Министерство образования и науки Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

ФАКУЛЬТЕТ ДИСТАНЦИОННОГО ОБУЧЕНИЯ (ФДО)

А. В. Гураков, О. И. Мещерякова, П. С. Мещеряков

ИНФОРМАТИКА II

Учебное пособие

Рекомендовано Сибирским региональным учебно-методическим центром высшего профессионального образования для межвузовского использования в качестве учебного пособия для студентов, обучающихся по направлениям подготовки бакалавров 11.03.04 (210100.62) «Электроника и наноэлектроника», 38.03.01 (080100.62) «Экономика», 38.03.02 (080200.62) «Менеджмент»

Томск
2015

УДК 004.4(075.8)
ББК 32.973.2-018.1
Г 950

Рецензенты:
Миньков Л. Л., докт. физ.-мат. наук, профессор кафедры математической физики Национального исследовательского Томского государственного университета;
Силич В. А., докт. техн. наук, профессор кафедры оптимизации систем управления Института кибернетики Томского политехнического университета.

Гураков А. В.

Г 950 Информатика II : учебное пособие / А. В. Гураков, О. И. Мещерякова, П. С. Мещеряков. — 2-е изд., доп. — Томск : ФДО, ТУСУР, 2015. — 112 с.

В учебном пособии по дисциплине «Информатика II» рассмотрены основы алгоритмического подхода для создания программ, основы языка программирования Free Pascal. Пособие снабжено большим количеством примеров, иллюстрирующих рассматриваемый материал.

Учебное пособие по дисциплине «Информатика II» предназначено для студентов факультета дистанционного обучения ТУСУРа.

УДК 004.4(075.8)
ББК 32.973.2-018.1

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Языки программирования высокого уровня предназначены для записи алгоритмов в форме, более удобной для человека по сравнению с машинными языками, низкоуровневыми и состоящими из одних цифр. С развитием информатики существует огромное количество языков программирования, однако в настоящее время используются немногие из них. Язык Pascal создал в 1970 году Никлаус Вирт специально для начального обучения программированию. Язык назван в честь великого физика и математика Блеза Паскаля, первого изобретателя суммирующего механического устройства (арифмометра).

Язык Pascal является универсальным, потому что на нём можно писать программы для обработки данных различных видов (числовых, текстовых), как простых, так и организованных в сложные структуры. На нём можно записывать алгоритмы самых различных видов и самой разной сложности.

В первой главе учебного пособия даётся краткое знакомство с теорией алгоритмов. Теория алгоритмов является основополагающей теорией для наук, связанных с вычислительной техникой — основным инструментарием информатики.

Во второй главе излагаются основные понятия и элементы языка Pascal: алфавит, структура программы, организация ввода/вывода на экран, простые типы данных, выражения, операторы. Овладение основами языка позволит создавать простые программы. Третья глава посвящена структурированным типам данных: массивам, строкам, записям и файлам. Такие структуры языка, как подпрограммы и библиотеки подпрограмм, позволяющие программировать сложные алгоритмы, рассматриваются в четвёртой главе.

В пятой главе рассматривается интегрированная среда программирования Free Pascal (в дальнейшем IDE), которая позволяет создавать тексты программ, компилировать их, находить ошибки и оперативно их исправлять, компоновать программы из отдельных частей, отлаживать и выполнять отлаженную программу. Приводятся команды вызова системы информационной контекстной помощи и даются некоторые рекомендации по настройке системы Free Pascal для работы.

В приложении приводятся сообщения об основных ошибках, выводимых системой Free Pascal при трансляции и выполнении программы.

Для более детального изучения языка Free Pascal следует обратиться к дополнительной литературе.

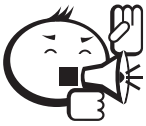
При изучении данного пособия рекомендуется набирать примеры программ на компьютере, запускать их на исполнение, вносить изменения для проверки неяс-

ных моментов и собственных предположений. Запомните — невозможно научиться программированию, не программируя. Если Вы не знакомы с интегрированной средой программирования Free Pascal, то, возможно, Вам в первую очередь стоит прочитать пятую главу.

Соглашения, принятые в книге

Для улучшения восприятия материала в данной книге используются пиктограммы и специальное выделение важной информации.

Эта пиктограмма означает определение или новое понятие.



Эта пиктограмма означает внимание. Здесь выделена важная информация, требующая акцента на ней. Автор здесь может поделиться с читателем опытом, чтобы помочь избежать некоторых ошибок.

Эта пиктограмма означает совет. В данном блоке можно указать более простые или иные способы выполнения определенной задачи. Совет может касаться практического применения только что изученного или содержать указания на то, как немного повысить эффективность и значительно упростить выполнение некоторых задач.

Пример

Эта пиктограмма означает пример. В данном блоке автор может привести практический пример для пояснения и разбора основных моментов, отраженных в теоретическом материале.

Контрольные вопросы по главе

Глава 1

ОСНОВНЫЕ ПОНЯТИЯ ТЕОРИИ АЛГОРИТМОВ

Наверняка можно утверждать, что каждый знаком с термином «алгоритм». Его применяют весьма широко не только в области вычислительной техники и программирования. Так же несомненно и то, что у каждого сформировалось своё (пусть даже большей частью интуитивное) понимание смысла этого термина.

Под алгоритмом всегда (и до возникновения строгой теории) понималась процедура, которая позволяла путём выполнения последовательности элементарных шагов получать однозначный результат (независящий от того, кто именно выполнял эти шаги) или за конечное число шагов прийти к выводу о том, что решения не существует.

Конечно, это нестрогое определение понятия алгоритма, и именно попытки сформулировать такое понятие привели к возникновению теории алгоритмов.

Теория алгоритмов *— это раздел математики, изучающий общие свойства алгоритмов. С возникновением и развитием вычислительной техники и смежных наук выяснилось, что в основе этих наук должна лежать теория алгоритмов.*

Существует несколько определений термина «алгоритм». Приведем некоторые из них.

Точное описание некоторого процесса (необязательно вычислительного), инструкция по его выполнению. . .[1]

[1] Демидович Н. Б., Монахов В. М. Программирование и ЭВМ. — М. : Просвещение, 1977. — 240 с.

Понятное и точное предписание (указание) исполнителю совершить последовательность действий, направленных на достижение указанной цели или на решение поставленной задачи[1].

Алгоритм — это последовательность действий со строго определенными правилами выполнения[2].

Алгоритм — это набор точных предписаний исполнителю выполнить заданную последовательность действий, направленных на достижение поставленной цели[3].

Несмотря на различие в формулировках рассматриваемого термина, есть и нечто общее — это словосочетание «последовательность действий». Действия выполняются строго друг за другом, т. е. эта последовательность строго определена. Если необходимо получить какой-то результат выполнения этих действий, их количество должно быть ограничено, т. е. конечно. Если алгоритм построен таким образом, что действия повторяются по нескольку раз, то мы говорим, что сам процесс должен быть, естественно, конечным. Что мы можем сказать о конкретном действии? Каждое действие должно быть элементарным, т. е. простым. Каждое действие должно быть понятным. Понятным оно должно быть исполнителю. А так как исполнителем алгоритма могут быть и машина, и человек, или кто-то еще, или что-то ещё, форма записи одного и того же алгоритма может быть различна. И каждое действие должно быть точным или однозначным.

Приведем простой пример составления алгоритма.

Пример 1.1

Составить алгоритм вычисления выражения $Y = (A + B) - \left(\frac{1}{A}\right)$.

Прежде всего нужно вспомнить приоритеты арифметических операций. Мы знаем, что первая операция вычисления в скобках, поэтому

- 1-й шаг: мы определим, как сложение A и B, результат запишем в переменную Z_1, т. е. $Z_1 = A + B$;
- 2-й шаг: операция деления, результат запишем в переменную Z_2, т. е. $Z_2 = \frac{1}{A}$;
- 3-й шаг: операция вычитания, результат запишем в искомую переменную Y, которая представляет собой разность сохраненных на предыдущих шагах операций Z_1 и Z_2, т. е. $Y = Z_1 - Z_2$.

Таким образом, простой алгоритм представлен тремя строчками. Для его выполнения исполнителю достаточно знать операции сложения, вычитания, деления.

---

[1] Ершов А. П., Монахов В. М. Основы информатики и вычислительной техники. — М. : Просвещение, 1985. — 96 с.

[2] Каймин В. А., Щеголев А. Г., Ерохина Е. А., Федюшин Д. П. Основы информатики и вычислительной техники. — М. : Просвещение, 1989. — 272 стр.

[3] Давыдова Н. А., Боровская Е. В. Программирование: учебное пособие. — М. : БИНОМ. Лаборатория знаний, 2012. — 238 с.

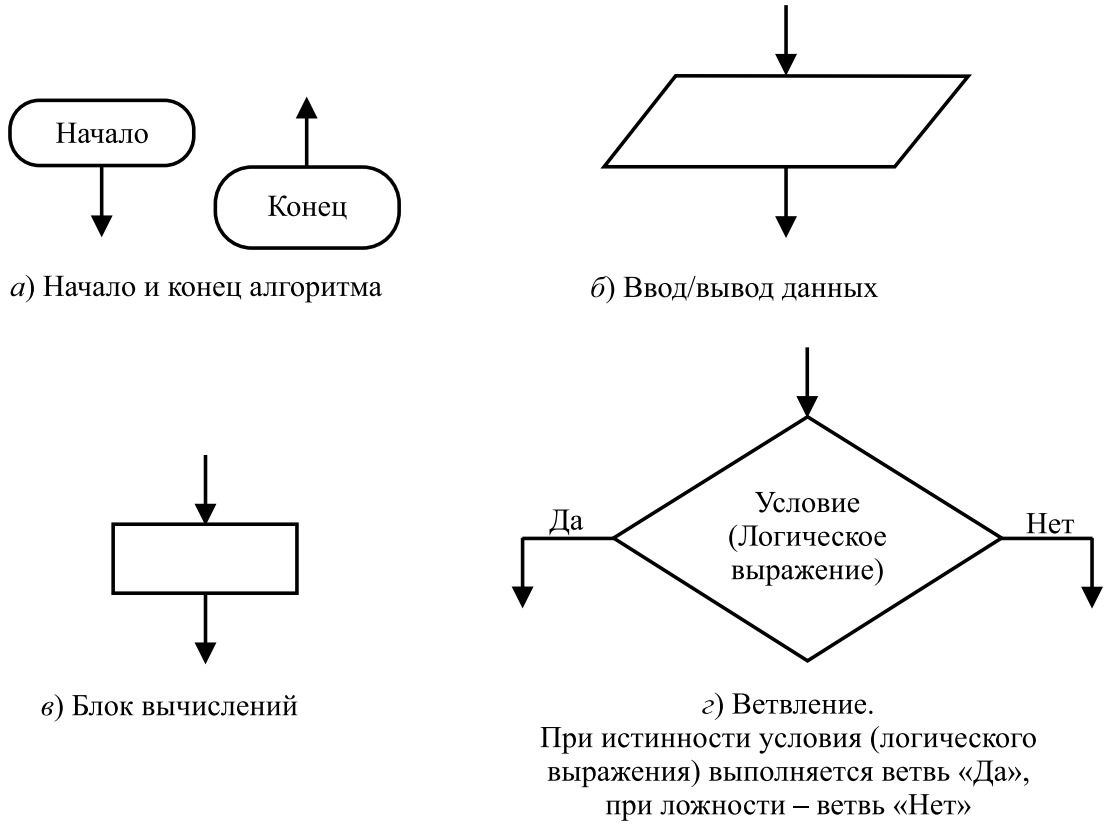
Объединяя вышесказанное, определим свойства, которыми должен обладать алгоритм.

1. Однозначность. Однозначность (она же понятность) — это свойство, которое говорит о том, что каждое действие должно быть понятно исполнителю. Каждое действие выполняется только так и никак иначе.
2. Конечность, т. е. алгоритм должен выполняться за определенное количество шагов. И если у нас некоторые действия алгоритма повторяются несколько раз, то речь идет, конечно же, о конечности процесса.
3. Корректность, т. е. алгоритм должен выдавать правильный результат. Недопустимо допущение ошибок при разработке алгоритма, поскольку это может привести как к неправильному результату, так и невозможности получить результат. Возвращаясь к примеру 1.1, можно увидеть, что ничего не сказано о переменных A и B. Составленный нами алгоритм будет некорректно работать при определенном наборе данных, а именно при A, равном B.
4. Наличие входных и выходных данных. Любой алгоритм направлен на достижение какой-то конкретной цели. Так как мы будем решать вычислительные задачи и составлять алгоритмы для написания в дальнейшем программ, то цели наших алгоритмов будут представлять собой набор некоторых данных. Нужно отметить, что сначала необходимо найти и выделить входные данные и определить цели, т. е. какие должны получиться выходные данные в данной задаче. И только после этого приступать к решению задачи и составлению алгоритма.
5. Массовость (или общность), т. е. алгоритм предназначен для решения некоторого класса задач.
6. Эффективность, т. е. составление алгоритма таким образом, чтобы поставленная задача была решена с наименьшими ресурсными затратами (например, процессорное время, объем оперативной памяти и т. д.). Оценить эффективность бывает очень тяжело. Существуют специальные методы, позволяющие оценить эффективность.

Для записи алгоритмов, например в примере 1.1, используется естественный язык. Иногда используют полуформальный язык с ограниченным словарем (часто на основе английского языка), промежуточный между естественным и языком программирования, который называют псевдокодом. Для разработки структуры программы удобнее пользоваться записью алгоритма в виде блок-схемы. Обозначения, применяемые при составлении блок-схем, приведены на рисунке 1.1.

Стрелки на блок-схемах показывают передачу управления.

Далее рассмотрим несколько примеров на составление алгоритмов.



а) Начало и конец алгоритма

б) Ввод/вывод данных

в) Блок вычислений

г) Ветвление.
При истинности условия (логического выражения) выполняется ветвь «Да», при ложности – ветвь «Нет»

Рис. 1.1 – Обозначения, используемые в блок-схемах алгоритмов

Пример 1.2

Заданы первый и второй элементы возрастающей арифметической прогрессии. Требуется вычислить N-й элемент. Причем N больше либо равно трем.

Прежде чем составлять алгоритм, необходимо проанализировать задание и найти решение задачи. Анализ задачи будет заключаться в следующем. Сначала определяем, какие данные на входе, то есть какие данные используются для решения задачи. И какие данные должны получить на выходе. Первое — на что следует обратить внимание — задана арифметическая прогрессия. Поэтому прежде чем говорить о входных и выходных данных, необходимо вспомнить, что это такое.

Арифметическая прогрессия — это некая последовательность чисел. Так как последовательность возрастающая, следовательно, для нее установлено строгое правило: то число, которое слева, меньше числа, стоящего справа. Это правило верно для любой пары чисел. Для арифметической прогрессии существует строгое правило: расстояние между соседними элементами одинаково, то есть разность между соседними элементами всегда одинакова и обозначается d.

Нам даны первый и второй элементы арифметической прогрессии. Необходимо вычислить N-ый элемент. Для этого нужно знать, чему он равен.

Итак, входными данными является: a_1 — первый элемент арифметической прогрессии, a_2 — второй элемент арифметической прогрессии и n — определяет поря-

док элемента арифметической прогрессии, который нужно найти. На выходе нужно получить значение этого элемента a_n. На основании этого можно составить алгоритм.

Блок-схема разработанного алгоритма данной задачи представлена на рисунке 1.2.

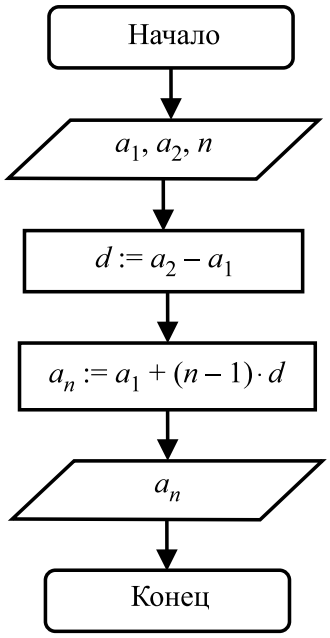


Рис. 1.2 – Блок-схема линейного алгоритма

Разработанный алгоритм является алгоритмом линейного типа.

Пример 1.3

Заданы первый и второй элементы арифметической прогрессии. Необходимо определить, является эта последовательность возрастающей или убывающей.

Начинаем с анализа задания и определения входных и выходных данных. У нас будет некий алгоритм, на входе которого есть какие-то данные. Мы уже определили, что такое арифметическая прогрессия, поэтому должно быть понятно, что заданы a_1 и a_2 — первый и второй соответственно элементы арифметической прогрессии. На выходе нужно получить сообщение: убывающая или возрастающая последовательность. Каким образом это определить? Можно сравнить a_2 и a_1. Если a_2 больше, чем a_1, последовательность возрастающая. Если наоборот, то последовательность убывающая. На основании этого можно составить алгоритм. Представим ниже его в виде блок-схемы.

При такой структуре алгоритма у нас выполняются не все действия. И такие алгоритмы называются алгоритмами с ветвлением.

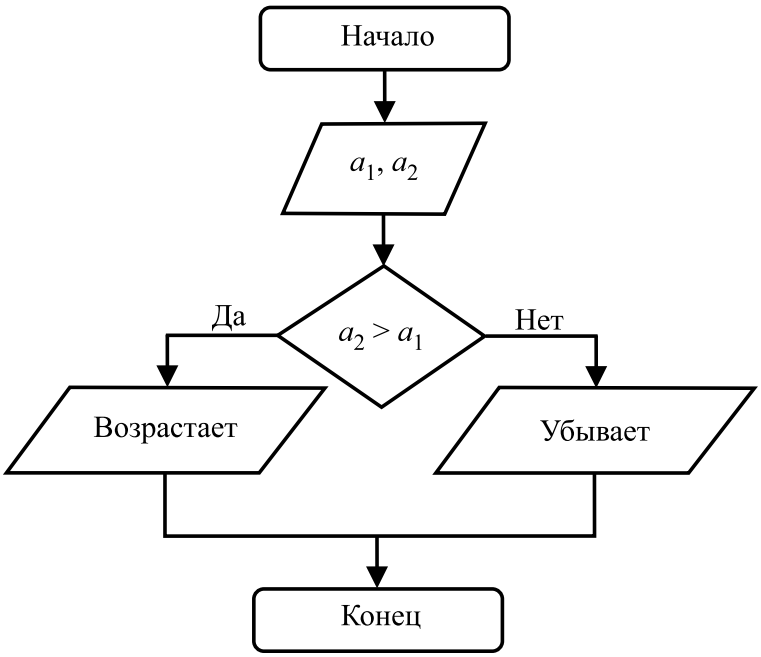


Рис. 1.3 – Блок-схема алгоритма с ветвлением

Пример 1.4

Заданы первый и второй элементы возрастающей арифметической прогрессии. Требуется вычислить N-й элемент. Но здесь есть небольшое ограничение. Исполнитель этого алгоритма умеет только складывать и сравнивать, то есть он не умеет умножать.

Данный пример похож на пример 1.2, за исключением введенного ограничения. Вспомним, что в предыдущем примере была использована формула $a_n := a_1 + (n - 1) \cdot d$. В данном примере она не может быть уже применена, поэтому разработанный ранее алгоритм требует доработки или требуется разработка нового алгоритма решения данной задачи.

Итак, нам дана арифметическая прогрессия, причем возрастающая, где каждый следующий элемент этой прогрессии отстает от предыдущего на одну и ту же величину, которая обозначается буквой d. То есть a_1 — первый элемент, следующий элемент можно записать, как $a_2 := a_1 + d$. Далее элемент $a_3 := a_2 + d$, $a_4 := a_3 + d$. Тогда соответственно $a_n := a_n - 1 + d$; $a_n + 1 := a_n + d$ и т. д.

Как узнать, что найден нужный элемент? Можно просто подсчитать. То есть каждый раз, когда находим очередной элемент, мы значение некоторой переменной увеличиваем на единицу. Эта переменная обычно называется счетчиком. И как только значение этой переменной достигнет искомого значения, можно с уверенностью сказать, что найден элемент a_n. Остается только определить, какого значения нужно достигнуть и из какого значения нужно искать эту последовательность.

Таким образом, на входе нашего алгоритма даны первый и второй члены арифметической прогрессии, а также номер того элемента арифметической прогрессии,

который нужно найти. На выходе нужно найти значение a_n — число, находящееся на n-й позиции данной арифметической прогрессии. Поскольку известны первый и второй члены прогрессии, сначала нужно найти третий элемент последовательности, затем четвертый, пятый и так до тех пор, пока не найдем n-ый. Если находить элементы начиная с a_3, для которого будем к a_2 прибавлять d, потом еще раз d, потом к полученному результату еще раз d и т. д., то таким образом мы должны выполнить $(n-2)$ сложений, чтобы достичь результата.

Нужно напомнить, что d мы должны определить в самом начале, как разность между вторым и первым членами ряда. Поскольку арифметическая прогрессия у нас возрастающая, то $a_2 > a_1$. Теперь можно записать алгоритм, используя для этого блок-схему.

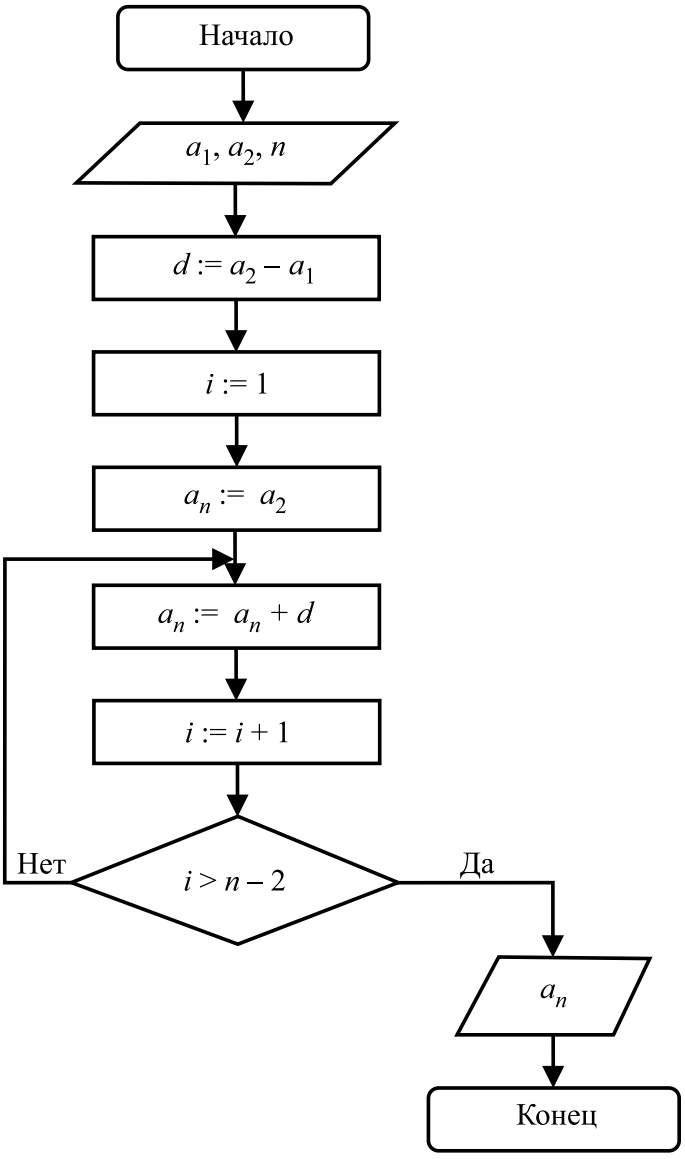


Рис. 1.4 – Блок-схема алгоритма с циклом

Представленный на рисунке 1.4 алгоритм называется алгоритмом с циклом, где последовательность действий повторяется после проверки условия цикла (на блок-схеме отображается в ромбе).

В заключение первой главы необходимо заметить, что к вопросам разработки алгоритмов придется возвращаться перед написанием программ. Рассмотренные примеры с их решением и составленными алгоритмами представляют собой необходимый минимум для понимания данной темы. Они охватывают малую часть из всего многообразия существующих классов задач. Тем не менее в приведенной литературе, рекомендованной к изучению, можно встретиться с уже разработанными алгоритмами, которым возможно найти практическое применение, что поспособствует более детальному освоению данной темы.

Контрольные вопросы по главе 1

1. Что изучает теория алгоритмов?
2. Дайте определение термину «алгоритм»?
3. Перечислите и поясните свойства алгоритмов.
4. Для чего используется блок-схема?
5. Какие обозначения используются при составлении блок-схем алгоритмов?

Глава 2

ОСНОВЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ FREE PASCAL

При решении задач программирования настоятельно рекомендуется придерживаться следующей последовательности действий.

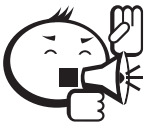
1. Постановка задачи. Здесь необходимо понять, что от вас требуется, и убедиться, что условия задачи непротиворечивы и достаточны (корректно поставлены). Сказочным примером некорректно поставленной задачи является приказ: «Пойди туда, не знаю куда, и найди то, не знаю что». Как известно, в сказке этот приказ был выполнен, в реальной жизни такой приказ является невыполнимым.
2. Выбрать метод решения. Как правило, сложные задачи могут быть решены несколькими способами. Если в условии задачи не указывается метод её решения, то выбираете оптимальный для вас или изобретаете свой.
3. Составить алгоритм решения задачи, представив его в виде блок-схемы.
4. Написать программу по блок-схеме. Начиная с этого этапа, вам понадобится знание какого-либо языка программирования, например Free Pascal, который, собственно, и рассматривается в данной главе.
5. Отладить и тестировать программу, т. е. исправить синтаксические и семантические ошибки при компиляции и логические ошибки при решении задачи с такими исходными данными, когда ответ известен заранее.
6. Исполнить программу с заданными исходными данными.
7. Проанализировать полученные результаты и сделать выводы.

2.1 Алфавит языка

Алфавит *— это совокупность допустимых в языке символов или групп символов, рассматриваемых как единое целое.*

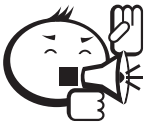
Идентификатор *— это неделимая последовательность символов, задающая имя любого объекта программы.*

В роли объектов выступают: константы, переменные, процедуры, функции, модули и подпрограммы.



Правила написания идентификаторов. Идентификатор может начинаться только с латинской буквы или символа подчёркивания «_», содержать только буквы латинского алфавита, символ подчёркивания и цифры (`1val`, `1_rez`, `first param`, `last-par`, `ab$d` — примеры неправильного написания идентификаторов). Прописные и строчные буквы в идентификаторах не различаются: так, например, идентификаторы `NAME, Name` или `nAmE` будут идентичны.

В данном пособии при написании примеров программ, с целью облегчить восприятие текста программы, зарезервированные слова и стандартные идентификаторы будут записываться с заглавной буквы, а идентификаторы, вводимые пользователем, — с маленькой буквы.



Разделители предназначены для отделения друг от друга элементов программы.

В качестве разделителей чаще всего используются пробел, [Enter], [Tab] и комментарии. Сочетания разделителей позволяет более наглядно представить структуру программы.

Комментарии предназначены для пояснения исходного текста программы.

Различают однострочные и многострочные комментарии. Однострочные комментарии начинаются с символов // (две подряд косые черты без пробела), после которых весь текст строки является комментарием. Многострочные помещаются либо в скобки `{ }`, либо в скобки вида `(* *)` и могут занимать любое количество строк. Не допускается вложение одного многострочного комментария в другой. В IDE комментарии выделяются серым цветом. При исполнении программы все комментарии игнорируются.

Пример 2.1

```
//однострочный комментарий
(*многострочный, занимающий одну строку*)
{а этот многострочный
 комментарий занимает
 несколько строк
}
//несколько строк можно сделать
//комментарием, используя однострочный
```

Специальные символы, выполняющие в языке определённые функции, можно разделить на знаки пунктуации (см. табл. 2.1), знаки операций и зарезервированные слова.

Таблица 2.1 – Специальные символы

Знак	Назначение
{ }	Скобки комментария
(* *)	Скобки комментария
//	Однострочный коментарий
[ ]	Обозначение множеств, выделение индексов элементов массивов или строк
( )	Выделение выражений, списков параметров
'	Апостроф для выделения символа или строки
:=	Знак присваивания значения переменной, типизированной константе или функции
;	Разделение операторов и объявлений в программе
:	Отделение переменной или типизированной константы от типа
=	Отделение идентификатора типа от описания типа или константы от её значения
,	Разделение элементов списка
..	Разделение границ диапазона
.	Обозначение конца программы, отделение целой части от дробной в вещественном числе, отделение полей в записи

Знаки операций обозначают арифметические, логические и другие действия. Они бывают двух типов: состоящие из небуквенных символов (например, `+, -, *, /` и т. п.) и сочетания букв (например, **`Not`**, **`Div`**, **`Mod`** и т. д.), представляющие собой зарезервированные слова.

Зарезервированные (ключевые) слова составляют основу языка и не могут переопределяться пользователем. При наборе программы в IDE зарезервированные слова выделяются белым цветом.

Неиспользуемые символы, такие, как, например, `%, &, «` и т. д., включая буквы русского алфавита, можно вставлять в комментарии и символьные строки.

2.2 Структура программы

В программе могут быть следующие, начинающиеся с собственного зарезервированного слова, разделы:

1. Заголовок программы.

   Program имя_программы;

   Слово **Program** означает начало программы, а имя_программы — идентификатор, вводимый пользователем по правилам, описанным в п. 2.1.

2. Раздел подключения модулей.

   Uses имя_модуля_1, имя_модуля_2,..., имя_модуля_N;

   После указателя используемых модулей Uses через запятую перечисляются все имена стандартных модулей (библиотек подпрограмм) и модулей пользователя (см. пп. 4.2, 4.3) необходимых для выполнения программы.

3. Раздел объявления констант.

```
Const имя_константы_1 = значение_1;
      имя_константы_2 = значение_2;
      ...;
      имя_константы_N = значение_N;
```

Константами *называются элементы данных программы, значения которых нельзя изменить в процессе её выполнения.*

Количество, имена констант и их значения задаются программистом в зависимости от выполняемой задачи.

Пример 2.2

Объявление обычных констант.

```
Const i=100;        //целочисленная константа
      r=0.5;        //вещественная константа
      expr=i/r;     //константное выражение
      c='Y';       //символьная константа
      s='Pascal'; //строковая константа
```

4. Раздел объявления типов.

```
Type имя_типа_1 = описание_1;
     имя_типа_2 = описание_2;
     ...;
     имя_типа_N = описание_N;
```

   Типы данных рассматриваются в п. 2.4 и п. 3.

5. Раздел объявления переменных.

```
Var список_переменных_1: тип_1;
    список_переменных_2: тип_2;
    ...;
    список_переменных_N: тип_N;
```

Переменными *называются элементы данных программы, значения которых могут изменяться в процессе её выполнения.*

Все используемые в программе переменные должны быть объявлены с указанием их типов. Раздел объявления переменных начинается зарезервированным словом **Var**, далее следуют объявления конкретных переменных, состоящие либо из одного идентификатора переменной, либо из группы переменных, в которой через запятую перечисляются идентификаторы одного типа, затем вводятся двоеточие и тип.

Пример 2.3

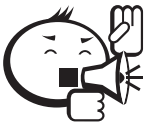
Объявление переменных простых стандартных типов.

```
Var
  i, j, k: Integer; //группа целочисленных переменных
  x, y, z: Real;    //группа вещественных переменных
  c: Char;          //символьная переменная
  flag: Boolean;    //логическая переменная
```

6. Раздел объявления подпрограмм.
   Разработка и применение процедур и подпрограмм-функций описываются в п. 4.1.
7. Тело программы (обязательная часть).

```
Begin
    оператор_1;
    оператор_2;
    ...;
    оператор_N;
End.
```

Тело программы содержит отделяемые друг от друга точкой с запятой операторы, которые выполняют алгоритмические действия, направленные на решение какой-либо задачи. Оно начинается зарезервированным словом **Begin** и заканчивается зарезервированным словом **End** с точкой (**End.**), которая является признаком конца программы. **Begin** и **End** ещё называют операторными скобками.



В ранних версиях языка Pascal заголовок программы был обязательной частью, и разделы должны были располагаться в указанном выше порядке. В Free Pascal заголовок является необязательной частью и может отсутствовать, а порядок размещения разделов произвольный (можно создавать несколько одинаковых разделов), следует только выполнять правило: в любом месте программы можно использовать лишь элементы (константы, типы, переменные и подпрограммы), определённые ранее по тексту программы.

Однако на этапе обучения программированию рекомендуется придерживаться вышеизложенной структуры программы.

Пример 2.4

Простая программа, которая вычисляет сумму двух чисел, вводимых с клавиатуры, и выводит результат на экран. В начале каждой строки в скобках приводится номер строки. Этот номер нужен только для удобства изложения, в программе его набирать, разумеется, не надо.

```
(1) Program summa;
(2) Var
(3)     x, y, sum: Real;
(4) Begin
(5)     WriteLN ('Введите два числа');
(6)     ReadLN (x, y);
(7)     sum:=x+y;
(8)     WriteLN ('Сумма равна ', sum)
(9) End.
```

Первая строка этого примера содержит заголовок программы с именем `summa`. Заголовок программы игнорируется при выполнении программы и носит характер пояснений. Во 2-й строке словом **`Var`** начинается раздел описания переменных. В третьей строке объявляются три переменных вещественного типа. 4-я строка — начало тела программы. В 5-й строке записан оператор, который выводит на экран строку, заключённую в апострофы. В 6-й строке записан оператор, который считывает значения переменных `x` и `y` с клавиатуры. В 7-й строке переменной sum присваивается значение суммы `x` и `y`. В 8-й строке на экран выводится текст, заключённый в апострофы, и значения переменной `sum`. Наконец, 9-я строка — это конец программы.

Запустим Free Pascal, наберём текст программы и разберём её.

Для решения поставленной задачи необходимы две переменные под вводимые числа и одна переменная под результат. Имена переменных (идентификаторы) выбираются такие, чтобы было понятно, для чего они предназначены (в программе `x`, `y` и `sum` соответственно). Стандартный вещественный тип *`Real`* выбираем по-

тому, что в условии не сказано, какого типа должны быть вводимые числа, а вещественный тип является более широким типом по сравнению с целым (например, *Integer*), т. е. он включает как дробные, так и целые числа. Другие элементы программы для решения задачи не нужны. Таким образом, из разделов со второго по седьмой (см. структуру) в программе присутствует только раздел объявления переменных **Var**.

Тело программы состоит из четырёх операторов, которые последовательно выполняются при запуске программы (комбинация клавиш [Ctrl]+[F9]). Первой выполняется стандартная процедура вывода с именем *WriteLN*, которая выводит на экран строку: `Введите два числа`. Данное указание необходимо для пояснения, что должен сделать пользователь, чтобы продолжить выполнение программы.

Второй выполняется стандартная процедура ввода с именем *ReadLN*, которая в режиме ожидания считывает с клавиатуры вводимые числа, первое для переменной `x`, второе — для `y`. Причём числа должны быть разделены одним или несколькими разделителями, а ввод заканчивается нажатием [Enter].

Третий оператор присваивает переменной `sum` значение выражения `x+y`, т. е. значение суммы переменных `x` и `y`.

Четвёртый оператор *WriteLN* выводит на экран строку: `Сумма равна`, а затем — значение переменной `sum`.

Если запустить данную программу и после строки:

```
Введите два числа
```

ввести:

`125`[Пробел]`-13`[Enter]

то на экране появится надпись:

```
Сумма равна 1.1200000000E+02
```

Увидеть эту надпись можно, нажав [Alt]+[F5].

Число `1.1200000000E+02` является вещественным представлением целого числа 112 и расшифровывается как $1.12 \cdot 10^2$.

В заключение параграфа приведем некоторые рекомендации по созданию исходных текстов программ.

При написании программы целесообразно использовать систему отступов, когда операторы, вложенные в другие операторы или операторные скобки, пишутся на строке с отступом вправо по отношению к другим операторам (обычно отступ делается клавишей [Tab]). Такое расположение операторов позволяет проще разобраться со структурой программы, понять её содержание, быстрее найти некоторые ошибки (например, отсутствие закрывающей операторной скобки **End**, соответствующей какой-либо открывающей скобке **Begin**).

Не следует в одной строке объединять несколько операторов, т. к. это затруднит локализацию ошибки во время отладки программы, поскольку минимальный выполняемый блок команд в процессе отладки соответствует одной строке текста.

Последовательности операторов, выполняющих какое-то законченное алгоритмическое действие, можно отделять от предыдущих и последующих операторов пустыми строками.

Идентификаторам программы следует давать имена, отражающие их суть. При написании идентификаторов следует широко использовать знак подчёркивания: `name_1` (имя 1), `real_vector` (вещественный одномерный массив), `type_of_mat-`

rix (тип матрицы) и т. п. Не страшно использовать длинные имена, т. к. можно скопировать их необходимое число раз.

Рекомендуется широко использовать комментарии, по крайней мере, для семантически связанных групп операторов или даже для отдельных операторов, чтобы пояснить их особенности.

Ещё раз подчеркнём, что все лишние пробелы, знаки табуляции, пустые строки, комментарии компилятором игнорируются и никак не сказываются на исполнении программы.

Любая программа из данного пособия приведена с применением вышеизложенных правил.

2.3 Организация ввода/вывода данных

Стандартные процедуры *Read* и *ReadLN* служат для выполнения операций ввода (*Read* — читать). После имени процедуры в круглых скобках через запятую перечисляются идентификаторы вводимых с клавиатуры переменных (см. пример 2.4). При исполнении этих процедур можно либо набирать данные с клавиатуры минимум через один пробел и нажимать [Enter], либо осуществлять ввод каждого значения через [Enter].

Процедура *ReadLN;* без параметров ожидает нажатия [Enter]. Если *ReadLN;* ставить в конце любой программы перед End., то можно спокойно ознакомиться с результатами и, нажав [Enter], вернуться в IDE, что удобно, т. к. не надо постоянно нажимать [Alt]+[F5] в IDE для просмотра результатов после выполнения программы (см. пример 2.5).

Операции вывода выполняются при помощи стандартных процедур *Write* и *WriteLN* (*Write* — писать). После имени процедуры в круглых скобках через запятую перечисляются выводимые элементы. Процедура *WriteLN;* без параметров выводит пустую строку.

Окончание *LN* в процедурах *ReadLN* и *WriteLN* является сокращением от *line* — строка и *new* — новая. После выполнения этих процедур курсор переводится на новую строку в отличие от процедур *Read* и *Write*.

Форматный вывод в процедурах *Write* и *WriteLN* позволяет задать необходимое представление выводимого элемента. Общий вид форматного вывода:

Write (элемент: d1: d2) или *WriteLN (элемент: d1: d2),*

где d1 — общее количество позиций, выделяемых под выводимый элемент; d2 — количество позиций, выделяемых под цифры после десятичной точки, в вещественном числе. Если количество символов выводимого элемента меньше d1, то происходит выравнивание по правому краю отведённых позиций, а если больше — автоматическое добавление недостающих позиций. При выводе чисел необходимо учитывать, что знак минус в отрицательном числе и десятичная точка в вещественном числе занимают по одной позиции.

Пример 2.5

```
//Программа, демонстрирующая форматный вывод
Begin
    //Вывод слова в середину строки
  WriteLN ('Привет': 43);
    {Вывод встроенной константы Pi с точностью до 2-го
     знака после десятичной точки}
  WriteLN ('Pi=', Pi: 4: 2);
    //Задержка выполнения программы до ввода Enter
  ReadLN;
End.
```

Данная программа состоит только из одного раздела — тела программы, содержащего три оператора. Заголовок программы заменён комментарием в первой строке.

В программе нет элементов, требующих подключения модулей и описаний в разделах объявлений. Поэтому разделы с 1 по 7 в структуре программы (п. 2.2) отсутствуют.

Первый оператор выводит слово `Привет` в середину строки. Откуда взялся формат `43`? В стандартном текстовом режиме в строке содержится 80 символов, т. о., 40-й символ — это середина строки. Само слово состоит из 6 символов, поэтому, учитывая выравнивание по правой границе, получаем $40 + 6/2 = 43$.

Второй оператор сначала выводит текст `Pi=`, а затем значение числа π с точностью до второго знака после десятичной точки. Формат `Pi: 4: 2` задаёт, что под выводимое вещественное число всего отводится 4 символа, из них 2 символа — под цифры после точки, которая также занимает позицию. На экран выводится строка:

```
Pi=3.14
```

Причём происходит не просто вывод цифр после десятичной точки, а округление по общим правилам до заданного знака.

Если в примере 2.4 строку:

```
WriteLN ('Сумма равна ', sum)
```

изменить как:

```
WriteLN ('Сумма равна ', sum: 0: 1)
```

то при вводе:

`125`[Пробел]`-13`[Enter]

на экране появится надпись:

```
Сумма равна 112.0
```

В заданном формате `sum: 0: 1` ноль означает, что количество позиций под вывод вещественного числа заранее неизвестно, поэтому необходимо их автоматическое добавление, единица задаёт округление до первого знака после десятичной точки. Очевидно, что форматный вывод вещественных чисел более нагляден и привычен.

При выполнении третьего оператора будет ожидаться ввод [Enter], что обеспечит просмотр результатов выполнения первого и второго операторов.

2.4 Типы данных

Каждая переменная, вводимая программистом, должна быть объявлена и определена в программе, чтобы под неё было отведено место в памяти компьютера. Определение переменных производится при помощи типов.

Типом данных *определяют множество допустимых значений этих данных, а также совокупность операций над ними.*

Типы данных в языке Free Pascal делятся на следующие группы:

1) простые типы;
2) структурированные типы;
3) указатели;
4) процедурные типы;
5) объектные типы.

Среди типов есть стандартные (предопределённые) и определяемые программистом в разделе, начинающемся со слова **`Type`**. В данном параграфе будут описаны только простые типы данных, структурированные типы будут описаны в п. 3, все остальные типы рассматриваться не будут.

Простые типы определяют упорядоченное множество значений элементов данных программы и делятся на вещественные, целые, символьный, логический, перечисляемый и тип-диапазон. Все простые типы, за исключением вещественных, называются *порядковыми типами* и характеризуются тем, что значения элементов таких типов расположены по порядку и можно точно назвать как предыдущее, так и последующее значение из диапазона.

Вещественные типы определяют дробные числа и представлены в языке шестью стандартными типами (см. табл. 2.2).

Таблица 2.2 – Вещественные типы данных

Тип	Диапазон значений	Число значащих цифр	Размер занимаемой памяти в байтах
`Real`	$2.9 \cdot 10^{-39}..1.7 \cdot 10^{38}$	15–16	8
`Single`	$1.5 \cdot 10^{-45}..3.4 \cdot 10^{38}$	7–8	4
`Double`	$5.0 \cdot 10^{-324}..1.7 \cdot 10^{308}$	15–16	8
`Extended`	$3.4 \cdot 10^{-4932}..1.7 \cdot 10^{4932}$	19–20	10
`Comp`	$-2^{63}..2^{63}$	19–20	8
`Currency`	–922337203685477.5808 ..922337203685477.5807	19–20	8

Целые типы определяют целые числа и представлены в языке девятью типами (см. табл. 2.3).

Таблица 2.3 – Целочисленные типы данных

Тип	Диапазон значений	Размер занимаемой памяти в байтах
Byte	0..255	1
Word	0..65535	2
LongWord	0..4294967295	4
ShortInt	–128..127	1
Integer	–2147483648..2147483647	4
LongInt	–2147483648..2147483647	4
SmallInt	–32768..32767	2
Int64	$-2^{63}..2^{63}$	8
Cardinal	0..4294967295	4

Стандартный *символьный тип* `Char` служит для хранения одного символа из набора в 256 символов. Занимает 1 байт.

Стандартный *логический тип* `Boolean` представляет собой тип данных, каждый элемент которого может принимать одно из двух значений: **`False`** (ложь) или **`True`** (истина).

Перечисляемый тип определяется набором идентификаторов (количеством до 255), с которыми может совпадать значение элемента данных.

```
Type имя_типа=(И1, И2,. . ., Иn);
```

где `И1, И2,..., Иn` — список идентификаторов.

Пример 2.6

Допустим, в программе присутствует раздел объявления типов:

```
Type operations=(plus, minus, multiply, divide);
```

и раздел объявления переменных:

```
Var math_characters: operations;
```

Тогда переменная `math_characters` может принимать в программе любое из четырёх значений: `plus`, `minus`, `multiply`, `divide`.

Логический тип *`Boolean`* является частным случаем перечисляемого типа, т. е. его можно представить как:

```
Type Boolean = (False, True);
```

В любом порядковом типе можно выделить подмножество значений, определяемое минимальным и максимальным значениями, в которое входят все значения исходного типа, находящиеся в этих границах, включая и сами границы. Такое подмножество определяет *тип-диапазон*.

```
Type имя_типа=ЛГ..ПГ;
```

где ЛГ — левая граница интервала (меньшее значение); ПГ — правая граница интервала (наибольшее значение).

Пример 2.7

Допустим, в программе присутствует раздел объявления типов:

```
Type dose=1..12; {подмножество целых чисел}
roman_alphabet='A'..'Z'; {подмножество символов}
```

и раздел объявления переменных:

```
Var standard1, standard2, standard3: dose;
    letter: roman_alphabet;
```

Тогда переменные standard1, standard2, standard3 могут принимать в программе 0.955 любые значения в диапазоне целых чисел от 1 до 12, а переменная letter — значения заглавных букв латинского алфавита.

Когда в тех или иных операциях или операторах присутствуют данные, относящиеся к различным типам, возникает вопрос о соответствии типов. В связи с этим говорят об идентичности и совместимости типов.

Совместимость типов требуется в выражениях (в том числе и в операциях отношения). Два типа (обозначим их T1 и T2) совместимы в следующих случаях:

- T1 и T2 — один и тот же тип или они идентичны;
- T1 и T2 — вещественные типы;
- T1 и T2 — целые типы;
- один тип вещественный, а другой — целый;
- один тип представляет собой тип-диапазон другого;
- оба типа являются типами-диапазонами какого-то одного типа;
- один тип является строкой, а другой — строкой или символом.

Совместимость для присваивания необходима, когда значение какого-то выражения присваивается переменной или функции. Если значение элемента типа T2 присваивается элементу типа T1, то это возможно в следующих случаях:

- T1 и T2 — идентичные типы и не являются файловыми типами или структурированными типами, содержащими элементы файлового типа;

- T1 и T2 — совместимые порядковые типы и значение типа T2 находится в границах возможных значений элемента типа T1;
- T1 и T2 — вещественные типы и значение типа T2 находится в границах возможных значений элемента типа T1;
- T1 — вещественный тип, а T2 — целый тип;
- T1 и T2 — строки;
- T1 — строка, а T2 — символ.

Речь о соответствии типов может идти и в некоторых других случаях, которые будут рассмотрены отдельно в разделах данного пособия.

2.5 Выражения

Выражение *— это синтаксическая единица языка, определяющая порядок вычисления некоторого значения.*

Выражения в языке Free Pascal формируются из операндов (констант, переменных, вызовов функций), знаков операций и круглых скобок.

Константы и переменные описаны в п. 2.2.

В таблице 2.4 представлены наиболее часто встречающиеся в выражениях стандартные арифметические функции.

Арифметические функции можно использовать только с величинами вещественных и целых типов.

Все операции в Free Pascal делятся на следующие группы:

- арифметические;
- логические;
- отношения;
- операции со строками;
- операции с битами информации;
- адресная операция `@`.

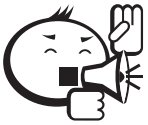
Первые три группы рассматриваются в данном разделе. Операции со строками описываются в п. 3.2. Последние две группы рассматриваться в данном пособии не будут.

Таблица 2.4 – Стандартные функции

Функция	Назначение	Тип результата
Abs(x)	Абсолютное значение аргумента `x`	Совпадает с типом `x`
Arctan(x)	Арктангенс аргумента `x`	Вещественный
продолжение на следующей странице		

Таблица 2.4 — Продолжение

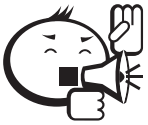
Функция	Назначение	Тип результата
`Cos(x)`	Косинус аргумента `x`	Вещественный
`Exp(x)`	e^x	Вещественный
`Ln(x)`	Натуральный логарифм аргумента `x`	Вещественный
`Pi`	Константа π = 3.1415926535897932385	Вещественный
`Sin(x)`	Синус аргумента `x`	Вещественный
`Sqr(x)`	Квадрат аргумента `x`	Совпадает с типом `x`
`Sqrt(x)`	Квадратный корень аргумента `x`	Вещественный



Арифметические операции применимы только к величинам вещественных и целых типов и делятся на унарные и бинарные операции.

В *унарной операции* присутствует только один элемент (*операнд*).

Так, унарный знак плюс «+», поставленный перед величиной либо вещественного, либо целого типа, не оказывает никакого влияния на значение этой величины. Унарный знак минус «−» в аналогичном случае приводит к изменению знака величины.



В *бинарных арифметических операциях* участвуют два операнда.

Эти операции и их знаки приведены в таблице 2.5.

Знаки операций «+», «−» и «*» применяются также и с другими типами операндов, но тогда они имеют иной смысл. В операциях деления знаменатель не должен равняться нулю. При использовании знака операции, являющегося зарезервированным словом, он должен быть отделён от операндов хотя бы одним разделителем.

Таблица 2.5 – Арифметические операции

Знак	Операция	Типы операндов	Тип результата
+	Сложение	Целые Хоть один вещественный	Целый Вещественный
−	Вычитание	Целые Хоть один вещественный	Целый Вещественный
*	Умножение	Целые Хоть один вещественный	Целый Вещественный
продолжение на следующей странице			

Таблица 2.5 — Продолжение

Знак	Операция	Типы операндов	Тип результата
/	Деление	Целые или вещественные	Вещественный
Div	Получение целого от деления целых чисел	Целые	Целый
Mod	Получение остатка от деления целых чисел	Целые	Целый

Пример 2.8

Записать выражение

$$\frac{(x+y+z)^2-\pi}{x^2-\sqrt[5]{|y|}}$$

по правилам языка Free Pascal.

Если в квадрат возводится некоторое выражение, то удобно использовать стандартную функцию *Sqr*. Получаем:

$$(x+y+z)^2 \Rightarrow \texttt{Sqr(x+y+z)}.$$

Если в квадрат возводится переменная с простым именем, то можно просто использовать перемножение: $x^2 \Rightarrow$ **x*x**. В Pascal нет стандартной функции возведения в любую степень, поэтому можно использовать свойства экспоненциальной и логарифмической функций с условием, что аргумент больше нуля. В нашем случае имеем $|y| \geqslant 0$, поэтому можно записать $\sqrt[5]{|y|} = |y|^{1/5} = e^{\ln(|y|^{1/5})} = e^{(\ln(|y|))/5}$ при $y \neq 0$. Следовательно, с использованием стандартной функции *Abs* получаем:

```
Exp(Ln(Abs(y))/5).
```

Окончательно выражение записывается в виде:

```
(Sqr(x+y+z)-Pi)/(x*x-Exp(Ln(Abs(y))/5)).
```

Числитель и знаменатель дроби необходимо помещать в скобки ( ) для сохранения порядка вычислений.

Пример 2.9

Рассмотрим операции целочисленного деления **Div** и **Mod**.

```
14 Div 3 {результат 4}
14 Mod 3 {результат 2}
```

Как получаются значения при использовании этих операций, хорошо иллюстрирует деление уголком:

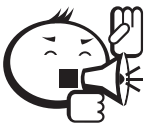
14	3
12	4 ← результат **Div**
2	← результат **Mod**

Логические операции применяются к величинам логического типа, результат таких операций — тоже логического типа.

Имеется одна унарная логическая операция **Not** (ОТРИЦАНИЕ) и три бинарные логические операции **And** (И), **Or** (ИЛИ), **Xor** (ИСКЛЮЧАЮЩЕЕ ИЛИ), которые определяются таблицей истинности 2.6.

Таблица 2.6 – Логические операции

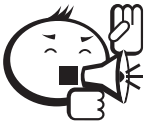
Значения операндов		**Результат операций**			
a	b	**Not** a	a **And** b	a **Or** b	a **Xor** b
False	**False**	**True**	**False**	**False**	**False**
False	**True**		**False**	**True**	**True**
True	**False**	**False**	**False**	**True**	**True**
True	**True**		**True**	**True**	**False**



Операции отношения предназначены для сравнения двух величин (величины должны быть сравнимых типов). Результат сравнения имеет логический тип.

Операции отношения имеют вид:

Символ	**Значение**
=	равно
<>	не равно
<	меньше
<=	меньше или равно
>	больше
>=	больше или равно



Круглые скобки используются для заключения в них части выражения, вычисление которой необходимо выполнить в первую очередь.

В выражении может быть любое количество круглых скобок, причём количество открывающих скобок должно быть равно количеству закрывающих скобок. Части выражений, заключённые в круглые скобки, должны быть либо непересекающимися, либо вложенными друг в друга.

Вычисление значений выражений выполняется в определённом порядке. Начинается вычисление с определения переменных и констант, входящих в выражение. Они являются основой для дальнейших вычислений. Действия выполняются в соответствии с их приоритетами. Так, в первую очередь вычисляются выражения, заключённые в круглые скобки. Для любых двух вложенных друг в друга пар круглых скобок вычисляется сначала внутреннее выражение, а затем внешнее. Далее вычисляются значения входящих в выражение функций и т. д. Приоритеты основных действий, выполняемых при вычислении выражений, приведены в таблице 2.7.

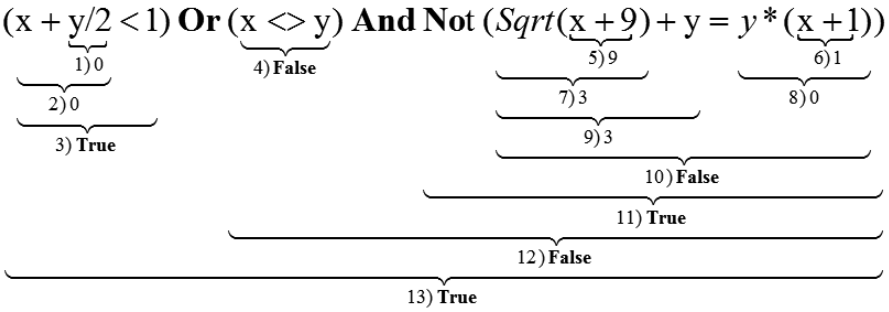
Таблица 2.7 – Приоритеты арифметических и логических операций

Группа	Тип действий	Операции или элементы
1	Вычисления в круглых скобках	`( )`
2	Вычисления значений функций	Функции
3	Унарные операции	**`Not`**, унарный +, унарный –
4	Операции типа умножения	`*`, `/`, **`Div`**, **`Mod`**, **`And`**
5	Операции типа сложения	+, –, **`Or`**, **`Xor`**
6	Операции отношения	=, <>, <, <=, >, >=

Если в выражении встречаются операции одного приоритета, то обычно они выполняются слева направо.

Пример 2.10

Расставим приоритеты выполнения операций в следующем выражении:



Первое число фигурной скобки показывает порядок выполнения операции, второе значение — её результат, если `x` и `y` равны нулю. Окончательный результат в 13-й операции — **`True`**.

2.6 Операторы языка

Операторы описывают некоторые алгоритмические действия, которые необходимо выполнить для решения задачи. Тело программы представляется как последовательность таких операторов. Идущие друг за другом операторы в программе разделяются точкой с запятой.

Все операторы делятся на две группы: простые и структурированные.

Простыми *называются операторы, которые не содержат в себе других операторов.*

К ним относятся: присваивание, обращение к процедуре, пустой оператор.

С помощью *оператора присваивания* переменной или функции присваивается значение выражения. Для этого используется знак присваивания `:=`, слева от которого записывается имя переменной или функции, а справа — выражение, значение которого вычисляется перед присваиванием. Тип выражения и тип переменной (или функции) должны быть совместимы для присваивания.

Не следует путать присвоение `:=` с операцией отношения равно `=`, когда сравниваются две величины. Также присвоение — это совсем не знак равно в математических формулах.

Пример 2.11

Совокупность математических формул $x = 5$, $x = x + 1$ является неверной и не имеет смысла, тогда как запись в Pascal:

```
x:=5;
x:=x+1;
```

вполне допустима (как и в любом другом алгоритмическом языке с учётом синтаксических особенностей). В первом операторе происходит присвоение переменной `x` значения `5`. Во втором операторе из ячейки памяти компьютера с именем `x` берётся значение `5`, складывается с `1`, и полученное значение заносится в ячейку памяти с тем же именем, т. е. окончательно в `x` будет `6`.

Такой приём широко используется, например, для изменения параметра цикла.

Оператор обращения к процедуре служит для вызова процедуры на исполнение и состоит из имени процедуры, после которого в круглых скобках через

запятую перечисляются её фактические параметры (если они есть). Примером использования таких операторов может служить любая программа, в которой предусмотрен ввод/вывод данных при помощи стандартных процедур, рассмотренных в п. 2.3. Более подробно обращение к процедурам описано в п. 4.1.

Пустой оператор не выполняет никакого действия и никак не отображается в программе.

Структурированными являются операторы, которые состоят из других операторов. К ним относятся: составной оператор, условные операторы **If** и **Case**, операторы циклов **While**, **Repeat** и **For**, оператор над записями **With**, который будет рассмотрен в п. 3.3.

Составной оператор представляет собой совокупность последовательно выполняемых операторов, заключённых в операторные скобки **Begin** и **End**:

```
Begin
    оператор_1;
    оператор_2;
    ...
    оператор_N;
End;
```

На рисунке 2.1 представлена блок-схема этого оператора. Он может потребоваться в тех случаях, когда в соответствии с правилами построения конструкций языка можно использовать один оператор, а выполнить нужно несколько действий. Тогда все операторы, выполняющие требуемые действия, помещаются в составной оператор.

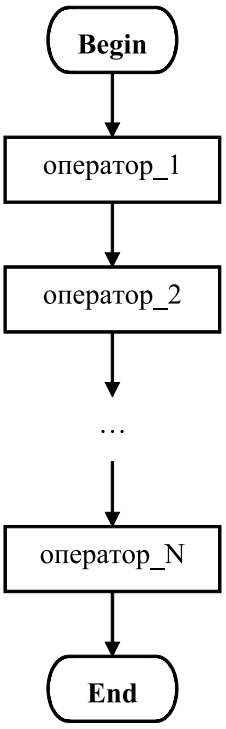


Рис. 2.1 – Блок-схема составного оператора

В дальнейшем везде, где будет указываться, что можно использовать один оператор, им может быть и составной оператор.

Тело программы, так как оно заключено в операторные скобки **Begin** и **End**, является глобальным составным оператором.

2.6.1 Условный оператор

Условный оператор **If** реализует алгоритмическую конструкцию разветвляющегося процесса и изменяет порядок выполнения операторов в зависимости от истинности или ложности некоторого логического выражения.

Существует две формы записи оператора: полная

```
If логическое_выражение Then
    оператор_1
Else
    оператор_2;
```

и сокращенная

```
If логическое_выражение Then
    оператор;
```

При истинности (**True**) логического выражения выполняется `оператор_1` в полной записи и `оператор` в сокращенной записи. При ложности (**False**) логического выражения выполняется `оператор_2` в полной записи, и ничего не выполняется в сокращенной записи (происходит переход к следующему элементу программы) (см. рис. 2.2).

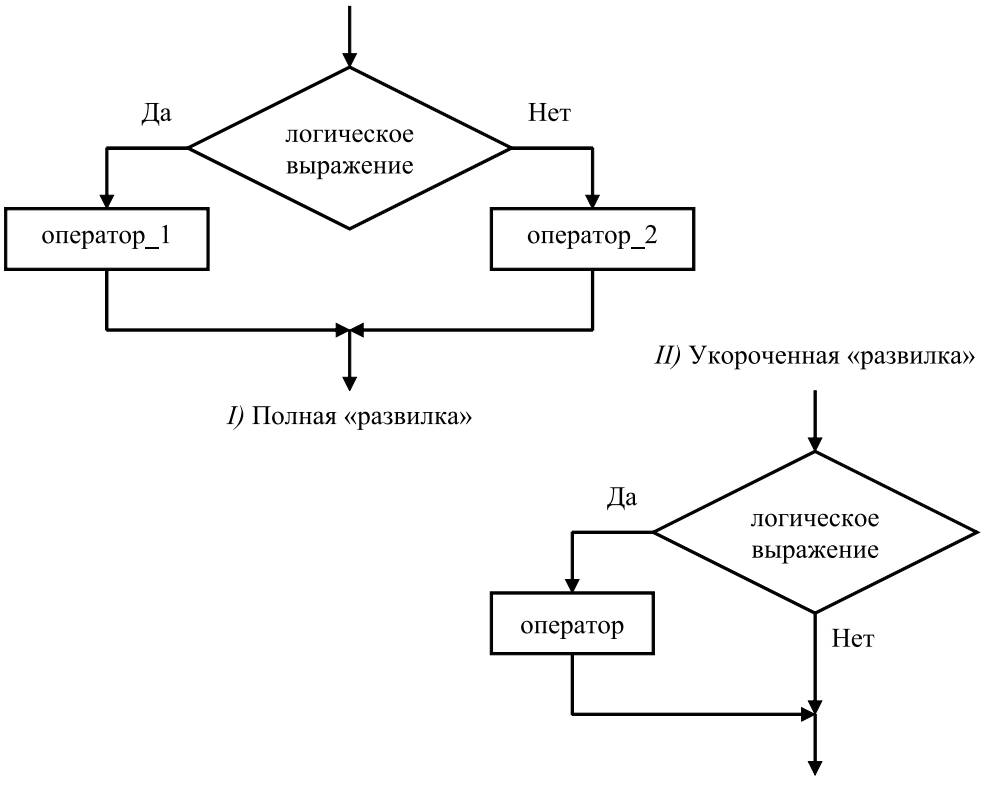


Рис. 2.2 – Блок-схемы двух видов условного оператора **If**

Так как оператор **If** является единым предложением, ни перед ветвью **Then**, ни перед ветвью **Else** точку с запятой ставить нельзя.

Пример 2.12

Рассмотрим очень подробно решение следующей задачи: найти действительные корни квадратного уравнения вида $ax^2 + bx + c = 0$*, где коэффициенты* a, b, c *могут быть любыми.*

На первый взгляд задача очень простая. Используя дискриминант $d = b^2 - 4ac$, корни уравнения вычисляем по формуле:

$$x_{1,2} = \frac{-b \mp \sqrt{d}}{2a}.$$

Составляем блок-схему алгоритма (рис. 2.3) и по ней в соответствии с правилами языка Free Pascal записываем программу.

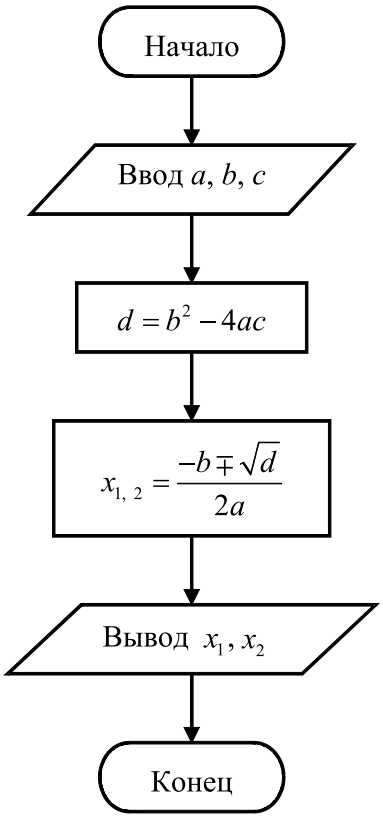


Рис. 2.3 – Блок-схема простого алгоритма вычисления корней квадратного уравнения

Программу назовём `quadratic_equation` (квадратное уравнение).

В блок-схеме на рисунке 2.3 видно, что в программе будут присутствовать следующие переменные: `a`, `b`, `c` вещественного типа, так как в условии задачи указано, что они могут быть любыми, а значит, и дробными числами; `d`, `x1`, `x2` также вещественного типа, так как при вычислении их значений используются переменные вещественного типа.

Таким образом, в программе будет раздел объявления переменных, начинающийся со слова **`Var`**, а остальные разделы объявлений будут отсутствовать (см. п. 2.2).

Блок «Начало» на рисунке 2.3 соответствует слову **Begin** — началу тела программы, блок «Конец» соответствует слову **End** с точкой — концу программы.

Ввод и вывод данных осуществляются при помощи стандартных процедур (см. п. 2.3).

Запускаем Free Pascal и в его интегрированной среде вводим следующую программу:

```
Program quadratic equation;
{Объявление и определение вещественным типом группы
переменных}
Var a, b, c, d, x1, x2: Real
//Начало тела программы
Begin
     //Ввод коэффициентов с клавиатуры
     ReadLN (a, b, c);
     //Вычисление дискриминанта
     d:=b*b-4*a*c;
     {Вычисление корней с использованием функции Sqrt
     см. таблицу 2.4}
     x1:=(-b-Sqrt(d))/2*a;
     x2:=(-b+Sqrt(d))/2*a;
     {Вывод на экран поясняющих надписей и значений корней
      точностью до третьего знака после десятичной точки}
     WriteLN (x1=, x1: 0: 3, x2=, x2: 0: 3);
//Конец программы
End.
```

Нажав [Ctrl]+[F9], запускаем программу и получаем первое сообщение об ошибке (см. рис. 2.4).

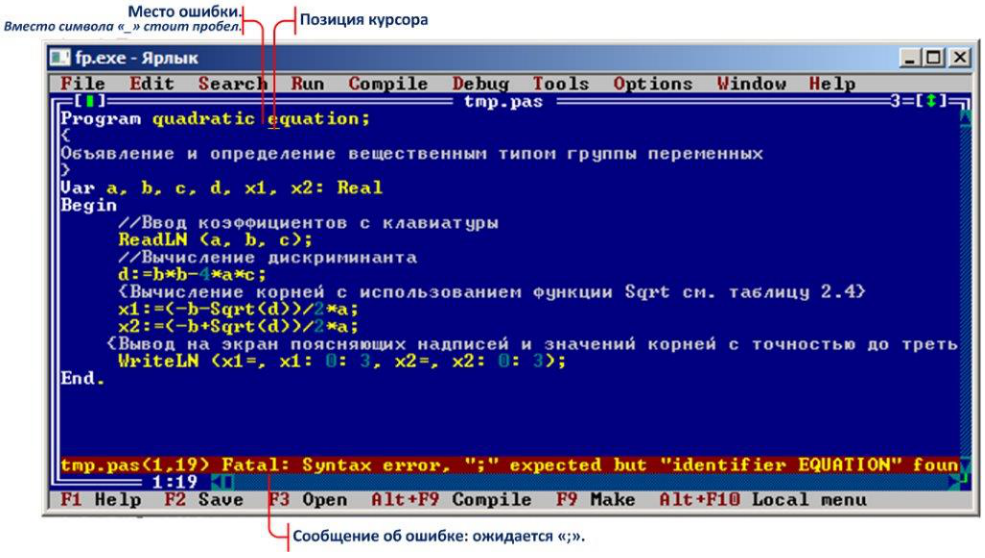


Рис. 2.4 – Экран IDE с первой ошибкой в программе `quadratic_equation`

В данном случае компилятор не вполне верно определил ошибку (Ожидается «;»), и если после `quadratic` поставить точку с запятой, то это не решит проблемы, а приведёт к следующей ошибке: **`BEGIN`** expected (Ожидается **`BEGIN`**). На самом деле, ошибка заключается в неправильном написании идентификатора — имени программы, а именно пропущен символ подчёркивания между `quadratic` и `equation` (см. п. 2.1).

Исправив ошибку, снова запускаем программу нажатием [Ctrl]+[F9].

На рисунке 2.5 представлено окно с ошибкой, аналогичной предыдущей, но определённой в другом месте программы.

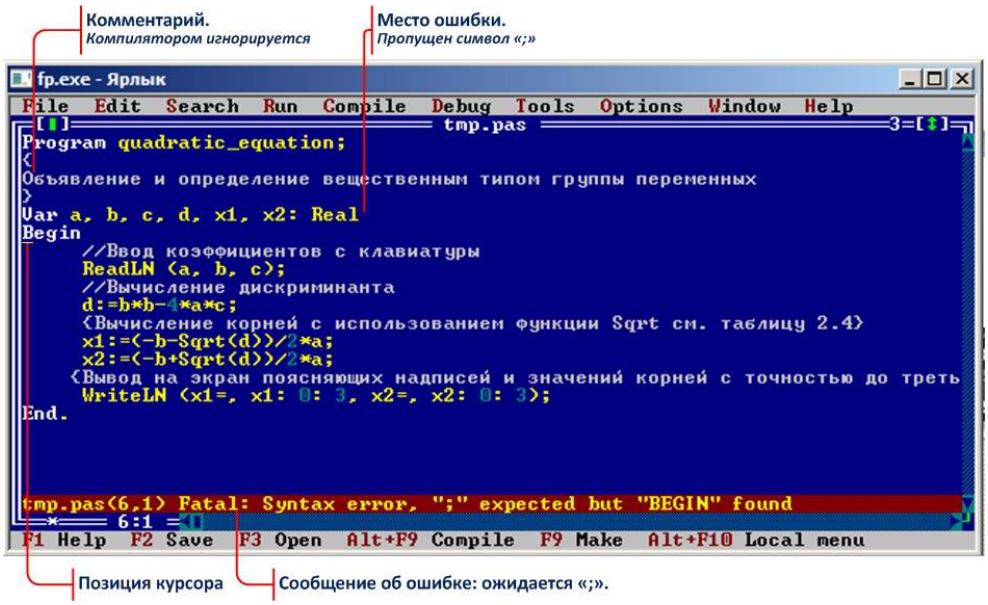


Рис. 2.5 – Экран IDE со второй ошибкой в программе `quadratic_equation`

Анализируя программу выше курсора и учитывая, что комментарии компилятором игнорируются, выясняем, что в разделе **`Var`** после группы переменных и их типа пропущена точка с запятой (см. п. 2.2 раздел объявления переменных).

Первая и вторая ошибки относятся к синтаксическим ошибкам.

Исправляем вторую ошибку и запускаем программу.

Третья ошибка на рисунке 2.6 является семантической ошибкой в задании параметров процедуры вывода *`WriteLN`*. Строковые константы `x1=` и `x2=`, выводящиеся на экран и играющие роль пояснительных надписей, должны быть заключены в апострофы. В начале строки `x2=` после апострофа можно поставить пробел, чтобы эта строка не сливалась с предварительно выводимым значением переменной `x1`.

После очередного запуска программы появляется чёрный экран с мигающим курсором, приглашающим к вводу данных с клавиатуры, если, конечно, помимо заложенных и разобранных выше ошибок вы не наделали собственных. В этом случае внимательно сверьте набранную программу с программой в данном пособии.

Для проверки работы программы введём такие коэффициенты, чтобы результат был известен, например корни уравнения $3x^2 + 4.5x - 3 = 0$, в котором $a = 3$, $b = 4.5$, $c = -3$, будут $x_1 = -2$ и $x_2 = 0.5$. Набираем с клавиатуры:

3[Пробел]4.5[Пробел]-3[Enter]

и получаем:

```
x1=-18.000 x2=4.500
```

Чтобы просмотреть результаты, необходимо нажать [Alt]+[F5].

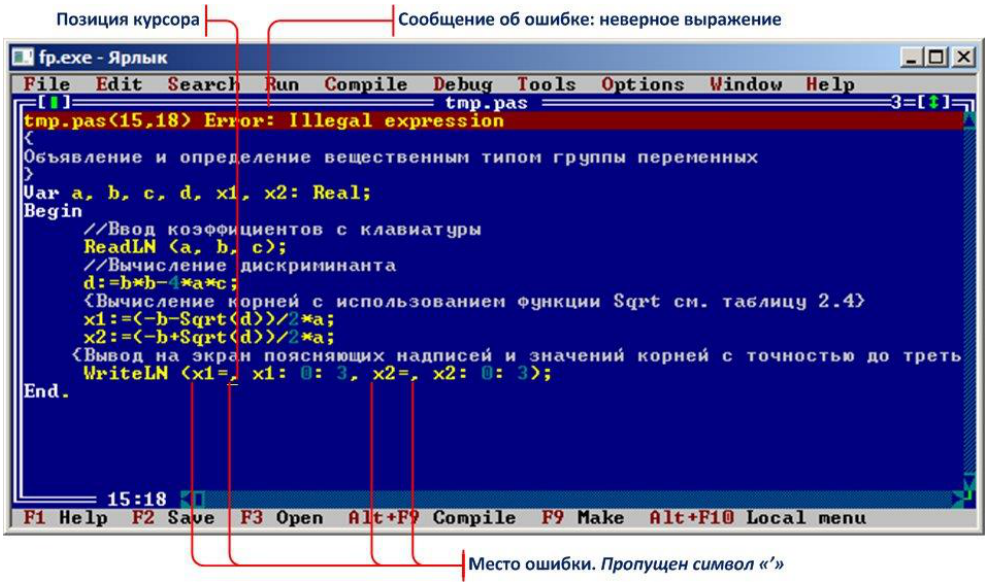


Рис. 2.6 – Экран IDE с третьей ошибкой в программе `quadratic_equation`

Очевидно, что программа производит неправильные вычисления. Для поиска логической ошибки используем отладчик.

В основном меню выбираем пункт Debug (отлаживать), в открывшемся подменю выполняем команду Watches (просмотр). В окно Watches вносим контролируемые элементы данных программы, нажимая клавишу [Insert] и набирая идентификатор с клавиатуры в окне Edit Watch в поле Expression to watch. В нашем случае контролируемыми элементами данных будут вычисляемые в программе переменные `d`, `x1`, `x2` (см. рис. 2.7).

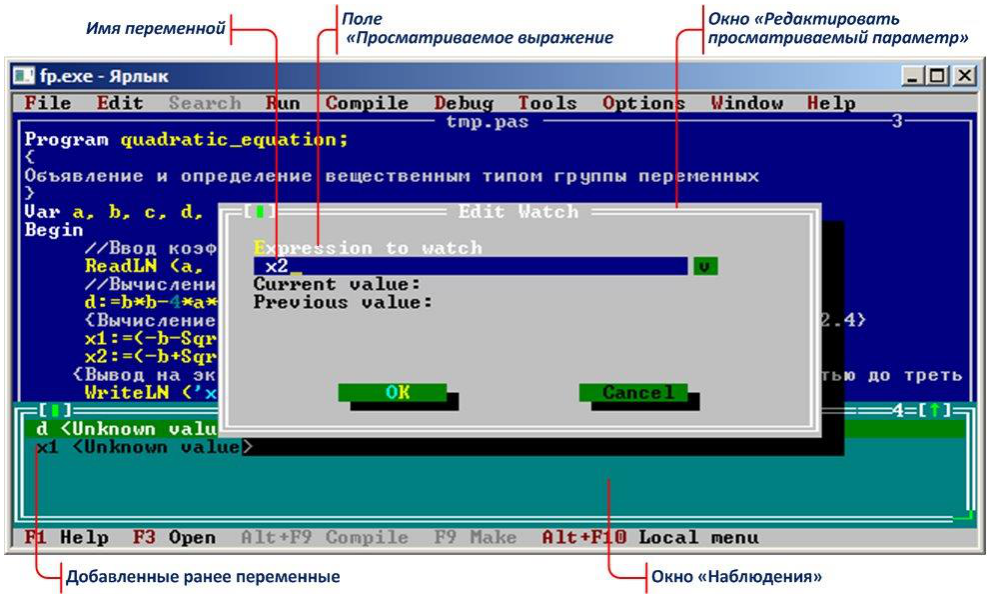


Рис. 2.7 – Экран IDE с окнами `Watches` и `Edit Watch`

После того, как в окно Watches введены все интересующие нас переменные, начинаем пошаговое выполнение (трассировку) программы. Это можно сделать, запуская команду Trace into (трассировка с заходом в подпрограммы) подменю Run (пуск) основного меню или, что удобнее, нажимая [F7]. При одном нажатии [F7] выполняется одна строка программы, а следующая выделяется голубым цветом, одновременно в окне Watches показываются значения переменных.

После вычисления дискриминанта в окне Watches появляется значение переменной d (см. рис. 2.8). Вычисляем это значение сами: $d = b^2 - 4ac = 4.5^2 - 4 \cdot 3 \cdot (-3) = 56.25$. Значения совпадают, поэтому делаем вывод, что оператор `d:=b*b-4*a*c` записан правильно.

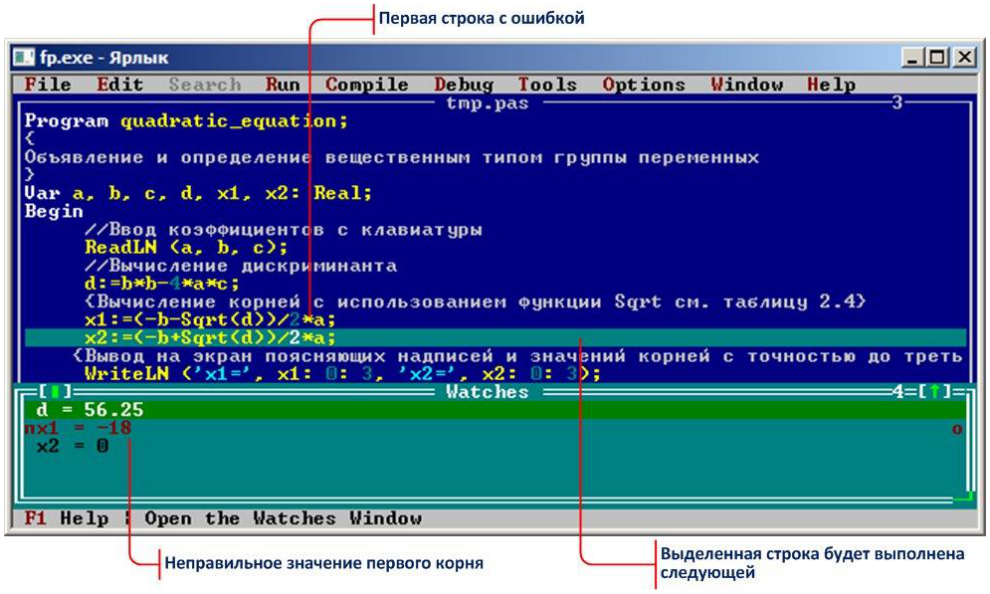


Рис. 2.8 – Экран IDE с трассировкой программы `quadratic_equation` и с окном `Watches`

При вычислении выражения `(-b-Sqrt(d))/2*a` получили −18.0, а должно быть −2 (см. рис. 2.8). Проверяя это выражение, находим, что знаменатель не взят в круглые скобки, и поэтому вместо $x_1 = \left(-b - \sqrt{d}\right)/2a$ имеем $x_1 = \left(-b - \sqrt{d}\right)/2a$. Аналогичная ошибка находится и в следующем операторе.

Исправляем эти ошибки:

```
x1:=(-b-Sqrt(d))/(2*a);
x2:=(-b+Sqrt(d))/(2*a);
```

и запускаем программу с исходными данными:

3[Пробел]4.5[Пробел]−3[Enter]

получая на этот раз правильный ответ:

```
x1=-2.000 x2=0.500
```

Казалось бы — задача решена, но запустим программу с исходными данными:

1[Пробел]1[Пробел]1[Enter]

Опять ошибка!

Exited with exitcode = 207. (Ошибка 207: Недопустимая операция с плавающей точкой.)

Вычисляем дискриминант $d = b^2 - 4ac = 1^2 - 4 \cdot 1 \cdot 1 = -3$. Оказывается, что в программе не предусмотрен случай, когда дискриминант меньше нуля (т. е. действительных корней уравнения нет) и вычисление функции `Sqrt(d)` приводит к вышеописанной ошибке. Чтобы её избежать, необходимо использовать условный оператор **If** в следующем алгоритме: если (**If**) дискриминант меньше нуля, тогда (**Then**) вывести сообщение, что действительных корней нет, иначе (**Else**) вычислить корни и вывести полученный результат.

Проверим другие комбинации вводимых коэффициентов a, b, c.

Если (**If**) $a = 0$ и $b = 0$, тогда (**Then**) уравнение $ax^2 + bx + c = 0$ не является уравнением и в общем случае при $c \neq 0$ тождеством.

Если (**If**) $a = 0$, тогда (**Then**) уравнение $ax^2 + bx + c = 0$ (при $b \neq 0$) является линейным уравнением, корень которого вычисляется по формуле $x = -c/b$.

Составляем новую блок-схему алгоритма вычисления корней квадратного уравнения (рис. 2.9). В этой блок-схеме предусмотрены все случаи вводимых коэффициентов a, b, c.

Следуя блок-схеме, напишем окончательный вариант программы. (В первых позициях каждой строки в круглых скобках приведен номер строки, который нужен только для удобства дальнейшего изложения.)

```
{Улучшенная программа вычисления корней
 квадратного уравнения}
(1) Program best_quadratic_equation;
(2) Var a, b, c, d, x1, x2: Real;
(3) Begin
(4)  WriteLN ('Введите коэффициенты a, b, c');
(5)  ReadLN (a, b, c);
(6) If (a=0) And (b=0) Then
(7)    WriteLN ('Неправильный ввод a, b, c')
(8) Else
(9)    If a=0 Then
         Begin
(10)        x1:=-c/b;
(11)        WriteLN ('Корень линейного уравнения ',
            x1: 0: 3);
(12)     End
(13)     Else
(14)     Begin
(15)        d:=b*b-4*a*c;
(16)        If d<0 Then
(17)           WriteLN ('Действительных корней нет')
(18)        Else
(19)        Begin
(20)           x1:=(-b-Sqrt(d))/(2*a);
(21)           x2:=(-b+Sqrt(d))/(2*a);
(22)           WriteLN ('x1=', x1: 0: 3, 'x2=',
               x2: 0: 3);
(23)        End;
```

```
(24)      End;
(25)    ReadLN;
(26) End.
```

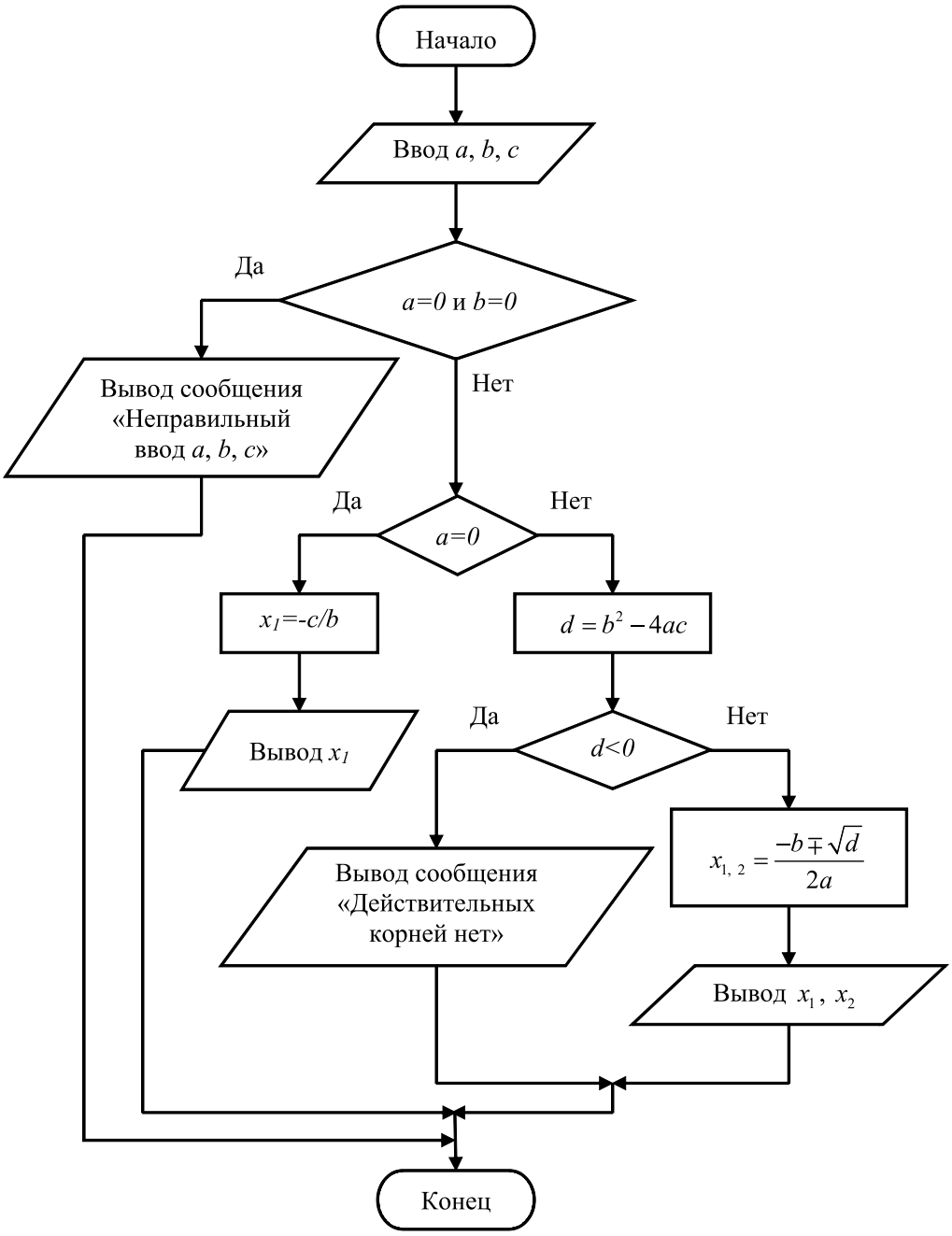


Рис. 2.9 – Блок-схема улучшенного алгоритма решения квадратного уравнения

Подробно разберем эту улучшенную программу вычисления корней квадратного уравнения. Во 2-й строке объявляется группа переменных вещественного типа. Следующая строка (**Begin**) — начало тела программы, которая завершается парной к ней 26-й строкой (**End.**). В следующей строке на экран выводится поясняющая надпись 'Введите коэффициенты a, b, c'. В 5-й строке осуществляется ввод коэффициентов с клавиатуры. Большую часть программы составляет услов-

ный оператор (строки с 6-й по 24-ю). Если записанное в заголовке этого оператора условие (a=0 и b=0) истинно, то выполняется записанный в 7-й строке оператор вывода на экран, после чего управление передается на 25-ю, предпоследнюю строку программы. В противном случае (условие (a=0 и b=0) ложно, т. е. хотя бы один из этих коэффициентов отличен от нуля) выполняется условный оператор, занимающий в программе строки с 9-й по 24-ю. Опять, если записанное в заголовке этого оператора условие (a=0) истинно, то выполняются заключенные в операторные скобки (**Begin** и **End**)два оператора, записанные в 10-й и 11-й строках (вычисление единственного корня уравнения и вывод его значения на экран), после чего управление передается на 25-ю, предпоследнюю строку программы. В противном случае (условие (a=0) ложно) выполняется составной оператор, занимающий в программе строки с 14-й по 24-ю.

Этот составной оператор, в свою очередь, состоит из двух операторов — оператора присваивания (вычисление дискриминанта d в 15-й строке) и условного оператора (строки с 16-й по 23-ю). В очередной раз, если записанное в заголовке этого оператора условие (d<0) истинно, то выполняются оператор вывода на экран информации об отсутствии действительных корней (17-я строка), после чего управление опять передается на 25-ю, предпоследнюю строку программы. В противном случае (условие (d<0) ложно) выполняется составной оператор, занимающий в программе строки с 19-й по 23-ю. Во всех случаях последним выполняется записанный в 25-й строке оператор *ReadLN*, предназначенный для задержки выполнения программы до нажатия Enter.

2.6.2 Оператор выбора

Оператор выбора **Case** позволяет выбрать вариант из любого количества вариантов. Общий вид оператора:

```
Case s Of
    c1: оператор_1;
    c2: оператор_2;
    ...
    cN: оператор_N
    Else оператор
End;
```

Здесь s — переменная или выражение порядкового типа; c1, c2,..., cN — список констант, с которыми сравнивается значение s. Если какая-либо константа c1, c2,..., cN совпадает со значением s, то выполняется соответствующий константе оператор, следующий после двоеточия. Если значение s не совпадает ни с одной из констант c1, c2,..., cN, то выполняется оператор после слова **Else** (см. рис. 2.10 и пример 2.13).

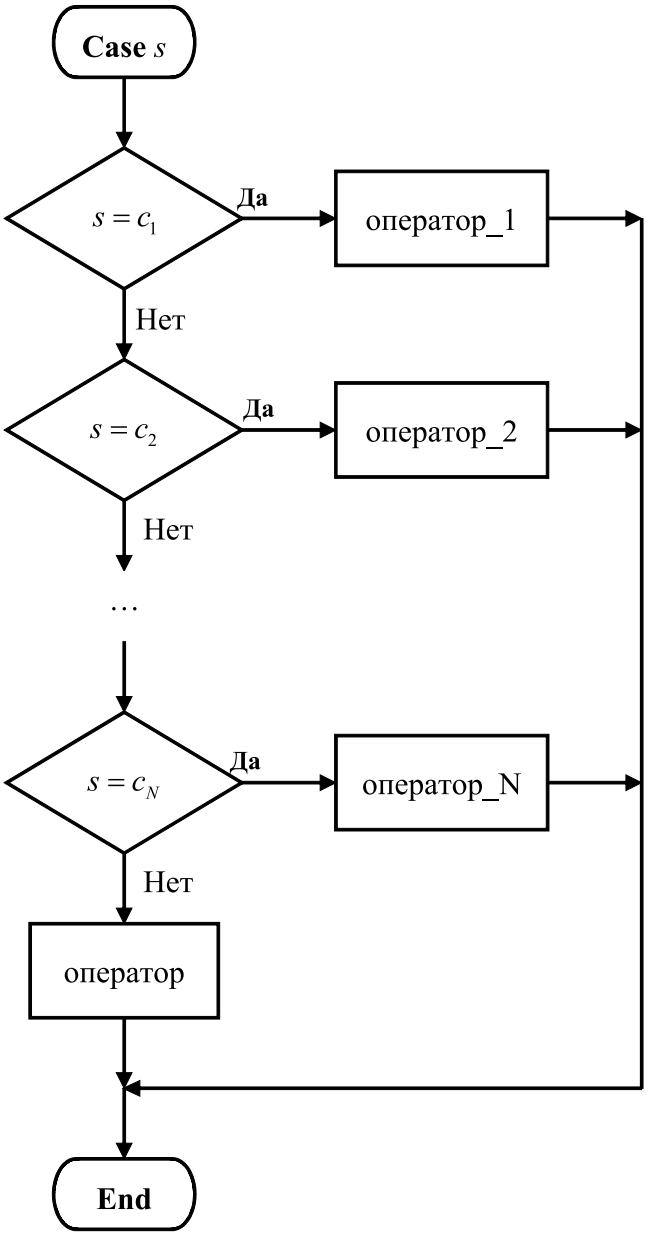
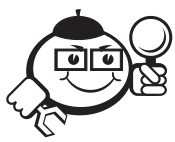


Рис. 2.10 – Блок-схема оператора выбора **Case**



Пример 2.13

```
Program example_Case; //Программа с оператором Case
Var i: Integer;
Begin //Начало тела программы
      WriteLN ('Введите целое число');
      ReadLN(i); //Ввод с клавиатуры числа
      Case i Of
              0: WriteLN ('Ноль');
       1,3,5,7,9: WriteLN ('Нечётное число');
```

```
      2,4,6,8,10: WriteLN ('Чётное число');
        11..100: WriteLN ('Число в диапазоне от 11 до 100');
         Else WriteLN ('Число отрицательное или больше 100');
      End; //Конец оператора Case
      ReadLN;
End. //Конец программы
```

В этой программе используется одна переменная `i` целого типа. В начале программы эта переменная считывается с клавиатуры при помощи оператора `ReadLN(i)`. Далее, в операторе **Case** значение этой переменной последовательно проверяется на совпадение с образцом. Если значение переменной `i` равняется 0, то выполняется оператор вывода

```
WriteLN ('Ноль')
```

и на этом работа оператора **Case** заканчивается. В противном случае значение переменной `i` последовательно сопоставляется с перечислениями `1,3,5,7,9`; `2,4,6,8,10` и диапазоном `11..100`. Если значение `i` присутствует среди перечисленных значений, то выполняется оператор, следующий в этой строке за двоеточием. Иными словами, оператор варианта **Case** приводит к выполнению оператора, которому предшествует константа выбора, равная значению переключателя `i` или диапазону выбора, в котором находится значение переключателя. Если такой константы выбора или такого диапазона выбора не существует и присутствует ветвь **Else**, то выполнются оператор, следующий за ключевым словом **Else**. Если же ветвь **Else** отсутствует, то никакой оператор не выполняется.

Ветвь **Else** является необязательной. Если она отсутствует и значение `s` не совпадает ни с одной из перечисленных констант, весь оператор **Case** рассматривается как пустой, т. е. он не выполняет никаких действий. В отличие от оператора **If** перед словом **Else** точку с запятой можно ставить.

Если для нескольких констант нужно выполнять один и тот же оператор, то их можно перечислить через запятую (или указать диапазон, если возможно), сопроводив их одним оператором, как в примере 2.13.

Оператор **Case** обязательно заканчивается словом **End**.

2.6.3 Оператор цикла While..do

Оператор цикла **While..do** организует выполнение одного оператора неизвестное заранее число раз.

Выход из цикла осуществляется, если логическое выражение, стоящее в начале цикла, окажется ложным. Так как ложность логического выражения проверяется в начале каждой итерации, тело цикла может не выполниться ни разу, поэтому этот цикл называют ещё *циклом с предусловием* (рис. 2.11).

Итерацией называется однократное выполнение *тела цикла*, т. е. операторов, входящих в цикл.

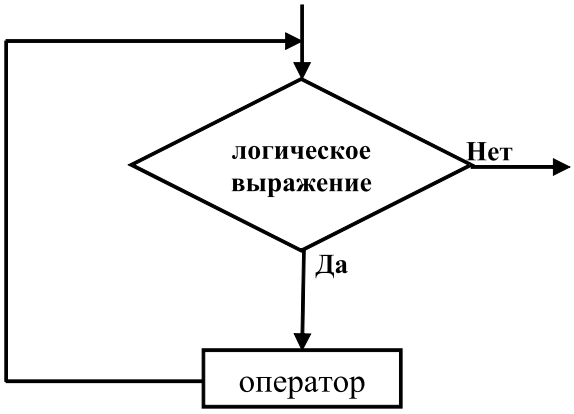


Рис. 2.11 – Блок-схема оператора цикла **While**

Обычно в операторе **While** логическое выражение — это некоторое условие, при истинности которого должны выполняться необходимые для решения задачи операторы, составляющие тело цикла и помещаемые в операторные скобки **Begin** и **End**, т. е. тело цикла зачастую является составным оператором.
Структура оператора:

```
While логическое_выражение Do оператор;
```

С английского языка запись данного оператора переводится как: пока (**While**) логическое выражение истинно (имеет значение **True**), делать (**Do**) оператор.

Пример 2.14

Написать программу, вычисляющую сумму ряда $\text{sum} = \sum_{i=1} i/(i^2 + 5i - 1)$, *пока члены ряда больше или равны 0.005. Вывести на экран полученную сумму и количество просуммированных членов ряда.*

Рассчитаем несколько первых членов ряда:

$$\text{1-й } \frac{1}{1^2 + 5 \cdot 1 - 1} = \frac{1}{5} = 0.2 \text{ при } i = 1;$$

$$\text{2-й } \frac{2}{2^2 + 5 \cdot 2 - 1} = \frac{2}{13} \approx 0.154 \text{ при } i = 2;$$

$$\text{3-й } \frac{3}{3^2 + 5 \cdot 3 - 1} = \frac{3}{23} \approx 0.130 \text{ при } i = 3;$$

$$\text{4-й } \frac{4}{4^2 + 5 \cdot 4 - 1} = \frac{4}{35} \approx 0.114 \text{ при } i = 4;$$

$$\text{5-й } \frac{5}{1^2 + 5 \cdot 5 - 1} = \frac{5}{49} \approx 0.102 \text{ при } i = 5 \text{ и т. д.}$$

Очевидно, что с увеличением i члены ряда уменьшаются.

Сумму членов ряда будем вычислять по алгоритму, аналогичному алгоритму вычисления суммы элементов матрицы (рис. 1.3).

```
//Программа вычисления суммы ряда
Program sum_of_series_1;
Const starting_i=1; //Начальное значение i
      min_term=0.005; //Минимальное значение члена ряда
Var sum, term: Real; {Сумма и член ряда,
                      переменные вещественного типа}
    i: Integer; {порядковый номер слагаемого,
                 переменная целого типа}
Begin
    sum:=0; //Обнуление переменной суммы
    i:=starting_i; //Задание начального значения i
    term:=i/(i*i+5*i-1); //Вычисление первого члена ряда
    While term>=min_term Do
    Begin
       sum:=sum+term;
       i:=i+1;
       term:=i/(i*i+5*i-1);
    End;
    WriteLN ('Сумма ряда ', sum: 0: 3);
    WriteLN ('Количество просуммированных членов ряда ',
             i-1);
    ReadLN;
End. //Конец программы
```

Рассмотрим подробнее этот пример. Описания всех используемых в примере констант и переменных сопровождаются комментариями в тексте программы. В начале программы стоят два типичных для такого рода программ оператора присваивания — обнуление переменной, в которой будет накапливаться сумма, и задание начального значения переменной цикла `i`.

```
sum:=0;
i:=starting_i;
```

Далее вычисляется первое слагаемое (первый члена ряда).

```
term:=i/(i*i+5*i-1);
```

В заголовке цикла значение текущего слагаемого `term` сравнивается с минимальным его минимально допустимым значением `min_term`. Составляющий тело цикла составной оператор

```
Begin
     sum:=sum+term;
     i:=i+1;
     term:=i/(i*i+5*i-1);
End
```

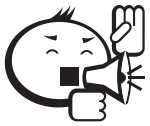
будет выполняться до тех пор, пока истинно логическое выражение в заголовке цикла (выражение `term>=min_term` принимает значение **`True`**). Первый оператор присваивания в этом составном операторе добавляет к текущему значению

суммы sum значение очередного слагаемого term. Второй оператор увеличивает на 1 значение переменной цикла i (порядковый номер слагаемого). Наконец, последний, третий оператор вычисляет значение следующего слагаемого term.

После завершения цикла (по достижению условия term>=min_term = = **False**) на экран выводятся сумма ряда с точностью до третьего знака после десятичной точки и количество членов ряда (параметр i надо уменьшить на единицу, так как последний вычисленный член ряда не суммируется).
Результатом исполнения данной программы будет:

```
Сумма ряда 3.568
Количество просуммированных членов ряда 178
```

2.6.4 Оператор цикла Repeat..until



Оператор цикла **Repeat..until** организует выполнение любого количества операторов неизвестное заранее число раз.

```
Repeat
      оператор_1;
      оператор_2;
      ...
      оператор_N;
Until логическое_выражение;
```

Запись данного оператора читается: повторять (**Repeat**) операторы до тех пор, пока (**Until**) логическое выражение не станет истинно (**True**).

Цикл **Repeat..until** похож на **While..do** тем, что также выполняется неизвестное заранее число раз, и поэтому большинство задач можно решить как при помощи одного цикла, так и при помощи другого. Цикл **Repeat..until** имеет следующие существенные отличия от цикла **While..do**:

1) В операторе **Repeat..until** проверка условия выхода выполняется в конце, а не в начале цикла.
2) Как следствие первого отличия, тело цикла **Repeat..until** всегда выполняется хотя бы один раз.
3) В операторе **Repeat..until** условие выхода удовлетворяется, если выражение истинно, а в операторе **While..do** — если ложно.
4) Между ключевыми словами **Repeat** и **Until** можно поместить любое количество операторов (без использования составного оператора), а в операторе **While..do** тело цикла должно содержать только один оператор (либо использоваться составной оператор).

Цикл **Repeat..until** называют ещё *циклом с постусловием* (рис. 2.12).

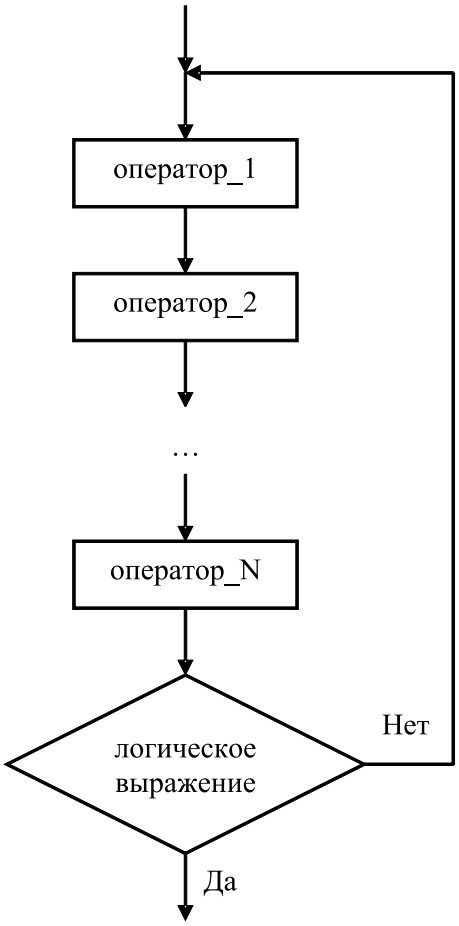


Рис. 2.12 – Блок-схема оператора цикла **Repeat**

Пример 2.15

В данном примере решим ту же задачу, что и в примере 2.14, но с использованием оператора цикла **Repeat..until**.

```
//Программа вычисления суммы ряда
Program sum_of_series_2;
Const min_term=0.005; //Минимальное значение члена ряда
Var sum, term: Real; {Сумма и член ряда,
                      переменные вещественного типа}
    i: Integer; {порядковый номер слагаемого,
                 переменная целого типа}
Begin
    sum:=0; //Обнуление переменной суммы
    i:=0; //Обнуление параметра цикла
    Repeat
       i:=i+1; //Увеличение параметра ряда на единицу
       term:=i/(i*i+5*i-1); {Вычисление очередного
                             слагаемого}
```

```
        sum:=sum+term; //Приращение суммы
    Until term<min_term; //Конец цикла
    WriteLN ('Сумма ряда ', sum: 0: 3);
    WriteLN ('Количество просуммированных членов ряда ', i);
    ReadLN;
End. //Конец программы
```

В результате исполнения данной программы на экране будет то же, что и при исполнении программы sum_of_series_1 из примера 2.14.

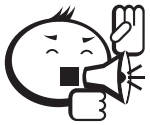
Часто бывает удобно использовать следующую конструкцию с оператором **Repeat..until**. В разделе объявления переменных необходимо описать какую-либо переменную символьным типом, например:

```
Var ch: Char;
```

Все операторы, направленные на решение какой-либо задачи, помещаются в тело цикла:

```
Begin //Начало любой программы
    Repeat //Начало цикла
      ...
      //Операторы, решающие любую задачу
      ...
      WriteLN ('Для завершения программы введите пробел');
      WriteLN ('Для продолжения программы введите любой
               символ');
      ReadLN (ch);
    Until ch=' '; //Конец цикла
End. //Конец программы
```

2.6.5 Оператор цикла For



Оператор цикла **For** организует выполнение одного оператора заранее известное число раз, в чём и заключается его основное отличие от циклов **While..do** и **Repeat..until**.

Существует два варианта оператора:

```
For i:= min To max Do оператор;
```

и

```
For i:= max DownTo min Do оператор;
```

Здесь `i` — параметр цикла, который должен быть переменной порядкового типа; `min`, `max` — константы, переменные или выражения порядкового типа, определяющие минимальное и максимальное значения параметра цикла. Цикл работает следующим образом (см. рис. 2.13). Сначала, если `min`, `max` — выражения, то вычисляются их значения. Далее параметру цикла `i` присваивается значение `min` (в первом варианте) или `max` (во втором варианте). Затем, пока `i` в зависимости от варианта либо меньше или равен `max`, либо больше или равен `min`, выполняется очередная итерация цикла; в противном случае происходит выход из цикла. Выполнение очередной итерации включает в себя сначала выполнение оператора, составляющего тело цикла, а затем присвоение параметру цикла `i` следующего большего значения (в первом варианте) или следующего меньшего значения (во втором варианте) соответственно.

Естественно, что если в первом варианте значение `min` больше `max`, а во втором варианте `max` меньше `min`, то тело цикла не выполнится ни разу.

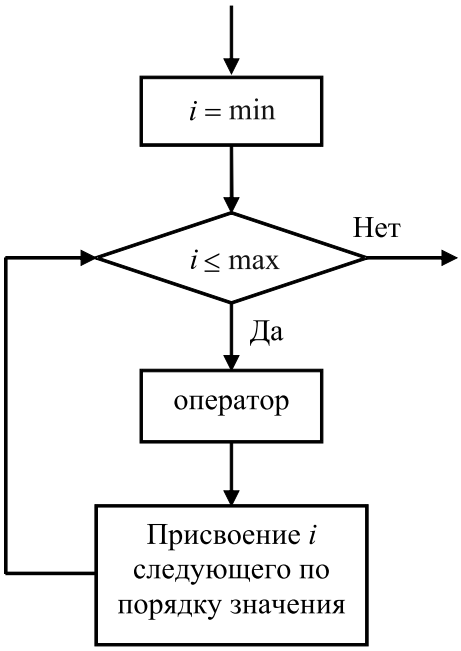
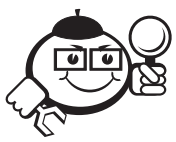


Рис. 2.13 – Блок-схема оператора цикла **`For`**

Если в цикле **`For`** необходимо выполнить не один, а несколько операторов, то используется составной оператор, начинающийся **`Begin`** и заканчивающийся **`End`**, аналогично циклу **`While..do`**.



Пример 2.16

Вывести на экран буквы от A до Z и числа от 9 до 0.

```
//Программа с примерами двух вариантов цикла
For
Program example_cycle_For;
Var roman_letter: Char; //Переменная символьного типа
```

```
        number: Integer; //Переменная целого типа
Begin
    {Первый вариант цикла, в котором выводятся в строку
       через один пробел буквы от A до Z}
    For roman_letter:='A' To 'Z' Do
       Write (roman_letter: 2);
       WriteLN; //Перевод курсора на новую строку
       {Второй вариант цикла, в котором выводятся
          в строку через один пробел числа от 9 до 0}
       For number:= 9 DownTo 0 Do Write (number: 2);
       ReadLN;
End. //Конец программы
```

Контрольные вопросы по главе 2

1. Из чего состоит алфавит языка?
2. Что такое идентификатор?
3. Как правильно записываются идентификаторы?
4. Что такое разделители?
5. Какие бывают комментарии?
6. Что относится к специальным символам?
7. Что составляет структуру программы?
8. С помощью каких процедур осуществляется ввод/вывод?
9. Что определяет тип данных?
10. Что такое перечисляемый тип данных?
11. Что такое выражение?
12. Какие бывают операции?

Глава 3

СТРУКТУРИРОВАННЫЕ ТИПЫ ДАННЫХ FREE PASCAL

Структурированные типы данных определяют последовательности однотипных или разнотипных элементов и образуются из других типов данных (простых или структурированных).

В языке Pascal существуют следующие структурированные типы: тип-массив, тип-строка, тип-запись, тип-файл, тип-множество.

В дальнейшем объекты структурированных типов для краткости будут называться теми же именами, что и их типы, без указания слова «тип»: массив, строка, запись, файл.

3.1 Массивы. Сортировки массивов

Массив *— это фиксированная последовательность упорядоченных однотипных элементов, снабжённых индексами.*

Массивы могут быть одномерными и многомерными. Мы будем рассматривать одномерные и двухмерные массивы, которые в высшей математике соответствуют векторам-столбцам (векторам-строкам) и двухмерным матрицам.

Чтобы задать массив, используется зарезервированное слово **`Array`**, после которого следует указать в квадратных скобках тип и количество индексов и далее после слова **`Of`** — тип элементов:

```
Type имя_типа = Array [тип_индексов] Of тип_элементов;
```

Размерность массива может быть любой, элементы массива могут быть любого, в том числе и структурированного, типа, индексы должны быть порядкового типа.

Описать переменные типом-массивом можно двумя способами:

1) предварительно в разделе объявления типов описать тип-массив, а затем в разделе объявления переменных этим типом описать переменную;
2) непосредственно в разделе объявления переменных описать переменную типом-массивом.

Пример 3.1

```
Type type_array = Array [1..5] Of Integer;
                  {описывается тип одномерного
                  массива с именем type_array,
                  состоящий из пяти элементов
                  типа Integer}
Var a: type_array; {Переменная a описана как
                  переменная типа type_array}
    b: Array [1..5] Of Integer;
    c: Array [1..5] Of Integer;
                  {Переменные b, c описаны как
                  одномерные массивы из пяти
                  элементов типа Integer}
    d: Array [1..3, 1..3] Of Real;
                  {Переменная d описана как
                  двухмерный массив из девяти
                  элементов типа Real}
```

В математике одномерные массивы a, b, c соответствуют, например, вектор-строкам:

$$a = (a_1, a_2, a_3, a_4, a_5),$$
$$b = (b_1, b_2, b_3, b_4, b_5),$$
$$c = (c_1, c_2, c_3, c_4, c_5),$$

а двухмерный массив d — матрице:

$$d = \begin{pmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \end{pmatrix}.$$

При создании программ рекомендуется использовать первый способ описания массивов. Доступ к элементам массива осуществляется указанием имени массива, за которым в квадратных скобках помещаются значения индексов элемента. К массивам одинакового типа во Free Pascal вполне допустимо применять операцию присваивания вида $a := b$, где a и b — массивы из примера 3.1. Все остальные операции применимы только к элементам массива.

Пример 3.2

Осуществить ввод с клавиатуры пяти элементов вещественного типа одномерного массива `vector` *и ввод случайным образом девяти элементов целого типа двухмерного массива* `matrix` *размера три строки на три столбца. Вывести данные массивы на экран.*

Ввод с клавиатуры элементов массива `vector` можно сделать при помощи операторов:

```
ReadLN (vector[1]);
ReadLN (vector[2]);
ReadLN (vector[3]);
ReadLN (vector[4]);
ReadLN (vector[5]);
```

что не рационально, так как зачастую элементов больше пяти или их число заранее неизвестно и вводится пользователем после запуска программы. Аналогичная ситуация и при выводе элементов массивов. На самом деле, при работе с массивами широко используется цикл с известным количеством повторений **`For`**, и при вводе/выводе массивов также надо применять его.

Для ввода элементов массивов случайным образом используется стандартная функция `Random`, которая формирует случайное целое число в диапазоне $0 \leqslant x \leqslant (rang - 1)$, если имеет вид:

```
x:=Random(rang),
```

где `rang` — целое число, задаваемое пользователем; или формирует случайное вещественное число в диапазоне $0.0 \leqslant x \leqslant 1.0$, если имеет вид:

```
x:=Random.
```

Обычно, совместно с функцией `Random` используется стандартная процедура без параметров `Randomize`, которая инициализирует генератор случайных чисел. Без этой процедуры база случайных чисел, основанная на показаниях системных часов компьютера, после первоначального запуска программы не будет меняться, поэтому при последующих запусках программы числа, сформированные `Random`, также не будут меняться.

На рисунках 3.1, 3.2 представлен алгоритм решения поставленной задачи. По блок-схеме пишем программу.

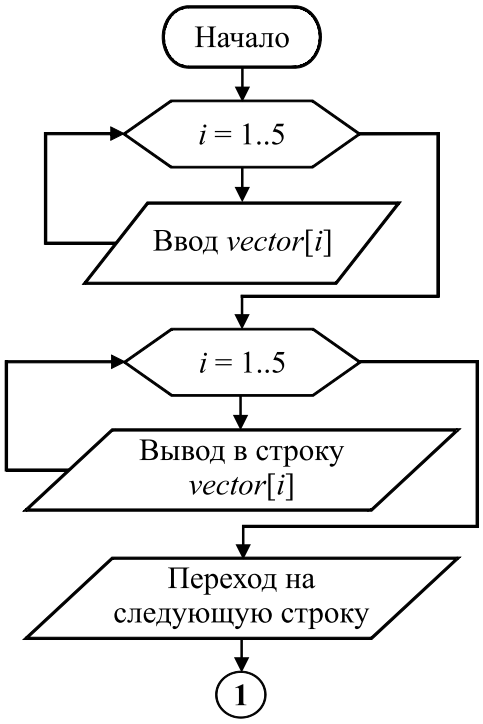


Рис. 3.1 – Блок-схема алгоритма ввода с клавиатуры
и вывода на экран одномерного массива

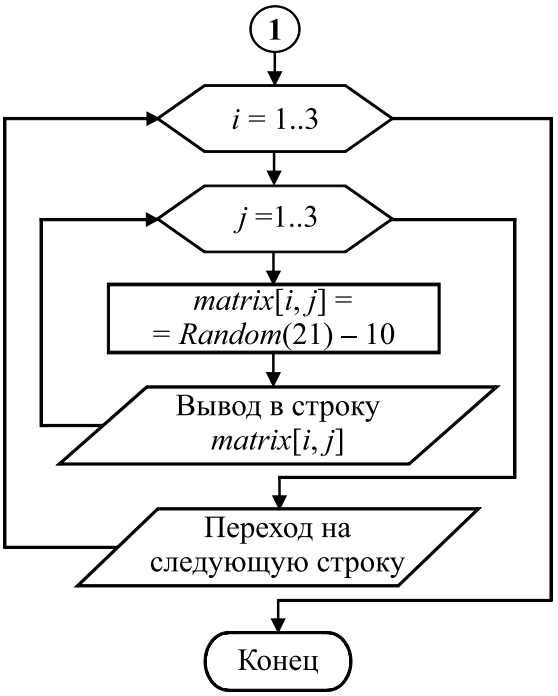


Рис. 3.2 – Блок-схема алгоритма ввода случайным образом
и вывода на экран двухмерного массива

```
//Программа ввода и вывода массивов
Program input_output_array;
```

```
Type
     type_vector = Array [1..5] Of Real;
         {Описание типа одномерного массива
         с именем type_vector,состоящего из пяти
         элементов типа Real}
     type_matrix =  Array [1..3, 1..3] Of Integer;
         {Описание типа двухмерного массива
         с именем type_matrix,состоящего из девяти
         элементов типа Integer}
Var vector: type_vector;
         //Массив vector типа type_vector
     matrix: type_matrix;
         //Массив matrix типа type_matrix
     i, j: Integer;
Begin
     WriteLN ('Введите элементы массива vector');
     For i:=1 To 5 Do ReadLN (vector[i]);
     WriteLN ('Массив vector');
     For i:=1 To 5 Do Write (vector[i]: 0: 3, ' ');
     WriteLN; {Перевод курсора на следующую строку}
     WriteLN ('Матрица matrix');
     Randomize;
       //Инициализация генератора случайных чисел
     For i:=1 To 3 Do
     Begin
       For j:=1 To 3 Do
       Begin
         matrix[i,j]:=Random(21)-10;
         Write (matrix[i,j]: 4);
       End; //Конец цикла по j
       WriteLN; //Перевод курсора на следующую строку
     End; //Конец цикла по i
     ReadLN;
End. //Конец программы
```

В первом цикле **For** в этом примере происходит поэлементный ввод массива vector, при этом параметр цикла `i` меняется от 1 до 5 с шагом 1. Во втором цикле происходит вывод в строку через один пробел элементов массива vector с точностью до третьего знака после десятичной точки. Далее в програме представлен пример двух вложенных циклов — телом внешнего цикла является внутренний цикл. Во внешнем цикле параметр `i` пробегает значения от 1 до 3 с шагом 1, во внутреннем цикле соответственно параметр `j` пробегает значения от 1 до 3. В теле внутреннего цикла задаются элементы матрицы случайными целыми числами в диапазоне от −10 до 10 и вывод элементов матрицы в строку минимум через один пробел.

Сортировкой массива *называется перераспределение элементов массива в порядке возрастания или убывания значений элементов.*

Рассмотрим три простейших метода сортировок массива, на которых основаны более сложные способы, использующиеся в специализированных программах обработки больших объёмов данных.

Метод простого выбора заключается в выборе наименьшего (в случае сортировки по возрастанию) или наибольшего (в случае сортировки по убыванию) значения элемента массива; затем этот элемент меняется местами с первым элементом. Далее выбирается наименьший (наибольший) элемент из оставшейся неотсортированной части массива и меняется местами со вторым элементом и т. д.

В таблице 3.1 приведён пример сортировки по возрастанию значений элементов массива $m = (5\ \ 3\ \ 2\ \ 1\ \ 4)$ методом простого выбора. Фигурной границей показаны элементы, среди которых происходит поиск наименьшего значения. Полужирным шрифтом выделены минимальные значения на каждом прогоне. Стрелками показаны элементы, значения которых меняются местами. Прогоном будем называть цикл, в котором происходит поиск минимального значения в неотсортированной части массива.

Таблица 3.1 – Пример сортировки массива по возрастанию методом простого выбора

Номер прогона	Элементы массива m				
1	5	3	2	**1**	4
2	1	3	**2**	5	4
3	1	2	**3**	5	4
4	1	2	3	5	**4**
Результат	1	2	3	4	5

Заметим, что количество прогонов будет на единицу меньше количества элементов массива. Поэтому если количество элементов массива m будет n, то количество прогонов составит $n - 1$.

Алгоритм метода простого выбора представлен на рисунке 3.3. Первый цикл по i отвечает за прогоны; вложенный в него второй цикл по j осуществляет перебор элементов массива для поиска минимального значения, которое запоминается в переменной *extr*, а индекс этого элемента — в переменной k. Параметры циклов i и j связаны таким образом, чтобы на каждом следующем прогоне исключать из поиска минимального значения отсортированный элемент. После выхода из цикла по j происходит переприсвоение значения элемента, занимающей первое место в неотсортированной части массива, и минимального значения элемента.

Если в условии $extr > m_j$ поменять знак на <, то получим сортировку по убыванию.

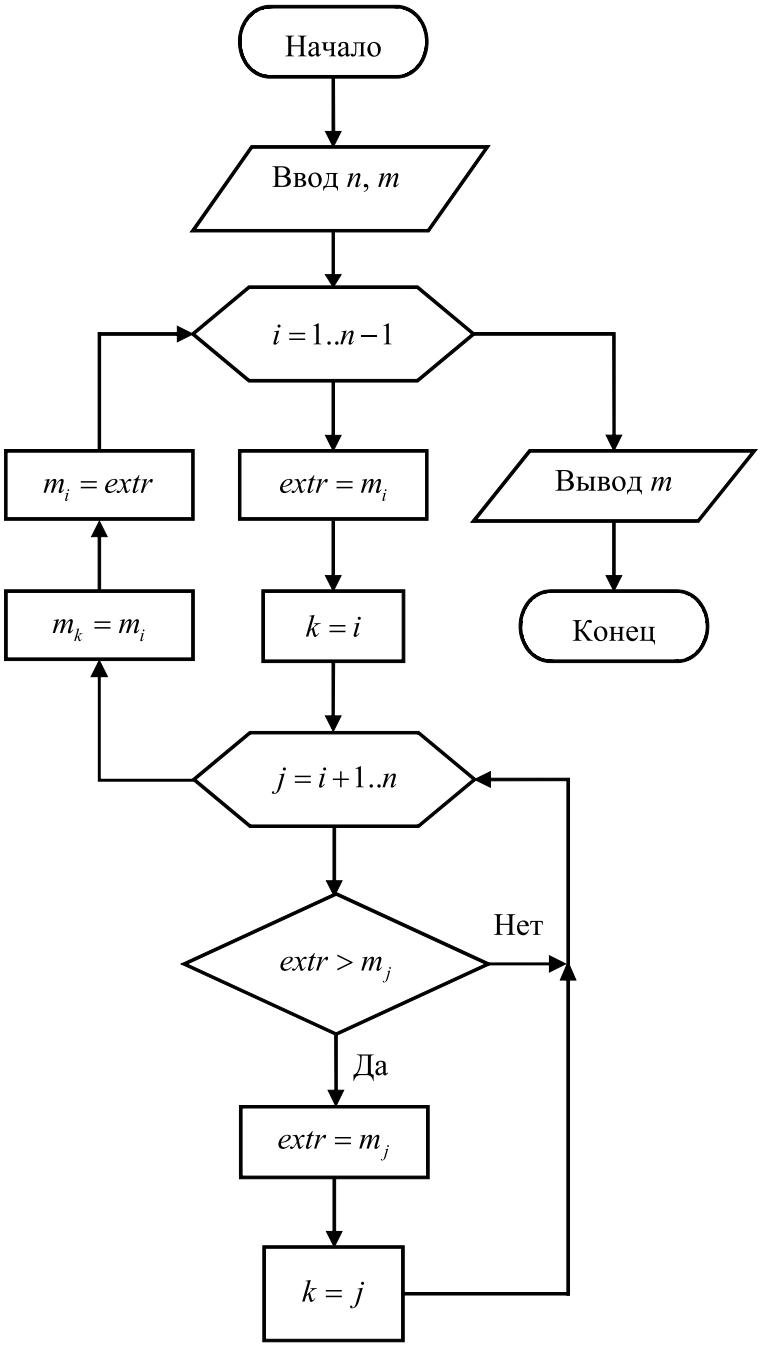


Рис. 3.3 – Блок-схема метода простого выбора

По блок-схеме на рисунке 3.3, осуществляя ввод массива *m* аналогично алгоритму на рисунке 3.2 и вывод, как на рисунке 3.1, записываем следующую программу.

```
//Программа сортировки массива методом простого выбора
Program simple_selection_sort_of_array;
Const
     n=100; //размер массива
Type
     type_vector = Array [1..n] Of Integer;
```

```
                {Описывается одномерный тип-массив
                с именем type_vector, состоящий из n
                элементов типа Integer, соответственно
                индекс элементов массива изменяется
                в диапазоне от 1 до n}
Var m: type_vector; //Массив m типа type_vector
    i, j, k, //Индексы элементов массива
    extr: Integer; {Переменная для хранения наименьшего
                    значения элементов массива}
Begin
    Randomize;
                //Инициализация генератора случайных чисел
    For i:=1 To n Do
    Begin
                m[i]:=Random(21)-10;
                Write (m[i]: 4);
    End; {Конец цикла по i}
    WriteLN; /Перевод курсора на следующую строку
    For i:=1 To n-1 Do
    Begin
                extr:=m[i];
                    //Задание начального наименьшего
                    значения
                k:=i;
                    {Задание индекса элемента
                    с наименьшим значением}
                For j:=i+1 To n Do
                    If extr>m[j] Then
                    Begin
                      extr:=m[j];
                      k:=j;
                    End; {Конец условного оператора
                          и цикла по j}
                m[k]:=m[i];
                m[i]:=extr;
    End; {Конец цикла по i}
    WriteLN ('Отсортированный массив m');
                For i:=1 To n Do Write (m[i]: 4);
                ReadLN;
End. //Конец программы
```

В представленной программе в первом цикле (по i) происходит ввод и вывод массива m. Элементы массива задаются случайными целыми числами в диапазоне от –10 до 10. Второй цикл (по i) собственно и осуществляет сортировку. На каждом шаге этого цикла находится минимальный элемент массива в диапазоне от i-го элемента до n-го и его значение обменивается со значением в i-м элементе массива.

Этот поиск осуществляется в цикле по `j` перебором элементов массива. Если значение `extr` больше значения текущего элемента массива, то номер этого элемента запоминается в переменной `k`, а значение этого элемента запоминается в переменной `extr`. После выхода из этого внутреннего цикла происходит обмен значениями между `i`-м и `k`-м элементами массива.

Недостатком метода простого выбора является то, что в случае если значение элемента уже находится на своём месте (см. табл. 3.1 3-й прогон), происходит присвоение этого значения тому же элементу, т. е. выполняются лишние действия, что приводит к увеличению времени сортировки. Данный недостаток устранён в методе «пузырька».

Метод простого обмена («пузырька») заключается в последовательном сравнении соседних элементов массива и переприсвоении значений этих элементов в зависимости от вида сортировки (по возрастанию или по убыванию). Таким образом, за один прогон на своё место встаёт одно значение, «всплывая», как пузырёк газа в жидкости, откуда произошло второе название метода (см. табл. 3.2 — пример сортировки по возрастанию).

Таблица 3.2 – Пример сортировки массива по возрастанию методом простого обмена

Номер прогона	Элементы массива *m*				
1	**5**	**2**	3	1	4
	2	**5**	**3**	1	4
	2	3	**5**	**1**	4
	2	3	1	**5**	**4**
2	**2**	**3**	1	4	5
	2	**3**	**1**	4	5
	2	1	**3**	**4**	5
3	**2**	**1**	3	4	5
	1	**2**	**3**	4	5
4	**1**	**2**	3	4	5
Результат	1	2	3	4	5

В таблице 3.2 фигурной границей показаны элементы, которые отсортированы в предыдущем прогоне и не участвуют в текущем прогоне. Прогоном будем называть цикл, в котором происходит перемещение значения на своё место в неотсортированной части массива. Полужирным шрифтом выделены сравниваемые значения. Стрелками показаны элементы, значения которых меняются местами.

Алгоритм метода «пузырька» представлен на рисунке 3.4. Аналогично методу простого выбора — первый цикл по i отвечает за прогоны, количество которых будет на единицу меньше количества элементов массива; вложенный в него второй цикл по j осуществляет перебор элементов массива, и если условие $m_j > m_{j+1}$ истинно, то значения элементов меняются местами (если в этом условии поменять знак на <, то получим сортировку по убыванию); параметры циклов i и j связаны таким образом, чтобы на каждом следующем прогоне исключать отсортированный элемент.

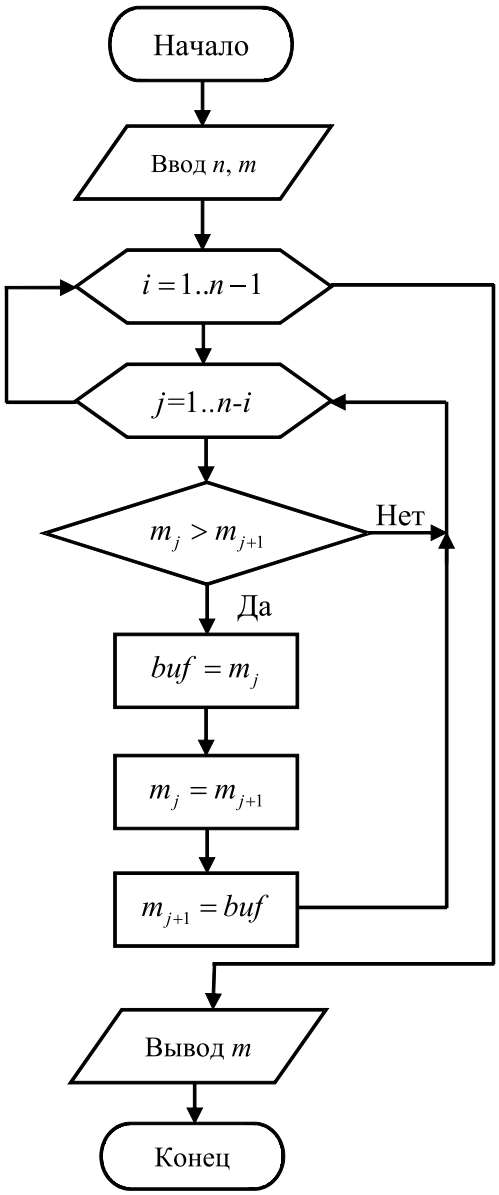


Рис. 3.4 – Блок-схема метода простого обмена

По блок-схеме на рисунке 3.4 записываем следующую программу.

```
//Программа сортировки массива методом пузырька
Program bubble_sort_of_array;
Const
     n=100; //размер массива
Type
     type_vector = Array [1..n] Of Integer;
Var m: type_vector;
     i, j,
     buf: Integer;
     //Буферная переменная
Begin
     WriteLN ('Массив m');
```

```
  Randomize;
  For i:=1 To n Do Begin
    m[i]:=Random(21)-10;
    Write (m[i]: 4);
  End;
  WriteLN;
  For i:=1 To n-1 Do
    For j:=1 To n-i Do
    If m[j]>m[j+1] Then
    Begin
      buf:=m[j];
      m[j]:=m[j+1];
      m[j+1]:=buf;
    End;
  WriteLN ('Отсортированный массив m');
    For i:=1 To n Do
      Write (m[i]: 4);
    ReadLN;
End. //Конец программы
```

Хотя в методе «пузырька» нет лишних переприсвоений, однако на каждом прогоне сравниваются все пары элементов, что также увеличивает время сортировки. Это устраняется в *методе простых вставок*, в котором значение элемента из неотсортированной части массива, меняясь местами со значениями из отсортированной части, встаёт на своё место.

В таблице 3.3 представлена сортировка по возрастанию следующего массива $m = (5 \quad 2 \quad 3 \quad 1 \quad 4)$ методом простых вставок. Полужирным шрифтом выделены сравниваемые значения. Стрелками показаны элементы, значения которых меняются местами. Фигурной границей показаны элементы, значения которых не сравниваются в текущем прогоне, что сокращает время сортировки. Прогоном будем называть цикл, в котором значение из неотсортированной части занимает своё место в отсортированной части массива.

Таблица 3.3 – Пример сортировки массива по возрастанию методом простых вставок

Номер прогона	Элементы массива m				
1	**5**	**2**	3	1	4
2	2	**5**	**3**	1	4
	2	**3**	5	1	4
3	2	3	**5**	**1**	4
	2	**3**	**1**	5	4
	2	**1**	3	5	4
4	1	2	3	**5**	**4**
	1	2	**3**	**4**	5
Результат	1	2	3	4	5

По блок-схеме на рисунке 3.5 записываем программу, в которой присутствуют, как и в предыдущих методах, цикл по i с количеством итераций (прогонов) на единицу меньше количества элементов массива и вложенный в него второй цикл с неизвестным заранее количеством итераций. При истинности логического выражения ($m_j > m_{j+1}$ и $j \geqslant 1$) во втором цикле осуществляется перебор элементов массива и значения элементов меняются местами (если в условии $m_j > m_{j+1}$ поменять знак на <, то получим сортировку по убыванию).

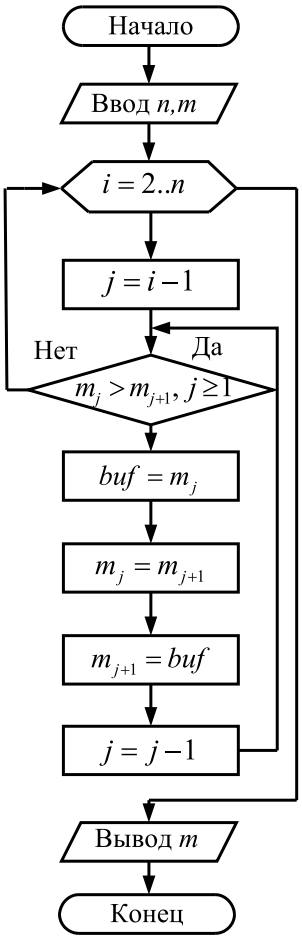


Рис. 3.5 – Блок-схема метода простых вставок

```
{Программа сортировки массива методом простых вставок}
Program simple_insertion_sort_of_array;
Const
     n=100; //размер массива
Type
     type_vector =  Array [1..n] Of Integer;
Var
     m: type_vector;
     i, j, //Индексы элементов
     buf: Integer;
     //Буферная переменная
Begin
```

```
      WriteLN ('Массив m');
      Randomize;
      For i:=1 To n Do
Begin
        m[i]:=Random(21)-10;
        Write (m[i]: 4);
      End; //Конец цикла по i
      WriteLN;
      For i:=2 To n Do
      Begin
        j:=i-1;
        While (m[j]>m[j+1])
          And (j>=1) Do
        Begin
          buf:=m[j];
          m[j]:=m[j+1];
          m[j+1]:=buf;
          j:=j-1;
        End; //Конец цикла While
      End; //Конец цикла For
      WriteLN ('Отсортированный массив m');
      For i:=1 To n Do
        Write (m[i]: 4);
      ReadLN;
End. //Конец программы
```

3.2 Строки типа `String`

Строка стандартного типа String — это последовательность символов длиной до 255 элементов. Строки можно рассматривать как массивы символов. Доступ к отдельному символу строки осуществляется так же как к элементу одномерного массива, т. е. указывается имя переменной, а затем в квадратных скобках порядковый номер символа в строке.

При объявлении переменной строкового типа после слова `String` в квадратных скобках можно указывать её размер (от 1 до 255), если размер не указан, то он по умолчанию принимается равным 255 символам.

Для строк применимы операции конкатенации (сложение — знак «+») и сравнения. При сложении строк следует помнить, что в общем случае $s1 + s2 \neq s2 + s1$, где $s1$ и $s2$ — строки. При сравнении двух строк последовательно сравниваются коды входящих в строки символов, начиная с первого, и большей считается та, код символа которой является большим в кодовой таблице символов. Равными считаются строки, все символы и длина которых одинаковы.

Пример 3.3

```
//Программа с операциями над строками
Program operations_with_String;
Var
     s1: String [5]; //Строка из 5-ти символов
     s2: String [6]; //Строка из 6-ти символов
     s_max: String; {Максимальная строка из 255-ти
                     символов}
Begin
     s1:='Free';
     s2:='Pascal';
     If s1>s2 Then
     Begin
       Write ('Символ ', s1[1], 'с кодом ', Ord(s1[1]));
       Write ('больше символа ', s2[1]);
       WriteLN ('с кодом ', Ord(s2[1]));
         {Переменной s_max присваивается строка,
         которая складывается из строки s1,
         пробела и строки s2}
       s_max:=s1+' '+s2;
     End
     Else
     Begin
       Write ('Символ ', s1[1], 'с кодом ', Ord(s1[1]));
       Write ('меньше или равен символу ', s2[1]);
       WriteLN ('с кодом ', Ord(s2[1]));
       s_max:=s2+' '+s1;
     End;
     WriteLN (s_max);
     ReadLN;
End. //Конец программы
```

Рассмотрим стандартные подпрограммы обработки строк. В фигурных скобках указывается тип задаваемых и возвращаемых параметров подпрограмм.

Функции работы со строками

1. `Concat (s1, s2,..., sn{: String})){: String}` — возвращает строку, которая является результатом сложения строк `s1`, `s2`,..., `sn`, и при необходимости усекает чрезмерно большую полученную строку до 255 символов. Например, оператор

```
s_max:=s1+' '+s2;
```

будет эквивалентен оператору

```
s_max:=Concat(s1,' ',s2);
```

2. `Copy (s{: String}, p, n{: Integer}){: String}` — возвращает строку длиной `n` символов, начиная с позиции `p`, выделенную из строки `s`. Например,

```
full_name:='Сидоров Иван Петрович';
name:=Copy(full_name,9,4);
```

в результате переменная

```
name='Иван'
```

3. `Length (s{: String}){: Integer}` — возвращает текущий размер строки `s`. Например,

Пример 3.4

```
s1:='Free';
s2:='Pascal';
s_max:=s1+' '+s2;
size_str:=Length(s_max);
```

в результате переменная

```
size_str=12
```

4. `Pos (s1, s2{: String}){: Byte}` — возвращает номер первого символа подстроки `s1`, входящей в строку `s2`. Если подстроки `s1` в строке `s2` нет, то возвращается ноль. Например,

Пример 3.5

```
full_name:='Сидоров Иван Петрович';
name:='Иван';
patronymic:='Викторович';
p_name:=Pos(name,full_name);
p_patr:=Pos(patronymic,full_name);
```

в результате переменные

```
p_name=9
p_patr=0
```

Процедуры работы со строками

1. `Delete ({Var}s{: String}, p, n{: Integer})` — удаляет `n` символов из строки `s`, начиная с позиции `p`. Например,

```
full_name:='Сидоров Иван Петрович';
```

```
Delete (full_name, 13, 9);
```

в результате переменная

```
full_name='Сидоров Иван'
```

2. `Insert (s1, {`**`Var`**`}s2{: `*`String`*`}, p{: `*`Integer`*`})` — вставляет строку `s1` в строку `s2`, начиная с позиции `p`. Например,

Пример 3.6

```
full_name:='Сидоров Петрович';
name:='Иван';
Insert (name+' ', full_name, 9);
```

в результате переменная

```
full_name='Сидоров Иван Петрович'
```

3. `Str (n{: Числовой тип}, {`**`Var`**`}s{: `*`String`*`})` — преобразует вещественное или целое число `n` в строку `s`. Для числа можно задать формат. Например,

Пример 3.7

```
n:=10;
Str (n, s1);
Str (Pi: 4: 2, s2);
```

в результате переменные

```
s1='10'
s2='3.14'
```

Рассмотрим в этом же параграфе две полезные стандартные функции преобразования типов:

1. `Chr (c{: Byte}){: `*`Char`*`}` — возвращает символ с кодом `c`. Например,

Пример 3.8

```
code:=65;
symbol:=Chr(code);
```

в результате переменная

```
symbol='A'
```

2. `Ord (v{: Порядковый тип}){: LongInt}` — возвращает порядковый номер переменной `v`. См. пример 3.3, в котором выводится код первых символов строк `s1` и `s2`.

Фактически строка из n символов представляет собой массив из $n + 1$ символа. Нулевой символ предназначен для указания используемого количества символов строки и может изменяться от символа с кодом 0 до символа с кодом n в кодовой таблице. С ним можно работать как и с остальными символами строки (записывать, читать его значение и др.), но не забывать о его основном предназначении.

3.3 Записи. Оператор над записями `With`

Запись представляет собой фиксированную последовательность элементов различного типа. Элементы записи называются *полями*, количество которых может быть любым.

Тип-запись не является стандартным типом и объявляется как

```
Type имя_типа = Record
       поле_1: тип_1;
       поле_2: тип_2;
       ...
       поле_n: тип_n;
     End;
```

После имени типа-записи ставится знак «=» и зарезервированное слово **Record**, затем перечисляются все поля с указанием через двоеточие их типов (если в записи несколько полей одного типа, то их можно объединить в одну группу, перечислив через запятые, и указать общий тип). Тип-запись завершается словом **End**. Объявления полей отделяются друг от друга точкой с запятой.

Пример 3.9

Объявление типа-записи библиотечных книг и переменной этого типа.

```
Type
     record_book = Record
       name, //Фамилия автора
       initials: String [30]; //Инициалы автора
       title, //Название книги
       publ_house: String; //Издательство
       publ_date, //Год издания
       pages: Integer; //Количество страниц
       artwork: Boolean; {Переменная логического типа.
                          Если иллюстрации в книге есть,
                          то переменная — True,
                          иначе — False}
```

```
        End; //Конец записи
    Var //Раздел объявления переменных
        book: record_book; {Переменная book типа record_book}
```

В программе доступ к полям записи осуществляется указанием имени переменной и имени поля, записываемого через точку.

Пример 3.10

Использование полей записи в программе.

```
book.name:='Вирт';
book.initials:='H.';
book.title:='АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ';
book.publ_house:='M.: Мир';
book.publ_date:=1989;
book.pages:=360;
book.artwork:= True;
```

Для того чтобы не выписывать каждый раз имя записи при обращении к её полям, можно использовать оператор над записями **With**, структура которого выглядит следующим образом:

With переменная_запись **Do** оператор_с_полями;

Пример 3.11

Использование оператора **With** для выполнения действий, аналогичных примеру 3.5.

```
    //С переменной book выполнять составной оператор
    With book Do
    Begin
           name:='Вирт';
           initials:='H.';
           title:='АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ';
           publ_house:='M.: Мир';
           publ_date:=1989;
           pages:=360;
artwork:= True;
    End;
```

3.4 Файлы

Тип-файл — это последовательность элементов одного типа, расположенных на дисках. Число элементов в файле не объявляется. В общем виде объявление типа-файла выглядит как

```
Type имя_типа = File Of тип_элементов;
```

В Free Pascal поддерживается три разновидности файлов: типизированные, нетипизированные и текстовые файлы. При определении нетипизированного файла не указывается тип его элементов. Нетипизированный файл рассматривается как совокупность байтов и используется при скоростной обработке файлов. Если в типизированных и текстовых файлах считывание и запись информации идёт поэлементно и каждый раз обращение к файлу идёт сравнительно медленно, то в нетипизированных файлах возможны считывание и запись блоками. При этом реальные элементы файла могут быть различного типа.

Пример 3.12

```
Type
     file_int = File Of Integer;
       //Тип целочисленного файла
     file_roman = File Of 'A'..'Z';
       {Тип файла, содержащего диапазон
       латинских заглавных букв}
Var
     f1: file_int; //Целочисленный файл
     f2: file_roman; //Файл типа file_roman
     f3: Text; //Текстовый файл
     f4: File; //Нетипизированный файл
```

Файловые переменные имеют специфическое применение. Над ними нельзя выполнять никаких операций (присваивать значение, сравнивать и т. д.). Их можно использовать только для выполнения операций с файлами (чтения, записи, удаления из файла и т. п.). Кроме того, через файловую переменную можно получить информацию о конкретном файле (тип, параметры, имя файла и т. д.).

Рассмотрим основные подпрограммы обработки файлов. Параметр `f` во всех подпрограммах является переменной файлового типа.

Процедуры:

- `Assign (f, s{: String})` — связывает файловую переменную `f` и файл, имя которого указывается в строке. Если в строке `s` не указывается полное имя файла, т. е. отсутствует путь к файлу, тогда он должен находиться в текущей папке. После использования процедуры `Assign` обращение к файлу осуществляется через файловую переменную, а не по имени файла.

- `ReWrite (f)` — создаёт и открывает новый файл для записи в него данных.
- `Reset (f)` — открывает существующий файл, т. е. файл к моменту действия этой процедуры уже должен быть создан на диске.
- `Append (f)` — открывает существующий текстовый файл для добавления в него данных.
- `Read (f, x1, x2,..., xn {группа читаемых параметров})` — читает информацию из типизированного или текстового файла. Для текстовых файлов также можно использовать процедуру *`ReadLN`* с теми же параметрами. Чтение информации из файла без типа выполняется процедурой `BlockRead [3]`.
- `Write (f, x1, x2,..., xn {группа записываемых параметров})` — записывает информацию в типизированный или текстовый файл. Для текстовых файлов также можно использовать процедуру *`WriteLN`* с теми же параметрами. Запись информации в файл без типа выполняется процедурой `BlockWrite [3]`.
- `Close (f)` — закрывает ранее открытый файл.

Функция логического типа `EOF(f){: Boolean}` возвращает истину (**`True`**), если указатель позиции в файле стоит за последним элементом, т. е. фиксирует конец файла, и ложь (**`False`**) в противном случае.

Пример 3.13

1. *Написать программу создания файла записей, содержащего информацию о книгах (фамилия, инициалы автора, название книги, издательство, год издания, количество страниц, наличие иллюстраций).*

2. *Написать программу создания текстового файла, содержащего прочитанную из существующего файла записей информацию об иллюстрированных книгах, отсортированных в порядке возрастания года издания.*

В программах будем использовать запись, рассмотренную в примере 3.9.

```
//Программа создания файла записей
Program creation_file_of_records;
Type
    record_book = Record
      name, //Фамилия автора
      initials: String [30]; //Инициалы автора
      title, //Название книги
      publ_house: String; //Издательство
      publ_date, //Год издания
      pages: Integer; //Количество страниц
      artwork: Boolean; {Переменная логического типа.
                         Если иллюстрации в книге есть,
                         то переменная — True,
                         иначе — False}
```

```
    End; //Конец записи
Var
    file_name: String [75];
    file_record: File Of record_book;
                 {Файловая переменная типа
                 record_book для файла записей}
    i: Integer;
    book: record_book; {Переменная book типа record_book}
    selection: Char; //Переменная символьного типа
Begin
    WriteLN ('Введите имя файла записей
            (без расширения)');
    ReadLN (file_name); {Считывание с клавиатуры
                        строки file_name}
    Assign (file_record, file_name+'.dat');
    ReWrite (file_record); //Создание и открытие файла
    i:=0; //Обнуление счётчика записей
    Repeat
      i:=i+1;
      WriteLN ('Введите ', i, '-ю запись книги:');
      With book Do
      Begin
        Write ('Фамилия автора - ');
        ReadLN (name);
        Write ('Инициалы автора - ');
        ReadLN (initials);
        Write ('Название книги - ');
        ReadLN (title);
        Write ('Издательство - ');
        ReadLN (publ_house);
        Write ('Год издания - ');
        ReadLN (publ_date);
        Write ('Количество страниц - ');
        ReadLN (pages);
        Write ('Если иллюстрации есть,', 'введите "1",
              иначе - "0"');
        Repeat
            ReadLN (selection);
            {До тех пор, пока значение selection
            не станет равно символу "1"или "0".}
        Until (selection='1') Or (selection='0');
        artwork := selection;
      End;
      Write (file_record, book);
      WriteLN ('Для продолжения введите',
              'любой символ');
```

```
      WriteLN ('Для выхода из программы ',
               'введите "0"');
      ReadLN (selection);
    Until selection='0';
    Close (file_record);
End. //Конец программы
```

В записанной выше программе первый цикл повторяет считывание информации с клавиатуры и запись её в файл, пока пользователем не будет введён символ ноль после соответствующего сообщения. Второй цикл предназначен для того, чтобы при определении — имеются или нет иллюстрации в книге, избежать ошибочного ввода любых символов, кроме «1» и «0».

```
//Программа создания текстового файла
Program creation_text_file;
Type
    record_book = Record
      name, //Фамилия автора
      initials: String [30]; //Инициалы автора
      title, //Название книги
      publ_house: String; //Издательство
      publ_date, //Год издания
      pages: Integer; //Количество страниц
      artwork: Boolean; {Переменная логического типа.
                         Если иллюстрации в книге есть,
                         то переменная — True,
                         иначе — False}
    End; //Конец записи
    array_books = Array [1..50] Of record_book;
Var
      {Имя файла, максимальный размер которого,
      включая имя устройства и путь, будет 79
      символов минус 4 символа под расширение}
    file_name: String [75];
    file_record: File Of record_book;
    i, j, //Индексы массива записей
    n: Integer; //Количество записей
    books: array_books; {Массив books типа array_books}
    buf: record_book; {Буферная переменная типа record_book}
    text_file: Text;
Begin
    WriteLN ('Введите имя файла записей ',
             '(без расширения)');
    ReadLN (file_name);
    Assign (file_record, file_name+'.dat');
    Reset (file_record);
```

```
        i:=1; //Задание начального индекса массива
        While Not EOF(file_record) Do
        Begin
          Read (file_record, books[i]);
          If books[i].artwork Then
            i:=i+1;
        End;
        Close (file_record);
        n:=i-1;
        {Сортировка массива books методом
        простых вставок}
        For i:=2 To n Do
        Begin
          j:=i-1;
          While (books[j].publ_date
          > books[j+1].publ_date) And (j>=1) Do
          Begin
            buf:=books[j];
            books[j]:=books[j+1];
            books[j+1]:=buf;
            j:=j-1;
          End;
        End;
        WriteLN ('Введите имя текстового ',
                 'файла (без расширения)');
        ReadLN (file_name);
        Assign (text_file, file_name+'.txt');
        ReWrite (text_file);
        WriteLN (text_file, 'ИЛЛЮСТРИРОВАННЫЕ КНИГИ');
        For i:=1 To n Do
          With books[i] Do
          Begin
          WriteLN (text_file, i, '-я книга');
          WriteLN (text_file, name, ' ', initials);
          WriteLN (text_file, title);
          WriteLN (text_file, publ_house);
          WriteLN (text_file, 'Год издания ', publ_date);
          WriteLN (text_file, 'Количество страниц ', pages);
        End;
        Close (text_file);
End.
```

В приведённой программе три основных блока.

В первом считывание информации из файла записей организовано таким образом, что запись книги сначала считывается в текущий элемент массива, а затем при условии наличия иллюстраций индекс элементов массива увеличивается на единицу, иначе индекс не изменяется. Поэтому элемент массива с записью книги

без иллюстраций перезаписывается при следующем считывании информации из файла. Однако в последнем элементе массива может оказаться как запись книги с иллюстрациями, так и без иллюстраций. В первом случае количество необходимых записей `n` меньше `i` на единицу потому, что условие в цикле выполняется и происходит лишнее приращение `i`. Во втором случае `n` должно быть меньше `i` на единицу, так как элемент массива с последней записью книги без иллюстраций не нужен по условию задачи.

Во втором блоке происходит сортировка методом простых вставок массива записей книг с иллюстрациями по возрастанию значений поля `publ_date` (года издания). Метод простых вставок подробно рассматривается в п. 3.1.

В третьем блоке записи книг отсортированного массива записываются в текстовый файл, имя которого вводится пользователем. Содержимое получаемого таким образом файла можно просмотреть в любом текстовом редакторе.

Контрольные вопросы по главе 3

1. Что такое массив?
2. Какие бывают массивы?
3. Что такое сортировка массивов?
4. Какие виды сортировок Вам известны?
5. Что такое строка?
6. Какие процедуры и функции работы со сторками Вам известны?
7. Что такое запись?
8. Как осуществляется доступ к полю записи?
9. Что такое файл?
10. Как осуществляется доступ к элементам файла?

Глава 4

ПОДПРОГРАММЫ. БИБЛИОТЕКИ ПОДПРОГРАММ FREE PASCAL

4.1 Подпрограммы

В языке Pascal имеются две разновидности подпрограмм — процедуры и функции.

Структура любой подпрограммы аналогична структуре всей программы. Подпрограмма должна быть описана до того, как она будет использована в основной программе или другой подпрограмме. Любая подпрограмма начинается с заголовка. За заголовком могут идти такие же разделы, что и в основной программе. В отличие от основной программы заголовок в подпрограмме обязателен, и завершается она не точкой, а точкой с запятой.

Все параметры, которые использует подпрограмма, делятся на две категории: *локальные параметры*, объявляемые внутри подпрограммы и доступные только ей самой, и *глобальные параметры*, объявляемые в основной программе и доступные как программе, так и всем её подпрограммам.

Процедура предназначена для выполнения какой-либо законченной последовательности действий. Заголовок процедуры начинается с зарезервированного слова **Procedure**, за которым следует идентификатор имени процедуры, а далее в круглых скобках — список формальных параметров:

```
Procedure имя_процедуры (список_формальных_параметров);
```

Подпрограмма-функция предназначена для вычисления какого-либо параметра.

У функции два основных отличия от процедуры.

Первое отличие в заголовке. В функции он состоит из слова **Function**, за которым следует имя функции, далее в круглых скобках — список формальных параметров, затем через двоеточие записывается тип функции — тип возвращаемого параметра. Функция может возвращать параметры следующих типов: любого порядкового, любого вещественного, стандартного типа *String*, любого указателя.

```
Function имя_функции(список_формальных_параметров):
                                              тип_функции;
```

Второе отличие заключается в том, что в теле функции её имени хотя бы раз должно быть присвоено некоторое значение.

Пример 4.1

Рассмотрим следующую задачу.

Пусть даны координаты вершин плоского n-угольника. Требуется найти координаты всех середин его сторон. Разумеется, данную задачу можно решить и без использования подпрограмм. Для определенности будем считать, что n = 4, а координаты всех вершин для простоты зададим при помощи операторов присваивания.

```
Program without_subprogram; //Программа без подпрограмм
Const
    n=4;
Type
    point = Record
      x,y:Real;
    End;
    pnts = Array[1..n] Of point;
Var
    arr1,arr2:pnts;
    i,j,k: Integer;

Begin
    arr1[1].x:=1.0; arr1[1].y:=1.0;
    arr1[2].x:=3.0; arr1[2].y:=1.0;
    arr1[3].x:=3.0; arr1[3].y:=3.0;
    arr1[4].x:=1.0; arr1[4].y:=3.0;
    For i:=1 To n do
    Begin
      j:=i+1;
      If j>n Then j:=1;
      arr2[i].x:=(arr1[i].x+arr1[j].x)/2;
      arr2[i].y:=(arr1[i].y+arr1[j].y)/2;
    End
End.
```

В этой программе для координаты точки введен тип `point`, координатам x и y соответствуют одноименные поля этой записи. Для координат всех вершин или середин сторон заведен массив таких записей (тип `pnts`).

Модифицируем теперь эту программу так, чтобы координаты середин сторон вычисляла специальная процедура с тремя параметрами — двумя входными (координаты вершин отрезка) и одним выходным (координаты середины отрезка).

```
Program with_subprogram_1; //Программа с подпрограммами
Const
    n=4;
Type
    point = Record
      x,y:Real;
    End;
    pnts = Array[1..n] Of point;
Var
    arr1,arr2:pnts;
    i,j,k:Integer;

Procedure middle(a,b:point; Var c:point);
Begin
    c.x:=(a.x+b.x)/2;
    c.y:=(a.y+b.y)/2;
End;
Begin
    arr1[1].x:=1.0; arr1[1].y:=1.0;
    arr1[2].x:=3.0; arr1[2].y:=1.0;
    arr1[3].x:=3.0; arr1[3].y:=3.0;
    arr1[4].x:=1.0; arr1[4].y:=3.0;
    For i:=1 To n do
    Begin
      j:=i+1;
      If j>n Then j:=1;
      middle(arr1[i],arr1[j],arr2[i]);
    End
End.
```

Модифицируем нашу программу еще раз. Пусть специальная процедура полностью решает поставленную задачу.

```
Program with_subprogram_2; //Программа с подпрограммами2
Const
    n=4;
Type
    point = Record
      x,y:Real;
    End;
    pnts = Array[1..n] Of point;
Var
    arr1,arr2:pnts;

Procedure middle(a,b:point; Var c:point);
Begin
    c.x:=(a.x+b.x)/2;
    c.y:=(a.y+b.y)/2;
End;
```

```
Procedure mid_arr;
Var
    i,j:Integer;
Begin
    For i:=1 To n do
    Begin
      j:=i+1;
      If j>n Then j:=1;
      middle(arr1[i],arr1[j],arr2[i]);
    End
End;

Begin
    arr1[1].x:=1.0; arr1[1].y:=1.0;
    arr1[2].x:=3.0; arr1[2].y:=1.0;
    arr1[3].x:=3.0; arr1[3].y:=3.0;
    arr1[4].x:=1.0; arr1[4].y:=3.0;
    mid_arr;
End.
```

Однако новый вариант программы не имеет особых преимуществ перед предыдущим вариантом. Усложним первоначальную задачу: пусть требуется найти координаты всех середин сторон n-угольника и координаты всех середин n-угольника, вершины которого лежат в серединах сторон первого n-угольника. В этом случае имеет смысл модифицировать процедуру `mid_arr`, сделав ее процедурой с параметрами. Тогда при первом обращении к этой процедуре можно будет определить координаты всех середин сторон n-угольника, а при втором — координаты всех середин n-угольника, вершины которого лежат в серединах сторон первого n-угольника. Заключительный вариант программы будет выглядеть следующим образом.

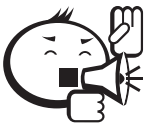
```
Program with_subprogram_3; //Программа с подпрограммами3
Const
    n=4;
Type
    point = Record
      x,y:Real;
    End;
    pnts = Array[1..n] Of point;
Var
    arr1,arr2,arr3:pnts;

Procedure middle(a,b:point; Var c:point);
Begin
    c.x:=(a.x+b.x)/2;
    c.y:=(a.y+b.y)/2;
End;

Procedure mid_arr(a:pnts; Var b:pnts);
Var
```

```
    i,j:Integer;
Begin
    For i:=1 To n do
    Begin
      j:=i+1;
      If j>n Then j:=1;
      middle(a[i],a[j],b[i]);
    End
End;

Begin
    arr1[1].x:=1.0; arr1[1].y:=1.0;
    arr1[2].x:=3.0; arr1[2].y:=1.0;
    arr1[3].x:=3.0; arr1[3].y:=3.0;
    arr1[4].x:=1.0; arr1[4].y:=3.0;
    mid_arr(arr1,arr2);
    mid_arr(arr2,arr3);
End.
```



Формальные параметры подпрограммы указывают, с какими параметрами следует обращаться к этой подпрограмме (количество параметров, их последовательность, типы).

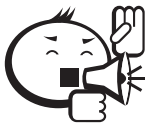
Они задаются в заголовке подпрограммы в виде списка, разбитого на группы, которые разделяются точками с запятыми. В группу формальных параметров включаются однотипные параметры одной категории.

Все формальные параметры разбиваются на четыре категории:

1. Параметры-значения, которые подпрограммой изменяться не могут. Они указываются в заголовке подпрограммы своим именем и через двоеточие — типом. Тип параметра-значения может быть любым за исключением файлового. Если параметров-значений одного типа несколько, то их можно объединить в одну группу, перечислив их имена через запятую, а затем указать общий тип.
2. Параметры-переменные, которые могут изменяться подпрограммой. Они указываются в заголовке подпрограммы аналогично параметрам-значениям, но только перед именем параметра-переменной записывается зарезервированное слово **Var**. Действие слова **Var** распространяется до ближайшей точки с запятой, т. е. в пределах одной группы.
3. Параметры-константы.
4. Параметры-процедуры и параметры-функции[1].

[1] Епанешников А. М., Епанешников В. А. Программирование в среде Turbo Pascal 7.0. — 3-е изд., стер. — М. : ДИАЛОГ-МИФИ, 1996.

Имена формальных параметров могут быть любыми, в том числе и совпадать с именами объектов основной программы. Необходимо лишь помнить, что в этом случае параметр основной программы с таким именем становится недоступным для непосредственного использования подпрограммой.

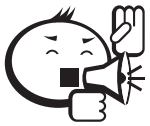


В заголовке подпрограммы нельзя вводить новый тип.

Например, нельзя писать

```
Procedure output_Real_vector (size_arr: Integer;
                    vector: Array [1..100] Of Real);
```

Чтобы правильно записать этот заголовок, следует в основной программе описать глобальный тип-массив, а затем использовать его в заголовке.



При обращении к подпрограмме формальные параметры заменяются соответствующими *фактическими параметрами* вызывающей программы или подпрограммы.

Таким образом, для вызова подпрограммы из основной программы или другой подпрограммы следует записать имя подпрограммы со списком фактических параметров, которые должны совпадать по количеству и типам с формальными параметрами подпрограммы.

Иногда бывает необходимо, чтобы подпрограмма вызывала саму себя (*рекурсивное обращение*, или коротко — *рекурсия*). В этом случае при каждом новом обращении к подпрограмме параметры, которые она использует, заносятся в стек (специальная область памяти), причём параметры предыдущего обращения также сохраняются. Подпрограммы с рекурсивными алгоритмами могут быть более компактными и эффективными, однако они являются более сложными для понимания логики выполняемых действий, и не следует забывать об опасности переполнения стека, размер которого ограничен.

Пример 4.2

Две подпрограммы-функции (обычная и с рекурсией) для вычисления факториала некоторого натурального числа. Вспомним, что факториал $n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$, причём 0!=1 и 1!=1. Например, $2! = \prod_{i=1}^{2} i = 1 \cdot 2 = 2$, $3! = \prod_{i=1}^{3} i = 1 \cdot 2 \cdot 3 = 6$, $4! = \prod_{i=1}^{4} i = 1 \cdot 2 \cdot 3 \cdot 4 = 24$, $5! = \prod_{i=1}^{5} i = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$ и т. д.

```
{Функция с именем factorial1 целого типа LongInt
для вычисления факториала числа с формальным
```

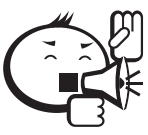
```
параметром-значением n целого типа Byte}
Function factorial1 (n: Byte): LongInt;
     Var
       f: LongInt; //Вспомогательная переменная
       i: Byte; //Параметр цикла типа Byte
     Begin
       f:=n; //Задание начального значения переменной f
       For i:=n-1 DownTo 2 Do
         If n=0 {Если n равно нулю} Then
           factorial1:=1
         Else
           factorial1:=f;
End; //Конец подпрограммы-функции factorial1
```

Код рекурсивной подпрограммы-функции будет несколько короче:

```
Function factorial12 (n: Byte): LongInt;
     Var
       f: LongInt; //Вспомогательная переменная
       i: Byte; //Параметр цикла типа Byte
     Begin
       If (n=0) Or (n = 1) Then
         factorial2:=1
Else
         factorial2:=n*factorial2(n-1);
             {присвоить функции значение выражения,
             в котором осуществляется умножение n на
             значение, возвращаемое рекурсивно вызванной
             функцией с параметром n без единицы}
End; //Конец подпрограммы-функции factorial2
```

4.2 Run-Time Library

В состав Free Pascal входят модули, реализующие основную функциональность. Конкретный состав модулей зависит от платформы. Кроме ядра динамической библиотеки Run-Time Library (RTL), оно присутствует всегда.



Модули RTL Free Pascal позволяют воспользоваться большим количеством уже готовых подпрограмм и упростить процедуру написания программы.

В программе модули, включая модули пользователя (см. п. 4.3), перечисляются через запятую после зарезервированного слова `Uses`.

RTL содержит много модулей, рассмотрим модули `System` и `Crt`.

Модуль System содержит подпрограммы, которые входят в стандарт языка Pascal. Он автоматически подключается к любой программе, поэтому его не следует указывать в разделе подключения библиотек, в случае явного указания будет инициирована ошибка двойного описания.

Помимо уже приведённых в других разделах данного пособия подпрограмм модуля System, познакомимся с подпрограммами, которые могут пригодиться при выполнении контрольных работ по курсу информатики.

Арифметические функции

1. **Frac**(x{: *Real*}){: *Real*} — возвращает дробную часть вещественного числа x. Например,

```
fraction:=Frac(Pi);
```

в результате переменная

```
fraction=0.1415926536...
```

2. **Int**(x{: *Real*}){: *Real*} — возвращает целую часть вещественного числа x, без преобразования типа. Например,

```
int_part:=Int(Pi);
```

в результате переменная

```
int_part=3.0
```

Функции для величин порядкового типа

1. **Odd**(x{: LongInt}){: *Boolean*} — возвращает истину (**True**), если целое число x нечётное, и ложь (**False**) — в противном случае. Например,

```
x:=1; parity_x:=Odd(x);
y:=0; parity_y:=Odd(y);
```

в результате переменные

```
parity_x = True //т.е. x — нечётное
parity_y = False //т.е. y — чётное
```

2. **Pred**(x{: порядковый тип}){: порядковый тип} — возвращает предшествующее значение x порядкового типа. Например,

```
x:=1; pred_x:=Pred(x);
y:='b'; pred_y:=Pred(y);
```

в результате переменные

```
pred_x=0 и pred_y='a'
```

3. **Succ**(x{: порядковый тип}){: порядковый тип} — возвращает следующее значение x порядкового типа. Например,

```
x:=1; succ_x:=Succ(x);
y:='b'; succ_y:=Succ(y);
```

в результате переменные

```
succ_x=2 и succ_y='c'
```

Функции преобразования типов

1. **Chr**(x{: Byte}){: *Char*} — возвращает символ с указанным кодом x. Например,

```
code:=65; symbol:=Chr(code);
```

в результате переменная

```
symbol='A'
```

2. **Ord**(x{: порядковый тип}){: LongInt} — возвращает порядковый номер значения x порядкового типа. Например,

```
symbol:='A'; code1:=Ord(symbol);
logic1:=True; code2:=Ord(logic1);
logic2:=False; code3:=Ord(logic2);
```

в результате переменные

```
code1=65, code2=1, code3=0
```

3. **High**(x{: порядковый тип, тип-строка, тип-массив, открытый массив}){: Word} — возвращает максимальное значение x. Например,

```
max_longint:=High(LongInt);
max_String:=High(String);
```

в результате переменные

```
max_longint=2147483647 и max_String=255
```

4. **Low**(x{: порядковый тип, тип-строка, тип-массив, открытый массив}){: Word} — возвращает минимальное значение x. Например,

```
min_longint:=Low(LongInt);
min_String:=Low(String);
```

в результате переменные

```
min_longint=-2147483648 и min_String=0
```

5. **Round**(x{: *Real*}){: LongInt} — округляет до целого вещественное значение x. Например,

```
x:=-6.5; round_x:=Round(x);
```

в результате переменная

```
round_x=-7
```

6. **Trunc**(x{: *Real*}){: LongInt} — возвращает целую часть вещественного значения x. Например,

```
x:=-6.5; trunc_x:=Trunc(x);
y:=6.5; trunc_y:=Trunc(y);
```

в результате переменные

```
x=-6 и y=6
```

Процедуры для величин порядкового типа

1. **Dec**({**Var**}x{: порядковый тип}, n{: LongInt}) — уменьшает значение x порядкового типа на величину необязательного параметра n целого типа. Если n отсутствует, то x уменьшается на единицу. Например,

```
i:=1; Dec (i);
k:=10; Dec (k, 5);
c:='Z'; Dec (c, 10);
```

в результате переменные

```
i=0, k=5, c='P'
```

2. **Inc**({**Var**}x{: порядковый тип}, n{: LongInt}) — увеличивает значение x порядкового типа на величину необязательного параметра n целого типа. Если n отсутствует, то x увеличивается на единицу. Например,

```
i:=1; Inc (i);
k:=10; Inc (k, 5);
c:='A'; Inc (c, 10);
```

в результате переменные

```
i=2, k=15, c='K'
```

Процедуры управления программой, не имеющие параметров:

- **Break** — осуществляет немедленный выход из циклов **Repeat**, **While** или **For**. Процедуру можно использовать только внутри цикла.
- **Continue** — прерывает текущую и начинает следующую итерацию циклов **Repeat**, **While** или **For**. Процедуру можно использовать только внутри цикла.
- **Exit** — осуществляет немедленный выход в вызывающую среду из подпрограммы или основной программы. В основной программе действует подобно процедуре `Halt`.
- **Halt** — прекращает выполнение программы.

Модуль `Crt` содержит константы, переменные и подпрограммы, предназначенные для работы с консолью. В отличие от стандартного ввода-вывода, когда он осуществляется через операционную систему, подпрограммы библиотеки `Crt` работают с базовой системой ввода-вывода (BIOS), а также непосредственно с видеопамятью.

При работе с экраном через `Crt` весь экран разбивается на отдельные строки, а каждая строка — на отельные позиции, в которые можно поместить символ (в том числе и пробел). Таким образом, весь экран разбивается на отдельные неделимые прямоугольные элементы. Для каждого элемента можно задать цвет фона (задний план) и цвет символа (передний план). Кроме того, при необходимости символ можно сделать мерцающим.

Рассмотрим основные подпрограммы `Crt`.

Процедуры

1. **TextMode** `(m{: Word})` — устанавливает текстовый режим `m` и увеличивает текущее окно до целого экрана. Значение `m` можно задавать как именем константы (`BW40` — 40 позиций на 25 строк режим чёрно-белый; `CO40` — цветной; `BW80` — ч/б…), так и номером (0, 1, 2…). Если значение `m` не совпадает ни с одной из констант, то по умолчанию устанавливается режим `CO80` (80×25 цветной) с номером 3.

2. **ClrScr** — очищает текущее окно, заполняя его цветом фона, и помещает курсор в его верхний левый угол с координатами $(1, 1)$.

3. **ClrEol** — удаляет все символы от курсора (включительно) до конца строки, заполняя этот участок строки цветом фона.

4. **DelLine** — удаляет строку, в которой находится курсор.

5. **InsLine** — вставляет пустую строку в месте расположения курсора и заполняет её цветом фона.

6. **GoToXY** `(x, y{: Byte})` — перемещает курсор к элементу экрана с координатами `x`, `y` (координаты отсчитываются от левого верхнего угла текущего окна). Если хотя бы одна из координат недопустима, процедура не выполняется.

7. **Window** (x1, y1, x2, y2 {: Byte}) — задаёт размеры окна координатами x1, y1 левого верхнего угла и x2, y2 правого нижнего угла, а также помещает курсор в левый верхний угол созданного окна с координатами $(1,1)$. Если хотя бы одна из координат недопустима, процедура не выполняется.

8. **TextBackGround** (c{: Byte}) — задаёт цвет фона c со значением от 0 до 7 либо непосредственно именем константы: Black (чёрный — 0), Blue (синий — 1) … LightGray (светло-серый — 7).

9. **TextColor** (c{: Byte}) — задаёт цвет символов c со значением от 0 до 15 либо непосредственно именем константы: Black (чёрный — 0), Blue (синий — 1) … White (белый — 15).

10. **Delay** (m{: Word}) — задерживает выполнение программы на m миллисекунд.

11. **Sound** (h{: Word}) — включает источник звука с частотой h герц.

12. **NoSound** — выключает источник звука.

Функции

1. **KeyPressed**{: *Boolean*} — возвращает истину (**True**), если нажата любая клавиша на клавиатуре, и ложь (**False**) — в противном случае.

2. **ReadKey**{: *Char*} — считывает символ с клавиатуры. Например,

```
c:=ReadKey;
```

в результате переменная c будет хранить значение введённого символа.

3. **WhereX**{: Byte} — возвращает текущую координату x курсора.

4. **WhereY**{: Byte} — возвращает текущую координату y курсора.

Пример 4.3

Программа, демонстрирующая работу подпрограмм модуля Crt.

```
Program demo_crt;
Uses Crt;
Var i, j: Byte;
Procedure music;
     Begin
       Sound (392); Delay (200); NoSound;
       Sound (330); Delay (100); NoSound;
       Sound (330); Delay (100); NoSound;
       Sound (392); Delay (200); NoSound;
       Sound (330); Delay (100); NoSound;
       Sound (330); Delay (100); NoSound;
       Sound (392); Delay (140); NoSound;
       Sound (349); Delay (140); NoSound;
       Sound (330); Delay (140); NoSound;
       Sound (294); Delay (140); NoSound;
       Sound (262); Delay (140); NoSound;
     End;
```

```
Begin
     TextMode (CO80);
     Repeat
       TextBackGround (LightGray);
       ClrScr;
       TextColor (Magenta);
       GoToXY (5, 2);
       Write ('ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
             СИСТЕМ ');
       Write ('УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ');
       GoToXY (19, 3);
       Write ('Кафедра прикладной математики
             и информатики');
       For i:=1 To 7 Do
       Begin
         Window (5, 7, 75, 22);
         TextBackGround (i);
         ClrScr;

         For j:=1 To 15 Do
         Begin
           TextColor (j);
           GoToXY (17, 8);
           Write ('Для прекращения нажмите любую клавишу');
           Delay (200);
         End;
         If KeyPressed Then Break;
       End;
       music;
     Until KeyPressed;
     TextMode (3);
End.
```

4.3 Модули пользователя

Создание собственных библиотек (*модулей пользователя*), содержащих подпрограммы и данные, позволяет программисту отлаживать программу по частям, а также практически неограниченно увеличивать код программы. Универсализация библиотечных подпрограмм позволяет использовать их в различных программах, что упрощает программирование сложных алгоритмов.

Модуль состоит из следующих частей:

1. Заголовок.
2. Интерфейс.

3. Исполнительная часть.
4. Секция инициализации.

Все разделы модуля, кроме секции инициализации, являются обязательными. Обязательна также указанная последовательность разделов.

Заголовок модуля состоит из зарезервированного слова `Unit` и идентификатора (имени) модуля. Идентификатор модуля должен быть уникальным.

Модуль помещается в файл, имя которого должно обязательно совпадать с именем модуля, указанным в заголовке, а расширение файла должно быть *.pas.

Через *интерфейс* осуществляется взаимодействие основной программы с модулем или модуля с модулем. Интерфейс начинается словом `Interface`. Далее идут разделы объявлений аналогично структуре основной программы. Идентификаторы из этих разделов могут быть использованы основной программой или другими модулями при вызове данного модуля. В разделе объявлений процедур и функций указываются лишь заголовки подпрограмм, а сами подпрограммы приводятся в исполнительной части модуля.

Исполнительная часть включает все подпрограммы модуля. В ней могут содержаться также локальные для неё разделы объявлений. Исполнительная часть начинается словом `Implementation`, после которого идут разделы объявлений аналогично структуре основной программы. При описании подпрограмм допустимо использовать сокращённые заголовки, т. е. без списка формальных параметров, если заголовок подпрограммы указан в секции `Interface`.

В некоторых случаях перед обращением к модулю следует провести ее инициализацию (например, установить связь с теми или иными файлами с помощью процедуры `Assign`). Необходимые действия можно выполнить в *секции инициализации* модуля. Эта секция начинается словом **`Begin`**, после которого идут исполняемые операторы, и заканчивается словом **`End`** с точкой. Если инициализация модуля не нужна, то в этой секции помещается только **`End`** с точкой.

После того, как текст модуля записан в файле с расширением *.pas, необходимо организовать модуль. Для этого в меню нужно открыть окно Compile и выполнить опцию Build. В результате на диске будут созданы файлы с именем модуля и расширениеми *.o и *.ppu. Файл с расширением *.o должен находиться в том же каталоге, где и программа, использующая этот модуль, или в каталоге, где расположены стандартные модули.

Пример 4.4

Библиотека, содержащая две процедуры заполнения одномерных или двумерных массивов задаваемых размеров случайными целыми или вещественными числами в задаваемом диапазоне и две процедуры вывода полученных массивов на экран.

```
Unit io_arr;
Interface
     Type real_vector = Array [1..100] Of Real;
```

```
          int_vector = Array [1..100] Of Integer;
          real_matrix = Array [1..10, 1..10] Of Real;
          int_matrix = Array [1..10, 1..10] Of Integer;
      Procedure input_vector (float: Boolean;
            Var n: Integer; Var vector);
      Procedure input_matrix (float: Boolean;
            Var n, m: Integer; Var matrix);
      Procedure output_vector (float: Boolean;
            Var n: Integer; Var vector);
      Procedure output_matrix (float: Boolean;
            Var n, m: Integer; Var matrix);
Implementation
  Procedure input_number_range (Var p1, p2: Real);
    Var a, b: Real;
    Begin
      WriteLN ('Введите границы диапазона',
        'случайных чисел');
      Write ('a='); ReadLN (a);
      Write ('b='); ReadLN (b);
      If a>b
        Then
        Begin
          p1:=a; a:=b; b:=p1;
        End;
      If a=b
        Then
        Begin
          p1:=0; p2:=b;
        End
        Else
          If b=0
            Then
            Begin
              p1:=a; p2:=0;
            End
            Else
            Begin
              p1:=b-a; p2:=a;
            End;
    End;
  Procedure input_vector (float: Boolean;
Var n: Integer; Var vector);
    Var par1, par2: Real;
      i: Integer;
      v: real_vector;
    Begin
```

```
        Write ('Введите количество элементов массива ');
        WriteLN ('(не более 100)');
        ReadLN (n);
        input_number_range (par1, par2);
        Randomize;
        For i:=1 To n Do v[i]:=Random*par1+par2;
        If float
          Then
            For i:=1 To n Do
              real_vector(vector)[i]:=v[i]
          Else
            For i:=1 To n Do
              int_vector(vector)[i]:=Round(v[i]);
        End;
    Procedure input_matrix(float: Boolean;
  Var n, m: Integer; Var matrix);
      Var par1, par2: Real;
        i, j: Integer;
        v: real_matrix;
      Begin
        Write ('Введите количество строк матрицы ');
        WriteLN ('(не более 10)');
        ReadLN (n);
        Write ('Введите количество столбцов матрицы ');
        WriteLN ('(не более 10)');
        ReadLN (m);
        input_number_range (par1, par2);
        Randomize;
        For i:=1 To n Do
          For j:=1 To m Do
            v[i,j]:=Random*par1+par2;
        If float
          Then
            For i:=1 To n Do
              For j:=1 To m Do
                real_matrix(matrix)[i,j]:=v[i,j]
          Else
            For i:=1 To n Do
              For j:=1 To m Do
                int_matrix(matrix)[i,j]:=Round(v[i,j]);
      End;
    Procedure output_vector(float: Boolean;
  Var n: Integer; Var vector);
      Var q2, i: Integer;
      Begin
        If float
```

```
          Then
          Begin
            Write ('Задайте: с точностью до ',
                  'какого знака ');
            Write ('после десятичной точки выводить ');
            WriteLN ('вещественные элементы массива');
            ReadLN (q2);
            WriteLN ('Массив:');
            For i:=1 To n Do
              Write (real_vector(vector)[i]: 0: q2, ' ');
          End
          Else
          Begin
            WriteLN ('Массив:');
            For i:=1 To n Do
              Write (int_vector(vector)[i], ' ');
          End;
        WriteLN;
        WriteLN ('Нажмите Enter');
        ReadLN;
      End;
    Procedure output_matrix(float: Boolean;
Var n, m: Integer; Var matrix);
      Var q1, q2, i, j: Integer;
      Begin
        If float
          Then
          Begin
            Write ('Задайте общее количество ',
                  'знаков под вывод ');
            WriteLN ('вещественного элемента матрицы');
            ReadLN (q1);
            Write ('Задайте: с точностью до какого ',
                  'знака после ');
            Write ('десятичной точки выводить ',
                  'вещественный ');
            WriteLN ('элемент матрицы');
            ReadLN (q2);
            WriteLN ('Матрица:');
            For i:=1 To n Do
            Begin
              For j:=1 To m Do
                Write (real_matrix(matrix)[i,j]: q1: q2, ' ');
              WriteLN;
            End;
          End
```

```
          Else
          Begin
            Write ('Задайте общее количество ',
                  'знаков под вывод ');
            WriteLN ('элемента матрицы');
            ReadLN (q1);
            WriteLN ('Матрица:');
            For i:=1 To n Do
            Begin
              For j:=1 To m Do
                Write (int_matrix(matrix))[i,j]: q1, ' ');
              WriteLN;
            End;
          End;
        WriteLN;
        WriteLN ('Нажмите Enter');
        ReadLN;
      End;
  End.
```

Программа, в которой вызываются процедуры библиотеки io_arr.

```
Program uses_io_arr;
Uses Crt, io_arr;
Var selection: Char;
      n, m: Integer;
      v_r: real_vector;
      v_i: int_vector;
      m_r: real_matrix;
      m_i: int_matrix;
Begin
    Repeat
      ClrScr;
      Write ('Для заполнения случайными числами ');
      WriteLN ('и вывода на экран');
      WriteLN ('вещественного одномерного ',
               'массива введите "1"');
      WriteLN ('целого одномерного массива ', '-"2"');
      WriteLN ('вещественной матрицы ', '-"3"');
      WriteLN ('целой матрицы ', '-"4"');
      WriteLN ('Выход из программы ', '-"0"');
      ReadLN (selection);
      Case selection Of
        '1': Begin
          input_vector (True, n, v_r);
          output_vector (True, n, v_r);
```

```
          End;
          '2': Begin
            input_vector (False, n, v_i);
            output_vector (False, n, v_i);
          End;
          '3': Begin
            input_matrix (True, n, m, m_r);
            output_matrix (True, n, m, m_r);
          End;
          '4': Begin
            input_matrix (False, n, m, m_i);
            output_matrix (False, n, m, m_i);
          End;
          '0': Halt;
          Else
            WriteLN ('Введите "1"2"3"4"или "0"');
        End;
      Until False;
End.
```

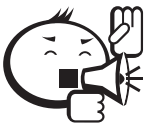
Контрольные вопросы по главе 4

1. Что такое подпрограмма?
2. Какие виды подпрограмм существуют во Free Pascal?
3. Что такое локальные параметры?
4. Что такое глобальные параметры?
5. Что такое формальные параметры?
6. Что такое фактические параметры?
7. В чем принцип рекурсии?
8. Для чего нужна RTL?
9. Что такое модули пользователя?
10. Для чего нужен интерфейс модуля?

Глава 5

ИНТЕГРИРОВАННАЯ СРЕДА ПРОГРАММИРОВАНИЯ FREE PASCAL

5.1 Система Free Pascal



Процессор не «понимает» никаких языков программирования, поэтому в компьютере используется специальная машинная программа, называемая *компилятором*.

Компилятор преобразует программу, записанную на любом языке высокого уровня, в программу на машинном языке. Только после компиляции программа может быть запущена на компьютере.

Для одного и того же языка программирования существуют, как правило, несколько различных компиляторов, разработанных разными фирмами.

Версии Free Pascal имеют интегрированную среду программирования (Integrated Development Environment – далее IDE), внешне очень напоминающую среду программирования Turbo Pascal, включающую в себя: 1) экранный редактор исходного текста; 2) компилятор; 3) отладчик; 4) модули подпрограмм; 5) систему контекстной информационной помощи. Основные особенности IDE:

- возможность использования многих перекрывающихся окон, у которых можно менять размеры и которые можно перемещать по экрану;
- наличие развитой системы меню;
- наличие диалоговых окон;
- поддержка работы с «мышью»;
- многофайловый экранный редактор, причём можно осуществлять обмен информацией между отдельными окнами редактирования, а также использовать сведения из системы информационной помощи;
- увеличенные возможности отладчика;
- возможность полной очистки и восстановления экрана.

С помощью IDE программист реализует весь цикл работ по созданию законченной, годной для дальнейшей эксплуатации машинной программы. Вначале он в *экранном редакторе исходного текста* вводит программу с помощью клавиатуры в компьютер и сохраняет на диске для последующего использования и редактирования. Затем *компилирует* её. В случае синтаксических (возникающих в результате нарушения правил написания предложений языка) и семантических (связанных с недопустимыми значениями параметров, недопустимыми действиями над параметрами и т. п.) ошибок в тексте программы компилятор выдаёт диагностические сообщения. Тогда программисту приходится возвращаться на этап редактирования, чтобы исправить ошибки. Наконец, после успешной трансляции программист запускает программу на пробное исполнение (тестирование). Здесь также возможно обнаружение ошибок, но уже логических, т. е. ошибок в последовательности выполняемых действий или в самих действиях. *Отладчик* помогает отследить исполнение программы и определить источник таких ошибок.

Модули подпрограмм позволяют программисту составлять программы, включая в них заранее подготовленные вспомогательные *подпрограммы*. Они разделены на группы по различным областям применения. С помощью библиотек подпрограмм программист ускоряет и облегчает себе процесс разработки сложных алгоритмов.

5.2 Настройка IDE Free Pascal для работы

IDE запускается с помощью файла FP.EXE и может по желанию пользователя легко модифицироваться. После загрузки файла FP.EXE под Windows на экране дисплея в небольшом окне появляется основной экран IDE, имеющий вид, показанный на рисунке 5.1.

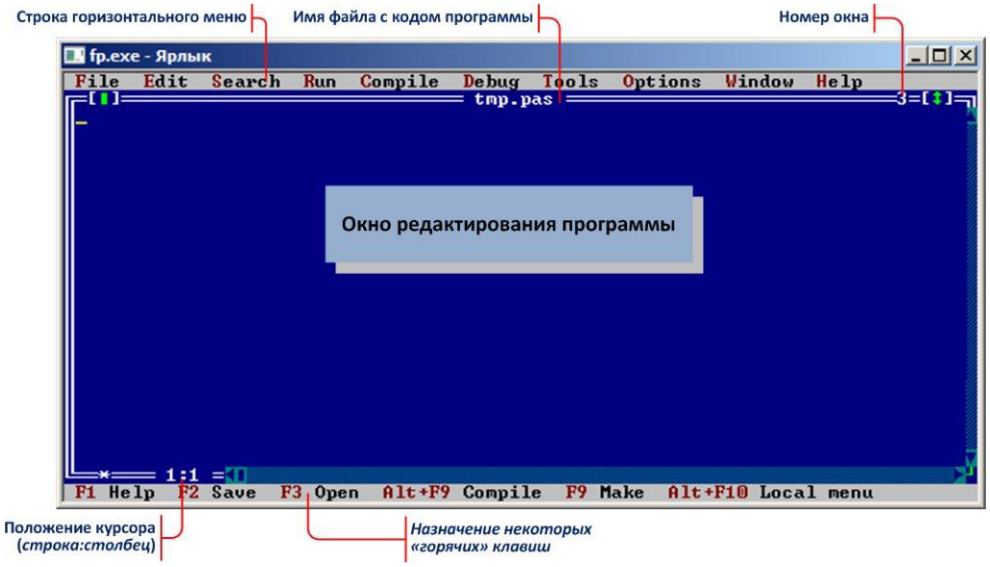


Рис. 5.1 – Экран интегрированной среды Free Pascal

Для настройки рабочей папки в IDE необходимо выполнить следующие действия.

1. С помощью «мыши» или клавиш [F10] ⇒ [←] и [→] ⇒ [Enter] выбрать пункт работы с файлами File (файл) из основного меню, расположенного в верхней строке окна.
2. В подменю выполнить команду New (новый). В результате будет открыт файл с именем NONAME00.PAS.
3. Ещё раз войти в подменю File основного меню.
4. Выбрать пункт Change dir... (изменение директории).
5. В появившемся окне можно либо в поле Directory name вручную набрать путь к своим файлам (в нашем случае — C:\MY_PROGR), либо в поле Directory tree найти и выделить свою папку, используя команды Chdir и Revert (см. рис. 5.2). Для перемещения по пунктам, полям и командам в окнах с помощью клавиатуры используются клавиши управления курсором (стрелки) и [Tab]. При этом элементы окон выделяются цветом.
6. После настройки рабочей папки для выхода из окна Change Directory выполнить команду OK.
7. Из основного меню открыть подменю Options (опции).
8. Выбрать пункт Directories... (директории).

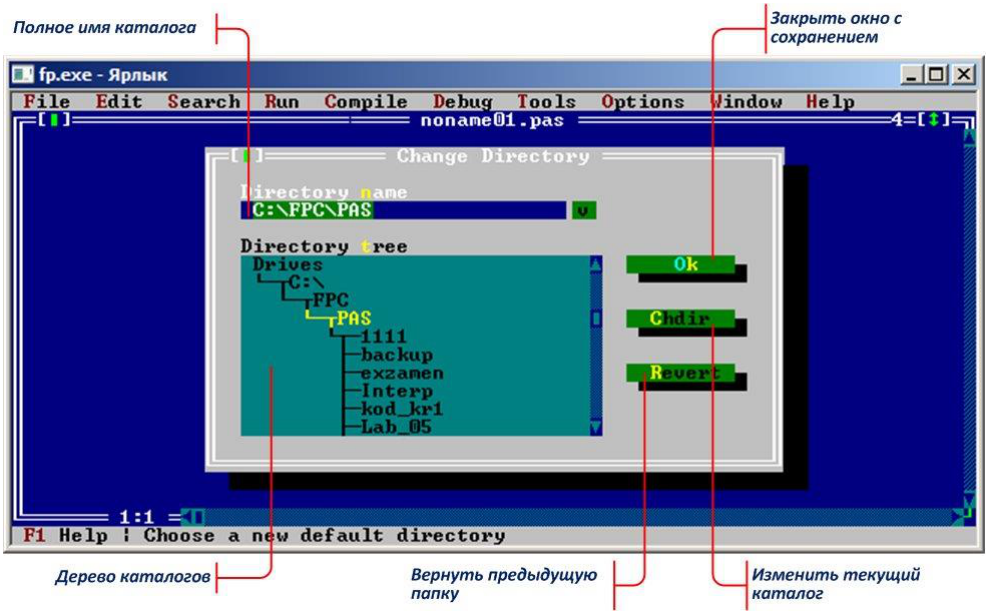


Рис. 5.2 – Экран IDE с окном Change Directory (изменение директории)

9. В открывшемся окне на закладке Misc. в поле EXE output directory ввести путь к своим файлам, как на рисунке 5.3.
10. Выполнить команду OK.
11. Для сохранения настроек необходимо ещё раз из основного меню открыть подменю Options и выполнить команду Save FP.INI (сохранить в файле FP.INI).

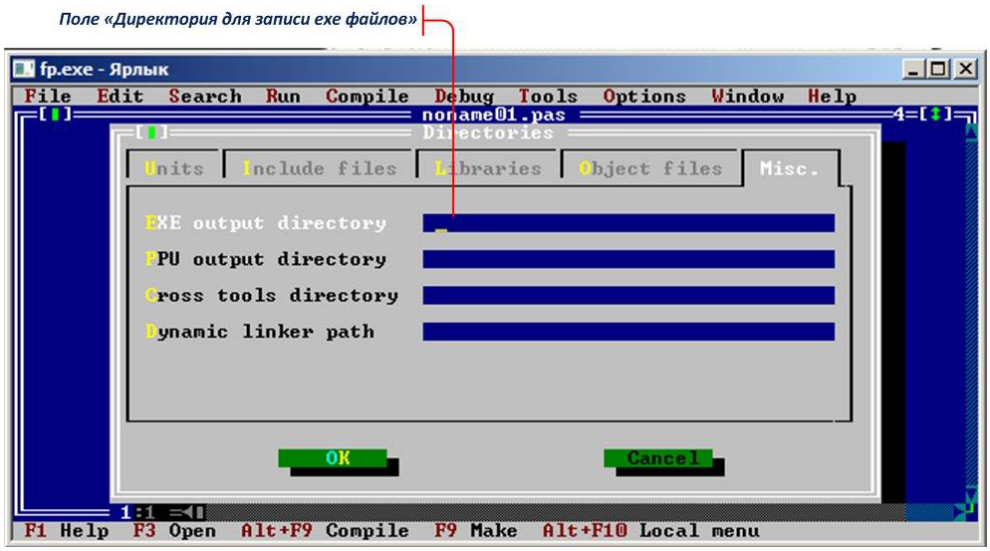


Рис. 5.3 – Экран IDE с окном Directories (директории)

5.3 Редактирование исходного текста программ

Мигающий курсор указывает, где будет воспроизводиться вводимый с клавиатуры символ. Передвижение курсора внутри текста производится клавишами со стрелками. Очередной символ вводится в двух режимах: со вставкой, тогда символы в строке раздвигаются, и с заменой ранее введённого в текущей позиции символа. Переключаются режимы нажатием [Insert]. Клавиша [Home] смещает курсор в начало строки, а [End] — в конец, [PageUp] и [PageDown] «листают» текст вверх и вниз соответственно. Клавиша [Backspase] удаляет символ слева от курсора, а [Delete] — символ над курсором.

Каждая введённая строка текста заканчивается невидимым символом перехода на новую строку. Этот символ вводится при нажатии [Enter], если включен режим вставки. В этом случае если курсор находится в конце строки, то вставляется новая пустая строка, а если в середине, то строка разрывается на две. Удаляется этот невидимый символ клавишей [Backspase] или [Delete], как и любой другой символ.

Более сложные действия по редактированию можно выполнять, нажимая комбинации клавиш. Так, для удаления всей строки текста, в которой находится курсор, надо, удерживая [Ctrl], нажать [Y].

Удобно работать с фрагментами (блоками) текста. Комбинация [Ctrl]+[K] включает работу с блоком. Если сразу после этого нажать [B], то отметится начало блока в позиции курсора. Перемещение курсора на другую позицию и нажатие [Ctrl]+[K] ⇒ [K] выделяет блок цветом. После выделения с блоками возможны следующие операции: [Ctrl]+[K] ⇒ [C] — копирование блока в местонахождение курсора; [Ctrl]+[K] ⇒ [V] — перемещение блока на новое место; [Ctrl]+[K] ⇒ [Y] — удаление блока. Нажатие [Ctrl]+[K] ⇒ [H] снимает цветовое выделение блока.

Можно выделять блоки, используя комбинации клавиш с [Shift], которые представлены в таблице 5.1.

Таблица 5.1 – Выделение текста с помощью клавиатуры

Комбинации клавиш	Действия
[Shift]+[←]	выделение одного символа слева от курсора
[Shift]+[→]	выделение одного символа справа от курсора
[Shift]+[↑]	выделение одной строки вверх от курсора
[Shift]+[↓]	выделение одной строки вниз от курсора
[Shift]+[Home]	выделение до начала строки от курсора
[Shift]+[End]	выделение до конца строки от курсора
[Shift]+[PageUp]	выделение одного экрана вверх от курсора
[Shift]+[PageDown]	выделение одного экрана вниз от курсора
[Shift]+[Ctrl]+[←]	выделение одного слова слева от курсора
[Shift]+[Ctrl]+[→]	выделение одного слова справа от курсора

Также выделение блоков возможно при помощи «мыши». Для этого курсор «мыши» устанавливаем в начальную позицию и, удерживая левую кнопку «мыши», перемещаем его в конечную позицию, при этом выделяется нужный фрагмент текста программы. Нажатие правой кнопки «мыши» вызывает выпадающее меню, содержащее команды работы с блоками (см. рис. 5.4). Эти команды дублированы в подменю Edit основного меню IDE.

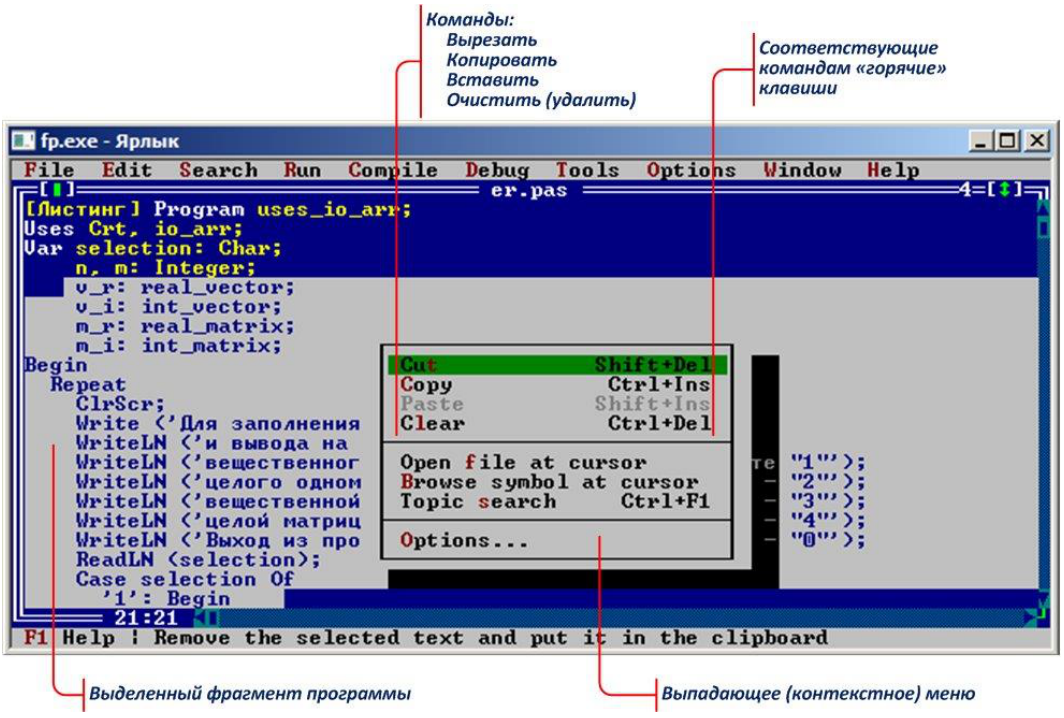


Рис. 5.4 – Экран IDE с выпадающим, при нажатии правой кнопки «мыши», меню

Таким образом, работа в IDE с фрагментами текста программы возможна тремя способами: с использованием комбинаций клавиш [Ctrl]+[K], сохранившихся из ранних версий Turbo Pascal, комбинаций с [Shift] и третий способ — при помощи «мыши», которые также могут применяться в известном текстовом редакторе MS Word. Выберете при работе наиболее удобный для Вас способ.

Сделанные в программе исправления отменяются командой Undo (отмена) подменю Edit или нажатием [Alt]+[Backspase]. Команда Redo (повтор) возвращает исправления после выполнения Undo.

5.4 Работа с файлами

Название файла в системах MS DOS и Windows состоит из имени и расширения, которые разделяются точкой. Файлы с исходными текстами программ на Free Pascal должны иметь расширение PAS. Имя файла может быть любым, однако при работе в Free Pascal для имён файлов рекомендуется использовать последовательности латинских букв, цифр и знака подчёркивания длиной до восьми символов.

После запуска Free Pascal для ввода текста программы по умолчанию открывается файл с названием NONAME00.PAS. Набрав текст программы, её необходимо сохранить. Это можно сделать, выполнив команду Save as... (сохранить как) подменю File основного меню или нажав [F2], в открывшемся окне Save File As (сохранить файл как) в поле Save file as нужно ввести имя файла (можно без расширения) и выполнить команду OK. В дальнейшем, чтобы открыть уже существующий файл, следует выполнить команду Open... (открыть) или нажать [F3], ввести имя файла в поле Name окна Open a File и выполнить OK. Сохранение изменений в существующем файле производится командой Save или нажатием [F2].

Команда Exit (выход) подменю File или комбинация клавиш [Alt]+[X] завершает работу с системой.

Удобно настроить компьютер на запуск Free Pascal по расширению файлов PAS, как, например, по умолчанию настроены документы Word, Excel и другие файлы MS Office. Под Windows это можно сделать следующим образом:

1) щёлкнув правой кнопкой «мыши» на любом файле с расширением PAS, вызвать выпадающее меню и выбрать Свойства;
2) в открывшемся окне нажать на кнопку «изменить»;
3) в окне Выбор программы нажать «обзор»;
4) найти и выделить файл FP.EXE на диске, нажать «открыть».

При выполнении описанных действий все окна открываются последовательно и закрываются в обратном порядке. После такой настройки, чтобы сразу открыть существующий файл с текстом программы, достаточно найти его на диске и запустить. При этом произойдёт запуск Free Pascal, и изменения в программе будут сохраняться в файл, расположенный в том же месте на диске, откуда произошёл запуск.

5.5 Компиляция и исполнение программ

После ввода с клавиатуры или чтения с диска программу можно запустить на исполнение, выполнив команду Run (пуск) одноимённого подменю или нажав [Ctrl]+[F9]. При этом сначала происходит компиляция программы и может быть обнаружена ошибка. Тогда будет выдано соответствующее сообщение. Перевод таких сообщений приводится в приложении. Для исправления ошибки необходимо

тщательно проанализировать текст программы выше курсора. После исправления одной ошибки может обнаружиться другая и т. д.

Наконец, после выдачи сообщения об успешной компиляции программа запускается на исполнение и произойдёт переключение экрана — вместо окна IDE Free Pascal появится окно, в котором производится ввод исходных данных и вывод результатов (если, конечно, ввод и вывод предусмотрен). После завершения работы программы снова появится окно IDE. Чтобы просмотреть экран с результатами, надо нажать [Alt]+[F5], а чтобы вернуться в IDE — любую клавишу.

Можно компилировать программу без её последующего запуска при помощи команды Compile одноимённого подменю или комбинации клавиш [Alt]+[F9].

Иногда случается, что после запуска программы никак не удаётся дождаться конца её работы. Это может быть из-за того, что исполнение алгоритма требует очень много действий, иными словами, алгоритм имеет большую трудоёмкость. Такое может случиться, если алгоритм не очень хорош (не эффективен), а количество обрабатываемых данных велико. Если же алгоритм исполняется на простом проверочном примере, то, скорее всего, он просто «зациклился», т. е. повторяет без всякого изменения бесконечную последовательность одних и тех же действий. Такой алгоритм применять бесполезно: результата вычислений никогда не получишь! Чтобы принудительно прекратить исполнение зациклившейся программы, необходимо нажать [Ctrl]+[Break].

Контрольные вопросы по главе 5

1. Что такое компилятор?
2. Для чего нужна IDE?
3. Где осуществляется редактирование текста программы?
4. Что такое отладчик?
5. Чем отличается выполнение программы по шагам, с использованием клавиш F7 и F8?
6. Как открыть для просмотра окно выполнения программы?

ЗАКЛЮЧЕНИЕ

Данное учебное пособие представляет собой возможность объяснить доступно и понятно, что же такое программирование и как научиться в короткие сроки создавать самостоятельно алгоритмы и программы на языке программирования. Авторами не преследовалась цель описать все возможности языка программирования Free Pascal и возможные конструкции. Содержание пособия является отражением опыта авторов, полученного в процессе многолетнего преподавания данной дисциплины. Максимально подробно расписаны те моменты, которые вызывали наибольшее затруднение в понимании слушателей при проведении занятий по соответствующей дисциплине. Авторы постарались дать ту основу, которая поможет студентам, не владеющим навыками программирования, в дальнейшем усовершенствовать полученные базовые знания при освоении и восприятии данной области.

ЛИТЕРАТУРА

[1] Острейковский В. А. Информатика : учебник для вузов / В. А. Острейковский. — М. : Высш. шк., 1999.

[2] Вирт Н. Алгоритмы и структуры данных / Н. Вирт. — М. : Мир, 1989.

[3] Епанешников А. М. Программирование в среде Turbo Pascal 7.0 / А. М. Епанешников, В. А. Епанешников. — 3-е изд., стер. — М. : ДИАЛОГ-МИФИ, 1996.

[4] Культин Н. Б. Turbo Pascal в задачах и примерах / Н. Б. Культин. — СПб. : БХВ-Петербург, 2000.

[5] Немнюгин С. А. Turbo Pascal / С. А. Немнюгин. — СПб. : Питер, 2000.

[6] Шпак Ю. А. Turbo Pascal 7.0 на примерах / Ю. А. Шпак ; под ред. Ю. С. Ковтанюка. — Киев : Юниор, 2003.

Приложение А

СООБЩЕНИЯ ОБ ОШИБКАХ В ПРОГРАММЕ

`BEGIN expected` — ожидается `BEGIN`.
`Boolean expression expected` — ожидается логическое выражение.
`Character expression expected` — ожидается выражение над символами.
`Compilation aborted` — компиляция прервана.
`Constant expected` — ожидается константа.
`Constant out of range` — значение константы вне допустимого диапазона.
`DO expected` — ожидается `DO`.
`Duplicate identifier` — повторное описание для одного и того же имени.
`END expected` — ожидается `END`.
`Error in expression` — ошибка в выражении.
`Error in integer constant` — ошибка в целой константе.
`Error in real constant` — ошибка в вещественной константе.
`Error in statement` — ошибка в операторе.
`Error in type` — ошибка в типе.
`Expected`... — ожидается... (какой-либо символ).
`Identifier expected` — ожидается имя.
`Integer constant expected` — ожидается константа целого типа.
`Integer expression expected` — ожидается выражение целого типа.
`Integer or real constant expected` — ожидается константа целого или вещественного типа.
`Integer or real expression expected` — ожидается выражение целого или вещественного типа.
`Integer or real variable expected` — ожидается переменная целого или вещественного типа.
`Integer variable expected` — ожидается переменная целого типа.
`Invalid FOR control variable` — неправильная переменная — параметр цикла `FOR`.

`Invalid floating point operation` — неправильная операция для вещественных операндов.

`Invalid function result type` — неправильный тип для результата функции.

`Invalid string length` — неправильная длина строки.

`Line too long` — строка в тексте программы слишком длинная.

`Lower bound greater than upper bound` — нижняя граница больше верхней (в описании массива).

`Operand types do not match operator` — несоответствие типов операндов и операций.

`Ordinal expression expected` — ожидается выражение целого или символьного типа.

`Ordinal type expected` — ожидается целый или символьный тип.

`Ordinal variable expected` — ожидается переменная целого или символьного типа.

`String constant exceeds line` — константа типа `String` должна записываться в программе без переноса на новую строку.

`String constant expected` — ожидается константа типа `String`.

`String expression expected` — ожидается выражение типа `String`.

`String length mismatch` — несоответствие длины для типа `String`.

`String variable expected` — ожидается переменная типа `String`.

`Syntax error` — синтаксическая ошибка.

`THEN expected` — ожидается `THEN`.

`TO or DOWNTO expected` — ожидается `TO` или `DOWNTO`.

`Too many symbols` — слишком много символов.

`Too many variables` — слишком много переменных.

`Type identifier expected` — ожидается имя типа.

`Type mismatch` — несоответствие типа.

`Undefined type` — неопределённый тип.

`Unexpected end of file` — неожиданный конец файла (нет точки в конце программы).

`Unknown identifier` — неизвестное имя.

`Variable identifier expected` — ожидается имя переменной.

Приложение Б

СООБЩЕНИЯ ПРИ ИСПОЛНЕНИИ ПРОГРАММЫ

`Abnormal program termination` — ненормальное окончание программы.
`Ctrl-Break to quit` — нажмите `Ctrl` и `Break` для выхода.
`Disk full` — на диске нет места.
`Disk is write-protected` — диск защищён от записи.
`Disk read error` — ошибка чтения с диска.
`Disk write error` — ошибка записи на диск.
`Division by zero` — деление на ноль.
`Drive not ready` — дисковод не готов (нет дискеты).
`Error on execute` — ошибка исполнения.
`Floating point overflow` — переполнение вещественного числа.
`Hardware failure` — ошибка оборудования.
`Heap overflow error` — не хватает основной памяти.
`I/O checking` — ошибка ввода/вывода.
`Invalid Breakpoint` — неправильная точка останова.
`Invalid format specifier` — неправильный формат.
`Must be in 8087 mode to compile this` — требуется компиляция с установленным режимом сопроцессора 8087.
`No debug info` — не задана информация для отладки.
`Not enough memory` — не хватает памяти.
`Out of memory` — не хватает памяти.
`Press any key` — нажмите любую клавишу.
`Program terminated` — исполнение программы прекращено.
`Range check error` — выход за пределы индекса в массиве.
`Runtime error` — ошибка во время исполнения.
`Stack overflow error` — переполнение стека.
`Too many breakpoints` — слишком много точек останова.
`User break` — пользователь прервал исполнение программы.

ГЛОССАРИЙ

Алгоритм — это точное предписание, которое задаёт алгоритмический процесс, начинающийся с произвольного исходного данного (из некоторой совокупности возможных исходных данных) и направленный на получение полностью определённого этим исходным данным результата.

Алгоритмический процесс — процесс последовательного преобразования конструктивных объектов (слов, чисел, пар слов, пар чисел, предложений и т. п.), происходящий элементарными «шагами». Каждый шаг состоит в смене одного конструктивного объекта другим.

Алфавит языка — совокупность допустимых в языке символов или групп символов, рассматриваемых как единое целое.

Арифметические операции — операции, применимые только к величинам вещественных и целых типов.

Бинарная операция — операция, в которой участвуют два операнда.

Блок-схема — это композиция ориентированных элементарных схем.

Вещественные типы — типы данных, которые определяют дробные числа.

Выражение — синтаксическая единица языка, определяющая порядок вычисления некоторого значения.

Глобальные параметры — переменные, объявляемые в основной программе и доступные как программе, так и всем её подпрограммам.

Запись — фиксированная последовательность элементов различного типа.

Зарезервированные (ключевые) слова — составляют основу языка, не могут переопределяться пользователем.

Знаки операций — зарезервированные слова, обозначающие арифметические, логические и другие действия. Могут состоять из небуквенных символов или сочетания букв.

Идентификатор — неделимая последовательность символов, задающая имя любого объекта программы.

Итерация — однократное выполнение тела цикла.

Комментарий — специальным образом выделенный фрагмент кода, не обрабатываемый компилятором, который предназначен для пояснения исходного текста программы.

Константа — элемент данных программы, значения которого нельзя изменить в процессе её выполнения.

Логический тип (`Boolean`) — тип данных, каждый элемент которого может принимать одно из двух значений: ***`False`*** (ложь) или ***`True`*** (истина).

Локальные параметры — переменные, объявляемые внутри подпрограммы и доступные только ей самой.

Массив — фиксированная последовательность упорядоченных однотипных компонент, снабжённых индексами.

Метод подъёма — метод программирования, при котором строятся решения задачи для нескольких частных случаев задания исходных данных, затем обобщаются полученные алгоритмы и записывается алгоритм, с помощью которого можно получить решение в случае задания любых исходных данных.

Метод частных целей — метод программирования при котором осуществляют разделение основной трудной задачи на последовательности более простых задач.

Многострочный комментарий — комментарий, расположенный между символами { } или (* *) , может занимать несколько строк.

Однострочный комментарий — комментарий, начинающийся с символов // и заканчивающийся концом строки в которой расположен.

Оператор — зарезервированное слово, которому соответствуют некоторые алгоритмические действия.

Оператор выбора (`Case`) — оператор, позволяющий выбрать вариант из любого количества вариантов.

Оператор обращения к процедуре — оператор, который служит для вызова процедуры.

Оператор цикла — оператор, позволяющий организовать выполнение одного оператора некоторое число раз.

Операции отношения — операции, предназначеные для сравнения двух величин.

Переменная — элементы данных программы, значение которого может изменяться в процессе её выполнения.

Процедура — подпрограмма, предназначенная для выполнения какой-либо законченной последовательности действий.

Подпрограмма-функция — подпрограмма, предназначенная для вычисления какого-либо параметра.

Программирование сверху вниз — это процесс пошагового разбиения алгоритма на всё более мелкие части с целью получения таких элементов, для которых можно написать конкретные команды.

Простой оператор — оператор, который не содержит в себе других операторов.

Простой тип — определяет упорядоченное множество значений элементов данных программы.

Пустой оператор — оператор, который не выполняет никакого действия и никак не отображается в программе.

Разделитель — специальный символ, предназначенный для отделения друг от друга элементов программы.

Рекурсия — алгоритм, который в процессе выполнения вызывает сам себя.

Символьный тип (`Char`) — тип данных, который служит для хранения одного символа из набора в 256 символов.

Сортировка массива — перераспределение элементов массива в порядке возрастания или убывания значений элементов.

Составной оператор — оператор, представляющий собой совокупность последовательно выполняемых операторов, заключённых в операторные скобки **`Begin`** и **`End`**.

Специальные символы — символы, выполняющие в языке определённые функции, включают в себя знаки пунктуации; знаки операций; зарезервированные слова.

Строка — последовательность символов длиной до 255 элементов.

Структурированный тип данных — последовательность однотипных или разнотипных элементов образованная из других типов данных.

Структурное программирование — это процесс разработки алгоритмов с помощью блок-схем.

Теория алгоритмов — раздел математики, изучающий общие свойства алгоритмов.

Тип-файл — последовательность элементов одного типа, расположенных на дисках.

Тип данных — определяет множество допустимых значений этих данных, а также совокупность операций над ними.

Унарная операция — операция, в которой присутствует только один элемент (*операнд*).

Условный оператор (`If`) — оператор, реализующий алгоритмическую конструкцию разветвляющегося процесса.

Целые типы — типы данных, которые определяют целые числа.

Предметный указатель

Учебное издание

Гураков Алексей Валерьевич
Мещерякова Ольга Ивановна
Мещеряков Павел Сергеевич

ИНФОРМАТИКА II

Учебное пособие

Корректор Осипова Е. А.
Компьютерная верстка Мурзагулова Н. Е.

Издано в Томском государственном университете
систем управления и радиоэлектроники.
634050, г. Томск, пр. Ленина, 40
Тел. (3822) 533018.