Глава 2

Методы программирования

Вы уже не новички в программировании. В 10 классе каждый составил не менее десятка программ на Паскале. Что же нового о программировании вы узнаете, изучив данную главу? Почему она называется «Методы программирования»?

Цель изучения этой главы состоит в том, чтобы приблизить ваши знания и умения в области программирования к профессиональному уровню. Для всякого профессионала важно иметь системное представление о предмете своей деятельности. Под фразой «методы программирования» понимается совокупность способов, средств, технологий создания программ для компьютера. В данной главе системно излагаются некоторые из этих методов.

Как принято в нашем учебнике, мы немного заглянем в историю программирования, чтобы понять логику его развития, а потом вы будете осваивать основы современных методов программирования.

2.1. Эволюция программирования

Программирование для компьютера — процесс создания программ управления работой компьютера.

Машинно-ориентированное программирование

С изобретением программно управляемых вычислительных машин появилась новая профессия — программист. Мы уже рассказывали, что первым в истории программистом была Ада Лавлейс, работавшая вместе с Чарльзом Беббиджем, она разрабатывала программы управления его Аналитической машиной. Но массовой профессия программиста стала только с изобретением электронных вычислительных машин — ЭВМ.

На ламповых ЭВМ первого поколения программисты составляли свои программы, используя непосредственно команды процессора. При этом программисту приходилось распределять ячейки памяти под данные и под команды программы. Нужно было знать систему команд процессора и коды всех команд. Исходные данные и команды представлялись в форме двоичного кода, т. е. непосредственно в том виде, в котором они хранились в памяти ЭВМ. Для сокращения записи программ на специальных бланках обычно использовали двоично-восьмеричный или двоично-шест-

надцатеричный код. Вот пример команды программы для одного из компьютеров первого поколения.

	Адрес команды	Код операции	1-й адрес	2-й адрес	3-й адрес
Шестнадцатеричный код	28	02	C0	C4	D8
Двоичный код	0010 1000	00000 0010	1100 0000	1100 0100	1101 1000

Такая команда называется трёхадресной. Код 02_{16} относится к команде сложения. 1-й и 2-й адреса — это адреса ячеек ОЗУ, в которых хранятся слагаемые, 3-й адрес — адрес ячейки, куда заносится сумма. Сама команда хранится в ячейке ОЗУ с адресом 28_{16}.

Программирование в машинных кодах представляло собой сложный процесс. По этой причине производительность работы программистов была довольно низкой. В 1950-х годах возникает направление, которое получило название «автоматизация программирования». Основная его цель: создание средств, облегчающих и ускоряющих процесс создания программы для ЭВМ. Появляются первые языки программирования.

Первыми **языками программирования** были машинно-ориентированные **автокоды**. Позднее за языками такого уровня закрепилось название **ассемблеры**. Первоначально ассемблером называли программу-переводчик с языка ассемблера в машинные команды. Позднее и сам язык ассемблера стали называть именем «ассемблер». Программирование на ассемблере снимает с программиста заботу о распределении памяти под данные и команды программы. Программист также не должен помнить внутренние коды всех команд процессора. Вот пример той же команды сложения на ассемблере (автокоде):

```
ADD a, b, c
```

Слово `ADD` обозначает команду «сложить», `a` и `b` — имена переменных-слагаемых, `c` — переменная, куда помещается результат.

Язык ассемблер называется машинно-ориентированным по той причине, что для каждой команды процессора существует свой аналог команды на ассемблере. Поскольку разные типы ЭВМ имели разные системы команд процессора, то и ассемблеры у них различались. Современные ассемблеры точно так же ориентированы на определенные типы процессоров. Позже появились так называемые *макроассемблеры*, в языке которых существуют макрокоманды, соответствующие сериям команд (подпрограммам) на языке процессора.

Составление программы на ассемблере проще, чем на языке команд процессора. Работу по распределению памяти под данные и команды, перевод команд ассемблера в машинные команды берет на себя специальная системная программа — **транслятор**.

Из машинной ориентированности программ на ассемблере следует, что такие программы нельзя переносить для исполнения на другие типы ЭВМ с другой системой команд процессора. Эта проблема создавала серьезные ограничения для прикладных программистов. Кроме того, само програм-

мирование на ассемблере является достаточно сложным для массового освоения, что ограничивало использование ЭВМ в прикладных областях.

Языки программирования высокого уровня

Следующим этапом развития программирования стало создание **языков программирования высокого уровня** (ЯПВУ). Примеры ЯПВУ: Паскаль, Бейсик, Фортран. Для каждого языка существует машинно-независимый стандарт. Возможность программирования на данном ЯПВУ зависит от наличия на вашем компьютере транслятора с этого языка. Трансляторы для каждого типа компьютеров составляют системные программисты.

Текст программы на ЯПВУ по своей форме ближе к естественным языкам (чаще всего — английскому), к языку математики. Та же команда сложения двух величин на ЯПВУ похожа на привычную форму математического равенства:

`c := a + b` (на Паскале);

`c = a + b` (на Фортране, Бейсике, Си).

Гораздо проще освоить программирование на языке высокого уровня, чем на ассемблере. Поэтому с появлением ЯПВУ значительно возросло число прикладных программистов, расширилось применение ЭВМ во многих областях.

Начиная с середины XX века и до нашего времени были созданы сотни языков программирования высокого уровня. Но распространенными и популярными из них стали не все. Одним из долгожителей в семействе ЯПВУ является язык Фортран. Fortran — сокращение словосочетания *for*mula *tran*slator — транслятор формул. Первая версия Фортрана была создана в 1954 году. Во времена ЭВМ второго и третьего поколений была популярна версия Фортран-IV. Фортран создавался как специализированный язык для математических расчетов, используемых в науке и технике. И в наше время этот язык, в стандарте Фортран-90 (и в последующих его модификациях Фортран-95, Фортран-2003), остаётся основным языком программирования для расчётов в области физико-технических проблем.

К числу первых ЯПВУ, созданных в 1950-х годах, относятся Кобол (создан в США) и Алгол (в Европе). Алгол, как и Фортран, был ориентирован на научно-технические расчеты математического характера. Кобол — язык для программирования экономических задач. В Коболе, по сравнению с двумя другими названными языками, слабее развиты математические средства, но зато хорошо представлены средства обработки текстов, организации вывода данных в форме требуемого документа. Для первых ЯПВУ предметная ориентация языков была характерной чертой.

Большое количество языков программирования появилось в 1960–1970-х годах. В 1965 году в Дартмутском университете был разработан язык Бейсик. По замыслу авторов это простой язык, легко изучаемый, предназначенный для программирования несложных расчетных задач. Наибольшее распространение Бейсик получил с появлением микро-ЭВМ и персональных компьютеров.

Значительным событием в истории языков программирования стало создание в 1969 году языка Паскаль. Его автор — швейцарский профессор Никлаус Вирт разрабатывал Паскаль как учебный язык структурного программирования.

Наибольший успех в распространении языка Паскаль обеспечили персональные компьютеры. Фирма Borland International, Inc (США) разработала систему программирования Turbo Pascal (Турбо Паскаль) для ПК. Турбо Паскаль — это не только язык и транслятор с него, но еще и **интегрированная среда программирования**, дающая пользователю возможность удобно работать на Паскале: вводить и редактировать текст программы, искать синтаксические ошибки, пользоваться библиотеками подпрограмм и модулей, работать с файлами и пр. Турбо Паскаль вышел за рамки учебного предназначения и стал языком профессионального программирования с универсальными возможностями. Паскаль стал источником многих основных современных языков программирования, например таких, как Ада, Модула-2 и др.

Модула-2 — это еще один язык, предложенный Виртом, являющийся развитием языка Паскаль и содержащий средства для создания больших программ.

Язык программирования Си (английское название — С) появился практически одновременно с Паскалем. Он создавался как инструментальный язык для разработки операционных систем, трансляторов, баз данных и других системных и прикладных программ. Хотя Си и является языком высокого уровня, однако в нем заложены возможности непосредственного обращения к некоторым машинным командам, к определенным участкам памяти компьютера, что ранее было возможно только в ассемблере. С появлением Си многие системные программисты перешли с ассемблера на Си. Дальнейшее развитие Си привело к созданию языка объектно-ориентированного программирования Си++.

Парадигмы программирования

Слово «парадигма» применительно к программированию означает определённый общепринятый подход к организации вычислений на компьютере. Парадигма определяет систему базовых понятий, на основе которой происходит программирование.

Языки программирования, о которых рассказывалось выше, основаны на **процедурной парадигме программирования**. Алгоритм, реализованный на процедурном языке, основывается на представлениях о фон-неймановской архитектуре компьютера. Базовым понятием является понятие величины, хранимой в памяти компьютера, базовой операцией — операция присваивания. Решение задачи на компьютере происходит путем изменения состояния памяти: в начале в память помещаются исходные данные, в конце получаются результаты. Программа — это описание процедуры перехода памяти из начального состояния в конечное в процессе выполнения последовательности команд. Мышление программиста, работающего в рамках процедурной парадигмы, направлено на сведéние решения задачи к последовательности команд, которые умеет выполнять процессор компьютера.

На другом подходе к программированию основана **функциональная парадигма программирования**. Наиболее известным языком, реализующим функциональную парадигму, является ЛИСП. Первое упоминание о ЛИСПе относится к 1958 году. Этот язык создан на основе понятия рекурсивной функции (это понятие будет раскрыто в разделе 2.3). В теоретическом программировании доказано, что любой алгоритм может быть описан с помощью некоторого набора рекурсивных функций. Поэтому ЛИСП, по сути, является универсальным языком. С его помощью на компьютере можно моделировать достаточно сложные процессы, в частности, интеллектуальную деятельность людей. Помимо ЛИСПа существуют и другие языки функционального программирования: Haskell, Scheme и др.

Логическая парадигма программирования реализована в языке Пролог. Пролог был разработан во Франции в 1972 году. Его, как и ЛИСП, относят к языкам искусственного интеллекта. Пролог основан на аппарате математической логики, отсюда название парадигмы. На Прологе строится база знаний в определённой предметной области, состоящая из совокупности фактов и правил. Для решения некоторой задачи формулируется запрос к базе знаний, который называется целью. Реализация цели (ответ на запрос) происходит через заложенный в интерпретатор Пролога алгоритм, который называется механизмом вывода. Используя Пролог, можно решать логические задачи, аналогичные тем, которые были рассмотрены в § 1.6.4 учебника для 10 класса. Следует отметить, что с помощью Пролога можно решать лишь ограниченный круг задач искусственного интеллекта. К языкам логического программирования также относятся Planner, Mercury, Fril и др.

Объектно-ориентированная парадигма программирования основана на концепции объектов и классов. Всякий объект характеризуется набором свойств и действий, которые с ним могут быть выполнены и которые он может выполнять. Объект относится к определенному классу. Объектно-ориентированное программирование сводится к выстраиванию иерархий классов, описаний объектов и их взаимодействий, программной реализации различных действий над объектами. Первым языком ООП был Симула, разработанный в 1967 году. Позже элементы ООП стали внедряться в процедурные языки, в том числе в Турбо Паскаль. Наиболее популярными средствами объектно-ориентированного программирования являются языки СИ++, Java. В нашем учебнике подробнее об ООП рассказывается в разделе 2.4.

Методологии и технологии программирования

На первых ЭВМ с «тесной» памятью и небольшим быстродействием основным показателем качества программы была её экономичность по занимаемой памяти и времени счёта. Чем программа получалась короче, тем класс программиста считался выше.

С ростом памяти и быстродействия ЭВМ, с совершенствованием языков программирования и трансляторов с этих языков проблема экономичности программы становится менее острой. Все более важной качественной характеристикой программ становится их простота, наглядность, надёжность. С появлением машин третьего поколения эти качества стали основными.

Уже в 1960-х годах программирование стало достаточно массовой профессиональной деятельностью. Возникают компании (фирмы) по разработке программ. Актуальной становится задача разработки общепринятой методологии программирования, повышающей производительность работы программистов и, что самое главное, качество программных продуктов. Основной качественный показатель программы — её работоспособность, отсутствие ошибок.

Методология программирования — это совокупность определённых способов написания, отладки и сопровождения программ. Первая наиболее известная и распространенная методология программирования получила название «**структурное программирование**».

Появление структурного программирования связано с именами Эдсгера Дейкстры и Чарльза Хоара. Начиная с 1960-х годов стали появляться языки структурного программирования. Первым из них был Алгол-60, разработанный Дейкстрой, затем был создан Паскаль. Другие, первоначально «не структурные» языки стали также приобретать «структурные свойства» (Турбо Бейсик, Фортран-77 и пр.). Структурное программирование до настоящего времени остается важнейшей методологией программирования. Соблюдение его принципов позволяет программисту составлять ясные, безошибочные, надёжные программы.

Технология — это массовый способ производства какого-то продукта, гарантирующий устойчивый результат. Технология предполагает наличие определённых средств, инструментов производства. В 1990-х годах с развитием объектно-ориентированной парадигмы программирования, а также средств графического интерфейса на персональных компьютерах, возникает новая **технология программирования — визуальное программирование**. Визуальная технология программирования позволяет программисту легко и быстро строить наглядный графический интерфейс для своих программ на основе стандартного набора шаблонов, графически отображаемых на экране объектов. О технологии визуального программирования в системе Delphi рассказывается в разделе 2.4 нашего учебника.

О профессиях: профессии, связанные с программированием

В перечне современных профессий имеется профессия «**программист**». Очень велико разнообразие профессиональной деятельности программистов. Принято всё это разнообразие делить на две группы: системное программирование и прикладное программирование. Исторически профессия системного программиста берёт свое начало с разработки трансляторов с языков программирования. Возникает разделение программистов на системных и прикладных. **Прикладные программисты** используют компьютер для решения прикладных задач из различных предметных областей: математики, физики, экономики и др. А работа **системных программистов** состоит в том, чтобы облегчить, упростить работу прикладным программистам, повысить эффективность использования компьютера для решения прикладных задач. Системные программисты разрабатывают не только трансляторы, но и операционные системы, утилиты, программное обеспечение для функционирования компьютерных сетей, информационных систем и пр.

Система основных понятий

<table>
<tr><th colspan="4">Что такое программирование</th></tr>
<tr><td colspan="4">Программирование для компьютера — процесс создания программ управления работой компьютера (создание программного обеспечения)</td></tr>
<tr><th colspan="4">Машинно-ориентированное программирование</th></tr>
<tr><td colspan="2">Программирование в машинных командах: требует от программиста распределения памяти под данные, знания кодов операций и форматов команд</td><td colspan="2">Программирование на автокодах (ассемблерах): символическое обозначение переменных, мнемоническое обозначение кодов операций. Нуждается в трансляции</td></tr>
<tr><td colspan="4">Машинно-независимое программирование — на языках высокого уровня (ЯПВУ)</td></tr>
<tr><th colspan="4">Парадигмы и языки программирования</th></tr>
<tr><td>Процедурное</td><td>Функциональное</td><td>Логическое</td><td>Объектно-ориентированное</td></tr>
<tr><td>Фортран, Алгол, Паскаль, Бейсик, Си и др.</td><td>ЛИСП, Haskell, Scheme и др.</td><td>Пролог, Planner, Mercury, Fril и др.</td><td>Симула, Си++, Java и др.</td></tr>
<tr><th colspan="4">Методологии и технологии программирования</th></tr>
<tr><td colspan="2">Структурная методология процедурного программирования</td><td colspan="2">Визуальная технология объектно-ориентированного программирования</td></tr>
</table>

Вопросы и задания

1. Что такое программирование?
2. В какой форме составлялись программы для первых ЭВМ?
3. Почему языки автокоды (ассемблеры) называются машинно-ориентированными языками программирования?
4. Назовите основные процедурные языки программирования в хронологической последовательности их создания.
5. Что такое парадигма программирования?
6. Назовите основные парадигмы программирования и их отличия друг от друга.
7. Что такое структурное программирование?
8. Что такое визуальное программирование?

2.2. Структурное программирование

2.2.1 Паскаль — язык структурного программирования

История Паскаля

Язык программирования Паскаль был создан швейцарским профессором Никлаусом Виртом в 1969 году как язык для обучения студентов структурной методике программирования. Язык получил свое название в честь Блеза Паскаля — изобретателя первого вычислительного механического устройства. Позднее фирма Borland International, Inc (США) разработала систему программирования Турбо Паскаль для персональных компьютеров, которая вышла за рамки учебного применения и стала использоваться для научных и производственных целей. В Турбо Паскаль были внесены некоторые дополнения к базовому стандарту Паскаля, описанного Виртом. Со временем язык развивался. Начиная с версии 5.5, в Турбо Паскаль вводятся средства поддержки объектно-ориентированного программирования. В дальнейшем это привело к созданию Object Pascal — языка с возможностями объектно-ориентированного программирования (ООП). В начале 1990-х годов объединение элементов ООП в Паскале с визуальной технологией программирования привело к созданию системы программирования Delphi.

У вас уже есть опыт программирования на Паскале, который вы получили, изучая информатику в 9 и 10 классах. Обучение там проводилось на примерах программ. Правила языка излагались по мере их использования в рассматриваемых программах. Целью данной главы является систематизация и расширение ваших знаний о языке Паскаль. Как это было и раньше, язык описывается в рамках расширенного стандарта Паскаля, соответствующего диалекту Турбо Паскаль.

В начале данной главы учебника мы будем рассматривать методику структурного программирования на Паскале. Позднее вы познакомитесь с основами ООП и визуального программирования на Delphi.

Структура процедурных языков программирования высокого уровня

Во всяком языке программирования определены способы организации данных и способы организаций действий над данными. Кроме того, существует понятие «элементы языка», включающее в себя множество символов (алфавит), служебных слов и других изобразительных средств языка программирования. Несмотря на разнообразие процедурных языков, их изучение происходит приблизительно по одной схеме. Это связано с общностью структуры различных процедурных языков программирования высокого уровня, которая схематически отражена на рис. 2.1.

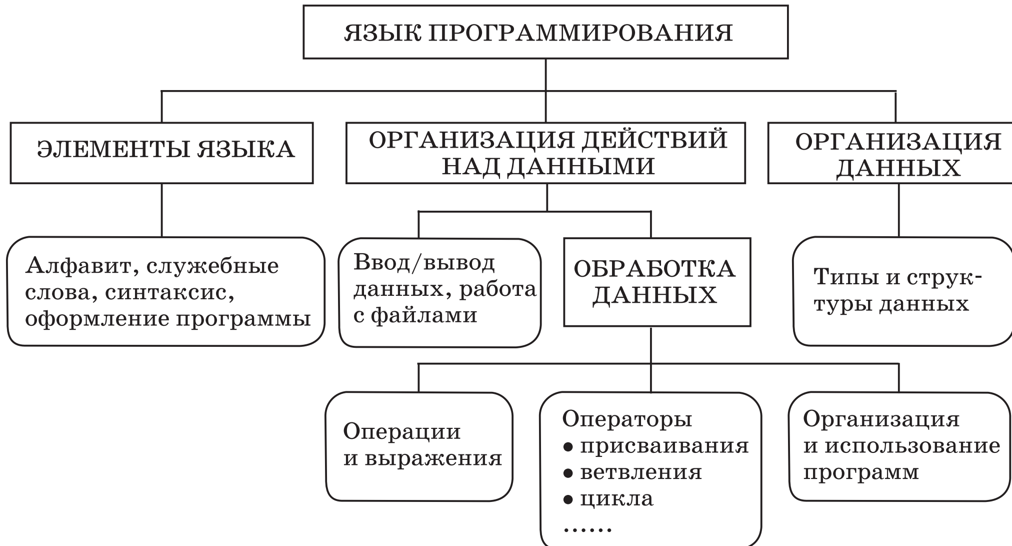


Рис. 2.1. Структура процедурного ЯПВУ

Всякий язык программирования образуют *три его основные составляющие*: *алфавит, синтаксис* и *семантика*. **Алфавит** — это множество символов, допустимых в записи текстов программ. **Синтаксис** — это правописание языковых конструкций (имён, констант, выражений, операторов и пр.). **Семантика** — это смысловое содержание языковой конструкции.

Соблюдение правил в языке программирования должно быть более строгим, чем в разговорном языке. Человеческая речь содержит значительное количество избыточной информации. Не расслышав какое-то слово, можно понять смысл фразы в целом. Слушающий или читающий человек может додумать, дополнить, исправить ошибки в воспринимаемом тексте. Компьютер же — автомат, воспринимающий всё буквально. В текстах программ нет избыточности, компьютер сам не исправит даже очевидной (с точки зрения человека) ошибки. Он может лишь указать на место, которое «не понял», и вывести замечание о предполагаемом характере ошибки. Исправить же ошибку должен программист.

Вы уже знакомы со многими элементами Паскаля, составляли программы на Паскале. Поэтому целью данного раздела нашего курса является упорядочение имеющихся у вас знаний языка и их дальнейшее развитие, необходимое для того, чтобы научиться решать задачи новых типов.

Структура программы на Паскале

По определению стандартного Паскаля программа состоит из **заголовка** программы и **тела** программы **(блока)**, за которым следует *точка* — признак конца программы. В свою очередь блок содержит **разделы описа-**

ний (меток, констант, типов, переменных, подпрограмм) и **раздел операторов**.

```
Program <имя программы>;
Label <раздел меток>;
Const <раздел констант>;
Type <раздел типов>;
Var <раздел переменных>;
Procedure (Function) <раздел подпрограмм>;
Begin
  <раздел операторов>
End.
```

Раздел операторов имеется в любой программе и является основным. Предшествующие разделы носят характер описаний и не все обязательно присутствуют в каждой программе.

В Турбо Паскале, в отличие от базового стандарта Паскаля, возможно:

- отсутствие заголовка программы;
- разделы **Const**, **Type**, **Var**, **Label** могут следовать друг за другом в любом порядке и повторяться в разделе описаний сколько угодно раз.

2.2.2 Элементы языка и типы данных

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

Латинские буквы: от *A* до *Z* (заглавные) и от *a* до *z* (строчные).

Цифры: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Специальные символы: + — * / = < > [ ] . , ( ) : ; { } ^ @ $ #.

Следующие комбинации специальных символов являются едиными символами (их нельзя разделять пробелами):

:=	знак присваивания;	<=	меньше или равно;
>=	больше или равно;	(* *)	ограничители комментариев (наряду с { });
<>	не равно;	(. .)	эквивалент [ ].

Пробелы — символ пробела (код ASCII 32) и все управляющие символы кода ASCII (от 0 до 31).

Служебные слова. К спецсимволам относятся и служебные слова, смысл которых определён однозначно. Служебные слова не могут быть использованы для других целей. С точки зрения языка, они являются едиными элементами алфавита. Вот некоторые служебные слова: **program**, **var**, **array**, **if**, **do**, **while** и др.

Идентификаторы. Идентификатором называется символическое имя определённого программного объекта. Такими объектами являются: имена констант, переменных, типов данных, процедур и функций, программ. *Идентификатор — это любая последовательность букв и цифр, начинающаяся с буквы.* К буквам приравнивается также знак подчёркивания. Длина идентификатора может быть произвольной, но значащими являются только первые 63 символа.

Комментарии. Следующие конструкции представляют собой комментарии и поэтому пропускаются компилятором:

{любой текст, не содержащий символ "фигурная скобка" }
(любой текст, не содержащий символы "звёздочка, круглая скобка" *)*
//последующий текст до конца строки

Буквы русского алфавита употребляются только в комментариях, символьных и текстовых константах.

Концепция типов данных в Паскале

Концепция типов данных является одной из центральных в любом языке программирования. *С* **типом величины** *связаны три её свойства: форма внутреннего представления, множество принимаемых значений и множество допустимых операций.*

Паскаль характеризуется большим разнообразием типов данных, отражённом на рис. 2.2.

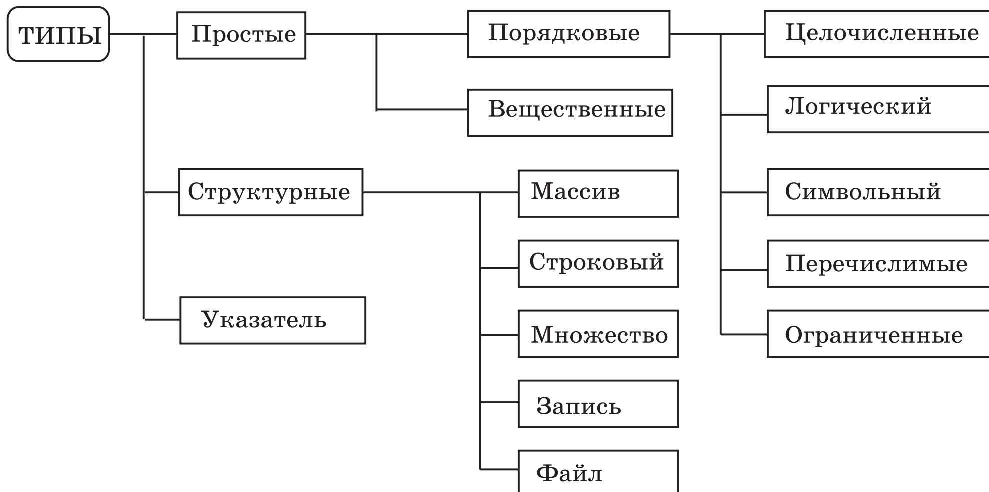


Рис. 2.2. Система типов данных Паскаля

Тип данных называется *порядковым*, если он состоит из счётного количества значений, которые можно пронумеровать. Отсюда следует, что на этом множестве значений существуют понятия «следующий» и «предыдущий».

В стандартном Паскале Вирта отсутствовал строковый тип. Он был внесён в Турбо Паскаль. Кроме того, в Турбо Паскале целые и вещественные — это группы типов. В старших версиях Турбо Паскаля появился процедурный тип и тип «объект».

Каждый тип имеет свой идентификатор. В таблице 2.1 представлена информация о простых типах данных, определённых в Турбо Паскале и последующих диалектах языка. Для вещественных типов в скобках указано количество сохраняемых значащих цифр мантиссы в десятичном представлении числа.

Таблица 2.1. **Типы данных**

Имя типа	Длина в байтах	Диапазон (множество) значений	Десятичных цифр в мантиссе
Целочисленные типы			
`integer`	2	–32768..32767	
`byte`	1	0..255	
`word`	2	0..65535	
`shortint`	1	–128..127	
`longint`	4	–2147483648..2147483647	
Вещественные типы			
`real`	6	$2{,}9 \cdot 10^{-39} - 1{,}7 \cdot 10^{38}$	(11–12)
`single`	4	$1{,}5 \cdot 10^{-45} - 3{,}4 \cdot 10^{38}$	(7–8)
`double`	8	$5 \cdot 10^{-324} - 1{,}7 \cdot 10^{308}$	(15–16)
`extended`	10	$3{,}4 \cdot 10^{-4932} - 1{,}1 \cdot 10^{4932}$	(19–20)
Логический тип			
`boolean`	1	`True`, `False`	
Символьный тип			
`char`	1	все символы кода ASCII	

Типы пользователя. Одна из принципиальных возможностей Паскаля состоит в том, что пользователю разрешается определять свои типы данных. Типы пользователя всегда базируются на стандартных типах данных Паскаля. Для описания типов пользователя в Паскале существует раздел типов, начинающийся со служебного слова **Type**. К простым типам пользователя относятся перечисляемый и интервальный типы данных.

Перечислимый тип задаётся непосредственно перечислением (списком) всех значений, которые может принимать переменная данного типа.

```
Type <имя типа> = (<список значений>)
```

Определённое таким образом имя типа затем используется для описания переменных. Например:

```
Type Gaz = (C, O, N, F);
     Metal = (Fe, Co, Na, Cu, Zn);
Var  G1, G2, G3: Gaz;
     Met1, Met2: Metall;
     Day: (Sun,Mon,Tue,Wed,Thu,Fri,Sat);
```

Здесь Gaz и Metal — имена перечислимых типов, которые ставятся в соответствие переменным G1, G2, G3 и Met1, Met2. Переменной Day назначается перечислимый тип, которому не присвоено имени.

Значения, входящие в перечислимый тип, являются *константами*. Действия над ними подчиняются правилам, применимым к константам. Каждое значение в перечислимом типе занимает в памяти 2 байта, поэтому число значений этого типа не должно превышать 65 535.

Перечислимый тип — упорядоченное множество. Его элементы пронумерованы, начиная от 0 в порядке следования в описании.

В программе, в которой присутствует данное выше описание переменной Day, возможен такой фрагмент:

```
if Day=Sun then write('Ура! Сегодня выходной!');
```

Ограниченный тип задаётся как упорядоченное ограниченное подмножество некоторого порядкового типа:

```
<константа 1> .. <константа 2>
```

Порядковый номер первой константы не должен превышать номера второй константы в соответствующем базовом типе.

При исполнении программы автоматически контролируется принадлежность значений переменной ограниченного типа установленному диапазону. При выходе из диапазона исполнение программы прерывается.

Пример

```
Type Numbers = 1..31;
     Alf= 'A'..'Z';
Var Data: Numbers;
    Bukva: Alf;
```

Структурные типы. Особенностью Паскаля является то, что в нем структуры данных рассматриваются как типы — структурные типы данных. Одна величина *простого типа* имеет одно значение: целое число, вещественное число, символ и пр. Одна величина *структурного типа* имеет множество значений; примеры — числовой массив, символьная строка и пр.

Автор Паскаля Вирт придавал большое значение разнообразию типов данных в языке программирования. В своей книге «Алгоритмы и структуры данных» он подчёркивает зависимость алгоритма решения задачи от способа организации данных в программе. Удачно выбранный способ организации данных упрощает алгоритм решения задачи.

Система основных понятий

<table>
<tr><th colspan="6">Базовые понятия Паскаля</th></tr>
<tr><td colspan="6">Паскаль — процедурный язык структурного программирования.
Этапы развития: стандартный Паскаль (Н. Вирт), Турбо Паскаль, Object Pascal, Delphi</td></tr>
<tr><th colspan="6">Состав программы на Паскале</th></tr>
<tr><th colspan="2">Элементы языка</th><th colspan="2">Разделы программы</th><th colspan="2">Комментарии</th></tr>
<tr><td colspan="2">Алфавит, служебные слова, идентификаторы</td><td colspan="2">Заголовок программы
Разделы описаний
Раздел операторов</td><td colspan="2">Поясняющие тексты, которые пропускаются компилятором</td></tr>
<tr><th colspan="6">Типы данных</th></tr>
<tr><td colspan="3">Стандартные (целочисленные, вещественные, логический, символьный)</td><td colspan="3">Определяемые в программе:
перечислимые, ограниченные</td></tr>
<tr><th colspan="3">Простые типы</th><th colspan="3">Структурные типы</th></tr>
<tr><td colspan="3">Одна величина — одно значение</td><td colspan="3">Одна величина — множество значений (массивы, строки, записи, множества, файлы)</td></tr>
</table>

Вопросы и задания

1. В какой парадигме программирования реализован язык Паскаль?
2. Входят ли в алфавит Паскаля русские буквы? Для чего их можно использовать?
3. Что такое идентификатор? Каковы правила задания идентификаторов?
4. Чем различаются разные типы данных из группы целочисленных типов?
5. Чем различаются разные типы данных из группы вещественных типов?
6. В чем разница между простыми и структурными типами?
7. Что такое перечислимый и ограниченный типы данных?

2.2.3 Операции, функции, выражения

Арифметические операции

К числовым типам данных относятся группы вещественных и целочисленных типов. К ним применимы арифметические операции и операции отношений.

Операции над данными бывают унарными (применимые к одному операнду) и бинарными (применимые к двум операндам). **Унарная арифметическая операция** в Паскале одна. Это операция изменения знака. Её формат:

–<величина>.

Бинарные арифметические операции стандартного Паскаля описаны в табл. 2.2. В ней символ I обозначает целые типы, символ R — вещественные типы.

Таблица 2.2. **Бинарные операции Паскаля**

Знак	Выражение	Типы операндов	Тип результатов	Операция
+	A + B	R, R I, I I, R; R, I	R I R	Сложение
-	A – B	R, R I, I I, R; R, I	R I R	Вычитание
*	A * B	R, R I, I I, R; R, I	R I R	Умножение
/	A/B	R, R I, I I, R; R, I	R R R	Вещественное деление
div	A div B	I, I	I	Целочисленне деление
mod	A mod B	I, I	I	Остаток от целочисленного деления

Стандартные функции и процедуры

В Паскале существует большое количество стандартных функций и процедур, к которым программист может обращаться в своих программах. Наиболее часто используются математические функции, например: `sqrt(x)` — корень квадратный, `abs(x)` — абсолютная величина, `sin(x)` и др. Часто используемые стандартные процедуры: `read(...)` — процедура ввода, `write(...)` — процедура вывода данных.

Стандартные функции и процедуры являются внешними подпрограммами по отношению к вызывающей их программе. Они объединены в **модули**, которые подключаются к основной программе и становятся доступными для использования. Наиболее часто используемые подпрограммы объединены в модуль под названием `SYSTEM`. Этот модуль подключается к программе автоматически.

Таблица 2.3 содержит описания стандартных математических функций Паскаля.

Таблица 2.3. **Стандартные математические функции Паскаля**

Обращение	Тип аргумента	Тип результата	Функция
`Pi`	—	R	Число π = 3,1415926536E+00
`abs(x)`	I, R	I, R	Модуль аргумента
`arctan(x)`	I, R	R	Арктангенс (в радианах)
`cos(x)`	I, R	R	Косинус (в радианах)
`exp(x)`	I, R	R	e^x — экспонента
`frac(x)`	I, R	R	Дробная часть x
`int(x)`	I, R	R	Целая часть x
`ln(x)`	I, R	R	Натуральный логарифм
`random`	—	R	Псевдослучайное число в интервале [0, 1)
`random(x)`	I	I	Псевдослучайное число в интервале [0, x)
`round(x)`	R	I	Округление до ближайшего целого
`sin(x)`	I, R	R	Синус (в радианах)
`sqr(x)`	I, R	I, R	Квадрат x
`sqrt(x)`	I, R	R	Корень квадратный
`trunc(x)`	R	I	Ближайшее целое, не превышающее x по модулю

Для подключения других модулей необходимо в начале программы (после заголовка) записать строку:

```
Uses <имя модуля>
```

Для управления символьным выводом на экран используется стандартный модуль `CRT`. К программе он подключается командой:

```
Uses CRT
```

В дальнейшем из этого модуля мы будем часто использовать *процедуру очистки экрана* для символьного вывода, обращение к которой производится оператором: `ClrScr`.

Арифметические выражения

Арифметическое выражение задаёт порядок выполнения действий над числовыми величинами. Арифметические выражения содержат числовые константы и переменные, арифметические операции, функции, круглые скобки. Одна константа или одна переменная — простейшая форма арифметического выражения.

Например, рассмотрим математическое выражение:

$$\frac{2a + \sqrt{0{,}5\sin(x+y)}}{0{,}2c - \ln(x-y)}.$$

На Паскале оно выглядит так:

```
(2*A + Sqrt(0.5*sin(X + Y)))/(0.2*C — Ln(X - Y))
```

Для того чтобы правильно записывать арифметические выражения, нужно соблюдать следующие правила.

1. Все символы пишутся в строчку на одном уровне. Проставляются все знаки операций (нельзя пропускать знак *).
2. Не допускаются два следующих подряд знака операций. (Нельзя: `A + - B`; можно: `A + (-B)`.)
3. Операции с более высоким приоритетом выполняются раньше операций с меньшим приоритетом. Порядок убывания приоритетов:
   вычисление функции;
   унарная операция смены знака (-);
   `*`, `/`, `div`, `mod`;
   `+`, `-`.
4. Несколько записанных подряд операций одинакового приоритета выполняются последовательно слева направо.
5. Часть выражения, заключённая в скобки, вычисляется в первую очередь. (Например, в выражении `(A+B)*(C-D)` умножение производится после сложения и вычитания.)

Не следует записывать выражения, не имеющие математического смысла, например: деление на нуль, логарифм отрицательного числа и т. п.

Пример. Цифрами сверху указан порядок выполнения операций:

```
   1   7   4  5  3      6   2     12  11 10   8      9
(1+y)*(2*x+sqrt(y)-(x+y))/(y+1/(sqr(x)-4))
```

Данное арифметическое выражение (на Паскале) соответствует следующему математическому выражению:

$$(1+y)\frac{2x + \sqrt{y} - (x+y)}{y + \dfrac{1}{x^2 - 4}}.$$

В Паскале нет операции или стандартной функции возведения числа в произвольную степень. Для вычисления x^y рекомендуется поступать следующим образом:

а) если y — целое значение, то его степень вычисляется через умножение; например, $x^3 \to x * x * x$; большие степени следует вычислять умножением в цикле;

б) если y — вещественное значение, то используется следующая математическая формула: $x^y = e^{y\ln(x)}$. На Паскале получим арифметическое выражение:

```
Exp(Y * Ln(x))
```

Очевидно, что при вещественном y не допускается нулевое или отрицательное значение x. Для целого y такого ограничения нет.

Пример

$$\sqrt[3]{a+1} = (a+1)^{\frac{1}{3}}$$

На Паскале это выражение выглядит так:

```
Exp(1/3 * Ln(A + 1))
```

Выражение имеет целочисленный тип, если в результате его вычисления получается величина целочисленнго типа. Выражение имеет вещественный тип, если результатом его вычисления является вещественная величина.

Логические выражения

С логическими величинами и выражениями вам уже приходилось иметь дело при работе с базами данных, с электронными таблицами, в разделе «Логические основы обработки информации» учебника 10 класса. Кратко сформулируем основные правила записи логических выражений на Паскале.

Логическое выражение есть логическая формула, записанная на языке программирования. Логическое выражение состоит из логических операндов, связанных логическими операциями и круглыми скобками. Результатом вычисления логического выражения является булевская величина (*false* или *true*). Логическими операндами могут быть логические константы, переменные, функции, операции отношения. Один отдельный логический операнд является простейшей формой логического выражения.

Логические константы: `True`, `False`.

Логические переменные: описываются с типом `Boolean`.

Операции отношения: осуществляют сравнение двух операндов и определяют, истинно или ложно соответствующее отношение между ними.

Знаки операций отношения: = (равно), (не равно), $>$ (больше), $<$ (меньше), $>=$ (больше или равно), $<=$ (меньше или равно).

Логические операции: **`not`** — отрицание; **`and`** — логическое умножение (конъюнкция); **`or`** — логическое сложение (дизъюнкция), **`xor`** — «исключающее ИЛИ». Таблица истинности для этих операций выглядит так:

A	B	**not** A	A **and** B	A **or** B	A **xor** B
False	False	True	False	False	False
False	True	True	False	True	True
True	False	False	False	True	True
True	True	False	True	True	False

Логические операции располагаются в следующем порядке по убыванию старшинства (приоритета): 1) **not**, 2) **and**, 3) **or**, **xor**. Операции отношения имеют самый низкий приоритет. Поэтому если операндами логической операции являются отношения, то их следует заключать в круглые скобки. Например, математическому неравенству $1 \le x \le 50$ соответствует следующее логическое выражение:

```
(1 <= X) and (X <= 50)
```

Логическая функция `odd(x)` принимает значение `True`, если значение целого аргумента `x` является нечётным, иначе — `False`.

Система основных понятий

Операции, функции, выражения
Арифметические операции — применяются к числовым величинам; нет операции возведения в степень
Тип результата операции зависит от типа операндов (табл. 2.2)
Возведение в степень (x^y): при целом y реализуется через умножение, при вещественном y реализуется так: `Exp(y*Ln(x))`
Стандартные функции и процедуры описаны в модулях, подключаемых к программе. Модуль `SYSTEM` подключается по умолчанию
Арифметическое выражение: языковая конструкция, определяющая порядок вычисления числовой величины в соответствии с математическим выражением. Типом выражения называется тип результата его вычисления
Логическое выражение: логическая формула, записанная на языке программирования. Логическое выражение состоит из логических операндов, связанных логическими операциями и круглыми скобками

Вопросы и задания

1. Для следующих математических выражений запишите соответствующие арифметические выражения на Паскале:

а) $a + bx + cyz$; б) $[(ax - b)x + c]x - d$; в) $\frac{a+b}{c} + \frac{c}{ab}$;

г) $\frac{x+y}{a_1} \cdot \frac{a_2}{x-y}$; д) $10^4\alpha + 3\frac{1}{5}\beta$; е) $\left(1 + \frac{x}{2!} + \frac{y}{3!}\right) / \left(1 + \frac{2}{3+xy}\right)$.

2. Запишите математические выражения, соответствующие следующим выражениям на Паскале:

а) `(p+q)/(r+s)-p*q/(r*s);`

б) `1E3+beta/(x-gamma*delta);`

в) `a/b*(c+d)-(a-b)/b/c+1E-8.`

3. Для следующих математических выражений запишите соответствующие арифметические выражения на Паскале:

а) $(1+x)^2$; б) $\sqrt{1+x^2}$; в) $\cos^2 x^2$; г) $\log_2 \frac{x}{5}$; д) $\arcsin x$; е) $\frac{e^x+e^{-x}}{2}$;

ж) $x^{\sqrt{2}}$; з) $\sqrt[3]{1+x}$; и) $\sqrt{x^8+8^x}$; к) $\frac{xyz-3{,}3|x+\sqrt[4]{y}|}{10^7+\ln 4!}$; л) $\frac{\beta+\sin^2\pi^4}{\cos 2+|\mathrm{ctg}\, y|}$.

☑ **4.** Вычислите значения выражений:

а) `trunc(6.9)`
б) `trunc(6.2)`
в) `20 div 6`
г) `2 div 5`
д) `round(6.9)`
е) `round(6.2)`
ж) `20 mod 6`
з) `2 mod 5`
и) `3*7 div 2 mod 7/3 — trunc(sin(1))`

5. Определите тип выражения:

а) `1+0.0`
б) `20/4`
в) `sqr(4)`
г) `sqrt(16)`
д) `sin(0)4`
е) `trunc(-3.14)`

☑ **6.** Вычислите значения логических выражений:

а) `K mod 7 = K div 5 - 1` при `K`=15;
б) `odd(trunc(10*P))` при `P` = 0.182;
в) `not odd(n)` при `n` = 0;
г) `t and (P mod 3 = 0)` при `t` = `true`, `P` = 10101;
д) `(x*y<>0)and(y>x)` при `x` = 2, `y` = 1;
е) `a or not b` при `a` = `False`, `b` = `True`.

2.2.4 Оператор присваивания, ввод и вывод данных

Присваивание — *это действие, в результате которого переменная величина получает определённое значение.* В программе на Паскале существуют три способа присваивания значения переменной:

1) оператор присваивания;
2) оператор ввода;
3) передача значения через параметры подпрограммы.

Оператор присваивания имеет следующий формат:

```
<переменная>:=<выражение>
```

Например:

1) `x:=2*a+sqrt(b);`
2) `b:=(x>y) and (k<>0);`

Сначала вычисляется выражение, затем полученное значение присваивается переменной. В первом примере приведён **арифметический опера-**

тор присваивания. Здесь x — переменная вещественного типа. Во втором примере — **логический оператор присваивания**. Здесь b — переменная типа boolean.

Типы переменной и выражения должны совпадать. Из этого правила есть одно исключение: переменной вещественного типа можно присваивать значение целочисленного выражения. В таком случае значение целого числа преобразуется к формату с плавающей точкой и присвоится вещественной переменной.

Ввод и вывод данных

Под вводом понимается передача данных с внешнего устройства компьютера в оперативную память. При выводе данные передаются из оперативной памяти на внешнее устройство (рис. 2.3).

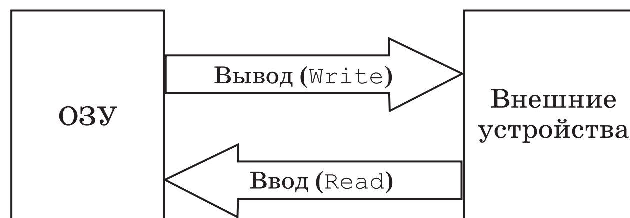


Рис. 2.3. Ввод и вывод

Операцию ввода называют чтением и выполняют с помощью оператора Read. Вывод называют записью, и для его выполнения используется оператор Write.

К внешним устройствам относятся устройства ввода и вывода (клавиатура, монитор, принтер и др.) и устройства внешней памяти (магнитные и оптические диски, флэш-память и др.). Данные на внешних устройствах организованы в виде файлов.

Для внешних запоминающих устройств (ВЗУ) **файл** — *это поименованная область памяти* этого устройства. В файлы на ВЗУ можно записывать данные по команде Write и можно читать данные из файлов по команде Read. На одном устройстве ВЗУ может храниться множество файлов одновременно. Правила именования файлов на ВЗУ определяются операционной системой. Имена для файлов, создаваемых пользователем, задает сам пользователь.

Устройства ввода с клавиатуры и вывода на экран монитора являются однофайловыми. Считается, что с клавиатурой связан один системный файл с именем INPUT, поэтому ввод с клавиатуры равнозначен чтению из файла INPUT. С монитором связан системный файл, который называется OUTPUT. Вывод на экран — это запись данных в файл OUTPUT.

По форме хранения данных файлы бывают *типизированными, нетипизированными* и *текстовыми*. В типизированных и нетипизированных файлах данные различных типов хранятся в том же формате, что и в оперативной памяти. Следовательно, при чтении и записи в такие файлы данные копируются без изменения объёма и формы представления. В тексто-

вых файлах данные хранятся в символьном формате, поэтому при вводе (чтении) чисел из текстового файла происходит преобразование их представления из символьной формы к форме их внутреннего представления (с фиксированной или плавающей точкой). А при выводе (записи) чисел в текстовой файл они преобразуются из внутренней формы в символьную.

Далее мы будем использовать только текстовые файлы.

Текстовые файлы. Текстовый файл — наиболее часто употребляемая разновидность файлов. Устройства ввода с клавиатуры и вывода на экран работают только с текстовыми файлами. Файлы, содержащие тексты программ на Паскале и других языках программирования, являются текстовыми. Различная документация, информация, передаваемая по каналам электронной связи, записана в текстовых файлах.

Содержимое текстового файла представляет собой символьную последовательность, разделённую на строки. Каждая строка заканчивается специальным признаком EOLN (end of line — конец строки). Весь файл заканчивается признаком EOF (end of file — конец файла). Схематически это выглядит так:

S_1	S_2	…	S_{k1}	EOLN	S_1	S_2	…	S_{k2}	EOLN	…	EOF

Здесь S_i обозначает i-й символ в строке. Каждый символ представлен во внутреннем коде (ASCII) и занимает 1 байт. Признак EOLN состоит из двух однобайтовых управляющих кодов: CR (код ASCII 13) — возврат к началу строки и LF (код ASCII 10) — перевод строки. При выводе содержимого текстового файла на экран или на печать признак EOLN обеспечивает визуальное разделение строк: переход к продолжению вывода с начала новой строки.

Текстовый файл можно создать или преобразовать с помощью текстового редактора. Его можно просмотреть на экране монитора или распечатать на принтере.

Ввод с клавиатуры производится путём обращения к стандартной процедуре `Read` в следующем формате:

```
Read(<список ввода>)
```

Чтение происходит из системного файла INPUT, всегда доступного для любой программы. Элементами списка ввода могут быть переменные символьного типа, числовых типов и строковые переменные.

Пример

```
Read(a, b, c, d)
```

При выполнении этого оператора происходит прерывание исполнения программы, после чего пользователь должен набрать на клавиатуре значения переменных `a`, `b`, `c`, `d`, отделяя их друг от друга пробелами. При этом вводимые значения высвечиваются на экране. В конце нажимается клавиша Enter. Значения следует вводить в строгом соответствии с синтаксисом Паскаля.

Пример:

```
Var T: Real; J: Integer; K: Char;
Begin
  Read(T, J, K);
```

Набираем на клавиатуре:

```
253.98  100  G  [Enter].
```

Если в программе имеются несколько подряд идущих операторов Read, то данные для них можно вводить последовательно (на экране они отображаются в одной строке), и лишь в конце ввода нужно нажать клавишу Enter.

Пример:

```
Var A, B: Integer;
    C, D: Real;
Begin
  Read(A, B); Read(C, D);
```

Набираем на клавиатуре и видим на экране:

```
18758 34[Enter] 2.62E-02 1.54E+01[Enter].
```

Другой вариант оператора ввода с клавиатуры имеет вид:

```
Readln(<список ввода>)
```

Здесь слово Readln означает *read line* — читать строку. Нажатие клавиши Enter в процессе ввода вырабатывает признак EOLN и данные при выполнении следующего оператора ввода будут отображаться на экране с начала новой строки. Если в предыдущем примере заменить операторы read на readln:

```
ReadLn(A, B); ReadLn(C, D);
```

то ввод значений будет происходить из двух строк, отображённых на экране:

```
18758 34 [Enter]
2.62E-02 1.54E+01 [Enter]
```

Ввод из файла на диске

Исходные данные могут быть заранее подготовлены с помощью текстового редактора и сохранены в файле на диске под определенным именем. Ввод исходных данных из файла производится автоматически, и при этом не происходит задержки выполнения программы, которая есть при клавиатурном вводе.

Для организации ввода данных из текстового файла следует:

1) объявить в программе переменную с типом text (она называется файловой переменной);
2) связать файловую переменную с файлом на ВЗУ, содержащим исходные данные с помощью оператора Assign;
3) открыть файл для чтения с помощью процедуры Reset;

4) осуществлять чтение из файла с помощью операторов Read или Readln;
5) закрыть файл с помощью оператора Close.

<u>**Пример.**</u> В текстовом файле с именем abc.txt хранятся пять чисел, разделённых на две строки:

```
2.5 3.1 4.0
0.7 1.5
```

В следующей программе организован ввод этих данных в вещественные переменные a, b, c, d, e:

```
Var a,b,c,d,e: real;
    FD: text; {Описание файловой переменной}
Begin
  Assign(FD, 'abc.txt');{переменная FD связывается с файлом
                         abc.txt}
  Reset(FD);{файл открывается для чтения с его начала}
  Readln(FD, a, b, c);{чтение первой строки файла}
  Readln(FD, d, e);{чтение второй строки файла}
  Close(FD);  {разрывается связь переменной FD с файлом}
```

Здесь FD — файловая переменная. Assign, Reset, Readln, Close — операторы обращения к стандартным процедурам, имеющим следующие форматы:

```
Assign(<файловая переменная>,<имя файла>);
Reset(<файловая переменная>);
Readln(<файловая переменная>,<список ввода>);
Close(<файловая переменная>).
```

Если файл хранится не в текущем каталоге, то в операторе Assign кроме имени файла надо указывать полный путь к нему. Имя файла можно задавать в строковой константе или переменной.

Вывод на экран производится по оператору обращения к стандартной процедуре:

```
Write(<список вывода>)
```

Здесь элементами списка вывода могут быть выражения различных типов (в частности, константы и переменные).

<u>**Пример**</u>

```
Write('Сумма ', A,'+', B,'=', A+B)
```

Если, например, A = 5 и B = 7, то на экране получим:

```
Сумма 5+7=12
```

При выводе на экран нескольких значений в строку они не отделяются друг от друга пробелами. Программист сам должен позаботиться о таком разделении. В приведенном примере предусмотрен пробел после слова «Сумма».

Второй вариант процедуры вывода на экран:

```
Writeln(<список вывода>)
```

Write line — писать строку. Действие оператора Writeln отличается от оператора Write тем, что после вывода последнего в списке значения происходит перевод курсора к началу следующей строки. Оператор Writeln, записанный без параметров, вызывает перевод строки.

В списке вывода могут присутствовать указатели форматов вывода (форматы). Формат определяет представление выводимого значения на экране. Формат отделяется от соответствующего ему элемента двоеточием. Если указатель формата отсутствует, то машина выводит значение по определенному правилу, предусмотренному «по умолчанию».

Вывод в текстовый файл

Запись результатов выполнения программы в текстовый файл позволяет их сохранить для того, чтобы в дальнейшем можно было бы их просмотреть с помощью текстового редактора, распечатать на принтере, а также использовать в качестве исходных данных для другой программы.

Для организации вывода данных в текстовый файл следует:

1) объявить в программе файловую переменную с типом text ;
2) связать файловую переменную с файлом на ВЗУ с помощью оператора Assign;
3) открыть файл для записи с помощью процедуры Rewrite;
4) осуществлять запись в файл с помощью операторов Write или Writeln;
5) закрыть файл с помощью оператора Close.

Пример

Требуется записать в текстовый файл таблицу умножения на 2.

```
Var A: integer;
    TM: text;{Описание файловой переменной}
Begin
  Assign(TM,'E:\TabMul.txt'); {Связывание переменной TM}
                              {с файлом}
  Rewrite(TM); {Открытие файла для записи}
  {Циклический вывод в файл таблицы умножения}
  For A:=2 To 9 Do Writeln(TM, 2, '*', A, '=', 2*A);
  Close(TM) {Закрытие файла}
End.
```

Процедуры открытия файла для записи и запись в файл имеют следующий формат:

```
Rewrite(<файловая переменна>);
Write (<файловая переменная>,<список ввода>);
Writeln (<файловая переменная>,<список ввода>);
```

Если файла с именем, указанным в операторе Assign, на диске не было, то программа его создаст. Если такой файл уже был, то его прежнее содержание будет утеряно и запишутся новые данные. В конце выполнения

оператора Writeln выставляется признак EOLN. Оператор Write этого не делает. Закрытие файла приводит к выставлению признака EOF.

В результате выполнения программы в корневом каталоге диска E появится файл с именем TabMul.txt. Открыв его в текстовом редакторе, увидим:

```
2*2=4
2*3=6
2*4=8
2*5=10
2*6=12
2*7=14
2*8=16
2*9=18
```

Система основных понятий

Присваивание, ввод, вывод	
Оператор присваивания: <переменная> := <выражение> Типы переменной и выражения должны совпадать. Исключение: вещественной переменной можно присваивать значение целого выражения	
Ввод — передача данных с внешнего устройства в ОЗУ	
Ввод с клавиатуры: Read(<список ввода>) или Readln(<список ввода>)	*Ввод из файла*: Read(<ФП>, <список ввода>) или Readln(<ФП>, <список ввода>) ФП — файловая переменная
Вывод — передача данных из ОЗУ на внешнее устройство	
Вывод на экран: Write(<список вывода>) или Writeln(<список вывода>)	*Вывод в файл*: Write(<ФП>, <список ввода>) или Writeln(<ФП>, <список ввода>) ФП — файловая переменная
Операторы (стандартные процедуры) работы с файлами	
Assign — назначение связи между файловой переменной и файлом на внешнем устройстве Reset — открытие файла для чтения Rewrite — открытие файла для записи Close — закрытие файла (разрыв связи с файловой переменной)	

Вопросы и задания

1. Назовите последовательность действий при выполнении оператора присваивания.
2. Сформулируйте правило соответствия типов для оператора присваивания. Какое существует исключение из этого правила?

☑ **3.** Если `y` — вещественная переменная, а `n` — целая, то какие из следующих операторов присваивания правильные, а какие — нет?

а) `y:=n + 1;` д) `y:=n div 2;`
б) `n:=y - 1;` е) `y:=y div 2;`
в) `n:=4.0;` ж) `n:=n/2;`
г) `y:=trunc(y);` з) `n:=sqr(sqrt(n)).`

☑ **4.** Напишите последовательность операторов присваивания, в результате выполнения которой целые переменные x и y обменяются значениями. При этом нельзя использовать дополнительные переменные. Найдя такой алгоритм, определите, в чём его недостаток по сравнению с методом обмена через третью переменную. Можно ли его применять для вещественных чисел?

☑ **5.** Напишите оператор присваивания, в результате выполнения которого целой переменной h присвоится значение цифры, стоящей в разряде сотен в записи положительного целого числа k (например, если $k = 28796$, то $h = 7$).

6. Напишите оператор присваивания, в результате выполнения которого целой переменной S присвоится значение суммы цифр трёхзначного целого числа k.

7. Напишите программу, по которой из текстового файла с именем kvur.txt будут прочитаны три числа: `a`, `b`, `c` — коэффициенты квадратного уравнения, затем будут вычислены корни этого уравнения и выведены на экран и в текстовый файл `korni.txt`.

Компьютерный практикум. Раздел «Программирование»

2.2.5 Структуры алгоритмов и программ

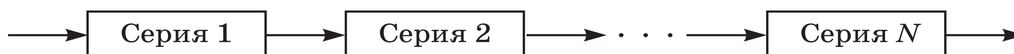
Базовые алгоритмические структуры

Эдсгер В. Дейкстра
(1930–2002)
Нидерланды

В основе структурного программирования лежит теорема, доказанная Эдсгером Дейкстрой в 1969 году. Суть её в том, что *алгоритм для решения любой логической задачи можно составить только из структур* СЛЕДОВАНИЕ, ВЕТВЛЕНИЕ, ЦИКЛ. Их называют **базовыми алгоритмическими структурами**.

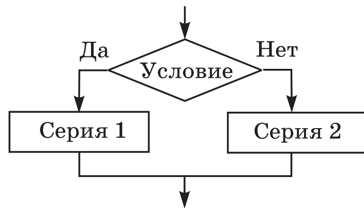
В § 1.7.1 учебника для 10 класса уже рассказывалось о базовых структурах. По сути дела, мы и раньше во всех рассматриваемых примерах программ придерживались принципов структурного программирования. Ещё раз покажем, как изображаются базовые структуры в схемах алгоритмов и как они программируются на Паскале.

Следование — это линейная последовательность действий:



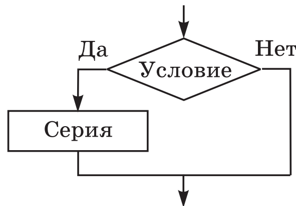
В программе на Паскале серия — это либо один отдельный оператор, либо **составной оператор:** последовательность операторов, заключённая в операторные скобки. Операторными скобками называются служебные слова `Begin` и `End`.

Ветвление — алгоритмическая альтернатива. Управление передается одному из двух блоков в зависимости от истинности или ложности условия. Затем происходит выход на общее продолжение. Вот как изображается ветвление на блок-схеме и программируется на Паскале с помощью условного оператора:




```
If <лог. выражение>
Then <серия 1>
Else <серия 2>;
```

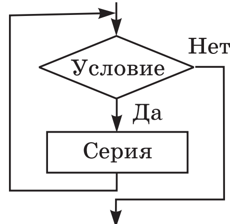
Неполная форма ветвления имеет место, когда на ветви «нет» пусто:




```
If <лог. выражение>
Then  <серия>;
```

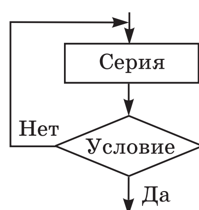
В конструкциях на Паскале операторы на ветвях могут быть как простыми, так и составными.

Цикл — повторение некоторой группы действий по условию. Различаются два типа цикла. Первый — **цикл с предусловием (цикл-пока):**




```
While <лог. выражение> Do
  <серия>;
```

Пока условие истинно, выполняется серия, образующая тело цикла.
Второй тип циклической структуры — **цикл с постусловием (цикл-до):**




```
Repeat
  <серия>
Until <лог. выражение>
```

Здесь тело цикла предшествует условию цикла. Тело цикла повторяет свое выполнение, если условие ложно. Повторение закончится, когда условие станет истинным. На Паскале для тела цикла с постусловием операторных скобок не требуется.

Теоретически необходимым и достаточным является лишь первый тип цикла — цикл с предусловием. Любой циклический алгоритм можно построить с его помощью. Это более общий вариант цикла, чем цикл-до. В самом деле, тело цикла-до хотя бы один раз обязательно выполнится, так как проверка условия происходит после завершения его выполнения. А для цикла-пока возможен такой вариант, когда тело цикла не выполнится ни разу, поэтому в любом языке программирования можно было бы ограничиться только циклом-пока. Однако в ряде случаев применение цикла-до оказывается более удобным и поэтому он используется.

Иногда в литературе структурное программирование называют программированием без `goto` — *оператора безусловного перехода*. Действительно, при таком подходе нет места безусловному переходу. Неоправданное использование в программах оператора `goto` лишает её структурности, а значит всех связанных с этим положительных свойств: прозрачности и надёжности алгоритма. Хотя во всех процедурных языках программирования этот оператор присутствует, однако, с точки зрения структурного подхода, его употребления следует избегать.

Комбинации базовых структур

Сложный алгоритм состоит из соединённых между собой базовых структур. Соединяться эти структуры могут двумя способами: *последовательным* и *вложенным*. Если блок, составляющий тело цикла, сам является циклической структурой, то имеют место вложенные циклы. В свою очередь внутренний цикл может иметь внутри себя еще один цикл и т. д. В связи с этим вводится представление о *глубине вложенности циклов*. Точно также и ветвления могут быть вложенными друг в друга.

Структурный подход требует соблюдения стандарта в изображении блок-схем алгоритмов. Чертить их нужно так, как это делалось во всех приведённых примерах. Каждая базовая структура должна иметь один вход и один выход. Нестандартно изображенная блок-схема плохо читается, теряется наглядность алгоритма. Несколько примеров структурных блок-схем алгоритмов приведены на рис. 2.4.

Такие блок-схемы легко читаются. Их структура хорошо воспринимается зрительно. Структуре каждого алгоритма можно дать название. Приведенные блок-схемы можно охарактеризовать следующим образом (в порядке номеров):

1. Вложенные ветвления. Глубина вложенности равна единице.
2. Цикл с вложенным ветвлением.
3. Вложенные циклы-пока. Глубина вложенности — 1.
4. Ветвление с вложенной последовательностью ветвлений на положительной ветви и с вложенным циклом-пока на отрицательной ветви.
5. Следование ветвления и цикла-до.
6. Вложенные циклы. Внешний — цикл-пока, внутренний — цикл-до.

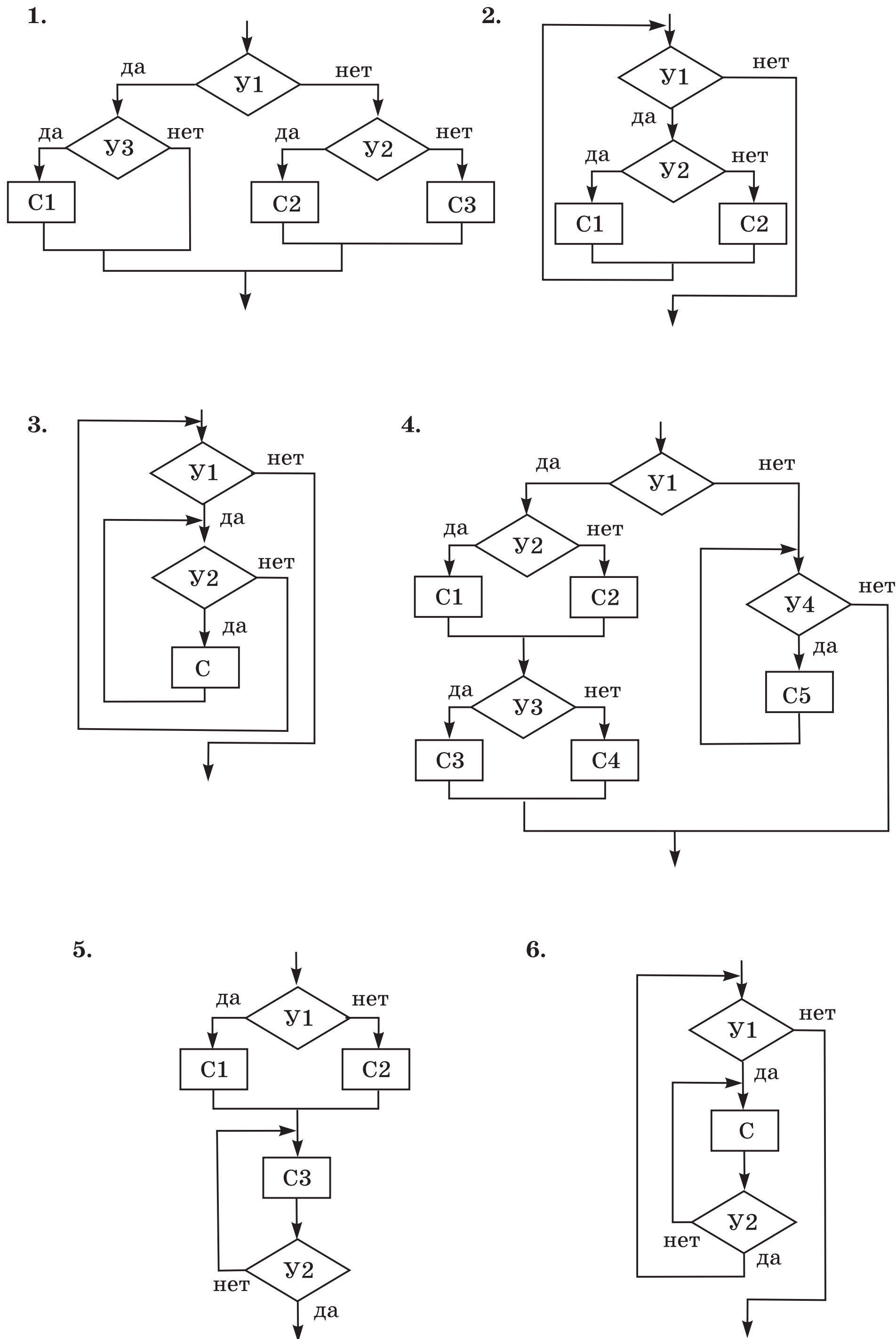


Рис. 2.4. Структурные схемы алгоритмов

Наглядность структуре программы на Паскале придает структуризация внешнего вида текста программы. Основной используемый для этого приём — сдвиги строк, которые должны подчиняться следующим правилам:

- конструкции одного уровня вложенности записываются на одном вертикальном уровне (начинаются с одной позиции в строке);
- вложенная конструкция записывается смещённой по строке на несколько позиций вправо относительно внешней для неё конструкции.

Для приведённых на рис. 2.4 блок-схем структура текстов программ на Паскале должна быть следующей.

1.

```
If <У1>
Then
  If <У2>
  Then <С1>
Else
  If <У3>
  Then <С2>
  Else <С3>;
```

2.

```
While <У1> Do
  If <У2>
  Then <С1>
  Else <С2>;
```

3.

```
While <У1> Do
  While <У2> Do
    <С>;
```

4.

```
If <У1>
Then
Begin
  If <U2>
  Then <С1>
  Else <С2>;
  If <У3>
  Then <С3>
  Else <С4>
End
Else While <У4> Do <С5>;
```

5.

```
If <У1>
Then <С1>
Else <С2>;
Repeat
  <С3>
Until <У2>;
```

6.

```
While <У1> Do
  Repeat
    <С1>
  Until <У2>;
```

Структурное программирование — это не только форма описания алгоритма и программы, но ещё и *способ мышления программиста*. Размышляя над алгоритмом, нужно стремиться составлять его из стандартных структур. Если использовать строительную аналогию, то структурная методика построения алгоритма подобна сборке здания из стандартных секций, в отличие от складывания по кирпичику.

Система основных понятий

Структуры алгоритмов и программ		
Базовые алгоритмические структуры		
Следование	**Ветвление**	**Цикл**
Линейная последовательность действий	Выбор одной из двух серий действий с выходом на общее продолжение	Повторение серии действий по условию. Цикл с предусловием, цикл с постусловием
Комбинации базовых структур: последовательность, вложенность		
Структурный алгоритм (программа): построенный из базовых алгоритмических структур, не содержащий команды безусловного перехода (goto)		
Структурирование текста программы: использование сдвигов строк для вложенных конструкций		

Вопросы и задания

☑ **1.** Перечислите основные базовые алгоритмические структуры и средства их программирования в Паскале.

2. Какой алгоритм называется структурным?

☑ **3.** Нарисуйте блок-схемы и напишите на Паскале два варианта программы решения задачи: выбрать из двух числовых величин большее значение. Первый вариант — с полным ветвлением, второй вариант — с неполным ветвлением.

☑ **4.** Нарисуйте блок-схемы и напишите на Паскале два варианта программы решения задачи: выбрать из трех числовых величин наименьшее значение. Первый вариант — с вложенными ветвлениями, второй вариант — с последовательными ветвлениями.

☑ **5.** Для данного натурального числа N требуется вычислить сумму: $1 + 1/2 + 1/3 + ... + 1/N$. Постройте алгоритм и напишите два варианта программы на Паскале: с циклом-до и с циклом-пока.

☑ **6.** Какую структуру будет иметь алгоритм решения следующей задачи? Дано целое положительное число N. Если N — чётное, то вычислить $N! = 1 \cdot 2 \cdot ... \cdot N$. Если N — нечётное, то вычислить сумму: $1 + 2 + ... + N$. Составьте программу на Паскале.

☑ **7.** В следующем фрагменте программы k и S — переменные целого типа:

```
Readln(k);
S:=0;
While k>0 Do
Begin
  S:=S+k mod 10;
  k:=k div 10
End;
Writeln(S);
```

Какая решается задача? Запрограммируйте решение этой же задачи с использованием цикла с постусловием.

2.2.6 Программирование ветвлений

Загляните в § 1.7.4 учебника для 10 класса. Там изображён структурный алгоритм решения квадратного уравнения и написана программа на Паскале, в которой соблюдены все вышеперечисленные правила структуризации текста программы. Алгоритм имеет структуру вложенных ветвлений. Рассмотрим другие задачи, которые решаются с помощью ветвящегося алгоритма.

Пример 1

Требуется перевести пятибалльную оценку в её наименование: 5 — «отлично», 4 — «хорошо», 3 — «удовлетворительно», 2 — «неудовлетворительно».

Блок-схема алгоритма приведена на рис. 2.5.

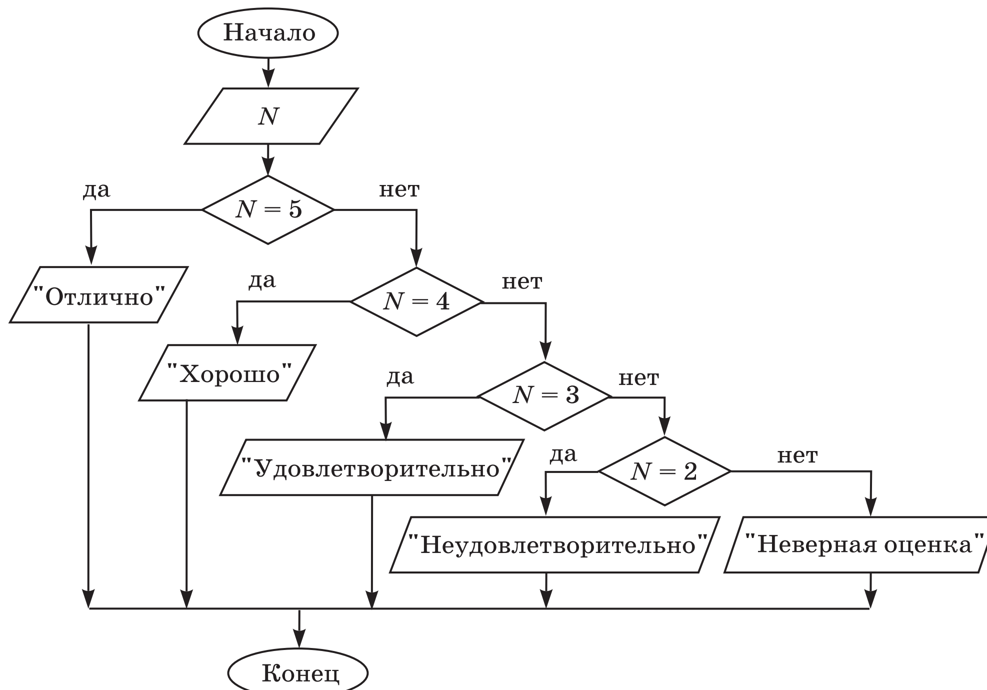


Рис. 2.5. Алгоритм перевода числовой оценки в словесную

Этот алгоритм имеет структуру вложенных ветвлений и может быть запрограммирован с использованием **условного оператора `If`** следующим образом:

```
Program Marks_1;
Var N: Integer;
Begin
  WriteLn('Введите оценку:');
  ReadLn(N);
  If (N=5)
  Then WriteLn('Отлично')
  Else
    If (N=4)
    Then WriteLn('Хорошо')
    Else
      If (N=3)
      Then WriteLn('Удовлетворительно')
      Else
        If (N=2)
        Then WriteLn('Неудовлетворительно')
        Else WriteLn('Неверная оценка')
End.
```

Пример 2

Решение рассмотренной в предыдущем примере задачи можно запрограммировать с помощью одного **оператора выбора**, имеющегося в языке Паскаль. Вот как будет выглядеть такая программа:

```
Program Marks_2;
Var N: Integer;
Begin
  WriteLn('Введите оценку:');
  ReadLn(N);
  Case N Of
    5: WriteLn('Отлично');
    4: WriteLn('Хорошо');
    3: WriteLn('Удовлетворительно');
    2: WriteLn('Неудовлетворительно');
  Else WriteLn('Нет такой оценки')
End;
```

Оператор выбора имеет следующий формат:

```
Case <селектор> Of
  <список констант>: <оператор>;
  ... ... ...
  <список констант>: <оператор>;
Else <оператор>
End
```

Здесь <селектор> — это выражение любого *порядкового типа*; <константа> — постоянная величина того же типа, что и селектор; <оператор> — любой простой или составной оператор.

Выполнение оператора выбора происходит так: вычисляется выражение — селектор; затем в списках констант ищется такое значение, которое совпадает с полученным значением селектора; далее исполняется оператор, помеченный данной константой. Если такой константы не найдено, то происходит переход к выполнению оператора, следующего после слова **Else**.

Пример 3

В этом примере демонстрируется использование списка констант в операторе выбора. Программа сообщает, сдал студент экзамен или не сдал. Если оценка одна из следующих: 3, 4, 5, то экзамен сдан; если оценка 2, то не сдан.

```
Case N Of
  3, 4, 5: WriteLn('Экзамен сдан');
  2: WriteLn('Экзамен не сдан');
Else WriteLn('Нет такой оценки');
```

Так же, как условный оператор, оператор выбора может использоваться в неполной форме, т. е. без ветви **Else**.

Если применить условный оператор, то эта программа запишется так:

```
If (N=3) or (N=4) or (N=5)
Then WriteLn('Экзамен сдан')
Else
  If N=2
  Then WriteLn('Экзамен не сдан')
  Else WriteLn('Нет такой оценки');
```

В условии ветвления использовано сложное логическое выражение, содержащее операции логического сложения **or** (или).

Пример 4

Составить программу определения принадлежности заданной точки с координатами (x, y) на плоскости области, заштрихованной на рис. 2.6, включая её границы[1].

Запишем систему неравенств, которой удовлетворяют точки, принадлежащие данной области:

$$\begin{cases} y \le 1; \\ y \ge \sin x; \\ x \ge 0; \\ x \le \pi/2. \end{cases}$$

[1] Задача взята из демоверсии ЕГЭ по информатике 2009 г.

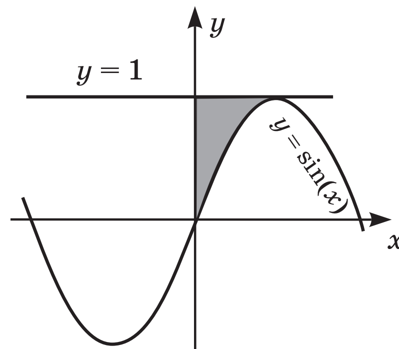


Рис. 2.6. Область на плоскости

Рассмотрим два варианта применения ветвления для решения этой задачи.

<table>
<tr>
<td>

```
Program Variant_1;
Var x, y: real;
Begin
  Write('Введите x, y:');
  Readln(x, y);
  If y<=1 Then
    If y>=sin(x)Then
      If x>=0 Then
        If x<=Pi/2 Then
           Write('Yes')
        Else Write('No')
      Else Write('No')
    Else Write('No')
  Else Write('No')
End.
```

</td>
<td>

```
Program Variant_2;
Var x, y: real;
Begin
  Write('Введите x, y:');
  Readln(x, y);
  If (y<=1) and (y>=sin(x)) and
     (x>0) and (x<=Pi/2)
  Then Write('Yes')
  Else Write('No')
End.
```

</td>
</tr>
</table>

Первый вариант имеет структуру вложенных ветвлений с простыми условиями. Во втором варианте — одно ветвление с условием, являющимся сложным логическим выражением. Рассмотрим еще один вариант программы решения этой задачи с линейной структурой.

```
Program Variant_3;
Var x, y: real; Flag: Boolean;
Begin
  Write('Введите x, y:'); Readln(x, y);
  Flag:=(y<=1) and (y>=sin(x)) and  (x>0) and (x<=Pi/2);
  Write('Точка принадлежит области?', Flag:6)
End.
```

Здесь используется логическая переменная Flag, которая принимает значение True или False в зависимости от положения точки. Например, если ввести значения x = 1, y = 1, то на экран выведется:

```
Точка принадлежит области? True.
```

А если x = 1, y = 2, то получим:

```
Точка принадлежит области? False.
```

Формат вывода :6 логической величины позволяет отделить логическое значение от предыдущего текста пробелами.

Система основных понятий

Программирование ветвлений	
Условный оператор	**Оператор выбора**
If (<логическое выражение>) **Then** <оператор 1> **Else** <оператор 2> Если <логическое выражение> истинно, то выполняется <оператор 1>, иначе — выполняется <оператор 2>. <Оператор 1> и <оператор 2> — простые или составные операторы. В неполном ветвлении отсутствует **Else** ...	**Case** <селектор> **Of** <список констант 1 >: <оператор 1>; ... <список констант N>: <оператор N> **Else** <оператор> **End** Селектор — выражение порядкового типа; константы имеют тот же тип. Выполняется только одна из ветвей выбора, которая содержит константу, совпадающую со значением селектора. Ветвь **Else** может отсутствовать

Вопросы и задания

1. Какие операторы используются для программирования ветвящихся алгоритмов?
2. Является ли оператор выбора необходимым для программирования ветвящихся алгоритмов?
3. В каких случаях удобно использование оператора выбора?
4. ☑ Составьте на Паскале программу полного решения биквадратного уравнения.
5. ☑ Используя оператор выбора, составьте программу, которая по введенному номеру месяца в году будет выводить соответствующее время года (зима, весна, лето, осень).

Компьютерный практикум. Раздел «Программирование»

2.2.7 Программирование циклов

Рассмотрим приёмы программирования циклов на Паскале. Различают циклы с заданным числом повторений и итеративные циклы.

Задача. Известно, что сумма следующего бесконечного числового ряда:

$$1+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\ldots+\frac{1}{i!}+\ \ldots$$

в пределе стремится к значению константы $e = 2{,}71828182\ldots$. Функция e^x называется экспонентой, а логарифм по основанию e называется натуральным логарифмом: $\ln(x)$.

Требуется составить программу, вычисляющую эту константу по сумме числового ряда.

Если слагаемые в этом выражении обозначить следующим образом:

$$a_0=1,\ a_1=\frac{1}{1!},\ a_2=\frac{1}{2!},\ a_3=\frac{1}{3!},\ \ldots\ ,$$

то обобщённая формула для i-го элемента будет такой:

$$a_i=\frac{1}{i!}\ .$$

Нетрудно увидеть, что между элементами данной последовательности имеется зависимость:

$$a_0 = 1,\ a_1=\frac{a_0}{1},\ a_2=\frac{a_1}{2},\ a_3=\frac{a_2}{3}\ \text{ и т. д.}$$

Такая зависимость называется **рекуррентной зависимостью**, а соответствующая числовая последовательность — **рекуррентной последовательностью**. Данная рекуррентная последовательность может быть описана следующим образом:

$$a_i=\begin{cases}1, & \text{при } i=0;\\ \dfrac{a_{i-1}}{i}, & \text{при } i>0.\end{cases}$$

Циклы с заданным числом повторений

Составим программу, по которой будет вычислена сумма заданного количества слагаемых. Постановка задачи такая: дано целое положительное значение n. Требуется вычислить сумму:

$$1+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\ldots+\frac{1}{n!}.$$

Ниже приводятся два варианта программы решения этой задачи. В первом варианте используется цикл с предусловием, во втором — цикл с постусловием.

<table>
<tr>
<td>

```
Program Summa_1;
Var E, a: real; N, i: integer;
Begin
  Write('N='); Readln(N);
  E:=0; i:=0; a:=1;
  While i<=N Do
  Begin
    E:=E+a;
    i:=i+1;
    a:=a/i
  End;
  Writeln('E=', E)
End.
```

</td>
<td>

```
Program Summa_2;
Var E, a: real; N, i: integer;
Begin
  Write('N='); Readln(N);
  E:=0; i:=0; a:=1;
  Repeat
    E:=E+a;
    i:=i+1
    a:=a/i;
  Until i>N;
  Writeln('E=', E)
End.
```

</td>
</tr>
</table>

Обратите внимание на то, как цикл с предусловием преобразуется в цикл с постусловием: условие цикла помещается после тела цикла и заменяется на противоположное:

```
Not(i<=N) = i>N.
```

И тот, и другой цикл повторит свое выполнение N раз. Переменная i выполняет роль не только знаменателя в дроби $1/i!$, но и является счетчиком числа повторений цикла. Такие переменные называются параметрами цикла.

Выполнение этих программ на компьютере для значения N = 7 приводит к следующему результату: E = 2,7182539.

Для программирования циклов с заданным числом повторений при постоянном шаге изменения параметра цикла в Паскале существует **цикл с параметром**. Вот как выглядит программа решения той же задачи с использованием цикла с параметром:

```
Program Summa_3;
Var E, a: real; N, i: integer;
Begin
  Write('N='); Readln(N);
  E:=1; a:=1;
  For i:=1 To N Do
  Begin
    a:=a/i;
    E:=E+a
  End;
  Writeln('E=', E)
End.
```

В программе используется оператор цикла **For**, для которого существуют два варианта:

1) **For** <параметр цикла>:=<выражение 1> **To** <выражение 2> **Do** <оператор>
2) **For** <параметр цикла>:=<выражение 1> **Downto** <выражение 2> **Do** <оператор>

Здесь <параметр цикла> — имя простой переменной порядкового типа. Выполнение оператора **For** в первом варианте (**To**) происходит по следующей схеме.

1. Вычисляются значения <выражения 1> и <выражения 2>. Это делается только один раз при входе в цикл.
2. Параметру цикла присваивается значение <выражения 1>.
3. Значение параметра цикла сравнивается со значением <выражения 2>. Если параметр цикла меньше или равен этому значению, то выполняется тело цикла, в противном случае выполнение цикла заканчивается.
4. Значение параметра цикла изменяется на следующее значение в его типе (для целых чисел — увеличивается на единицу); происходит возврат к пункту 3.

Оператор цикла **For** объединяет в себе действия, которые при использовании цикла **While** выполняют несколько операторов: присваивание параметру начального значения, сравнение с конечным значением, изменение значения на следующее.

Во втором варианте оператора **For** слово **Downto** буквально можно перевести как «вниз до». В таком случае параметр цикла изменяется по убыванию, т. е. при каждом повторении цикла параметр изменяет свое значение на предыдущее (равносильно `i:=pred(i)`).

Работая с оператором **For**, учитывайте следующие правила:

- параметр цикла не может иметь вещественный тип;
- в теле цикла нельзя изменять переменную — параметр цикла;
- при выходе из цикла значение переменной-параметра является неопределённым.

В следующем примере в качестве параметра цикла **For** используется символьная переменная. Пусть требуется получить на экране десятичные коды букв латинского алфавита. Как известно, латинские буквы в таблице кодировки упорядочены по алфавиту. Вот фрагмент такой программы:

```
For C := 'a' To 'z' Do
  Write (C, ' - ', Ord(c));
```

Здесь переменная `c` имеет тип `char`.

А теперь подумайте сами, как вывести кодировку латинского алфавита в обратном порядке (от `'z'` до `'a'`).

Итерационные циклы

Итерационными называются циклы, точное число повторений которых заранее не известно. Оно определится только в результате выполнения цикла. В итерационном цикле при каждом его повторении происходит последовательное приближение к вычисляемой величине и проверка условия достижения искомого результата. Выход из итерационного цикла осуществляется в случае выполнения заданного условия.

Снова рассмотрим задачу вычисления суммы того же числового ряда. Но теперь условие будет таким: в сумму нужно включить только слагаемые, значение которых больше некоторой малой величины ε. При этом полученная сумма будет отличаться от предельного значения (константы *e*) на величину, не большую ε.

Поскольку с увеличением значения `i` величина $1/i!$ уменьшается, то следовательно в сумму надо включать все слагаемые, предшествующие первому значению, меньшему ε. Вот две программы решения этой задачи: с циклом с предусловием и циклом с постусловием:

```
Program Summa_4;
Var E,a,eps: real; i:integer;
Begin
  Write('Eps='); Readln(eps);
  E:=0; i:=0; a:=1;
  While a>eps Do
  Begin
    E:=E+a;
    i:=i+1;
    a:=a/i
  End;
  Writeln('E=', E,
         'Слагаемых:', i)
End.
```

```
Program Summa_5;
Var E,a,eps: real; i:integer;
Begin
  Write('Eps='); Readln(eps);
  E:=0; i:=0; a:=1;
  Repeat
    E:=E+a;
    i:=i+1;
    a:=a/i
  Until a<=eps;
  Writeln('E=', E,
         'Слагаемых:', i)
End.
```

Решить эту задачу, используя цикл с параметром, нельзя. Итерационные циклы программируются путём использования либо цикла-пока, либо цикла-до.

В качестве результата выводится значение суммы и число вошедших в неё слагаемых. Выполнение этих программ для значения $\varepsilon = 10^{-8}$ дает в результате:

```
E=2,71828182, Слагаемых: 12
```

Таким образом, за 12 повторений цикла значение константы *e* получено с точностью до 8 знаков после запятой.

Система основных понятий

<table>
<tr><th colspan="4">Программирование циклов</th></tr>
<tr><td colspan="4">Разновидности циклических алгоритмов</td></tr>
<tr><td colspan="2">Циклы с заданным числом повторений</td><td colspan="2">Итерационные циклы</td></tr>
<tr><td colspan="2">Имеется управляющий параметр, изменяющийся с постоянным шагом в определенном диапазоне значений.
Реализуется всеми типами операторов цикла</td><td colspan="2">Число повторений цикла заранее не известно.
Реализуется операторами цикла-пока и цикла-до</td></tr>
<tr><td colspan="4">Операторы цикла</td></tr>
<tr><td>Цикл-пока</td><td colspan="2">Цикл с параметром</td><td>Цикл-до</td></tr>
<tr><td>While <лог. выражение> Do <оператор>;
<оператор> — тело цикла. Цикл повторяет выполнение, пока истинно <лог. выражение></td><td colspan="2">Параметр — переменная порядкового типа
1) For <параметр цикла> := <выражение 1> To <выражение 2> Do <оператор>
— по возрастанию параметра
2) For <параметр цикла> := <выражение 1> Downto <выражение 2> Do <оператор>
— по убыванию параметра</td><td>Repeat <оператор> Until <лог.выражение>
Повторяется выполнение тела цикла до того, как <лог. выражение> станет истинным</td></tr>
</table>

Вопросы и задания

1. Чем отличается итерационный цикл от цикла с заданным числом повторений?
2. Почему для программирования итерационных циклов не используется оператор цикла с параметром?
3. ☑ Значение функции e^x (экспонента от x) равно сходящейся сумме бесконечного ряда:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots$$

   Получите рекуррентную формулу для слагаемых. Используя операторы цикла **While**, **Repeat** и **For**, составьте три варианта программы вычисления суммы с заданным числом слагаемых.
4. Запрограммируйте итерационный цикл вычисления функции e^x (см. задачу 1 в этом параграфе), с заданной точностью ε. Составьте два варианта программы: с циклами **While** и **Repeat**. Сопоставьте полученный результат со значением стандартной функции $\exp(x)$.
5. Вычислите сумму квадратов всех целых чисел, попадающих в интервал $(\ln x,\ e^x)$, для заданного $x > 1$.
6. Вычислите количество точек с целочисленными координатами, попадающих в круг радиуса R ($R > 0$) с центром в начале координат.

7. Получите таблицу значений функции $\sin x$ и $\cos x$ на отрезке $[0, 1]$ с шагом 0.1 в следующем виде:

x	sin x	cos x
0.0000	0.0000	1.0000
0.1000	0.0998	0.9950
...	...	...
1.0000	0.8415	0.5403

8. Получите в возрастающем порядке все трёхзначные числа, в десятичной записи которых нет одинаковых цифр.
9. Дано целое $n > 2$. Получите все простые числа из диапазона $[2, n]$.

Компьютерный практикум. Раздел «Программирование»

2.2.8 Вспомогательные алгоритмы и подпрограммы

Еще одним важнейшим методологическим приёмом структурного программирования является *декомпозиция решаемой задачи на подзадачи* — более простые, с точки зрения программирования, части исходной задачи. Алгоритмы решения таких подзадач называются **вспомогательными алгоритмами**.

В языках программирования вспомогательные алгоритмы называются **подпрограммами**. В Паскале различаются две разновидности подпрограмм: **процедуры** и **функции**. Рассмотрим этот вопрос на примере следующей задачи: даны два натуральных числа a и b. Требуется определить наибольший общий делитель трёх величин: $a + b$, $|a - b|$, $a \cdot b$. Запишем это так: НОД($a + b$, $|a - b|$, $a \cdot b$).

Идея решения состоит в следующем математическом факте: если x, y, z — три натуральных числа, то НОД(x, y, z) = НОД(НОД(x, y), z). Иначе говоря, нужно найти НОД двух величин, а затем НОД полученного значения и третьего числа (попробуйте это доказать).

Очевидно, что вспомогательным алгоритмом для решения поставленной задачи является алгоритм получения наибольшего общего делителя двух чисел. Эта задача решается с помощью алгоритма Евклида, который подробно обсуждался в 9 классе. Напомним, что идея алгоритма Евклида основана на следующей формуле:

$$\text{НОД}(M, N) = \begin{cases} M, \text{ при } M = N; \\ \text{НОД}(M - N,\ N), \text{ при } M > N; \\ \text{НОД}(N,\ N - M), \text{ при } N > M. \end{cases}$$

Приведём алгоритм решения поставленной задачи на учебном Алгоритмическом языке. Алгоритм состоит из процедуры «Евклид» и основного алгоритма «Задача», в котором присутствуют два обращения к процедуре:

```
Процедура Евклид(цел M, N, K);
нач
  пока M<>N
  нц
    если M>N
    то M:=M-N
    иначе N:=N-M
    кв
  кц;
  K:=M
кон

алг задача;
цел a, b, c;
нач
  ввод(a, b);
  Евклид(a+b, |a-b|, c);
  Евклид(c, a*b, c);
  вывод(c)
кон
```

Здесь M, N и K являются формальными параметрами процедуры. M и N параметры-аргументы, K — параметр-результат.

Процедуры в Паскале. Основное отличие процедур в Паскале от процедур в Алгоритмическом языке (АЯ) состоит в том, что процедуры в Паскале описываются в разделе описания подпрограмм, а в АЯ процедура является внешней по отношению к вызывающей программе. Теперь посмотрим, как решение поставленной задачи программируется на Паскале.

```
Program NOD1;
Var A, B, C: Integer;
Procedure Evklid(M, N: Integer; Var K: Integer);
Begin
  While M<>N Do
    If M>N
    Then M:=M-N
    Else N:=N-M;
  K:=M
End;
Begin
  Write('a = '); ReadLn(A);
  Write('b = '); ReadLn(B);
  Evklid(A+B, Abs(A-B), C);
  Evklid(C, A*B, C);
  WriteLn('НОД = ', C)
End.
```

В данном примере **обмен** аргументами и результатами между основной программой и процедурой производится **через параметры**. Описание процедуры на Паскале имеет следующий формат:

```
Procedure <имя процедуры> [(список формальных параметров)];
<блок>
```

Квадратные скобки указывают на то, что список формальных параметром может отсутствовать, т. е. возможна процедура без параметров. Параметры могут быть параметрами-переменными и параметрами-значениями. Параметры-переменные записываются следующим образом:

```
Var <список переменных>: <тип>
```

Параметры-значения указываются так:

```
<список переменных>: <тип>
```

Чаще всего аргументы представляются как параметры-значения (хотя могут быть и параметрами-переменными). А для передачи результатов используются параметры-переменные. Процедура в качестве результата может передавать в вызывающую программу множество значений (в частном случае — одно), а может и ни одного. Теперь рассмотрим правила обращения к процедуре. Обращение к процедуре производится в форме **оператора процедуры**:

```
<имя процедуры>[(список фактических параметров)]
```

Если описана процедура с формальными параметрами, то обращение к ней производится оператором процедуры с фактическими параметрами. Правила соответствия между формальными и фактическими параметрами: соответствие *по количеству*, соответствие *по последовательности* и соответствие *по типам*.

Взаимодействие формальных и фактических параметров через параметры-пременные называется передачей **по ссылке**: при обращении к процедуре ей передаётся ссылка на переменную, заданную в качестве фактического параметра. Эта ссылка и используется процедурой для доступа к этой переменной.

Другой вариант взаимодействия формальных и фактических параметров называется передачей **по значению**: вычисляется значение фактического параметра (выражения), и это значение присваивается соответствующему формальному параметру.

В рассмотренном нами примере формальные параметры `M` и `N` являются параметрами-значениями. Это аргументы процедуры. При обращении к ней первый раз им соответствуют значения выражений `A+B` и `abs(A-B)`; второй раз — `C` и `A*B`. Параметр `K` является параметром-переменной. В ней получается результат работы процедуры. В обоих обращениях к процедуре соответствующим фактическим параметром является переменная `C`. Через эту переменную основная программа получает результат.

Теперь рассмотрим другой вариант программы, решающей ту же задачу. В ней используется процедура **без параметров**.

```
Program NOD2;
Var A, B, K, M, N: Integer;
Procedure Evklid;
Begin
  While M<>N Do
    If M>N
    Then M:=M-N
    Else N:=N-M;
  K:=M
End;
Begin
  Write('a= '); ReadLn(A);
  Write('b= '); ReadLn(B);
  M:=A+B; N:=Abs(A-B);
  Evklid;
  M:=K; N:=A*B;
  Evklid;
  WriteLn('НОД равен ', K)
End.
```

Чтобы разобраться в этом примере, требуется объяснить новое для нас понятие: область действия описания.

Областью действия описания любого программного объекта (переменной, типа, константы и т. д.) является тот блок, на который это описание распространяется. Если данный блок вложен в другой (подпрограмма), то присутствующие во вложенном блоке описания являются **локальными**. Они действуют только в пределах внутреннего блока. Описания же, расположенные во внешнем блоке, называются **глобальными** по отношению к внутреннему блоку. Если глобально описанный объект используется во внутреннем блоке, то на него распространяется внешнее (глобальное) описание.

В программе NOD1 переменные M, N, K являются локальными внутри процедуры; переменные A, B, C — глобальные. Однако внутри процедуры переменные A, B, C не используются. Связь между внешним блоком и процедурой осуществляется через параметры.

В программе NOD2 все переменные являются глобальными. В процедуре Evklid нет ни одной локальной переменной (нет и параметров). Переменные M и N, используемые в процедуре, получают свои значения через оператор присваивания в основном блоке программы и изменяют значения в подпрограмме. Результат получается в глобальной переменной K, значение которой выводится на экран. Здесь **обмен** значениями между основной программой и процедурой производится **через глобальные переменные**.

Использование механизма передачи через параметры делает процедуру более универсальной, не зависимой от основной программы. Однако в некоторых случаях оказывается удобнее использовать передачу через глобальные переменные. Чаще такое бывает с процедурами, работающими с большими объемами информации. В этой ситуации глобальное взаимодействие экономит память компьютера.

Функции. Теперь выясним, что такое подпрограмма-функция. Обычно функция используется в том случае, когда результатом работы подпрограммы должна быть скалярная (простая) величина. Тип результата называется типом функции. В Турбо Паскале допускаются функции строкового типа. Формат описания функции следующий:

```
Function <имя функции> [(<список формальных параметров>)]:
         <тип функции>; <блок>
```

Как и у процедуры, у функции в списке формальных параметров могут присутствовать параметры-переменные и параметры-значения. Всё это — аргументы функции. Параметры вообще могут отсутствовать, если аргументы передаются глобально.

Программа решения рассмотренной выше задачи с использованием функции будет выглядеть следующим образом:

```
Program NOD3;
Var A, B, Rez: Integer;
Function Evklid(M, N: Integer): Integer;
Begin
  While M<>N Do
    If M>N
    Then M:=M-N
    Else N:=N-M;
  Evklid:=M
End;
Begin
  Write('a = '); ReadLn(A);
  Write('b = '); ReadLn(B);
  Rez:= Evklid(Evklid(A+B, Abs(A-B)), A*B);
  WriteLn('NOD равен ', Rez)
End.
```

Из примера видно, что тело функции отличается от тела процедуры только тем, что в функции *результат присваивается идентификатору функции*: `Evklid:=M`

Обращение к функции является операндом в выражении. Оно записывается в следующей форме:

```
<Имя функции> (<Список фактических параметров>)
```

Правила соответствия между формальными и фактическими параметрами всё те же. Сравнивая приведённые выше программы, можно сделать вывод, что программа `NOD3` имеет определённые преимущества перед другими. Функция позволяет получить результат путем выполнения одного оператора присваивания. Здесь также иллюстрируется возможность того, что фактическим аргументом при обращении к функции может быть эта же функция.

По правилам стандарта Паскаля возврат в вызывающую программу из подпрограммы происходит, когда выполнение подпрограммы доходит до ее конца (последний **End**). Однако в Турбо Паскале есть средство, позволя-

ющее выйти из подпрограммы в любом её месте. Это оператор-процедура Exit. Например, функцию определения большего из двух данных вещественных чисел можно описать так:

```
Function Max(X, Y: Real): Real;
Begin
  Max:=X;
  If X>Y Then Exit Else Max:=Y
End;
```

Модифицированный алгоритм Евклида. Подпрограмму алгоритма Евклида можно составить иначе, если воспользоваться операцией **mod** — получением остатка от деления, имеющейся в Паскале. Идея алгоритма исходит из справедливости следующих равенств:

$$\text{НОД}(M, N) = \begin{cases} M, \text{ при } N = 0; \\ \text{НОД}(N,\ M \bmod N), \text{ при } N \neq 0. \end{cases}$$

В таком случае функцию Evklid можно переписать так:

```
Function Evklid(M, N: integer): integer;
Var R: integer;
Begin
  While N<>0 Do
  Begin R:=M mod N; M:=N; N:=R End;
  Evklid:=M
End;
```

Система основных понятий

Подпрограммы	
Процедуры	**Функции**
Результат — *любое число величин* Описание: **Procedure** <имя процедуры> [(список формальных параметров)]; <блок>	Результат — *одна величина* Описание: **Function** <имя функции> [(<список формальных параметров>)]: <тип функции>; <блок>
Обращение — *оператор процедуры:* <Имя процедуры> [(список фактических параметров)]	Обращение — *операнд выражения:* <Имя функции> (<Список фактических параметров>)
Параметры подпрограмм	
Параметры-переменные	Параметры-значения
Описание: Var <список переменных>: <тип>	Описание: <список переменных>: <тип>
Фактические параметры: *переменные*	Фактические параметры: *выражения*

Вопросы и задания

1. Для чего используются подпрограммы?
2. В чем различие между процедурами и функциями?
3. Какие существуют способы передачи данных между подпрограммой и вызывающей её программой?
4. ☑ Составьте программу вычисления площади кольца по значениям внутреннего и внешнего радиусов, используя подпрограмму вычисления площади круга (два варианта: с процедурой и с функцией).
5. ☑ Составьте программу сложения двух простых дробей. Результат должен быть несократимой дробью. Используйте подпрограмму вычисления НОД по алгоритму Евклида. Простая дробь задается двумя целыми числами: числителем и знаменателем.
6. ☑ По координатам вершин треугольника вычислите его периметр, используя подпрограмму вычисления длины отрезка между двумя точками.
7. ☑ Даны три целых числа. Определите, у которого из них больше сумма цифр. Подсчёт суммы цифр организуйте через подпрограмму.

Компьютерный практикум. Раздел «Программирование»

2.2.9 Массивы

Массивом в Паскале называют переменную величину регулярного типа.

Регулярный тип — это структурный тип данных, представляющих собой совокупность пронумерованных однотипных величин.

Описание массивов. Переменная регулярного типа описывается в разделе описания переменных в следующей форме:

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

В данном случае квадратные скобки — это обязательные символы, которые называются индексными скобками. Чаще всего в качестве типа индекса употребляется ограниченный тип. Например, массив вещественных чисел, хранящий 12 значений среднемесячных температур в течение года, опишется так:

```
Var T: array [1..12] of Real;
```

Описание массива определяет, во-первых, размещение массива в памяти, во-вторых, правила его дальнейшего употребления в программе.

Элемент массива идентифицируется в виде переменной с индексами:

```
<идентификатор массива>[<индексы элемента>]
```

Для одномерного массива индекс — это одно значение. Для многомерных массивов индекс — множество значений. В качестве индекса может употребляться любое выражение соответствующего типа. Например, для элементов массива температур возможны обозначения: `T[5]`, `T[k]`, `T[i+j]`, `T[m div 2]`.

Последовательные элементы массива располагаются в последовательных ячейках памяти (`T[1]`, `T[2]` и т. д.), причём значения индекса не должны выходить за диапазон `1..12`.

Тип индекса может быть любым скалярным порядковым типом, кроме `integer`. Например, в программе могут присутствовать следующие описания:

```
Var Cod: array[Char] of 1..100;
    L: array[Boolean] of Char;
```

В такой программе допустимы следующие обозначения элементов массивов:

```
cod['x']; L[true]; cod[chr(65)]; L[a>0].
```

В некоторых случаях бывает удобно в качестве индекса использовать перечислимый тип. Например, данные о количестве учеников в четырёх десятых классах одной школы могут храниться в следующем массиве:

```
Type Index = (A, B, C, D);
Var Class_10: array[Index] of Byte;
```

И если, например, элемент `class_10[A]` равен 35, то это означает, что в 10А классе 35 человек. Такое индексирование улучшает наглядность программы.

Часто структурному типу присваивается имя в разделе типов, которое затем используется в разделе описания переменных.

```
Type Mas1 = array [1..100] of Integer;
     Mas2 = array [-10..10] of Char;
Var Num: Mas1; Sim: Mas2;
```

До сих пор речь шла об одномерных массивах, в которых типы элементов скалярные.

Многомерный массив в Паскале трактуется как одномерный массив, тип элементов которого также является массивом (массив массивов).

В качестве примера рассмотрим таблицу с информацией о среднемесячных температурах за 10 лет, например с 2001 по 2010 год. Очевидно, для этого удобна прямоугольная (двумерная) таблица, в которой столбцы соответствуют годам, а строки — месяцам.

	Месяц											
Год	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
2001	–23	– 17	– 8,4	6,5	14	18,6	25	19	12.3	5,6	– 4,5	– 19
2001	– 16	– 8,5	–3,2	7,1	8,4	13,8	28,5	21	6,5	2	– 13	– 20
2003	– 9,8	– 14	–9,2	4,6	15,6	21	17,8	20	11,2	8,1	– 16	– 21
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
2010	– 25	– 9,7	– 3,8	8,5	13,9	17,8	23,5	17,5	10	5,7	– 14	– 20

Для обработки такой таблицы в программе следует описать массив:

```
Var Tabl: array[2001..2010] of array[1..12] of Real;
```

Вот примеры обозначения некоторых элементов этого массива:

```
Tabl[2001][1]; Tabl[2005][10]; Tabl[2010][12].
```

Однако чаще употребляется другая, эквивалентная форма обозначения элементов двумерного массива:

```
Tabl[2001,1]; Tabl[2005,10]; Tabl[2010,12].
```

Переменная `Tabl[2001]` обозначает всю первую строку таблицы, т. е. весь массив температур за 2001 год. Другим эквивалентным вариантом приведённому выше описанию является следующее:

```
Type Month = array [1..12] of Real;
Year = array [2001..2010] of Month;
Var Tabl: Year;
```

Наиболее краткий вариант описания данного массива такой:

```
Var Tabl: array[2001..2010, 1..12] of Real;
```

Продолжая по аналогии, можно определить трёхмерный массив как одномерный массив, у которого элементами являются двумерные массивы. Вот пример описания трёхмерного массива:

```
Var A: array[1..10, 1..20, 1..30] of Integer;
```

Это массив, состоящий из $10 \cdot 20 \cdot 30 = 6000$ целых чисел и занимающий в памяти $6000 \cdot 2 = 12\,000$ байтов. В Паскале нет ограничения сверху на размерность массива. Однако в каждой конкретной реализации Паскаля ограничивается объем памяти, выделяемый под массивы. В Турбо Паскале это ограничение равно 64 килобайтам.

По аналогии с математикой одномерные числовые массивы часто называют векторами, а двумерные — матрицами.

В Паскале не допускается употребление динамических массивов, т. е. таких, размер которых определяется в процессе выполнения. Изменение размеров массива происходит через изменение в тексте программы и повторную компиляцию. Для упрощения таких изменений удобно определять индексные параметры в разделе констант:

```
Const Imax = 10; Jmax = 20;
Var Mas: array[1..Imax, 1..Jmax] of Integer;
```

Теперь для изменения размеров массива `Mas` и всех операторов программы, связанных с этими размерами, достаточно отредактировать только одну строку в программе — раздел констант.

Действия над массивом как единым целым. Такие действия допустимы лишь в двух случаях:

- присваивание значений одного массива другому;
- применение к массивам операций отношения «равно», «не равно».

В обоих случаях массивы должны иметь одинаковые типы (тип индексов и тип элементов).

Пример 1

```
Var P,Q: array[1..5, 1..10] of Real;
```

При выполнении операции присваивания

```
P:=Q
```

все элементы массива P станут равными соответствующим элементам массива Q.

Пример 2

Как уже отмечалось, в многомерных массивах переменная с индексом может обозначать целый массив. Тогда, если массив Tabl описан так:

```
Type mas=array [1..12] of Real;
Var Tabl: array[2001..2010] of mas;
```

и в нём требуется данные за 2009 год сделать такими же, как за 2001 год (девятой строке присвоить значение первой строки), то это можно сделать одним присваиванием:

```
Tabl[2009]:= Tabl[2001].
```

А если нужно поменять местами значения этих строк, то это делается через третью переменную того же типа:

```
P:=Tabl[2009]; Tabl[2009]:=Tabl[2001]; Tabl[2001]:=P;
```

где P описана так:

```
Var P: mas;
```

Ввод и вывод массивов производится покомпонентно. Вот примеры ввода с клавиатуры значений одномерного и двумерного массивов:

```
For I:=1 To 12 Do
  ReadLn(T[I]);
For I:=1 To Imax Do
  For J:=1 To Jmax Do
    ReadLn(Mas[I, J]);
```

Здесь каждое следующее значение будет вводиться с новой строки. Для построчного ввода используется оператор Read.

Аналогично в цикле по индексной переменной организуется вывод значений массива на экран. Например:

```
For I:=1 To 12 Do Write(T[I]:8:4);
```

Напомним, что модификатор формата 8:4 означает вывод числа в формате с фиксированной точкой в 8 позиций, из которых в 4-х последних позициях размещается дробная часть.

Следующий фрагмент программы организует построчный вывод матрицы на экран:

```
For I:=1 To Imax Do
Begin
  For J:=1 to Jmax Do
    Write(Mas[I,J]:6);
  WriteLn
End;
```

После вывода очередной строки матрицы оператор Writeln без параметров переведет курсор в начало новой строки. Следует заметить, что в последнем примере матрица на экране будет получена в естественной форме прямоугольной таблицы, если Jmax не превышает 12 (подумайте, почему).

Для массивов большого размера удобно производить **ввод значений из текстового файла**, заранее подготовив такой файл с исходными данными. Пусть в текстовом файле с именем matr.txt с помощью текстового редактора записана следующая числовая матрица размером 4×4:

```
5   7  10   3
3   2   1  23
7  12   6  10
9   2   6  14
```

В следующей программе производится ввод этой матрицы в двумерный массив M:

```
Var M: array[1..4,1..4] of integer;
    i, j: byte;
    F1: text; {Файловая переменная}
Begin
  assign(F1, 'matr.txt'); {Связывание F1 с файлом matr.txt}
  Reset(F1);              {Открытие файла для чтения}
  For i:=1 To 4 Do
  Begin
    For j:=1 To 4 Do
      Read(F1, M[i,j]); {Последовательное чтение}
                        {из одной строки}
      Readln(F1)    {Переход к следующей строке}
  End;
  Close(F1);    {Закрытие файла}
  ...
```

Система основных понятий

<table>
<tr><td colspan="2">Массив — переменная величина регулярного типа</td></tr>
<tr><td colspan="2">Регулярный тип — структурный тип данных, представляющих собой совокупность пронумерованных однотипных величин</td></tr>
<tr><th>Описание массива</th><th>Идентификация элементов массива</th></tr>
<tr><td><code>Var <идентификатор>: array[<тип индекса>] of <тип компонентов></code>
<code><тип индекса></code> — любой порядковый тип, кроме <code>integer</code>; тип компонентов — любой простой или структурный тип</td><td><code><идентификатор массива>[<индексы элемента>]</code>
Для одномерного массива индекс — одно значение, для многомерного массива — множество значений</td></tr>
<tr><th colspan="2">Действия над массивом как единым целым</th></tr>
<tr><td>Присваивание однотипных массивов</td><td>Отношения «равно», «не равно» для однотипных массивов</td></tr>
<tr><td colspan="2">Ввод/вывод массивов производится покомпонентно с клавиатуры или из файла</td></tr>
</table>

Вопросы и задания

1. Что такое регулярный тип данных? Что такое массив?

2. Какие типы допустимы для индексов массива?

3. Как в Паскале трактуется многомерный массив?

4. Какие действия можно выполнять над массивом как единым целым?

☑ **5.** Дан вектор $\{z_i\}$, $i = 1,...,50$. Составьте программу ввода значений и вычисления длины этого вектора по следующей формуле:

$$L = \sqrt{z_1^2 + z_2^2 + \ldots + z_{50}^2}.$$

☑ **6.** Даны значения массива $\{a_i\}$, $i = 0, \ldots, 10$ и переменной x. Составьте программу вычисления алгебраического многочлена 10-й степени по формуле Горнера:

$$a_{10}x^{10} + a_9x^9 + \ldots + a_1x + a_0 = ((\ldots(a_{10}x + a_9)x + a_8)x + \ldots + a_1)x + a_0.$$

☑ **7.** Значения вектора $\{x_i\}$, $i = 1, \ldots, 20$ вводятся из текстового файла. Требуется подсчитать количество его элементов, значения которых лежат в интервале [0, 1].

☑ **8.** Введите из текстового файла целочисленную матрицу размером 6 × 8. Переверните матрицу, поменяв 1-ю строку с 6-й, 2-ю — с 5-й, 3-ю — с 4-й, и запишите полученную матрицу в другой файл.

☑ **9.** Введите с клавиатуры одномерный числовой массив из 9 элементов. Сверните его в матрицу размером 3 × 3, разместив первую тройку элементов в 1-й строке матрицы, 2-ю тройку — во второй строке, 3-ю тройку — в третьей строке.

☑ **10.** Введите с клавиатуры построчно в двумерный массив числовую матрицу размером 4 × 4. Разверните её по столбцам в одномерный массив. Запишите массив в текстовый файл.

2.2.10 Типовые задачи обработки массивов

Заполнение массива

Значения массива могут задаваться вводом с клавиатуры, чтением из файла или вычислением в программе. В некоторых задачах статистического характера требуется заполнять массивы случайными числами.

Пример 1

Заполнить массив равномерно распределёнными целыми случайными числами в диапазоне от 0 до 100.

```
Var X: array[1..20] of integer; i: integer;
Begin
  Randomize;
  For i:=1 To 20 Do X[i]:=Random(100);
  For i:=1 To 20 Do write(X[i]:4)
End.
```

Стандартная функция `Random(x)` возвращает псевдослучайное целое число в диапазоне от 0 до $x - 1$. Такие функции называют **датчиками случайных чисел**. Стандартная процедура `Randomize` случайным образом устанавливает начальное состояние датчика. Благодаря этому при повторном запуске программы последовательности чисел, вырабатываемые датчиком, повторяться не будут.

Если требуется изменить диапазон случайных чисел, то это всегда можно сделать путем сдвига. Например, если нужно получить числа в диапазоне от –50 до 50, то в программе перепишется оператор присваивания:

```
X[i]:=Random(100)-50;
```

Для получения вещественных случайных чисел используется функция `Random` без аргументов. Она возвращает случайные дробные значения в диапазоне [0, 1). С помощью сдвига и множителя эти значения можно привести к любому диапазону. Например, следующее выражение будет вычислять случайное вещественное число в диапазоне значений от –5 до 5: `10*Random - 5`.

☑ **Пример 2**

Заполнить верхнетреугольную матрицу указанного вида и вывести её на экран.

Матрица:

```
1 2 3 4
0 2 3 4
0 0 3 4
0 0 0 4
```

Программа решения задачи:

```
Var M: array[1..4, 1..4] of integer;
    i, j: integer;
Begin
  For i:=1 To 4 Do
    For j:=1 To 4 Do
      If j<I Then M[I,j]:=0 Else M[I,j]:=j;
  For i:=1 To 4 Do
  Begin
    For j:=1 To 4 Do
      Write (M[i,j]:3);
    Writeln
  End;
End.
```

Поиск в массиве

Алгоритмы поиска рассматривались в § 1.7.6 учебника 10 класса. Напомним, что если значения массива не упорядочены по возрастанию или убыванию, то поиск искомой величины приходится выполнять методом последовательного просмотра всего массива. В упорядоченном массиве поиск можно ускорить, применив алгоритм бинарного поиска.

Ещё одной часто решаемой задачей является поиск в массиве наибольшего или наименьшего значения.

☑ **Пример 3**

Оформить в виде процедуры подпрограмму поиска максимального элемента в одномерном массиве. Заполнить одномерный массив случайными числами (как в примере 1). С помощью процедуры найти в нём максимальное значение и индекс его первого вхождения этого значения в массив.

```
Const n=20;
Type: mas=array[1..n] of integer;
Var X: array[1..n] of integer; i,Xmax,imax: integer;

{Начало описания процедуры}
Procedure MaxArray(Var A: mas]; Var MaxA, k: integer);
Var j: integer;
Begin
  MaxA:=A[1]; k:=1;
  For j:=2 To n Do
    If A[j]>MaxA Then
    Begin MaxA:=A[j]; k:=j End
End; {Конец описания процедуры}

Begin
  Randomize;
  For i:=1 To n Do X[i]:=Random(100); {Заполнение массива}
  MaxArray(X, Xmax,imax);             {Обращение к процедуре}
  Write('Xmax=', Xmax, 'imax=', imax) {Вывод результатов}
End.
```

Процедура MaxArray имеет три параметра: A — исходный массив, MaxA — переменная для найденного максимального значения, k — переменная для индекса максимального значения. При обращении к процедуре им соответствуют фактические параметры: X, Xmax, imax. Размер массива определяется глобальной константой n, значение которой используется как в основной программе, так и в процедуре.

Сортировка массива

В § 1.7.7 учебника для 10 класса рассматривались два алгоритма сортировки: сортировка методом выбора максимального элемента и сортировка методом пузырька. Используем алгоритм метода пузырька для построчной сортировки матрицы.

☑ **Пример 4**

Заполнить квадратную матрицу случайными целыми числами. Упорядочить каждую строку матрицы по возрастанию значений элементов.

```
Const n=4;
Type: vector=array[1..n] of integer;
Var Matr: array[1..n] of vector; k, l: integer;

{Начало процедуры сортировки}
Procedure SortVector(Var A: vector);
Var i, j, X: integer; Flag: boolean;
Begin
  Flag:=true; i:=1;
  While (i<=n-1) and Flag Do
  Begin
    Flag:=false;
    For j:=1 To n-i Do
      If A[j]>A[j+1] Then
      Begin X:=A[j]; A[j]:=A[J+1]; A[j+1]:=X; flag:=true End;
    i:=i+1
  End
End; {Конец процедуры сортировки}

{Начало процедуры вывода матрицы}
Procedure PrintMatr(Var M: array[1..n] of vector);
Var i, j: integer;
Begin
  For i:=1 To n Do
  Begin
    For j:=1 To n Do Write (M[i][j]:3);
    Writeln
  End
End;{Конец процедуры печати матрицы}

{Основная программа}
Begin
  Randomize;
```

```
    {Заполнение матрицы}
    For k:=1 To n Do
      For l:=1 To n Do
        Matr[k][l]:=Random(10);
    {Вывод исходной матрицы}
    Writeln('Исходная матрица:'); PrintMatr(Matr);
    {Построчная сортировка}
    For k:=1 To n Do SortVector(Matr[k]);
    {Вывод отсортированной матрицы}
    Writeln('Отсортированная матрица:'); PrintMatr(Matr)
  End.
```

В этой программе имеются две процедуры: `SortVector` — сортировка одномерного массива и `PrintMatr` — печать матрицы. `Matr` — сортируемая квадратная матрица размером n × n. Каждая строка матрицы — это одномерный массив (вектор). Сортировка строк производится поочередным обращением к процедуре `SortVector` с указанием в переменной k номера сортируемой строки. После заполнения матрицы случайными числами она выводится на экран. После сортировки ее вывод повторяется. Чтобы не программировать дважды вывод матрицы, создана процедура `PrintMatr`. В основной программе к ней дважды происходит обращение. Исполнение этой программы на компьютере дало следующие результаты:

```
Исходная матрица:
4 1 3 9
4 2 9 0
8 4 3 7
3 6 2 9
Отсортированная матрица:
1 3 4 9
0 2 4 9
3 4 7 8
2 3 6 9
```

Система основных понятий

Задачи обработки массива	
Типовые задачи: • заполнение массива путём ввода, вычисления, случайными числами; • поиск в массиве: заданного значения, максимального или минимального значения; • сортировка массива	
Датчик случайных равномерно распределённых чисел	
Целые числа: случайное число в диапазоне $[0, x - 1]$:	Вещественные числа: случайное число в диапазоне $(0, 1)$:
Функция `Random(x)`, где `x` — целое число	Функция `Random` (без аргумента)

Вопросы и задания

1. Какими способами можно заполнить массив значениями?
2. Что такое датчик случайных чисел?
3. Как можно получать целые случайные числа в диапазоне от –50 до 0?
4. Как можно получать вещественные случайные числа в диапазоне от 2,5 до 10?
5. ☑ Даны два вектора $\{x_i\}$, $\{y_i\}$, $i = 1, \ldots, 10$, упорядоченные по возрастанию. Соедините их в один вектор $\{z_i\}$, $i = 1, \ldots, 20$ так, чтобы сохранилась упорядоченность.
6. ☑ Дан массив, состоящий из 100 целых чисел:
   а) вывести все числа, которые встречаются в этом массиве по нескольку раз;
   б) вывести все числа, которые встречаются в массиве только по одному разу.
7. ☑ В целочисленной матрице размером 10×10 найдите максимальное значение и индексы всех элементов, равных ему.
8. ☑ Матрицу размером 5×10 заполните случайными двоичными цифрами (0 и 1). Определите номер строки с наибольшим количеством нулей.
9. ☑ В двоичной матрице размером 10×10 найдите совпадающие строки.

Компьютерный практикум. Раздел «Программирование»

2.2.11 Метод последовательной детализации

В соответствии с методологией структурного программирования метод последовательной детализации является основным подходом при проектировании сложных алгоритмов. Суть метода состоит в следующем:

1) анализируется исходная задача, в ней выделяются подзадачи, строится иерархия таких подзадач (рис. 2.7);
2) составляются алгоритмы (или программы), начиная с основного алгоритма (основной программы), далее — вспомогательные алгоритмы (подпрограммы) с последовательным углублением уровня.

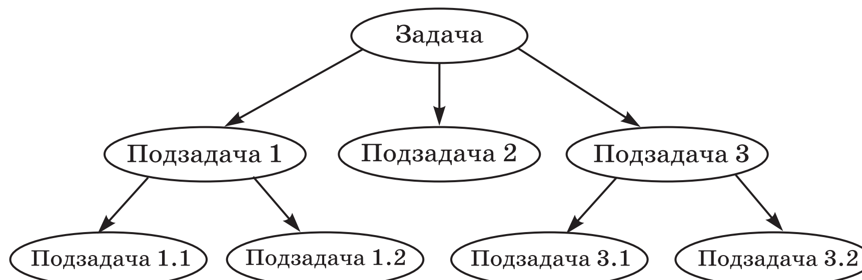


Рис. 2.7. Иерархия подзадач

Проиллюстрируем применение метода последовательной детализации на несложном примере.

Пример 1

Вычислить площадь выпуклого N-угольника, заданного координатами своих вершин (рис. 2.8).

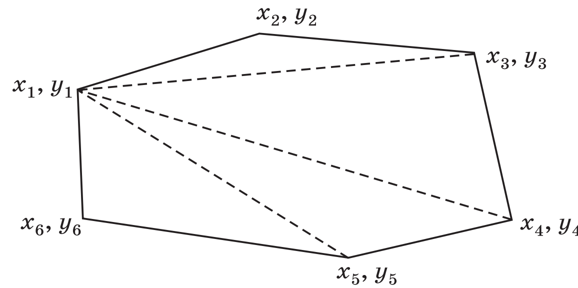


Рис. 2.8. Выпуклый многоугольник

Метод решения задачи заключается в следующем: выпуклый N-угольник разбивается диагональными линиями, выходящими из одной вершины на $N - 2$ треугольника. Площадь многоугольника вычисляется как сумма площадей треугольников. Площади треугольников вычисляются по формуле Герона:

$$S = \sqrt{p(p-a)(p-b)(p-c)},$$

где p — полупериметр треугольника, a, b, c — длины сторон треугольника. Длины сторон вычисляются по формуле, следующей из теоремы Пифагора. Например, длина отрезка между точками с координатами (x_1, y_1), (x_2, y_2) равна:

$$L_{12} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Основной задачей является вычисление площади многоугольника. Подзадачей для основной задачи является вычисление площади треугольника. Подзадачей вычисления площади треугольника является вычисление длины отрезка. Соотношение между этими задачами и соответствующими программами показано на рис. 2.9.

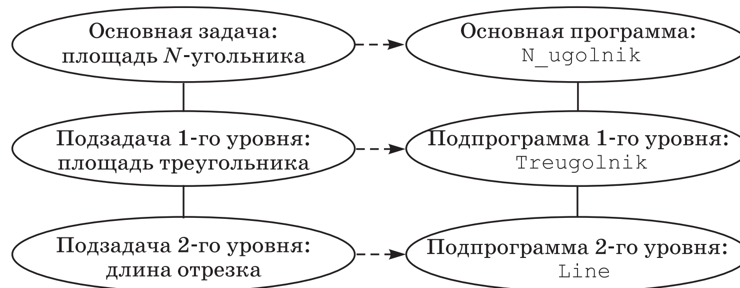


Рис. 2.9. Связи: структура задачи — структура программы

Организация данных: исходные данные — координаты вершин N-угольника будут храниться в двух массивах: X[1..N], Y[1..N].

На первом шаге детализации составляется основная программа без подробного программирования используемых в ней подпрограмм первого уровня. Однако должны быть записаны интерфейсы подпрограмм первого уровня. **Интерфейс** здесь — это заголовок подпрограммы: имя и список формальных параметров. Интерфейсы необходимы для того, чтобы в основной программе можно было бы организовать обращения к подпрограммам первого уровня детализации.

В программе об N-угольнике подпрограмму Treugolnik сделаем процедурой.

```
Program N_ugolnik;
Const N=6;
Var X, Y: array[1..N] of real; S, SNugol: real; i: integer;
Procedure Treugolnik(Var x1, y1, x2, y2, x3, y3, R: real);
{блок процедуры не записывается}
Begin
  {Ввод координат вершин многоугольника}
  For i:=1 To N Do
    Begin
      Write('X[', I, ']='); Readln(X[i]);
      Write('Y[', I, ']='); Readln(Y[i])
    End;
  SNugol:=0; {Переменная для вычисления площади фигуры}
  {Суммирование площадей треугольников}
  For i:=2 To N-1 Do
  Begin
    Treugolnik(X[1], Y[1], X[i], Y[i], X[i+1], Y[i+1], S);
    SNugol:=SNugol+S
  End;
  Writeln('Площадь фигуры =',Snugol)
End.
```

На втором шаге детализации запрограммируем процедуру Treugolnik. В разделе подпрограмм этой процедуры запишем лишь интерфейс подпрограммы Line, которую сделаем функцией.

```
Procedure Treugolnik(Var x1, y1, x2, y2, x3, y3, R: real);
Var L1, L2, L3, p: real;
Function Line(Var Xa, Ya, Xb, Yb: real): real;
{Блок функции не записывается}
Begin
  L1:= Line(x1,y1,x2,y2); {Длина 1-й стороны}
  L2:= Line(x2,y2,x3,y3); {Длина 2-й стороны}
  L3:= Line(x1,y1,x3,y3); {Длина 3-й стороны}
  P:=(L1+L2+L3)/2;        {Полупериметр}
  R:=sqrt(p*(p-L1)*(p-L2)*(p-L3)) {Площадь треугольника}
End;
```

На третьем шаге детализации запрограммируем функцию Line. Формальные координаты концов отрезка заданы параметрами: (Xa, Ya) — первая точка, (Xb, Yb) — вторая точка.

```
Function Line(Var Xa, Ya, Xb, Yb: real): real;
Begin
  Line:=sqrt(sqr(Xa-Xb)+sqr(Ya-Yb))
End;
```

Из составленных фрагментов собираем окончательный вариант программы.

```
Program N_ugolnik;
Const N=6;
Var X, Y: array[1..N] of real; S, SNugol: real; i: integer;
Procedure Treugolnik(Var x1, y1, x2, y2, x3, y3, R: real);
Var L1, L2, L3, p: real;
Function Line(Var Xa, Ya, Xb, Yb: real): real;
begin
  Line:=sqrt(sqr(Xa-Xb)+sqr(Ya-Yb))
End;
Begin
  L1:= Line(x1,y1,x2,y2);
  L2:= Line(x2,y2,x3,y3);
  L3:= Line(x1,y1,x3,y3);
  P:=(L1+L2+L3)/2;
  R:=sqrt(p*(p-L1)*(p-L2)*(p-L3))
End;
Begin
  For i:=1 To N Do
  Begin
    Write('X[', I, ']='); Readln(X[i]);
    Write('Y[', I, ']='); Readln(Y[i])
  End;
  SNugol:=0;
  For i:=2 To N-1 Do
  Begin
    Treugolnik(X[1], Y[1], X[i], Y[i], X[i+1], Y[i+1], S);
    SNugol:=SNugol+S
  End;
  Writeln('Площадь фигуры =', SNugol)
End.
```

В этой программе значение N может быть любым, начиная с 3, т. е. N ≥ 3. Константа N описана глобально в основной программе, поэтому область этого описания распространяется на все подпрограммы. Все остальные величины в подпрограммах определены локально.

Показанный в рассмотренном примере способ построения программы называют еще **программированием «сверху вниз»**: начиная от основной программы, последовательно переходя к подпрограммам всё более глубокого уровня детализации.

Работу полученной программы можно проверить на простом примере. Пусть N = 4. Вычислим площадь квадрата с длинами сторон, равными 2 и следующими координатами вершин:

```
X[1]=0, X[2]=0, X[3]=2, X[4]=2
Y[1]=0, Y[2]=2, Y[3]=2, Y[4]=0
```

После ввода этих значений в результате получим:

```
Площадь фигуры =4
```

Применение метода последовательной детализации позволяет разделить работу над большим программным проектом между несколькими программистами. Один человек — руководитель группы проектирует многоуровневую структуру алгоритма и составляет основную программу, а написание подпрограмм поручается другим членам группы. Для согласования работы программ договариваются лишь о интерфейсах: именах и параметрах подпрограмм. А внутреннее устройство подпрограммы — дело программиста, ее составляющего. При составлении больших проектов подпрограммы объединяются в модули.

Система основных понятий

Метод последовательной детализации
Метод последовательной детализации — один из основных методов структурного программирования, заключающийся в разработке сложных алгоритмов путём построения иерархии подзадач
Программирование «сверху вниз»: первой составляется основная программа, затем подпрограмма 1-го уровня детализации, затем — 2-го уровня и т. д. Подпрограммы последнего уровня содержат только простые команды, не вызывающие других подпрограмм
Интерфейс подпрограммы: это её заголовок, позволяющий организовать обращение к подпрограмме (имя, тип, формальные параметры)

Вопросы и задания

1. В чём заключается основная идея метода последовательной детализации?
2. Каким образом применение метода последовательной детализации позволяет организовать работу коллектива программистов над большим проектом?
3. Что такое интерфейс подпрограммы?

Компьютерный практикум. Раздел «Программирование»

2.2.12 Символьный тип данных

Величина типа «символ» может принимать значения любых символов компьютерного алфавита. Символьная величина занимает 1 байт памяти, в котором хранится код этого символа, соответствующий используемой кодовой таблице. Заметим, что в Delphi наряду с однобайтовой кодировкой символов используется и двухбайтовая.

Символьная константа записывается между апострофами. Например: `'R'`, `'+'`, `'9'`, `']'`.

Символьной тип называется `char`. Пример описания символьных переменных:

```
Var c1, c2: char;
```

Символьный тип относится к порядковым типам данных. Из этого следует:

- символы — упорядоченное множество;
- у каждого символа в этом множестве есть свой порядковый номер;
- между символами работает соотношение «следующий – предыдущий».

Порядковый номер символа — это его десятичный код, который лежит в диапазоне от 0 до 255. Например, в кодовой таблице ASCII десятичный код латинской буквы 'A' равен 65, а цифры '5' — 53. О стандартах кодирования символов подробно рассказывалось в § 1.4.2 учебника для 10 класса.

Функция `Ord(x)`

`Ord(x)` — функция от аргумента порядкового типа, которая возвращает порядковый номер значения `x` в этом типе данных. Если `x` — символьная величина, то результатом функции будет десятичный код `x` в кодовой таблице. Например:

```
Ord('A')=65, Ord('5')=53
```

Функция `Chr(x)`

`Chr(x)` — функция от целочисленного аргумента, результатом которой является символ с кодом, равным x. Например:

```
Chr(65)='A', Chr(53)='5'
```

Поскольку коды символов лежат в диапазоне от 0 до 255, то желательно тип `x` определять либо как `byte`, либо как ограниченный тип 0..255. В § 1.4.2 учебника для 10 класса содержится программа получения таблицы кодировки символов с номерами от 20 до 255. В этой программе последовательность кодов и соответствующих им символов получается в следующем цикле:

```
For kod:=20 To 255 Do Write(Chr(kod):3, kod:4);
```

Напомним, что коды меньше 20 являются управляющими и на экране не отражаются.

Функция Chr является обратной к функции Ord. Отсюда следует: Chr(Ord(x))=x. Например: Chr(Ord('A'))='A'.

Принцип последовательного кодирования алфавитов

В любой кодовой таблице выполняется принцип последовательного кодирования латинского (английского) алфавита и алфавита десятичной системы счисления. Это важное обстоятельство, которое часто учитывается в программах обработки символьной информации.

При выполнении операций отношений, применительно к символьным величинам, учитываются коды этих величин. Чем больше значение кода, тем символ считается больше. Истинными являются следующие отношения: 'A'<'B', 'Z'>'Y', 'a'>'A'. Значение символьной переменной C является заглавной латинской буквой, если истинно логическое выражение:

```
(C>='A') and (C<='Z')
```

Значение символьной переменной C является цифрой, если истинно логическое выражение:

```
(C>='0') and (C<='9').
```

В латинском алфавите 26 букв. Поэтому разница между кодами букв 'Z' и 'A', а также 'z' и 'a' равна 25.

☑ **Задача 1**

С помощью датчика случайных чисел заполнить массив Sim[0..10] строчными английскими буквами. Затем массив отсортировать в алфавитном порядке.

```
Uses CRT;
Var Sim: array[0..10] of char;
    C: char; i, k: integer;
Begin
  ClrScr;
  Randomize; {Заполнение массива случайными буквами}
  Writeln('Исходный массив:');
  For i:= 0 To 10 Do
  Begin
    Sim[i]:=Chr(Random(26)+Ord('a'));
    Write(Sim[i])
  End;
  Writeln;
  {Сортировка методом пузырька}
  For i:=0 To 9 Do
    For k:=0 To 9-i Do
      If Sim[k]>Sim[k+1] Then
      Begin
        C:=Sim[k]; Sim[k]:=Sim[k+1]; Sim[k+1]:=C
      End;
```

```
  Writeln('Отсортированный массив:');
  For i:=0 To 10 Do Write(Sim[i]);
End.
```

При тестировании программы было получено:

```
Исходный массив:
gnkbeqgmsin
Отсортированный массив:
beggikmnnqs
```

☑ **Задача 2**

На вход программе подаются строчные английские буквы. Ввод этих символов заканчивается точкой (другие символы, отличные от '.' и букв 'a'..'z', во входных данных отсутствуют). Написать программу на Паскале, которая будет выводить буквы, встречающиеся во входной последовательности, в порядке уменьшения частоты их встречаемости. Каждая буква должна быть выведена один раз. Точка при этом не учитывается[1].

Идея алгоритма. Формируется массив символов английского алфавита от 'a' до 'z'. Дадим имя массиву Alf, а элементы пронумеруем от 0 до 25. В другом массиве Schet[0..25] будем вести счётчики повторений каждой буквы в последовательности вводимых символов. В Schet[0] — счетчик 'a', в Schet[1] — счетчик 'b' и т. д. В начале счётчики обнуляются.

В цикле посимвольно вводятся данные и подсчитывается количество каждой буквы в своём счётчике. Цикл заканчивается, когда будет введена точка. Затем сортируется массив счётчиков по убыванию значений. Применяется алгоритм сортировки методом пузырька. Одновременно с перестановками в массиве Schet производятся аналогичные перестановки в массиве Alf. В конце по порядку выводятся оба эти массива, при этом пропускаются счётчики, равные нулю, и соответствующие им буквы.

```
Program Bukvy;
Uses CRT;
Var Alf: array[0..25] of char;
    Schet: array[0..25] of byte;
    X, I, k: byte;
    C: char;
Begin
  ClrScr;
  {Заполнение массива Alf буквами алфавита и массива Schet}
  {нулями}
  For i:=0 To 25 Do
  Begin Alf[i]:=Chr(Ord('a')+i); Schet[i]:=0 End;
  Readln(c); {Ввод первого символа}
  {Циклический ввод символов до точки}
  While C<>'.' Do
```

[1] Задача взята из заданий ЕГЭ по информатике 2008 г.

```
    Begin
      K:=Ord(C)-Ord('a');   {Номер символа C}
      Schet[k]:=Schet[k]+1;  {+1 в счетчик данного символа}
      Readln(c);       {Ввод очередного символа}
    End;
    {Сортировка массива Schet методом пузырька}
    For i:=0 To 24 Do
      For k:=0 To 24-i Do
        If Schet[k]<Schet[k+1] Then
        Begin
          X:=Schet[k]; Schet[k]:=Schet[k+1]; Schet[k+1]:=X;
          C:=Alf[k]; Alf[k]:=Alf[k+1]; Alf[k+1]:=c
        End;
    For i:=0 To 25 Do
      If Schet[i]>0 Then Writeln(Alf[i]:2,'-', Schet[i]:3)
  End.
```

При тестировании программы была введена последовательность символов:

```
asadsdghka.
```

В результате получено:

```
a-3 d-2 s-2 g-1 h-1 k-1
```

Система основных понятий

Символьный тип данных
Величины символьного типа (char): константы и переменные, принимающие значения символов компьютерного алфавита
1 символ занимает 1 байт памяти (в 8-битовых кодировках)
Ord(x) — функция от аргумента порядкового типа, которая возвращает порядковый номер значения x в этом типе данных. Если x — символьная величина, то функция возвращает код символа
Chr(x) — функция от целочисленного аргумента, результатом которой является символ с кодом, равным x
В любой кодовой таблице выполняется **принцип последовательного кодирования** латинского (английского) алфавита и алфавита десятичной системы счисления

Вопросы и задания

1. Как в программе на Паскале обозначаются символьные константы и переменные?
2. С помощью какой стандартной функции определяется код символа?
3. С помощью какой стандартной функции можно определить символ по его коду?

4. Что такое принцип последовательного кодирования алфавитов? Приведите примеры алгоритмов, где он может быть использован.

☑ 5. Определите результаты вычисления выражений (типы и значения):

1) `Chr(Ord('B'));`
2) `Ord('A')-Ord('Z');`
3) `Ord('A')-Ord('a')= Ord('Z')-Ord('z');`
4) `Ord('9')-Ord('0');`
5) `Chr(Ord('a')+Ord('R')-Ord('r')).`

2.2.13 Строки символов

Рассмотрим еще один структурный тип данных — строковый тип. **Строковый тип данных** был введен в Турбо Паскале. Он позволяет программировать обработку слов, предложений, текстов.

Строка — это последовательность символов. Каждый символ занимает 1 байт памяти (код ASCII). Количество символов в строке называется её длиной. Длина строки может находиться в диапазоне от 0 до 255. Строковые величины могут быть константами и переменными.

Строковая константа записывается как последовательность символов, заключенная в апострофы. Например:

```
' Язык программирования ПАСКАЛЬ'
' IBM PC - computer',
'33-45-12'
```

Строковая переменная описывается в разделе описания переменных следующим образом:

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

Например:

```
Var Name: String[20]
```

Параметр длины может и не указываться в описании. В таком случае подразумевается, что он равен максимальной величине — 255. Например:

```
Var slovo: String
```

Строковая переменная занимает в памяти на 1 байт больше, чем указанная в описании длина. Дело в том, что один (нулевой) байт содержит значение *текущей длины строки*. Если строковой переменной не присвоено никакого значения, то её текущая длина равна нулю. По мере заполнения строки символами её текущая длина возрастает, но она не должна превышать максимальной по описанию величины.

Символы внутри строки индексируются (нумеруются), начиная с единицы. Каждый отдельный символ идентифицируется именем строки с индексом, заключенным в квадратные скобки. Например:

```
Name[5], Name[i], slovo[k+l].
```

Значение индекса может быть задано положительной константой, переменной, выражением целочисленного типа. Оно не должно выходить за границы описания.

Тип `String` и стандартный тип `Char` совместимы: строки и символы могут употребляться в одних и тех же выражениях.

Строковые выражения строятся из строковых констант, переменных, функций и знаков операций. Над строковыми данными допустимы операции сцепления и операции отношения.

Операция сцепления (+) применяется для соединения нескольких строк в одну результирующую строку. Сцеплять можно как строковые константы, так и переменные.

Например:

```
'ЭВМ'+' IBM'+' PC'
```

В результате получится строка:

```
'ЭВМ IBM PC'
```

Длина результирующей строки не должна превышать 255.

Операции отношения : =, < , > , <= , >= , < > производят сравнение двух строк, в результате чего получается логическая величина (`true` или `false`). Операция отношения имеет более низкий приоритет, чем операция сцепления. Сравнение строк производится слева направо до первого несовпадающего символа, и та строка считается больше, в которой первый несовпадающий символ имеет больший номер в таблице символьной кодировки.

Если строки имеют различную длину, но в общей части символы совпадают, считается, что более короткая строка меньше, чем более длинная. Строки равны, если они полностью совпадают по длине и содержат одни и те же символы.

Пример

Выражение:	Результат:
'cosm1'<'cosm2'	True
'pascal'>'PASCAL'	True
'Ключ_''Ключ'	True
'MS DOS'='MS DOS'	True

Функции и процедуры

Функция `Copy(S, Poz, N)` выделяет из строки `S` подстроку длиной `N` символов, начиная с позиции `Poz`. `N` и `Poz` — целочисленные выражения.

Пример

Значение S:	Оператор:	Результат:
'ABCDEFG'	Copy(S, 2, 3)	'BCD'
'ABCDEFG'	Copy(S, 4, 4)	'DEFG'

Функция Concat(S1,S2,...,SN) выполняет сцепление (конкатенацию) строк S1, ..., SN в одну строку.

Пример

Выражение:	Результат:
Concat('AA', 'XX', 'Y')	'AAXXY'

Функция Length(S) определяет текущую длину строки S. Результат — значение целочисленного типа.

Пример

Значение S:	Оператор:	Результат:
'test-5'	Length(S)	6
'(A+B)*C'	Length(S)	7

Функция Pos(S1, S2) обнаруживает первое появление в строке S2 подстроки S1. Результат — целое число, равное номеру позиции, где находится первый символ подстроки S1. Если в S2 не обнаружена подстрока S1, то результат равен 0.

Пример

Значение S2:	Оператор:	Результат:
'abcdef'	Pos('cd', S2)	3
'abcdcdef'	Pos('cd', S2)	3
'abcdef'	Pos('k', S2)	0

Процедура Delete(S, Poz, N) — удаление N символов из строки S, начиная с позиции Poz.

Пример

Исходное значение:	Оператор:	Конечное значение:
'abcdefg'	Delete(S,3,2)	'abefg'
'abcdefg'	Delete(S,2,6)	'a'

В результате выполнения процедуры уменьшается текущая длина строки в переменной S.

Процедура Insert(S1,S2,Poz) — вставка строки S1 в строку S2, начиная с позиции Poz.

Пример

Начальное S2:	Оператор:	Конечное S2:
'ЭВМ PC'	Insert('IBM-',S2,5)	'ЭВМ IBM-PC'
'Рис.2'	Insert('N',S2,6)	'Рис.N2'

Примеры программ обработки строк

Пример 1

Составить программу, формирующую символьную строку, состоящую из N звездочек (N — целое число, $1 \le N \le 255$).

```
Program Stars;
Var A: String;
    N, I: Byte;
Begin
  Write ('Введите число звездочек');
  ReadLn(N);
  A:='';
  For I:=1 To N Do
    A:=A+'*';
  WriteLn(A)
End.
```

Здесь строковой переменной A вначале присваивается значение **пустой строки**, обозначаемой двумя апострофами (''). Затем к ней присоединяются звёздочки.

Пример 2

В символьной строке подсчитать количество цифр, предшествующих первому символу '!'.

```
Program C;
Var S: String;
    K, I: Byte;
Begin
  WriteLn('Введите строку');
  ReadLn(S);
  K:=0; I:=1;
  While (I <= Length(S)) And (S[I] <> '!') Do
  Begin
    If (S[I] >= '0') And (S[i] <= '9')
    Then K:=K+1;
    I:=I+1
  End;
  WriteLn ('Количество цифр до символа "!" равно', K)
End.
```

В этой программе переменная K играет роль счетчика цифр, а переменная I – роль параметра цикла. Цикл закончит выполнение при первом же выходе на символ '!' или, если в строке такого символа нет, то при выхо-

де на конец строки. Символ `S[I]` является цифрой, если истинно отношение: `'0'<S[I]<'9'`.

Пример 3

В этом примере запрограммируем одну из функций текстового редактора: выравнивание строки по ширине. Например, дан текст, состоящий из четырех строк:

```
Буря мглою небо кроет,
Вихри снежные крутя.
То как зверь она завоет,
То заплачет, как дитя.
```

Строки выровнены по левому краю. Пусть длина строки должна быть равна `L`. Нужно путём равномерной вставки пробелов между словами растянуть строки таким образом, чтобы последний символ каждой строки оказался в позиции номер `L`. Например, если `L` = 30, то текст должен приобрести такой вид:

```
Буря    мглою    небо   кроет,
Вихри      снежные      крутя.
То   как   зверь  она  завоет,
То    заплачет,    как   дитя.
```

Будем строить программу методом последовательной детализации. Сначала составим основную программу, которая будет вводить исходный текст в массив `Tekst`, состоящий из `N` строк, последовательно выравнивать строки путем обращения к процедуре `RovStr` и выводить преобразованный текст на экран.

```
Program Editor;
Const N=4; L=30;
Var Tekst: array[1..N] of string;
    i: byte;
Procedure RovStr(Var Str: string);
{Блок процедуры RovStr}
Begin
  {Ввод текста}
  Writeln('Введите текст');
  For i:=1 To N Do Readln(Tekst[i]);
  {Выравнивание строк}
  For i:=1 To N Do RovStr(Tekst[i]);
  {Вывод после выравнивания}
  For i:=1 To N Do Writeln(Tekst[i])
End.
```

Здесь константа `N` указывает на число строк в тексте, константа `L` — длину строки. Процедура имеет один параметр — строку `Str`. Она представляет одновременно и входные и выходные данные. На входе — невыровненная строка, на выходе — эта же строка после выравнивания.

На втором шаге детализации запрограммируем процедуру `RovStr`. Идея алгоритма состоит в следующем: в промежутки между словами

вставляются по очереди дополнительные пробелы до тех пор, пока длина строки не станет равной L.

```
Procedure RovStr(Var Str: string);
Var j: byte;
Begin
  j:=1; {Номер первого символа в слове}
  {Цикл, пока длина строки не станет равной L}
  While length(Str)<L Do
  Begin
    j:=j mod length(Str);
    {Обнаружение конца слова и вставка дополнительного}
    {пробела}
    If (Str[j]<>' ') and (Str[j+1]=' ') Then
    Begin Insert(' ',Str,j+1); j:=j+1 End;
    j:=j+1 {Номер следующего символа}
  End
End;
```

Постарайтесь самостоятельно понять назначение следующего оператора в этой программе:

```
j:=j mod length(Str);
```

Объединив основную программу с процедурой, получаем окончательный текст программы:

```
Program Editor;
Const N=4; L=30;
Var Tekst: array[1..N] of string;
    i: byte;
Procedure RovStr(Var Str: string);
Var j: byte;
Begin
  j:=1;
  While length(Str)<L Do
  Begin
    j:=j mod length(Str);
    If (Str[j]' ') and (Str[j+1]=' ') Then
    Begin Insert(' ',Str,j+1); j:=j+1 End;
    j:=j+1 {Номер следующего символа}
  End
End;
Begin
  Writeln('Введите текст');
  For i:=1 To N Do Readln(Tekst[i]);
  For i:=1 To N Do RovStr(Tekst[i]);
  For i:=1 To N Do writeln(Tekst[i])
End.
```

Система основных понятий

Строки символов
Строка — последовательность символов
Описание строковой переменной: **Var** <идентификатор>: String[<длина строки>] Максимальная длина строки — 255
Обозначение символа в строке: <идентификатор строки>[<индекс>]
Операции над строками: сцепление (+), отношение (=, < , > , <= , >= , < >)
Стандартные функции: Copy(S, Poz, N) — выделение подстроки; Concat(S1, S2, ..., SN) — сцепление (конкатенация) строк; Length(S) — определение текущей длины строки; Pos(S1, S2) — определение первого вхождения подстроки в строку
Стандартные процедуры: Delete(S, Poz, N) — удаление подстроки; Insert(S1, S2, Poz) — вставка подстроки

Вопросы и задания

1. Как в программе обозначается строковая константа, как определяется строковая переменная?
2. Какой может быть максимальная длина строки?
3. Составьте программу получения из слова «дисковод» слова «воск», используя операцию сцепления и функцию Copy.
4. Составьте программу получения слова «правило» из слова «операция», используя процедуры Delete, Insert.
5. В данном слове замените первый и последний символы на '*'.
6. В данном слове произведите обмен первого и последнего символов.
7. К данному слову присоедините столько '!', сколько в нём имеется букв (например, из строки «УРА» надо получить «УРА!!!»).
8. В данной строке вставьте пробел после каждого символа.
9. Переверните введённую строку (например, из 'ДИСК' должно получиться 'КСИД').
10. В данной строке удалите все пробелы.
11. Строка представляет собой запись целого числа. Составьте программу её перевода в соответствующую величину целого типа.

Компьютерный практикум. Раздел «Программирование»

2.2.14 Комбинированный тип данных

Все структурные типы данных, с которыми вы уже познакомились (массивы, строки), представляют собой совокупности однотипных величин. **Комбинированный тип данных** — это структурный тип, состоящий из фиксированного числа компонентов (полей) разных типов.

Комбинированный тип объявляется в программе в разделе типов:

```
Type <имя> = record
     <имя поля 1>: <тип>;
     ...
     <имя поля N>: <тип>
End
```

Поля могут иметь любые типы, в том числе и комбинированный тип.

Например, данные о результатах экзаменов, полученных учеником по трём предметам, могут быть представлены одной величиной комбинированного типа:

```
Type results = record
     Family: string[15]; {Фамилия ученика}
     Rus: 2..5;          {Оценка по русскому языку}
     Alg: 2..5;          {Оценка по алгебре}
     Phiz: 2..5          {Оценка по физике}
End;
```

После этого в разделе переменных следует описание:

```
Var exam: results;
```

Величина комбинированного типа называется **записью**. Элементы записи идентифицируются составными именами следующей структуры:

```
<имя переменной >.<имя поля>
```

Например: `exam.family, exam.rus`

В программе может использоваться массив, элементами которого являются записи.

☑ **Пример 1**

На экзаменационном листе содержатся сведения о результатах экзаменов, сданных 30 учениками класса. Ввести эти данные в компьютер и получить список всех отличников.

В программе используется описание комбинированного типа `result`, приведенное выше. Исходные данные организуются в массив следующей структуры.

```
Var list: array[1..30] of results;
```

После ввода в этот массив исходных данных, следует фрагмент программы:

```
For i:=1 To 30 Do
  If (list[i].rus=5) and (list[i].alg=5) and (list[i].phiz=5)
  Then Writeln(list[i].family);
```

Программа отбирает записи, в которых все поля с оценками равны 5, и выводит соответствующие поля фамилий.

А теперь обсудим проблему: как наиболее удобным способом организовать ввод данных в этой программе? Вводить с клавиатуры неудобно из-за большого объёма данных. При каждом повторном запуске программы нужно начинать ввод сначала. А при отладке это наверняка придется делать многократно. Гораздо удобнее подготовить файл с исходными данными с помощью текстового редактора. После этого без проблем можно повторять ввод многократно. Так и поступим. Подготовим текстовый файл следующего вида:

```
     Таблица успеваемости 10А класса
Фамилия     Русский язык   Алгебра    Физика
Антонов
               4             5          5
Андреева
               5             3          4
Боброва
               5             5          5
...
```

Таблица содержит данные с фамилиями и оценками 30 учеников класса. Обратите внимание на то, что фамилии записываются в отдельных строках. Необходимость этого связана с реализацией алгоритма (см. далее): при вводе символьной строки прочитывается полностью очередная строка текстового файла до признака EOLN. При этом фамилии должны содержать не более 15 символов, а первые оценки (по русскому языку) — располагаться не раньше 16-й позиции в своей строке.

Сохраним этот файл в корневом каталоге логического диска E под именем 10_a.txt. Составим программу с вводом таблицы успеваемости и выводом списка отличников. Фамилии отличников выведем на экран и сохраним в файле с именем Best.txt .

```
Program Examen;
Type results = record
     Fam: string[15];
     Rus: 2..5;
     Alg: 2..5;
     Phiz: 2..5
End;
Var list: array[1..30] of results; {Массив записей}
    i: integer; F1, F2: text;
Begin
  Assign(F1, 'E:\10_a.txt'); {Связывание F1 с файлом 10_a.txt}
```

```
  Assign(F2, 'E:\Best.txt'); {Связывание F2 с файлом Best.txt}
  Reset(F1);                {Открытие файла F1 для чтения}
  Rewrite(F2);              {Открытие файла F2 для записи}
  Readln(F1); Readln(F1);   {Пропуск 2-х строк в файле F1}
  {Цикл ввода из файла F1}
  For i:=1 To 30 Do
    Readln(F1,list[i].Fam,list[i].Rus,list[i].Alg,list[i].Phiz);
  {Цикл отбора отличников и вывода их фамилий}
  For i:=1 To 30 Do
    If (list[i].rus=5) and (list[i].alg=5) and
       (list[i].phiz=5)
    Then
    Begin
      Writeln(list[i].fam);     {Вывод фамилии на экран}
      Writeln(F2, list[i].fam)  {Запись фамилии в файл F2 }
    End;
  Close(F1); Close(F2)    {Закрытие файлов}
End.
```

☑ **Пример 2**

Решая рассмотренную задачу с оценками, можно обойтись без массива записей. Кроме того, можно не ставить ограничения на число учеников в классе. Их число выяснится в процессе чтения файла с таблицей успеваемости. Составим программу, которая кроме вывода списка фамилий отличников подсчитает их количество и процент отличников по отношению к полному составу класса.

```
Program Examen_2;
Type results = record
     Fam: string[15];
     Rus: 2..5;
     Alg: 2..5;
     Phiz: 2..5
End;
Var list: results;  {Одна переменная комбинированного типа}
    I, K: integer;
    F1, F2: text;
Begin
  Assign(F1, 'E:\10_a.txt');
  Assign(F2, 'E:\Best.txt');
  Reset(F1); Rewrite(F2);
  Readln(F1); Readln(F1);
  I:=0; K:=0;  {Инициализация счетчиков}
  {Цикл до конца чтения файла}
  While not EOF(F1) Do
  Begin
    Readln(F1, list.Fam, list.Rus, list.Alg, list.Phiz);
    I:=I+1;    {Подсчёт числа учеников}
    If (list.rus=5) and (list.alg=5) and (list.phiz=5) Then
```

```
    Begin
      Writeln(list.fam);
      Writeln(F2, list.fam);
      K:=K+1;  {Подсчёт числа отличников}
    End
  End;
  Writeln('Из ', I, ' учеников в классе ', K,
          ' отличников, что составляет ', K/I*100:4:1, '%');
  Close(F1); Close(F2)  {Закрытие файлов}
End.
```

В этой программе переменная I используется как счётчик числа учеников, а переменная K — как счётчик числа отличников.

Стандартная логическая функция EOF (end of file) примет значение True, когда процесс чтения из файла дойдет до его конца.

В результате выполнения программы кроме списка отличников на экран выведется строка:

```
Из 30 учеников в классе 10 отличников, что составляет 33,3%.
```

Система основных понятий

Комбинированный тип данных
Комбинированный тип данных — структурный тип, объединяющий разнотипные компоненты (поля) данных
Тип поля: любой простой или структурированный тип (кроме файлового)
Запись — величина комбинированного типа
Идентификация поля записи — составное имя: <имя записи>.<имя поля>

Вопросы и задания

1. Чем комбинированный тип данных отличается от регулярного типа данных (массива)?
2. Что такое запись?
3. Опишите омбинированный тип для записей, содержащих следующие данные учеников: фамилию, имя, год рождения, рост (в сантиметрах), вес (в килограммах).
4. Опишите содержимое текстового файла, из которого будут вводиться данные, соответствующие описанию из предыдущего задания, для нескольких учеников класса (не менее пяти).
5. ☑ Напишите программу, по которой будут введены данные из файла, описанного в предыдущем задании, и выполнена следующая обработка:
   - определение среднего роста и среднего веса всех учеников;
   - вывод на экран и в файл rost.txt списка (фамилии, имена, возраст) учеников, рост которых выше среднего;
   - вывод на экран и в файл ves.txt списка учеников, вес которых ниже среднего.

☑ **6.** Решите предыдущую задачу, не используя в программе массива записей. Подсказка: оператор `Reset` можно использовать в программе многократно для повторного чтения файла, начиная с его первой записи.

Компьютерный практикум. Раздел «Программирование»

2.3. Рекурсивные методы программирования *

2.3.1 Рекурсивные подпрограммы

Частично рекурсивная функция

В § 2.2.7 было введено понятие **рекуррентной последовательности** на примере следующего числового ряда:

$$a_0 = 1,\ a_1 = \frac{1}{1!},\ a_2 = \frac{1}{2!},\ a_3 = \frac{1}{3!},\ \ldots,\ a_n = \frac{1}{n!},\ \ldots$$

Было доказано, что вычисление элементов этого ряда можно производить по формуле:

$$a_i = \begin{cases} 1, & \text{при } i = 0; \\ \dfrac{a_{i-1}}{i}, & \text{при } i > 0. \end{cases}$$

Такая формула называется **одношаговой рекуррентной формулой**. Одношаговость обозначает тот факт, что имеется одно начальное значение и каждое следующее значение ряда вычисляется по одному предыдущему значению.

Будем рассматривать a_n в качестве значений функции $F(n)$ от целого аргумента n. Определение этой функции будет выглядеть так:

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

Определение функции через саму себя (здесь $F(n)$ определяется через $F(n - 1)$) называется в математике рекурсивным определением. Но поскольку при $n = 0$ функция вычисляется другим способом, т. е. не через саму себя, такую функцию называют **частично-рекурсивной функцией**. Вычисление на компьютере частично-рекурсивных функций можно про-

изводить с помощью **рекурсивных алгоритмов**, которые на языках программирования реализуются через **рекурсивные подпрограммы**.

Рекурсивные подпрограммы-функции и процедуры

В описаниях подпрограмм-функций на Паскале допускается использование в теле функции вызова этой же самой функции. Такая подпрограмма-функция называется рекурсивной.

Пример 1

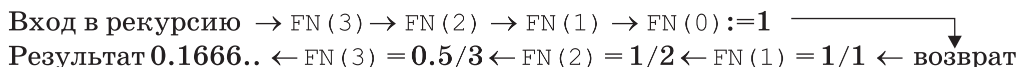
Вот два варианта подпрограммы-функции вычисления $F(n) = 1/n!$

Не рекурсивная функция	Рекурсивная функция

Function FN(n: integer): real;
Var i: integer; F: real;
Begin
  F:=1;
  For i:=1 **To** n **Do** F:=F/i;
  FN:=F
End;
	Function FN(n: integer): real;
	Begin
	  If n=0 **Then** FN:=1
	  Else FN:=FN(n-1)/n
	End;

Первый, не рекурсивный вариант подпрограммы имеет циклическую структуру. Во втором, рекурсивном варианте используется ветвление. На положительной ветви (`Then`) значение функции вычисляется явно (`FN:=1`), на отрицательной ветви (`Else`) происходит обращение функции к себе самой с уменьшенным на единицу значением аргумента.

Пусть в основной программе для вычисления $1/3!$ имеется следующий оператор присваивания: `P:=FN(3)`. При его выполнении произойдет цепочка обращений к рекурсивной функции `FN` с последующим возвратом:



При каждом обращении к функции в специальный раздел памяти помещается необходимая информация для реализации обратного процесса вычислений: адрес команды, к которой надо вернуться, ячейки для размещения промежуточных значений функции и т. д. Такой раздел памяти называется **стеком**. После выхода на граничное значение (присваивание единицы) с помощью стека происходит обратный процесс последовательного вычисления промежуточных значений функции, пока не будет получен окончательный результат. Выполнение рекурсивно определённой функции займет на компьютере больше времени, чем функции без рекурсии.

Вычисление частично-рекурсивной функции может быть реализовано также и в форме рекурсивно определённой процедуры. Ниже показан пример программы, в которой описана рекурсивная процедура вычисления $1/n!$. С помощью этой процедуры вычисляется значение $1/5!$.

```
Var X: real;
Procedure FN(n: integer; Var F: real);
Begin
  If n=0 Then F:=1
  Else Begin FN(n-1, F); F:=F/n End
End;
Begin
 FN(5,X); Write(X)
End.
```

В результате выполнения программы получено число: 0.008333333333

Пример 2

Вспомним задачу вычисления наибольшего общего делителя двух чисел, описанную в § 2.2.8. Задача решается с помощью алгоритма Евклида. Модифицированный алгоритм Евклида основан на следующей системе равенств:

$$\text{НОД}(M, N) = \begin{cases} M, \text{ при } N = 0; \\ \text{НОД}(N, M \bmod N), \text{ при } N \neq 0. \end{cases}$$

Но это частично-рекурсивное определение функции НОД от двух аргументов! Рекурсивная подпрограмма-функция получается путём непосредственной реализации этого определения в форме оператора ветвления. В следующей программе описана рекурсивная функция и обращение к ней.

```
Function NOD(M, N: integer): integer;
Begin
  If N=0 Then NOD:=M
  Else NOD:=NOD(N, M mod N)
End;
Begin
  Writeln(NOD(12,32))
End.
```

В результате выполнения программы получим число 4. В этой рекурсивной функции, как и в примере с факториалом, дано одно начальное значение. А теперь рассмотрим пример с двумя начальными значениями.

Пример 3

С XIII века в математике известна замечательная числовая последовательность, названная именем своего автора: числа Фибоначчи. У этой последовательности имеется множество приложений, с одним из которых вы познакомились в 10 классе: основание фибоначчиевой системы счисления. Первые два значения числового ряда Фибоначчи равны единице. Каждое следующее значение равно сумме двух предыдущих. Если функ-

цию вычисления n-го элемента ряда обозначить как $Fib(n)$, то ее математическое определение запишется так:

$$\mathrm{Fib}(n) = \begin{cases} 1, \text{при } n = 1 \text{ и } n = 2; \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2), \text{при } n > 2. \end{cases}$$

Очевидно, что это частично-рекурсивная функция с двумя начальными (граничными) значениями. На основании данной формулы можно запрограммировать рекурсивную подпрограмму-функцию на Паскале. В следующей программе описана такая функция и с её помощью вычислены первые 10 чисел Фибоначчи:

```
Program Fibonachi;
var i: integer;
Function Fib(n: integer): integer;
Begin
  If (n=1) or (n=2) Then Fib:=1
  Else Fib:=Fib(n-1)+Fib(n-2)
End;
Begin
  For i:=1 To 10 Do Write(Fib(i):4)
End.
```

В результате выполнения программы получим числовую последовательность первых десяти чисел Фибоначчи:

```
1   1   2   3   5   8  13  21  34  55
```

При выполнении этой программы компьютер будет строить два стека, поскольку в определении функции присутствуют два рекурсивных обращения к ней самой. Вопреки лаконичной простоте текста программы, её выполнение — достаточно тяжёлый процесс для компьютера с точки зрения времени счёта и расхода памяти на построение стека.

Пример 4

Снова вернёмся к задаче из § 2.2.7 — вычислению суммы числового ряда вида:

$$1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!} + \ldots$$

В примере 1 мы получили частично-рекурсивную функцию для вычисления слагаемых этой суммы. Оказывается, что и саму сумму можно вычислить как частично-рекурсивную функцию. Если для слагаемых использовать обозначения: $a_0 = 1$, $a_n = 1/n!$ для $n = 1, 2, \ldots$, а сумму обозначить как функцию $S(n)$, то будет справедливым следующее определение этой функции:

$$S(n) = \begin{cases} 1, \text{при } n = 0; \\ S(n-1) + a_n, \text{при } n > 0. \end{cases}$$

Ниже приведены два варианта решения этой задачи. Первый — циклическая программа из § 2.2.7. Во втором варианте используется рекурсивная подпрограмма-функция SUM для вычисления $S(n)$. В ней, в свою очередь, использована рекурсивная функция FN для вычисления a_n, составленная в примере 1.

<table>
<tr><th>Не рекурсивная программа</th><th>Программа с рекурсивными функциями</th></tr>
<tr><td>
<pre>
Program Summa_1;
Var E, a: real;
    N, i: integer;
Begin
  Write('N='); Readln(N);
  E:=0; i:=0; a:=1;
  While i<=N Do
  Begin
    E:=E+a;
    i:=i+1;
    a:=a/i
  End;
  Writeln('E=', E)
End.
</pre>
</td><td>
<pre>
Program Summa_4;
Var M: integer;
Function SUM(K: integer): real;
Function FN(n: integer): real;
Begin
  If n=0 Then FN:=1
  Else FN:=FN(n-1)/n
End;
Begin
  If K=0 Then SUM:=1
  Else SUM:=SUM(K-1)+FN(K);
End;
{Основная программа}
Begin
  Write('M='); Readln(M);
  Writeln('E=', SUM(M))
End.
</pre>
</td></tr>
</table>

Сопоставьте эти программы и сделайте вывод, какая из них вам кажется проще с точки зрения процесса программирования. С точки зрения оптимизации работы компьютера, предпочтительным является не рекурсивный вариант.

Сделаем выводы из сказанного в этом параграфе. Частично-рекурсивную функцию можно запрограммировать на Паскале, используя рекурсивную подпрограмму-функцию или подпрограмму-процедуру. Такая подпрограмма имеет ветвящуюся структуру алгоритма вместо циклической структуры в нерекурсивной программе. В некоторых случаях рекурсивный подход упрощает программирование. Однако рекурсивно определённая подпрограмма занимает больше машинного времени при исполнении и требует дополнительного расхода памяти на организацию стека.

Система основных понятий

Рекурсивные подпрограммы
n-шаговая **рекуррентная последовательность:** числовая последовательность, имеющая n заданных начальных элементов и формулу для вычисления последующих элементов через n предшествующих
Рекуррентная формула: формула для вычисления элементов рекуррентной последовательности
Частично-рекурсивная функция (математическая) — функция, частично определенная через саму себя
Рекурсивная подпрограмма (функция, процедура): подпрограмма, содержащая в своем описании вызов себя самой. Должна иметь выход, не содержащий рекурсивного обращения

Вопросы и задания

1. Можно ли арифметическую и геометрическую прогрессии назвать рекуррентными последовательностями? Есла да, то напишите для них рекуррентные формулы.
2. Напишите два варианта подпрограммы-функции вычисления n-го элемента арифметической прогрессии: не рекурсивную и рекурсивную.
3. Опишите рекурсивную подпрограмму-функцию `pow(x, n)` от вещественного `x` (`x` $\neq 0$) и целого `n`, которая вычисляет величину x^n согласно формуле:

$$x^n = \begin{cases} 1, \text{при } n = 0; \\ \dfrac{1}{x^{-n}}, \text{ при } n < 0; \\ x \cdot x^{n-1}, \text{при } n > 0. \end{cases}$$

2.3.2 Задача о Ханойской башне

Применение рекурсивных методов для решения вычислительных задач не всегда эффективно. В большинстве случаев для решения той же задачи можно построить оптимальный не рекурсивный алгоритм. В то же время существуют задачи не вычислительного содержания, решить которые без использования рекурсии оказывается крайне проблематичным. К числу таких задач относится известная головоломка под названием «Ханойские башни» (рис. 2.10).

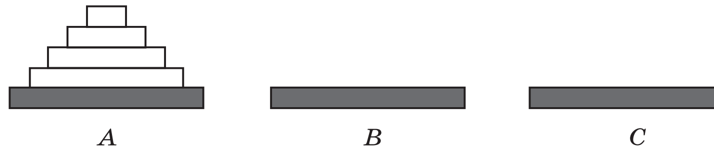


Рис. 2.10. Исходное состояние задачи

На площадке (назовем её A) находится пирамида, составленная из дисков уменьшающегося от основания пирамиды к её вершине размера. Эту пирамиду в том же виде требуется переместить на площадку B. При выполнении работы необходимо соблюдать следующие ограничения:

- перекладывать можно только по одному диску, взятому сверху пирамиды;
- класть диск можно только либо на основание площадки, либо на диск большего размера;
- в качестве дополнительной можно использовать площадку C.

Название «Ханойская башня» связано с легендой, согласно которой в давние времена монахи одного ханойского храма взялись переместить по этим правилам башню, состоящую из 64 дисков. С завершением их работы должен наступить конец света.

Нетрудно решить эту задачу для двух дисков. Обозначая перемещения диска, например, с площадки A на площадку B так: $A \Rightarrow B$, напишем алгоритм для этого случая:

$A \Rightarrow C;\ A \Rightarrow B;\ C \Rightarrow B.$

Всего 3 хода! Для трех дисков алгоритм длиннее:

$A \Rightarrow B;\ A \Rightarrow C;\ B \Rightarrow C;\ A \Rightarrow B;\ C \Rightarrow A;\ C \Rightarrow B;\ A \Rightarrow B.$

Уже 7 ходов.

Подсчитать количество ходов (N) для k дисков можно по следующей рекуррентной формуле:

$N(1) = 1;\ N(k) = 2 \cdot N(k - 1) + 1.$

Например, $N(10) = 1023$, $N(20) = 104857$. А вот сколько перемещений нужно сделать ханойским монахам:

$N(64) = 18\,446\,744\,073\,709\,551\,615.$

Попробуйте оценить, сколько лет на это потребуется.

Составим программу, по которой машина рассчитает алгоритм перемещения дисков и выведет его для любого значения n (количества дисков). Пусть на площадке A находится n дисков. Алгоритм решения задачи будет следующим:

1. Если $n = 0$, то ничего не делать.
2. Если $n > 0$, то:
   - переместить $n - 1$ диск на C через B;
   - переместить диск с A на B ($A \Rightarrow B$);
   - переместить $n - 1$ диск с C на B через A.

При выполнении пункта 2 последовательно будем иметь три состояния (рис. 2.11).

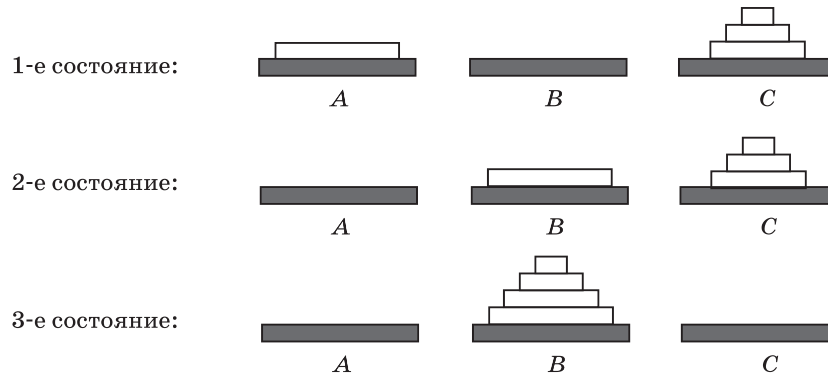


Рис. 2.11. Процесс перемещения башни

Описание алгоритма имеет явно рекурсивный характер. Перемещение n дисков описывается через перемещение $n - 1$ диска. А где же выход из этой последовательности рекурсивных ссылок? Он в пункте 1, как бы не показалось странным его тривиальное содержание.

А теперь составим программу на Паскале. В ней имеется рекурсивная процедура Hanoy, выполнение которой заканчивается только при $n = 0$. При обращении к процедуре используются фактические имена площадок, заданные их номерами: 1, 2, 3. Поэтому на выходе цепочка перемещений будет описываться в таком виде:

$1 \Rightarrow 2 \;\; 1 \Rightarrow 3 \;\; 2 \Rightarrow 3$ и т. д.

```
Program Monahi;
Var N: Byte;
Procedure Hanoi(N: Byte; A, B, C: Char);
Begin
  If N > 0 Then
  Begin Hanoi(N-1, A, C, B);
    WriteLn(A, '=>', B);
    Hanoi(N-1, C, B, A)
  End
End;
Begin
  WriteLn('Укажите число дисков: ');
  ReadLn(N);
  Hanoi(N, '1', '2', '3')
End.
```

Удивительная программа, не правда ли? Попробуйте воспроизвести её на машине. Проследите, как изменяется число ходов с ростом n. Для этой цели можете сами добавить в программу счётчик ходов и в конце вывести его значение или выводить ходы с порядковыми номерами.

2.3.3 Алгоритм быстрой сортировки

Об алгоритмах сортировки рассказывалось в § 1.7.7 учебника 10 класса. Там описаны два алгоритма сортировки: методом поиска максимального элемента и методом пузырька. Сейчас мы рассмотрим самый оптимальный по времени работы алгоритм, который называется алгоритмом быстрой сортировки. Этот алгоритм был разработан Энтони Хоаром в 1960 году. Покажем, как он реализуется через рекурсивную процедуру.

В алгоритме быстрой сортировки используются три идеи:

1) разделение сортируемого массива на две части: левую и правую;
2) взаимное упорядочение двух частей (подмассивов) так, чтобы все элементы левой части не превосходили значений элементов правой части;
3) рекурсия, при которой подмассив упорядочивается точно таким же способом, как и весь массив.

Для первоначального разделения массива на две части нужно выбрать некоторое «барьерное» значение. Это значение должно удовлетворять единственному условию: лежать в диапазоне значений для данного массива (т. е. между минимальной и максимальной величинами). В качестве «барьера» можно выбрать значение любого элемента массива, например первого или последнего, или находящегося в середине.

Далее нужно сделать так, чтобы в левом подмассиве оказались все элементы со значениями, меньшими или равными барьеру, а в правом — бóльшими или равными. Для этого, просматривая массив слева направо, нужно найти позицию первого элемента со значением, бóльшим или равным барьеру, а просматривая справа налево, найти первый элемент со значением, меньшим или равным барьеру. Поменять местами эти значения. Затем продолжить встречное движение до следующей пары элементов, предназначенных для обмена. Так продолжать до тех пор, пока индекс левого просмотра не станет больше индекса правого просмотра. Эти индексы будут разделителями двух взаимно упорядоченных подмассивов. Далее алгоритм рекурсивно применяется к каждому из подмассивов (левому и правому). В конечном итоге приходим к совокупности из n взаимно упорядоченных одноэлементных массивов, которые делить дальше невозможно. Эта совокупность образует один полностью упорядоченный массив. Сортировка завершена!

В следующей программе вещественный массив `A` заполняется случайными числами в диапазоне от 0 до 10. Затем этот массив сортируется с помощью процедуры быстрой сортировки `QSort`.

```
Program Sortirovka;
Const N=20;
Var A: array[1..N] of real;
    i: integer;
```

```
Procedure Qsort(L, R: Integer);
Var i, j: integer; bar, w: real;
Begin i:=L; j:=R;
  bar:=A[(L+R)div 2]; {Установка барьера}
  Repeat
    {Поиск элемента слева для обмена}
    While A[i]<bar Do i:=i+1;
    {Поиск элемента справа для обмена}
    While A[j]>bar Do j:=j-1;
    {Обмен элементов и смещение по массиву}
    If i<=j Then
    Begin
      w:=A[i]; A[i]:=A[j]; A[j]:=w;
      i:=i+1; j:=j-1
    End;
  Until i>j;
  {сформированы взаимно упорядоченные подмассивы}
  {Сортировка левого подмассива}
  If L<j Then Qsort(L, j);
  {Сортировка правого подмассива}
  If i<R Then Qsort(I, R);
End; {Qsort}

{Основная программа}
Begin
  Randomize;
  For i:=1 To N Do A[i]:=Random*10; {Заполнение случайными}
                                    {числами}
  Qsort(1, N);           {Обращение к процедуре сортировки}
  For i:=1 To N Do Write(A[i]:5:1) {Вывод отсортированного}
                                   {массива}
End.
```

Константа N и массив A описаны в программе глобально. Процедура Qsort(L,R) сортирует по возрастанию значений элементы массива A. Параметр L процедуры обозначает нижнее значение индекса сортируемого массива (крайний левый индекс), параметр R — верхнее значение индекса (крайний правый индекс). Барьерное значение bar вычисляется как значение элемента, имеющего индекс, равный целой части среднего арифметического между L и R.

Система основных понятий

Примеры рекурсивного программирования
Задача о Ханойской башне
Быстрая сортировка (Э. Хоар)

Вопросы и задания

1. Сформулируйте идею рекурсивного алгоритма решения задачи о Ханойской башне.
2. Реализуйте на компьютере программу решения задачи о Ханойской башне.
3. Сформулируйте идею алгоритма быстрой сортировки.
4. Реализуйте на компьютере программу сортировки массива с использованием процедуры быстрой сортировки.

Компьютерный практикум. Раздел «Программирование»

2.4. Объектно-ориентированное программирование

2.4.1 Базовые понятия объектно-ориентированного программирования

В этом разделе вы познакомитесь с основами объектно-ориентированной парадигмы программирования (ООП). Как уже рассказывалось в § 2.2.1, язык Паскаль первоначально предназначался только для процедурного программирования с опорой на структурную методику. Затем в поздних версиях языка Турбо Паскаль появились элементы объектно-ориентированного программирования. Этот процесс привёл к созданию объектно-ориентированной версии Паскаля, которая получила название Object Pascal. Результатом объединение языка программирования Object Pascal с визуальной технологией программирования и библиотекой визуальных компонентов стала система программирования Delphi. Далее будет рассмотрена объектная модель, положенная в основу Delphi.

Классы, объекты, инкапсуляция

Центральными понятиями объектно-ориентированного программирования являются «класс» и «объект». **Класс** — это тип данных, описываемый программистом; это категория объектов, обладающих одинаковым набором свойств и методов воздействия на них. **Объект** — это экземпляр определенного класса с конкретными значениями свойств.

Класс в Object Pascal является структурированным типом данных, элементы которого — поля и методы. Формат описания типа «класс» следующий:

```
Type <имя класса> = class[(<имя класса родителя>)]
     <описания полей>
     <объявление методов и описания свойств>
End;
```

Поля — это переменные величины (простые или структурные), методы — это процедуры и функции, работающие с полями этого класса. Процесс объединения в единую структуру данных **(полей)** и действий над этими данными **(методов)**, называется в ОПП **инкапсуляцией**. Инкапсуляция исключает возможность изменения значений полей другими способами, кроме методов данного класса. *Инкапсуляция — первый базовый принцип ООП*.

Рассмотрим пример программы вычисления длины отрезка прямой на языке Object Pascal, в которой используется тип данных класс.

```
Program Geometry;
{----  Описание класса TLine---------- }
Type TLine = class
  FX1, FY1, FX2, FY2: real; // Поля — координаты концов линии
  Function Length: real;  // Метод — функция вычисления длины
  Procedure Input;       // Метод — процедура ввода координат
End;
{ -------Описание объекта типа TLine------ }
Var Line: TLine; //
{---------Реализация методов---------}
Function Line.Length: real;
Begin
  Line.Length:=sqrt(sqr(FX1-FX2)+sqr(FY1-FY2))
End;
Procedure Line.Input;
Begin
  Write('x1='); Readln(FX1);
  Write('y1='); Readln(FY1);
  Write('x2='); Readln(FX2);
  Write('y2='); Readln(FY2)
End;
{------Основная программа-----------}
Begin
  Line:=TLine.Create; //Создание объекта в динам. памяти
  Line.Input;         //Ввод данных
                      //Вычисление и вывод длины отрезка
  Writeln('Длина отрезка=', Line.Length)
  Line.Destroy     //Удаление объекта из динаммческой памяти
End.
```

В описании класса методы представлены в виде объявлений заголовков процедур и функций, их реализующих. Подробное описание этих процедур и функций помещается ниже в разделе подпрограмм. Обратите внимание на то, что в описаниях функций `Length` и процедуры `Input` не указываются параметры, поскольку доступными для них величинами являются поля класса.

Конструктор (`Create`) — это стандартная функция, предназначенная для выделения в динамической памяти компьютера места под данные объекта. Деструктор (`Destroy`) — стандартная процедура, освобождающая динамическую память от объекта.

Обращение к полям и методом объекта данного класса производится с помощью составного имени, как это делается в записях:

```
<имя объекта>.<имя поля>,  <имя объекта>.<имя метода>
```

Наследование и полиморфизм

Следующая программа также будет иметь геометрическое содержание. В ней объявлены два класса: класс выпуклых четырёхугольников (TFourAngl) и класс квадратов (TKvadrat). Четырёхугольник — более общее понятие, чем квадрат. Квадрат является частным случаем четырёхугольника. Общее свойство, характерное для любых четырёхугольников, — наличие четырёх вершин. Поэтому полями класса TFourAngl будут координаты четырёх вершин. Кроме того, в число полей класса четырёхугольников включим длины четырёх сторон и двух диагоналей четырёхугольника. Среди методов класса TFourAngl объявим ввод координат вершин, вычисление длин отрезков (сторон и диагоналей) и вычисление площади произвольного выпуклого четырёхугольника.

Класс TKvadrat будет объявлен как **потомок** класса TFourAngl. Имя родительского класса указывается в объявлении класса-потомка в круглых скобках после слова class. Класс-потомок наследует у класса-родителя все его элементы, т. е. поля и методы, поэтому в объявлении класса-потомка их повторять не следует. Хотя в классе TFourAngl существует метод вычисления площади произвольного выпуклого четырёхугольника — функция Square, однако в классе TKvadrat объявлена функция с тем же именем Square с той целью, чтобы для квадрата можно было применять более простой способ вычисления площади (как квадрат длины стороны), что повысит точность вычислений и сократит машинное время.

Рассмотрим полный текст программы:

```
Program Geometry;
Type TKoord = record
  X, Y: real
End; //Тип координат вершин
{------ Объявление базового класса (родителя)------}
Type TFourAngl = class
  P: array[0..3] of TKoord; //Координаты 4-х вершин
  L: array[0..5] of real;   //Длины сторон и диагоналей
  Procedure Init;     //Метод: ввод координат
  Procedure Line;     //Метод: вычисление длин отрезков
  Function Square: real; //Метод: вычисление площади
End;
{------ Объявление порожденного класса (потомка)-----}
Type TKvadrat = class(TFourAngl)
  Function Square: real; //Метод: вычисление площади квадрата
End;
{----Описание объектов Четырехугольник и Квадрат ----}
Var FourAngl: TFourAngl;
  Kvadrat: TKvadrat;
```

```
{------ Описание процедур реализации методов -----}
Procedure TFourAngl.Init;
Var i: integer;
Begin
  For i:=0 To 3 Do
  Begin
    Write('Input X', i, ':'); Readln(P[i].X);
    Write('Input Y', i, ':'); Readln(P[i].Y)
  End
End;

Procedure TFourAngl.Line;
Var i: integer;
Begin
  For i:=0 To 3 Do
    L[i]:=sqrt(sqr(P[i].X-P[(i+1) mod 4].X)+
          sqr(P[i].Y-P[(i+1) mod 4].Y));
  L[4]:=sqrt(sqr(P[0].X-P[2].X)+ sqr(P[0].Y-P[2].Y));
  L[5]:=sqrt(sqr(P[1].X-P[3].X)+ sqr(P[1].Y-P[3].Y))
End;

Function TFourAngl.Square: real;
Var pp1, pp2: real;
Begin
  Line;
  pp1:=(L[0]+L[1]+L[4])/2; pp2:=(L[2]+L[3]+L[4])/2;
  Square:=sqrt(pp1*(pp1-L[0])*(pp1-L[1])*(pp1-L[4]))+
          sqrt(pp2*(pp2-L[2])*(pp2-L[3])*(pp2-L[4]))
End;

Function TKvadrat.Square: real;
Begin
  Line;
  Square:=sqr(FourAngl.L[0])
End;

 {---------Основная программа-----------}
Begin
  FourAngl:=TFourAngl.Create; //Создание объекта FourAngl
  Kvadrat:=TKvadrat.Create;   //Создание объекта Kvadrat
  FourAngl.Init;      //Ввод координат вершин
  FourAngl.Line;      //Вычисление длин сторон и диагоналей
  With FourAngl Do
  Begin
    {----Распознавание квадрата и вычисление площади------}
    If (L[0]=L[1]) and (L[1]=L[2]) and (L[4]=L[5])
    Then Writeln('Это квадрат, площадь = ', Kvadrat.Square)
    Else Writeln('Это не квадрат, площадь = ', Square)
  End
End.
```

Встроенные в текст комментарии позволяют понять назначение отдельных фрагментов программы. В этой программе применен второй базовый принцип объектно-ориентированного программирования (помимо инкапсуляции), который называется **«наследование»**. Наследование — способ создания новых классов в качестве наследников уже существующих.

Класс-потомок наследует от своего родительского класса все поля и методы. При этом если родительский класс также унаследовал от своих предков некоторые элементы, то они передаются по наследству его потомку. Технология ООП позволяет выстраивать ветвящиеся иерархии наследования, т. е. один потомок может иметь несколько родителей. В нашем примере используется простое наследование: один родитель — один потомок, что соответствует ограничению, действующему в Object Pascal.

Третий базовый принцип ООП называется **«полиморфизм»** — многообразие. Элементы классов с одинаковым интерфейсом могут иметь разную реализацию. В рассмотренной программе полиморфизм проявляется в том, что функция с одним и тем же именем `Square` определена дважды, по-разному: сначала в родительском классе `TFourAngl`, а затем она *переопределена* в классе-потомке `TKvadrat`. В результате площадь произвольного четырёхугольника и площадь квадрата будут вычисляться по разным алгоритмам.

Система основных понятий

Базовые понятия ООП	
Класс	Структурный тип данных, определяющий категорию объектов, обладающих одинаковым набором свойств и методов воздействия на них
Объект	Экземпляр определённого класса с конкретными значениями свойств (величина типа «класс»)
Инкапсуляция	Первый базовый принцип ООП: объединение в единую структуру (класс) данных — полей и действий над этими данными — методов
Наследование	Второй базовый принцип ООП: класс-потомок получает от своего родительского класса все поля и методы
Полиморфизм	Третий базовый принцип ООП: возможность переопределения метода с одним интерфейсом в разных классах

Вопросы и задания

1. Что такое класс и что такое объект в Object Pascal?
2. Дайте определения понятий: инкапсуляция, наследование, полиморфизм.
3. Реализуйте на компьютере (в системе программирования на Object Pascal) программу `Geometry`.
4. Добавьте в описание класса `TFourAngl` поля для значений углов четырёхугольника и метод для вычисления угла. В основной части программы реализуйте вычисление всех углов.

Компьютерный практикум. Раздел «Программирование»

2.4.2 Система программирования Delphi

Delphi — система программирования, предназначенная для создания **объектно-ориентированных** приложений Windows путём использования **визуальной технологии программирования**. Визуальная среда Delphi относится к средам категории RAD (Rapid Application Development — среда быстрой разработки приложений). Договоримся также термином Delphi называть и язык программирования, являющийся современным диалектом Паскаля, который также называют Delphi Pascal.

Основной подход к разработке программного обеспечения в подобных средах заключается в использовании стандартных визуальных компонентов — заранее подготовленных классов, которые программист подключает к программе, помещая их в свой проект.

Процесс разработки программы в Delphi может быть предельно упрощён. В первую очередь это относится к созданию интерфейса, на который обычно уходит порядка 80% времени разработки программы. Программисту необходимо просто поместить нужные компоненты на поверхность окна (в Delphi оно называется *формой*) и настроить их свойства с помощью специального инструмента (Object Inspector). С его помощью можно связать *события* компонентов (нажатие на кнопку, выбор мышью элемента в списке и т. д.) с процедурами их обработки, которые должен составить сам программист, — и простое приложение готово. При этом программист имеет в своем распоряжении необходимые средства отладки, удобную контекстную справочную систему, средства коллективной работы над проектом и пр.

Среда системы программирования Delphi

Среда системы программирования Delphi показана на рис. 2.12. Она состоит из следующих элементов:

Строка заголовка (вверху окна).

Строка главного меню и командные кнопки (под строкой заголовка).

Окно конструктора форм. Располагается в центре экрана на вкладке *Design* (см. рис. 2.12). Форма используется для конструирования интерфейса проектируемого приложения путём размещения на ней элементов управления (элементов интерфейса).

Окно программного кода. Располагается в центре экрана на вкладке *Code*. В начале создания программы в окне программного кода помещается стандартный шаблон (рис. 2.13). Далее в процессе программирования он будет заполняться текстом программы.

Окно элементов управления (элементов интерфейса). Располагается справа, озаглавлено *Tool Palette*. Содержит пиктограммы элементов управления, которые разделены на группы. Стандартная группа пиктограмм озаглавлена *Standard*, дополнительная группа — *Additional*, и т. д. Элементы управления с помощью мыши перемещаются на форму проектируемого приложения.

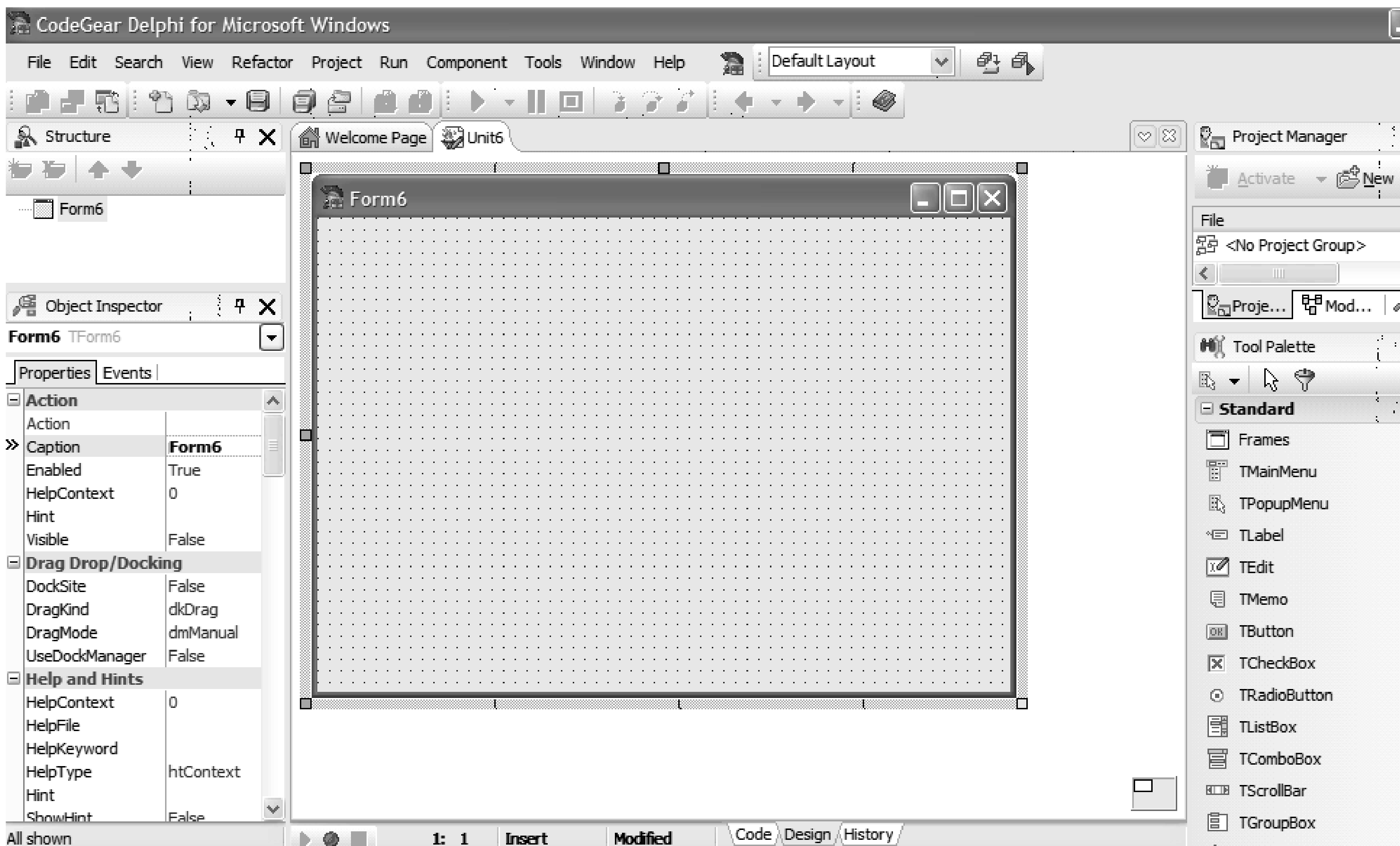


Рис. 2.12. Среда программирования Delphi с окном конструктора форм

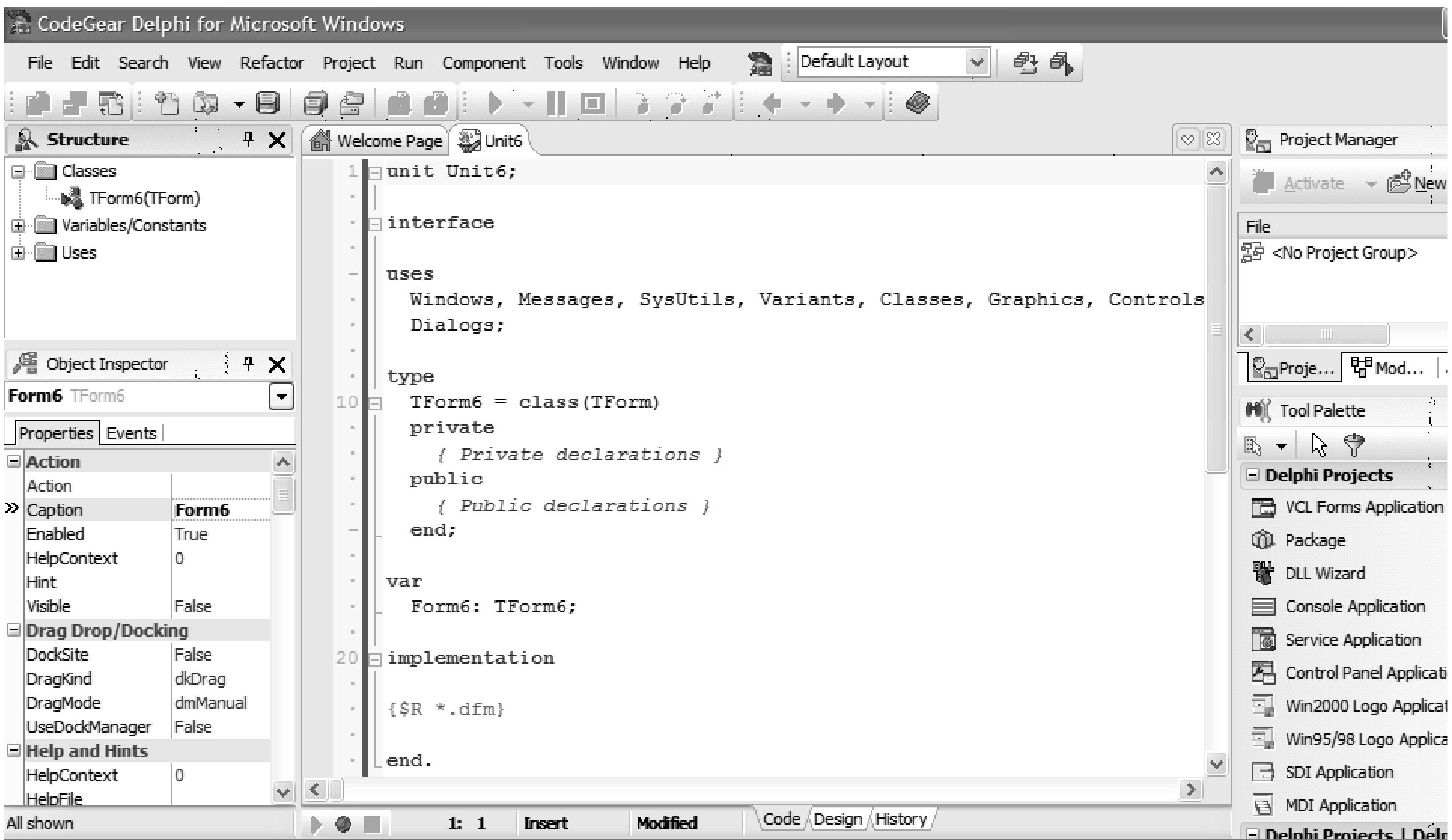


Рис. 2.13. Среда программирования Delphi с окном программного кода

Окно инспектора объектов. Располагается в левой нижней части экрана под заголовком *Object Inspector*. Вверху окна находится выпадающий список с названиями объектов (на рис. 2.12 имеется единственный объект с названием *Form 6*). Под ним на вкладке *Properties* (Свойства) содержится список полей — свойств объекта с их названиями и значениями, принятыми по умолчанию, которые могут быть изменены вручную или программным путем. На вкладке *Events* (События) содержится список событий, относящихся к выбранному объекту. Каждое событие имеет имя и указание на метод — программу обработки события, выполняемую при наступлении этого события.

Окно менеджера проектов. Располагается в верхнем правом углу экрана под заголовком *Project Manager*. Отображает файловое дерево компонентов, составляющих текущий проект (см. далее). Позволяет щелчком мышью переходить от одного компонента к другому.

Окно дерева объектов. Располагается в верхнем левом углу под заголовком *Structure*. Отображает состав объектов графического интерфейса в создаваемом проекте. Позволяет переключаться между объектами щелчком мышью.

Проект. Форма

Проектом (Project) на Delphi называется весь комплекс модулей и ресурсов, из которого создаётся исполняемая программа. Проект включает в себя программные модули, описания экранных форм, графических ресурсов, общих параметров создания программы и т. д.

Форма — базовый графический объект Delphi для создания рабочих окон. Форма имеет все признаки окна традиционных приложений: значок, заголовок, кнопки «Свернуть», «Развернуть», «Закрыть», размерную рамку и управляется мышью (см. рис. 2.12). В программе, предусматривающей интерактивное взаимодействие с пользователем, назначается главная форма, описывающая основное окно программы.

Форма является своеобразным контейнером, который включает в себя все другие компоненты графического интерфейса проекта: кнопки, метки, окна ввода и пр. **Конструктор форм** позволяет выполнить во время разработки проекта следующие действия:

- добавить компоненты на форму;
- модифицировать форму и её компоненты;
- связать обработчики событий компонента с процедурой или функцией, содержащейся в редакторе кода.

В общих чертах процесс разработки программы на Delphi выглядит следующим образом: из окна элементов управления с помощью мыши выбираются компоненты интерфейса (кнопка, надпись, редактор текста и др.), помещаются на форму, и задаются значения их свойств в области *Свойства*. Среда Delphi анализирует содержимое формы, создает соответствующий программный модуль (*Unit*), связанный с формой, а программист вносит в него программные коды процедур — **обработчиков событий**.

Проект может содержать несколько форм и, следовательно, несколько программных модулей. Помимо программных модулей форм, в проект включаются файлы с некоторыми другими программными модулями.

Элементы управления. Свойства

Элементы управления — это классы объектов, являющихся компонентами графического интерфейса проекта, обладающие набором свойств, определяющих их внешний вид и состояние, а также реагирующие на события, производимые пользователем или программой.

Примеры элементов управления и их свойств:

TLabel — метка. Служит для отображения текста на экране. В числе свойств: надпись (*Caption*), имя (*Name*), параметры шрифта (*Font*), цвет (*Color*), размер на экране, координаты размещения на экране и др.

TEdit — экранное поле редактирования. Служит для ввода данных пользователем в процессе выполнения программы. Основные свойства: имя (*Name*), текст в поле ввода/редактирования (*Text*), шрифт (*Font*), размеры и др.

TButton — командная кнопка. Позволяет выполнять какие-либо действия при нажатии кнопки во время выполнения программы. Основные свойства: надпись (*Caption*), имя (*Name*), размеры и положение на экране.

Элементы управления. События

Событие — изменение некоторого состояния объекта в результате действия пользователя или программного воздействия на объект. Для каждого класса объектов имеется свой набор событий, отображенный на вкладке Events окна Object Inspector.

Примерами событий, вызываемых действиями пользователя, являются:

OnClick — щелчок указателем мыши на объекте;
OnDblClick — двойной щелчок указателем мыши на объекте;
OnMouseMove — движение указателя мыши над объектом;
OnChange — изменение содержания (например, текста в окне редактирования).

Методы — процедуры обработки событий

Реакцией Delphi на событие является вызов процедуры — обработчика события. Такую процедуру составляет сам программист. Выбрав событие на вкладке Events, следует произвести двойной щелчок на поле, напротив имени события. В окне программного кода появится заготовка процедуры, в которую нужно вписать программный код реакции на это событие.

Программы, создаваемые в системе Delphi, называются *событийно-управляемыми программами*. Это значит, что выполнение различных задач, решаемых программой, инициируются определенными событиями.

Система основных понятий

Система программирования Delphi	
Delphi	Система программирования для создания объектно-ориентированных приложений Windows. Используется визуальная технология разработки интерфейса
Форма	Базовый графический объект Delphi, на основе которого создается программный проект (Project)
Элементы управления	Классы объектов графического интерфейса проекта, вкладываемых в форму (метки, кнопки, окна редактирования и пр.). Обладают набором свойств (Properties) и событий (Events)
Методы	Процедуры обработки событий (составляются программистом)
Событийно-управляемое программирование	Выполнение различных задач, решаемых программой, инициируется определёнными событиями, которые запускают методы их обработки

Вопросы и задания

1. Для чего предназначена система Delphi?
2. Назовите основные компоненты среды программирования Delphi.
3. Что такое форма?
4. Что такое элементы управления? Чем они характеризуются?
5. Что такое методы?
6. Почему программирование на Delphi называется событийно-управляемым программированием?

2.4.3 Этапы программирования на Delphi

С помощью системы программирования Delphi можно создать несколько различных типов приложений. В том числе можно реализовать и приложение без графического интерфейса, которое называется **консольным приложением.** Такие приложения вам уже приходилось создавать, например в системе Турбо Паскаль или PascalABC.

Создание консольного приложения

Приведём в качестве первого примера для программирования задачу перевода целого недесятичного числа в десятичную систему счисления. В § 1.3.3 учебника для 10 класса приведена такая программа на Паскале. Практически в том же виде её можно перенести в среду Delphi в режим

создания консольного приложения. Делается это следующим образом: после запуска системы Delphi через главное меню отдать команду на создание нового проекта: *File, New*. Затем в открывшемся окне выбрать тип приложения *Console Application*. В открывшемся окне редактора программного кода с готовым шаблоном ввести текст программы. Программа будет выглядеть следующим образом:

```
Program Project1;
{$APPTYPE CONSOLE}  //Директива компилятору о типе приложения
Uses SysUtils;      //Подключение системной библиотеки
Var N10, Np, k: longint;
    p: 2..9;
Begin
  Write('p='); Readln(p); //Ввод основания системы счисления
  Write('N', p, '=');
  Readln(Np);      //Ввод исходного p-ичного числа
  k:=1; N10:=0;
  While (Np<>0) Do  //Цикл выполняется, пока Np не равно нулю
  Begin
    N10:=N10+(Np mod 10)*k;//Суммирование развёрнутой формы
    k:=k*p;      //Вычисление базиса: p, p в степени 2,
                 //p в степени 3, ...
    Np:=Np div 10         //Отбрасывание младшей цифры
  End;
  Writeln('N10=', N10);  //Вывод десятичного числа
  Readln              //Задержка окна результата на экране
End.
```

Выполнение программы инициируется через главное меню по команде *Run, Run*.

Создание оконного приложения

Теперь решение задачи о переводе чисел реализуем с помощью оконного проекта с графическим интерфейсом. Для этого после команды *File, New* выберем тип приложения *VCL Forms Application*.

1. Проектирование и конструирование интерфейса. С самого начала работы надо представить себе конечный результат: в какой форме мы хотим увидеть на экране результат выполнения программы. Такое представление для задачи о числах показано на рис. 2.14.

Используя конструктор формы и панель элементов управления, наносим на шаблон формы нужные элементы интерфейса, как это показано на рис. 2.15. Здесь присутствуют 5 элементов типа **Метка** (*Label*), два **окна редактирования** (*Edit*) и одна командная кнопка (*Button*). В окна редактирования будут вводиться исходные данные: основание системы счисления и число в этой системе. Результат будет выводиться в поле *Label5*.

Затем для объекта формы изменяется надпись (свойство *Caption*) на «Системы счисления». Для меток с 1-й по 4-ю назначаются требуемые надписи. На метке *Label5* надпись убирается. Для командной кнопки назначается надпись «Выполнить перевод». Окончательный вид макета интерфейса показан на рис. 2.16.

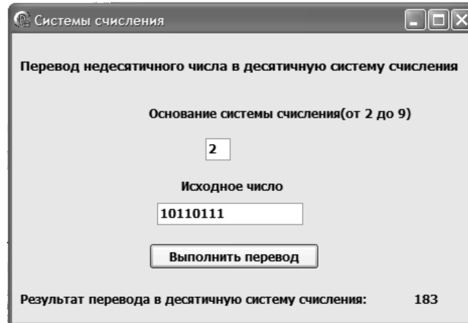


Рис. 2.14. Интерфейс задачи о числах

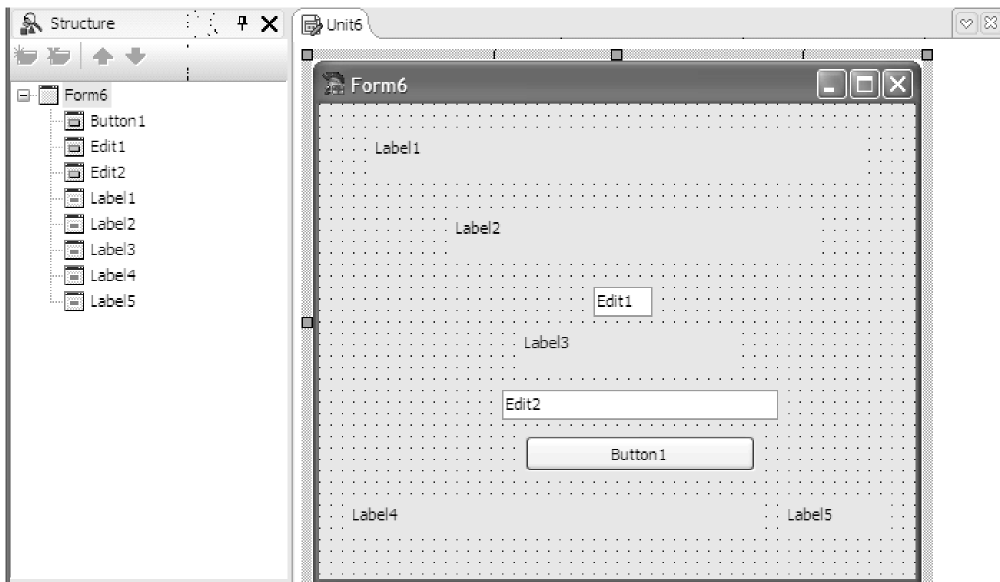


Рис. 2.15. Структура интерфейса

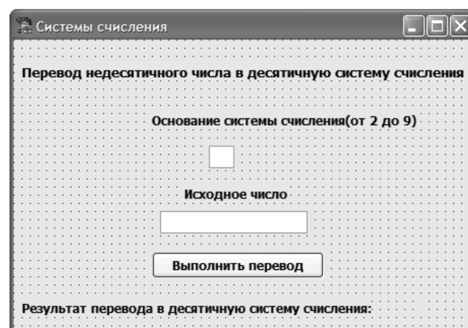


Рис. 2.16. Макет интерфейса программы

2. Реализация обработки событий. Выполнение перевода числа в десятичную систему счисления должно происходить по щелчку на кнопке «Выполнить перевод». Инициируем вызов процедуры обработки события *OnClick* для объекта *Button1*. В окне редактора программного кода появляется шаблон для процедуры с заголовком:

```
Procedure TForm6.Button1Click(Sender: TObject);
```

Здесь в скобках указывается класс `TObject` — базовый класс Delphi, потомками которого являются все другие классы. В тело этой процедуры вписываем программу перевода недесятичного числа в десятичную систему счисления. В итоге весь программный модуль (*Unit*), связанный с созданной формой, будет иметь следующий вид:

```
Unit Unit6;
Interface
Uses Windows, Messages, SysUtils, Variants, Classes,
     Graphics, Controls, Forms, Dialogs, StdCtrls;
Type
  TForm6 = class(TForm)
  Label1: TLabel;
  Label2: TLabel;
  Edit1: TEdit;
  Label3: TLabel;
  Edit2: TEdit;
  Label4: TLabel;
  Label5: TLabel;
  Button1: TButton;
Procedure Button1Click(Sender: TObject);
  private
  {Private declarations}
  public
  {Public declarations}
 end;

Var Form6: TForm6;
implementation
{$R *.dfm}
{-------Процедура обработки события--------}
Procedure TForm6.Button1Click(Sender: TObject);
Var N10, Np, k: longint;
    p: 2..9;
Begin
  p:=StrToInt(Edit1.Text);    //Ввод из окна Edit1
  Np:=StrToInt(Edit2.Text);    //Ввод из окна Edit2
  k:=1; N10:=0;
  While (Np<>0) Do
  Begin
    N10:=N10+(Np mod 10)*k;
    k:=k*p;
    Np:=Np div 10
  End;
```

```
  Label5.Caption:=IntToStr(N10) //Вывод в поле метки Label5
End      //Конец процедуры обработки события
End.
```

Выполнение программы инициируется через главное меню по команде *Run*, *Run*. На экране появляется окно интерфейса программы. В окна редактирования вводятся исходные данные: основание системы счисления и переводимое число. Затем производится щелчок на командной кнопке, и в поле метки *Label5* получаем результат (см. рис. 2.14).

Ввод и вывод данных. Ввод исходных данных производится с помощью поля *Text* элемента интерфейса *Edit*. При вводе с клавиатуры числа в окно редактирования оно воспринимается как символьная строка. Если вводится целое число, то строка затем преобразуется в величину целого типа с помощью функции `StrToInt(строка)`. Если нужно ввести вещественное число, то используется функция `StrToFloat(строка)`.

Вывод результата выполнен путем присваивания свойству *Caption* метки *Label5* строки со значением итогового числа. Перевод целого числа в строку символов происходит с помощью функции `IntToStr(целое число)`. Если требуется вывести вещественное число, то используется функция `FloatToStr(вещественное число)`.

Если требуется вводить одно из конечного множества значений, то удобно использовать другие элементы интерфейса: **списки** (*ListBox*), **поля со списком** (*ComboBox*), **счётчики** (*UpDown*), **ползунки** (*TracBar*), **переключатели** (*RadioButton*). На рисунке 2.17 продемонстрировано использование поля со списком в программе «Системы счисления» для ввода основания системы. При нажатии на кнопку справа выпадает список значений для выбора одного из них. Значение выбирается щелчком мышью.

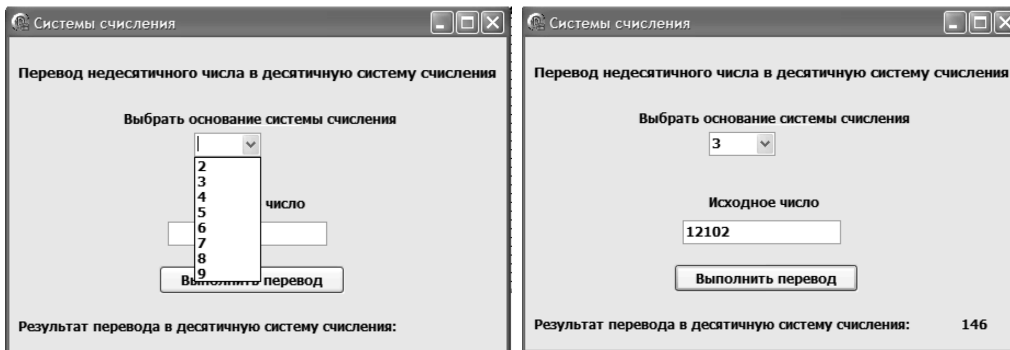


Рис. 2.17. Использование для ввода поля со списком

Значения, включаемые в список, вводятся во время программирования с использованием свойства `ComboBox1.Items`. Элементы поля со списком проиндексированы — получают порядковые номера, начиная с нуля. Каждый элемент идентифицируется индексированным именем элемента: `ComboBox1.Items[индекс]`. После выбора элемента из списка его индекс

присваивается полю ComboBox1.ItemIndex. В программе перевода чисел ввод значения переменной p реализуется следующим оператором:

```
p:=StrToInt(ComboBox1.Items[ComboBox1.ItemIndex]);
```

Поле со списком является способом организации интерфейса для представления одномерного массива.

Система основных понятий

Этапы программирования на Delphi	
Консольное приложение:	приложение Delphi без графического интерфейса
Оконное приложение:	приложение Delphi с графическим интерфейсом
Проектирование интерфейса:	разработка эскиза графического интерфейса
Конструирование интерфейса:	создание формы и её наполнение элементами управления
Реализация обработки событий:	программирование процедур методов обработки событий

Вопросы и задания

1. В чем различие консольного приложения и оконного приложения?
2. В какой последовательности создается оконное приложение на Delphi?
3. Какие функции используются для ввода и вывода данных в оконном приложении?
4. Реализуйте на компьютере приведенную в параграфе программу перевода недесятичного числа в десятичную систему счисления.

Компьютерный практикум. Раздел «Программирование»

2.4.4 Программирование метода статистических испытаний

Метод статистических испытаний (метод Монте-Карло) — численный метод, использующий моделирование случайных величин и получение статистических оценок искомых величин. Одно из применений этого метода — вычисление площадей фигур и объемов тел. Составим программу вычисления числа Пифагора — π с помощью метода статистических испытаний.

Идея метода состоит в следующем. Около единичной окружности описывается квадрат, длина стороны которого равна 2 (рис. 2.18). С помощью датчика случайных чисел с равномерным законом распределения вероятности производится «стрельба» по квадрату, т. е. случайный выбор точек внутри квадрата. Каждый такой выбор будем называть испытанием. Испытание заключается в том, что вычисляются координаты точки (x, y) с помощью функции `Random` в пределах значений от -1 до 1. Затем определяется, лежит ли эта точка внутри круга. Условие выполняется, если $x^2 + y^2 \le 1$. Если точка попадает в круг, то в счётчик попаданий добавляется единица.

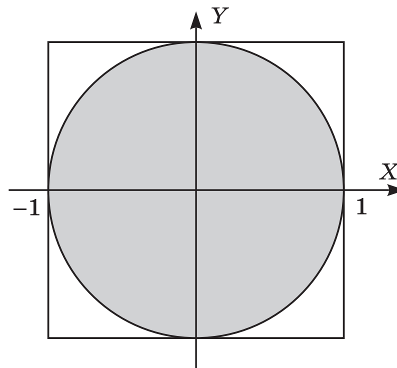


Рис. 2.18. Погружение в квадрат

Пусть P — общее число испытаний. Из них произошло M попаданий в круг. Площадь квадрата равна 4. При условии равномерного покрытия испытательными точками площади квадрата, для площади круга справедлива формула:

$$S_{\text{кр}} = 4 \cdot \lim_{P \to \infty} \frac{M}{P}.$$

Смысл формулы состоит в том, что с увеличением количества испытаний отношение M/P все больше приближается к отношению площадей круга и квадрата и в пределе при P, стремящемся к бесконечности, становится ему равно. Поскольку площадь круга радиуса 1 равна π, то при достаточно большом значении P будет выполняться приближенное равенство:

$$\pi \approx 4 \cdot \frac{M}{P}.$$

Чем больше P, тем это равенство точнее.

Интерфейс программы на Delphi решения этой задачи показан на рис. 2.19. Чтобы можно было проследить за установлением значения числа π, испытания разбиваются на серии. В одной серии производится N испытаний, а число таких серий равно K. После завершения каждой серии на экран выводится результат.

Окно с формой включает в себя следующие элементы интерфейса:

- *Memo1* — поле для ввода/редактирования многострочных текстов (до 32 Кб);
- *Edit1* и *Edit2* — поля для ввода значений K и N;
- *Label1* и *Label2* — метки к полям ввода;
- *Button1* — командная кнопка для запуска вычислений;
- *ListBox1* — поле для вывода прокручиваемого списка.

Обработка события `Button1Click` производится следующей процедурой:

```
Procedure TForm6.Button1Click(Sender: TObject);
Var i, j: integer; K, N, M: Int64;
    X, Y, Pi: double;
Begin
  K:=StrToInt(Edit1.Text); //Ввод числа серий
  N:=StrToInt(Edit2.Text); //Ввод длины серии испытаний
  Randomize;
  M:=0;           //Инициализация счетчика попаданий в круг
  For i:=1 To K Do    //Цикл по номеру серии испытаний
  Begin
    For j:=1 To N Do   //Повторение испытаний
    Begin
      X:=2*Random - 1;    //Вычисление X-координаты точки
      Y:=2*Random - 1;    //Вычисление Y-координаты точки
      If X*X+Y*Y<=1 Then M:=M+1 //Подсчёт числа попаданий
                                //в круг
    End;
    Pi:=4*M/(i*N);       //Вычисление p после окончания серии
    //Вывод числа испытаний и 'пи'
    ListBox1.Items.Add(IntToStr(i*N) + ' испытаний: Pi='
                       + FloatToStr(Pi))
  End
End;
```

Вывод в поле *ListBox* производится вызовом метода `ListBox1.Items.Add(строка)`. Каждое обращение к этому методу добавляет строку к списку *ListBox*. Для просмотра всего списка выведенных строк используется линейка прокрутки.

Обсудим результаты, представленные на рис. 2.19. Выполнены расчёты для 1000 серий, каждая из которых состоит из 10 000 испытаний. Последний результат соответствует десяти миллионам испытаний. В результате получилось только четыре верные цифры в значении числа π: 3,141. Более точное значение: $\pi = 3{,}14159265$.

Возникает вопрос: почему такой дорогой ценой (десять миллионов испытаний!) дались всего 4 цифры числа π? И отсюда следующий вопрос: а если продолжать увеличивать число испытаний, то можно ли таким же

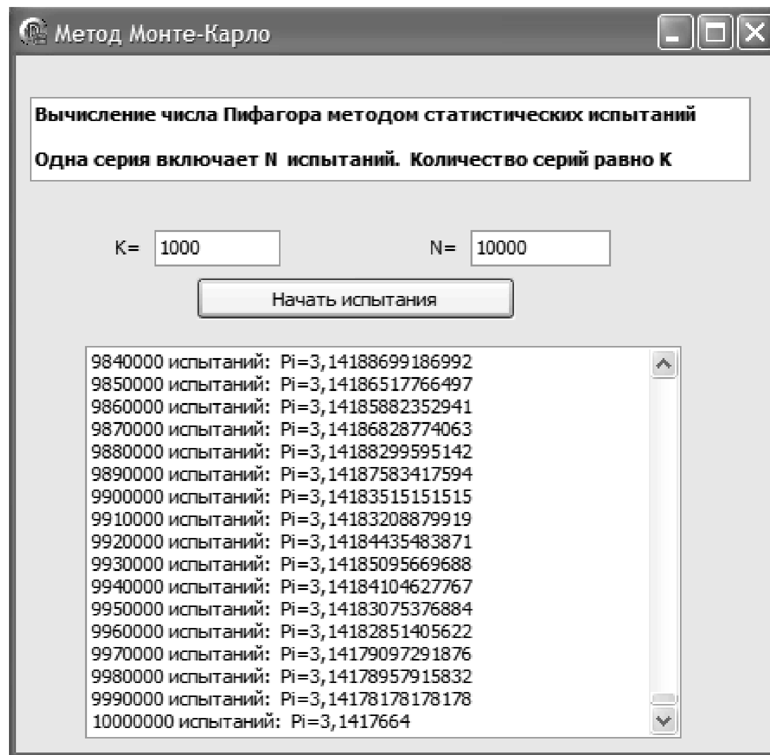


Рис. 2.19. Интерфейс программы «Метод Монте-Карло»

способом получить сколько угодно верных цифр числа π? Ответ: теоретически — да, практически — нет! В чём же дело?

Причина заключается в погрешности машинных вычислений с вещественными числами. Об этом говорилось в § 2.4.2 учебника для 10 класса. В результате этой погрешности для точек, которые отстоят от границы круга на расстояние порядка машинной ошибки, диагностика попадания в круг оказывается не всегда верной. Существует некоторое критическое число испытаний, превышение которого ведёт не к повышению, а к понижению точности вычисления числа π. Попробуйте определить его экспериментально!

С помощью метода Монте-Карло можно вычислять площади фигур сложной формы на плоскости или объёмы трёхмерных тел. Идея все та же: фигура помещается в другую фигуру простой формы, для которой известна площадь или объём. Например, для плоской фигуры — прямоугольник или круг, для объемного тела — прямоугольный параллелепипед или шар. Затем, с помощью датчика случайных чисел с равномерным законом распределения, производится стрельба по цели. Каким-либо образом диагностируются попадания в исследуемую фигуру. Далее вычисляется искомая площадь или объём по отношению числа попаданий к общему числу испытаний (выстрелов).

Система основных понятий

Метод статистических испытаний
Метод статистических испытаний (метод Монте-Карло) — численный метод, использующий моделирование случайных величин и получение статистических оценок искомых величин
Вычисление числа Пифагора: круг единичного радиуса, площадь которого равна π, вписывается в квадрат со стороной 2. Производятся испытания — случайный выбор точек внутри квадрата с подсчётом числа точек, попавших в круг. Точная формула: $$\pi = 4 \cdot \lim_{P \to \infty} \frac{M}{P},$$ где P — число испытаний, M — число попаданий в круг
Статистическая оценка: $\pi \approx 4 \cdot \frac{M}{P}$ для больших значений P
Причины погрешности результата: конечное значение числа испытаний P и погрешность машинных вычислений

Вопросы и задания

1. Что такое метод статистических испытаний (метод Монте-Карло)?
2. В чем состоит идея применения метода Монте-Карло для вычисления площадей и объёмов?
3. Реализуйте на компьютере приведенную в параграфе программу вычисления числа Пифагора. Проследите за изменением результата с ростом числа испытаний.

Компьютерный практикум. Раздел «Программирование»

2.4.5 Построение графика функции

Составим программу получения в окне вывода графика функции $y = f(x)$. Для получения графических изображений в Delphi используется объект *Canvas* — холст. Объект *Canvas* применяется не как самостоятельный компонент, а как свойство формы (`Form.Canvas`). Вывод рисунка на холст происходит с возникновением события `Paint` формы. Это событие появляется каждый раз, когда возникает необходимость вывода окна приложения (при запуске приложения или при обновлении вида окна). Обработка этого события осуществляется процедурой `FormPaint`.

Рисование на холсте происходит путём закрашивания точек на *графической поверхности* окна. Совокупность таких точек (пикселей) образует графическую сетку. Положение каждой точки сетки характеризуется двумя координатами: горизонтальной (x) и вертикальной (y). Следует учитывать, что координаты отсчитываются от левого верхнего угла. Один шаг графической сетки равен расстоянию между соседними пикселями (рис. 2.20).

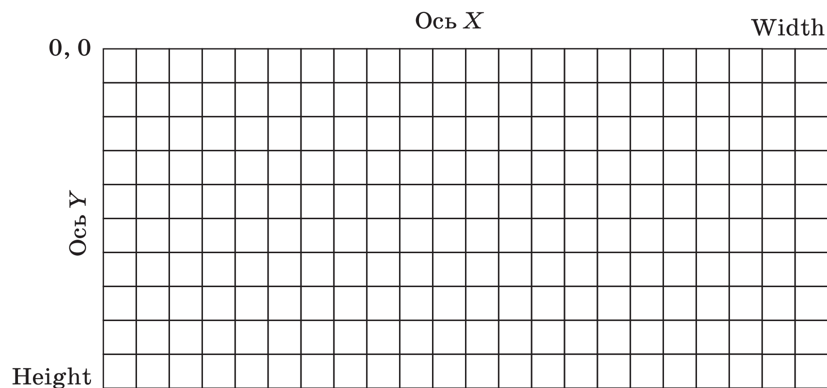


Рис. 2.20. Структура и параметры графической сетки

Максимальное число шагов по оси X хранит свойство формы `ClientWidth`, а по оси Y — свойство `ClientHeight`.

Для создания любых рисунков используются основные графические примитивы — точки, линии, окружности, прямоугольники и др. Чтобы нарисовать такие примитивы, используются соответствующие методы объекта *Canvas*.

В приведенной ниже программе рисования графика функции использованы следующие методы для построения изображения на холсте и нанесения надписей:

`MoveTo(x, y: integer)`	Устанавливает указатель текущей точки для рисования в позицию с указанными значениями координат `x`, `y`
`LineTo(x, y: integer)`	Вычерчивает линию из текущей точки в точку с указанными координатами. Вид линии определяет свойство *Pen* (перо)
`TextOut(x, y: integer; s: string)`	Выводит на экран строку `s` от точки с координатами (`x`, `y`). Шрифт определяет свойство *Font* поверхности (*Canvas*), на которую выводится текст. Цвет закраски области вывода текста определяет свойство *Brush* (Кисть) этой же поверхности.

Сведения о других методах см. в описаниях Delphi.

График функции строится по точкам с помощью установки значения свойства `Pixels[x, y]:=z`, где `x`, `y` — координаты пикселя, `z` — цвет пикселя.

Программа выполняет построение графика функции вида: $f(x) = 2\sin x \cdot e^{x/5}$ на отрезке $x \in [-10, 10]$. Для вычисления значений функции составлена подпрограмма-функция $f(x)$. Построение графика выполняет процедура с именем `GrOfFunc`. Текст программы снабжен подробными комментариями. Постарайтесь внимательно его изучить и понять смысл программы!

```
// ФУНКЦИЯ ДЛЯ ГРАФИЧЕСКОГО ПРЕДСТАВЛЕНИЯ
Function f(x: real): real;
Begin
  f:=2*sin(x)*exp(x/5)
End;
//ПРОЦЕДУРА ПОСТРОЕНИЯ ГРАФИКА
Procedure GrOfFunc;
Var x1, x2, //границы измерения аргумента функции
    y1, y2, // границы изменения значения функции
    x,      // аргумент функции
    y,      // значение функции в точке x
    dx: real;   // приращение аргумента
    l, b: integer; // левый нижний угол области вывода графика
    w, h: integer; // ширина и высота области вывода графика
    mx, my: real;  // масштаб по осям X и Y
    x0, y0: integer; // точка — начало координат
    n, sh, s: integer; // число единичных меток на осях
Begin
  // Расчёт области вывода графика
  l:=20;
  b:=Form6.ClientHeight-20; // координата y левого нижнего
                            // угла
  h:=Form6.ClientHeight-40; // высота
  w:=Form6.ClientWidth-40;  // ширина
  x1:=-10;   // нижняя граница диапазона аргумента
  x2:=10;  // верхняя граница диапазона аргумента
  dx:=0.01; // шаг аргумента
  // Вычисление максимального и минимального значения
  // функции на отрезке [x1,x2]
  y1:=f(x1); // переменная для минимума
  y2:=f(x1); // переменная для максимума
  x:=x1;
  Repeat
    y := f(x);
    If y < y1 Then y1:=y;
    If y > y2 Then y2:=y;
    x:=x+dx;
  Until (x>=x2);
  // Вычисление масштаба
  my:=h/abs(y2-y1); // масштаб по оси Y
```

```
mx:=w/abs(x2-x1); // масштаб по оси X
x0:=l+abs(Round(x1*mx)); //положение начала
y0:=b-abs(Round(y1*my)); // координат
// РИСОВАНИЕ И РАЗМЕТКА ОСЕЙ КООРДИНАТ
With Form6.Canvas Do
Begin
  MoveTo (x0, b); LineTo(x0, b-h); //рисуем ось Y
  MoveTo(l, y0); LineTo(l+w, y0);  //рисуем ось X
  //Разметка оси OY
  TextOut(x0+5,b-h,FloatToStrF(y2,ffGeneral,6,3));//нанесём
                                              //на ось Y max
  TextOut(x0+5,b,FloatToStrF(y1,ffGeneral,6,3));    //и min
                                          //значения функции
  n:=Round(Form6.ClientHeight/40); //количество меток
  If (y2-y1)<100 Then   //подбираем шаг расстановки меток
                        //в зависимости от диапазона
  Begin                 //изменения значений функции
    sh:=round((y2-y1)/n);
    s:=Round(y1)
  End
  Else
  Begin
    sh:=(Round((y2-y1)/n) div 10)*10; //выделим только
                                      //десятки
    s:=(Round(y1) div 10)*10
  End;
  Repeat
    MoveTo(Round(x0-2), Round(y0+s*my)); //рисуем риски
                                         //меток
    LineTo(Round(x0+2), Round(y0+S*my)); //и проставляем
                                         //числа
    TextOut(x0-30,Round(y0-5+s*my),
            FloatToStrF(-1*S,ffGeneral,6,3));
    S:=S+sh;
  Until (s>y2);
  //Разметка оси Ox
  n:=Round(w/40);
  If (x2-x1)<100
  Then //аналогично, для оси Ox
  Begin
    sh:=round((x2-x1)/n);
    s:=Round(x1)
  End
  Else
  Begin
    sh:=(Round((x2-x1)/n) div 10)*10;
    s:=(Round(x1) div 10)*10
  End;
  Repeat
    MoveTo(Round(x0+s*mx),y0-2);
```

```
        LineTo(Round(x0+S*mx),y0+2);
        TextOut(Round(x0-5+s*mx),y0+15,
                FloatToStrF(S,ffGeneral,6,3));
        S:=S+sh;
      Until (s>x2);
      // ПОСТРОЕНИЕ ГРАФИКА ФУНКЦИИ
      x:=x1;
      Repeat
        y:=f(x);
        //Закрашивание точки графической сетки красным цветом
        Pixels [x0+Round(x*mx),y0-Round(y*my)]:=clRed;
        x:=x+dx;
      Until (x>=x2);
    End;
  End;

  //Запуск построения графика с открытием формы
  Procedure TForm6.FormPaint(Sender: TObject);
  Begin
    GrOfFunc;
  End;

  //Перерисовывание графика с изменением размера формы
  Procedure TForm6.FormResize(Sender: TObject);
  Begin
    // очистить форму
    Form6.Canvas.FillRect(Rect(0,0,ClientWidth,ClientHeight));
    // построить график
    GrOfFunc
  End;
```

График строится в пределах прямоугольника, имеющего в окне вывода левое, правое, верхнее и нижнее поля, равные 20 шагам графической сетки. Шаг изменения математического значения координаты x равен 0,01. Масштабы по оси X и оси Y (переменные `mx` и `my`) равны числу шагов графической сетки, приходящихся на единицу изменения математических значений аргумента x и функции y. Принято использовать следующую терминологию: математические значения аргумента x и функции y определяют положение графика функции *в мировой системе координат*. А структура, представленная на рис. 2.20, называется *экранной системой координат*. Масштабы `mx` и `my` используются для пересчёта положения точек графика из мировой системы в экранную систему координат.

Если вручную изменять размеры окна вывода, то происходит событие формы, которое называется *Resize*. Это событие обрабатывается процедурой `FormResize`, которая производит очистку формы от старого рисунка и обеспечивает перерисовывание графика для нового размера окна. На рисунке 2.21 показаны три варианта изображения графика для разных размеров холста (окна вывода).

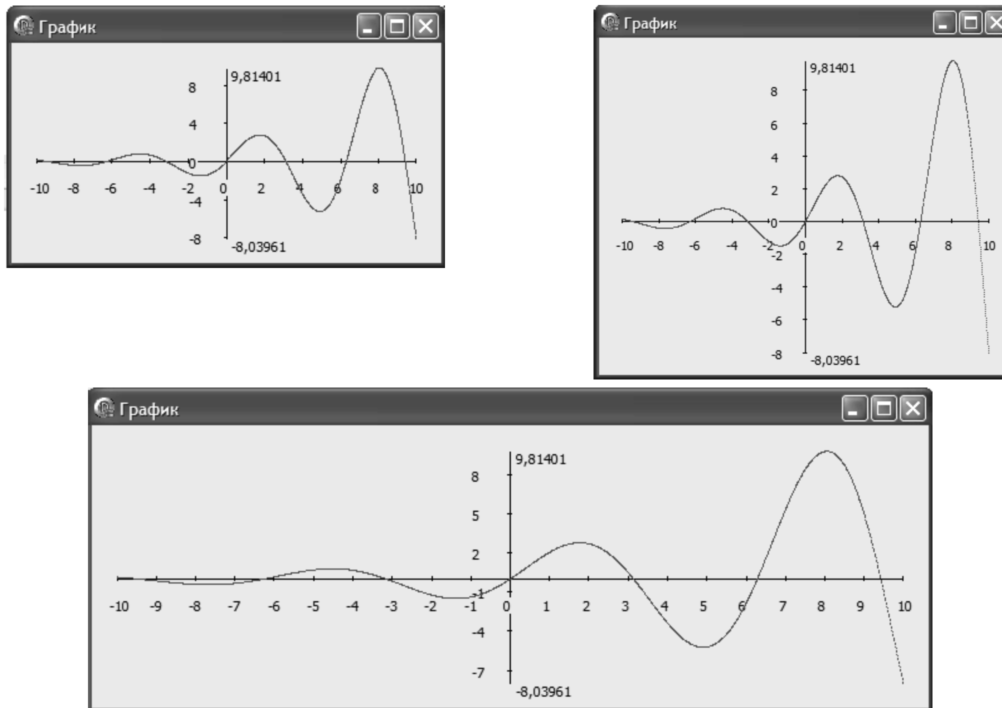


Рис. 2.21. График функции для разных размеров окна вывода, полученные с помощью процедуры `GrOfFunc`

Система основных понятий

Построение графика функции
Canvas (холст) — свойство формы, реализованное как объект, позволяющее наносить на форму графические изображения
Свойства холста: параметры графической сетки `ClientWidth` и `ClientHeight`, определяющие горизонтальный и вертикальный размеры
Рисование на холсте: производится с помощью команд изображения графических примитивов или закрашиванием точек графической сетки посредством установки значения свойства `Pixels[x,y]:=z`, где `x`, `y` — координаты пикселя, `z` — цвет пикселя
Для рисования графика функции: 1) строятся оси мировой системы координат; 2) осуществляется разметка осей; 3) изменяется с некоторым шагом значение аргумента и вычисляется соответствующее значение функции; 4) пересчитываются значения x и y из мировой системы координат в экранную систему; 5) рисуется точка на экране в соответствующей позиции

Вопросы и задания

1. Какое свойство формы позволяет создавать на ней графические изображения?
2. Какие параметры характеризуют размер холста?
3. Как расположены оси экранной системы координат?
4. С помощью каких методов можно наносить на холст рисунки и надписи?
5. Что такое мировая система координат?
6. Для чего в процедуре `GrOfFunc` используются величины `mx` и `my`?
7. При наступлении какого события выполняется программа рисования графика?
8. За счёт чего происходит перерисовывание графика с изменением размеров окна вывода (формы)?
9. Что нужно изменить в программе для того, чтобы нарисовать график другой функции в другом диапазоне значений x?
10. Какие операторы надо добавить в программу для того, чтобы на концах осей координат нарисовать стрелочки и написать обозначения осей X и Y?
11. Каким образом можно изменить цвет линии графика?
12. Почему при изменении размера окна вывода изменяется разметка осей? Проанализируйте, как это делается в программе.

Компьютерный практикум. Раздел «Программирование»