МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Поволжский государственный университет телекоммуникаций и информатики»

КОЛЛЕДЖ СВЯЗИ

Основы алгоритмизации и программирования

Учебное пособие

для специальности:

– 10.02.05 – «Обеспечение информационной безопасности автоматизированных систем»

Составила

преподаватель Лобачева М.Е.

Самара

2021

Рассмотрено на заседании П(Ц)К
«Информационные системы и технологии»
Протокол № 2 от 07.10.2021г.
Председатель П(Ц)К _____ Шомас Е.А.

Утверждаю
Зам. директора по УВР
_____ Логвинов А.В.
« 07 » октября 2021г.

Основы алгоритмизации и программирования. Учебное пособие. Составлено преподавателем КС ПГУТИ Лобачевой М.Е., Самара КС ПГУТИ, 2021г. – 5,6 п.л.

Содержание

Раздел 1. Основные принципы алгоритмизации и программирования

Тема 1.1 Основные понятия алгоритмизации

Одним из фундаментальных понятий в информатике является понятие алгоритма. Происхождение самого термина «алгоритм» связано с математикой. Это слово происходит от Algorithmi – латинского написания имени Мухаммеда аль-Хорезми (787 – 850) выдающегося математика средневекового Востока. В своей книге "Об индийском счете" он сформулировал правила записи натуральных чисел с помощью арабских цифр и правила действий над ними столбиком. Алгоритм может быть предназначен для выполнения его человеком или автоматическим устройством. В XII в. был выполнен латинский перевод его математического трактата, из которого европейцы узнали о десятичной позиционной системе счисления и правилах арифметики многозначных чисел. Именно эти правила в то время называли алгоритмами.

В настоящее время понятие алгоритма трактуется шире.

Алгоритмом называется точное описание, определяющее последовательность действий исполнителя, направленных на решение поставленной задачи.

Мы постоянно сталкиваемся с этим понятием в различных сферах деятельности (кулинарные книги, инструкции по использованию различных приборов, правила решения математических задач и т.д.). Мир алгоритмов очень разнообразен. Несмотря на это, удается выделить общие свойства, которыми обладает любой алгоритм.

Свойства алгоритма

1. Дискретность - разбиение алгоритма на ряд отдельных законченных действий - шагов. Каждое действие должно быть закончено исполнителем алгоритма прежде, чем он приступит к исполнению следующего действия.
2. Понятность - однозначное понимание и исполнение каждого шага алгоритма его исполнителем. Алгоритм должен быть записан на понятном для исполнителя языке.
3. Результативность (или конечность) - обязательное получение результата за конечное число шагов. Каждый шаг (и алгоритм в целом) после своего завершения дает среду, в которой все объекты однозначно определены. Если это по каким-либо причинам невозможно, то алгоритм должен сообщать, что решение задачи не существует. Работа алгоритма должна быть завершена за конечное число шагов.

4. Определенность (детерминированность, точность) - свойство алгоритма, указывающее на то, что каждый шаг алгоритма должен быть строго определен и не допускать различных толкований. Также строго должен быть определен порядок выполнения отдельных шагов.

5. Массовость - применимость алгоритма ко всем задачам рассматриваемого типа, при любых исходных данных

Формы записи алгоритмов

На практике наиболее распространены следующие формы представления алгоритмов: словесная, графическая (блок-схемы), псевдокод и программа.

Словесный способ записи алгоритмов представляет собой описание алгоритма на естественном языке.

Пример. Записать алгоритм нахождения наибольшего общего делителя (НОД) двух натуральных чисел.

Алгоритм может быть следующим:

1. задать два числа;

2. если числа равны, то взять любое из них в качестве ответа и остановиться, в противном случае продолжить выполнение алгоритма;

3. определить большее из чисел;

4. заменить большее из чисел разностью большего и меньшего из чисел;

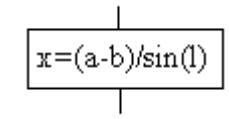
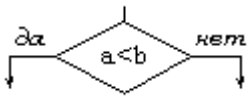
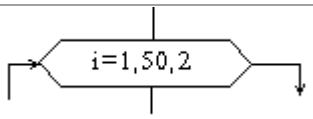
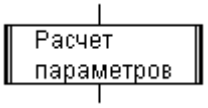
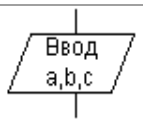
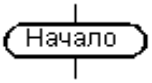
5. повторить алгоритм с шага 2.

Описанный алгоритм применим к любым натуральным числам и должен приводить к решению поставленной задачи.

Словесный способ не имеет широкого распространения по следующим причинам:

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

Графический способ (или блок-схема) - описание структуры алгоритма с помощью последовательности связанных между собой функциональных блоков, каждый из которых соответствует выполнению одного или нескольких действий. В блок-схеме каждому типу действий (вводу исходных данных, вычислению значений выражений, проверке условий, управлению повторением действий, окончанию обработки и т.п.) соответствует геометрическая фигура, представленная в виде блочного символа. Блочные символы соединяются линиями переходов, определяющими очередность выполнения действий. В таблице приведены наиболее часто употребляемые символы.

Название символа	Обозначение и пример заполнения	Пояснение
Процесс		Вычислительное действие или последовательность действий
Решение		Проверка условий
Модификация		Начало цикла
Предопределенный процесс		Вычисления по подпрограмме, стандартной подпрограмме
Ввод-вывод		Ввод-вывод в общем виде
Пуск-останов		Начало, конец алгоритма, вход и выход в подпрограмму
Документ		Вывод результатов на печать

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

Блок **"решение"** используется для обозначения переходов управления по условию. В каждом блоке "решение" должны быть указаны вопрос, условие или сравнение, которые он определяет.

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

Блок **"предопределенный процесс"** используется для указания обращений к вспомогательным алгоритмам, существующим автономно в виде некоторых самостоятельных модулей, и для обращений к библиотечным подпрограммам.

Псевдокод - описание структуры алгоритма на естественном, частично формализованном языке, позволяющее выявить основные этапы решения задачи, перед точной его записью на языке программирования. В псевдокоде используются некоторые формальные конструкции и общепринятая математическая символика.

Псевдокод занимает промежуточное место между естественным и формальным языками.

• Он близок к обычному естественному языку, поэтому алгоритмы могут на нем записываться и читаться как обычный текст.

• В псевдокоде используются некоторые формальные конструкции и математическая символика, что приближает запись алгоритма к общепринятой математической записи.

В псевдокоде не приняты строгие синтаксические правила для записи команд, присущие формальным языкам, что облегчает запись алгоритма на стадии его проектирования и дает возможность использовать более широкий набор команд, рассчитанный на абстрактного исполнителя.

Однако в псевдокоде обычно имеются некоторые конструкции, присущие формальным языкам, что облегчает переход от записи на псевдокоде к записи алгоритма на формальном языке. В частности, в псевдокоде, так же, как и в формальных языках, есть служебные слова, смысл которых определен раз и навсегда. Они выделяются в печатном тексте жирным шрифтом, а в рукописном тексте подчеркиваются.

Единого или формального определения псевдокода не существует, поэтому возможны различные псевдокоды, отличающиеся набором служебных слов и основных (базовых) конструкций.

Программа - описание структуры алгоритма на языке алгоритмического программирования.

Этапы решения задач на ЭВМ

Решение задачи разбивается на этапы:

1. Постановка задачи. При постановке задачи выясняется конечная цель и вырабатывается общий подход к решению задачи. Выясняется, сколько решений имеет задача и имеет ли их вообще. Изучаются общие свойства рассматриваемого физического явления или объекта, анализируются возможности данной системы программирования.

2. Формализация (математическая постановка). На этом этапе все объекты задачи описываются на языке математики, выбирается форма хранения данных, составляются все необходимые формулы.

3. Построение алгоритма. На этом этапе метод решения записывается применительно к данной задаче на одном из алгоритмических языков (чаще на графическом).

4. Составление программы на языке программирования. Переводим решение задачи на язык, понятный машине.
5. Отладка и тестирование программы.
6. Проведение расчетов и анализ полученных результатов.

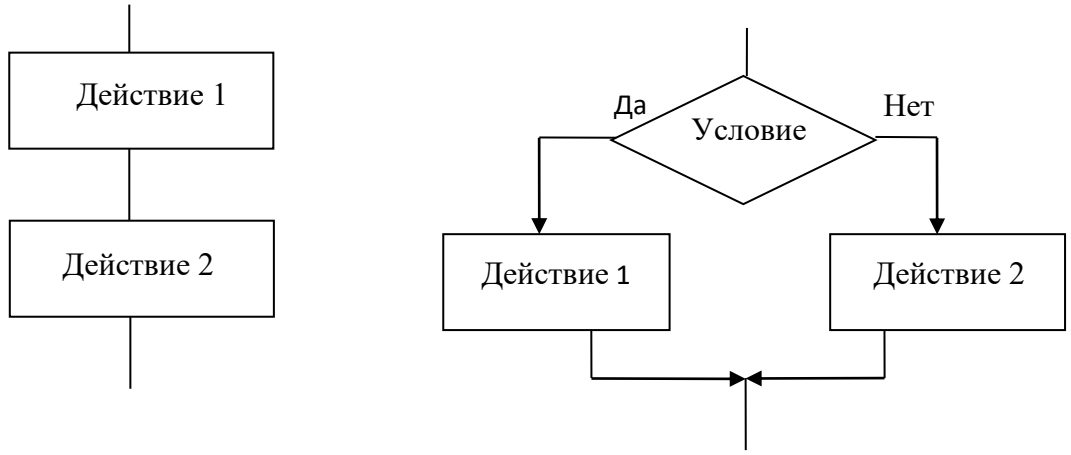
Основные алгоритмические конструкции

Элементарные шаги алгоритма можно объединить в следующие алгоритмические конструкции: линейные (последовательные), разветвляющиеся, циклические.

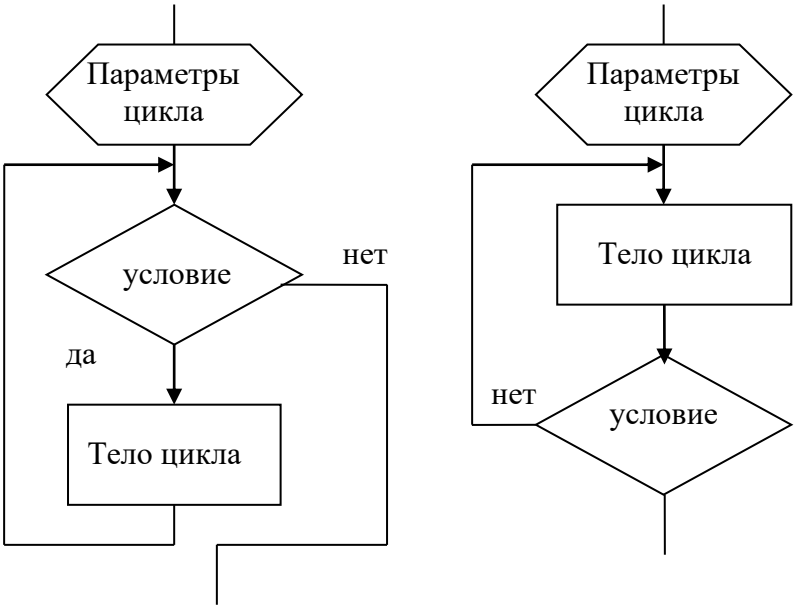
Линейной называют алгоритмическую конструкцию, реализованную в виде последовательности действий (шагов), в которой каждое действие (шаг) алгоритма выполняется ровно один раз.

Разветвляющейся называется алгоритмическая конструкция, обеспечивающая выбор между двумя альтернативами в зависимости от значения входных данных.

Линейный алгоритм **Разветвляющийся алгоритм**



Циклической называют алгоритмическую конструкцию, в которой некоторая, идущая подряд группа действий (шагов) алгоритма может выполняться несколько раз, в зависимости от входных данных или условия задачи.



Данные и величины

Совокупность величин, с которыми работает компьютер, принято называть *данными*. По отношению к программе данные делятся на исходные, результаты (окончательные данные) и промежуточные, которые получаются в процессе вычислений.

Любые данные, т.е. константы, переменные, значения функций или выражения характеризуются своими типами. *Тип* определяет множество допустимых значений, которые может принимать тот или иной объект, а также множество допустимых операций, которые применимы к нему. Кроме того, тип определяет формат внутреннего представления данных в памяти ПК. В каждом языке программирования существует своя концепция типов данных. Тем не менее, в любой язык входит минимальный набор основных типов данных, к которому относятся: целый, вещественный, логический и символьный типы.

Тема 1.2 Языки и системы программирования

Развитие вычислительной техники сопровождается созданием новых и совершенствованием существующих средств общения программистов с ЭВМ - языков программирования (ЯП).

Под ЯП понимают правила представления данных и записи алгоритмов их обработки, которые автоматически выполняются ЭВМ. В более абстрактном виде ЯП является средством создания программных моделей объектов и явлений внешнего мира.

К настоящему времени созданы десятки различных ЯП от самых примитивных до близких к естественному языку человека. Чтобы разобраться во всем многообразии ЯП, нужно знать их классификацию, а также историю создания, эволюцию и тенденции развития.

Чтобы понимать тенденции развития ЯП, нужно знать движущие силы их эволюции. Для выяснения этого вопроса будем рассматривать ЯП с различных точек зрения.

Во-первых, ЯП является инструментом программиста для создания программ. Для создания хороших программ нужны хорошие ЯП. Поэтому одной из движущих сил эволюции ЯП является стремление разработчиков к созданию более совершенных программ.

Во-вторых, процесс разработки программы можно сравнивать с промышленным производством, в котором определяющими факторами являются производительность труда коллектива программистов, себестоимость и качество программной продукции. Создаются различные технологии разработки программ (структурное, модульное, объектно-ориентированное программирование и другие), которые должны поддерживаться ЯП. Поэтому второй движущей силой эволюции ЯП является стремление к повышению эффективности процесса производства программной продукции.

В-третьих, программы можно рассматривать как аналог радиоэлектронных устройств обработки информации, в которых вместо радиодеталей и микросхем используют конструкции ЯП (элементная база программы). Как и электронные устройства, программы могут быть простейшими (уровня детекторного приемника) и очень сложными (уровня автоматической космической станции), при этом уровень инструмента должен соответствовать сложности изделия. Кроме того, человеку удобнее описывать моделируемый объект в терминах предметной области, а не языком цифр. Поэтому третьей движущей силой, ведущей к созданию новых, специализированных, ориентированных на проблемную область и более мощных ЯП, является увеличение разнообразия и повышение сложности задач, решаемых с помощью ЭВМ.

В-четвертых, совершенствование самих ЭВМ приводит к необходимости создания языков, максимально реализующих новые возможности ЭВМ.

В-пятых, программы являются интеллектуальным продуктом, который нужно накапливать и приумножать. Но программы, как и технические изделия, обладают свойством морального старения, одной из причин которого является их зависимость от типа ЭВМ и операционной сре-

ды. С моральным старением программ борются путем их модернизации и выпуска новых версий, однако при высокой динамике смены типов ЭВМ и операционных сред разработчики будут только тем и заниматься, что модернизировать старые программы. Поэтому, ЯП должен обеспечивать продолжительный жизненный цикл программы, и стремление к этому является пятой движущей силой развития ЯП.

История развития языков программирования

Первые ЭВМ, созданные человеком, имели небольшой набор команд и встроенных типов данных, но позволяли выполнять программы на машинном языке. Машинный язык (МЯ) - единственный язык, понятный ЭВМ. Он реализуется аппаратно: каждую команду выполняет некоторое электронное устройство. Программа на МЯ представляет собой последовательность команд и данных, заданных в цифровом виде. Данные на МЯ представлены числами и символами. Операции являются элементарными, и из них строится вся программа. Ввод программы в цифровом виде производился непосредственно в память с пульта ЭВМ либо с примитивных устройств ввода. Естественно, что процесс программирования был очень трудоемким, разобраться в программе даже автору было довольно сложно, а эффект от применения ЭВМ был довольно низким. Этот этап в развитии ЯП показал, что программирование является сложной проблемой, трудно поддающейся автоматизации, но именно программное обеспечение определяет в конечном счете эффективность применения ЭВМ. Поэтому на всех последующих этапах усилия направлялись на совершенствование интерфейса между программистом и ЭВМ - языка программирования.

Стремление программистов оперировать не цифрами, а символами, привело к созданию мнемонического языка программирования, который называют ассемблером, мнемокодом, автокодом. Этот язык имеет определенный синтаксис записи программ, в котором, в частности, цифровой код операции заменен мнемоническим кодом. Например, команда сложения записывается в виде AR 1,2 и означает сложение (Addition) типа регистр-регистр (Register) для регистров 1 и 2. Теперь программа имеет более удобочитаемую форму, но ее не понимает ЭВМ. Поэтому понадобилось создать специальную программу транслятор, который преобразует программу с языка ассемблера на МЯ. Эта проблема потребовала, в свою очередь, глубоких научных исследований и разработки различных теорий, например теорию формальных языков, которые легли в основу создания трансляторов. Практически любой класс ЭВМ имеет свой язык ассемблера. На сегодняшний день язык ассемблера используется для создания системных программ, использующих специфические аппаратные возможности данного класса ЭВМ.

Следующий этап характеризуется созданием языков высокого уровня (ЯВУ). Эти языки являются универсальными (на них можно создавать любые прикладные программы) и алгоритмически полными, имеют более широкий спектр типов данных и операций, поддерживают тех-

нологии программирования. На этих языках создается неисчислимое множество различных прикладных программ. Принципиальными отличиями ЯВУ от языков низкого уровня являются:

· использование переменных;

· возможность записи сложных выражений;

· расширяемость типов данных за счет конструирования новых типов из базовых;

· расширяемость набора операций за счет подключения библиотек подпрограмм;

· слабая зависимость от типа ЭВМ.

С усложнением ЯП усложняются и трансляторы для них. Теперь в набор инструментов программиста, кроме транслятора, входит текстовый редактор для ввода текста программ, отладчик для устранения ошибок, библиотекарь для создания библиотек программных модулей и множество других служебных программ. Все вместе это называется системой программирования. Наиболее яркими представителями ЯВУ являются FORTRAN, PL/1, Pascal, C, Basic, Ada.

Одновременно с развитием универсальных ЯВУ стали развиваться проблемно-ориентированные ЯП, которые решали экономические задачи (COBOL), задачи реального времени (Modula-2, Ada), символьной обработки (Snobol), моделирования (GPSS, Simula, SmallTalk), численно-аналитические задачи (Analitic) и другие. Эти специализированные языки позволяли более адекватно описывать объекты и явления реального мира, приближая язык программирования к языку специалиста в проблемной области.

Другим направлением развития ЯП является создание языков сверхвысокого уровня (ЯСВУ). На языке высокого уровня программист задает процедуру (алгоритм) получения результата по известным исходным данным, поэтому они называются процедурными ЯП. На ЯСВУ программист задает отношения между объектами в программе, например систему линейных уравнений, и определяет, что нужно найти, но не задает как получить результат. Такие языки еще называют непроцедурными, т.к. сама процедура поиска решения встроена в язык (в его интерпретатор). Такие языки используются, например, для решения задач искусственного интеллекта (Lisp, Prolog) и позволяют моделировать мыслительную деятельность человека в процессе поиска решений. К непроцедурным языкам относят и языки запросов систем управления базами данных (QBE, SQL).

Классификация языков программирования

Исходя из вышесказанного, ЯП можно классифицировать по следующим признакам.

1. По степени ориентации на специфические возможности ЭВМ ЯП делятся на:

· машинно-зависимые;

· машинно-независимые.

К машинно-зависимым ЯП относятся машинные языки, ассемблеры и автокоды, которые используются в системном программировании. Программа на машинно-зависимом ЯП может выпол-

няться только на ЭВМ данного типа. Программа на машинно-независимом ЯП после трансляции на машинный язык становится машинно-зависимой. Этот признак ЯП определяет мобильность получаемых программ (возможность переноса на ЭВМ другого типа).

2. По степени детализации алгоритма получения результата ЯП делятся на:
   · языки низкого уровня;
   · языки высокого уровня;
   · языки сверхвысокого уровня.
3. По степени ориентации на решение определенного класса задач:
   · проблемно-ориентированные;
   · универсальные.
4. По возможности дополнения новыми типами данных и операциями:
   · расширяемые;
   · нерасширяемые.
5. По возможности управления реальными объектами и процессами:
   · языки систем реального времени;
   · языки систем условного времени.
6. По способу получения результата:
   · процедурные;
   · непроцедурные.
7. По типу решаемых задач:
   · языки системного программирования;
   · языки прикладного программирования.
8. Непроцедурные языки по типу встроенной процедуры поиска решений делятся на:
   · реляционные;
   · функциональные;
   · логические.

Рассмотренная схема классификации позволяет каждому ЯП присвоить один из признаков каждого класса.

Системы программирования

Система программирования – это комплекс средств, предназначенный для создания и эксплуатации программ на конкретном языке программирования на ЭВМ определенного типа.

Традиционными средствами разработки программ являются алгоритмические (процедурные) языки программирования. Для создания программы на выбранном языке программирования нужно иметь следующие компоненты:

- Текстовый редактор – это редактор, который позволяет набрать текст программы на языке программирования. Для этой цели можно использовать любые текстовые редакторы, но лучше пользоваться специализированным текстовым редактором.
- Транслятор – это основа систем программирования. Трансляторы языков программирования, т. е. программы, обеспечивающие перевод исходного текста программы на машинный язык (объектный код), бывают двух типов: интерпретаторы и компиляторы.
- Интерпретатор - это транслятор, который обеспечивает последовательный синхронный «перевод» и выполнение каждой строки программы, причем при каждом запуске программы на выполнение вся процедура полностью повторяется. Достоинством интерпретатора является удобство отладки программы в интерактивном режиме, а недостатком - малая скорость работы.
- Компилятор – это транслятор, который исходный текст программы переводит в машинный код. Если в тексте программы нет синтаксических ошибок, то машинный код будет создан. Но это, как правило, не работоспособный код, т.к. в этой программе не хватает подпрограмм стандартных функций, поэтому компилятор выдает промежуточный код, который называется объектным кодом и имеет расширение .obj.
- Редактор связей (сборщик) – это программа, которая объединяет объектные модули отдельных частей программы и добавляет к ним стандартные модули подпрограмм стандартных функций (файлы с расширением .lib), которые содержатся в библиотеках, поставляемых вместе с компилятором, в единую программу, готовую к исполнению, т.е. создает исполнимый .exe файл. Этот файл имеет самостоятельное значение и может работать под управлением той (или такой же) операционной системы, в которой он создан.

В стандартную поставку, как правило, входят текстовый редактор, компилятор, редактор связей (сборщик), библиотеки стандартных функций. Но хорошая интегрированная система обязательно включает в себя специализированный текстовый редактор, в котором выделяются ключевые слова различными цветами и шрифтами. Все этапы создания программы в ней автоматизированы: после того, как исходный текст программы введен, его компиляция и сборка осуществляются одним нажатием клавиши.

В современных интегрированных системах имеется еще один компонент – отладчик. Он позволяет анализировать работу программы по шагам во время ее выполнения, наблюдая, как меняются значения различных переменных.

В последние несколько лет созданы среды быстрого проектирования, в которых программирование, по сути, заменяется проектированием. В проектируемое окно готовые визуальные компоненты перетаскиваются с помощью мыши, затем свойства и поведение компонентов настраивается с помощью редактора. Исходный же текст программы, ответственный за работу этих элементов, генерируется автоматически с помощью среды быстрого проектирования, которая называется RAD-средой. Подобный подход называется визуальным программированием.

Тема 1.3 Методы программирования

Структурное программирование

Основная идея структурного программирования состоит в том, что структура программы должна отражать структуру решаемой задачи, чтобы алгоритм программы был ясно виден из исходного текста. Следовательно, надо разбить программу на последовательность модулей, каждый из которых выполняет одно или несколько действий. Требование к модулю – чтобы его выполнение начиналось с первой команды и заканчивалось последней. Модульность – это основная характеристика структурного программирования. А для этого надо иметь средства для создания программы не только с помощью трех простых операторов, но и с помощью средств более точно отражающих конкретную структуру алгоритма.

С этой целью в программирование введено понятие подпрограммы – набора операторов, выполняющих нужное действие и не зависящих от других частей исходного кода. Программа разбивается на множество подпрограмм, каждая из которых выполняет одно из действий исходного кода. Комбинируя эти блоки, удается сформировать итоговый алгоритм уже не из операторов, а из законченных блоков. Обращаться к блокам надо по названиям, а название несет смысловую нагрузку. Например, Call Summa, означает обращение к подпрограмме с именем Summa, Call - вызов. При структурном подходе к составлению алгоритмов и программ используются три основные типа алгоритмов: условные, циклические алгоритмы и подпрограммы.

Модульное программирование

Модульное программирование является развитием и совершенствованием процедурного программирования и библиотек специальных программ. Основная черта модульного программирования - стандартизация интерфейса между отдельными программными единицами. Модуль - это отдельная функционально-законченная программная единица, которая структурно оформляется стандартным образом по отношению к компилятору и по отношению к объединению ее с другими аналогичными единицами и загрузке. Как правило, каждый модуль содержит паспорт, в котором указаны все основные его характеристики: язык программирования, объем, входные и выходные переменные, их формат, ограничения на них, точки входа, параметры настройки и т.д.

Модульное программирование - это искусство разбиения задачи на некоторое число различных модулей, умение широко использовать стандартные модули путем их параметрической настройки, автоматизация сборки готовых модулей из библиотек, банков модулей.

Основные концепции модульного программирования:

-каждый модуль реализует единственную независимую функцию;

-каждый модуль имеет единственную точку входа и выхода;

-размер модуля по возможности должен быть минимизирован;

-каждый модуль может быть разработан и закодирован различными членами бригады программистов и может быть отдельно протестирован;

-вся система построена из модулей;

-модуль не должен давать побочных эффектов;

-каждый модуль не зависит от того, как реализованы другие модули.

Объектно-ориентированное программирование

Концепция ООП возникла в середине 80-х годов. Главная ее идея в том, что программное приложение, как и окружающий нас мир, должно состоять из объектов, обладающих собственными свойствами и поведением. ООП объединяет исполняемый код программы и ее данные в объекты, что упрощает создание сложных программных приложений. Например, можно организовать коллективную работу над проектом, где каждый участник создает собственный класс объектов, который становится доступным другим участникам проекта.

При объектно-ориентированном подходе программные задачи распределяются между объектами программы. Объекты обладают определенным набором свойств, методов и способностью реагировать на события (нажатие кнопок мыши, интервалы времени и т.д.). В отличие от процедурного программирования, где порядок выполнения операторов программы определяется порядком их следования и командами управления, в ООП порядок выполнения процедур и функций определяется, прежде всего, событиями.

Чтобы проект можно было считать объектно-ориентированным, объекты должны удовлетворять некоторым требованиям. Этими требованиями являются инкапсуляция, наследование и полиморфизм.

Инкапсуляция — означает, что объекты скрывают детали своей работы. Инкапсуляция позволяет разработчику объекта изменять внутренние принципы его функционирования, не оказывая никакого влияния на пользователя объекта.

Наследование — означает, что новый объект можно определить на основе уже существующих объектов, при этом он будет содержать все свойства и методы родительского. Наследование полезно, когда требуется создать новый объект, обладающий дополнительными свойствами по сравнению со старым.

Полиморфизм — многие объекты могут иметь одноименные методы, которые могут выполнять разные действия для разных объектов. Например, оператор "+" для числовых величин выполняет сложение, а для текстовых — склеивание.

В ООП центральным является понятие класса. Класс – это шаблон, по которому создаются объекты определенного типа. Класс объединяет в себе данные и методы их обработки.

Объекты — это экземпляры определенного класса.

Элементы управления — это объекты, используемые при разработке пользовательского интерфейса.

Раздел 2. Программирование на языке Pascal

Тема 2.1 Элементы языка

Разработка программ на Паскале включает в себя следующие действия (этапы разработки программы): ввод и редактирование текста программы на языке программирования Паскаль, ее трансляцию, отладку.

Для выполнения каждого этапа применяются специальные средства: для ввода и редактирования текста используется *редактор текстов*, для трансляции программы - *компилятор*, для построения исполняемого компьютером программного модуля с объединением разрозненных откомпилированных модулей и библиотекой стандартных процедур Паскаля - *компоновщик (linker),* для отладки программ с анализом ее поведения, поиском ошибок, просмотром и изменением содержимого ячеек памяти компьютера- *отладчик (debugger).*

Основные элементы языка Pascal

Алфавит языка состоит из множества символов, включающих в себя буквы, цифры и специальные символы.

Латинские буквы: от A до Z (прописные) и от a до z (строчные), символ “подчеркивания”, который в языке считается буквой.

Цифры: арабские от 0 до 9 и шестнадцатеричные (первые 10 цифр от 0 до 9 - арабские, остальные шесть - латинские буквы: A, B, C, D, E, F).

Специальные символы: + - * / = , . : ; < > [ ] ( ) { } ', $, пары <> <= >= := (* *) (. .), пробел (символы (. .) соответствуют символам [ ], несколько пробелов считаются одним).

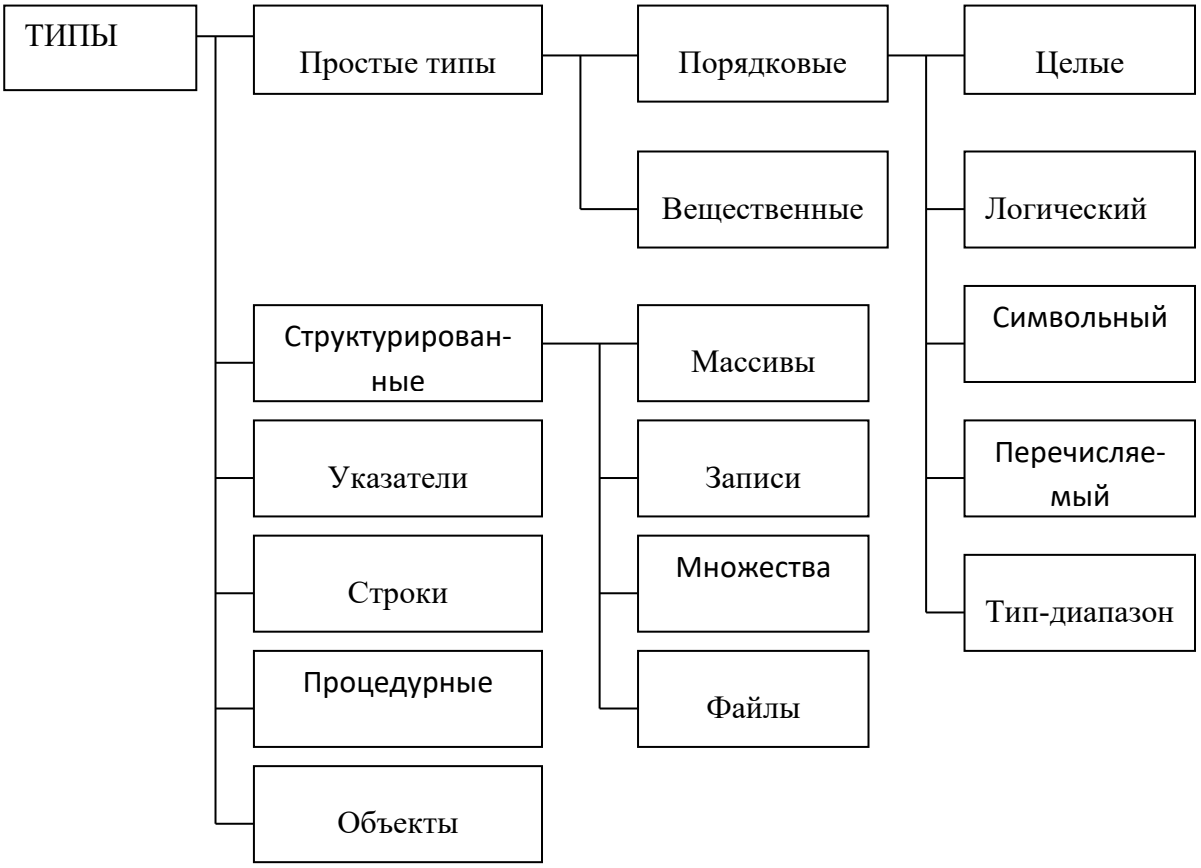
К спецсимволам относятся также служебные слова - abs, and, array, begin, case, const, dir, do, downto, else, end, for, function, goto, if, int, label, mod, not, of, or, procedure, program, repeat, shr, then, to, type, var, while, with и др. Смысл зарезервированных слов фиксирован строго. При этом набор зарезервированных слов может меняться от версии к версии.

Идентификатор - это последовательность букв, цифр и знаков подчеркивания, начинающихся не с цифры. Под идентификатором мы будем понимать ячейку памяти ЭВМ, которая имеет свое имя и в которой хранится информация. В Паскале строчные и прописные буквы в идентификаторах и служебных словах не различаются. Идентификаторы могут иметь произвольную длину, но значащими являются только первые 63 символа. Хорошим стилем является осмысленный выбор имени идентификатора. Зарезервированные слова не могут использоваться в качестве идентификаторов.

Комментарии заключаются либо в фигурные скобки **{** комментарий 1 **}**, либо в символы **(*** комментарий 2 ***)** и могут занимать любое количество строк. Последовательность из трех символов **(*)** начинает комментарий до конца строки. Текст комментария игнорируется при компиляции, если это не директивы компилятора, которые имеют вид **{$ }**.

Типы данных

Концепция типов данных является одной из центральных в любом языке программирования. Паскаль характеризуется разветвлённой структурой типов данных.



ПРОСТЫЕ ТИПЫ

К простым типам относятся порядковые и вещественные типы.

Порядковые типы отличаются тем, что каждый из них имеет конечное число возможных значений. Эти значения можно определённым образом упорядочить (отсюда – название типов) и, следовательно, с каждым из них можно сопоставить некоторое целое число – порядковый номер значения.

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

Порядковые типы

К порядковым типам относятся целые, логический, символьный, перечисляемый и тип-диапазон.

Целые типы. Диапазон возможных значений целых типов зависит от их внутреннего представления, которое может занимать один, два или четыре байта

ЦЕЛЫЕ ТИПЫ		
Название	Длина, байт	Диапазон значений
Byte	1	0…255
ShortInt	1	-128…+127
Word	2	0…65535
Integer	2	-32768…+32767
LongInt	4	-2 147 483 648…+2 147 483 647

При использовании процедур и функций с целочисленными параметрами следует руководствоваться «вложенностью» типов, т.е. везде, где может использоваться Word, допускается использование Byte (но не наоборот), в LongInt “входит” Integer, который, в свою очередь, включает в себя ShortInt.

При действии с целыми числами тип результата будет соответствовать типу операндов, если операнды относятся к различным целым типам,- типу того операнда, который имеет максимальную мощность (максимальный диапазон значений).

Логический тип. Значениями логического типа может быть одна из предварительно объявленных констант FALSE (ложь) или TRUE (истина).

Поскольку логический тип относится к порядковым типам, его можно использовать в операторе счётного типа.

ЛОГИЧЕСКИЕ ТИПЫ		
Название	Длина, байт	Значение
Boolean	1	False, True
Bytebool	1	False, True
Wordbool	2	False, True
Longbool	4	False, True

Символьный тип. CHAR – занимает 1 байт. Значением символьного типа является множество всех символов ПК. Каждому символу присваивается целое число в диапазоне 0…255. Это число служит кодом внутреннего представления символа.

Перечисляемый тип. Перечисляемый тип задаётся перечислением тех значений, которые он может получать. Каждое значение именуется некоторым идентификатором и располагается в списке, обрамлённом круглыми скобками, например,

```
Type
    Colors = (red, white, blue);
```

Применение перечисляемых типов делает программы нагляднее.

Соответствие между значениями перечисляемого типа и порядковыми номерами этих значений устанавливается порядком перечисления: первое значение списке получает порядковый номер 0, второе – 1 и т.д. максимальная мощность перечисляемого типа составляет 65536 значений, поэтому фактически перечисляемый тип задаёт некоторое подмножество целого типа WORD и может рассматриваться как компактное объявление сразу группы целочисленных констант со значениями 0,1 и т.д.

Использование перечисляемых типов повышает надёжность программы, благодаря возможности контроля тех значений, которые получают соответствующие переменные.

Тип-диапазон. Тип-диапазон есть подмножество своего базового типа, в качестве которого может выступать любой порядковый тип, кроме типа-диапазона.

Тип-диапазон задаётся границами своих значений внутри базового типа:

<мин.знач.>..<макс.знач.>, где

<мин.знач.> - минимальное значение типа-диапазона, <макс.знач.> - максимальное значение.

Например,

```
Type
    Digit = ‘0’ ..’9’;
    Dig2 = 48 .. 57;
```

При определении типа-диапазона нужно руководствоваться следующими правилами:

1. два символа «..» рассматриваются как один символ, поэтому между ними недопустимы пробелы.
2. левая граница диапазона не должна превышать его правую границу.

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

Вещественные типы

В отличие от порядковых типов, значения которых всегда сопоставляются с рядом целых чисел и, следовательно, представляется в ПК абсолютно точно, значения вещественных типов определяют произвольное число лишь с некоторой конечной точностью, зависящей от внутреннего формата вещественного числа.

Длина, байт	Название	Количество значащих цифр	Диапазон десятичного порядка
6	Real	11…12	-39…+38
4	Single	7…8	-45…+38
8	Double	15…16	-324…+308
10	Extended	19…20	-4951…+4932
8	comp	19…20	$-2*10^{63}+1…+2*10^{63}-1$

СТРУКТУРИРОВАННЫЕ ТИПЫ

Любой из структурированных типов характеризуется множественностью образующих этот тип элементов, т.е. переменная или константа структурированного типа всегда имеет несколько компонентов. Каждый компонент, в свою очередь, может принадлежать структурированному типу, что позволяет говорить о возможной вложености типов. В Паскале допускается произвольная глубина вложености типов, однако суммарная длина любого из них во внутреннем представлении не должна превышать 65520 байт.

Массивы в Паскале во многом схожи с аналогичными типами данных в других языках программирования. Отличительная особенность массивов заключается в том, что все их компоненты суть данные одного типа (возможно структурированного). Эти компоненты можно легко упорядочить и обеспечить доступ к любому из них простым указанием порядкового номера.

Запись – это структура данных, состоящая из фиксированного числа компонентов, называемых полями записи. В отличие от массива, компоненты (поля) записи могут быть различного типа. Чтобы можно было ссылаться на тот или иной компонент записи, поля именуются.

Множества – это набор однотипных логических связанных друг с другом объектов. Характер связей между объектами лишь подразумевается программистом и никак не контролируется Паскалем. Количество элементов, входящих в множество, может меняться в пределах от 0до 256 (множество, не содержащее элементов, называется пустым).именно непостоянством количества своих элементов множества отличаются от массивов и записей.

Для задания множества используется так называемый конструктор множества: список спецификаций элементов множества, отделяемых друг от друга запятыми; список обрамляется квадратными скобками. Спецификациями элементов могут быть константы или выражения базового типа, а также – тип-диапазон того же базового типа.

Строки. Тип STRING (строка) в Турбо Паскале широко используется для обработки текстов. Он во многом похож на одномерный массив символов ARRAY [0..N] OF CHAR, однако, в отличие от последнего, количество символов в строке – переменной может меняться от 0 до N, где N – максимальное количество символов в строке. Значение N определяется объявлением типа

STRING[N] N и может быть любой константой порядкового типа, но не больше 255. Турбо Паскаль разрешает не указывать N, в том случае длина строки принимается максимально возможной, а именно N=255.

Файлы. Под файлом понимается либо именованная область внешней памяти ПК, либо логическое устройство – потенциальный источник или приёмник информации.

Любой файл имеет три характерные особенности. Во-первых, у него есть имя, что даёт возможность программе работать одновременно с несколькими файлами. Во-вторых, он содержит компоненты одного типа. Типом компонентов может быть любой тип Турбо Паскаля, кроме файлов. В-третьих, длина вновь создаваемого файла никак не оговаривается при его объявлении и ограничивается только ёмкостью устройств внешней памяти.

Процедурные типы. Основное назначение этих типов — дать программисту гибкие средства передачи функций и процедур в качестве фактических параметров обращения к другим процедурам и функциям.

Переменным процедурных типов допускается присваивать в качестве значений имена соответствующих подпрограмм. После такого присваивания имя переменной становится синонимом имени подпрограммы.

Объект — это структура, состоящая из фиксированного числа компонент. Каждая компонента — это или поле, которое содержит данные определенного типа, или метод, который производит операции над объектом. Тип объект может наследовать компоненты от другого типа объекта.

Указатель (ссылочный тип) определяет множество значений, которые указывают на динамические переменные определенного типа, называемого базовым типом. Переменная с типом указатель содержит адрес динамической переменной в памяти.

Если базовый тип является еще не описанным идентификатором, то он должен быть описан в той же самой части описания типов, что и тип указатель.

Переменные и константы

Константа - это величина, которая в ходе выполнения программы принимает одно значение. Ее значение устанавливается еще до того, как программа начнет выполняться, а в ходе ее запуска сохраняет свое значение неизменной на всем протяжении работы программы.

В качестве констант могут использоваться целые, вещественные, шестнадцатеричные числа, логические константы, символы, строки символов, множества:

а) целые числа записываются без дробной части со знаком или без него;

б) вещественные числа записываются со знаком или без него, с фиксированной или плавающей точкой (например, +3.14 или -19e-5);

в) логическая константа - либо false либо true (ложь или истина);

г) символьная константа - любой символ, заключенный в апострофы (например, '<>', 'Y');

д) строковая константа - любая последовательность символов, заключенная в апострофы (например, 'это моя строка', ' '").

В Паскале существуют типизированные константы, представляющие собой переменные простых или составных типов (кроме файлов) с начальным значением.

Переменными называются параметры программы, которые могут менять свое значение в процессе ее выполнения. Все без исключения переменные должны быть описаны в разделе программы, начинающемся со слова **VAR**. Затем следуют конструкции вида:

список идентификаторов переменных**:** тип1;

список идентификаторов переменных**:** тип2;

В списке имена переменных перечисляются через запятую. Кроме базовых типов Турбо Паскаля здесь можно использовать свои типы (описанные ранее в разделе **Type**).

Арифметические операции, функции, выражения

В Паскале определены следующие операции:

- Унарные: not (отрицание);
- Мультипликативные: * (умножение), / (деление), div (деление нацело), mod (остаток от целочисленного деления), and (логическое “И”);
- Аддитивные: + (сложение), - (вычитание), or (логическое “ИЛИ”);
- Отношения: = (равно), <> (не равно), < (меньше), > (больше), <= (меньше или равно), >= (больше или равно).

Примечание: в Паскале нет операции возведения в степень. Вместо этого для возведения в целую степень можно использовать операцию умножения, а в дробную – выражение, использующее операцию логарифмирования и потенцирования: xy=ey*lnx, что на языке Паскаль соответствует выражению exp(y*ln(x)) с использованием двух стандартных функций exp(<выражение>) и ln(<выражение>).

Стандартные функции Паскаля

Для вычисления наиболее распространенных математических функций в Турбо-Паскале предусмотрены следующие стандартные функции:

Обращение	Функция
Pi	Число π
Sin(x)	Синус x (x в радианах)

Cos(x)	Косинус x (x в радианах)
Arctan(x)	Арктангенс x (x в радианах)
Exp(x)	e^x – экспонента
Ln(x)	Натуральный логарифм x
Sqr(x)	Квадрат числа x
Sqrt(x)	Квадратный корень из числа x
Abs(x)	Абсолютная величина числа x
Trunc(x)	Ближайшее целое, не превышающее x по модулю
Frac(x)	Дробная часть числа x
Int(x)	Целая часть числа x
Round(x)	Округление числа до ближайшего целого
Succ(x)	Определение следующего по порядку элемента из списка
Pred(x)	Определение предыдущего элемента из списка
Random	Псевдослучайное число в интервале [0; 1)
Random(x)	Псевдослучайное число в интервале [0; x)

.

Выражение - это единица языка, которая определяет способ вычисления некоторого значения. Выражения формируются из констант, переменных, функций, знаков операций и круглых скобок по определенным синтаксическим правилам.

В Паскале имеется большое количество встроенных функций для работы с данными каждого типа. Имена (указатели) этих функций с аргументом в круглых скобках могут также встречаться в выражениях.

Круглые скобки используются для изменения порядка вычисления частей выражения. Выражения без скобок вычисляются в порядке, соответствующем приоритету операций. Приоритеты расставлены таким образом:

- вычисления в круглых скобках;
- вычисление значений функций;
- унарные операции (**not**,+,-);
- операции типа умножения (*,/,**div**,**mod**,**and**);
- операции типа сложения (+,-, **or**, **xor**);
- операции отношения (=, <>, <, >, <=, >=).

В логическом выражении 2<=4 **and** 5>3 Паскаль выдаст ошибку, поскольку операция **and** будет выполнена раньше операций сравнения. Верная запись - (2<=4) **and** (5>3).

Структура программы

Программа, написанная на Паскале, состоит из заголовка и тела (блока), в конце которого следует точка – признак конца программы. В свою очередь, блок содержит разделы описаний и раздел операторов:

Program <имя программы>; { Заголовок программы }

Uses … ; { Подключение модулей }

Label ... ; { Раздел объявления меток }

Const ... ; { Раздел объявления констант }

Type ... ; { Раздел объявления новых типов }

Var ... ; { Раздел объявления переменных }

Procedure ... ; { Описание своих процедур }

Function ... ; { Описание своих функций }

Begin { начало основной программы }

<раздел операторов>

End.

Заголовок программы, начинается с зарезервированного слова program и имени, состоящего из букв латинского алфавита и цифр. Турбо-Паскаль позволяет опускать объявление program, поэтому данная строка является необязательной. Однако использование слова program является признаком хорошего стиля программирования.

Раздел описаний содержит описание каждого элемента информации, над которыми эти действия будут производиться. Эта часть должна предшествовать исполнительной части.

1. В разделе подключения модулей объявляются имена всех внешних модулей, ресурсы которых используются в программе.

2. Определение меток, начинающееся с зарезервированного слова Label и перечисления через запятую используемых в программе меток, представляющих собой любые цифры от 1 до 9999 или символьные имена.

3. Раздел объявления констант начинается с зарезервированного слова Const и содержит перечисления используемых в программе констант с присвоенными им именами, знаками = и их значениями и отделяемыми друг от друга точкой с запятой. Типизированные константы также объявляются после слова const, но имена через двоеточие связываются с типом, и лишь потом ставится знак равенства, и указываются значения.

4. Объявление новых типов начинается с зарезервированного слова Type и состоит из перечисления через точку с запятой имен типов, вводимых пользователем, с пояснением через знак равенства (=), от каких базовых типов и как они происходят.

5. Раздел описания переменных начинается с зарезервированного слова Var и содержит перечисления используемых в программе переменных с указанием их типа. При этом имена пе-

ременных одного и того же типа, могут перечисляться через запятую, с указанием в конце записи после двоеточия их типа, а имена переменных разных типов описываются отдельно и отделяются друг от друга точкой с запятой. В языке Турбо-Паскаль все используемые в программе переменные должны быть объявлены.

6. Раздел описания процедур и функций, начинается с зарезервированного слова Procedure и имени процедуры или с зарезервированного слова Function и имени функции. В раздел описания каждой процедуры и функции могут входить все перечисленные выше разделы, а также раздел операторов, составляющий непосредственно сами процедуры и функции.

Раздел операторов, начинающийся со слова begin и заканчивающийся словом end, включает в себя непосредственно весь процесс вычислений. Зарезервированные слова begin и end отмечают начало и конец программы.

Порядок разделов описаний может быть любым. Разделы друг от друга отделяются точкой с запятой. Раздел операторов является основным в программе. Все другие разделы, кроме раздела операторов могут отсутствовать. Операторы друг от друга отделяются точкой с запятой. В конце раздела операторов после ключевого слова end обязательно ставится точка, которая является признаком конца записи программы.

При записи программы на каждой строке можно писать либо по одному, либо по несколько операторов. Запись оператора можно начинать в любом месте строки. Ее можно продолжить на следующую строку, не разрывая имен констант, переменных, типов и символов. Кроме того, можно использовать пустые строки, чтобы отделить одну часть программы от другой.

Тема 2.2 Операторы языка

Оператор – это описание действий, которые будут выполнены при реализации алгоритма. В Паскале существуют 2 вида операторов: простые и структурные.

Простые операторы: оператор присваивания, оператор перехода, процедуры и функции, пустой оператор. Структурные операторы: составной оператор, оператор условия, оператор варианта, оператор цикла.

Пустой оператор

В программе может применяться пустой оператор, не выполняющий никакого действия, представляющий собой точку с запятой. Он может потребоваться для осуществления на него безусловного перехода или для более наглядного просмотра программы.

```
Goto M1;
. . . .
M1:
. . . .
```

Оператор присваивания

Оператор присваивания используется для задания значения переменных и имеет следующий синтаксис:

имя_переменной**:=** выражение;

Вычисляется выражение, стоящее в правой части оператора, после чего его значение записывается в переменную, имя которой стоит слева. Тип выражения и тип переменной должны быть совместимы, т.е. множество допустимых значений для типа выражения содержится во множестве допустимых значений для типа переменной.

Составной оператор

Составным оператором считается последовательность произвольных операторов, заключенная в операторные скобки - зарезервированные слова **begin** ... **end**. Допускается произвольная глубина вложенности составных операторов. Составной оператор применяется там, где по синтаксическим правилам языка может стоять только один оператор, а нам надо выполнить несколько действий. В этом случае набор необходимых команд должен быть оформлен как составной оператор. По сути, все тело программы представляет собой один составной оператор.

Простейший ввод и вывод

Рассмотрим простейшие процедуры ввода и вывода. По умолчанию ввод осуществляется с клавиатуры, а вывод на экран. К операторам ввода относятся:

Read(<список переменных через запятую>);

Readln(<список переменных>);

Readln;

Второй отличается от первого тем, что после ввода переводит курсор на новую строку, точнее, в конце своей работы считывает с клавиатуры код клавиши <**Enter**>. Третий оператор используется для организации паузы - выполнение программы продолжится, как правило, только после нажатия на клавиатуре клавиши <**Enter**>.

К операторам вывода относятся:

Write(<список вывода>);

Writeln(<список вывода>);

Writeln;

В списке вывода кроме имен переменных можно писать строковые константы (последовательность символов в апострофах) и даже выражения (выводятся их значения). Второй оператор отличается от первого тем, что после вывода переводит курсор на новую строку. Третий оператор просто переводит курсор на новую строку.

Форматированный вывод

В процедурах вывода *Write* и *WriteLn* имеется возможность записи выражения, определяющего ширину поля вывода. В приведенных ниже форматах используются следующие обозначения:

I, p, q – целочисленное выражение;

R –выражение вещественного типа;

B – выражение булевского типа;

Ch –выражение символьного типа;

S –выражение строкового типа;

-цифра;

* - знак “+” или “-“;

_-пробел.

I-выводится десятичное представление величины I, начиная с позиции расположения курсора.

Значение I	*Выражение*	*Результат*
134	Write (I);	134
287	Write (I,I,I);	287287287

I:p -выводится десятичное представление величины I в крайние правые позиции поля шириной p.

Значение I	*Выражение*	*Результат*
134	Write (I: 6);	_ _ _134
1	Write (I: 1);	_ _ _ _ _ _ _ _ _ 1
312	Write (I+I:7)	_ _ _ _624

R- в поле шириной 18 символов выводится десятичное представление величины R в формате с плавающей точкой. Если R>=0.0, используется формат _#.##########E*##.

Если R<0.0, формат имеет вид: -#.##########E*##.

Значение R	*Выражение*	*Результат*
715.432	Write (R);	_ _ 7.1543200000E+02

-1.919E+01 Write (R); _ -1.9190000000E+01

567.986 Write (R/2); _ _ 2.8399300000E+02

R:p –в крайние правые позиции поля шириной p символов выводится десятичное представление значения R в формате с плавающей точкой. Если R>=0.0, используется формат _ _…_##..#E*##, причем минимальная длина поля вывода составляет 7 символов. Если R<0.0, формат имеет вид:

_ _ …_--#.##..#E*##. Минимальная длина поля вывода 8 символов. После десятичной точки выводится, по крайней мере, одна цифра.

Значение R *Выражение* *Результат* 511.04

Write (R:15); 5.110400000E+02

-511.04 Write (R:15); -5.11040000E+02

46.78 Write (-R:12); -4.67800E+01

R:p:q –в крайние правые позиции поля шириной p символов выводится десятичное представление значения R в формате с фиксированной точкой, причем после десятичной точки выводится q цифр (0<=q<=24), представляющих дробную часть числа. Если q=0, ни дробная часть, ни десятичная точка не выводится. Если q>24, то при выводе используется формат с плавающей точкой.

Значение R	*Выражение*	*Результат*
511.04	Write (R:8:4);	511.0400
-46.78	Write (R:15);	_-46.78
-46.78	Write (R:9:4);	_-46.7800

Ch-начиная с позиции курсора выводится значение Ch.

Значение Ch	*Выражение*	*Результат*
’X’	Write (Ch);	X
‘Y’	Write (Ch);	Y
’!’	Write (Ch, Ch, Ch);	!!!

Ch:p-в крайнюю правую позицию поля шириной p выводится значение Ch.

Значение Ch	*Выражение*	*Результат*
’X’	Write (Ch:3);	_ _X
‘Y’	Write (Ch:5);	_ _ _ _Y
’!’	Write (Ch:2, Ch:4);	_!_ _ _!

S- начиная с позиции курсора, выводится значение S (строка или массив символов, если его длина соответствует длине строки).

Значение S *Выражение* *Результат*

’Day N’ Write (S); Day N ‘Ведомость 11’

Write (S); Ведомость 11

’RRRDDD’ Write (S, S); RRRDDDRRRDDD

S:p- значение S выводится в крайние правые позиции поля шириной p символов.

Значение S *Выражение* *Результат*

’Day N’ Write (S:10); _ _ _ _ _ Day N ‘Ведомость 11’

Write (S:13); _Ведомость 11

’RRRDDD’ Write (S:7, S:7); _RRRDDD_RRRDDD

B- выводится результат выражения B True или False, начиная с текущей позиции курсора.

Значение B	*Выражение*	*Результат*
True	Write (B);	True
False	Write (B, not B);	False True

B:p- в крайние правые позиции поля шириной p символов выводится результат булевского выражения B True или False.

Значение B	*Выражение*	*Результат*
True	Write (B:6);	_ _ True
False	*Write (B:10);*	*_ _ _ _ _False*
True	Write (B:5,not B:7);	_True_ _False

Оператор записи WriteLn аналогичен процедуре Write, но после вывода последнего в списке значения для текущей процедуры WriteLn происходит перевод курсора к началу следующей строки.

Процедура WriteLn, записанная без параметров, вызывает перевод строки.

Пример программы с использованием процедур ввода-вывода данных с различными форматами выводимых данных

```
Program Demo;
Var
    A,B,S:Integer;
Begin
    Writeln(‘Введите сторону A = ’);
    Readln(A);
    Writeln(‘Введите сторону B = ’);
    Readln(B);
    S:=A*B;
    Writeln(‘---------------------------------------------‘);
    Writeln(‘ | Сторона A |  | Сторона B |   | Площадь | ‘);
    Writeln(‘---------------------------------------------‘);
    Writeln(‘|‘,A:7,B:11,S:11, ‘|‘ :5);
    Writeln(‘---------------------------------------------‘);
End.
```

В результате работы данной программы на экране будет изображена следующая таблица:

```
-------------------------------------------------
| Сторона A |  | Сторона B |   | Площадь |
-------------------------------------------------
|        8                4                 32      |
-------------------------------------------------
```

каждая строка которой будет печататься с первой позиции новой строки экрана.

РАЗВЕТВЛЯЮЩИЕСЯ АЛГОРИТМЫ

В Паскале имеется возможность нелинейного хода программы, т.е. выполнения операторов не в том порядке, в котором они записаны. Такую возможность нам предоставляют разветвляющиеся алгоритмы. Они могут быть реализованы одним из трех способов: с использованием операторов перехода, условного оператора или оператора выбора.

Оператор перехода

Оператор перехода имеет вид:

GOTO <метка>.

Он позволяет передать управление непосредственно на нужный оператор программы. Перед этим оператором должна располагаться метка, отделенная от него двоеточием. В Паскале в качестве меток выступают либо целые числа от 0 до 9999, либо идентификаторы. Все метки должны быть описаны в разделе объявления меток следующим образом:

label <список меток через запятую> ;

Каждой меткой в программе может быть помечен только один оператор. Операторов перехода с одной и той же меткой можно писать любое количество. Необходимо, чтобы раздел описания метки, сама метка и оператор перехода с ее использованием располагались в пределах одного блока программы. Кроме того, нельзя передавать управление внутрь структурированных операторов (например, if, for, while, repeat и др.).

Условный оператор

Условный оператор **IF** позволяет изменить порядок выполнения команд в зависимости от некоторого логического условия, т.е. он осуществляет ветвление вычислительного процесса. Условный оператор имеет вид:

IF <условие> **THEN** <оператор1> [**ELSE** <оператор2>];

В случае истинности логического выражения, стоящего в условии, выполняется <оператор1>, а <оператор2> пропускается. При ложном значении логического выражения пропускается <оператор1> и выполняется <оператор2>.

Оператор **IF** может быть полным (присутствуют обе ветви) или неполным (**Else**-ветви нет, при ложном условии ничего не делается). По правилам каждая из ветвей может содержать либо один выполняемый оператор, либо несколько, объединенных в составной. Точка с запятой перед **Else** считается ошибкой.

Пример. *Ввести целое число. Вывести соответствующий ему символ ASCII-таблицы, либо сообщить, что такого символа нет (0-31 - управляющие коды, затем до 256 - печатаемые символы).*

```
program ascii_symbol;
var     i:word;
begin
   write('Введите целое число: ');  readln(i);
   if (i>31) and (i<256) then
      writeln('Соответствующий символ - ', Chr(i))
      else writeln('Такого символа нет');
   readln
end.
```

Оператор выбора

Оператор выбора является обобщением условного оператора: он дает возможность выполнить один из нескольких операторов в зависимости от значения некоторого выражения, называемого селектором.

```
CASE <селектор> OF
  <список меток 1> : <оператор 1>;
  <список меток 2> : <оператор 2>;
  . . . . . . . . . . . . . . .;
  <список меток N> : <оператор N>;
  ELSE <оператор>
END;
```

селектор - выражение любого перечисляемого типа, кроме вещественного;

оператор - любой оператор языка, в том числе и составной;

список меток - список разделенных запятыми значений выражения 'селектор' или одно его значение; тип метки и селектора должен быть одинаков;

Оператор варианта выбирает для исполнения тот ОПЕРАТОР, одна из меток которого равна текущему значению выражения СЕЛЕКТОР. Если ни одна из меток не равна текущему значению селектора, то никакие операторы не выполняются, либо выполняются операторы, следующие за зарезервированным словом ELSE (если такое имеется).

Ветвь **Else** не обязательна, и в отличие от оператора **if**, перед ней ставится точка с запятой. Если для нескольких значений <селектора> действия совпадают, то эти константы можно перечислить через запятую перед двоеточием или задать диапазон значений (нижняя граница .. верхняя граница).

***Пример**: Вводится целое число, если это цифра, то определить четная она или нет, а если число, то определить попадает ли оно в диапазон от 10 до 100, если нет, то выдать соответствующее сообщение.*

```
program chislo;
var     i:integer;
begin
   write('Введите целое число: ');
   readln(i);
   case i of
      0,2,4,6,8 : writeln('Четная цифра');
      1,3,5,7,9 : writeln('Нечетная цифра');
      10...100,200 : writeln('Число от 10 до 100 или 200');
      else writeln('Число либо отрицательное, либо > 100, но не 200');
   end;
   readln
end.
```

ЦИКЛИЧЕСКИЕ АЛГОРИТМЫ

Последовательность команд, выполняющихся несколько раз в зависимости от некоторого условия, называется циклом.

Цикл с параметром

Если заранее известно число повторений цикла, то в программе используются циклы с параметром. Оператор цикла **For** организует выполнение одного оператора заранее определенное число раз. Его еще называют цикл со счетчиком. В общем виде цикл с параметром задается следующим образом:

FOR <параметр> := <nz> **TO** <kz> **DO** <оператор>

Здесь параметр цикла (счетчик) представляет собой переменную порядкового типа; <nz> и <kz> - выражения, определяющие начальное и конечное значение счетчика; <оператор> - один (возможно составной) оператор, который называют телом цикла, повторяемый определенное число раз.

На первом шаге цикла параметр принимает значение nz. После каждого выполнения тела цикла, если параметр цикла не равен kz, происходит увеличение параметра на единицу. Если nz > kz, то <тело цикла> не будет выполнено ни разу и выполнение цикла с параметром сразу же закончится.

Возможна другая форма цикла с параметром.

FOR <параметр> := < kz > **DOWNTO** < nz > **DO** <оператор>,

которая выполняется аналогичным образом, но значение <параметра> изменяется с шагом, равным -1.

***Рекомендации*:** Использовать цикл **for** при заранее известном количестве повторений. Не изменять параметр в теле цикла. При использовании кратных (вложенных) циклов применять разные переменные в качестве параметров. Определять до цикла значения всех используемых в нем переменных. Не ставить точку с запятой после **do**.

Примеры программ с использованием оператора for

Пример 1. Программа *DemoFor1* выводит на экран таблицу перевода из градусов по шкале Цельсия(C) в градусы по Фаренгейту(P) для значений от 15°C до 30°C с шагом 1 градус. Перевод осуществляется по формуле: F = C*1.8+32.

```
program DemoFor1;
var
I: integer;
F: real;
begin
  Writeln ('    Температура    ') ;
   for I:= 15 to 30 do {Заголовок цикла с параметром}
    begin            {Начало тела цикла}
    F:= I*1.8+32;
    Writeln('по Цельсию=  ',I,' по Фаренгейту= ', F:5:2)
    end;             {Конец тела цикла}
end.
```

В блоке описания переменных описаны параметр цикла *I* типа *integer* и переменная *F* — температура по Фаренгейту типа *real*. Переменная *I,* помимо функций управляющей переменной, является переменной, хранящей целочисленные значения температуры по шкале Цельсия. В начале выполнения программы на экран выводится надпись *' Температура '*, а затем оператором повтора выводится таблица соотношения температуры в шкалах Цельсия и Фаренгейта. Печать таблицы выполняется оператором *Writeln('По Цельсию= ',I,' по Фаренгейту= ' , F: 5:2).*

Цикл выполняется следующим образом.

При первом обращении к оператору *for* вычисляются значения начального *(15)* конечного *(30)* параметров цикла, и управляющей переменной *I* присваивается начальное значение *15*.

Затем циклически выполняется следующее:

1. Проверяется условие *I<=30.*

2. Если оно соблюдается, то выполняется составной оператор в теле цикла, т.е. рассчитывается значение выражения *I* 1.8+32*, затем оно присваивается переменной *F,* и на экран выводится сообщение: 'По Цельсию= ', I, ' по Фаренгейту= ', F:5:2.

Если условие *I<=30* не соблюдается, т. е. как только *I* станет *> 30,* оператор тела цикла не выполняется, а управление в программе передается за пределы оператора *for*, в нашем примере на оператор *end*. Программа завершает работу.

3. Значение параметра цикла *I* увеличивается на единицу, и управление передается в заголовок цикла *for* для проверки условия.

Далее цикл повторяется, начиная с пункта 1.

Пример 2. Программа *DemoFor2*, которая печатает на экране символы американскою стандартного кода обмена информацией *(ASCII)* в порядке *убывания* кода.

```
program DemoFor2;
var
A: integer;
begin
     for A:= 255 downto 0 do   {Цикл с убыванием параметра}
    Writeln('код символа = ',A, ' символ == ',Chr(A));
end.
```

В данной программе применяется *цикл for с убыванием значения управляющей переменной* A (используется указание *downto* - убывание).

Пример 3*: Вводятся 10 чисел, посчитать среди них количество положительных.*

```
program cycle_for1;
var     i,kn:byte;       x:real;
begin
   kn:=0;
   for i:=1 to 10 do
   begin
      writeln('Введите ',i,' число: ');
```

```
        readln(x);
        if x>0 then kn:=kn+1 {увеличиваем количество на 1}
    end;
    writeln('Вы ввели ',kn,' положительных чисел.');
    readln
end.
```

***Пример 4**: Напечатать буквы от 'Z' до 'A'.*

```
program cycle_for2;
var     c:char;
begin
    for c:='Z' downto 'A' do write(c);
    readln
end
```

***Пример 5**: Вычислить N-е число Фиббоначчи. Числа Фиббоначчи строятся следующим образом: F(0)=F(1)=1; F(i+1)=F(i)+F(i-1); для i>=1. Это пример вычислений по рекуррентным формулам.*

```
program Fib;
var     a,b,c:word;     i,n:byte;
begin
    write('введите номер числа Фиббоначчи ');
    readln(N);
    a:=1; {a=F(0), a соответствует F(i-2)}
    b:=1; {b=F(1), b соответствует F(i-1)}
    for i:=2 to N do
    begin
        c:=a+b; {c соответствует F(i)}
        a:=b; b:=c; {в качестве a и b берется следующая пара чисел}
    end;
    writeln(N,'-е число Фиббоначчи =',b); {для N>=2 b=c}
    readln
end.
```

Циклы с условием

Если заранее неизвестно число повторений цикла, то используются циклы с условием. В Паскале имеется два типа таких циклов. Это циклы с предусловием и постусловием.

Цикл с предусловием

WHILE <логическое выражение> **DO** <оператор>;

Выполнение оператора цикла с предусловием начинается с проверки условия, записанного после слова while. Если оно соблюдается, то выполняется <тело цикла>, а затем вновь проверяется условие и т.д. Как только на очередном шаге окажется, что условие не соблюдается, то выполнение <тела цикла> прекратится.

Если <тело цикла> состоит из нескольких операторов, то они объединяются операторными скобками.

В теле цикла обязательно должен быть оператор, влияющий на соблюдение условия, в противном случае произойдет зацикливание.

Пример программы с использованием цикла while

Программа *DemoWhile* производит суммирование 10 произвольно введенных целых чисел.

```
program DemoWhile;
const
Limit =10;   {Ограничение на количество вводимых чисел}
var Count, Item, Sum: integer;
begin
  Count:=0;    {Счетчик чисел}
  Sum:= 0;      {Сумма чисел}
          while (Count < Limit) do  {Условие выполнения цикла}
                  begin
                       Count:= Count+1;
                Write('Введите ', Count, ' - е целое число: ');
                Readln(Item);{Ввод очередного числа с клавиатуры}
                       Sum:= Sum+Item;
                  end;
                Writeln('Сумма введенных чисел равна ', Sum) ;
end.
```

В данном примере в разделе описания констант описана константа *Limit=10,* задающая ограничение на количество вводимых чисел. В разделе описания переменных описаны переменные *Count, Item, Sum* целочисленного типа. В начале выполнения программы обнуляются значения счетчика введенных чисел *Count* и их суммы *Sum*. Затем выполняются цикл ввода 10 чисел и их суммирование. Вначале оператор *условия while* проверяет условие *Count < Limit*. Если условие верно, то выполняется составной оператор в теле цикла:

```
begin
  Count:= Count+1;
  Write('Введите ', Count, '-е целое число: ');
  Readln(Item) ;
  Sum:= Sum+Item;
End;
```

в котором вводится значение очередного числа, и на это значение увеличивается значение суммы. После этого управление в программе вновь передается оператору цикла *while*, опять проверяется условие *Count < Limit*. Если условие верно, то выполняется составной оператор и т. д., пока значение переменной *Count* будет меньше 10. Как только значение *Count* станет равно 10 и условие *Count < Limit* не будет соблюдено, выполнение цикла завершится, а управление в программе будет передано на оператор, находящийся за словом *end*, т. е. первый оператор за границей *while.* Это вызов процедуры *Writeln,* которая выведет сообщение *'Сумма введенных чисел равна'* и напечатает значение переменной *Sum.*

Цикл с постусловием

REPEAT
<оператор 1>
...
<оператор N>
UNTIL <логическое выражение>

Оператор **Repeat** организует повторяющееся выполнение нескольких операторов до тех пор пока не станет истинным условие, стоящее в **Until**-части. Тело цикла обязательно выполняется хотя бы один раз. Таким образом, в этом цикле логическое выражение - это условие выхода из цикла.

При создании циклических алгоритмов Турбо Паскаль позволяет использовать процедуры **Continue** и **Break**. Процедура **Continue** досрочно завершает очередной шаг цикла, передает управление на заголовок. Процедура **Break** реализует немедленный выход из цикла.

***Рекомендации*:** Для того чтобы избежать зацикливания программы необходимо обеспечить изменение на каждом шаге цикла значения хотя бы одной переменной, входящей в условие цикла. После выхода из цикла со сложным условием (с использованием операций **and**, **or**, **xor**) как правило, необходима проверка того, по какому условию цикл завершен.

***Пример 1**: Пары неотрицательных вещественных чисел вводятся с клавиатуры. Посчитать произведение для каждой пары и сумму всех чисел.*

```
program cycle_while;
var     x,y,sum:real;   otv:char;
begin
   sum:=0;
   otv='Д';
   while (otv='Д') or (otv='д') do
   begin
      write('Введите числа x,y > 0 ');
      readln(x,y);
      writeln('Их произведение = ',x*y:8:3);
      sum:=sum+x+y;
      write('Завершить программу (Д/Н)? ');
      readln(otv);
   end;
   writeln('Общая сумма = ',sum:8:3);
   readln
end.
```

***Пример 2**: В той же задаче можно использовать другой цикл с условием:*

```
program cycle_repeat;
var     x,y,sum:real;   otv:char;
begin
```

```
    sum:=0;
    repeat
        write('Введите числа x,y > 0 ');
        readln(x,y);
        writeln('Их произведение = ',x*y:8:3);
        sum:=sum+x+y;
        write('Завершить программу (Д/Н)? ');
        readln(otv);
    until (otv='Д') or (otv='д');
    writeln('Общая сумма = ',sum:8:3);
    readln
end.
```

Пример 3*: Нахождение наибольшего общего делителя двух целых чисел с помощью Алгоритма Эвклида.*

```
program Evklid;
var     a,b,c:integer;
begin
    write('введите два целых числа : ');
    readln(a,b);
    while b<>0 do
    begin
        c:=a mod b;
        a:=b;
        b:=c;
    end;
    writeln('наибольший общий делитель = ',a);
    readln
end.
```

Вложенные циклы

Если для решения задачи требуется организовать два цикла, один из которых помещается внутрь другого, то такие конструкции называют вложенными циклами.

Например, даны натуральные числа n и k. Составить программу вычисления выражения $1^k + 2^k + ... + n^k$.

Для вычисления указанной суммы целесообразно организовать цикл с параметром i, в котором, во-первых, вычислялось бы очередное значение $y = i^k$ и, во-вторых, осуществлялось бы накопление суммы прибавлением полученного слагаемого к сумме всех предшествующих (s = s + y).

```
Program Example_13;
 Var n, k, y, i, s, m: Integer;
Begin
 Writeln ('Введите исходные данные n и k');
 Readln(n,k);
 s:=0;
 For i:=1 To n Do
 Begin y:=1;
```

```
 For m:=1 To k Do y:=y*i; {нахождение степени k числа i}
 s:=s+y;
 End;
 Writeln('Ответ: ',s);
End.
```

Внутренний и внешний циклы могут быть любыми из трёх рассмотренных ранее видов: циклами с параметром, циклами с предусловием или циклами с постусловием. Правила организации как внешнего, так и внутреннего циклов такие же, как и для простых циклов каждого из этих видов. Но при использовании вложенных циклов необходимо соблюдать следующее условие: внутренний цикл должен полностью укладываться в циклическую часть внешнего цикла.

Тема 2.3 Массивы

Рассмотренные ранее *простые типы данных* определяют различные множества атомарных (неразделимых) значений. В отличие от них *структурные типы данных* задают множества сложных значений, каждое из которых образует совокупность нескольких значений другого типа. В структурных типах выделяют регулярный тип (массивы).

С понятием "массив" приходится сталкиваться при решении научно-технических и экономических задач обработки совокупностей большого количества значений. В общем случае ***массив*** - это структурированный тип данных, состоящий из фиксированного числа элементов, имеющих один и тот же тип.

Название *регулярный тип* (или ряды) массивы получили за то, что в них объединены однотипные (логически однородные) элементы, упорядоченные по индексам, определяющим положение каждого элемента в массиве.

В качестве элементов массива можно использовать и любой другой ранее описанный тип, поэтому вполне правомерно существование массивов записей, массивов указателей, массивов строк, массивов и т.д. Элементами массива могут быть данные любого типа, включая структурированные. Тип элементов массива называется *базовым*. Особенностью языка Паскаль является то, что *число элементов массива фиксируется* при описании и в процессе выполнения программы не меняется.

Элементы, образующие массив, упорядочены таким образом, что каждому элементу соответствует совокупность номеров (индексов), определяющих его местоположение в общей последовательности. *Доступ* к каждому отдельному элементу осуществляется путем индексирования элементов массива. *Индексы* представляют собой выражения любого скалярного типа, кроме вещественного. Тип индекса определяет границы изменения значений индекса. Для описания массива предназначено словосочетание: *array of* .

Формат записи массивов:

```
Type
        <имя типа> = array [тип индекса] of  <тип компонента>;
Var
        <идентификатор> : <имя типа>;
```

Массив может быть описан и без представления типа в разделе описания типов данных:

```
Var   <идентификатор> : array [тип индекса] of  <тип компонента>;
```

Примеры описания одномерных массивов

Если в описании массива задан один индекс, массив называется *одномерным*. Одномерный массив соответствует понятию линейной таблицы (вектора).

Одномерные массивы описываются следующим образом:

```
Type
```

```
Klass = (К1, К2, К3, К4) ;
Znak = array [1..255] of char;
Var
Ml: Znak;        {Тип Znak предварительно описан в разделе типов}
M2: array[1..4] of Klass;
M3: array[1..60] of integer; {Прямое описание массива М3}
Mas: array[1..4] of integer;
```

В Паскале количество элементов массива всегда должно быть фиксировано, т. е. определяться при трансляции программы.

Действия над массивами

Для работы с массивом как единым целым используется идентификатор массива без указания индекса в квадратных скобках. Массив может участвовать только в операциях отношения *"равно", "не равно" и в операторе присваивания*. Массивы, участвующие в этих действиях, должны быть идентичны по структуре, т. е. иметь одинаковые типы индексов и одинаковые типы компонентов.

Например, если массивы А и В описаны как

Var A, B: array[1..20] of real,

то применение к ним допустимых операций даст следующий результат:

Выражение	*Результат*
A=B	*True*, если значение каждого элемента массива А равно соответствующему значению элемента массива В
A<>B	*True*, если хотя бы одно значение элемента массива А не равно значению соответствующего элемента массива В
A:=B	Все значения элементов массива В присваиваются соответствующим элементам массива А. Значения элементов массива В остаются неизменны.

Действия над элементами массива

После объявления массива каждый его элемент можно обработать, указав *идентификатор (имя) массива* и *индекс элемента* в квадратных скобках. Например, запись *Mas[2], Vector[10]* позволяет обратиться ко второму элементу массива *Mas* и десятому элементу массива *Vector*.

Индексированные элементы массива называются *индексированными переменными* и могут быть использованы так же, как и простые переменные. Например, они могут находиться в выражениях в качестве операндов, использоваться в операторах *for, while, repeat*, входить в качестве параметров в операторы *Read, Readin, Write, Writeln*; им можно присваивать любые значения, соответствующие их типу.

Рассмотрим типичные ситуации, возникающие при работе с данными типа *array*. Для этого опишем три массива и четыре вспомогательные переменные:

```
Var
   A,D    : array[l.,4] of real;
   B      : array[1..10,1..15] of integer;
   I, J, K : integer;
   S : real;
```

Инициализация (присваивание начальных значений) массива заключается в присваивании каждому элементу массива одного и того же значения, соответствующего базовому типу. Наиболее эффективно эта операция выполняется с помощью оператора for, например:

Инициализация элементов одномерного массива A:

```
for I := 1 to 4 do
A[I] := 0;
```

Ввод-вывод элементов массива

Паскаль не имеет средств ввода-вывода элементов массива сразу, поэтому ввод и вывод значений производится поэлементно. Значения элементам массива можно присвоить с помощью оператора присваивания, как показано в примере инициализации, однако чаще всего они вводятся с экрана с помощью оператора *Read* или *Readln* с использованием оператора организации *цикла for*:

Ввод элементов одномерного массива A:

```
for I:=l to 4 do
Readln(A[I]) ;
```

В связи с тем, что использовался оператор *Readln*, каждое значение будет вводиться с новой строки. Можно ввести и значения отдельных элементов, а не всего массива. Так, операторами:

```
Read(A[3]);
```

вводится значение третьего элемента массива A.

Вывод значений элементов массива выполняется аналогичным образом, но используются операторы *Write* или *Writeln*:

Вывод элементов одномерного массива A:

```
for I := I to 4 do
Writeln (A[I]);
```

Копированием массивов называется присваивание значений всех элементов одного массива всем соответствующим элементам другого массива. Копирование можно выполнить одним оператором присваивания, например *A:=D;* или с помощью оператора *for*:

```
for I := 1 to 4 do
A[I] := D[I];
```

В обоих случаях значение элементов массива D не изменяется, а значения элементов массива A становятся равными значениям соответствующих элементов массива D. Очевидно, что оба массива должны быть идентичны по структуре.

Иногда требуется осуществить *поиск в массиве* каких-либо элементов, удовлетворяющих некоторым известным условиям. Пусть, например, надо выяснить, сколько элементов массива A имеют нулевое значение.

```
K := 0;
for I := 1 to 4 do
  if A[I] = 0 then K := K + 1;
```

После выполнения цикла переменная *K* будет содержать количество элементов массива *A* с нулевым значением.

Перестановка значений элементов массива осуществляется с помощью дополнительной переменной того же типа, что и базовый тип массива.

Например, так запишется фрагмент программы, обменивающий значения первого и пятого элементов массива A:

```
Vs:= A[5];        {Vs - вспомогательная переменная}
A[5]:= A[1];
A[1]:= Vs;
```

Примеры решения типовых задач

1. *Сформировать и вывести на экран последовательность из n элементов одномерного массива, вводимых с клавиатуры.*

```
program primer1;
var
mas:array [1..100] of integer; n,i:integer;
BEGIN
writeln('Введите количество элементов массива не больше 100: ');
readln(n);
for i:=1 to n do   {ввод элементов массива}
   begin
      writeln('Введите  ',i,' элемент массива');
      readln(mas[i]);
   end;{конец ввода}
writeln('Введенный массив: ');
for i:=1 to n do   {вывод элементов массива}
   begin
      write(' ',mas[i],' ');
   end;          {конец вывода}
END.
```

2. ***Сформировать и вывести на экран последовательность из n элементов, заданных датчиком случайных чисел на интервале [-23, 34].***

```
Program posled;
Var a: array[1..100] of integer;
```

```
    i, n: integer;
Begin
  Write (‘Сколько элементов? ’);  Readln (n);
   For i=1 to n do
      begin
          a[i]:= Random(58)-23;
           writeln (a[i],’ ‘);
      end;
End.
```

3. Найти сумму элементов одномерного массива. Размер произвольный. Элементы вводятся с клавиатуры.

```
Program summa;
Var a: array[1..100] of real;
     i, n: integer;
      s: real;
Begin
  Write (‘n=’); Readln (n);
  s:=0;
  For i:=1 to n do
    begin
      write (‘введите число’); readln (a[i]);
      s:=s+a[i];
    end;
    writeln(‘сумма  элементов равна  ‘,s);
End.
```

4. Найти номер наименьшего элемента в массиве, заданного датчиком случайных чисел на интервале [-20, 25]. Размер произвольный.

```
Program numberminim;
Var a: array[1..100] of integer;
     i, n, num, min: integer;
Begin
  Write (‘n=’); Readln (n);
  For i:=1 to n do
   begin
       a[i]:= Random(46)-20;
       writeln (a[i]);
   end;
  min:=a[1];
  num:=1;
  For i:=2 to n do
      If a[i]< min then
          begin
             min:=a[i];
             num:=i;
          end;
  Writeln(‘ номер наименьшего элемента:   ‘,num);
End.
```

5. Найти произведение элементов целочисленного одномерного массива с четными номерами, состоящего из n элементов. Элементы вводятся с клавиатуры.

```
Program proizved_chet;
Var a: array [1..100] of integer;
   i, n, p: integer;
Begin
  p:=1;
 write ('n=');  readln (n);
  for i:=1 to n do
    begin
      write ('a[',i,']='); readln (a[i]);
      if i mod 2=0 then p:=p*a[i];
    end;
  Writeln ('произведение элементов массива с четными номерами равно ',p);
End.
```

6. Сортировка целочисленного массива в порядке возрастания

```
program z5_3;uses crt;
const n=10;
var x:array [1..n] of integer; i,j,h:integer;
BEGIN
clrscr; randomize;
writeln('исходный массив:');
for i:=1 to n do begin
  x[i]:=random(n+1); write(x[i]:3);
  end;
writeln; writeln('упорядоч. массив:');
for i:=2 to n do begin
  for j:=n downto i do begin
    if x[j-1]>x[j] then begin
      h:=x[j-1];
      x[j-1]:=x[j];
      x[j]:=h;
      end; end; end;
  for j:=1 to n do write(x[j]:3);
END.
```

Двумерные массивы

Двумерный массив можно представить как таблицу или матрицу. Для получения доступа к его элементам используются два индекса: номер строки и номер столбца. При описании в типе индексов надо указать диапазон для двух индексов массива.

Примеры описания двумерных массивов:

```
Type
        Vector = array[1..4] of integer;
                Massiv = array[1..4] of Vector;
Var
        Matrix : Massiv;
```

Ту же структуру можно получить, используя другую форму записи:

```
Var
        Matrix : array[1..4,1..4] of integer ;
```
Для описания массива можно использовать предварительно определенные константы:

```
Const
        G1 = 4; G2 = 6;
Var
        MasY: array[1..Gl, l..G2] of real;
```

Элементы массива располагаются в памяти последовательно. Элементы с меньшими значениями индекса хранятся в более низких адресах памяти. Многомерные массивы располагаются таким образом, что самый правый индекс возрастает самым первым.

Например, если имеется массив:

```
        A:array[1..5,1..5] of integer;
```

то в памяти элементы массива будут размещены по возрастанию адресов:

A[1,1] A[1,2] … A[1,5] A[2,1] A[2,2] … A[5,5]

При работе с *двумерным массивом* указываются два индекса, с n-мерным массивом - n индексов. Например, запись *Matr[4,4]* делает доступным для обработки значение элемента, находящегося в *четвертой строке четвертого столбца* массива *Matr*.

Процедуры обработки матриц

Сумма указанной строки:

```
S:=0; i:=3;
For J:=1 to m do S:=S+b[i,j];
```

Транспонирование квадратной матрицы:

```
For i:=1 to n-1 do
   For j:=1 to n do
      Begin
        P:= a[i,j];
        a[i,j]:=a[j,i];
        a[j,i]:=p;
       end;
```

Удаление строки из матрицы:

```
n:=n-1;
For i:=1 to n do
  For j:=1 to m do b[i,j]:=b[i+1,j];
```

Включение строки в матицу:

```
i:=n;
while i>=k do
  begin
    for j:=1 to m do b[i+1,j]:=b[i,j];
    i:=i-1;
  end;
for j:=1 to m do b[k,j]:=c[j];
```

```
n:=n+1;
```

Перестановка строк матрицы:

```
For k:=1 to k do c[k]:=a[i,k];
For k:=1 to k do a[i,k]:=a[j,k];
For k:=1 to k do a[j,k]:=c[k];
```

Поиск минимального элемента матрицы:

```
Min:= a[1,1]; k;=1; L:=1;
For i:=1 to n do
  For j:=1 to m do
    IF min>a[i,j] then begin min:=a[i,j]; k:=i; l:=j; end;
```

Сложение строк матрицы:

```
For j:=1 to m do
   a[k,j]:=a[k,j]+a[l,j]*b;
```

Примеры решения типовых задач

1. *Программа ввода-вывода двумерного массива*

```
program primer;
var
i,j,n,m:integer;
massiv:array[1..10,1..10] of integer; {описан двумерный массив с именем massiv}
BEGiN
write('Введите количество строк массива (не больше 10): ');
read(n);                                {n-количество строк}
write('Введите количество столбцов массива (не больше 10):');
read(m);                                {m-количество столбцов}
        for i:=1 to n do        {ввод элементов двумерного массива}
                begin
                   for j:=1 to m do
                     read(massiv[i,j]);
                end;                            {конец ввода}
        writeln('Введенный массив: ');
         for i:=1 to n do       {вывод элементов двумерного массива}
                begin
                   for j:=1 to m do
                      write(massiv[i,j]:5);
                end;                            {конец вывода}
END.
```

2. Сформировать с помощью датчика случайных чисел и вывести на экран матрицу, размером MxN. Элементы задаются на интервале [-20, 25].

```
Var a: array[1..50,1..50] of integer;
    i, j, n, m: integer;
Begin
Write('сколько строк?'); Readln(m);
Write('сколько столбцов?'); Readln(n);
For i:=1 to m do
  begin
     For j:=1 to n do
```

```
        begin
            a[i,j]:=int(rnd*46)-20;
             write(a[i,j],' ');
        end;
     writeln;
  end;
End.
```

3. В двумерном массиве, состоящем из целых чисел, найти наименьший элемент и номер строки, в которой он находится. Элементы вводятся с клавиатуры. Размер MXN.

```
Program minim;
Var a: array[1..50,1..50] of integer;
     i, j, m, n, min, K: integer;
Begin
 Write('сколько строк?'); Readln(m);
Write('сколько столбцов?'); Readln(n);
For i:=1 to m do
For j:=1 to n do
      begin       write('a[',i,',',j,']=');  readln (a[i,j]);  end;
min:=a[1,1]; K:=1;
  For i:=1 to m do
  For j:=1 to n do
    If a[i,j]< min then
       begin
          min:=a[i,j];    K:=i;
       end;
  Writeln('наименьшее число   ',min,'  находится в ', k , ' строке');
End.
```

Тема 2.4 Строки и множества

Строка — это последовательность символов кодовой таблицы персонального компьютера. При использовании в выражениях строка заключается в апострофы. Количество символов в строке (длина строки) может динамически изменяться от 0 до 255. Для определения данных строкового типа используется идентификатор *String*, за которым следует заключенное в квадратные скобки значение максимально допустимой длины строки данного типа. Если это значение не указывается, то по умолчанию длина строки равна 255 байт.

Переменную строкового типа можно определить через описание типа в разделе определения типов или непосредственно в разделе описания переменных. Строковые данные могут использоваться в программе также в качестве констант.

Недопустимо применение строковых переменных в качестве *селектора* в операторе *Case.*

Определение строкового типа устанавливает максимальное количество символов, которое может содержать строка.

Формат описания строкового типа

```
Type
        <имя типа> =String [максимальная длина строки];
Var
        <идентификатор, . . . > : <имя типа>;
```

Переменную типа *String* можно задать и без описания типа:

```
Var
        <идентификатор, . . . > : String [максимальная длина строки];
```

Пример описания строковых данных

```
Const
      Address = 'ул. Переверткина, 25'; {Строковая константа}
Type
      Flot = string[125];
Var
      Fstr : Flot;          {Описание с заданием типа)
      St1 : String;         {По умолчанию длина строки St1= 255}
      St2, St3 : string[50] ;
      Nazv   : string[280];   (Ошибка, длина строки Nazv превышает 255}
```

Для ввода и вывода переменной типа STRING используются операторы READLN и WRITELN.

Операции над строками

1. Операции сравнения. Сравнение происходит посимвольно слева направо: сравниваются коды соответствующих символов, пока не нарушится равенство. Две строки считаются равными, если они равны по длине и совпадают посимвольно.
2. Строки можно объединять с помощью операции сцепления. Знак операции - «+».

3. Индексирование. Так как строку можно рассматривать как массив символов, то при помощи индексирования можно организовать доступ к его отдельным символам по их номерам.

Стандартные строковые процедуры и функции

Delete (Str,Poz,N) — удаление *N* символов строки *Str*, начиная с позиции *Poz*. Если значение Poz > 255, возникает программное прерывание.

Например:

Значение Str	*Выражение*	*Результат*
'абвгде'	Delete(Str, 4, 2);	'абве'
'река Волга'	Delete(Str, 1, 5);	'Волга'

Insert (Strl, Str2, Poz) –вставка строки *Str1* в строку *Str2*, начиная с позиции *Poz*.

Например:

```
Var
  Sl, S2 : string[ll] ;
  …
  S1 := '  EC  ';
  S2 := 'ЭВМ1841';
  Insert(S1,S2,4) ;
```

В результате выполнения последнего выражения значение строки *S2* станет равным *'ЭВМ EC 1841'*.

Str (IBR,St) — преобразование числового значения величины *IBR* и помещение результата в строку *St*. После IBR может записываться формат, аналогичный формату вывода. Если в формате указано недостаточное для вывода количество разрядов, поле вывода расширяется автоматически до нужной длины.

Например:

Значение IBR	*Выражение*	*Результат*
1500	Str(IBR:6,St)	'_1500'
4.8E+03	Str(IBR:10,St)	'____4800'
76854	Str(-IBR:3,St)	'—76854'

Val (St,IBR,Code) — преобразует значение *St* в величину целочисленного или вещественного типа и помещает результат в *IBR*. Значение *St* не должно содержать незначащих пробелов в

начале и в конце. *Code* — целочисленная переменная. Если во время операции преобразования ошибки не обнаружено, значение *Code* равно нулю, если ошибка обнаружена (например, литерное значение переводится в цифровое), *Code* будет содержать номер позиции первого ошибочного символа, а значение *IBR* не определено.

Например:

Значение St	*Выражение*	*Результат*
'1450'	Val(St,IBR,Cod)	Code=0
'14.2E+02'	Val(St,lBR,Cod)	Code=0
'14.2A+02	Val(St,IBR,Cod)	Code=5

Copy (St,Poz,N) — выделяет из строки *St* подстроку длиной *N* символов, начиная с позиции *Poz.* Если *Poz > Length(St),* то результатом будет пробел; если *Poz > 255*, возникнет ошибка при выполнении. Функция *Length* описана ниже. *Poz, N* — целочисленные выражения.

Например:

Значение St	*Выражение*	*Результат*
'ABCDEFG'	Copy(St, 2, 3)	'BCD'
'ABCDEFG'	Copy(St, 4, 10)	'DEFG'

Concat (Strl,Str2,...,StrN) — выполняет сцепление строк *Strl, Str2,..,StrN* в том порядке, в каком они указаны в списке параметров. Сумма символов всех сцепленных строк не должна превышать 255.

Например:

Выражение	*Результат*
Concat('AA','XX','Y')	'AAXXY' 'Ин-
Concat('Индекс','394063')	декс 394063'

Length (St) — вычисляет текущую длину в символах строки *St.* Результат имеет целочисленный тип.

Например:

Значение St	*Выражение*	*Результат*
'123456789'	Length(St)	9
'System 370'	Length(St)	10

Pos (Strl,Str2) — обнаруживает первое появление в строке *Str2* подстроки *Strl*. Результат имеет целочисленный тип и равен номеру той позиции, где находится первый символ подстроки *Strl.* Если в *Str2* подстроки *Strl* не найдено, результат равен 0.

Например:

Значение Str1	*Выражение*	*Результат*
'abcdef '	Pos('de',Strl)	4
'abcdef'	Pos('r',Strl)	0

UpCase (Ch) — преобразует строчную букву в прописную. Параметр и результат имеют литерный тип. Обрабатывает буквы только латинского алфавита.

Значение Ch	*Выражение*	*Результат*
'd'	UpCase(Ch)	'D'

Множества

Множество – это структурированный тип данных, представляющий собой набор взаимосвязанных по какому-либо признаку или группе признаков объектов, которые можно рассматривать как единое целое. Каждый объект во множестве называется *элементом множества*.

Все элементы множества должны принадлежать одному из скалярных типов, кроме вещественного. Этот тип называется *базовым типом множества*. Базовый тип задается диапазоном или перечислением. *Область значений* типа множество – набор всевозможных подмножеств, составленных из элементов базового типа.

В выражениях на языке Паскаль значения элементов множества указываются в квадратных скобках:[1,2,3,4], [‘a’,’b’,’c’], [‘a’..’z’]. Если множество не имеет элементов, оно называется *пустым* и обозначается, как [ ]. Количество элементов множества называется его *мощностью*. В отличие от массива порядок перечисления элементов во множестве не имеет значения, и количество элементов заранее не определено. Количество элементов, входящих во множество, может меняться от 0 до 256. Значения множества задаются в квадратных скобках перечислением элементов через запятую.

Например:

[ ]- пустое множество;

[2, 3, 7,11] - множество из целых чисел;

['a', 'c', T]; - множество из символов;

[1..10] - множество из элемента ограниченного типа;

[k,.2*k] - элемент множества задастся текущим значением переменной k.

Формат записи множественных типов:

Type
<имя типа> = **set** of <элемент 1,…., элементN>;
Var
<идентификатор, ….> : <имя типа>;

Можно задать множественный тип и без предварительного описания:

Var
<идентификатор, …> : set of <элемент1, …>;

Например,

```
Type
   Simply  = set of  ‘a’ ..’h’;
   Number = set of  1..31;
Var
  Pr : Simply;
  N : Number;
  Letter : set of char; {определение множества без предварительного описания в разделе типов}
```

В данном примере переменная *Pr* может принимать значения символов латинского алфавита от ‘a’ до ‘h’; *N* – любое значение в диапазоне 1..31; *Letter* – любой символ. Попытка присвоить другие значения вызовет программное прерывание.

Количество элементов множества не должно превышать 256, соответственно номера значений базового типа должны находиться в диапазоне 0..255.

Операции над множествами

При работе с множествами допускается использование операций отношения “=”, “<>”, “>=”, “<=”, объединения, пересечения, разности множеств и операции in. Результатом выражения с применением этих операций является значение *True* или *False*.

Операция “равно” (=). Два множества A и B считаются равными, если состоят из одних и тех же элементов. Порядок следования элементов в сравниваемых множествах значения не имеет.

Например:

Значение A	*Значение B*	*Выражение*	*Результат*
[1,2,3,4]	[1,2,3,4]	A=B	True
[‘a’, ’b’, ’c’]	[‘c’, ‘a’]	A=B	False
[‘a’..’z’]	[‘z’..’a’]	A=B	True

Операция “не равно” (<>). Два множества A и B считаются не равными, если они отличаются по мощности или по значению хотя бы одного элемента.

Например:

Значение A	*Значение B*	*Выражение*	*Результат*
[1,2,3]	[3,1,2,4]	A<>B	True
[‘a’..’z’]	[‘b’.. ‘z’]	A<>B	True
[‘c’..’t’]	[‘t’..’c’]	A<>B	False

Операция “больше или равно” (>=). Операция “больше или равно” (>=) используется для определения принадлежности множеств. Результат операции A >=B равен True, если все элементы множества B содержаться в множестве A. В противном случае результат равен False.

Например:

Значение A	*Значение B*	*Выражение*	*Результат*
[1,2,3,4]	[2,3,4]	A>=B	True
[‘a’..’z’]	[‘b’.. ‘t’]	A>=B	True
[‘z’,’x’,’c’]	[‘c’,’x’]	A>=B	True

Операция “меньше или равно” (<=). Эта операция используется аналогично предыдущей операции, но результат выражения A<=B равен True, если все элементы множества A содержаться во множестве B. В противном случае результат равен False.

Значение A	*Значение B*	*Выражение*	*Результат*
[1,2,3]	[1,2,3,4]	A<=B	True
[‘d’..’h’]	[‘z’.. ‘a’]	A<=B	True
[‘a’,’v’]	[‘a’,’n’,’v’]	A<=B	True

Операция in. Операция in используется для проверки принадлежности какого-либо значения указанному множеству. Обычно применяется в условных операторах.

Значение A	*Значение B*	*Результат*
2	if A in [1,2,3] then..	True
‘v’	if A in [‘a’..’n’] then..	True
X1	if A in [X0,X1,X2,X3] then..	True

При использовании операции in проверяемое на принадлежность значение и множество в квадратных скобках не обязательно предварительно описывать в разделе описаний. Операция in позволяет эффективно и наглядно производить сложные проверки условий, заменяя иногда десятки других операций. Например, выражение if(a=1) or (a=2) or (a=3) or (a=4) or (a=5) or (a=6) then…можно заменить более коротким выражением

if a in [1..6] then… .

Часто операцию in пытаются записать с отрицанием: *X NOT in M.* Такая запись является ошибочной, так как две операции следуют подряд; правильная инструкция имеет вид*: NOT (X in M).*

Объединение множеств (+). Объединением двух множеств является третье множество, содержащее элементы обоих множеств.

Например:

Значение A	*Значение B*	*Выражение*	*Результат*
[1,2,3]	[1,4,5]	A+B	[1,2,3,4,5]
[‘A’..’D’]	[‘E’.. ‘Z’]	A+B	[‘A’..’Z’]
[]	[]	A+B	[]

Пересечение множеств (*). Пересечением двух множеств является третье множество, которое содержит элементы, входящие одновременно в оба множества.

Например:

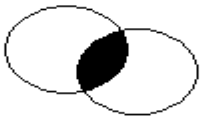
Значение A	*Значение B*	*Выражение*	*Результат*
[1,2,3]	[1,4,2,5]	A*B	[1,2]
[‘A’..’Z’]	[‘B’.. ‘R’]	A*B	[‘B’..’R’]
[]	[]	A*B	[]

Разность множеств (-). Разностью двух множеств является третье множество, которое содержит элементы первого множества, не входящие во второе множество.

Например:

Значение A	*Значение B*	*Выражение*	*Результат*
[1,2,3,4]	[3,4,1]	A-B	[2]
[‘A’..’Z’]	[‘D’.. ‘Z’]	A-B	[‘A’..’C’]
[X1,X2,X3,X4]]	[X4,X1]	A-B	[X2,X3]

Результат операций рад двумя множествами можно наглядно представить с помощью закрашенных частей двух кружочков:


Объединение **Пересечение** **Разность**

Использование в программе данных типа **set** дает ряд преимуществ: значительно упрощаются сложные операторы if , увеличивается степень наглядности программы и понимания алгоритма решения задачи, экономятся память, время компиляции и выполнения.

Имеются и отрицательные моменты, основной из них – отсутствие в языке Паскаль средств ввода-вывода элементов множества, поэтому программист сам должен писать соответствующие процедуры.

Пример 1.

```
Program Dem_Mno;   {демонстрация операций над множествами}
Type
   Digits=set of 0..9;
Var
   D1, D2, D3, D:Digits;
Begin
     D1:=[2,4,6,8];      {заполнение множеств}
     D2:=[0..3,5];
     D3:=[1,3,5,7,9];
     D:=D1+D2;     {объединение множеств D1 и D2}
     D:=D+D3;        {объединение множеств D и D3 }
     D:=D-D2;        {разность множеств D и D2 }
     D:=D*D1;        {пересечение множеств D и D1}
end.
```

Сначала описан тип *Digits=set of 0..9,* затем описаны переменные *D1,D2,D3,D* этого типа. В первой части программы осуществляется заполнение множеств, а затем над множествами выполняются операции объединения, пересечения, разности.

Пример 2. *Описать множество M (1..50) и сделать его пустым. Вводя целые числа с клавиатуры, заполнить множество 10 элементами.*

```
Program Input_Mno;
Var
     M:set of 1..50;
     X,I:integer;
Begin
     M:= [ ];              {M – пустое множество}
     for I:=1 to 10 do
          begin
             write('введите ', I, ' –й элемент множества: ');
```

```
            readln (X);
            if (X in M) then    {если введенное число входит в множество M}
                        begin
                        writeln(X,' помещен в множество 1..50');
                        M:=M+[X];
                        end;
        end;
writeln;
end.
```

В разделе описания переменных описано множество целых чисел от 1 до 50, переменная X целого типа, которая используется для считывания числа-кандидата в множество, и целая переменная I, используемая для подсчета количества введенных чисел. В начале программы применена операция инициализации множества M, так как оно не имеет элементов и является пустым:

M:= [ ];

Заполнение множества элементами производится с использованием оператора повтора for, параметр которого I будет указывать порядковый номер вводимого элемента. Операция заполнения множества записывается оператором присваивания:

M:=M+[X];

Контроль заполнения множества записан операцией проверки принадлежности in. Если условие X in M выполняется, выводится сообщение о том, что число X помещено в множество.

Пример 3. Описывается множество гласных и согласных букв русского языка и определяется количество гласных и согласных букв в предложении, введенном с клавиатуры пользователем.

```
Program Glasn_Sogl;
Type
      Letters=set of 'А'..'я';
Var
      Glasn, Sogl:Letters;
      Text:String;
      I:Byte;
      G,S:Byte;
Begin
      Glasn:=['А','а', 'Е','е','И','и','О','о','У','у','Э','Ю','ю', 'Я','я'];
      Sogl:=['Б'..'Д','б'..'д','Ж','ж','З','з','К'..'Н','к'..'н','П'..'Т','п'..'т','Ф'..'Щ','
      'ф'..'щ','Ъ','ъ','Ь','ь'];
      Write('Введите предложение');
      Readln(Text);
      G:=0;
      S:=0;
For I:=1 to Length(Text) do
        Begin
           If Text[I] in Glasn then G:=G+1;
           If Text[I] in Sogl then S:=S+1;
```

```
    End;
Writeln('В предложении" ',Text,' "',G,'гласных и ',S,'согласных букв');
End.
```

Зададим тип Letters –множество букв русского языка, затем опишем переменные этого типа: Glasn-множество гласных букв, Sogl-множество согласных букв. Вводимое с клавиатуры предложение опишем переменной Text типа String. Для указания символа в строке Text применим переменную I типа Byte.Для подсчета количества гласных и согласных букв опишем переменные Gи S. Проверку принадлежности символов, составляющих предложение множествам гласных или согласных букв русского языка запишем с использованием цикла for , параметр I которого, изменяясь от 1 до значения длины предложения, будет указывать порядковый номер символа в предложении. Принадлежность очередного символа предложения множеству гласных или согласных букв запишем операцией in. Если символ является гласной буквой (Text[I] in Glasn), то счетчик гласных букв G увеличивается на 1. Аналогично с согласными буквами.

Тема 2.5 Подпрограммы

Подпрограмма – именованная, логически законченная группа операторов языка, которую можно вызвать для выполнения любое количество раз из разных мест программы. В языке Паскаль существует два вида подпрограмм: процедуры и функции.

Процедуры

Процедура – это подпрограмма, которая решает некоторую частную задачу или объединяет группу часто встречающихся операторов. Каждая процедура должна быть предварительно описана в разделе описаний процедур и функций. Описание процедуры состоит из заголовка и тела процедуры. Описание процедуры содержит служебное слово procedure, имя процедуры и заключенный в скобки список формальных параметров с указанием их типов. Формальные параметры отделяются точкой с запятой (список однотипных параметров может быть перечислен через запятую).

Procedure <имя процедуры> (<список формальных параметров>);

После заголовка идут разделы описаний (констант, типов, переменных, процедур и функций, используемых в процедуре) и операторы языка Паскаль, реализующие алгоритм процедуры.

```
const ... ;
type . . . ;
var   . . . ;
begin
        <операторы>
end ;
```

Формальные параметры фиктивно присутствуют в процедуре и определяют тип и действия, которые над ними производятся. Их нельзя описывать в разделе описаний процедуры.

Во время выполнения формальные параметры заменяются фактическими параметрами. Фактические параметры – это параметры, которые передаются процедуре при обращении к ней.

Формальные параметры делятся на параметры – значения и параметры - переменные. Параметры – значения используются в качестве входных данных подпрограммы, при обращении к которой фактические параметры передают свое значение формальным и больше не меняются. Параметры-значения задаются следующим образом:

<Имя параметра>:< тип параметра>.

Параметры – переменные могут использоваться как в качестве входных, так и в качестве выходных. В заголовке процедуры перед ними необходимо указывать служебное слово var:

Var <имя параметра>:<тип параметра>

При обращении к подпрограмме фактические параметры замещают формальные. В результате выполнения подпрограммы изменяются фактические параметры.

Вызов процедуры. Для обращения к процедуре необходимо использовать оператор вызова процедуры. Он имеет следующий вид:

<имя процедуры> (<список фактических параметров>);

При вызове процедуры основная программа приостанавливает свою работу и передает управление в процедуру. При завершении процедуры передается управление на команду, следующую за вызовом процедуры.

Количество, типы и порядок следования формальных и фактических параметров должны совпадать.

Пример. Подпрограмма должна заполнить массив исходными данными.

```
Program proba;
Const k=100
Type Tm=array[1…k] of integer;
Var M:TM ; i; fk; byte;
Procedure InputMas (Var M:TM ; i; fk; byte;)
var i: byte;
begin
for i:=1 to10 do
begin
Mas[i]:=round(random(50)-25);
{= = = = = = = = = = = = = = = = = = = = = = = =}
begin
write('Введите количество элементов');
repeat read (fk); until fk<=k;
inputMas(M,fk);
for i:=1 to fk do
write(M[i]:5);
end.
```

Функции

Функция, определенная пользователем, состоит из *заголовка* и тела *функции*.

Заголовок содержит зарезервированное слово *function*, имя функции, заключенное в круглые скобки, необязательный список формальных параметров и тип возвращаемого функцией значения. Тело функции представляет собой локальный блок, по структуре аналогичный программе. В целом структура функции, определенной пользователем имеет вид:

```
function <имя> (формальные параметры) : <тип результата>;
const ...
type ...
var
begin
        <операторы>
end;
```

В разделе операторов должен находиться, по крайней мере, один оператор, *присваивающий имени функции значение*. В точку вызова возвращается результат последнего присваивания.

Обращение к функции осуществляется по имени с необязательным указанием списка аргументов. Каждый аргумент должен соответствовать формальным параметрам, указанным в заголовке, и иметь тот же тип.

Пример. Составить программу вычисления выражения $Z=(A^{5} + A^{-3})/2* A^{M}$, в которой возведение в степень выполняется *функцией Step*.

```
program DemoFunc;
Var
M     : integer;
A,Z,R : real ;
{Функция вычисления  степени. N - степень, X – число, возводимое в данную степень. N, X — формальные параметры; результат, возвращаемый функцией в точку вызова, имеет вещественный тип}
function Step(N : integer; X : real): real;
Var
I : integer;
Y : real;
begin
 Y:=1;
       for I:=1 to N do{Цикл вычисления N—й степени числа X)
              Y:=Y*X;
              Step:=Y ;          {Присваивание функции результата вычисления степени}
end; {Конец функции}
       Begin          {Начало основной программы}
       Write('Введите значение числа А и показатель степени M');
       Readln(A,M) ;
       Z:=Step(5,A) ; {Вызов функции с передачей ей фактических параметров N=5, X=A}
       Z:=Z+ Step(3,l/A); {Вызов функции с передачей ей фактических параметров N=3, X=1/A}
         if M=0 then R:=l {если число возводится в нулевую степень, то результат всегда равен 1}
         else if M>0 then R:=Step(M,A){Вызов функции Step с передачей ей фактических параметров M, A}
         else R:=Step(-M,A); {Вызов функции с передачей ей фактических параметров: - M, отрицательная степень}
         Z:=Z/(2*R) ;
         Writeln(' Для A= ',A,'M= ',M,' Значение выражения= ',Z);
end.
```

В начале программы описываются переменная целого типа *M* и переменные вещественного типа *A, Z, R*, после этого описывается функция вычисления степени числа *Step* с формальными параметрами *N* и *X*, результат, возвращаемый функцией в точку вызова, - вещественного типа.

В описании функции вводятся две *локальных (местных) переменных I* и *Y*. Переменная *I* служит для подсчета числа повторений цикла, а в *Y* накапливается значение степени как произведения *N* одинаковых сомножителей. В заключение функции присваивается значение вычисленного произведения.

В начале выполнения основной программы на экран выводится запрос *"Введите значение числа А и показатель степени М"* и считывается с клавиатуры значение вещественного числа А и целого числа М.

Затем выполняется оператор:

Z:=Step(5,A);

Осуществляется вызов функции *Step* с передачей ей фактических параметров *5, А*. Их значения присваиваются формальным параметрам функции *N* и *X*. По окончании вычисления степени числа значение функции *Step*, вычисленное для фактических параметров *5* и *А*, присваивается переменной *Z*. Аналогично в операторе:

Z := Z + Step(3,l/A);

сначала осуществляется вызов функции *Step* с передачей ей фактических параметров *3, 1/А*, после чего значение переменной *Z* увеличивается на величину возвращенного в основную программу результата вычисления функции *Step*.

Операторы:

```
if M=0 then R:=1
    else if M>0 then R:=Step(M,A)
    else R:=Step(- M,A);
```

проверяют условия *M=0, M>0* и в зависимости от их соблюдения либо присваивает переменной *R* значение 1 *(при М=О)*, либо выполняет вызов функции *Step* для *фактических значений М, А* или *-М, А*, а после вычисления значения функции *Step* присваивает его переменной R.

Оператор:

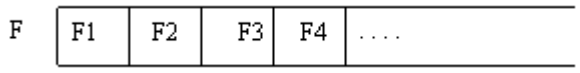
```
Z:=Z/(2*R);
```

выполняет вычисление значения выражения, а затем присваивает вычисленное значение переменной *Z*.

В заключение программы стандартная процедура *Writeln* выводит на экран сообщение о результате вычислений степени *М* числа *А*.

Тема 2.6 Файлы

Существенной особенностью всех рассмотренных до сих пор значений производных типов является наличие в них конечного, наперед заданного числа компонент. Так, в значении многомерного массива это число можно определить, зная количество компонент по каждому измерению, а в значении записи это число определяется количеством и типом полей. Таким образом, заранее, еще до выполнения программы, по этому описанию можно выделить необходимый объем памяти машины для хранения значений переменных этих типов. Но существует определенный класс задач и определенные ситуации, когда количество компонент (пусть даже одного и того же из известных уже типов) заранее определить невозможно, оно выясняется только в процессе решения задачи. Поэтому возникает необходимость в специальном типе значений, которые представляют собой произвольные последовательности элементов одного и того же типа, причем длина этих последовательностей заранее не определяется, а конкретизируется в процессе выполнения программы. Этот тип значений получил название файлового типа. Условно файл в Паскале можно изобразить как некоторую ленту, у которой есть начало, а конец не фиксируется. Элементы файла записываются на эту ленту последовательно друг за другом:



где F – имя файла, а F1, F2, F3, F4 – его элементы. Файл во многом напоминает магнитную ленту, начало которой заполнено записями, а конец пока свободен. В программировании существует несколько разновидностей файлов, отличающихся методом доступа к его компонентам: файлы последовательного доступа и файлы произвольного доступа.

Простейший метод доступа состоит в том, что по файлу можно двигаться только последовательно, начиная с первого его элемента, и, кроме этого, всегда существует возможность начать просмотр файла с его начала. Таким образом, чтобы добраться до пятого элемента файла, необходимо, начав с первого элемента, пройти через предыдущие четыре. Такие файлы называют файлами последовательного доступа. У последовательного файла доступен всегда лишь очередной элемент. Если в процессе решения задачи необходим какой-либо из предыдущих элементов, то необходимо вернуться в начало файла и последовательно пройти все его элементы до нужного.

Файлы произвольного доступа Паскаля позволяют вызывать компоненты в любом порядке по их номеру.

Важной особенностью файлов является то, что данные, содержащиеся в файле, переносятся на внешние носители. Файловый тип Паскаля – это единственный тип значений, посредством которого данные, обрабатываемые программой, могут быть получены извне, а результаты могут

быть переданы во внешний мир. Это единственный тип значений, который связывает программу с внешними устройствами ЭВМ.

Работа с файлами в Паскале

Любой файл имеет три характерные особенности. Во-первых, у него есть имя, что дает возможность программе работать одновременно с несколькими файлами. Во-вторых, он содержит компоненты одного типа. Типом компонентов может быть любой тип Паскаля, кроме файлов. Иными словами, нельзя создать «файл файлов». В-третьих, длина вновь создаваемого файла никак не оговаривается при его объявлении и ограничивается только емкостью устройств внешней памяти.

Файловый тип или переменную файлового типа в Паскале можно задать одним из трех способов:

Type <имя_ф_типа>=file of<тип_элементов>;

<имя_ф_типа>=text;

<имя_ф_типа>=file;

Здесь <имя_ф_типа> – имя файлового типа (правильный идентификатор); File, of – зарезервированные слова (файл, из); <тип_элементов> – любой тип Паскаля, кроме файлов.

Пример описания файлового типа в Паскале

```
Type
   Product= record
      Name: string;
      Code: word;
   End;
   Text80= file of string[80];
Var
   F1: file of char;
   F2: text;
   F3: file;
   F4: Text80;
   F5: file of Product;
```

В зависимости от способа объявления можно выделить три вида файлов Паскаля:

- типизированные файлы Паскаля(задаются предложением file of..);
- текстовые файлы Паскаля(определяются типом text);
- нетипизированные файлы Паскаля(определяются типом file).

Следует помнить, что физические файлы на магнитных дисках и переменные файлового типа в программе на Паскале – объекты различные. Переменные файлового типа в Паскале могут

соответствовать не только физическим файлам, но и логическим устройствам, связанным с вводом/выводом информации. Например, клавиатуре и экрану соответствуют файлы со стандартными именами Input, Output.

Каждый тип данных в Паскале, вообще говоря, определяет множество значений и множество операций над значениями этого типа. Однако над значениями файлового типа Паскаля не определены какие-либо операции, в том числе операции отношения и присваивания, так что даже такое простое действие, как присваивание значения одной файловой переменной другой файловой переменной, имеющей тот же самый тип, запрещено. Все операции могут производиться лишь с элементами (компонентами) файлов. Естественно, что множество операций над компонентами файла определяется типом компонент.

Переменные файлового типа используются в программе только в качестве параметров собственных и стандартных процедур и функций.

Основные процедуры и функции для работы с файлами

1. До начала работы с файлами в Паскале необходимо установить связь между файловой переменной и именем физического дискового файла:

Assign(<файловая_переменная>, <имя_дискового_файла>)

Следует помнить, что имя дискового файла при необходимости должно содержать путь доступа к этому файлу, включая имя дисковода. При этом имя дискового файла – строковая величина, т.е. должна быть заключена в апострофы. Например:

Assign (chf, 'G:\Home\ Student\ Lang\ Pascal\ primer.dat');

Если путь не указан, то программа будет искать файл в своем рабочем каталоге и по указанным путям в autoexec.bat.

Вместо имени дискового файла можно указать имя логического устройства, каждое из которых имеет стандартное имя:

CON – консоль, т.е. клавиатура-дисплей;

PRN – принтер. Если к компьютеру подключено несколько принтеров, доступ к ним осуществляется по именам LPT1, LPT2, LPT3.

Не разрешается связывать с одним физическим файлом более одной файловой переменной.

2. После окончания работы с файлами на Паскале, они должны быть закрыты.

Close(<список файловых переменных>);

При выполнении этой процедуры закрываются соответствующие физические файлы и фиксируются сделанные изменения. Следует иметь в виду, что при выполнении процедуры close связь файловой переменной с именем дискового файла, установленная ранее процедурой assign, сохра-

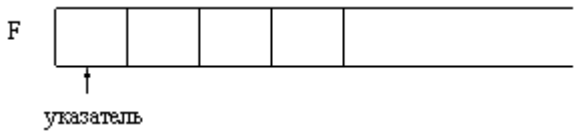
няется, следовательно, файл можно повторно открыть без дополнительного использования процедуры assign.

Работа с файлами заключается, в основном, в записи элементов в файл и считывании их из файла. Для удобства описания этих процедур введем понятие указателя, который определяет позицию доступа, т.е. ту позицию файла, которая доступна для чтения (в режиме чтения), либо для записи (в режиме записи). Позиция файла, следующая за последней компонентой файла (или первая позиция пустого файла) помечается специальным маркером, который отличается от любых компонент файла. Благодаря этому маркеру определяется конец файла.

3. Подготовка к записи в файл Паскаля

Rewrite(<имя_ф_переменной>);

Процедура Rewrite(f) (где f – имя файловой переменной) устанавливает файл с именем f в начальное состояние режима записи, в результате чего указатель устанавливается на первую позицию файла. Если ранее в этот файл были записаны какие-либо элементы, то они становятся недоступными. Результат выполнения процедуры rewrite(f); выглядит следующим образом:

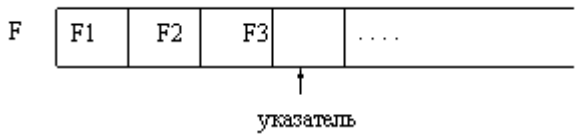


4. Запись в файл Паскаля

Write(<имя_ф_переменной>, <список записи>);

При выполнении процедуры write(f, x) в ту позицию, на которую показывает указатель, записывается очередная компонента, после чего указатель смещается на следующую позицию. Естественно, тип выражения x должен совпадать с типом компонент файла. Результат действия процедуры write(f, x) можно изобразить так:

Состояние файла f до выполнения процедуры



Состояние файла f после выполнения процедуры

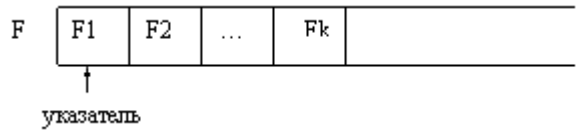
Для типизированных файлов выполняется следующее утверждение: если в списке записи перечислено несколько выражений, то они записываются в файл, начиная с первой доступной пози-

ции, а указатель смещается на число позиций, равное числу записываемых выражений.

5. Подготовка файла к чтению Паскаля

Reset(<имя_ф_переменной>);

Эта процедура ищет на диске уже существующий файл и переводит его в режим чтения, устанавливая указатель на первую позицию файла. Результат выполнения этой процедуры можно изобразить следующим образом:



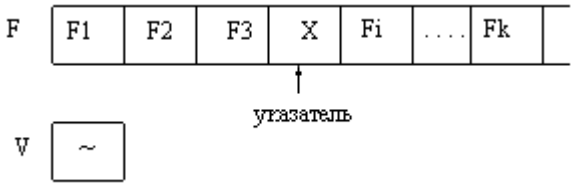
Если происходит попытка открыть для чтения не существующий еще на диске файл, то возникает ошибка ввода/вывода, и выполнение программы будет прервано.

6. Чтение из файла в Паскале

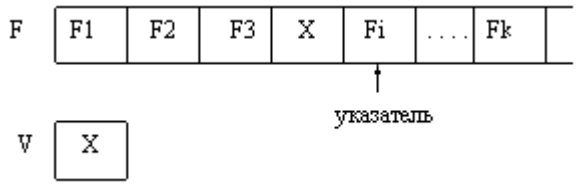
Read(<имя_ф_переменной>,<список переменных>);

Рассмотрим результат действия процедуры read(f, v):

Состояние файла f и переменной v до выполнения процедуры:



Состояние файла f и переменной v после выполнения процедуры:



Для типизированных файлов при выполнении процедуры read() последовательно считывается, начиная с текущей позиции указателя, число компонент файла, соответствующее числу переменных в списке, а указатель смещается на это число позиций.

В большинстве задач, в которых используются файлы, необходимо последовательно перебрать компоненты и произвести их обработку. В таком случае необходимо иметь возможность

определять, указывает ли указатель на какую-то компоненту файла, или он уже вышел за пределы файла и указывает на маркер конца файла.

7. Функция определения достижения конца файла в Паскале

   Eof(<имя_ф_переменной>);

Название этой функции является сложносокращенным словом от end of file. Значение этой функции имеет значение true, если конец файла уже достигнут, т.е. указатель стоит на позиции, следующей за последней компонентой файла. В противном случае значение функции – false.

8. Изменение имени файла в Паскале

   Rename(<имя_ф_переменной>, <новое_имя_файла>);

Здесь новое_ имя_ файла – строковое выражение, содержащее новое имя файла, возможно с указанием пути доступа к нему.

Перед выполнением этой процедуры необходимо закрыть файл, если он ранее был открыт.

9. Уничтожение файла в Паскале

   Erase(<имя_ф_переменной>);

Перед выполнением этой процедуры необходимо закрыть файл, если он ранее был открыт.

10. Уничтожение части файла от текущей позиции указателя до конца в Паскале

    Truncate(<имя_ф_переменной>);

11. Файл Паскаля может быть открыт для добавления записей в конец файла

    Append(<имя_ф_переменной>);

Типизированные файлы Паскаля.

Длина любого компонента типизированного файла строго постоянна, т.к. тип компонент определяется при описании, а, следовательно, определяется объем памяти, отводимый под каждую компоненту. Это дает возможность организовать прямой доступ к каждой компоненте (т.е. доступ по порядковому номеру).

Перед первым обращением к процедурам ввода/вывода указатель файла стоит в его начале и указывает на его первый компонент с номером 0. После каждого чтения или записи указатель сдвигается к следующему компоненту файла. Переменные и выражения в списках ввода и вывода в процедурах read() и write() должны иметь тот же тип, что и компоненты файла Паскаля. Если этих переменных или выражений в списке несколько, то указатель будет смещаться после каждой операции обмена данными на соответствующее число позиций.

Для облегчения перемещения указателя по файлу и доступа к компонентам типизированного файла существуют специальные процедуры и функции:

- функция Паскаля, определяющая число компонентов в файле:

fileSize(<имя_ф_переменной>)

– функция Паскаля, значением которой является текущая позиция указателя:

filePos(<имя_ф_переменной>)

– процедура Паскаля, смещающая указатель на компоненту файла с номером n:

seek(<имя_ф_переменной>,n)

Так, процедура seek(<имя_ф_переменной>,0) установит указатель в начало файла, а процедура seek(<имя_ф_переменной>, FileSize(<имя_ф_переменной>)) установит указатель на признак конца файла.

Текстовые файлы Паскаля

Текстовые файлы предназначены для хранения текстовой информации. Именно в таких файлах хранятся, например, исходные тексты программ. Компоненты текстовых файлов могут иметь переменную длину, что существенно влияет на характер работы с ними. Доступ к каждой строке текстового файла Паскаля возможен лишь последовательно, начиная с первой. К текстовым файлам применимы процедуры assign, reset, rewrite, read, write и функция eof. Процедуры и функции seek, filepos, filesize к ним не применяются. При создании текстового файла в конце каждой записи (строки) ставится специальный признак EOLN(end of line – конец строки). Для определения достижения конца строки существует одноименная логическая функция EOLN(<имя_ф_переменной>), которая принимает значение true, если конец строки достигнут.

Форма обращения к процедурам write и read для текстовых и типизированных файлов одинакова, но их использование принципиально различается.

В списке записываемых в текстовый файл элементов могут чередоваться в произвольном порядке числовые, символьные, строковые выражения. При этом строковые и символьные элементы записываются непосредственно, а числовые из машинной формы автоматически преобразуются в строку символов.

текстовые файлы удобнее для восприятия человеком, а типизированные соответствуют машинному представлению объектов;

текстовые файлы, как правило, длиннее типизированных;

длина текстовых файлов зависит не только от количества записей, но и от величины переменных.

Так, в типизированном файле числа 6, 65 и 165 как целые будут представлены одним и тем же числом байт. А в текстовых файлах, после преобразования в строку, они будут иметь разную длину. Это вызывает проблемы при расшифровке текстовых файлов. Пусть в текстовый файл пишутся подряд целые числа (типа byte): 2, 12, 2, 128. Тогда в файле образуется запись

2122128. При попытке прочитать из такого файла переменную типа byte программа прочитает всю строку и выдаст сообщение об ошибке, связанной с переполнением диапазона.

Чтобы избежать этой ошибки, достаточно вставить при записи в файл после каждой переменной пробел. Тогда программа при каждом чтении берет символы от пробела до пробела и правильно преобразует текстовое представление в число.

Кроме процедур read и write при работе с текстовыми файлами используются их разновидности readln и writeln. Отличие заключается в том, что процедура writeln после записи заданного списка записывает в файл специальный маркер конца строки. Этот признак воспринимается как переход к новой строке. Процедура readln после считывания заданного списка ищет в файле следующий признак конца строки и подготавливается к чтению с начала следующей строки.

Пример. *Пусть нам необходимо сформировать текстовый файл с помощью Паскаля, а затем переписать из данного файла во второй только те строки, которые начинаются с буквы «А» или «а».*

Пояснения: нам понадобятся две файловые переменные f1 и f2, поскольку оба файла текстовые, то тип переменных будет text. Задача разбивается на два этапа: первый – формирование первого файла; второй – чтение первого файла и формирование второго.

Для завершенности решения задачи есть смысл добавить еще одну часть, которая в задаче явно не указана – вывод на экран содержимого второго файла.

```
Program primer;
 Var f1,f2:text;
    I,n: integer;
    S: string;
 Begin
 {формируем первый файл}
    Assign(f1, ‘file1.txt’); {устанавливаем связь файловой переменной с физическим файлом на диске}
    Rewrite(f1);  {открываем файл для записи}
    Readln(n) {определим количество вводимых строк}
    for i:=1 to n do
    begin
       readln(s); {вводим с клавиатуры строки}
       writeln(f1,s); {записываем последовательно строки в файл}
    end;
    close(f1); {заканчиваем работу с первым файлом, теперь на диске существует файл с именем file1.txt, содержащий введенные нами строки. На этом программу можно закончить, работу с файлом можно продолжить в другой программе, в другое время, но мы продолжим}

 {часть вторая: чтение из первого файла и формирование второго}
    Reset(f1); {открываем первый файл для чтения}
    Assign(f2, ‘file2.txt’); {устанавливаем связь второй файловой переменной с физическим файлом}
    Rewrite(f2); {открываем второй файл для записи}
```

```
{Дальше необходимо последовательно считывать строки из первого файла, проверять выполнение условия и записывать нужные строки во второй файл. Для чтения из текстового файла рекомендуется использовать цикл по условию «пока не конец файла»}
  While not eof(f1) do
  Begin
    Readln(f1,s);{считываем очередную строку из первого файла}
    If (s[1]='A') or (s[1]='a') then
    Writeln(f2,s); {записываем во второй файл строки, удовлетворяющие условию}
  End;
  Close(f1,f2); {заканчиваем работу с файлами}

{часть третья: выводим на экран второй файл}
  Writeln;
  Writeln('Второй файл содержит строки:');
  Reset(f2); {открываем второй файл для чтения}
  While not eof(f2) do {пока не конец второго файла}
  Begin
    Readln(f2,s);{считываем очередную строку из второго файла}
    Writeln(s); {выводим строку на экран}
  End;
End.
```

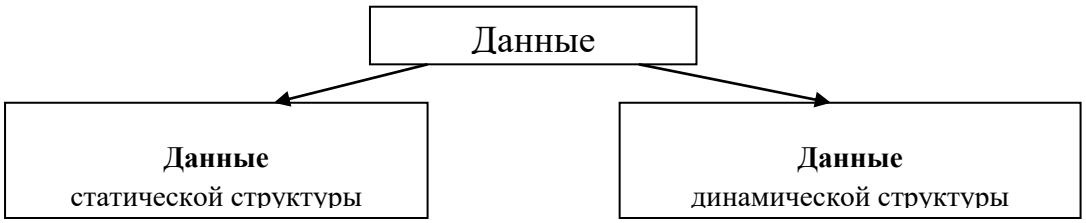
Тема 2.7 Указатели и динамическая память

Динамические структуры данных

Объект данных обладает динамической структурой, если его размер изменяется в процессе выполнения программы или он потенциально бесконечен.

Классификация структур данных

Используемые в программировании данные можно разделить на две большие группы:



Данные статической структуры – это данные, взаиморасположение и взаимосвязи элементов которых всегда остаются постоянными.

Данные динамической структуры – это данные, внутреннее строение которых формируется по какому-либо закону, но количество элементов, их взаиморасположение и взаимосвязи могут динамически изменяться во время выполнения программы, согласно закону формирования.

К данным динамической структуры относят файлы, несвязанные и связанные динамические данные.

Файлы в данной классификации отнесены к динамическим структурам данных. Это сделано исходя из вышеприведенного определения. Хотя удаление и вставка элементов в середину файла не допускается, зато длина файла в процессе работы программы может изменяться – увеличиваться или уменьшаться до нуля. А это уже динамическое свойство файла как структуры данных.

Статические и динамические переменные в Паскале

В Паскале одной из задач описания типов является то, чтобы зафиксировать на время выполнения программы размер значений, а, следовательно, и размер выделяемой области памяти для них. Описанные таким образом переменные называются *статическими*.

Все переменные, объявленные в программе, размещаются в одной непрерывной области оперативной памяти – сегмент данных. Длина сегмента данных определяется архитектурой микропроцессора и составляет обычно 65536 байт.

Однако порой заранее не известны не только размеры значений, но и сам факт существования значения той или иной переменной. Для результата переменной приходится отводить память в расчете на самое большое значение, что приводит к нерациональному использованию памяти. Особенно это затруднительно при обработке больших массивов данных.

Предположим, например, что у вас есть программа, требующая массива в 400 строк по 100 символов каждая. Для этого массива требуется примерно 40К, что меньше максимума в 64К. Если остальные ваши переменные помещаются в оставшиеся 24К, массив такого объема проблемы не представляет.

Но что если вам нужно два таких массива? Это потребовало бы 80К, и 64К сегмента данных не хватит.

Другим общим примером является сортировка. Обычно когда вы сортируете большой объем данных, то делаете копию массива, сортируете копию, а затем записываете отсортированные данные обратно в исходный массив. Это сохраняет целостность ваших данных, но требует также наличия во время сортировки двух копий данных.

С другой стороны объем памяти компьютера достаточно велик для успешного решения задач с большой размерностью данных. Выходом из положения может служить использование так называемой *динамической памяти.*

Динамическая память (ДП) – это оперативная память ПК, предоставляемая программе при ее работе, за вычетом сегмента данных (64 Кб), стека (16 Кб) и собственно тела программы. Размер динамической памяти можно варьировать. По умолчанию ДП – вся доступная память ПК.

ДП – это фактически единственная возможность обработки массивов данных большой размерности. Многие практические задачи трудно или невозможно решить без использования ДП. Например, при разработке САПР статическое распределение памяти невозможно, т.к. размерность математических моделей в разных проектах может значительно различаться.

И статические и динамические переменные вызываются по их адресам. Без адреса не получить доступа к нужной ячейке памяти, но при использовании статических переменных, адрес непосредственно не указывается. Обращение осуществляется по имени. Компилятор размещает переменные в памяти и подставляет нужные адреса в коды команд.

Адресация динамических переменных осуществляется через *указатели*. Их значения определяют адрес объекта.

Для работы с динамическими переменными в программе должны быть выполнены следующие действия:

- Выделение памяти под динамическую переменную;
- Инициализация указателя;
- Освобождение памяти после использования динамической переменной.

Программист должен сам резервировать место, определять значение указателей, освобождать ДП.

Вместо любой статической переменной можно использовать динамическую, но без реальной необходимости этого делать не стоит.

Указатели

Для работы с динамическими программными объектами в Паскале предусмотрен *ссылочный тип* или *тип указателей*. В переменной ссылочного типа хранится ссылка на программный объект (адрес объекта).

Указатель – это переменная, которая в качестве своего значения содержит адрес байта памяти.

Объявление указателей

Указатель, связанный с некоторым определенным типом данных, называют *типизированным указателем*. Его описание имеет вид:

Имя_переменной: ^ базовый-тип;

Например:

```
Type A= array [1..100] of integer;
     TA= ^A ;     {тип указатель на массив}
Var
     P1: ^integer; {переменная типа указатель на целое число}
     P2: ^ real;     {переменная типа указатель на вещественное число}
```

Указатель, не связанный с каким-либо конкретным типом данных, называется *нетипизированным указателем*. Для описания нетипизированного указателя в Паскале существует стандартный тип ***pointer***. Описание такого указателя имеет вид:

Имя-переменной: pointer;

С помощью нетипизированных указателей удобно динамически размещать данные, структура и тип которых меняются в ходе выполнения программы.

Значения указателей – адреса переменных в памяти. Адрес занимает четыре байта и хранится в виде двух слов, одно из которых определяет сегмент, второе – смещение.

Следовало бы ожидать, что значение одного указателя можно передать другому. На самом деле можно передавать значения только между указателями, связанными с одним типом данных. Указатели на различные типы данных имеют различный тип, причем эти типы несовместимы.

Например,

```
Var     p1,p2: ^integer;
        p3: ^real;
        pp: pointer;
……
p1:= p2;    {допустимое действие}
p1:= p3;    {недопустимое действие}
```

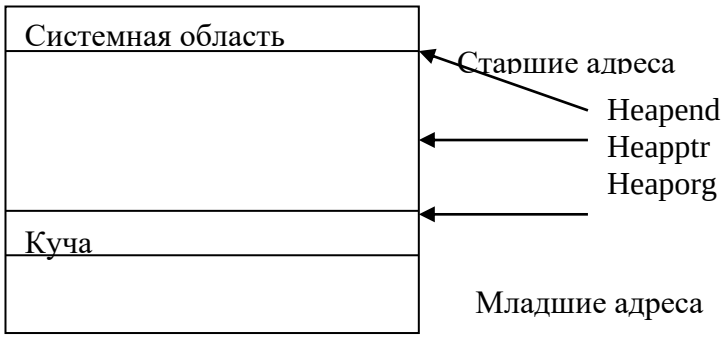
Однако это ограничение не распространяется на нетипизированный указатель. В программе допустимы будут следующие действия:

```
pp:= p3;
p1:= pp;
```

Выделение и освобождение динамической памяти

Вся ДП рассматривается как сплошной массив байтов, который называется *кучей*.

Расположение кучи в памяти ПК.



Существуют стандартные переменные, в которых хранятся значения адресов начала, конца и текущей границы кучи:

Heaporg – начало кучи;

Heapend – конец кучи;

Heapptr – текущая граница незанятой ДП.

Выделение памяти под динамическую переменную осуществляется процедурой:

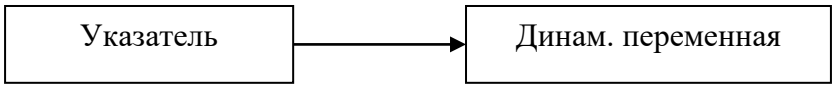
New (переменная_типа_указатель)

В результате обращения к этой процедуре указатель получает значение, соответствующее адресу в динамической памяти, начиная с которого можно разместить данные.

Например,

```
Var  i, j: ^integer;
      r: ^real;
begin
     new(i); {после этого указатель i приобретает значение адреса
               Heapptr, а Heapptr смещается на 2 байта}
……
     new(r) ;  {r приобретает значение Heapptr, а Heapptr смещается
                на 6 байт}
```

Графически действие процедуры new можно изобразить так:



Здесь хранится адрес переменной

В этой области памяти будет храниться значение переменной

Освобождение динамической памяти осуществляется процедурой:

Dispose (переменная_типа_указатель)

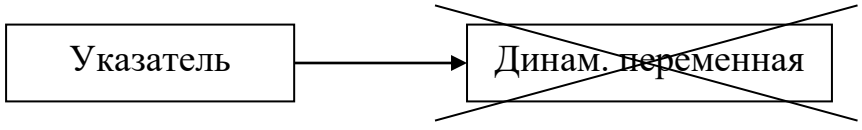
Например:

Dispose (i); {возвращает в кучу 2 байта}

Dispose (r); {возвращает в кучу 6 байт}

Следует помнить, что повторное применение процедуры dispose к свободному указателю может привести к ошибке.

Процедура dispose освобождает память, занятую динамической переменной. При этом значение указателя становится неопределенным.



Любые действия над указателем в программе располагаются между процедурами new и dispose.

При использовании динамически распределяемых переменных часто возникает общая проблема, называемая утечкой динамической памяти. Утечка памяти – это ситуация, когда пространство выделяется в динамически распределяемой памяти и затем теряется – по каким-то

причинам ваш указатель не указывает больше на распределенную область, так что вы не можете освободить пространство.

Общей причиной утечек памяти является переприсваивание динамических переменных без освобождения предыдущих. Простейшим случаем является следующий:

```
var IntPointer: ^Integer;
begin
  New(IntPointer);
  New(IntPointer);
end.
```

При первом вызове New в динамически распределяемой памяти выделяется 2 байта, и на них устанавливается указатель IntPointer. Второй вызов New выделяет другие 2 байта, и IntPointer устанавливается на них. Теперь у вас нет указателя, ссылающегося на первые 2 байта, поэтому вы не можете их освободить. В программе эти байты будут потеряны.

Присваивание значений указателю

Для инициализации указателей существует несколько возможностей.

1) процедура new отводит блок памяти в области динамических переменных и сохраняет адрес этой области в указателе;
2) специальная операция **@** ориентирует переменную-указатель на область памяти, содержащую уже существующую переменную. Эту операцию можно применять для ориентации на статическую и динамическую переменную. Например,

   ```
   type A= array [0..99] of char;
   var X: array [0..49] of integer;
       pA: ^A; {указатель на массив символов}
   ```

   Объявлены переменные разных типов: массив из 50 целых чисел и указатель на массив символов. Чтобы указатель pA указывал на массив X, надо присвоить ему адрес X

   ```
   pA:= @X;
   ```

3) Существует единственная константа ссылочного типа ***nil***, которая обозначает «пустой» адрес. Ее можно присваивать любому указателю.
4) Переменной-указателю можно присвоить значение другого указателя того же типа. Используя указательный тип pointer как промежуточный, можно присвоить значение одного указателя другому при несовпадении типов.

Операции с указателями

Для указателей определены только операции присваивания и проверки на равенство и неравенство. В Паскале запрещаются любые арифметические операции с указателями, их ввод-вывод и сравнение на больше-меньше.

Еще раз повторим правила присваивания указателей:

- любому указателю можно присвоить стандартную константу nil, которая означает, что указатель не ссылается на какую-либо конкретную ячейку памяти;
- указатели стандартного типа pointer совместимы с указателями любого типа;
- указателю на конкретный тип данных можно присвоить только значение указателя того же или стандартного типа данных.

Указатели можно сравнивать на равенство и неравенство, например:

If p1=p2 then …..

If p1<>p2 then …..

В Паскале определены стандартные функции для работы с указателями:

- addr(x) – тип результата pointer, возвращает адрес x (аналогично операции @), где x – имя переменной или подпрограммы;
- seg(x) – тип результата word, возвращает адрес сегмента для x;
- ofs(x) – тип результата word, возвращает смещение для x;
- ptr(seg, ofs) – тип результата pointer, по заданному сегменту и смещению формирует адрес типа pointer.

Присваивание значений динамическим переменным

После того, как динамическая переменная объявлена, ей можно присваивать значения, изменять их, использовать в выражениях и т.д. Для этого используют следующее обращение: переменная_указатель^. Такое обращение называется *операция разадресации (разыменования)*. Таким образом происходит обращение к значению, на которое указывает указатель, т.е. к данным. Если же за переменной_указателем значок ^ не стоит, то имеется в виду адрес, по которому расположены данные.

Динамически размещенные данные можно использовать в любом месте программы, где допустимо использование выражений соответствующего типа.

Например:

```
R^:= sqr(R^) + I^ -17;
q^:= 2; inc(q^); writeln(q^)
```

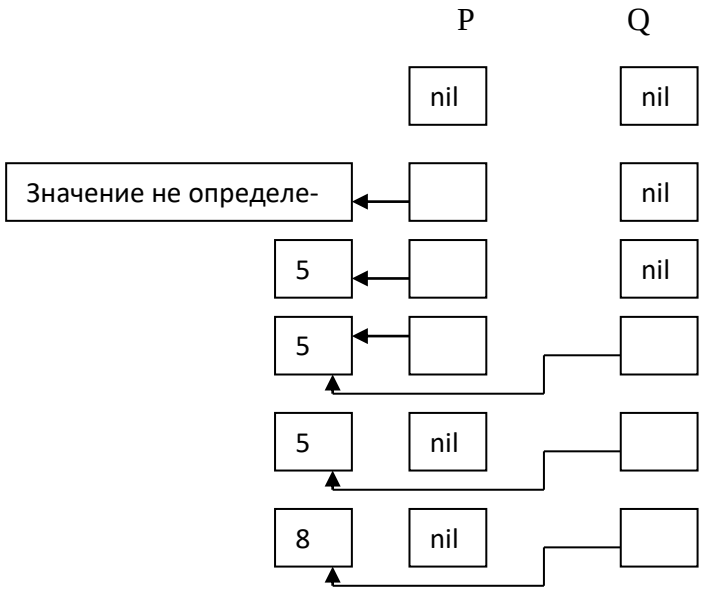
Недопустимо использовать выражения, подобные следующим:

Адрес ⟶ R:= sqr(R^) + I^ -17 ⟵ вещественное выражение.

Вещественная переменная ⟶ R^:= sqr(R) ⟵ аргумент – адрес.

Рассмотрим пример работы с указателями:

```
Var P,Q: ^integer;
   begin

new (P);

P^:= 5;

Q:= P;

P:= nil;

Q^:= 8;
Write(Q^);
```



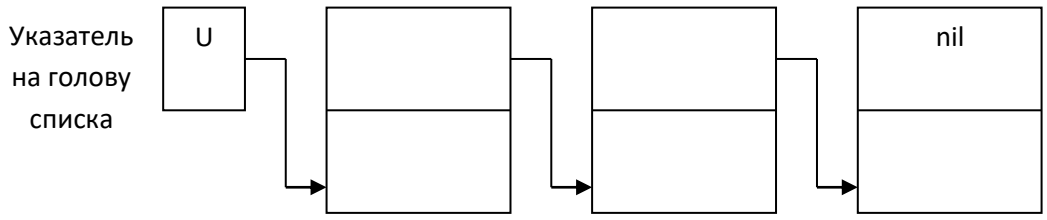
Динамические структуры

Линейные списки (однонаправленные цепочки)

Списком называется структура данных, каждый элемент которой посредством указателя связывается со следующим элементом.

Каждый элемент связанного списка, во-первых, хранит какую-либо информацию, во-вторых, указывает на следующий за ним элемент. Так как элемент списка хранит разнотипные части (хранимая информация и указатель), то его естественно представить записью, в которой в одном поле располагается объект, а в другом – указатель на следующую запись такого же типа. Такая запись называется *звеном*, а структура из таких записей называется *списком или цепочкой*.

Лишь на самый первый элемент списка (голову) имеется отдельный указатель. Последний элемент списка никуда не указывает.



Описание списка

```
Type    ukazat= ^S;
        S= record
           Inf: integer;
           Next: ukazat;
           End;
```

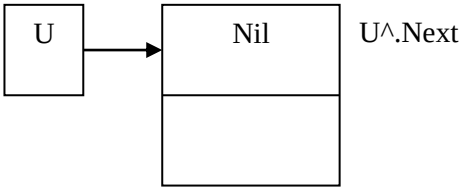
В Паскале существует основное правило: перед использованием какого-либо объекта он должен быть описан. Исключение сделано лишь для указателей, которые могут ссылаться на еще не объявленный тип.

Формирование списка

Чтобы список существовал, надо определить указатель на его начало.

```
Type    ukazat= ^S;
        S= record
           Inf: integer;
           Next: ukazat;
        End;
Var u,x: ukazat;
```

Создадим первый элемент списка:




```
New (u); {выделяем место в памяти}
u^. Next:= nil; {указатель пуст}
u^.Inf:=3;
```

Продолжим формирование списка. Для этого нужно добавить элемент либо в конец списка, либо в голову.

А) *Добавим элемент в голову списка*. Для этого необходимо выполнить последовательность действий:

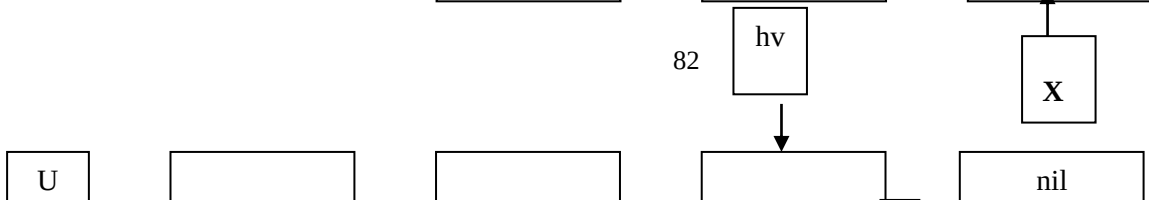
1) получить память для нового элемента;
2) поместить туда информацию;
3) присоединить элемент к голове списка.

```
New(x);
Readln(x^.Inf);
x^.Next:=u;
u:=x;
```

Б) *Добавление элемента в конец списка*. Для этого введем вспомогательную переменную, которая будет хранить адрес последнего элемента. Пусть это будет указатель с именем hv (хвост).

```
x:= hv;
```


```
New(x^.next); {выделяем память для следующего элемента}
```

x:= x^.next;

x^.next:= nil;

x^.inf:= 5;

hv:=x;

Просмотр списка

While u<> nil do

Begin

Writeln (u^.inf);

u:= u^.next;

end;

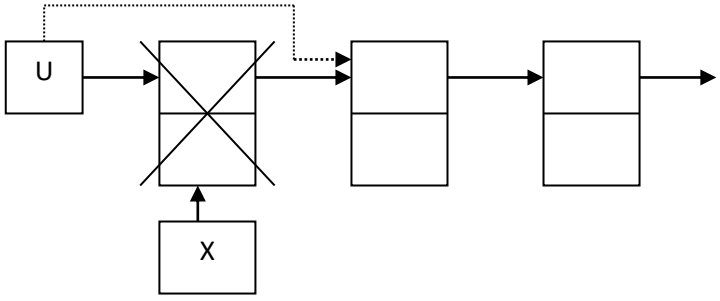
Удаление элемента из списка

А) *Удаление первого элемента*. Для этого во вспомогательном указателе запомним первый элемент, указатель на голову списка переключим на следующий элемент списка и освободим область динамической памяти, на которую указывает вспомогательный указатель.

x:= u;

u:= u^.next;

dispose(x);

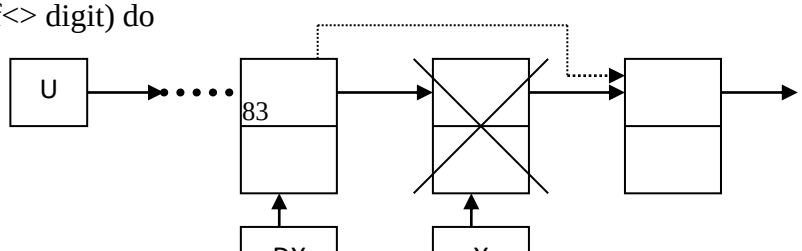


Б) *Удаление элемента из середины списка*. Для этого нужно знать адреса удаляемого элемента и элемента, стоящего перед ним. Допустим, что digit – это значение удаляемого элемента.

x:= u;

while (x<> nil) and (x^.inf<> digit) do

begin



```
dx:= x;
x:= x^.next;
end;
dx:= x^.next:
dispose(x);
```

В) *Удаление из конца списка*. Для этого нужно найти предпоследний элемент.

```
x:= u; dx:= u;
while x^.next<> nil do
begin
dx:= x; x:= x^.next;
end;
dx^.next:= nil;
dispose(x);
```

Прохождение списка

Важно уметь перебирать элементы списка, выполняя над ними какую-либо операцию. Пусть необходимо найти сумму элементов списка.

```
summa:= 0;
x:= u;
while x<> nil do
begin
summa:= summa+ x^.inf;
x:= x^.next;
end;
```

Динамические объекты сложной структуры

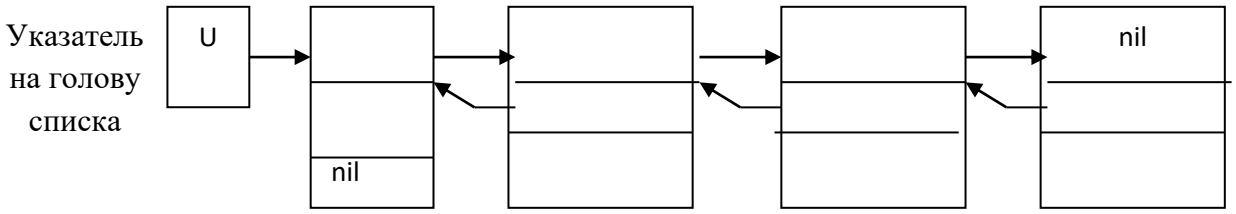
Использование однонаправленных списков при решении ряда задач может вызвать определенные трудности. Дело в том, что по однонаправленному списку можно двигаться только в одном направлении, от головы списка к последнему звену. Между тем нередко возникает необходимость произвести какую-либо операцию с элементом, предшествующим элементу с заданным свойством. Однако после нахождения элемента с данным свойством в однонаправленном списке у нас нет возможности получить удобный и быстрый способ доступа к предыдущему элементу.

Для устранения этого неудобства добавим в каждое звено списка еще одно поле, значением которого будет ссылка на предыдущее звено.

```
Type     ukazat= ^S;
```

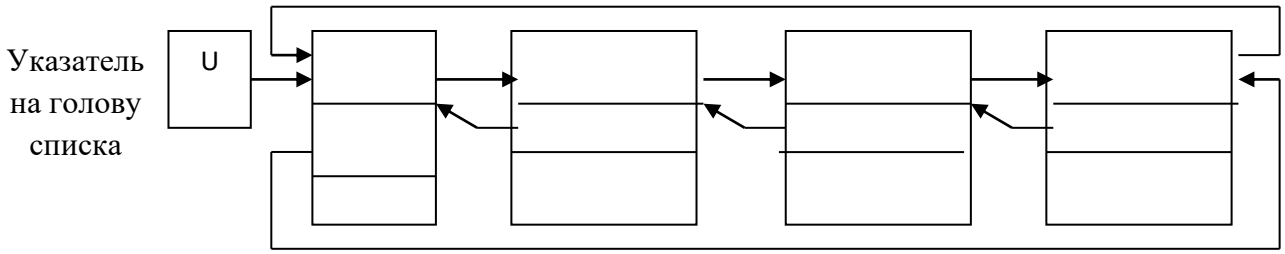
```
S= record
   Inf: integer;
   Next: ukazat;
   Pred: ukazat;
End;
```

Динамическая структура, состоящая из звеньев такого типа, называется ***двунаправленным списком***, который схематично можно изобразить так:



Наличие в каждом звене двунаправленного списка ссылки как на следующее, так и на предыдущее звено позволяет от каждого звена двигаться по списку в любом направлении. По аналогии с однонаправленным списком здесь есть заглавное звено. В поле Pred этого звена фигурирует пустая ссылка nil, свидетельствующая, что у заглавного звена нет предыдущего (так же, как у последнего нет следующего).

В программировании двунаправленные списки часто обобщают следующим образом: в качестве значения поля Next последнего звена принимают ссылку на заглавное звено, а в качестве значения поля Pred заглавного звена – ссылку на последнее звено:



Как видно, здесь список замыкается в своеобразное «кольцо»: двигаясь по ссылкам, можно от последнего звена переходить к заглавному звену, а при движении в обратном направлении – от заглавного звена переходить к последнему. Списки подобного рода называют ***кольцевыми списками***.

Существуют различные методы использования динамических списков:

1) ***Стек*** – особый вид списка, обращение к которому идет только через указатель на первый элемент. Если в стек нужно добавить элемент, то он добавляется впереди первого элемента, при этом указатель на начало стека переключается на новый элемент. Алго-

ритм работы со стеком характеризуется правилом: «последним пришел – первым вышел».

2) ***Очередь*** – это вид списка, имеющего два указателя на первый и последний элемент цепочки. Новые элементы записываются вслед за последним, а выборка элементов идет с первого. Этот алгоритм типа «первым пришел – первым вышел».
3) Возможно организовать списки с произвольным доступом к элементам. В этом случае необходим дополнительный указатель на текущий элемент.

Тема 2.8 Модули

Модуль — это набор ресурсов (функций, процедур, констант, переменных, типов и т.д.), разрабатываемых и хранимых независимо от использующих их программ. В отличие от внешних подпрограмм модуль может содержать достаточно большой набор процедур и функций, а также других ресурсов для разработки программ. Обычно каждый модуль содержит логически связанные между собой программные ресурсы.

В основе идеи модульности лежат принципы структурного программирования. Существуют стандартные модули Turbo Pascal, которые обычно описываются в литературе по данному языку.

Модуль имеет следующую структуру:

```
Unit <имя модуля>; {заголовок модуля}
Interface
   {интерфейсная часть}
Implementation
   {раздел реализации}
Begin
   {раздел инициализации модуля}
End.
```

После служебного слова Unit записывается имя модуля, которое (для удобства дальнейших действий) должно совпадать с именем файла, содержащего данный модуль. Поэтому (как принято в MS DOS) имя не должно содержать более 8 символов.

В разделе Interface объявляются все ресурсы, которые будут в дальнейшем доступны программисту при подключении модуля. Для подпрограмм здесь указывается лишь полный заголовок.

В разделе Implementation реализуются все подпрограммы, которые были ранее объявлены. Кроме того, здесь могут содержаться свои константы, переменные, типы, подпрограммы и т.д., которые носят вспомогательный характер и используются для написания основных подпрограмм. В отличие от ресурсов, объявленных в разделе Interface, все, что дополнительно объявляется в Implementation, уже не будет доступно при подключении модуля. При написании основных подпрограмм достаточно указать их имя (т.е. не нужно полностью переписывать весь заголовок), а затем записать тело подпрограммы.

Раздел инициализации (который часто отсутствует) содержит операторы, которые должны быть выполнены сразу же после запуска программы, использующей модуль.

Пример 1. *Модуль для работы с одномерными массивами до 100 целых чисел.*

```
{модуль описаний, глобальных для основной программы и всех модулей}
```

```
Unit Globals;
Interface
   const Len=100;
   type Vector = array[1..Len] of integer;
Implementation
End.
Unit Vectors;
Interface
   uses Globals;
   {находит максимальный элемент массива}
   function Max_V(A:Vector; n:byte):integer;
   {поэлементное сложение двух векторов}
   procedure Add_V(A,B:Vector; n:byte; var C:Vector);
   {скалярное произведение векторов}
   function Scal_V(A,B:Vector; n:byte):integer;
Implementation
   function Max_V; {заголовок без параметров}
   var i,max:integer;
   begin
      max:=A[1];
      for i:=2 to n do if A[i]>max then max:=A[i];
      Max_V:=max;
   end;
   procedure Add_V;
   var i:integer;
   begin
      for i:=1 to n do C[i]:=A[i]+B[i];
   end;
   function Scal_V(A,B:Vector; n:byte):integer;
   {заголовок из interface}
   var s:integer;       i:byte;
   begin
      s:=0;
      for i:=1 to n do s:=s+A[i]*B[i];
      Scal_V:=s;
   end;
End. {раздел инициализации модуля отсутствует}
```

Пример 2. *Разработать личную библиотеку, включив в нее процедуры:*

- *ввода элементов числовой матрицы размером N*N;*
- *транспонирования матрицы;*
- *вывода результирующей матрицы.*

В основной программе ввести размер матрицы N.

Начнем разработку модуля, который будет носить название Matrix. Программно это будет выглядеть так:

```
Unit Matrix;
 {Зарезервированное слово Unit служит для указания имени библиотеки. Это имя
 должно совпадать с именем PAS-файла библиотеки (т.е библиотека Matrix должна
 находиться с файле Matrix.Pas), а иначе компилятор даст ошибку при попытке
 использования такой библиотеки}

Interface
{Секция Interface содержит описания общедоступных типов данных, констант,
 процедур и функций. Т.е. все, что будет здесь находиться можно будет
 использовать при подключении данной библиотеки.}

Type
  TMatrix = array [1..10,1..10] of Integer;          { Квадратная матрица }
  procedure MatrInput  (Var m : TMatrix; n : Integer); { ввод матрицы }
  procedure MatrOutput (Var m : TMatrix; n : Integer); { вывод матрицы }
  procedure MatrTransp (Var m : TMatrix; n : Integer); { транспонирование }

Implementation
{Секция Implementation содержит реализацию тел процедур и функций, описанных
 в Interface. Также здесь могут содержаться типы данных, константы, процедуры
 и функции, необходимые для работы, но которые не будут видны программе при
 подключении библиотеки.}
{Процедура обмена местами двух элементов матрицы (x1,y1) и (x2,y2).
 Эта процедура используется при транспонировании матрицы, но ее
 нельзя вызвать при подключении библиотеки, т.к. она не объявлена
 в секции Interface.}
procedure Swap (Var m : TMatrix; x1,y1,x2,y2 : Integer);
 var
   temp : Integer;
 begin
   temp := m[x1,y1];
   m[x1,y1] := m[x2,y2];
   m[x2,y2] := temp;
 end;
{Ввод матрицы с клавиатуры. Параметры процедуры здесь не указаны, т.к. они есть в секции Interface }
procedure MatrInput;
 var
   i,j : Integer;
```

```
 begin
   for i:=1 to n do
    begin
     Write(i:3,'-я строка : ');
     for j:=1 to n do Read(m[i,j]);
     ReadLn;
    end;
 end;
{Транспонирование матрицы.}
procedure MatrTransp;
 var
   i,j : Integer;
 begin
   for i:=1 to n-1 do
    for j:=i+1 to n do
     Swap (m,i,j,j,i);
 end;
{Вывод матрицы на экран.}
procedure MatrOutput;
 var
   i,j : Integer;
 begin
   for i:=1 to n do
    begin
     Write(i:3,'-я строка : ');
     for j:=1 to n do Write (m[i,j]:4);
     WriteLn;
    end;
 end;
{Эта секция может использоваться для инициализации работы библиотеки.}
Begin
End.
```

Создание модуля закончено.

Теперь необходимо создать файл, который будет содержать текст основной программы, в которой будет подключаться разработанный выше модуль.

```
{Это отдельный файл, содержащий основную программу}
Uses
  Crt, { Библиотека стандартных процедур управления экраном и клавиатурой }
  Matrix; {Наш разработанный модуль-библиотека работы с квадратными матрицами (личная)}
Var
   : TMatrix;  {Объявляем матрицу - максимальный размер 10*10 }
```

```
  n : Integer;  { Размер матрицы }
Begin
  { Повторяем ввод размера, пока не будет введено корректное значение}
  repeat
    ClrScr;
    Write('Введите размер матрицы (1..10) : ');
    ReadLn(n);
  until (n >= 1) and (n <= 10);
WriteLn;
WriteLn('Введите матрицу размера',n,'*',n,'по строкам:');
MatrInput (m,n); {вызов процедуры ввода матрицы, определенной в модуле Matrix}
  {Транспонируем ее }
  MatrTransp (m,n); {вызов процедуры транспонирования матрицы, определенной в модуле
Matrix}
  { Выведем результат на экран }
  WriteLn;
  WriteLn('Транспонированная матрица :');
  MatrOutput (m,n); {вызов процедуры вывода матрицы, определенной в модуле Matrix}
End.
```

Как видно из примеров, для подключения модуля используется служебное слово USES, после чего указывается имя модуля и происходит это сразу же после заголовка программы. Если необходимо подключить несколько модулей, они перечисляются через запятую.

При использовании ресурсов модуля совсем не нужно знать, как работают его подпрограммы. Достаточно обладать информацией, как выглядят их заголовки, и какое действие эти подпрограммы выполняют. По такому принципу осуществляется работа со всеми стандартными модулями. Поэтому, если программист разрабатывает модули не только для личного пользования, ему необходимо сделать полное описание всех доступных при подключении ресурсов. В таком случае возможна полноценная работа с таким продуктом.

Если в программе, использующей модуль, имеются идентификаторы, совпадающие с точностью до символа с идентификаторами модуля, то они «перекрывают» соответствующие ресурсы модуля. Тем не менее, даже в такой ситуации доступ к этим ресурсам модуля может быть получен таким образом: <имя модуля>.<имя ресурса>.

Пример модуля, выводящего требуемую последовательность символов в заданное место экрана.

```
Unit Display;   {Содержит простую программу вывода на экран}
Interface {Интерфейсная часть}
{Заголовок процедуры}
Procedure WriteXY(X,Y: Integer; Message: String);
```

```
Implementation {Раздел реализации}
Uses Crt; {Подключение модуля Crt для выполнения процедуры GotoXY}
{Реализация процедуры}
Procedure WriteXY(X,Y: Integer; Message: String);
Begin
  If (X in [1..80]) And (Y in [1..25]) Then  Begin
 GoToXY(X, Y);
 Write(Message);
End;
End;
End. {Конец модуля}
```

Все модули хранятся в файлах с расширением TPU, имена которых совпадают с именами модулей. Для создания на диске файла с именем Display.tpu нужно выполнить компиляцию этого модуля (клавиши Alt+F9), предварительно установив опцию Compile/Distination/Disk.