МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)» (СГАУ)

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Мультимедийный-образовательный модуль
в системе дистанционного обучения Moodle

Работа выполнена по мероприятию блока 1 «Совершенствование
образовательной деятельности» Программы развития СГАУ
на 2009 – 2018 годы по проекту «Установка, настройка и использование в
учебном процессе факультета летательных аппаратов
системы дистанционного обучения (СДО) Moodle
совместно с блоком «Электронный деканат»
Соглашение № 1/16 от 03 июня 2013 г.

САМАРА
2013

УДК 043.43 (075)
Я 411

Автор-составитель: **Глушков Сергей Валериевич, Громова Екатерина Георгиевна**

Языки программирования [Электронный ресурс] : мультимедийный образоват. модуль в системе дистанц. обучения Moodle / М-во образования и науки РФ, Самар. гос. аэрокосм. ун-т им. С. П. Королева (нац. исслед. ун-т); авт.-сост. С. В. Глушков, Е. Г. Громова. - Электрон. текстовые и граф. дан. - Самара, 2013. – 1 эл. опт. диск (CD-ROM).

В состав мультимедийно-образовательного модуля входят:

1. Языки программирования. Конспект лекций.
2. Языки программирования. Материалы для подготовки к лабораторным работам.
3. Языки программирования. Набор тестов для самоконтроля по лекционному курсу.
4. Языки программирования. Список тем для подготовки докладов.
5. Языки программирования. Комплект индивидуальных заданий.

Мультимедийный-образовательный модуль предназначен для студентов факультета летательных аппаратов, обучающихся по направлению подготовки бакалавров 151600.62 «Прикладная механика» изучающих дисциплину «Языки программирования» во 2 семестре.

Модуль разработан на кафедре космического машиностроения СГАУ.

© Самарский государственный
аэрокосмический университет, 2013

Введение. Программирование как вид практической деятельности. Современные языки программирования. Средства разработки программ

Процесс решения задачи на компьютере состоит из ряда этапов, включающих как подготовку задачи к решению, так и собственно решение.

Эти этапы включают:

1. постановку задачи;
2. построение модели изучаемого процесса или явления;
3. выбор метода решения;
4. разработка алгоритма решения задачи;
5. составление программы (кодирование);
6. отладка и тестирование программы;
7. собственно вычисления;
8. анализ полученных результатов.

Рассмотрим эти этапы.

Постановка задачи

Под постановкой задачи подразумевается определение цели или целей, которых необходимо достичь в результате решения данной задачи. В постановку задачи входит определение необходимой информации (что дано), результата решения задачи (что требуется определить) и выработка общего подхода к решению задачи.

Построение соответствующей модели изучаемого процесса или явления.

На этом этапе производится выбор из всего множества зависимостей и связей основных, определяющих тот или иной реальный процесс, явление и формирование гипотез, позволяющих представить реальный, обычно достаточно сложный процесс, в виде уже известных процессов. Моделирование предусматривает некоторую разумную абстракцию, дающую

возможность с достаточной точностью представить себе реальные физические, информационные или другие процессы.

Для адекватного отражения сути изучаемого процесса или явления приходится разрабатывать различные модели. Чаще всего используются информационные и математические модели. В информационной модели указываются наиболее значимые характеристики объекта, имеющих существенное значение для данной задачи. Например, если создается база данных таксопарка, то наиболее значимыми характеристиками такого объекта как автомобиль являются марка автомобиля, год выпуска, государственный номер и др.

Под математической формулировкой задачи или как ее иногда называют математической моделью, подразумевается любое математическое описание изучаемого процесса или явления в виде уравнений или неравенств. В качестве уравнений могут быть алгебраические и трансцендентные уравнения, системы линейных алгебраических уравнений, дифференциальные уравнения, интегральные уравнения и т.д.

К математическому описанию предъявляются, в общем, противоречивые требования. С одной стороны математическое описание должно быть полным, с другой стороны желательно, чтобы математические зависимости были проще.

Выбор метода решения

Выбор метода решения зависит от вида модели, постановки задачи и возможностей имеющихся средств вычислительной техники. Многие задачи можно решить разными методами и способами, поэтому актуальным становится выбор оптимального метода. Причем критерии оптимальности даже для одной и той же задачи могут быть разными.

Поскольку компьютеры оперируют с числами, то в качестве методов решения математических моделей обычно применяются численные методы. Преобразование математических выражений, характерное для классической

математики, не является типичным при применении компьютеров. Хотя в существуют мощные программные системы типа MAPLE, которые позволяют производить и символьные вычисления. В то же время численные методы часто позволяют решать задачи, которые методами классической математики обычно неразрешимы.

Разработка алгоритма решения задачи

Характер работы на этом этапе существенно зависит от предыдущих этапов. Кроме того, имеет значение и размер задачи. Если это достаточно простая задача, то с ней может справиться один человек.

Для средних и крупных проектов, для реализации которых могут потребоваться группа или даже целый коллектив программистов, возникают проблемы определения методологии проектирования, планирования и распределения работ между членами группы, их взаимодействия и пр.

Важное значение имеет выбор средств проектирования и разработки ПО. В настоящее время широкое распространение получили так называемые RAD-системы (Rapid Application Development – быстрая разработка приложений). В качестве можно привести такие системы так Microsoft Visual Studio, Code Gear RAD Studio, Embarcadero RAD Studio, Lazarus и другие в которых имеются развитые средства для разработки ПО в коллективе.

Составление программы

Под этим этапом подразумевается непосредственная запись полученных ранее алгоритмов на выбранном языке программирования. Современные системы программирования позволяют значительно облегчить этот процесс, хотя этот этап по-прежнему остается одним из самых трудоемких. Разумеется, этот этап предполагает хорошее знание того языка программирования, на котором ведутся работы.

Отладка и тестирование программы

Под этим понимается поиск и исправление ошибок в программе. Причем под отладкой понимается исправление ошибок непосредственно в процессе кодирования. Огромную помощь программисту в этом деле оказывает компилятор, который указывает программисту место возникновения ошибки и характер ошибки. Однако факт того, что компилятор не сообщил об ошибке и программа стала работать, еще не гарантирует от отсутствия ошибок. Это так называемые логические ошибки или ошибки времени исполнения. Для выявления таких ошибок разрабатываются система тестов – специальным образом подобранные контрольные примеры, для которых решение задачи известно.

В крупных проектах программы подразделяются на версии. Альфа-версия это первая работоспособная версия программы. Бета-версия это версия или версии, которые передаются заказчику для дополнительного тестирования в уже реальных условиях функционирования программы.

Алгоритм. Свойства. Представления алгоритмов

Под алгоритмом понимают набор инструкций, описывающих порядок действий исполнителя для достижения результата решения задачи за конечное число действий.

Иногда вместо слова «порядок» используется слово «последовательность», но по мере развития параллельности в работе компьютеров слово «последовательность» стали заменять более общим словом «порядок». Это обусловлено тем, что работа каких-то инструкций алгоритма может быть зависима от других инструкций или результатов их работы. Например, некоторые инструкции должны выполняться строго после завершения работы инструкций, от которых они зависят. Независимые инструкции или инструкции, ставшие независимыми из-за завершения работы инструкций, от которых они зависят, могут выполняться в произвольном порядке, параллельно или одновременно, если это позволяют используемые процессор и операционная система.

Часто в качестве исполнителя выступает некоторый механизм (компьютер, станок с ЧПУ, автоматизированная линия производства, стиральная машина и т.д.), но понятие алгоритма необязательно относится к компьютерным программам, так, например, рецепт приготовления пиццы также является алгоритмом, в таком случае исполнителем является повар (человек).

Понятие алгоритма относится к первоначальным, основным, базисным понятиям математики. Вычислительные процессы алгоритмического характера (например, нахождение наибольшего общего делителя двух чисел, вычисление последовательности простых чисел) известны человечеству с глубокой древности. Однако понятие алгоритма в явном виде было сформировано лишь в начале XX века.

Существует несколько вероятных теорий происхождения термина Алгоритм. Наиболее популярное объяснение связывает это название с

именем хорезмского учёного Абу Абдуллах Мухаммеда ибн Муса аль-Хорезми (алгоритм == аль-Хорезми).

Современное формальное определение алгоритма было дано в 1930..50-е годы в работах Тьюринга, Поста, Винера.

Свойства алгоритма

Любой алгоритм должен обладать рядом следующих свойств:

— дискретность (алгоритм должен представлять процесс решения задачи как последовательное выполнение некоторых простых шагов, при этом для выполнения каждого шага алгоритма требуется конечный отрезок времени, то есть преобразование исходных данных в результат осуществляется во времени дискретно);

— детерминированность (в каждый момент времени следующий шаг работы однозначно определяется состоянием системы, т.е. алгоритм выдаёт один и тот же результат для одних и тех же исходных данных);

— понятность (алгоритм должен оперировать только теми командами, которые доступны исполнителю и входят в его систему команд);

— массовостью (алгоритм должен уметь решать не одну конкретную задачу, а целый класс однотипных задач при различных исходных данных);

— результативностью (алгоритм должен выдавать результат своей работы);

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

Эти же свойства присущи и программам, реализующим алгоритмы. Если же хотя бы одно из них оказывается невыполненным, программа полностью теряет смысл.

Алгоритм содержит ошибки, если он приводит к получению неверных результатов, либо не дает результатов вовсе. Алгоритм не содержит ошибок, если он дает результаты для любого корректного набора исходных данных.

Для разработки алгоритмов и программ используется алгоритмизация – процесс систематического составления алгоритмов для решения

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

Представление алгоритмов

Для записи алгоритмов возможно использовать самые различные формы. Среди них можно выделить:

- словесная (вербальная);
- псевдокод;
- схематическая (графическая).

Наиболее простой формой является словесная или вербальная форма. В этом случае алгоритм представляет собой запись последовательности команд в виде инструкций на русском языке (например, рецептура приготовления блюда). Однако следует отметить, что такая форма чрезвычайно громоздка и, в некоторых случаях, допускает возможность двоякого толкования. Кроме того, описать в вербальной форме разветвленные алгоритмы чрезвычайно тяжело из-за запутанности перекрестных ссылок на различные ветви решения. Вербальная форма алгоритмов нашла применение в должностных инструкциях, рекомендациях к действию при чрезвычайных происшествиях и т.п.

Следующей по сложности формой является представление алгоритма в виде псевдокода. Это компактный (зачастую неформальный) язык описания алгоритмов, использующий ключевые слова императивных языков программирования, но при этом опускающий несущественные подробности и специфический синтаксис. Псевдокод обычно опускает детали, несущественные для понимания алгоритма человеком, например описания переменных, системно-зависимый код и подпрограммы. Главная цель использования псевдокода – обеспечить понимание алгоритма человеком,

сделать описание более воспринимаемым, чем исходный код на языке программирования. Псевдокод широко используется в учебниках и научно-технических публикациях, а также на начальных стадиях разработки компьютерных программ. Как правило, инструкции псевдокода описываются на естественном для человека языке (русский, английский и др.). Пример – алгоритмический язык Ершова.

Схематическое изображение алгоритмов представляется наиболее популярным видом описания алгоритмов. В этом случае алгоритм заменяется блок-схемой, в которой отдельные шаги изображаются в виде блоков различной формы, соединенных между собой линиями, указывающими направление последовательности действий. Правила выполнения блок-схем регламентируются ГОСТ 19.701-90. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения.

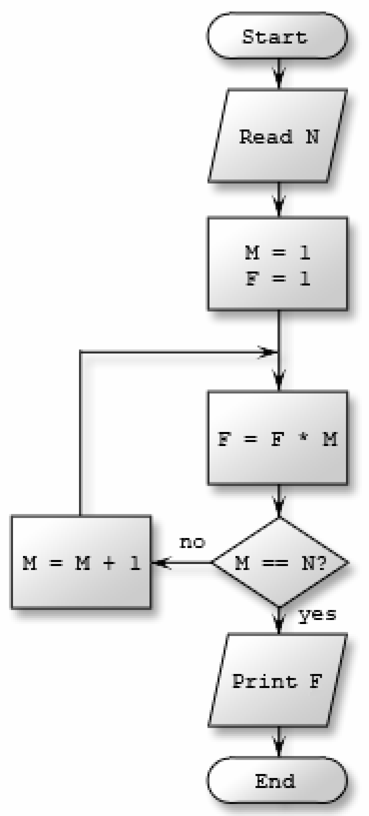
Основные элементы блок схем

Наименование	**Обозначение**	**Функция**
Блок начало-конец		Начало и конец программы/подпрограммы. Внутри фигуры записывается соответствующее действие.
Блок действия		Выполнение одной или нескольких операций, обработка данных любого вида (изменение значения данных, формы представления, расположения). Внутри фигуры записывают непосредственно сами операции, например, присваивание: x = 10 y+ln(z).
Логический блок (блок условия)		Отображает решение или функцию переключательного типа с одним входом и двумя или более альтернативными выходами, из которых только один может быть выбран после вычисления условий, определенных внутри этого элемента. Вход в элемент обозначается линией, входящей обычно в верхнюю вершину элемента. Если выходов два или три, то обычно каждый выход обозначается линией, выходящей из оставшихся вершин (боковых и нижней). Если выходов больше трех, то их следует

		показывать одной линией, выходящей из вершины (чаще нижней) элемента, которая затем разветвляется. Соответствующие результаты вычислений могут записываться рядом с линиями, отображающими эти пути. Примеры решения: в общем случае − сравнение (три выхода: >, <, =). Эквивалент оператора if или case
Предопределён ный процесс		Символ отображает выполнение процесса, состоящего из одной или нескольких операций, который определен в другом месте программы (в подпрограмме, модуле). Внутри символа записывается название процесса и передаваемые в него данные. Например, в программировании − вызов процедуры или функции.
Ввод-вывод данных		Преобразование данных в форму, пригодную для обработки (ввод) или отображения результатов обработки (вывод)
Граница цикла		Символ состоит из двух частей − соответственно, начало и конец цикла − операции, выполняемые внутри цикла, размещаются между ними. Условия цикла и приращения записываются внутри символа начала или конца цикла − в зависимости от типа организации цикла. Часто для изображения на блок-схеме цикла вместо данного символа используют символ условия, указывая в нём решение, а одну из линий выхода замыкают выше в блок-схеме (перед операциями цикла).
Соединитель		Символ отображает вход в часть схемы и выход из другой части этой схемы. Используется для обрыва линии и продолжения её в другом месте (для предотвращения излишних пересечений или слишком длинных линий, а также, если схема состоит из нескольких страниц). Соответствующие соединительные символы должны иметь одинаковое (при том уникальное) обозначение.
Комментарий		Используется для более подробного описания шага, процесса или группы процессов. Описание помещается со стороны квадратной скобки и охватывается

		ей по всей высоте. Пунктирная линия идет к описываемому элементу, либо группе элементов (при этом группа выделяется замкнутой пунктирной линией). Также символ комментария следует использовать в тех случаях, когда объём текста, помещаемого внутри некоего символа (например, символ процесса, символ данных и др.), превышает размер самого этого символа, т.е. в роли выносного элемента

Пример блок схемы для алгоритма вычисления факториала числа



Алгоритм работает так. Запрашивается число N. Устанавливаются начальные значения M и F, равными 1. Переменная F принимает новое значение, равное произведению F на M. Если M <> N то переменная M увеличивает значение на 1 и алгоритм повторяется с момента вычисления нового значения F. Как только выполнится условие M=N, на печать выводится результат вычисления. Программа завершается.

Язык программирования Pascal. История. Синтаксические особенности. Структура программы

Язык программирования Pascal это язык программирования общего назначения. До сегодняшнего дня остается одним из наиболее популярных языков (в том числе и для целей обучения основам программирования), является базой для ряда других языков.

Pascal это алголоподобный язык программирования, который ввел в широкое употребление понятие типа данных и принципы структурного программирования. Свое название получил в честь французского математика 17 века Блеза Паскаля.

Первая версия языка Паскаль была разработана швейцарским ученым Никлаусом Виртом, сотрудником Высшей технической школы в Цюрихе, в 1968. Изначально Вирт ставил перед собой задачу создания учебного языка программирования, который бы строился на небольшом количестве базовых понятий, имел простой синтаксис, допускал перевод программ в машинный код простым компилятором. Язык Паскаль обеспечивает возможность создания больших программ, поддерживая их строгую логическую структуру. Однако для коротких программ Паскаль может оказаться излишне громоздким.

Широкое признание программистов и простых пользователей этот язык получил после появления диалекта Турбо Паскаль, созданного американской фирмой Борланд. Паскаль считается важнейшим инструментом для обучения методам структурного программирования и с 1983 введен во всех средних школах США в учебные курсы для учащихся, которые специализируются в области информатики. Наличие специальных методик создания трансляторов с Паскаля упростило их разработку и способствовало широкому распространению языка.

В 1965 году был объявлен конкурс по созданию нового языка программирования - преемника языка АЛГОЛ-60. Участие в конкурсе принял

швейцарский учёный, поляк по происхождению, Никлаус Вирт, работавший доцентом на факультете информатики Стэнфордского университета. Проект предложенный им был отвергнут комиссией в 1967 году. Но Вирт не прекратил работу над созданием нового языка. Вернувшись в Швейцарию, совместно с сотрудниками Швейцарского федерального института технологии в Цюрихе, он уже в 1968 году разработал первую версию языка Паскаль. Язык назван в честь великого французского математика и механика Блеза Паскаля, в 1648 г. создавшего первую счётную машину. В 1971 г. Н.Вирт выпустил описание своего языка, а в 1975 г. было разработано руководство для пользователей Паскаля, практически легшей в основу стандарта. Но такой документ появился только в 1982 г. - международный стандарт ISO на язык Паскаль.

Паскаль переживал громадный успех и в конце 70-х годов получил широкое распространение в университетах. Но разработчики программного обеспечения, попытавшиеся приспособить Паскаль для микрокомпьютеров и использовать его в коммерческих целях, внесли в язык значительные изменения. Поскольку Вирт изначально разрабатывал Паскаль для обучения студентов, язык почти не имел ввода-вывода и других средств, существенных для практического программирования. По этой причине в компиляторах Паскаля появлялись всё новые расширения, выводящие язык за пределы чисто академических приложений. Это породило множество диалектов. Вирт не признал новые диалекты, провозгласив в 1977 г. свою позицию: "Если язык едва подходит для применения, на который его создатель явно не рассчитывал, то следует набраться смелости и создать новый, полностью адекватный язык, а не усложнять существующий". Именно это Вирт и сделал. В 1981 г. он разработал язык Модула-2, который должен был заменить Паскаль в универсальных применениях. До этого времени, различные компиляторы Паскаля для микрокомпьютеров не имели особого коммерческого успеха, поскольку были дороги, занимали большой объём памяти на диске и работали с черепашьей скоростью. Но тут на арену вышел

бывший студент Вирта со своим компилятором для микрокомпьютеров, который он назвал Турбо-Паскаль. Это был небольшой, мгновенно срабатывающий и удивительно дешёвый компилятор. Разработчиком Турбо-Паскаля стал француз Филип Кан. Родом из Парижа, Кан некоторое время учился в Цюрихе, где прослушал вводный курс Вирта по Паскалю. Вернувшись во Францию, он защитил диссертацию на степень кандидата наук по математике. Кан занимался математикой, а компьютерами интересовался лишь постольку, поскольку это помогало ему в решении задач и доказательстве теорем. Затем он приобрёл свой первый микрокомпьютер. Вместе с ещё двумя приятелями он начал зарабатывать небольшие деньги, составляя на Паскале прикладные программы для микрокомпьютеров. Неудовлетворённость существующими компиляторами Паскаля побудила Кана взяться за разработку Турбо-Паскаля. В 1982 г., имея на руках эту программу и 2000 $ в кармане, он покинул Францию и уехал в Калифорнию.

Вначале Кан не мог получить там даже работу. Путешествуя по туристской визе, он не имел права на так называемую зелёную карточку – правительственное разрешение на получение работы в США. В отчаянии он решил сделать ставку на Турбо-Паскаль и создал новую фирму "Borland International" – название, подсказанное телевизионной передачей о бывшем космонавте с похожим именем. Кан решил, что имя, имеющее "всеамериканское звучание", будет подходящим прикрытием для чужака в мире бизнеса.

Несмотря на громкое имя фирмы, Кану не удалось заинтересовать даже представителей рискового капитала. Кан привлёк к работе в своём новом предприятии других образованных новичков. Не имея возможности получить хотя бы доллар из официальных источников финансирования, Кан собрал небольшую сумму (20 000 $) у членов своей семьи. В марте 1984 г. он организовал контору в двухкомнатном офисе над авторемонтным гаражом. Приступив к делу, Кан уговорил служащего одного популярного

компьютерного журнала провести широкую рекламу Турбо-Паскаля в кредит.

Кан понимал, что рекламное объявление – единственный возможный способ пробиться, и сделал всё возможное. Прочитав книгу о продаже товаров по почте, где рекомендовалось использовать яркие цвета для привлечения читателей, он оформил объявление в резких зелёных, синих и розовых тонах. В книге также говорилось, что для быстрого успеха, стоимость товара не должна превышать 50 $. Поэтому он оценил свой сложный компилятор и редактор программ в 49,95$. Это была предельно низкая граница, поскольку все остальные системы продавались в 10 раз дороже. Реклама оказалась гениальной находкой для завоевания рынка. Только за первый месяц она принесла Кану заказов на 150 000 $. Эксперты отмечали также быстродействие компилятора Кана, который работал во много раз быстрее, чем его соперники.

За первые два года было продано не менее 300 тыс. копий Турбо-Паскаля, что превзошло объём продажи всех прочих языков для микрокомпьютеров.

Турбо-Паскаль почти за одну ночь перевёл "Borland International" в разряд основных производителей программного обеспечения. В 1992 году фирма Borland International выпустила два пакета программирования, основанные на использовании языка Паскаль, - Borland Pascal 7.0 и Turbo Pascal 7.0, которые получили наибольшее признание.

Пакет Borland Pascal 7.0 учитывает многие новейшие достижения в программировании и практике создания программ и включает в себя три режима работы: в режиме операционной системы MS DOS, в защищённом режиме MS DOS и в среде Windows.

Никлаус Вирт в настоящее время продолжает работать в Швейцарском федеральном технологическом институте. Созданный им язык программирования Модула-2 не имел широкого успеха. Как утверждают специалисты, основной причиной этого является то, что идеи, заложенные в

Модула-2, были настолько революционными, что опережали время. Последним изобретением Никлауса Вирта является язык программирования Oberon, которому прочат большое будущее.

Фирма Borland International была впоследствии приобретена фирмой Inprise, и в настоящее время продолжает выпускать свои продукты, но уже под логотипом фирмы Inprise. Одним из продуктов этой фирмы является очень популярная на сегодняшний день среда визуального программирования Borland Delphi, разработанная на базе языка программирования Pascal.

Однако высокая стоимость пакета Delphi привела к появлению большого числа альтернативных сред разработки, основанных на бесплатных компиляторах языка Pascal. В настоящем курсе будем рассматривать такую бесплатную среду визуального программирования Lazarus, основанную на свободно-распространяемом компиляторе Free Pascal.

Структура Pascal-программы

Для того чтобы Pascal-компилятор правильно понял, какие именно действия от него ожидаются, ваша программа должна быть оформлена в полном соответствии с синтаксисом этого языка.

Любая Pascal-программа может состоять из следующих блоков (здесь и далее квадратными скобками помечены необязательные части):

```
program <имя_программы>;
    [ uses <имена_подключаемых_модулей>;]
    [ label <список_меток>;]
    [ const <имя_константы> = <значение_константы>;]
    [ type <имя_типа> = <определение_типа>;]
    [ var <имя_переменной> : <тип_переменной>;]
    [ procedure <имя_процедуры> <описание_процедуры>;]
    [ function <имя_функции> <описание_функции>;]
    begin   {начало основного тела программы}
    <совокупность операторов программы>
    end.  (* конец основного тела программы *)
```

Здесь следует сделать оговорку: название программы, то есть строку

```
program <имя_программы>;
```

проще говоря, можно опустить, поскольку она не является обязательной. Но, для лучшего понимания сути разрабатываемой программы, следует все же приводить имя программы, и формулировать его таким, чтобы оно являлось информативным.

Любой из перечисленных необязательных разделов может встречаться в тексте программы более одного раза, их общая последовательность также может меняться, но при этом всегда должно выполняться главное правило языка – прежде чем объект будет использован, он должен быть объявлен и описан.

Внешний вид исходного текста программы

Компиляторы языка Pascal не различают строчные и прописные буквы, а пробельные символы игнорируют, поэтому текст программы можно структурировать так, чтобы читать и отлаживать его было наиболее удобно.

Например, операторы каждого логически единого блока программы стоит записывать с небольшим отступом от левого края экрана, и чем глубже вложенность блока, тем шире должны быть отступы перед входящими в него операторами. Кроме того, встроенный редактор среды Lazarus автоматически выравнивает левые края строк.

Для облегчения отладки программы не следует записывать на одну строку несколько операторов.

Хотя компилятор Pascal не различает строчные и прописные буквы, общепринятым правилом именования переменных, методов, функций и других объектов принято, что если название состоит из нескольких слов, то каждое слово писать с заглавной буквы.

Помимо отступов, большие логически замкнутые блоки программы удобно разделять строками-комментариями, содержащими информацию о

смысле последующего блока. Комментарий - это строка (или несколько строк) из произвольных символов, заключенная в фигурные скобки:

```
{ комментарий }
```

Другой вариант оформления комментария:

```
(* комментарий *)
```

Одну строку комментария можно установить так:

```
// строчный комментарий
```

Внутри самого комментария символы } или *) встречаться не должны.

Во время компилирования программы любые комментарии игнорируются. Следовательно, их можно добавлять в любом месте программы. Можно даже разорвать оператор вставкой комментария. Кроме того, все, что находится после ключевого слова end., завершающего текст программы, компилятор тоже воспринимает как комментарий.

Идентификаторы

Имена, даваемые программным объектам (константам, типам, переменным, функциям и процедурам, да и всей программе целиком) называются идентификаторами. Они могут состоять только из цифр, латинских букв и знака подчеркивание. Однако цифра не может начинать имя. Идентификаторы могут иметь любую длину, но если у двух имен первые 63 символа совпадают, то такие имена считаются идентичными.

Вы можете давать программным объектам любые имена, но необходимо, чтобы они отличались от зарезервированных слов, используемых языком Pascal.

Список наиболее часто встречающихся зарезервированных слов:

```
and            goto            set
array          implementation  shl
```

begin	in	shr
case	interface	string
const	label	then
div	mod	text
do	nil	to
downto	not	type
else	of	unit
end	or	until
file	pointer	uses
far	procedure	var
for	program	while
forward	record	with
function	repeat	xor

Переменные. Константы. Операторы ввода и вывода

Переменная

Переменная – это программный объект, значение которого может изменяться в процессе работы программы.

Тип данных – это характеристика диапазона значений, которые могут принимать переменные, относящиеся к этому типу данных.

Все используемые в программе переменные должны быть описаны в специальном разделе var по следующему шаблону:

```
var <имя_перем_1> [,<имя_перем_2>,...] : <имя_типа_1>;
    <имя_перем_3> [,<имя_перем_4>,...] : <имя_типа_2>;
```

Язык Pascal обладает большим набором разнообразных типов данных, однако сейчас мы рассмотрим лишь некоторые из них. Обо всех же типах данных мы поговорим в следующей лекции, там же будут показаны различные примеры описания переменных.

Константа

Константа – это объект, значение которого известно еще до начала выполнения программы. К константам относятся фундаментальные математические или физические значения – например, число пи и т.п.

Константы необходимы для оформления наглядных программ, незаменимы при использовании в тексте программы многократно повторяемых значений, поскольку удобны в случае необходимости изменения этих значений сразу во всей программе.

В языке Pascal существует три вида констант:

— неименованные константы (цифры и числа, символы и строки, множества);

— именованные нетипизированные константы;

— именованные типизированные константы.

Рассмотрим каждый вид подробнее.

Неименованные константы

Неименованные константы не имеют имен, и потому их не нужно предварительно описывать. Свои значения они получают на этапе разбора исходного кода программы компилятором.

Тип неименованной константы определяется автоматически, по умолчанию:

— любая последовательность цифр (возможно, предваряемая знаком "-" или "+" или разбиваемая одной точкой) воспринимается компилятором как неименованная константа – число (целое или вещественное);

— любая последовательность символов, заключенная в апострофы, воспринимается как неименованная константа – строка или символ (если он один);

— любая последовательность целых чисел либо символов через запятую, обрамленная квадратными скобками, воспринимается как неименованная константа – множество.

Кроме того, существуют две специальные константы true и false, относящиеся к логическому типу данных, описывающих соответственно истинное и ложное значения.

Примерами использования неименованных констант могут послужить следующие операторы:

```
i1  := 199;
r2  := 1.05 – x;
b3  := true;
c4  := 'h';
str5 := 'ssau';
```

Именованные нетипизированные константны

Как следует из названия, именованные константы, должны иметь имя. Эти имена необходимо предварительно сообщить компилятору, то есть описать в специальном разделе исходного кода программы – блоке const.

Если не указывать тип константы, то по ее внешнему виду компилятор сам автоматически определит, к какому (базовому) типу ее отнести. Любую уже описанную константу можно использовать при объявлении других констант, переменных и типов данных. Вот несколько примеров описания нетипизированных именованных констант:

```
const    StudCount = 18;
         million = 1000000;
         g = 9.81;
         Pi = 3.1415;
         UniverName = 'SSAU';
         GenderM = 'M';
```

Именованные типизированные константны

Типизированные именованные константы представляют собой переменные (!) с начальным значением, которое к моменту старта программы уже известно. Следовательно, во-первых, типизированные константы нельзя использовать для определения других констант, типов данных и переменных, а во-вторых, их значения можно изменять в процессе работы программы.

Описание типизированных констант производится по следующему шаблону:

```
const <имя_конст> : <тип_конст> = <начальное_значение>;
```

Из приведенных ниже примеров видно, как это сделать:

```
const    GroupNumber: integer = 1104;
         x: real = 1.2;
         c: char = 'q';
         b: boolean = true;
```

Простейшие операторы

Оператором называется минимальная структурно завершенная единица программы.

Все операторы языка Pascal должны заканчиваться знаком ";" (точка с запятой), и ни один оператор не может разрываться этим знаком. Единственная возможность не ставить после оператора ";" появляется лишь в том случае, когда сразу за этим оператором следует ключевое слово end.

К простейшим операторам языка Pascal относятся:

1. ";" – так называемый пустой оператор, который можно вставлять куда угодно, а также вычеркивать откуда угодно, поскольку на целостность программы это никак не влияет.
2. "a:= b;" – оператор присваивания переменной a значения переменной b. В правой части присваивания может находиться переменная, константа, арифметическое выражение или вызов функции, слева – только переменная.
3. Операторные скобки, превращающие несколько операторов в один:

```
begin
  <несколько операторов>
end;
```

Далее всегда, когда в записи конструкций языка Pascal мы будем использовать обозначение <один_оператор>, его следует понимать как "один оператор или несколько операторов, заключенные в операторные скобки begin – end".

Операторы ввода-вывода

Как мы уже говорили, любой алгоритм должен быть результативным. В общем случае это означает, что он должен сообщать результат своей работы потребителю: пользователю-человеку или другой программе. Не будем рассматривать здесь внутренние автоматические процессы, использующие сигналы непрерывно функционирующих программ, а сосредоточим

внимание на взаимодействии программы и человека, то есть на процессах ввода информации с клавиатуры и вывода ее на экран.

В программировании существует специальное понятие консоль, которое обозначает клавиатуру при вводе и монитор при выводе.

Ввод с консоли

Для того чтобы получить данные, вводимые пользователем вручную (то есть с консоли), применяются команды

```
read(<список_ввода>) и readln(<список_ввода>)
```

Первая из этих команд считывает все предложенные ей данные, оставляя курсор в конце последней строки ввода, а вторая – сразу после окончания ввода переводит курсор на начало следующей строки.

Список ввода - это последовательность имен переменных, разделенных запятыми. Например, при помощи команды

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

программа может получить с клавиатуры данные сразу для четырех переменных.

Вводимые значения необходимо разделять пробелами, а завершать ввод – нажатием клавиши Enter. Ввод данных заканчивается в тот момент, когда последняя переменная из списка ввода получила свое значение. Следовательно, вводя данные при помощи приведенной выше команды, вы можете нажать Enter четыре раза – после каждой из вводимых переменных, либо же только один раз, предварительно введя все четыре переменные в одну строчку (разделив их пробелами).

Типы вводимых значений должны совпадать с типами указанных переменных, иначе возникает ошибка. Поэтому нужно внимательно следить за правильностью вводимых данных.

Вообще, вводить с клавиатуры можно только данные базовых типов (за исключением логического). Если же программе все-таки необходимо получить с консоли значение для boolean-величины, придется действовать более хитро: вводить оговоренный символ, а уже на его основе присваивать логической переменной соответствующее значение. Например:

```
repeat
  writeln('Согласны ли Вы с этим утверждением?
          1 - да, 0 - нет');
  readln(a); {a:integer}
  case a of
    1: b:= true;
    0: b:= false;
     else writeln('Ошибка!');
   end;
until (c=0) or (c=1);
```

Второе исключение: строки, хотя они и не являются базовым типом, вводить тоже разрешается. Признаком окончания ввода строки является нажатие клавиши Enter, поэтому все следующие за нею переменные необходимо вводить с новой строчки.

Вывод на консоль

Ожидая от человека ввода с клавиатуры, не следует полагать, что он догадается, какого типа переменная нужна ожидающей программе. Старайтесь всегда придерживаться правила: каждый запрос данных должен быть прокомментирован. Это значит, что перед тем как считывать что-либо с консоли, необходимо сообщить пользователю, что именно он должен ввести: смысл вводимой информации, тип данных, максимальное и минимальное допустимые значения и т.п.

Примером неплохого приглашения служит, скажем, такая строчка:

```
Введите два положительных вещественных числа
(0.1<a,b<1000) - длины катетов треугольника.
```

Впрочем, и ее можно улучшить, сообщив пользователю не только допустимый диапазон ввода, но и ожидаемую точность (количество знаков после запятой).

Средства, позволяющие организовать выдачу информации на экран, мы здесь и рассмотрим.

Для того чтобы вывести на экран какое-либо сообщение, воспользуйтесь процедурой

```
write(<список_вывода>) или writeln(<список_вывода>).
```

Первая из них, напечатав на экране все, о чем ее просили, оставит курсор в конце выведенной строки, а вторая переведет его в начало следующей строчки.

Список вывода может состоять из нескольких переменных, записанных через запятую; все эти переменные должны иметь тип либо базовый, либо строчный. Например,

```
writeln(a,b,c);
```

Форматный вывод

Если для вывода информации воспользоваться последней командой, то выводимые символы окажутся "слепленными" на экране, т.е. три числа будут записаны непрерывно, без промежуточных интервалов. Чтобы этого не случилось, нужно либо позаботиться о пробелах между выводимыми переменными:

```
writeln(a,' ',b,' ',c);
```

либо задать для всех (или для некоторых) переменных формат вывода:

```
writeln(a:5,b,c:20:5);
```

Здесь первое число после знака ":" обозначает количество позиций, выделяемых под всю переменную, а второе – под дробную часть числа. Десятичная точка тоже считается отдельным символом.

Если число длиннее, чем отведенное под него пространство, количество позиций будет автоматически увеличено. Если же выводимое число короче заданного формата, то спереди к нему припишется необходимое количество пробелов. Таким образом, можно производить вывод ровными колонками, а также следить за тем, чтобы переменные не сливались друг с другом.

Например, если a = 5, b = 'c', а c = 11.3, то после выполнения команды writeln(a:5,' ',b,c:10:5) на экране или в файле будет записано следующее (подчерки в данном случае служат лишь для визуализации пробелов):

```
___ 5_c_ _11.30000
```

Особенно важен формат при выводе вещественных переменных. К примеру, если не указать формат, то число 11.3 будет выведено как 1.1300000000E+0001. Этот формат называется записью с плавающей точкой.

Если же задать только общую длину вещественного числа, не указывая длину дробной части, то оно будет занимать на экране заданное количество символов (при необходимости, спереди будет добавлено соответствующее количество пробелов), но при этом останется в формате плавающей точки. Минимальной длиной для вывода вещественных чисел является 10 (при формате _x.xE+yyyy). Первая позиция зарезервирована под знак минуса для отрицательных значений.

Необходимо помнить, что в случае недостаточной длины вывода число будет автоматически округлено, например (напоминаем, что символ подчеркивания служит лишь для визуализации пробела):

```
Оператор вывода            Результат вывода на экран
writeln(125.2367:10);      _1.3E+0002
writeln(125.2367:11);      _1.25E+0002
writeln(125.2367:12);      _1.252E+0002
writeln(125.2367:13);      _1.2524E+0002
writeln(125.2367:14);      _1.25237E+0002
writeln(125.2367:15);      _1.252367E+0002
writeln(125.2367:16);      _1.2523670E+0002
```

Пример простейшей программы на языке Pascal

```
program hello;
var s: string;
begin
 write('Пожалуйста, введите Ваше имя: ');
 readln(s);
 writeln('Hello, ', s, '!');
end.
```

Во время работы этой программы на экране появится примерно следующее:

```
Пожалуйста, введите Ваше имя: Михаил
Hello, Михаил!
```

Жирным шрифтом показана строка, введенная пользователем после приглашения.

Базовые типы данных. Арифметические операции. Вычисление выражений

Типы данных языка Pascal

Компиляторы языка Pascal требуют, чтобы сведения об объеме памяти, необходимой для работы программы, были предоставлены до начала ее работы. Для этого в разделе описания переменных (блок `var`, `см. лекцию 3`) нужно перечислить все переменные, используемые в программе. Кроме того, необходимо также сообщить компилятору, сколько памяти каждая из этих переменных будет занимать, а также заранее условиться о различных операциях, применимых к тем или иным переменным.

Все это можно сообщить программе, указав тип декларируемой (объявляемой) переменной. Имея информацию о типе переменной, компилятор "понимает", сколько байт необходимо отвести под нее, какие действия с ней можно производить и в каких конструкциях она может участвовать.

Для удобства программистов в языке Pascal существует множество стандартных типов данных и плюс к тому возможность создавать новые типы.

Конструируя новые типы данных на основе уже имеющихся (стандартных или определенных самим программистом), нужно помнить, что любое здание должно строиться на хорошем фундаменте. Поэтому сейчас мы и поговорим об этом "фундаменте".

На основании базовых типов данных строятся все остальные типы языка Pascal, которые так и называются: конструируемые.

Все типы данных языкаTurbo Pascal разделяются на две группы: скалярные (простые) и структурированные (составные).

Простые типы данных

Целочисленные типы данных (Integer, Byte, Word) представляют собой значения, которые могут использоваться в арифметических выражениях и занимать память от 1 до 4 байт.

Тип	Диапазон	Размер, байт
Byte	0...255	1
Shortint	-128…127	1
Word	0...65535	2
Longint	-2147483648...2147483647	4

Вещественные типы данных(Real, Double) представляют собой вещественные значения, которые могут использоваться в арифметических выражениях и занимать память от 4 до 10 байт.

Тип	Диапазон	Знач. цифр	Размер, байт
Real	2.9E-39…1.7E38	11-12	6
Single	1.5E-45...3.4E38	7-8	4
Double	5E-324...1.7E308	15-16	8
Extended	1.9E–4951...1.1E4932	19-20	10
Comp	(-2E+63)+1 ...(2E+63)-1	10-20	8

Символьный тип данных(char) определяется множеством значений кодовой таблицы ASCII. Для переменной символьного типа требуется 1 байт.

Логический тип данных(Boolean) представлен двумя значениями: true (истина) и false (ложь). Он широко применяется в логических выражениях и выражениях отношения.

Пользовательские типы данных

В языке Pascal существуют типы данных, определяемые непосредственно пользователем. Это перечислимый тип (когда непосредственно, в разделе описания типов, заранее записываются все значения для переменных этого типа) и интервальный (когда задаются границы диапазона значений для данной переменной).

Перечислимый тип данных задается непосредственно перечислением всех значений, которые может принимать переменная данного типа. При описании отдельные значения указываются через запятую, а весь список заключается в круглые скобки.

Например:

```
Var
    Season: (winter, spring, summer, autumn);
    Temp: (20,22,24,26);
```

Интервальный тип позволяет задавать две константы, определяющие границы диапазона значений для каждой переменной. Обе константы должны принадлежать одному и тому же стандартному типу (кроме, разумеется, real).

Например:

```
Var
    Grup: 1101..1699;
    SmallChar: ’a’..’z’;
```

Типы данных, конструируемые программистом, рекомендуется описывать в разделе `type` (см. лекцию 3) по следующему шаблону:

```
type <имя_типа> = <описание_типа>;
```

Например:

```
type     lat_bukvy = 'a'..'z';
var      SmallChar: lat_bukvy;
```

Стандартные конструируемые типы также можно не описывать в разделе `type`. Однако в некоторых случаях это все равно приходится делать из-за требований синтаксиса.

Структурированные типы данных

Составные типы данных определяют упорядоченную совокупность скалярных переменных и характеризуются типом своих компонентов.

К структурированным типам данных в языке Pascal относят:

— тип–массив (array);
— строковый тип (string).
— тип–множество (set);
— тип–запись (record);
— файловый тип (file);
— объектный тип (object).

Структурированные типы будут рассмотрены в более поздних лекциях.

Порядковые типы данных

Среди простых типов данных (кроме вещественных чисел) особо выделяются порядковые типы. Такое название можно обосновать двояко:

1. Каждому элементу порядкового типа может быть сопоставлен уникальный (порядковый) номер.
2. Кроме того, на элементах любого порядкового типа определен порядок (в математическом смысле этого слова), который напрямую зависит от нумерации. Таким образом, для любых двух элементов порядкового типа можно точно сказать, который из них меньше, а который - больше

Только для величин порядковых типов определены следующие функции и процедуры:

1. Функция `ord(x)` возвращает порядковый номер значения переменной x (относительно того типа, к которому принадлежит переменная x).
2. Функция `pred(x)` возвращает значение, предшествующее x (к первому элементу типа неприменима).
3. Функция `succ(x)` возвращает значение, следующее за x (к последнему элементу типа неприменима).
4. Процедура `inc(x)` возвращает значение, следующее за x (для арифметических типов данных это эквивалентно оператору x:=x+1).
5. Процедура `inc(x,k)` возвращает k-е значение, следующее за x (для арифметических типов данных это эквивалентно оператору x:=x+k).
6. Процедура dec`(x)` возвращает значение, предшествующее x (для арифметических типов данных это эквивалентно оператору x:=x-1).
7. Процедура dec`(x,k)` возвращает k-е значение, предшествующее x (для арифметических типов данных это эквивалентно оператору x:=x-k).

На первый взгляд кажется, будто результат применения процедуры `inc(x)` полностью совпадает с результатом применения функции `succ(x)`. Однако разница между ними проявляется на границах допустимого диапазона. Функция `succ(x)` неприменима к максимальному элементу типа, а вот процедура `inc(x)` не выдаст никакой ошибки, но, действуя по правилам машинного сложения, прибавит очередную единицу к номеру элемента. Номер, конечно же, выйдет за пределы диапазона и за счет усечения превратится в номер минимального значения диапазона. Получается, что процедуры `inc()` и dec`()` воспринимают любой

порядковый тип словно бы "замкнутым в кольцо": сразу после последнего вновь идет первое значение.

Опишем теперь порядковые типы данных более подробно.

Логический тип `boolean` имеет два значения: `false` и `true`, и для них выполняются следующие равенства:

```
ord(false)=0, ord(true)=1, false<true,
pred(true)=false, succ(false)=true,
inc(true)=false, inc(false)=true,
dec(true)=false, dec(false)=true.
```

В символьный тип `char` входит 256 символов расширенной таблицы ASCII (например, 'a', 'b', 'я', '7', '#'). Номер символа, возвращаемый функцией `ord()`, совпадает с номером этого символа в таблице ASCII.

Операции и выражения

Как говорилось ранее, для каждого типа данных определены действия, применимые к его значениям. Например, если переменная относится к порядковому типу данных, то она может фигурировать в качестве аргумента стандартных функций `ord()`, `pred()` и `succ()`. А к вещественным типам эти функции применить невозможно.

Рассмотрим операции – стандартные действия, разрешенные для переменных того или иного базового типа данных. Основу будут составлять арифметические операции, но, будут затронуты и логический тип данных. Операции, определенные для значений символьного типа, будут подробно рассмотрены в лекции посвященной строкам.

Замечание: Все перечисленные ниже операции (за исключением унарных '-' и not) требуют двух операндов.

Логические операции (and, or, not, xor) применимы только к значениям типа `boolean`. Их результатом также служат величины типа `boolean`. Приведем таблицы значений для этих операций:

Оператор	Операнд 1	Операнд 2	Результат
not	false	-	true
	true	-	false
and	false	false	false
	false	true	false
	true	false	false
	true	true	true
or	false	false	false
	false	true	true
	true	false	true
	true	true	true
xor	false	false	false
	false	true	true
	true	false	true
	true	true	false

Операции сравнения (=, <>, >, <, <=, >=) применимы ко всем базовым типам. Их результатами также являются значения типа `boolean`.

Операции целочисленной арифметики применимы, как легко догадаться, только к целым типам. Их результат - целое число, тип которого зависит от типов операндов.

`a div b` – деление a на b нацело (напоминаем, что деление на 0 запрещено, поэтому в таких случаях операция выдает ошибку). Результат будет принадлежать к типу данных, общему для тех типов, к которым принадлежат операнды. Например, (shortint div byte = integer). Пояснить это можно так: integer – это минимальный тип, подмножествами которого являются одновременно и byte, и shortint.

`a mod b` – взятие остатка при делении a на b нацело. Тип результата, как и в предыдущем случае, определяется типами операндов, а 0 является запрещенным значением для b. В отличие от математической операции mod,

результатом которой всегда является неотрицательное число, знак результата "программистской" операции mod определяется знаком ее первого операнда. Таким образом, если в математике (-2 mod 5)=3, то в Pascal (-2 mod 5)= -2.

Операции общей арифметики (+, -, *, /) применимы ко всем арифметическим типам. Их результат принадлежит к типу данных, общему для обоих операндов (исключение составляет только операция дробного деления /, результат которой всегда относится к вещественному типу данных).

Стандартные арифметические функции

К арифметическим операциям примыкают и стандартные арифметические функции. Их список с кратким описанием мы приводим в таблице.

	Описание	**Тип аргумента**	**Тип результата**
`abs(x)`	Абсолютное значение (модуль) числа	Арифметический	Совпадает с типом аргумента
`arctan(x)`	Арктангенс (в радианах)	Арифметический	Вещественный
`cos(x)`	Косинус (в радианах)	Арифметический	Вещественный
`exp(x)`	Экспонента (e^x)	Арифметический	Вещественный
`frac(x)`	Взятие дробной части числа	Арифметический	Вещественный
`int(x)`	Взятие целой части числа	Арифметический	Вещественный
`ln(x)`	Натуральный	Арифметический	Вещественный

	логарифм (по основанию е)		
`odd(x)`	Проверка нечетности числа	Целый	boolean
`pi`	Значение числа пи	-	Вещественный
`round(x)`	Округление к ближайшему целому	Арифметический	Целый
`trunc(x)`	Округление "вниз" - к ближайшему меньшему целому	Арифметический	Целый
`sin(x)`	Синус (в радианах)	Арифметический	Вещественный
`sqr(x)`	Возведение в квадрат	Арифметический	Вещественный
`sqrt(x)`	Извлечение квадратного корня	Арифметический	Вещественный

Арифметические выражения

Все арифметические операции можно сочетать друг с другом - конечно, с учетом допустимых для их операндов типов данных.

В роли операндов любой операции могут выступать переменные, константы, вызовы функций или выражения, построенные на основе других операций.

Порядок вычислений

Если в выражении расставлены скобки, то вычисления производятся в порядке, определенном в математике: чем меньше глубина вложенности скобок, тем позже вычисляется заключенная в них операция. Если же скобки отсутствуют, то сначала вычисляются значения операций с более высоким приоритетом, затем - с менее высоким. Несколько подряд идущих операций одного приоритета вычисляются в последовательности "слева направо".

	Операции	**Приоритет**
Унарные операции	+, -, not	Первый(высший)
Операции, эквивалентные умножению	*, /, div, mod, and	Второй
Операции, эквивалентные сложению	+,-, or, xor	Третий
Операции сравнения	=, <>, >, <, <=, >=, in	Четвертый

Примечание: вызов любой функции имеет более высокий приоритет, чем все внешние относительно этого вызова операции. Выражения, являющиеся аргументами вызываемой функции, вычисляются в момент вызова.

Совместимость типов данных

В общем случае при выполнении арифметических (и любых других) операций компилятору требуется, чтобы типы операндов совпадали: нельзя, например, сложить массив и множество, нельзя передать вещественное число функции, ожидающей целый аргумент, и т.п.

В то же время, любая процедура или функция, написанная в расчете на вещественные значения, сможет работать и с целыми числами.

Правила, по которым различные типы данных считаются взаимозаменяемыми, мы приводим ниже.

Эквивалентность

Эквивалентность - это наиболее высокий уровень соответствия типов.

Два типа - T1 и T2 - будут эквивалентными, если верен хотя бы один вариант из перечисленных ниже:

— T1 и T2 совпадают;
— T1 и T2 определены в одном объявлении типа;
— T1 эквивалентен некоторому типу T3, который эквивалентен типу T2.

Пример:

```
type     T2 = T1;
         T3 = T1;
         T4,T5 = T2;
```

Здесь эквивалентными будут T1 и T2; T1 и T3; T1 и T4; T1 и T5; T4 и T5. В то время как T2 и T3 - не являются эквивалентными.

Совместимость

Совместимость типов требуется при конструировании выражений, а также при вызовах подпрограмм (для параметров-значений). Совместимость означает, что для переменных этих типов возможна операция присваивания - хотя во время этой операции присваиваемое значение может измениться: произойдет неявное приведение типов данных.

Два типа T1 и T2 будут совместимыми, если верен хотя бы один вариант из перечисленных ниже:

— T1 и T2 эквивалентны (в том числе совпадают);
— T1 и T2 - оба целочисленные или оба вещественные;
— T1 и T2 являются подмножествами одного типа;
— T1 является некоторым подмножеством T2;
— T1 - строка, а T2 - символ.

Совместимость по присваиванию

В отличие от простой совместимости, совместимость по присваиванию гарантирует, что в тех случаях, когда производится какое-либо присваивание (используется запись вида a:=b; или происходит передача значений в подпрограмму или из нее и т.п.), не произойдет никаких изменений присваиваемого значения.

Два типа данных T1 и T2 называются совместимыми по присваиванию, если выполняется хотя бы один вариант из перечисленных ниже:

- T1 и T2 эквивалентны;
- T1 и T2 совместимы, причем T2 - некоторое подмножество в T1;
- T1 - вещественный тип, а T2 - целый.

Приведение типов данных

Неявное приведение типов данных

Как мы упомянули выше, тип результата арифметических операций (а следовательно, и выражений) может отличаться от типов исходных операндов. Например, при "дробном" делении (операция /) одного целого числа на другое целое в ответе все равно получается вещественное. Такое изменение типа данных называется неявным приведением типов.

Если в некоторой операции присваивания участвуют два типа данных совместимых, но не совместимых по присваиванию, то тип присваиваемого выражения автоматически заменяется на подходящий. Это тоже неявное приведение. Причем в этих случаях могут возникать изменения значений. Скажем, если выполнить такую последовательность операторов

```
a:= 10;           {здесь a: byte}
a:= -a;
writeln(a);
```

то на экране мы увидим не -10, а 246 (246 = 256 - 10).

Явное приведение типов данных

Тип значения можно изменить и явным способом: просто указав новый тип выражения, например: a:= byte(b). В этом случае переменной a будет присвоено значение, полученное новой интерпретацией значения переменной b. Скажем, если b имеет тип shortint и значение -23, то в a запишется 233 (= 256 - 23).

Приводить явным образом можно и типы, различающиеся по длине. Тогда значение может измениться в соответствии с новым типом. Скажем, если преобразовать тип longint в тип integer, то возможны потери из-за отсечения первых двух байтов исходного числа. Например, результатом попытки преобразовать число 100 000 к типу integer станет число 31 072, а к типу word - число 34 464.

Функции, изменяющие тип данных

В заключение мы рассмотрим список стандартных функций, аргумент и результат которых принадлежат к совершенно различным типам данных:

```
trunc: real -> integer;
round: real -> integer;
val: string -> byte/integer/real;
chr: byte -> char;
ord: <порядковый_тип> -> longint;
```

Операторы ветвления. Массивы. Циклы

Операторы ветвления

Практически в каждой задаче требуется выбрать необходимые действия из двух, либо из некоторого множества действий (с числом элементов более двух). Для решения этой задачи служат операторы ветвления.

Условный оператор if

Оператор if выбирает между двумя вариантами развития событий:

```
if <условие>
  then <один_оператор>
  [else <один_оператор>];
```

Обратите внимание, что перед словом else (когда оно присутствует) символ ";" не ставится, поскольку он разорвал бы оператор на две части.

Условный оператор if работает следующим образом:

1. Сначала вычисляется значение <условия> – это может быть любое выражение, возвращающее значение типа boolean.
2. Затем, если в результате получена "истина" (true), то выполняется оператор, стоящий после ключевого слова then, а если "ложь" (false) – без дополнительных проверок выполняется оператор, стоящий после ключевого слова else. Если же else-ветвь отсутствует, то не выполняется ничего и управление переходит на следующий по порядку программы оператор.

Оператор If-Then-Else допускает многократное вложение друг в друга. В случае, когда каждый оператор if имеет собственную else-ветвь, все будет в порядке. А вот если некоторые из них этой ветви не имеют, может возникнуть ошибка. Компилятор языка Pascal всегда считает, что else относится к самому ближнему оператору if. Таким образом, если написать

```
if i>0 then if s>2
             then s:= 1
            else s:= -1;
```

подразумевая, что else-ветвь относится к внешнему оператору if, то компилятор все равно воспримет эту запись как

```
if i>0 then if s>2
            then s:= 1
            else s:= -1
       else;
```

Очевидно, что в этом случае правильного результата ожидать не приходится.

Для того чтобы избежать подобных ошибок, стоит всегда (или по крайней мере при наличии нескольких вложенных условных операторов) указывать оба ключевых слова, даже если одна из ветвей будет пустовать. Так вы застрахуетесь от одной из частых ошибок по невнимательности, которые очень сложно найти в процессе отладки программы. Альтернативным решением является обрамление вложенного оператора If-Then-Else в операторные скобки begin-end.

Итак, исходный вариант нужно переписать следующим образом:

```
if i>0 then if s>2
            then s:=1
            else
       else s:=-1;
```

либо так:

```
if i>0 then begin
                if s>2 then s:=1
              end
        else s:=-1;
```

Вообще же, если есть возможность переписать несколько вложенных условных операторов как один оператор выбора, это стоит сделать.

Оператор выбора case

Оператор case позволяет сделать выбор между несколькими вариантами:

```
case <переключатель> of
          <список_констант> : <один_оператор>;
          [<список_констант> : <один_оператор>;]
          [<список_констант> : <один_оператор>;]
          [else <один_оператор>;]
end;
```

Замечание: Обратите внимание, что после else двоеточие не ставится.

Существуют дополнительные правила, относящиеся к структуре этого оператора:

1. Переключатель должен относиться только к порядковому типу данных, но не к типу longint.
2. Переключатель может быть переменной или выражением.
3. Список констант может задаваться как явным перечислением, так и интервалом или их объединением.
4. Повторение констант не допускается.
5. Тип переключателя и типы всех констант должны быть совместимыми).

Пример оператора выбора:

```
case symbol(* :char *) of
    'a'..'z','A'..'Z' : writeln('Это латинская буква');
    'а'..'я','А'..'Я' : writeln('Это русская буква');
    '0'..'9' :          writeln('Это цифра');
    ' '      :          writeln('Это пробел');
    else                 writeln('Это спец. символ');
end;
```

Выполнение оператора case происходит следующим образом:

1. вычисляется значение переключателя;
2. полученный результат проверяется на принадлежность к тому или иному списку констант;

3. если такой список найден, то дальнейшие проверки уже не производятся, а выполняется оператор, соответствующий выбранной ветви, после чего управление передается оператору, следующему за ключевым словом end, которое закрывает всю конструкцию case.
4. если подходящего списка констант нет, то выполняется оператор, стоящий за ключевым словом else. Если else-ветви нет, то не выполняется ничего, а управление передается следующему за end оператору программы.

Иллюстрация if и case

В качестве примера, иллюстрирующего использование операторов ветвления, приведем несколько различных реализаций функции sgn(x) – определяющей знак числа x. Из математики известно, что эта функция имеет следующие значения:

```
sgn(x) = -1, если  x < 0;
sgn(x) = 0, если  x = 0;
sgn(x) = 1, если  x > 0.
```

Реализовать эту функцию для случая, когда x вещественное, можно следующими способами (при условии, что x: real; sgn: -1..1;):

```
if x=0 then sgn:= 0;
if x<0 then sgn:= -1;
if x>0 then sgn:= 1;
```

Это прямая реализация, построенная на определении математической функции. Здесь нет никаких хитростей и никаких попыток оптимизации: даже если сработает первый вариант, второй и третий все равно будут проверены, невзирая на то, что результат уже получен.

```
if x=0
   then sgn:= 0
   else if x<0 then sgn:= -1
               else sgn:= 1;
```

Второй вариант свободен от излишних проверок в том случае, если значение переменной не положительно. Эту реализацию следует признать более эффективной, чем предыдущая.

```
if x=0
   then sgn:=0
   else sgn:=x/abs(x);
```

Успешная попытка сократить текст программы. Здесь используется стандартная функция abs(), которая возвращает абсолютное значение аргумента. Проблема в данном случае состоит в том, что "/" - деление дробное, но ведь нам необходим целый, а не вещественный ответ! Можно использовать функцию округления

```
if x=0
 then sgn:=0
 else sgn:=round(x/abs(x));
```

Этот вариант будет выдавать верный результат.

```
case x=0 of
 true:  sgn:=0;
 false:
   sgn:=round(x/abs(x));
end;
```

Как вариант можно использовать оператор выбора case. Вся хитрость этого варианта в том, что выбирающий ветви переключатель обязан принадлежать к перечислимому типу, именно поэтому пришлось заменить "x" на "x = 0". Напомним, что эта операция сравнения выдает результат логического типа boolean, и именно логические константы true и false фигурируют в качестве меток выбора.

Здесь разобраны далеко не все возможные способы реализации функции sgn(x), однако приведенные примеры показывают, что способов решить задачу всегда больше, чем один, и вряд ли самое простое решение будет и оптимальным.

Массивы

Массив это структурированный тип данных, предназначенный для хранения нескольких однотипных элементов.

Для того чтобы задать массив, необходимо в разделе описания переменных (var) указать его размеры и тип его компонент.

Общий вид описания (одномерного) массива:

```
var   array[<тип_индексов>] of <тип_компонент>;
```

Чаще всего это трактуется так:

```
Var array[<лев_гран>..<прав_гран>] of <тип_компонент>;
```

Например, одномерный (линейный) массив, состоящий не более чем из 10 целых чисел, можно описать следующим образом:

```
var  W: array [1..10] of integer;
```

Нумерация компонент массива не обязана начинаться с 1 или с 0 – возможно описывать массив, пронумерованный любыми целыми числами. Необходимо лишь, чтобы номер последней компоненты был больше, чем номер первой:

```
var YY: array [-99..33] of integer;
```

Собственно говоря, нумеровать компоненты массива можно не только целыми числами. Любой порядковый тип данных (перечислимый, интервальный, символьный, логический, а также произвольный тип,

созданный на их основе) имеет право выступать в роли нумератора. Таким образом, допустимы следующие описания массивов:

```
type ch = 'a','c'..'z'; (- пропущен символ "b")
var  a: array [ch] of integer;
     c: array [shortint] of real;
```

Общий размер массива не должен превосходить 65 520 байт. Следовательно, попытка задать массив

```
z: array [integer] of  byte;
```

не увенчается успехом, поскольку тип integer покрывает 65 535 различных элементов. А про тип longint в данном случае лучше и вовсе не вспоминать.

Тип компонент массива может быть любым:

```
var  ar: array[10..20] of real;
     ac: array[1..5] of char;
     aa: array[1..100] of array[1..100] of integer;
```

Для краткости и удобства многомерные массивы можно описывать и более простым способом:

```
var bb: array[1..10,1..20] of real;
    zx: array[1..2, -1..1,1..10] of word;
```

Общее ограничение на размер массива - не более 65 520 байт - сохраняется и для многомерных массивов. Количество компонент многомерного массива вычисляется как произведение всех его "измерений". Таким образом, в массиве bb содержится 200 компонент, а в массиве zx – 60 компонент.

Описание переменных размерностей

Если ваша программа должна обрабатывать матрицы переменных размерностей (например, N по горизонтали и M по вертикали), то существует проблема изначального описания такого массива, ведь в разделе var не

допускается использование переменных. Следовательно, самый логичный, казалось бы, вариант

```
var m,n: integer;
    a: array[1..m,1..n] of real;
```

является неверным.

Предположим, однако, что вам известны максимальные границы, в которые могут попасть индексы обрабатываемого массива. Скажем, N и M заведомо не могут превосходить 100. Тогда можно выделить место под наибольший возможный массив, а реально работать только с малой его частью:

```
const nnn=100;
var a: array[1..nnn,1..nnn] of real;
    m,n: integer;
```

Обращение к компонентам массива

Массивы относятся к структурам прямого доступа. Это означает, что возможно напрямую (не перебирая предварительно все предшествующие компоненты) обратиться к любой интересующей нас компоненте массива.

Доступ к компонентам линейного массива осуществляется так:

```
<имя_массива>[<индекс_компоненты>]
```

а многомерного - так:

```
<имя_массива>[<индекс>,_,<индекс>]
```

или так

```
<имя_массива>[<индекс>][<индекс>][...[<индекс>]]
```

Правила употребления индексов при обращении к компонентам массива таковы:

1. Индекс компоненты может быть константой, переменной или выражением, куда входят операции и вызовы функций.

2. Тип каждого индекса должен быть совместим с типом, объявленным в описании массива именно для соответствующего "измерения"; менять индексы местами нельзя.
3. Количество индексов не должно превышать количество "измерений" массива. Попытка обратиться к линейному массиву как к многомерному обязательно вызовет ошибку. А вот обратная ситуация вполне возможна: например, если вы описали N-мерный массив, то его можно воспринимать как линейный массив, состоящий из (N-1)-мерных массивов.

Примеры использования компонент массива:

```
ar[5]:= aa[1]+1;
zx[1,0,4]:= 5;
bb[i+j]:= aa[i+j];
```

Задание массива константой

Для того чтобы не вводить массивы вручную во время отладки программы, входные данные можно задавать прямо в тексте программы при помощи типизированных констант.

Если массив линейный (вектор), то начальные значения для компонент этого вектора задаются через запятую, а сам вектор заключается в круглые скобки.

Многомерный массив также можно рассматривать как линейный, предполагая, что его компонентами служат другие массивы. Таким образом, для системы вложенных векторов действует то же правило задания типизированной константы: каждый вектор ограничивается снаружи круглыми скобками.

Исключение составляют только массивы, компонентами которых являются величины типа char. Такие массивы можно задавать проще – строкой символов.

Примеры задания массивов типизированными константами:

```
type  mass = array[1..3,1..2] of byte;
const a: array[-1..1] of byte = (0,0,0); {линейный}
      b: mass = ((1,2),(3,4),(5,6)); {двумерный}
      s: array[0..9] of char = '0123456789';
```

Замечание: Невозможно задать неименованную или нетипизированную константу, относящуюся к типу данных array.

Операторы циклов

Для того чтобы обработать несколько однотипных элементов, совершить несколько одинаковых действий и т.п., следует воспользоваться оператором цикла, который наилучшим образом подходит к поставленной задаче.

Оператор цикла повторяет некоторую последовательность операторов заданное число раз, которое может быть определено как заранее, так и динамически – уже во время работы программы.

Замечание: Алгоритмы, построенные только с использованием циклов, называются итеративными – от слова итерация, которое обозначает повторяемую последовательность действий.

Циклы for-to и for-downto (циклы с параметром)

В случае, когда количество однотипных действий заранее известно (например, необходимо обработать все компоненты массива известной размерности), стоит отдать предпочтение циклу с параметром (for).

Общий вид оператора for-to:

```
for i:= first to last do <оператор>;
```

Здесь счетчик i (переменная), нижняя граница first (переменная, константа или выражение) и верхняя граница last (переменная, константа или выражение) должны относиться к эквивалентным порядковым типам данных. Если тип нижней или верхней границы не эквивалентен типу счетчика, а

лишь совместим с ним, то осуществляется неявное приведение: значение границы преобразуется к типу счетчика, в результате чего возможны ошибки.

Цикл for-to работает следующим образом:

1. вычисляется значение верхней границы last;
2. переменной i присваивается значение нижней границы first;
3. производится проверка того, что i<=last;
4. если это так, то выполняется <оператор> ;
5. значение переменной i увеличивается на единицу;
6. пункты 3-5, составляющие одну итерацию цикла, выполняются до тех пор, пока i не станет строго больше, чем last; как только это произошло, выполнение цикла прекращается, а управление передается следующему за ним оператору.

Из этой последовательности действий можно понять, какое количество раз отработает цикл for-to в каждом из трех случаев:

— first < last: цикл будет работать last-first+1 раз;

— first = last: цикл отработает ровно один раз;

— first > last: цикл вообще не будет работать.

После окончания работы цикла переменная-счетчик может потерять свое значение. Таким образом, нельзя с уверенностью утверждать, что после того, как цикл завершил работу, обязательно окажется, что i=last+1. Поэтому попытки использовать переменную-счетчик сразу после завершения цикла (без присваивания ей какого-либо нового значения) могут привести к непредсказуемому поведению программы при отладке.

Существует аналогичный вариант цикла for, который позволяет производить обработку не от меньшего к большему, а в противоположном направлении:

```
for i:= first downto last do <оператор>;
```

Счетчик i (переменная), верхняя граница first (переменная, константа или выражение) и нижняя граница last (переменная, константа или

выражение) должны иметь эквивалентные порядковые типы. Если тип нижней или верхней границы не эквивалентен типу счетчика, а лишь совместим с ним, то осуществляется неявное приведение типов.

Цикл for-downto работает следующим образом:

1. переменной i присваивается значение first ;
2. производится проверка того, что i>=last ;
3. если это так, то выполняется <оператор> ;
4. значение переменной i уменьшается на единицу;
5. пункты 2-4 выполняются до тех пор, пока i не станет меньше, чем last ; как только это произошло, выполнение цикла прекращается, а управление передается следующему за ним оператору.

Если при этом

— first < last, то цикл вообще не будет работать;

— first = last, то цикл отработает один раз;

— first > last, то цикл будет работать first-last+1 раз.

Замечание о неопределенности значения счетчика после окончания работы цикла справедливо и в этом случае.

Циклы с условием (while-do и repeat-until)

Если заранее неизвестно, сколько раз необходимо выполнить тело цикла, то удобнее всего пользоваться циклом с предусловием (while-do) или циклом с постусловием (repeat-until).

Общий вид этих операторов таков:

```
while <условие_1> do <оператор>;
repeat <операторы> until <условие_2>;
```

Условие окончания цикла может быть выражено переменной, константой или выражением, имеющим логический тип.

Замечание: Обратите внимание, что на каждой итерации циклы for и while выполняют только по одному оператору (либо группу операторов,

заключенную в операторные скобки begin-end и потому воспринимаемую как единый составной оператор). В отличие от них, цикл repeat-until позволяет выполнить сразу несколько операторов: ключевые слова repeat и until сами выступают в роли операторных скобок.

Последовательности действий при выполнении этих циклов таковы:

While-do	**repeat-until**
1. Проверяется, истинно ли <условие_1>.	1. Выполняются <операторы>.
2. Если это так, то выполняется <оператор>.	2. Проверяется, ложно ли <условие_2>
3. Пункты 1 и 2 выполняются до тех пор, пока <условие_1> не станет ложным.	3. Пункты 1 и 2 выполняются до тех пор, пока <условие_2> не станет истинным.

Таким образом, если <условие_1> изначально ложно, то цикл while не выполнится ни разу. Если же <условие_2> изначально истинно, то цикл repeat-until выполнится один раз. Это отличительная особенность цикла repeat-until – он выполняется как минимум один раз.

Операторы прерывания цикла

Существует возможность прервать выполнение цикла (или одной его итерации), не дождавшись конца его (или ее) работы. Для этого внутрь цикла (обычно в одну из ветвей условного оператора) помещают оператор break или continue. Их действия такокы:

— оператор break прерывает работу всего цикла и передает управление на следующий за ним оператор.

— оператор continue прерывает работу текущей итерации цикла и передает управление следующей итерации (цикл repeat-until) или на предшествующую ей проверку (циклы for-to, for-downto, while).

Замечание: При прерывании работы циклов for-to и for-downto с помощью функции break переменная цикла (счетчик) сохраняет свое текущее значение, не "портится".

Пример использования циклов

Распечатать двумерный массив размерности MxN удобным для пользователя способом. (Известно, что массив содержит только целые числа из промежутка [0..100].)

```
for i:= 1 to n do
begin
    for j:= 1 to m do write(a[i,j]:4);
    writeln;
end;
```

Сортировка массивов

Сортировкой называется задача упорядочивания данных в массиве по возрастанию или убыванию.

Часто нужно упорядочить предметы по какому-то признаку: записать данные числа в порядке возрастания, слова — по алфавиту, людей выстроить по росту. Если можно сравнить любые два предмета из данного набора, то этот набор всегда можно упорядочить. Процесс упорядочивания информации и называют «сортировкой»

Необходимость отсортировать какие-либо величины возникает в программировании очень часто. К примеру, входные данные подаются в произвольном состоянии, а программа работает с упорядоченной последовательностью. Существуют ситуации, когда предварительная сортировка данных позволяет сократить содержательную часть алгоритма в разы, а время его работы – уменьшить в десятки раз.

Однако верно и обратное. Сколь бы хорошим и эффективным ни был выбранный вами алгоритм, но если в качестве подзадачи он использует "плохую" сортировку, то вся работа по его оптимизации оказывается бесполезной. Неудачно реализованная сортировка входных данных способна заметно понизить эффективность алгоритма в целом.

Методы упорядочения подразделяются на внутренние (обрабатывающие массивы) и внешние (занимающиеся только файлами).

Их принципиальное отличие заключается в том, что для алгоритмов внутренней сортировке весь массив упорядочиваемых данных находится в оперативной памяти, а операция внешней сортировки находит применение для упорядочивания огромных массивов, которые не могут быть целиком загружены в оперативную память и считываются с носителей информации отдельными блоками – фрагментами.

В настоящем курсе будем рассматривать только внутренние сортировки. Дополнительная особенность состоит в том, что эти алгоритмы

не требуют дополнительной памяти: вся работа по упорядочению производится внутри одного и того же массива, т.е. путем перестановки элементов.

Простые сортировки

К простым внутренним сортировкам относят методы, сложность которых пропорциональна квадрату размерности входных данных.

Количество действий, необходимых для упорядочения некоторой последовательности данных, в общем случае, зависит не только от длины этой последовательности, но и от ее структуры. Например, если на вход алгоритма сортировки подается уже упорядоченная последовательность, то количество действий будет значительно меньше, чем в случае произвольно перемешанных входных данных.

Как правило, сложность алгоритмов подсчитывают раздельно по количеству сравнений и по количеству перемещений данных в памяти (пересылок), поскольку выполнение этих операций занимает различное время. Однако точные значения удается найти редко, поэтому для оценки алгоритмов ограничиваются лишь понятием "пропорционально", которое не учитывает конкретные значения констант, входящих в итоговую формулу. Общую же эффективность алгоритма обычно оценивают "в среднем": как среднее арифметическое от сложности алгоритма "в лучшем случае" и "в худшем случае", то есть (Eff_best + Eff_worst)/2.

Сортировка "пузырьком"

Аналогию сортировки этим методом можно наблюдать, рассматривая поведения пузырька воздуха в воде – он постепенно поднимается вверх. Это явление связано с тем, что плотность воздуха намного меньше плотности воды.

Суть метода сортировки пузырьком заключается в следующем. Последовательно просматриваются все пары рядом стоящих элементов

массива и, если они расположены в неправильном порядке, они меняются местами. Это правило действует до тех пор, пока все пары не будут стоять в правильном порядке (то есть, по возрастанию или убыванию).

Приведем текст программы, реализующий метод сортировки "пузырьком"

```
{входные параметры}
const N=10; {Количество элементов массива}
var a: array[1..N] of integer; {массив}
    i: integer; {счётчик для цикла}
    f: boolean; {Признак наличия неупорядоченных пар}
    c: integer; {промежуточная переменная}
{алгоритм сортировки}
repeat
  f:=false; {Пока неупорядоченных пар не было}
{Просматриваем все пары рядом стоящих элементов}
  for i:=1 to N-1 do
  begin
    if a[i]>a[i+1] then {Если порядок пары неверный}
    begin
      f:=true; {Есть неупорядоченная пара}
      c:=a[i];        {Меняем местами }
      a[i]:=a[i+1];   {эти элементы}
      a[i+1]:=c;
    end;
  end;
until not f; { пока не исчезнут все неправильные пары}
```

Метод сортировки "пузырьком" является одним из самых простых и, одновремнно, самых медленных методов.

Следующий метод обладает такими же характеристиками.

Метод простого выбора

Суть метода заключается в следующем. Находится минимальный элемент массива и записывается в первую ячейку массива, содержимое которой записывается на место найденного минимального элемента. После чего находится минимальный элемент массива, начиная со второго элемента,

он записывается во вторую ячейку массива, содержимое которой записывается на место найденного минимального элемента. Таким образом, постепенно выстраивается упорядоченный массив. На языке Pascal это выглядит так:

```
{входные параметры}
const N=10; {Количество элементов массива}
var a: array[1..N] of integer; {массив}
    i,j: integer; {счётчики для цикла}
    c:   integer; {Промежуточная переменная}
    c2:  integer; {Промежуточная переменная}
{алгоритм сортировки}
for i:=1 to N-1 do begin
  {цикл по первому обрабатываемому элементу массива}
  c2:=i; {индекс предполагаемого минимального элемента}
  for j:=i+1 to N do
    {поиск минимального элемента}
    if a[c2]>a[j] then c2:=j;
        {если в с2 индекс не минимального элемента,
        то в с2 записывается индекс меньшего элемента}
    c:=a[i];
    a[i]:=a[c2];
    a[c2]:=c; {Меняем местами элемент массива}
end;
```

Сортировка простыми вставками

Один из простых способов сортировки – это упорядочение данных по мере их поступления. В этом случае при вводе каждого нового значения можно опираться на тот факт, что все предыдущие элементы уже образуют отсортированную последовательность.

На каждом шаге алгоритма мы выбираем один из элементов входных данных и вставляем его на нужную позицию в уже отсортированном списке, до тех пор, пока набор входных данных не будет исчерпан. Метод выбора очередного элемента из исходного массива произволен; может использоваться практически любой алгоритм выбора. Обычно (и с целью получения устойчивого алгоритма сортировки), элементы вставляются по

порядку их появления во входном массиве. Приведенный ниже алгоритм использует именно эту стратегию выбора:

1. Первый элемент записать "не раздумывая".
2. Пока не закончится последовательность вводимых данных, для каждого нового ее элемента выполнять следующие действия:
   - начав с конца уже существующей упорядоченной последовательности, все ее элементы, которые больше, чем вновь вводимый элемент, сдвинуть на 1 шаг назад;
   - записать новый элемент на освободившееся место.

Реализация на языке Pascal может выглядеть так:

```
{входные параметры}
const N=10; {Количество элементов массива}
var a: array[1..N] of integer; {массив}
    i,j: integer; {счётчики для цикла}
    c:   integer; {Промежуточная переменная}
{алгоритм сортировки}
for i := 2 to N do
begin
   c := a[i];
   j := i - 1;
   while (j >= 1) and (a[j] > c) do
   begin
     a[j + 1] := a[j];
     j := j - 1;
   end;
   a[j + 1] := c;
end;
```

Улучшенные сортировки

В отличие от простых сортировок, имеющих сложность ~N2, к улучшенным сортировкам относятся алгоритмы с общей сложностью ~N*logN.

Необходимо, однако, отметить, что на небольших наборах сортируемых данных (N<100) эффективность быстрых сортировок не столь очевидна: выигрыш становится заметным только при больших N.

Следовательно, если необходимо отсортировать маленький набор данных, то выгоднее взять одну из простых сортировок.

Сортировка Шелла

Эта сортировка базируется на уже известном нам алгоритме простых вставок. Смысл ее состоит в раздельной сортировке методом простых вставок нескольких частей, на которые разбивается исходный массив. Эти разбиения помогают сократить количество пересылок: для того, чтобы освободить "правильное" место для очередного элемента, поскольку приходится уже сдвигать меньшее количество элементов.

Алгоритм выглядит следующим образом. На каждом шаге (пусть переменная t хранит номер этого шага) нужно произвести следующие действия:

1. вычленить все подпоследовательности, расстояние между элементами которых составляет k_t;
2. каждую из этих подпоследовательностей отсортировать методом простых вставок.

Нахождение убывающей последовательности расстояний k_t, $k_{t-1...}$, k_1 составляет главную проблему этого алгоритма. Многочисленные исследования позволили выявить ее обязательные свойства:

— $k_1 = 1$;
— для всех t: $k_t > k_{t-1}$;
— желательно также, чтобы все k_t не были кратными друг другу (для того, чтобы не повторялась обработка ранее отсортированных элементов).

Дональд Кнут предлагает две "хорошие" последовательности расстояний:

```
1, 4, 13, 40, 121, _ (kt  = 1+3*kt-1)
1, 3, 7, 15, 31, _  (kt  = 1+2*kt-1 = 2t -1)
```

Первая из них подходит для сортировок достаточно длинных массивов, вторая же более удобна для коротких. Поэтому мы будем рассматривать второй случай.

Как же определить начальное значение для t (а вместе с ним, естественно, и для k_t)?

Можно, конечно, шаг за шагом проверять, возможно ли вычленить из сортируемого массива подпоследовательность (хотя бы длины 2) с расстояниями 1, 3, 7, 15 и т.д. между ее элементами. Однако такой способ довольно неэффективен. Мы поступим иначе, ведь у нас есть формула для вычисления $k_t = 2^t - 1$.

Итак, длина нашего массива (N) должна попадать в такие границы:

k_t <= N -1 < k_{t+1}

или, что то же самое,

2^t <= N < 2^{t+1}

Прологарифмируем эти неравенства (по основанию 2):

```
t <= log N < t+1
```

Таким образом, стало ясно, что t можно вычислить по следующей формуле:

```
t = trunc(log N))
```

К сожалению, язык Pascal предоставляет возможность логарифмировать только по основанию e (натуральный логарифм). Поэтому нам используем правило "превращения" логарифмов:

$\log_m x = \log_z x / \log_z m$

В нашем случае m = 2, z = e. Таким образом, для начального t получаем:

```
t:= trunc(ln(N)/ln(2)).
```

Однако при таком t часть подпоследовательностей будет иметь длину 2, а часть – и вовсе 1. Сортировать такие подпоследовательности незачем, поэтому стоит сразу же отступить еще на 1 шаг:

```
t:= trunc(ln(N)/ln(2))-1
```

Расстояние между элементами в любой подпоследовательности вычисляется так:

```
k:= (1 shl t)-1;          {k= 2^t-1}
```

Количество подпоследовательностей будет равно в точности k. В самом деле, каждый из первых k элементов служит началом для очередной подпоследовательности. А дальше, начиная с (k+1)-го, все элементы уже являются членами некоторой, ранее появившейся подпоследовательности, значит, никакая новая подпоследовательность не сможет начаться в середине массива.

Сколько же элементов будет входить в каждую подпоследовательность? Ответ таков: если длину всей сортируемой последовательности (N) можно разделить на шаг k без остатка, тогда все подпоследовательности будут иметь одинаковую длину, а именно:

```
s:= N div k;
```

Если же N не делится на шаг k нацело, то первые p подпоследовательностей будут длиннее на 1. Количество таких "удлиненных" подпоследовательностей совпадает с длиной "хвоста" - остатка от деления N на шаг k:

```
P:= N mod k;
```

На языке Pascal сортировка Шелла описывается следующим кодом.

```
{входные параметры}
const N=10; {Количество элементов массива}
var a: array[1..N] of integer; {массив}
    i,j: integer; {счётчики для цикла}
```

```
    m,x,s,p,t,k,r: integer; {промежуточные переменные}
{алгоритм сортировки}
t:= trunc(ln(n)/ln(2));
repeat
 t:= t-1;
 k:= (1 shl t)-1;
 p:= n mod k;
 s:= n div k;
 if p=0 then p:= k
        else s:= s+1;
for i:= 1 to k do
{и длинные, и короткие подпоследовательности}
begin
    if i= p+1 then s:= s-1;
    (для коротких - уменьшаем     длину}
    for j:= 1 to s-1 do  {метод простых вставок,шаг=k}
       if a[i+(j-1)*k]>a[i+j*k]
            then begin
                x:= a[i+j*k];
                m:= i+(j-1)*k;
                while (m>0) and (a[m]>x) do
                begin
                    a[m+k]:= a[m];
                    m:= m-k;
                end;
                a[m+k]:= x;
            end;
    end;
until k=1;
```

Довольно сложными методами, в изложение которых мы не будем углубляться, показано, что алгоритм Шелла имеет сложность ~N3/2. И хотя это несколько хуже, чем N*logN, все-таки эта сортировка относится к улучшенным.

Существует множество других специальных методов сортировки, отличающихся быстротой, но их рассмотрение выходит за рамки лекции.

Символы и строки

Предыдущие лекции были посвящены массивам, состоящим из числовых значений. Рассмотрим массивы специального вида – массивы, состоящие только из символов. Условно их можно назвать строками.

Строковые переменные описываются в разделе var следующим образом:

```
var <имя_строки>: string[ [<длина>] ];
```

Здесь необязательный параметр длина представляет собой положительное целое число, не превышающее 255.

Компоненты этого массива нумеруются с нуля, причем нулевая ячейка хранит информацию об общей длине строки.

Если параметр длины в объявлении строки не указан, то принимается, что строка максимальной длины, т.е. 255 символов.

Примеры описаний:

```
var s1: string[15];   (*строка длиной 15 символов*)
    s2: string;       (*строка длиной 255 символов*)
```

Необходимо отметить, что один символ и строка длиной в один символ

```
var c: char;
    s: string[1];
```

совершенно не эквивалентны друг другу. Вне зависимости от своей реальной длины, строка относится к конструируемым структурированным типам данных, а не к базовым порядковым (лекция 5).

Символ-константа и строка-константа

Неименованные константы

В тексте программы на языке Pascal последовательность любых символов, заключенная в апострофы, воспринимается как символ или строка. Например:

```
c:='w';       {c: char}
s:='SSAU';    {s: string}
```

Константе автоматически присваивается "минимальный" тип данных, достаточный для ее представления: `char` или `string[k]`. Поэтому попытка написать

```
c:='zzz'; {c: char}
```

вызовет ошибку уже на этапе компиляции программы.

Кроме того, если константа длиннее той строковой переменной, куда программа пытается ее записать, то в момент присваивания произойдет усечение (отбрасывание правой части) ее до нужной длины.

Пустая строка задается двумя последовательными апострофами:

```
st:= '';
```

Если же необходимо сделать так, чтобы среди символов строки содержался и сам апостроф, его нужно удвоить:

```
s:='Don''t worry about the apostrophe!';
```

Если теперь вывести на экран эту строку, то получится следующее:

```
Don't worry about the apostrophe!
```

Нетипизированные константы

Все правила задания символов и строк как неименованных констант остаются в силе и при задании именованных нетипизированных констант в специальном разделе `const`. Например:

```
const   c3 = '''';   {это один символ - апостроф}
        s3 = 'This is a string';
```

Типизированные константы

Типизированная константа, которая будет иметь тип `char` или `string`, задается в разделе `const` следующим образом:

```
const   c4: char = ''''; {это один символ - апостроф}
        s4: string[20] = 'This is a string';
```

Действия с символами

Операции

Результатом унарной операции

#<положительная_неименованная_целочисленная_константа>

является символ, код которого в таблице ASCII соответствует заданному числу. Например,

```
#100  = 'd'
#39   = '''' {апостроф}
#232  = 'ш'
#1000 = 'ш'  {потому что (1000 mod 256)= 232}
```

Кроме того, к символьным переменным, как и к значениям всех порядковых типов данных, применимы операции сравнения <, <>, >, =, результат которых также опирается на номера символов из таблицы ASCII.

Стандартные функции

Функция `chr(k:byte):char` "превращает"; номер символа в символ. Действие этой функции аналогично действию операции `#`. Например:

```
c:= chr(100); {c: char}
{c = 'd'}
```

Обратной к функции `chr()` является уже изученная нами функция `ord()`. Таким образом, для любого числа `k` и для любого символа `c`

```
ord(chr(k)) = k       и       chr(ord(c)) = c
```

Отметим, что стандартные процедуры и функции `pred()`, `succ()`, `inc()` и `dec()`, определенные для значений любого порядкового типа, применимы также и к символам (значениям порядкового типа данных `char`). Например:

```
pred('[')  = 'Z'
succ('z')  = '{'
inc('a')   = 'b'
inc('c',2) = 'e'
dec('z')   = 'y'
dec(#0,4)  = '№' {#252}
```

Стандартная функция `upcase(c: char):char` превращает строчную букву в прописную. Символы, не являющиеся строчными латинскими буквами, остаются без изменения (к сожалению, в их число попадают и все русские буквы).

Стандартные функции и процедуры обработки строк

Для обработки символьных массивов, которыми являются строки, в языке Pascal существуют специальные подпрограммы:

Функция `concat(s1,...,sN:string):string` осуществляет склеивание (конкатенацию) всех перечисленных строк или символов в

указанном порядке. Если длина итоговой строки больше 255-ти символов, то произойдет отсечение "хвоста". Кроме того, даже если результат конкатенации не был усечен, но программа пытается сохранить его в переменную заведомо меньшей длины, то усечение все равно состоится:

```
concat('AbC','*DeF','%','q','We') = 'AbC*DeF%qWe'
```

Функция `copy(s:string;i,k:byte):string` вычленяет из строки `s` подстроку длиной `k` символов, начиная с `i` -го. Если `i` больше длины строки, то результатом будет пустая строка. Если же `k` больше, чем длина оставшейся части строки, то результатом будет только ее "хвост":

```
copy('abc3de Xyz',2,4)  = 'bc3d'
copy('abc3de Xyz',12,4) = ''
copy('abc3de Xyz',8,14) = 'Xyz'
```

Процедура `delete(s:string;i,k:byte)` удаляет из строки `s` подстроку длиной `k` символов, начиная с `i` -го. Если `i` больше длины строки, то ничего удалено не будет. Если же `k` больше, чем длина оставшейся части строки, то удален будет только ее "хвост":

```
{s = 'abc3de Xyz'}                {s = 'abc3de Xyz'}
delete(s,2,3);                    delete(s,8,13);
{s = 'ade Xyz'}                   {s = 'abc3de '}
```

Процедура `insert(ss,s:string;i:byte)` вставляет подстроку `ss` в строку `s`, начиная с `i` -го символа. Если `i` выходит за конец строки, то подстрока `ss` припишется в конец строки `s` (если результат длиннее, чем допускается для строки `s`, произойдет его усечение):

```
{s = 'abc3de Xyz'}            {s = 'abc3de'}
    insert('xyz',s,2);            insert('xyz',s,12);
    {s = 'axyzbc3de Xyz'}     {s = 'abc3dexyz'}
```

Функция `length(s:string):byte` возвращает длину строки `s`:

```
length('abc3de Xyz') = 10
```

Функция pos(ss,s:string):byte определяет позицию, с которой начинается первое (считая слева направо) вхождение подстроки ss в строку s. Если ss не встречается в s ни разу, функция вернет 0:

```
pos('X', 'abc3de Xyz') = 8
```

Процедура str(x[:w[:d]],s:string) превращает десятичное число x (можно указать, что в этом числе w цифр, из них d дробных) в строку s. Если число короче указанных величин, то спереди и/или сзади оно будет дополнено пробелами:

```
str(156.4:7:2,s);
{s = ' 156.4 '}
```

Процедура val(s:string;i:<арифмет_тип>;err:byte) превращает строку s в десятичное число x (в случае ошибки в переменную err будет записан номер первого недопустимого символа, если ошибки нет, то err=0):

```
{s = '99.17'}
val(s,x,err);
{x = 99.17; err = 0}
```

Операции со строками

Сравнения

Строки – это единственный структурированный тип данных, для элементов которого определен порядок и, следовательно, возможны операции сравнения (=, >, <).

На строках определен так называемый лексикографический порядок: из двух строк меньшей считается та, у которой первый различный символ меньше (подразумевается, что меньше тот символ, у котого ASCII код меньше). Считается, что пустая строка меньше любой другой строки (поскольку содержит единственный символ с кодом #0).

Таким образом, если начальные символы двух сравниваемых строк совпадают, то эта совпадающая часть никак не повлияет на отношение порядка между строками, поэтому ее можно откинуть и сравнивать только первые символы оставшихся подстрок. Если одна из строк полностью совпадает с началом другой, то после удаления совпадающих частей она превратится в пустую строку. Это с очевидностью будет свидетельствовать о том, что начало слова всегда меньше, чем все слово.

Пример:

```
'abc' < 'xyz'
'a' < 'abc'
'1200' < '45'
'Anny' < 'anny'
```

Обращение к компонентам строки

Доступ к `k`-му символу строки осуществляется так же, как к `k` -й компоненте массива (прямые скобки здесь являются обязательным элементом синтаксиса):

```
<имя_строки>[<индекс>]
```

Например:

```
{s = 'w5.17'}
c:= s[3];
{c = '.'}
```

Однако, в отличие от массива, нельзя напрямую заменять символы в строке, то есть действие

```
s[i]:= 'a';
```

хотя и не вызовет ошибки при компиляции, но, скорее всего, не станет работать во время выполнения программы. Для того чтобы изменить символ в строке, нужно воспользоваться стандартными функциями `length()`,

`concat()` и `copy()`. В этом случае простое, казалось бы, действие приходится представлять как последовательность четырех операций:

1. В качестве первой под строки взять из строки `s` символы с `1` -го по (`k-1`)-й:

```
s1:= copy(s,1,k-1);
```

2. В качестве второй под строки взять новое значение заменяемого символа:

```
s2:= new_char;
```

3. В качестве третьей подстроки взять оставшуюся часть строки `s`:

```
s3:= copy(s,k+1,length(s)-k);
```

4. Слить эти строки воедино, а результат записать вместо исходной строки `s`:

```
s:= concat(s1,s2,s3);
```

Или можно объединить все четыре действия в одном операторе:

```
s:=concat(copy(s,1,k-1), new_char,
    copy(s,k+1,length(s)-k));
```

Конкатенация

Единственная операция, которую разрешается производить с переменными строкового типа, – это слияние строк или символов (так называемая конкатенация). Она полностью эквивалентна функции `concat()` и записывается при помощи знака "+". Таким образом, предыдущий оператор можно сделать более простым:

```
s:= copy(s,1,k-1) + new_char + copy(s,k+1,length(s)-k);
```

Структурированный тип данных – запись

Как и массивы, записи являются структурами прямого доступа, однако, в отличие от массивов, могут хранить элементы, относящиеся к разным типам данных.

Таким образом, запись – это такой вектор, компоненты которого (называемые полями) могут относиться к разным типам данных.

Описание записей

В разделе `var` переменную типа запись описывают так:

```
var <имя_записи>:     record
                          <имя_поля1>: <тип_поля1>;
                          [<имя_поля2>: <тип_поля2>;]
                          [...]
                      end;
```

Имена полей должны подчиняться общим правилам построения идентификаторов. Повторение имен полей **внутри одной** записи не допускается.

Замечание: Имена полей могут совпадать с именами других переменных, поскольку на самом деле являются составными:

```
<имя_записи>.<имя_поля>.
```

Поэтому можно записать:

```
var x:   string;
    r:   record
                 x: real;
                 y: real
         end;
```

Поля могут относиться к любым стандартным (базовым или сконструированным) или определенным ранее типам.

Если несколько подряд идущих полей принадлежат к одному типу данных, их описания можно объединить:

```
var <имя_записи>: record
            <имя_поля1>,_,<имя_поляN>: <тип_полей>;
            <имя_поляS>: <тип_поляS>;
            ...
        end;
```

Например:

```
var zap1: record
            x, y: real;
            flag: boolean;
            c: set of 'a'..'z';
            a: array[1..100] of byte;
            data: record
                    day:1..31;
                    month: 1..12;
                    year: 1900..2100;
                end;
        end;
```

Эта запись содержит 6 полей, три из которых сами являются составными (диапазон, массив и запись).

Наиболее распространенный способ представления записей это двумерная таблица, каждый столбец которой имеет свой тип. Такую структуру описывают, например, следующим образом:

```
var tabl: array[1..100] of zap1;
```

Задание записей константой

Как и массивы, записи не могут быть заданы неименованной или нетипизированной константой.

Для того чтобы задать запись типизированной константой, следует вначале описать соответствующий тип в разделе type, а затем воспользоваться им в разделе const:

```
type <имя_типа> = record
                <имя_поля1>: <тип_поля1>;
                [<имя_поля2>: <тип_поля2>;]
                [...]
                end;
```

```
const <имя_конст>: <имя_типа> = <начальное_значение>;
```

Начальное значение для переменной типа запись задается перечислением в круглых скобках начальных значений для всех полей (соответствующих типов!) с обязательным указанием имени задаваемого поля. Имя поля от его начального значения отделяется двоеточием, значения соседних полей разделяются точкой с запятой:

```
(<имя_поля1>:<знач_1>;...; <имя_поляN>: <знач_N>);
```

Например:

```
type data = record
                day:     1..31;
                month:   1..12;
                year:    1900..2100;
            end;
const today: data = (day:13; month:11; year:2013);
```

Допускается не описывать тип константы отдельно, а объединить оба определения:

```
const today: record
            day:     1..31;
            month:   1..12;
            year:    1900..2100;
        end;
        = (day:13; month:11; year:2013);
```

Если описана двумерная таблица, то ее начальные значения задаются как вектор, каждый компонент которого является записью. Таким образом, правила задания типизированной константы-таблицы сочетают в себе правила задания массива и записи:

```
type family = (mother, father, child);
const birthdays : array[family] of data
        = ((day: 18; month: 4; year: 1966),
             (day: 13; month: 7; year: 1961),
             (day: 11; month: 2; year: 2000));
```

Доступ к полям

Обратиться к полю записи можно следующим способом:

```
<имя_записи>.<имя_поля>
```

Например:

```
Month:= Today.Month + 1;
```

При этом, несмотря на одинаковое наименование переменой month и полем записи `Today.`month не возникает никаких конфликтов.

Доступ к полю двумерной таблицы осуществляется аналогичным образом (скобки здесь являются обязательным элементом синтаксиса):

```
<имя_таблицы>[<индекс>].<имя_поля>
```

Эту запись можно трактовать так:

```
(<имя_таблицы>[<индекс>]).<имя_поля>
```

Например:

```
birthdays[mother].day := 18;
```

Оперирование несколькими полями

Если в тексте программе необходимо несколько раз подряд обращаться к полям одной и той же записи, может оказаться неудобным записывать это обращение полностью по причине громоздкости:

```
today.day:= 17;
today.month:= 11;
today.year:= 2013;
```

Для сокращения таких участков служит оператор `with`, позволяющий обращаться к полям, не указывая каждый раз имя всей записи:

```
with <имя_записи> do
    begin
        <операторы>
        {имена полей здесь используются как <имя_поля>,
```

```
        а не как <имя_записи>.<имя_поля>}
    end;
```

Например:

```
with today do
    begin
        day:= 17;
        month:= 11;
        year:= 2013;
    end;
```

Замечание. Для того чтобы внутри оператора with можно было обратиться не к полю записи, а к глобальной переменной с таким же именем, перед этой переменной нужно указать (через точку) имя программы: <имя_программы>.<имя_переменной>.

Например:

```
with todayday do
    begin
        day:= 17;
        month:= 11;  {поле записи today.month}
        year:= 2013;
        MyProg.month:= 5; {глобальная переменная month}
    end;
```

Вложенные операторы with

Если возникает необходимость расположить один оператор with внутри другого, то любую переменную (если перед ней явно не указано имя записи), находящуюся под внутренним оператором with, компилятор пытается интерпретировать в такой последовательности:

1. если во внутренней записи есть поле с искомым именем, то поиск заканчивается;
2. если во внутренней записи поля с таким именем нет, то поиск производится среди полей внешней записи (если вложенных

операторов with больше, чем два, то поиск ведется последовательно во всех задействованных записях в направлении "изнутри наружу");

3. если среди полей всех вложенных записей нет искомого идентификатора, компилятор считает его глобальной переменной.

Например:

```
type Tdate = record
            day: 1..31;
            month: 1..12;
            year: 1900..2005;
            end;
    Tstudent = record
            name: string[100];
            year: 2008..2030; {год поступления}
            gruppa: string[5];
            birth: Tdate;
            end;
var Stud: Tstudent;

begin
    ...
    with Stud do
    begin
            year := 2013;    {Stud.year}
            with birth do
             begin
                ...
                year:= 2001;     {birth.year}
                gruppa:= '1104'; {Stud.gruppa}
                ...
             end;
            ...
    end;
end;
```

Запись с вариантной частью

Если заранее известно, что в массиве записей (таблице) некоторые поля могут оставаться пустыми (наборы пустых полей могут быть разными для разных записей), то становится возможным сократить неиспользуемый, но резервируемый объем памяти.

Для этих случаев применяется запись с вариативной часть.

В разделе var запись с вариантной частью описывают так:

```
var <имя_записи>: record
                  <поле1>: <тип1>;
                  [<поле2>: <тип2>;]
                  [...]
                  case <поле_переключатель>: <тип> of
                  <варианты1>:       (<поле3>: <тип3>;
                                     <поле4>: <тип4>;
                                     ...);
                  <варианты2>:       (<поле5>: <тип5>;
                                     <поле6>: <тип6>;
                                     ...);
                  [...]
                  end;
```

Невариантная часть записи (до ключевого слова case) подчиняется тем же правилам, что и обычная запись. Вообще говоря, невариантная часть может и вовсе отсутствовать.

Вариантная часть начинается зарезервированным словом case, после которого указывается то поле записи, которое в дальнейшем будет служить переключателем. Как и в случае обычного оператора case, переключатель обязан принадлежать к одному из перечислимых типов данных (лекция 5). Список вариантов может быть константой, диапазоном или объединением нескольких констант или диапазонов. Набор полей, которые должны быть включены в структуру записи, если выполнился соответствующий вариант, заключается в круглые скобки.

Количество байтов, выделяемых компилятором под запись с вариантной частью, определяется самым "длинным" ее вариантом. Более "короткие" наборы полей из других вариантов занимают лишь некоторую часть выделяемой памяти.

Работа с текстовыми файлами

В четвертой лекции был рассмотрен ввод информации с клавиатуры и вывод ее на экран. Однако процесс ввода с консоли весьма трудоемок, а результат вывода на консоль сохраняется только до завершения работы программы. Существует более удобный способ записывать, хранить, пересылать и по необходимости считывать информацию из постоянной памяти компьютера. Для этого применяются файлы.

Под файлом понимается самостоятельная последовательность символов, записанная в постоянную память компьютера.

В английском языке слово " file " имеет вполне понятный смысл: "вереница", что очень хорошо отражает внутреннюю структуру любого файла. Файл это именно вереница символов, причем связанных в определенной последовательности: символы файла сохраняют упорядоченность.

Сохранность файлов состоит в том, что они не зависят от работы какой-либо программы. И даже если выключить компьютер, файлы будут продолжать свое существование на носителе информации, в отличии от данных (значений переменных), хранящихся в оперативной памяти (только на время работы программы).

Файлы могут хранить в себе все, что поддается кодированию:

— исходные тексты программ или входные данные;
— машинные коды выполняемых программ;
— информацию о текущем состоянии какого-либо процесса, или его результат работы;
— различные документы;
— мультимедийную информацию (картинки, звуковые или видео-фрагменты)
— другие виды данных.

Когда нужно использовать файлы

Выбор между консолью и файлами необходимо делать всякий раз, когда разрабатывается очередная программа.

Для ответа на вопрос, вынесенный в заголовок этого пункта, следует рассмотреть следующие обстоятельства:

- Файлы полезны, если объем входных данных превосходит посильный при ручном вводе. (Крайним является случай, когда входные или выходные данные заведомо не могут поместиться в оперативной памяти)
- Файлы нужны, если приходится многократно вводить одну и ту же информацию, с минимальными изменениями или вовсе без изменений (например, при отладке программы).
- Файлы необходимы, если нужно сохранять информацию о результатах работы программы, полученных при вводе различных входных данных (например, при поиске ошибок в программе).

Например, если вашей программе необходимо получить несколько чисел или пару строк длиной десяток символов, то вполне можно задавать такие данные с клавиатуры вручную. Если же необходимо вводить, например, массив чисел размерностью 10x10, то вероятность ошибки при ручном вводе возрастает многократно. Значит, возможность этой ошибки нужно исключить: записать исходные данные в файл, который легко отредактировать в случае необходимости. Кроме того, однажды созданный файл можно использовать многократно (может быть, с незначительными изменениями).

Разновидности файлов

В языке Pascal имеется возможность работы с тремя видами файлов:

- текстовыми;
- типизированными;
- нетипизированными.

Последние два типа объединяются под названием бинарные: информация в них записывается по байтам и потому недоступна для просмотра или редактирования в удобных для человека текстовых редакторах, зато такие файлы более компактны, чем текстовые.

В отличие от бинарных, текстовые файлы возможно создавать, просматривать и редактировать "вручную" – в любом доступном в системе текстовом редакторе. Кроме того, при считывании данных из текстового файла нет необходимости заботиться об их преобразовании, поскольку в языке Pascal имеются средства автоматического перевода содержимого текстовых файлов в нужный тип и формат, и это позволяет сэкономить немало времени и сил при разработке программы.

Описание файлов

В разделе var переменные, используемые для работы с файлами (так называемые файловые переменные), описываются следующим образом:

```
var f1,f2: text;     {текстовые файлы}
    g: file of <тип_элементов>; {типизированные файлы}
    in, out: file;   {нетипизированные файлы}
```

Отметим, что файловая переменная не может быть задана константой.

Текстовые файлы

В этом курсе мы ограничимся рассмотрением лишь текстовых файлов. С этого момента и до конца лекции под словом "файл" мы будем подразумевать "текстовый файл" (если иное специально не оговорено). Однако многие описываемые ниже команды пригодны не только для текстовых, но и для бинарных файлов.

Назначение файла

Процедура `assignfile(f,'<имя_файла>');` служит для установления связи между файловой переменной f и именем того файла, за действия с которым эта переменная будет отвечать.

На разных этапах работы программы одной и той же файловой переменной можно присваивать разные значения. Например, если в начале программы мы напишем

```
assignfile(f, 'input.txt');
```

то переменной f будет соответствовать файл, из которого производится считывание входных данных, вплоть до того момента, когда в программе встретится, скажем, команда

```
assignfile(f, 'output.txt');
```

после которой переменной f будет уже соответствовать тот файл, куда выводятся результаты.

Строка '<имя_файла>' может содержать полный (абсолютный или относительный) путь к файлу. В том случае, если путь не указан, файл считается расположенным в той же директории, что и исполняемый модуль программы. Именно этот вариант обычно считается наиболее удобным.

Открытие файла

В зависимости от того, какие действия ваша программа собирается производить с открываемым файлом, возможно трояков его открытие:

```
reset(f);
```

Открытие файла для считывания из него информации; если такого файла не существует, попытка открыть его вызовет ошибку и аварийную остановку работы программы. Эта же команда служит для возвращения указателя на начало файла;

```
rewrite(f);
```

Открытие файла для записи в него информации; если такого файла не существует, он будет создан; если файл с таким именем уже есть, вся содержавшаяся в нем ранее информация исчезнет;

```
append(f);
```

Открытие файла для записи в него информации (указатель помещается в конец этого файла). Если такого файла не существует, он будет создан; а если файл с таким именем уже есть, вся содержащаяся в нем ранее информация будет сохранена, потому что запись будет производиться в его конец.

Закрытие файла

После того как ваша программа закончит работу с файлом, обязательно следует закрыть его командой:

```
close(f);
```

В противном случае информация, содержащаяся в этом файле, может быть потеряна.

Считывание из файла

Чтение данных из файла, открытого для считывания, производится с помощью команд read() и readln(). В скобках сначала указывается имя файловой переменной, а затем - список ввода. Например:

`read(f,a,b,c);` считать из файла f три переменные a, b и c. После выполнения этой процедуры указатель в файле передвинется за переменную c;

`readln(f,a,b,c);` Считать из файла f три переменные a, b и c, а затем перевести указатель ("курсор") на начало следующей строки; если кроме уже считанных переменных в строке содержалось еще что-то, то этот "хвост" будет проигнорирован.

Если вспомнить то обстоятельство, что в памяти компьютера любой файл записывается линейной последовательностью символов и никакой разбивки на строки там фактически нет, то действия процедуры readln() можно пояснить так: **читать все указанные переменные, а затем игнорировать все символы вплоть до ближайшего символа "конец строки" или "конец файла"**. Указатель при этом перемещается на позицию непосредственно за первым найденным символом **"конец строки"**.

Если же символ конца строки встретится где-нибудь между переменными, указанными в списке ввода, то обе процедуры его просто проигнорируют.

Считывать из текстового файла можно только переменные простых типов: целых, вещественных, символьных, а также строковых. Численные переменные, считываемые из файла, должны разделяться хотя бы одним пробельным символом. Типы вводимых данных и типы тех переменных, куда эти данные считываются, обязаны быть совместимыми. Здесь действуют все те же правила, что и при считывании с клавиатуры.

Считываемые переменные могут иметь различные типы. Например, если в файле f записана строка

```
1104 ssau
```

то командой read(f,a,s); можно прочитать одновременно значения для двух переменных, причем все - разных типов:

```
a: integer;
s: string[4];
```

Из файла невозможно считать переменную составного типа (например, если a - массив, то нельзя написать read(f,a), можно считывать его только поэлементно при помощи цикла), файлового, а также логического (следует использовать числовые значения и преобразовывать их в логические после считывания).

Особенно внимательно нужно считывать строки (string[length] и string): эти переменные "забирают" из файла столько символов, сколько позволяет им длина (либо вплоть до ближайшего конца строки). Если строковая переменная неопределенной длины (тип данных string) является последней в текущей строке файла, то никаких проблем с ее считыванием не возникнет. Но в случае, когда необходимо считывать переменную типа string из начала или из середины строки файла, это придется делать с помощью цикла - посимвольно. Аналогичным образом – посимвольно, от пробела до пробела - считываются из текстового файла слова.

Существует еще один способ: считать из файла всю строку целиком, а затем разобрать ее на части специальной процедурой выделения подстрок, при помощи функции copy(). После чего (если необходимо) применить процедуру превращения символьной записи числа в само число, применяя стандартную процедуру val(). Однако, совсем не очевидно, что длина вводимой строки не будет превышать 256 символов, поэтому такой способ следует использовать с осторожностью.

Запись в файл

Сохранять переменные в файл, открытый для записи командами rewrite(f) или append(f), можно при помощи команд write() и writeln(). Так же как в случае считывания, первой указывается файловая переменная, а за ней - список вывода:

`write(f,a,b,c);` Записать в файл f переменные a, b и c ;
`writeln(f,a,b,c);` записать в файл f переменные a, b и c, а затем записать туда же символ " конец строки ".

Выводить в текстовый файл можно переменные любых базовых типов (вместо значений логического типа выведется их строковый аналог TRUE или FALSE) или строки.

Структурированные типы данных можно записывать только поэлементно (обычно при помощи циклов).

Пробельные символы

К пробельным символам (присутствующим в файле, но невидимым на экране) относятся:

- символ горизонтальной табуляции (#9);
- символ перевода строки (#10) (смещение курсора на следующую строку, в той же позиции);
- символ вертикальной табуляции (#11);
- символ возврата каретки (#13) (смещение курсора на начальную позицию текущей строки);
- символ конца файла (#26);
- символ пробела (#32).

Замечание: Пара символов #13 и #10 является признаком конца строки текстового файла (в кодировках DOS и Windows).

В (текстовом) файле границами чисел служат пробельные символы, и при считывании чисел эти пробельные символы игнорируются, сколько бы их ни было. Таким образом, если ввод многих чисел производится с помощью команды read(), то нет никакой разницы, как именно записаны эти числа: в одну строку, в несколько строк или вовсе в один столбик. В любом случае считывание пройдет корректно и завершится только по достижении конца файла.

Если же считывание тестового файла производится посимвольно, то нужно аккуратно отслеживать пробельные (и особенно концевые) символы.

Поиск специальных пробельных символов (с кодами #10, #13 и #26) можно осуществлять при помощи стандартных функций:

`eof(f)` - возвращает значение TRUE, если уже достигнут конец файла f (указатель находится сразу за последним элементом файла), и FALSE в противном случае;

`seekeof(f)` - возвращает значение TRUE, если "почти" достигнут конец файла f (между указателем и концом файла нет никаких символов, кроме пробельных), и FALSE в противном случае;

`eoln(f)` - возвращает значение TRUE, если достигнут конец строки в файле f (указатель находится сразу за последним элементом строки), и FALSE в противном случае;

`seekeoln(f)` - возвращает значение TRUE, если "почти" достигнут конец строки в файле f (между указателем и концом строки нет никаких символов, кроме пробельных), и FALSE в противном случае.

Очевидно, что в большинстве случаев предпочтительнее использовать функции seekeof(f) и seekeoln(f): они предназначены специально для текстовых файлов, игнорируют концы строк (и вообще все пробельные символы) и потому позволяют автоматически обработать сразу несколько частных случаев.

Например, если по условию задачи файл входных данных может содержать только одну строку, то правильнее всего будет написать программу, обрабатывающую все возможные варианты:

— одна строка, заканчивающаяся символом конца файла;

— одна строка, заканчивающаяся несколькими концевыми пробелами, а затем - символом конца файла;

— одна строка, заканчивающаяся символом конца строки, а затем - символом конца файла (на самом деле получается, что в файле содержится не одна, а две строки, но вторая - пустая);

— одна строка, заканчивающаяся несколькими концевыми пробелами, затем - символом конца строки, а затем - символом конца файла.

Поскольку функции seekeof() и seekeoln() при каждой проверке пытаются проигнорировать все пробельные символы, то и результаты их работы отличаются от результатов работы функций eof() и eoln(). Эти различия нужно учитывать.

Например, для входного файла f, состоящего из двух строк

```
1_2__3_
#13#10
```

содержащего всего 9 символов, причем вторая строка пустая (подчеркивания здесь обозначают пробелы), следующие фрагменты программ будут выдавать такие результаты:

Фрагмент программы	Содержимое результирующего файла g	Длина файла g

`while not eof(f) do`
`begin read(f,c); {c: char}`
`      write(g,c)`
`end;`
	1_2_ _3_#13#10	9 байт

`while not seekeof(f) do`
`begin read(f,c); {c: char}`
`      write(g,c)`
`end;`
	123	3 байта

`while not eof(f) do`
`while not seekeoln(f) do`
`begin read(f,c); {c: char}`
`      write(g,c)`
`end;`
	Зацикливание программы	

`while not seekeof(f) do`
`while not eoln(f) do`
`begin read(f,c); {c: char}`
`      write(g,c)`
`end;`
	1_2_ _3_	7 байт

`while not seekeof(f) do`
`while not  eoln(f) do`
`begin read(f,k); {k: byte}`
`      write(g,k)`
`end;`
	1230	4 байта

Материалы для подготовки к ЛР №1

Разработка алгоритмов для простых задач

Построение алгоритмов подчиняется отдельным законам, использует специальный язык для обозначений основных понятий и терминов, активно используется для описания методов решения задач.

Алгоритм, реализующий решение задачи, можно задать различными способами на основе различных языковых средств: графических, текстовых, табличных.

Графические средства представления алгоритмов имеют ряд преимуществ благодаря визуальности и явному отображению процесса решения задачи. Алгоритмы, представленные на языке графических объектов, получили название визуальные алгоритмы.

Текстовое описание алгоритма является достаточно компактным и может быть реализовано на абстрактном или реальном языках программирования в виде программы для ЭВМ. Таблицы значений представляют алгоритм неявно, как некоторое преобразование конкретных исходных данных в выходные.

Табличный способ описания алгоритмов может быть с успехом применен для проверки правильности функционирования разработанного алгоритма на конкретных тестовых наборах входных данных, которые вместе с результатами выполнения алгоритма фиксируются в«таблицах трассировки».

Основные элементы блок схем (графические алгоритмы)

Наименование	Обозначение	Функция
Блок начало-конец		Начало и конец программы/подпрограммы. Внутри фигуры записывается соответствующее действие.
Блок действия		Выполнение одной или нескольких операций, обработка данных любого вида (изменение значения данных, формы представления, расположения). Внутри фигуры записывают непосредственно сами операции, например, присваивание: x = 10 y+ln(z).
Логический блок (блок условия)		Отображает решение или функцию переключательного типа с одним входом и двумя или более альтернативными выходами, из которых только один может быть выбран после вычисления условий, определенных внутри этого элемента. Вход в элемент обозначается линией, входящей обычно в верхнюю вершину элемента. Если выходов два или три, то обычно каждый выход обозначается линией, выходящей из оставшихся вершин (боковых и нижней). Если выходов больше трех, то их следует показывать одной линией, выходящей из вершины (чаще нижней) элемента, которая затем разветвляется. Соответствующие результаты вычислений могут записываться рядом с линиями, отображающими эти пути. Примеры решения: в общем случае − сравнение (три выхода: >, <, =). Эквивалент оператора if или case
Предопределённый процесс		Символ отображает выполнение процесса, состоящего из одной или нескольких операций, который определен в другом месте программы (в подпрограмме, модуле). Внутри символа записывается название процесса и передаваемые в него данные. Например, в программировании − вызов процедуры или функции.
Ввод-вывод данных		Преобразование данных в форму, пригодную для обработки (ввод) или отображения результатов обработки (вывод)

Граница цикла		Символ состоит из двух частей − соответственно, начало и конец цикла − операции, выполняемые внутри цикла, размещаются между ними. Условия цикла и приращения записываются внутри символа начала или конца цикла − в зависимости от типа организации цикла. Часто для изображения на блок-схеме цикла вместо данного символа используют символ условия, указывая в нём решение, а одну из линий выхода замыкают выше в блок-схеме (перед операциями цикла).
Соединитель		Символ отображает вход в часть схемы и выход из другой части этой схемы. Используется для обрыва линии и продолжения её в другом месте (для предотвращения излишних пересечений или слишком длинных линий, а также, если схема состоит из нескольких страниц). Соответствующие соединительные символы должны иметь одинаковое (при том уникальное) обозначение.
Комментарий		Используется для более подробного описания шага, процесса или группы процессов. Описание помещается со стороны квадратной скобки и охватывается ей по всей высоте. Пунктирная линия идет к описываемому элементу, либо группе элементов (при этом группа выделяется замкнутой пунктирной линией). Также символ комментария следует использовать в тех случаях, когда объём текста, помещаемого внутри некоего символа (например, символ процесса, символ данных и др.), превышает размер самого этого символа, т.е. в роли выносного элемента

Таким образом, все три способа представления алгоритмов можно считать взаимодополняющими друг друга. На этапе проектирования алгоритмов наилучшим способом является графическое представление, на

этапе проверки алгоритма – табличное описание, на этапе применения– текстовая запись в виде программы.

Визуальные алгоритмы

При проектировании визуальных алгоритмов используют графические блоки, представленные в таблице.

Рассмотрим общие правила при проектировании визуальных алгоритмов:

— в начале алгоритма должны быть блоки ввода значений входных данных;

— после ввода значений входных данных могут следовать блоки обработки и блоки условия;

— в конце алгоритма должны располагаться блоки вывода значений выходных данных;

— в алгоритме должен быть только один блок начала и один блок окончания;

— связи между блоками указываются направленными или ненаправленными линиями.

Этап проектирования алгоритма следует за этапом выбора формальной модели и методов решения задачи, на котором определены входные и выходные данные, а также зависимости между ними.

При построении алгоритмов для сложной задачи используют системный подход с использованием последовательной декомпозиции решаемой задачи на ряд подзадач до тех пор, пока не будут определены задачи, для которых алгоритмическое решение известно или оно может быть оперативно найдено.

Одним из системных методов разработки алгоритмов является метод структурной алгоритмизации. Этот метод основан на визуальном представлении алгоритма в виде последовательности управляющих

структурных фрагментов. Выделяют три базовые управляющие процессом обработки информации структуры: композицию, альтернативу и итерацию.

С помощью этих структур можно описать решение большого класса задач обработки информации.

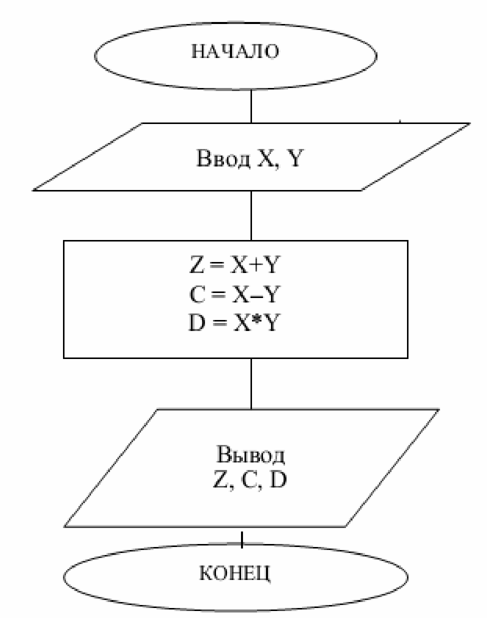
Композиция (следование)– это линейная управляющая конструкция, не содержащая альтернативу и итерацию. Она предназначена для описания единственного процесса обработки информации.

Альтернатива– это нелинейная управляющая конструкция, не содержащая итерацию. Она предназначена для описания процессов решения различных задач обработки информации, выбор которых зависит от значений входных данных.

Итерация – это циклическая управляющая конструкция, которая содержит композицию и ветвление. Она предназначена для организации повторяющихся процессов обработки последовательности значений данных.

Линейные алгоритмы не содержат блока условия. Они предназначены для представления линейных процессов. Такие алгоритмы применяют для описания обобщенного решения задачи в виде последовательности модулей.

Пример линейного алгоритма приведен на рисунке.

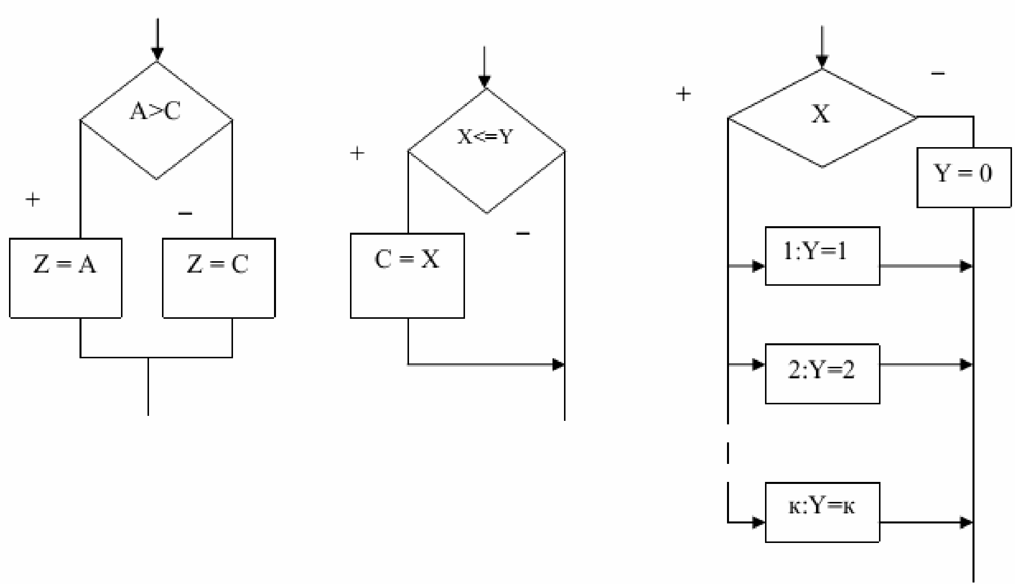


В соответствии с наличием в алгоритмах управляющих структур композиции, альтернативы и итерации алгоритмы классифицируют на линейные, разветвленные и циклические алгоритмы.

Разветвленные алгоритмы

Разветвленные алгоритмы в своем составе содержат блок условия и различные конструкции ветвления. Ветвление – это структура, обеспечивающая выбор между альтернативами.

Каждая управляющая структура ветвления имеет один вход и один выход. Ветвления содержат блок условия, в котором записывают логические условия, например такие как A > C, X <= Y. В зависимости от значений переменных A, C в управляющей структуре ветвления на рисунке, условие A> C принимает значение «истина» или «ложь» и процесс вычислений включает блок действия Z = A или Z = C. Аналогично происходит и в управляющей структуре неполного ветвления. Только в этом случае, если условие X <= Y истинно, то выполняется действие C= X, в противном случае никаких действий не выполняется.

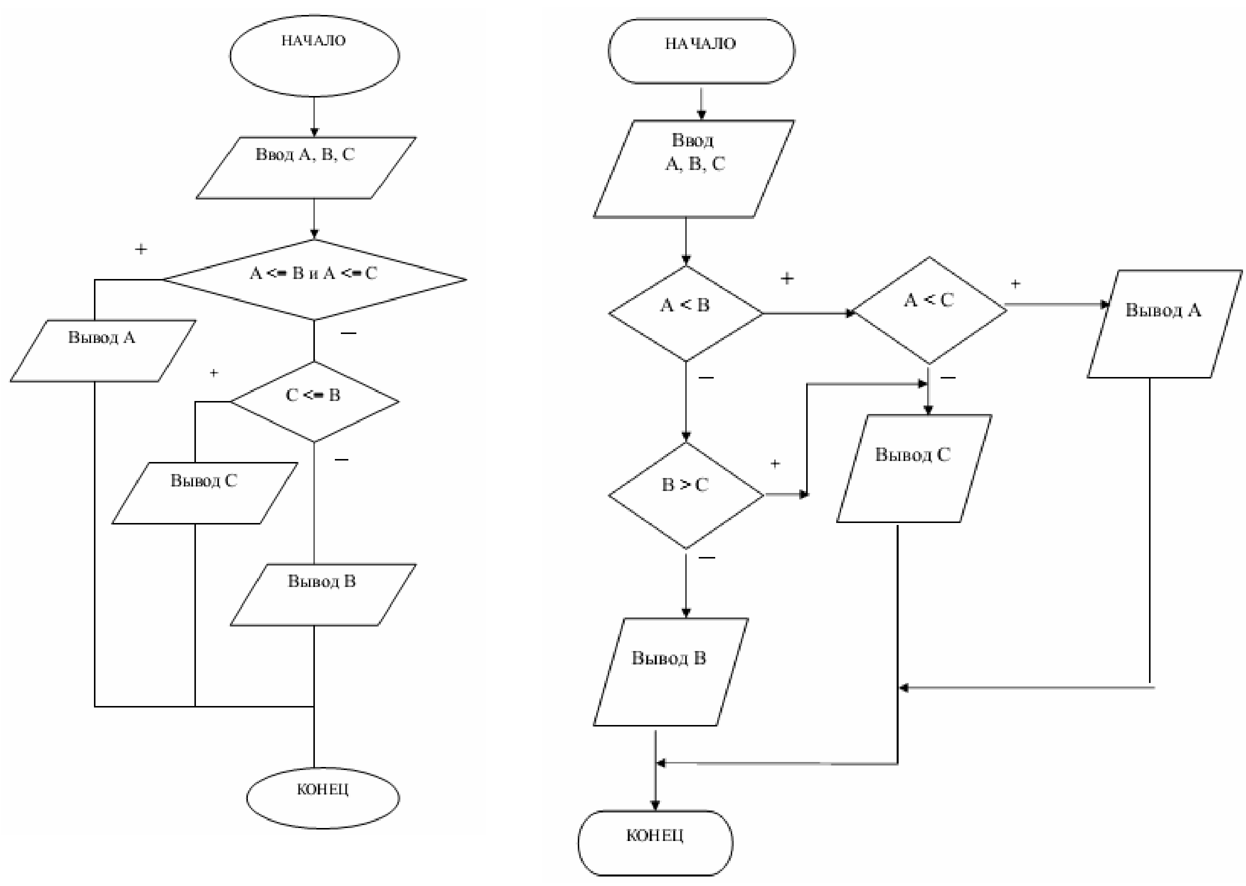


В управляющей структуре многоальтернативный выбор в блоке условия записывается переменная, в данном случае X, которая может принимать различные значения. Если значение переменной X совпадет с

одним из значений в блоке действия, то выполняются действия, записанные в этом блоке. Например, если X= 1, то выполнится действие Y = 1. Если значение X не совпало ни с одним из значений, указанных в блоках слева, то выполняется действие в блоке справа, которого также, как и в неполном ветвлении, может и не быть.

Пример. Составить алгоритм нахождения минимального значения из3 чисел.

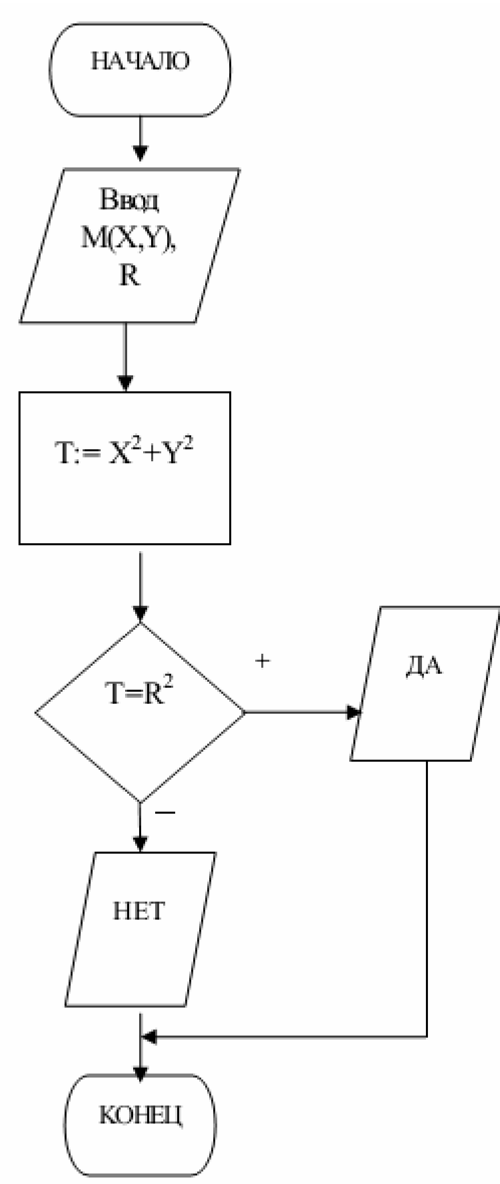
Для определения минимального значения будем использовать проверку пары значений. Визуальные разветвленные алгоритмы приведены на рисунках ниже. Эти алгоритмы используют для обозначения чисел переменные A, B, C и различные подходы для анализа исходных данных.



Пример. Составить алгоритм определения: находится ли точка M с координатами X, Y на окружности радиуса R.

Визуальный алгоритм приведен на рисунке. Для решения в нем используется математическая модель в виде формулы окружности

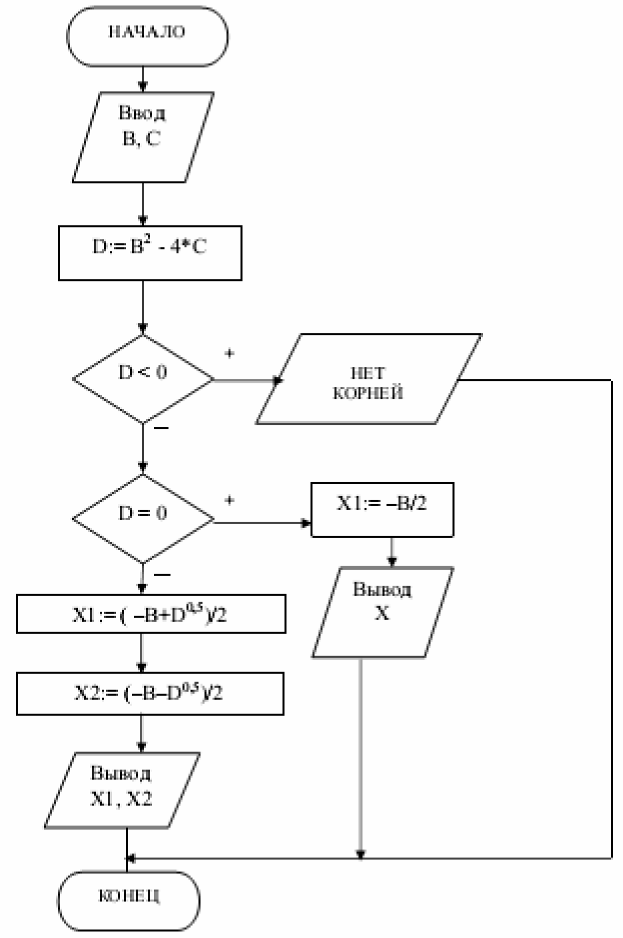
$$R^2 = X^2 + Y^2.$$



Пример. Составить алгоритм определения корней уравнения $x^2 + Bx + C = 0$

При составлении этого алгоритма надо рассмотреть случаи, когда уравнение не имеет корней и когда имеется только один корень. Обозначим корни уравнения через переменные X1, X2. Пусть D – промежуточная переменная для вычисления дискриминанта.

Алгоритм вычисления корней уравнения заданного вида приведен на рисунке.



Задания для самоподготовки:

1. Составить алгоритм проверки является ли точка с координатами X, Y – точкой пересечения диагоналей квадрата со стороной a, построенного в первом квадранте так, что левый нижний угол совпадает с началом координат.

2. Составить алгоритм поездки из дома в университет разными видами транспорта в зависимости от времени начала занятий и статистической информации о пробках на дорогах.

3. Для трех длин отрезков A, B и C проверить, возможно ли построить из этих отрезков треугольник. Если возможно, проверить является ли он прямоугольным, равнобедренным или равносторонним.

Материалы для подготовки к ЛР №2

Среда разработки Lazarus

Существует большое количество компиляторов для языка Pascal. В настоящем курсе будем использовать компилятор Free Pascal Compiler.

Free Pascal Compiler (часто применяется сокращение FPC) это свободно распространяемый компилятор языка Паскаль с открытыми исходными кодами, распространяется на условиях GNU General Public License (GNU GPL). Он совместим с Borland Pascal 7 и Object Pascal – Delphi, но при этом обладает рядом дополнительных возможностей, например, поддерживает перегрузку операторов. FPC — кроссплатформенный инструмент, поддерживающий большое количество платформ. Среди них — DOS, Linux, *BSD, OS/2, MacOS(X) и Windows.

Free Pascal поддерживает компиляцию в нескольких режимах, обеспечивающих совместимость с различными диалектами и реализациями языка:

- TP – режим совместимости с Turbo Pascal: совместимость практически полная, за исключением нескольких моментов, связанных с тем, что FPC компилирует программы для защищѐнного режима процессора, где невозможно прямое обращение к памяти, портам и т. д.
- FPC – собственный диалект: соответствует предыдущему, расширенному дополнительными возможностями, такими как, например, перегрузка операторов.
- DELPHI – режим совместимости с Delphi: включает поддержку классов и интерфейсов.
- OBJFPC – совмещает объектно-ориентированные возможности Delphi и собственные расширения языка.
- MACPAS – режим совместимости с Mac Pascal.
- GNU – режим частичной совместимости с GNU Pascal.

Free Pascal Compiler имеет свою собственную интегрированную среду разработки. Применяется также аббревиатура IDE (Integrated Development Environment). Среда имеет текстовый интерфейс очень похожий на интерфейс Turbo Pascal 7.0.

Однако, в 1999 г. Cliff Baeseman, Shane Miller и Michael A. Hess. предприняли попытку написать бесплатную графическую среду для бесплатного компилятора FPC. Проект получает название Lazarus. На сегодняшний день следует признать, что идея оказалась весьма плодотворной потому, что среда пользуется популярностью и активно развивается.

Lazarus это бесплатный инструмент разработки с открытым кодом. IDE Lazarus представляет собой среду с графическим интерфейсом для быстрой разработки программ, аналогичную Delphi, и базируется на оригинальной кроссплатформенной библиотеке визуальных компонентов LCL (Lazarus Component Library), совместимых с VCL Delphi. В состав IDE входят и не визуальные компоненты. В принципе такого набора достаточно для создания программ с графическим интерфейсом и приложений, работающих с базами данных.

В среде Lazarus используются собственный формат управления пакетами и свои файлы проектов.

Lazarus это стабильная богатая возможностями среда разработки для создания самостоятельных графических и консольных приложений. В настоящее время она работает на Linux, FreeBSD и Windows и предоставляет настраиваемый редактор кода и визуальную среду создания форм вместе с менеджером пакетов, отладчиком и графическим интерфейсом, полностью интегрированным с компилятором FreePascal.

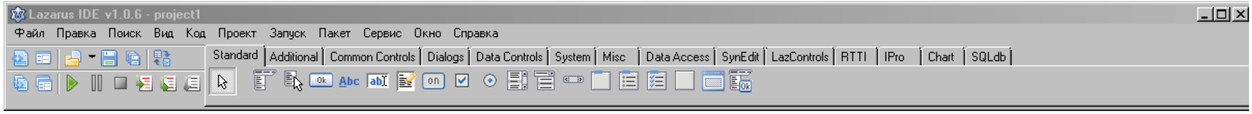
Рассмотрим основные элементы среды разработки Lazarus.

Дистрибутив доступен по адресу адресу http://sourceforge.net/projects/lazarus/ либо http://www.lazarus.freepascal.org/

IDE Lazarus имеет вид, показанный на рисунке.

Среда Lazarus состоит из нескольких, вообще говоря, не связанных окон.

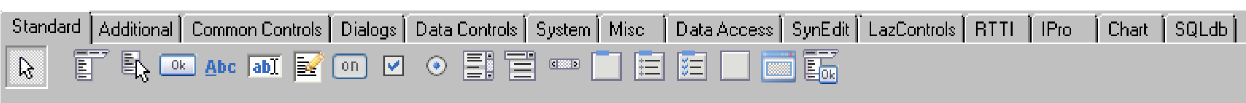
Главное окно IDE Lazarus, состоит из главного меню, панели инструментов и палитры компонентов (см. рисунки)



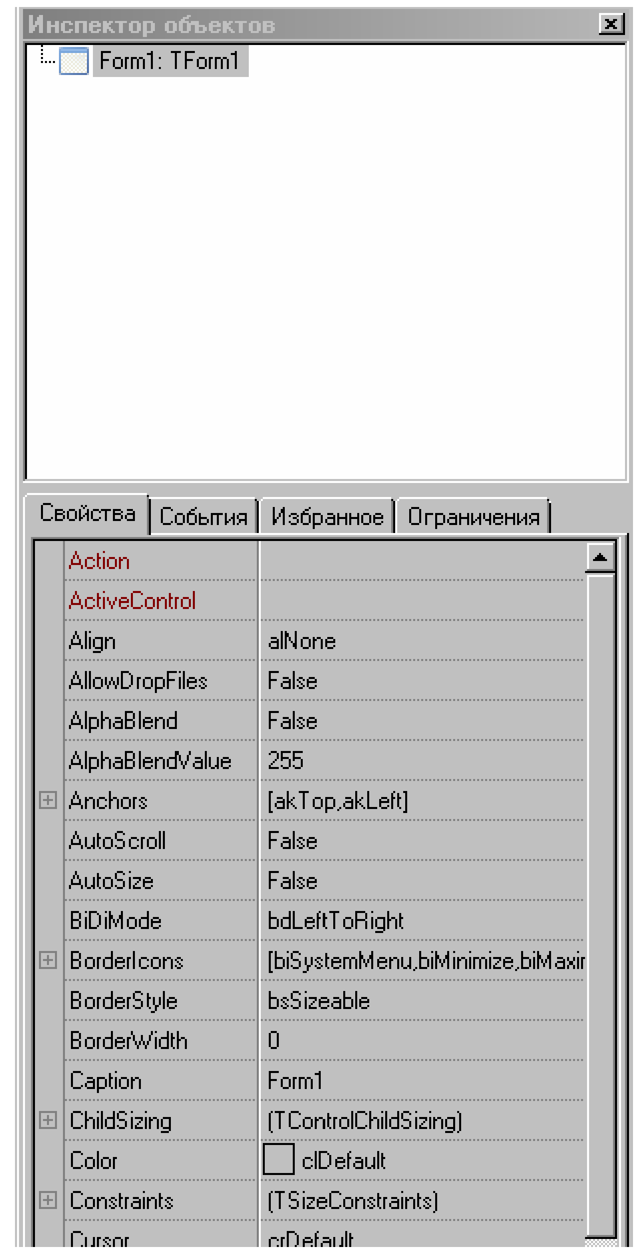
Главное меню:

Панель инструментов:

Палитра компонентов:



Второе окно представляет инспектор объектов:



В верхней части окна показывается иерархия объектов, а снизу, расположены три вкладки: "Свойства", "События", "Избранное". Назначение инспектора объекта – это просмотр всех свойств и методов объектов. На вкладке "Свойства" перечисляются все свойства выбранного объекта. На вкладке "События" перечисляются все события для объекта. На вкладке "Избранное" избранные свойства и методы.

Центральное окно представляет редактор исходного кода:

Редактор исходного кода

*Unit1

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs;

type
  TForm1 = class(TForm)
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.lfm}

end.
```

Именно в этом окне мы будем набирать тексты своих программ. Многие функции и возможности этого редактора совпадают с возможностями обычных текстовых редакторов. Текст в редакторе можно выделять, копировать, вырезать, вставлять. Кроме того, в редакторе можно осуществлять поиск заданного фрагмента текста, выполнять вставку и замену. Но, конечно, редактор исходных текстов Lazarus обладает еще рядом дополнительных возможностей для комфортной работы применительно к разработке программ.

Основное преимущество редактора заключается в том, что он обладает возможностями подсветки синтаксиса, причѐм не только Pascal, но и других языков, а также рядом других удобств. В частности, выделенный фрагмент текста можно сдвигать вправо или влево на количество позиций, что очень удобно для форматирования с целью структурирования кода. Выделенный фрагмент можно закомментировать или раскомментировать, перевести в верхний или нижний регистр и т.д.

Все возможные операции в редакторе собраны в меню Правка и Поиск главного меню Lazarus.

Информационное окно сообщений:

В этом окне выводятся сообщения компилятора, компоновщика и отладчика, помогающие найти ошибки в написанной программе.

Материалы для подготовки к ЛР №3

Вычисление кусочно-заданной функции. Табулирование функции. Массивы

Операторы ветвления

Операторы ветвления предназначены для реализации выбора действия в зависимости от выполнения некоторого условия (или равенства величины некоторому значению).

В Pascal существует два оператора, реализующих логику разветвления: оператор IF-THEN-ELSE и оператор CASE.

Условный оператор IF-THEN-ELSE записывается в форме

```
if <условие>
  then <оператор_если_истина>
  [else <оператор_если_ложь>];
```

причем в зависимости от истинности выражения <условие> выбирается то или иное действие – в случае истинности выполняется оператор после ключевого слова `then`, в противном случае – оператор после слова `else`.

Существует сокращенная форма условного оператора, когда ветвь `else` отсутствует, в этом случае при ложном значении выражения <условие> управление переходит к следующему оператору программы.

Множественный выбор вариантов позволяет организовать оператор CASE. Он имеет следующую форму:

```
case <переключатель> of
          <список_констант> : <один_оператор>;
          [<список_констант> : <один_оператор>;]
          [<список_констант> : <один_оператор>;]
          [else <один_оператор>;]
end;
```

В этом случае, в зависимости от значения переменной <переключатель> происходит выполнение соответствующего действия. Если

же значение переменной <переключатель> не совпадает ни с одной из констант, выполняется оператор ветки else (при ее наличии), или же управление переходит к следующему оператору программы.

В качестве переключателя может выступать любая переменная порядкового типа. В качестве констант могут выступать как одиночные значения, так и целые интервалы, причем тип констант должен совпадать с типом переменной переключателя.

Рассмотрим применение условных операторов на примере вычисления кусочно-заданной функции.

Пример. Требуется составить алгоритм вычисления значения функции

$$y = \begin{cases} \sin(x), & \text{при } x \le 0 \\ \sqrt{x}, & \text{при } x > 0 \end{cases},$$

для произвольно введенного значения x.

Эта задача предполагает выбор одного из двух вариантов вычисления значения функции в зависимости от того, является ли введенное значение x положительной величиной или нет.

Реализация вычисления такой функции на языке Pascal имеет вид:

```
program Func;
var x, y: real;
begin
    Readln(x);
    if x > 0     then y := sqr(x)
            else y := sin(x);
    writeln('F(', x:5:2, ')=', y:5:3);
    Readln;
end.
```

Пример. Для введенного номера месяца (целое число в диапазоне 1..12) вывести к какому кварталу года этот месяц относится.

Решение этой задачи возможно путем использования нескольких операторов IF-THEN, но оно будет выглядеть громоздко. Гораздо удобнее воспользоваться оператором CASE.

Текст программы в этом случае будет выглядеть так:

```
Program CheckKvartal;
Var Month: Byte;
begin
    Writeln('Введите номер месяца [1..12]');
    Readln(Month);
    case Month of
        1, 2, 3 : writeln ('Первый квартал');
        4, 5, 6 : writeln ('Второй квартал');
        7, 8, 9 : writeln ('Третий квартал');
        10, 11, 12 : writeln ('Четвёртый квартал');
    else writeln('Неверный номер месяца!');
    end;
    Readln;
end.
```

Логика работы программы в пояснениях не нуждается.

Циклические алгоритмы

Циклические алгоритмы являются наиболее распространенным видом алгоритмов, в них предусматривается повторное выполнение определенного набора действий в зависимости от некоторого условия, называемого условием окончания повторений. Такое повторное выполнение называют циклом. Повторяющиеся действия в цикле называются «телом цикла».

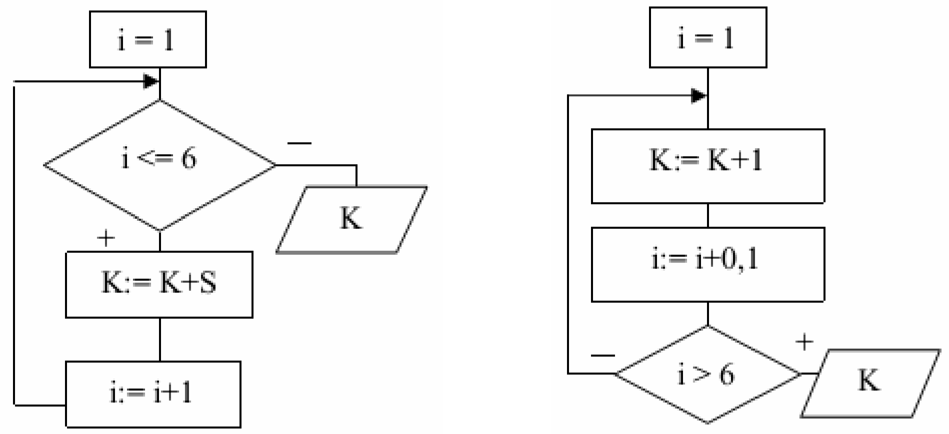
Существуют два основных вида циклических алгоритмов: циклические алгоритмы с предусловием, циклические алгоритмы с постусловием. Они отличаются друг от друга местоположением условия выхода их цикла.

Цикл с предусловием (цикл while) начинается с проверки условия выхода из цикла. Это логическое выражение, например I <= 6 (эквивалентно математическому выражению $I \leq 6$). Пока оно истинно, то выполняются действия цикла, которые должны повторяться. При этом в теле цикла

переменные (хотя бы одна из них), составляющие условие должны изменяться, в противном случае цикл будет иметь бесконечный характер.

Если при изменении переменной I логическое выражение I <= 6 станет ложным, то есть I станет больше 6, то цикл с предусловием прекратит свои действия.

Цикл с постусловием (цикл repeat-until) функционирует иначе. Сначала выполняются один раз те действия, которые подлежат повторению, затем проверяется логическое выражение, определяющее условие выхода из цикла, например, I > 6. Цикл повторяет действия, указанные в теле цикла, до тех пор, пока условие выхода не станет истинным, в противном случае происходит повторение действий, указанных в цикле. Разновидности циклов с условием приведены на рисунке.



В записи на языке Pascal эти блоки можно записать так:

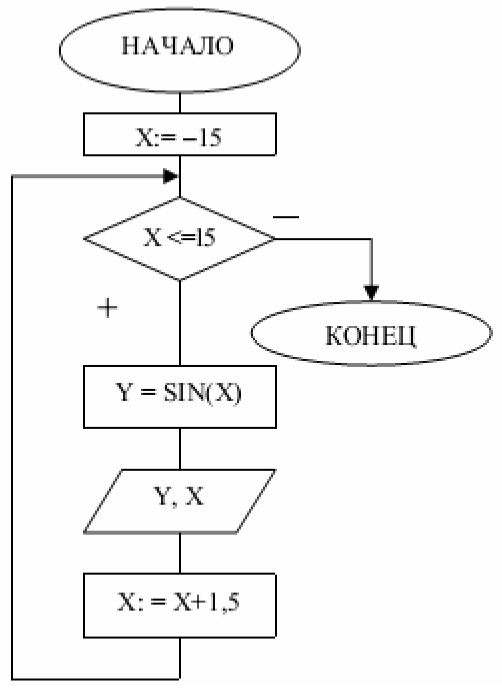
```
i := 1;
while i <= 6 do
begin
  k := k + s;
i := i + 1;
end;
writeln(k);
```

```
i := 1;
repeat
k:= k + 1;
i := i + 0.1;
until i >6;
writeln(k);
```

Пример. Требуется составить алгоритм получения на отрезке [-15,15] последовательности значений функции Y=SIN(X) в виде таблицы значений(X,Y) при изменении аргумента X по формуле $X_k = X_{k-1} + h$, где h = 1,5.

Такие задачи носят название табулирования функций. Из условия задачи определяем, что начальное значение отрезка табулирования X = –15, конечное значение X = 15. Процесс получения множества пар (X, Y) является итерационным, значит разрабатываемый алгоритм будет циклическим. Условие выхода из цикла X> 15.

На рисунке представлен циклический алгоритм с предусловием вычисления табличного значения функции Y = SIN(X) на отрезке -15 < X < 15 при изменении X на каждом шаге итерации на величину 1,5. Результатом выполнения алгоритма является циклический вывод множеств пар (X,Y).



На языке Pascal программа, соответствующая приведенной блок-схеме будет выглядеть так:

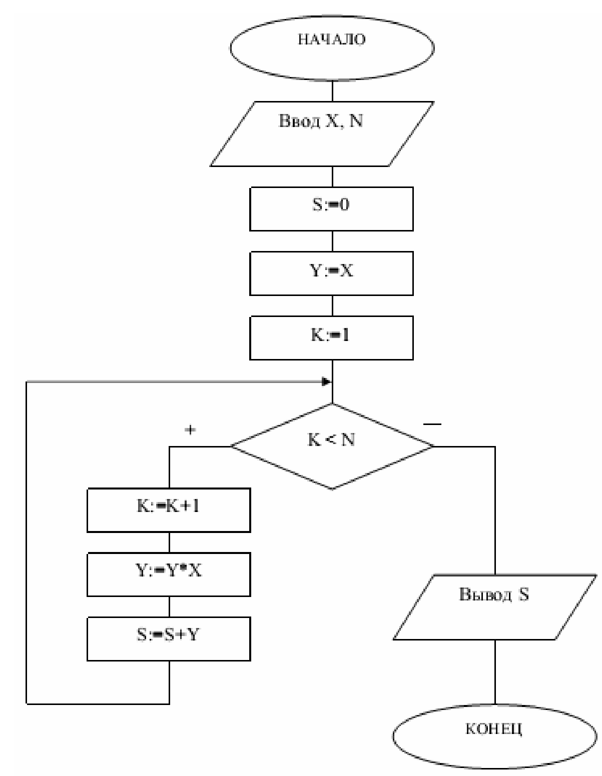
```
program tabulate;
var x, y, h: real;
begin
    x:=-15;
    h:=1.5;
```

```
    while x<= 15 do
    begin
        y := sin(x);
        writeln(x:3:1,y:7:4);
        x:=x+h;
    end;
    readln;
end.
```

Пример. Требуется составить алгоритм вычисления суммы ряда $S = x + x^2 + x^3 + ... + x^N$ (здесь N – целое положительное число).

Исходные данные для алгоритма – это переменные x и N. На каждом шаге будем вычислять Y (очередной член суммы) и прибавлять его к предыдущему значению суммы S. Для этого используем рекуррентную формулу вычисления степени $Y = Y \cdot x$, тогда сумма ряда на каждом шаге итерации будет вычисляться по формуле S=S+Y. Количество итераций K изменяется от 1 до N и равно количеству членов ряда. Начальное значение суммы ряда S устанавливается равным 0.

На рисунке представлен циклический алгоритм с предусловием для вычисления заданной суммы ряда.



Соответствующая программа на языке Pascal запишется так.

```
program Summa;
var X, Y, S: real;
    N, K: word;
begin
    readln(X, N);
    S := 0;
    Y := x;
    K := 1;
    while K < N do
    begin
        K := K + 1;
        Y := Y * X;
        S := S + Y;
    end;
    Writeln(S):
    readln;
end.
```

Работа с массивами

Под массивом в программирование понимается структурированный тип данных, предназначенный для хранения совокупности однотипных элементов.

Для задания переменной – массива необходимо использовать следующий формат описания

```
var   array[<тип_индексов>] of <тип_компонент>;
```

Обращение к элементам массива производится путем указания имени переменной и индекса элемента в квадратных скобках.

```
<имя_массива>[<индекс_компоненты>]
<имя_массива>[<индекс>,...,<индекс>]
```

Пример ввода массива из 5 элементов и вывода его на экран в обратном порядке с использованием циклов:

```
program Massiv1;
Var x: array [1..5] of integer;
    i: byte;
begin
    Writeln('Введите массив из 5 элементов');
    for i := 1 to 5 do
    begin
        Write(' X[ ', i, ']=');
        ReadLn(x[i]);
    end;
    Writeln('------------------');
    for i := 5 downto 1 do
        Writeln('X[',   i,   ']=',   x[i]);
    readln;
end.
```

Пример заполнения массива случайными целыми числами и вывода его на экран.

```
program Massiv2;
Var x: array [1..10] of integer;
    i: byte;
begin
    Randomize;
    for i := 1 to 10 do
        x[i] := Random(99);
    for i := 1 to 10 do
        Writeln('X[',   i,   ']=',   x[i]);
    readln;
end.
```

Задания для самоподготовки:

1. Составить алгоритм вычисления функции, заданной выражением

$$y(x)=\begin{cases} 2\left(x+\dfrac{\pi}{2}\right), & \text{при } x<\dfrac{\pi}{2} \\ \cos(x), & \text{при} |x|\le\dfrac{\pi}{2} \\ \left(x-\dfrac{\pi}{2}\right)^2, & \text{при } x>\dfrac{\pi}{2} \end{cases}$$

2. Выполнить табулирование функции из задания 1 в диапазоне (-3,14; 3,14) с шагом 0,1. Значения вывести на печать с 3 цифрами после запятой.

3. Сформировать массив из 10 случайных чисел в диапазоне от 0 до 99. Вывести на печать те элементы массива, значение которых превышает 50.

Материалы для подготовки к ЛР №4

Вложенные циклы. Сортировка массивов

Вложенные циклы

Существует возможность организовать цикл внутри тела другого цикла. Такой цикл будет называться вложенным циклом. Вложенный цикл по отношению к циклу, в тело которого он вложен, будет именоваться внутренним циклом, и наоборот цикл, в теле которого существует вложенный цикл, будет именоваться внешним по отношению к вложенному. Внутри вложенного цикла в свою очередь может быть вложен еще один цикл, образуя следующий уровень вложенности и так далее. Количество уровней вложенности, как правило, не ограничивается.

Полное число исполнений тела внутреннего цикла не превышает произведения числа итераций внутреннего и всех внешних циклов. Например, взяв три цикла, вложенных друг в друга, каждый по 10 итераций, получим 10 исполнений тела для внешнего цикла, 100 для цикла второго уровня и 1000 итераций в самом внутреннем цикле.

Подобная организация циклов бывает полезна в операциях перебора всех возможных вариантов,

Правила организации внешнего и внутреннего циклов такие же, как и для простого цикла любого вида. Однако, при программировании вложенных циклов необходимо соблюдать следующее дополнительное условие: все операторы внутреннего цикла должны полностью располагаться в теле внешнего цикла.

Вторым условием, накладываемым на вложенные циклы, является обязательное использование различных переменных для внутренних и внешних циклов. Однако это ограничение не запрещает устанавливать для внутренних циклов переменные пределы, зависящие от значения переменной

внешнего цикла. Следует помнить, что изменять значение параметра внешнего цикла в цикле внутреннем нельзя.

Рассмотрим пример простой задачи, решение которой предполагает использование вложенных циклов – задачи вывода на экран таблицы умножения. С использованием цикла for вариант решения данной задачи может быть следующим:

```
program TablUmn;
var
     I, J : byte;
begin
     for I:=1 to 9 do {Внешний цикл}
          for J:=1 to 9 do {Внутренний цикл}
               {Тело внутреннего цикла}
               Writeln (I, ' * ', J, ' = ', I*J);
     readln;
end.
```

Вывод этой программы получается в одну длинную колонку, что не слишком удобно. Модифицируем программу для получения табличной структуры.

```
program TablUmnMod;
var
     I, J : byte;
begin
     write(' ');
     for I:=1 to 9 do {Внешний цикл}
          write(i:4)
     writeln;
     for I:=1 to 9 do {Внешний цикл}
     begin
          write(i:1);
          for J:=1 to 9 do {Внутренний цикл}
               {Тело внутреннего цикла}
               Write((I*J):4);
          writeln;
     end;
     readln;
end.
```

Сортировка массивов

Под сортировкой массива подразумевается процесс перестановки элементов с целью упорядочивания их в соответствии с каким-либо критерием. Например, если имеется массив x целых чисел, то после сортировки по возрастанию должно выполняться условие: $x_1 \le x_2 \le ... \le x_n$, где n – верхняя граница индекса массива.

Существует много методов сортировки массивов. Рассмотрим некоторые из них:

– метод простого выбора;
– метод прямого обмена (пузырьковая сортировка).

Генерацию массивов будем производить случайным образом, заполнять целыми числами. Размер массива для определенности положим равным 10 элементам.

Метод простого выбора

Алгоритм сортировки массива по возрастанию методом простого выбора может быть представлен так:

1. Просматривая массив от первого элемента, найти минимальный и поместить на место первого элемента, а первый на место минимального.
2. Просматривая массив от второго элемента, найти минимальный и поместить его на место второго элемента, а второй на место минимального.
3. И так далее до предпоследнего элемента.

Ниже представлена программа сортировки массива целых чисел по возрастанию на языке Pascal.

```
program ProstSort;
const    n = 10;
var x: array [1..n] of integer;
    i, j, k: integer;
```

```
begin
{ заполнение массива случайными числами }
    Randomize;
    for j := 1 to n do
        x[j] := Random(100);
{ вывод исходного массива на экран }
    for j := 1 to n do
        Write(x[j]:3);
    Writeln;
{ сортировка массива методом простого выбора }
    for i := 1 to n-1 do
    begin
        k := i;
        for j := i + 1 to n do
            if x[j] < x[k] then k := j;
        c := x[i];
        x[i] := x[k];
        x[k] := c;
    end;
{ вывод упорядоченного массива на экран }
    for j := 1 to n do
        Write(x[j]:3);
    readln;
end.
```

Метод прямого обмена

В основе алгоритма лежит обмен соседних элементов массива. Каждый элемент массива, начиная с первого сравнивается со следующим и если он больше следующего, то элементы меняются местами. Таким образом, элементы с наименьшим значением продвигаются к началу массива (всплывают), а элементы с большим значением – к концу массива (тонут), поэтому метод получил распространенное название – пузырьковая сортировка. Этот процесс повторяется на единицу меньшее число раз чем элементов в массиве.

Программа сортировки массива целых чисел по возрастанию на языке Pascal методом прямого обмена представлена ниже.

```
program SortChange;
const    n = 10;
var x: array [1..n] of integer;
    i, j, tmp: integer;
begin
{ заполнение массива случайными числами }
    Randomize;
    for j := 1 to n do
        x[j] := Random(100);
{ вывод исходного массива на экран }
    for j := 1 to n do
        Write(x[j]:3);
    Writeln;
{ сортировка массива методом прямого обмена }
    for i:=n-1 downto 1 do
        for j:=1 to i do
            if x[j]>x[j+1] then
                begin
                    tmp:= M[j];
                    M[j]:= M[j+1];
                    M[j+1]:= tmp;
                end;
{ вывод упорядоченного массива на экран }
    for j := 1 to n do
        Write(x[j]:3);
    readln;
end.
```

Задания для самоподготовки:

1. Составить алгоритм подсчета суммы элементов для каждого столбца массива размерностью $n \times n$ (принять $n = 8$). Заполнение массива производить случайными целыми положительными числами не превышающими 50.

2. Для двумерного массива размерностью $m \times n$ (принять $m = 4$, $n = 5$) разработать алгоритм вычисления произведения тех элементов, сумма индексов которых – четное число. Заполнение массива производить случайным образом, целыми положительными числами 0...30.

3. Модифицировать приведенные алгоритмы для сортировки массивов по убыванию элементов.

Материалы для подготовки к ЛР №5

Работа с текстовыми строками

Подсчет количества букв

Дана символьная строка. Подсчитать в ней количество букв r, k, t.

```
program bukvy;
var s: string;
    r,k,t,i:integer;
begin
    write('введите строку');
    readln(s);
    a:=0; b:=0; c:=0;
    for i:=1 to length(s) do
    begin
        if s[i]='r' then r:=r+1;
        if s[i]='k' then k:=k+1;
        if s[i]='t' then t:=t+1;
    end;
    writeln('количество букв r=',r);
    writeln('количество букв k=',k);
    writeln('количество букв t=',t);
    readln;
end.
```

Подсчет количества цифр в строке

```
Program CifrCount;
var s : string;
    i, nd : byte;
begin
    write('введите произвольную строку');
    readln(s);
    nd:=0;
    for i:=1 to length(s) do
        if s[i] in ['0'..'9'] then inc(nd);
    writeln(nd);
    readln;
end.
```

Определить количество пробелов в строке

```
program SpaceCount;
var s:string;
    k, i:integer;
begin
    Write('введите строку ');
    Readln(s);
    i:=1;
    repeat
        if s[i]=' ' then k:=k+1;
        i:=i+1
    until i=length(s);
    writeln('количество пробелов=',k);
    readln;
end.
```

Определить наличие слово длиной 3 буквы

```
program Slovo3;
var s, slovo:string;
    i,k:integer;
    p:boolean;
begin
    write('введите строку:');
    readln(s);
    i:=1;
    p:=true;
    slovo:='';
    k:=0;
    while s[i]<>'.' do
    begin
        if s[i]<>' ' then
        begin
                slovo:=slovo+s[i];
                k:=k+1
        end
        else
        begin
                if k=3 then
                begin
                    writeln(slovo);
                    p:=false
                end;
                slovo:='';
```

```
            k:=0;
        end;
        i:=i+1
    end;
    if k=3 then
    begin
        writeln(slovo);
        p:=false;
    end;
    if p then
        writeln('в строке нет слова из трех букв');
    readln;
end.
```

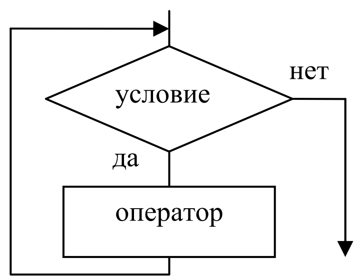
Слова начинающиеся и заканчивающиеся на одну букву

```
program SlovoSym;
var sl,s:string;
    i:integer;
begin
    write('введите строку:');
    readln(s);
    sl:='';
    i:=1;
    s:=s+'.';
    repeat
        if (s[i]=' ') or (s[i]=',') or (s[i]=';') or
            (s[i]='.') or (s[i]='!') or (s[i]='?')
        then begin
            if sl[1]=sl[length(sl)] then writeln(sl);
            sl:='';
        end
        else sl:=sl+s[i];
        i:=i+1
    until i>length(s);
readln;
end.
```

Тесты для самоконтроля

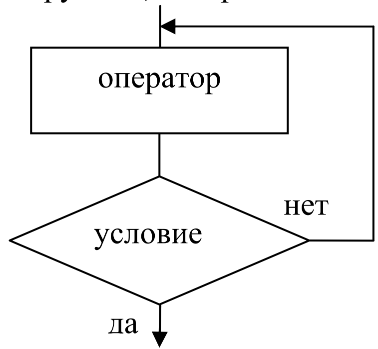
Алгоритмы

1 Укажите вид циклической конструкции, изображенной на рисунке



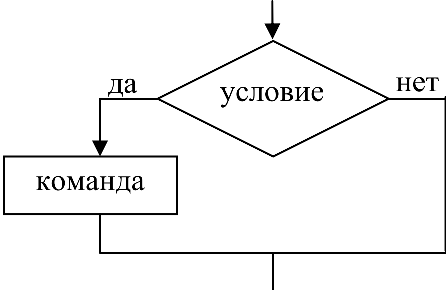
+ цикл с предусловием
- цикл с постусловием
- цикл с параметром

2 Укажите вид циклической конструкции, изображенной на рисунке



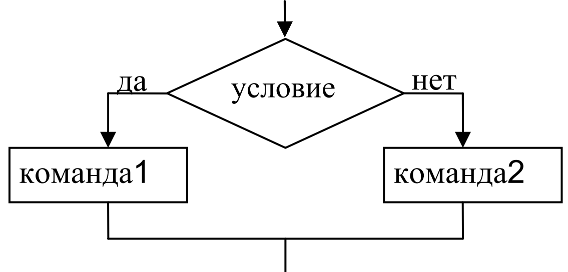
+ цикл с постусловием
- цикл с предусловием
- цикл с параметром

3. Какая алгоритмическая конструкция изображена на рисунке



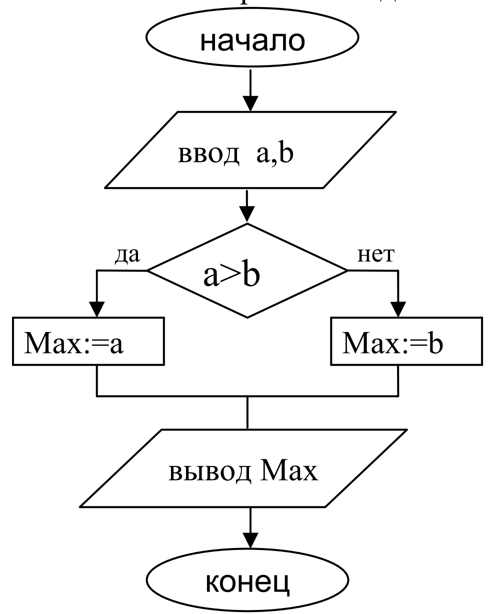
+ неполное ветвление
- полное ветвление
- цикл с предусловием
- цикл с параметром

4. Какая алгоритмическая конструкция изображена на рисунке



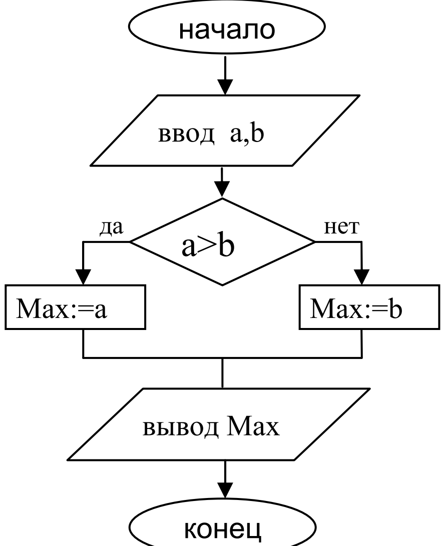
- неполное ветвление
+ полное ветвление
- цикл с предусловием
- цикл с параметром

5. При каких значениях a и b выполнение алгоритма пойдет по левой ветви



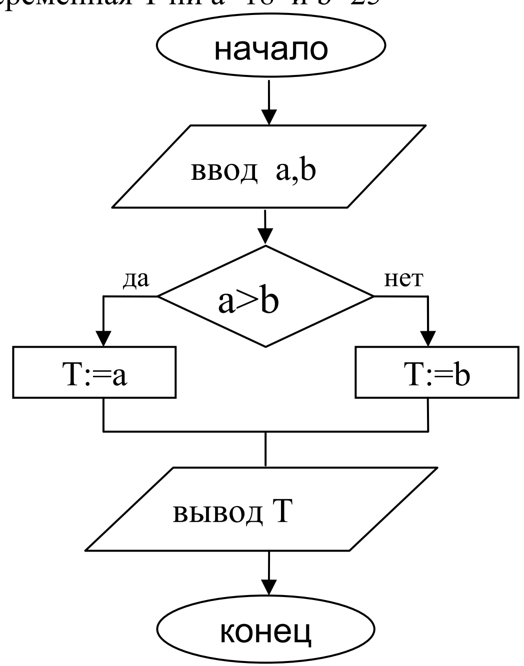
+ a=7 b=4
- a=7 b=7
- a=5 b=8

6. При каких значениях a и b выполнение алгоритма пойдет по правой ветви



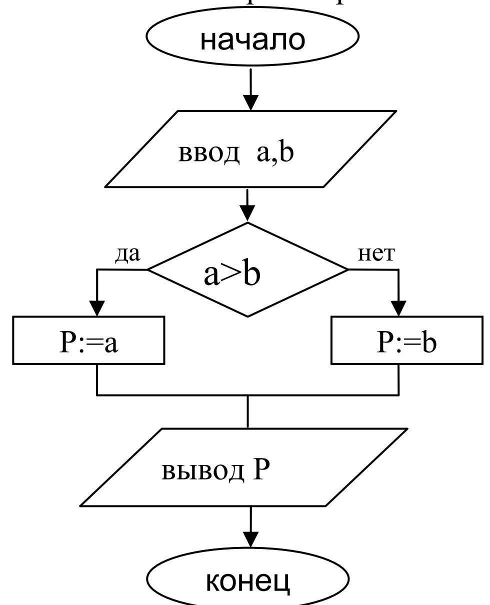
- a=7 b=4
+ a=7 b=7
+ a=5 b=8
- a=6 b=1

7. Какое значение примет переменная T пи a=18 и b=25



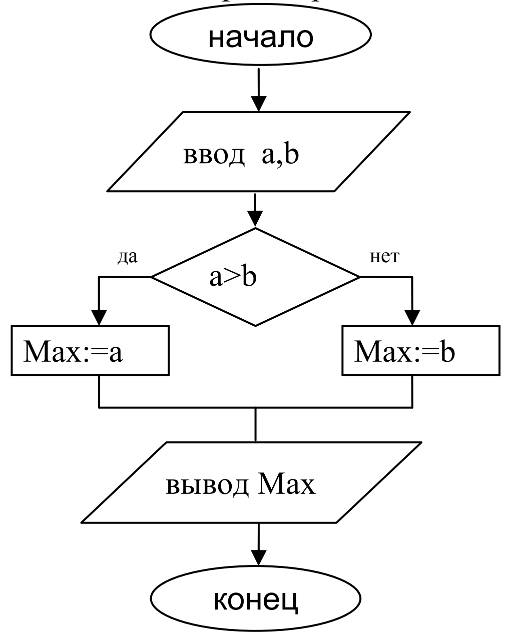
Ответ: 25.

8. При a=17 и b=11 результат выполнения алгоритма равен



Ответ: 17.

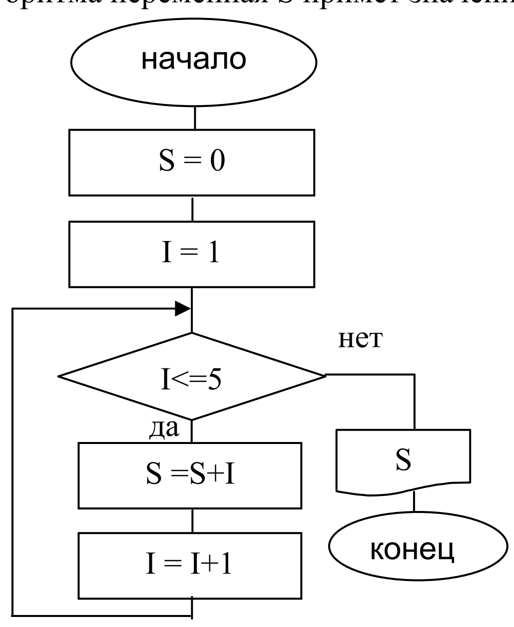
9. При a=8 и b=8 результат выполнения алгоритма равен



Ответ: 8.

10. Задание {{ 10 }} ТЗ № 186

В результате выполнения алгоритма переменная S примет значение



Ответ: 15.

11. Основными способами записи алгоритма являются...

+ словесно-формульный
+ графический
+ на алгоритмическом языке
- знаковый
- числовой

12. Свойство алгоритма, определяющее, что решение задачи должно быть представлено в виде последовательности отдельных действий, называется

+ дискретностью
- определенностью
- результативностью
- массовостью
- понятностью

13. Свойство алгоритма, определяющее, что каждый шаг алгоритма должен восприниматься однозначно и не допускать произвольной трактовки, называется

- дискретностью
+ определенностью
- результативностью
- массовостью
- понятностью

14. Свойство алгоритма, определяющее, что решение задачи должно быть получено за определенное конечное число шагов, называется

- дискретностью
- определенностью
+ результативностью
- массовостью
- понятностью

15. Свойство алгоритма, определяющее, что алгоритм должен решать некоторый класс задач, отличающихся исходными данными, называется

- дискретностью
- определенностью
- результативностью
+ массовостью
- понятностью

16. Алгоритм - это

+ четко определенная последовательность действий, которые необходимо выполнить для решения задач.
- набор данных
- результат решения задачи
- поиск решения задачи
- набор данных, которые необходимо задать для решения задачи

ФАЙЛЫ

1. Укажите процедуру закрытия файла
   + Closefile
   - Reset
   - Read
   - Eof
2. Укажите процедуру открытия файла для чтения
   - CloseFile
   + Reset
   - Read
   - Rewrite
3. Укажите процедуру открытия файла для записи
   - Close
   - Reset
   - Read
   + Rewrite
4. Установите правильную последовательность действий при работе с файлами данных
   4: Обработать файл, используя файловые процедуры и функции
   3: Открыть файл
   5: Закрыть файл
   2: Связать файловую переменную с физическим именем файла
   1: В разделе описаний объявить файловую переменную

5. Текстовые файлы могут быть объявлены следующим образом:
   - Var F: file of char;
   - Var F: file;
   + Var F: text;
   - Var F: file as text;

СТРУКТУРИРОВАННЫЙ ТИП ДАННЫХ ЗАПИСЬ

1. Структура данных, которая может содержать информацию разных типов, объединенную под одним названием, называется

    + запись
    - массив
    - множество
    - диапазон

2. Компоненты типа данных запись называются...

    + поля
    - элементы
    - данные
    - множества

3. Объявлен тип данных запись:

```
type Men = Record
      FIO, Address : string;         Year : byte;
   End;
var A : Men;
```

Отметьте правильное обращение к полям записи

    + A.Address:='пр. Ленина, д. 40, кв. 10';
    - Adress:='пр. Ленина, д. 40, кв. 10';
    - A:='пр. Ленина, д. 40, кв. 10';
    - A[Adress]:='пр. Ленина, д. 40, кв. 10';

4. Объявлен тип данных запись:

```
type Men = Record
      FIO, Address : string;         Year : byte;
   End;
var A : Men;
```

Отметьте правильное обращение к полям записи

    + A.FIO:='Иванов И.И.';
    - A:='Иванов И.И.';
    - FIO:='Иванов И.И.';
    - A[FIO]:='Иванов И.И.';

5. Объявлен тип данных запись:

```
Type Men = Record
      FIO,Address : string;         Year : byte;
   End;
Var A : Men;
```

Отметьте правильное обращение к полям записи

    + A.Year:=1981;
    - Year:=1981;
    - A:=1981;
    - A[Year]:=1981;

6. Объявлен тип данных запись:

```
Type student = Record
      FIO : string;         Year, Ball : byte;
   End;
Var A : student;
```

Отметьте правильное обращение к полям записи

    + A.Year:=1981;
    - Year:=1981;
    - A:=1981;
    - A[Year]:=1981;

КОМАНДЫ ПРИСВАИВАНИЯ, ВВОДА И ВЫВОДА

1. Какое значение примет переменная D после выполнения команд:

D:=3; D:=D*D; D:=D*D;

Ответ: 81;

2. Результат выполнения команд: C:=14; C:=C mod 3; Writeln(C);

Ответ: 2;

3. Укажите правильные формы записи оператора вывода

+ write (x, y);
+ write (x, x+1, x+2);
- write (x; y; z);
+ write (x:7:3);
- write (x-2; 2);

4. Операторы в языке PASCAL отделяются друг от друга...

- Пробелом
+ Точкой с запятой
- Точкой
- Запятой

5. Переменная X после выполнения команды X:=SQR(4)/4*2 примет значение

- 4
- 2
- 6
+ 8

6. Команда ввода значений переменных в PASCAL

+ READLN
- GET
- APPEND
- WRITELN

7. Результат выполнения следующего фрагмента кода:

X:= 5; Y:= X+1;
Writeln('X=', X, ' Y=',Y);

- X=6 Y=5
- X=5 Y=5
+ X=5 Y=6
- X=6 Y=6

8. После выполнения команды WRITELN

+ курсор переводится на новую строку
- курсор остается на прежней строке
- выводится строка пробелов

9. При выполнении команды READLN(A,B,C) вводимые значения переменных разделяются...

+ пробелом
- запятой
- точкой с запятой
- ничем не разделяются

10. Результат выполнения следующего фрагмента кода:

X:= 18; Y:= X mod 5; Y:=Y*Y;
Writeln('Y=',Y);

- Y=5
+ Y=9
- Y=18

ЦИКЛЫ

1. Оператор, реализующий в Паскале цикл с предусловием...
   - FOR...
   - REPEAT...
   + WHILE...
   - WRITE...
2. Оператор, реализующий в Паскале цикл с постусловием...
   - FOR...
   + REPEAT...
   - WHILE...
   - WRITE...
3. Оператор, реализующий в Паскале цикл с параметром...
   + FOR...
   - REPEAT...
   - WHILE...
   - WRITE...
4. Какой из перечисленных операторов цикла всегда выполняется хотя бы один раз
   - FOR...
   + REPEAT...
   - WHILE...
5. Цикл WHILE выполняется
   - всегда многократно
   + может не выполниться ни разу
   - всегда выполняется хотя бы один раз
6. Цикл FOR выполняется
   - всегда многократно
   + может не выполниться ни разу
   - всегда выполняется хотя бы один раз

УСЛОВНЫЙ ОПЕРАТОР. МАССИВЫ

1.Выберите правильные формы записи условного оператора
   - IF a>0 TO a:=1;
   + IF a>0 THEN a:=1;
   - IF a>0 ELSE a:=1;
   + IF a>0 THEN a:=1 ELSE a:=0;
   - IF a>0 TO a:=1 ELSE a:=0;
2. Укажите правильные формы записи условного оператора в языке Паскаль
   + IF a>0 THEN a:=1 ELSE begin a:=0; b:=b+1 end;
   - IF a>0 THEN a:=1 ELSE a:=0 end;
   - IF a>0 THEN a:=1 ELSE begin a:=0; b:=b+1;
   + IF a>0 THEN begin a:=1; b:=b+1; end ELSE a:=0;
3. Переменная X после выполнения команд примет значение

```
X:=-2;
If X >=0 then x:=x*2 else x:=abs(x);
```

Ответ: 2;

4. Какое значение примет переменная K в результате выполнения программы:

```
VAR
   I,K :integer;
   B: Array[1..10] of integer;
Begin
   K=0;  For I=1 to 10 do
            begin
               B[I]:=I+1;
```

```
                    If B[I] mod 3 =0 Then K:=K+1;
                end;
        Write(k);
    End.
```

Ответ: 3;

5. Имеется двухмерный массив Y. Чему равна сумма элементов Y[2, 3] и Y[3, 1]

3	1	-5
5	0	9
-2	-6	8

Ответ: 7;

6. В результате выполнения программы, переменная P примет значение равное

```
VAR   I, P :integer; A: Array[1..8] of integer;
Begin
  P:=1;
  For I:=1 to 8 do
      begin
          A[I]:=I;
          If A[I] mod 3 =0 Then P:=P*A[I];
      end;
   Write(P);
End.
```

Ответ: 18.

7. Одномерные массивы имеют описание

```
type mas=array[1..10] of integer;
var A,B:mas;
```

Каким способом не может быть заполнен массив B

- B:=A;
- For I:=1 to 10 do B[I]:=random(1);
+ Read(B);
- Все перечисленные способы не подходят для заполнения массива.

8. Массив из 10 целых чисел в языке Паскаль может быть объявлен следующим образом

+ A: array[1..10] of integer;
- A: array[10] of integer;
- A: array(10) of integer;
- A: array[1..10] integer;

9. Массив из 15 вещественных чисел в языке Паскаль может быть объявлен следующим образом

+ A: array[1..15] of real;
- A: array[15] of real;
- A: array(15) of real;
- A: array[1..15] real;

10. Таблицу из 5 строк и 7 столбцов, содержащую целые числа, можно описать следующим образом

+ A: array[1..5, 1..7] of integer;
- A: array[1..7, 1..5] of integer;
- A: array(1..5, 1..7) of integer;
- A: array[5,7] of intereg;

РАБОТА СО СТРОКАМИ

1. Какая операция не допустима над строковыми переменными A и B в языке Паскаль
   - A:=B;
   - A:=A+B;
   + A:=A-B;
   - A[1]:='B';
2. Каким ключевым словом описывается строковый тип данных
   - RECORD
   + STRING
   - ARRAY
3. Какая функция вычисляет позицию подстроки в строке
   + POS
   - LENGTH
   - INSERT
4. Какая функция возвращает длину строки
   - POS
   + LENGTH
   - INSERT
5. Какая процедура удаляет подстроку из строки
   + DELETE
   - STR
   - INSERT
6. Какая процедура преобразует число в его строковое представление
   - VAL
   + STR
   - INSERT
7. Какая процедура преобразует строку в число
   + VAL
   - STR
   - INSERT

Список тем для подготовки докладов

1. История понятия "алгоритм".
2. Известнейшие в истории математики проблемные алгоритмы.
3. Проблема существования алгоритмов в математике.
4. Средства представления алгоритмов.
5. Методы разработки алгоритмов.
6. Эволюция языков программирования.
7. История языка Паскаль.
8. Обзор современных тенденций в программировании.
9. Современные парадигмы программирования.
10. Биография Н. Вирта.
11. Роль личности в истории языков программирования.
12. Кроссплатформенные языки программирования.
13. Оценка популярности языков программирования (за последние годы).
14. Жизненный цикл разработки программных систем.
15. Модульный подход к программированию.
16. Структурный подход к программированию.
17. Объектно-ориентированный подход к программированию.
18. Декларативный подход к программированию.
19. Системы логического программирования.
20. Особенности программирования мобильных приложений.

Индивидуальные домашние задания

Вычисление сложного выражения

Составить алгоритм, реализующий вычисление сложного математического выражения $y(x)$ при заданном x.

Вариант	Выражение
1	$y(x)=\sin\left(x^2+\cos(x)-1\right)$
2	$y(x)=e^{x^2+2x-1}-5$
3	$y(x)=2\,\mathrm{tg}\left(x^2-6x+7\right)+4$
4	$y(x)=\ln\frac{2x^2+7}{5x^2+1}$
5	$y(x)=\sqrt{\frac{3x^4+2x^2+1}{x^2+7x+5}}$
6	$y(x)=\sqrt{x^2+3x+2}+\frac{9x-7}{13+x}$
7	$y(x)=\frac{3x^3-1}{9x^2+3x+1}$
8	$y(x)=\sin\frac{2x^2-x+5}{x^2+4}$
9	$y(x)=\ln\left\|\frac{3x^2+7}{9x^4+1}\right\|$
10	$y(x)=\frac{x-1}{x^2+1}+\frac{x+1}{x^2+4}$
11	$y(x)=\sin x+\cos 2x+\sin^2 2x$
12	$y(x)=\mathrm{arctg}\left(\frac{x^2+2x+1}{x^2+9}\right)$
13	$y(x)=\sqrt{\frac{x^2+2x+1}{x^4+81}}$

14	$y(x)=\frac{x+1}{x^2+4}-\frac{x-1}{x^2+9}$
15	$y(x)=\sqrt{x^2+4}+\sqrt{x^4+81}+\sqrt{x^6+1}$
16	$y(x)=\ln\left(\sin^2 x+1\right)$
17	$y(x)=\frac{x+\frac{x+1}{x^2+4}}{x^2+\frac{x+2}{x+5}}$
18	$y(x)=\frac{x+\sqrt{x^2+9}}{x^2+\sqrt{x^4+27}}$
19	$y(x)=\sqrt{x^2+\sqrt{x^2+\sqrt{x^2+9}}}$
20	$y(x)=1+\frac{1}{x^2+\frac{1}{x^2+4}}$

Вычисление кусочно-заданной функции. Табулирование функции

Составить алгоритм в виде блок-схемы и написать программу, которая будет реализовывать табулирование функции, заданной кусочно, на некотором интервале $[a,b]$ числовой оси с шагом h.

Вариант	Функция
1	$f(x)=\begin{cases}\ln x-\sqrt{x^2+\left\lvert\frac{1}{x}\right\rvert}, & \text{при } x<0;\\ 20, & \text{при } x=0;\\ \left\lvert 1-x^7\right\rvert-\sqrt{x}, & \text{при } x>0.\end{cases}$
2	$f(x)=\begin{cases}\sin x-\sqrt{x^2+3}, & \text{при } x<0;\\ 4, & \text{при } x=0;\\ \left\lvert 1-x^2\right\rvert-x, & \text{при } x>0.\end{cases}$

3	$f(x)=\begin{cases} \ln\lvert x\rvert, & \text{при } x<-2; \\ \sin x, & \text{при } -2\le x\le 2; \\ \lvert 1-x^7\rvert-\sqrt{x}, & \text{при } x>2. \end{cases}$
4	$f(x)=\begin{cases} \sqrt{x^2+1}, & \text{при } x<0; \\ 10, & \text{при } x=0; \\ -\sqrt{x}, & \text{при } x>0. \end{cases}$
5	$f(x)=\begin{cases} \sqrt{3+x^5}, & \text{при } x<0; \\ -1, & \text{при } 0\le x\le 3; \\ \sin x-\cos x, & \text{при } x>3. \end{cases}$
6	$f(x)=\begin{cases} (\sin 5x)^3, & \text{при } \lvert x\rvert<2; \\ e^{3x}, & \text{при } \lvert x\rvert\ge 2. \end{cases}$
7	$f(x)=\begin{cases} x-\sqrt{x^4+\left\lvert\frac{1}{x}\right\rvert}, & \text{при } x<0; \\ \cos 4x, & \text{при } x=0; \\ \lvert 1+x^5\rvert-\sqrt{2x}, & \text{при } x>0. \end{cases}$
8	$f(x)=\begin{cases} 7x+4\arccos^2(2x), & \text{при } \lvert x\rvert<5; \\ \frac{\operatorname{arctg} x}{3x}, & \text{при } \lvert x\rvert\ge 5. \end{cases}$
9	$f(x)=\begin{cases} 5-e^{2x}, & \text{при } x<0; \\ \cos 3x, & \text{при } x=0; \\ arcctg\, 5x^{5x}, & \text{при } x>0. \end{cases}$
10	$f(x)=\begin{cases} \sin^2 2x+3\operatorname{tg}^4 x, & \text{при } \lvert x\rvert<3; \\ \sqrt{\lvert arcctg\, 2x-5\rvert}, & \text{при } \lvert x\rvert\ge 3. \end{cases}$
11	$f(x)=\begin{cases} \sin^2 2x-6\cos^3 x, & \text{при } \lvert x\rvert<5; \\ \operatorname{tg} 2x+4, & \text{при } \lvert x\rvert\ge 5. \end{cases}$

12	$f(x)=\begin{cases}\sqrt{1+x^2}, & \text{при } x<0;\\ 1, & \text{при } x=0;\\ 3x^{2x-1}, & \text{при } x>0.\end{cases}$
13	$f(x)=\begin{cases}\operatorname{ctg}^2 2x+2\cos x, & \text{при } \lvert x\rvert<2;\\ \ln\lvert 3x\rvert-5, & \text{при } \lvert x\rvert\ge 5.\end{cases}$
14	$f(x)=\begin{cases}\sin^3 2x+3\operatorname{arctg} x, & \text{при } \lvert x\rvert<2;\\ e^{3x}-5\sin x, & \text{при } \lvert x\rvert\ge 2.\end{cases}$
15	$f(x)=\begin{cases}\sqrt{x^6-10x^2}, & \text{при } x<0;\\ 13, & \text{при } x=0;\\ 10\sqrt{x}+1, & \text{при } x>0.\end{cases}$
16	$f(x)=\begin{cases}3\sin^2 5x, & \text{при } \lvert x\rvert<2;\\ e^{2x}-1, & \text{при } \lvert x\rvert\ge 2.\end{cases}$
17	$f(x)=\begin{cases}\sin^3 5x+2\cos^2 x, & \text{при } \lvert x\rvert<2;\\ \operatorname{ctg} 2x-1, & \text{при } \lvert x\rvert\ge 2.\end{cases}$
18	$f(x)=\begin{cases}2x-4\arcsin^2 x, & \text{при } \lvert x\rvert<3;\\ \operatorname{tg} 2x+\sin x, & \text{при } \lvert x\rvert\ge 3.\end{cases}$
19	$f(x)=\begin{cases}3^x, & \text{при } \lvert x\rvert<3;\\ \sqrt{x^2-3}, & \text{при } \lvert x\rvert\ge 3.\end{cases}$
20	$f(x)=\begin{cases}\sin^3 2x-\sqrt{x^5+10}, & \text{при } x<0;\\ 15, & \text{при } x=0;\\ \lvert 8-x^2\rvert-10x, & \text{при } x>0.\end{cases}$

Обработка массивов

Разработать программу, генерирующую массив размерностью 7х7, заполняемый случайными целыми числами в диапазоне [-10, 10] и обрабатывающую этот массив в соответствии с заданием из таблицы.

Предусмотреть форматированный вывод исходного и конечного содержимого массива. Нумерацию элементов начинать с единицы.

Вариант	Обработка массива
1	Переставить строки исходного массива так, чтобы на главной диагонали были максимальные элементы
2	Переставить столбцы исходного массива так, чтобы на главной диагонали были максимальные элементы
3	В каждой строке осуществить перестановку по следующему правилу – если первый элемент нечетный, то упорядочить элементы по возрастанию, если четный – по убыванию, если ноль – оставить строку без изменений.
4	В каждом столбце осуществить перестановку по следующему правилу – если первый элемент столбца нечетный, то упорядочить элементы по возрастанию, если четный – по убыванию, если ноль – оставить порядок элементов без изменений.
5	Заменить все отрицательные элементы нулями
6	Поменять местами строки и столбцы массива (транспонировать массив)
7	У каждого элемента строки с четным номером изменить знак на противоположный
8	У каждого элемента столбца с нечетным номером изменить знак на противоположный
9	Для каждого элемента, сумма индексов которого четная, изменить знак на противоположный
10	Поменять местами максимальный и минимальный элемент в каждой строке
11	Поменять местами максимальный и минимальный элемент в каждом столбце

12	Поменять местами максимальный и минимальный элемент всего массива
13	Для элементов, величина которого меньше центрального (индекс 3-3), удвоить значения
14	Для элементов, величина которых меньше полусуммы максимального и минимального элементов, удвоить значения
15	Для всех отрицательных элементов изменить знак на противоположный
16	Для всех положительных элементом изменить знак на противоположный
17	Для всех элементов с четным значением выполнить целочисленное деление на 2
18	Все элементы с нечетным значением увеличить вдвое
19	Для элементов, величина которых по модулю больше 5, изменить знак на противоположный
20	Для элементов, величина которых по модулю больше 7, изменить знак на противоположный

Обработка текстовых строк

Разработать программу, считывающую с клавиатуры текстовую строку (не более 100 символов) из символов английского алфавита, и выполняющую задание из таблицы.

Вариант	Задание на обработку строки
1	Подсчитать количество заглавных букв
2	Подсчитать количество различных заглавных букв
3	Подсчитать количество знаков препинания (точка, запятая, тире)
4	Убрать все подряд идущие одинаковые буквы

5	Исправить ошибочные пробелы перед знаками препинания
6	Убрать дублирование пробелов
7	Подсчитать количество введенных слов (под словом понимается часть строки, ограниченная с обеих сторон пробелами и/или знаком препинания, а также началом или концом строки)
8	Подсчитать количество гласных букв
9	Заменить все строчные гласные буквы на заглавные
10	Заменить все заглавные гласные буквы на строчные
11	Найти наиболее часто встречающуюся букву (регистр различать)
12	Найти наиболее часто встречающуюся букву (без учета регистра)
13	При наличии в строке цифр – оставить только их, в противном случае вывести строку – цифр нет
14	Найти самую длинную цепочку повторяющихся символов
15	Найти самую длинную цепочку, состоящую из цифр
16	Сделать первую букву каждого слова заглавной
17	Сделать все буквы каждого слова строчными
18	Подсчитать количество слов с заглавными буквами
19	После каждой цифры вставить пробел
20	Убрать пробелы между цифрами