КОМПЬЮТЕРНЫЕ НАУКИ И ТЕХНОЛОГИИ

Министерство образования и науки Украины
Донецкий национальный технический университет

Центр компетентности в области компьютерных наук и технологий

Алексеев Е.Р., Чеснокова О.В., Кучер Т.В.

Самоучитель по программированию на Free Pascal и Lazarus

УНИТЕХ
ДОНЕЦК 2009

УДК 004.43

ISBN 978-966-8248-26-9

Рецензенты: *Аноприенко А.Я.* — кандидат технических наук, профессор, декан факультета компьютерных наук и технологий Донецкого национального технического университета.
Кононов Ю.Н. — доктор физико-математических наук, профессор кафедры прикладной механики и компьютерных технологий Донецкого национального университета.

А 47 **Алексеев Е.Р., Чеснокова О.В., Кучер Т.В.** Самоучитель по программированию на Free Pascal и Lazarus. - Донецк.: ДонНТУ, Технопарк ДонНТУ УНИТЕХ, 2009. - 503 с.

Печатается по решению Ученого совета факультета компьютерных наук и технологий Донецкого национального технического университета, протокол №7 от 30 октября 2009 г.

Ответственный за выпуск: заведующий кафедрой вычислительной математики и программирования Донецкого национального технического университета, доктор технических наук, профессор Павлыш В.Н.

Книга является учебником по алгоритмизации и программированию. В учебнике описан язык Free Pascal и среда визуального программирования Lazarus. Free Pascal и Lazarus являются мощными и свободно распространяемыми средствами программирования. В книге приведено большое количество примеров алгоритмов и программ. Особое внимание уделено работе с визуальными компонентами, их свойствами и методами. Подробно описаны такие этапы программирования как работа с подпрограммами и файлами. Также в книге можно познакомиться с основами объектно-ориентированного программирования и графическими средствами Lazarus. Книга содержит 25 вариантов заданий для самостоятельного решения по всем рассматриваемым темам.

Издание предназначено для школьников, студентов, аспирантов и преподавателей, а также для всех, кто изучает программирование на Free Pascal и Lazarus.

ISBN 978-966-8248-26-9 **ООО «**Технопарк ДонНТУ УНИТЕХ**»**

Содержание

Введение

Авторы книги давно хотели написать учебник по программированию, предназначенный для пользователей различных операционных систем.

Учебник основан на курсе, который авторы читали в Донецком национальном техническом университете (ДонНТУ) студентам общеинженерных специальностей. Многие годы в ДонНТУ в качестве языка обучения программированию будущих инженеров используется Pascal, который является ясным, логичным и гибким языком и приучает к хорошему стилю программирования. Кроме того, в средней школе основы программирования преподают именно на базе Pascal. Вместе с тем именно Pascal лежит в основе современной мощной системы визуального программирования Delphi, с помощью которой разрабатываются многие современные программные продукты.

В учебнике используется язык программирования Free Pascal, компиляторы с которого являются свободно распространяемыми. Free Pascal является очень мощным средством программирования, и вместе с тем за использование компиляторов студенту, школьнику и преподавателю не придется платить. Этим компилятором можно пользоваться абсолютно легально.

Свободно распространяемые компиляторы с языка Free Pascal реализованы во многих дистрибутивах Linux, есть свободные компиляторы и для ОС Windows. Кроме того, в этой книге мы попытались познакомить читателя с принципами создания визуальных приложений в среде Lazarus. Бурно развивающаяся среда визуального программирования Lazarus является серьезным конкурентом Delphi.

В настоящее время существует множество подходов к изучению программирования. По мнению авторов, нельзя изучать программирование на каком-либо языке, не изучив методы разработки алгоритмов. Как свидетельствует опыт авторов, одним из наиболее наглядных методов составления алгоритмов является язык *блок-схем*. Мы попытались написать учебник по алгоритмизации и программированию. Насколько нам это удалось судить читателю.

Авторы надеются, что читатель имеет первоначальные навыки работы на персональном компьютере под управлением ОС Linux или Windows и знаком со школьным курсом математики.

Книга состоит из десяти глав.

В *первой главе* читатель узнает о средствах разработки программ на Free Pascal, напишет свои первые программы.

Во *второй* главе изложены основные элементы языка (переменные, выражения, операторы) Free Pascal. Описаны простейшие операторы языка: присваивания и ввода-вывода. Приведена структура программы на Free Pascal, а также примеры программ линейной структуры.

Третья глава является одной из ключевых в изучении программирования. В ней изложена методика составления алгоритмов с помощью блок-схем. Приведено большое количество примеров алгоритмов и программ различной сложности. Авторы рекомендуют внимательно разобрать все примеры и выполнить упражнения этой главы и только после этого приступать к изучению последующих глав книги.

В *четвертой* главе читатель на большом количестве примеров познакомится с подпрограммами. Описан механизм передачи параметров между подпрограммами. Один из параграфов посвящен рекурсивным подпрограммам. В завершении главы рассмотрен вопрос создания личных модулей.

Пятая и *шестая* главы посвящены алгоритмам обработки массивов и матриц. Здесь же читатель познакомится и с реализацией этих алгоритмов на языке Free Pascal. Именно эти главы совместно с третьей являются ключом к пониманию принципов программирования.

Седьмая глава знакомит читателя с обработкой файлов на языке Free Pascal под управлением операционных систем Linux и Windows. На практических примерах изложен механизм прямого и последовательного доступа к файлам и обработки ошибок ввода-вывода. Описана работа с бестиповыми и текстовыми файлами.

Восьмая глава посвящена обработке строк и записей. Приведенные примеры позволят читателю разобраться с принципами обработки таблиц в языке Free Pascal.

В *девятой* главе авторы описали принципы объектно-ориентированного программирования и их реализацию в языке Free Pascal.

В *десятой* главе описаны графические возможности Lazarus, подробно описан алгоритм построения графиков непрерывных функций на экране дисплея. Приведены тексты программ изображения графиков функций с подробными комментариями.

К каждой теме прилагаются 25 вариантов задач для самостоятельного решения, что позволит использовать книгу не только начинающим самостоятельно изучать программирование, но и преподавателям в учебном процессе.

С рабочими материалами книги можно познакомиться на сайтах Евгения Ростиславовича Алексеева — www.teacher.dn-ua.com, www.teacher.ucoz.net.

Авторы выражают благодарность своим родным за помощь и понимание.

Алексеев Е.Р., Чеснокова О.В., Кучер Т.В.
Донецк, ноябрь 2009 г.

Сведения об авторах

Алексеев Евгений Ростиславович — кандидат технических наук, доцент кафедры «Вычислительная математика и программирование» Донецкого национального технического университета, доцент кафедры «Документоведение и информационная деятельность» Донецкого инженерно-экономического колледжа. Преподавательский стаж Алексеева Е.Р. — 17 лет. Евгений Ростиславович — автор двенадцати книг, вышедших в Москве и Донецке, общий тираж которых превышает 65 тыс. экземпляров. Е.Р. Алексеев — автор более 60 научных и методических работ. Область его научных интересов — программирование, вычислительная математика, Интернет-технологии, использование свободно распространяемого программного обеспечения.

Чеснокова Оксана Витальевна — старший преподаватель кафедры «Вычислительная математика и программирование» Донецкого национального технического университета. Преподавательский стаж Чесноковой О.В. — 15 лет. Оксана Витальевна — автор девяти книг, общий тираж которых превышает 40 тыс. экземпляров, а также — около 40 научных и методических работ. Область ее научных интересов — программирование, вычислительная математика.

Кучер Татьяна Викторовна — ассистент кафедры «Вычислительная математика и программирование» Донецкого национального технического университета, ассистент кафедры «Документоведение и информационная деятельность» Донецкого инженерно-экономического колледжа. Татьяна Викторовна — автор 30 научных и методических работ. Преподавательский стаж Кучер Т. В. — 14 лет. Область ее научных интересов — программирование, вычислительная математика.

1 Средства разработки программ на языке Free Pascal

В этой главе мы начинаем знакомство с программированием на языке Free Pascal. *Язык программирования* Free Pascal ведет свое начало от классического языка Pascal, который был разработан в конце 60-х годов XX века Николасом Виртом. Он разрабатывал этот язык как учебный язык для своих студентов. С тех пор Pascal, сохранив простоту и структуру языка, разработанного Н. Виртом, превратился в мощное средство программирования. С помощью современного языка Pascal можно производить простые расчеты, разрабатывать программы для проведения сложных инженерных и экономических вычислений.

1.1 Процесс разработки программы

Разработку программы можно разбить на следующие этапы:

1. Составление алгоритма решения задачи. *Алгоритм* — это описание последовательности действий, которые необходимо выполнить для решения поставленной задачи.

2. Написание текста программы. *Текст программы* пишут на каком-либо языке программирования (например на Free Pascal) и вводят его в компьютер с помощью текстового редактора.

3. Отладка программы. *Отладка программы* — это процесс устранения ошибок из текста программы. Все ошибки делятся на синтаксические и логические. При наличии синтаксических ошибок (ошибок в написании операторов) программа не запускается. Подобные ошибки исправляются проще всего. Логические ошибки — это ошибки, при которых программа работает, но неправильно, выдавая не те результаты, которые ожидает разработчик или пользователь. Логические ошибки исправить сложнее, чем синтаксические, иногда для этого приходится переписывать отдельные участки программы, а иногда перерабатывать весь алгоритм.

4. Тестирование программы. *Тестирование программы* — процесс выявления ошибок в ее работе.

Процессы отладки и тестирования сопровождаются неоднократным *запуском программы на выполнение*. Процесс запуска может быть осуществлен только после того, как введенная в компьютер про-

грамма на алгоритмическом языке Pascal[1] будет переведена в *двоичный машинный код* и создан *исполняемый файл*.

Процесс перевода текста программы в машинный код называют *трансляцией*. Все трансляторы делятся на два класса:

- *интерпретаторы* — трансляторы, которые переводят каждый оператор программы в машинный код и по мере перевода операторы выполняются процессором;
- *компиляторы* переводят всю программу целиком, и если этот перевод прошел без ошибок, то полученный двоичный код можно запускать на выполнение.

Если в качестве транслятора выступает компилятор, то процесс перевода текста программы в машинный код называют *компиляцией*. При переводе программы с языка Pascal в машинный код используются именно компиляторы[2].

Рассмотрим основные этапы обработки компилятором программы на языке Pascal.

1. Компилятор анализирует, какие внешние библиотеки[3] нужно подключить, разбирает текст программы на составляющие элементы, проверяет синтаксические ошибки и в случае их отсутствия формирует объектный код (в Windows — файл с расширением **.obj**, в Linux — файл с расширением **.o**). Получаемый на этом этапе двоичный файл (объектный код) не включает в себя объектные коды подключаемых библиотек.

2. На втором этапе компоновщик подключает к объектному коду программы объектные коды библиотек и генерирует исполняемый код программы. Этот этап называется *компоновкой*, или *сборкой*, *программы*. Полученный на этом этапе исполняемый код можно запускать на выполнение.

На сегодняшний день существует множество компиляторов языка Pascal, среди которых можно выделить компиляторы языка Pascal разработки фирмы Borland (Borland Pascal), а также свободно распространяемые кроссплатформенные компиляторы языка Free Pascal и среду визуального программирования Lazarus.

---

1 Как и на любом другом языке.

2 Вместо термина «компилятор» в литературе иногда используют термин «транслятор компилирующего типа».

3 В библиотеках языка Pascal хранится объектный (двоичный) код стандартных (таких, как sin(x), cos(x) и др.) функций и процедур языка.

1.2 Среда программирования Free Pascal

Рассмотрим *процесс установки компилятора* Free Pascal в ОС Linux. Для установки программ в операционной системе Linux служит менеджер пакетов Synaptic. В ОС Ubuntu он вызывается с помощью команды **Система — Администрирование — Менеджер пакетов Synaptic**. В ALT Linux менеджер пакетов вызывается командой **Система — Программа управления пакетами Synaptic**. Окно Synaptic представлено на рис. 1.1.

Обратите внимание, что для установки программ необходимо установить список источников программ (список репозиториев[4]). После установки ОС Ubuntu Linux первоначальный список репозиториев установлен. В ОС ALT Linux первоначальный список репозиториев надо установить.

Для установки окне Synaptic (см. рис. 1.1) необходимо щелкнуть по кнопке **Найти**. И в открывшемся окне ввести **fpc** (см. рис. 1.2). Менеджер программ находит программу Free Pascal, после чего в окне Synaptic необходимо пометить программы *fpc* (Free Pascal Compiler Meta Package) для установки (с помощью контекстного меню или с помощью кнопки **Отметить для обновления**) и начать установку, щелкнув по кнопке **Применить**. После этого начнется процесс скачивания пакетов из Интернета и их установки.

В состав метапакета `fpc` входит компилятор языка Free Pascal `fpc` и среда разработки `fp-ide`. Для запуска среды разработки в Linux необходимо просто в терминале набрать `fp`. На рис. 1.3 представлено окно среды разработки программ на языке Free Pascal в ОС Linux.

Для установки Free Pascal в операционной системе Windows необходимо запустить скачанный со страницы загрузки http://www.-freepascal.org/down/i386/win32.var инсталяционный файл. Первое диалоговое окно сообщит о начале процесса установки Free Pascal на компьютер. Для продолжения процесса установки во всех следующих окнах нужно выбирать кнопку **Next**, для возврата к предыдущему шагу — кнопку **Back**, а для прерывания процесса установки — кнопку **Cancel**. В следующем окне нужно определить путь для установки Free Pascal. По умолчанию установка происходит в корневой каталог диска C. Для выбора другого пути установки можно воспользоваться

4 Список репозиториев — список официальных сайтов, с которых можно устанавливать программы.

кнопкой **Browse...**. Кроме того, в этом окне выводится информация о количестве свободного места на диске. В следующих четырех окнах пользователь сможет выбрать из списка тип установки: **Full Installation** (полная), **Minimum Installation** (минимальная), **Custom Installation** (выбор компонентов), указать название устанавливаемого приложения в главном меню, выбрать типы файлов, поддерживаемых средой, и начать процесс инсталляции Free Pascal, нажав кнопку **Install**. Контролировать процесс установки можно с помощью линейного индикатора, а прервать кнопкой **Cancel**.

Запуск среды программирования Free Pascal в Windows можно осуществить из главного меню: **Пуск — Программы — Free Pascal — Free Pascal**. На экране появится окно, представленное на рис. 1.4.

Установив пакет Free Pascal, мы получили компилятор и среду программирования.

Компилятор Free Pascal работает в командной строке. Для того чтобы создать исполняемый файл из текста программы, написанного на языке Pascal, необходимо выполнить команду

```
fpc name.pas
```

здесь `fpc` — имя исполняемого файла компилятора командной строки Free Pascal, `name.pas` — имя файла с текстом программы. В результате в Linux будет создан исполняемый файл с именем `name` (в Windows — имя исполняемого файла `name.exe`).

При использовании компилятора fpc после компиляции автоматически происходит компоновка программы (запуск компоновщика make).

Технология работы с компилятором Free Pascal может быть такой: набираем текст программы в стандартном текстовом редакторе, затем в консоли запускаем компилятор, после исправления синтаксических ошибок повторно запускаем компилятор. После успешной компиляции запускаем исполняемый файл. При необходимости опять вносим изменения в текст программы и запускаем компилятор. При такой технологии работы с компилятором необходимо не забывать сохранять текст программы, иначе при запуске компилятора будет компилироваться старая версия текста программы.

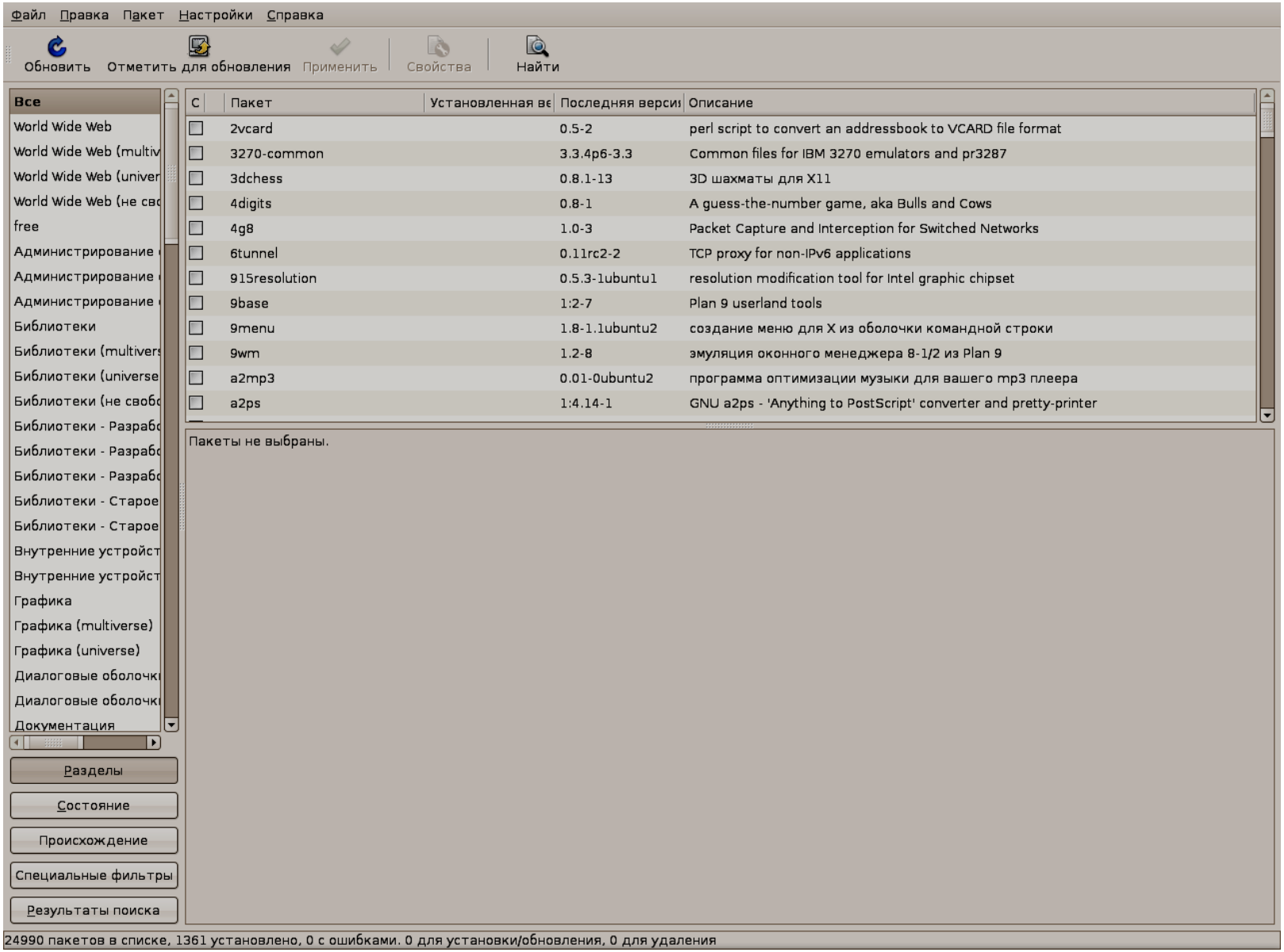

Рисунок 1.1: Менеджер пакетов Synaptic

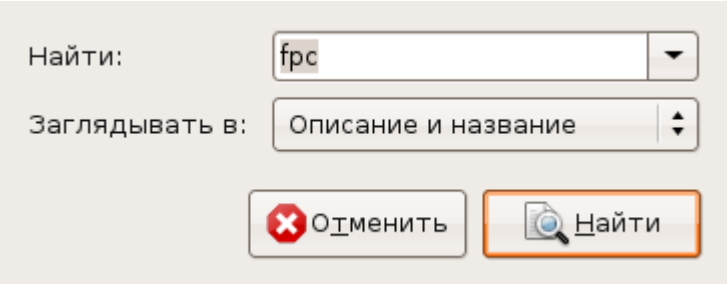

Рисунок 1.2: Окно поиска компилятора Free Pascal в Synaptic

Рисунок 1.3: Среда программирования Free Pascal в ОС Linux

Рисунок 1.4: Окно компилятора Free Pascal

Среда программирования Free Pascal позволяет значительно упростить процесс разработки программ. В состав среды программирования входит текстовый редактор, транслятор и отладчик. Рассмотрим их работу подробнее.

1.2.1 Работа в текстовом редакторе Free Pascal

С помощью редактора Free Pascal можно создавать и редактировать тексты программ. После открытия пустого окна (**File — New**) или загрузки текста программы (**File — Open**) пользователь находится в *режиме редактирования*, признаком чего является наличие в окне *курсора* (небольшого мигающего прямоугольника). Для перехода из режима редактирования к главному меню нужно нажать клавишу **F10**, обратно — **Esc**. Кроме того, этот переход можно осуществить щелчком мыши либо по строке главного меню, либо по полю редактора.

Редактор Free Pascal обладает возможностями, характерными для большинства текстовых редакторов. Остановимся на некоторых особенностях.

Работа с фрагментами текста (блоками) в редакторе Free Pascal может осуществляться с помощью главного меню и функциональных клавиш.

В главном меню для работы с фрагментами текста предназначены команды *пункта редактирования* **Edit**:

Copy (Ctrl+C) — копировать фрагмент в буфер;
Cut (Ctrl+X) — вырезать фрагмент в буфер;
Paste (Ctrl+V) — вставить фрагмент из буфера;
Clear (Ctrl+Del) — очистить буфер;
Select All — выделить весь текст в окне;
Unselect — отменить выделение.

Команды **Copy** и **Cut** применяют только к выделенным фрагментам текста. *Выделить фрагмент текста* можно с помощью клавиши **Shift** и клавиш перемещения курсора (стрелок). Кроме того, пункт меню **Edit** содержит команды **Undo** и **Redo,** с помощью которых можно отменять и возвращать выполненные действия.

Комбинации клавиш, предназначенные для работы с блоком:

Ctrl+K+B – пометить начало блока;
Ctrl+K+K – пометить конец блока;
Ctrl+K+T – пометить в качестве блока слово слева от курсора;
Ctrl+K+Y – стереть блок;

Ctrl+K+C – копировать блок в позицию, где находится курсор;
Ctrl+K+V – переместить блок в позицию, где находится курсор;
Ctrl+K+W – записать блок в файл;
Ctrl+K+R – прочитать блок из файла;
Ctrl+K+P – напечатать блок;
Ctrl+K+H – снять пометку блока; повторное использование этой комбинации клавиш вновь выделит блок.

Работа с файлами в среде Free Pascal осуществляется с помощью команд **File** главного меню и функциональных клавиш:

New — открыть окно для создания новой программы;
Open (F3) — открыть ранее созданный файл;
Save (F2) — сохранить созданный файл;
Save As — сохранить файл под другим именем;
Exit (Alt+X) — выйти из среды программирования;

При *создании новой программы* ей по умолчанию присваивается стандартное имя NONAMEOO.PAS (NO NAME — нет имени).

При *первом сохранении файла* пользователю будет предложено ввести его имя. При *повторном сохранении* файл сохраняется под тем же именем. Команда **Save As** аналогична первому сохранению. Если файл не был сохранен, то при попытке завершить работу со средой появится запрос о необходимости сохранить изменения в файле. При *открытии ранее созданного файла* его имя выбирают из списка существующих файлов.

В редакторе Free Pascal допускается *работа с несколькими окнами*. Для переключения в окно с номером от первого до девятого нажать комбинацию клавиш **Alt+i**, где i – номер окна (например **Alt+5** — вызов пятого окна). Для вывода списка окон на экран нажать комбинацию клавиш **Alt+0**, появится с*писок активных окон*, в котором необходимо будет выбрать нужное и нажать **Enter**.

1.2.2 Запуск программы в среде Free Pascal и просмотр результатов

После того как текст программы был набран, его следует перевести в машинный код. Для этого необходимо *вызвать транслятор* с помощью команды **Compile — Compile** (комбинация клавиш **Alt+F9**). На первом этапе транслятор проверяет наличие синтаксических ошибок. Если в программе нет синтаксических ошибок, то на экране сообщается о количестве строк транслированной программы и объеме доступной оперативной памяти.

Если на каком-либо этапе транслятор обнаружит ошибку, то в окне редактора курсор указывает ту строку программы, где при трансляции обнаружена ошибка. При этом в верхней строке редактора появляется краткое диагностическое сообщение о причине ошибки.

Для *запуска транслированной программы* необходимо выполнить команду **Run — Run** (комбинация клавиш **Ctrl+F9**), после чего на экране появляется окно командной строки, в котором пользователь и осуществляет диалог с программой. После завершения работы программы пользователь опять видит экран среды Free Pascal.

Для *просмотра результатов работы программы* в ОС Windows необходимо нажать комбинацию клавиш **Alt+F5**. Для возврата в оболочку следует нажать любую клавишу.

При использовании среды программирования в ОС Linux существует проблема вывода кириллического текста в консольном приложении. Альтернативой среды программирования Free Pascal является текстовый редактор Geany (http://www.geany.org), который позволяет разрабатывать программы на различных языках программирования.

1.3 Текстовый редактор Geany

Текстовый редактор Geany есть в репозитарии большинства современных дистрибутивов Linux[5]. Разработка программ с использованием Geany довольна удобна. Установка Geany также может быть осуществлена с помощью менеджера пакетов Synaptic.

Последовательно рассмотрим основные этапы разработки программы с использованием Geany.

1. Необходимо *создать шаблон* приложения на Pascal (или другом языке программирования) с помощью команды **Файл — New (with Template) — Pascal source file**. После чего появится окно с шаблоном исходного кода (рис. 1.5), в котором необходимо ввести текст программы и сохранить его (рис. 1.6).

2. Для *компиляции* и *запуска программы* на выполнение служит пункт меню **Построить**. Для компиляции программы следует использовать команду **Построить — Собрать** (**F8**). В этом случае будут созданы файл с объектным кодом программы и исполняемый файл. После компиляции программы ниже окна с программой будет представлен подробный отчет (рис. 1.6) о результатах компиляции. Следует его внимательно изучить. Этот отчет позволит быстро исправлять

5 Существует версия Geany и для Windows (http://www.geany.org/Download/Releases)

синтаксические ошибки. В сообщении об ошибке указана строка, где она найдена и ее краткое описание на английском языке. В отчетах по результатам компиляции могут быть *ошибки* (*error*) и *сообщения* (*warning*). Сообщения — это обнаруженные компилятором неточности, при которых возможно создание исполняемого кода программы.

3. Для *запуска программы* следует выполнить команду **Построить — Выполнить (F5)**. После чего на экране появляется окно терминала (рис. 1.7), в котором можно вводить данные и увидеть результаты работы программы.

В редакторе `Geany` (хотя часто его называют и средой программирования) можно настроить команду вызова компиляции, компоновки и запуск. Для это служит команда **Построить — Установить включения и аргументы**. Это окно для работы с файлами с расширением pas представлено на рис. 1.8. При настройке строк **Compile** и **Запуск** следует учитывать, что `%f` — имя компилируемого файла, `%e` — имя файла без расширения.

Какую среду выбрать для разработки консольных программ на Free Pascal — это дело пользователя. Авторы советуют под управлением ОС Linux использовать `Geany`[6]. Хотя можно использовать для набора текста программы обычный текстовый редактор (например, `gedit`, `tea`, `kate` и др.), а компиляцию осуществлять в терминале. Под управлением Windows логичнее использовать `fp-ide`.

1.4 Среда визуального программирования Lazarus

Lazarus – это *среда визуального программирования*. Здесь программист получает возможность не просто создавать программный код, но и наглядно (визуально) показывать системе, что бы он хотел увидеть.

Технология визуального программирования позволяет строить интерфейс[7] будущей программы из специальных компонентов, реализующих нужные свойства. Количество таких компонентов достаточно велико. Каждый из них содержит готовый программный код и все необходимые для работы данные, что избавляет программиста от создания того, что уже создано ранее.

6 Это субъективный совет авторов.

7 Интерфейс – диалог, обмен информацией.

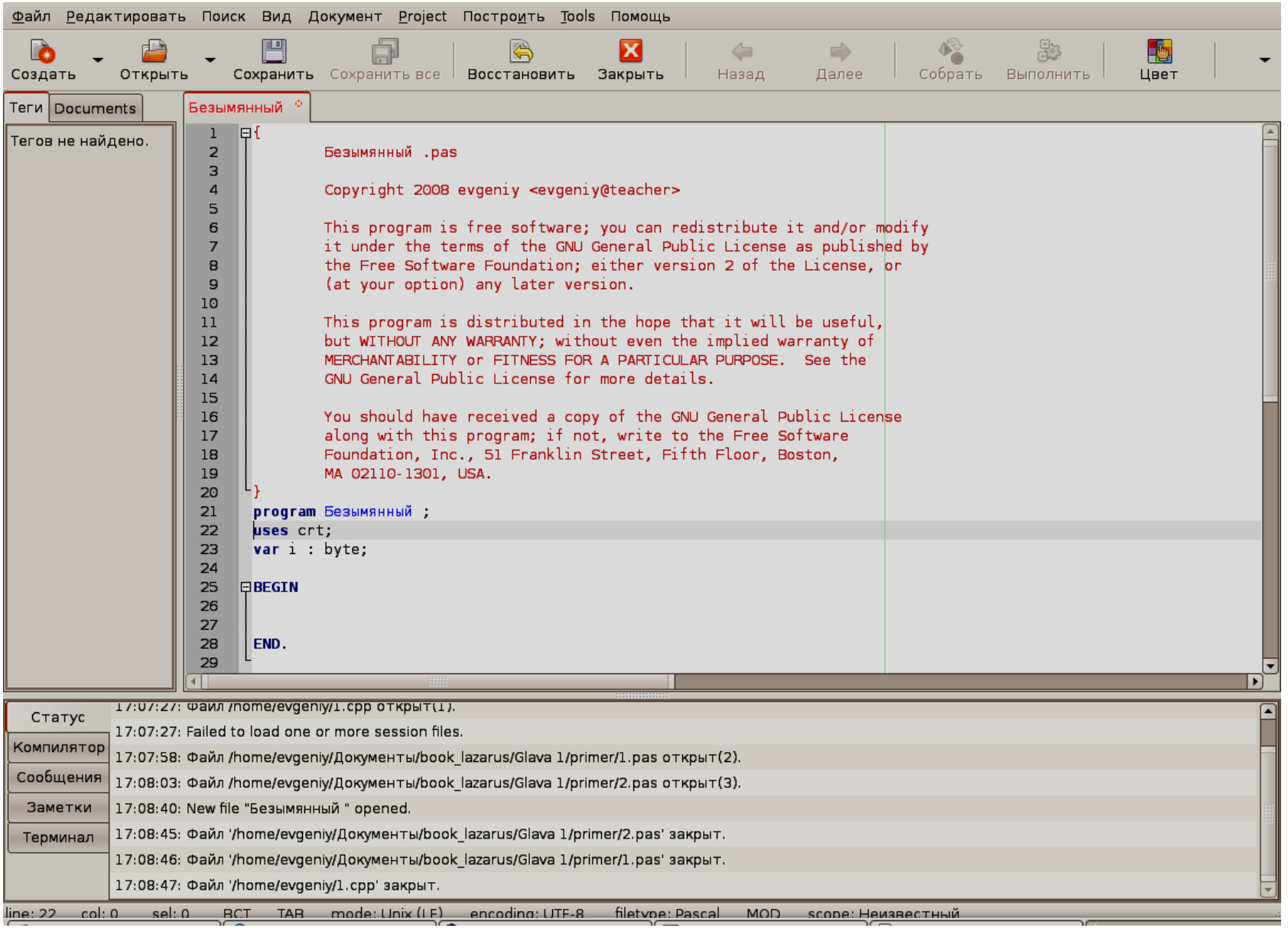

Рисунок 1.5: Окно Geany с шаблоном программы на Free Pascal

Подобный подход во много раз сокращает время написания программы. Кроме того, быстрота создания программного кода в Lazarus достигается за счет того, что значительная часть текста формируется автоматически.

Среда визуального программирования Lazarus сочетает в себе компилятор, объектно-ориентированные средства визуального программирования и различные технологии, облегчающие и ускоряющие создание программы.

1.4.1 Установка Lazarus в ОС Linux

Для установки Lazarus в окне Synaptic (см. рис. 1.1) необходимо щелкнуть по кнопке **Найти**. В появившемся окне поиска (см. рис. 1.9) вводим имена необходимых программ (*Lazarus, fpc, fpc-source*) и щелкаем по кнопке **Найти**.

Менеджер программ находит программы Lazarus и Free Pascal, после чего в *Lazarus, fpc, fpc-source* для установки (с помощью контекстного меню или с помощью кнопки **Отметить для обновления**) и начинает установку, щелкнув по кнопке **Применить**. После этого Synaptic предложит установить еще несколько пакетов, которые необходимы для установки Lazarus. Надо соглашаться. После этого начнется процесс скачивания файлов пакетов и установки Lazarus на компьютер. После установки запуск программы осуществляется с помощью команды меню **Программирование — Lazarus**[8].

Можно начинать работать. В старых версиях операционной системы Linux (например, Ubuntu 8.10 и более ранних) при запуске Lazarus вместо русских пунктов меню появятся непонятные символы.

Подробно о том, как добиться правильного отображения символов кириллицы в меню Lazarus, описано на следующих страницах:

http://www.freepascal.ru/article//lazarus/20080316091540/,
http://forum.sources.ru/index.php?showtopic=243159,
http://forum.ubuntu.ru/index.php?topic=18539.0;all.

Кроме того, можно использовать и запуск с английским интерфейсом командой терминала `LANG=C startlazarus`.

Но, на взгляд авторов, наиболее универсальным и простым методом добиться корректного отображения символов кириллицы будет следующий.

8 Не исключено, что вызов Lazarus в других дистрибутивах Linux можно осуществлять и с помощью другой команды главного меню.

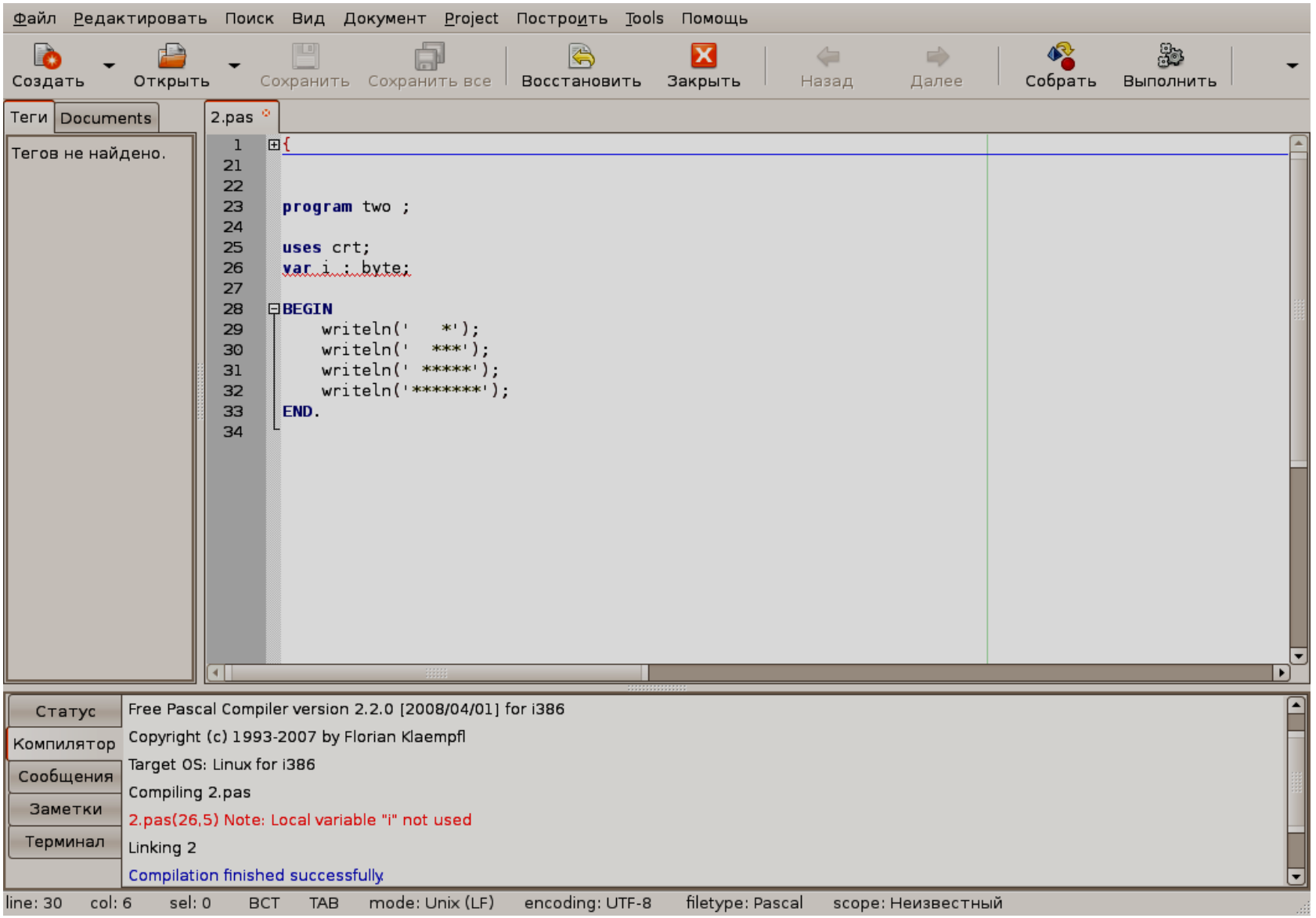

Рисунок 1.6: Окно Geany с текстом программы на языке Free Pascal

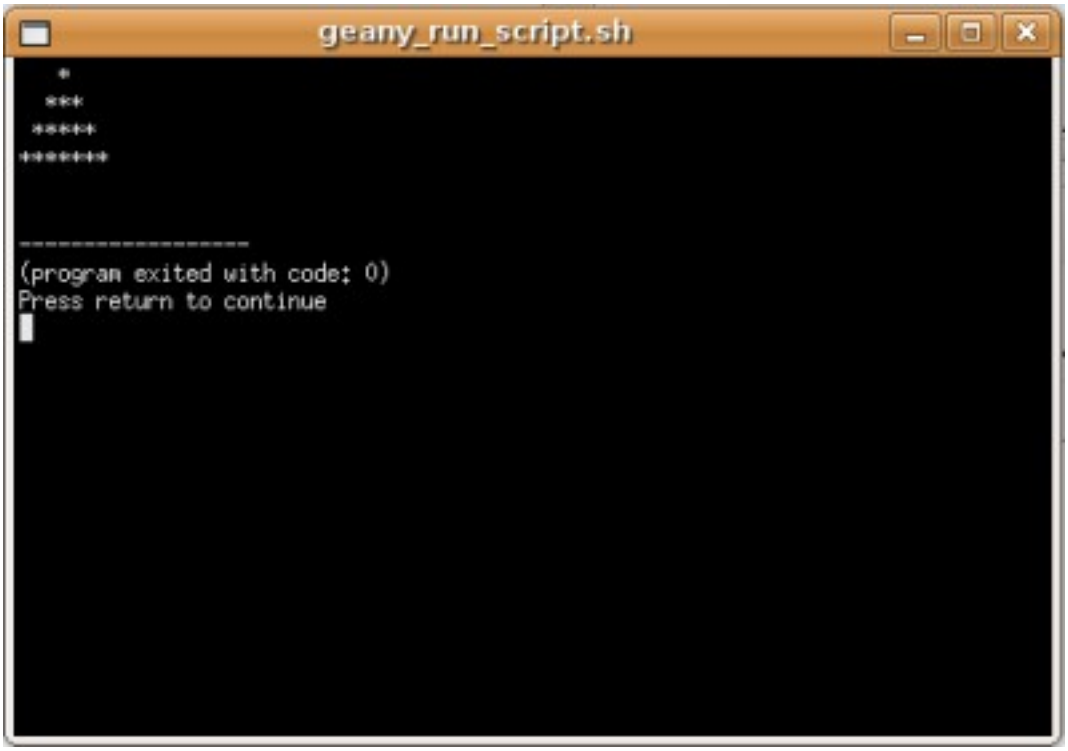

Рисунок 1.7: Окно терминала с результатами работы программы

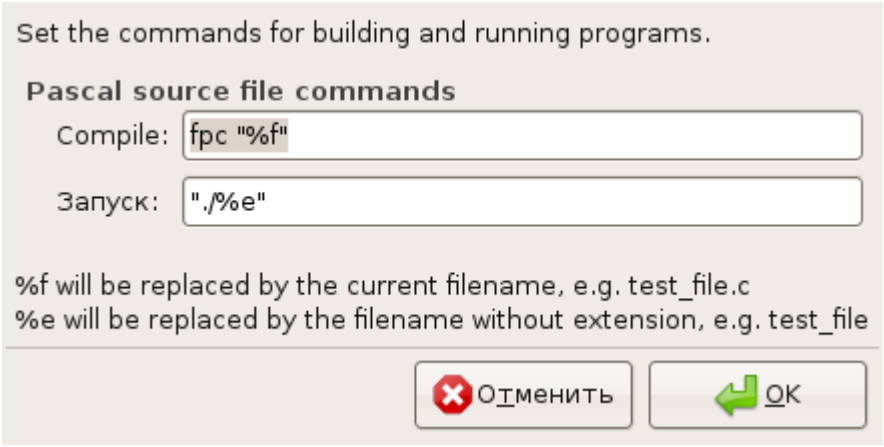

*Рисунок 1.8: Окно **Установить включения и аргументы** для Free Pascal*

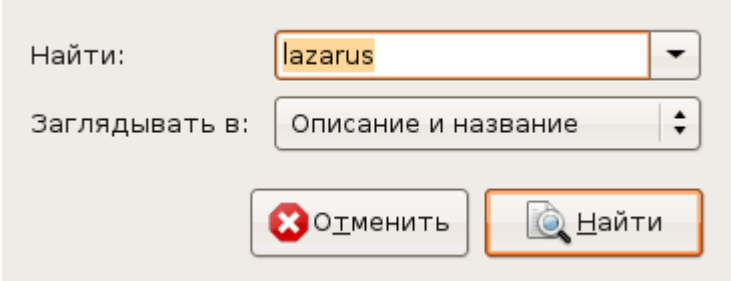

Рисунок 1.9: Окно поиска пакета Lazarus для установки

1. Запускаем Lazarus с английским интерфейсом с правами администратора[9].

2. Выполняем команду **Tools — Conigure «Build — Lazarus»...** . В открывшемся окне **Conigure «Build — Lazarus»** на вкладке Quick Build Options устанавливаем следующие параметры компиляции **Build Options — Build All**, **LCL interface — IDE — gtk2 (betta)**, после чего щелкаем по кнопке Build и ждем, пока произойдет перекомпиляция среды Lazarus с новыми параметрами.

3. После этого запускаем Lazarus из меню (или из терминала командой startlazarus) и видим корректную русскоязычную среду Lazarus.

Конечно, для установки Lazarus можно не использовать менеджер пакетов Synaptic, а самостоятельно скачать все необходимые пакеты с сайта http://sourceforge.net/project/showfiles.php?group_id=89339 и затем их вручную установить. Подробно процесс ручной установки можно найти в Интернете, например на странице http://freepascal.ru/article//lazarus/20080316091540.

1.4.2 Установка Lazarus под управлением ОС Windows

Рассмотрим особенности установки среды визуального программирования Lazarus для операционной системы Windows. Перед установкой необходимо скачать установочный файл со страницы загрузки http://sourceforge.net/project/showfiles.php?group_id=89339. Установка Lazarus на компьютер осуществляется с помощью ряда окон Мастера установки. Для того чтобы Мастер установки начал свою работу, необходимо запустить программу установки Lazarus Setup. Появится диалоговое окно (рис. 1.10), в котором пользователь может выбрать из списка язык для дальнейшего диалога с Мастером установки. Нажатие кнопки **ОК** приведет к появлению следующего окна[10].

Следующее окно (рис. 1.11) информационное. Оно сообщает пользователю о начале процесса установки. Здесь кнопка **Далее** приведет к следующему шагу Мастера установки, кнопка **Отмена** прервет процесс установки приложения.

---

9 В Ubuntu команда будет такой `sudo LANG=C startlazarus`

10 В этом и во всех последующих окнах использование кнопки **Отмена** приведет к прерыванию работы Мастера установки.

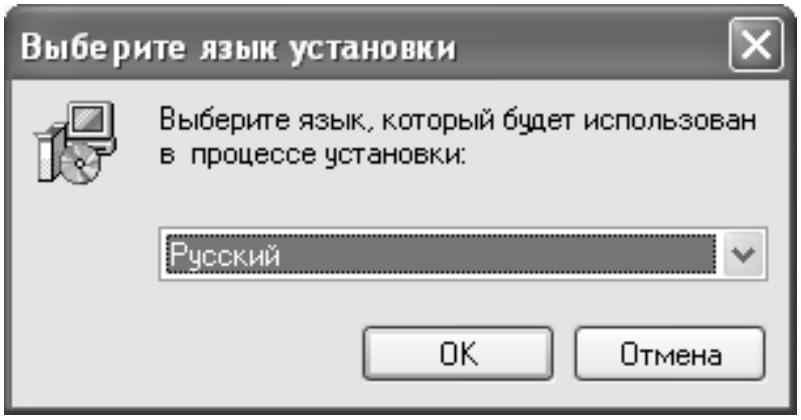

Рисунок 1.10: Окно Мастера установки. Выбор языка.

На следующем этапе (рис. 1.12) необходимо выбрать путь для установки Lazarus. По умолчанию программа будет установлена на диск C. Для выбора иного пути для установки следует воспользоваться кнопкой **Обзор**. Щелчок по кнопке **Далее** приведет к следующему шагу процесса установки[11].

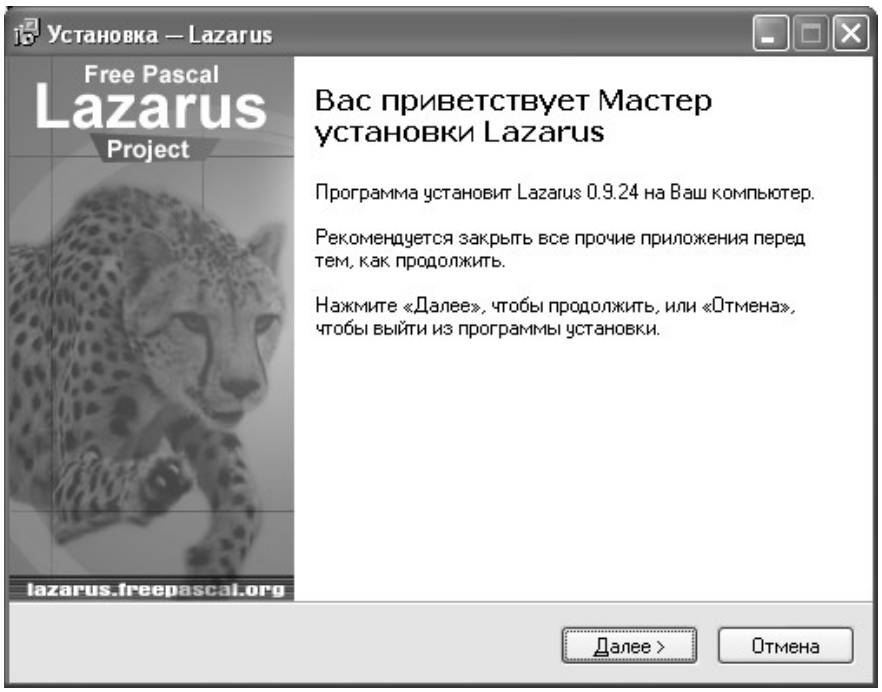

Рисунок 1.11: Начало процесса установки Lazarus

11 В этом и во всех последующих окнах с помощью кнопки **Назад** можно вернуться к предыдущему шагу Мастера установки.

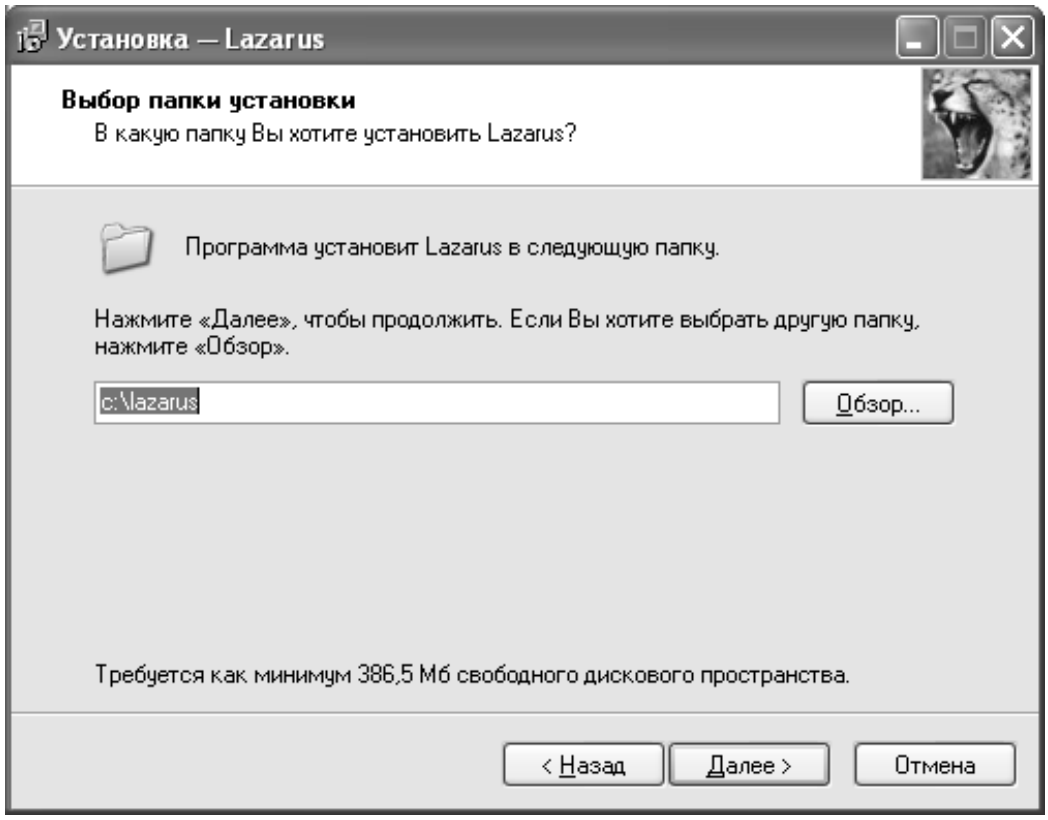

Рисунок 1.12: Окно выбора пути для установки Lazarus

Следующий шаг – это выбор необходимых компонентов из полного перечня средств системы (рис. 1.13). По умолчанию будут установлены все компоненты системы. Отменить установку компонента можно, если щелчком мыши убрать метку рядом с его именем. Кнопка **Далее** продолжает процесс установки. В следующих окнах Мастера установки пользователь сможет выбрать папку меню **Пуск**, в которой будет создан ярлык для устанавливаемого приложения[12], и создать на рабочем столе ярлык для приложения Lazarus[13].

Далее Мастер установки сообщит о том, куда будет установлено приложение Lazarus, каков тип установки, какие компоненты системы были выбраны, где будет создан ярлык и будет ли создан значок на рабочем столе. После этого начнется процесс установки приложения на компьютер. Контролировать инсталляцию приложения пользователь может с помощью линейного индикатора, а прервать – кнопкой **Отмена**. Завершается процесс установки щелчком по кнопке **Завершить**. Запуск Lazarus можно осуществить из главного меню командой **Пуск — Программы — Lazarus — Lazarus.**

12 По умолчанию ярлык создается в меню **Пуск — Программы.**

13 С помощью щелчка мыши установить метку в поле рядом с командой **Создать значок на рабочем столе**.

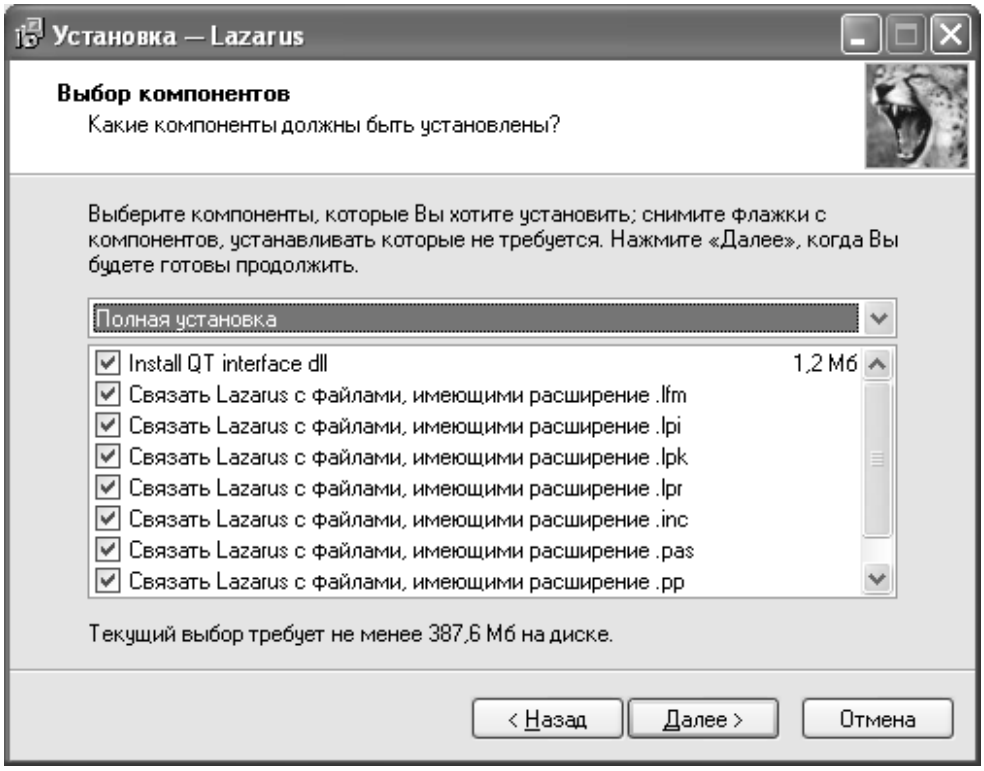

Рисунок 1.13: Окно выбора компонентов для установки приложения

1.4.3 Среда Lazarus

На рис. 1.15 показано окно, которое появится после запуска Lazarus. В верхней части этого окна размещается *главное меню* и *панель инструментов*. Слева расположено окно *инспектора объектов*, а справа окно *редактора исходного кода*. Если свернуть или сдвинуть окно редактора, то станет доступным *окно формы*, представленное на рис. 1.14.

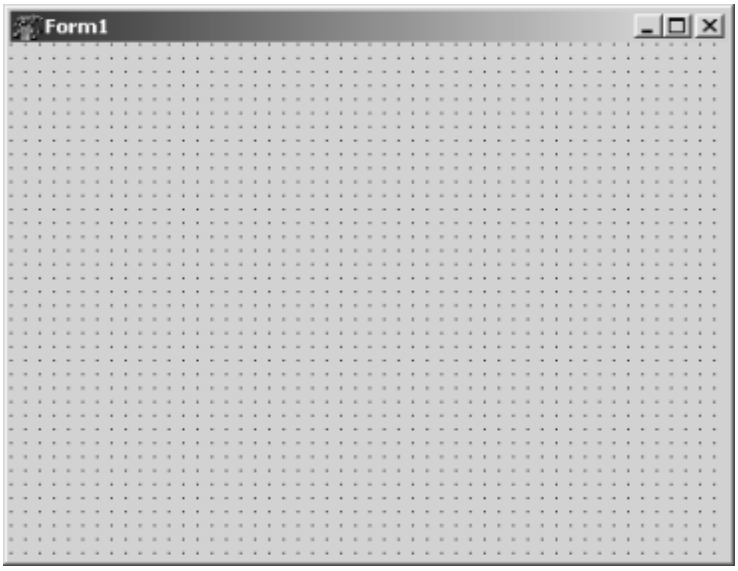

Рисунок 1.14: Окно формы

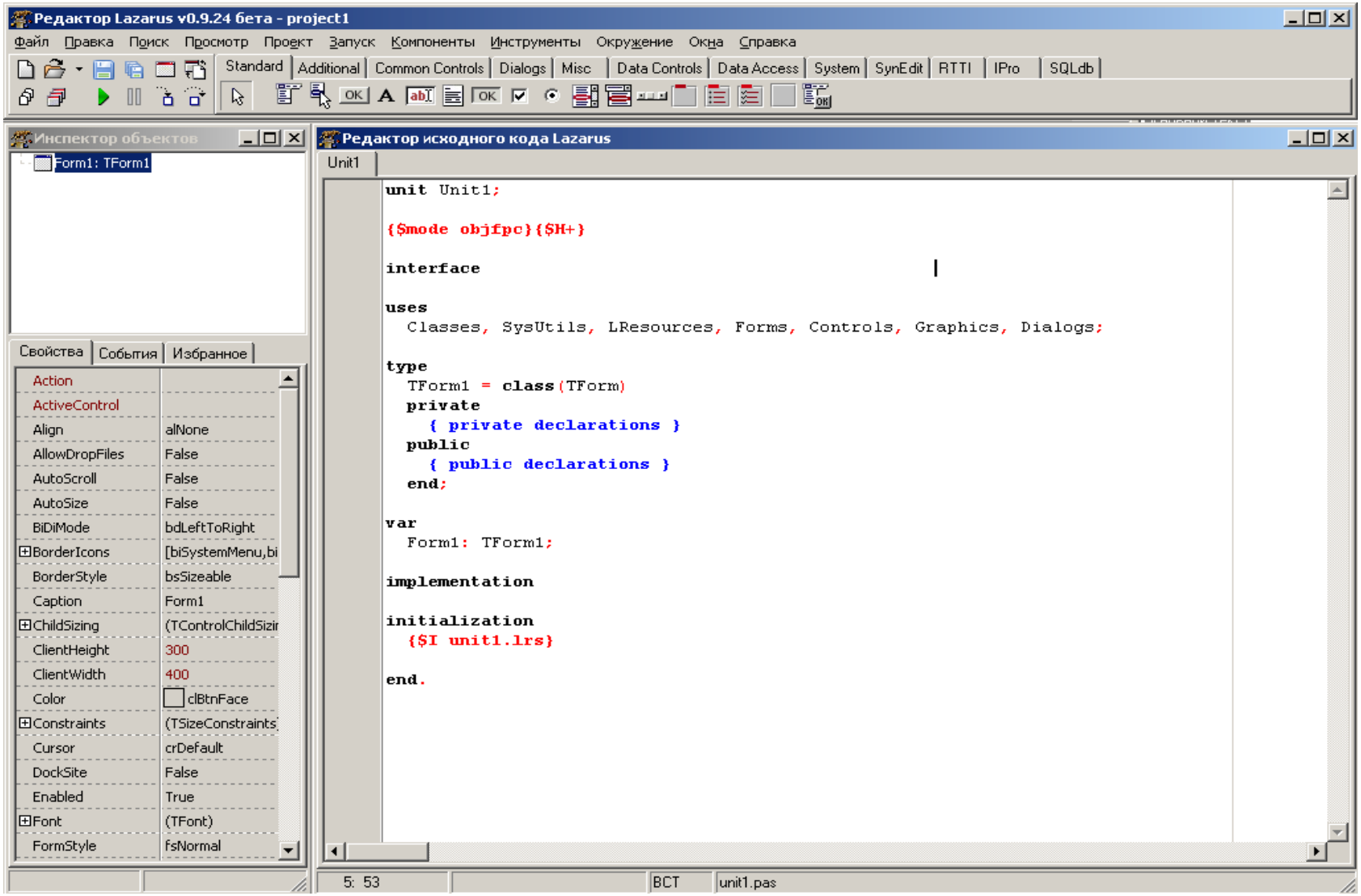

Рисунок 1.15: Среда визуального программирования Lazarus

Работу над программой в среде визуального программирования условно можно разбить на две части. Первая это создание внешнего вида (интерфейса) будущей программы, вторая — написание программного кода. Итак, *инспектора объектов* и *окно формы* нужны для создания интерфейса программы, а *редактор исходного кода* — для работы с ее текстом. Файлы, из которых в результате получается программа, называют *проектом*.

1.4.4 Главное меню Lazarus

Все команды, необходимые для работы в среде визуального программирования Lazarus, содержатся в *главном меню*. Доступ к командам главного меню осуществляется одинарным щелчком левой кнопкой мыши.

Работа с файлами в среде Lazarus осуществляется при помощи пункта меню **Файл**. Команды этого пункта меню можно разбить на группы:

- создание новых файлов – **Создать модуль**, **Создать форму**, **Создать...**;
- загрузка ранее созданных файлов – **Открыть**, **Открыть недавнее**, **Вернуть**;
- сохранение файлов – **Сохранить**, **Сохранить как…**, **Сохранить все**;
- закрытие файлов – **Закрыть**, **Закрыть все файлы редактора**;
- вывод на печать – **Печать…**;
- перезагрузка среды – **Перезапуск**;
- выход из среды – **Выход**.

Команды, предназначенные для *редактирования текста программного кода,* собраны в меню **Правка**. В основном это команды характерные для большинства текстовых редакторов:

- команды отмены или возврата последней операции – **Отменить**, **Вернуть**;
- команды переноса, копирования и вставки выделенного фрагмента текста в буфер – **Вырезать**, **Копировать**, **Вставить**;
- команды, работающие с выделенным блоком текста – **Сдвинуть блок вправо**, **Сдвинуть блок влево**;
- команды смены регистра – **Верхний регистр выделения**, **Нижний регистр выделения**;
- команды выделения фрагмента текста собраны в пункте меню **Выделить**.

Далее будут перечислены специфические команды редактора программного кода Lazarus.

Команда **Закомментировать** добавляет в начале каждой строки выделенного фрагмента два символа «косая черта», что превращает выделенный текст в *комментарий*[14], команда **Раскомментировать** выполняет обратное действие.

Команда **Заключить выделение в...** приводит к открытию диалогового окна (рис. 1.16), в этом окне пользователь может выбрать конструкцию языка программирования, в которую будет заключен выделенный фрагмент текста.

Команда **Сортировка выбранного...** открывает диалоговое окно, в котором можно установить параметры сортировки текста в выделенном фрагменте.

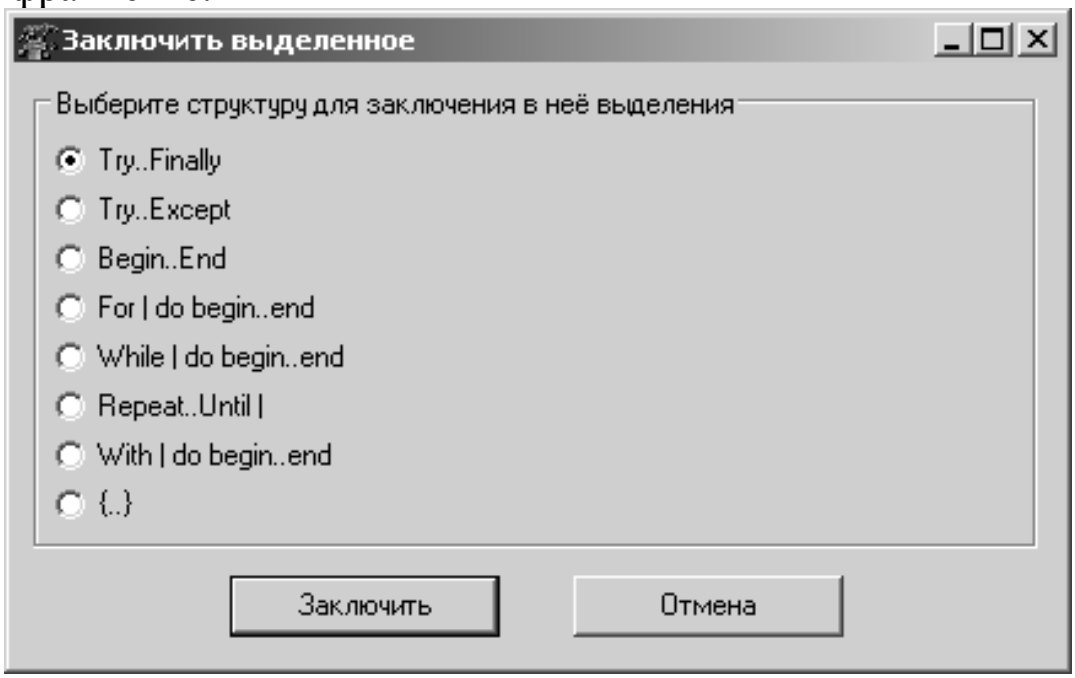

Рисунок 1.16: Выбор конструкции языка для заключения в нее выделенного фрагмента программного кода

Команды меню **Поиск** можно разделить на группы. Первая группа — это непосредственно *команды поиска и замены*, вторая - это команды перехода, а третья — работа с закладкой. В четвертой группе объединены команды поиска, замены и перехода в выделенном фрагменте. Большинство из этих команд используются в текстовых редакторах, смысл остальных понятен из названия.

Пункт меню **Просмотр** применяют для *настройки внешнего вида*

14 Комментарий — фрагмент программного кода, который игнорируется компилятором. Обычно комментарии используют для пояснений к программе или для временного исключения фрагментов текста при отладке.

среды программирования. Первая группа команд открывает или активизирует следующие окна:

- **Инспектор объектов** — окно, с помощью которого можно описать внешний вид и поведение выделенного объекта (подробно см. п. 1.4.8);
- **Редактор исходного кода** — окно, в котором можно создавать и редактировать текст программы (подробно см. п. Ошибка: источник перекрестной ссылки не найден);
- **Обозреватель кода —** содержит общую информацию о проекте;
- **Редактор LazDoc —** редактор шаблонов;
- **Браузер кода** — окно проводника проекта.

Следующая группа команд пункта меню **Просмотр** тоже открывает диалоговые окна. Эти окна носят информационный характер. Так команды **Модуль...** и **Форма...** выводя список модулей и форм данного проекта. Назначение команд **Показать зависимости модулей** и **Показать сведения о модуле** говорят сами за себя. Последняя команда этой группы **Переключить модуль/форму** активизирует либо окно редактора, либо форму.

В последней группе команд следует отметить команды **Показать палитру компонентов** и **Показать кнопки быстрого доступа**. Устанавливая метки рядом с этими командами, пользователь выводит на экран или наоборот убирает *панели инструментов*. Командой **Окна отладки** пользуются во время отладки программного кода. Здесь можно вызывать **Окно наблюдений**, **Окно отладчика**, просматривать точки останова и значения переменных в процессе выполнения программы.

Команды пункта меню **Проект** предназначены для выполнения различных *операций с проектом*:

- команды создания проекта — **Создать проект** и **Создать проект из файла**;
- команды вызова ранее созданного проекта — **Открыть проект**, **Открыть недавний проект**, **Закрыть проект**;
- команды сохранения проекта — **Сохранить проект**, **Сохранить проект как...**, **Опубликовать проект**;
- команды управления проектом — **Инспектор проекта**, **Настройка проекта...**, **Параметры компилятора** и т.д.

Команды, позволяющие *запускать проект* на выполнение и *выполнять его отладку*, содержатся в пункте главного меню **Запуск**:

- **Собрать** — сборка программы из откомпилированных файлов;
- **Собрать все** — скомпоновать все файлы проекта;
- **Быстрая компиляция** — компиляция файлов программы;
- **Запуск** — запуск проекта с помощью отладчика (компиляция, компоновка и выполнение);
- **Пауза** — приостановка выполнения программы до нажатия любой клавиши;
- **Шаг с входом** — режим пошагового отладочного выполнения программы с входом в вызываемые процедуры и функции;
- **Шаг в обход** — режим пошагового отладочного выполнения программы без входа в вызываемые процедуры и функции;
- **Запуск до курсора** - отладка и выполнение программы в этом режиме осуществляются до оператора, стоящего в строке помеченной курсором;
- **Останов** — прерывание выполнения программы;
- **Сброс отладчика** - сброс всех ранее задействованных отладочных средств и прекращение выполнения программы;
- **Настройка сборки+запуска...** - настройка параметров компоновки и запуска;
- **Вычислить\Изменить** - возможность просмотреть значение переменной и/или найти значение выражения в процессе выполнения программы, при необходимости можно изменить значения любой переменной;
- **Добавить наблюдения** — в открывающемся окне можно указать переменные и/или выражения, за изменением значений которых следует понаблюдать при отладке программы;
- **Добавить точку останова** - установка в текущей строке контрольной точки; после запуска программного кода отладчик прекратит его выполнение перед оператором, помеченным точкой останова; в программе можно указать произвольное количество таких точек. Точку останова также можно добавить щелчком мыши по номеру строки программного кода.

Работа с компонентами[15] организована при помощи команд пункта меню **Компоненты**. Пункты меню **Инструменты** и **Окружение** применяют для настройки свойств среды программирования.

15 Компонент – это готовый программный код, который можно использовать при написании программы. Пункт меню **Компоненты** предназначен для расширения стандартного набора за счет добавления компонентов других разработчиков.

Работа с окнами в Lazarus выполняется при помощи пункта меню **Окна**. Названия команд этого пункта совпадают с названиями окон, и выбор любой команды приведет к активации соответствующего окна.

Пункт меню **Справка** это справочная информация о системе. Здесь можно вызывать средства справочной системы и выполнять ее настройку.

1.4.5 Окно формы

Окно формы (рис. 1.14) — это проект интерфейса будущего программного продукта.

Вначале это окно содержит только стандартные элементы – строку заголовка и кнопки развертывания, свертывания и закрытия. Рабочая область окна заполнена точками координатной сетки[16].

Задача программиста – используя *панель компонентов*, заполнить форму различными интерфейсными элементами, создавая тем самым внешний вид своей программы.

Команды настройки окна формы находятся на вкладке **Редактор форм** (рис. 1.17) в меню **Окружение — Настройки окружения...**

1.4.6 Окно редактора Lazarus

Окно редактора (рис. 1.15) тесно связано с окном формы (рис. 1.18) и появляется вместе с ним при создании нового проекта.

Окно редактора по умолчанию находится на первом плане и закрывает окно формы. Переключаться между этими окнами можно командой **Просмотр - Переключить модуль/форму**, клавишей **F12** или просто щелчком мыши.

Окно редактора предназначено для создания и редактирования текста программы, который создается по определенным правилам и описывает некий алгоритм. Если окно формы определяет внешний вид будущей программы, то программный код, записанный в окне редактора, отвечает за ее поведение.

Вначале окно редактора содержит текст, обеспечивающий работу пустой формы. Этот программный код появляется в окне редактора автоматически, а программист в ходе работы над проектом вносит в него дополнения, соответствующие функциям программы.

16 Координатная сетка отображается только на этапе создания программы.

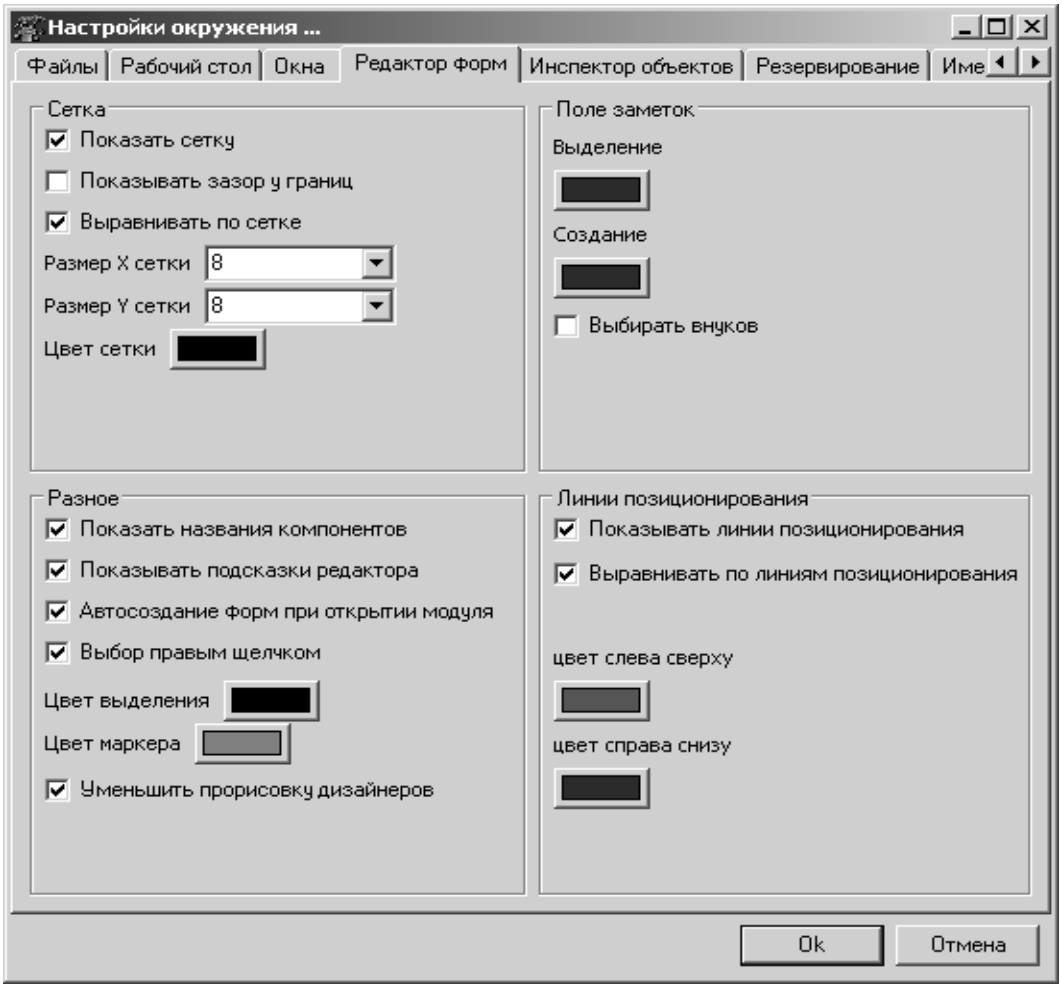

Рисунок 1.17: Настройка окна формы

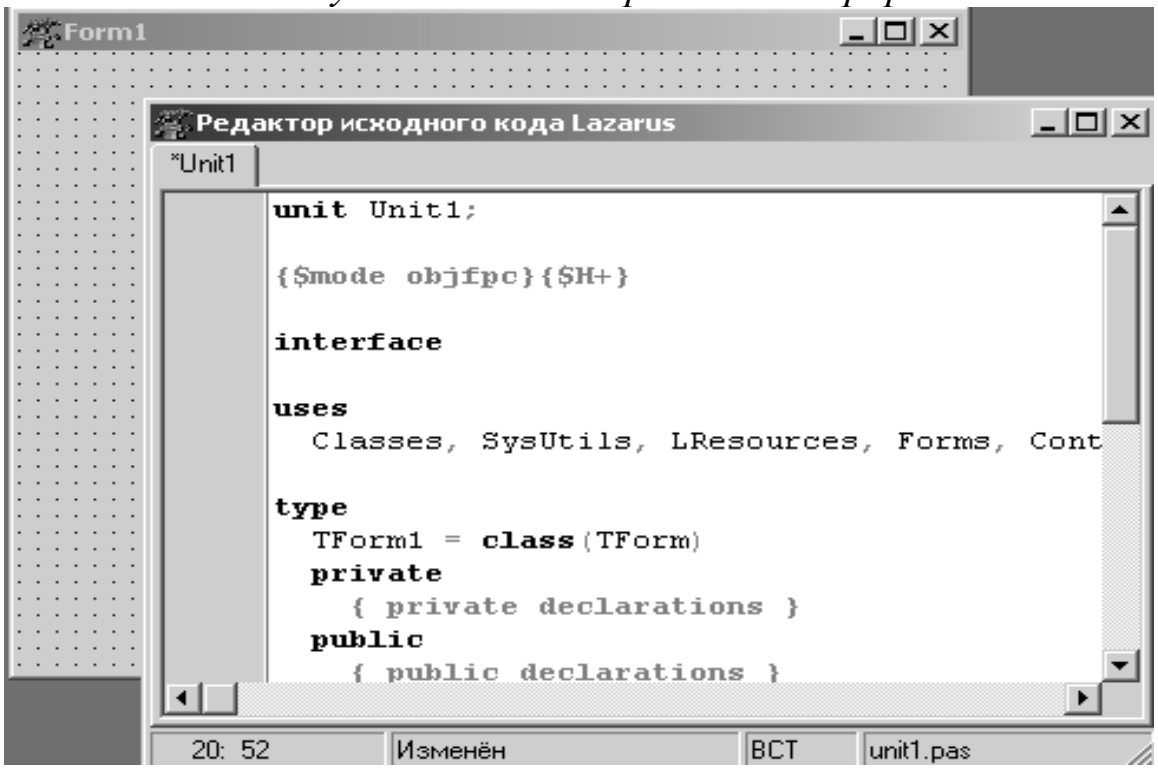

Рисунок 1.18: Окно формы и окно редактора

Обратите внимание, что при загрузке Lazarus автоматически загружается *последний проект*, с которым работал пользователь. Происходит это благодаря установке **Открывать последний проект при запуске** (рис. 1.19, 1.20), которая находится на вкладке **Файлы** меню **Окружение — Настройки окружения...**. Если убрать метку рядом с командой **Открывать последний проект при запуске**, то при загрузке Lazarus будет создавать *новый проект*.

Настроить окно редактора можно с помощью вкладки **Общие** меню **Окружение — Настройки редактора...**(рис. 1.21). Для того чтобы установить (отменить) ту или иную настройку, достаточно установить (убрать) маркер рядом с ее названием и подтвердить изменения нажатием кнопки **Ok**.

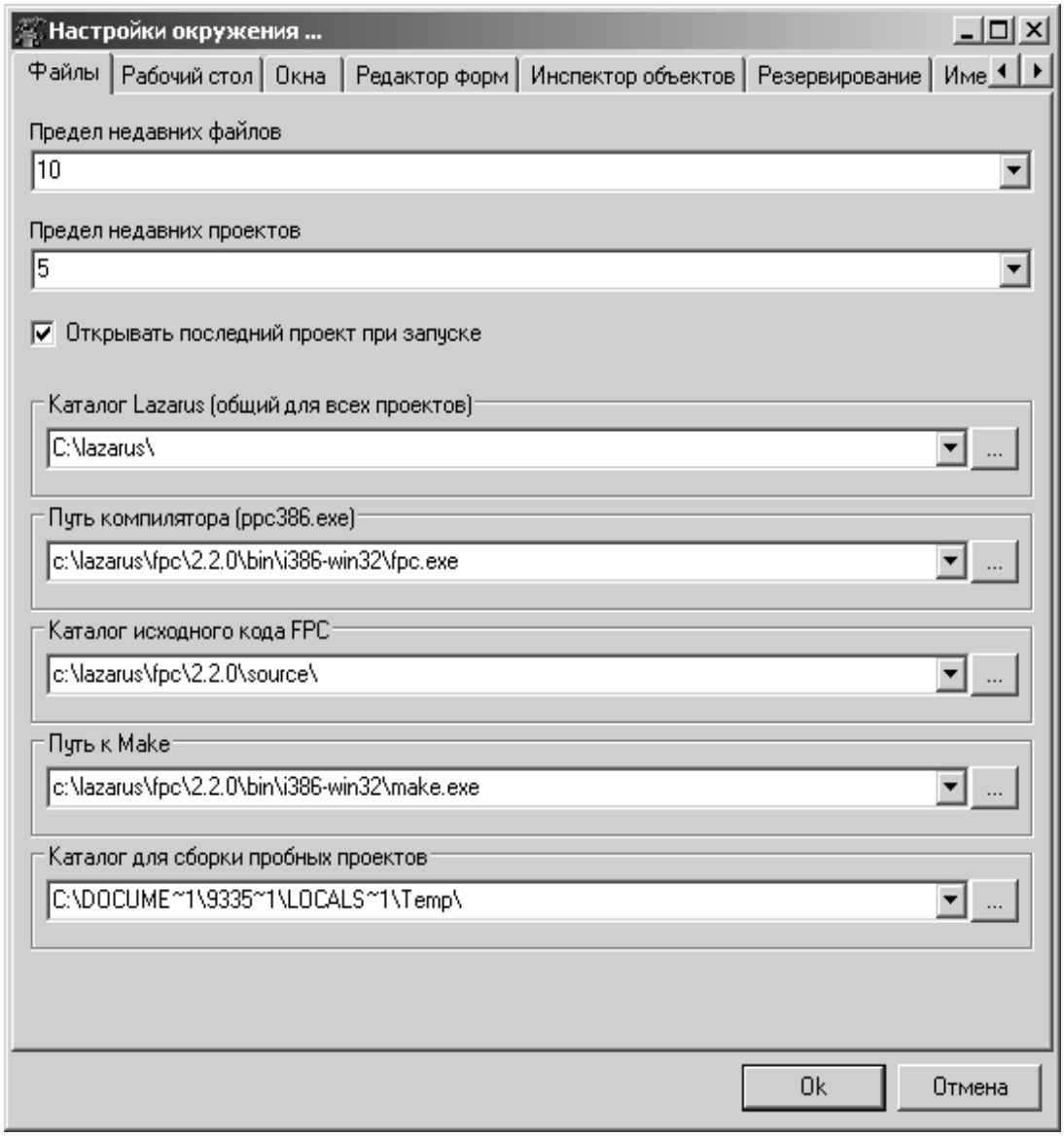

Рисунок 1.19: Окно настройки файлов в среде Windows

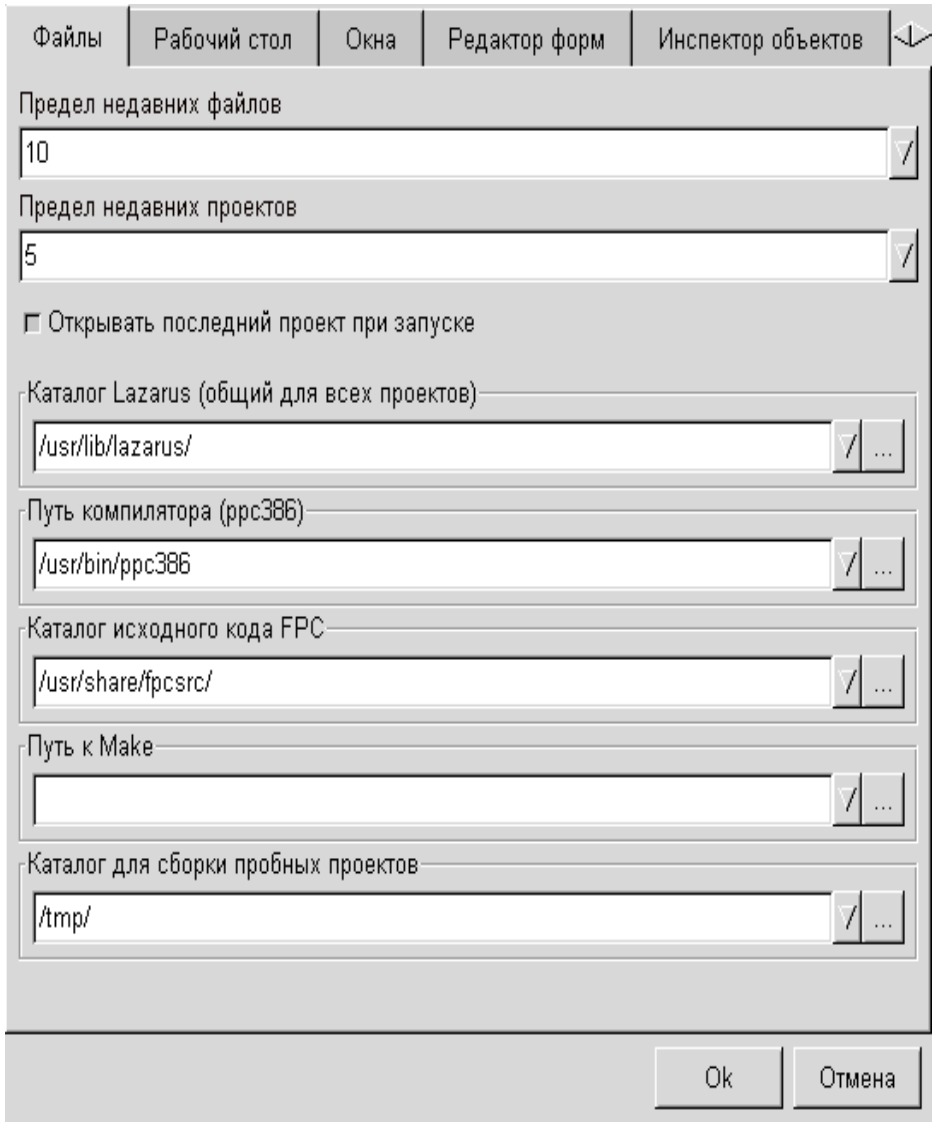

Рисунок 1.20: Окно настройки файлов в Linux

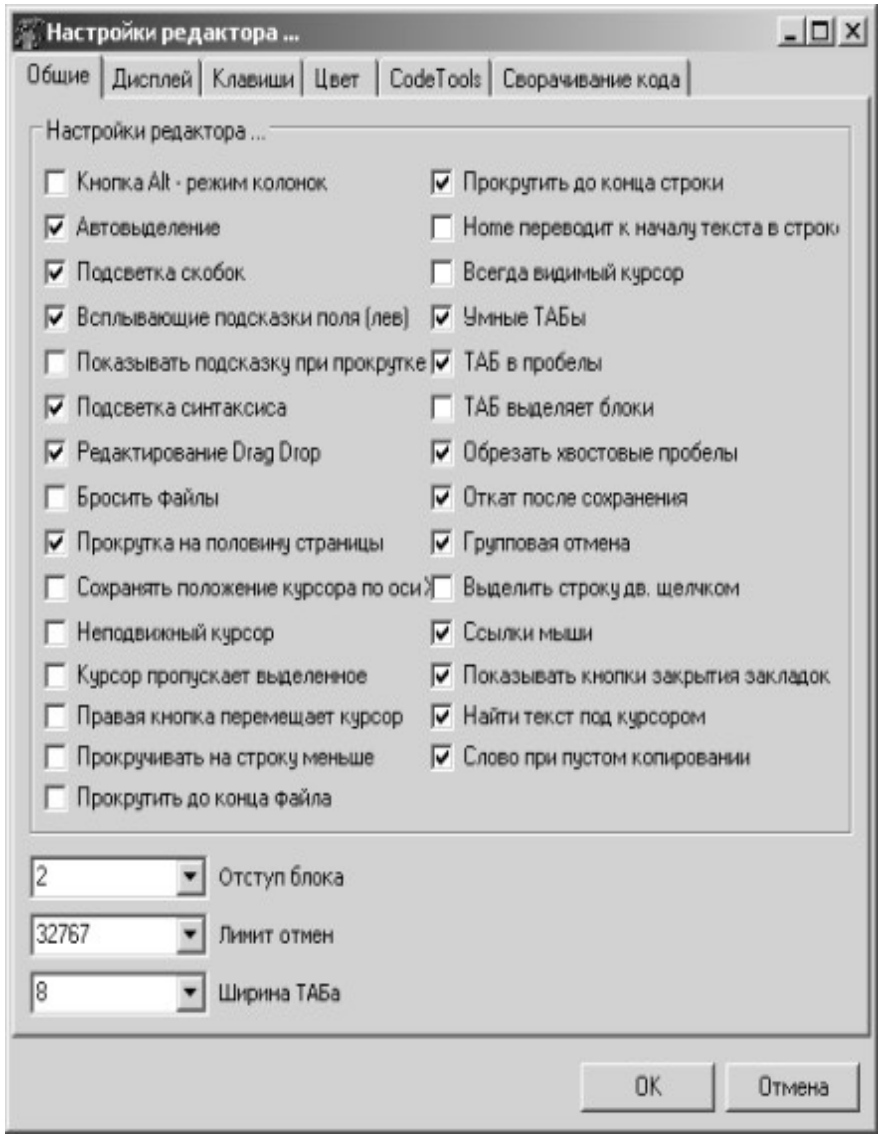

Рисунок 1.21: Настройки окна редактора

На вкладке **Дисплей** меню **Окружение — Настройки редактора...**(рис. 1.22) можно *изменить шрифт* текста программного кода.

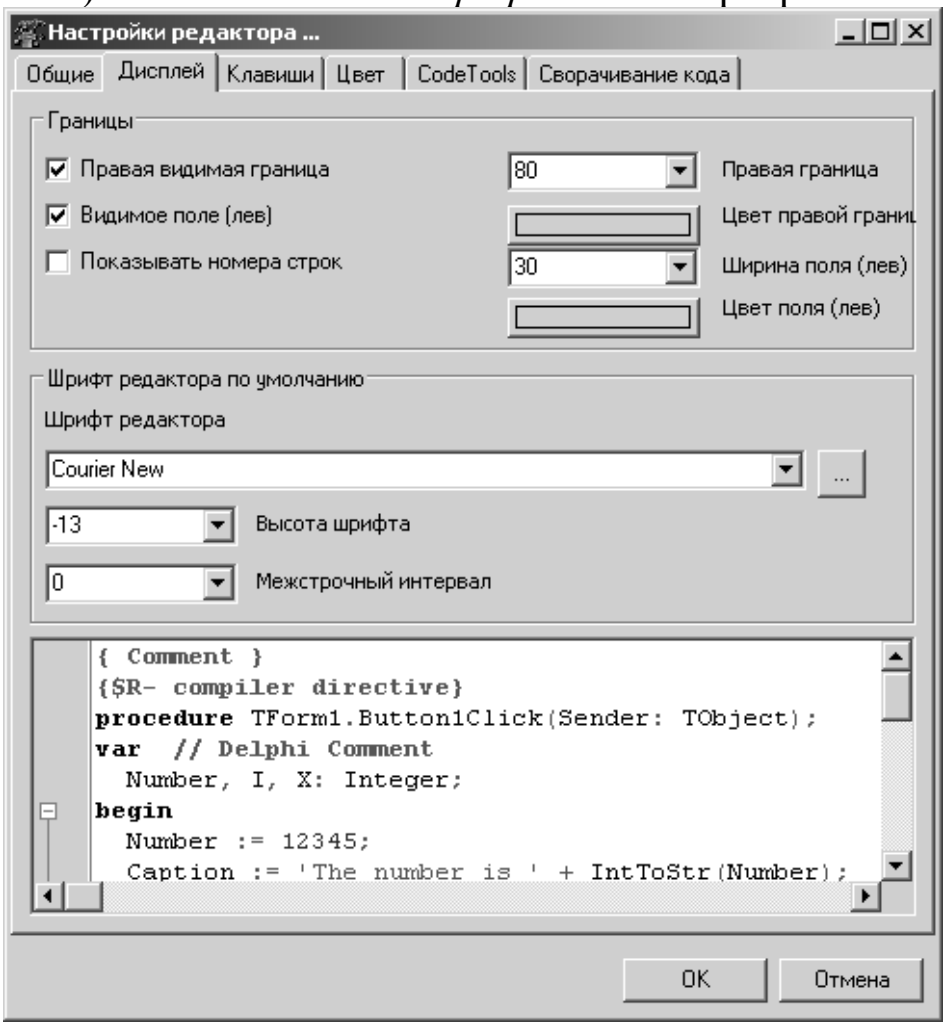

Рисунок 1.22: Окно настройки редактора программного кода, вкладка Дисплей

Для корректного отображения символов кириллицы в окне редактора кода под управлением Ubuntu Linux необходимо в окне настроек редактора в поле **Шрифт редактора** установить тип шрифта — **Fixed Misc**, после чего выбрать кодировку **ISO 10646-1** (рис. 1.23).

Как и в большинстве текстовых редакторов, в редакторе программного кода Lazarus предусмотрена *работа с шаблонами*. Здесь под шаблоном подразумевается автоматический ввод конструкций языка программирования.

Например, пользователь вводит символ `b`, затем символ пробел, а на экране появляется конструкция `begin ... end`. Настройку шаблонов можно выполнить в диалоговом окне (рис. 1.24, 1.25), которое появится после выполнения команды **Окружение — Шаблоны кода...**. Например, чтобы заработал рассмотренный шаблон, нужно

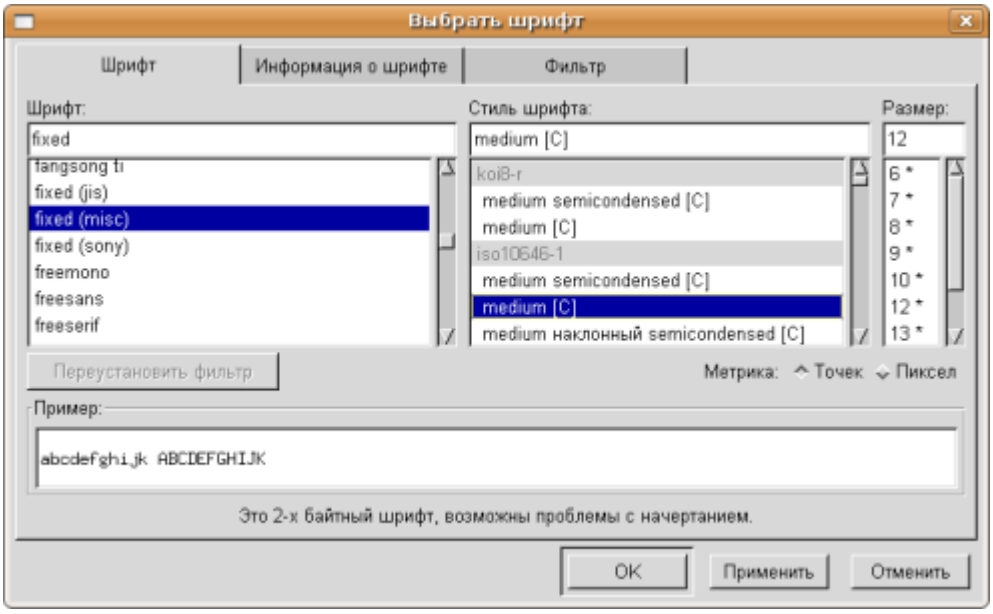

Рисунок 1.23: Настройка кириллицы в ОС Ubuntu

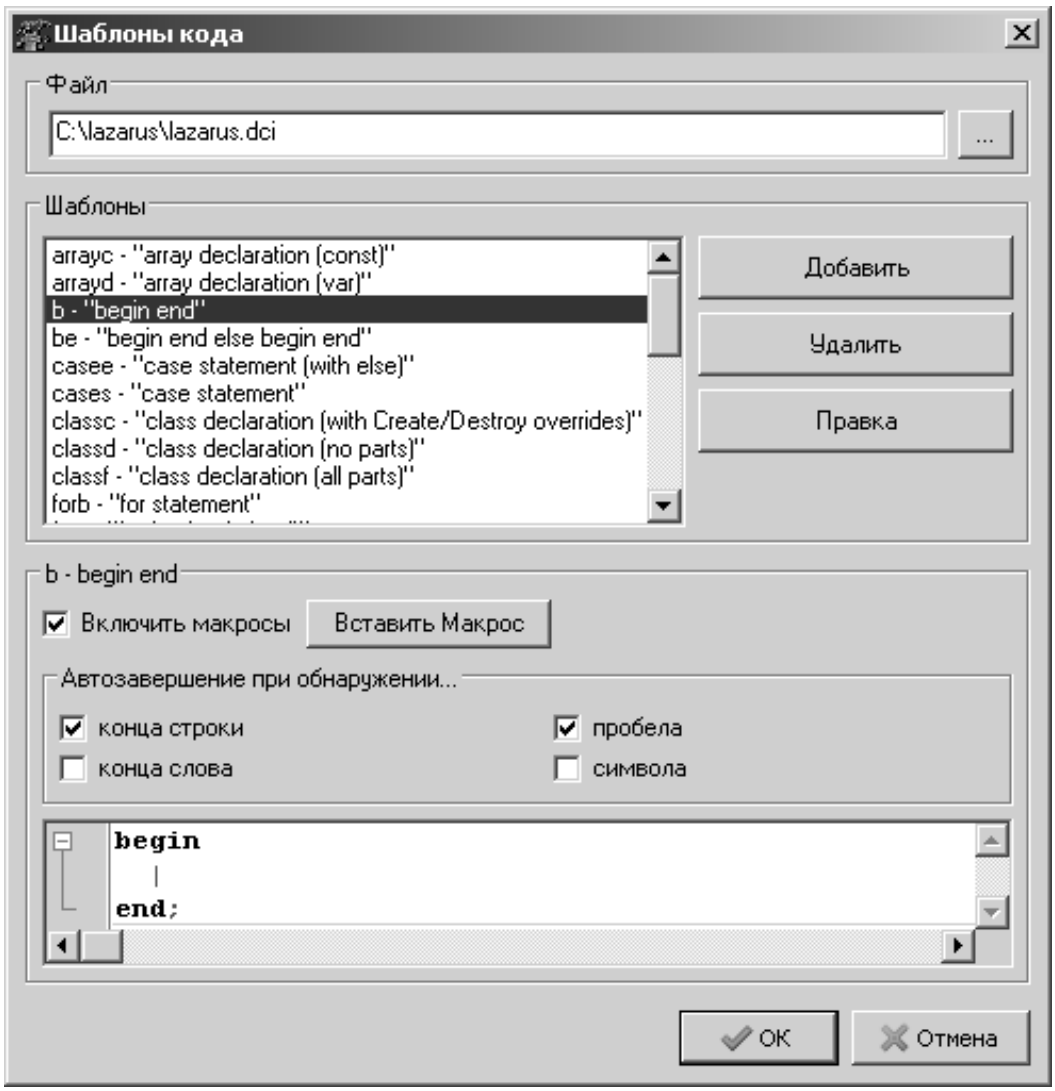

Рисунок 1.24: Окно настройки шаблонов редактора программного кода в среде Windows

выбрать (щелкнуть по нему мышкой) в списке шаблонов строку `b - begin ... end` и установить маркер рядом с командой **Включить макросы**. Затем в группе команд **Автозавершение при обнаружении...** включить флажок возле слова **пробела**. Если среди команд автозавершения выбрать **конец строки**, то шаблон будет появляться в тексте программы после ввода символа `b` и нажатия клавиши **Enter**.

Шаблоны можно добавлять, удалять и править. Для этого в окне **Шаблоны кода** (рис. 1.25) предусмотрены специальные кнопки. В нижней части окна, над кнопками **Ok** и **Отмена**, расположено поле ввода и редактирования шаблонов с активным курсором.

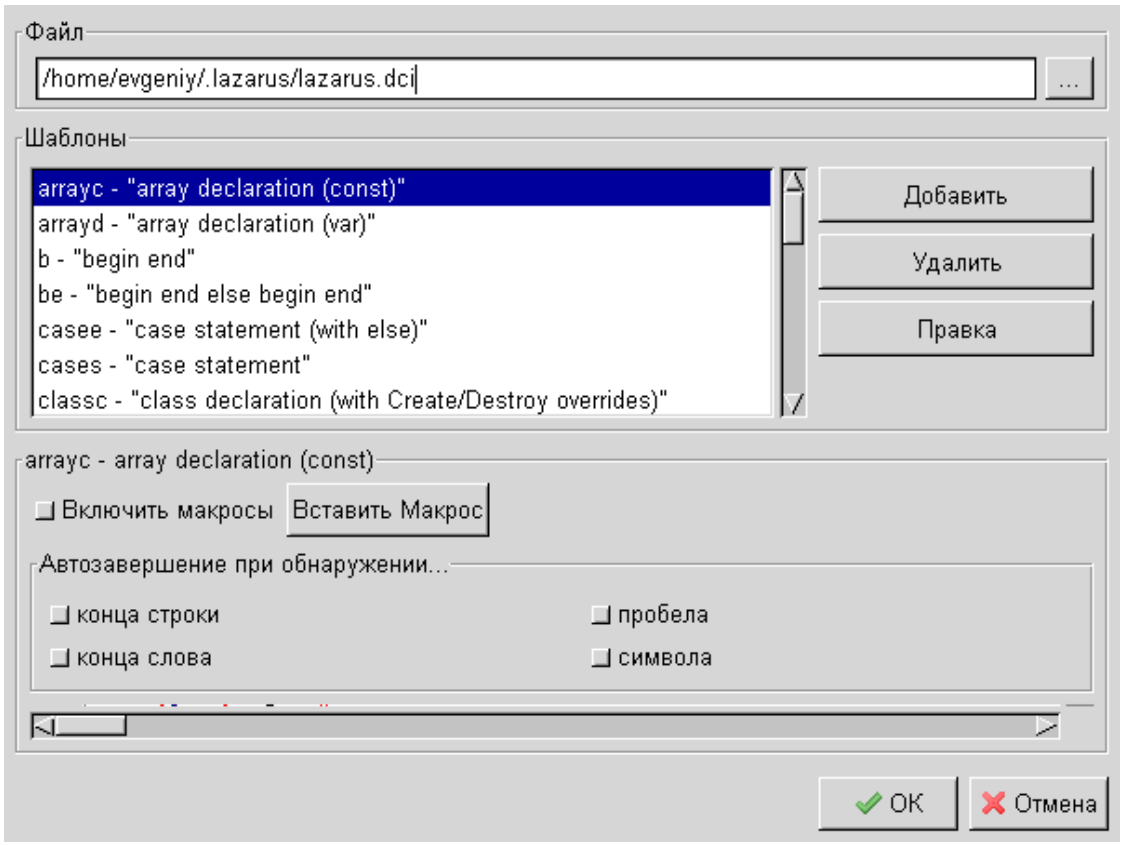

Рисунок 1.25: Окно настройки шаблонов редактора программного кода в среде Linux

Если выделить любой шаблон из списка, щелкнув по нему мышкой, то в поле ввода появится соответствующая конструкция. Если воспользоваться кнопкой **Добавить**, то появится диалоговое окно (рис. 1.26) для *создания шаблона*. Здесь в поле **Элемент** следует указать символ или группу символов, ввод которых будет связан с создаваемым шаблоном.

В поле **Комментарий** можно дать краткое описание шаблона и нажать кнопку **Добавить**. Элемент и комментарий к нему будут добавлены в список шаблонов в окне **Шаблоны кода**. Теперь нужно создать сам шаблон, то есть указать, какая именно конструкция языка появится при вводе элемента в текст программы.

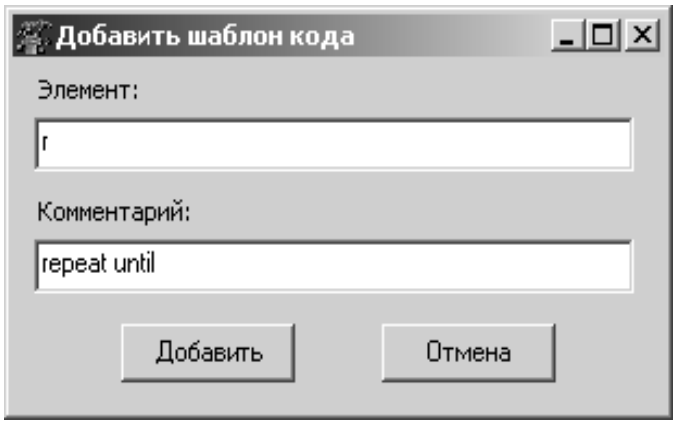

Рисунок 1.26: Окно создания шаблона

Для этого надо выделить вновь созданный шаблон и в поле ввода, расположенном в нижней части окна **Шаблоны кода,** ввести соответствующий текст. Например, создадим шаблон для конструкции `repeat ... until`, которая будет добавлена в текст, если пользователь введет символ `r`. Выполним следующие действия:

- щелкнем по кнопке **Добавить** в окне **Шаблоны кода**;
- в диалоговом окне **Добавить шаблон кода** (рис. 1.26) в поле **Элемент** введем символ `r`, а в поле **Комментарий** фразу `repeat until` и нажмем кнопку **Добавить**;
- в окне **Шаблоны кода** (рис. 1.27, 1.28) выделим в списке шаблонов строку `r - repeat until`;
- введем в поле ввода, расположенном в нижней части окна, конструкцию языка `repeat until;`;
- нажмем кнопку **Ok** в окне **Шаблоны кода**.

Для *изменения шаблона* служит кнопка **Правка**. Шаблон, который нужно откорректировать, должен быть выделен. Дальнейшие действия аналогичны тем, которые выполняются при создании шаблона.

Для того чтобы *удалить шаблон* из списка, его нужно выделить, а затем нажать кнопку **Удалить**.

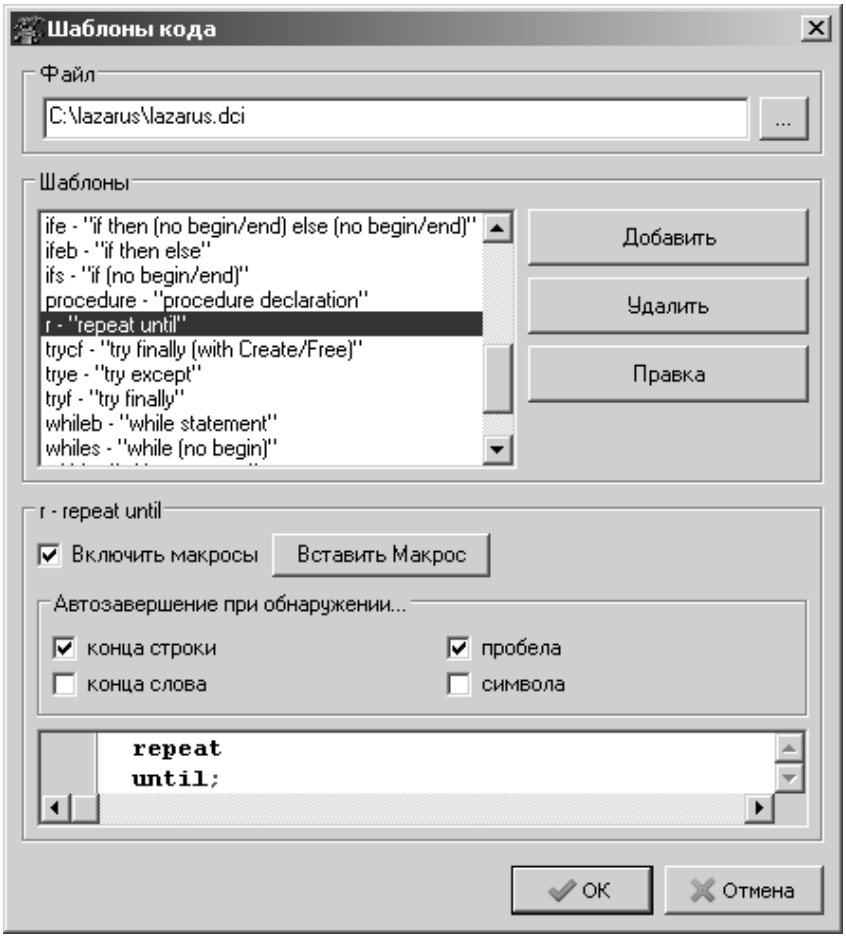

Рисунок 1.27: Создание шаблона в Windows

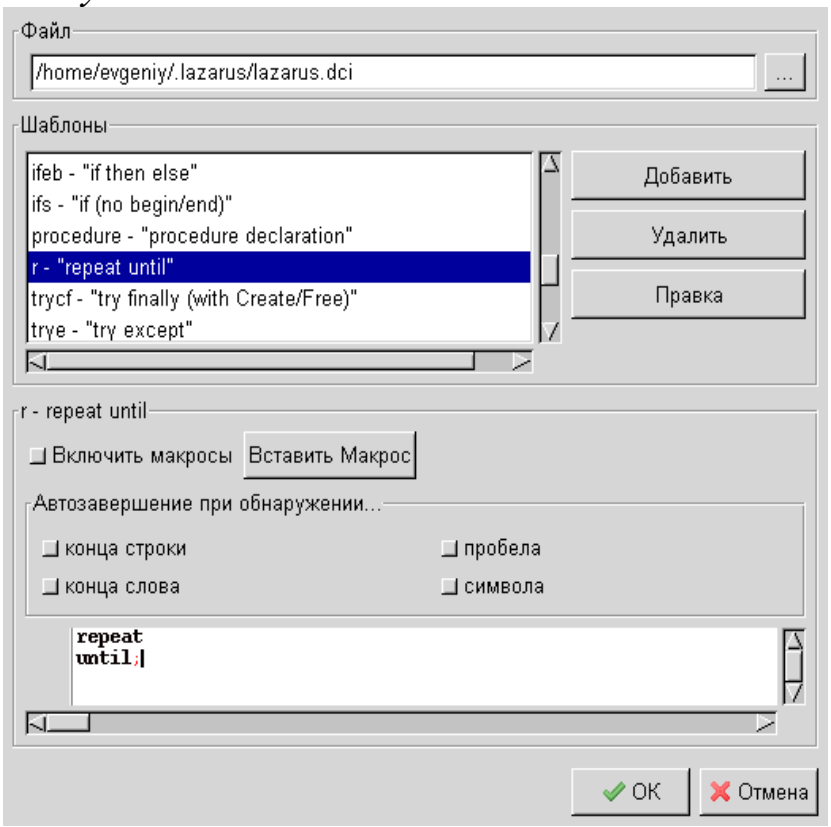

Рисунок 1.28: Создание шаблона в Linux

1.4.7 Панель компонентов

Панель компонентов расположена[17] под главным меню (рис. 1.15). Она состоит из большого числа групп, в которых располагаются соответствующие компоненты (рис. 1.29).

Рисунок 1.29: Панель компонентов

Компонент – это некий функциональный элемент интерфейса, обладающий определенными свойствами. Размещая компоненты на форме, программист создает внешний вид своей будущей программы – окна, кнопки, переключатели, поля ввода и т.п.

Для *внедрения нового компонента* на форму нужно сделать два щелчка мышкой:

- в панели компонентов, для выбора компонента;
- в рабочем пространстве формы, для указания положения левого верхнего угла компонента.

Компоненты объединяются в группы по функциональному признаку. После создания проекта по умолчанию открывается список группы **Standard**, содержащий основные элементы диалоговых окон. Просмотреть другие группы можно, раскрывая их щелчком по соответствующей вкладке.

1.4.8 Инспектор объектов

Окно *инспектора объектов* располагается слева от окна редактирования. Как правило, оно содержит информацию о выделенном объекте. На рис. 1.30 представлен инспектор объектов с информацией о вновь созданной форме. Окно инспектора объектов имеет три вкладки: **Свойства**, **События**, **Избранное.** Эти вкладки используются для редактирования свойств объекта и описания событий, на которые будет реагировать данный объект. Совокупность *свойств* отображает внешнюю сторону объекта, совокупность *событий* – его поведение.

Вкладки инспектора объектов представляют собой таблицу. В левой колонке расположены названия свойств или событий, в правой – конкретные значения свойств или имена подпрограмм, обрабатывающих события. Чтобы выбрать свойство или событие, необходимо щелкнуть левой кнопкой мыши по соответствующей строке.

17 Включить (выключить) **Панель компонентов** можно, установив (убрав) маркер рядом с командой **Показать палитру компонентов** меню **Просмотр**.

Свойства, отображенные в таблице, могут быть простыми или сложными. Простые свойства определены единственным значением.

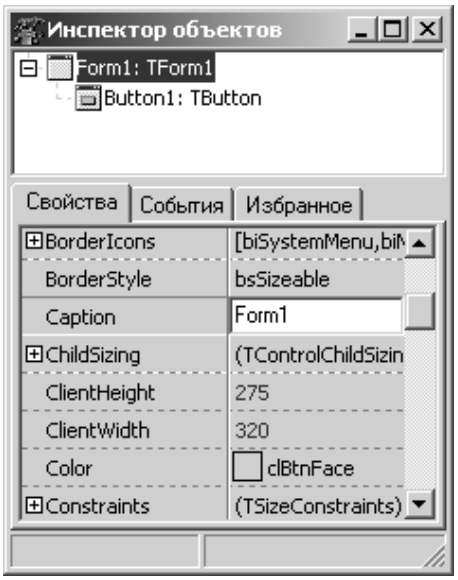

Рисунок 1.30: Окно инспектора объектов

Например, свойство `Caption` (Заголовок) определяется строкой символов, свойства `Height` (Высота) и `Width` (Ширина) – числом, свойство `Enabled` (Доступность) – значениями `True` (Истина) и `False` (Ложь). Сложные свойства определяются совокупностью значений. Например, свойство `Font` (Шрифт). Слева от имени этого свойства стоит знак «+». Это означает, что свойство сложное, и при щелчке по значку «+» откроется список его составляющих.

Активизировать значение любого свойства можно обычным щелчком мыши. При этом в конце строки может появиться либо кнопка с символом `троеточия`, либо кнопка со стрелкой, направленной вниз. Щелчок по `троеточию` откроет диалоговое окно для установки значений сложных свойств (например, `Font`). Обращение к стрелке приведет к раскрытию списка возможных значений простого свойства (например, `Enabled`).

1.4.9 Первая программа в Lazarus

Процесс создания программы в Lazarus состоит из двух этапов: формирование внешнего вида программы, ее интерфейса и написание программного кода на языке программирования Free Pascal, заставляющего работать элементы интерфейса.

Как мы уже выяснили, для создания интерфейса программы существует *окно формы*, а для написания программного кода – *окно редактора*. Эти окна тесно связаны между собой, и размещение *компонентов* на форме приводит к автоматическому изменению программного кода.

Начнем знакомство с визуальным программированием с создания простой программы про кнопку, которая хочет, чтобы по ней щелкнули. После чего появляется сообщение о работе программы.

Начнем с *создания нового проекта*. Для этого выполним команду

главного меню **Проект — Создать проект...**. В появившемся диалоговом окне (рис. 1.31, 1.32) выберем из списка слово **Приложение** и нажмем кнопку **Создать**. Результатом этих действий будет появление *окна формы* и *окна редактора программного кода*.

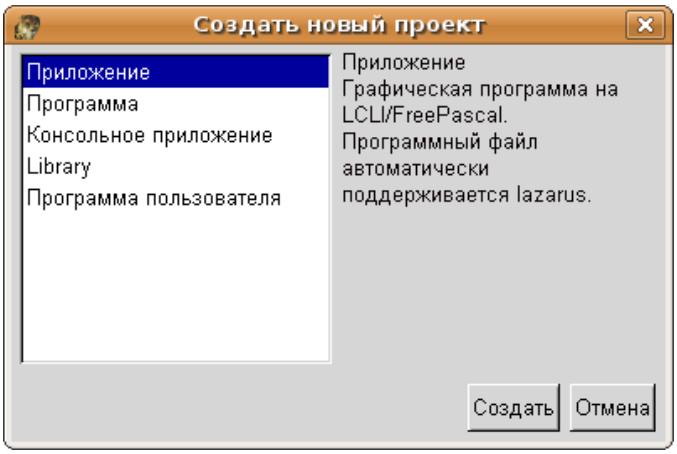

Рисунок 1.31: Создание нового проекта в Linux

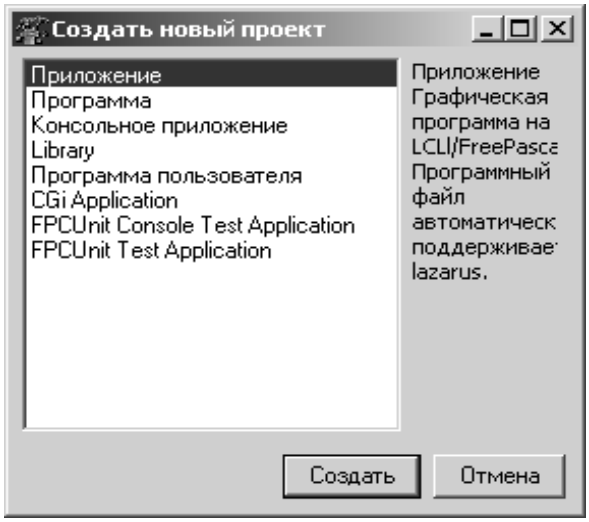

Рисунок 1.32: Создание нового проекта в среде Windows

Сохраним созданный проект, воспользовавшись командой **Проект — Сохранить проект как…**. Откроется окно сохранения программного кода **Сохранить Unit1** (рис. 1.33, 1.34). Создадим в нем новую папку **Primer_1** (можно воспользоваться кнопкой **Создание новой папки (Создание каталога)**), откроем ее и щелкнем по кнопке **Сохранить**. Тем самым мы сохраним файл **Unit1.pas**, содержащий текст программы.

Сразу же откроется окно **Сохранить проект** (рис. 1.35, 1.36), в котором также необходимо щелкнуть по кнопке **Сохранить**.

Теперь мы сохранили файл **Project1**[18], содержащий общие сведения о проекте.

На самом деле все наши манипуляции привели к сохранению более чем двух файлов. Теперь в каталоге **Primer_1** (рис. 1.37, 1.38) хранится файл с текстом программы **Unit1.pas**, файл **Unit1.lfm**[19], со сведениями о форме **Form1**, а также файлы **Project1.lpn** и **Project1.lpi,** содержащие настройки системы программирования и параметры конфигурации проекта.

18 Имена **Unit1** и **Project1** могут быть любыми другими.
19 Имена файлов можно задать и отличные от стандартных Unit1.

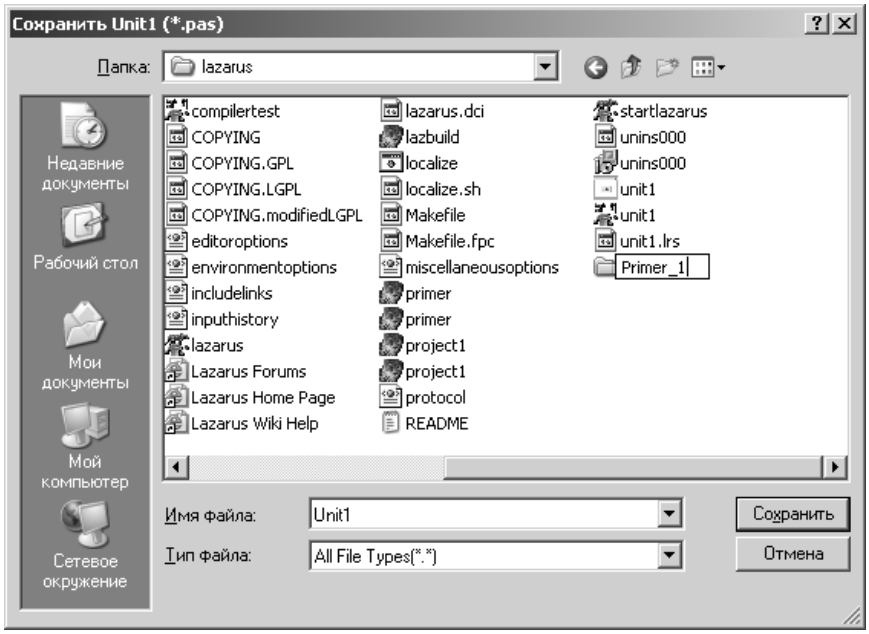

Рисунок 1.33: Сохранение файла в Windows

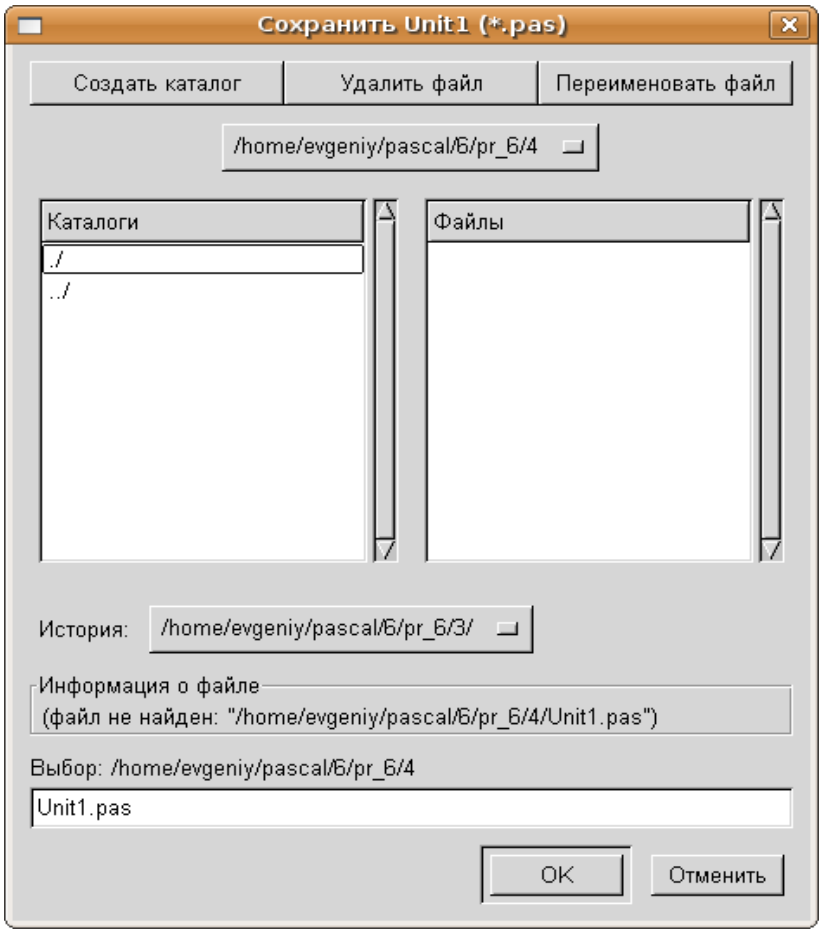

Рисунок 1.34: Сохранение файла в Linux

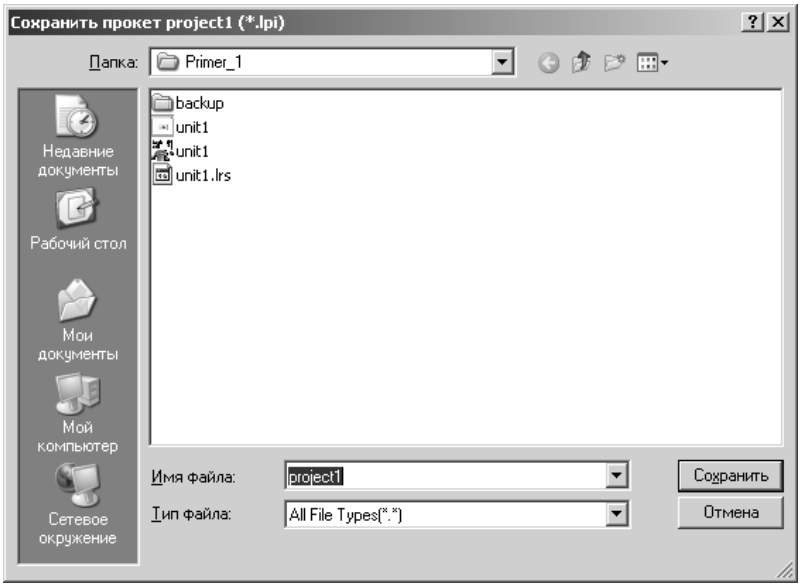

Рисунок 1.35: Сохранение проекта в Windows

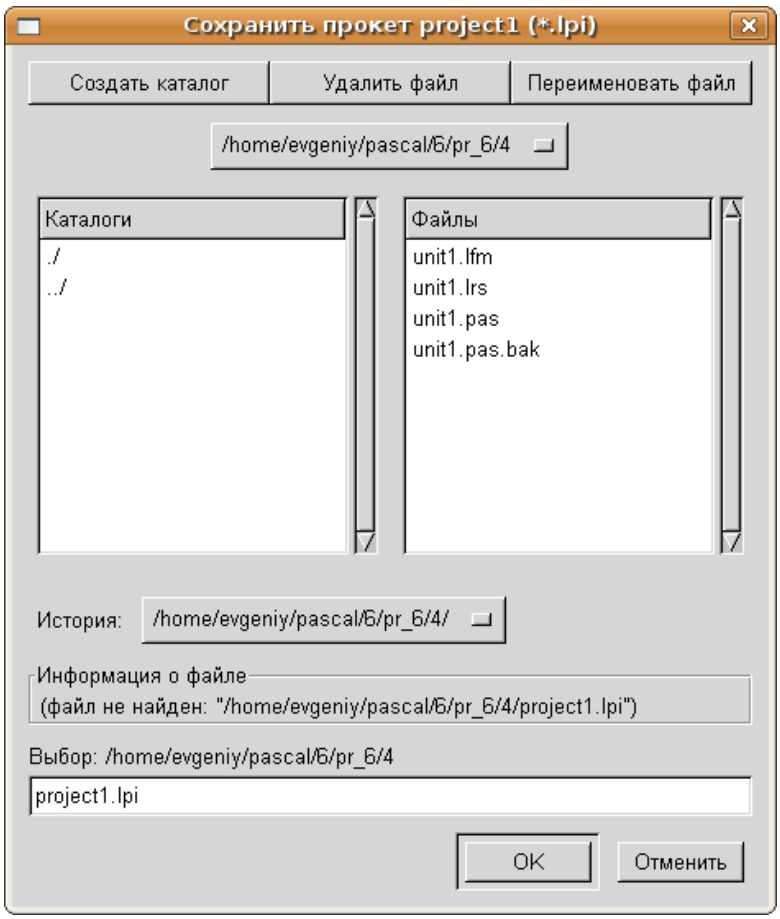

Рисунок 1.36: Сохранение проекта в Linux

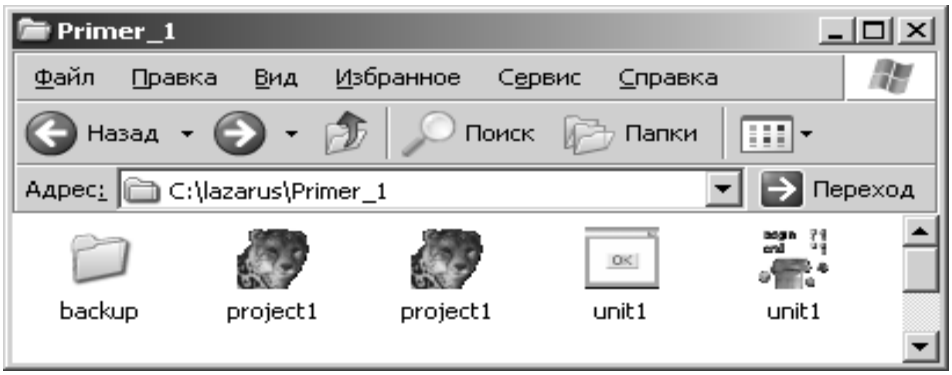

Рисунок 1.37: Файлы проекта в Windows

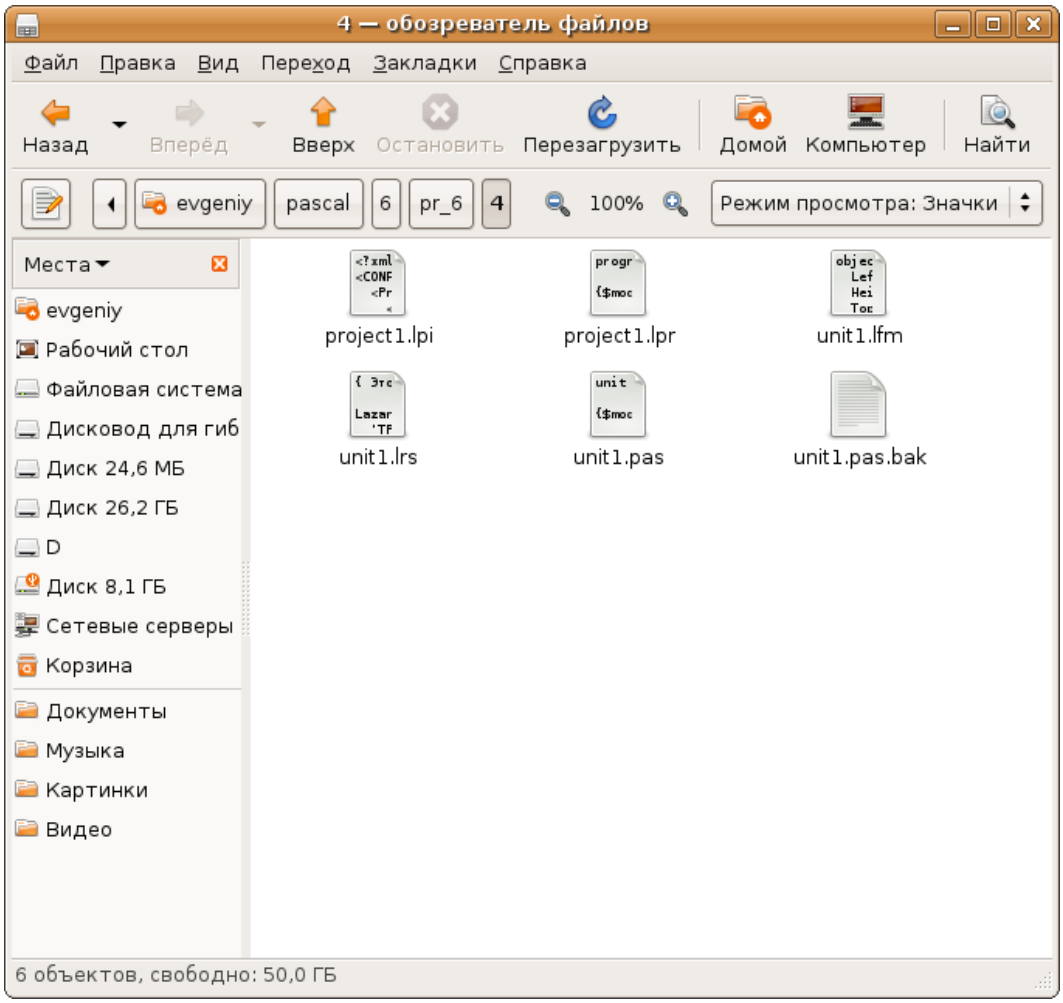

Рисунок 1.38: Файлы проекта в Linux

Итак, проект сохранен. Все дальнейшие изменения будем сохранять командой **Проект - Сохранить проект**.

Теперь можно приступить к визуальному программированию. У нас есть один объект – форма `Form1`. Изменим некоторые его свойства с помощью инспектора объектов. Перейдем в окно инспектора объектов и найдем там свойство `Caption` (Заголовок). По умолчанию это свойство имеет значение `Form1`. Изменим его на слово ПРИ-

МЕР 1 (рис. 1.39). Это изменение сразу же отразится на форме – в поле заголовка появится надпись – ПРИМЕР 1 (рис. 1.40). Аналогично можно изменить размеры формы, присвоив новые значения свойствам `Height` (Высота) и `Width` (Ширина), еще проще это сделать, изменив положение границы формы с помощью кнопки мыши, как это делается со всеми стандартными окнами.

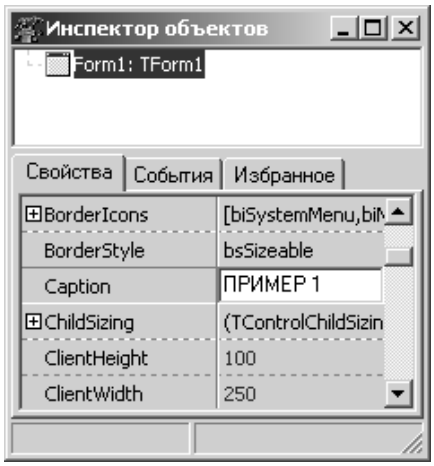

Рисунок 1.39: Изменение свойства формы Caption

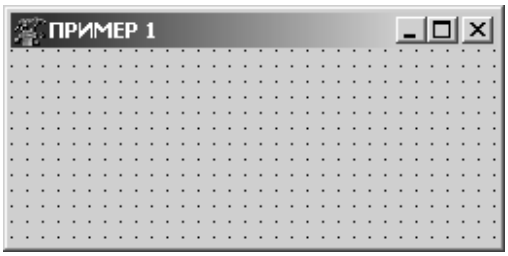

Рисунок 1.40: Изменение заголовка формы

Изменим еще одно свойство формы – `Position` (Положение формы на экране). Значения этого свойства не вводятся вручную, а выбираются из списка (рис. 1.41). Если выбрать свойство `poScreenCenter`, то форма всегда будет появляться в центре экрана.

Напомним, что мы пишем программу про кнопку. Пришло время поместить ее на форму. Для этого обратимся к панели компонентов (рис. 1.29) и найдем там компонент `Tbutton`. Чтобы *разместить компонент* на форме, нужно сделать два щелчка мышкой: первый — по компоненту, второй — по окну формы. В результате форма примет вид, представленный на рис. 1.42.

Теперь у нас два объекта: форма `Form1` и кнопка `Button1`. Напомним, что изменения значений свойств в окне инспектора объектов относятся к выделенному объекту. Поэтому выделим объект `Button1` и изменим его заголовок `Caption` и размеры `Height`, `Width`. В заголовке напишем фразу «`Щелкни мышкой!`», установим ширину 135 пикселей, а высоту – 25 пикселей. *Сохраним* проект (**Проект — Сохранить проект**).

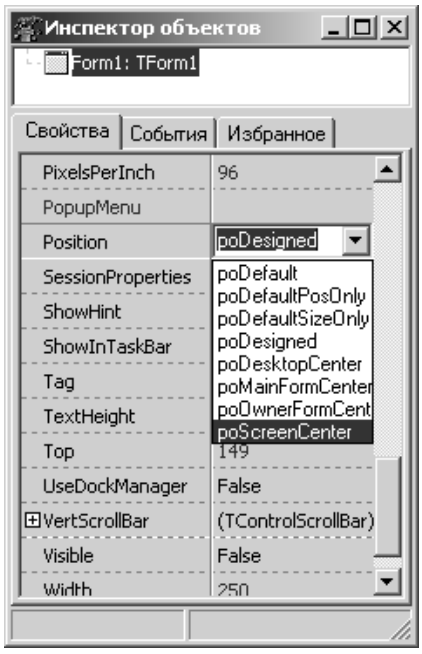

Рисунок 1.41: Изменение свойства формы Position

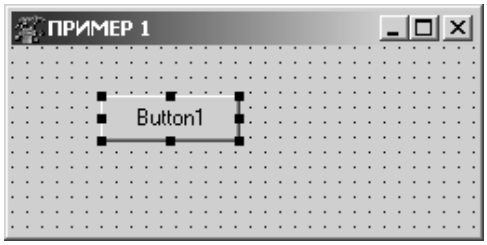

Рисунок 1.42: Размещение кнопки на форме

Для того чтобы посмотреть, как работает наша программа, ее необходимо *запустить на выполнение*. Сделать это можно командой **Запуск — Запуск**, функциональной клавишей **F9** или кнопкой **Запуск** в панели инструментов (рис. 1.43). Созданное окно с кнопкой должно выглядеть, как на рис. 1.44.

Рисунок 1.43: Кнопка Запуск в панели инструментов Lazarus

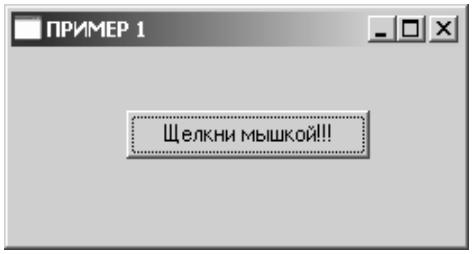

Рисунок 1.44: Результат работы программы

Теперь щелкните по кнопке и убедитесь, что ничего не произойдет. В этом нет ничего удивительного, ведь пока мы не написали ни одной строчки программного кода. Можно сказать, что мы разработали только внешнюю часть программы, но не позаботились о функциональной. Несмотря на это, в окне редактора появился текст программы:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes, SysUtils, LResources, Forms,
   Controls,   Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
 TForm1 = class(TForm)
 Button1: TButton;
 private
     { private declarations }
 public
     { public declarations }
end;
var
 Form1: TForm1;
implementation
initialization
 {$I unit1.lrs}
end.
```

О назначении всех перечисленных в листинге команд мы поговорим позже. Вернемся к программе и подумаем, что же мы хотим от нашей кнопки. Скорее всего, это должна быть реакция на какое-то событие, например, на щелчок мыши. После запуска программы на экране появилось окно, на котором расположена кнопка с надписью «Щелкни мышкой!». Предположим, что если мы щелкнем по ней, она «обрадуется» и сообщит: «УРА! ЗАРАБОТАЛО!».

Для воплощения этой идеи завершим работу программы, закрыв окно с кнопкой обычным способом (щелчком по крестику в правом верхнем углу) и перейдем в режим редактирования. Убедимся в том, что объект кнопка `Button1` выделен, и обратимся к вкладке **События** инспектора объектов. Напомним, что здесь расположены описания событий. Выберем событие `OnClick` – *обработка щелчка мыши* и дважды щелкнем в поле справа от названия (рис. 1.45).

В результате описанных действий в окне редактора появиться следующий текст:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
end;
```

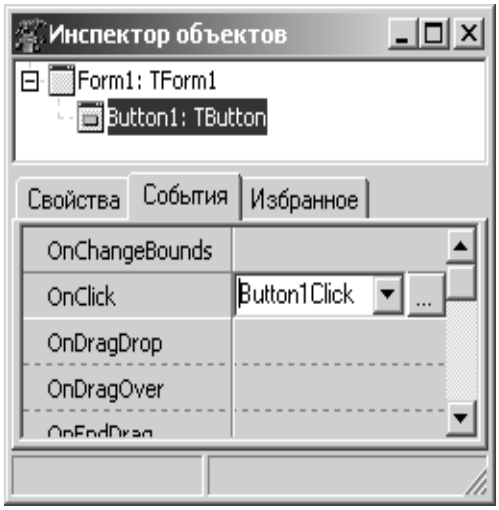

Рисунок 1.45: Выбор события OnClick

На данном этапе изложения материала данный текст – это фрагмент программного кода, именуемый *подпрограммой*. О назначении этой подпрограммы можно догадаться по ее имени `TForm1.Button1Click`: на форме `Form1` для объекта кнопка `Button1` обрабатывается событие «щелчок мыши» `Click`. Все команды, написанные между словами `begin` и `end`, будут выполняться при наступлении указанного события.

Теперь установим курсор между словами `begin` и `end` созданной подпрограммы и напишем:

```
Button1.Caption:='УРА! ЗАРАБОТАЛО!';
```

Эта запись означает изменение свойства кнопки. Только теперь мы выполнили его не с помощью инспектора объектов, а записав оператор языка программирования. Прочитать его можно так: присвоить (`:=`) свойству `Caption` объекта `Button1` значение `'УРА! ЗАРАБОТАЛО!'`. Поскольку присваиваемое значение – строка, оно заключено в одинарные кавычки. Теперь, текст подпрограммы в окне редактора имеет вид:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
     Button1.Caption:='УРА! ЗАРАБОТАЛО!';
end;
```

Рисунок 1.46: Окончательный результат работы программы

Сохраним созданную программу, запустим ее на выполнение и убедимся, что кнопка реагирует на щелчок мыши (рис. 1.46). *Закрыть проект* можно командой **Проект — Закрыть проект**.

1.4.10 Полезная программа

Итак, первая программа написана. Она не несет особой смысловой нагрузки. Главная ее задача – показать, что же такое визуальное программирование. Следующая программа также будет несложной, но полезной. Мы автоматизируем процесс перевода старой русской меры веса в современную. Эта задача формулируется так: выполнить перевод пудов и фунтов в килограммы, если известно, что 1 пуд = 40 фунтам = 16.38 кг.

Итак, наша программа должна позволить пользователю ввести два числа (пуды и фунты) и после щелчка по кнопке сообщить, каково значение в килограммах.

Создадим новый проект, выполнив команду **Проект — Создать проект...**, и сохраним его в папке **Primer_2** командой **Проект — Сохранить проект как..**.

Воспользуемся вкладкой **Standard** панели компонентов и разместим на форме следующие компоненты. Четыре *объекта типа надпись* `Label`. Этот компонент используется для размещения в окне не очень длинных однострочных надписей. Два *поля ввода* `Edit`. Это редактируемое текстовое поле, предназначенное для ввода, отображения или редактирования одной текстовой строки. И одну *кнопку* `Button`. Этот компонент обычно используют для реализации некоторой команды. Компоненты должны располагаться на форме примерно так, как показано на рис. 1.47.

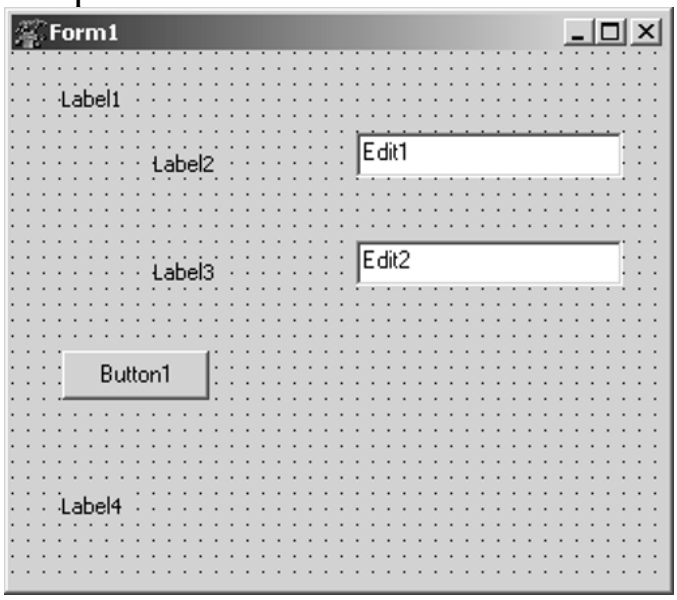

Рисунок 1.47: Конструирование формы

В первой надписи будет храниться текст «Меры веса». Две надписи `Label2` и `Label3` нужны для пояснения того, какие именно данные должны вводиться, надпись `Label4` понадобится для вывода результата.

В поля ввода `Edit1` и `Edit2` будут введены конкретные значения: вес в пудах и фунтах.

Вычисление веса в килограммах произойдет при щелчке по кнопке `Button1` и будет выведено в `Label4`.

Изменим свойства формы и размещенных на ней компонентов в соответствии с табл. 1.1-1.4.

Таблица 1.1 Значение свойств формы

Свойство	**Значение**	**Описание свойства**
`Caption`	Перевод из старой русской меры веса в современную	Заголовок формы
`Height`	230	Высота формы
`Width`	400	Ширина формы
`Font.Name`	Arial	Название шрифта
`Font.Size`	10	Размер шрифта

Таблица 1.2. Свойства компонентов типа надпись

Свойство	**Label1**	**Label2**	**Label3**	**Label4**	**Описание свойства**
`Caption`	Меры веса	пуды	фунты	В килограммах:	Заголовок компонента
`Height`	15	15	15	15	Высота компонента
`Width`	60	30	35	85	Ширина компонента
`Top`	20	50	105	25	Координата верхнего края
`Left`	25	115	115	200	Координата левого края

Таблица 1.3. Свойства компонентов поле ввода

Свойство	Edit1	Edit2	Описание свойства
Text	пробел	пробел	Текст в поле ввода
Height	25	25	Высота компонента
Width	135	135	Ширина компонента
Top	40	95	Координата верхнего края
Left	175	175	Координата левого края

Таблица 1.4. Свойства компонента кнопка

Свойство	Значение	Описание свойства
Caption	ВЫЧИСЛИТЬ	Текст на кнопке
Height	25	Высота компонента
Left	200	Ширина компонента
Top[20]	150	Координата верхнего края
Width	80	Координата левого края

После всех изменений форма будет иметь вид, представленный на рис. 1.48.

Рисунок 1.48: Изменение свойств формы и ее компонентов

Интерфейс программы готов. Займемся вычислительной частью задачи. Для этого создадим для кнопки Button1 обработчик событий OnClick, для чего дважды щелкнем по объекту Button1 левой кнопкой мыши. Между словами begin и end созданной подпрограм-

20 Свойства Top и Left определяют расположение компонента относительно верхнего левого угла формы.

мы запишем несколько команд, чтобы получился следующий программный код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
pud, funt: integer; kg: real;
begin
     pud:=StrToInt(Edit1.Text);
     funt:=StrToInt(Edit2.Text);
     kg:=pud*16.38+funt*16.38/40;
     Label4.Caption:='В килограммах:
                              '+FloatToStr(kg);
end;
```

Рассмотрим программный код построчно.

Строка 1. *Заголовок подпрограммы*. Он состоит из ключевого слова `procedure` и имени подпрограммы `TForm1.Button1Click`, а в скобках обычно указывают список параметров подпрограммы, если таковые имеются.

Строка 2. Открывает *блок описания переменных* (`var` – от `variable` переменная);

Строка 3. *Описывает переменные* `pud` и `funt`. Компьютер понимает эту запись как команду выделить в памяти место для хранения двух целых чисел[21].

Строка 4. Описывает одной вещественной переменной `kg`.

Строка 5. Начинает программный блок (`begin` – *начало*).

Строка 6. Команда, выполняющая следующие действия. Из свойства `Text` поля ввода `Edit1` считывает введенную туда информацию. Эта информация воспринимается компилятором как строка текста (например, '360' – это строка из трех символов), а поскольку нам для вычислений нужны числовые значения, то функция `StrToInt` преобразовывает ее в целое число. Результат этих преобразований записывается в память компьютера под именем `pud`. Символы «:=» обозначают *оператор присваивания*. Например, запись `a:=3.14` читается так: переменной `a` присвоить значение `3.14`. Выражение `3.14:=a` не имеет смысла.

Строка 7. Информация из поля ввода `Edit2` преобразовывается в целое число и записывается в переменную `funt`. Фактически ко-

21 О переменных и их типах подробнее рассказывается в следующей главе.

манды, записанные в шестой и седьмой строках, осуществляют *ввод исходных данных*.

Строка 8. *Вычисляется значение выражения,* результат которого присваивается переменной `kg`. Обратите внимание, что *умножение* здесь обозначается звездочкой, *деление* – наклонной чертой, *сложение* – знаком «+». Записывая математическое выражение на языке программирования, нужно четко указывать все операции. Нельзя, например, опустить знак умножения, как это принято в математике.

Строка 9. Изменяет свойство `Caption` объекта `Label4`. В него будет помещен текст, состоящий из двух строк. Первая – `'В килограммах:'`, вторая значение переменной `kg`. Знак «+» в этом выражении применяют для конкатенации (слияния) строк. Кроме того, поскольку значение переменной `kg` – вещественное число, оно преобразовывается в строку функцией `FloatToStr`. Фактически в этой строке был выполнен *вывод результатов*.

Строка 10. Конец программного блока (`end` – *конец*).

Теперь наша программа может быть откомпилирована и запущена на выполнение. Процесс *компиляции* представляет собой перевод программы с языка программирования в машинные коды. Для компиляции нужно выполнить команду **Запуск — Быстрая компиляция**. Во время компиляции проверяется наличие синтаксических ошибок. В случае их обнаружения процесс компиляции прерывается. Мы специально допустили ошибку в восьмой строке программного кода (пропустили двоеточие в операторе присваивания). Результат компиляции программы с синтаксической ошибкой показан на рис. 1.49. Строка с ошибкой выделена красным цветом, а под окном редактора появилось окно **Сообщение**[22], в котором описана допущенная ошибка. Ошибку нужно исправить, а программу еще раз откомпилировать.

После успешной компиляции всех файлов, входящих в проект, необходимо *собрать* (скомпоновать) проект из объектных (откомпилированных) файлов (с расширением **.o —** для Linux, .**obj** — для Windows). Для этого следует выполнить команду **Запуск — Собрать** или воспользоваться комбинацией клавиш **Ctrl+F9**.

В результате компоновки будет сгенерирован выполнимый файл, имя которого совпадает с именем проекта.

22 Если окно **Сообщение** отсутствует, его можно вывести на экран командой **Просмотр — Сообщения**.

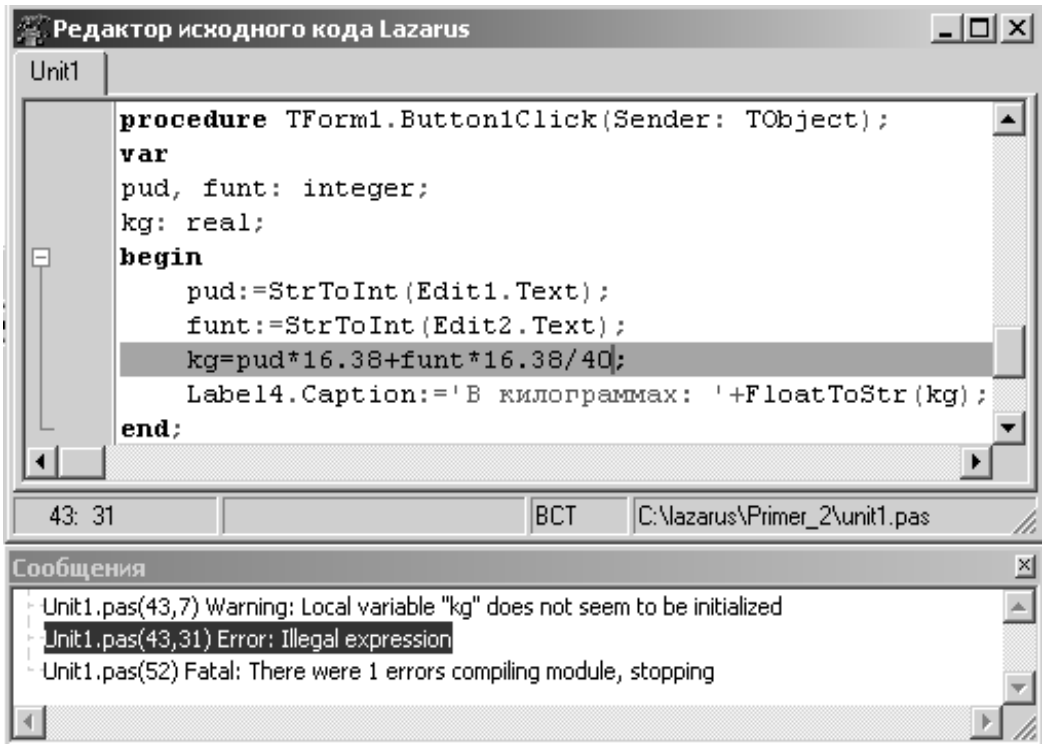

Рисунок 1.49: Сообщение об ошибке в программе

Вообще говоря, если команду **Запуск — Быстрая компиляция** опустить, а сразу выполнить команду **Запуск — Собрать**, то произойдет и компиляция всех файлов проекта и его компоновка.

После успешной компиляции и сборки программу нужно *сохранить* (**Проект - Сохранить**) и *запустить*.

Запуск осуществляется из среды разработки (**Запуск - Запуск**) или выполнением файла, Linux — **./project1**, в Windows — **projec1.exe**. На рис. 1.50 показаны результаты работы нашей программы.

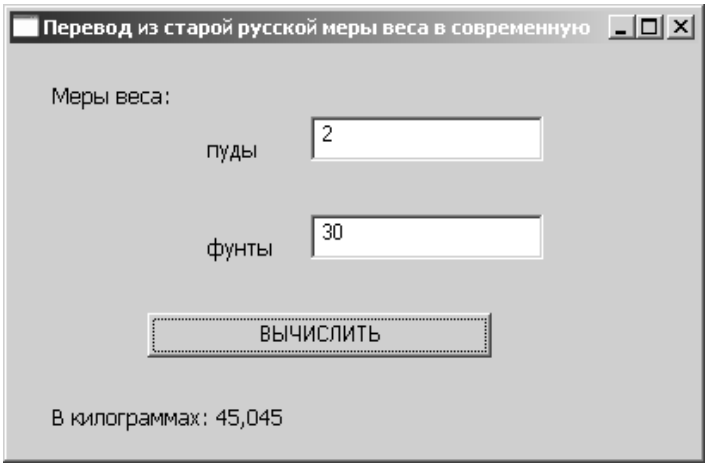

Рисунок 1.50: Результат работы программы

Обратите внимание, что ошибки могут возникать и после запуска программы. Например, в нашем примере исходные данные – целые числа. Если ввести дробь, появится аварийное сообщение. В подобной ситуации следует *прервать выполнение программы* командой **Запуск—Останов** (**Ctrl+F2**).

Вернуться к созданному проекту, например с целью его усовершенствования, несложно. Нужно выполнить команду **Проект — Открыть проект**… (**Ctrl+F11**), перейти к папке с проектом и выбрать из списка файл с расширением **.lpi** (в нашем случае — **project1.lpi**).

1.4.11 Консольное приложение среды Lazarus

В Lazarus можно создавать не только *визуальные приложения* с графическим интерфейсом, но и писать программы на Free Pascal в *текстовом режиме*. Для этого существует *консольное приложение*. Чтобы *создать новое консольное приложение,* нужно выполнить команду **Проект — Создать проект...**, в появившемся диалоговом окне (рис. 1.31, 1.32) выбрать фразу **Программа пользователя** и нажать кнопку **Создать**.

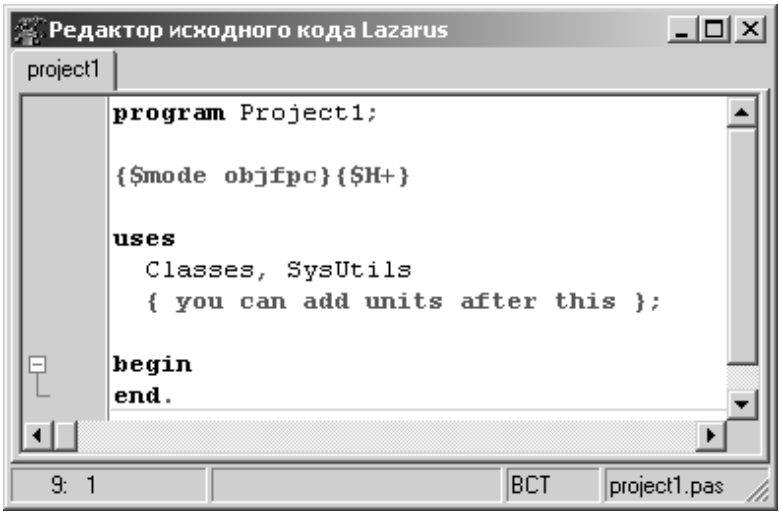

Рисунок 1.51: Окно редактора в консольном приложении

На экране появится окно редактора программного кода (рис. 1.51), в котором уже представлена *общая структура программы* на языке Паскаль.

В общем виде *программа на языке Free Pascal* состоит из заголовка программы, раздела описаний и непосредственно тела программы.

Заголовок программы состоит из ключевого слова `program` и имени программы. В нашем случае (рис. 1.51) это `Project1`. Имя было присвоено программе автоматически. При желании его можно изменить.

Раздел описаний обычно включает в себя описание констант, типов, переменных, процедур и функций. На рис. 1.51 видно, что раздел описаний начинается со слова `uses`. Это раздел подключения модулей. *Модуль* — это специальная программа, которая расширяет возможности языка программирования. В нашем случае происходит подключение модуля `SysUtils`, который позволяет работать с файлами и каталогами, и модуля `Classes`, который работает с компонентами.

За разделом описаний следует исполняемая часть программы, или *тело программы*. Оно начинается со служебного слова `begin` и заканчивается служебным словом `end` и точкой. Тело программы содержит операторы языка, предназначенные для реализации поставленной задачи.

Кроме того в тексте программы могут встречаться комментарии. *Комментарий* — это текст, заключенный в фигурные скобки или начинающийся с двух наклонных. Этот текст не является программным кодом, а носит информационный характер. Например, в нашем случае текст заключенный в фигурные скобки, сразу после описания модулей сообщает пользователю о том, что остальные элементы языка он может добавить самостоятельно.

Рассмотрим **пример**. Пусть нужно решить *задачу перевода градусной меры угла в радианную*. Эта задача известна из школьного курса и формулируется так: чтобы найти радианную меру какого-нибудь угла по данной градусной мере, нужно умножить число градусов на $\frac{\pi}{180}$, число минут на $\frac{\pi}{180 \cdot 60}$ и найденные произведения сложить.

Текст программы для решения поставленной задачи в консольном приложении будет иметь вид:

```
program Project1;
{$mode objfpc}{$H+}
uses
 Classes, SysUtils
     { you can add units after this };
var
 gradus,minuta:integer; radian: real;
begin
 write('gradus=');
 readln(gradus);
 write('minuta=');
 readln(minuta);
 radian:=gradus*pi/180+minuta*pi/(180*60);
 writeln('radian=', radian);
end.
```

Сохранить, *открыть*, *откомпилировать*, *скомпоновать* и *запустить* на выполнение программу в консольном приложении можно так же, как в визуальном проекте.

Результаты работы нашей программы будут иметь вид:

```
gradus=165
minuta=30
radian=2.88851991205O6E+000
```

Нетрудно заметить, что к тексту, созданному автоматически, мы добавили описание переменных (все используемые в программе переменные должны быть описаны):

```
Var
 //Описаны две целочисленные переменные.
 gradus,minuta:integer;
 //Описана вещественная переменная.
 radian: real;
```

и тело программы:

```
begin                 //Начало тела программы.
 //Вывод на экран строки символов gradus=
 write('gradus=');
 //Ввод переменной gradus.
 readln(gradus);
 //Вывод на экран символов minuta=.
 write('minuta=');
 //Ввод переменной minuta.
 readln(minuta);
 //Вычисление.
 radian:=gradus*pi/180+minuta*pi/(180*60);
 //Вывод результата вычислений.
 writeln('radian=', radian);
end. //Конец программы.
```

Так как в этой программе графический интерфейс отсутствует, мы разработали элементарный текстовый диалог компьютер — пользователь. Для этого были использованы операторы *ввода* (`read`) и *вывода* (`write`) данных.

1.4.12 Операторы ввода - вывода данных

Ввод информации с клавиатуры осуществляется с помощью оператора `read`. Он может иметь один из следующих форматов:

```
read (x1, x2, …, xn); readln (x1, x2, …, xn);
```

где `x1`, `x2`, …, `xn` – список вводимых переменных. При вводе вещественных значений целую и дробную часть числа следует разделять точкой.

Когда в программе встречается оператор `read`, ее действие приостанавливается до тех пор, пока не будут введены исходные данные. При вводе числовых значений два числа считаются разделенными, если между ними есть хотя бы один пробел, символ табуляции или конца строки (**Enter**). После ввода последнего значения следует нажать **Enter**.

Оператор `readln` аналогичен оператору `read`, разница заключается в том, что после считывания последнего в списке значения для одного оператора `readln` данные для следующего оператора `readln` будут считываться с начала новой строки. Но следует помнить, что **Enter** переведет курсор на новую строку независимо от того, как именно происходит считывание данных.

Для *вывода информации* на экран служат операторы `write` и `writeln`. В общем случае эти операторы имеют вид:

```
write(x1, x2, …, xn); writeln(x1, x2, …, xn);
```

где `x1`, `x2`, …, `xn` представляют собой список выводимых переменных, констант, выражений. Если элемент списка — текстовая информация, ее необходимо взять в кавычки.

Операторы `write` и `writeln` последовательно выводят все переменные. Если используется оператор `writeln`, то после вывода информации курсор перемещается в новую строку.

Итак, в нашем примере оператор `write('gradus=');` выводит на экран символы `gradus=`, которые подсказывают пользователю, что он должен ввести значение переменной `gradus`, а оператор `readln(gradus);` предназначен для ввода значения переменной `gradus`. Оператор `writeln('radian=', radian);` выводит на экран два значения: строку `radian=` и значение переменной `radian`.

Как правило, вещественные данные выводятся в *формате с плавающей точкой* # . # # # # # # # # # # # # E ± # # #, где # - любая десятичная цифра от 0 до 9. Для того чтобы перейти к *формату с фиксированной точкой* нужно, число, расположенное до символа E, умножить на 10, возведенное в степень, значение которой указано после числа E. Например,

0.35469000000E-01=0.35469000000*10^{-1}=0.035469,

-5.43286710000E+02=-5.43286710000*10^{2}=-543.28671.

Чтобы выводить числа в формате с фиксированной точкой, необходимо использовать *форматированный вывод*. Для этого оператор `write` или `writeln` нужно задать следующим образом:

```
write(идентификатор:ширина_поля_вывода:дробная_часть);
```

где

`идентификатор` — это имя переменной, которая будет выводится на экран;

`ширина_поля_вывода` — количество позиций (целая часть, точка, дробная часть), которое будет занимать значение переменной при выводе;

`дробная_часть` — количество позиций, необходимых для дробной части числа.

Например, пусть результат работы нашей программы имеет вид

```
gradus=165
minuta=30
radian=2.8885199120506E+000
```

Понятно, что оператор `writeln('radian=', radian);` вывел на экран значение переменной `radian` в формате с плавающей точкой.

Если воспользоваться оператором

```
writeln('radian=', radian:5:3);
```

то результат работы программы будет таким:

```
gradus=165
minuta=30
radian=2.889
```

где значение переменной `radian` представлено в формате с фиксированной точкой (все число занимает пять позиций и три из них после запятой).

2 Общие сведения о языке программирования Free Pascal

В этой главе читатель познакомится со структурой проекта в среде Lazarus и основными элементами языка программирования Free Pascal: переменными, константами, их типами, основными операциями и функциями языка.

2.1 Структура проекта Lazarus

Любой *проект* в Lazarus – это совокупность файлов, из которых создается единый *выполняемый файл*. В простейшем случае список файлов проекта имеет вид:

- файл описания проекта (`.lpi`);
- файл проекта (`.lpr`);
- файл ресурсов (`.lrs`);
- модуль формы (`.lfm`);
- программный модуль (`.pas`);

После *компиляции* программы из всех файлов проекта создается единый выполняемый файл, имя этого файла совпадает с именем проекта.

Программный модуль, или просто модуль, – это отдельно компилируемая программная единица, которая представляет собой набор типов данных, констант, переменных, процедур и функций. Любой модуль имеет следующую структуру:

```
unit имя_модуля;    //Заголовок модуля.
interface
//Раздел описаний.
implementation
//Раздел реализаций.
end.              //Конец модуля.
```

Заголовок модуля – это зарезервированное слово `unit`, за которым следует имя модуля и точка с запятой. В разделе описаний, который открывается служебным словом `interface`, описывают *программные элементы* – типы, классы, процедуры и функции:

```
interface
uses список_модулей;
type список_типов;
```

```
const список_констант;
var список_переменных;
procedure имя_процедуры;
…
function имя_функции;
…
```

Раздел `implementation` содержит *программный код*, реализующий механизм работы описанных программных элементов (тексты процедур обработки событий, процедуры и функции, созданные программистом). *Процедуры* и *функции* в Lazarus также построены по модульному принципу[23].

Наряду с визуальными приложениями, Lazarus позволяет разрабатывать и обычные консольные приложения, которые также могут быть созданы в оболочке Free Pascal, и в текстовом редакторе Geany. Авторы настоятельно рекомендуют начинать изучение программирования именно с создания консольных приложений. Поэтому рассмотрим подробно структуру консольного приложения.

2.2 Структура консольного приложения

Структура консольного приложения имеет вид:

```
заголовок программы;
uses modul1, modul2, …, moduln;
раздел описаний;
тело программы.
```

Заголовок программы состоит из служебного слова *program*, имени программы, образованного по правилам использования идентификаторов (см. п. 2.3), и точки с запятой, например:

```
program my_prog001;
```

Предложение `uses modul1, modul2, …, moduln` предназначено для подключения модулей. В модулях находятся функции и процедуры языка. Для использования функций и процедур, находящихся в модуле, необходимо в тексте программы подключить их с помощью предложения `uses`.

Раздел описаний включает следующие подразделы:

```
раздел описания констант;
раздел описания типов;
раздел описания переменных;
```

23 Подробно о процедурах и функциях см. в главе 4.

раздел описания процедур и функций.

В языке Free Pascal должны быть описаны все переменные, типы, константы, которые будут использоваться программой. В стандартном языке Pascal порядок следования разделов в программе жестко установлен, во Free Pascal такого строгого требования нет. В программе может быть несколько разделов описания констант, переменных и т.д., но все они должны быть до тела программы. Более подробно структуру консольной программы на языке Free Pascal можно представить следующим образом:

```
program имя_программы;
uses modul1, modul2, …, moduln;
const описания_констант;
type описания_типов;
var описания_переменных;
begin
    операторы_языка;
end.
```

Тело программы начинается со слова begin, затем следуют операторы языка Pascal, реализующие алгоритм решаемой задачи. Операторы в языке Pascal отделяются друг от друга точкой с запятой и могут располагаться в одну строчку или начинаться с новой строки (в этом случае их также необходимо разделять точкой с запятой). Назначение символа « ; » - отделение операторов друг от друга. Тело программы заканчивается служебным словом end. Несмотря на то что операторы могут располагаться в строке как угодно, рекомендуется размещать их по одному в строке, а в случае сложных операторов отводить для каждого несколько строк. Рассмотрим более подробно структуру программы:

```
program имя_программы;
uses modul1, modul2, …, moduln;
const описания_констант;
type описания_типов;
var описания_переменных;
begin
оператор_1;
оператор_2;
...
end.
```

Приведем пример текста программы на Free Pascal:

```
program one;
const a=7;
var b,c: real;
begin
 c:=a+2; b:=c-a*sin(a)
end.
```

2.3 Элементы языка

Программа на языке Free Pascal может содержать следующие символы:

- латинские буквы A, B, C…, x, y, z;
- цифры 0, 1, 2…, 9;
- специальные символы + ,–, /, =, <, >, [,], ., (,), ;, :, {, }, $, #, _, @, ‘, ^.

Из символов алфавита формируют ключевые слова и идентификаторы. *Ключевые слова* – это зарезервированные слова, которые имеют специальное значение для компилятора. Примером ключевых слов являются операторы языка, типы данных и т.п. Ключевые слова используются только так, как они определены при описании языка. *Идентификатор* – это имя программного объекта[24], представляющее собой совокупность букв, цифр и символа подчеркивания. Первый символ идентификатора – буква или знак подчеркивания, но не цифра. Идентификатор не может содержать пробел. Прописные и строчные буквы в именах не различаются, например `ABC`, `abc`, `Abc` – одно и то же имя. Каждое имя (идентификатор) должно быть уникальным и не совпадать с ключевыми словами.

В тексте программы можно использовать *комментарии*. Комментарий это текст, который компилятор игнорирует. Комментарии используют для пояснений программного кода или для временного отключения команд программного кода при отладке. Комментарии бывают однострочные и многострочные. Однострочный комментарий начинается с двух символов «косая черта» `//` и заканчивается символом перехода на новую строку. Многострочный комментарий заключен в фигурные скобки `{ }` или располагается между парами символов `(*` и `*)`. Понятно, что фигурные скобки `{ }` или символы `(*` и `*)`

24 К программным объектам относятся константы, переменные, метки, процедуры, функции, модули и программы.

можно использовать и для однострочного комментария.

Например:

```
{Комментарий может
 выглядеть так!}
(*Или так.*)
//А если вы используете такой способ,
//то каждая строка должна начинаться
//с двух символов «косая черта».
```

2.4 Данные в языке Free Pascal

Для решения задачи в любой программе выполняется обработка каких-либо данных. Они хранятся в памяти компьютера и могут быть самых различных типов: целыми и вещественными числами, символами, строками, массивами. *Типы данных* определяют способ хранения чисел или символов в памяти компьютера. Они задают размер ячейки, в которую будет записано то или иное значение, определяя тем самым его максимальную величину или точность задания. Область памяти (ячейка), в которой хранится значение определенного типа, называется *переменной*. У переменной есть *имя* (*идентификатор*), *тип* и *значение*. Имя служит для обращения к области памяти, в которой хранится значение. Во время выполнения программы значение переменной можно изменить. Перед использованием любая переменная должна быть описана. Описание переменной в языке Free Pascal осуществляется с помощью служебного слова `var`:

```
var имя_переменной: тип_переменной;
```

Если объявляется несколько переменных одного типа, то описание выглядит следующим образом:

```
var переменная_1,…,переменная_N: тип_переменных;
```

Например:

```
var
//Объявлена целочисленная переменная.
ha: integer;
//Объявлены две вещественные переменные.
hb, c: real;
```

Константа – это величина, которая не изменяет своего значения в процессе выполнения программы. Тип константы определяется ее значением. Описание константы имеет вид:

```
const имя_константы = значение;
```

Например:

```
const
h=3;          //Целочисленная константа.
bk=-7.521;    //Вещественная константа.
c='abcde';    //Символьная константа.
```

Рассмотрим основные типы данных.

2.4.1 Символьный тип данных

Данные *символьного типа* в памяти компьютера всегда занимают один байт. Это связано с тем, что обычно под величину символьного типа отводят столько памяти, сколько необходимо для хранения одного символа.

Описывают символьный тип с помощью служебного слова `char`. Например:

```
var
 c: char;
```

В тексте программы значения переменных и константы символьного типа должны быть заключены в апострофы: `'a'`, `'b'`, `'+'`.

2.4.2 Целочисленный тип данных

Целочисленные типы данных могут занимать в памяти компьютера один, два, четыре или восемь байт. Диапазоны значений данных целочисленного типа представлены в табл. 2.1.

Таблица 2.1 Целочисленные типы данных

Тип	Диапазон	Размер
`Byte`	0 .. 255	1 байт
`Word`	0 .. 65535	2 байта
`LongWord`	0 .. 4294967295	4 байта
`ShortInt`	-128 .. 127	1 байт
`Integer`	-2147483648 .. 2147483647	4 байта
`LongInt`	-2147483648 .. 2147483647	4 байта
`Smallint`	-32768 .. 32767	2 байта
`Int64`	-2^{63} .. 2^{63}	8 байт
`Cardinal`	0 .. 4294967295	4 байта

Описание целочисленных переменных в программе может быть таким:

```
var
b: byte;
i, j: integer;
W: word;
L_1, L_2: longint;
```

2.4.3 Вещественный тип данных

Внутреннее представление *вещественного числа* в памяти компьютера отличается от представления целого числа. Оно представлено в экспоненциальной форме $m\text{E} \pm p$, где m – мантисса (целое или дробное число с десятичной точкой), p – порядок (целое число)[25]. Чтобы перейти от экспоненциальной формы к обычному представлению числа[26], необходимо мантиссу умножить на десять в степени порядок. Например:

$-36.142\text{E} + 2 = -36.142 \cdot 10^2 = -3614.2$;

$7.25\text{E} - 5 = 7.25 \cdot 10^{-5} = 0.0000725$.

Вещественное число в Pascal может занимать от четырех до десяти байтов. Диапазоны значений вещественного типа представлены в табл. 2.2.

Таблица 2.2 Вещественные типы данных

Тип	Диапазон	Количество значащих цифр	Размер
`Single`	1.5E-45 .. 3.4E+38	7 — 8	4 байта
`Real`	2.9E-39 .. 1.7E+38	15 — 16	8 байтов
`Double`	5.0E-324 .. 1.7E+308	15 — 16	8 байтов
`Extended`	3.4E-4932 .. 3.4E+4932	19 — 20	10 байтов
`Comp`	-2^{63} .. 2^{63}	19 — 20	8 байтов
`Currency`	-922337203685477.5808 .. 922337203685477.5807	19 — 20	8 байтов

Примеры описания вещественных переменных:

```
var
r1, r2: real; D: double;
```

25 Действия над числами, представленными в экспоненциальной форме, называют арифметикой с плавающей точкой, так как положение десятичной точки меняется в зависимости от порядка числа.

26 Число в обычном его представлении называют числом с фиксированной точкой.

2.4.4 Тип дата-время

Тип данных дата-время `TDateTime` предназначен для одновременного хранения даты и времени. В памяти компьютера он занимает восемь байтов и фактически представляет собой вещественное число, где в целой части хранится дата, а в дробной – время.

2.4.5 Логический тип данных

Данные *логического типа* могут принимать только два значения: истина (`true`) или ложь (`false`). В стандартном языке Pascal был определен лишь один логический тип данных – `boolean`. Логические типы данных, определенные в среде Lazarus, представлены в табл. 2.3.

Таблица 2.3 Логические типы данных

Тип	Размер
`Boolean`	1 байт
`ByteBool`	1 байт
`WordBool`	2 байта
`LongBool`	4 байта

Пример объявления логической переменной:

```
var
 FL: boolean;
```

2.4.6 Создание новых типов данных

Несмотря на достаточно мощную структуру встроенных типов данных, в среде Lazarus предусмотрен механизм создания *новых типов*. Для этого используют служебное слово `type`:

```
type новый_тип_данных = определение_типа;
```

Когда новый тип данных создан, можно объявлять соответствующие ему переменные:

```
var список_переменных: новый_тип_данных;
```

Применение этого механизма мы рассмотрим в следующих параграфах.

2.4.7 Перечислимый тип данных

Перечислимый тип задается непосредственным перечислением значений, которые он может принимать:

```
var имя_переменной:(знач_1,знач_2,…,знач_N);
```

Такой тип может быть полезен, если необходимо описать данное, которое принимает ограниченное число значений. Например:

```
var
animal: (fox, rabbit);
color: (yellow, blue, green);
```

Применение перечислимых типов делает программу нагляднее:

```
//Создание нового типа данных – времена года.
type
 year_times = (winter, spring, summer, autumn);
//Описание соответствующей переменной.
var  yt: year_times;
```

2.4.8 Интервальный тип

Интервальный тип задается границами своих значений внутри базового типа:

```
var  имя_переменной: мин_знач .. макс_знач;
```

Обратите внимание, что в данной записи два символа точки рассматриваются как один, поэтому пробел между ними не допускается. Кроме того, границы диапазона могут быть только целого или символьного типов и левая граница диапазона не должна превышать правую. Например:

```
var
 date: 1..31;
 symb: 'a'..'h';
```

Применим механизм создания нового типа данных:

```
type
//Создание перечислимого типа данных –
//дни недели.
Week_days = (Mo, Tu, We, Th, Fr, Sa, Su);
//Создание интервального типа данных –
//рабочие дни.
Working_days = Mo.. Fr
//Описание соответствующей переменной.
var
 days: Working_days;
```

2.4.9 Структурированные типы

Структурированный тип данных характеризуется множественностью образующих его элементов. В языке Free Pascal это массивы, строки, записи, множества и файлы.

Массив – совокупность данных одного и того же типа[27]. Число

27 Подробно о массивах см. в главе 5.

элементов массива фиксируется при описании типа и в процессе выполнения программы не изменяется. Для описания массива используют ключевые слова `array … of`:

```
имя: array [список_индексов] of тип_данных;
```

где:

• `имя` – любой допустимый идентификатор;

• `тип_данных` – любой тип языка.

• `список индексов` – перечисление диапазонов изменения номеров элементов массива; количество диапазонов совпадает с количеством измерений массива; диапазоны отделяются друг от друга запятой, а границы диапазона, представляющие собой интервальный тип данных, отделяют друг от друга двумя символами точки:

```
[индекс1_начальный..индекс1_конечный,
    индекс2_начальный..индекс2_конечный, …, ]
```

Например:

```
var
//Одномерный массив из 10 целых чисел.
a:array [1..10] of byte;
//Двумерный массив вещественных чисел
//(3 строки, 3 столбца).
b:array [1..3, 1..3] of real;
```

Еще один способ описать массив – создать новый тип данных. Например, так:

```
type
//Объявляется новый тип данных –
//трехмерный массив целых чисел.
massiv=array [0..4, 1..2, 3..5] of word;
//Описывается переменная соответствующего типа.
var
M: massiv;
```

Для *доступа к элементу массива* достаточно указать его порядковый номер, а если массив многомерный (например, таблица), то несколько номеров:

```
имя_массива[номер_элемента]
```

Например: `a[5]`, `b[2, 1]`, `M[3, 2, 4]`.

Строка – последовательность символов. В Lazarus строка трактуется как массив символов, то есть каждый символ строки пронумерован, начиная с единицы.

При использовании в выражениях строка заключается в апострофы. Описывают переменные строкового типа так:

```
имя_переменной: string;
```

или:

```
имя_переменной: string[длина_строки];
```

Например:

```
const S='СТРОКА';
var
 Str1: string;
 Str2: string[255];
 Stroka: string[100];
```

Если при описании строковой переменной длина строки не указывается, это означает, что она составляет 255 символов. В приведенном примере строки `Str1` и `Str2` одинаковой длины.

Запись[28] – это структура данных, состоящая из фиксированного количества компонентов, называемых полями записи. В отличие от массива поля записи могут быть разного типа. При объявлении типа записи используют ключевые слова `record … end`:

```
имя_записи = record список_полей end;
```

здесь, `имя_ записи` – любой допустимый идентификатор, `список_полей` – описания полей записи. Например:

```
//Описана структура записи.
//Каждая запись содержит информацию
//о студенте и состоит из двух полей –
//имя и возраст.
type
 student = record
     name: string;
     age: byte;
     end;
var
//Объявлены соответствующие переменные –
//три объекта типа запись.
 a, b, c: student;
```

Доступ к полям записи осуществляется с помощью составного имени:

```
имя_записи.имя_поля
```

28 Подробно о структурах см. в главе .

Например:

```
a.name:='Ivanov Ivan';
a.age:=18;
b.name:=a.name;
```

Оператор присоединения `with имя_записи do` упрощает доступ к полям записи:

```
with a do
begin
name:='Petrov Petr';
age:=19;
end;
```

Множество – это набор логически связанных друг с другом объектов. Количество элементов множества может изменяться от 0 до 255. Множество, не содержащее элементов, называется пустым. Для описания множества используют ключевые слова `set of`:

```
имя_множества = set of базовый_тип_данных;
```

Например:

```
type  TwoNumbers = set of 0..1;
var   Num1, Num2, Num3: TwoNumbers;
```

Файл[29] – это именованная область внешней памяти компьютера. Файл содержит компоненты одного типа (любого типа, кроме файлов). Длина созданного файла не оговаривается при его объявлении и ограничивается только емкостью диска, на котором он хранится. В Lazarus можно объявить *типизированный файл*:

```
имя_файловой_переменной = file of тип_данных;
```

бестиповый файл:

```
имя_файловой_переменной = file;
```

и *текстовый файл*:

```
имя_файловой_переменной = TextFile;
```

Например:

```
var
f1: file of byte;
f2: file;
f3: TextFile;
```

2.4.10 Указатели

Как правило, при обработке оператора объявления переменной

```
имя_переменной: тип_переменной
```

29 Подробно о файлах см. в главе 7 .

компилятор автоматически выделяет память под переменную имя_переменной в соответствии с указанным типом. Доступ к объявленной переменной осуществляется по ее имени. При этом все обращения к переменной заменяются адресом ячейки памяти, в которой хранится ее значение. При завершении подпрограммы, в которой была описана переменная, память автоматически освобождается.

Доступ к значению переменной можно получить иным способом – определить собственные переменные для хранения адресов памяти. Такие переменные называют указателями.

Указатель[30] – это переменная, значением которой является адрес памяти, где хранится объект определенного типа (другая переменная). Как и любая переменная, указатель должен быть объявлен. При объявлении указателей всегда указывается тип объекта, который будет храниться по данному адресу:

```
var имя_переменной: ^тип;
```

Такие указатели называют *типизированными*. Например:

```
var p: ^integer;
//По адресу, записанному в переменной p
//будет храниться переменная типа int
//или, другими словами p, указывает
//на тип данных integer.
```

В языке Free Pascal можно объявить указатель, не связанный с каким-либо конкретным типом данных. Для этого применяют служебное слово pointer:

```
var имя_переменной: pointer;
```

Например:

```
var x, y: pointer;
```

Подобные указатели называют *нетипизированными* и используют для обработки данных, структура и тип которых меняется в процессе выполнения программы, то есть динамически.

2.5 Операции и выражения

Выражение задает порядок выполнения действий над данными и состоит из операндов (констант, переменных, обращений к функциям), круглых скобок и знаков операций, например

```
a+b*sin(cos(x)).
```

30 Подробно об указателях см. 5.11

В табл. 2.4 представлены *основные операции* языка программирования Free Pascal.

Таблица 2.4. Основные операции языка Free Pascal

Операция	**Действие**	**Тип операндов**	**Тип результата**
+	сложение	целый/вещественный	целый/вещественный
+	сцепление строк	строковый	строковый
−	вычитание	целый/вещественный	целый/вещественный
*	умножение	целый/вещественный	целый/вещественный
/	деление	целый/вещественный	вещественный
div	целочисленное деление	целый	целый
mod	остаток от деления	целый	целый
not	отрицание	целый/логический	целый/логический
and	логическое И	целый/логический	целый/логический
or	логическое ИЛИ	целый/логический	целый/логический
xor	исключающее ИЛИ	целый/логический	целый/логический
shl	сдвиг влево	целый	целый
shr	сдвиг вправо	целый	целый
in	вхождение в множество	множество	логический
<	меньше	числовой/символьный/логический	логический
>	больше	числовой/символьный/логический	логический
<=	меньше или равно	числовой/символьный/логический	логический
>=	больше или равно	числовой/символьный/логический	логический
=	равно	числовой/символьный/логический	логический
<>	не равно	числовой/символьный/логический	логический

В сложных выражениях порядок, в котором выполняются операции, соответствует приоритету операций. В языке Free Pascal приняты следующие приоритеты:

1. `not`.
2. `*`, `/`, `div`, `mod`, `and`, `shl`, `shr`.
3. `+`, `-`, `or`, `xor`.
4. =, <>, >, <, >=, <=.

Использование скобок в выражениях позволяет менять порядок вычислений.

Перейдем к подробному рассмотрению основных операций.

2.5.1 Арифметические операции

Операции `+`, `-`, `*`, `/` относят к *арифметическим операциям*. Их назначение понятно и не требует дополнительных пояснений.

Операции целочисленной арифметики (применяется только к целочисленным операндам):

• `div` – целочисленное деление (возвращает целую часть частного, дробная часть отбрасывается), например, 17 `div` 10=1;

• `mod` – остаток от деления, например, 17 `mod` 10=7.

К *операциям битовой арифметики* относятся следующие операции: `and`, `or`, `xor`, `not`, `shl`, `shr`. В операциях битовой арифметики действия происходят над двоичным представлением целых чисел.

Арифметическое И (`and`)[31]. Оба операнда переводятся в двоичную систему, затем над ними происходит логическое поразрядное умножение операндов по следующим правилам:

1 `and` 1=1, 1 `and` 0=0, 0 `and` 1=0, 0 `and` 0=0.

Например, если A=14 и B=24, то их двоичное представление – A=0000000000001110 и B=0000000000011000. В результате логического умножения A `and` B получим 0000000000001000 или 8 в десятичной системе счисления (рис. 2.1). Таким образом, A&B=14&24=8.

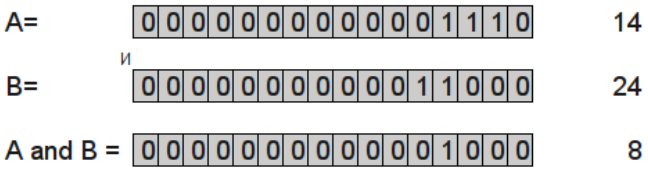

Рисунок 2.1: Пример логического умножения A and B

31 Конъюнкция.

Арифметическое ИЛИ (`or`)[32]. Здесь также оба операнда переводятся в двоичную систему, после чего над ними происходит логическое поразрядное сложение операндов по следующим правилам:

1 `or` 1=1, 1 `or` 0=1, 0 `or` 1=1, 0 `or` 0=0.

Например, результат логического сложения чисел A=14 и B=24 будет равен A `or` B=30 (рис. 2.2).

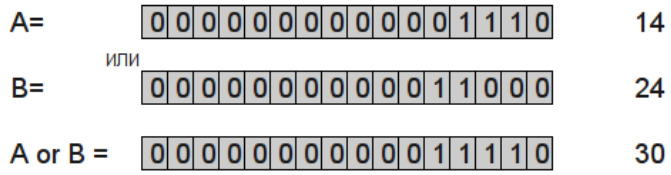

Рисунок 2.2: Пример логического сложения A or B

Арифметическое исключающее ИЛИ (`xor`). Оба операнда переводятся в двоичную систему, после чего над ними происходит логическая поразрядная операция `xor` по следующим правилам:

1 `xor` 1=0, 1 `xor` 0=1, 0 `xor` 1=1, 0 `xor` 0=0.

Арифметическое отрицание (`not`). Эта операция выполняется над одним операндом. Применение операции `not` вызывает побитную инверсию двоичного представления числа (рис. 2.3).

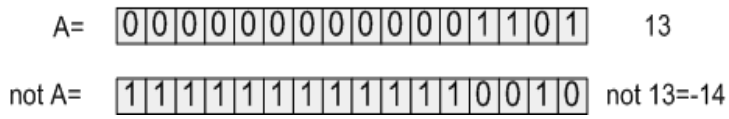

Рисунок 2.3: Пример арифметического отрицания not A

Сдвиг влево (`M shl L`). Двоичное представление числа M сдвигается влево на L позиций. Рассмотрим операцию 15 `shl` 3. Число 15 в двоичной системе имеет вид 1111. При сдвиге его на 3 позиции влево получим 1111000. В десятичной системе это двоичное число равно 120. Итак, 15 `shl` 3 =120 (рис. 2.4).

Рисунок 2.4: Сдвиг влево 15 shl 3

32 Дизъюнкция

Заметим, что сдвиг на один разряд влево соответствует умножению на два, на два разряда – умножению на четыре, на три – умножению на восемь. Таким образом, операция M `shl` L эквивалентна умножению числа M на 2 в степени L.

Сдвиг вправо (`M shr L`). В этом случае двоичное представление числа M сдвигается вправо на L позиций, что эквивалентно целочисленному делению числа M на 2 в степени L. Например, 15 `shr` 1=7 (рис. 2.5), 15 `shr` 3= 2.

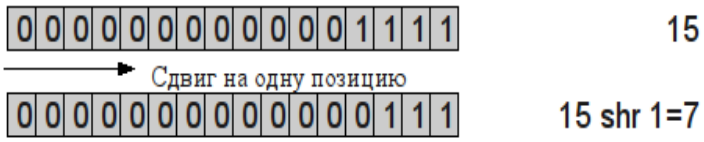

Рисунок 2.5: Сдвиг вправо 15 shr 1

2.5.2 Операции отношения

Операции отношения применяются к двум операндам и возвращают в качестве результата логическое значение. Таких операций семь: >, >=, <, <=, =, <>, `in`. Результат операции отношения – логическое значение `true` (истина) или `false` (ложь).

Назначение операций >, >=, <, <=, =, <> понятно (см. табл. 2.4). Поясним, как работает операция `in`. Первым операндом этой операции должно быть любое выражение, вторым – множество, состоящее из элементов того же типа. Результат операции `true` (истина), если левый операнд принадлежит множеству, указанному справа.

2.5.3 Логические операции

В языке Free Pascal определены следующие *логические операции* `or`, `and`, `xor`, `not`. Логические операции выполняются над логическими значениями `true` (истина) и `false` (ложь). В табл. 2.5 приведены результаты логических операций.

Таблица 2.5. Логические операции

A	B	Not A	A and B	A or B	A xor B
t	t	f	t	t	f
t	f	f	f	t	t
f	t	t	f	t	t
f	f	t	f	f	f

В логических выражениях могут использоваться операции отношения, логические и арифметические.

2.5.4 Операции над указателями

При работе с указателями используют операции получения адреса и разадресации.

Операция получения адреса `@` возвращает адрес своего операнда. Например:

```
Var
a:real; //Объявлена вещественная переменная a
adr_a:^real; //Объявлен указатель на тип real
…
//Оператор записывает в переменную adr_a
//адрес переменной a
adr_a:=@a;
```

Операция разадресации `^` возвращает значение переменной, хранящееся по заданному адресу:

```
Var
a:real; //Объявлена вещественная переменная a
adr_a:^real; //Объявлен указатель на тип real
…
//Оператор записывает в переменную a
//значение, хранящееся по адресу adr_a.
a:=adr_a^;
```

2.6 Стандартные функции

В языке Free Pascal определены *стандартные функции* над арифметическими операндами (табл. 2.6):

Таблица 2.6. Некоторые арифметические функции

Обозначение	Тип аргументов	Тип результата	Действие
`abs(x)`	целый/вещественный	целый/вещественный	модуль числа
`sin(x)`	вещественный	вещественный	синус
`cos(x)`	вещественный	вещественный	косинус
`arctan(x)`	вещественный	вещественный	арктангенс
`pi`	без аргумента	вещественный	число π
`exp(x)`	вещественный	вещественный	экспонента e^x
`ln(x)`	вещественный	вещественный	натуральный логарифм

Обозначение	Тип аргументов	Тип результата	Действие
`sqr(x)`	вещественный	вещественный	квадрат числа
`sqrt(x)`	вещественный	вещественный	корень квадратный
`int(x)`	вещественный	вещественный	целая часть числа
`frac(x)`	вещественный	вещественный	дробная часть числа
`round(x)`	вещественный	целый	округление числа
`trunc(x)`	вещественный	целый	отсекание дробной части числа
`random(n)`	целый	целый	случайное число от 0 до n
Функции, определенные в программном модуле Math[33]			
`arccos(x)`	вещественный	вещественный	арккосинус
`arcsin(x)`	вещественный	вещественный	арксинус
`arccot(x)`	вещественный	вещественный	арккотангенс
`arctan2(y,x)`	вещественный	вещественный	арктангенс y/x
`cosecans(x)`	вещественный	вещественный	косеканс
`sec(x)`	вещественный	вещественный	секанс
`cot(x)`	вещественный	вещественный	котангенс
`tan(x)`	вещественный	вещественный	тангенс
`lnXP1(x)`	вещественный	вещественный	логарифм натуральный от (x+1)
`log10(x)`	вещественный	вещественный	десятичный логарифм
`log2(x)`	вещественный	вещественный	логарифм по основанию два
`logN(n,x)`	вещественный	вещественный	логарифм от x по основанию n

Определенную проблему представляет возведение X в степень n. Если значение степени `n` – целое положительное число, то можно n раз перемножить X (что дает более точный результат и при целом `n` предпочтительней) или воспользоваться формулой[34]:

33 Эти функции будут работать только в том случае, если в тексте основной программы после ключевого слова Unit указать имя Math.

34 Формула формируется так: логарифмируем выражение x^n, получается n ln(x), экспоненцируем последнее.

$$\begin{cases} X^n = e^{n\cdot\ln(X)}, X>0 \\ X^n = -e^{n\cdot\ln(X)}, X<0 \end{cases},$$

которая программируется с помощью стандартных функций языка:

- `exp(n*ln(x))` – для положительного *X*;
- `-exp(n*ln(abs(x)))` – для отрицательного *X*.

Данную же формулу можно использовать для возведения *X* в дробную степень *n*, где *n* – обыкновенная правильная дробь вида *k*/*l*, а знаменатель *l* — нечетный. Если знаменатель *l* – четный, это означает извлечение корня четной степени, следовательно, есть ограничения на выполнение операции: подкоренное выражение не должно быть отрицательным.

При возведении числа *X* в отрицательную степень следует помнить, что

$$x^{-n} = \frac{1}{x^n} .$$

Таким образом, для программирования выражения, содержащего возведение в степень, надо внимательно проанализировать значения, которые могут принимать *X* и *n*, так как в некоторых случаях возведение *X* в степень *n* невыполнимо.

Некоторые *функции*, предназначенные для *работы со строками*, представлены в табл. 2.7.

Таблица 2.7. Функции обработки строк

Обозначение	**Тип аргументов**	**Тип результата**	**Действие**
Работа со строками			
`length(S)`	строка	целое	текущая длина строки `S`
`concat(S1,S2, …)`	строки	строка	объединение строк `S1`, `S2`, …
`copy(S, n, m)`	строка, целое, целое	строка	копирование `n` символов строки `S` начиная с `m`-й позиции
`delete(S, n, m)`	строка, целое, целое	строка	удаление `n` символов из строки `S` начиная с `m`-й позиции

Обозначение	Тип аргументов	Тип результата	Действие
`insert(S, n, m)`	строка, целое, целое	строка	вставка `n` символов в строку `S` начиная с `m`-й позиции
`pos(S1, S2)`	строки	целое	номер позиции, с которой начинается вхождение `S2` в `S1`
`chr(x)`	целое	символ	возвращает символ с кодом `x`
`ord(c)`	символ	целое	возвращает код символа `c`
Преобразование строк в другие типы			
`StrToDateTame(S)`	строка	дата и время	преобразует символы из строки `s` в дату и время
`StrToFloat(S)`	строка	вещественное	преобразует символы из строки `s` в вещественное число
`StrToInt(S)`	строка	целое	преобразует символы из строки `s` в вещественное число
`Val(S,X,Kod)`	строка		преобразует строку символов `S` во внутреннее представление числовой переменной `X`, если преобразование прошло успешно, `Kod`=0.
Обратное преобразование			
`DataTimeToStr(V)`	дата и время	строка	преобразует дату и время в строку.
`FloatToStr(V)`	вещественное	строка	преобразует вещественное число в строку
`IntToStr(V)`	целое	строка	преобразует целочисленное число в строку
`FloatToStrF( V, F,P,D)`	вещественное	строка	преобразует вещественное число `V` в строку символов с учетом формата `F` и параметров `P`, `D`.

Поясним назначение функции `FloatToStrF(V,F,P,D)`. Обычно ее используют для форматированного вывода вещественного числа. В табл. 2.8 содержатся значения параметров этой функции.

Таблица 2.8. Параметры функции `FloatToStrF`

Формат	Назначение
`ffExponent`	Экспоненциальная форма представления числа, *P* – мантисса, *D* – порядок: 1.2345E+10.
`ffFixed`	Число в формате с фиксированной точкой, *P* – общее количество цифр в представлении числа, *D* – количество цифр в дробной части: 12.345.
`ffGtneral`	Универсальный формат, использует наиболее удобную форму представления числа.
`ffNumber`	Число в формате с фикс. точкой, использует символ разделителя тысяч при выводе больших чисел.
`ffCurency`	Денежный формат, соответствует `ffNumber`, но в конце ставит символ денежной единицы.

Примером работы функции `FloatToStrF` служит фрагмент программы:

```
var
n:integer;
m:real;
St:string;
begin
n:=5;
m:=4.8;
St:='Иванов А.';
//Выражение chr(13) – символ окончания строки.
//Для вывода вещественного числа m отводится
//четыре позиции вместе с точкой,
//две позиции после точки.
Label1.Caption:='Студент'+St+'сдал '
        +IntToStr(n)+'экзаменов.'+chr(13)+
                'Средний балл составил '+
                   FloatToStrF(m,ffFixed,4,2);
End;
```

Результатом работы программы будет фраза:

```
Студент Иванов А. сдал 5 экзаменов.
Средний балл составил 4.80.
```

В табл. 2.9 приведены функции, предназначенные для работы с датой и временем.

Таблица 2.9. Функции для работы с датой и временем

Обозначение	Тип аргументов	Тип результата	Действие
`date`	без аргумента	дата-время	возвращает текущую дату
`now`	без аргумента	дата-время	возвращает текущую дату и время
`time`	без аргумента	дата-время	возвращает текущее время

Если в процессе работы над программой возникает необходимость в создании переменной, размер которой неизвестен заранее, используют *динамическую память*[35].

В языке Free Pascal *операции распределения памяти* осуществляются по средством функций представленных в табл. 2.10.

Таблица 2.10. Функции для работы с памятью

Обозначение	Действие
`adr(x)`	Возвращает адрес аргумента `x`.
`dispose(p)`	Возвращает фрагмент динамической памяти, который ранее был зарезервирован за типизированным указателем `p`.
`Freemen(p,size)`	Освобождает фрагмент динамической памяти, который ранее был зарезервирован за бестиповым указателем `p`.
`GetMem(p,size)`	Резервирует за бестиповым указателем p фрагмент динамической памяти размера `size.`
`New(p)`	Резервирует фрагмент динамической памяти для размещения переменной и помещает в типизированный указатель `p` адрес первого байта.
`SizeOf(x)`	Возвращает длину в байтах внутреннего представления указанного объекта `x`.

С подробным описанием приведенных в этой главе функций и множеством других функций можно ознакомиться в справочной системе Lazarus.

Рассмотрим решение задачи с использованием стандартных функций.

ЗАДАЧА 2.1. Известны длины сторон треугольника a, b и c. Вычислить площадь S, периметр P и величины углов α, β и γ треугольника (рис. 2.6).

35 Динамическая память – это область оперативной памяти, которая выделяется программе при ее работе.

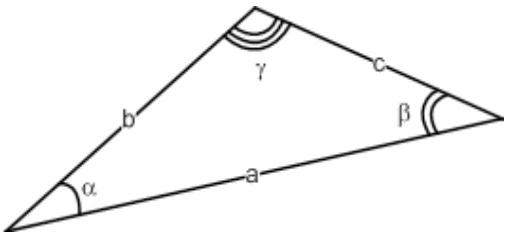

Рисунок 2.6: Иллюстрация к задаче 2.1

Прежде чем приступить к написанию программы, вспомним математические формулы, необходимые для решения задачи.

Для вычисления площади треугольника применим теорему Герона:

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

где полупериметр:

$$r=\frac{a+b+c}{2}\ ;$$

один из углов найдем по теореме косинусов:

$$cos(\alpha)=\frac{b^2+c^2-a^2}{2\cdot b\cdot c}\ ;$$

второй — по теореме синусов:

$$sin(\beta)=\frac{b}{a}\cdot\sin(a)\ ;$$

третий — по формуле:

$$\gamma=\pi-(\alpha+\beta);$$

Решение задачи можно разбить на следующие этапы:

1. Определение значений *a*, *b* и *c* (ввод величин *a, b* и *c* в память компьютера).
2. Расчет значений *S*, *P*, *α, β и γ* по приведенным формулам.
3. Вывод значений *S*, *P*, *α, β и γ*.

Процесс разработки несложного программного интерфейса описан в главе 1. Попытайтесь самостоятельно разработать внешний вид данной программы. Разместите на форме десять меток, три поля ввода и одну кнопку. Измените их заголовки (свойство `Caption`) в соответствии с табл. 2.11. В результате форма должна выглядеть так, как показано на рис. 2.7.

Таблица 2.11. Заголовки компонентов формы (рис. 2.7)

Компонент	**Свойство** `Caption`
`Form1`	`Параметры треугольника`
`Label1`	`Введите длины сторон`
`Label2`	`a=`
`Label3`	`b=`
`Label4`	`c=`
`Label5`	`Величины углов`
`Label6`	`alfa=`
`Label7`	`betta=`
`Label8`	`gamma=`

Компонент	**Свойство** Caption
Label9	Периметр P=
Label10	Площадь S=
Button1	ВЫЧИСЛИТЬ

Рисунок 2.7: Проект формы к задаче 2.1

Итак, проект формы готов. В окне программного кода среды Lazarus автоматически сформировал структуру модуля, перечислив названия основных разделов. Двойной щелчок по кнопке Вычислить приведет к созданию процедуры

TForm1.Button1Click

в разделе implementation:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

и ее описанию в разделе interface. Понятно, что созданная процедура не содержит ни одной команды. Задача программиста заполнить шаблон описаниями и операторами. Все команды, указанные в процедуре между словами begin и end, будут выполнены при щелчке по кнопке Выполнить.

В нашем случае процедура TForm1.Button1Click будет иметь вид:

```
procedure TForm1.Button1Click(Sender: TObject);
//Описание переменных:
//   a, b, c – стороны треугольника;
//   alfa, betta, gamma – углы треугольника;
//       S - площадь треугольника;
//       r - полупериметр треугольника
//Все переменные вещественного типа.
var a, b, c, alfa, betta, gamma, S, r: real;
begin
//Из полей ввода Edit1, Edit2, Edit3
//считываются введенные строки,
```

```
//с помощью функции StrToFloat(x)
//преобразовываются в вещественные числа
//и записываются в переменные a, b, c.
a:=StrToFloat(Edit1.Text);
b:=StrToFloat(Edit2.Text);
c:=StrToFloat(Edit3.Text);
//Вычисление значения полупериметра.
R:=(a+b+c)/2;
//Вычисление значения площади,
//для вычисления применяется функция:
//  sqrt(x) – корень квадратный из x.
S:=sqrt(r*(r-a)*(r-b)*(r-c));
//Вычисление значения угла alfa в радианах.
//Для вычисления применяем функции:
//         arccos(x) - арккосинус x;
//         sqr(x) – возведение x в квадрат.
alfa:=arccos((sqr(b)+sqr(c)-sqr(a))/2/b/c);
//Вычисление значения угла betta в радианах.
//Для вычисления применяем функции:
//          arcsin(x) - арксинус x;
betta:=arcsin(b/a*sin(alfa));
//Вычисление значения угла gamma в радианах.
//Математическая постоянная определена
//функцией без аргумента pi.
gamma:=pi-(alfa+betta);
//Перевод радиан в градусы.
alfa:=alfa*180/pi;
betta:=betta*180/pi;
gamma:=gamma*180/pi;
//Для вывода результатов вычислений используем
//операцию слияния строк «+»
//и функцию FloatToStrF(x), которая
//преобразовывает вещественную переменную x
//в строку и выводит ее в указанном формате,
//в нашем случае под переменную отводится
//три позиции, включая точку
//и ноль позиций после точки.
//Величины углов в градусах выводятся на форму
```

```
//в соответствующие объекты типа надпись.
Label6.Caption:='alfa='+
                FloatToStrF(alfa,ffFixed,3,0);
Label7.Caption:='betta='+
                FloatToStrF(betta,ffFixed,3,0);
Label8.Caption:='gamma='+
                FloatToStrF(gamma,ffFixed,3,0);
//Используем функцию FloatToStrF(x)
//для форматированного вывода, в нашем случае
//под все число отводится пять позиций,
//включая точку, и две позиций после точки.
//Значения площади и периметра
//выводятся на форму.
Label9.Caption:='Периметр P='+
                FloatToStrF(2*r,ffFixed,5,2);
Label10.Caption:='Площадь S='+
                FloatToStrF(S,ffFixed,5,2);
end;
```

Обратите внимание, что было написано всего десять команд, предназначенных для решения поставленной задачи. Остальной текст в окне редактора создается автоматически. В результате весь программный код имеет вид:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
Classes, SysUtils, LResources, Forms, Controls,
             Graphics, Dialogs, StdCtrls, Math;
type
 { TForm1 }
 TForm1 = class(TForm)
     Button1: TButton;
     Edit1: TEdit;
     Edit2: TEdit;
     Edit3: TEdit;
     Label1: TLabel;
     Label10: TLabel;
     Label2: TLabel;
```

```
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    Label9: TLabel;
 procedure Button1Click(Sender: TObject);
 private
    { private declarations }
 public
    { public declarations }
 end;
var
 Form1: TForm1;
implementation
{ TForm1 }
procedure TForm1.Button1Click(Sender: TObject);
 var a, b, c, alfa, betta, gamma, S, r: real;
 begin
    a:=StrToFloat(Edit1.Text);
    b:=StrToFloat(Edit2.Text);
    c:=StrToFloat(Edit3.Text);
    r:=(a+b+c)/2;
    S:=sqrt(r*(r-a)*(r-b)*(r-c));
    alfa:=arccos((sqr(b)+sqr(c)-
                        sqr(a))/2/b/c);
    betta:=arcsin(b/a*sin(alfa));
    gamma:=pi-(alfa+betta);
    alfa:=alfa*180/pi; betta:=betta*180/pi;
    gamma:=gamma*180/pi;
    Label6.Caption:='alfa='+
               FloatToStrF(alfa,ffFixed,3,0);
    Label7.Caption:='betta='+
               FloatToStrF(betta,ffFixed,3,0);
    Label8.Caption:='gamma='+
               FloatToStrF(gamma,ffFixed,3,0);
    Label9.Caption:='Периметр P='+
```

```
                FloatToStrF(2*r,ffFixed,5,2);
    Label10.Caption:='Площадь S='+
                FloatToStrF(S,ffFixed,5,2);
 end;
initialization
 {$I unit1.lrs}
end.
```

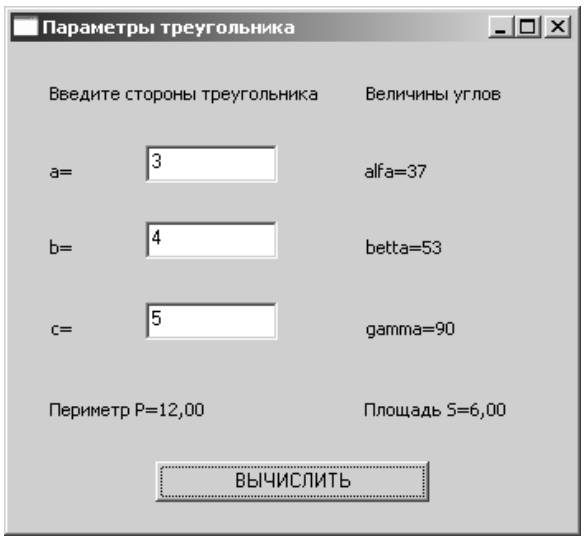

Рисунок 2.8: Результат работы программы к задаче 2.1

На рис. 2.8 представлено диалоговое окно, которое появится, если запустить эту программу, щелкнув по кнопке Вычислить.

Теперь напишем консольное приложение для решения этой задачи. Для этого запустим текстовый редактор Geany. Выполним команду **Файл — New with Tamplate — Pascal Source File**.

В открывшемся окне редактора введем следующий текст программы.

```
program pr3;
//Подключение модуля для работы
//с математическими функциями, см. табл.2.6.
uses math;
//Описание всех используемых переменных.
var a,b,c,r,s,alfa,betta,gamma:real;
BEGIN
//Ввод сторон трегольника.
writeln('Введите стороны треугольника');
readln(a,b,c);
//Вычисление значения полупериметра.
R:=(a+b+c)/2;
//Вычисление значения площади,
//для вычисления применяется функция:
//   sqrt(x) – корень квадратный из x.
S:=sqrt(r*(r-a)*(r-b)*(r-c));
//Вычисление значения угла alfa в радианах.
```

```
//Для вычисления применяем функции:
//           arccos(x) - арккосинус x;
//           sqr(x) - возведение x в квадрат.
alfa:=arccos((sqr(b)+sqr(c)-sqr(a))/2/b/c);
//Вычисление значения угла betta в радианах.
//Для вычисления применяем функции:
//           arcsin(x) - арксинус x;
betta:=arcsin(b/a*sin(alfa));
//Вычисление значения угла gamma в радианах.
//Математическая постоянная определена
//функцией без аргумента pi.
gamma:=pi-(alfa+betta);
//Перевод из радиан в градусы.
alfa:=alfa*180/pi;
betta:=betta*180/pi;
gamma:=gamma*180/pi;
//Вывод площади и сторон треугольника.
writeln('Площадь треугольника =',S:6:2);
writeln('alf=',alfa:6:2,
        ' bet=',betta:6:2,' gam=',gamma:6:2);
end.
```

Для компиляции программы в Geany необходимо выполнить команду **Построить — Собрать (F8)**, для запуска — **Построить — Выполнить (F5)**. На рис. 2.9 представлены результаты работы программы.

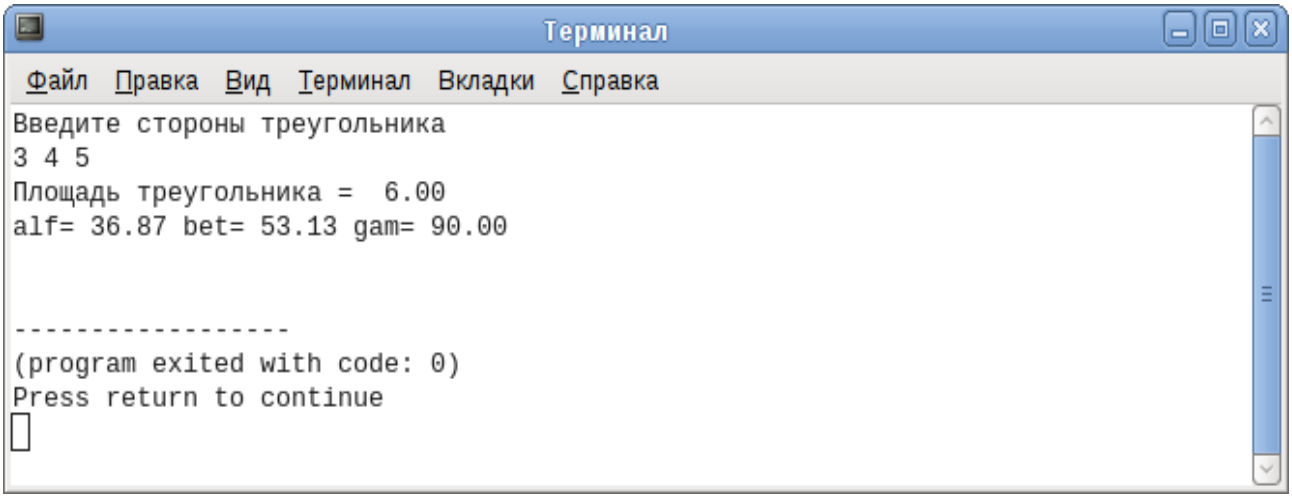

Рисунок 2.9: Результаты работы консольного приложения решения задачи 2.1

2.7 Задачи для самостоятельного решения

Разработать программу в среде программирования Lazarus. Для каждой задачи создать интерфейс, соответствующий условию.

1. Заданы два катета прямоугольного треугольника. Найти гипотенузу и углы треугольника.
2. Известна гипотенуза c и прилежащий угол α прямоугольного треугольника. Найти площадь треугольника.
3. Известна диагональ квадрата d. Вычислить площадь S и периметр P квадрата.
4. Известна диагональ прямоугольника d и угол α между диагональю и большей стороной. Вычислить площадь S прямоугольника.
5. Треугольник задан величинами своих сторон – a, b, c. Найти углы треугольника – α, β, γ.
6. Тело имеет форму параллелепипеда с высотой h. Прямоугольник в основании имеет диагональ d. Известно, что диагонали основания пересекаются под углом α. Найти объем тела V и площадь поверхности S.
7. В треугольнике известен катет a и площадь S. Найти величину гипотенузы c, второго катета b и углов α и β.
8. Известна площадь квадрата S. Вычислить сторону квадрата a, диагональ d и площадь S1 описанного вокруг квадрата круга.
9. В равнобедренном треугольнике известно основание c и угол при нем α. Найти площадь треугольника S и величину боковой стороны a.
10. Известны координаты трех вершин прямоугольника ABCD: A(x1,y1), B(x2,y2) и C(x3,y3). Найти его площадь и периметр.
11. Заданы два катета прямоугольного треугольника. Вычислить его площадь и периметр.
12. Известна гипотенуза c и противолежащий угол α прямоугольного треугольника. Найти периметр треугольника.
13. Известна диагональ ромба d. Вычислить его площадь S и периметр P.
14. Известна длина диагоналей прямоугольника d и угол α между ними. Вычислить площадь S прямоугольника.
15. В прямоугольном треугольнике известен катет b и площадь S. Вычислить периметр треугольника.

16. Известно значение периметра P равностороннего треугольника. Вычислить его площадь.

19. Задан периметр квадрата P. Вычислить сторону квадрата a, диагональ d и площадь S.

20. В равнобедренном треугольнике известно основание c и высота h. Найти площадь треугольника S и периметр P.

21. Известны координаты вершин треугольника ABC: A(x1,y1), B(x2,y2) и C(x3,y3). Найти его площадь и периметр.

22. Металлический слиток имеет форму цилиндра, площадь поверхности S, высота h, плотность α. Вычислить массу m слитка.

23. Задан первый член арифметической прогрессии и ее шаг. Вычислить сумму n членов арифметической прогрессии и значение n-го члена.

24. Задан первый член геометрической прогрессии и ее знаменатель. Вычислить сумму n членов геометрической прогрессии и значение n-го члена.

25. Тело падает с высоты h. Какова его скорость в момент соприкосновения с землей и когда это произойдет.

3 Операторы управления

В этой главе изложена методика составления алгоритмов с помощью блок-схем и описаны основные операторы языка: условный оператор `if`, оператор выбора `case`, операторы цикла `while..do`, `repeat..until`, `for..do`. Приводится большое количество примеров составления программ различной сложности.

3.1 Основные конструкции алгоритма

Как правило, созданию программы предшествует разработка алгоритма[36]. *Алгоритм* — это четкое описание последовательности действий, которые необходимо выполнить для того, чтобы при соответствующих исходных данных получить требуемый результат. Одним из способов представления алгоритма является *блок-схема*. При ее составлении все этапы решения задачи изображаются с помощью различных геометрических фигур. Эти фигуры называют блоками и, как правило, сопровождают надписями. Последовательность выполнения этапов указывают при помощи стрелок, соединяющих эти блоки.

Типичные этапы решения задачи изображаются следующими геометрическими фигурами:

- *блок начала* (*конца*) (рис. 3.1). Надпись внутри блока: «начало» («конец»);
- *блок ввода* (*вывода*) данных (рис. 3.2). Надпись внутри блока: ввод (вывод) и список вводимых (выводимых) переменных;
- *блок решения,* или *арифметический* (рис. 3.3). Внутри блока записывается действие, вычислительная операция или группа операций;
- *условный блок* (рис. 3.4). Логическое условие записывается внутри блока. В результате проверки условия осуществляется выбор одного из возможных путей (ветвей) вычислительного процесса.

Рассмотренные блоки позволяют описать три *основные конструкции алгоритма*: линейный, разветвляющийся и циклический процессы.

36 Алгоритм - от algorithmi, algorismus, первоначально латинская транслитерация имени математика аль-Хорезми.

Рисунок 3.1: Блок начала (конца) алгоритма

Рисунок 3.2: Блок ввода (вывода)

Рисунок 3.3: Арифметический блок

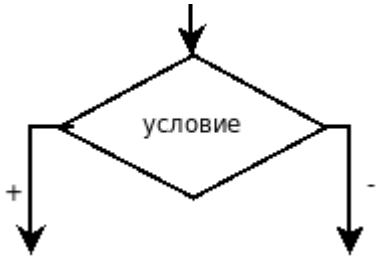

Рисунок 3.4: Условный блок

Линейный процесс — это конструкция, представляющая собой последовательное выполнение двух или более блоков (рис. 3.5).

Разветвляющийся процесс задает выполнение одного или другого оператора в зависимости от выполнения условия (рис. 3.6).

Циклический процесс задает многократное выполнение оператора или группы операторов (рис. 3.7).

Нетрудно заметить, что каждая из основных конструкций алгоритма имеет один вход и один выход. Это позволяет вкладывать конструкции друг в друга произвольным образом и составлять алгоритмы для решения задач любой сложности.

Рассмотрим операторы языка программирования Free Pascal, реализующие основные конструкции алгоритма.

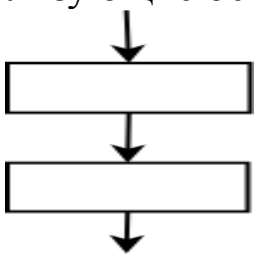

Рисунок 3.5: Линейный процесс

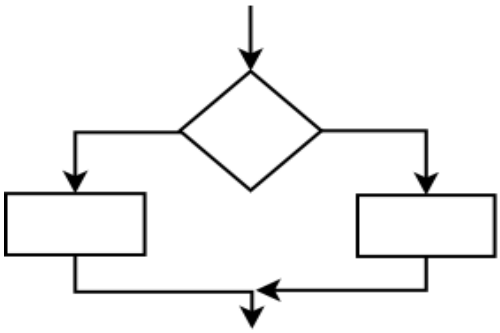

Рисунок 3.6: Разветвляющийся процесс

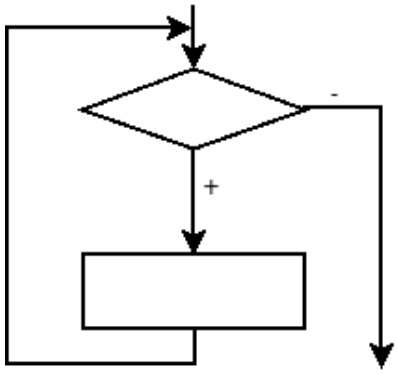

Рисунок 3.7: Циклический процесс

3.2 Оператор присваивания

Оператор присваивания в языке Free Pascal состоит из двух символов: двоеточия и знака равенства. Символы `:=` всегда пишут слитно. Пробелы допускаются перед символом двоеточия и после символа равенства.

В общем случае оператор присваивания имеет вид:

```
имя_переменной := значение;
```

где значение – это выражение, переменная, константа или функция. Выполняется оператор так. Сначала вычисляется значение выражения, указанного в правой части оператора, а затем его результат записывается в область памяти (переменную), имя которой указано слева. Например, запись a:=b означает, что переменной a присваивается значение b.

Типы переменных a и b должны совпадать или быть совместимыми для присваивания, то есть тип, к которому принадлежит переменная b, должен находиться в границах типа переменной a.

3.3 Составной оператор

Составной оператор – группа операторов языка Free Pascal, отделенных друг от друга точкой с запятой, начинающихся со служебного слова begin и заканчивающихся служебным словом end:

```
begin
оператор_1;
...
оператор_n
end;
```

Транслятор воспринимает составной оператор как один оператор.

3.4 Условные операторы

В языке Free Pascal одна из основных конструкций алгоритма, *разветвляющийся процесс,* реализована двумя условными операторами: if и case. Рассмотрим каждый из них.

3.4.1 Условный оператор if…then…else

При решении большинства задач порядок вычислений зависит от определенных условий, например, от исходных данных или от промежуточных результатов, полученных на предыдущих шагах программы. Для организации вычислений в зависимости от какого-либо условия в языке Free Pascal используется *условный оператор* if…then…else, который в общем виде записывается так:

```
if условие then оператор_1 else оператор_2;
```

где if…then…else – зарезервированные слова, условие – выражение логического типа[37], оператор_1 и оператор_2 – любые операторы языка Free Pascal.

37 Логическое выражение может принимать одно из двух значений: истина или ложь.

Работа условного оператора организована следующим образом. Сначала вычисляется выражение, записанное в условии. Если оно имеет значение истина (True), то выполняется оператор_1. В противном случае, когда выражение имеет значение ложь (False), оператор_1 игнорируется и управление передается оператору_2. Алгоритм, который реализован в условном операторе if…then…else, представлен на рис. 3.8.

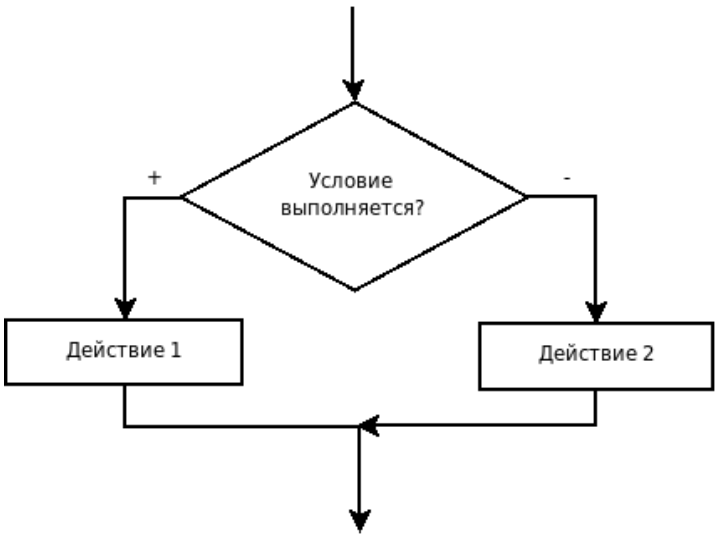

Рисунок 3.8: Алгоритм условного оператора if...then...else

Например, чтобы сравнить значения переменных x и y, нужно создать следующий программный код:

```
write('x='); readln(x);
write('y='); readln(y);
if x=y then
writeln('значение x равно значению y')
else
writeln('значение x не равно значению y');
```

Если в задаче требуется, чтобы в зависимости от значения условия выполнялся не один оператор, а несколько, их необходимо заключать в операторные скобки как составной оператор:

```
if условие then
 begin
     оператор_1;
     оператор_2;
     …
 end
```

```
else
 begin
     оператор_1;
     оператор_2;
     …
 end;
```

Альтернативная ветвь `else` в условном операторе может отсутствовать, если в ней нет необходимости:

```
if условие then оператор;
```

или

```
if условие then
 begin
     оператор_1;
     оператор_2;
     …
     оператор_n;
 end;
```

В таком «усеченном» виде условный оператор работает так: оператор (группа операторов) либо выполняется, либо пропускается, в зависимости от значения выражения, представляющего условие. Алгоритм этого условного процесса представлен на рис. 3.9.

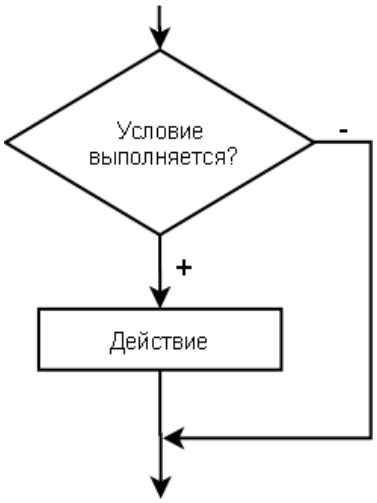

Рисунок 3.9: Алгоритм условного оператора if без альтернативной ветви else

Пример применения условного оператора, без альтернативной ветви `else` может быть таким:

```
write('x=');
readln(x);
write('y=');
readln(y);
z:=0;
{Значение с изменяется только при
условии,что x не равно y.}
if (x<>y) then    z:=x+y;
{Вывод на экран значения перемен-
ной z выполняется в любом случае.}
writeln('Значение z=',z);
```

Условные операторы могут быть вложены друг в друга. При вложениях условных операторов всегда действует правило: альтернатива `else` считается принадлежащей ближайшему `if`, имеющему ветвь `else`. Например, в записи

```
if условие_1 then
 if условие_2 then
     оператор_А
 else оператор_Б;
```

`оператор_Б` относится к `условию_2`, а в конструкции

```
if условие_1 then
 begin
     if условие_2 then
         оператор_А;
     end
 else оператор_Б;
```

он принадлежит оператору `if` с `условием_1`.

Для сравнения переменных в условных выражениях применяют *операции отношения*: =, <>, <, >, <=, >=. Условные выражения составляют с использованием логических операций `and`, `or` и `not`. В языке Free Pascal приоритет *операций отношения* меньше, чем у *логических операций*, поэтому составные части сложного логического выражения заключают в скобки.

Допустим, нужно проверить, принадлежит ли переменная `x` интервалу [`a`, `b`]. Условный оператор будет иметь вид:

```
if (x>=a) and (x<=b) then….
```

Запись

```
if x>=a and x<=b then…
```

не верна, так как фактически будет вычисляться значение выражения

```
x>= (a and x) <=b.
```

Рассмотрим использование оператора `if` на примерах[38].

ЗАДАЧА 3.1. Дано вещественное число `x`. Для функции, график которой приведен на рис. 3.10, вычислить $y=f(x)$.

38 В задачах этой главы мы не будем уделять много внимания интерфейсу создаваемых программ, чтобы у читателя была возможность разобраться в алгоритмах и способах их записи на языке Free Pascal.

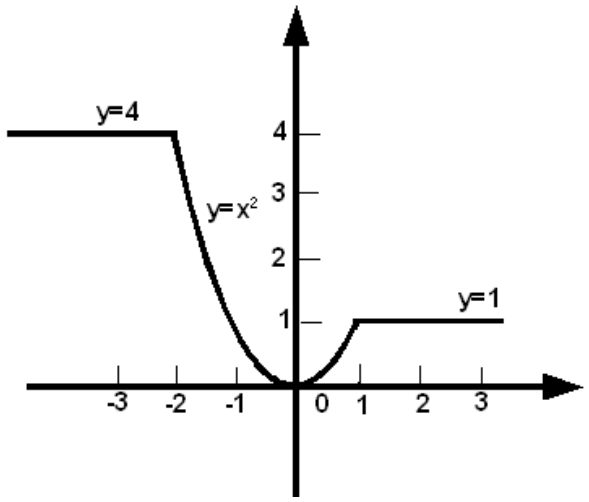

Рисунок 3.10: Графическое представление задачи 3.1

Аналитически функцию, представленную на рис. 3.10, можно записать так:

$$y(x)=\begin{cases} 4, & x\leqslant -2 \\ 1, & x\geqslant 1 \\ x^2, & -2<x<1 \end{cases}$$

Составим словесный алгоритм решения этой задачи:

1. Начало алгоритма.
2. Ввод числа `x` (аргумент функции).
3. Если значение `x` меньше либо равно `-2`, то переход к п. 4, иначе переход к п. 5.
4. Вычисление значения функции: `y=4`, переход к п. 8.
5. Если значение `x` больше либо равно `1`, то переход к п. 6, иначе переход к п. 7.
6. Вычисление значения функции: `y=1`, переход к п. 8.
7. Вычисление значения функции: $y=x^2$.
8. Вывод значений аргумента `x` и функции `y`.
9. Конец алгоритма.

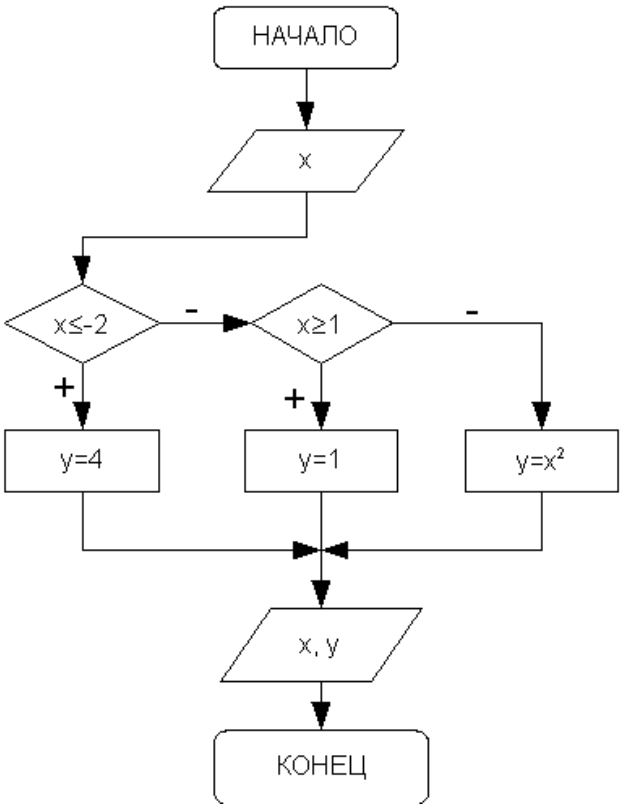

Рисунок 3.11: Блок-схема алгоритма решения задачи 3.1

Блок-схема, соответствующая описанному алгоритму, представлена на рис. 3.11.

Текст программы на языке Free Pascal будет иметь вид:

```
var x,y: real; begin
 write('x='); readln(x);
 if x<=-2 then y:=4 else if x>=1 then y:=1
 else y:=sqr(x);
writeln('x=',x:5:2,'    y=',y:5:2);end.
```

ЗАДАЧА 3.2. Даны вещественные числа x и y. Определить принадлежит ли точка с координатами (x; y) заштрихованной части плоскости (рис. 3.12).

Как показано на рис. 3.12, плоскость ограничена линиями x=-1, x=3, y=-2 и y=4. Значит точка с координатами (x; y) будет принадлежать этой плоскости, если будут выполняться следующие условия: $x \geq -1$, $x \leq 3$, $y \geq -2$ и $y \leq 4$. Иначе точка лежит за пределами плоскости.

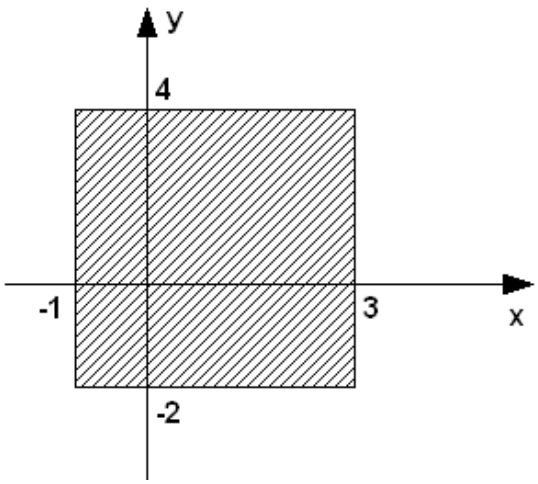

Рисунок 3.12: Графическое представление задачи 3.2

Блок-схема, описывающая алгоритм решения данной задачи, представлена на рис. 3.13. Текст программы к задаче 3.2 приведен далее.

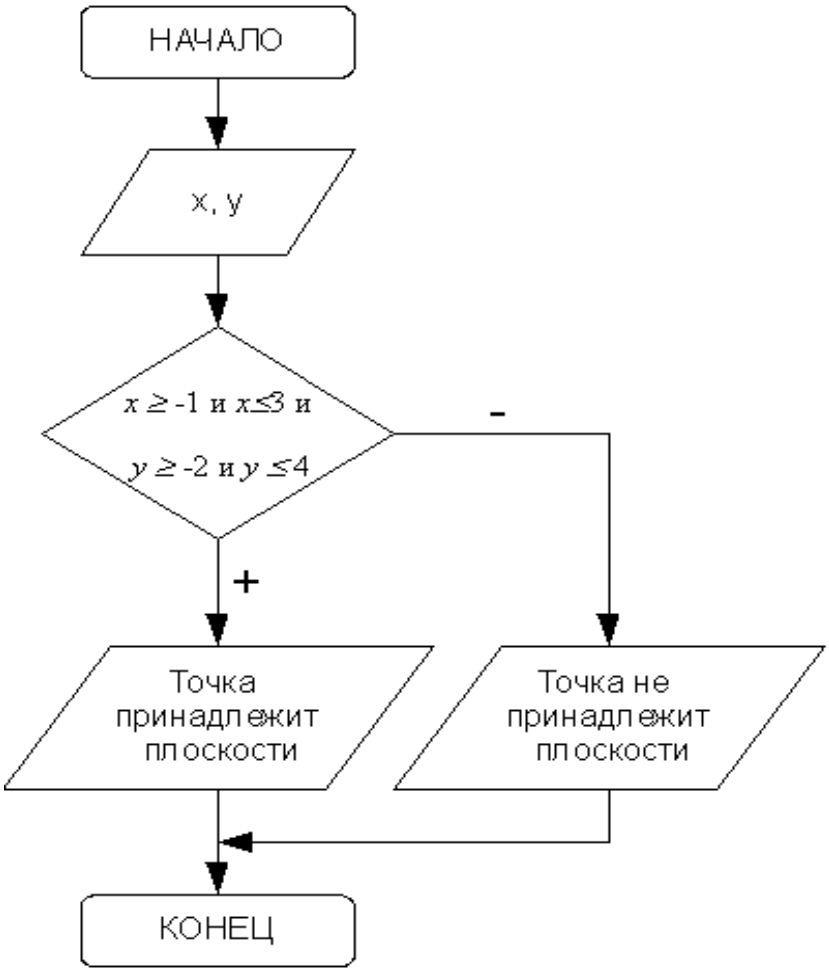

Рисунок 3.13: Алгоритм решения задачи 3.2

```
var
    x,y:real;
begin
    write('x=');
    readln(x);
    write('y=');
    readln(y);
    if (x>=-1) and (x<=3) and (y>=-2) and (y<=4)
                                                then
        writeln('Точка принадлежит плоскости')
    else
        writeln('Точка не принадлежит плоскости');
end.
```

ЗАДАЧА 3.3. Написать программу решения квадратного уравнения $ax^2+bx+c=0$.

Исходные данные: вещественные числа a, b и c – коэффициенты квадратного уравнения.

Результаты работы программы: вещественные числа x1 и x2 – корни квадратного уравнения либо сообщение о том, что корней нет.

Вспомогательные переменные: вещественная переменная d, в которой будет храниться дискриминант квадратного уравнения.

Составим словесный алгоритм решения этой задачи.

1. Начало алгоритма.
2. Ввод числовых значений переменных a, b и c.
3. Вычисление значения дискриминанта d по формуле $d = b^2 - 4ac$.
4. Если d<0, то переход к п.5, иначе переход к п.6.
5. Вывод сообщения Вещественных корней нет и переход к п.8.
6. Вычисление корней $x1 = \frac{-b+\sqrt{d}}{2a}$ и $x2 = \frac{-b-\sqrt{d}}{2a}$.
7. Вывод значений x1 и x2 на экран.
8. Конец алгоритма.

Блок-схема, соответствующая этому описанию, представлена на рис. 3.14.

Текст консольной программы, которая реализует решение квадратного уравнения:

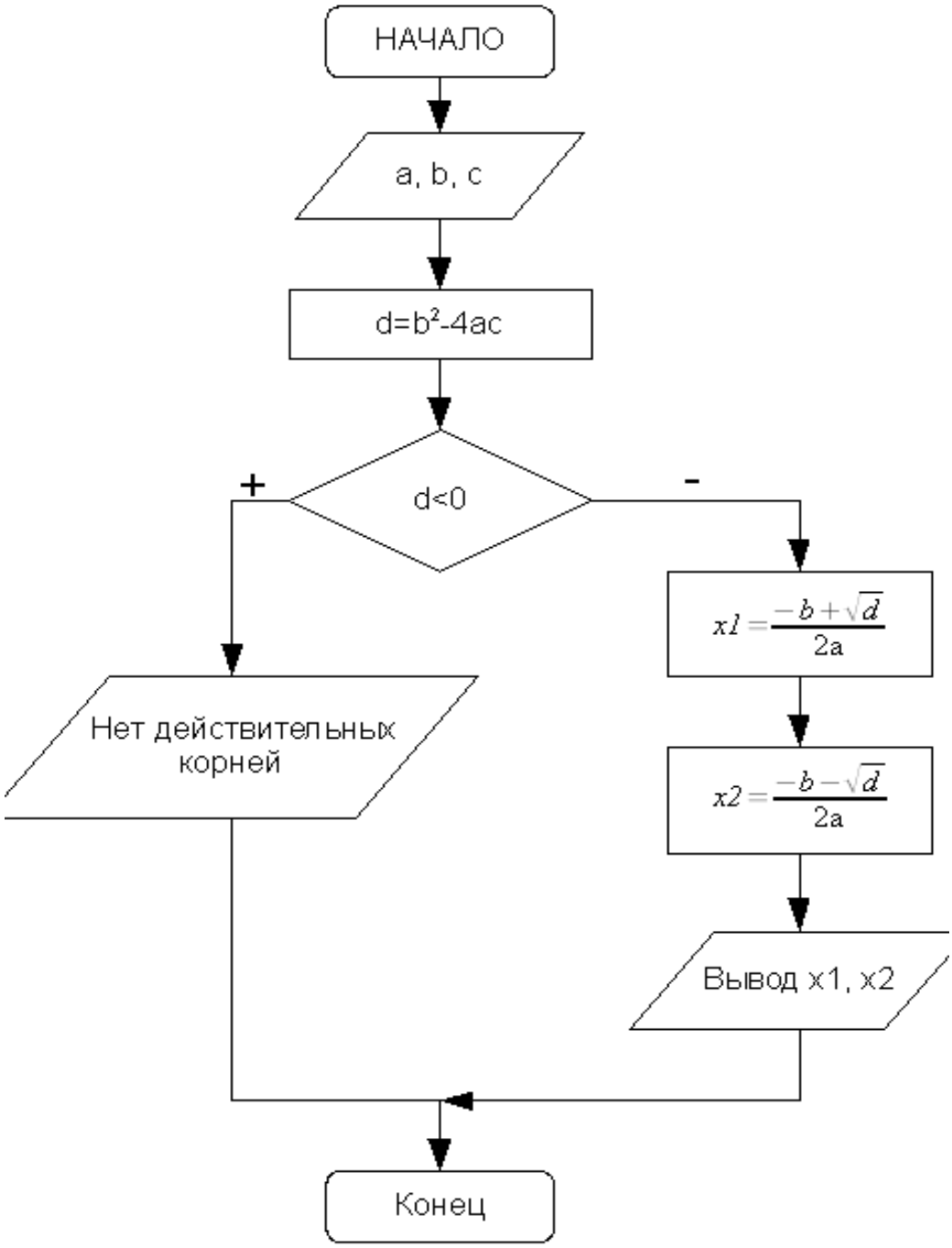

Рисунок 3.14: Алгоритм решения квадратного уравнения

```
{Описание переменных.}
var a,b,c,d,x1,x2: real;
begin
 {Ввод коэффициентов квадратного уравнения.}
 writeln('Введите коэффициенты уравнения');
 readln(a,b,c);
 {Вычисление дискриминанта.}
```

```
    d:=b*b-4*a*c;
    {Если дискриминант отрицателен,}
    if d<0 then
        {то вывод сообщения, что корней нет,}
        writeln('Корней нет')
    else
    begin
        {иначе вычисление корней x1, x2}
        x1:=(-b+sqrt(d))/2/a;
        x2:=(-b-sqrt(d))/(2*a);
        {и вывод их на экран.}
        writeln('X1=',x1:6:3, ' X2=',x2:6:3)
    end
    end.
```

ЗАДАЧА 3.4. Составить программу нахождения действительных и комплексных корней квадратного уравнения $ax^2+bx+c=0$.

Исходные данные: вещественные числа `a`, `b` и `c` – коэффициенты квадратного уравнения.

Результаты работы программы: вещественные числа `x1` и `x2` – действительные корни квадратного уравнения либо `x1` и `x2` – действительная и мнимая части комплексного числа.

Вспомогательные переменные: вещественная переменная `d`, в которой будет храниться дискриминант квадратного уравнения.

Можно выделить следующие этапы решения задачи:

1. Ввод коэффициентов квадратного уравнения `a`, `b` и `c`.
2. Вычисление дискриминанта d по формуле $d = b^2 - 4ac$.
3. Проверка знака дискриминанта. Если $d \geq 0$, то вычисление действительных корней:

$$x1 = \frac{-b+\sqrt{d}}{2a} \quad \text{и} \quad x2 = \frac{-b-\sqrt{d}}{2a}$$

и вывод их на экран. При отрицательном дискриминанте выводится сообщение о том, что действительных корней нет, и вычисляются комплексные корни[39] $\frac{-b}{2a} + i\frac{\sqrt{|d|}}{2a}$, $\frac{-b}{2a} - i\frac{\sqrt{|d|}}{2a}$.

У обоих комплексных корней действительные части одинаковые,

39 Комплексные числа записываются в виде a+ib, где a – действительная часть комплексного числа, b – мнимая часть комплексного числа, i – мнимая единица $\sqrt{-1}$.

а мнимые отличаются знаком. Поэтому можно в переменной `x1` хранить действительную часть числа $\frac{-b}{2a}$, в переменной `x2` – модуль мнимой части $\frac{\sqrt{|d|}}{2a}$, а в качестве корней вывести `x1+ix2` и `x1-ix2`.

На рис. 3.15 изображена блок-схема решения задачи.

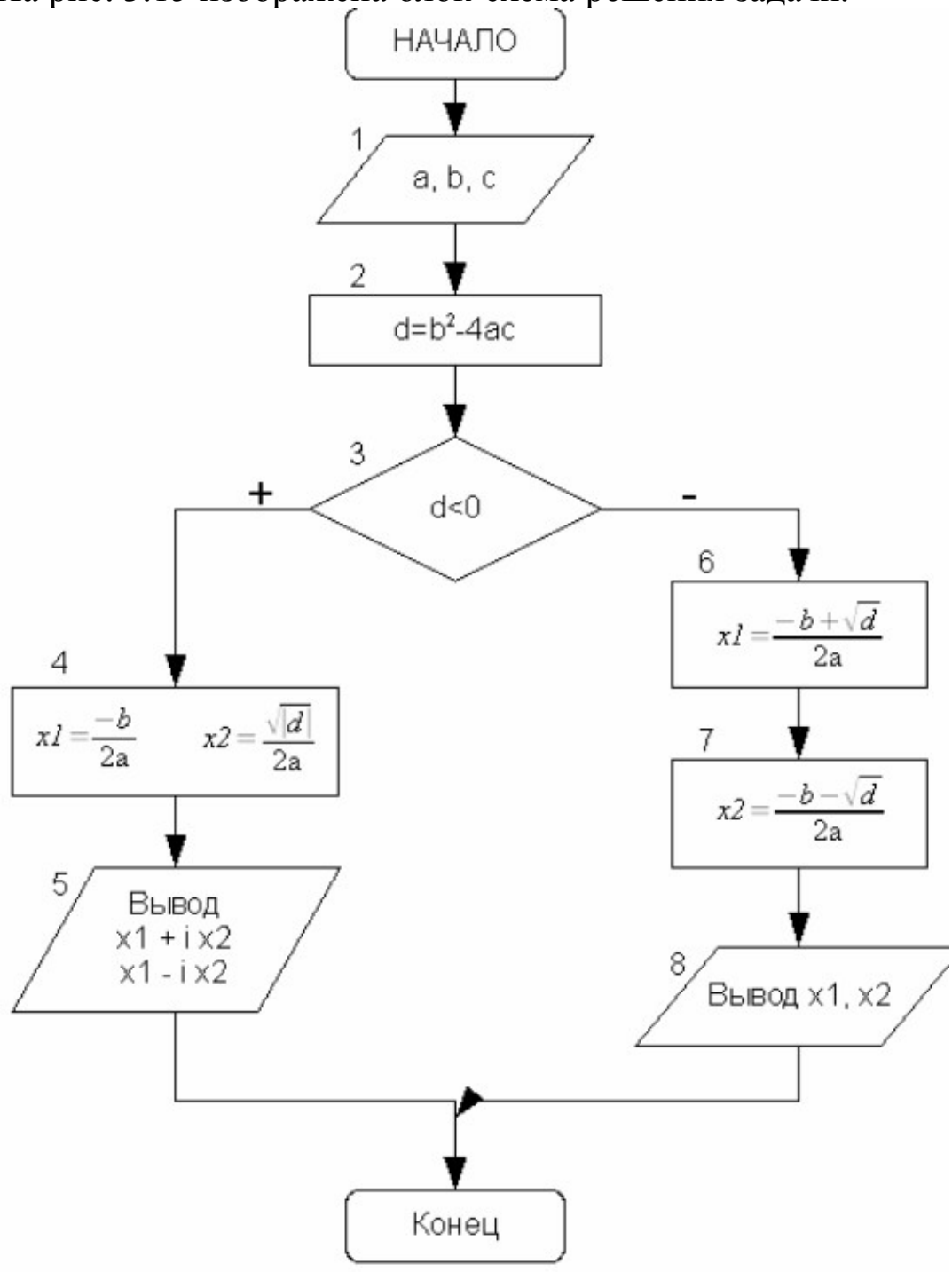

Рисунок 3.15: Алгоритм решения задачи 3.4

Описание блок-схемы, изображенной на рис. 3.15. Блок 1 предназначен для ввода коэффициентов квадратного уравнения. В блоке 2 осуществляется вычисление дискриминанта. Блок 3 осуществляет проверку знака дискриминанта, если дискриминант отрицателен, то корни комплексные, их расчет происходит в блоке 4 (действительная часть корня записывается в переменную `x1`, модуль мнимой – в переменную `x2`), а вывод – в блоке 5 (первый корень `x1+ix2`, второй – `x1-ix2`). Если дискриминант положителен, то вычисляются действительные корни уравнения (блоки 6, 7) и выводятся на экран (блок 8).

Текст программы, реализующей поставленную задачу:

```
var a,b,c,d,x1,x2: real;
begin
 writeln('Введите коэффициенты уравнения');
 readln(a,b,c);
 d:=b*b-4*a*c;
 if d<0 then
 begin
 {Если дискриминант отрицателен,
 то вывод сообщения,
 что действительных корней нет, и вычисление
 комплексных корней.}
     writeln('Действительных корней нет');
     {Вычисление действительной части
         комплексных корней.}
     x1:=-b/(2*a);
     {Вычисление модуля мнимой части
         комплексных корней.}
     x2:=sqrt(abs(d))/(2*a);
     writeln('Комплексные корни уравнения ',
         a:1:2,'x^2+',b:1:2,'x+',c:1:2,'=0');
     {Вывод значений комплексных корней в виде
      x1±ix2}
     writeln(x1:1:2,'+i*(',x2:1:2,')');
     writeln(x1:1:2,'-i*(',x2:1:2,')');
 end
 else
 begin
     {иначе вычисление действительных
```

```
        корней x1, x2}
        x1:=(-b+sqrt(d))/2/a;
        x2:=(-b-sqrt(d))/(2*a);
        {и вывод их на экран.}
        writeln('Действительные корни уравнения ',
             a:1:2,'x^2+',b:1:2,'x+',c:1:2,'=0');
        writeln('X1=',x1:1:2, ' X2=',x2:1:2)
    end
end.
```

ЗАДАЧА 3.5. Составить программу для решения кубического уравнения $ax^3+bx^2+cx+d=0$.

Кубическое уравнение имеет вид $ax^3+bx^2+cx+d=0$ (3.1)

После деления на a уравнение (3.1) принимает канонический вид:

$$x^3+rx^2+sx+t=0 \quad (3.2)$$

где $r=\frac{b}{a}$, $s=\frac{c}{a}$, $t=\frac{d}{a}$. В уравнении (3.2) сделаем замену $x=y-\frac{r}{3}$ и получим приведенное уравнение: $y^3+py+q=0$ (3.3),

где $p=\frac{3s-r^2}{3}$, $q=\frac{2r^3}{27}-\frac{rs}{3}+t$.

Число действительных корней приведенного уравнения (3.3) зависит от знака дискриминанта (табл. 3.1): $D=(\frac{p}{3})^3+(\frac{q}{2})^2$.

Таблица 3.1. Количество корней кубического уравнения

Дискриминант	**Количество действительных корней**	**Количество комплексных корней**
$D\geqslant 0$	1	2
$D<0$	3	-

Корни приведенного уравнения могут быть рассчитаны по формулам Кардано:

$$\begin{aligned} y_1&=u+v \\ y_2&=\frac{-u+v}{2}+\frac{u-v}{2}i\sqrt{3} \\ y_3&=\frac{-u+v}{2}-\frac{u-v}{2}i\sqrt{3} \end{aligned} \quad (3.4)$$

где $u=\sqrt[3]{\frac{-q}{2}+\sqrt{D}}$, $v=\sqrt[3]{\frac{-q}{2}-\sqrt{D}}$.

При отрицательном дискриминанте уравнение (3.1) имеет три действительных корня, но они будут вычисляться через вспомогательные комплексные величины. Чтобы избавиться от этого, можно воспользоваться формулами:

$$y_1=2\sqrt[3]{\rho}\cos(\frac{\phi}{3}) ,$$

$$y_2=2\sqrt[3]{\rho}\cos(\frac{\phi}{3}+\frac{2\pi}{3}) , \quad y_3=2\sqrt[3]{\rho}\cos(\frac{\phi}{3}+\frac{4\pi}{3}) , \qquad (3.5)$$

где $\rho=\sqrt{\frac{-p^3}{27}}$, $\cos(\phi)=\frac{-q}{2\rho}$.

Таким образом, при положительном дискриминанте кубического уравнения (3.3) расчет корней будем вести по формулам (3.4), а при отрицательном – по формулам (3.5). После расчета корней приведенного уравнения (3.3) по формулам (3.4) или (3.5) необходимо по формулам $x_k=y_k-\frac{r}{3}$, $k=1,2,3...$ перейти к корням заданного кубического уравнения (3.1).

Блок-схема решения кубического уравнения представлена на рис.3.16 - 3.17.

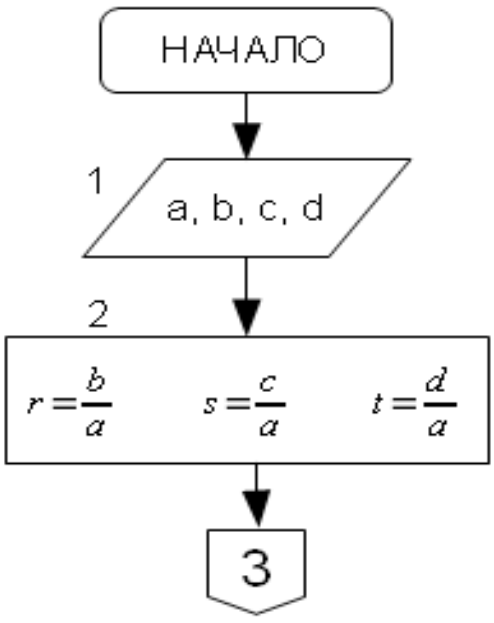

Рисунок 3.16: Алгоритм решения кубического уравнения

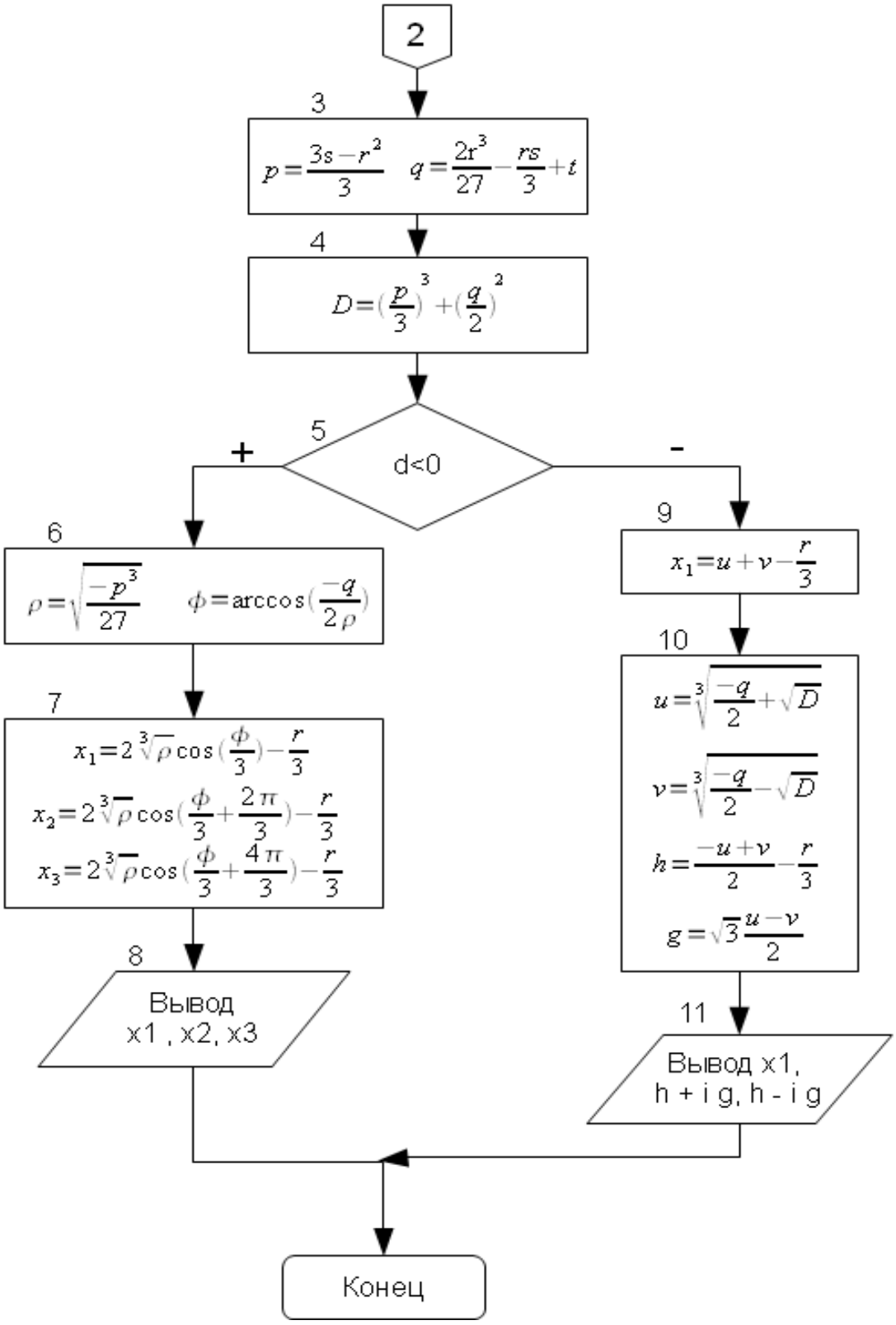

Рисунок 3.17: Алгоритм решения кубического уравнения (продолжение)

Описание алгоритма решения кубического уравнения (рис. 3.16 - 3.17). В блоке 1 вводятся коэффициенты кубического уравнения, в

блоках 2-3 рассчитываются коэффициенты канонического и приведенного уравнений. Блок 4 предназначен для вычисления дискриминанта. В блоке 5 проверяется знак дискриминанта кубического уравнения. Если он отрицателен, то корни вычисляются по формулам (3.5) (блоки 6-7). При положительном значении дискриминанта расчет идет по формулам (3.4) (блок 9, 10). Блоки 8 и 11 предназначены для вывода результатов на экран.

Текст программы приведен далее[40].

```
var
 a,b,c,d,r,s,t,p,q,ro,fi,x1,x2,x3,u,v,h,g:real;
begin
 {Ввод коэффициентов кубического уравнения.}
 write('a=');readln(a);
 write('b=');readln(b);
 write('c=');readln(c);
 write('d=');readln(d);
 {Расчет коэффициентов канонического
    уравнения по (3.2).}
 r:=b/a; s:=c/a; t:=d/a;
 {Вычисление коэффициентов
    приведенного уравнения (3.3).}
 p:=(3*s-r*r)/3;
 q:=2*r*r*r/27-r*s/3+t;
 {Вычисление дискриминанта
    кубического уравнения.}
 d:=(p/3)*sqr(p/3)+sqr(q/2);
 {Проверка знака дискриминанта,
 ветка then реализует формулы (3.5),
 ветка else - формулы (3.4)}
 if d<0 then
 begin
    ro:=sqrt(-p*p*p/27);
```

40 При расчете величин u и v в программе предусмотрена проверка значения подкоренного выражения.

Если $\frac{-q}{2}\mp\sqrt{D}>0$, *то* $u=\sqrt[3]{\frac{-q}{2}+\sqrt{D}}, a \;\; v=\sqrt[3]{\frac{-q}{2}-\sqrt{D}}$.

Если $\frac{-q}{2}\mp\sqrt{D}<0$, *то* $u=-\sqrt[3]{|\frac{-q}{2}+\sqrt{D}|}, a \;\; v=-\sqrt[3]{|\frac{-q}{2}-\sqrt{D}|}$.

Соответственно, при нулевом значении подкоренного выражения u и v обращаются в ноль.

```
    {Следующие два оператора реализуют
    расчет угла fi, сначала вычисляется
    величина косинуса угла, затем вычисляется
    его арккосинус через арктангенс.}
    fi:=-q/(2*ro);
    fi:=pi/2-arctan(fi/sqrt(1-fi*fi));
    {Вычисление действительных
    корней уравнения x1, x2 и x3}
    x1:=2*exp(1/3*ln(ro))*cos(fi/3)-r/3;
    x2:=2*exp(1/3*ln(ro))*
                  cos(fi/3+2*pi/3)-r/3;
    x3:=2*exp(1/3*ln(ro))*
                  cos(fi/3+4*pi/3)-r/3;
    writeln('x1=',x1:1:3,
           ' x2=',x2:1:3,' x3=',x3:1:3);
end
else
begin
    {Вычисление u и v с проверкой знака
              подкоренного выражения. }
    if -q/2+sqrt(d)>0 then
       u:=exp(1/3*ln(-q/2+sqrt(d)))
    else
       if -q/2+sqrt(d)<0 then
          u:=-exp(1/3*ln(abs(-q/2+sqrt(d))))
       else
          u:=0;
    if -q/2-sqrt(d)>0 then
       v:=exp(1/3*ln(-q/2-sqrt(d)))
    else
       if -q/2-sqrt(d)<0 then
          v:=-exp(1/3*ln(abs(-q/2-sqrt(d))))
       else
          v:=0;
    {Вычисление действительного
       корня кубического уравнения.}
    x1:=u+v-r/3;
    {Вычисление действительной и
```

```
        мнимой частей комплексных корней.}
        h:=-(u+v)/2-r/3;
        g:=(u-v)/2*sqrt(3);
        writeln('x1=',x1:1:3,' x2=',h:1:3,'+i*',
                g:1:3,' x3=',h:1:3,'-i*',g:1:3);
    end
end.
```

ЗАДАЧА 3.6. Заданы коэффициенты *a*, *b* и *c* биквадратного уравнения $ax4+bx^2+c=0$. Найти все его действительные корни.

Входные данные: a, b, c.

Выходные данные: x1, x2, x3, x4.

Для решения биквадратного уравнения необходимо заменой $y=x^2$ привести его к квадратному уравнению $ay^2+by+c=0$ и решить это уравнение.

Опишем алгоритм решения этой задачи (рис. 3.18):

1. Ввод коэффициентов биквадратного уравнения a, b и c (блок 1).
2. Вычисление дискриминанта уравнения d (блок 2).
3. Если d < 0 (блок 3), вывод сообщения, что корней нет (блок 4), а иначе определяются корни соответствующего квадратного уравнения y1 и y2 (блок 5).
4. Если y1<0 и y2<0 (блок 6) , то вывод сообщения, что корней нет (блок 7).
5. Если y1≥0 и y2≥0 (блок 8), то вычисляются четыре корня по формулам $\pm\sqrt{y_1}, \pm\sqrt{y_2}$ (блок 9) и выводятся значения корней (блок 10).
6. Если условия 4) и 5) не выполняются, то необходимо проверить знак y1. Если y1≥0 (блок 11), то вычисляются два корня по формуле $\pm\sqrt{y_1}$ (блок 12), иначе (если y2≥0) вычисляются два корня по формуле $\pm\sqrt{y_2}$ (блок 13). Вывод вычисленных корней (блок 14).

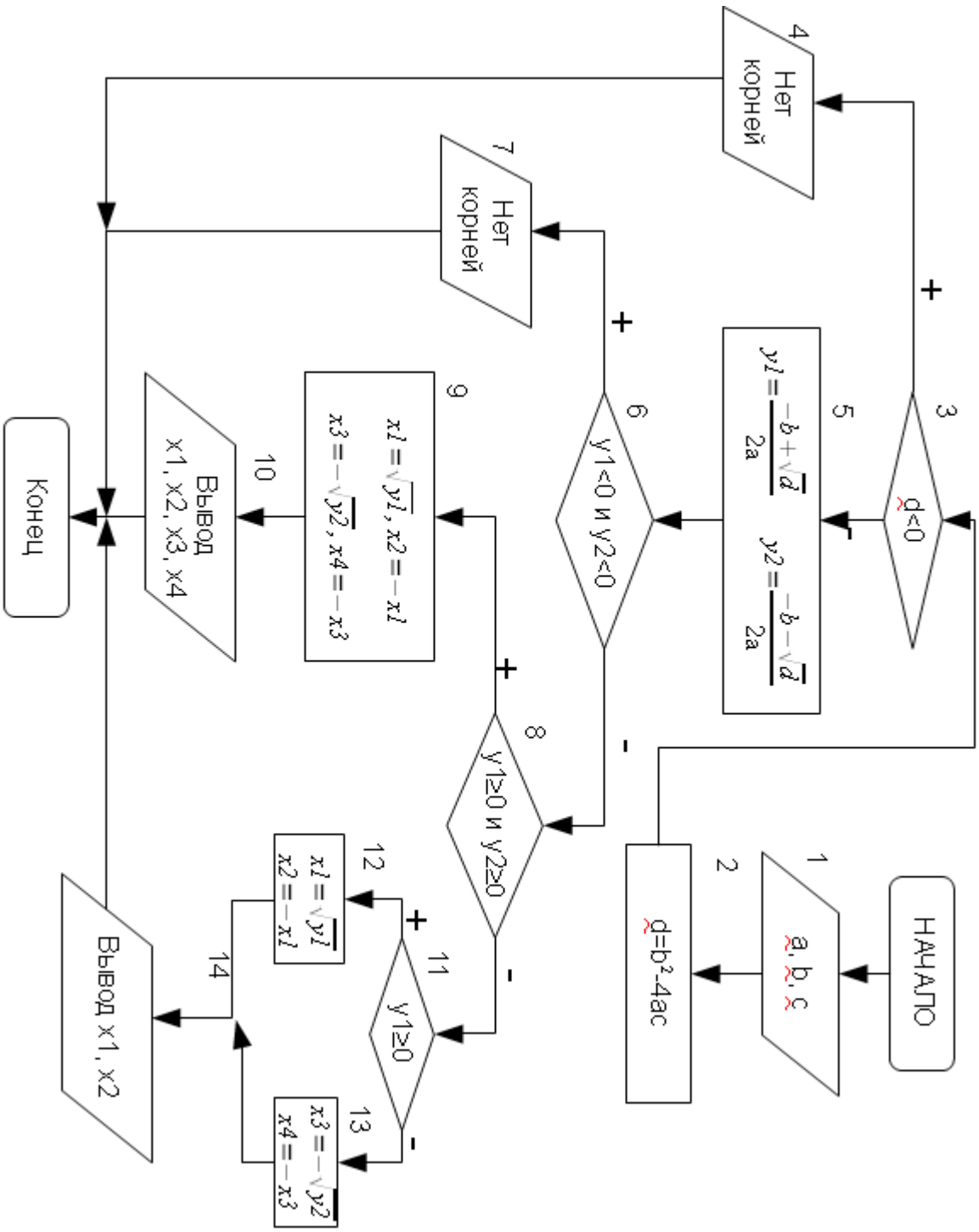

Рисунок 3.18: Алгоритм решения биквадратного уравнения

Текст программы на языке Free Pascal с комментариями:

```
var
    a,b,c,d,x1,x2,x3,x4,y1,y2: real;
begin
 {Ввод коэффициентов уравнения.}
 writeln('Введите коэффициенты уравнения');
 readln(a,b,c);
 {Вычисление дискриминанта.}
 d:=b*b-4*a*c;
 {Если он отрицателен,}
 if d<0 then
    {вывод сообщения «Корней нет»}
    writeln('Корней нет')
 {Если дискриминант ≥0,}
 else
{Описание переменных:
a,b,c – коэффициенты биквадратного уравнения,
d – дискриминант,
x1,x2,x3,x4 – корни биквадратного уравнения,
y1,y2 – корни уравнения ay^2+by+c=0.}
 begin
    {вычисление корней квадратного уравнения.}
    y1:=(-b+sqrt(d))/2/a;
    y2:=(-b-sqrt(d))/(2*a);
    {Если оба корня квадратного уравнения <0,}
    if (y1<0) and (y2<0) then
       {вывод сообщения «Корней нет».}
       writeln('Корней нет')
    {Если оба корня квадратного уравнения ≥0,}
    else if (y1>=0) and (y2>=0) then
    begin
       {вычисление четырех корней уравнения.}
       x1:=sqrt(y1);  x2:=-x1;
       x3:=sqrt(y2);  x4:=-sqrt(y2);
       {Вывод корней уравнения на экран.}
       writeln('X1=',x1:6:3,' X2=',x2:6:3);
       writeln('X3=',x3:6:3,' X4=',x4:6:3);
    end
```

```
        {Если не выполнились оба условия
            1.  y1<0 И y2<0
            2.  y1>=0 И y2>=0,
        то проверяем условие y1>=0}
        else if (y1>=0) then
        {Если оно истинно}
        begin
            x1:=sqrt(y1);  x2:=-x1;
            writeln('X1=',x1:6:3,'   X2=',x2:6:3);
        end
        else
        {Если условие y1>=0 ложно, то}
        begin
            x1:=sqrt(y2);  x2:=-x1;
            writeln('X1=',x1:6:3,'   X2=',x2:6:3);
        end
    end
end.
```

3.4.2 Оператор варианта case

Оператор варианта `case` необходим в тех случаях, когда в зависимости от значений какой-либо переменной надо выполнить те или иные операторы.

```
case переменная of
 набор_значений_1: оператор_1;
 набор_значений_2: оператор_2;
 …
 набор_значений_N: оператор_N
else
 альтернативный_оператор
end;
```

Оператор работает следующим образом. Если `переменная` принимает значение из `набора_значений_1`, то выполняется `оператор_1`. Если `переменная` принимает значение из `набора_значений_2`, то выполняется `оператор_2` и так далее. Если `переменная` не принимает значений из имеющихся наборов, то выполняется `альтернативный_оператор`, расположенный после ключевого слова `else`.

Тип переменной должен быть только перечислимым (включая char и boolean), диапазоном или целочисленным. Набор_значений – это конкретное значение управляющей переменной или выражение, при котором необходимо выполнить соответствующий оператор, игнорируя остальные варианты. Значения в каждом наборе должны быть уникальны, то есть они могут появляться только в одном варианте. Пересечение наборов значений для разных вариантов является ошибкой.

Альтернативная ветвь else может отсутствовать, тогда оператор имеет вид:

```
case переменная of
 набор_значений_1: оператор_1;
 набор_значений_2: оператор_2;
 …
 набор_значений_N: оператор_N;
end;
```

Кроме того, в операторе case допустимо использование составного оператора. Например:

```
case переменная of
 набор_значений_1: begin
                     оператор_A;
                     оператор_B;
                   end;
 набор_значений_2: begin
                     оператор_C;
                     оператор_D;
                     оператор_E;
                   end;
 …
 набор_значений_N: оператор_N;
end;
```

ЗАДАЧА 3.7. Вывести на печать название дня недели, соответствующее заданному числу D, при условии, что в месяце 31 день и первое число – понедельник.

Для решения задачи воспользуемся операцией mod, позволяющей вычислить остаток от деления двух чисел, и условием, что 1-е число – понедельник. Если в результате остаток от деления (обозначим его R)

заданного числа D на семь будет равен единице, то это понедельник, двойке – вторник, тройке – среда и так далее. Следовательно, при построении алгоритма необходимо использовать семь условных операторов, как показано рис. 3.19.

Решение задачи станет значительно проще, если при написании программы воспользоваться оператором варианта:

```
var d:byte;
begin
 write('Введите число D='); readln(D);
{Вычисляется остаток от деления D на 7.}
```

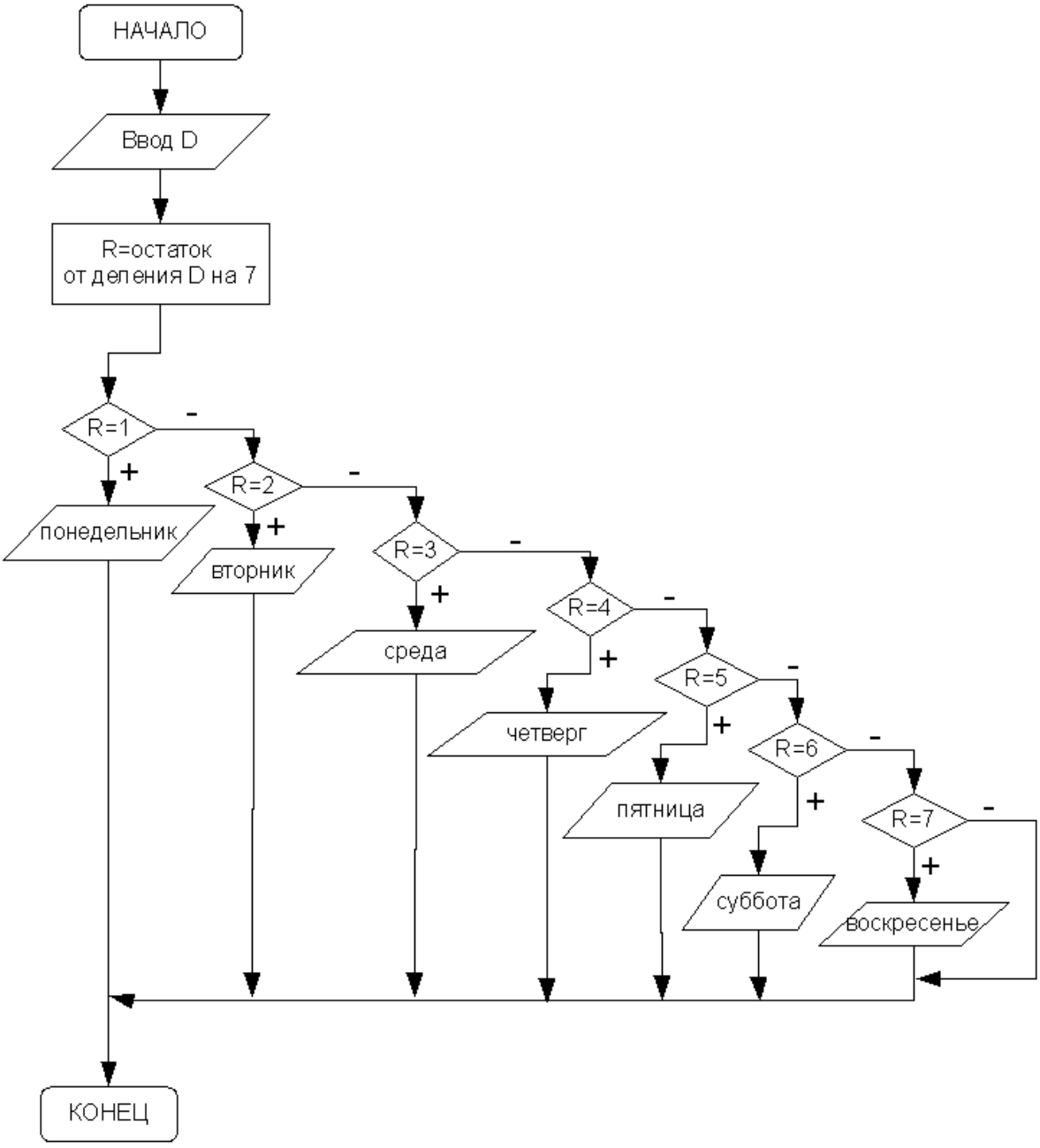

Рисунок 3.19: Алгоритм решения задачи 3.7

```
case D mod 7 of
    {В зависимости от полученного значения}
    {на печать выводится название дня недели}
    1:  writeln('ПОНЕДЕЛЬНИК');
    2:  writeln('ВТОРНИК');
    3:  writeln('СРЕДА');
    4:  writeln('ЧЕТВЕРГ');
    5:  writeln('ПЯТНИЦА');
    6:  writeln('СУББОТА');
    0:  writeln('ВОСКРЕСЕНЬЕ');end;end.
```

В предложенной записи оператора варианта отсутствует ветвь `else`. Это объясняется тем, что переменная `R` может принимать только одно из указанных значений, т.е. 1, 2, 3, 4, 5, 6 или 0.

ЗАДАЧА 3.8. По заданному номеру месяца `m` вывести на печать название времени года.

Для решения данной задачи необходимо проверить выполнение четырех условий. Если заданное число `m` равно 12, 1 или 2, то это зима, если `m` попадает в диапазон от 3 до 5, то – весна, лето определяется принадлежностью числа `m` диапазону от 6 до 8 и, соответственно, при равенстве переменной `m` 9, 10 или 11 – это осень. Понятно, что область возможных значений переменной `m` находится в диапазоне от 1 до 12 и если пользователь введет число, не входящее в этот интервал, то появится сообщение об ошибке. Для этого в операторе `case` программы предусмотрена альтернативная ветка `else`.

```
Var m:byte;
begin
 write('Введите номер месяца m=');
 readln(m);
 {Проверка допустимых значений переменной m.}
 case m of
    {В зависимости от значения m на печать}
    {выводится название времени года.}
    12,1,2:writeln('ЗИМА');
    3..5:  writeln('ВЕСНА');
    6..8:  writeln('ЛЕТО');
    9..11: writeln('ОСЕНЬ')
```

```
    {Если значение переменной m выходит за пределы
    области допустимых значений, то выдается
    сообщение об ошибке.}
        else
            writeln('ОШИБКА ПРИ ВВОДЕ!!!');
        end
end.
```

3.5 Обработка ошибок. Вывод сообщений в среде Lazarus

Понятно, что чем меньше в программе ошибок, тем она лучше. В очень хорошей программе ошибок нет вообще. А это значит, что программист должен не только основательно продумать алгоритм, поставленной задачи, но и предугадать ошибки, которые может допустить пользователь, работая с программой.

Если пользователь допустил ошибку, например, при вводе данных, его необходимо проинформировать об этом. Для этого можно воспользоваться функцией `MessageDlg`, которая выводит *сообщение* в отдельном окне. В общем виде функцию записывают так:

```
MessageDlg(сообщение, тип_сообщения,
                        [список_кнопок], справка);
```

где

- `сообщение` – текст, который будет отображен в окне сообщения;
- `тип_ сообщения` – определяет внешний вид окна (табл. 3.2);
- `список_кнопок` – константы (перечисляются через запятую), определяющие тип кнопок окна сообщения (табл. 3.3);
- `справка` – номер окна справочной системы, которое будет выведено на экран, если нажать **F1**, параметр равен нулю, если использование справки не предусмотрено.

Таблица. 3.2. Тип окна сообщения.

Параметр	Тип окна сообщения
mtInformation	информационное
mtWarning	предупредительное
mtError	сообщение об ошибке
mtConfirmation	запрос на подтверждение
mtCustom	обычное

Таблица. 3.3. Тип кнопки в окне сообщения.

Константа	Кнопка в окне сообщения
mbYes	Да
mbNo	Нет
mbOk	Ок
mbCancel	Отмена
mbAbort	Прервать
mbRetry	Повторить
mbIgnore	Пропустить
mbHelp	Помощь

Вернемся к задаче решения квадратного уравнения (задача 3.3). Нами был рассмотрен алгоритм решения этой задачи и написана программа на языке программирования Free Pascal. Реализуем эту задачу в среде Lazarus. Создадим новый проект[41] (рис. 3.20).

Рисунок 3.20: Форма для решения квадратного уравнения

Для организации ввода коэффициентов уравнения внедрим на форму четыре объекта типа надпись (Label1, Label2, Label3, Label4) и три поля ввода (Edit1, Edit2, Edit3). Корни уравнения или сообщение об их отсутствии будем выводить в надпись Label5[42].

Все действия по вычислению корней квадратного уравнения будут выполняться при нажатии кнопки Button1.

При вводе данных в программе могут возникнуть следующие ошибки:

- в поле ввода оказалась строка, которую невозможно преобразовать в число;
- значение коэффициента a равно нулю[43].

Для того чтобы не допустить подобных ошибок необходимо контролировать данные, вводимые пользователем. Применим для этой цели встроенную процедуру Val(S,X,Kod), которая преобразовывает строку S в целое или вещественное число X. Если преоб-

41 Подробно о создании проекта см. в первой главе.
42 На этапе конструирования формы Label5.Visible:=false.
43 В этом случае при вычислении корней произойдет деление на ноль.

разование прошло успешно, то параметр Kod принимает значение, равное нулю, а результат преобразования записывается в переменную x. В противном случае Kod содержит номер позиции в строке S, где произошла ошибка, и содержимое переменной X не меняется. Далее приведен фрагмент программы с подробными комментариями:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 a,b,c,d,x1,x2: real;
 kod1,kod2,kod3:integer;
begin
 //Ввод значений коэффициентов уравнения.
 //Из поля ввода Edit1 считывается строка
//символов и преобразовывается в вещественное
//число, если преобразование прошло успешно,
//то kod1=0 и полученное число записывается
//в переменную a.
 val(Edit1.Text,a,kod1);
 val(Edit2.Text,b,kod2);
 val(Edit3.Text,c,kod3);
 //Если преобразования прошли успешно, то
 if (kod1=0) and (kod2=0) and (kod3=0) then
     //проверить чему равен первый коэффициент.
     //Если значение первого коэффициента
     //равно нулю, то
     if a=0 then
         //выдать соответствующее сообщение.
         MessageDlg('Введите не нулевое
         значение a', mtInformation,[mbOk],0)
     else
     //иначе перейти к решению уравнения
     begin
         d:=b*b-4*a*c;
         Label5.Visible:=true;
         if d<0 then
             Label5.Caption:='В уравнении'+
             chr(13)+'нет действительных корней'
         else
         begin
```

```
            x1:=(-b+sqrt(d))/2/a;
            x2:=(-b-sqrt(d))/(2*a);
        Label5.Caption:='X1='+
                    FloatToStr(x1)+chr(13)+
                    'X2='+FloatToStr(x2);
        end;
    end
 else
//Преобразование не выполнено,
//выдать сообщение.
    MessageDlg('Введите числовое значение',
                    mtInformation,[mbOk],0);
end;
```

Результаты работы программы показаны на рис. 3.21 - 3.24.

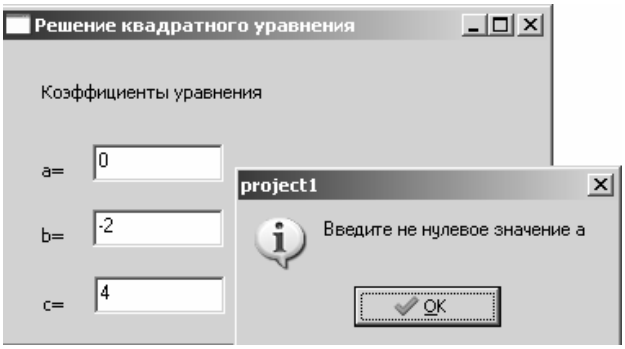

Рисунок 3.21: Обработка ошибки ввода данных — коэффициент a равен 0

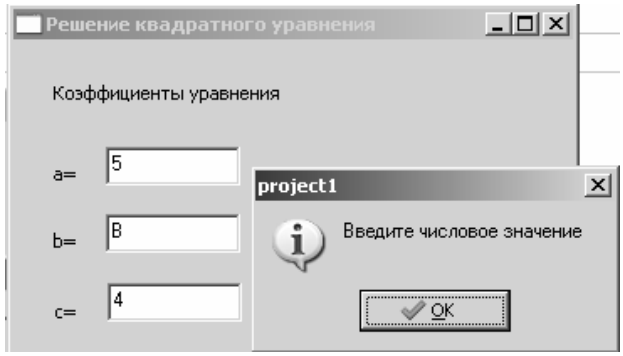

Рисунок 3.22: Обработка ошибки ввода данных - в поле ввода строка, которую невозможно преобразовать в число (коэффициент равен символу B)

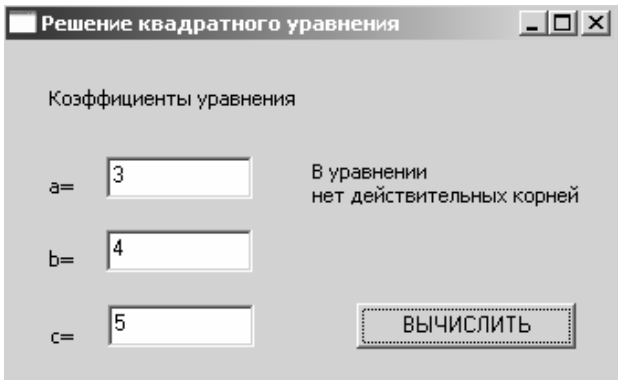

Рисунок 3.23: Решение квадратного уравнения $3x^2+4x+5=0$ (корней нет)

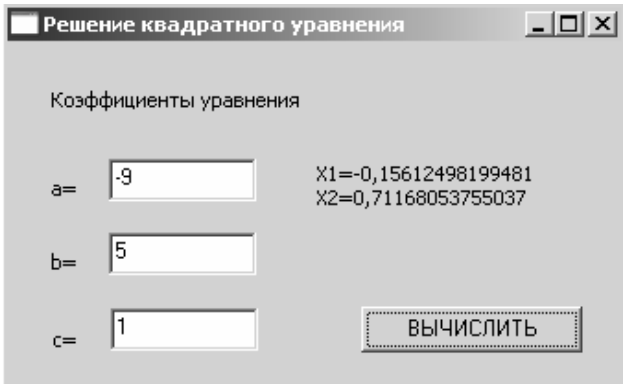

Рисунок 3.24: Вычисление корней квадратного уравнения

Метод обработки ошибок ввода, изложенный в этом примере, не единственный. Можно попытаться контролировать символы, поступающие в поля ввода. Набор символов, которые могут быть преобразованы в число, невелик. Это все цифры от 0 до 9, знак минус и символ запятой. Кроме того, пользователь должен иметь возможность удалять символы из поля ввода. Для этого ему понадобится клавиша `BackSpace` (`#8` – символ с кодом восемь). Представим набор перечисленных символов в виде множества, и если окажется, что введенный символ не принадлежит ему, то будем выводить сообщение об ошибке при вводе.

Возможность контролировать ввод символов обеспечивает событие `OnKeyPress`. Если выделить компонент `Edit2` и на вкладке инспектора объектов дважды щелкнуть возле события нажатия клавиши `OnKeyPress`, то будет создана процедура обработки этого события. Текст процедуры с комментариями:

```
//Обработка события ввод символа в поле Edit2.
procedure TForm1.Edit2KeyPress(Sender: TObject;
                               var Key: Char);
begin       //Если символ не принадлежит
            //множеству допустимых символов,то
 if not(Key in [#8,',','-','0'..'9']) then
 begin             //выдать сообщение и
    MessageDlg('Введите числовое значение',
                       mtInformation,[mbOk],0);
    Abort;  //прервать выполнение подпрограммы.
 end;
end;
```

3.6 Операторы цикла

Циклический процесс, или просто *цикл,* – это повторение одних и тех же действий. Последовательность действий, которые повторяются в цикле, называют *телом цикла*. Один проход цикла называют *шагом*, или *итерацией*[44]. Переменные, которые изменяются внутри цикла и влияют на его окончание, называются *параметрами цикла*.

44 Понятие итерации в математике и программировании несколько отличаются. В математике под итерацией понимают повторение какой-либо математической операции, использующее результат предыдущей аналогичной операции. В программировании итерация — это организация обработки данных, при котором действия повторяются многократно, не приводя при этом к вызовам самих себя (http://ru.wikipedia.org/wiki/%D0%98%D1%82%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F).

При написании циклических алгоритмов следует помнить следующее. Во-первых, чтобы цикл имел шанс когда-нибудь закончиться, содержимое его тела должно обязательно влиять на условие цикла. Во-вторых, условие должно состоять из корректных выражений и значений, определенных еще до первого выполнения тела цикла.

В языке Free Pascal для удобства программиста предусмотрены три оператора, реализующих циклический процесс: `while`, `repeat... until` и `for`.

3.6.1 Оператор цикла с предусловием while .. do

На рис. 3.25 изображена блок-схема алгоритма *цикла с предусловием*. Оператор, реализующий этот алгоритм в языке Free Pascal, имеет вид:

```
while выражение do оператор;
```

здесь `while .. do` – зарезервированные слова языка Free Pascal, `выражение` – логическая константа, переменная или логическое выражение, `оператор` – любой допустимый оператор языка.

Работает оператор `while` следующим образом. Вычисляется значение `выражения`. Если оно истинно (`True`), выполняется `оператор`. В противном случае цикл заканчивается, и управление передается оператору, следующему за телом цикла. `Выражение` вычисляется перед каждой итерацией цикла. Если при первой проверке `выражение` ложно (`False`), цикл не выполнится ни разу.

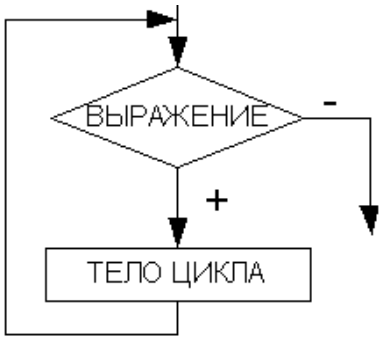

Рисунок 3.25: Алгоритм циклической структуры с предусловием

Если в цикле надо выполнить более одного оператора, необходимо использовать *составной оператор*:

```
while условие do
 begin
     оператор_1;
     оператор_2;
     …
     оператор_n;
 end;
```

Рассмотрим пример.

Пусть необходимо вывести на экран значения функции $y = e^{\sin(x)}\cos(x)$ на отрезке [0;π] с шагом 0.1. Применим *цикл с предусловием*:

```
var  x,y:real;
begin
 {Присваивание параметру цикла
 стартового значения.}
 x:=0;
 {Цикл с предусловием.}
 while x<=pi do     {Пока параметр цикла
        не превышает конечное значение,
        выполнять тело цикла.}
 begin
     {Вычислить значение y.}
     y:=exp(sin(x))*cos(x);
     {Вывод на экран пары x и y.}
     writeln('x=', x,'   y=', y);
     {Изменение параметра цикла -
      переход к следующему значению x.}
     x:=x+0.1;
 end; {Конец цикла.}
end.
```

В результате работы данного фрагмента программы на экран последовательно будут выводиться сообщения со значениями переменных x и y:

```
x= 0; y=1
x= 0.1; y=1.0995
…
x= 3.1; y=-1.0415
```

3.6.2 Оператор цикла с постусловием repeat … until

Если в цикле с предусловием проверка условия осуществляется до тела цикла, то в цикле с постусловием условие проверяется после тела цикла (см. рис. 3.26). Сначала выполняются операторы, являющиеся телом цикла, после чего проверяется условие, если последнее ложно, то цикл повторяется. Выполнение цикла прекратится, если условие станет истинным.

В языке Free Pascal *цикл с постусловием* реализован конструкцией

```
repeat
 оператор;
until выражение;
```

здесь `repeat .. until` – зарезервированные слова языка Free Pascal, `выражение` – логическая константа, переменная или логическое выражение, `оператор` – любой допустимый оператор языка.

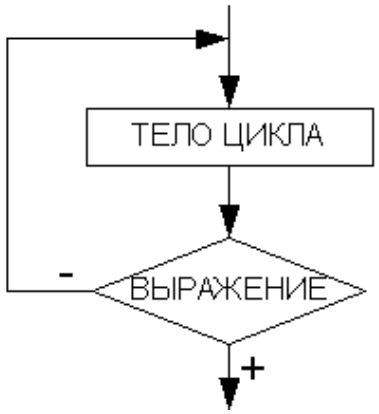

Рисунок 3.26: Алгоритм цикла с постусловием

Если тело цикла состоит более чем из одного оператора, то цикл с постусловием имеет вид:

```
repeat
 оператор_1;
 оператор_2;
 ..............
 оператор_N;
until выражение;
```

Работает цикл следующим образом. В начале выполняется `оператор`, представляющий собой тело цикла. Затем вычисляется значение `выражения`. Если оно ложно (`False`), `оператор` тела цикла выполняется еще раз. В противном случае цикл завершается, и управление передается оператору, следующему за циклом.

Таким образом, нетрудно заметить, что цикл с постусловием всегда будет выполнен хотя бы один раз в отличие от цикла с предусловием, который может не выполниться ни разу.

Если применить цикл с постусловием для создания подпрограммы, которая выводит значения функции $y=e^{\sin(x)}\cos(x)$ на отрезке $[0;\pi]$ с шагом 0.1, получим:

```
var
 x,y:real;
 begin
 {Присваивание параметру цикла
 стартового значения.}
 x:=0;
 {Цикл с постусловием.}
 repeat          {Начало цикла}
     y:=exp(sin(x))*cos(x);
     writeln('x=',x,'   y=',y);
     x:=x+0.1;   {Изменение параметра цикла.}
 until x>pi;     {Закончить работу цикла,
```

```
когда параметр превысит конечное значение.}
end.
```

3.6.3 Оператор цикла for … do

Операторы *цикла с условием* обладают значительной гибкостью, но не слишком удобны для организации «строгих» циклов, которые должны быть выполнены заданное число раз. Оператор цикла `for … do` используется именно в таких случаях:

```
for i:= in to ik do оператор;
for i:= ik downto in do оператор;
```

где `оператор` – любой оператор языка, `i`, `in` и `ik` — переменная целочисленного или перечислимого типов.

Переменную `i` называют *параметром цикла.* Переменные `in` и `ik` — *диапазон* изменения параметра цикла: `in` — начальное значение, а `ik` — конечное значение параметра цикла.

Шаг изменения цикла `for` всегда постоянен и равен интервалу между двумя ближайшими значениями типа параметра цикла (при целочисленном значении параметра цикла шаг равен 1).

В случае если тело цикла состоит более чем из одного оператора, необходимо использовать *составной оператор*:

```
for i:= in to ik do
 begin
     оператор_1;
     оператор_2;
     ..........
     оператор_n;
 end;
```

Опишем (рис.3.27) алгоритм работы *цикла* `for … do`:

1. Параметру цикла `i` присваивается начальное значение `in`.
2. Если значение параметра цикла превосходит конечное значение (`i>ik`), то цикл завершает свою работу. В противном случае выполняется пункт 3.
3. Выполняется `оператор`.
4. Значение параметра цикла `i` изменяется на соответствующий `шаг` и осуществляется переход к п.2 и т. д.

Понятно, что этот алгоритм представляет собой цикл с предусловием.

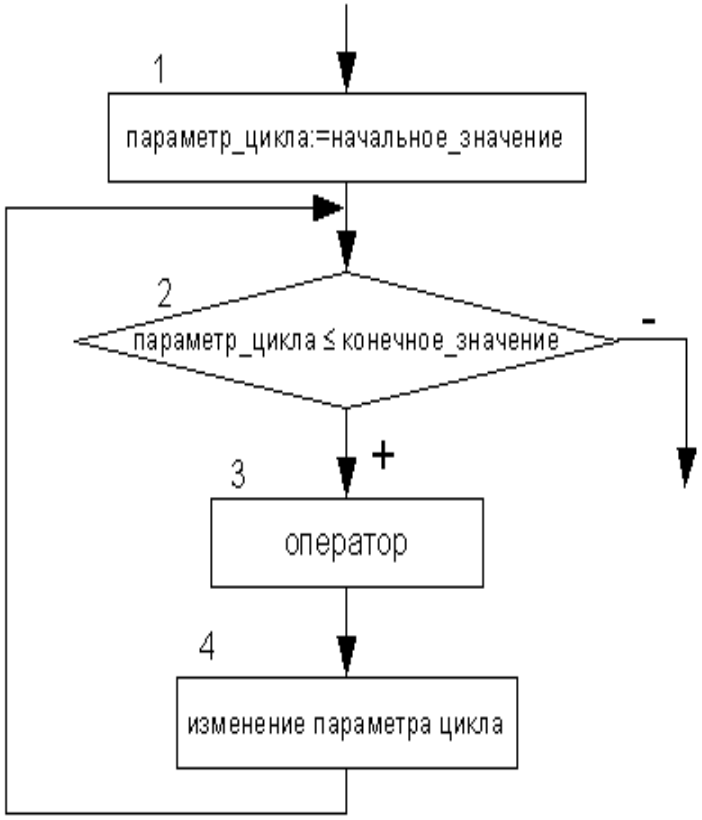

Рисунок 3.27: Алгоритм работы цикла for...do

В дальнейшем, чтобы избежать создания слишком громоздких алгоритмов, в блок-схемах цикл `for` будем изображать так, как показано на рис. 3.28[45].

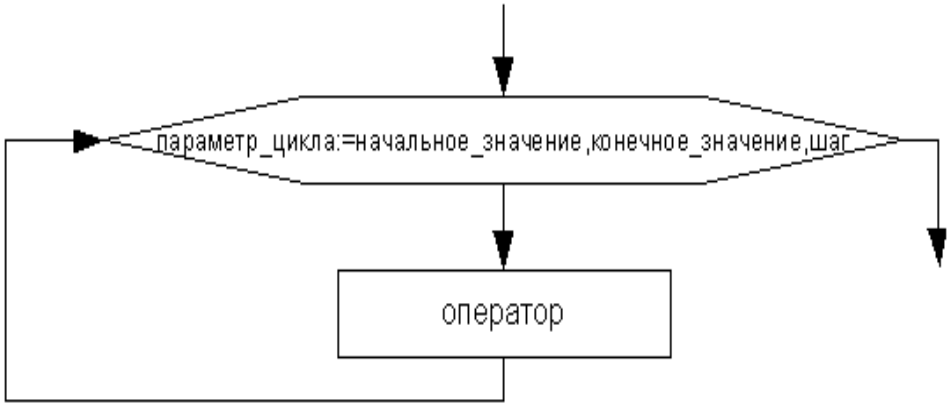

Рисунок 3.28: Представление цикла for...do с помощью блок-схемы

Фрагмент подпрограммы, приведенный далее, демонстрирует применение цикла `for`:

```
var i:integer; c:char;
 begin
```

45 Если шаг изменения параметра цикла равен единице, его в блок-схемах можно не указывать.

```
    {Вывод на экран чисел от 1 до 10.}
    for i:=1 to 10 do  writeln(i);
    {Вывод на экран чисел от 10 до -10.}
    for i:=10 downto -10 do writeln(i);
    {Вывод на экран символов от a до r.}
    for c:='a' to 'r' do writeln(c);
end;
```

Вернемся к задаче вывода значений функции $y=e^{\sin(x)}\cos(x)$ на отрезке [0;π] с шагом 0.1. Как видим, здесь количество повторений цикла явно не задано. Однако это значение, можно легко вычислить. Предположим, что параметр цикла x принимает значения в диапазоне от xn до xk, изменяясь с шагом dx, тогда количество повторений тела цикла можно определить по формуле:

$$n=\frac{xk-xn}{dx}+1 \quad , \qquad (3.6)$$

округлив результат деления до целого числа. Следовательно, фрагмент программы вывода значений функции $y=e^{\sin(x)}\cos(x)$ на отрезке [0;π] с шагом 0.1 будет иметь вид:

```
var i,n:integer; x,y:real;
 begin
 {Количество повторений цикла:}
 {n=(xk-xn)/dx+1; xk=pi, xn=0, dx=0.1}
 n:=round((pi-0)/0.1)+1;
 x:=0;   {Начальное значение аргумента.}
 {Цикл с известным числом повторений,}
 {i – параметр цикла,
 изменяется от 1 до n с шагом 1.}
 for i:=1 to n do
 begin              {Начало цикла.}
    {Вычисление значения функции }
    y:=exp(sin(x))*cos(x);
    {для соответствующего значения аргумента.}
    writeln('x=',x,'   y=',y);
    x:=x+0.1;       {Вычисление нового
                    значения аргумента.}
 end;               {Конец цикла.}
end.
```

3.7 Операторы передачи управления

Операторы передачи управления принудительно изменяют порядок выполнения команд. В языке Free Pascal таких операторов пять: `goto`, `break`, `continue`, `exit` и `halt`.

Оператор `goto метка`, где `метка` обычный идентификатор, применяют для безусловного перехода, он передает управление оператору с меткой:

```
метка: оператор;
```
[46].

Операторы `break` и `continue` используют только внутри циклов. Так, оператор `break` осуществляет немедленный выход из циклов `repeat`, `while`, `for` и управление передается оператору, находящемуся непосредственно за циклом. Оператор `continue` начинает новую итерацию цикла, даже если предыдущая не была завершена.

Оператор `exit` осуществляет выход из подпрограммы.

Оператор `halt` прекращает выполнение программы.

3.8 Решение задач с использованием циклов

Рассмотрим использование циклических операторов на конкретных примерах.

ЗАДАЧА 3.9. Найти наибольший общий делитель (НОД) двух натуральных чисел A и B.

Входные данные: `A` и `B`. *Выходные данные*: `A` – НОД.

Для решения поставленной задачи воспользуемся алгоритмом Евклида: будем уменьшать каждый раз большее из чисел на величину меньшего до тех пор, пока оба значения не станут равными, так как показано в таблице 3.4.

Таблица 3.4. Поиск НОД для чисел A=25 и B=15.

Исходные данные	**Первый шаг**	**Второй шаг**	**Третий шаг**	**Значение НОД для A и B равно 5**
A=25	A=10	A=10	A=5	
B=15	B=15	B=5	B=5	

В блок–схеме решения задачи, представленной на рис. 3.29, для решения поставленной задачи используется цикл с предусловием, то

46 Обычно применение оператора goto приводит к усложнению программы и затрудняет отладку. Он нарушает принцип структурного программирования, согласно которому все блоки, составляющие программу, должны иметь только один вход и один выход. В большинстве алгоритмов применения этого оператора можно избежать.

есть тело цикла повторяется до тех пор, пока A не равно B. Следовательно, при создании программы воспользуемся циклом while … do.

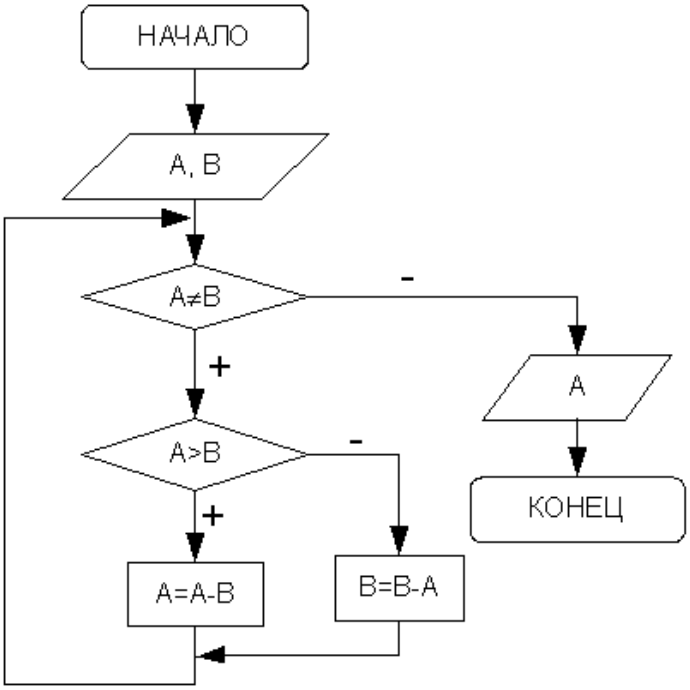

Рисунок 3.29: Алгоритм поиска наибольшего общего делителя двух чисел

Программа на языке Free Pascal, реализующая задачу 3.9:

```
var  a,b:word;
 begin
 writeln('введите два натуральных числа');
 write('A='); readln(A);
 write('B='); readln(B);
 {Если числа не равны, выполнять тело цикла.}
 while a<>b do
 {Если число A больше, чем B, то уменьшить
 его значение на B,}
 if a>b then
        a:=a-b
 {иначе уменьшить значение числа B на A.}
 else
        b:=b-a;
 writeln('НОД=',A);
end.
```

ЗАДАЧА 3.10. Вычислить факториал числа N (N!=1·2·3 …·N).

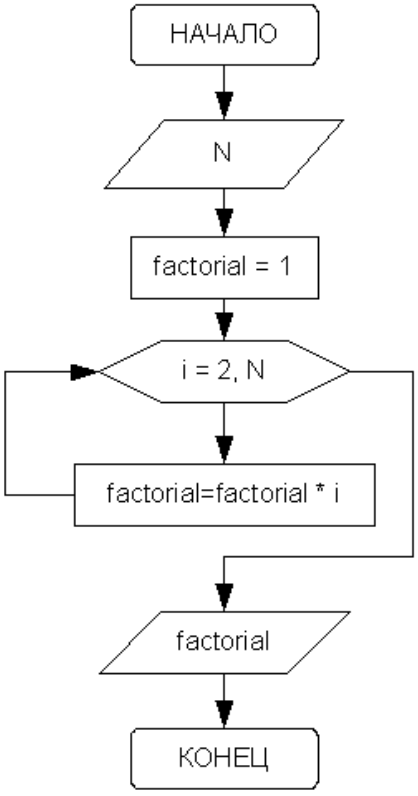

Рисунок 3.30: Алгоритм вычисления факториала

Входные данные: `N` – целое число, факториал которого необходимо вычислить.

Выходные данные: `factorial` *– значение факториала числа* `N`*, произведение чисел от 1 до* `N`*, целое число.*

Промежуточные переменные: `i` – параметр цикла, целочисленная переменная, последовательно принимающая значения 2, 3, 4 и так далее до `N`. Блок-схема приведена на рис. 3.30.

Итак, вводится число `N`. Переменной `factorial`, предназначенной для хранения значения произведения последовательности чисел, присваивается начальное значение, равное единице. Затем организуется цикл, параметром которого выступает переменная `i`.

Если значение параметра цикла меньше или равно `N`, то выполняется оператор тела цикла, в котором из участка памяти с именем `factorial` считывается предыдущее значение произведения, умножается на текущее значение параметра цикла, а результат снова помещается в участок памяти с именем `factorial`.

Когда параметр `i` становится больше `N`, цикл заканчивается, и выводится значение переменой `factorial`.

Теперь рассмотрим текст программы вычисления факториала на языке Free Pascal.

```
Var factorial, n, i:integer;
 begin
     write('n='); readln(n);
     factorial:=1;
     for i:=2 to n do
         factorial:=factorial*i;
     writeln(factorial);
 end.
```

ЗАДАЧА 3.11. Вычислить a^n , где n – целое положительное число.

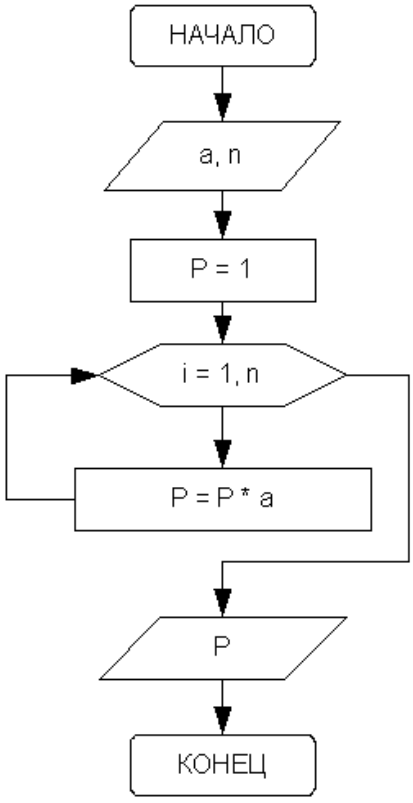

Рисунок 3.31: Алгоритм возведения вещественного числа в целую степень

Входные данные: a – вещественное число, которое необходимо возвести в целую положительную степень n. *Выходные данные*: p (вещественное число) – результат возведения вещественного числа a в целую положительную степень n. *Промежуточные данные*: i – целочисленная переменная, принимающая значения от 1 до n с шагом 1, параметр цикла.

Блок-схема приведена на рис. 3.31.

Известно, что для того чтобы получить целую степень n числа a, нужно умножить его само на себя n раз. Результат этого умножения будет храниться в участке памяти с именем p. При выполнении очередного цикла из этого участка предыдущее значение будет считываться, умножаться на основание степени a и снова записываться в участок памяти p. Цикл выполняется n раз.

В таблице 3.5 отображен протокол выполнения алгоритма при возведении числа 2 в пятую степень: a=2, n=5.

Подобные таблицы, заполненные вручную, используются для тестирования – проверки всех этапов работы программы.

Таблица 3.5. Процесс возведения числа a в степень n

i		1	2	3	4	5
P	1	2	4	8	16	32

Далее приведен текст программы, составленной для решения поставленной задачи.

```
Var  a,p:real; i,n:word;
 begin
     write('Введите основание степени a=');
     readln(a);
     write('Введите показатель степени n=');
     readln(n);
     p:=1;
```

```
        for i:=1 to n do p:=p*a;
        writeln('P=',P:1:3);
    end.
```

ЗАДАЧА 3.12. Вычислить сумму натуральных четных чисел, не превышающих N.

Входные данные: N – целое число.

Выходные данные: S – сумма четных чисел.

Промежуточные переменные: i – параметр цикла, принимает значения 2, 4, 6, 8 и так далее, также имеет целочисленное значение.

При сложении нескольких чисел необходимо накапливать результат в определенном участке памяти, каждый раз считывая из этого участка предыдущее значение суммы и прибавляя к нему следующее слагаемое. Для выполнения первого оператора накапливания суммы из участка памяти необходимо взять такое число, которое не влияло бы на результат сложения. Перед началом цикла переменной, предназначенной для накапливания суммы, необходимо присвоить значение нуль (s=0).

Блок-схема решения этой задачи представлена на рис. 3.32.Так как параметр цикла i изменяется с шагом 2, в блок–схеме, построенной для решения данной задачи, использован цикл с предусловием, который реализуется при составлении программы с помощью оператора while … do:

```
var
 n,i,S:word;
 begin
     write('n=');
     readln(n);
     S:=0;
     i:=2;
     while i<=n do
     begin
         S:=S+i;
         i:=i+2;
     end;
     writeln('S=',S);
end.
```

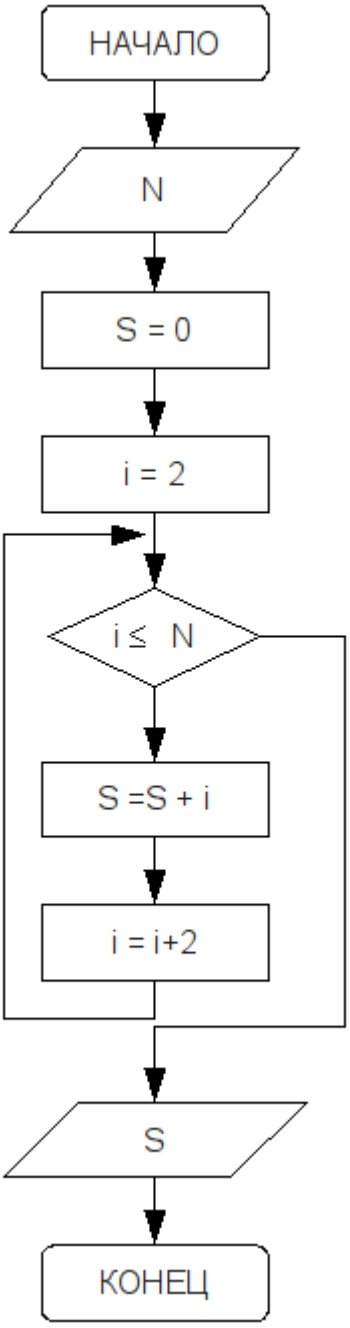

Рисунок 3.32: Алгоритм вычисления суммы четных натуральных чисел

Эту же задачу можно решить иначе, используя цикл `for … do`:

```
var
    n,i,S:word;
begin
    write('n=');
    readln(n);
```

```
    S:=0;
    for i:=1 to n do
    {Если остаток от деления параметра цикла
    на 2 равен 0, то это число четное,
    значит, происходит накапливание суммы.}
    if i mod 2 = 0 then
            S:=S+i;
    writeln('S=',S);
end.
```

В таблице 3.6 приведены результаты тестирования программы для n=7.

Таблица 3.6. Суммирование четных чисел

i		1	2	3	4	5	6	7
S	0	0	2	2	6	6	12	12

Несложно заметить, что при нечетных значениях параметра цикла значение переменной, предназначенной для накапливания суммы, не изменяется.

ЗАДАЧА 3.13. Дано натуральное число N. Определить К – количество делителей этого числа, не превышающих его (Например, для N=12 делители 1, 2, 3, 4, 6. Количество К=5).

Входные данные: N – целое число.

Выходные данные: целое число K – количество делителей N.

Промежуточные переменные: i – параметр цикла, возможные делители числа N.

В блок-схеме, изображенной на рис. 3.33, реализован следующий алгоритм: в переменную K, предназначенную для подсчета количества делителей заданного числа, помещается значение, которое не влияло бы на результат(k=0).

Далее организовывается цикл, в котором изменяющийся параметр i выполняет роль возможных делителей числа N.

Параметр цикла меняется от 1 до N/2 с шагом 1. Если заданное число делится нацело на параметр цикла, это означает, что i является делителем N, и значение переменной K следует увеличить на единицу. Цикл необходимо повторить N/2 раз.

Текст программы, соответствующий описанному алгоритму, приведен далее.

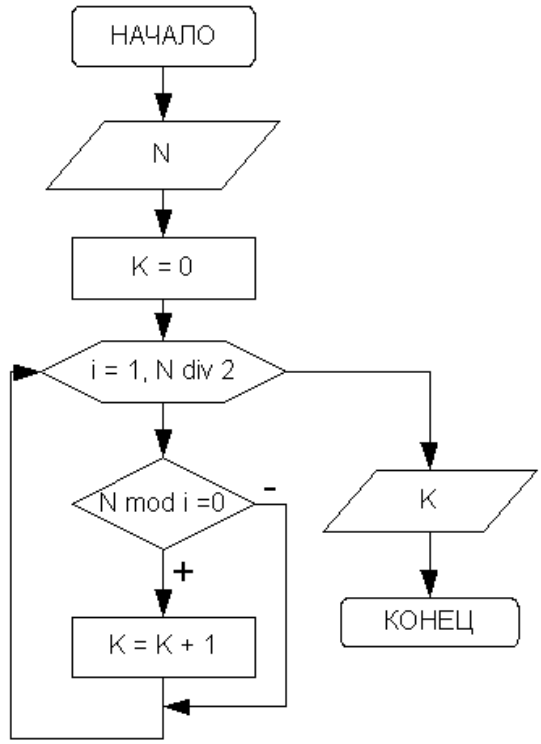

Рисунок 3.33: Алгоритм подсчета делителей натурального числа

```
var
N,i,K:word;
begin
write('N=');
readln(N);
K:=0;
for i:=1 to N div 2 do
{Если N делится
 нацело на i, то}
if N mod i= 0 then
{увеличить счетчик
 на один.}
k:=k+1;
writeln(' K=',K);
end.
```

В таблице 3.7 отображены результаты тестирования алгоритма при определении делителей числа `N=12`.

Таблица 3.7. Определение количества делителей числа `N`

i		1	2	3	4	5	6
K	0	1	2	3	4	4	5

ЗАДАЧА 3.14. Дано натуральное число N. Определить, является ли оно простым. Натуральное число N называется простым, если оно делится нацело без остатка только на единицу и N. Число 13 – простое, так как делится только на 1 и 13, N=12 не является простым, так как делится на 1, 2, 3, 4, 6 и 12.

Входные данные: `N` – целое число.

Выходные данные: сообщение.

Промежуточные данные: `i` – параметр цикла, возможные делители числа `N`.

Алгоритм решения этой задачи (рис. 3.34) заключается в том, что необходимо определить, есть ли у числа `N` делители среди чисел от `2` до `N/2`. Если делителей нет — число простое.

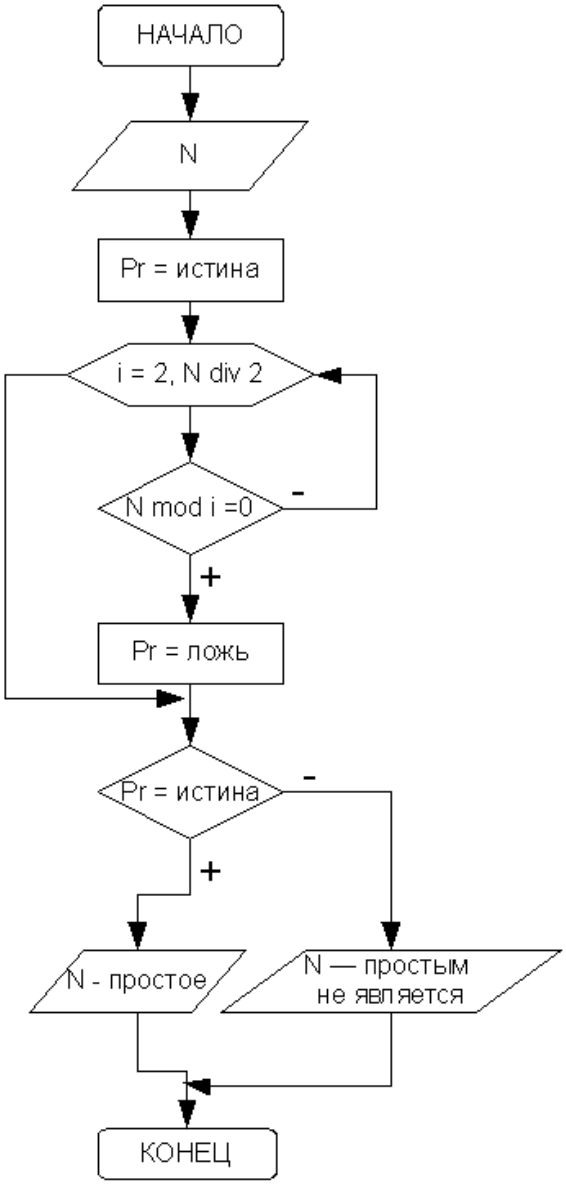

Рисунок 3.34: Алгоритм определения проситого числа

Предположим, что число N является простым (pr:=true). Организуем цикл, в котором переменная i будет изменяться от 2 до N/2.

В цикле будем проверять, делится ли N на i. Если делится, то мы нашли делитель, N не является простым (Pr:=false). Проверка остальных делителей не имеет смысла, поэтому покидаем цикл. В алгоритме предусмотрено два выхода из цикла. Первый – естественный, при исчерпании всех значений параметра, а второй — досрочный.

После выхода из цикла надо проверить значение Pr. Если Pr=true, то число N — простое, иначе N не является простым числом.

При составлении программы на языке Free Pascal досрочный выход из цикла удобно выполнять при помощи оператора break.

Текст программы:

```
var N,i:integer; Pr:boolean;
begin
    write('N='); readln(N);
    Pr:=true;  {Предположим, что число простое.}
    for i:=2 to N div 2 do
    {Если найдется хотя бы один делитель, то}
    if N mod i = 0 then
    begin
        Pr:=false; {число простым не является}
        break;     {досрочный выход из цикла.}
```

```
    end;
{Проверка значения логического параметра и}
if Pr then
{вывод на печать соответствующего сообщения.}
    writeln('Число ',N,' - простое')
else
    writeln('Число ',N,' простым не является');
end.
```

ЗАДАЧА 3.15. Определить количество простых чисел в интервале от N до M, где N и M – натуральные числа, причем N ≤ M.

Алгоритм решения данной задачи представлен на рис. 3.35.

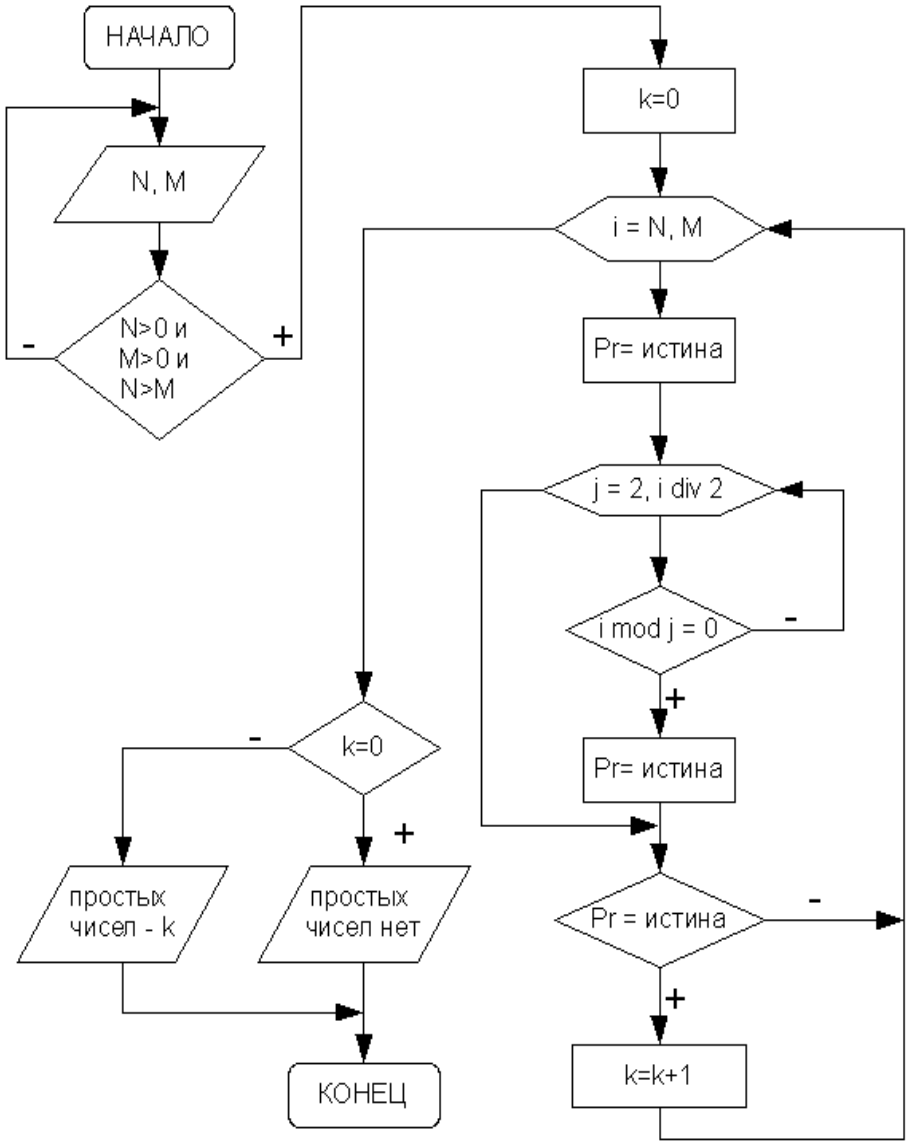

Рисунок 3.35: Алгоритм определения простых чисел в заданном интервале

Обратите внимание, что здесь осуществляется проверка корректности ввода исходных данных. Если границы интервала не положительны или значение N превышает M, ввод данных повторяется в цикле с постусловием до тех пор, пока не будут введены корректные исходные данные.

Далее для каждого числа из указанного интервала (параметр i принимает значения от N до M) происходит проверка. Если число является простым, то количество простых чисел k увеличивается на единицу. Подробно определение простого числа описано в задаче 3.14.

Программа на языке Free Pascal, реализующая алгоритм подсчета количества простых чисел в заданном диапазоне:

```
var N,M,i,j,k: longint; Pr:boolean;
begin
 repeat
     write('N='); readln(N);
     write('M='); readln(M);
 until (N>0) and (M>0) and (N<M);
 k:=0;       {Количество простых чисел}
{Параметр i принимает значения от N до M}
 for i:=N to M do
 begin
     {Определение простого числа.}
     Pr:=true;
     for j:=2 to i div 2 do
         if i mod j = 0 then
         begin
             Pr:=false;
             reak;
         end;
         {Если число простое,
         увеличиваем количество на 1.}
         if Pr then k:=k+1;
 end;
 if k=0 then writeln('Простых чисел нет')
 else writeln('Простых чисел в диапазоне ',k);
end.
```

ЗАДАЧА 3.16. Дано натуральное число N. Определить количество цифр в числе.

Входные данные: `N` – целое число.

Выходные данные: `kol` – количество цифр в числе.

Промежуточные данные: `M` – переменная для временного хранения значения N.

Для того чтобы подсчитать количество цифр в числе, необходимо определить, сколько раз заданное число можно разделить на десять нацело. Например, пусть `N=12345`, тогда количество цифр `kol = 5`. Результаты вычислений сведены в таблицу 3.8.

Таблица 3.8. Определение количества цифр числа

kol	N
1	12345
2	12345 div10=1234
3	1234 div 10=123
4	123 div 10=12
5	12 div 10=1
	1 div 10=0

Алгоритм определения количества цифр в числе представлен на рис. 3.36.Текст программы, реализующей задачу, можно записать так:

```
var M,N:longint; kol:word;
begin
{Так как речь идет о натуральных числах, при вводе предусмотрена проверка. Закончить цикл, если введено положительное число, иначе повторить ввод.}
repeat
    write('N=');readln(N);
until N>0;
M:=N;  {Сохранить значение переменной N. }
kol:=1;{Пусть число состоит из одной цифры.}
while M div 10 > 0 do
{Выполнять тело цикла, пока число делится на 10.}
begin
    kol:=kol+1;    {Счетчик количества цифр.}
    M:=M div 10;   {Изменение числа.}
end;
writeln('kol=', kol); end.
```

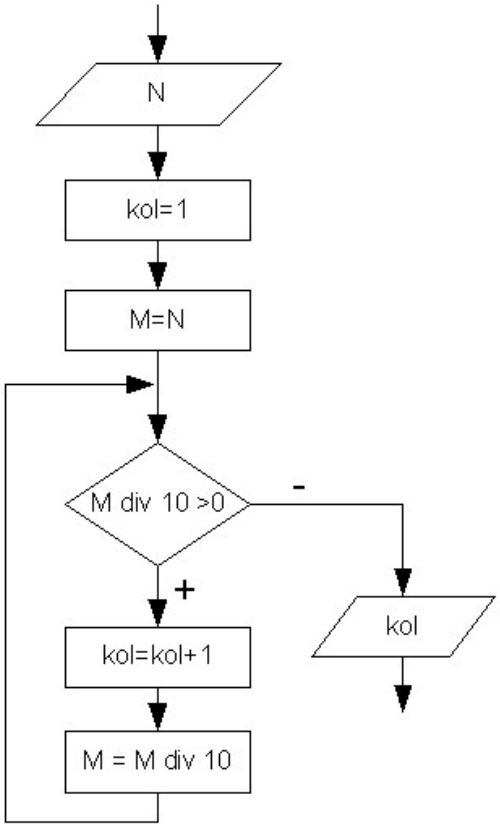

Рисунок 3.36: Алгоритм определения количества цифр в числе

ЗАДАЧА 3.17. Дано натуральное число N. Определить, содержит ли это число нули и в каких разрядах они расположены (например, число 1101111011 содержит ноль в третьем и восьмом разрядах).

Входные данные: `N` – целое число. *Выходные данные*: `pos` – позиция цифры в числе. *Промежуточные данные*: `i` – параметр цикла, `M` – переменная для временного хранения значения `N`.

В связи с тем что разряды в числе выделяются начиная с последнего, то для определения номера разряда в числе, необходимо знать количество цифр в числе[47]. Таким образом, на первом этапе решения задачи необходимо определить `kol` – количество цифр в числе. Затем начинаем выделять из числа цифры, если очередная цифра равна нулю, вывести на экран номер разряда, который занимает эта цифра. Процесс определения текущей цифры числа N=12345 представлен в таблице 3.9.

47 Алгоритм нахождения количества цифр в числе был рассмотрен в предыдущей задаче.

Таблица 3.9. Определение текущей цифры числа

i	Число M	Цифра	Номер позиции
1	120405	120405 mod 10= 5	6
2	12040 div10=1204	12040 mod 10 = 0	5
3	1204 div 10=120	1204 mod 10 = 4	4
4	120 div 10=12	120 mod 10 = 0	3
5	12 div 10=1	12 mod 10 = 2	2
6	1 div = 0	1 div 10 = 1	1

Блок-схема алгоритма решения данной задачи показана на рис. 3.37.

Текст программы, реализующей данный алгоритм:

```
var
 M,N:longint;
 i,pos,kol:word;
begin
 {Так как речь идет о натуральных числах,}
 {при вводе предусмотрена проверка.}
 {Закончить цикл, если введено}
 {положительное число, иначе повторить ввод}
 repeat
     write('N=');
     readln(N);
 until N>0;
//Определение kol - количества разрядов.
 M:=N;  {Сохранить значение переменной N. }
 kol:=1; {Пусть число состоит из одной цифры.}
 while M div 10 > 0 do
 {Выполнять тело цикла,}
 {пока число делится нацело на 10.}
 begin
     kol:=kol+1;    {Счетчик количества цифр.}
     M:=M div 10;   {Изменение числа.}
 end;
 writeln('kol=', kol);
 M:=N;
 pos:=0;        {Пусть в числе нет нулей.}
```

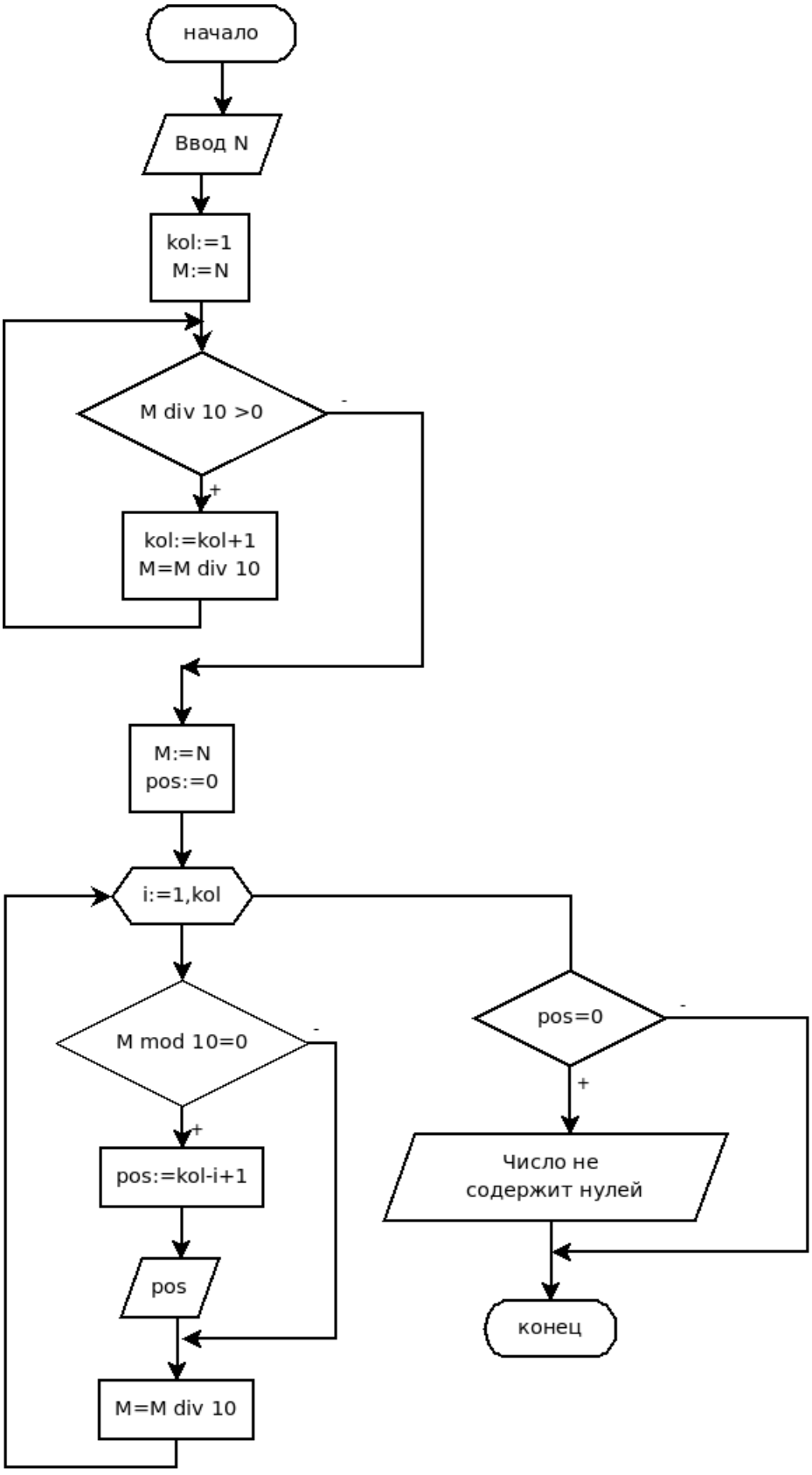

Рисунок 3.37: Алгоритм решения задачи 3.17

```
    for i:=1 to kol do
    begin
    {Выделение цифры из числа и сравнение ее с 0.}
        if (M mod 10 = 0) then
        begin
            pos:=kol-i+1;  {Позиция нуля в числе.}
            writeln('Ноль в ', pos, '-м разряде.');
        end;
        M:=M div 10;       {Изменение числа.}
    end;
    if pos=0 then
        writeln('Число не содержит цифру 0.');
end.
```

3.9 Ввод данных из диалогового окна в среде Lazarus

Окно ввода – это стандартное диалоговое окно, которое появляется на экране в результате вызова функции `InputBox`. В общем виде оператор ввода данных с использованием этой функции записывают так:

```
имя:=InputBox(заголовок_окна,подсказка,значение);
```

где

- `заголовок_окна` – строка, определяющая название окна;
- `подсказка` – текст поясняющего сообщения;
- `значение` – строка, которая будет находиться в поле ввода при появлении окна на экране;
- `имя` — переменная строкового типа, которой будет присвоено `значение` из поля ввода;

После выполнения фрагмента программы

```
var
 S:string;
begin
 S:=InputBox('ЗАГОЛОВОК ОКНА',
 'Подсказка: введите исходные данные',
                    'Данное значение');
end;
```

появится окно, представленное на рис. 3.38. У пользователя есть возможность изменять текст в поле ввода. Щелчок по кнопке **ОК** приве-

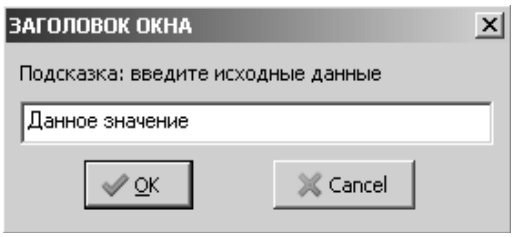

Рисунок 3.38: Окно ввода

дет к тому, что в переменную, указанную слева от оператора присваивания, будет занесена строка, находящаяся в поле ввода. В данном случае в переменную `S` будет записана строка `'Данное значение'`. Щелчок по кнопке **Cancel** закроет окно ввода.

Учитывая, что функция `InputBox` возвращает строковое значение, при вводе числовых данных применяют функции преобразования типов:

```
var S:string; gradus,radian:real;
begin
 S:=InputBox('Ввод данных',
 'Введите величину угла в радианах','0,000');
 gradus:=StrToFloat(S);
 radian:=gradus*pi/180;
 MessageDlg('Величина угла в градусах'
 +FloatToStr(radian),MtInformation,[mbOk],0);
end;
```

Можно применять диалоговое окно при решении задач, обрабатывающих некоторые *числовые последовательности*. Рассмотрим несколько таких задач.

ЗАДАЧА 3.18. Поступает последовательность из N вещественных чисел. Определить наибольший элемент последовательности.

Входные данные: `N` – целое число; `X` – вещественное число, определяет текущий элемент последовательности.

Выходные данные: `Max` – вещественное число, элемент последовательности с наибольшим значением.

Промежуточные переменные: `i` – параметр цикла, номер вводимого элемента последовательности.

Алгоритм поиска наибольшего элемента в последовательности следующий (рис. 3.39). В памяти компьютера отводится ячейка, например с именем `Max`, в которой будет храниться *наибольший элемент* последовательности – *максимум*.

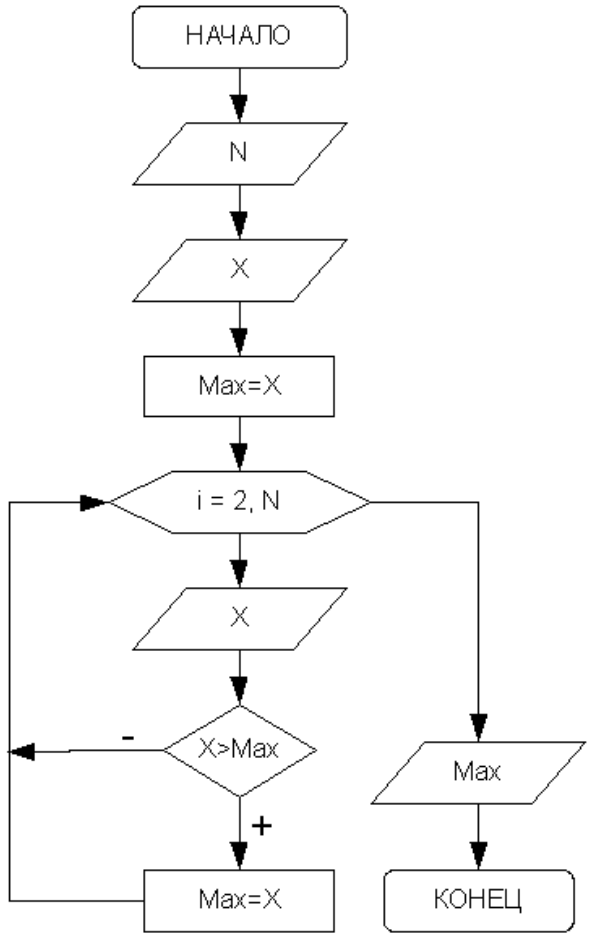

Рисунок 3.39: Алгоритм поиска наибольшего числа в последовательности

Вводим количество элементов последовательности и первый элемент последовательности. Предполагаем, что первый элемент последовательности наибольший и записываем его в `Max`. Затем вводится второй элемент последовательности и сравнивается с предполагаемым максимумом. Если окажется, что второй элемент больше, его записываем в ячейку `Max`. В противном случае никаких действий не предпринимает. Потом переходим к вводу следующего элемента последовательности, и алгоритм повторяется с начала. В результате в ячейке `Max` будет храниться элемент последовательности с наибольшим значением[48].

Разместим на форме объект типа надпись `Label1` и кнопку `Button1` (рис. 3.40).

Щелчок по кнопке приведет к выполнению приведенной далее процедуры.

```
procedure Tform1.Button1Click(
      Sender: TObject);
var
 i,N:integer;
 max,X:real;
 S:string;
begin
//Ввод количества элементов
//последовательности.
S:=InputBox('Ввод',
```

48 Для поиска *наименьшего элемента* последовательности (*минимума*) предполагают, что первый элемент – наименьший, записывают его в ячейку min, а затем среди элементов последовательности ищут число, значение которого будет меньше, чем предполагаемый минимум.

```
'Введите количество элементов в последователь-
ности.','0');
N:=StrToFloat(S);
//Ввод первого элемента последовательности.
    S:=InputBox('Ввод элементов
        последовательности','Введите число.','0');
    X:=StrToFloat(S);
    //Предположим, что 1-й элемент максимальный.
    max:=X;
    //Параметр цикла принимает стартовое
   //значение i=2, т.к. первый элемент уже введен.
    for i:=2 to N do
    begin
    //Ввод следующих элементов последовательности.
        S:=InputBox('Ввод элементов
        последовательности','Введите число.','0');
        X:=StrToInt(S);
        //Если найдется элемент превышающий
        //максимум, записать его в ячейку Max -
        //теперь он предполагаемый максимум.
        if X>max then max:=X;
    end;
    //Вывод наибольшего элемента.
    MessageDlg('Значение наибольшего элемента - '
    +FloatToStr(max),MtInformation,[mbOk],0);
   end;
```

Результаты работы программы представлены на рис. 3.40 - 3.42.

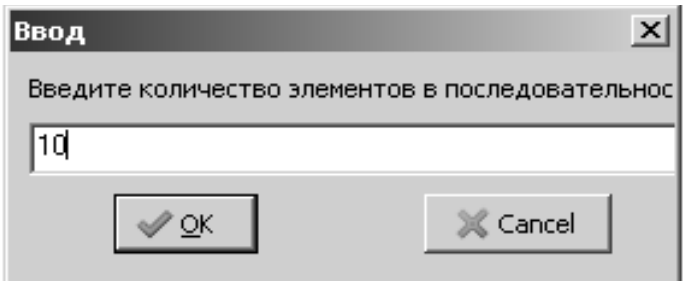

Рисунок 3.40: Второе окно диалога к задаче 3.18

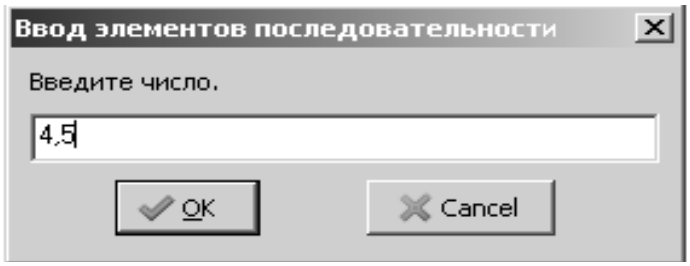

Рисунок 3.41: Третье окно диалога к задаче 3.18

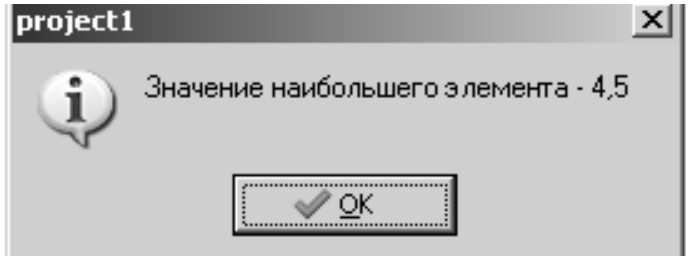

Рисунок 3.42: Результат работы программы к задаче 3.18

ЗАДАЧА 3.19. Вводится последовательность целых чисел, 0 – конец последовательности. Найти наименьшее число среди положительных, если таких значений несколько[49], определить, сколько их.

Блок-схема решения задачи приведена на рис. 3.43.

Далее приведен текст подпрограммы с подробными комментариями[50]. Подпрограмма выполняется при обращении к кнопке `Button1`, предварительно размещенной на форме.

```
procedure TForm1.Button1Click(Sender: TObject);
 var
    N,k,min:integer;
    S:string;
 begin
 //Ввод первого элемента последовательности.
 S:=InputBox('Ввод элементов последовательности', 'введите число. 0 - конец последовательности','0');
 N:=StrToInt(S);
```

49 Предположим, вводится последовательность чисел 11, -3, 5, 12, -7, 5, 8,-9, 7, -6, 10, 5,0.Наименьшим положительным числом является 5. Таких минимумов в последовательности 3.

50 Алгоритм поиска максимального (минимального) элемента последовательности подробно описан в задаче 3.18.

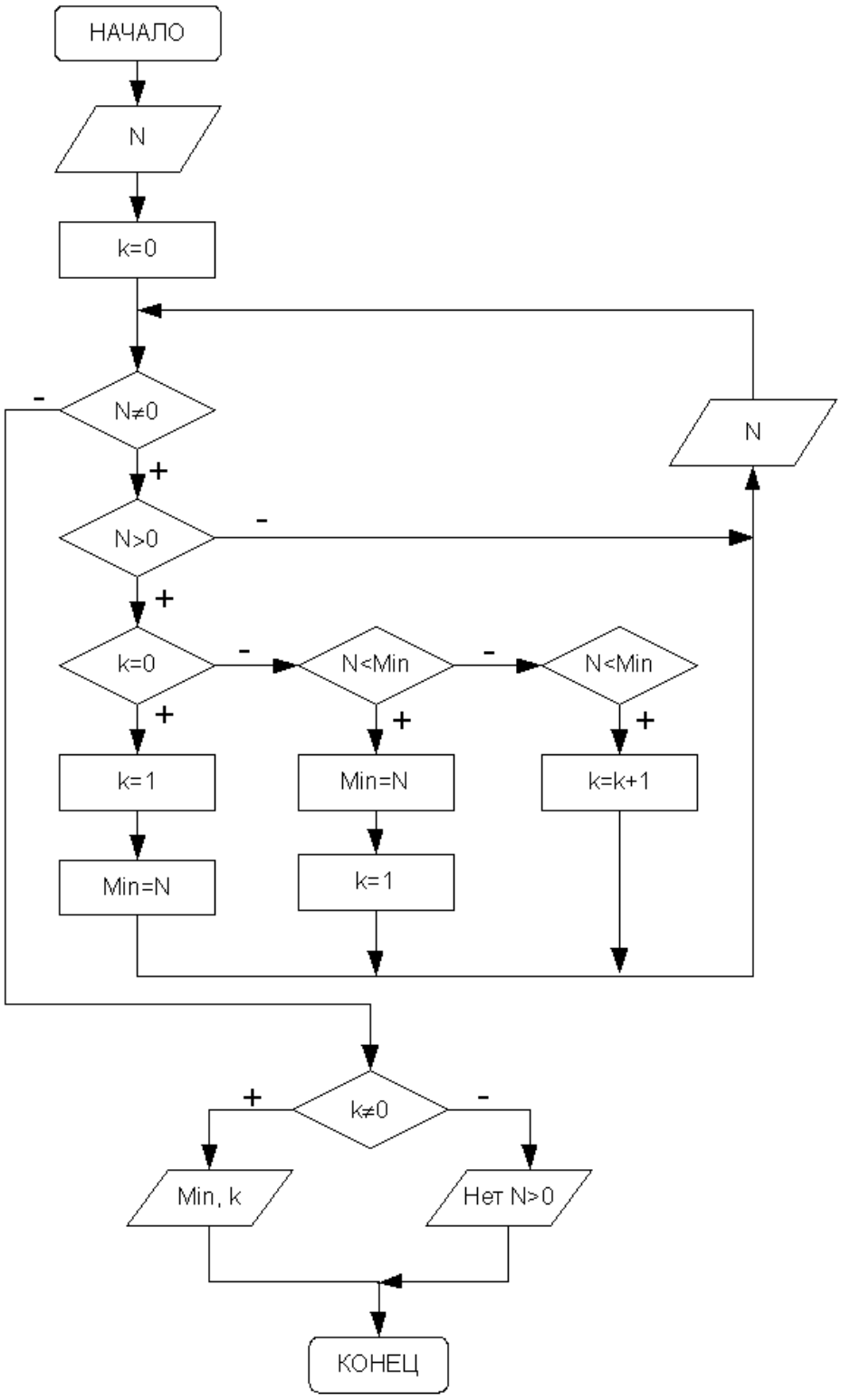

Рисунок 3.43: Алгоритм поиска минимального положительного числа в последовательности

```
//Предположим, что положительных чисел
//нет – К=0. В переменной К будет храниться
//кол-во минимумов среди положительных чисел.
 k:=0;
```

```
    //Пока введенное число не равно нулю,
   //выполнять тело цикла.
    while N<>0 do
    begin
    //Является ли введенное число положительным?
        if N>0 then
        begin
   //Если N>0 и K=0, поступил 1-й
   //положительный элемент, предположим,
   //что он минимальный Min=N, соответственно
   //количество минимумов равно 1.
            if k=0 then
            begin
                k:=1;
                min:=N;
            end
            //Если элемент не первый, сравниваем
            //его с предполагаемым минимумом, если
            //элемент меньше, записываем его в Min.
            //Соответственно количество
            //минимумов равно 1.
            else if N<min then
                begin
                    min:=N;
                    k:=1;
                end
            //Если элемент равен минимуму,
            //увеличиваем количество минимальных
            //элементов на 1.
            else if N=min then k:=k+1;
        end;
        //Ввод следующего элемента.
        S:=InputBox('Ввод элементов последователь-
ности', 'введите число. 0 - конец последовательно-
сти','0');
        N:=StrToInt(S);
    end;        //конец цикла
    if k<>0 then
```

```
        //Если значение счетчика не равно нулю,
        //выводим значение минимального элемента и
        //количество таких элементов,
        MessageDlg('MIN = '+IntToStr(min)+
        ' K='+IntToStr(k),MtInformation,[mbOk],0)
    else
        //в противном случае сообщаем,
        //что положительных чисел нет.
        MessageDlg('Положительных чисел нет',
                    MtInformation,[mbOk],0);
    end;
```

ЗАДАЧА 3.20. Определить, сколько раз последовательность из N произвольных чисел меняет знак.

Чтобы решить задачу, нужно попарно перемножать элементы последовательности. Если результат произведения пары чисел – отрицательное число, значит, эти числа имеют разные знаки.

Пусть `k` — количество смен знака последовательности равно 0. Пусть в переменной `A` храниться текущий элемент последовательности, а в переменой `B` – предыдущий. Введем `N` — количество элементов последовательности. Организуем цикл (переменная `i` меняется от `1` до `N`). В цикле будем делать следующее: вводим очередной элемент последовательности (`A`), если это первый элемент последовательности (`i=1`), то сравнивать его не с чем, и просто переписываем переменную `A` в переменную `B` (`B:=A`). Если это не первый элемент последовательности (`i≠1`), то проверяем знак произведения $A \cdot B$ (текущего и предыдущего элементов последовательности). Если произведение `<0`, то счетчик `k` увеличиваем на `1`. После чего не забываем в переменную `B` записать `A`.

Блок-схема алгоритма приведена на рис. 3.44.

Разметим на форме два объекта типа надпись `Label1` и `Label2`, объект поле ввода `Edit1` и кнопку `Button1` (рис. 3.45). Текст подпрограммы, которая будет выполнена при обращении к кнопке, приведен далее.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    N,A,B,i,k:integer;
    S:string;
```

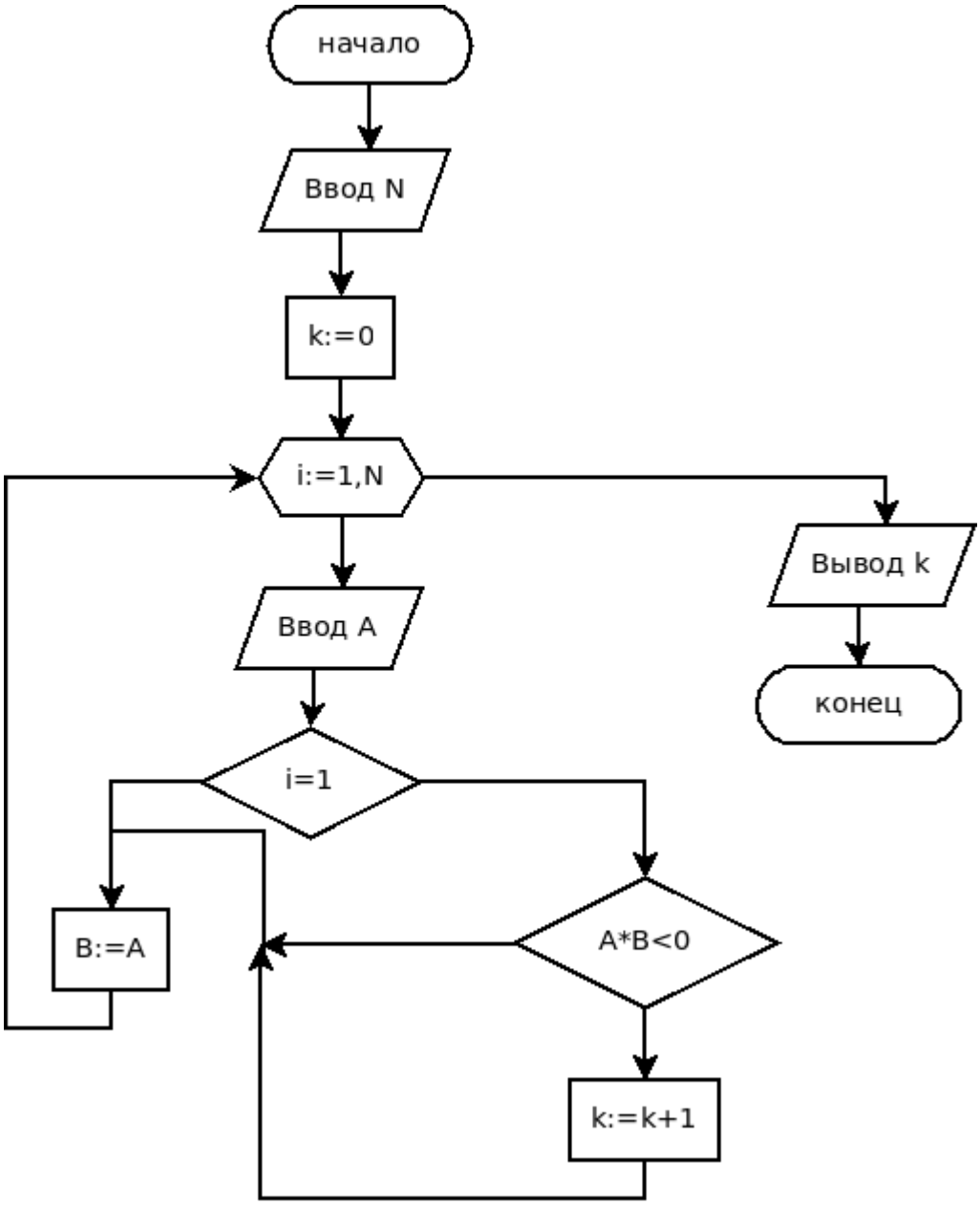

Рисунок 3.44: Алгоритм к задаче 3.20

```
   begin
    N:=StrToInt(Edit1.Text);
    S:=InputBox('Ввод элементов последователь-
ности', 'Введите число. 0 - конец последователь-
ности','0');
    A:=StrToInt(S);
    k:=0;
    for i:=1 to N do
    begin
        S:=InputBox('Ввод элементов последователь-
ности', 'введите число. 0 -конец последователь-
```

```
сти','0');
        A:=StrToInt(S);
        if (i<>1) then
        if A*B<0 then
                k:=k+1;
        B:=A;
    end;
    MessageDlg('K = '+IntToStr(k),
                    MtInformation,[mbOk],0);
end;
```

Результаты работы программы представлены на рис. 3.46-3.47.

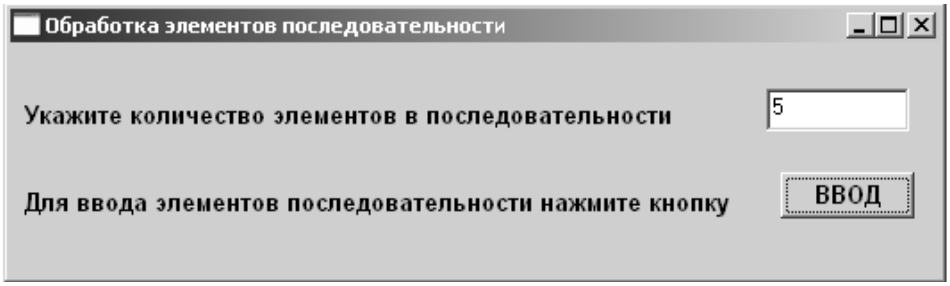

Рисунок 3.45: Первое окно диалога к задаче 3.19

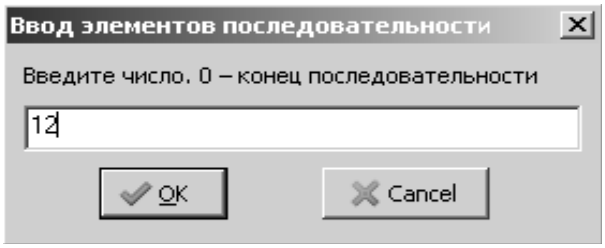

Рисунок 3.46: Второе окно диалога к задаче 3.19

Рисунок 3.47: Результат работы программы к задаче 3.19

3.10 Задачи для самостоятельного решения

Изобразите блок-схему решения задачи и напишите программу.

3.10.1 Разветвляющийся процесс

Дано вещественное число *a*. Для функции *y=f*(*x*), график которой приведен ниже, вычислить *f*(*a*). Варианты заданий представлены на рис. 3.48 -3.55.

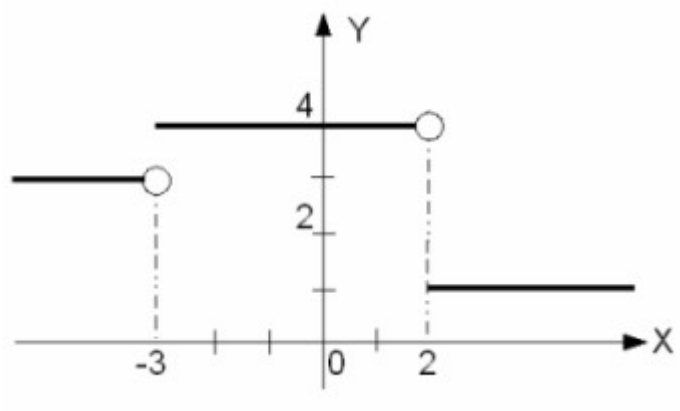

Рисунок 3.48: Задание 1

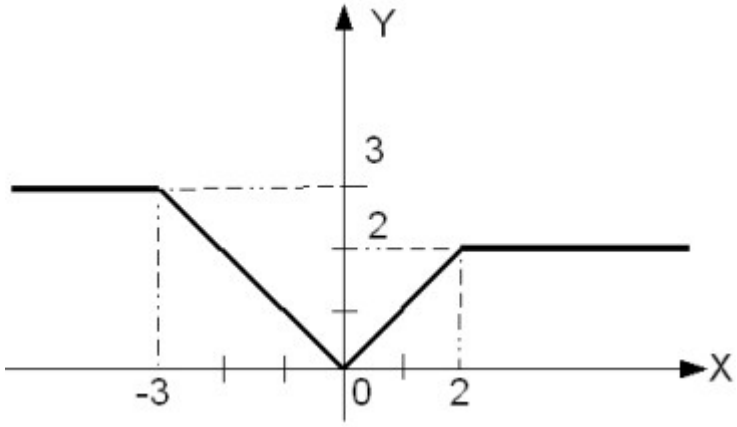

Рисунок 3.49: Задание 2

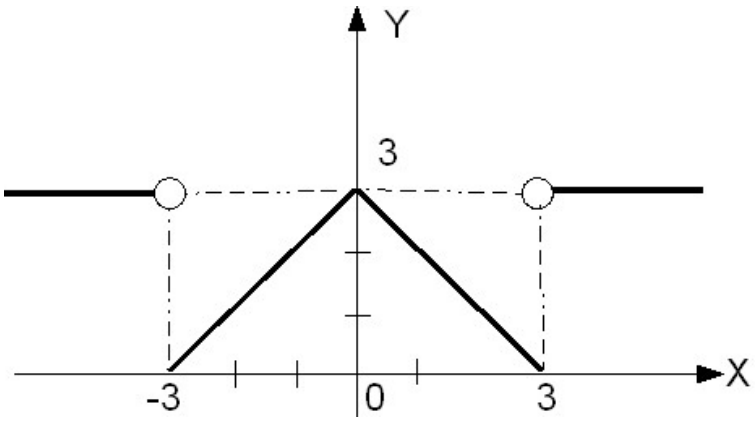

Рисунок 3.50: Задание 3

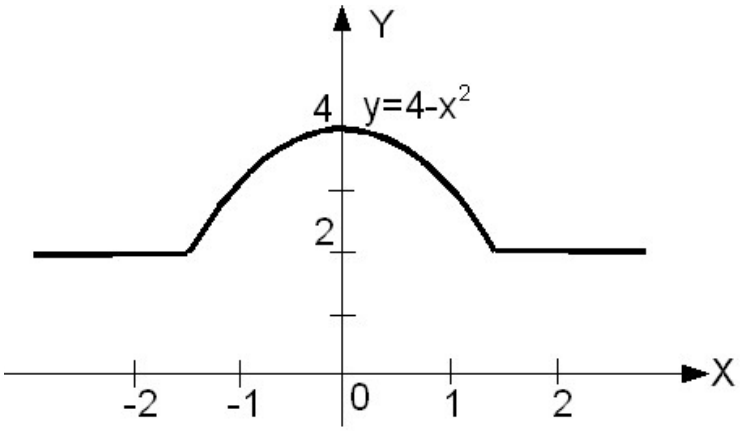

Рисунок 3.51: Задание 4

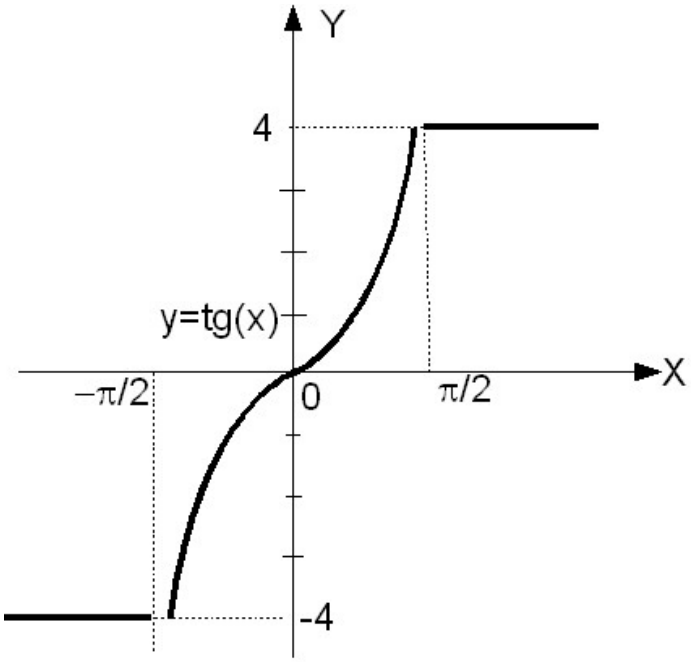

Рисунок 3.52: Задание 5

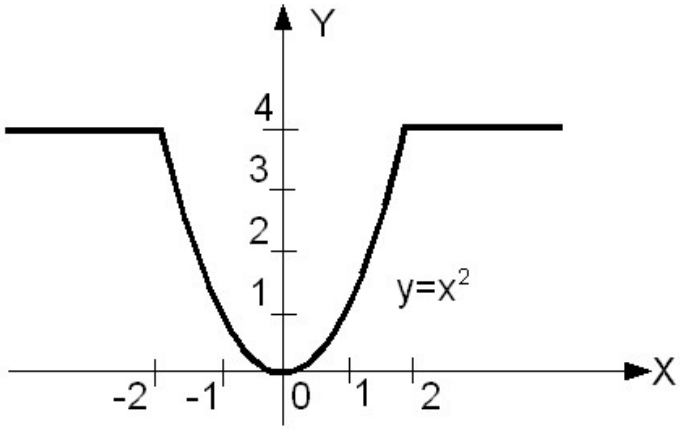

Рисунок 3.53: Задание 6

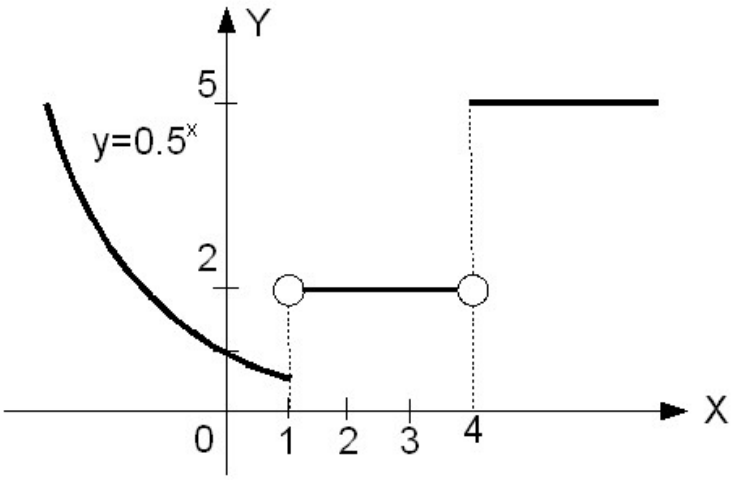

Рисунок 3.54: Задание 7

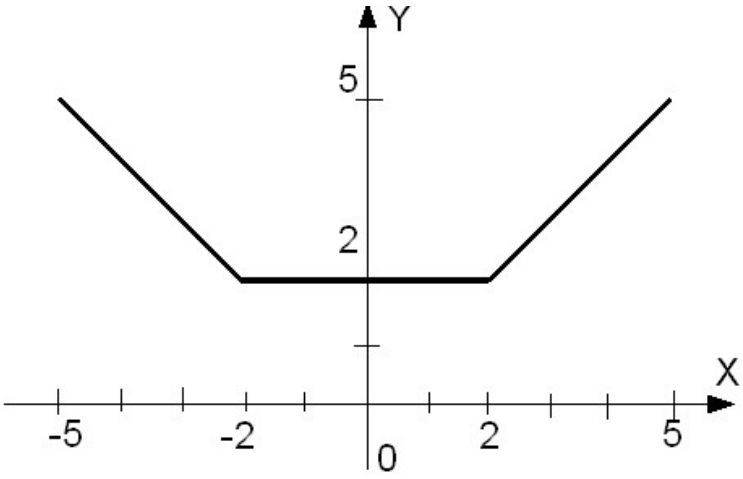

Рисунок 3.55: Задание 8

Даны вещественные числа *x* и *y*. Определить принадлежит ли точка с координатами (*x*; *y*) заштрихованной части плоскости. Варианты заданий представлены на рис. 3.56 - 3.63.

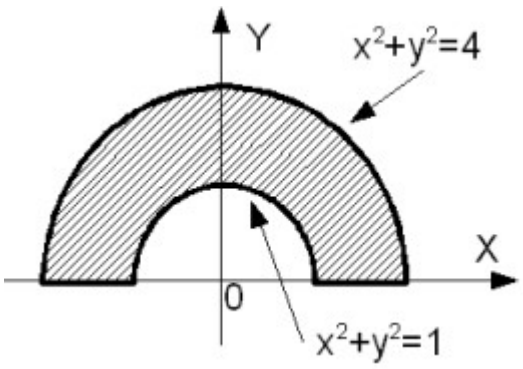

Рисунок 3.56: Задание 9

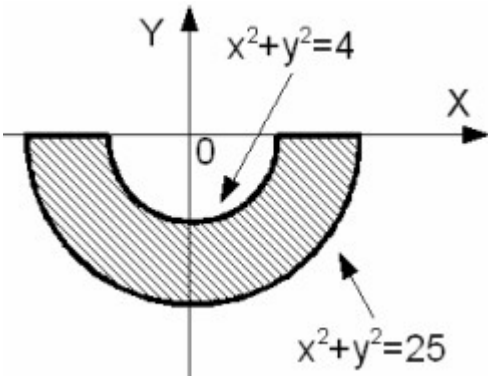

Рисунок 3.57: Задание 10

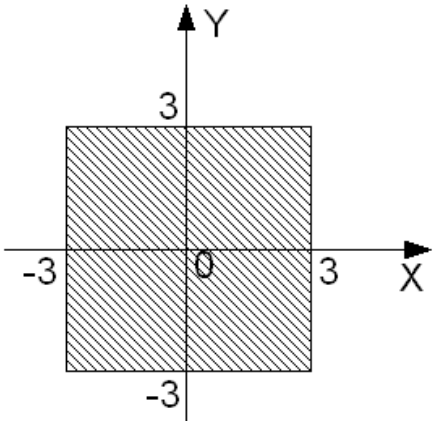

Рисунок 3.58: Задание 11

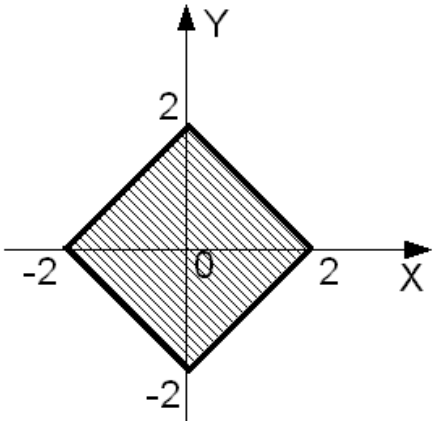

Рисунок 3.59: Задание 12

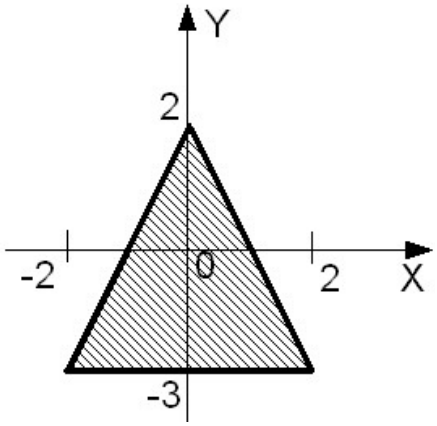

Рисунок 3.60: Задание 13

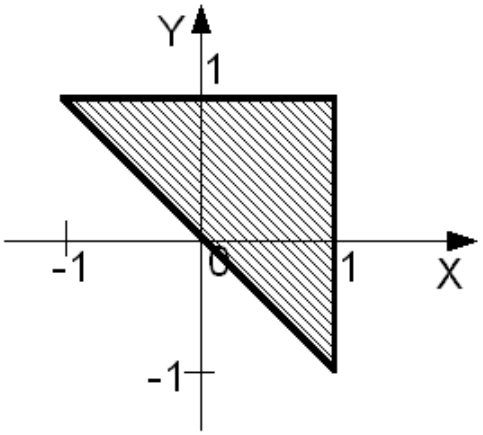

Рисунок 3.61: Задание 14

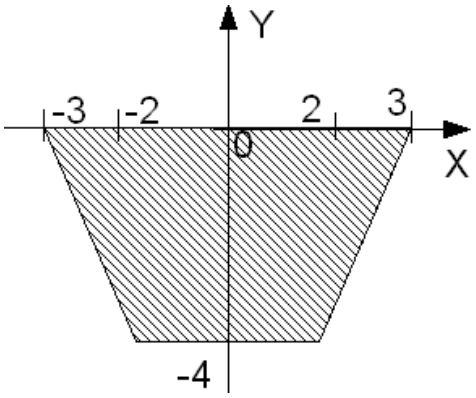

Рисунок 3.62: Задание 15

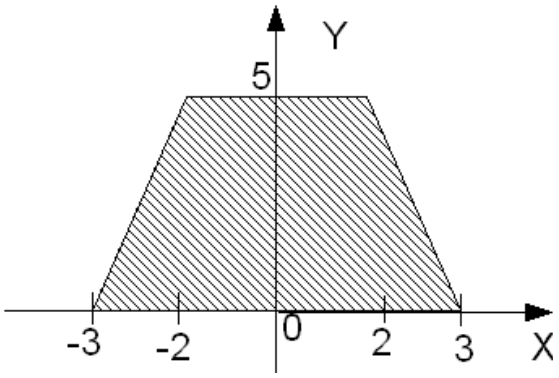

Рисунок 3.63: Задание 16

Решить следующие задачи:

17. Задан круг с центром в точке $O(x_0, y_0)$ и радиусом R_0 и точка $A(x_1, y_1)$. Определить, находится ли точка внутри круга.

18. Определить, пересекаются ли параболы $y=ax^2+bx+c$ и $y=dx^2+mx+n$. Если пересекаются, то найти точку пересечения.

19. Определить, пересекаются ли линии $y=ax^3+bx^2+cx+d$ и $y=kx^3+mx^2+nx+p$. Если пересекаются, найти точку пересечения.

20. Задана окружность с центром в точке $O(x_0, y_0)$ и радиусом R_0, найти точки пересечения линии с осью абсцисс.

21. Задана окружность с центром в точке $O(x_0, y_0)$ и радиусом R_0, найти точки пересечения линии с осью ординат.

22. Определить, пересекаются ли линии $y=bx^2+cx+d$ и $y=kx+m$. Если пересекаются, найти точки пересечения.

23. Задана окружность с центром в точке $O(0,0)$ и радиусом R_0 и прямая $y=ax+b$. Определить, пересекаются ли прямая и окружность. Если пересекаются, найти точку пересечения.

24. Найти точки пересечения линии $y= ax^2+bx+c$ с осью абсцисс.

25. Определить, пересекаются ли линии $y=ax^4+bx^3+cx^2+dx+f$ и $y=bx^3+mx^2+dx+p$. Если пересекаются, найти точку пересечения.

3.10.2 Циклический процесс

1. Вычислить сумму натуральных нечетных чисел, не превышающих N.

2. Вычислить произведение натуральных четных чисел, не превышающих N.

3. Вычислить количество натуральных чисел, кратных трем и не превышающих N.

4. Задано число n. Определить значение выражения:

$$P = \frac{n!}{\sum_{i=1}^{n} i}.$$

5. Вводится последовательность ненулевых чисел, 0 – конец последовательности. Определить сумму положительных элементов последовательности.

6. Вводится последовательность ненулевых чисел, 0 – конец последовательности. Определить, сколько раз последовательность поменяет знак.

7. Вычислить сумму отрицательных элементов последовательности из *N* произвольных чисел.

8. В последовательности из *N* произвольных чисел подсчитать количество нулей.

9. Вводится последовательность ненулевых чисел, 0 – конец последовательности. Определить наибольшее число в последовательности.

10. Дано натуральное число *P*. Определить все простые числа, не превосходящие *P*.

11. Определить, является ли число *L* совершенным. Совершенное число L равно сумме всех своих делителей, не превосходящих *L*. Например, 6=1+2+3 или 28=1+2+4+7+14. В основе решения задачи лежит алгоритм из задачи 3.13.

12. Вводится последовательность ненулевых чисел, 0 – конец последовательности. Определить среднее значение элементов последовательности.

13. Вводится последовательность из *N* произвольных чисел, найти наименьшее положительное число.

14. Вводится последовательность из *N* произвольных чисел, найти среднее значение положительных элементов последовательности.

15. Вводится последовательность ненулевых чисел, 0 – конец последовательности. Подсчитать процент положительных и отрицательных чисел.

16. Вводится последовательность из *N* произвольных чисел. Определить процент положительных, отрицательных и нулевых элементов.

17. Вводится последовательность положительных целых чисел, 0 – конец последовательности. Определить количество совершенных чисел (см. вариант 11).

18. Вводится последовательность из *N* произвольных чисел. Вычислить разность между наименьшим и наибольшим значениями последовательности.

19. Дано натуральное число *P*. Определить все совершенные числа (см. вариант 11), не превосходящие *P*.

20. Вводится последовательность из *N* положительных целых чисел. Найти наименьшее число среди четных элементов последовательности.

21. Вводится последовательность положительных целых чисел, 0 – конец последовательности. Определить, является ли эта последовательность знакочередующейся.

22. Задано число *P*. Если это число простое вычислить *P*!.

23. Вводится последовательность из *N* произвольных чисел. Найти наибольшее число в последовательности. Если таких чисел несколько, определить, сколько их.

24. Задано число *P*. Определить количество его четных и нечетных делителей.

25. Определить, является ли последовательность из *N* произвольных чисел строго возрастающей (то есть каждый следующий элемент больше предыдущего).

4 Подпрограммы

В практике программирования часто складываются ситуации, когда одну и ту же группу операторов, реализующих определенную цель, требуется повторить без изменений в нескольких местах программы. Для избавления от столь нерациональной траты времени была предложена концепция подпрограммы.

Подпрограмма – именованная, логически законченная группа операторов языка, которую можно вызвать для выполнения любое количество раз из различных мест программы. В языке Free Pascal существуют два вида подпрограмм: *процедуры* и *функции*. Главное *отличие процедуры от функции* заключается в том, что результатом исполнения операторов, составляющих тело функции, всегда является некоторое значение, поэтому функцию можно использовать непосредственно в выражениях, наряду с переменными и константами.

4.1 Общие сведения о подпрограммах. Локальные и глобальные переменные

Итак, *подпрограмма* – это поименованный набор описаний и операторов, выполняющих определенную задачу. Информация, передаваемая в подпрограмму для обработки, называется *параметрами*, а результат вычислений – *значениями*. Обращение к подпрограмме называют *вызовом*. Перед вызовом подпрограмма должны быть обязательно описана в разделе описаний. *Описание подпрограммы* состоит из заголовка и тела. В *заголовке* объявляется имя подпрограммы и в круглых скобках ее параметры, если они есть (для функции необходимо сообщить тип возвращаемого ею результата). *Тело подпрограммы* следует за заголовком и состоит из описаний и исполняемых операторов.

Любая подпрограмма может содержать описание других подпрограмм. Константы, переменные, типы данных могут быть объявлены как в основной программе, так и в подпрограммах различной степени вложенности. Переменные, константы и типы, объявленные в основной программе до определения подпрограмм, называются *глобальными*, они доступны всем функциям и процедурам. Переменные, константы и типы, описанные в какой-либо подпрограмме, доступны только в ней и называются *локальными*.

Для правильного определения области действия идентификаторов (переменных) необходимо придерживаться следующих правил:

- каждая переменная, константа или тип должны быть описаны перед использованием;
- областью действия переменной, константы или типа является та подпрограмма, в которой они описаны;
- все имена в пределах подпрограммы, в которой они объявлены, должны быть уникальными и не должны совпадать с именем самой подпрограммы;
- одноименные локальные и глобальные переменные – это разные переменные, обращение к таким переменным в подпрограмме трактуется как обращение к локальным переменным (глобальные переменные недоступны);
- при обращении к подпрограмме доступны объекты, которые объявлены в ней и до ее описания.

4.2 Формальные и фактические параметры. Передача параметров в подпрограмму

Обмен информацией между вызываемой и вызывающей функциями осуществляется с помощью механизма передачи параметров. Переменные, указанные в заголовке подпрограммы называются *формальными параметрами* или просто параметрами подпрограммы. Эти переменные могут использоваться внутри подпрограммы. Список переменных в операторе вызова подпрограммы – это *фактические параметры,* или *аргументы*.

Механизм передачи параметров обеспечивает обмен данных между формальными и фактическими параметрами, что позволяет выполнять подпрограмму с различными данными. Между фактическими параметрами в операторе вызова и формальными параметрами в заголовке подпрограммы устанавливается взаимно однозначное соответствие. *Количество, типы и порядок следования формальных и фактических параметров должны совпадать*.

Передача параметров выполняется следующим образом. Вычисляются выражения, стоящие на месте фактических параметров. В памяти выделяется место под формальные параметры в соответствии с их типами. Выполняется проверка типов и при их несоответствии выдается диагностическое сообщение. Если количество и типы формальных и фактических параметров совпадают, то начинает работать

механизм передачи данных между фактическими и формальными параметрами.

Формальные параметры процедуры можно разделить на два класса: *параметры-значения* и *параметры-переменные*.

При передаче данных через *параметры-значения* в подпрограмму передаются значения фактических параметров, и доступа к самим фактическим параметрам из подпрограммы нет.

При передаче данных *параметры-переменные* заменяют[51] формальные параметры, и, следовательно, в подпрограмме есть доступ к значениям фактических параметров. Любое изменение параметров-переменных в подпрограмме приводит к изменению соответствующих им формальных параметров. Следовательно, *входные данные следует передавать через параметры-значения*, для *передачи изменяемых в результате работы подпрограммы данных следует использовать параметры-переменные*.

От общетеоретических положений перейдем к практическому использованию подпрограмм при решении задач. Изучение подпрограмм начнем с процедур.

4.3 Процедуры

Описание процедуры имеет вид:

```
procedure имя_процедуры(формальные_параметры);
    label
        описание_меток;
    const
        описание_констант;
    type
        описание_типов;
    var
        описание_переменных;

    begin
        //Тело процедуры.
    end;
```

Начинается описание с *заголовка процедуры*, где `procedure` – ключевое слово языка, `имя_процедуры` – любой допустимый в языке Free Pasacal идентификатор, `формальные_параметры` – имена

51 Реально в подпрограмму передаются адреса фактических параметров.

формальных параметров и их типы, разделенные точкой с запятой. Рассмотрим примеры заголовков процедур с параметрами-значениями:

```
procedure name_1(r:real; i:integer; c:char);
```

Однотипные параметры могут быть перечислены через запятую:

```
procedure name_2(a,b:real; i,j,k:integer);
```

Список формальных параметров не обязателен и может отсутствовать:

```
procedure name_3;
```

Если в заголовке процедуры будут применяться *параметры-переменные*, то перед ними необходимо указывать служебное слово `var`, перед *параметрами-значениями* слово `var` отсутствует:

```
procedure name_4(x,y:real; var z:real);
                  //x, y – параметры-значения,
                  //z – параметр-переменная.
```

После заголовка идет *тело процедуры*, которое состоит из раздела описаний[52] (константы, типы, переменные, процедуры и функции, используемые в процедуре) и операторов языка, реализующих алгоритм процедуры.

Для обращения к процедуре необходимо использовать *оператор вызова*:

```
имя_процедуры(список_фактических_параметров);
```

Фактические параметры в списке оператора вызова отделяются друг от друга запятой:

```
a:=5.3; k:=2;
s:='a';
name_1(a, k, s);
```

Если в описании процедуры формальные параметры отсутствовали, то и при вызове их быть не должно:

```
name_3;
```

ЗАДАЧА 4.1. Найти действительные корни квадратного уравнения $ax^2+bx+c=0$.

Алгоритм решения этой задачи был подробно описан в задаче 3.3 (рис. 3.14). Однако там не была рассмотрена ситуация некорректного ввода значений коэффициентов. Например, если пользователь введет $a=0$, тот уравнение из квадратного превратится в линейное. Алгоритм

52 Раздел описаний в процедуре может отсутствовать, если в нем нет необходимости.

решения линейного уравнения тривиален: $x=-c/b$, при условии, что $b\neq0$. Чтобы не усложнять уже составленный алгоритм решения квадратного уравнения, запишем его в виде *подпрограммы-процедуры*. Далее приведен фрагмент программы с комментариями:

```
//Процедура для вычисления действительных
//корней квадратного уравнения.
procedure korni(a,b,c:real;var x1,x2:real;
                                  var pr:boolean);
 //Входные параметры процедуры
 //(параметры-значения):
 //a,b,c – коэффициенты квадратного уравнения;
 //Выходные параметры процедуры
 //(параметры-переменные):
 //x1,x2 – корни квадратного уравнения;
 //pr – логическая переменная,
 //принимает значение ложь,
//если в уравнении нет корней
 //и значение истина в противном случае.
 var
 d:real;
 begin
     d:=b*b-4*a*c;
     if d<0 then
         pr:=false
     else
     begin
         pr:=true;
         x1:=(-b+sqrt(d))/2/a;
         x2:=(-b-sqrt(d))/(2*a);
     end
 end;    //Конец подпрограммы
//Основная программа
var a_,b_,c_,x1_,x2_,x_:real; pr_:boolean;
begin
 write('a_:='); readln(a_);
 write('b_:='); readln(b_);
 write('c_:='); readln(c_);
 if a_=0 then   //Если a_=0, то уравнение
```

```
                    //квадратным не является.
 begin
     //Решение линейного уравнения bx+c=0.
     if b_<>0 then
     begin
         x_:=-c_/b_;
         writeln('x=',x_);
     end
     else
         writeln('Нет корней');
 end
 else       //Решение уравнения ax^2+bx+c=0.
 Begin
     //Вызов процедуры.
     korni(a_,b_,c_,x1_,x2_,pr_);
     if pr_=false then
         writeln('Нет корней')
     else
         writeln('x1=',x1_,' x2=',x2_);
 end;
end.
```

ЗАДАЧА 4.2. Вводится последовательность из N целых положительных чисел. В каждом числе найти наибольшую и наименьшую цифры.

Для решения задачи создадим процедуру max_min, результатом работы которой будут два значения: минимальная и максимальная цифры в заданном числе.

Текст программы:

```
//Процедура возвращает max наибольшую
//и min наименьшую цифры в числе M.
//В списке параметров:
//M параметр-значение (входной параметр),
//max и min параметры-переменные
//(выходные параметры).
procedure max_min(M:longint; var max:byte;
                                 var min:byte);
var  i: byte;
begin
```

```
 i:=1;
 while M div 10>0 do
 begin
     if i=1 then
     begin
//Предположим, что первая цифра является
         max:=M mod 10; //наибольшей или
         min:=M mod 10; //наименьшей.
         i:=i+1;
     end;
     //Поиск цифры больше max или меньше  min
     if M mod 10 > max then max:=M mod 10;
     if M mod 10 < min then min:=M mod 10;
     M:=M div 10;
 end;
end;
var
 X:longint; N,i,X_max, X_min:byte;
begin
 //Количество элементов в последовательности
 write('N='); readln(N);
 for i:=1 to N do
 begin
     //Элемент последовательности.
     write('X=');
     readln(X);
     if X>0 then //Если элемент положительный,
                 //то
     begin
         //вызов процедуры.
         max_min(X,X_max,X_min);
         //Печать результатов.
         writeln(' max=',X_max,' min=',X_min);
     end;
 end;
end.
```

4.4 Функции

Описание функции также состоит из заголовка и тела:

```
function имя_функции(формальные_параметры):тип;
 label
    описание_меток;
 const
    описание_констант;
 type
    описание_типов;
 var
    описание_переменных;
 begin
    //Тело функции.
 end;
```

Заголовок функции содержит: служебное слово `function`, любой допустимый в языке Free Pascal идентификатор - `имя_функции`; имена формальных параметров и их типы, разделенные точкой с запятой - `формальные_параметры`, тип возвращаемого функцией значения - `тип` (функции могут возвращать скалярные значения целого, вещественного, логического, символьного или ссылочного типа).

Примеры описания функций:

```
function fun_1 (x:real):real;
function fun_2(a, b:integer):real;
```

Тело функции состоит из раздела описаний[53] (константы, типы, переменные, процедуры и функции, используемые в процедуре) и операторов языка, реализующих ее алгоритм. *В теле функции всегда должен быть хотя бы один оператор, присваивающий значение имени функции*.

Например:

```
function fun_2(a, b:integer):real;
 begin
    fun_2:=(a+b)/2;
 end;
```

Обращение к функции осуществляется по имени с указанием списка фактических параметров, разделенных запятой:

```
имя_функции (список_фактических_параметров);
```

53 Раздел описаний в функции может отсутствовать, если в нем нет необходимости.

Например:

```
y:=fun_1(1.28);
z:=fun_1(1.28)/2+fun_2(3,8);
```

ЗАДАЧА 4.3. Вводится последовательность из N целых чисел, найти среднее арифметическое совершенных чисел и среднее геометрическое простых чисел последовательности.

Напомним, что целое число называется простым, если оно делится нацело только на само себя и единицу. Подробно алгоритм определения простого числа описан в задаче 3.14 (рис. 3.33). Кроме простых чисел, в этой задаче фигурируют совершенные числа. Число называется совершенным, если сумма всех делителей, меньших его самого, равна этому числу. Алгоритм, с помощью которого можно определить делители числа, подробно рассмотрен в задаче 3.13 (рис. 3.32).

Для решения поставленной задачи понадобятся две функции:

•`prostoe` – определяет, является ли число простым, аргумент функции целое число `N`, функция возвращает `true` (истина), если число простое и `false` (ложь) – в противном случае;

•`soversh` – определяет, является ли число совершенным; входной параметр целое число `N`, функция возвращает `true` (истина), если число простое и `false` (ложь) – в противном случае.

Фрагмент программы с комментариями:

```
//Функция, которая определяет простое число.
function prostoe(N:word):boolean;
 var i:word;
 begin
     prostoe:=true;
     for i:=2 to N div 2 do
         if N mod i = 0 then
         begin
             prostoe:=false;
             break;
         end;
 end;
//Функция, которая определяет
//совершенное число.
function soversh(N:word):boolean;
 var i:word; S:word;
```

```
 begin
     soversh:=false;
     S:=0;
     for i:=1 to N div 2 do
         if N mod i =0 then S:=S+i;
         if S=N then soversh:=true;
 end;
var X:word; K,kol_p,kol_s,i:byte; Sum,Pro:real;
begin   //Начало основной программы.
 //Ввод количества элементов.
 write('K='); readln(K);
 Sum:=0; //Переменная для накапливания суммы.
 Pro:=1; //Переменная для вычисления произвед.
 kol_p:=0;      //Счетчик простых чисел.
 kol_s:=0;      //Счетчик совершенных чисел.
 for i:=1 to K do
 begin
     //Ввод элемента последовательности.
     Writeln('X='); readln(X);
     //Если число простое,
     if prostoe(X) then
     begin
         //выполнить операцию умножения,
         Pro:=Pro*X;
         //увеличить счетчик простых чисел.
         kol_p:=kol_p+1;
     end;
     //Если число совершенное,
     if soversh(X) then
     begin
         //выполнить операцию умножения,
         Sum:=Sum+X;
         //увеличить счетчик совершенных чисел.
         kol_s:=kol_s+1;
     end;
 end;
 //Если были найдены совершенные числа,
 if kol_s<> 0 then
```

```
begin
    //вычислить среднее арифметическое.
    Sum:=Sum/kol_s;
    writeln('Среднее арифметическое
                совершенных чисел ', Sum:5:2);
end
else   //иначе вывести сообщение:
    writeln('Совершенных чисел
                    в последовательности нет.');
//Если были найдены простые числа,
if kol_p<>0 then
begin
    //вычислить среднее геометрическое.
    Pro:= exp(1/kol_p*ln(Pro));
    writeln('Среднее геометрическое
                    простых чисел ',Pro:5:2);
end
else//иначе вывести сообщение:
    writeln('Простых чисел
                    в последовательности нет');
end.
```

Задача 4.4. Вводится последовательность целых чисел. 0 — конец последовательности. Определить, содержит ли последовательность хотя бы одно число-палиндром.

Палиндром — это число, симметричное относительно своей середины. Например, 123454321, 678876 — палиндромы. Чтобы определить, является ли число палиндромом нужно сравнивать первую и последнюю цифры, затем вторую и предпоследнюю и так далее. Если хотя бы в одной паре цифры не совпадут, то число палиндромом не является.

Для решения поставленной задачи понадобятся две функции:

- `cifra_kol` — определяет количество цифр в числе (подробно алгоритм описан в задаче 3.16);
- `palindrom` — возвращает значение истина, если переданное в нее число является палиндромом.

Текст программы с комментариями:

```
//Функция для вычисления количества
//цифр в числе M.
```

```
function cifra_kol(M:longint):byte;
 begin
     cifra_kol:=1;
     while M div 10 > 0 do
     begin
         cifra_kol:=cifra_kol+1;
         M:=M div 10;
     end;
 end;
//Функция возвращает значение истина,
//если число M,
//состоящее из kol цифр палиндром,
//и значение ложь в противном случае.
function palindrom(M:longint;kol:byte):boolean;
 var i:byte; j:longint;
begin
 j:=1;
//Возведение числа 10 в степень kol-1
//(разрядность числа).
 for i:=1 to kol-1 do
     j:=j*10;
 palindrom:=true;   //Пусть число - палиндром.
 for i:=1 to kol div 2 do
 begin
//Выделение старшего разряда M div j
//(первая цифра).
//Выделение младшего разряда  M mod 10
//(последняя цифра).
//Если первая и последняя цифры не совпадают,
     if M div j <> M mod 10 then
     begin
         //то число не палиндром.
         palindrom:=false;
         break;          //выход из цикла.
     end;
     //Изменение числа
         //Удаление 1-й цифры числа
     M:=M-(M div j)*j;
```

```
            //Удаление последней цифры числа.
        M:=M div 10;
            /Уменьшение разрядности.
        j:=j div 100;      /
        end;
    end;
    //Основная программа.
    var X:longint; pr:boolean;
    begin
     //Ввод элемента последовательности.
     write('X=');readln(X);
    //Пусть в последовательности нет палиндромов.
     pr:=false;
     while X<>0 do //Пока не ноль,
     begin
        if palindrom(X,cifra_kol(X)) then
        begin
            pr:=true;  //Найдено число палиндром,
            break;     //досрочный выход из цикла.
        End;
    //Ввод следующего элемента последовательности.
        write('X=');readln(X);
     end;
     if pr then writeln('Последовательность
                         содержит число-палиндром.')
     else writeln('Последовательность не содержит
палиндромов.');
    end.
```

4.5 Решение задач с использованием подпрограмм

В этом разделе мы рассмотрим задачи с несложными алгоритмами, но больше внимания уделим их интерфейсу в среде Lazarus.

ЗАДАЧА 4.5. Создать программу, которая автоматизирует процесс перевода градусной меры угла в радианную и наоборот, в зависимости от выбора пользователя. То есть пользователь должен выбрать, как он будет вводить угол, в радианах или в градусах. Введет в радианах, ответ получит в градусах и, соответственно, введет в градусах, ответ получит в радианах.

С точки зрения математика задача не вызывает сложности:

• чтобы найти радианную меру какого-нибудь угла по данной градусной его мере, нужно умножить число градусов на $\frac{\pi}{180}$, число минут - на $\frac{\pi}{180 \cdot 60}$, число секунд - на $\frac{\pi}{180 \cdot 60 \cdot 60}$ и найденные произведения сложить;

• чтобы найти градусную меру угла по заданной радианной нужно умножить число радиан на $\frac{180}{\pi}$; если из полученной дроби выделить целую часть, то получим градусы; если из числа полученного умножением оставшейся дробной части на 60, выделить целую часть получим минуты; секунды вычисляются аналогично из дробной части минут.

Для перевода угла из градусной меры в радианную создадим функцию

```
function gradus_radian(gradus,minuta,secunda:byte):real;
```

в которую будем передавать целочисленные значения градусов, минут и секунд. Результат работы функции – вещественное число, величина угла в радианах.

Задачу перевода из радианной меры в градусную решим, создав процедуру

```
procedure radian_gradus(radian:real;
                        var gradus,minuta,secunda:byte);
```

у которой один входной параметр – радианная мера угла и три выходных – градусы, минуты и секунды.

Разработаем интерфейс будущей программы в среде Lazarus. Создадим новый проект, установим свойства формы так, как показано в табл. 4.1, и разместим на ней компоненты в соответствии с рис. 4.2.

Таблица 4.1. Свойства формы

Свойство	Значение	Описание свойства
Caption	Величина угла	Заголовок формы
Height	265	Высота формы
Width	325	Ширина формы
BorderIcons.BiMaximize	false	Кнопка развертывания окна недоступна

Свойство	Значение	Описание свойства
BorderStyle	bdDialog	Стиль рамки – диалоговое окно, не изменяет размеры
Position	poScreenCenter	Окно появится в центре экрана

С компонентами Edit, Label и Button мы уже хорошо знакомы.

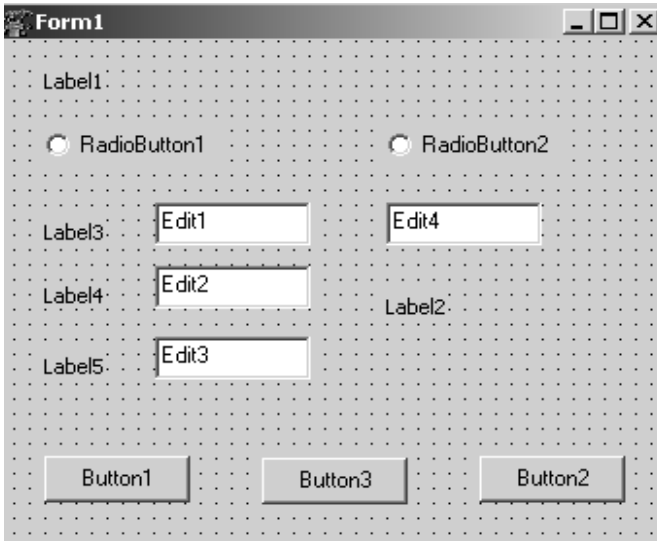

Рисунок 4.1: Настройка формы к задаче 4.5

Компонент RadioButton – это *переключатель*. Его используют для выбора одного из нескольких взаимоисключающих решений. Обычно на форму помещается, по меньшей мере, два таких компонента. Они могут иметь только два состояния, определяемых свойством Checked. Если у одного из компонентов это свойство истинно true, то во всех остальных ложно false. В данной задаче используется два компонента RadioButton1 и RadioButton2, предоставляя пользователю выбор: включен первый компонент – будет осуществлен перевод из градусной меры в радианную, включен второй – наоборот. Двойной щелчок по компоненту RadioButton1 приведет к созданию процедуры TForm1.RadioButton1Click обработки события «Щелчок мыши по кнопке переключателя». В тексте процедуры следует указать команды, которые будут выполняться, если пользователь включил или выключил компонент.

Нам уже известно, что свойства компонентов могут изменяться как в окне конструирования формы, так и непосредственно в программе. Если дважды щелкнуть по форме, вне размещенных на ней компонентов, то будет создана процедура TForm1.FormCreate обработки события открытия формы. На вкладке События инспектора

объектов это событие носит название `OnCreate`. В процедуре `TForm1.FormCreate` можно задать свойства всех компонентов на момент открытия формы.

Кнопки, расположенные на форме, несут следующую функциональную нагрузку:

•`Button1` запускает процесс перевода в зависимости от установок переключателей;

•`Button3` возвращает внешний вид формы в первоначальное состояние (до ввода и вывода данных);

•`Button2` – завершает процесс выполнения программы.

Текст программы с необходимыми комментариями приведен ниже. Результаты работы программы представлены на рис. 4.4 и рис. 4.5.

```
   unit Unit1;
   interface
   uses
   Windows, Messages, SysUtils, Variants, Classes,
Graphics, Controls, Forms, Dialogs, StdCtrls;
   type
   TForm1 = class(TForm)
   Button1: TButton;
   Label1: TLabel;
   RadioButton1: TRadioButton;
   RadioButton2: TRadioButton;
   Edit1: TEdit;
   Button2: TButton;
   Label2: TLabel;
   Edit2: TEdit;
   Edit3: TEdit;
   Label3: TLabel;
   Label4: TLabel;
   Label5: TLabel;
   Edit4: TEdit;
   Button3: TButton;
   procedure Button1Click(Sender: TObject);
   procedure RadioButton1Click(Sender: TObject);
   procedure RadioButton2Click(Sender: TObject);
   procedure FormCreate(Sender: TObject);
```

```
procedure Button2Click(Sender: TObject);
procedure Button3Click(Sender: TObject);
private
{ Private declarations }
public
{ Public declarations }
end;
var
Form1: TForm1;
implementation
{$R *.dfm}

//Щелчок по кнопке ВЫЧИСЛИТЬ.
procedure TForm1.Button1Click(Sender: TObject);
//Функция перевода данных из градусов
//в радианы.
function gradus_radian
            (gradus,minuta,secunda:byte):real;
 begin
    gradus_radian:=
           gradus*pi/180+minuta*pi/180/60+
                     secunda*pi/180/60/60;
 end;

 //Процедура перевода из радиан в градусы.
 procedure radian_gradus(radian:real;
            var gradus,minuta,secunda:byte);
 begin
    gradus:=trunc(radian*180/pi);
    minuta:= trunc((radian*180/pi-gradus)*60);
    secunda:=trunc(((radian*180/pi-
                  gradus)*60-minuta)*60);
 end;

var                      //Описание переменных.
grad,min,sec:byte;       //Градусная мера угла.
rad:real;                //Радианная мера угла.
```

```
//Контроль ввода.
kod_g,kod_m,kod_s,kod_r:integer;
begin
//Если первый переключатель вкл.
if RadioButton1.Checked then
begin
 Val(Edit1.Text,grad,kod_g);   //Ввод градусов.
 Val(Edit2.Text,min,kod_m);    //Ввод минут.
 Val(Edit3.Text,sec,kod_s);    //Ввод секунд.
 //Если ошибки при вводе не было, то
 if (kod_g=0) and (kod_m=0) and (kod_s=0) then
 begin
 //сделать видимым компонент Label2
 Label2.Visible:=true;
 //и вывести туда результат вычислений.
 //Вызов функции gradus_radian
 //перевода из градусов в радианы.
 Label2.Caption:='Величина угла ' +chr(13)+
 FloatToStrF(gradus_radian(grad,min,sec),
 ffFixed,8,6)+' радиан';
 end
 else
//Иначе выдать сообщение об ошибке при вводе.
 MessageDlg('Ошибка при вводе данных!',
                        MtWarning,[mbOk],0);
end;
//Если второй переключатель вкл.
if RadioButton2.Checked  then
begin
 //Ввод радианной меры угла.
 Val(Edit4.Text,rad,kod_r);
 //Если нет ошибок при вводе, то
 if (kod_r=0)then
 begin
 //сделать видимым компонент Label2
 Label2.Visible:=true;
//и вывести туда результат вычислений.
//Вызов процедуры перевода из радиан в градусы.
```

```
 radian_gradus(rad,grad,min,sec);
 Label2.Caption:='Величина угла' +chr(13)
            +IntToStr(grad)+' г '+IntToStr(min)
                  +' м '+IntToStr(sec)+' с';
 end
 else
//Иначе выдать сообщение об ошибке при вводе.
 MessageDlg('Ошибка при вводе данных!',
                             MtWarning,[mbOk],0);
end;
end;
//Щелчок по кнопке ВЫХОД.
procedure TForm1.Button2Click(Sender: TObject);
begin
 close;
end;
//Щелчок по кнопке ОЧИСТИТЬ.
procedure TForm1.Button3Click(Sender: TObject);
begin
 //Установка свойств компонентов
//в первоначальное состояние.
 Edit1.Text:='00';
 Edit2.Text:='00';
 Edit3.Text:='00';
 Edit4.Text:='00.000';
 Label1.Caption:='Введите значение';
 Label1.Font.Size:=10;
 Label3.Caption:='Градусы';
 Label4.Caption:='Минуты';
 Label5.Caption:='Секунды';
 Button1.Caption:='ВЫЧИСЛИТЬ';
 Button2.Caption:='ВЫХОД';
 Button3.Caption:='ОЧИСТИТЬ';
 Edit4.Enabled:=false;
 Label2.Visible:=false;
 RadioButton1.Checked:=true;
 RadioButton2.Checked:=false;
end;
```

```
//Обработка события открытие формы.
procedure TForm1.FormCreate(Sender: TObject);
begin
 //Установка свойств компонентов.
 Edit1.Text:='00';
 Edit2.Text:='00';
 Edit3.Text:='00';
 Edit4.Text:='00.000';
 Label1.Caption:='Введите значение';
 Label1.Font.Size:=10;
 Label3.Caption:='Градусы';
 Label4.Caption:='Минуты';
 Label5.Caption:='Секунды';
 Button1.Caption:='ВЫЧИСЛИТЬ';
 Button2.Caption:='ВЫХОД';
 Button3.Caption:='ОЧИСТИТЬ';
 Edit4.Enabled:=false;
 Label2.Visible:=false;
 RadioButton1.Checked:=true;
 RadioButton2.Checked:=false;
end;
//Обработка события щелчок
//по переключателю RadioButton1.
procedure Tform1.RadioButton1Click(
                              Sender: TObject);
begin
 if RadioButton1.Checked then
 begin
     Edit1.Enabled:=true;
     Edit2.Enabled:=true;
     Edit3.Enabled:=true;
     Label5.Enabled:=true;
     Label3.Enabled:=true;
     Label4.Enabled:=true;
     Edit4.Enabled:=false;
 end;
end;
```

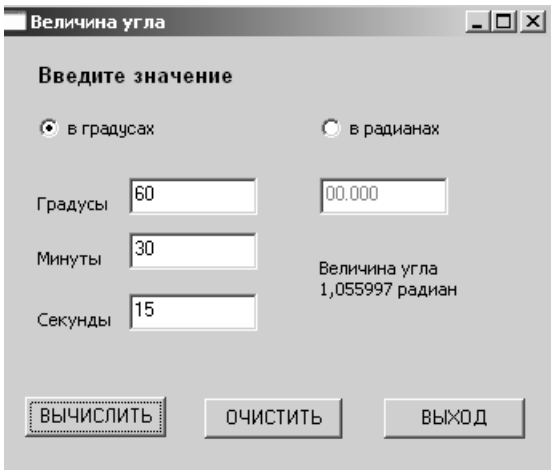

Рисунок 4.2: Перевод значений из градусной меры в радианную

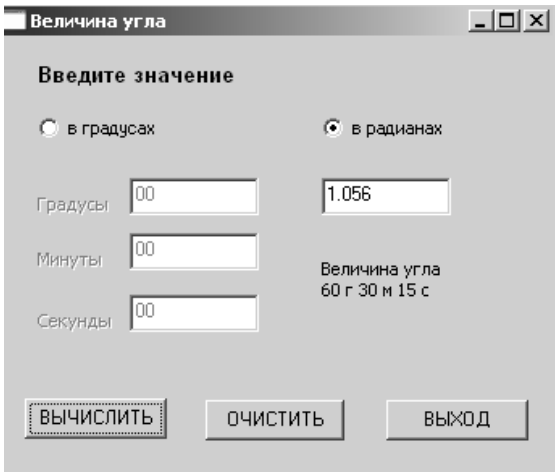

Рисунок 4.3: Перевод значений из радианной меры в градусную

```
//Обработка события щелчок
//по переключателю RadioButton2.
procedure Tform1.RadioButton2Click(
                              Sender: TObject);
 begin
 if RadioButton2.Checked then
 begin
     Edit4.Enabled:=true;
     Button1.Enabled:=true;
     Edit1.Enabled:=false;
     Edit2.Enabled:=false;
     Edit3.Enabled:=false;
     Label3.Enabled:=false;
     Label4.Enabled:=false;
     Label5.Enabled:=false;
 end;
end;
end.
```

ЗАДАЧА 4.6. Создать программу для решения уравнений:
•линейное $ax+b=0$;
•квадратное $ax^2+bx+c=0$;
•кубическое $ax^3+bx^2+cx+d=0$.

Решение линейного уравнения тривиально $x=-b/a$, алгоритмы решения квадратного и кубического уравнений подробно рассмотрены в задачах 3.4 и 3.5.

Создадим новый проект (рис. 4.4). Свойства формы настроим по табл. 4.1, за исключением свойства `Caption`, которому присвоим значение **Решение уравнения**.

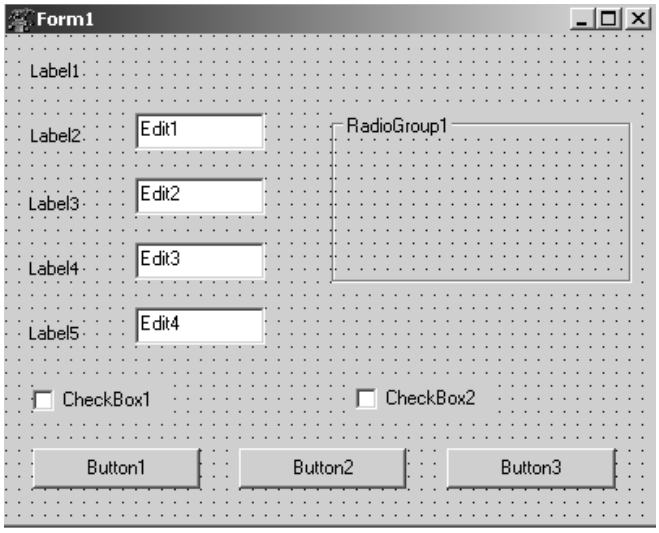

Рисунок 4.4: Процесс создания формы к задаче 4.6

Обратите внимание, что на форме появились не знакомые нам компоненты. Это `CheckBox` – флажок и `RadioGroup` – группа переключателей.

Компонент *флажок* `CheckBox` используется для того, чтобы пользователь мог включить или выключить значение какого-либо параметра. Установлен флажок или нет, определяет свойство `Checked` (`true`, `false`). В составе диалогового окна может быть несколько таких компонентов, причем состояние любого из них не зависит от состояния остальных.

Компонент *группа переключателей* `RadioGroup` объединяет в себе несколько переключателей. Каждый размещенный в нем переключатель помещается в специальный список `Items` и доступен по номеру, установленному в свойстве `ItemIndex`. После размещения на форме компонент пуст. Чтобы создать в нем хотя бы один переключатель, нужно выделить его, обратиться к инспектору объектов и выбрать свойство `Items` – *редактор списка*. Строки, набранные в редакторе, используются как поясняющие надписи справа от переключателей, а их количество определяет количество переключателей в группе. В нашем случае окно редактора списка будет иметь вид, как на рис. 4.5.

После создания компонента группа переключателей его свойство номер переключателя `ItemIndex` по умолчанию равно –1. Это означает, что ни один компонент в группе не установлен.

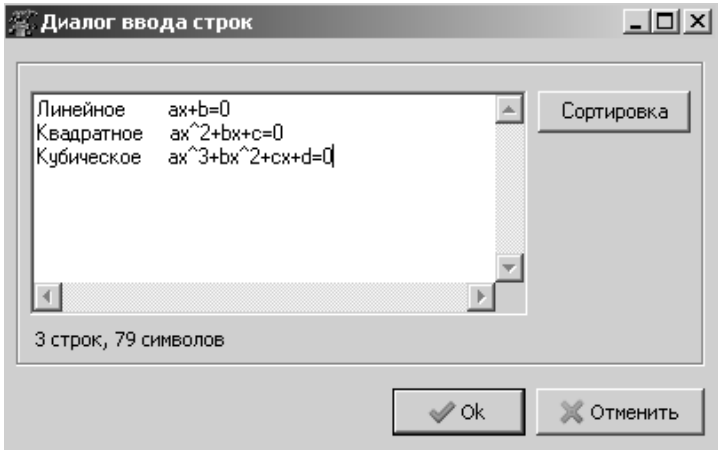

Рисунок 4.5: Окно редактора списка

Чтобы в момент появления компонента на экране один из переключателей был отмечен, нужно либо на этапе конструирования формы, либо программно присвоить свойству `ItemIndex` номер одного из элементов списка, учитывая, что нумерация начинается с нуля.

С остальными компонентами, размещенными на форме, пользователь уже знаком. На рис. 4.6 - 4.8 видно, как работает программа. Пользователю предоставляется возможность выбрать вид решаемого уравнения, ввести его коэффициенты и указать, какие решения — действительные или комплексные (если это возможно) — он хотел бы получить. Далее приведен текст программного модуля с комментариями:

```
unit Unit1;
interface
uses
Windows, Messages, SysUtils, Variants, Classes,
Graphics, Controls, Forms, Dialogs, StdCtrls,
                                        ExtCtrls;
type
TForm1 = class(TForm)
Label1: TLabel;
Label2: TLabel;
Label3: TLabel;
Label4: TLabel;
Label5: TLabel;
Edit1: TEdit;
Edit2: TEdit;
Edit3: TEdit;
Edit4: TEdit;
CheckBox1: TCheckBox;
CheckBox2: TCheckBox;
Button1: TButton;
```

```
Button2: TButton;
RadioGroup1: TRadioGroup;
Button3: TButton;
procedure FormCreate(Sender: TObject);
procedure Button2Click(Sender: TObject);
procedure Button3Click(Sender: TObject);
procedure Button1Click(Sender: TObject);
procedure RadioGroup1Click(Sender: TObject);
private
{ Private declarations }
public
{ Public declarations }
end;
var
Form1: TForm1;
implementation
{$R *.dfm}
//Щелчок по кнопке НАЙТИ КОРНИ.
procedure TForm1.Button1Click(Sender: TObject);
 //Решение линейного уравнения.
 //Параметры процедуры korni_1:
 //a,b — вещественные переменные,
 //коэффициенты уравнения;
 //x_ - строковая переменная,
 //решение уравнения в символьном виде.
 procedure korni_1(a,b:real;var x_:string);
     //x - решение уравнения в численном виде
     var x:real;
 begin
     x:=-b/a;
     x_:=FloatToStrF(x,ffFixed,5,2);
 end;
 //Решение квадратного уравнения.
 //Параметры процедуры korni_2:
 //a,b,c — вещественные переменные,
 //коэффициенты уравнения;
 //x1_,x2_ - строковые переменные,
 //решение уравнения в символьном виде;
```

```
//pr — целочисленная переменная,
//принимает значение, равное 1, если
//уравнение имеет действительные корни,
//и значение, равное 2, если корни мнимые.
procedure korni_2(a,b,c:real;
           var x1_,x2_:string;var pr:byte);
//x1,x2 - решение уравнения в численном виде,
//d — дискриминант.
    var d,x1,x2:real;
begin
    d:=b*b-4*a*c;
    if d<0 then
    begin
        x1:=-b/(2*a);
        x2:=sqrt(abs(d))/(2*a);
        x1_:= FloatToStrF(x1,ffFixed,5,2)+
        '+i*'+FloatToStrF(x2,ffFixed,5,2);
        x2_:= FloatToStrF(x1,ffFixed,5,2)+
        '-i*'+FloatToStrF(x2,ffFixed,5,2);
        pr:=2;     //Комплексные корни.
    end
    else
    begin
        x1:=(-b+sqrt(d))/2/a;
        x2:=(-b-sqrt(d))/(2*a);
        x1_:= FloatToStrF(x1,ffFixed,5,2);
        x2_:= FloatToStrF(x2,ffFixed,5,2);
        pr:=1;     //Действительные корни.
    end
end;
//Решение кубического уравнения.
//Параметры процедуры korni_3:
//a,b,c,d — вещественные переменные,
//коэффициенты уравнения;
//x1_,x2_,x3_ - строковые переменные,
//решение уравнения в символьном виде;
//pr — целочисленная переменная,
//принимает значение, равное 1, если
```

```
//уравнение имеет действительные корни,
//и значение, равное 2, если в уравнении
//один корень действительный и два мнимых.
procedure korni_3(a,b,c,d:real;
    var x1_,x2_,x3_:string;var pr:byte);
//x1,x2,x3 - решения в численном виде.
var r,s,t,p,q,ro,fi,u,v,x1,x2,x3,h,g:real;
begin
    r:=b/a;
    s:=c/a;
    t:=d/a;p:=(3*s-r*r)/3;
    q:=2*r*r*r/27-r*s/3+t;
    d:=(p/3)*sqr(p/3)+sqr(q/2);
    if d<0 then
    begin
        ro:=sqrt(-p*p*p/27);
        fi:=-q/(2*ro);
        fi:=pi/2-arctan(fi/sqrt(1-fi*fi));
//Вычисление действительных корней уравнения.
x1:=2*exp(1/3*ln(ro))*cos(fi/3)-r/3;
x2:=2*exp(1/3*ln(ro))*cos(fi/3+2*pi/3)-r/3;
x3:=2*exp(1/3*ln(ro))*cos(fi/3+4*pi/3)-r/3;
        x1_:=FloatToStrF(x1,ffFixed,5,2);
        x2_:=FloatToStrF(x2,ffFixed,5,2);
        x3_:=FloatToStrF(x3,ffFixed,5,2);
        pr:=1;  //Действительные корни.
    end
    else
    begin
        if -q/2+sqrt(d)>0 then
            u:=exp(1/3*ln(-q/2+sqrt(d)))
        else
            if -q/2+sqrt(d)<0 then
            u:=-exp(1/3*ln(abs(-q/2+sqrt(d))))
            else u:=0;
        if -q/2-sqrt(d)>0 then
            v:=exp(1/3*ln(-q/2-sqrt(d)))
        else
```

```
            if -q/2-sqrt(d)<0 then
            v:=-exp(1/3*ln(abs(-q/2-sqrt(d))))
            else v:=0;
        //Вычисление действительного корня.
        x1:=u+v-r/3;
        //Вычисление комплексных корней.
        h:=-(u+v)/2-r/3;
        g:=(u-v)/2*sqrt(3);
        x1_:=FloatToStrF(x1,ffFixed,5,2);
        x2_:=FloatToStrF(h,ffFixed,5,2)+
        '+i*'+FloatToStrF(g,ffFixed,5,2);
        x3_:=FloatToStrF(h,ffFixed,5,2)+
        '-i*'+FloatToStrF(g,ffFixed,5,2);
        pr:=2;  //1 действительный и
                //2 комплексных корня.
    end
 end;
//Обработка события
//щелчок по кнопке НАЙТИ КОРНИ.
var
    a_,b_,c_,d_:real;
    kod_a,kod_b,kod_c,kod_d:integer;
    _x,_x1,_x2,_x3:string; _pr:byte;
begin
 case RadioGroup1.ItemIndex of
 0:  //Пользователь выбрал 1-й переключатель.
 Begin  //Решение линейного уравнения.
    //Ввод исходных данных.
    val(Edit1.Text,a_,kod_a);
    val(Edit2.Text,b_,kod_b);
    //Ввод прошел успешно.
    if (kod_a=0) and (kod_b=0) then
    begin
        //1-й коэффициент не ноль.
        if a_<>0 then
        begin
            //Решение линейного уравнения.
            korni_1(a_,b_,_x);
```

```
            //Вывод найденного значения.
MessageDlg('Решение линейного уравнения x='+
                    _x,mtInformation,[mbOk],0);
        end
        //1-й коэффициент равен нулю,
        else
            //вывод соответствующего сообщения.
            MessageDlg('Нет корней!',
                    mtInformation,[mbOk],0);
    end
    else //Некорректный ввод данных.
        MessageDlg('Ошибка при вводе!',
                        mtInformation,[mbOk],0);
end;
1:  //Пользователь выбрал 2-й переключатель.
begin
    //Ввод исходных данных.
    val(Edit1.Text,a_,kod_a);
    val(Edit2.Text,b_,kod_b);
    val(Edit3.Text,c_,kod_c);
    //Ввод прошел успешно.
    if (kod_a=0) and (kod_b=0) and (kod_c=0)
    then begin
        //Первый коэффициент не ноль.
        if a_<>0 then
        begin
        //Решение квадратного уравнения.
            korni_2(a_,b_,c_,_x1,_x2,_pr);
//В переменной _pr содержится информация
//о типе корней:
//1 - действительные, 2 - комплексные.
//Оба флажка не установлены.
if (CheckBox1.Checked=false) and
         (CheckBox2.Checked=false) then
    MessageDlg('Выберите тип решения',
                mtInformation,[mbOk],0);
//Оба флажка установлены.
if CheckBox1.Checked and
```

```
                    CheckBox2.Checked then
    MessageDlg('Решение уравнения'+
                        chr(13)+' X1='+_x1+
    ' X2='+_x2,mtInformation,[mbOk],0);
//Установлен первый флажок,
//корни действительные.
if CheckBox1.Checked and
    (CheckBox2.Checked=false) and (_pr=1)
    then
        MessageDlg('Действительные корни'+
            chr(13)+' X1='+_x1+' X2='+_x2,
                mtInformation,[mbOk],0);
//Установлен второй флажок,
//корни действительные.
if (CheckBox1.Checked=false) and
    CheckBox2.Checked and (_pr=1) then
    MessageDlg('Уравнение имеет только
    действительные корни.',
                mtInformation,[mbOk],0);
//Установлен первый флажок, корни комплексные.
if CheckBox1.Checked and
    (CheckBox2.Checked=false) and (_pr=2) then
    MessageDlg('Действительных корней нет',
                        mtInformation,[mbOk],0);
//Установлен второй флажок, корни комплексные.
if (CheckBox1.Checked=false) and
    CheckBox2.Checked and (_pr=2) then
    MessageDlg('Комплексные корни уравнения'+
                chr(13)+' X1='+_x1+' X2='+_x2,
                    mtInformation,[mbOk],0);
end
else   //Первый коэффициент равен нулю.
MessageDlg('Первый коэффициент не равен 0!',
                        mtInformation,[mbOk],0);
end                 //Некорректный ввод данных.
else

MessageDlg('Ошибка при вводе коэффициентов!',
```

```
                        mtInformation,[mbOk],0);
end;
2:  //Пользователь выбрал 3-й переключатель.
Begin
    //Ввод исходных данных.
    val(Edit1.Text,a_,kod_a);
    val(Edit2.Text,b_,kod_b);
    val(Edit3.Text,c_,kod_c);
    val(Edit4.Text,d_,kod_d);
    //Ввод прошел успешно.
    if (kod_a=0) and (kod_b=0) and (kod_c=0)
    and (kod_d=0)then
    begin
    //Первый коэффициент не ноль.
    if a_<>0 then
        begin
//Решение кубического уравнения.
//В переменной _pr содержится информация
//о типе корней:
//1 - действительные,
//2 - один действительный и два комплексных.
korni_3(a_,b_,c_,d_,_x1,_x2,_x3,_pr);
//Оба флажка не установлены.
if (CheckBox1.Checked=false) and
(CheckBox2.Checked=false) then
MessageDlg('Выберите тип решения',
                  mtInformation,[mbOk],0);
//Оба флажка установлены.
if CheckBox1.Checked
    and CheckBox2.Checked then
    MessageDlg('Корни кубического уравнения'+
chr(13)+' X1='+_x1+    ' X2='+_x2+' X3='+_x3,
                  mtInformation,[mbOk],0);
    //Установлен первый флажок,
    //корни действительные.
    if CheckBox1.Checked and
    (CheckBox2.Checked=false) and (_pr=1) then
MessageDlg('Действительные корни уравнения'+
```

```
      chr(13)+' X1='+_x1+' X2='+_x2+' X3='+_x3,
                    mtInformation,[mbOk],0);
  //Установлен первый флажок, корни комплексные.
      if CheckBox1.Checked and
      (CheckBox2.Checked=false) and (_pr=2) then
  MessageDlg('Действительные корни уравнения'+
  chr(13)+' X1='+_x1,mtInformation,[mbOk],0);
  //Установлен второй флажок, корни комплексные.
      if (CheckBox1.Checked=false) and
          CheckBox2.Checked and (_pr=2) then
      MessageDlg('Комплексные корни уравнения'+
                 chr(13)+' X1='+_x2+' X2='+_x3,
                    mtInformation,[mbOk],0);
      //Установлен второй флажок,
      //корни действительные.
      if (CheckBox1.Checked=false) and
      CheckBox2.Checked and (_pr=1) then
      MessageDlg('Уравнение имеет только
                    действительные корни.',
                       mtInformation,[mbOk],0);
      end
      else
  MessageDlg('Первый коэффициент не равен 0!',
                       mtInformation,[mbOk],0);
      end
      else           //Некорректный ввод данных.
      MessageDlg('Ошибка при вводе !',
                       mtInformation,[mbOk],0);
  end;
  end;
  end;
 //Щелчок по кнопке ОЧИСТИТЬ.
 procedure TForm1.Button2Click(Sender: TObject);
 begin
  Label1.Caption:='Введите коэффициенты';
  Label2.Caption:='a=';
  Label3.Caption:='b=';
  Label4.Caption:='c=';
```

```
 Label5.Caption:='d=';
 Edit1.Text:='0.00';
 Edit2.Text:='0.00';
 Edit3.Text:='0.00';
 Edit4.Text:='0.00';
 Button1.Caption:='НАЙТИ КОРНИ';
 Button2.Caption:='ОЧИСТИТЬ';
 Button3.Caption:='ВЫХОД';
 CheckBox1.Caption:='Действительные корни';
 CheckBox2.Caption:='Комплексные корни';
 CheckBox1.Checked:=true;
 Label4.Enabled:=false;
 Label5.Enabled:=false;
 Edit3.Enabled:=false;
 Edit4.Enabled:=false;
 CheckBox2.Enabled:=false;
 RadioGroup1.ItemIndex:=0;
end;
//Щелчок по кнопке ВЫХОД.
procedure TForm1.Button3Click(Sender: TObject);
begin
 close;
end;
//Событие открытие формы.
procedure TForm1.FormCreate(Sender: TObject);
begin
 Label1.Caption:='Введите коэффициенты';
 Label2.Caption:='a=';
 Label3.Caption:='b=';
 Label4.Caption:='c=';
 Label5.Caption:='d=';
 Edit1.Text:='0.00';
 Edit2.Text:='0.00';
 Edit3.Text:='0.00';
 Edit4.Text:='0.00';
 Button1.Caption:='НАЙТИ КОРНИ';
 Button2.Caption:='ОЧИСТИТЬ';
 Button3.Caption:='ВЫХОД';
```

```
 CheckBox1.Caption:='Действительные корни';
 CheckBox2.Caption:='Комплексные корни';
 CheckBox1.Checked:=true;
 Label4.Enabled:=false;
 Label5.Enabled:=false;
 Edit3.Enabled:=false;
 Edit4.Enabled:=false;
 CheckBox2.Enabled:=false;
 RadioGroup1.ItemIndex:=0;
end;
//Выбор переключателя из группы.
procedure Tform1.RadioGroup1Click(
                              Sender: TObject);
begin
 case RadioGroup1.ItemIndex of
 0:          //Выбран первый из списка.
 begin
     Label2.Enabled:=true;
     Label3.Enabled:=true;
     Edit1.Enabled:=true;
     Edit2.Enabled:=true;
     Label2.Caption:='a=';
     Label3.Caption:='b=';
     Edit1.Text:='0.00';
     Edit2.Text:='0.00';
     Label4.Enabled:=false;
     Label5.Enabled:=false;
     Edit3.Enabled:=false;
     Edit4.Enabled:=false;
     CheckBox2.Enabled:=false;
 end;
 1:          //Выбран второй из списка.
 begin
     Label2.Enabled:=true;
     Label3.Enabled:=true;
     Label4.Enabled:=true;
     Edit1.Enabled:=true;
     Edit2.Enabled:=true;
```

```
        Edit3.Enabled:=true;
        Label2.Caption:='a=';
        Label3.Caption:='b=';
        Label4.Caption:='c=';
        Edit1.Text:='0.00';
        Edit2.Text:='0.00';
        Edit3.Text:='0.00';
        Label5.Enabled:=false;
        Edit4.Enabled:=false;
        CheckBox2.Enabled:=true;
    end;
    2:          //Выбран третий из списка.
    begin
        Label2.Enabled:=true;
        Label3.Enabled:=true;
        Label4.Enabled:=true;
        Label5.Enabled:=true;
        Edit1.Enabled:=true;
        Edit2.Enabled:=true;
        Edit3.Enabled:=true;
        Edit4.Enabled:=true;
        Label2.Caption:='a=';
        Label3.Caption:='b=';
        Label4.Caption:='c=';
        Label4.Caption:='d=';
        Edit1.Text:='0.00';
        Edit2.Text:='0.00';
        Edit3.Text:='0.00';
        Edit4.Text:='0.00';
        CheckBox2.Enabled:=true;
    end;
end;
end;end.
```

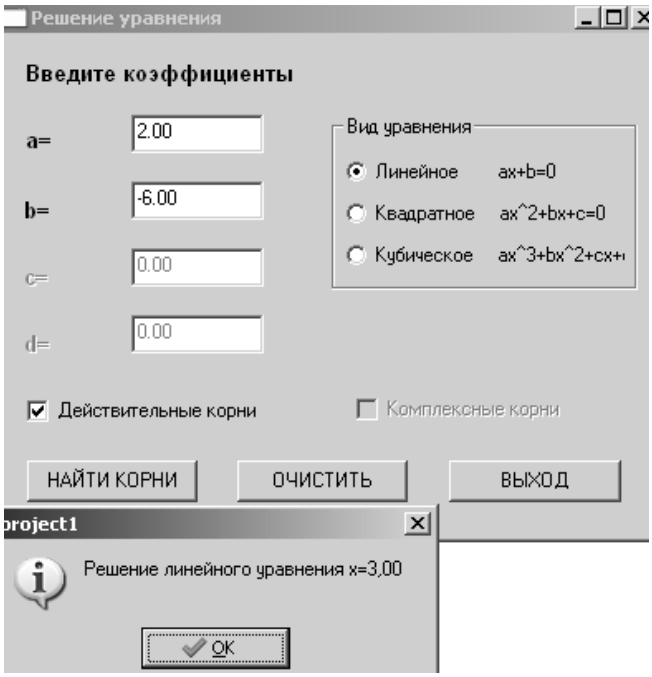

Рисунок 4.6: Решение линейного уравнения

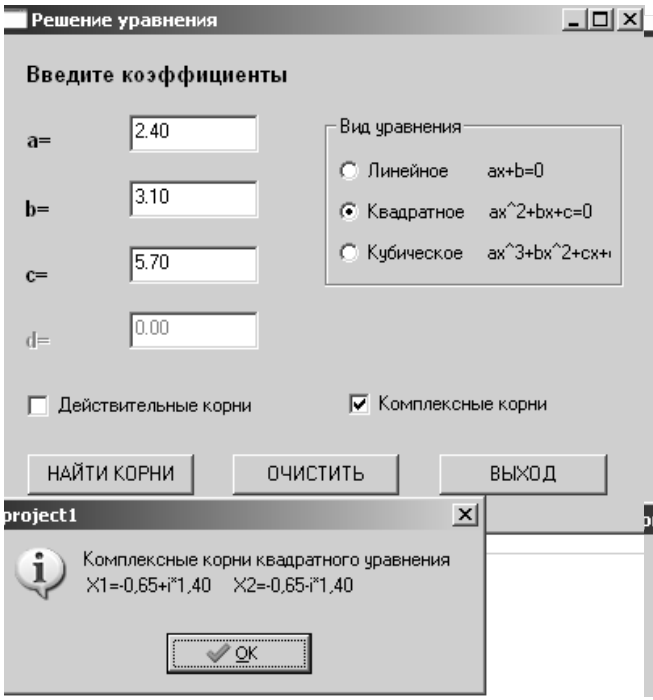

Рисунок 4.7: Решение квадратного уравнения

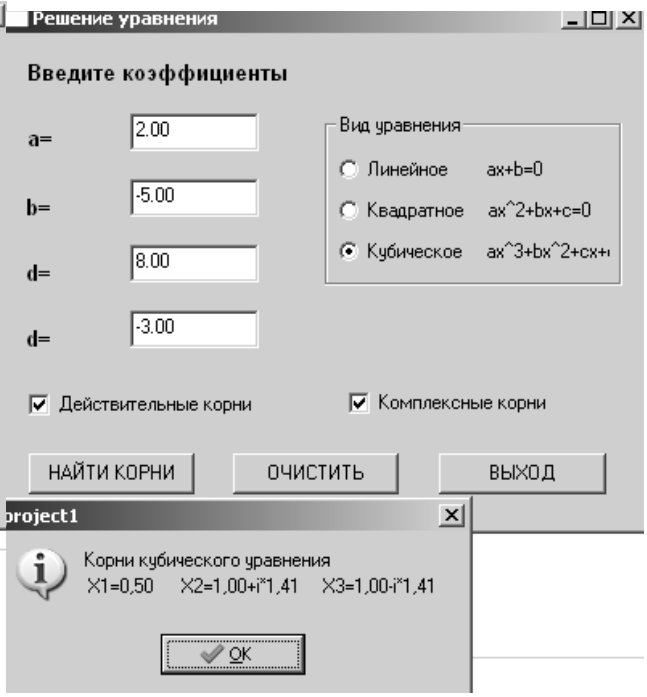

Рисунок 4.8: Решение кубического уравнения

4.6 Рекурсивные функции

Под *рекурсией* в программировании понимают подпрограмму, которая вызывает сама себя. Рекурсивные функции чаще всего используют для компактной реализации рекурсивных алгоритмов. Классиче-

скими рекурсивными алгоритмами могут быть возведение числа в целую положительную степень, вычисление факториала. С другой стороны, любой рекурсивный алгоритм можно реализовать без применения рекурсий. Достоинством рекурсии является компактная запись, а недостатком расход памяти на повторные вызовы функций и передачу параметров, кроме того, существует опасность переполнения памяти.

В рекурсивной функции необходимо обязательно предусмотреть завершение рекурсивного вызова. Иначе функция никогда не завершит свою работу.

Рассмотрим применение рекурсии на примерах.

ЗАДАЧА 4.7. Вычислить факториал числа *n*.

Вычисление факториала подробно рассмотрено в задаче 3.10 (рис. 3.29). Для решения этой задачи с применением рекурсии создадим функцию `factorial`, алгоритм которой представлен на рис. 4.9.

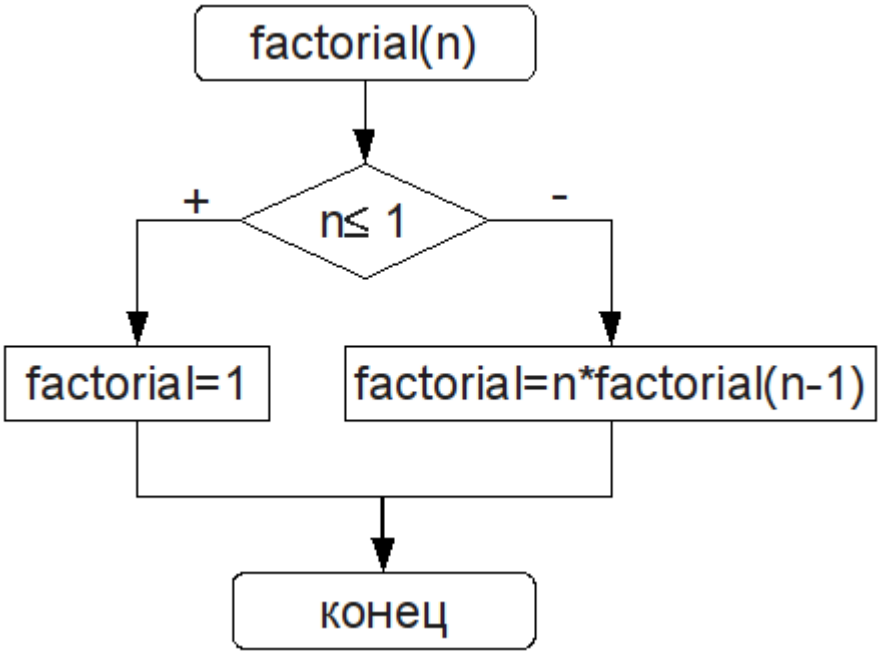

Рисунок 4.9: Алгоритм вычисления факториала

Текст подпрограммы с применением рекурсии:

```
function factorial(n:word):longint;
begin
 if n<=1 then   factorial:=1
 else   factorial:=n*factorial(n-1)
end;
var i:integer;
begin
```

```
 write('i=');
 read(i);
 write(i,'!=',factorial(i));
end.
```

ЗАДАЧА 4.8. Вычислить *n*-ю степень числа *a* (*n* – целое число)

Результатом возведения числа *a* в целую степень *n* является умножение этого числа на себя *n* раз. Но это утверждение верно только для положительных значений *n*. Если *n* принимает отрицательные значения, то $a^{-n}=\frac{1}{a^n}$. В случае если *n*=0, то $a^0=1$.

Для решения задачи создадим рекурсивную функцию `stepen`, алгоритм которой представлен на рис. 4.10.

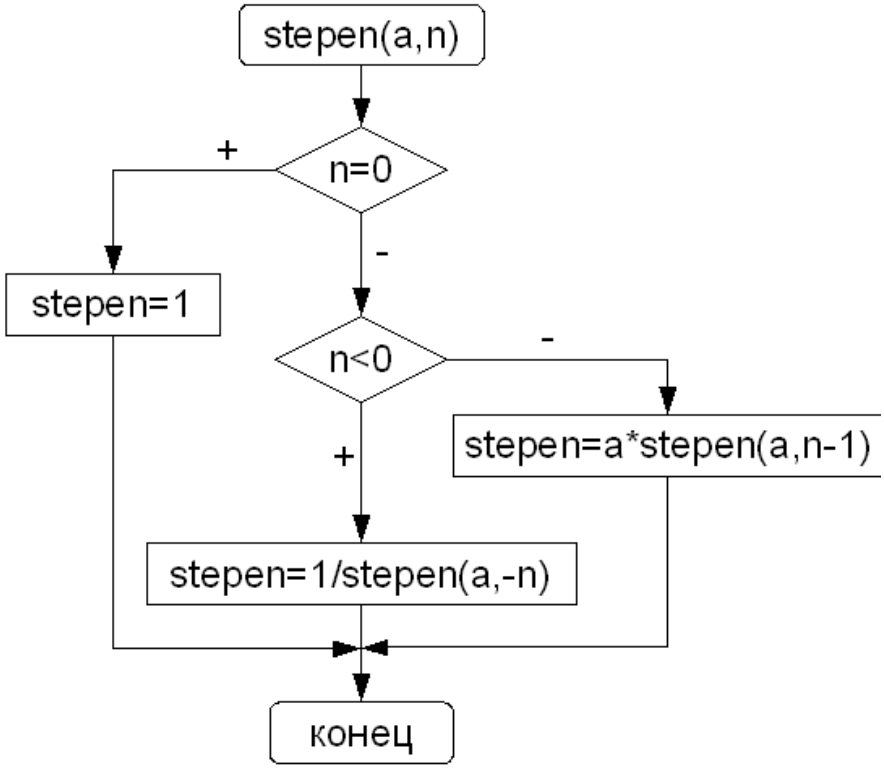

Рисунок 4.10: Рекурсивный алгоритм вычисления степени числа

Фрагмент программы с применением рекурсии:

```
function stepen(a:real;n:word):real;
begin
 if n=0 then stepen:=1
 else
     if n<0 then
         stepen:=1/stepen(a,-n)
```

```
    else
        stepen:=a*stepen(a,n-1);
end;
var
 x:real;k:word;
begin
 writeln('x='); readln(x);
 writeln('k='); readln(k);
 writeln(x:5:2,'^',k,'=',stepen(x,k):5:2);
end.
```

ЗАДАЧА 4.9. Вычислить *n*-е число Фибоначчи.

Если нулевой элемент последовательности равен нулю, первый – единице, а каждый последующий представляет собой сумму двух предыдущих, то это последовательность чисел Фибоначчи (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...).

Алгоритм рекурсивной функции `fibonachi` изображен на рис. 4.11.

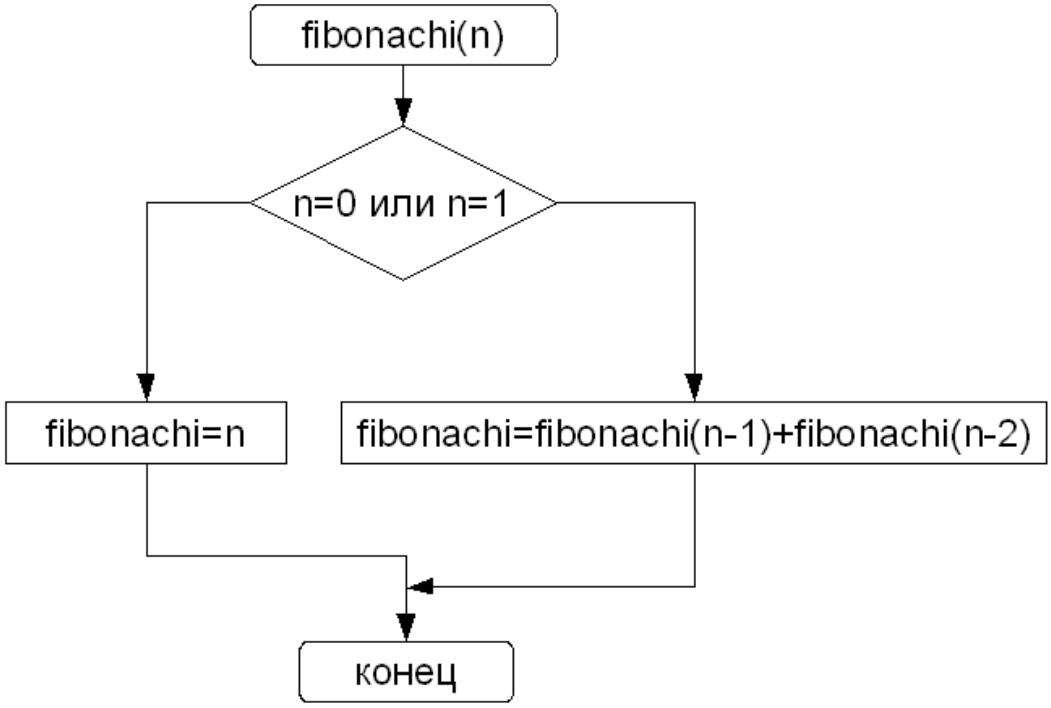

Рисунок 4.11: Рекурсивный алгоритм вычисления числа Фибоначчи

Текст подпрограммы:

```
function fibonachi(n:word):word;
begin
 if (n=0)or(n=1) then  fibonachi:=n
```

```
 else fibonachi:=fibonachi(n-1)+fibonachi(n-2);
end;
var x:word;
begin
 write('Введите номер числа Фибоначчи x=');
 readln(x);
 writeln(x,' -е число Фибоначчи = ',
                                  fibonachi(x));
end.
```

4.7 Особенности работы с подпрограммами

Рассмотрим еще несколько способов *передачи параметров* в подпрограмму.

4.7.1 Параметры-константы

Параметр-константа указывается в заголовке подпрограммы подобно параметру-значению, но перед его именем записывается зарезервированное слово `const`, действие которого распространяется до ближайшей точки с запятой. Параметр-константа передается в подпрограмму как параметр-переменная, но компилятор блокирует любые присваивания параметру-константе нового значения в теле подпрограммы.

В языке Free Pascal можно использовать параметры-переменные и параметры-константы без указания типа, и тогда фактический параметр может быть переменной любого типа, а ответственность за правильность использования того или иного параметра возлагается на программиста.

4.7.2 Процедурные типы

Процедурные типы были разработаны как средство передачи функций и процедур в качестве фактических параметров обращения к другим процедурам и функциям. Для объявления процедурного типа используется заголовок подпрограммы, в котором пропущено ее имя. Существует два процедурных типа: *тип-процедура* и *тип-функция*:

```
type
тип_процедура=procedure (формальные_параметры);
тип_функция=function (формальные_параметры):тип;
```

Кроме того, в программе могут быть объявлены переменные процедурного типа:

```
var
 имя_переменной_1: тип_процедура;
 имя_переменной_2: тип_функция;
```

Например:

```
type
 Fun1=function (x,y:real):real;
 Fun2=function:string;
 Proc1=procedure (x,y:real; var c,z:real);
 Proc2=procedure ();
Var
 F1,f2: Fun1;
 p1,p2: Proc1;
 mass:array [1..5] of Proc2;
```

Обращаться к переменным процедурного типа следует по их адресу[54]:

```
@имя_переменной
```

Рассмотрим механизм передачи подпрограмм в качестве параметра на примере следующей задачи.

ЗАДАЧА 4.10. Создать программу, которая выводит на экран таблицу значений функций $f(x)$ и $g(x)$.

Вычисление и вывод значений осуществляются с помощью функции `VivodFunc`. Ее параметры:

- `a` и `b` — границы интервала изменения аргумента функции;
- `n` — количество точек, на которые будет разбит интервал (`a`, `b`);
- `ff` – имя функции.

```
type    {Описание процедурного типа func.}
 func=function(x:real):real;
function VivodFunc (a,b:real;
                    N:word;ff:func):integer;
 var
     x,y,hx:real;
     f:text;
 begin
     {Шаг изменения переменной x.}
     hx:=(b-a)/N;
     x:=a;
```

54 Операция взятия адреса (п. 2.4.4)

```
        while(x<=b) do
        begin
            y:=ff(x);
            writeln('x=',x:5:2,' y=',y:7:2);
            x:=x+hx;
        end;
    end;
   {Определение функций f и g с описателем far.}
   function f(x:real):real;far;
    begin
        f:=exp(sin(x))*cos(x);
    end;
   function g(x:real):real;far;
    begin
        g:=exp(cos(x))*sin(x);
    end;
   begin
   {Вызов Vivodfunc с функцией f в качестве пара-
метра.}
        VivodFunc(0,1,7,@f);
        writeln;
   {Вызов Vivodfunc с функцией g в качестве пара-
метра.}
        VivodFunc(0,2,8,@g);
   end.
```

В языке Free Pascal есть возможность использования в качестве формальных параметров *массивов функций*.

Модифицируем задачу 4.10. Программа, представленная далее, выводит на экран таблицу значений нескольких функций с помощью функции `VivodFunc`. Здесь в роли параметров функции выступают:

- интервал (`a`, `b`);
- количество узлов `n` интервал (`a`, `b`);
- массив функций `ff`, в которых необходимо вычислить значения;
- количество функций `m` в массиве `ff`.

```
type
 func=function(x:real):real;
{В процедуру VivodFunc передается}
{входной параметр ff -
```

```
                        открытый массив функций[55].}
function VivodFunc(a,b:real;N:word;
            ff:array of func; m:word):integer;
 var
     x,y,hx:real;
     i:integer;
 begin
     hx:=(b-a)/N;
     {Цикл по всем функциям.}
     for i:=0 to m-1 do
     begin
         x:=a;
         while(x<=b) do
         begin
 {Вычисление значения i-й функции в точке x.}
             y:=ff[i](x);
             writeln('x=',x:5:2,' y=',y:7:2);
             x:=x+hx;
         end;
         writeln;
     end;
 end;
function f(x:real):real;far;
 begin
     f:=exp(sin(x))*cos(x);
 end;
function g(x:real):real;far;
 begin
     g:=exp(cos(x))*sin(x);
 end;
{Описание массива восьми функций fff.}
var fff:array[1..8] of func;
 begin
     {Запись реальных функций в массив fff.}
     fff[1]:=@f;  fff[2]:=@g;
     VivodFunc(0,1,7,fff,2);{Вызов процедуры.}
 end.
```

55 Подробнее о передаче массива в подпрограмму см. п. 5.10

Несколько подпрограмм можно объединять в модуль и затем использовать в других программах. Рассмотрим процесс создания модуля более подробно.

4.8 Разработка модулей

Модуль – это автономная программная единица, включающая в себя различные компоненты: константы, переменные, типы, процедуры и функции. Мы использовали модули Lazarus при разработке визуальных приложений. Теперь рассмотрим, как создавать личный модуль.

Модуль имеет следующую структуру:

```
UNIT имя модуля;
INTERFACE
 интерфейсная часть
IMPLEMENTATION
 исполняемая часть
BEGIN
 инициирующая часть
END.
```

Заголовок модуля состоит из служебного слова `UNIT` и следующего за ним имени. Причем имя модуля должно совпадать с именем файла, в котором он хранится. Модуль `EDIT` должен храниться в файле `edit.pas`.

Интерфейсная часть начинается служебным словом `INTERFACE`, за которым находятся объявления всех глобальных объектов модуля: типов, констант, переменных и подпрограмм. Эти объекты будут доступны всем модулям и программам, вызывающим данный модуль.

Исполняемая часть начинается служебным словом `IMPLEMENTATION` и содержит описания подпрограмм, объявленных в интерфейсной части. Здесь же могут объявляться локальные объекты, которые используются только в интерфейсной части и остаются недоступными программам и модулям, вызывающим данный модуль.

В инициирующей части размещаются исполняемые операторы, содержащие некоторый фрагмент программы. Эти программы исполняются до передачи управления основной программе и обычно используются для подготовки ее работы. Инициирующая часть может отсутствовать вместе с начинающим ее словом `begin`.

В качестве примера рассмотрим модуль работы с комплексными числа, в котором будут представлены подпрограммы, реализующие основные операции[56] с комплексными числами:

·`procedure sum(a,b:complex;var c:complex)` — процедура сложения двух комплексных чисел `a` и `b`, результат возвращается в переменной `c`;

·`procedure razn(a,b:complex;var c:complex)` — процедура вычитания двух комплексных чисел `a` и `b`, результат возвращается в переменной `c`;

·`procedure umn(a,b:complex;var c:complex)` — процедура умножения двух комплексных чисел `a` и `b`, результат возвращается в переменной `c`;

·`procedure delenie(a,b:complex;var c:complex)` — процедура деления двух комплексных чисел `a` и `b`, результат возвращается в переменной `c`;

Кроме того, в модуле будет процедура вывода комплексного числа на экран `procedure vivod(a:complex)`.

Текст модуля приведен ниже.

```
UNIT compl;
INTERFACE
type
complex=record
x:real;
y:real;
end;
procedure sum(a,b:complex;var c:complex);
procedure razn(a,b:complex;var c:complex);
procedure umn(a,b:complex;var c:complex);
procedure delenie(a,b:complex;var c:complex);
procedure vivod(a:complex);
IMPLEMENTATION
procedure sum(a,b:complex;var c:complex);
begin
```

56 Суммой двух чисел a+bi и c+di является число (a+c) + (b+d)i, разностью этих чисел является число (a−c) + (b−d)i, произведением — число (ac−bd) + (bc+ ad)i, частным число - $\left(\frac{ab+bd}{c^2+d^2}\right)+\left(\frac{bc+ad}{c^2+d^2}\right)i$.

```
c.x:=a.x+b.x;
c.y:=a.y+b.y;
end;
procedure razn(a,b:complex;var c:complex);
begin
c.x:=a.x-b.x;
c.y:=a.y-b.y;
end;
procedure umn(a,b:complex;var c:complex);
begin
c.x:=a.x*b.x-a.y*b.y;
c.y:=a.y*b.x+a.x*b.y;
end;
procedure delenie(a,b:complex;var c:complex);
begin
c.x:=(a.x*b.x+a.y*b.y)/(b.x*b.x+b.y*b.y);
c.y:=(a.y*b.x-a.x*b.y)/(b.x*b.x+b.y*b.y);
end;
procedure vivod(a:complex);
begin
if a.y>=0 then
    writeln(a.x:1:3,'+',a.y:1:3,'i')
else
    writeln(a.x:1:3,'-',-a.y:1:3,'i')
end;
end.
```

Для работы с модулем запустим Geany, введем текст модуля, сохраним его под именем `compl.pas`. После этого выполним команду **Собрать**. В каталоге, где сохранен модуль, в результате компиляции появятся два файла — с расширением **.o** (в Linux и **.obj** в Windows), и с расширением **.ppu**. Для использования нужен именно файл с расширением **.o**. Теперь скомпилированный файл модуля с расширением **.o** должен находится либо в том же каталоге, что и программа, которая будет его вызывать, либо в том, где находятся стандартные модули вашего компилятора.

Теперь можно написать программу, использующую этот модуль. Текст этой консольной программы приведен ниже.

```
uses compl;
```

```
var g,h,e:complex;
BEGIN
 writeln('Введите первое комплексное число');
 read(g.x,g.y);
 vivod(g);
 writeln('Введите второе комплексное число');
 read(h.x,h.y);
 vivod(g);
 sum(g,h,e);
 writeln('Сумма чисел');
 vivod(e);
 razn(g,h,e);
 writeln('Разность чисел');
 vivod(e);
 umn(g,h,e);
 writeln('Произведение чисел');
 vivod(e);
 delenie(g,h,e);
 writeln('Частное чисел');
 vivod(e);
end.
```

Результаты работы программы представлены на рис. 4.12.

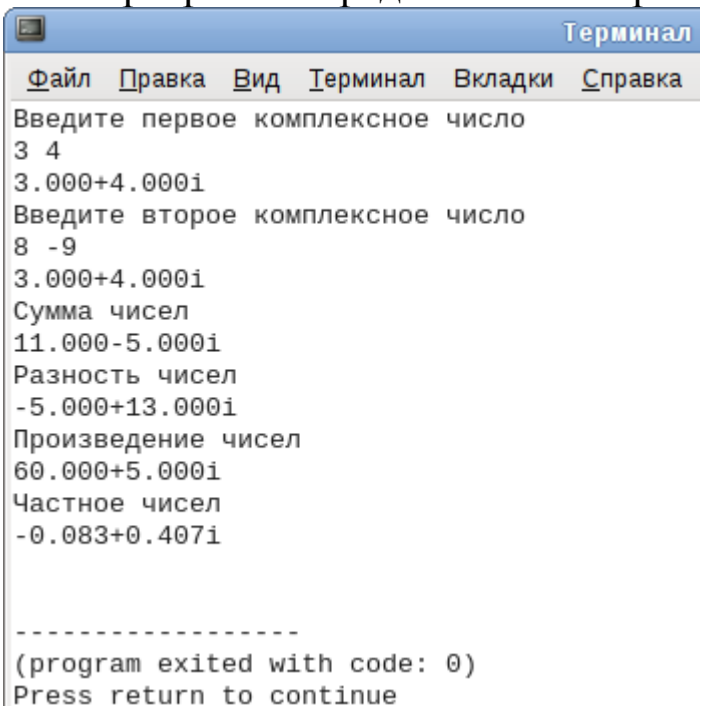

Рисунок 4.12: Операции с комплексными числами

4.9 Задачи для самостоятельного решения

Напишите программу, используя процедуры и функции.

1. Вводится последовательность целых чисел, 0 – конец последовательности. Определить, содержит ли последовательность хотя бы одно число, сумма цифр в котором равна их количеству. Создать процедуру, которая возвращает сумму и количество цифр в числе.

2. Вводится последовательность целых чисел, 0 – конец последовательности. Определить, содержит ли последовательность хотя бы одно совершенное число. Для определения совершенного числа создать функцию.

3. Вводится последовательность из N целых положительных элементов. Определить, содержит ли последовательность хотя бы одно простое число. Для определения простого числа создать функцию.

4. Вводится последовательность из N целых положительных элементов. Посчитать количество чисел палиндромов. Для определения палиндрома создать функцию.

5. Вводится последовательность из N целых положительных элементов. Подсчитать количество совершенных чисел в последовательности. Для определения совершенного числа создать функцию.

6. Поступает последовательность целых положительных чисел, 0 – конец последовательности. Определить, в каком из чисел больше всего делителей. Для подсчета делителей числа использовать функцию.

7. Поступает последовательность целых положительных чисел, 0 – конец последовательности. Определить, в каком из чисел больше всего цифр. Для подсчета количества цифр числа использовать функцию.

8. Вывести на экран значения функции $f(x)=x-2e^x$ и ее первой производной $f'(x)$, в диапазоне от -5 до 5, с шагом 0.5. Для вычисления значений $f(x)$ и $f'(x)$ создать функции.

9. Вводится последовательность из N целых положительных элементов. Найти число с минимальным количеством цифр. Для определения количества цифр в числе использовать функцию.

10. Вводится последовательность из N целых элементов. Для всех положительных элементов последовательности вычислить значение факториала и вывести его на печать. Вычисление факториала оформить в виде функции.

11. Поступает последовательность целых положительных чисел, 0 – конец последовательности. Вывести на экран все числа последовательности, не являющиеся простыми, и их делители. Определение простого числа оформить в виде функций.

12. Вводится последовательность из N целых элементов. Вывести на экран все числа последовательности, являющиеся совершенными, и их делители. Определение совершенного числа оформить в виде функций.

13. Поступает последовательность целых положительных чисел, 0 – конец последовательности. Найти среднее арифметическое простых чисел в этой последовательности. Определение простого числа оформить в виде функций.

14. В последовательности из N целых положительных элементов найти число с наибольшим количеством нулей в своем представлении. Составить функцию для подсчета нулей в числе.

15. В последовательности из N целых положительных элементов найти сумму всех палиндромов. Для определения палиндрома создать функцию.

16. Поступает последовательность целых положительных чисел, 0 – конец последовательности. Посчитать количество элементов последовательности, имеющих в своем представлении цифру 0. Создать процедуру, возвращающую значение истина, если в числе есть нули, и ложь в противном случае.

17. Поступает последовательность целых положительных чисел, 0 – конец последовательности. Для каждого числа найти количество нулей и единиц. Создать процедуру, которая возвращает количество нулей и единиц в заданном числе.

18. Вводится последовательность из N целых элементов. Для каждого элемента последовательности найти среднее значение его цифр. Создать функцию для расчета среднего значения цифр в числе.

19. Поступает последовательность целых положительных чисел, 0 – конец последовательности. Определить количество цифр и наименьшую цифру для каждого числа последовательности. Написать процедуру, которая для заданного числа возвращает два параметра: количество цифр в нем и наименьшую цифру.

20. Вводится последовательность из N целых элементов. Для каждого элемента последовательности вывести на экран количество цифр и количество делителей. Написать процедуру, которая рассчитывает оба параметра.

21. Поступает последовательность целых положительных чисел, 0 – конец последовательности. Записать каждое число последовательности в обратном порядке. Например, 12345 → 54321. Создать функцию для преобразования числа.

22. Поступает последовательность целых положительных чисел, 0 – конец последовательности. Для каждого элемента последовательности вывести на экран количество цифр в числе и наибольшую цифру. Написать процедуру, которая возвращает количество цифр и наибольшую цифру заданного числа.

23. Вводится последовательность из N целых положительных элементов. Для простых элементов последовательности определить сумму цифр. Написать процедуру, которая проверяет, является ли число простым, и вычисляет сумму цифр в нем. Если число простым не является, то процедура выдает соответствующее сообщение.

24. Поступает последовательность целых положительных чисел, 0 – конец последовательности. Для каждого числа определить сумму и количество цифр в числе. Написать процедуру для подсчета суммы и количества цифр в числе.

25. Вывести на экран значения функций $f(x)=\sqrt{(5x^3)}\cdot\sin(x^2)$ и $g(x)=\begin{cases} e^{\frac{x}{4}}, если\ x\geq 0 \\ \sqrt[5]{x^2}, если\ x<0 \end{cases}$ в диапазоне от *a* до *b,* с шагом *h*. Для вычисления значений *f*(*x*) и g(*x*) создать функции.

5 Использование языка Free Pascal для обработки массивов

5.1 Общие сведения о массивах

В предыдущих главах мы рассматривали задачи, в которых использовались скалярные переменные. Однако при обработке однотипных данных (целочисленных значений, строк, дат и т.п.) оказывается удобным использовать массивы. Например, можно создать массив для хранения значений температуры в течение года. Вместо создания множества (365) переменных для хранения каждой температуры, например `temperature1`, `temperature2`, `temperature3`, ... `temperature365` можно использовать один массив с именем `temperature`, где каждому значению будет соответствовать порядковый номер (рис. 5.1).

№ элемента	1	2	3	4	...	364	365
temperature	-1.5	-3	-6.7	1		2	-3

Рисунок 5.1: Массив значений температур

Таким образом, можно дать следующее определение.

Массив – структурированный тип данных, состоящий из фиксированного числа элементов одного типа.

Массив, представленный на рисунке 5.2, имеет 7 элементов, каждый элемент сохраняет число вещественного типа. Элементы в массиве пронумерованы от 1 до 7. Такого рода массив, представляющий собой просто список данных одного и того же типа, называют простым, или одномерным массивом. Для доступа к данным, хранящимся в определенном элементе массива, необходимо указать имя массива и порядковый номер этого элемента, называемый индексом.

1-й элемент массива	2-й элемент массива	3-й элемент массива	4-й элемент массива	5-й элемент массива	6-й элемент массива	7-й элемент массива
-1.5	-3.913	13.672	-1.56	45.89	4.008	-3.61

Рисунок 5.2: Одномерный массив из 7 вещественных чисел

Если возникает необходимость хранения данных в виде таблиц, в формате строк и столбцов, то необходимо использовать многомерные массивы.

На рисунке 5.3 приведен пример массива, состоящего из трех строк и четырех столбцов. Это *двумерный массив*. Строки в нем мож-

но считать первым измерением, а столбцы — вторым. Для доступа к данным, хранящимся в этом массиве, необходимо указать имя массива и два индекса, первый должен соответствовать номеру строки, а второй — номеру столбца, в которых хранится необходимый элемент.

		Номера столбцов			
		1	**2**	**3**	**4**
Номера строк	**1**	6.3	4.3	-1.34	5.02
	2	1.1	4.7	8.12	8.5
	3	-2.4	-6.2	11.23	8.18

Рисунок 5.3: Двумерный числовой массив

После общего знакомства с понятием «массив», рассмотрим работу с массивами в языке Free Pascal.

5.2 Описание массивов

Для описания массива служат служебные слова `array of`. Описать массив можно двумя способами:

Ввести новый тип данных, а потом описать переменные нового типа. В этом случае формат оператора `type` следующий:

```
type
имя_типа = array [тип_индекса] of
                                    тип_компонентов;
```

В качестве **типа_индекса** следует использовать перечислимый тип. **Тип_компонентов** — это любой ранее определенный тип данных, например:

```
type
massiv=array[0..12] of real;
//Тип данных massiv из 13 элементов,
//элементы нумеруются от 0 до 12.
dabc=array[-3..6] of integer;
//Тип данных dabc из 10 элементов,
//элементы нумеруются от -3 до 6.
var
x,y:massiv;
z: dabc;
```

Можно не вводить новый тип, а просто описать переменную следующим образом:

```
var
переменная: array [тип_индекса] of
                                  тип_переменной;
```

Например:

```
var
z,x: array[1..25] of word;
//Массивы z и x из 25 значений типа word,
//элементы нумеруются от 1 до 25.
g:array[-3..7] of real;
//Массив g из 11 значений типа real,
//которые нумеруюся от -3 до 7.
```

Для описания массива можно использовать предварительно определенные константы:

```
const
n=10;
m=12;
var
a: array[1..n] of real;
b: array[0..m] of byte;
```

Константы должны быть определены до использования, так как массив не может быть переменной длины!

Двумерный массив (матрицу) можно описать, применив в качестве базового типа (типа компонентов) одномерный:

```
type
massiv=array[1..200] of real;
matrica=array[1..300] of massiv;
var
ab:matrica;
```

Такая же структура получается при использовании другой формы записи:

```
Type
matrica = array [1..300,1..200] of real;
var
ab:matrica;
```

или

```
var ab:array [1..300,1..200] of real;
```

При всех трех определениях мы получали матрицу вещественных чисел, состоящую из 300 строк и 200 столбцов.

Аналогично можно ввести трехмерный массив, или массив большего числа измерений:

```
type
abc=array [1..4,0..6,-7..8,3..11] of real;
var b:abc;
```

5.3 Операции над массивами

Для работы с массивом как с единым целым надо использовать имя массива (без указания индекса в квадратных скобках). Для доступа к элементу массива необходимо указать имя массива и в квадратных скобках порядковый номер элемента массива, например x[1], y[5], c[25], A[8].

В языке Free Pascal определена операция присваивания для массивов, идентичных по структуре (с одинаковыми типами индексов и компонентов). Например, если массивы `C` и `D` описаны как

```
var C,D: array [0..30] of real;
```

то можно записать оператор `C:=D;`. Такая операция сразу всем элементам массива `C` присвоит значения соответствующих им по номерам элементов массива `D`.

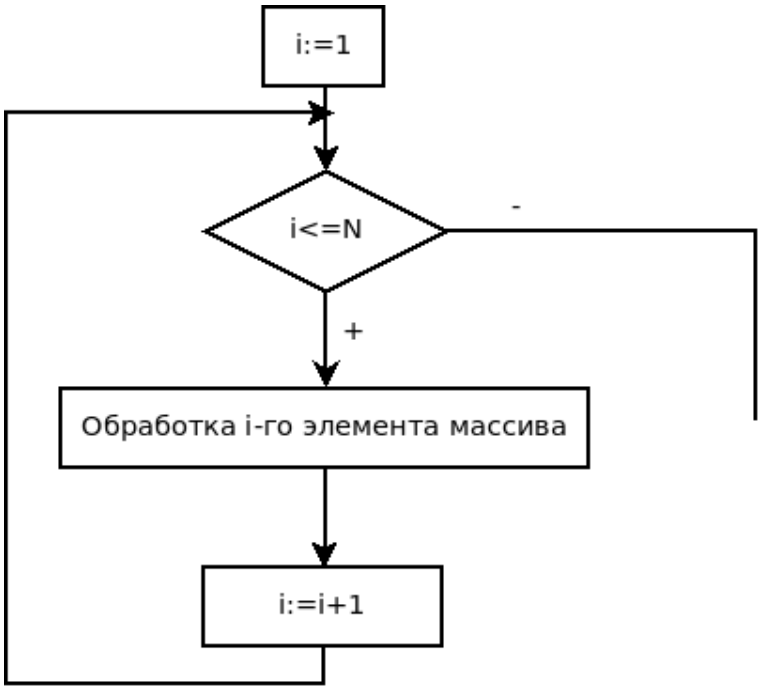

Рисунок 5.4: Блок-схема обработки элементов массива с использованием цикла с предусловием

Выполнение любой другой операции над массивом надо организовывать поэлементно, для чего необходимо организовать цикл, в котором последовательно обрабатывать элементы массива, сначала обрабатываем первый элемент массива, затем второй, третий, …, n-й (рис. 5.4-5.5).

Для обработки элементов массива удобно использовать цикл `for`.

Рисунок 5.5: Блок-схема обработки элементов массива с использованием цикла с for

5.4 Ввод-вывод элементов массива

Язык Free Pascal не имеет специальных средств ввода-вывода всего массива, поэтому данную операцию следует организовывать поэлементно.

При вводе массива необходимо последовательно вводить 1-й, 2-й, 3-й и т.д. элементы массива, аналогичным образом поступить и при выводе. Следовательно, как для ввода, так и для вывода необходимо организовать стандартный цикл обработки массива (рис. 5.6 - 5.7).

Для обращения к элементу массива необходимо указать имя массива и в квадратных скобках номер элемента, например `X[5]`, `b[2]` и т.д.

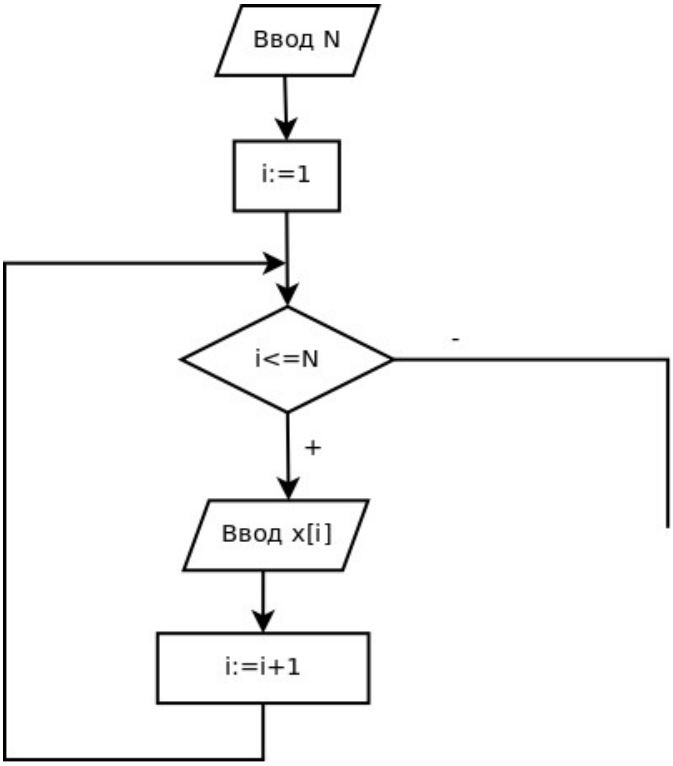

Рисунок 5.6: Алгоритм ввода массива X с использованием цикла с предусловием

5.4.1 Организация ввода-вывода

Реализуем эти алгоритмы в консольных приложениях.

```
/Ввод элементов массива X с помощью цикла while.
var x: array [1..100] of real;
    i,n: integer;
begin
```

```
    writeln ('введите размер массива'); readln(N);
    i:=1;
    while (i<=N) do
    begin
        write('x(',i,')= ');
        readln(x[i]);
        i:=i+1
    end;
end.
//Ввод элементов массива X с помощью цикла for.
var i,n: integer;
    x: array [1..100] of real;
begin
    readln(N);
    for i:=1 to N do
    begin
        write('x(',i,')= ');
        readln(x[i])
    end;
end.
```

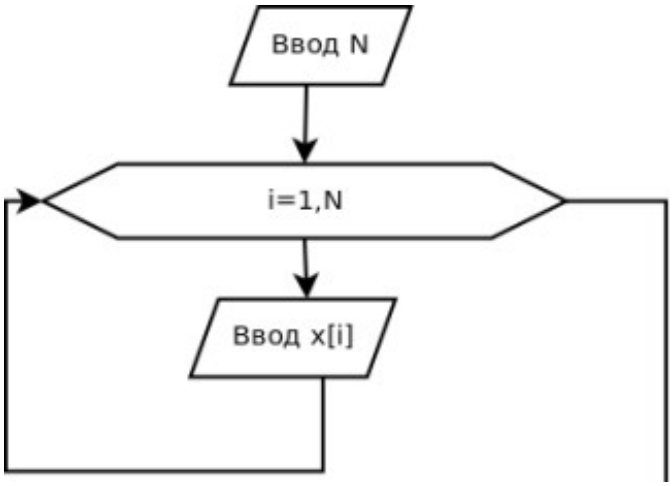

Рисунок 5.7: Алгоритм ввода массива X с использованием блока модификации

Цикл `for … do` удобнее использовать для обработки всего массива, и в дальнейшем при выполнении подобных операций с массивами мы чаще будем применять именно его.

Вывод массива организуется аналогично вводу, только вместо блока ввода элемента массива будет блок вывода.

Предлагаем читателю рассмотреть несколько вариантов вывода массива вещественных чисел

`a=(1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8),`

самостоятельно разобраться, чем они отличаются друг от друга, и решить, какой из вариантов удобнее в конкретной задаче.

```
//Вариант 1
for i: = 1 to n do
        write (a[i]:3:1);
```

Результат первого варианта вывода массива на экран представлен на рис. 5.8. Обратите внимание, что между числами в данном варианте отсутствует пробел.

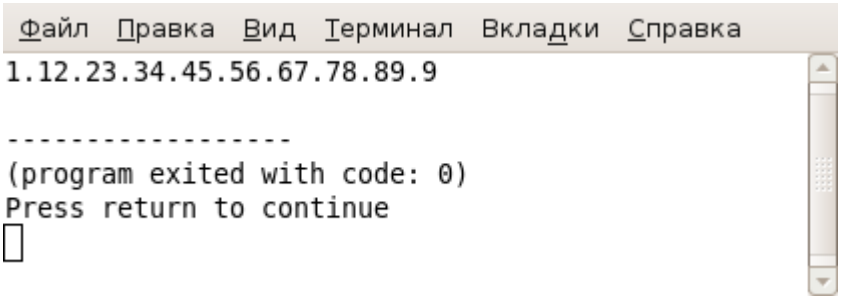

Рисунок 5.8: Результат первого варианта вывода массива

```
//Вариант 2
for i: = 1 to n do
            write (a[i]:6:2);
```

Результат второго варианта вывода массива на экран представлен на рис. 5.9

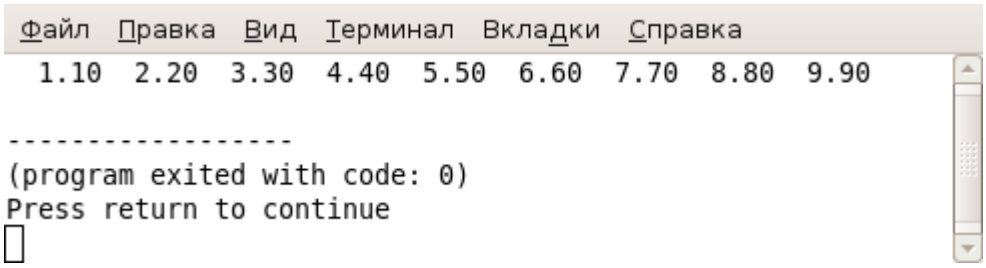

Рисунок 5.9: Результат второго варианта вывода массива

```
// Вариант 3
for i: = 1 to n do
            write (a[i]:3:1,' ');
```

Результат третьего варианта вывода массива на экран представлен на рис. 5.10.

```
// Вариант 4
writeln ('массив A');
for i:=1 to n do
            writeln(a[i]:6:2);
```

Результат четвертого варианта вывода массива на экран представлен на рис. 5.11.

```
// Вариант 5
 for i:=1 to n do
        write ('a(',i,')=',a[i]:3:1,' ');
```

Результат пятого варианта вывода массива на экран представлен на рис. 5.12.

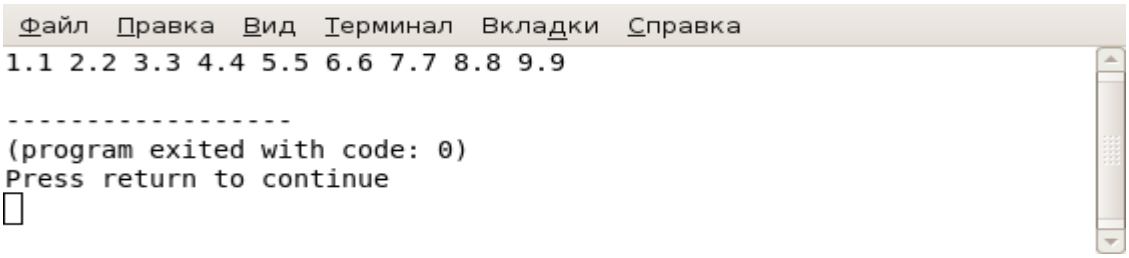

Рисунок 5.10: Результат третьего варианта вывода массива

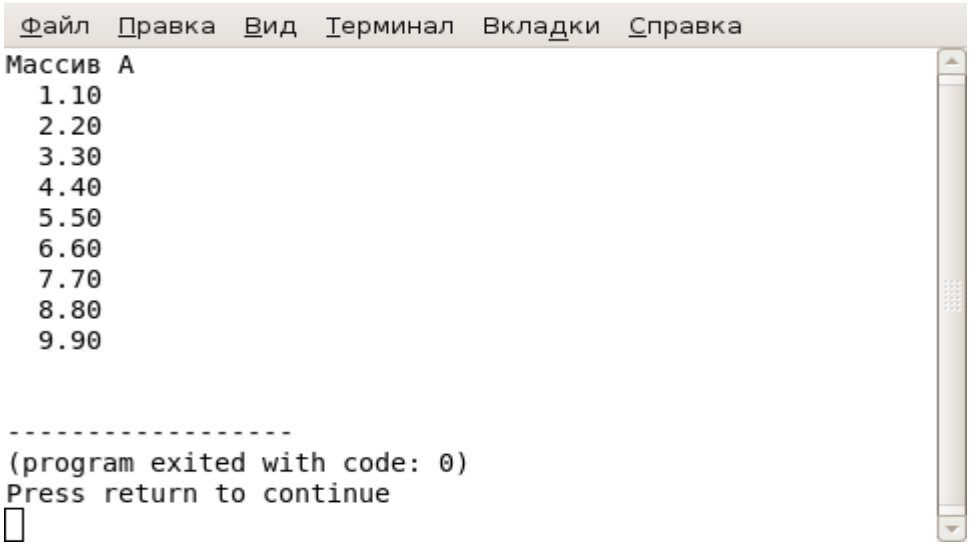

Рисунок 5.11: Результат четвертого варианта вывода массива

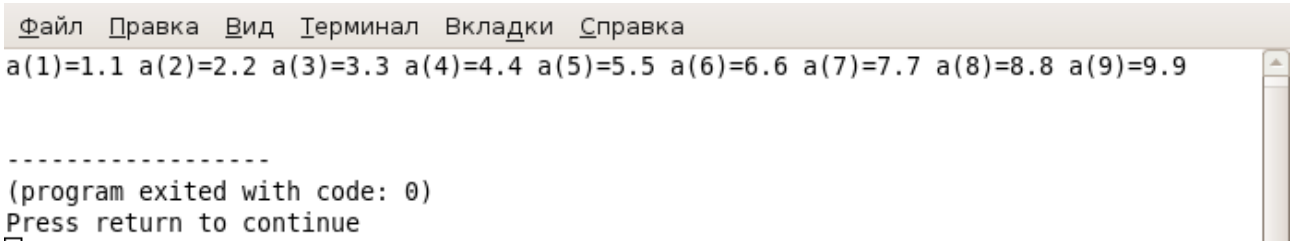

Рисунок 5.12: Результат пятого варианта вывода массива

```
// Вариант 6
for i:=1 to n do
    writeln ('a(',i, ')= ',a[i]:6:2);
```

Результат шестого варианта вывода массива на экран представлен на рис. 5.13.

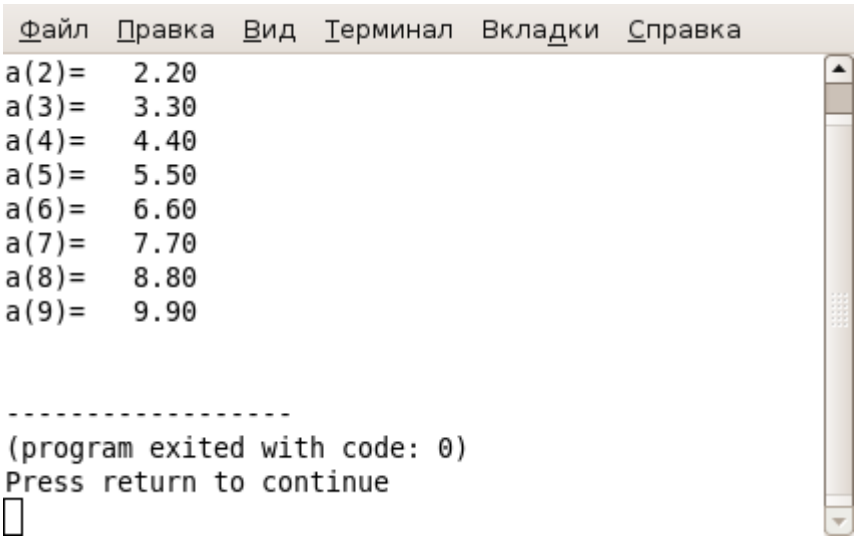

Рисунок 5.13: Результат шестого варианта вывода массива

5.4.2 Ввод-вывод данных в визуальных приложениях

Рассмотрим возможности организации ввода-вывода массивов в визуальных приложениях, для вывода массивов можно использовать стандартный компонент типа `TEdit`.

Рассмотрим эти возможности на примере стандартной задачи вывода массива вещественных чисел

`a=(1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8).`

Расположим на форме кнопку (компонент типа `TButton`) и компонент типа `TEdit` (см. рис. 5.14).

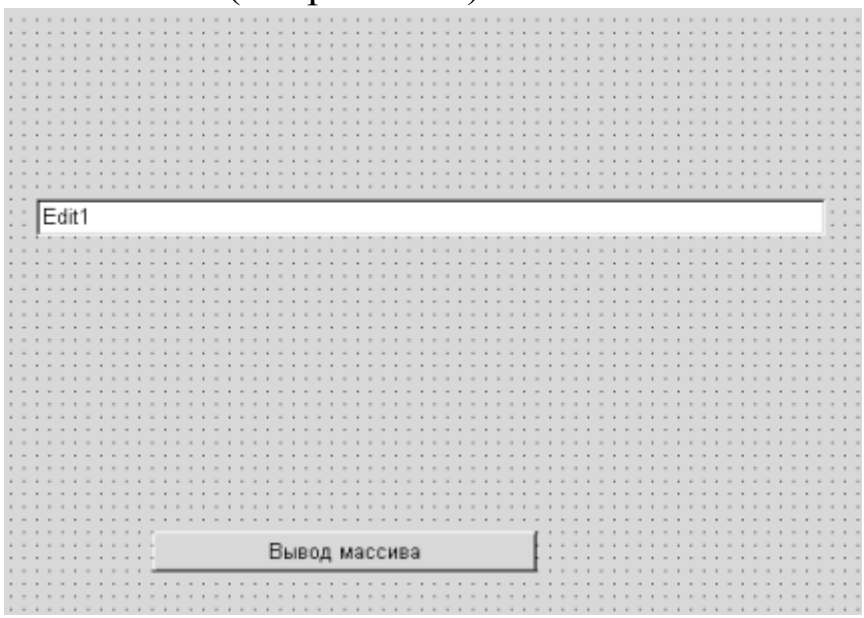

Рисунок 5.14: Форма для задачи вывода массива вещественных чисел

В табл. 5.1 - 5.2 приведены свойства компонентов типа TButton и TEdit.

Таблица 5.1: Свойства компонента Edit1

Свойство	Name1	Text	Height	Left	Top	Width	ReadOnly
Значение	Edit1	' '	23	103	184	492	True

Таблица 5.2: Свойства компонента Button1

Свойство	Name1	Caption	Height	Left	Top	Width
Значение	label1	Button1	25	177	392	239

Наше приложение по щелчку по кнопке «**Вывод массива**» будет выводить массив a в поле ввода Edit1. Алгоритм вывода массива заключается в следующем: каждое число переводится в строку с помощью функции FloatToStr, после чего полученная строка добавляется к полю вывода. Текст обработчика события Button1Click с комментариями приведен ниже.

```
procedure TForm1.Button1Click(Sender: TObject);
var
//Исходный массив a.
a:array [1..8] of
 real=(1.1,2.2,3.3,4.4, 5.5, 6.6, 7.7, 8.8);
 i:integer;
//В переменной n хранится
//количество элементов в массиве
//вещественных чисел.
 n:integer=8;   S:string='';
begin
 Edit1.Text:='';
//Цикл for для последовательной
//обработки всех элементов массива a.
     for i:=1 to n do
//Добавление в поле ввода Edit1
//строки, которая получена из
//элемента массива a[i].
 Edit1.Text:=Edit1.Text+FloatToStr(a[i])+' ';
end;
```

После щелчка по кнопке «**Вывод массива**» окно приложения станет подобным тому, как представлено на рис. 5.15.

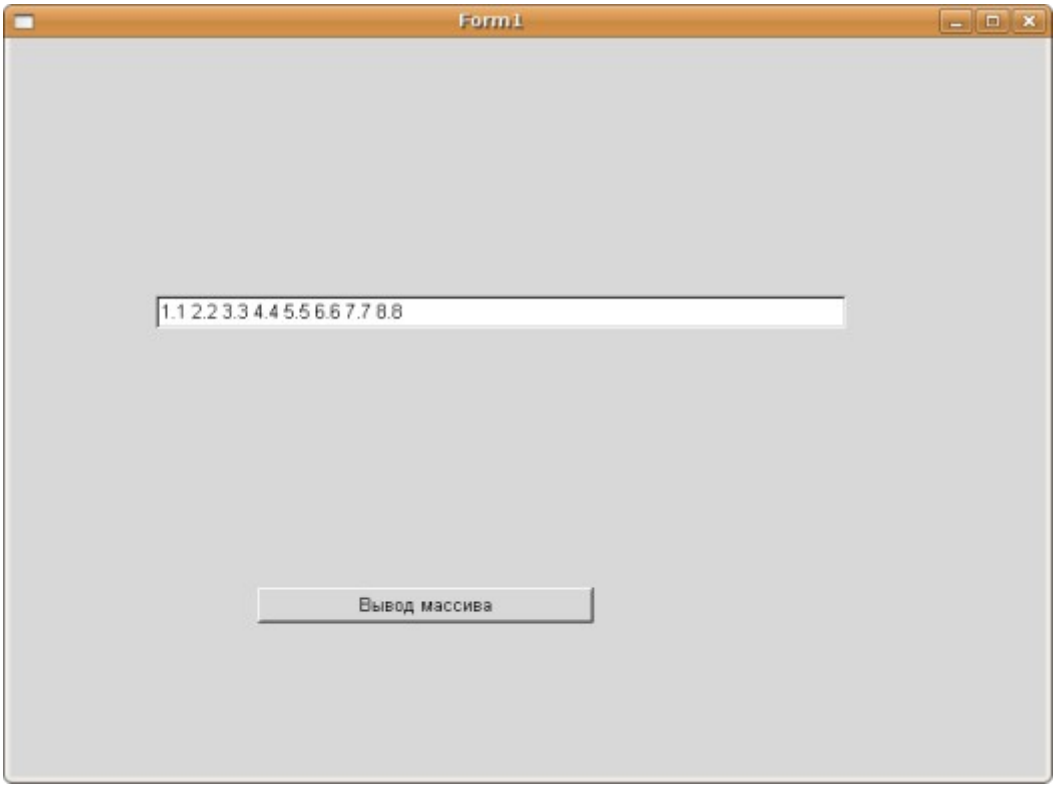

Рисунок 5.15: Вывод массива в поле ввода

Для ввода массивов в Lazarus в простейшем случае можно воспользоваться функцией `InputBox`. В качестве примера рассмотрим проект, в котором будет осуществляться ввод массива при щелчке по кнопке. На форме расположим единственную кнопку. Текст обработчик события `Button1Click` с комментариями приведен ниже.

```
procedure TForm1.Button1Click(Sender: TObject);
var i,n:byte; X:array [1..20] of real;
begin
//Количество элементов массива.
n:=StrToInt(InputBox('Ввод элементов
                     массива','n=','7'));
for i := 1 to n do     //Поэлементный ввод.
//Ввод очередного элемента массива
X[i]:=StrToFloat(InputBox('Ввод элементов
        массива','Введите '+IntToStr(i)+ '
                              элемент','0,00'));
end;
```

При щелчке по кнопке на экране появится окно для ввода размера массива (см. рис. 5.16). После корректного ввода размера массива последовательно будут появляться диалоговые окна для ввода очередного элемента, подобные представленному на рис. 5.17.

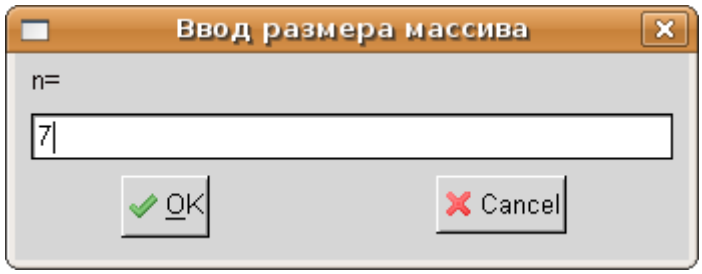

Рисунок 5.16: Ввод размера массива

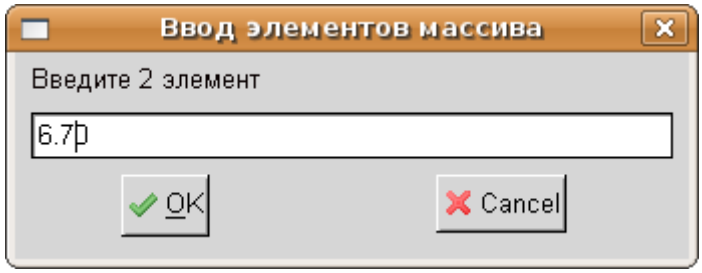

Рисунок 5.17: Ввод второго элемента массива

Для вывода массива с помощью диалогового окна можно применить функцию `MessageDlg`:

```
for i:= 1 to n do
 MessageDlg('X['+IntToStr(i)+']=
 '+FloatToStr(X[i]), MtInformation, [mbOk],0),
```

которая будет открывать отдельное окно для каждого элемента (см. рис. 5.18).

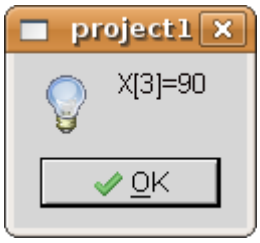

Рисунок 5.18: Вывод третьего элемента массива

Чтобы у пользователя была возможность просматривать элементы массива одновременно, можно из них сформировать строку, а затем вывести ее, например, на форму в виде метки или в окне сообщения:

```
var
i,n:byte;
X:array  [1..20]  of  real;
S:string;
begin
//В переменную строкового типа записывается
//пустая строка.
   S:='';
   for i:=1 to n do
```

```
//Выполняется слияние элементов массива,
//преобразованных в строки и символов пробела -
//результат строка, в которой элементы массива
//перечислены через пробел.
S:=S+FloatToStrF(X[i],ffFixed,5,2)+' ';
//Вывод строки на форму в виде метки.
Label2.Caption:=S;
//Аналогично строку можно вывести в виде
//сообщения, используя функцию
//MessageDlg(S,MtInformation,[mbOk],0);
```

Наиболее универсальным способом ввода-вывода как одномерных, так и двумерных массивов является компонент типа `TStringGrid` («таблица строк»). Познакомимся с этим компонентом подробнее.

На рис. 5.19 представлен проект формы, на которой расположен компонент типа `TStringGrid`[57] (таблица строк). Ячейки таблицы строк предназначены для чтения или редактирования данных. Этот компонент может служить как для ввода, так и для вывода массива.

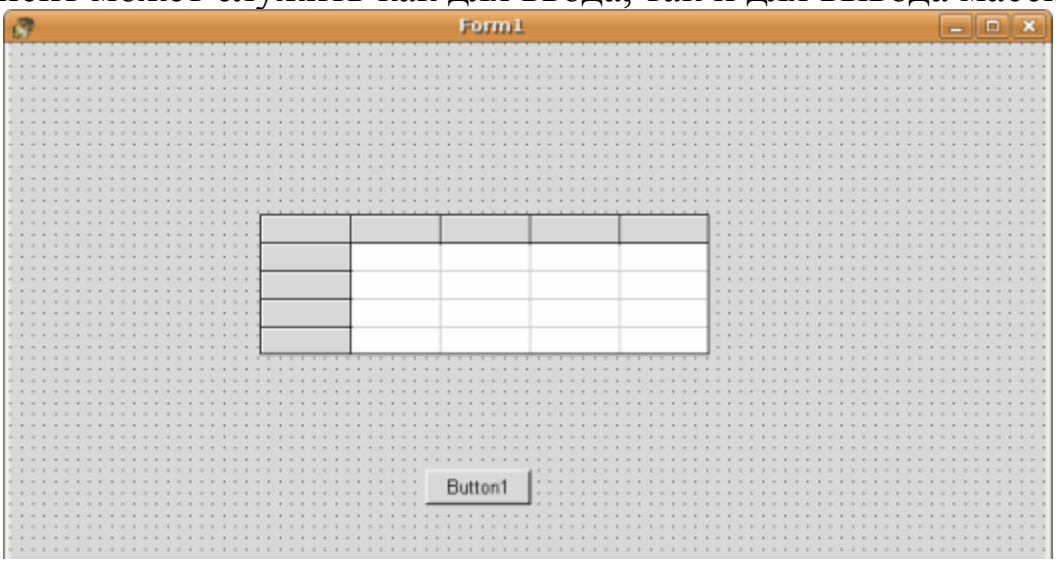

Рисунок 5.19: Форма с компонентом "таблица"

Основные свойства этого компонента представлены в таблице 5.3.

Таблица 5.3: Основные свойства компонента типа `TstringGrid`

Свойство	Описание
`Name`	Имя компонента
`ColCount`	Количество столбцов таблицы

57 Компонент типа `TStringGrid` расположен на странице `Additional`.

Свойство	Описание
RowCount	Количество строк таблицы
Cells	Двумерный массив, в котором хранится содержимое таблицы. Ячейка таблицы, находящаяся на пересечении столбца номер col и строки номер row, определяется элементом Cells [col,row]; строки в компоненте нумеруются от 0 до RowCount-1, (столбцы от 0 до ColCount-1)
FixedCols	Количество зафиксированных слева столбцов таблицы, которые выделяются цветом и при горизонтальной прокрутке таблицы остаются на месте
FixedRows	Количество зафиксированных сверху строк таблицы, которые выделяются цветом и при вертикальной прокрутке таблицы остаются на месте
ScrollBars	Параметр определяет наличие полос прокрутки, возможны следующие значения параметра: •ssNone — отсутствие полос прокурутки (пользователь может в этом случае перемещаться только с помощью курсора) •ssHorizontal, ssVertical или ssBoth — наличие горизонтальной, вертикальной или обеих полос прокрутки; •ssAutoHorizontal, ssAutoVertical или ssAutoBoth — появление горизонтальной, вертикальной или обеих полос прокрутки по мере необходимости.
Options.goEditing	Логическая переменная, которая определяет, может пользователь (True) или нет (False) редактировать содержимое ячеек таблицы
Options.goTab	Логическая переменная, которая разрешает (True) или запрещает (False) использование клавиши **Tab** для перемещения курсора в следующую ячейку таблицы
DefaultColWidth	Ширина столбцов таблицы
DefaultRowHeight	Высота строк таблицы

Свойство	Описание
GridLineWidth	Ширина разграничительных линий между ячейками таблицы
Left	Расстояние от таблицы до левой границы формы
Top	Расстояние от таблицы до верхней границы формы
DefaultColWidth	Ширина столбцов таблицы
DefaultRowHeight	Высота строк таблицы
Height	Высота компонента типа TStringGrid
Width	Ширина компонента типа TStringGrid
Font	Шрифт, которым отображается содержимое ячеек таблицы

Рассмотрим использование компонента для ввода-вывода массивов на примере программы, с помощью которой можно осуществить ввод массива из восьми вещественных чисел, а затем вывести его в обратном порядке.

Разместим на форме две метки, два компонента типа TStringGrid и одну кнопку. Свойства компонентов StringGrid1 и StringGrid2 можно задать такими же, как показано в табл. 5.4.

Таблица 5.4: Основные свойства компонента типа TStringGrid

Свойство	StringGrid1	StringGrid2	Описание
ColCount	8	8	Количество столбцов таблицы
RowCount	1	1	Количество строк таблицы
FixedCols	0	0	Количество зафиксированных слева столбцов таблицы
FixedRows	0	0	Количество зафиксированных сверху строк таблицы

Свойство	StringGrid1	StringGrid2	Описание
`Options.goEditing`	True	False	Логическая переменная, которая определяет возможность пользователю редактировать содержимое ячеек таблицы
`Left`	186	186	Расстояние от таблицы до левой границы формы
`Top`	72	216	Расстояние от таблицы до верхней границы формы
`Height`	24	24	Высота компонента
`Width`	518	518	Ширина компонента

Окно формы приложения ввода—вывода массива будет подобным представленному на рис. 5.20.

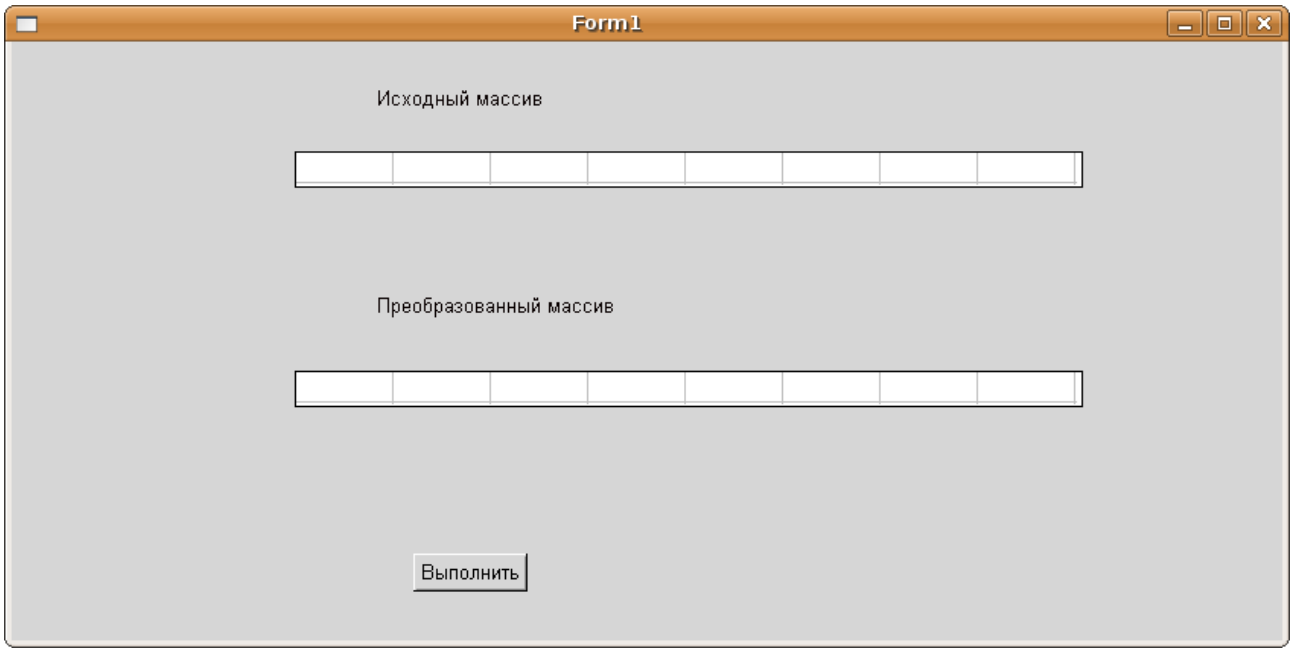

Рисунок 5.20: Окно формы приложения

В первую таблицу будем вводить элементы массива, во вторую выводить преобразованный массив. Щелчок по кнопке **Выполнить** вызовет следующую подпрограмму:

```
procedure Tform1.Button1Click(
                                Sender: TObject) ;
```

```
var  n,i:integer;   a:array [0..7] of real;
begin
for i:=0 to 7 do        //Ввод массива.
//Из поля таблицы считывается элемент,
//преобразовывается в число и
//присваивается элементу массива.
a[i]:=StrToFloat(StringGrid1.Cells[i,0]);
for i:=0 to 7 do        //Вывод массива.
//Элемент массива преобразовывается в строку
//и помещается в поле таблицы.
StringGrid2.Cells[i,0]:=
            FloatToStrF(a[7-i],ffFixed,5,2);
end;
```

При запуске программы на выполнение появляется окно приложения, подобное представленному на рис. 5.20, пользователь вводит исходный массив, щелкаем по кнопке **Выполнить**, после чего окно приложения принимает вид, подобный представленному на рис. 5.21.

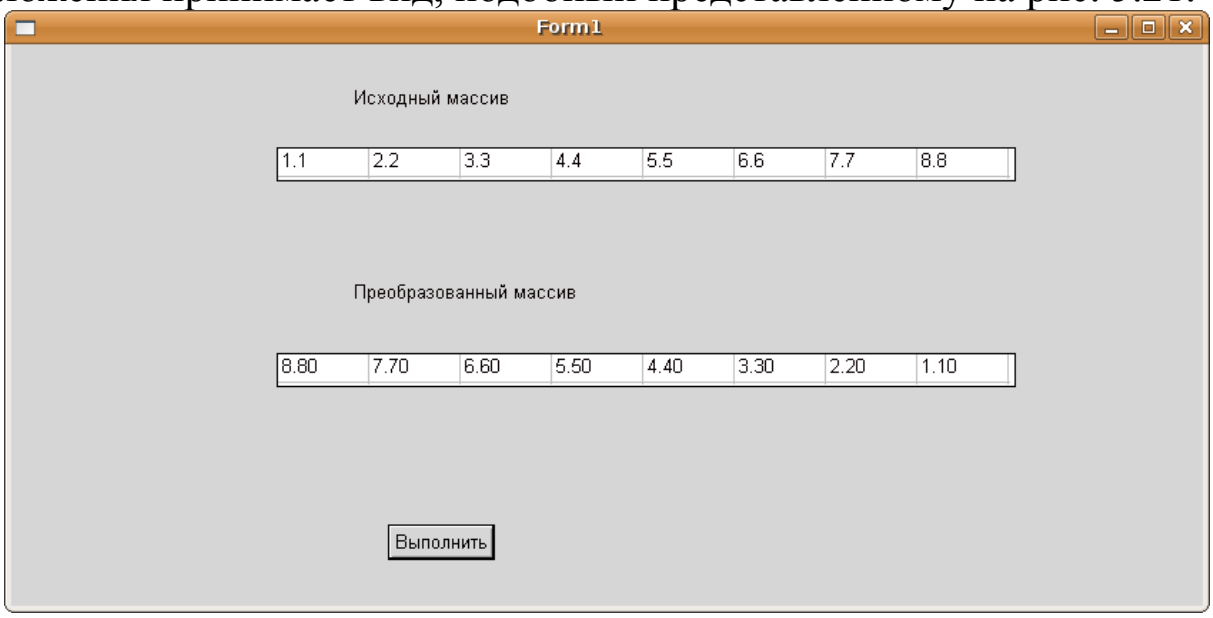

Рисунок 5.21: Окно программы ввода-вывода массива

В этом параграфе были рассмотрены различные способы ввода-вывода как в консольных, так и в визуальных приложениях. В дальнейшем мы будем использовать те из них, которые удобнее при решении конкретной задачи.

Теперь перейдем к рассмотрению основных алгоритмов обработки одномерных массивов, многие из которых аналогичны соответствующим алгоритмам обработки последовательностей (вычисление

суммы, произведения, поиск элементов по определенному признаку, выборки и т. д.). Отличие заключается в том, что в массиве одновременно доступны все его компоненты, поэтому становятся возможными более сложные действия с массивами (например, сортировка элементов массива, удаление и вставка элементов и т.д.).

5.5 Вычисление суммы и произведения элементов массива

Нахождение суммы и произведения элементов массива аналогично подобным алгоритмам нахождения суммы и произведения элементов последовательности.

Дан массив `X`, состоящий из `n` элементов. Найти *сумму элементов* этого массива. Переменной `S` присваивается значение, равное нулю, затем последовательно к переменной `S` добавляются элементы массива `X`.

Блок-схема алгоритма расчета суммы приведена на рис. 5.22. Соответствующий алгоритму фрагмент программы будет иметь вид:

```
s:=0;
for i:=1 to n do s:=s+x[i];
writeln('s=',s:7:3);
```

Найдем *произведение элементов* массива `X`. Решение задачи сводится к тому, что значение переменной `P`, в которую предварительно была записана единица, последовательно умножается на значение `i`–го элемента массива. Блок-схема алгоритма приведена на рис. 5.23.

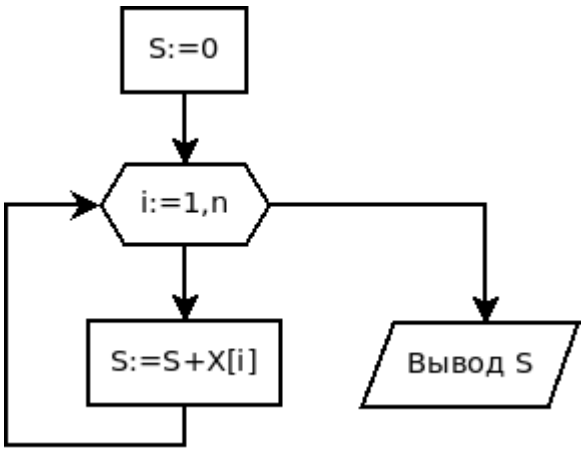

Рисунок 5.22: Алгоритм нахождения суммы элементов массива

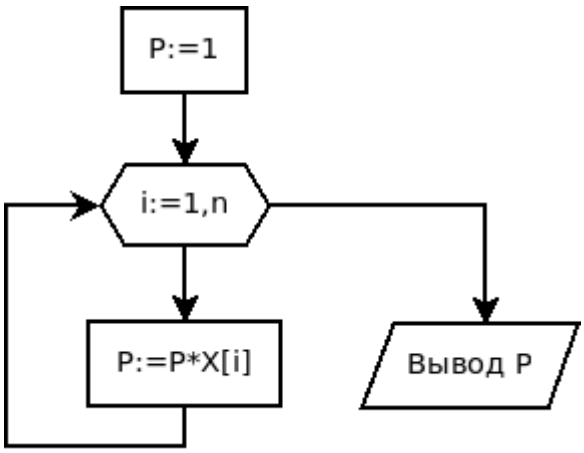

Рисунок 5.23: Алгоритм нахождения произведения элементов массива

Соответствующий фрагмент программы будет иметь вид:

```
p:=1;
for i:=1 to n do p:=p*x[i];
writeln('P=',P:7:3);
```

5.6 Поиск максимального элемента в массиве и его номера

Рассмотрим задачу поиска максимального элемента (Max) и его номера (Nmax) в массиве X, состоящем из n элементов.

Алгоритм решения задачи следующий. Предположим, что первый элемент массива является максимальным, и запишем его в переменную Max, а в Nmax – его номер (число 1).

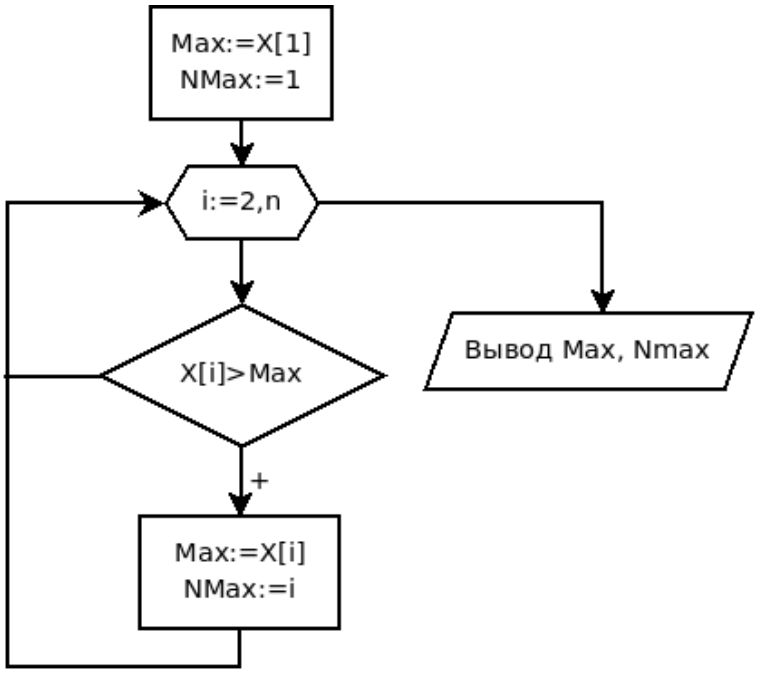

Рисунок 5.24: Алгоритм поиска максимального элемента массива и его номера

Затем все элементы, начиная со второго, сравниваем в цикле с максимальным. Если текущий элемент массива оказывается больше максимального, то записываем его в переменную Max, а в переменную Nmax – текущее значение индекса i. Процесс определения максимального элемента в массиве изображен при помощи блок-схемы на рис. 5.24. Соответствующий фрагмент программы имеет вид:

```
Max:=X[1]; Nmax:=1;
for i:=2 to n do
    if X[i]>Max then
    begin
        Max:=X[i];
        Nmax:=i;
   end;
write(' Max=',Max:1:3,' Nmax=',Nmax);
```

Алгоритм поиска минимального элемента в массиве будет отличаться от приведенного выше лишь тем, что в условном блоке и, соответственно, в конструкции `if` текста программы знак поменяется с > на <.

5.7 Сортировка элементов в массиве

Сортировка представляет собой процесс упорядочения элементов в массиве в порядке возрастания или убывания их значений. Например, массив `X` из `n` элементов будет отсортирован в порядке возрастания значений его элементов, если

`X[1] ≤ X[2] ≤… ≤ X[n],`

и в порядке убывания, если

`X[1] ≥ X[2] ≥… ≥ X[n].`

Многие алгоритмы сортировки основаны на том факте, что надо переставлять два элемента таким образом, чтобы после перестановки они были правильно расположены друг относительно друга. При сортировке по возрастанию после перестановки элемент с меньшим индексом должен быть не больше элемента с большим индексом[58]. Рассмотрим некоторые из алгоритмов.

5.7.1 Сортировка методом «пузырька»

Наиболее известным методом сортировки является сортировка массивов пузырьковым методом. Ее популярность объясняется запоминающимся названием[59] и простотой алгоритма. Сортировка методом пузырька основана на выполнении в цикле операций сравнения и при необходимости обмена соседних элементов. Рассмотрим алгоритм пузырьковой сортировки на примере *сортировки по возрастанию* более подробно.

Сравним первый элемент массива со вторым, если первый окажется больше второго, то поменяем их местами. Затем сравним второй с третьим, если второй больше третьего, то также поменяем их, далее сравниваем третий и четвертый, и если третий большего четвертого, меняем их местами. После трех этих сравнений самым большим элементом станет элемент с номером `4`. Если продолжить сравнения соседних элементов, сравнить четвертый с пятым, пятый с

58 При сортировке по убыванию после перестановки элемент с меньшим индексом должен быть не меньше элемента с большим индексом.

59 Название алгоритма происходит из-за подобия процессу движения пузырьков в резервуаре с водой, когда каждый пузырек находит свой собственный уровень.

шестым и т. д. до сравнения (n-1)–го и n–го элементов, то в результате этих действий самый большой элемент станет на последнее (n-е) место. Теперь повторим данный алгоритм сначала, с 1-го до n−1 элемента (последний, n-й элемент, рассматривать не будем, так как он уже занял свое место). После проведения данной операции самый большой элемент оставшейся части массива станет на свое (n−1)-е место. Так повторяем до тех пор, пока не упорядочим весь массив.

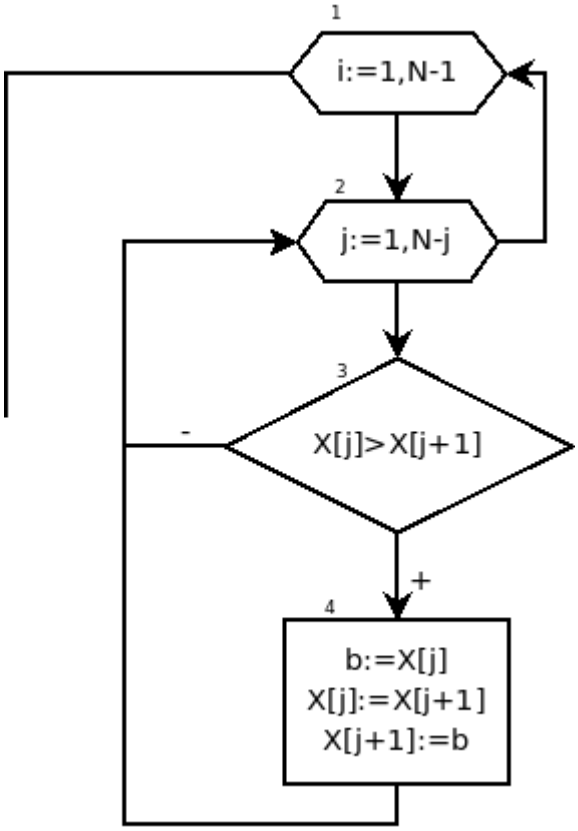

Рисунок 5.25: Алгоритм упорядочивания по возрастанию методом пузырька

В табл. 5.5 подробно показан процесс упорядочивания элементов в массиве.

Нетрудно заметить, что для преобразования массива, состоящего из n элементов, необходимо просмотреть его n−1 раз, каждый раз уменьшая диапазон просмотра на один элемент.

Блок–схема описанного алгоритма приведена на рис. 5.25.

Для обмена двух элементов в массиве (блок 4) используется буферная переменная b, в которой временно хранится значение элемента, подлежащего замене.

Таблица 5.5: Процесс упорядочивания элементов в массиве по возрастанию

Номер элемента	**1**	**2**	**3**	**4**	**5**
Исходный массив	7	3	5	4	2
Первый просмотр	3	5	4	2	7
Второй просмотр	3	4	2	5	7
Третий просмотр	3	2	4	5	7
Четвертый просмотр	2	3	4	5	7

Ниже приведен текст консольного приложения, предназначенного для упорядочивания массива по возрастанию методом пузырька.

На рис. 5.26 приведены результаты работы этой программы.

```
var  i,j,n: byte; b:real;
 X: array [1..100] of real;
begin
 writeln ('введите размер массива ');
 readln (n);
 for i:=1 to n do
 begin
     write('X[',i,']=');
     readln (X[i]);
 end;
 writeln ('массив X ');
 for i:=1 to n do
     write (x[i]:5:2,' ');
 writeln;
 for j:=1 to n-1 do
     for i:=1 to n-j do
         if X[i] > X[i+1] then
         { Если текущий элемент больше
           следующего, то }
         begin  { поменять их местами. }
             b:=X[i];   { Сохранить значение
                          текущего элемента. }
             X[i]:=X[i+1];{Заменить текущий
                          элемент следующим. }
             X[i+1]:=b; {Заменить следующий
                          элемент переменной b.}
         end;
 writeln('упорядоченный массив');
 for i:=1 to n do
     write (X[i]:5:2,' ');
 writeln;
end.
```

Для упорядочивания элементов в массиве по убыванию их значений необходимо при сравнении элементов массива заменить знак > на < (см. блок 3 на рис. 5.25).

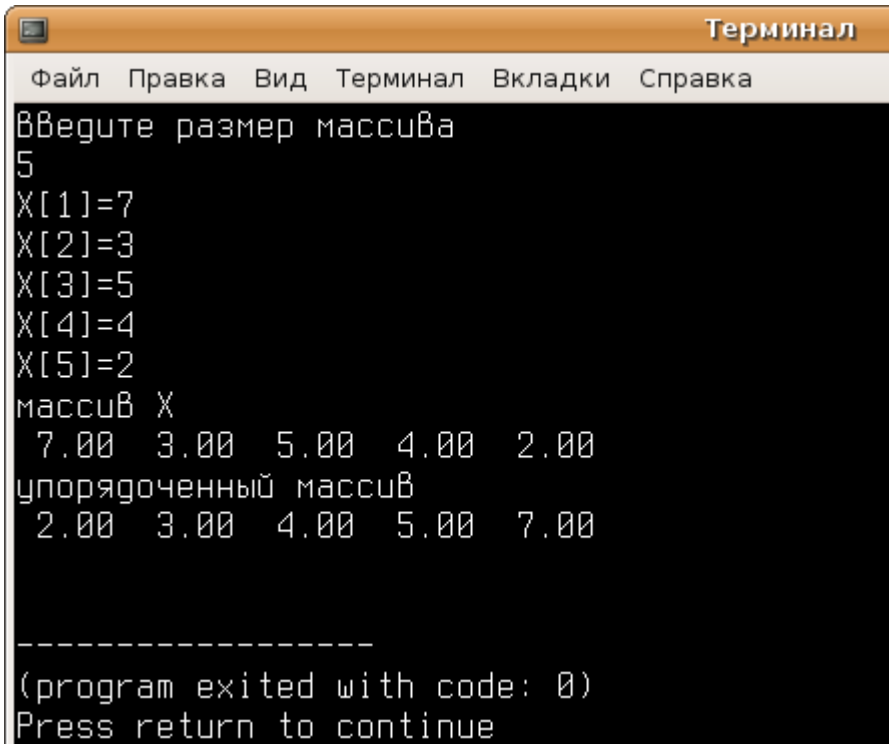

Рисунок 5.26: Результат программы упорядочивания массива по возрастанию

5.7.2 Сортировка выбором

Для сортировки элементов массива по возрастанию (убыванию) можно воспользоваться алгоритмом сортировки выбора максимального (минимального) элемента.

Алгоритм выбором приведен в виде блок-схемы на рис. 5.27. Найдем в массиве самый большой элемент (блоки 2–5) и поменяем его местами с последним элементом (блок 6). После этого максимальный элемент станет на свое место. Теперь надо повторять эти действия (блоки 2–6), уменьшив количество просматриваемых элементов на единицу (блок 7), до тех пор, пока количество рассматриваемых элементов не станет равным одному (блок 8). В связи с тем что мы на каждом шаге уменьшаем количество элементов на `1`, то, чтобы не потерять размер массива (`N`), необходимо в начале алгоритма переписать `N` в переменную `K` (блок 1) и уменьшать уже значение `K`.

При упорядочивании массива по убыванию необходимо перемещать минимальный элемент. Для чего в алгоритме (рис. 5.27) в блоке 4 достаточно знак > поменять на знак <.

Ниже приведен фрагмент программы упорядочения массива по возрастанию с использованием сортировки выбором максимального элемента.

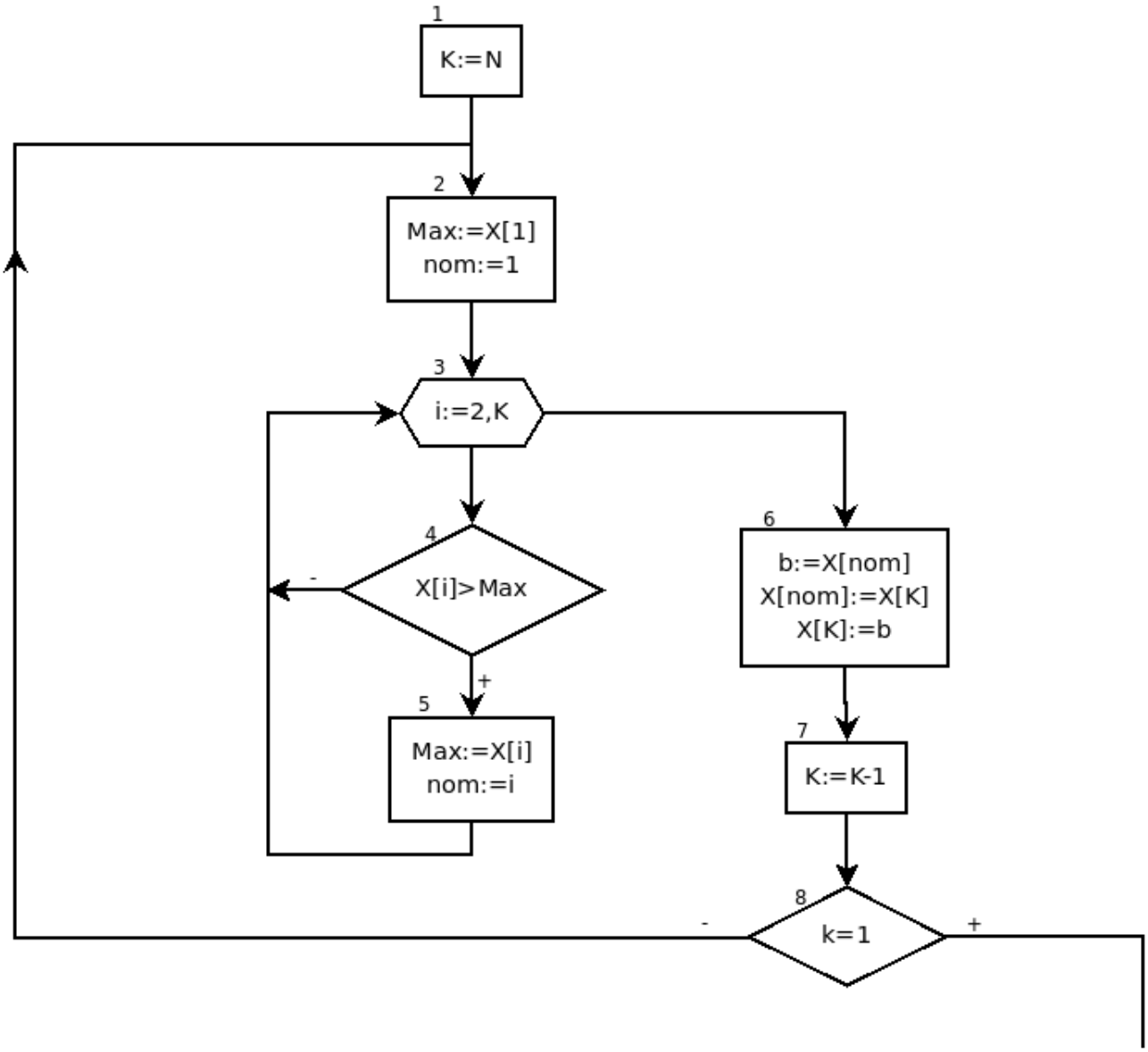

Рисунок 5.27: Сортировка массива по возрастанию выбором наибольшего элемента

```
//Сортировка выбором.
repeat
max:=x[1]; nom:=1;
for i:=2 to k do
 if max < X[i] then
 begin
     max:=X[i]; nom:=i;
 end;
 b:=x[nom]; x[nom]:=x[k];  x[k]:=b;
 k:=k-1;
until k=1;
```

5.8 Удаление элемента из массива

Знакомство с алгоритмом удаления элемента из массива начнем со следующей простой задачи. Необходимо удалить третий элемент из массива X, состоящего из 6 элементов. Алгоритм удаления третьего элемента заключается в том, что на место третьего элемента следует записать четвертый, на место четвертого — пятый, а на место пятого — шестой.

```
X[3]:=X[4]; X[4]:=X[5]; X[5]:=X[6];
```

Таким образом, все элементы с третьего по пятый надо переместить влево на один, на место i-го элемента нужно записать (i+1)-й. Блок-схема алгоритма представлена на рис. 5.28.

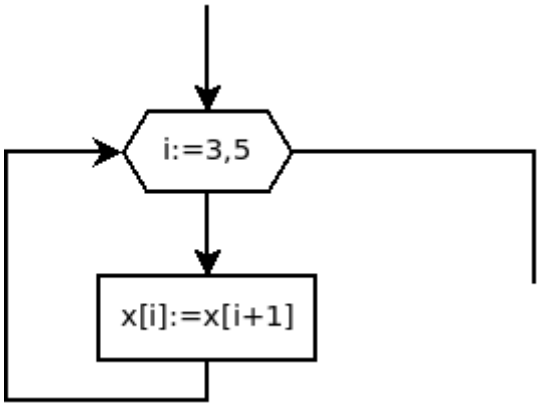

Рисунок 5.28: Алгоритм удаления 3-го элемента из массива

Теперь рассмотрим более общую задачу: необходимо удалить m-й элемент из массива X, состоящего из n элементов. Для этого достаточно записать (m+1)-й элемент на место элемента с номером m, (m+2)-й элемент – на место (m+1)-го и т.д., n-й элемент – на место (n−1)-го. Процесс удаления элемента из массива представлен на рис. 5.29.

Рисунок 5.29: Процесс удаления элемента из массива

Алгоритм удаления из массива X размерностью n элемента с номером m приведен на рис. 5.30.

После удаления элемента[60] из массива, изменится количество элементов в массиве (уменьшается на один) и у части элементов изменяется индекс. Если элемент удален, то на место него приходит следующий, передвигаться к которому (путем увеличения индекса на один) нет необходимости. Следующий элемент сам сдвинулся влево после удаления.

60 А фактически сдвига части массива на один элемент влево.

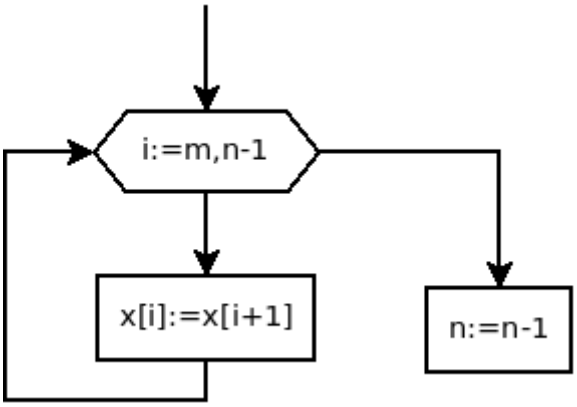

Рисунок 5.30: Алгоритм удаления m-го элемента из массива из n элементов

Если обрабатывается массив, в котором часть элементов удаляется, то после удаления элемента не надо переходить к следующему (при этом уменьшается количество элементов). В качестве примера рассмотрим следующую задачу.

ЗАДАЧА 5.1. Удалить из массива X(n) отрицательные элементы.

Алгоритм решения задачи довольно прост, перебираем все элементы массива, если элемент отрицателен, то удаляем его путем сдвига всех последующих на один влево. Единственное, о чем стоит помнить, что после удаления не надо переходить к следующему для последующей обработки, он сам сдвигается на место текущего. Блок-схема решения задачи 5.1 представлена на рис. 5.31. Ниже представлен текст программы с комментариями. Результаты работы программы представлены на рис. 5.32.

```
program upor_massiv;
var
 i,n,j:byte;
 X: array [1..100] of real;
 begin
 writeln ('введите размер массива ');
 readln (n);
 {Ввод массива.}
 for i:=1 to n do
 begin
     write('X[',i,']=');
     readln (X[i]);
 end;
writeln ('массив X ');
for i:=1 to n do
     write (x[i]:5:2,' ');
writeln;
```

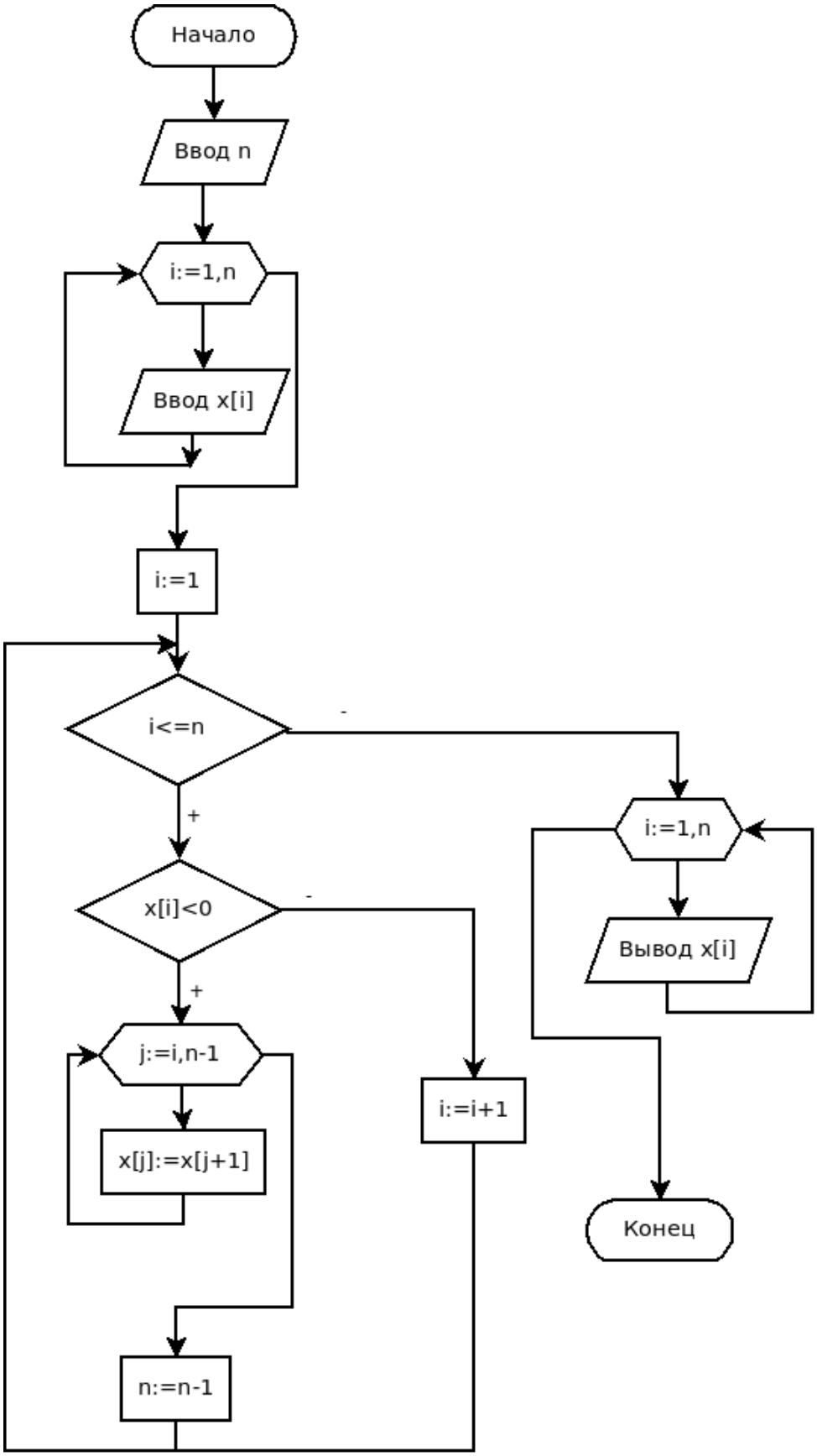

Рисунок 5.31: Блок-схема решения задачи 5.1

```
i:=1;
while(i<=n)do
{Если очередной элемент массива X[i]
отрицателен, то }
 if x[i]<0 then
 begin
{Удаляем элемент массива с номером i.}
     for j:=i to n-1 do x[j]:=x[j+1];
{Уменьшаем размер массива.}
{Не переходим к следующему элементу массива.}
     n:=n-1;
 end
else
{Если элемент не удалялся, то переходим к
следующему элементу массива.}
     i:=i+1;
writeln('Измененный массив:');
for i:=1 to n do write (X[i]:5:2,' ');
writeln;end.
```

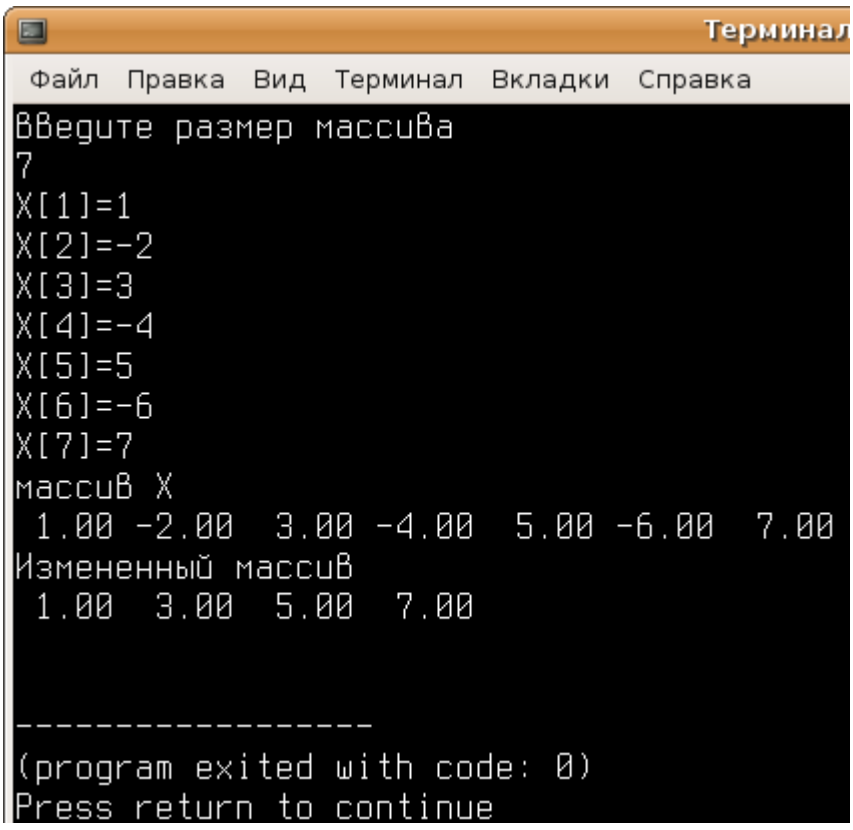

Рисунок 5.32: Результаты работы программы решения задачи 5.1

5.9 Вставка элемента в массив

Рассмотрим несложную задачу: вставить число `b` в массив `X(10)` между третьим и четвертым элементами.

Для решения этой задачи необходимо все элементы, начиная с четвертого, сдвинуть вправо на один элемент. А потом в четвертый элемент массива надо будет записать `b` (`X[4]:=b;`).

Но чтобы не потерять соседнее значение, сдвигать надо сначала десятый на один вправо, затем девятый, восьмой и т. д. до четвертого.

Блок-схема алгоритма вставки приведена на рис. 5.33.

В общем случае блок-схема вставки числа `b` в массив `X(N)` между элементами с номерами `m` и `m+1` представлена на рис. 5.34.

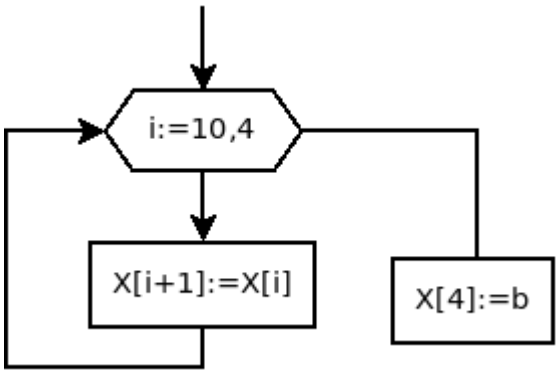

Рисунок 5.33: Вставка числа b между третьим и четвертым элементов массива X

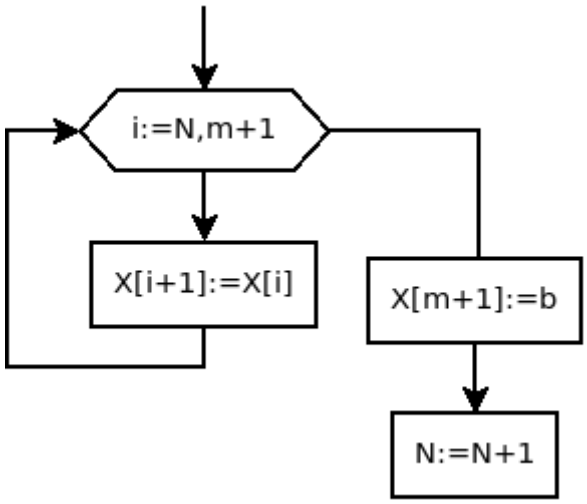

Рисунок 5.34: Вставка числа b в массив X

Ниже представлен фрагмент программы, реализующий этот алгоритм[61].

```
var i,n,m:byte; X: array [1..100] of real;
b:real;
begin
 writeln ('N='); readln (n);
 for i:=1 to n do
 begin
     write('X[',i,']='); readln (X[i]);
 end;
 writeln ('Массив X');
 for i:=1 to n do write (x[i]:5:2,' ');
```

61 При описании массива X необходимо предусмотреть достаточный размер для вставки одного элемента.

```
writeln;
writeln ('m='); readln (m);
writeln ('b='); readln(b);
for i:=n downto m+1 do
    x[i+1]:=x[i];
x[m+1]:=b;
n:=n+1;
writeln('Измененный массив');
for i:=1 to n do write (X[i]:5:2,' ');
writeln;
end.
```

5.10 Использование подпрограмм для работы с массивами

Рассмотрим, как можно передавать массивы в подпрограмму. Как известно (см. главу 4), чтобы объявить переменные в списке формальных параметров подпрограммы, необходимо указать их имена и типы. Однако типом любого параметра в списке может быть только стандартный или ранее объявленный тип. Поэтому для того чтобы передать в подпрограмму массив, необходимо вначале описать его тип[62], а затем объявлять процедуру:

```
type тип_массива= array [список_индексов] of тип;
procedure имя_поцедуры(имя_массива: тип_массива);
    …
```

Например:

```
type
vector=array [1..10] of byte;
matrica=array [1..3, 1..3] of real;
procedure proc(A:matrica;b:vector;var x:vector);
```

Понятно, что передача в подпрограмму строки вида

```
имя_переменной: string[длина_строки];
```

которая фактически является массивом[63], должна осуществляться аналогично:

```
type тип_строки= string [длина_строки];
procedure имя_поцедуры(имя_строки: тип_ строки);
    …
```

62 Тип данных массив, объявление массива см. в п.2.3.9.

63 Тип данных строка, объявление строки см. в п.2.3.9.

Например:

```
type
stroka_5=string[5];
stroka_10=string[10];
function fun(Str: stroka_5):stroka_10;
```

Массивы в подпрограмму можно передавать, используя понятие открытого массива. *Открытый массив* — это массив[64], при описании которого указывается тип составляющих его элементов, но не определяются границы изменения индексов:

```
имя_открытого_массива: array of array of … тип;
```

Например:

```
var
massiv_1: array of real;
massiv_2: array of array of char;
massiv_3: array of array of array of byte;
```

Распределение памяти и указание границ индексов по каждому измерению открытых массивов осуществляется в ходе выполнения программы с помощью функции `SetLength`:

```
SetLength(имя_массива, список_границ_индексов);
```

Для освобождения выделенной памяти нужно выполнить оператор:

```
имя_массива:=NIL;
```

Нижняя граница открытого массива (минимальное значение номера элемента) всегда равна нулю. *Верхняя граница* (максимальное значение номера элемента) возвращается стандартной функцией:

```
high(имя_массива)
```

Открытый массив можно использовать и при обычной обработке массивов в языке Free Pascal. Рассмотрим использование открытого массива на примере простейшей задачи нахождения суммы элементов массива.

```
var
//Описание открытого массива.
 x: array of real;
 s:real;
 i,n:integer;
Begin
write('n='); readln(n);
```

64 Тип данных массив, объявление массива, обращение к массиву см. в п.2.2.9.

```
//Выделяется память для размещения
//n вещественных значений:
SetLength(x,n);
for i:=0 to high(x) do read(x[i]);
s:=0;
for i:=0 to high(x) do s:=s+x[i];
writeln('сумма=',s:7:3);
x:=NIL;     //Освобождение памяти.
end.
```

Открытый массив может быть формальным параметром подпрограммы:

```
procedure   имя_поцедуры(имя_открытого_массива:
array of тип;);
```

Применение открытого массива в подпрограмме позволяет обрабатывать одномерные массивы произвольной длины:

```
//Процедура предназначена для вывода на экран
//сообщений о значениях элементов
//одномерного массива.
//Параметром подпрограммы является
//открытый массив целых чисел.
procedure outputArray(X:array of integer);
var i:byte;
begin
 //Элементы в открытом массиве
//пронумерованы от 0 до high(X).
 for i:=0 to high(X) do
 //Вывод сообщения:
//X[номер_элемента]=значение_элемента.
     writeln('X[',i,']=',X[i]);
end;
var
 A:array [1..10] of integer;
 C: array of integer;
 i:byte;
begin
 //Формирование одномерного массива A
//из 10 элементов.
 for i:=1 to 10 do A[i]:= 2*i+1;
```

```
 //Выделяется память для размещения
//3 целочисленных значений:
 SetLengTh(C,3);
 //Формирование одномерного массива C
//из 3 элементов.
 for i:=0 to 2 do C[i]:= 1-2*i;
 //Обращение к подпрограмме.
 outputArray(A);
 outputArray(C);
end.
```

Без использования открытых массивов процедуру `outputArray` пришлось бы записать так:

```
//Описание типа: массив целых чисел,
//пронумерованных от 0 до10.
type massiv=array [0..10] of integer;
//Процедура предназначена для вывода сообщений
//о значениях элементов одномерного массива.
procedure outputArray(X:massiv;nN,nK:byte);
//Параметры подпрограммы:
//1. Массив целых чисел X.
//2. Нижняя граница индекса nN.
//3. Верхняя граница индекса nK.
var i:byte;
begin
 //Элементы массива нумеруются от nN до nK.
 for i:=nN to nK do
     writeln('X[',i,']=',X[i]);
end;
```

5.11 Использование указателей для работы с динамическими массивами

Все объявленные в программе статические переменные, которые мы рассматривали до этого момента, размещаются в одной непрерывной области оперативной памяти, которая называется сегментом данных. Для работы с массивами большой размерности можно воспользоваться так называемой динамической памятью, которая выделяется программе после запуска программы на выполнение. Размер динамической памяти можно варьировать в широких пределах. По умолча-

нию этот размер определяется всей доступной памятью ПК.

Динамическое размещение данных осуществляется компилятором непосредственно в процессе выполнения программы. При динамическом размещении заранее неизвестно количество размещаемых данных. Кроме того, к ним нельзя обращаться по именам, как к статическим переменным.

Оперативная память ПК представляет собой совокупность элементарных ячеек для хранения информации – байтов, каждый из которых имеет собственный номер. Эти номера называются адресами, они позволяют обращаться к любому байту памяти.

5.11.1 Работа с динамическими переменными и указателями

Free Pascal имеет гибкое средство управления памятью – указатели.

Указатель – переменная, которая в качестве своего значения содержит адрес байта памяти. Указатель занимает 4 байта.

Как правило, указатель связывается с некоторым типом данных. В таком случае он называется типизированным. Для его объявления используется знак ^, который помещается перед соответствующим типом, например:

```
type massiv=array [1..2500] of real;
var a:^integer; b,c:^real; d:^massiv;
```

В языке Free Pascal можно объявлять указатель, не связывая его с конкретным типом данных. Для этого служит стандартный тип `pointer`, например:

```
var p,c,h: pointer;
```

Указатели такого рода будем называть *нетипизированными*. Поскольку нетипизированные указатели не связаны с конкретным типом, с их помощью удобно динамически размещать данные, структура и тип которых меняются в ходе работы программы.

Значениями указателей являются адреса переменных памяти, поэтому следовало ожидать, что значение одного из них можно передавать другому. На самом деле это не совсем так. Эта операция проводится только среди указателей, связанных с одними и теми же типами данных.

Например:

```
Var
p1, p2:^integer; p3:^real; pp:pointer;
```

В этом случае присваивание `p1:=p2;` допустимо, в то время как `p1:=p3;` запрещено, поскольку `p1` и `p3` указывают на разные типы данных. Это ограничение не распространяется на нетипизированные указатели, поэтому можно записать `pp:=p3; p1:=pp;` и достичь необходимого результата.

Вся динамическая память во Free Pascal представляет собой сплошной массив байтов, называемый кучей. Физически куча располагается за областью памяти, которую занимает тело программы.

Начало кучи хранится в стандартной переменной `heaporg`, конец – в переменной `heapend`. Текущая граница незанятой динамической памяти хранится в указателе `heapprt`.

Память под любую динамическую переменную выделяется процедурой `new`, параметром обращения к которой является типизированный указатель. В результате обращения последний принимает значение, соответствующее динамическому адресу, начиная с которого можно разместить данные, например:

```
var
i,j:^integer;
r:^real;
begin
new(i);
new(R);
new(j)
```

В результате выполнения первого оператора указатель `i` принимает значение, которое перед этим имел указатель кучи `heapprt`. Сам `heapprt` увеличивает свое значение на 4, так как длина внутреннего представления типа `integer`, связанного с указателем `i`, составляет 4 байта. Оператор `new(r)` вызывает еще одно смещение указателя `heapprt`, но уже на 8 байтов, потому что такова длина внутреннего представления типа `real`. Аналогичная процедура применяется и для переменной любого другого типа. После того как указатель стал определять конкретный физический байт памяти, по этому адресу можно разместить любое значение соответствующего типа, для чего сразу за указателем без каких-либо пробелов ставится значок `^`, например:

```
i^:=4+3;
j^:=17;
r^:=2 * pi;
```

Таким образом, значение, на которое указывает указатель, то есть собственно данные, размещенные в куче, обозначаются значком `^`. Значок `^` ставится сразу за указателем. Если после указателя значок `^` отсутствует, то имеется в виду адрес, по которому размещаются данные. Динамически размещенные данные (но не их адрес!) можно использовать для констант и переменных соответствующего типа в любом месте, где это допустимо, например:

```
r^:=sqr(r^)+sin(r^+i^)-2.3
```

Невозможен оператор

```
r:=sqr(r^)+i^;
```

так как указателю r нельзя присвоить значение вещественного типа.

Точно так же недопустим оператор

```
r^:=sqr(r);
```

поскольку значением указателя `r` является адрес, и его (в отличие от того значения, которое размещено по данному адресу) нельзя возводить в квадрат. Ошибочным будет и присваивание `r^:=i`, так как вещественным данным, на которые указывает `r^`, нельзя давать значение указателя (адрес). Динамическую память можно не только забирать из кучи, но и возвращать обратно. Для этого используется процедура `dispose(p)`, где `p` – указатель, который не изменяет значение указателя, а лишь возвращает в кучу память, ранее связанную с указателем.

При работе с указателями и динамической памятью необходимо самостоятельно следить за правильностью использования процедур `new`, `dispose` и работы с адресами и динамическими переменными, так как транслятор эти ошибки не контролирует. Ошибки этого класса могут привести к зависанию компьютера, а то и к более серьезным последствиям!

Другая возможность состоит в освобождении целого фрагмента кучи. С этой целью перед началом выделения динамической памяти текущее значение указателя `heapptr` запоминается в переменной-указателе с помощью процедуры `mark`. Теперь можно в любой момент освободить фрагмент кучи, начиная с того адреса, который запомнила процедура `mark`, и до конца динамической памяти. Для этого используется процедура `release`.

Процедура `mark` запоминает текущее указание кучи `heapptr`

(обращение `mark(ptr)`, где `ptr` – указатель любого типа, в котором будет возвращено текущее значение `heapptr`). Процедура `release(ptr)`, где `ptr` – указатель любого типа, освобождает участок кучи от адреса, хранящегося в указателе до конца кучи.

5.11.2 Работа с динамическими массивами с помощью процедур getmem и freemem

Для работы с указателями любого типа используются процедуры `getmem`, `freemem`. Процедура `getmem(p,size)`, где `p` – указатель, `size` – размер в байтах выделяемого фрагмента динамической памяти (`size` типа `word`), резервирует за указателем фрагмент динамической памяти требуемого размера.

Процедура `freemem(p,size)`, где `p` – указатель, `size` – размер в байтах освобождаемого фрагмента динамической памяти (`size` типа `word`), возвращает в кучу фрагмент динамической памяти, который был зарезервирован за указателем. При применении процедуры к уже освобожденному участку памяти возникает ошибка.

После рассмотрения основных принципов и процедур работы с указателями возникает вопрос: а зачем это нужно? В основном для того, чтобы работать с так называемыми динамическими массивами. Последние представляют собой массивы переменной длины, память под которые может выделяться (и изменяться) в процессе выполнения программы, как при каждом новом запуске программы, так и в разных её частях. Обращение к `i`-му элементу динамического массива `x` имеет вид `x^[i]`.

Рассмотрим процесс функционирования динамических массивов на примере решения следующей задачи.

ЗАДАЧА 5.2. Найти максимальный и минимальный элементы массива X(N).

Вспомним решение задачи традиционным способом.

```
Program din_mas1;
Var
 x:array [1..150] of real;
i,n:integer; max,min:real;
begin
 writeln('введите размер массива');
 readln (n);
for i:=1 to N do
```

```
begin
 write('x[',i,']='); readln(x[i]);
end;
max:=x[1]; min:=x[1];
for i:=2 to N do
begin
 if x[i] > max then max:=x[i];
 if x[i] < min  then  min:=x[i];
end;
 writeln('максимум=',max:1:4);
 writeln('минимум=',min:1:4);
end.
```

Теперь рассмотрим процесс решения задачи с использованием указателей. Распределение памяти проводим с помощью процедур `new-dispose` (программа din_mas2) или `getmem-freemem` (программа din_mas2).

```
 type massiw= array[1..150]of real;
 var
 x:^massiw;
 i,n:integer;
 max,min:real;
begin
{Выделяем память под динамический
массив из 150 вещественных чисел.}
 new(x);
 writeln('Введите размер массива');
 readln(n);
for i:=1 to N do
begin
 write('x(',i,')=');
 readln(x^[i]);
end;
max:=x^[1];min:=x^[1];
for i:=2 to N do
begin
 if x^[i] > max then max:=x^[i];
 if x^[i] < min then min:=x^[i];
end;
```

```
writeln('максимум=',max:1:4,
                    ' минимум=',min:1:4);
{ Освобождаем память. }
dispose(x);
end.
type
 massiw=array[1..150]of real;
var
 x:^massiw;
 i,n:integer;max,min:real;
begin
writeln('Введите размер массива');
readln(n);
{ Выделяем память под n элементов массива. }
getmem(x,n*sizeof(real));
for i:=1 to N do
begin
 write('x(',i,')=');
 readln(x^[i]);
end;
max:=x^[1];min:=x^[1];
for i:=2 to N do
begin
 if x^[i] > max then max:=x^[i];
 if x^[i] < min then min:=x^[i];
end;
writeln('максимум=',max:1:4,
                         ' минимум=',min:1:4);
{ Освобождаем память. }
freemem(x,n*sizeof(real));
end.
```

При работе с динамическими переменными необходимо соблюдать следующий порядок работы:

1. Описать указатели.
2. Выделить память под массив (функции `new` или `getmem`).
3. Обработать динамический массив.
4. Освободить память (функции `dispose` или `freemem`).

5.12 Примеры программ

ЗАДАЧА 5.3. Дан массив A, состоящий из k целых чисел. Записать все отрицательные элементы массива A в массив B.

Решение задачи заключается в следующем. Последовательно перебираются элементы массива A. Если среди них находятся отрицательные, то они записываются в массив B. На рисунке 5.35 видно, что

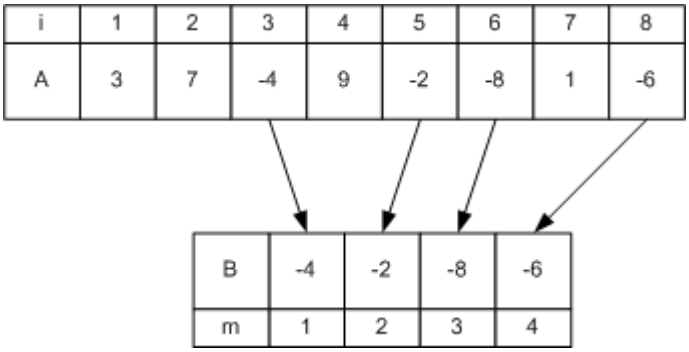

Рисунок 5.35: Процесс формирования массива B из отрицательных элементов массива A

первый отрицательный элемент хранится в массиве A под номером три, второй и третий — под номерами пять и шесть соответственно, а четвертый — под номером восемь. В массиве B этим элементам присваиваются номера *один*, *два*, *три* и *четыре*.

Поэтому для их формирования необходимо определить дополнительную переменную. В блок–схеме, приведенной на рисунке 5.36, роль такой переменной выполняет переменная m. В процессе формирования массива B в переменной m хранится номер сформированного элемента. Вначале в массиве B нет ни одного элемента, и поэтому m=0 (блок 2). В цикле (блок 3) последовательно перебираем все элементы A, если очередной элемент массива A отрицателен (блок 5), то переменная m увеличивается на единицу, а значение элемента массива A записывается в массив B под номером m (блок 6). В блоке 7 проверяем, были ли в массиве A отрицательные элементы и сформирован ли массив B. В результате этого алгоритма будет сформирован массив B отрицательных чисел, состоящий из k чисел.

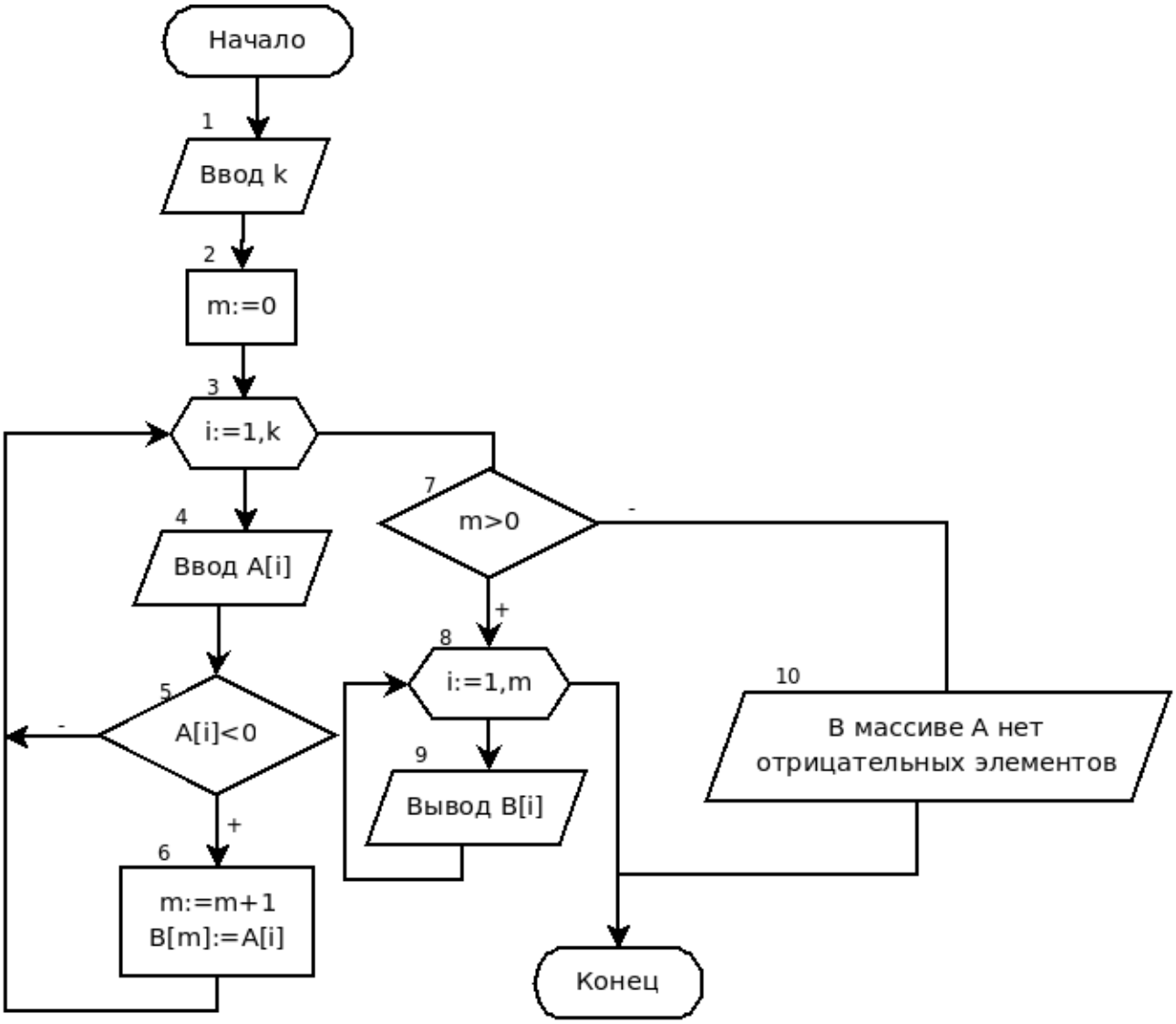

Рисунок 5.36: Блок-схема формирования массива B из отрицательных элементов массива A

Приведенная ниже программа реализует описанный алгоритм.

```
var a,b:array [1..200] of word; k,m,i:byte;
begin
 write('введите размерность массива к=');
 readln(k);
 m:=0;
 for i:=1 to k do
 begin
     write('A[',i,']=');readln(A[i]);
 if A[i]<0 then
 begin
     m:=m+1;
     B[m]:=A[i];
```

```
 end;
 end;
 if m>0 then
     for i:=1 to m do   write(B[i],' ')
 else
write('В массиве нет отрицательных элементов');
end.
```

ЗАДАЧА 5.4. Задан массив y из n целых чисел. Сформировать массив z таким образом, чтобы в начале шли отрицательные элементы массива y, затем — положительные и, наконец, — нулевые.

Алгоритм решения этой задачи основывается на алгоритме перезаписи элементов, удовлетворяющих какому-либо условию из одного массива в другой, который был подробно рассмотрен в предыдущей задаче. Блок-схема решения задачи 5.4 представлена на рис. 5.37.

Текст программы с комментариями приведен ниже.

```
var i,k,n: integer;
 y,z:array [1..50] of integer;
begin
 writeln('Введите n<=50'); readln (n);
for i:=1 to n do //Ввод массива y.
 begin
     write('y[',i,']='); readln(y[i]);
 end;
 k:=0;
//Перезапись отрицательных чисел
//из массива y в массив z.
 for i:=1 to n do
     if y[i] < 0 then
     begin
         k:=k+1; z[k]:=y[i];
     end;
//Перезапись положительных чисел
//из массива y в массив z.
 for i:=1 to n do
     if y[i] >0 then
     begin k:=k+1;
             z[k]:=y[i]
     end;
```

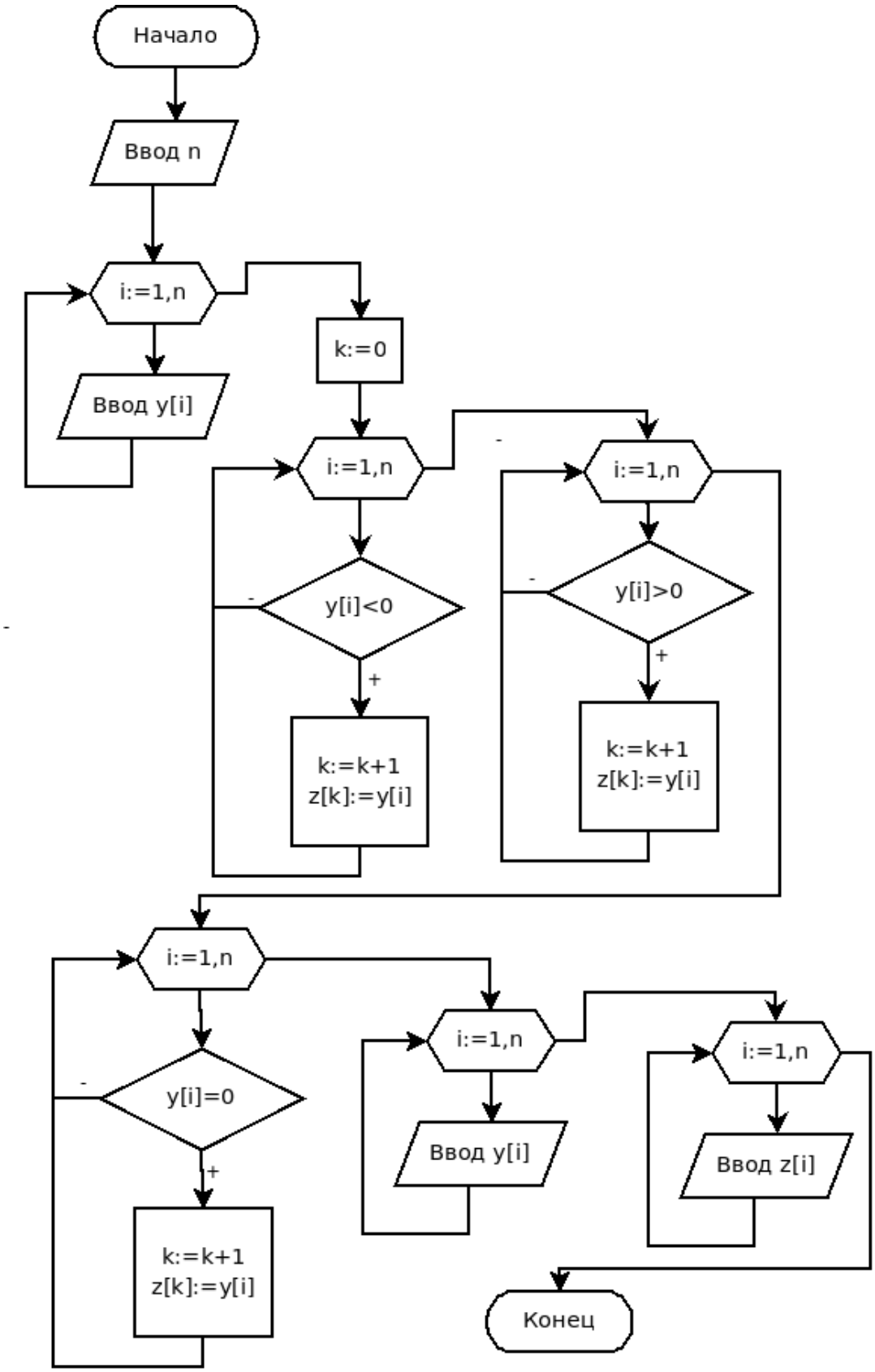

Рисунок 5.37: Блок схема решения задачи 5.4

```
//Перезапись нулевых чисел из массива
//y в массив z.
 for i:=1 to n do
     if y[i]=0 then
     begin
         k:=k+1;
         z[k]:= y[i];
     end;
// Вывод массива Y.
 writeln ('Массив Y:');
 for i:=1 to n do
         write(y[i],' ');
 writeln;
// Вывод массива Z.
 writeln ('Массив Z:');
 for i:=1 to n do
         write (z[i],' ');
 writeln;
end.
```

ЗАДАЧА 5.5. Переписать элементы в массиве X в обратном порядке.

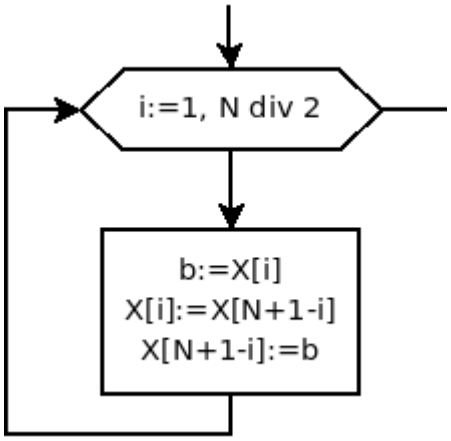

Рисунок 5.38: Фрагмент блок–схемы к задаче 5.5

Алгоритм решения задачи состоит в следующем: меняем местами 1-й и n-й элементы, затем 2-й и n−1-й элементы, и так до середины массива, элемент с номером i следует обменять с элементом n+1−i. Блок-схема обмена элементов в массиве представлена на рис. 5.38.

Ниже приведен текст консольного приложения задачи 5.5.

```
type
 massiv=array [1..100] of real;
var
 x:massiv;
 i,n:integer;
```

```
 b:real;
begin
//Ввод размера массива.
writeln ('Введите размер массива'); readln(n);
//Ввод массива.
for i:=1 to n do
begin
     write ('x[',i,']=');
     readln(x[i]);
end;
//Перебирается первая половина массива,
//и меняются элементы местами 1-й с n-м,
//2-й с (n-1), … i-й с (n+1-i)-м элементом.
for i:=1 to n div 2  do
begin
     b:=x[n+1-i];
     x[n+1-i]:=x[i];
     x[i]:=b;
end;
//Ввод преобразованного массива.
writeln('Преобразованный массив');
for i:=1 to n do
     write(x[i]:1:2,' ');
end.
```

ЗАДАЧА 5.6. Удалить из массива X, состоящего из n элементов, первые четыре нулевых элемента.

Вначале количество нулевых элементов равно нулю (`k=0`). Последовательно перебираем все элементы массива. Если встречается нулевой элемент, то количество нулевых элементов увеличиваем на `1` (`k:=k+1`). Если количество нулевых элементов меньше или равно `4`, то удаляем очередной нулевой элемент. Если встречаем пятый нулевой элемент (`k>4`), то аварийно выходим из цикла (дальнейшая обработка массива бесполезна).

Блок-схема представлена на рис. 5.39.

Текст программы с комментариями приведен ниже.

```
const n=20;
var
```

```
    X:array [1..n] of byte;
    k,i,j:integer;
begin
 for i:=1 to n do
    readln(X[i]);
k:=0;   {Количество нулевых элементов.}
j:=1;   {Номер элемента в массиве X.}
while j<=n do  {Пока не конец массива.}
begin
{Если нашли нулевой элемент, то}
    if x[j]=0 then
    begin
        k:=k+1; {посчитать его номер}
        if k>4 then
                break
    {Если k превышает 4, то выйти из цикла.}
    {Иначе удаляем j-й элемент из массива.}
        else
            for i:=j to n-k do
                X[i]:=X[i+1];
    end
{ Если встретился ненулевой элемент,
то просто переходим к следующему.}
else
    j:=j+1; {Если элемент ненулевой }
end;
{Вывод на печать измененного массива.}
{Количество элементов в массиве
 уменьшилось на k.}
for i:=1 to n-k do
        write(X[i],' ');
end.
```

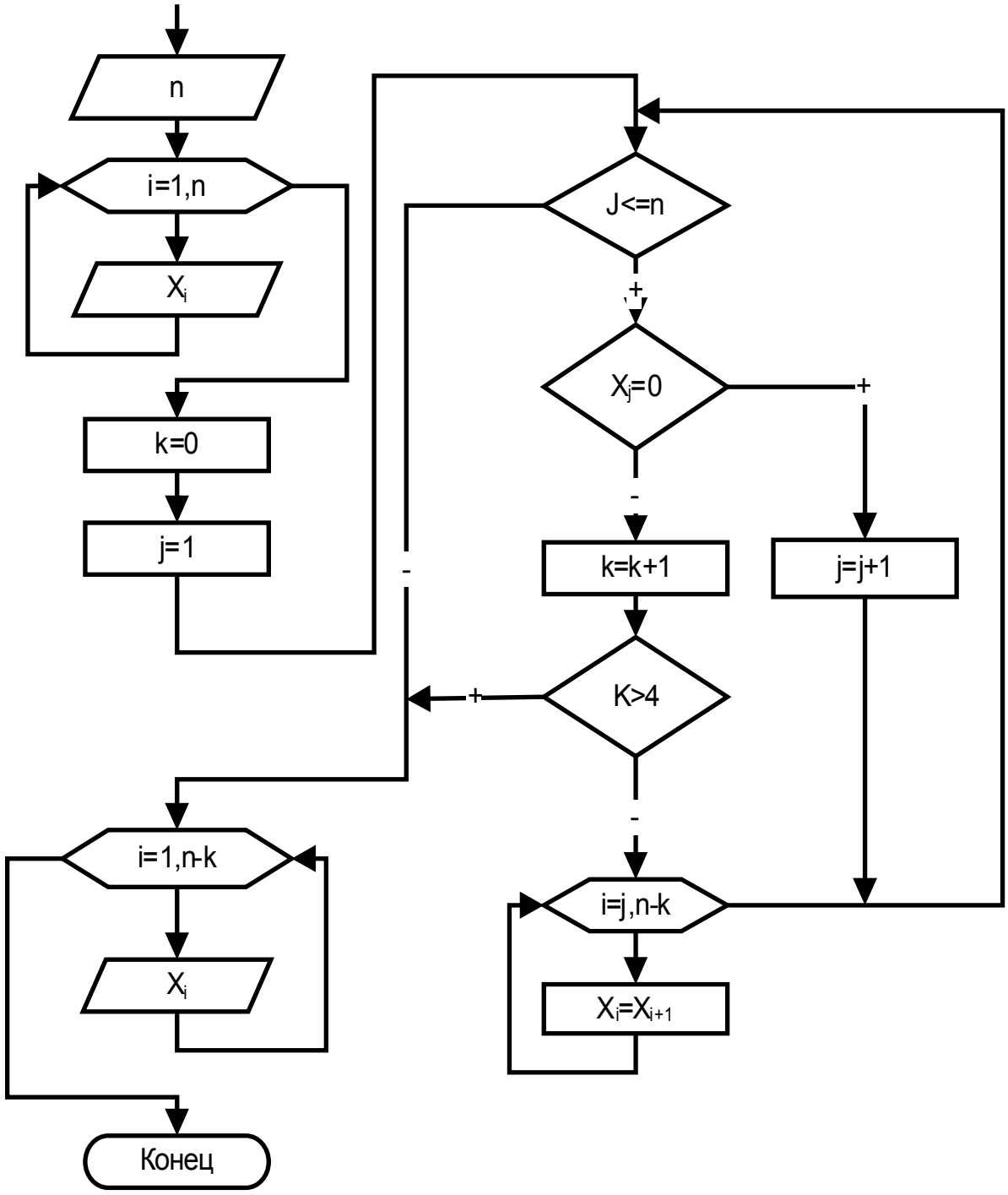

Рисунок 5.39: Алгоритм решения задачи 5.6

ЗАДАЧА 5.7. Найти сумму простых чисел целочисленного массива C(N).

Идея алгоритма состоит в следующем. Сначала сумма равна 0. Последовательно перебираем все элементы, если очередной элемент простой, то добавляем его к сумме. Для проверки, является ли число простым, напишем функцию `prostoe`, которая проверяет, является ли число простым.

Блок-схема этой функции представлена на рис.5.40.

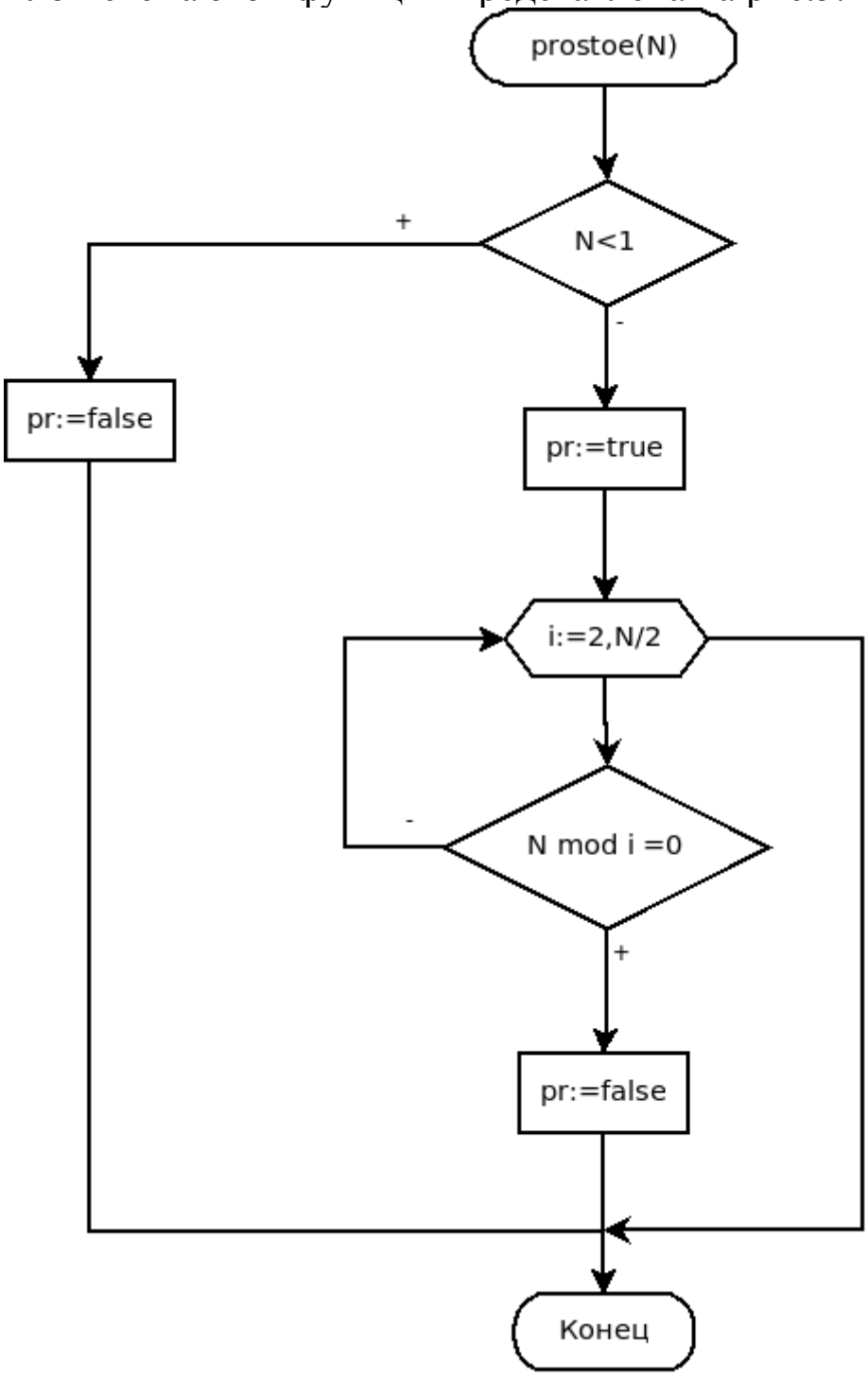

Рисунок 5.40: Блок-схема функции prostoe

Заголовок функции `Prostoe` имеет вид:

```
function Prostoe(N:integer):boolean;
```

Функция возвращает значение `true`, если число `N` является простым. В противном случае результатом функции является значение `false`. Блок–схема решения задачи 5.7 изображена на рис. 5.41.

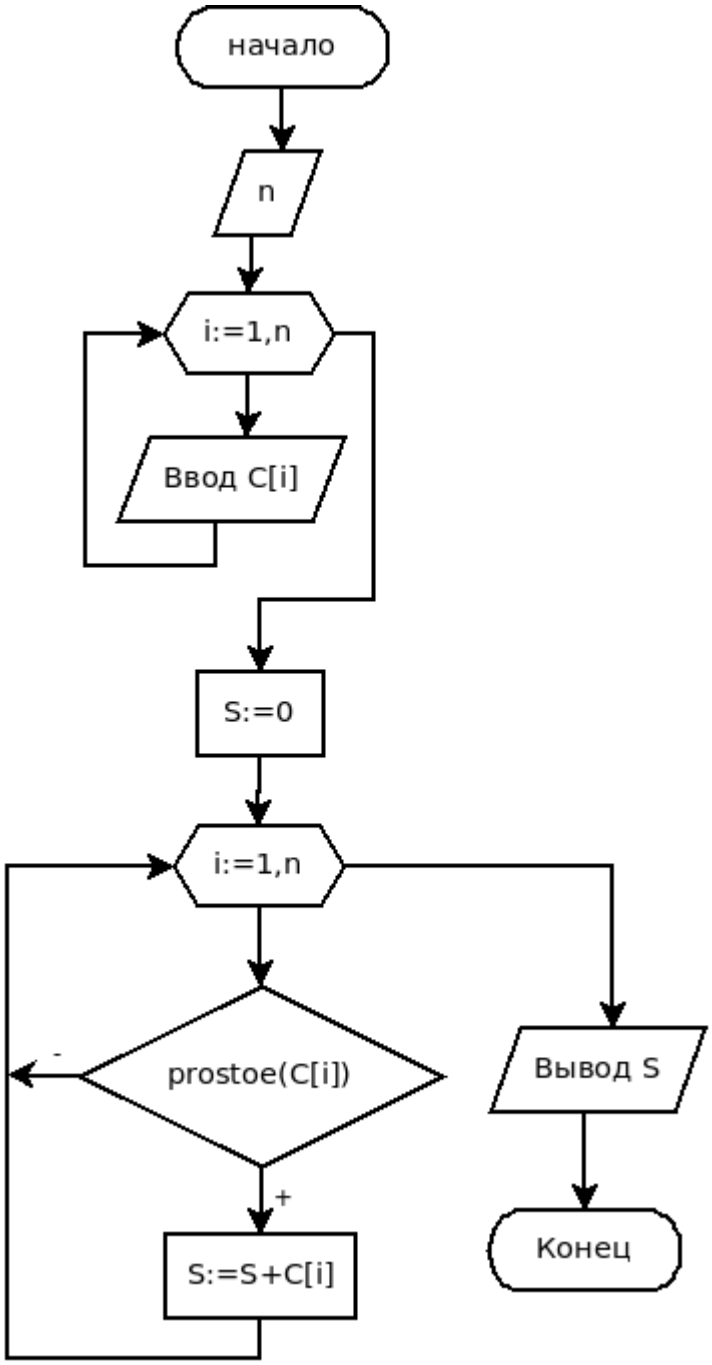

Рисунок 5.41: Блок-схема решения задачи 5.7

Далее приведен текст программы, реализующей этот алгоритм, с комментариями.

```
function prostoe(N:integer):boolean;
var i:integer;  pr:boolean;
begin
if N<1 then
     pr:=false
else
```

```
begin
{ Предположим, что текущее число простое. }
pr:=true;
for i:=2 to N div 2 do
 if (N mod i = 0) then
{ Если найдется хотя бы один делитель,
кроме единицы и самого числа, то }
 begin
{ изменить значение логической переменной }
    pr:=false;
{ и досрочно выйти из цикла. }
    break;
 end; end;
prostoe:=pr;
end;
{Основная программа}
var
 c:array [1..50] of word;
 i,n:byte; S:word;
begin
 write('Введите размерность массива n=');
 readln(n);
 for i:=1 to n do
 begin
    write('Введите ',i,'-й элемент массива ');
    readln(C[i]);
 end;
 S:=0;
 for i:=1 to n do
 { Если число простое, то накапливать сумму. }
 if prostoe(C[i]) then
    S:=S+C[i];
 {Вывод найденной суммы.}
 writeln('Сумма простых чисел массива S=',S);
end.
```

ЗАДАЧА 5.8. Определить, есть ли в заданном массиве серии элементов, состоящих из знакочередующихся чисел (рис. 5.42). Если есть, то вывести на экран количество таких серий.

Рисунок 5.42: Массив с тремя сериями знакочередующихся элементов

Идея алгоритма состоит в следующем. Последовательно в цикле от `1` до `n-1` сравниваются соседние элементы (первый и второй, второй и третий, … , предпоследний и последний). Если соседние элементы имеют разные знаки (их произведение отрицательно), то количество элементов в серии (переменная `k`) увеличиваем на один[65]. Если произведение соседних элементов в серии не отрицательно, то возможны два варианта: либо сейчас оборвалась серия из знакочередующихся элементов, либо очередные два элемента одного знака. Выбор из этих двух вариантов можно осуществить, сравнив переменную `k` с единицей. Если `k>1`, сейчас оборвалась серия из знакочередующихся элементов, и количество таких серий (`kol`) надо увеличить на 1.

После выхода из цикла проверяем, не было ли в конце массива серии из знакочередующихся элементов. Если такая серия был (`k>1`), то количество серий (`kol`) опять увеличиваем на 1. В завершение выводим количество серий из знакочередующихся элементов — переменную `kol`. Блок-схема решения задачи 5.8 представлена на рис. 5.43. Ниже приведен текст консольного приложения на языке Free Pascal.

```
var
 x:array[1..50] of real;
 n,i,k,kol:integer;
begin
 write('n=');
 readln(n);
 for i:=1 to n do
     read(x[i]);
{Так как минимальная серия состоит
из двух элементов, }
{ k присвоим значение 1. }
     k:=1; { Длина серии. }
```

65 Количество элементов в серии изначально равно нулю, так как после встречи в массиве первой пары знакочередующихся элементов в серии станет сразу два элемента, а все последующие идущие подряд элементы разного знака в серии будут увеличивать длину серии на один.

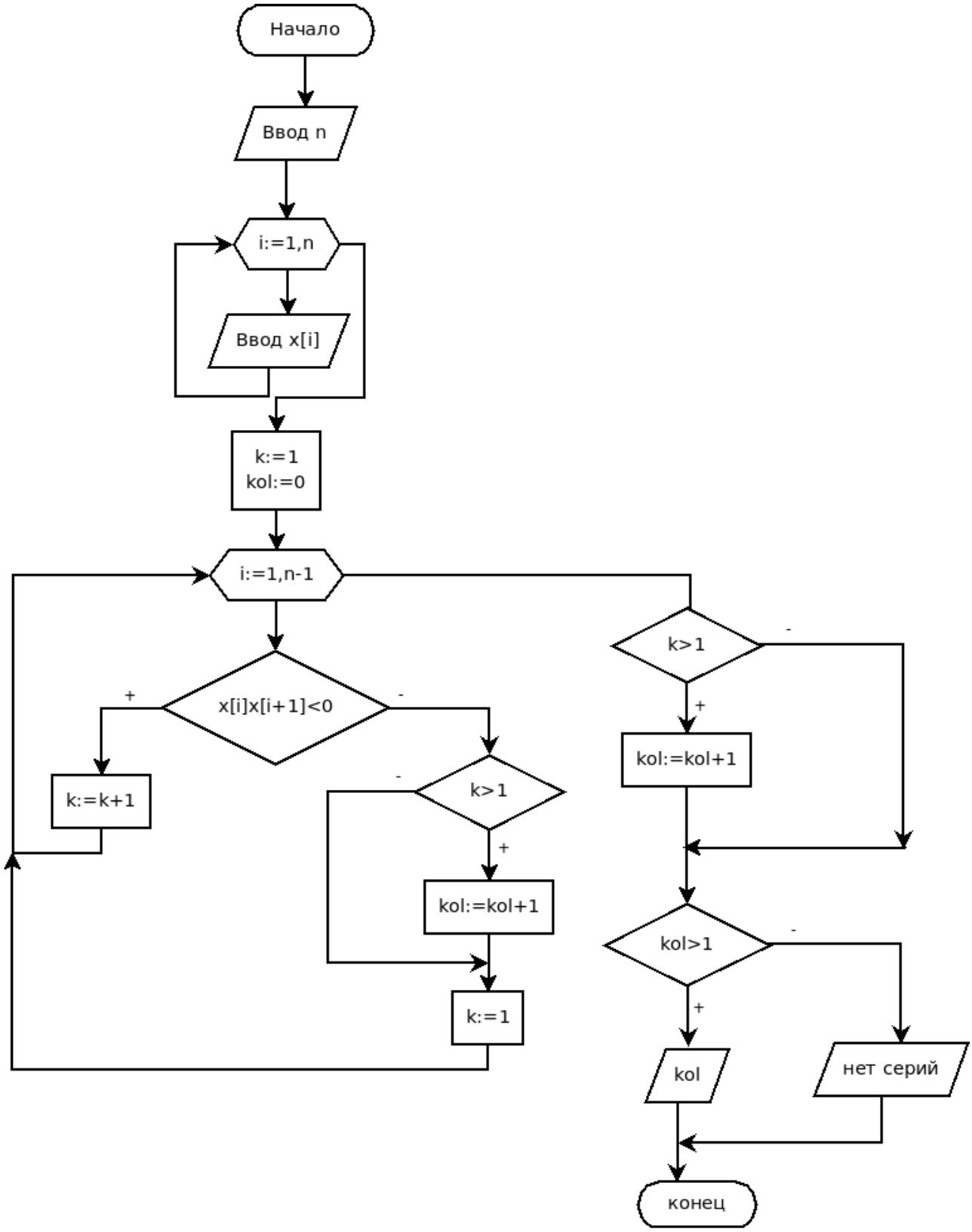

Рисунок 5.43: Блок–схема решения задачи 5.8

```
    kol:=0; { Количество серий в массиве. }
 for i:=1 to n-1 do
{ Если при умножении двух соседних элементов
```

```
 результат – отрицательное число,
то элементы имеют разный знак. }
     if x[i]*x[i+1]<0 then
     k:=k+1 { Показатель продолжения серии. }
     else
     begin
{ Если серия разорвалась, то увеличить счетчик
  количества серий. }
          if k>1 then
              kol:=kol+1;
{ Подготовить показатель продолжения серии }
{ к возможному появлению следующей серии. }
          k:=1;
     end;
{Проверка, не было ли серии в конце массива. }
 if k>1 then
{ Если да, увеличить счетчик еще на единицу. }
     kol:=kol+1;
     if kol>0 then
         write('Количество знакочередующихся
                                    серий=',kol)
     else
         write('Знакочередующихся серий нет')
end.
```

ЗАДАЧА 5.9. В заданном массиве найти самую длинную серию элементов, состоящую из единиц.

Для максимальной серии будем хранить ее длину (`max`), номер последнего элемента (`kon_max`).

Эта задача похожа на предыдущую, отличие заключается в том, что надо фиксировать не только тот факт, что серия кончилась, но и саму серию. Серия может характеризоваться двумя из трех параметров: первый элемент серии, последний элемент серии, длина серии. В связи с тем что мы фиксируем серию в момент ее окончания, в качестве параметров серии будем использовать последний элемент серии (`kon`) и ее длину (`k`).

Алгоритм решения этой задачи следующий. Вначале количество серий (`kol`) и ее длина (`k`) равны нулю. Перебираем последовательно

все элементы. Если текущий элемент равен 1, то количество элементов в серии[66] увеличиваем на 1. Если текущий элемент не равен 1, то возможны два варианта: либо сейчас оборвалась серия из единиц, либо встретился очереной не равный единице элемент. Выбор из этих двух вариантов можно осуществить, сравнив переменную k с единицей. Если k>1, сейчас оборвалась серия из единиц, и количество таких серий (kol) надо увеличить на 1, зафиксировать конец серии (kon:=i-1), длину серии (dlina:=k). После этого необходимо проверить, какая по счету серия. Если это первая серия (kol=1), то объявляем ее максимальной, в переменную max записываем длину текущей серии k, в переменную kon_max — kon (последний элемент текущей серии). Если это не первая серия (kol>1), то длину текущей серии (k) сравниваем с длиной серии максимальной длины (max). И если k>max, то текущую серию объявляем серией максимальной длины (max:=k; kon_max:=kon;). Если встретился не равный единице элемент, надо количество элементов в серии положить равным нулю (k:=0).

После выхода из цикла надо также проверить, не было ли в конце серии, состоящей из единиц. Если серия была в конце, то следует обработать ее так же как и серию, которая встретилась в цикле.

Блок-схема решения задачи приведена на рис. 5.44.

Ниже приведен текст консольного приложения решения задачи.

```
var x:array[1..50] of integer;
n,i,k,kol,kon,max,kon_max,dlina:integer;
begin
{Ввод размера массива.}
write('n=');
readln(n);
{Ввод массива}
writeln('Массив X');
for i:=1 to n do
read(x[i]);
{Начальное присваивание длины серии
и количества серий}
k:=0; { Длина серии. }
kol:=0; { Количество серий в массиве. }
```

66 Количество подряд идущих единиц.

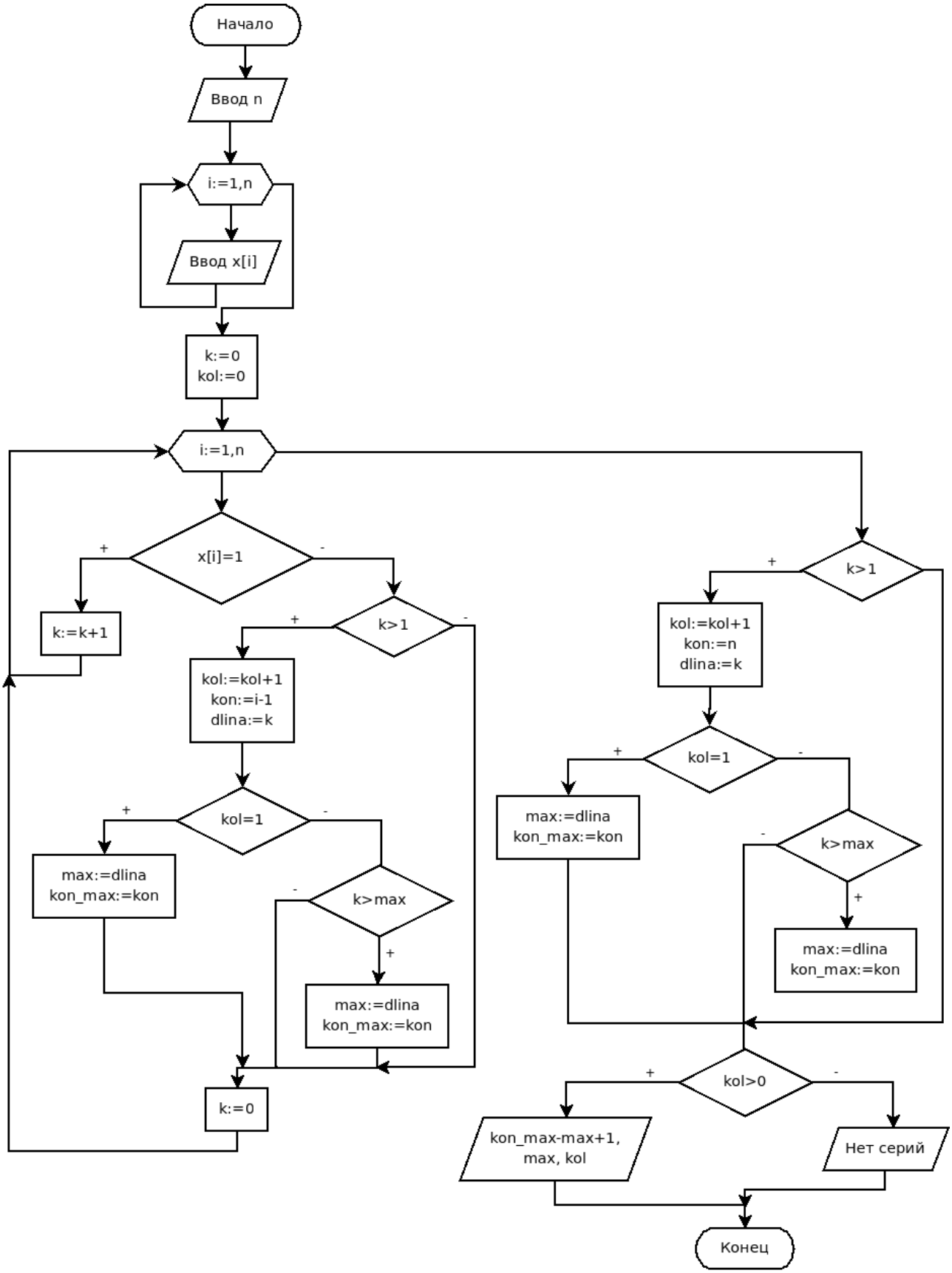

Рисунок 5.44: Блок-схема решения задачи 5.9

```
{Перебираем все элементы в массиве}
for i:=1 to n do
{Если текущий элемент равен 1, то }
if x[i]=1 then
{количество подряд идущих единиц
увеличить на 1.}
k:=k+1
else
{ Если текущий элемент не равен 1, то}
begin
{ проверяем, была ли серия до этого, k>1?}
 if k>1 then
{Если только что оборвалась серия, то}
 begin
{увеличиваем количество серий.}
    kol:=kol+1;
{Фиксируем тот факт, что на предыдущем
элементе серия закончилась,}
    kon:=i-1;
{Длина серии равна k.}
    dlina:=k;
{Если это первая серия,}
    if kol=1 then
{объявляем ее максимальной.}
    begin
{Длина максимальной серии единиц.}
        max:=dlina;
{Конец максимальной серии, состоящей из
 единиц, хранится в переменной kon_max.}
        kon_max:=kon;
    end
{Если это не первая серия,
состоящая из единиц,}
    else
{то ее длину сравниваем с длиной серии
с максимальным количеством единиц.}
        if k>max then
{Если длина текущей серии больше,}
```

```
        begin
{то объявляем ее максимальной.}
            max:=dlina; kon_max:=kon;
        end;
 end;
{Если текущий элемент массива не равен 1,
то количество подряд встречающихся единиц
начинаем считать сначала (k:=0).}
 k:=0;
end;
{Проверка, не было ли серии в конце массива. }
if k>1 then
{ Если да, увеличить счетчик еще на единицу. }
begin
    kol:=kol+1;
{Серия закончилась на последнем элементе}
    kon:=n; dlina:=k;
{Обработка последней серии так,
как это происходило в цикле.}
    if kol=1 then
    begin
        max:=dlina; kon_max:=kon;
    end
    else
        if k>max then
        begin
            max:=dlina; kon_max:=kon;
        end;
end;
{Если серии были, то}
 if kol>0 then
{вывод информации о серии с максимальным
количеством единиц.}
 begin
    writeln('Количество серий, состоящих из
                                единиц=',kol);
    writeln('Наибольшая серия начинается
    с номера ',kon_max-max+1,', заканчивается
```

```
    номером ', kon_max,', ее длина равна ', max)
    end {Вывод информации об отсутствии серий.}
  else writeln('Нет серий, состоящих из единиц')
  end.
```

ЗАДАЧА 5.10. Задан массив вещественных чисел. Перевести все элементы массива в *p*-ричную систему счисления.

Перед решением всей задачи давайте разберемся с алгоритмом перевода вещественного числа из десятеричной в другую систему счисления. Этот алгоритм можно разделить на следующие этапы:

1. Выделение целой и дробной частей числа.
2. Перевод целой части числа в другую систему счисления.
3. Перевод дробной части числа в другую систему счисления.
4. Объединение целой и дробной частей числа в новой системе счисления.

Алгоритм перевода целого числа в другую систему счисления.

Разделить нацело число на основание новой системы счисления. Получим остаток и частное. Остаток от деления будет младшим разрядом числа. Его необходимо будет умножить на 10 в нулевой степени. Если частное не равно нулю, то продолжим деление; новый остаток даст нам следующий разряд числа, который надо будет умножить на десять в первой степени, и т. д. Деление будем продолжать до тех пор, пока частное не станет равным 0. Особенностью алгоритма является то, что число формируется в обратном порядке от младшего разряда к старшему, что позволит в один проход собрать число в новой системе счисления.

Алгоритм перевода дробной части числа в новую систему счисления.

Умножить дробную часть числа на основание системы счисления. В полученном произведении выделить целую часть числа, это будет старший разряд числа, который необходимо умножить на 10^{-1}. Дробную часть опять умножить на основание системы счисления. В произведении целая часть будет очередным разрядом (его умножать надо будет на 10^{-2}), а дробную часть необходимо опять умножать на основание системы счисления до получения необходимого количества разрядов исходного числа.

Блок-схема функции перевода вещественного число N из 10-й системы счисления в другую систему представлена на рис. 5.45.

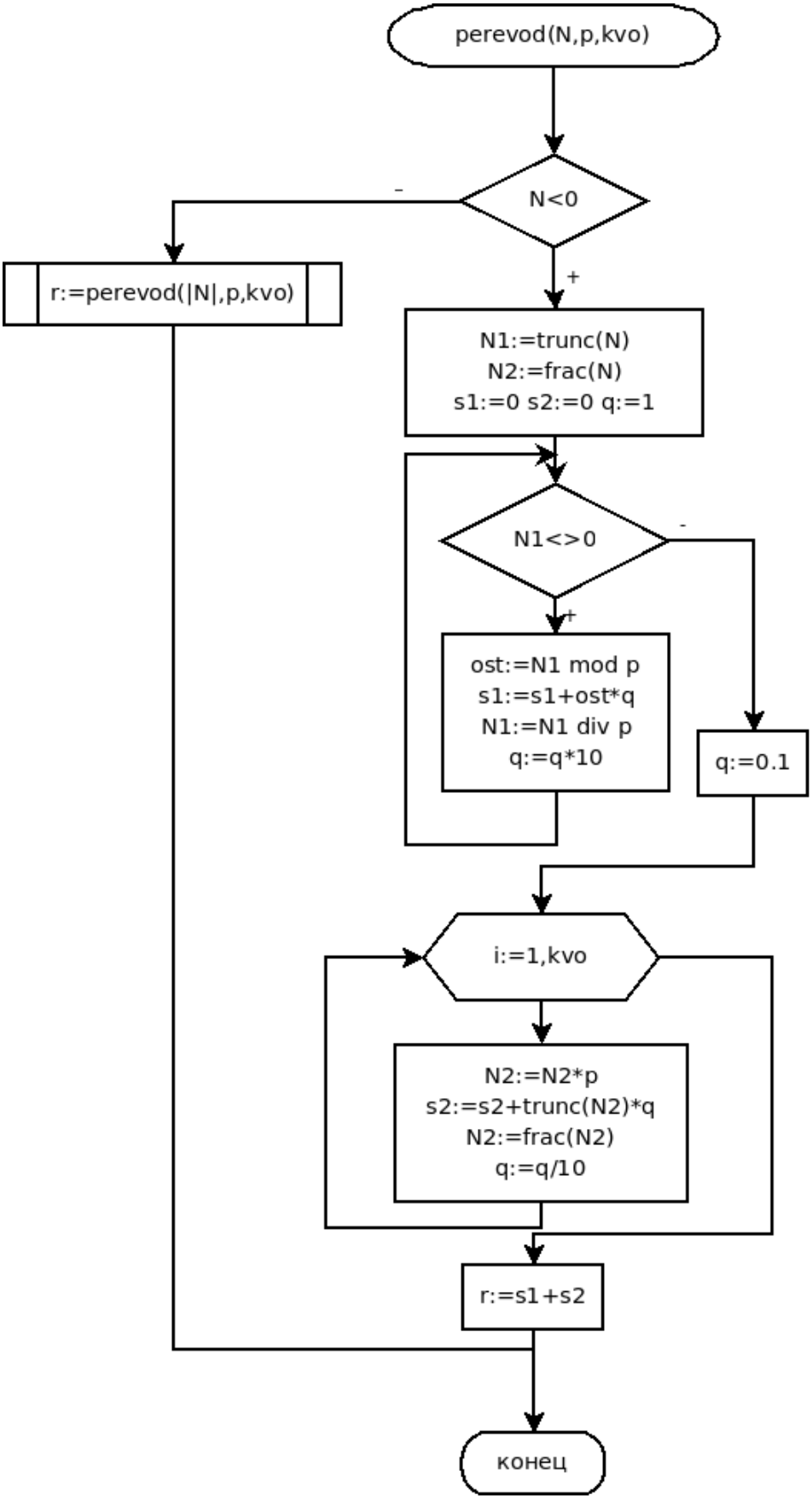

Рисунок 5.45: Блок-схема функции перевода вещественного числа в p-ричную систему счисления

Обратите внимание, как в блок-схеме и в функции реализовано возведение в степень. В связи с тем что при переводе целой части числа последовательно используются степени *10*, начиная *0*, для формирования степеней десяти, вводится переменная *q*, которая в начале равна *1*, а затем последовательно в цикле умножается на *10*. При переводе дробной части числа последовательно нужны отрицательные степени *10*: $10^{-1}, 10^{-2}, \ldots$. Поэтому при формировании дробной части числа переменная *q:=0.1*, которая последовательно в цикле делится на *10*.

Ниже приведен текст консольной программы решения задачи 5.10 с комментариями.

```
{ Функция перевода вещественного числа в p-рич-
ную систему счисления, входные параметры функции:
вещественное число N, основание системы счисления
— целое число p, kvo — количество разрядов в дроб-
ной части формируемого числа.}
   function perevod(N:real;P:word;kvo:word):real;
   var i ,N1, ost: word;
    s1, N2, r, s2:real;
    q:real;
   begin
   {Если исходное число отрицательно, то для его
перевода рекурсивно обращаемся к функции perevod,
передавая в качестве параметра модуль числа.}
   if N<0 then r:=-perevod(abs(N),P,kvo)
   else
   begin
   {Выделяем целую N1 и дробную N2 части веще-
ственного числа N.}
    N1:=trunc(N);N2:=frac(N);
    s1:=0;s2:=0;
   {В переменной q будем последовательно хранить
степени десяти, вначале туда записываем 1 — десять
в 0 степени, а затем последовательно в цикле будем
умножать q на 10.}
   q:=1;
   {Перевод целой части числа. Пока число не ста-
нет равным 0.}
```

```
    while (N1<>0) do
    begin
   {Вычисляем ost — очередной разряд числа, как остаток от деления N1 на основание системы счисления.}
        ost:=N1 mod P;
   {Очередной разряд числа умножаем на 10 в степени i и добавляем к формируемому числу s1.}
        s1:=s1+ost*q;
{Уменьшаем число N1 в p раз путем целочисленного деления на p.}
        N1:=N1 div P;
   {Формируем следующую степень десятки.}
        q:=q*10;
    end;
   {В переменной q будем последовательно хранить отрицательные степени десяти, вначале туда записываем 0.1 — десять в минус первой степени, а затем последовательно в цикле будем делить q на 10.}
    q:=0.1;
    for i:=1 to kvo do
    begin
   {Умножаем дробную часть на 10.}
        N2:=N2*p;
   {Вычисляем очередной разряд числа как целую часть от умножения N2 на основание системы счисления. Очередной разряд числа умножаем на 10 в степени i и добавляем к формируемому числу s2.}
        s2:=s2+trunc(N2)*q;
   {Выделяем дробную часть от сформированного числа}
        N2:=frac(N2);
   {Формируем очередную отрицательную степень 10.}
        q:=q/10;
    end;
   {Суммируем целую и дробную части числа в p-ричной системе счисления.}
    r:=s1+s2;
```

```
    end;
     perevod:=r;
    end;
    var C:array[1..100] of real;
    p,i,n:word;
    begin
    {Ввод размера массива.}
     Write('n=');readln(n);
    {Ввод массива.}
     writeln('Массив C');
     for i:=1 to n do  read(C[i]);
    {Ввод системы счисления.}
     writeln('Введите основание
                        системы счисления');
    readln(p);
    {Перевод всех элементов массива в другую систе-
му счисления.}
     for i:=1 to n do
     c[i]:=perevod(C[i],p,5);
    {Вывод преобразованного массива.}
     writeln('Преобразованный массив C');
     for i:=1 to n do
     write(C[i]:1:5,' ');
    end.
```

Результаты работы программы представлены на рис. 5.46.

```
Файл  Правка  Вид  Терминал  Вкладки  Справка
n=10
Массив C
5.7 9.905 -78.946 76.894 -11.1111 11.11 0.89654 -0.1324 132.13 90.87
Введите основание системы счисления
5
Преобразованный массив C
10.32222 14.42303 -303.43311 301.42133 -21.02342 21.02333 0.42201 -0.03123 1012.03111 330.41333

------------------
(program exited with code: 0)
Press return to continue
```

Рисунок 5.46: Результаты работы консольной программы решения задачи 5.10

ЗАДАЧА 5.11. Из массива целых положительных чисел Y удалить 3 последних числа, цифры которых в восьмеричном представлении образуют убывающую последовательность.

Для решения этой задачи понадобится функция, которая будет проверять, образуют ли цифры числа в восьмеричном представлении убывающую последовательность цифр.

Заголовок этой функции будет иметь вид:

```
function vosem(N:word):boolean;
```

На вход функции `vosem` приходит целое десятичное число (формальный параметр `N`), функция возвращает `true`, если цифры числа в восьмеричном представлении образуют убывающую последовательность и `false` в противном случае.

При разработке алгоритма этой задачи следует помнить, что при переводе числа из десятеричной системы в восьмеричную разряды числа мы будем получать в обратном порядке. Значит, получаемые восьмеричные разряды наоборот должны формировать возрастающую последовательность цифр.

Текст функции с комментариями приведен ниже.

```
function vosem(N:word):boolean;
var pr:boolean;
tsifra,tsifra_st:word; i:integer;
begin
 i:=0;
{Предположим, что цифры числа N в восьмеричном представлении образуют убывающую последовательность.}
pr:=true;
{Пока число N не равно 0, }
 while N<>0 do
 begin
{Достаем очередной разряд числа в восьмеричной системе.}
     tsifra:=N mod 8;
{Уменьшаем число в 8 раз.}
     N:=N div 8; i:=i+1;
{Если разряд не первый}
     if i>1 then {и текущий разряд меньше или равен предыдущему, цифры в 8-м представлении числа N не образуют убывающую последовательность (pr:=false) и аварийно покидаем цикл.}
     if tsifra<=tsifra_st then
```

```
        begin
            pr:=false; break;
        end;
        tsifra_st:=tsifra;
    end;
    vosem:=pr;
end;
```

Алгоритм решения задачи будет следующий. Перебираем все числа в массиве в обратном порядке, проверяем, образуют ли цифры текущего элемента массива в восьмеричном представлении убывающую последовательность. Если образуют, то количество таких чисел (k) увеличиваем на 1. Если $k \le 3$, то удаляем текущий элемент массива.

Создадим визуальное приложение, предназначенное для решения задачи 5.11. Расположим на форме следующие компоненты: три кнопки, три метки, одно поле ввода и две таблицы строк. Расположим их примерно так, как показано на рис. 5.47.

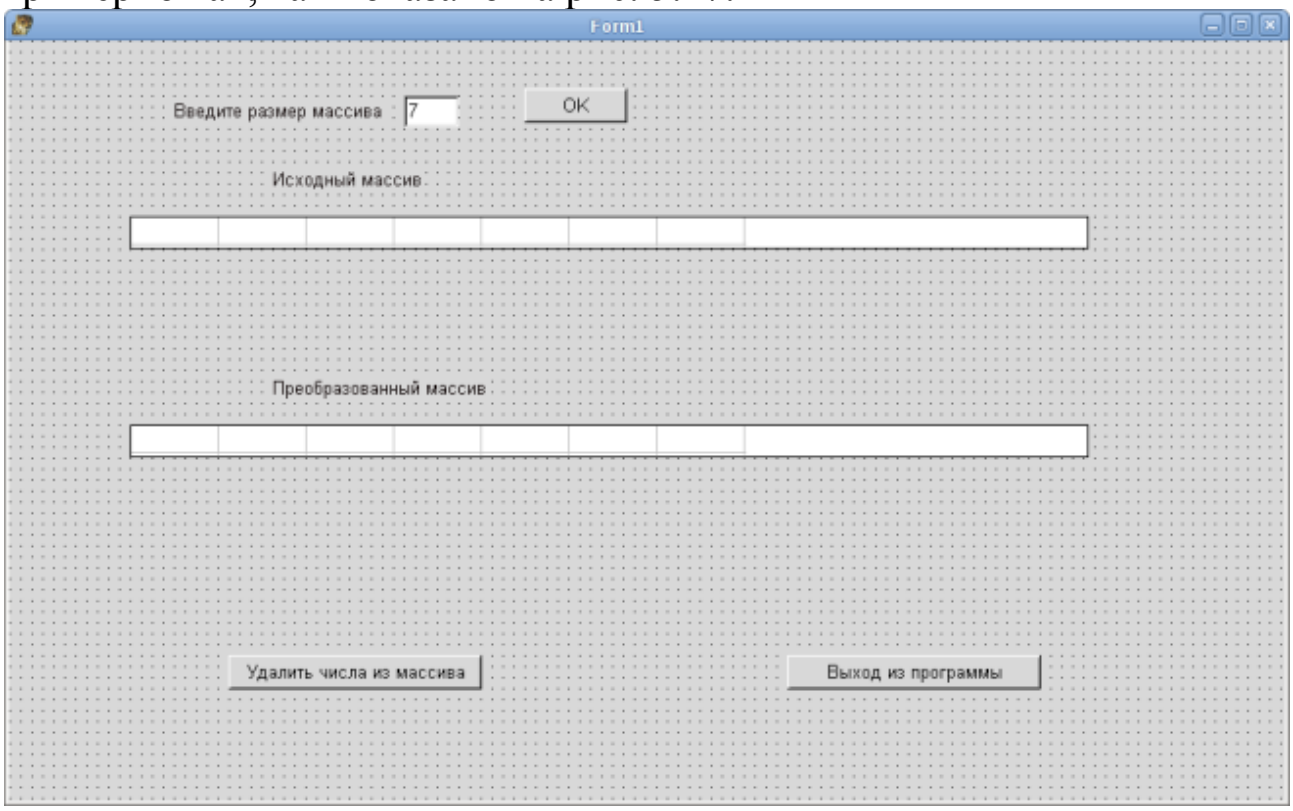

Рисунок 5.47: Форма для решения задачи 5.11

Свойства основных компонентов представлены в таблицах 5.6-5.7. При запуске приложения видимы метка Label1, поле для ввода размера массива Edit1 и кнопка Button1.

Таблица 5.6: Свойства меток, кнопок и текстового поля

Name	Caption (Text)	Width	Visible	Left	Top
label1	Введите размер массива	153	true	120	46
label2	Исходный массив	110	false	192	96
label3	Преобразованный массив	157	false	192	210
Edit1	7	40	true	288	40
Button1	OK	75	true	376	35
Button2	Удалить числа из файла	185	false	160	448
Button3	Выход из программы	185	false	568	448

Таблица 5.7: Свойства таблиц строк

Свойства	Таблица 1	Таблица 2
Name	StringGrid1	StringGrid2
ColCount	7	7
RowCount	1	1
Visible	false	false
FixedCols	0	0
FixedRows	0	0
Options.goEditing	true	true

При щелчке по кнопке OK (Button1) из поля ввода Edit1 считывается размер массива и становятся видимыми две другие кнопки, метка Label2, таблица строк StringGrid1 для ввода элементов массива. Метка Label1, поле для ввода размера массива Edit1 и кнопка Button1 становятся невидимыми. Окно приложения после щелчка по кнопке OK станет подобным представленному на рис. 5.48.

При щелчке по кнопке **Удалить числа из массива** происходят следующие действия:

- считывание массива из StringGrid1;
- удаление из массива трех последних чисел, цифры которых в восьмеричном представлении образуют убывающую последовательность;
- становятся видимыми метка Label3, таблица строк StringGrid2 для вывода элементов преобразованного массива.

Ниже приведен текст модуля с необходимыми комментариями. В результате работы программы решения задачи 5.11 окно приложения примет вид, представленный на рис. 5.49

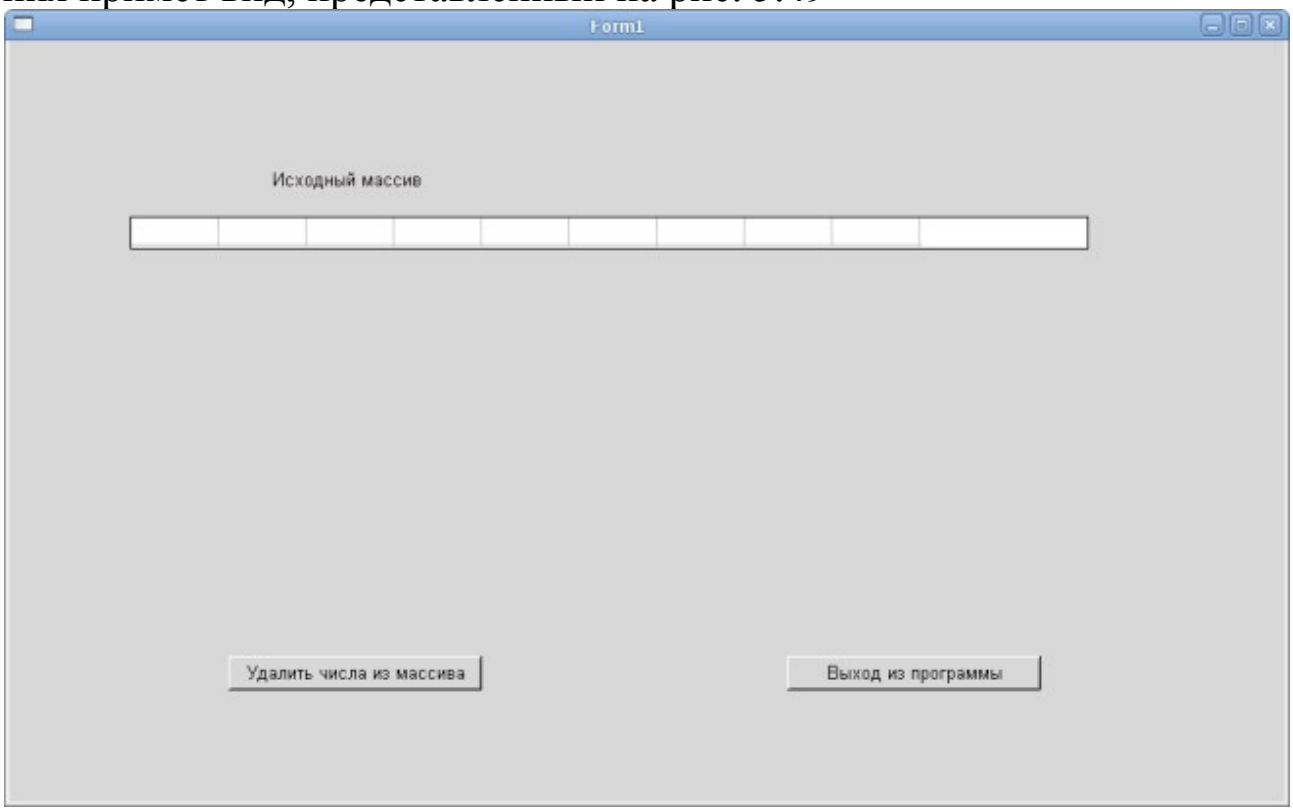

Рисунок 5.48: Окно приложения после щелчка по кнопке ОК

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses Classes, SysUtils, LResources, Forms,
 Controls, Graphics, Dialogs, StdCtrls, Grids;
{Описание формы}
type
 { TForm1 }
 TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    StringGrid1: TStringGrid;
    StringGrid2: TStringGrid;
```

```
        procedure Button1Click(Sender: TObject);
         procedure Button2Click(Sender: TObject);
        procedure Button3Click(Sender: TObject);
    private
        { private declarations }
    public
        { public declarations }
     end;
    type massiv=array[1..100] of word;
    var
     Form1: TForm1;
     N:word;
     X:massiv;
    implementation
    { TForm1 }
    {Функция vosem проверяет, образуют ли цифры
числа N в восьмеричном представлении убывающую по-
следовательность}
    function vosem(N:word):boolean;
    var pr:boolean;
     tsifra,tsifra_st:word;
     i:word;
    begin
     i:=0;
    {Предположим, что цифры числа N в восьмеричном
представлении образуют убывающую последователь-
ность.}
    pr:=true;
    {Пока число N не равно 0, }
     while N<>0 do
     begin
    {Достаем очередной разряд числа в восьмеричной
системе.}
        tsifra:=N mod 8;
    {Уменьшаем число в 8 раз.}
        N:=N div 8;
        i:=i+1;
    {Если разряд не первый}
```

```
        if i>1 then
   {И текущий разряд меньше или равен предыдущему,
цифры в восьмеричном представлении числа N не об-
разуют убывающую последовательность (pr:=false) и
аварийно покидаем цикл.}
        if tsifra<=tsifra_st then
        begin
            pr:=false;
            break;
        end;
        tsifra_st:=tsifra;
    end;
    vosem:=pr;
   end;
   {Функция удаления из массива X(N) элемента с
номером m.}
procedure udal(var X:Massiv; m:word;var N:word);
   var i:word;
   begin
    for i:=m to N-1 do
        x[i]:=x[i+1];
    N:=N-1;
   end;
   {Обработчик щелчка по кнопке OK.}
   procedure TForm1.Button1Click(Sender: TObject);
   begin
   {Считываем размер массива из поля ввода.}
    N:=StrToInt(Edit1.Text);
   {Делаем невидимыми первую метку, поле ввода и
кнопку OK.}
    Label1.Visible:=False;
    Edit1.Visible:=False;
    Button1.Visible:=False;
   {Делаем видимыми вторую метку, таблицу строк.}
    label2.Visible:=True;
    StringGrid1.Visible:=True;
   {Устанавливаем количество элементов
   в таблице строк.}
```

```
    StringGrid1.ColCount:=N;
   {Делаем видимыми вторую и третью кнопки.}
    Button2.Visible:=True;
    Button3.Visible:=True;
   end;
   {Обрабочик событий кнопки «Удалить числа из
массива».}
   procedure TForm1.Button2Click(Sender: TObject);
   var k,i:word;
   begin
   {Считываем массив из таблицы строк.}
    for i:=0 to N-1 do
    X[i+1]:=StrToInt(StringGrid1.Cells[i,0]);
   k:=0;
   {Перебираем все элементы массива в обратном по-
рядке.}
    for i:=N-1 downto 0 do
   {Если цифры очередного элемента массива в вось-
меричном представлении образуют убывающую последо-
вательность}
        if vosem(x[i]) then   begin
   {Увеличиваем счетчик таких чисел на 1.}
          k:=k+1;
   {Если это первое, второе или третье число,
удовлетворяющее условию, то удаляем его из масси-
ва.}
          if k<=3 then
              udal(x,i,N);   end;
   {Делаем видимыми третью кнопку и вторую таблицу
строк.}
    label3.Visible:=True;
    StringGrid2.Visible:=True;
    StringGrid2.ColCount:=N;
   {Вывод преобразованного массива.}
    for i:=0 to N-1 do
       StringGrid2.Cells[i,0]:=IntToStr(X[i+1]);
   end;
   {Обработчик кнопки закрытия окна.}
```

```
procedure TForm1.Button3Click(Sender: TObject);
begin
 Close;
end;
initialization
 {$I unit1.lrs}
end.
```

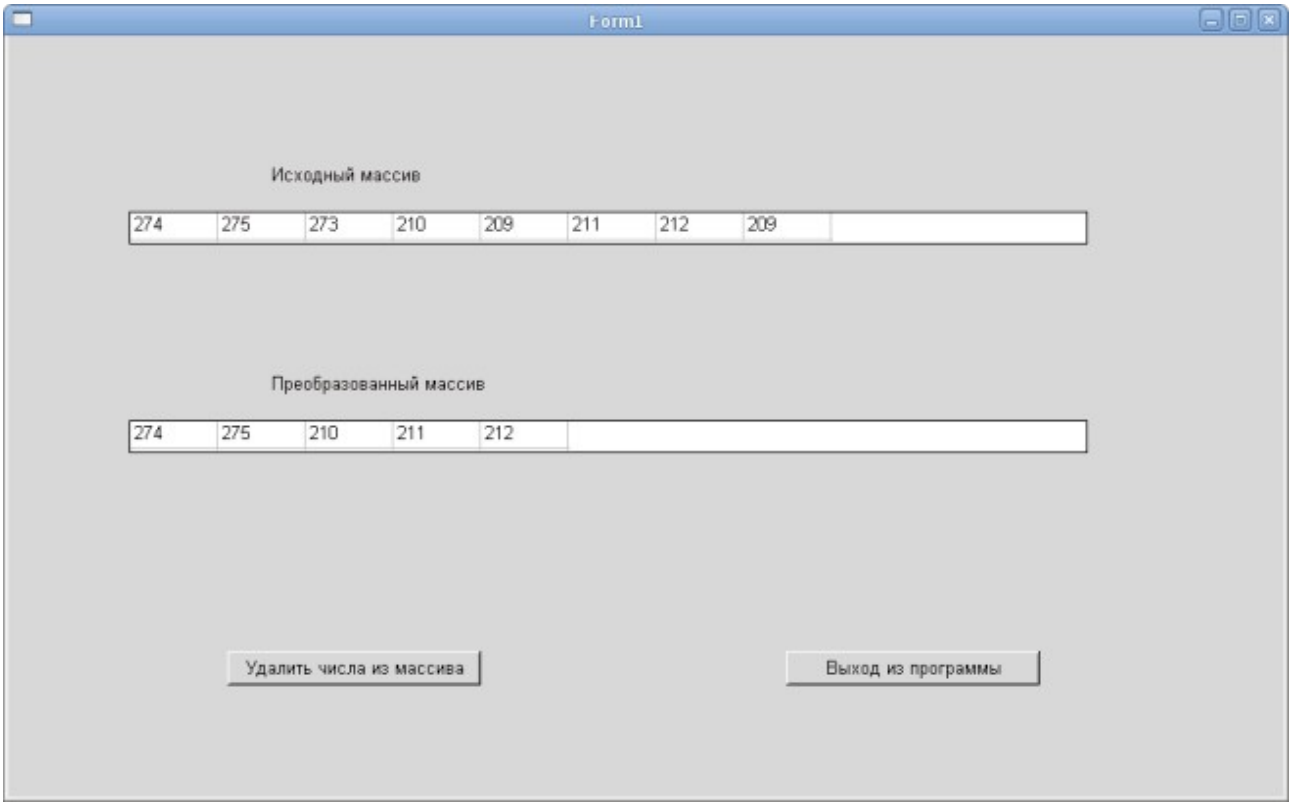

Рисунок 5.49: Окно с результатами решения задачи 5.11

5.13 Задачи для самостоятельного решения

1. Записать положительные элементы массива X подряд в массив Y. Вычислить сумму элементов массива X и произведение элементов массива Y. Из массива Y удалить элементы, расположенные между максимальным и минимальным элементами.

2. Сформировать массив B, записав в него элементы массива A с нечетными индексами. Вычислить среднее арифметическое элементов массива B и удалить из него максимальный, минимальный и пятый элементы.

3. Дан массив целых чисел X. Переписать пять первых положительных элементов массива и последних два простых элемента в массив Y. Найти максимальный отрицательный элемент массива X.

4. Записать элементы массива X, удовлетворяющие условию $1 \le x_i \le 2$, подряд в массив Y. Поменять местами максимальный и минимальный элементы в массиве Y.

5. Переписать элементы массива целых чисел X в обратном порядке в массив Y. Вычислить количество четных, нечетных и нулевых элементов массива Y.

6. Определить максимальный и минимальный элементы среди положительных нечетных элементов целочисленного массива X. Удалить из массива все нулевые элементы.

7. Переписать элементы целочисленного массива $X=(x_1, x_2, \ldots, x_{12})$ в массив $Y = (y_1, y_2, \ldots, y_{12})$, сдвинув элементы массива X вправо на три позиции. При этом три элемента с конца массива X перемещаются в начало: $(y_1, y_2, \ldots, y_{12}) = (x_{10}, x_{11}, x_{12}, x_1, x_2, \ldots, x_9)$. Определить номера максимального простого и минимального положительного элементов в массивах X и Y.

8. Записать элементы массива $X = (x_1, x_2, \ldots, x_{15})$ в массив $Y = (y_1, y_2, \ldots, y_{15})$, сдвинув элементы массива X влево на четыре позиции. При этом четыре элемента, стоящие в начале массива X, перемещаются в конец: $(y_1, y_2, \ldots, y_{15}) = (x_5, x_6, \ldots, x_{15}, x_1, x_2, x_3, x_4)$. Поменять местами минимальный и максимальный элементы массива Y.

9. В массиве X определить количество элементов меньших среднего арифметического значения. Удалить из массива положительные элементы, расположенные между максимальным и минимальным.

10. Вычислить среднее арифметическое элементов массива X, расположенных между его минимальным и максимальным значениями. Если минимальный элемент размещается в массиве раньше максимального, то упорядочить массив на данном промежутке по возрастанию его элементов.

11. Определить, содержит ли заданный массив группы элементов, расположенные в порядке возрастания их значений. Если да, то определить количество таких групп.

12. В заданном массиве целых чисел найти самую маленькую серию подряд стоящих нечетных элементов.

13. Удалить из массива целых чисел все простые числа, расположенные до максимального значения.

14. Удалить из массива предпоследнюю группу элементов, представляющих собой знакочередующийся ряд.

15. Задан массив целых положительных чисел X. Определить количество совершенных чисел в массиве. Удалить из массива последних два отрицательных числа. Сформировать массив Y, куда записать номера элементов массива X, являющихся простыми числами.

16. Переписать положительные элементы массива целых чисел X в обратном порядке в массив Y. Вычислить процент четных, нечетных и нулевых элементов массива Y. Перевести элементы массива Y в двоичную систему счисления.

17. Определить максимальный и минимальный элементы среди положительных четных элементов целочисленного массива X. Удалить из массива X совершенные числа, расположенные после максимального значения.

18. Заданы массивы вещественных чисел X и Y. Сформировать массив Z, куда записать положительные элементы массивов Y и Z в семеричной системе счисления. Определить номера максимального и минимального элементов в массиве Z.

19. Записать четные положительные элементы целочисленных массивов X и Y в массив Z. Поменять местами минимальный и максимальный элементы массива Z. Вывести элементы массива Z в четверичной системе счисления.

20. Из целочисленного массива X удалить все числа, превышающие среднее арифметическое простых элементов массива.

21. В массивах вещественных чисел X и Y записаны координаты точек на плоскости. Найти две точки, расстояние между которыми наименьшее.

22. Определить, содержит ли заданный массив вещественных чисел группы элементов, расположенные в порядке убывания их значений. Если да, то определить группу наименьшей длины.

23. В заданном массиве целых чисел найти самую большую серию подряд стоящих четных элементов.

24. Удалить из массива целых чисел все элементы, которые в пятеричном представлении не содержат нулей.

25. Из массивов вещественных чисел A и B сформировать массив C, записав в него элементы массивов A и B, которые не содержат «семерок» в восьмеричном представлении.

6 Обработка матриц во Free Pascal

Матрица – это двумерный массив, каждый элемент которого имеет два индекса: номер строки и номер столбца.

Объявить двумерный массив (матрицу) можно так:

```
имя: array [индекс1_нач .. индекс1_кон,
          индекс2_нач .. индекс2_кон] of тип;
```

где тип определяет тип элементов массива, имя — имя матрицы, индекс1_нач .. индекс1_кон — диапазон изменения номеров строк, индекс2_нач .. индекс2_кон — диапазон изменения номеров столбцов матрицы.

Например,

```
var h: array [0..11,1..10] of real;
```

Описана матрица вещественных чисел h, состоящая из двенадцати строк и десяти столбцов (строки нумеруются от 0 до 11, столбцы — от 1 до 10).

Существует ещё один способ описать матрицы, для этого надо создать новый тип данных:

```
type
новый_тип=array [индекс1_нач .. индекс1_кон]
                                        of тип;
var
имя: array [индекс2_нач .. индекс2_кон]
                                    of новый_тип;
```

или

```
type
новый_тип=array [список_диапазонов] of тип;
var
    имя: новый_тип;
```

Например:

```
type
 massiv=array [1..30] of integer;
 matrica=array [0..15,0..13] of real;
var
 a, b: array [1..10] of massiv;
 c:matrica;
```

В данном случае в матрицах `a` и `b` есть 10 строк и 30 столбцов, а `c` – матрица, в которой есть шестнадцать строк и четырнадцать столбцов.

Для обращения к элементу матрицы необходимо указать ее имя и в квадратных скобках через запятую номер строки и номер столбца:

```
имя[номер_строки, номер_столбца]
```

или так

```
имя[номер_строки] [номер_столбца]
```

Например, `h[2,4]`[67] – элемент матрицы `h`, находящийся в строке под номером два и столбце под номером четыре.

Для обработки всех элементов матрицы необходимо использовать два цикла. Если обрабатываем матрицу построчно, то во внешнем цикле последовательно перебираем строки от первой до последней, затем во внутреннем — перебираем все (первый, второй, третий и т. д.) элементы текущей строки. При обработке элементов матрицы по столбцам, внешний цикл будет по столбцам, внутренний — по строкам.

На рис. 6.1 представлена блок-схема алгоритма обработки матрицы по строкам, на рис. 6.2 — по столбцам. Здесь `i` — номер строки, `j` — номер столбца, `N` — количество строк, `M` — количество столбцов матрицы `A`.

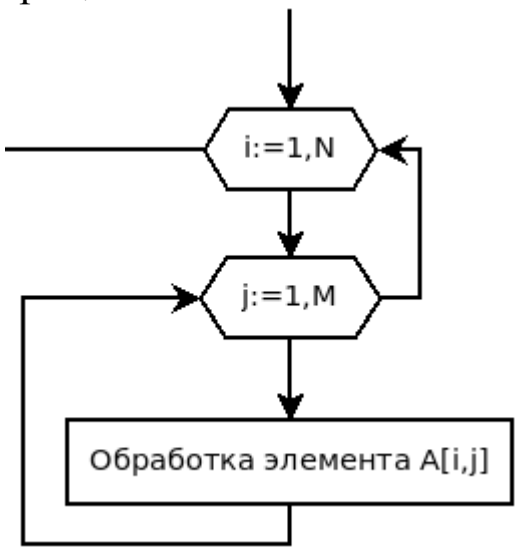

Рисунок 6.1: Построчная обработка матрицы

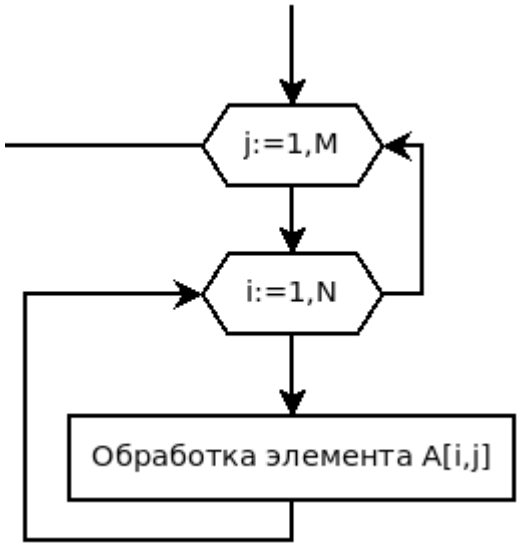

Рисунок 6.2: Алгоритм обработки матрицы по столбцам

67 или h[2][4]

6.1 Ввод-вывод матриц

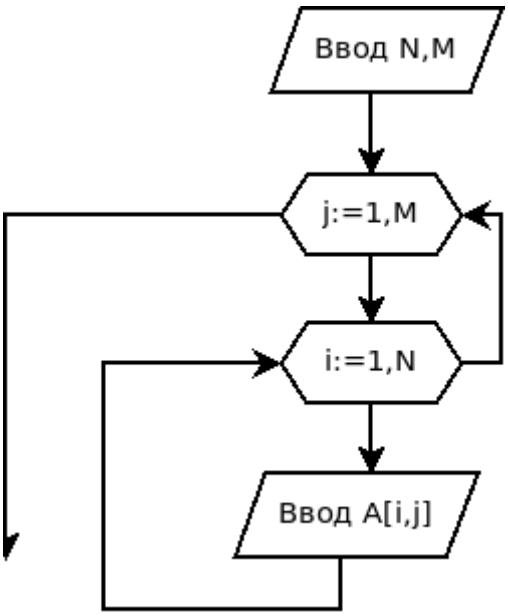

Рисунок 6.3: Блок-схема ввода элементов матрицы

Матрицы, как и массивы, нужно вводить (выводить) поэлементно. Вначале следует ввести размеры матрицы, а затем уже в двойном цикле вводить элементы. Блок-схема ввода элементов матрицы изображена на рис. 6.3.

Вывод можно осуществлять по строкам или по столбцам, но лучше, если элементы располагаются построчно, например,

```
 2   3  13  35
 5  26  76  37
52  61  79  17
```

Алгоритм построчного вывода элементов матрицы приведен на рис. 6.4.

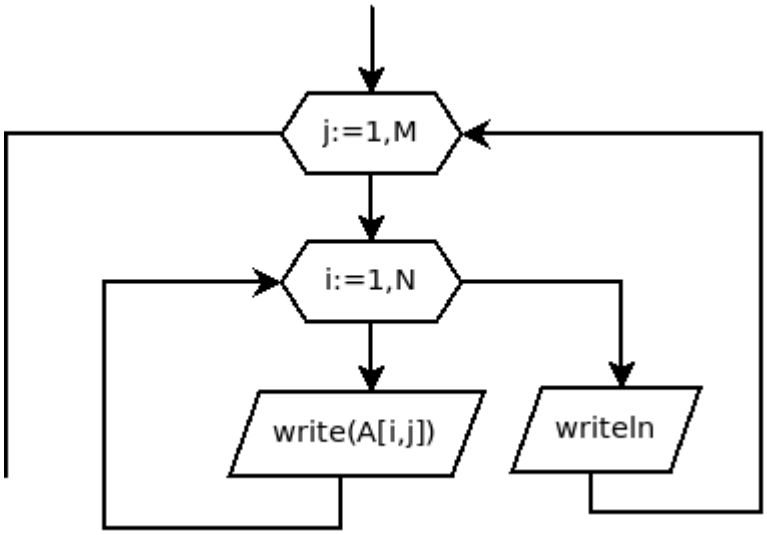

Рисунок 6.4: Построчный вывод матрицы

Об описании матриц на языке Free Pascal было рассказано в разделе 5.2 пятой главы, обращение к элементу $A_{i,j}$ матрицы можно осуществить с помощью конструкции `A[i,j]` или `A[i][j]`.

Рассмотрим реализацию ввода-вывода матриц в консольных приложениях. Для организации построчного ввода матрицы в двойном цикле по строкам и столбцам можно использовать оператор `read`.

```
for i:=1 to N do
for j:=1 to m do
    read(A[i,j]);
```

В этом случае элементы каждой строки матрицы можно разделять символами пробела или табуляции и только в конце строки нажимать **Enter**.

ЗАДАЧА 6.1. Написать консольное приложение ввода матрицы вещественных чисел и вывода ее на экран дисплея.

Ниже приведен пример программы ввода - вывода матрицы.

```
var
a: array [1..20,1..20] of real;
i,j,n,m: integer;
begin
{Ввод размеров матрицы}
writeln('Введите количество строк и столбцов
                                        матрицы A');
readln(n,m);
{ Ввод элементов матрицы. }
writeln('Введите матрицу');
for i:=1 to n do
for j:=1 to m do
 read(A[i,j]);
{ Вывод элементов матрицы. }
writeln ('матрица A ' );
for i:=1 to n do
begin
 for j:=1 to m do
     write(a[i,j]:8:3, ' ');   {Печать строки.}
 writeln    {Переход на новую строку.}
end;
```

На рис. 6.5 представлены результаты работы программы.

Ввод матрицы можно также организовать с помощью следующего цикла. Авторы предлагают читателю самостоятельно разобраться, в чем будет отличие ввода матрицы в этом случае.

```
for i:=1 to N do
 for j:=1 to m do
 begin
     write('A(',i,',',j,')=');
     readln(A[i,j])
 end;
```

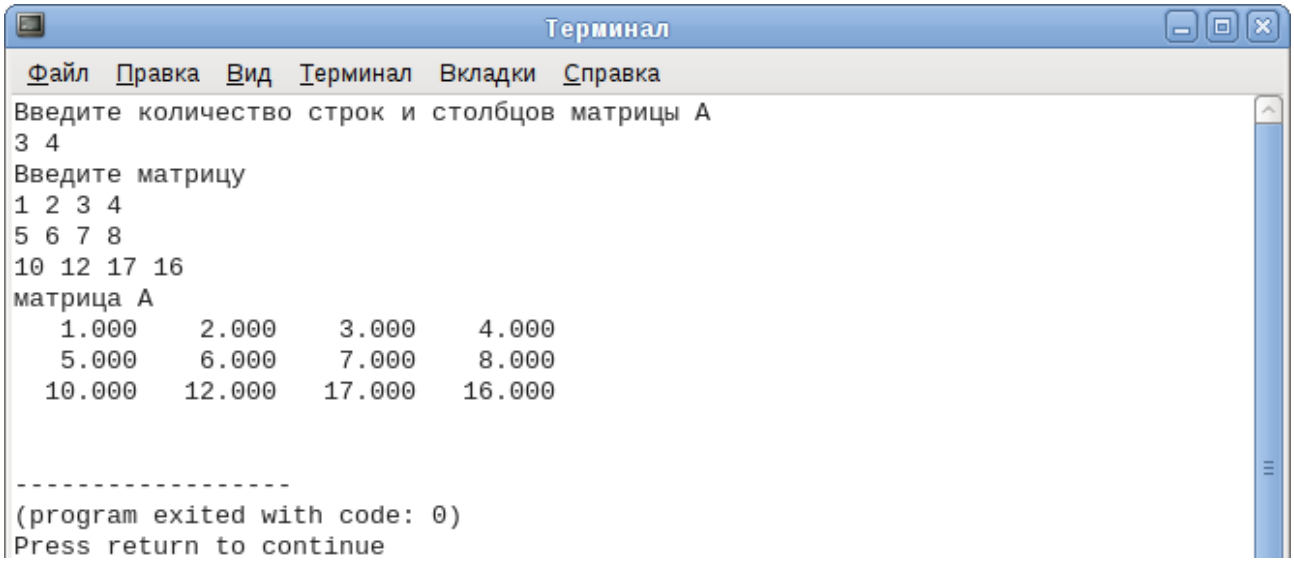

Рисунок 6.5: Результаты работы программы решения задачи 6.1

Для ввода-вывода матриц можно использовать компонент типа `TStringGrid`, с которым мы познакомились в пятой главе.

В качестве примера рассмотрим следующую задачу.

ЗАДАЧА 6.2. Составить программу транспонирования[68] матрицы A.

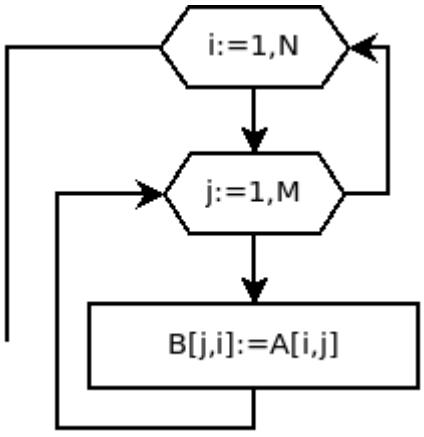

Рисунок 6.6: Блок-схема транспонирования матрицы A

Блок-схема транспонирования матрицы приведена на рис. 6.6. При транспонировании матрицы A(N,M) получается матрица B(M,N).

Рассмотрим частный случай транспонирования матрицы фиксированного размера `A(4,3)`. На форме разместим метки `Label1` и `Label2` со свойствами `Caption – Заданная матрица A` и `Транспонированная матрица B`, два компонента типа `TStringGrid`, изменив их свойства, так как показано в табл. 6.1, и кнопку `Транспонирование матрицы`.

Таблица 6.1: Свойства компонентов `StringGrid1`, `StringGrid2`.

Свойство	StringGrid1	StringGrid2	Описание свойства
Top	30	30	Расстояние от верхней границы таблицы до верхнего края формы

68 Транспонированная матрица — матрица, полученная из исходной матрицы A(N,M) заменой строк на столбцы.

Свойство	StringGrid1	StringGrid2	Описание свойства
`Left`	15	240	Расстояние от левой границы таблицы до левого края формы
`Height`	130	130	Высота таблицы
`Width`	200	200	Ширина таблицы
`ColCount`	4	5	Количество столбцов
`RowCount`	5	4	Количество строк
`DefaultColWidth`	30	30	Ширина столбца
`DefaultRowHeight`	20	20	Высота строки
`Options.goEditing`	true	false	Возможность редактирования таблицы

Окно формы приложения представлено на рис. 6.7.

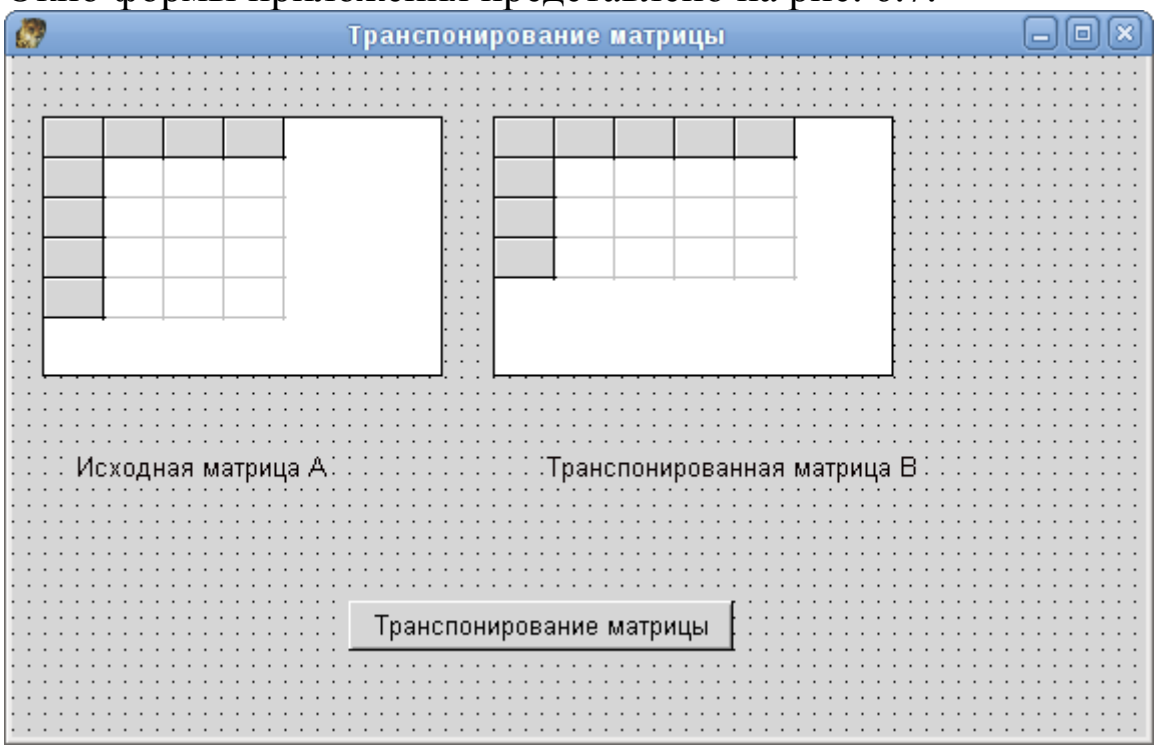

Рисунок 6.7: Форма приложения транспонирования матрицы

Ниже приведен текст подпрограммы с комментариями, которая будет выполняться, если пользователь щелкнет по кнопке `Транспонирование матрицы`.

```
procedure TForm1.Button1Click(Sender: TObject);
const n=4;m=3;  //Размерность матрицы A(n,m).
var
i,j:byte;               //Индексы матрицы:
//i – строки, j – столбцы.
//Исходная матрица.
A:array [1..n,1..m] of integer;
//Транспонированная матрица.
B:array [1..m,1..n] of integer;
Begin
//Исходные данные считываются
//из ячеек таблицы на форме,  и их значения
//записываются в двумерный массив A.
//Цикл по номерам строк.
for i:=1 to n do
//Цикл по номерам столбцов.
for j:=1 to m do
//Считывание элементов матрицы
//A из компонента StringGrid1.
 A[i,j]:=StrToInt(StringGrid1.Cells[j,i]);
//Формирование транспонированной матрицы B,
//см. блок-схему на рис. 6.6.
//Цикл по номерам строк.
for i:=1 to n do
//Цикл по номерам столбцов.
for j:=1 to m do
B[j,i]:=A[i,j];
//Элементы матрицы B выводятся
//в ячейки таблицы на форме.
//Цикл по номерам строк.
for i:=1 to n do
//Цикл по номерам столбцов.
for j:=1 to m do
//Обращение к элементам матрицы
//происходит по столбцам.
StringGrid2.Cells[i,j]:=IntToStr(B[j,i]);
end;
```

Результаты работы программы представлены на рис. 6.8.

Рисунок 6.8: Результаты работы программы транспонирования матрицы A(3,4)

Для демонстрации ввода-вывода матриц с помощью компонента типа `TStringGrid` мы рассмотрели работу с матрицами фиксированного размера `A(4,3)` и `B(3,4)`. Теперь рассмотрим общий случай решения задачи транспонирования матрицы `A(N,M)`.

Расположим на форме следующие компоненты:

- метку `label1` с надписью «`Введите размерность матрицы`»;
- метку `label2` с надписью «`N=`»;
- метку `label3` с надписью «`M=`»;
- метку `label4` с надписью «`Исходная матрица A`»;
- метку `label5` с надписью «`Преобразованная матрица B`»;
- поле ввода `Edit1` для ввода числа `N`;
- поле ввода `Edit2` для ввода числа `M`;
- компонент `StringGrid1` для ввода исходной матрицы `A`;
- компонент `StringGrid2` для хранения транспонированной матрицы `B`;
- кнопку `Button1` с надписью «`Ввод`» для ввода размеров матрицы `A`;

- кнопку `Button2` с надписью «`Очистить`» для очистки содержимого матриц;
- кнопку `Button3` с надписью «`Транспонирование`» для решения задачи транспонирования матрицы `A`;
- кнопку `Button4` с надписью «`Выход из программы`» для завершения работы программы.

Можно разместить компоненты на форме таким образом, как показано на рис. 6.9.

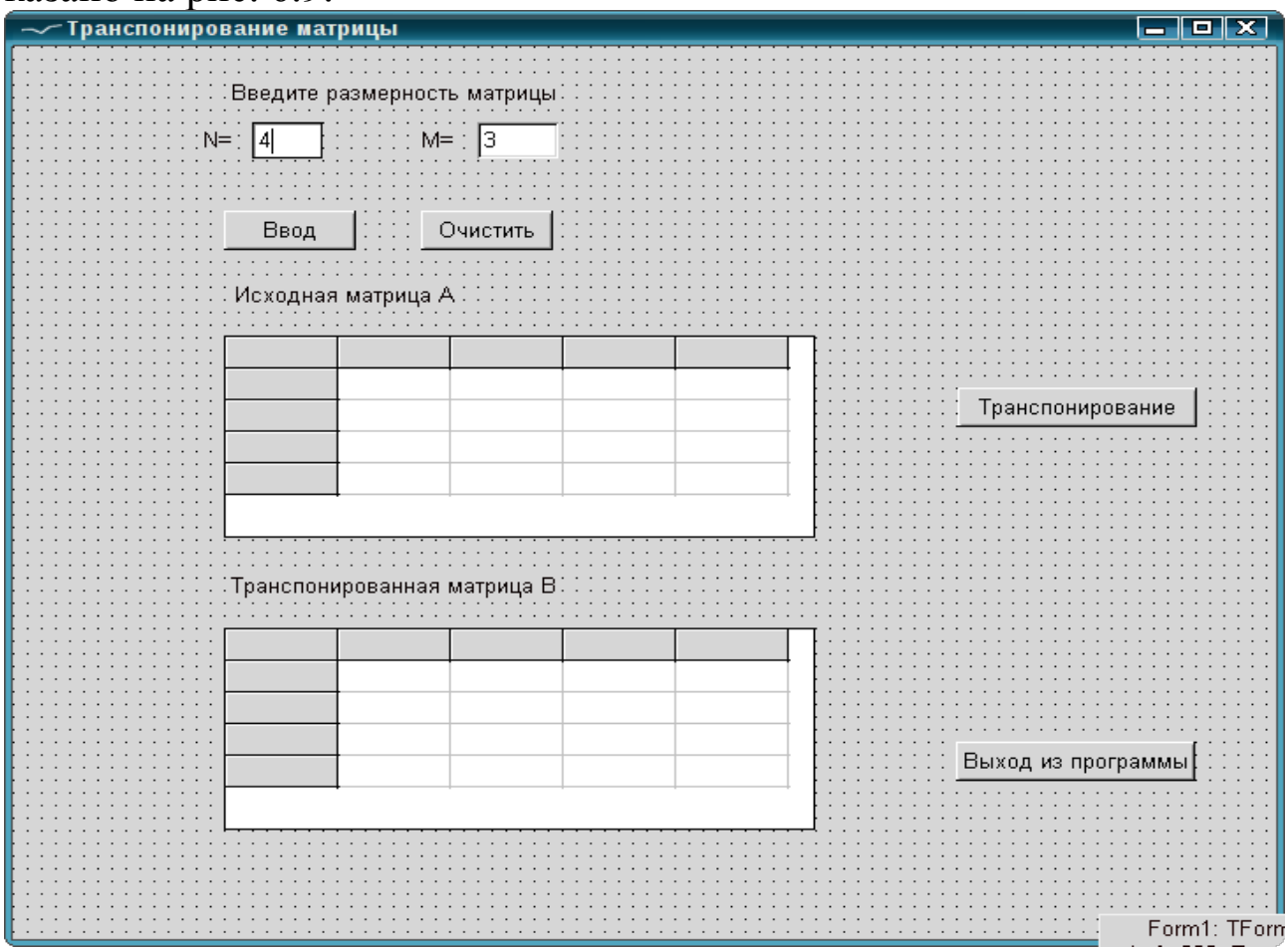

Рисунок 6.9: Окно формы решения задачи транспонирования матрицы A(N,M)

Установим свойство видимости `Visible` в `False` у компонентов метки `label4` и `label5`, `StringGrid1`, `StringGrid2`, кнопки `Button2` и `Button3`. После этого при старте программы будут видны только компоненты, отвечающие за ввод размеров матрицы, и кнопка `Выход из программы` (см. рис. 6.10)[69]. Матрицы `A` и `B`, их размеры `N`, `M` объявим глобально.

69 Свойству Caption формы присвоено значение **Транспонирование матрицы**.

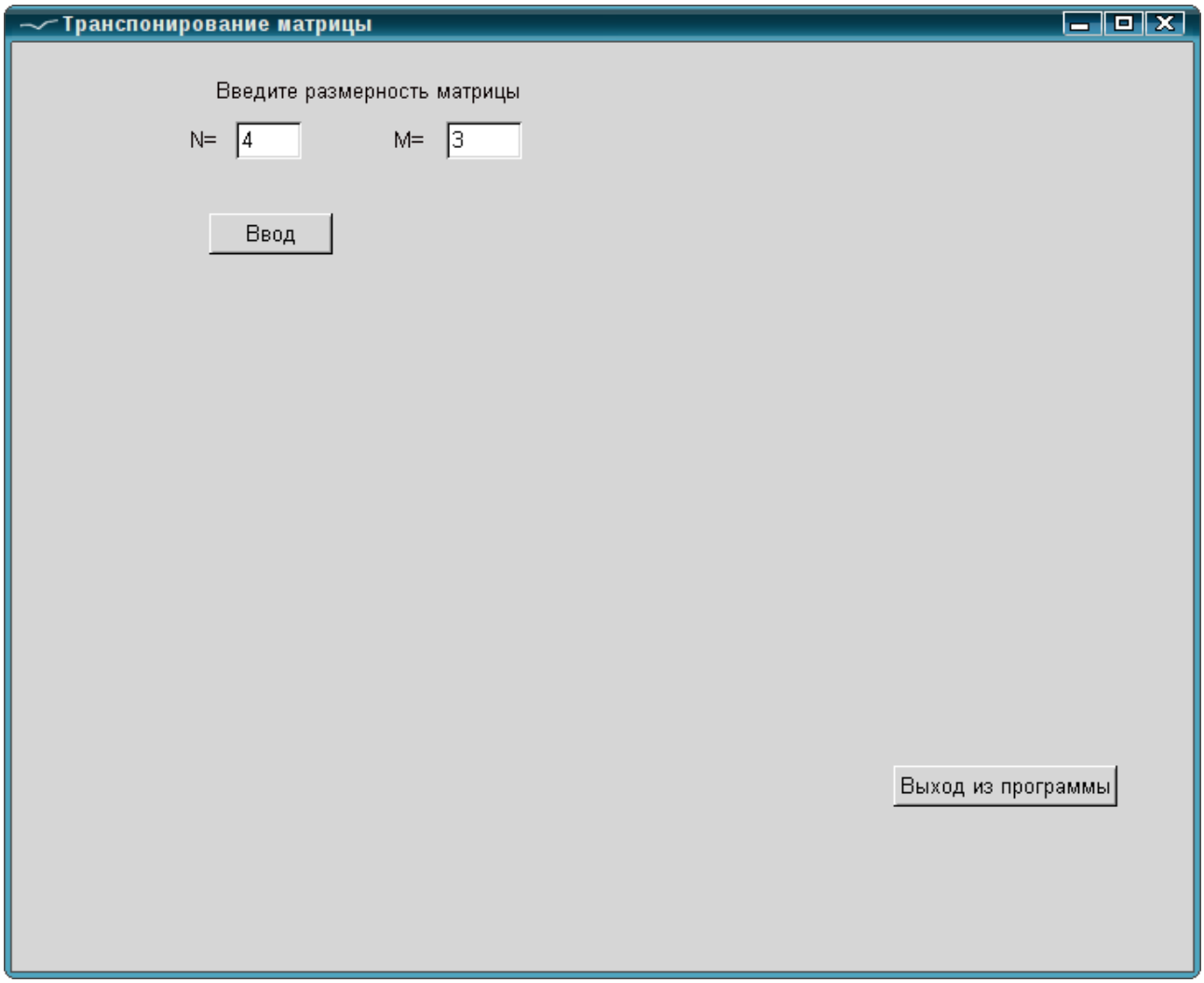

Рисунок 6.10: Стартовое окно программы транспонирования матрицы A(N,M)

```
type
     { TForm1 }
{Описание формы}
 TForm1 = class(TForm)
     Button1: TButton;
     Button2: TButton;
     Button3: TButton;
     Button4: TButton;
     Edit1: TEdit;
     Edit2: TEdit;
     Label1: TLabel;
     Label2: TLabel;
     Label3: TLabel;
     Label4: TLabel;
```

```
    Label5: TLabel;
    StringGrid1: TStringGrid;
    StringGrid2: TStringGrid;
 private
 { private declarations }
 public
 { public declarations }
end;
var
{Матрицы A,B}
    A,B:array[1..25,1..25] of integer;
{и их размеры}
    N,M:integer;
    Form1: TForm1;
```

Обработчик кнопки `Выход из программы` стандартен и представлен ниже.

```
procedure TForm1.Button4Click(Sender: TObject);
begin
 Close;
end;
```

Теперь напишем обработчик кнопки `Ввод`, который должен вводить и проверять корректность введения размеров матрицы, устанавливать свойства компонентов `StringGrid1` и `StringGrid2` (количество строк и столбцов), делать видимым компонент `StringGrid1`, кнопку `Транспонирование`, невидимыми компоненты, отвечающие за ввод размеров матрицы (метки `label1`, `label2`, `label3`, поля ввода `Edit1` и `Edit2`, кнопку `Ввод`).

```
procedure TForm1.Button1Click(Sender: TObject);
var i:byte;kod_n,kod_m,kod:integer;
begin
//Ввод размерности матрицы.
//Символьная информация преобразовывается
//в числовую и записывается в
 Val(Edit1.Text,N,kod_m); //переменную M
 Val(Edit2.Text,M,kod_n); //и переменную N.
//Если преобразование прошло успешно и
//введенные размеры удовлетворяют описанию
```

```
   //матриц A и B,
    if  (kod_n=0)  and  (kod_m=0)  and  (N>0)  and
(N<26) and (M>0) and (M< 26) then
   //то
    begin
   //визуализируется первая матрица,
       StringGrid1.Visible:=true;
       //соответствующая ей надпись,
       Label4.Visible:=true;
       Button2.Visible:=true;//кнопки Очистить
       //и транспонирование.
       Button3.Visible:=true;
       with StringGrid1 do
       begin
   //Определяем число строк (RowCount)
   //и столбцов(ColCount) в
   //компоненте StringGrid1.
           ColCount:=M+1;
           RowCount:=N+1;
   //и нумеруются строки и столбцы матрицы
           for i:=1 to RowCount-1 do
               Cells[0,i]:=IntToStr(i);
           for i:=1 to ColCount-1 do
               Cells[i,0]:=IntToStr(i);
       end;
   StringGrid2.ColCount:=N+1;
   StringGrid2.RowCount:=M+1;
   end
    else
    begin
   //При некорректном вводе выдается
   //соответствующее сообщение.
    MessageDlg('Размеры матрицы введены неверно!',
                          MtInformation,[mbOk],0);
   //Устанавливаются стартовые параметры
   //в поля ввода.
    Edit1.Text:='4';
    Edit2.Text:='3';
```

```
end;
end;
```

Теперь напишем обработчик кнопки Транспонирование. При щелчке по этой кнопке становится видимым компонент StrigGrid2, предназначенный для хранения транспонированной матрицы B, соответствующая ему надпись (label5), формируется матрица B. Матрица B выводится в компонент StringGrid2. Кнопка Ввод становится невидимой. Текст обработчика приведен ниже.

```
procedure TForm1.Button2Click(Sender: TObject);
var i,j:integer;
begin
//визуализируется вторая матрица,
//соответствующая ей надпись,
StringGrid2.Visible:=true;
label5.Visible:=true;
for i:=1 to N do       //Цикл по номерам строк.
for j:=1 to M do   //Цикл по номерам столбцов.
//Считывание элементов матрицы A
//из компонента StringGrid1.
A[i,j]:=StrToInt(StringGrid1.Cells[j,i]);
with StringGrid2 do
begin
for i:=1 to RowCount-1 do  //нумеруются строки
Cells[0,i]:=IntToStr(i);
//и столбцы компонента StringGrid2, в
//котором отображается матрица B.
for i:=1 to ColCount-1 do
Cells[i,0]:=IntToStr(i);
end;
//Формирование транспонированной матрицы B.
for i:=1 to N do       //Цикл по номерам строк.
for j:=1 to M do   //Цикл по номерам столбцов.
B[j,i]:=A[i,j];
//Элементы матрицы B выводятся
//в ячейки таблицы на форме.
for i:=1 to n do       //Цикл по номерам строк.
for j:=1 to m do   //Цикл по номерам столбцов.
//Обращение к элементам матрицы
```

```
//происходит по столбцам.
StringGrid2.Cells[i,j]:=IntToStr(B[j,i]);
Buuton1.Visible:=False; end;
```

Осталось написать обработчик события, которое произойдет при нажатии на кнопку Очистить. При щелчке по этой кнопке должны выполниться следующие действия:

- очистка содержимого компонентов StringGrid1, StringGrid2;
- компоненты StringGrid1, StringGrid2 и соответствующие им метки labe4 и label5, а также кнопки Транспонировать и Очистить должны стать невидимыми;
- становится видимой кнопка Ввод;
- в поля ввода записываются начальные значения размеров матрицы (N=4, M=3).

Текст обработчика кнопки Очистить с комментариями приведен ниже:

```
procedure TForm1.Button3Click(Sender: TObject);
var i,j:integer;
begin
//Очистка компонента StringGrid1.
 with StringGrid1 do
     for i:=1 to RowCount-1 do
     for j:=1 to ColCount-1 do
         Cells[j,i]:='';
//Очистка компонента StringGrid2.
 with StringGrid2 do
     for i:=1 to RowCount-1 do
     for j:=1 to ColCount-1 do
     Cells[j,i]:='';
//Делаем невидимыми компоненты
//StringGrid1, StringGrid2,
//labe4, label5.
 StringGrid1.Visible:=False;
 StringGrid2.Visible:=False;
 label4.Visible:=False;
 label5.Visible:=False;
//Делаем невидимыми кнопки «Транспонировать»
```

```
//и «Очистить».
 Button2.Visible:=False;
 Button3.Visible:=False;
//Делаем видимой кнопку «Ввод».
 Button1.Visible:=True;
//Запись начальных значений
//размеров матрицы (N=4, M=3).
 Edit1.Text:='4';
 Edit2.Text:='3';
end;
```

Получили работающую программу для транспонирования матрицы. На рис. 6.11 представлены результаты транспонирования матрицы `A(2,4)`.

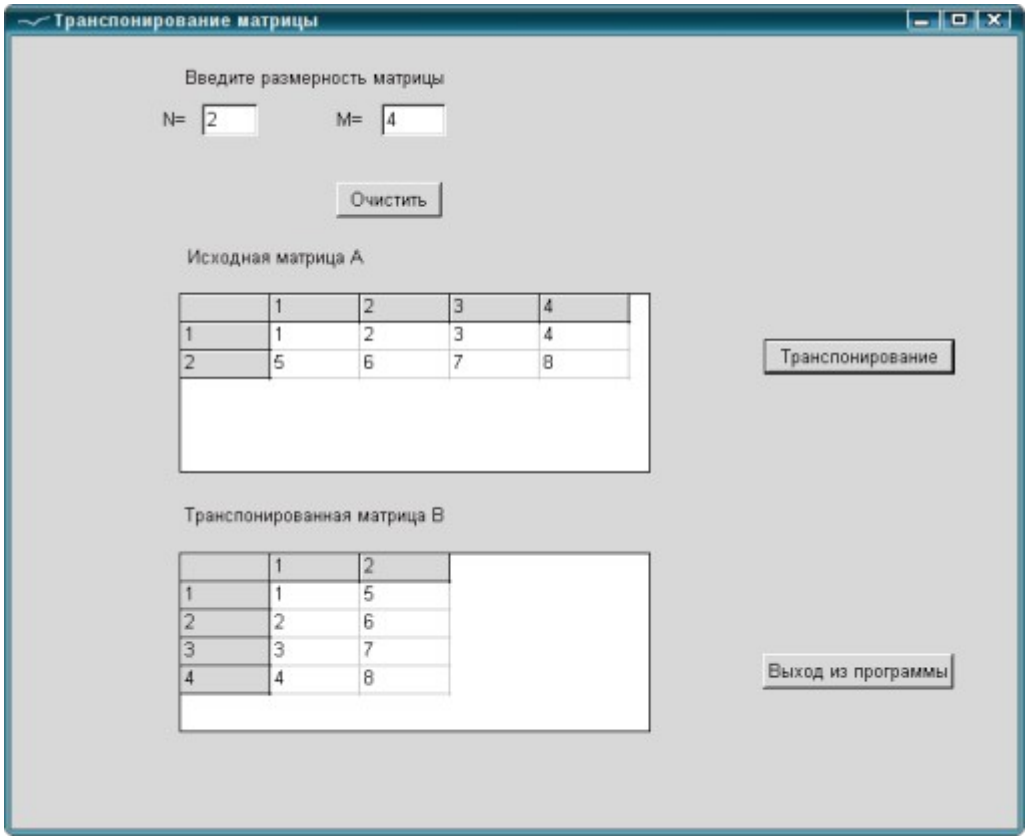

Рисунок 6.11: Транспонирование матрицы A(2,4)

Обратите внимание на использование оператора присоединения

```
with имя_компонента do оператор;
```

который упрощает доступ к свойствам компонента. Внутри оператора `With` имя компонента для обращения к его свойствам можно не использовать.

Например, для очистки элементов матрицы `A` вместо операторов

```
for i:=1 to StringGrid1.RowCount-1 do
    for j:=1 to StringGrid1.ColCount-1 do
        StringGrid1.Cells[j,i]:='';
```

был использован оператор

```
with StringGrid1 do
 for i:=1 to RowCount-1 do
    for j:=1 to ColCount-1 do
                Cells[j,i]:='';
```

Рассмотрим несколько задач обработки матриц. Для их решения напомним читателю некоторые свойства матриц (рис. 6.12):

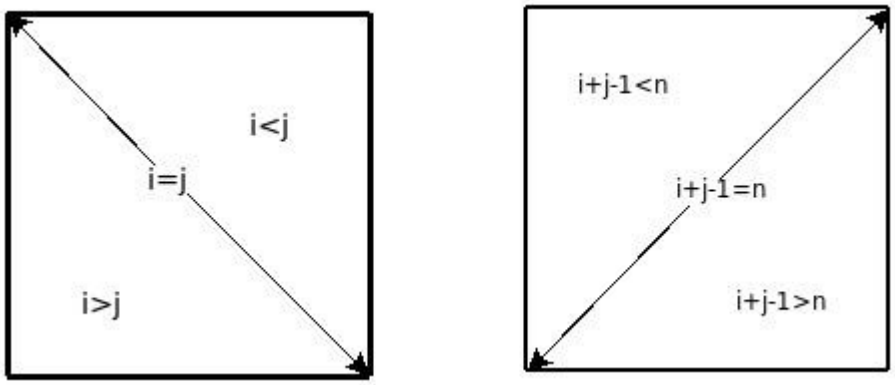

Рисунок 6.12: Свойства элементов матрицы

- если номер строки элемента совпадает с номером столбца (`i=j`), это означает что элемент лежит на главной диагонали матрицы;
- если номер строки превышает номер столбца (`i>j`), то элемент находится ниже главной диагонали;
- если номер столбца больше номера строки (`i<j`), то элемент находится выше главной диагонали;
- элемент лежит на побочной диагонали, если его индексы удовлетворяют равенству `i+j-1=n`;
- неравенство `i+j-1<n` характерно для элемента, находящегося выше побочной диагонали;
- соответственно, элементу, лежащему ниже побочной диагонали, соответствует выражение `i+j-1>n`.

6.2 Алгоритмы и программы работы с матрицами

Рассмотри несколько примеров решения задач обработки матриц.

ЗАДАЧА 6.3. Найти сумму элементов матрицы, лежащих выше главной диагонали (см. рис. 6.13).

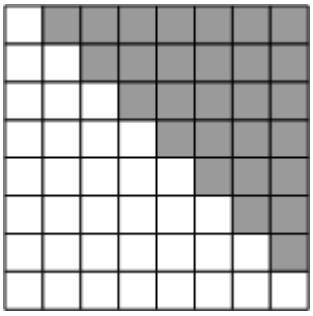

Рисунок 6.13: Иллюстрация к задаче 6.3

Рассмотрим два алгоритма решения данной задачи. *Первый алгоритм* (см. рис. 6.14) построен следующим образом: вначале переменная S для накапливания суммы обнуляется (`S:=0`). Затем с помощью двух циклов (первый по строкам, второй по столбцам) перебираются все элементы матрицы, но накапливание суммы происходит только в том случае, если этот элемент находится выше главной диагонали (если выполняется свойство `i<j`).

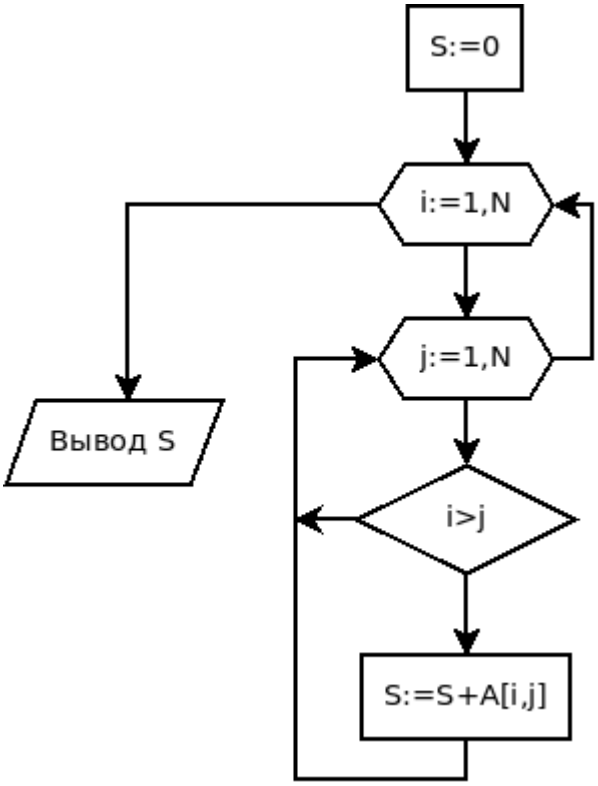

Рисунок 6.14: Блок-схема задачи 6.3 (алгоритм 1)

Текст консольного приложения с комментариями:

```
var
a:array [1..15,1..10]
                  of real;
i,j,n,m: integer;
s: real;

begin
write('Введите размеры');
writeln('матрицы');
write('n - количество');
writeln('строк: ');
readln (n);
write('m — количество');
writeln('столбцов: ');
readln (m);
writeln('Матрица A:');

for i:=1 to n do
    for j:=1 to m do
        read(a[i,j]);
s:=0;
for i:=1 to n do
    for j:=1 to m do
   {Если элемент лежит выше главной диагонали,то}
    if j>i then
        s:=s+a[i,j];{ накапливаем сумму. }
```

```
writeln('матрица A');
for i:=1 to n do
begin
 for j:=1 to m do
{Здесь формат важен,
особенно общая ширина поля!}
    write(a[i,j]:8:3,' ');
 writeln
end;
writeln('сумма элементов  матрицы', s:8:3);
end.
```

Результаты работы программы представлены на рис. 6.15.

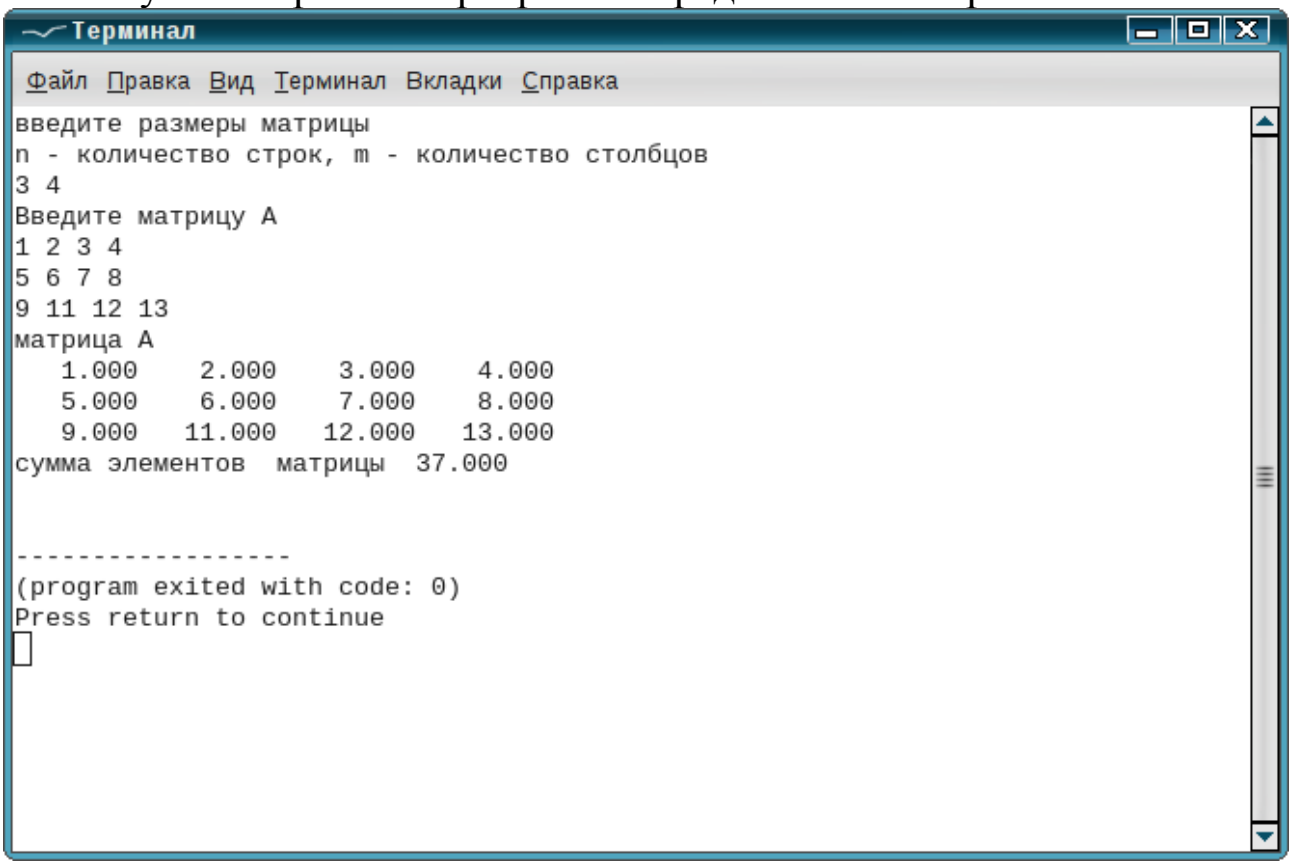

Рисунок 6.15: Результаты работы программы решения задачи 6.3

Второй алгоритм решения этой задачи представлен на рис. 6.16.

В нем проверка условия `i<j` не выполняется, но, тем не менее, в нем так же суммируются элементы матрицы, находящиеся выше главной диагонали. Для пояснения функционирования алгоритма обратимся к рисунку 6.13.

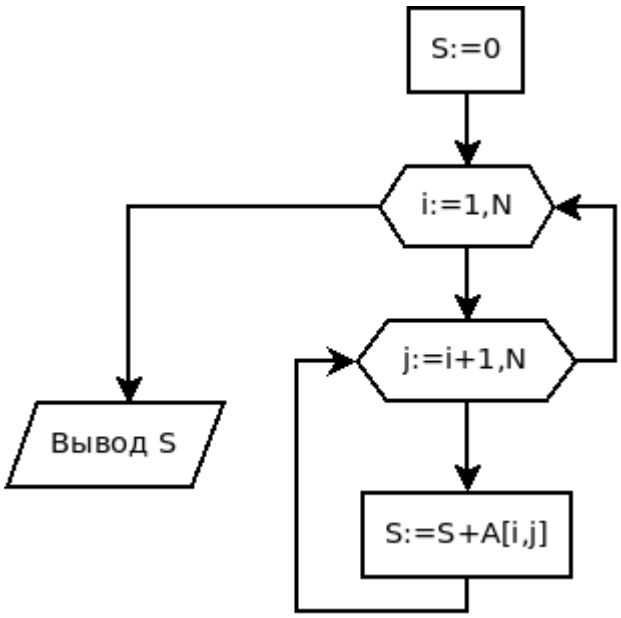

Рисунок 6.16: Блок-схема задачи 6.3 (алгоритм 2)

В первой строке заданной матрицы необходимо сложить все элементы, начиная со второго. Во второй – все, начиная с третьего, в `i`–й строке процесс суммирования начнется с (`i+1`)–го элемента и так далее. Таким образом, первый цикл работает от `1` до `N`, а второй от `i+1` до `M`.

Предлагаем читателю самостоятельно составить программу, соответствующую описанному алгоритму.

ЗАДАЧА 6.4. Вычислить количество положительных элементов квадратной матрицы A, расположенных по ее периметру и на диагоналях.

В квадратной матрице число строк равно числу столбцов. Прежде чем приступить к решению задачи, рассмотрим рисунок 6.17, на котором изображена схема диагоналей квадратных матриц различной размерности.

Из рисунка видно, что нет необходимости рассматривать все элементы матрицы. Достаточно рассмотреть элементы, расположенные в первой и последней строках, в первом и последнем столбцах, а также на диагоналях квадратной матрицы. Все эти элементы отмечены на рис. 6.17, причем черным цветом выделены элементы, которые принадлежат строкам, столбцам и диагоналям. Например, элемент $A_{1,1}$ принадлежит первой строке, первому столбцу и главной диагонали матрицы, элемент $A_{N,N}$ находится в последней строке, последнем столбце и принадлежит главной диагонали. Кроме того, если `N` – число нечетное (на рисунке 6.17 эта матрица расположена слева), то существует элемент с индексом (`N div 2 +1, N div 2 +1`), который находится на пересечении главной и побочной диагоналей. При четном значении `N` (матрица справа на рис. 6.17) диагонали не пересекаются.

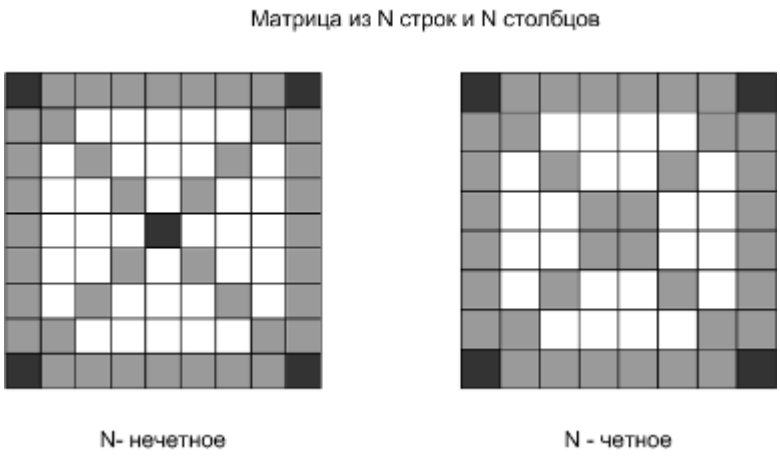

Рисунок 6.17: Рисунок к условию задачи 6.4

Рассмотрим алгоритм решения задачи. Для обращения к элементам главной диагонали вспомним, что номера строк этих элементов всегда равны номерам столбцов. Поэтому, если параметр `i` изменяется циклически от `1` до `N`, то $A_{i,i}$ — элементы расположенные на главной диагонали. Воспользовавшись свойством, характерным для элементов побочной диагонали, получим:

`i+j-1=N → j=N-i+1,`

следовательно, для строк `i=1,2,…,N` элемент $A_{i,N-i+1}$ — элемент побочной диагонали. Элементы, находящиеся по периметру матрицы, записываются следующим образом: $A_{1,i}$ — элементы, расположенные в первой строке (`i=1,2,…,N`), $A_{N,i}$ — элементы, расположенные в последней строке (`i=1,2,…,N`) и соответственно $A_{i,1}$ — элементы, расположенные в первом столбце (`i=1,2,…,N`), $A_{i,N}$ — в последнем столбце (`i=1,2,…,N`).

Алгоритм обработки построим следующим образом, сначала обработаем элементы, расположенные на диагоналях квадратной матрицы. Для этого необходимо в каждой строке (`i=1,2,…,N`) проверять знак элементов $A_{i,i}$ и $A_{i,N-i+1}$.

```
for i:=1 to N do
 begin
     if (a[i,i]>0) then     k:=k+1;
     if a[i,N-i+1]>0 then   k:=k+1;
 end;
```

Так как при проверке диагональных элементов матрицы угловые элементы были учтены, то при обработке элементов, расположенных по периметру матрицы, угловые элементы учитывать не нужно.

Поэтому надо перебрать элементы со второго до предпоследнего в первой и последней строках, в первом и последнем столбцах.

```
for i:=2 to N-1 do
 begin
{ Если элемент находится в первой строке. }
     if (a[1,i]>0) then        k:=k+1;
{ Если элемент находится в последней строке. }
     if (a[N,i]>0) then        k:=k+1;
{ Если элемент находится в первом столбце. }
     if (a[i,1]>0) then        k:=k+1;
{ Если элемент находится в последнем столбце. }
     if (a[i,N]>0) then        k:=k+1;
end;
```

Затем надо проверить, не был ли элемент, находящийся на пересечении диагоналей, подсчитан дважды. Это могло произойти только в том случае, если `N` – нечетно и элемент, расположенный на пересечении диагоналей[70], положителен.

```
if (N mod 2 <>0) and (a[n div 2+1,n div 2+1]>0)
then k:=k-1;
```

Ниже приведен полный текст консольного приложения решения задачи 6.4 с комментариями. На рис. 6.18 представлены результаты работы программы решения задачи 6.4.

```
var
     a:array [1..10,1..10] of integer;
     i,j,N,k:integer;
begin
     write('N=');
     readln(N);
//Ввод исходной матрицы.
 writeln('Введите матрицу A');
 for i:=1 to N do
     for j:=1 to N do
         read(a[i,j]);
//Вывод исходной матрицы.
 writeln('Была введена матрица A:');
```

70 На пересечении диагоналей находится элемент с индексами (`N div 2 +1, N div 2 +1`)

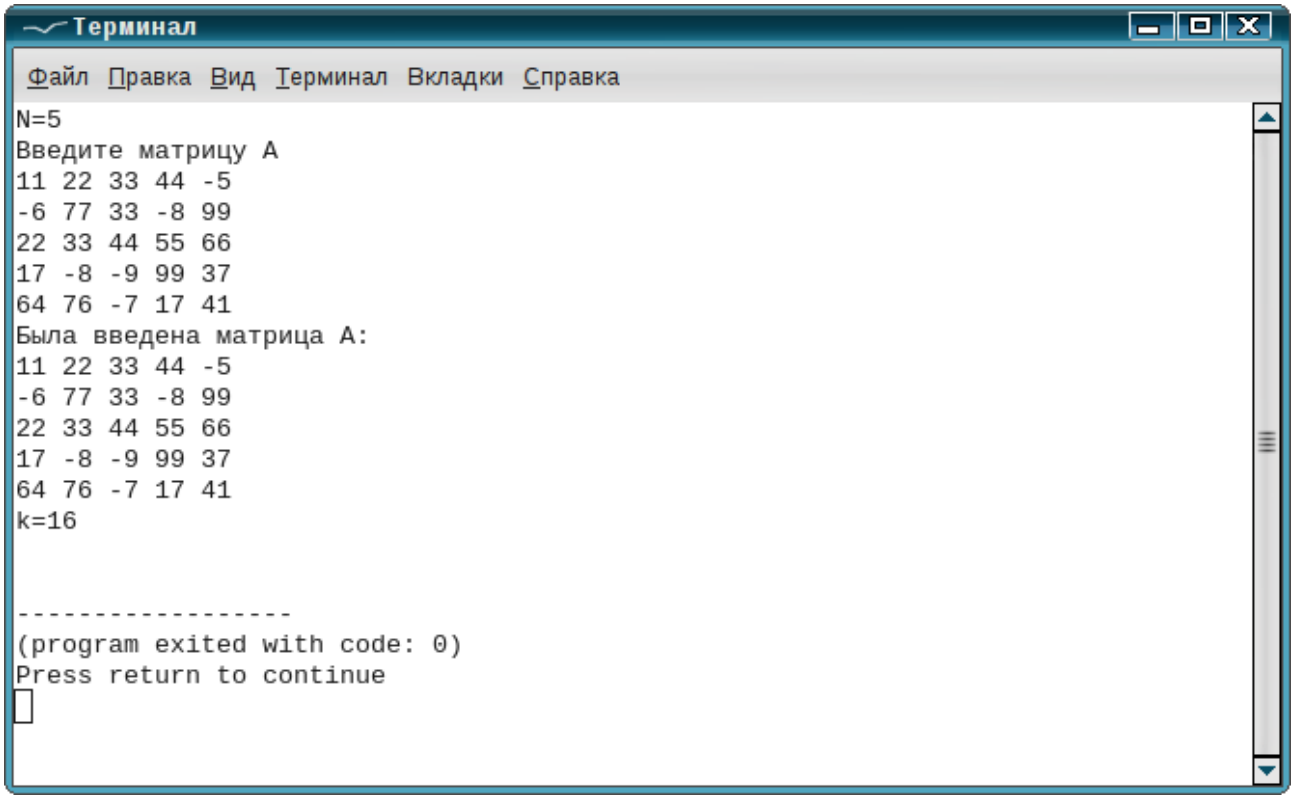

Рисунок 6.18: Результаты решения задачи 6.4

```
 for i:=1 to N do
 begin
     for j:=1 to N do
         write(a[i,j],' ');
     writeln;
 end;
 k:=0;
//Обработка элементов, расположенных
//на диагоналях матрицы.
 for i:=1 to N do
 begin
     if (a[i,i]>0) then    k:=k+1;
     if a[i,N-i+1]>0 then k:=k+1;
end;
//Обработка элементов, расположенных
//по периметру матрицы.
 for i:=2 to N-1 do
 begin
     if (a[1,i]>0) then k:=k+1;
     if (a[N,i]>0) then k:=k+1;
     if (a[i,1]>0) then k:=k+1;
```

```
        if (a[i,N]>0) then k:=k+1;
    end;
    { Если элемент, находящийся на пересечении диа-
гоналей, подсчитан дважды,  то уменьшить вычислен-
ное значение k на один. }
     if (n mod 2<>0) and (a[N div 2+1,N div 2+1]>0)
        then
                k:=k-1;
     writeln('k=',k);
    end.
```

ЗАДАЧА 6.5. Проверить, является ли заданная квадратная матрица единичной.

Единичной называют матрицу, у которой элементы главной диагонали – единицы, а все остальные – нули. Например,

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Решать задачу будем так. Предположим, что матрица единичная, (`pr:=true`) и попытаемся доказать обратное. В двойном цикле по строкам (`i:=1,2,...,N`) и по столбцам (`j:=1,2,...,N`) перебираем все элементы матрицы. Если диагональный элемент ($i=j$) не равен единице или элемент, расположенный вне диагонали ($i\neq j$), не равен нулю[71], то в логическую переменную `pr` записываем значение `false` и прекращаем проверку (аварийно покидаем цикл). После цикла проверяем значение `pr`, если переменная `pr` попрежнему равна `true`, то матрица единична, иначе она такой не является.

```
    var a:array[1..10,1..10] of real;
     i,j,n:integer;
     pr:boolean;
    begin
     writeln('Введите размер матрицы');
     readln(n);
     writeln('Введите матрицу');
     for i:=1 to n do
```

71 Воспользовавшись логическими операциями **and** и **or,** это сложное условие можно записать так *if ((i=j) and (a[i,j]<>1)) or ((i<>j) and (a[i,j]<>0)) then*

```
        for j:=1 to n do
        read(a[i,j]);
    {Предположим, что матрица единичная,
     и присвоим логической переменной значение
     истина. Если значение этой переменной при
     выходе из цикла не изменится, это будет
     означать, что матрица единичная.}
     pr:=true;
     for i:=1 to n do
        for j:=1 to n do
        if ((i=j) and (a[i,j]<>1)) or ((i<>j)
                          and (a[i,j]<>0))  then
    {Если элемент лежит на главной диагонали и не
равен единице или элемент лежит вне главной диаго-
нали и не равен нулю , то }
    begin
    {логической переменной присвоить значение ложь,
    это будет означать, что матрица не единичная.}
            pr:=false;
    { выйти из цикла. }
            break;
        end;
    {Проверка значения логической переменной
     и печать результата.}
    if pr then
        writeln('Матрица единичная')
    else
        writeln('Матрица не является единичной');
    end.
```

ЗАДАЧА 6.6. Преобразовать исходную матрицу так, чтобы последний элемент каждого столбца был заменен разностью минимального и максимального элемента в этом же столбце.

Для решения данной задачи необходимо в каждом столбце найти максимальный и минимальный элементы, после чего в последний элемент столбца записать их разность. Блок-схема алгоритма решения приведена на рис. 6.19.

Ниже приведен текст консольного приложения с комментариями.

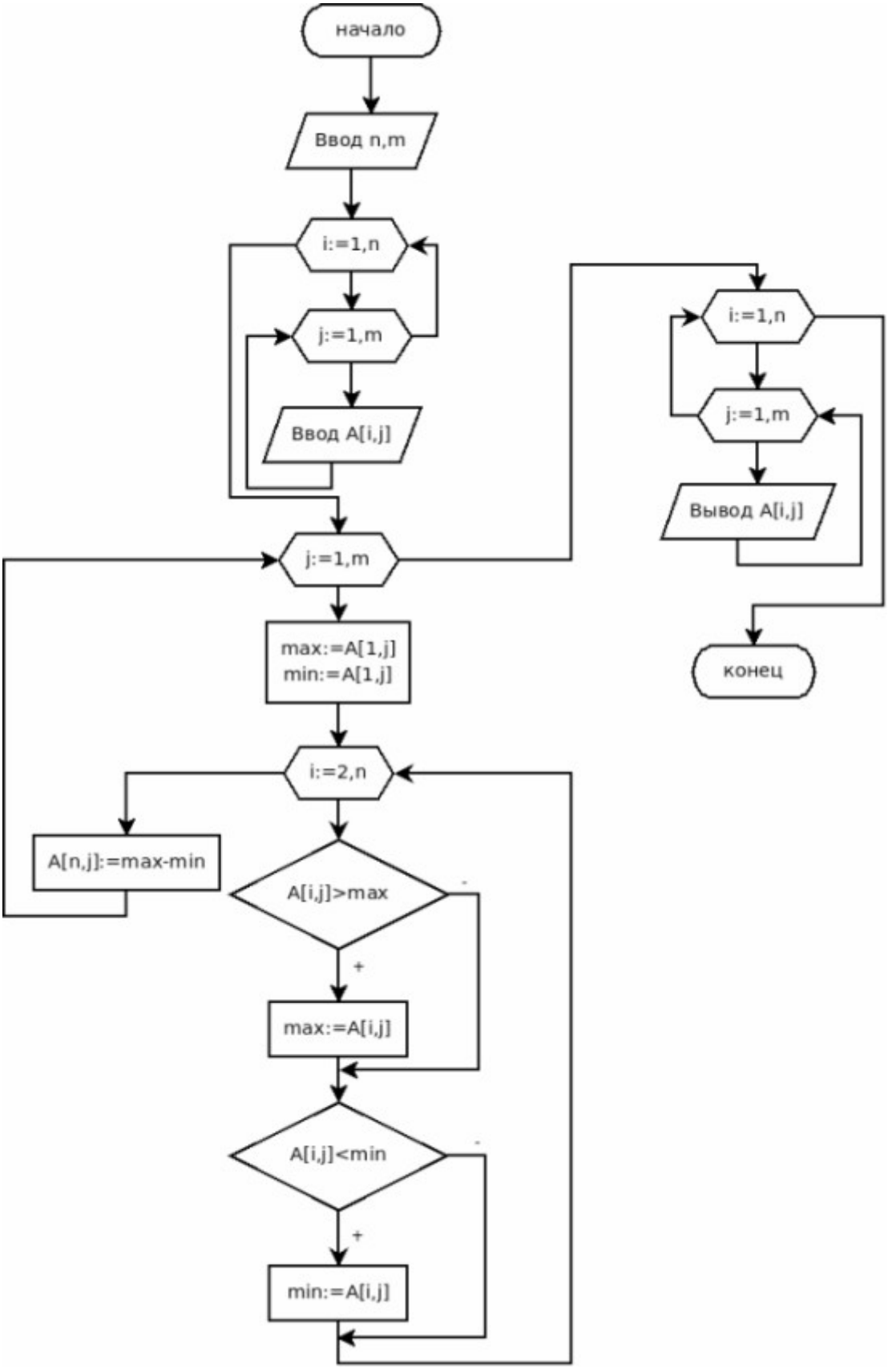

Рисунок 6.19: Блок-схема алгоритма решения задачи 6.6

```
    var a:array[1..25,1..25] of real;
    i,j,n,m:integer;
    max,min:real;
    begin
   {Ввод размеров матрицы.}
    writeln('Введите размеры матрицы');
    readln(n,m);
   {Ввод исходной матрицы.}
    writeln('Введите матрицу');
    for i:=1 to n do
        for j:=1 to m do
            read(a[i,j]);
   {Последовательно перебираем все столбцы матрицы.}
    for j:=1 to m do
    begin
   {Максимальным и минимальным объявляем первый элемент текущего (j-го) столбца матрицы.}
        max:=a[1,j];
        min:=a[1,j];
   {Последовательно перебираем все элементы в текущем (j-м) столбце матрицы.}
        for i:=2 to n do
        begin
   {Если текущий элемент больше максимального, то его и объявляем максимальным.}
            if a[i,j]>max then max:=a[i,j];
   {Если текущий элемент меньше минимального, то его и объявляем минимальным.}
            if a[i,j]<min then min:=a[i,j];
        end;
   {В последний элемент столбца записываем разность между максимальным и минимальным элементами столбца.}
        a[n,j]:=max-min;
    end;
   {Вывод преобразованной матрицы.}
    writeln('Преобразованная матрица');
```

```
    for i:=1 to n do
        begin
            for j:=1 to m do
                write(a[i,j]:7:3,' ');
            writeln;
        end;
end.
```

Теперь давайте создадим визуальное приложение, реализующее рассмотренный алгоритм. За основу возьмем форму (рис. 6.9) и проект транспонирования матрицы `A(N,M)`, разработанные для задачи 6.2. Окно формы несколько изменим. Кнопку `Транспонирование` переименуем в `Преобразование матрицы`, а свойство `Caption` метки `label5` установим `Преобразованная матрица A`. Кроме того изменим свойство `Caption` формы. Окно измененной формы представлено на рис. 6.20.

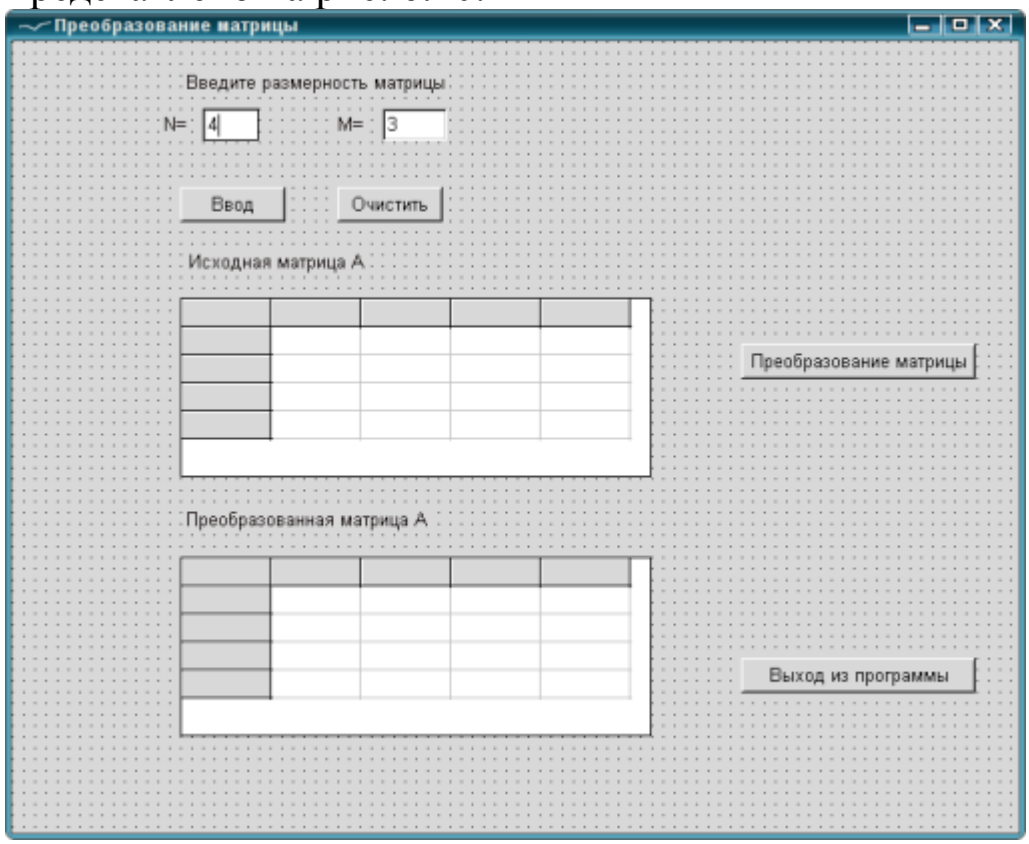

Рисунок 6.20: Окно формы решения задачи 6.6

Обработчики кнопок `Ввод`, `Очистить`, `Выход из программы` изменятся мало. Рассмотрим алгоритм работы обработчика кнопки `Преобразование матрицы`. В этом обработчике будет считывание матрицы из компонента `StringGrid1`, преобразование матрицы `A`

по алгоритму, представленному на рис. 6.19, вывод преобразованной матрицы в компонент StringGrid2.

Текст модуля визуального приложения решения задачи 6.6 с комментариями приведен ниже.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes,    SysUtils,    LResources,    Forms,
Controls, Graphics, Dialogs, StdCtrls,Grids;
//Описание формы
type
 { TForm1 }
 TForm1 = class(TForm)
 Button1: TButton;
 Button2: TButton;
 Button3: TButton;
 Button4: TButton;
 Edit1: TEdit;
 Edit2: TEdit;
 Label1: TLabel;
 Label2: TLabel;
 Label3: TLabel;
 Label4: TLabel;
 Label5: TLabel;
 StringGrid1: TStringGrid;
 StringGrid2: TStringGrid;
 procedure Button1Click(Sender: TObject);
 procedure Button2Click(Sender: TObject);
 procedure Button3Click(Sender: TObject);
 procedure Button4Click(Sender: TObject);
private
 { private declarations }
public
 { public declarations }
 end;
var
 A:array[1..25,1..25] of real;
```

```
    N,M:integer;
    Form1: TForm1;
   implementation
   { TForm1 }
   //Обработчик кнопки «Выход из программы».
   procedure TForm1.Button4Click(Sender: TObject);
   begin
    Close;
   end;
   //Обработчик первой кнопки — кнопки
   //ввода размерности матрицы.
   procedure TForm1.Button1Click(Sender: TObject);
   var i:byte;kod_n,kod_m,kod:integer;
   begin
   //Ввод размерности матрицы.
   //Символьная информация преобразовывается
   //в числовую и записывается в переменные N и M.
    Val(Edit1.Text,N,kod_m);
    Val(Edit2.Text,M,kod_n);
    //Если преобразование прошло успешно
   //и введенные размеры удовлетворяют
   //писанию матриц A и B,
    if  (kod_n=0)  and  (kod_m=0)  and  (N>0)  and
(N<26) and (M>0) and (M< 26) then
   //то
    begin
   //визуализируется первая матрица,
        StringGrid1.Visible:=true;
   //соответствующая ей надпись,
        Label4.Visible:=true;
   //кнопки Очистить
        Button2.Visible:=true;
   //и преобразование матрицы.
        Button3.Visible:=true;
        with StringGrid1 do
        begin
            ColCount:=M+1;
            RowCount:=N+1;
```

```
    //и нумеруются строки и
            for i:=1 to RowCount-1 do
                Cells[0,i]:=IntToStr(i);
    //столбцы первой таблицы.
            for i:=1 to ColCount-1 do
                Cells[i,0]:=IntToStr(i);
        end;
        StringGrid2.ColCount:=M+1;
        StringGrid2.RowCount:=N+1;
     end
     else
     begin
    //При некорректном вводе выдается
    //соответствующее сообщение.
        MessageDlg('Размеры матрицы введены не
верно!',MtInformation,[mbOk],0);
    //Устанавливаются стартовые параметры
    //в поля ввода.
        Edit1.Text:='4';
        Edit2.Text:='3';
     end; end;
    //Обработчик кнопки «Преобразование матрицы».
    procedure TForm1.Button2Click(Sender: TObject);
    var i,j:integer;
     max,min:real;
    begin
     StringGrid2.Visible:=true;
     label5.Visible:=true;
     for i:=1 to N do       //Цикл по номерам строк.
     for j:=1 to M do   //Цикл по номерам столбцов.
    //Считывание элементов матрицы A
    //из компонента StringGrid1.
     A[i,j]:=StrToFloat(StringGrid1.Cells[j,i]);
     with StringGrid2 do
     begin
        for i:=1 to RowCount-1 do //Нумеруются
            Cells[0,i]:=IntToStr(i);  //строки и
                          //столбцы 2-й матрицы.
```

```
        for i:=1 to ColCount-1 do
            Cells[i,0]:=IntToStr(i);
    end;
   //Решение задачи 6.6.
    for j:=1 to m do
    begin
   {Максимальным и минимальным объявляем первый
элемент текущего (j-го) столбца матрицы.}
        max:=a[1,j];
        min:=a[1,j];
   {Последовательно перебираем все элементы в те-
кущем (j-м) столбце матрицы.}
        for i:=2 to n do
        begin
   {Если текущий элемент больше максимального, то
его и объявляем максимальным.}
            if a[i,j]>max then max:=a[i,j];
   {Если текущий элемент меньше минимального, то
его и объявляем минимальным.}
            if a[i,j]<min then min:=a[i,j];
        end;
   {В последний элемент столбца записываем раз-
ность между максимальным и минимальным элементами
столбца.}
    a[n,j]:=max-min;
    end;
   //Элементы преобразованной матрицы A выводятся
в ячейки
   //таблицы на форме.
    for i:=1 to N do       //Цикл по номерам строк.
    for j:=1 to M do  //Цикл по номерам столбцов.
//Запись элемента преобразованной матрицы A в
ячейку StringGrid2.

    StringGrid2.Cells[j,i]:=FloatToStr(A[i,j]);
   //Делаем первую кнопку невидимой.
    Button1.Visible:=False;
   end;
```

```
//Обработчик кнопки «Очистить».
procedure TForm1.Button3Click(Sender: TObject);
 var i,j:integer;
begin
//Очистка компонента StringGrid1.
 with StringGrid1 do
     for i:=1 to RowCount-1 do
         for j:=1 to ColCount-1 do
             Cells[j,i]:='';
//Очистка компонента StringGrid2.
 with StringGrid2 do
     for i:=1 to RowCount-1 do
         for j:=1 to ColCount-1 do
             Cells[j,i]:='';
//Делаем невидимыми компоненты
//StringGrid1, StringGrid2,labe4, label5.
 StringGrid1.Visible:=False;
 StringGrid2.Visible:=False;
 label4.Visible:=False;
 label5.Visible:=False;
//Делаем невидимыми кнопки
//«Преобразование матрицы» и «Очистить».
 Button2.Visible:=False;
 Button3.Visible:=False;
//Делаем видимой кнопку «Ввод».
 Button1.Visible:=True;
//Запись начальных значений размеров
//матрицы (N=4, M=3).
 Edit1.Text:='4';
 Edit2.Text:='3';
end;
initialization
 {$I unit1.lrs}
end.
```

На рис. 6.21 представлено окно с результатами решения задачи.

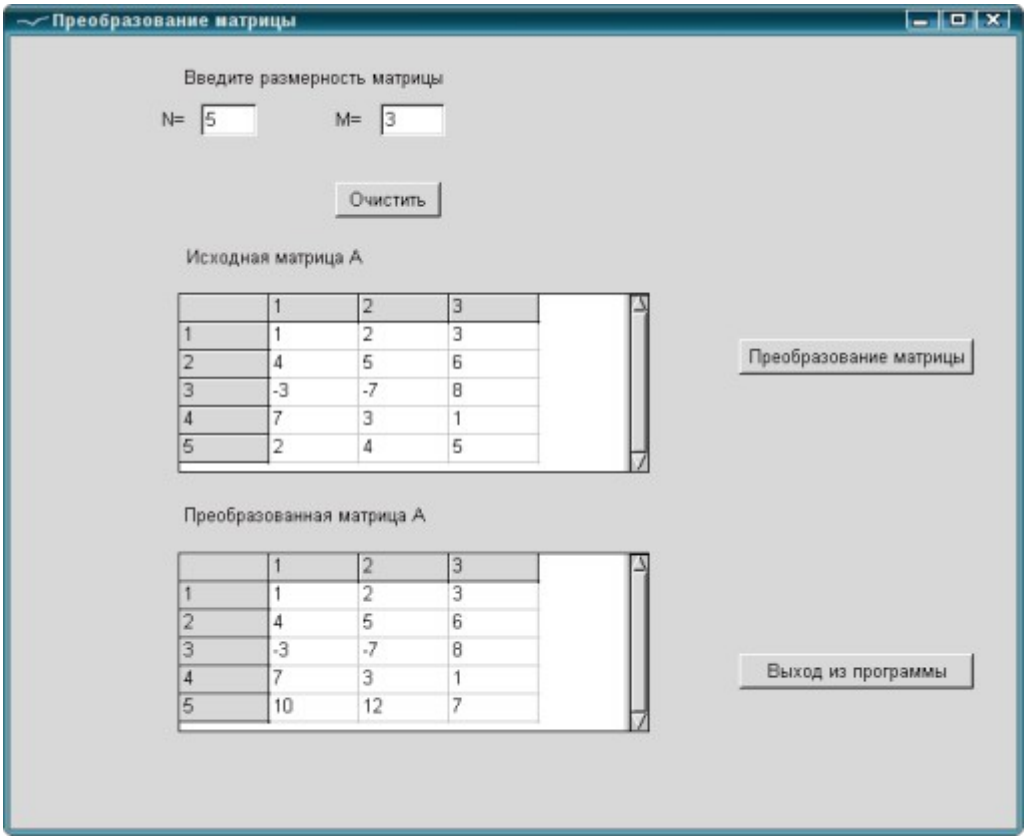

Рисунок 6.21: Результаты решения задачи 6.6

ЗАДАЧА 6.7. Поменять местами n-й и r-й столбцы матрицы A(K,M).

Задача сводится к обмену n-го и r-го элементов во всех строках матрицы. Блок-схема приведена на рис. 6.22.

Ниже приведен листинг консольного приложения с комментариями.

Результаты работы программы приведены на рис. 6.23.

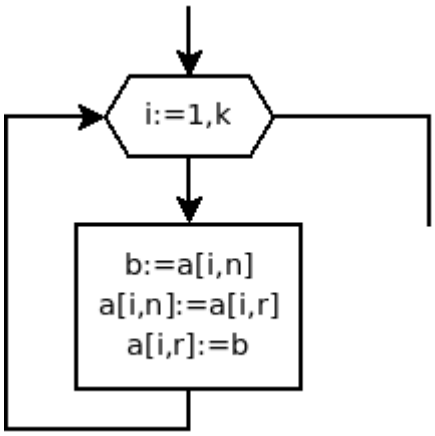

Рисунок 6.22: Блок-схема алгоритма решения задачи 6.7

```
type matrica=
  array [1..15,1..15] of real;
var
a:matrica; b:real;
i,j,k,m,n,r:byte;
begin
   //Ввод размеров матрицы.
   write ('k=');readln(k);
   write ('m=');readln(m);
   //Ввод матрицы A.
```

```
writeln('Матрица A');
for i:=1 to k do
 for j:=1 to m do read(a[i,j]);
//Ввод номеров столбцов матрицы,
//подлежащих обмену.
repeat
 write('n=');readln(n);
 write('r=');readln(r);
{Ввод считается верным, если n и r
не больше m и не равны друг другу.}
until (n<=m) and (r<=m) and (n<>r);
```

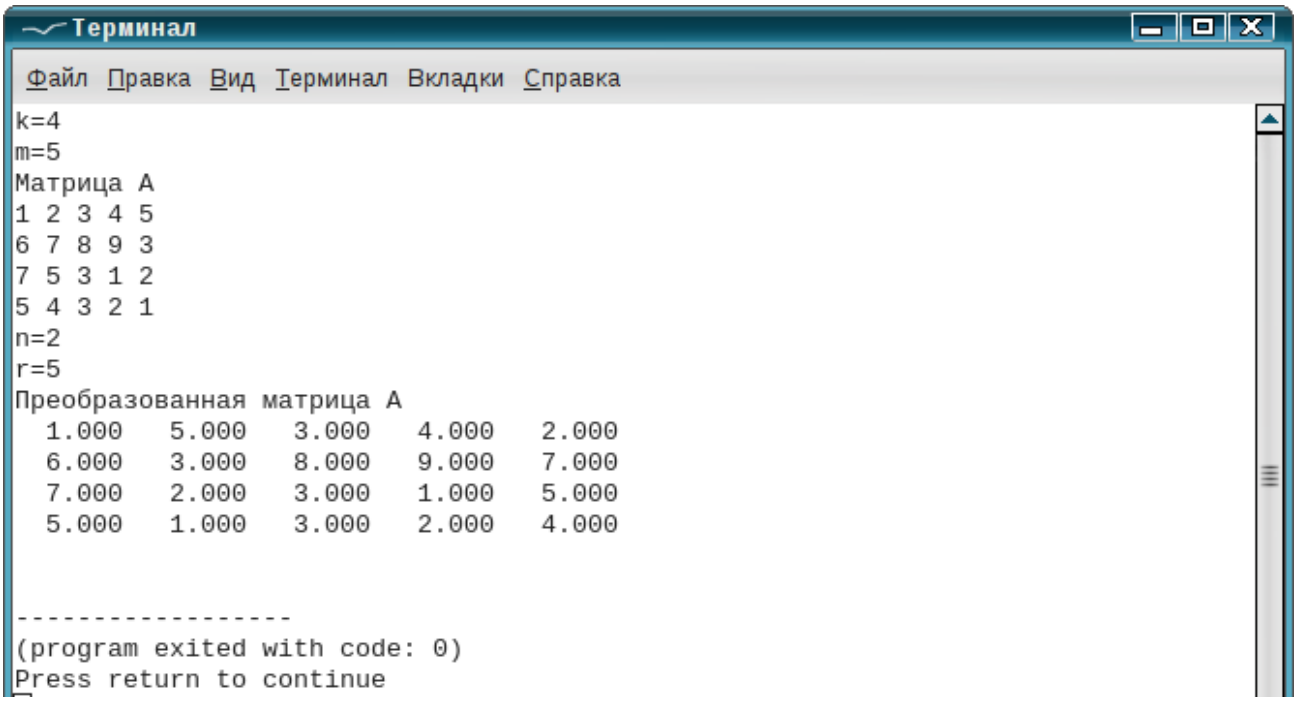

Рисунок 6.23: Результаты решения задачи 6.7

```
{Элементы столбца с номером r заменить
элементами столбца с номером n.}
for i:=1 to k do
begin
 b:=a[i,n];
 a[i,n]:=a[i,r];
 a[i,r]:=b
end;
writeln('Преобразованная матрица A');
for i:=1 to k do
begin
 for j:=1 to m do
```

```
        write(a[i,j]:7:3,' ');
     writeln;
   end;
end.
```

ЗАДАЧА 6.8. Преобразовать матрицу A(m,n) так, чтобы строки с нечетными индексами были упорядочены по убыванию, с четными – по возрастанию.

Каждая строка матрицы является одномерным массивом. Поэтому для упорядочивания строки или столбца можно использовать обычные алгоритмы сортировки массивов. При решении задачи необходимо последовательно просматривать все строки матрицы, если номер строки нечетен, то сортируем строку методом пузырька по убыванию, иначе — по возрастанию.

Блок-схема этого алгоритма представлена на рис. 6.24. Ниже представлено консольное приложение решения этой задачи с комментариями.

На рис. 6.25 приведены результаты работы программы.

```
var
 a:array [1..15,1..15] of real;
 j,i,k,m,n:byte;
b:real;
begin
//Ввод размеров матрицы.
 writeln('введите m и n');
 readln(m,n);
//Ввод матрицы.
 writeln('Матрица A');
 for i:=1 to m do
     for j:=1 to n do
         read(a[i,j]);
//Преобразование матрицы.
for i:=1 to m do
if (i mod 2)=0 then
{ Если номер строки четный, то }
begin
{ упорядочить ее элементы по возрастанию. }
```

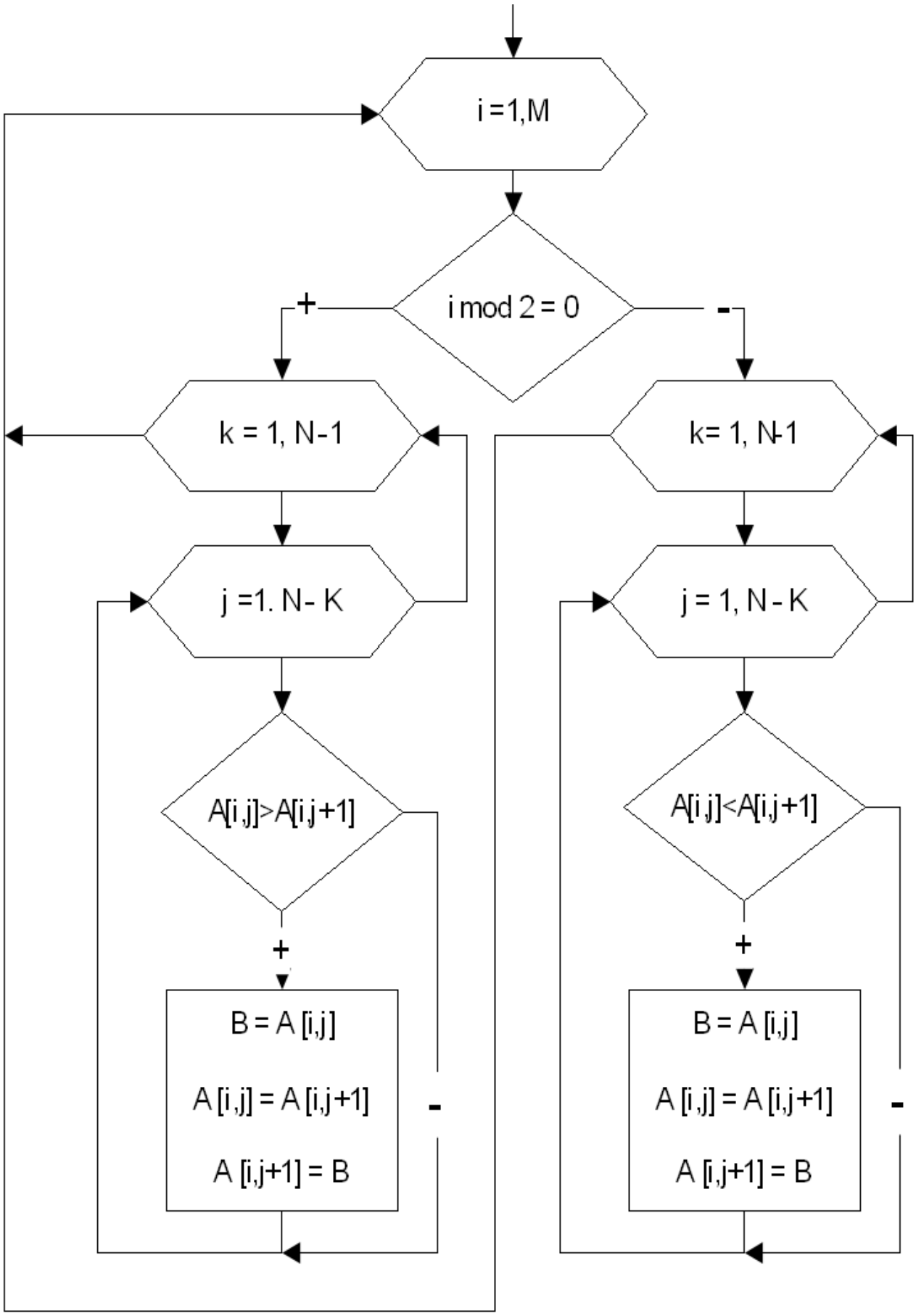

Рисунок 6.24: Блок-схема алгоритма решения задачи 6.8

```
{Упорядочивание строки матрицы методом
пузырька по возрастанию.}
     for k:=1 to n-1 do
         for j:=1 to n-k do
             if  a[i,j] > a[i,j+1] then
             begin
                 b:=a[i,j];
                 a[i,j]:=a[i,j+1];
                 a[i,j+1]:=b;
             end
end
else{ Если номер строки нечетный,
     то упорядочить ее элементы по убыванию.}
```

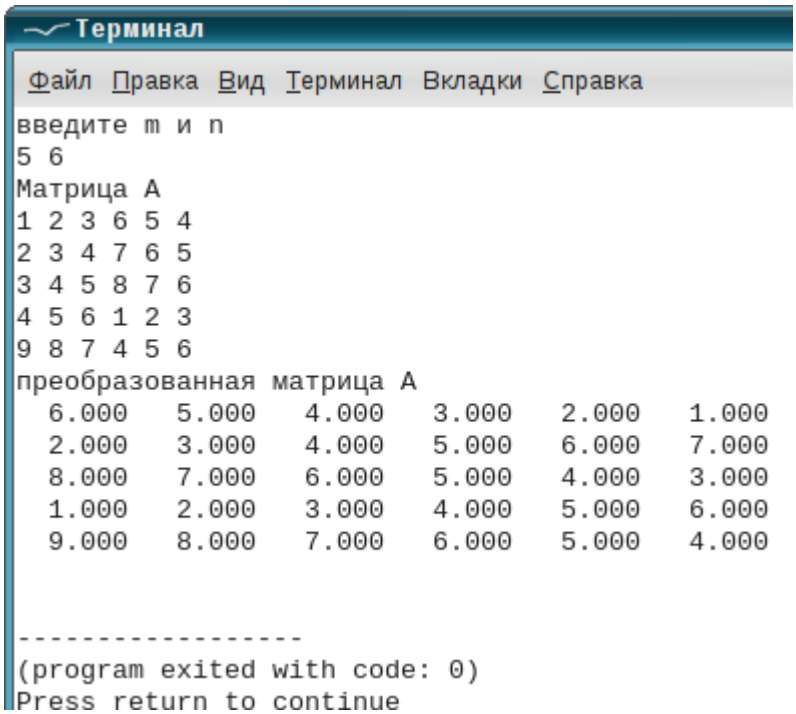

Рисунок 6.25: Результаты решения задачи 6.8

```
{Упорядочивание строки матрицы методом
пузырька по убыванию.}
 for k:=1 to n-1 do
     for j:=1 to n-k do
         if a[i,j] < a[i,j+1] then
         begin
             b:=a[i,j];
             a[i,j]:=a[i,j+1];
             a[i,j+1]:=b;
         end;
```

```
//Вывод преобразованной матрицы.
writeln('преобразованная матрица A');
for i:=1 to m do
begin
 for j:=1 to n do
     write (a[i,j]:7:3,' ');
writeln
end
end.
```

ЗАДАЧА 6.9. Задана матрица целых положительных чисел A(n,m). Сформировать вектор P(n), в который записать сумму простых чисел каждой строки матрицы в четверичной системе счисления, если в строке нет простых чисел, в соответствующий элемент массива записать число 0.

Для решения этой задачи нам понадобятся функции: проверки, является ли число простым, и перевода целого числа в четверичную систему счисления. Функция проверки, является ли число простым, подробно рассматривалась в пятой главе при решении задачи 5.7. Поэтому здесь просто приведем ее текст.

```
function prostoe(N:integer):boolean;
var i:integer;  pr:boolean;
begin
if N<1 then pr:=false
else
begin
 pr:=true;
 for i:=2 to N div 2 do
     if (N mod i = 0) then
     begin
         pr:=false;
         break;
     end;
end;
prostoe:=pr;
end;
```

Кроме того в пятой главе мы рассматривали алгоритм (рис. 5.45) и функцию перевода

```
function  perevod(N:real;P:word;kvo:word):real;
```

вещественного числа в *p*-чную систему счисления (задача 5.10). Нужная нам функция перевода целого числа в четверичную систему счисления является частным случаем рассмотренной ранее функции perevod. Ниже приведен текст функции perevod4 , которая переводит целое положительное число в четверичную систему счисления.

```
   {Функция перевода целого числа N в четверичную
систему счисления.}
   function perevod4(N:word):word;
    var
        s1,i ,q, ost: word;
   begin
   {В переменной s1 мы будем собирать число в чет-
веричной системе счисления.}
    s1:=0;
   {В переменной q будем последовательно хранить
степени десяти, вначале туда записываем 1 — десять
в 0 степени, а затем последовательно в цикле будем
умножать q на 10.}
   q:=1;
   {Перевод целого числа. Пока число не станет
равным 0.}
    while (N<>0) do
    begin
   {Вычисляем ost — очередной разряд числа, как
остаток от деления N на 4 (основание системы счис-
ления).}
        ost:=N mod 4;
   {Очередной разряд числа умножаем на 10 в степе-
ни i и добавляем к формируемому числу s1.}
        s1:=s1+ost*q;
   {Уменьшаем число N в 4 раза путем целочисленно-
го деления на 4.}
        N1:=N1 div 4;
   {Формируем следующую степень десятки.}
        q:=q*10;
    end;
   {Возвращаем число в четверичной системе счисле-
ния.}
```

```
 perevod:=s1;
end;
```

В каждой строке надо найти сумму простых чисел, а затем полученное число перевести в четверичную систему счисления. Поэтому необходимо для каждой строки (`i:=1,2,..,n`) выполнить следующее: обнулить сумму S (`S:=0`), организовать цикл по элементам строки (`j:=1,2,...,m`), внутри которого проверять, является ли текущий элемент $A_{i,j}$ простым и, если является, добавлять его к сумме `S`. После выхода из цикла по `j` необходимо проверить, были ли в строке с номером `i` простые числа (`S>0`), и, если были, перевести `S` в четверичную систему счисления и сформировать соответствующий элемент массива `P` (`P[i]:=perevod4(S)`).

Блок-схема алгоритма приведена на рис. 6.26.

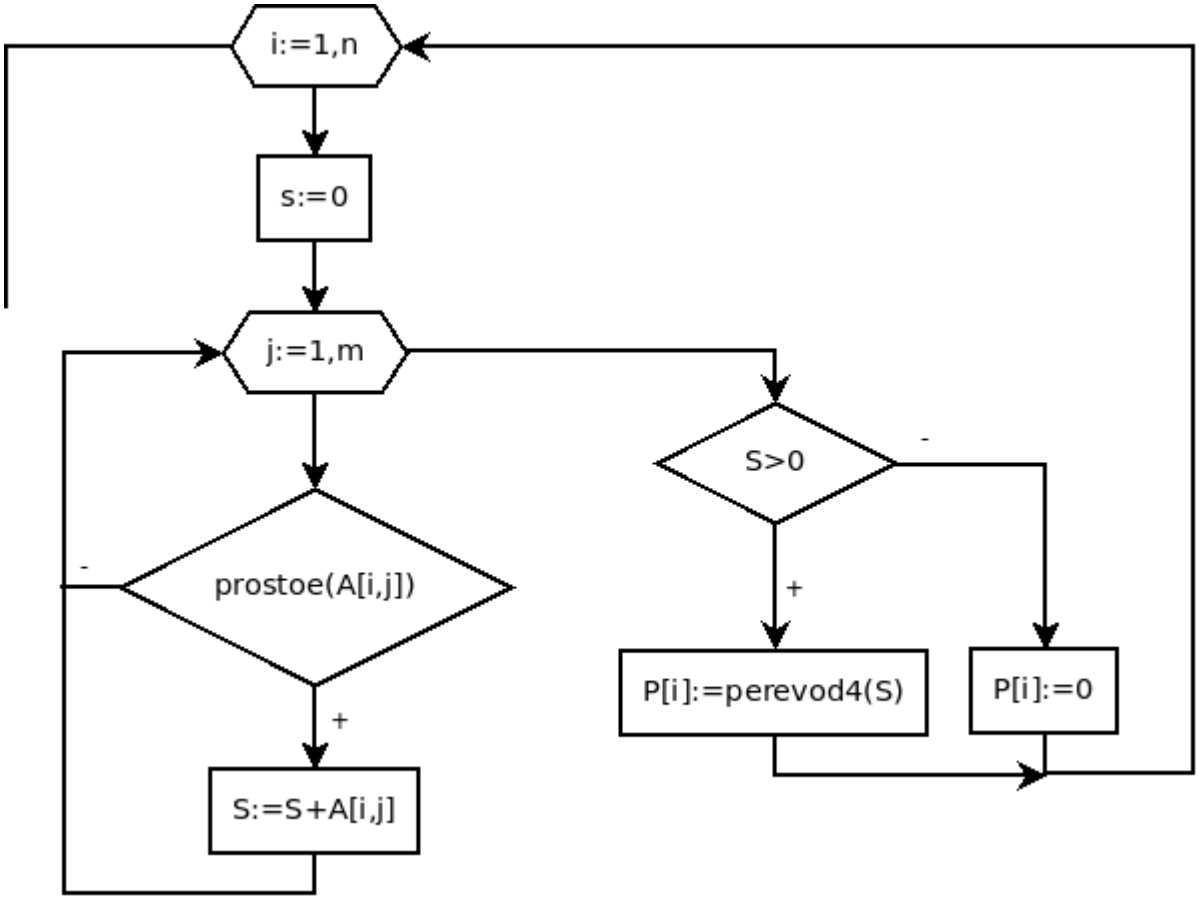

Рисунок 6.26: Блок-схема алгоритма решения задачи 6.9

Полный текст консольного приложения:

```
function prostoe(N:integer):boolean;
var i:integer;  pr:boolean;
begin
if N<1 then pr:=false
```

```
    else
    begin
     pr:=true;
     for i:=2 to N div 2 do
         if (N mod i = 0) then
         begin
             pr:=false; break;
         end;
    end;
    prostoe:=pr;
    end;
    function perevod4(N:word):word;
     var s1,q, ost: word;
    begin
    {В переменной s1 мы будем собирать число в чет-
веричной системе счисления.}
     s1:=0;
    {В переменной q будем последовательно хранить
степени десяти, вначале туда записываем 1 — десять
в 0 степени, а затем последовательно в цикле будем
умножать q на 10.}
    q:=1;
    {Перевод целого числа.
     Пока число не станет равным 0.}
     while (N<>0) do
     begin
    {Вычисляем ost — очередной разряд числа, как
остаток от деления N на 4 (основание системы счис-
ления).}
         ost:=N mod 4;
    {Очередной разряд числа умножаем на 10 в степе-
ни i и добавляем к формируемому числу s1.}
         s1:=s1+ost*q;
    {Уменьшаем число N в 4 раза путем целочисленно-
го деления на 4.}
         N:=N div 4;
    {Формируем следующую степень десятки.}
         q:=q*10;
```

```
    end;
   {Возвращаем число в 4-ной системе счисления.}
    perevod4:=s1;
   end;
   var S,i,j,n,m: word;
    a:array[1..25,1..25] of word;
    p:array[1..25] of word;
   begin
   //Ввод размеров матрицы.
    writeln('Введите размеры матрицы');
    readln(n,m);
    writeln('Введите матрицу A');{Ввод матрицы.}
    for i:=1 to n do
        for j:=1 to m do read(A[i,j]);
   {Последовательно перебираем все строки матрицы
для формирования суммы простых чисел каждой стро-
ки.}
    for i:=1 to n do
    begin
        S:=0;{Вначале сумма равна нулю.}
   {Перебираем все элементы в i-й строке матрицы.}
        for j:=1 to m do
   {Если очередной элемент в i-й строке матрицы —
простое число, то добавляем его к сумме.}
            if prostoe(A[i,j]) then
                        s:=s+A[i,j];
   {Если в строке были простые числа, то их сумму
переводим в четверичную систему счисления и запи-
сываем в p[i].}
        if s>0 then p[i]:=perevod4(s)
   {Если в строке не было простых чисел, то
p[i]:=0.}
        else p[i]:=0;
    end;
   //Вывод сформированного массива P.
    writeln('Массив P');
    for i:=1 to n do  write(P[i],' ');
    writeln;   end.
```

Результаты работы программы представлены на рис. 6.27.

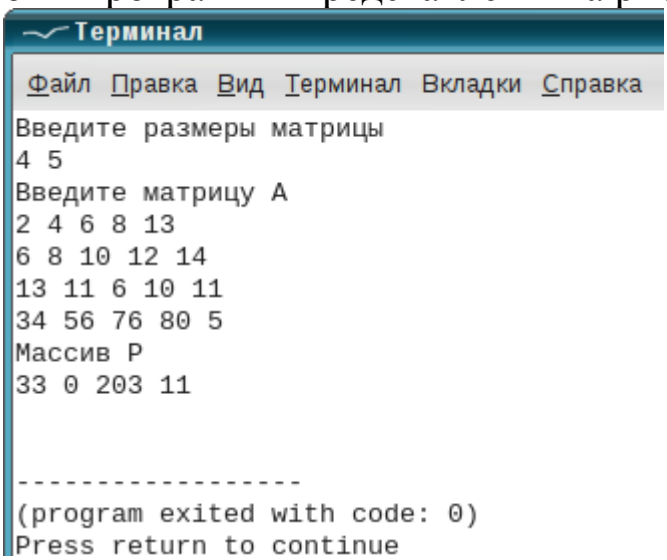

Рисунок 6.27: Результаты работы программы решения задачи 6.9

ЗАДАЧА 6.10. Написать программу умножения двух матриц A(N,M) и B(M,L).

Напомним некоторые сведения из курса математики.

Умножать можно только матрицы, у которых количество столбцов в первой матрице совпадает с количеством строк во второй матрице.

Матрица-произведение имеет столько строк, сколько было в первой матрице и столько столбцов, сколько было во второй. Таким образом, при умножении матрицы `A(N,M)` на матрицу `B(M,L)` получается матрица `C(N,L)`. Каждый элемент матрицы $C_{i,j}$ является скалярным произведением `i`-й строки матрицы `A` и `j`-го столбца матрицы `B`. В общем виде формула для нахождения элемента $C_{i,j}$ матрицы имеет вид:

$$C_{i,j} = \sum_{k=1}^{M} A_{ik} B_{kj} \text{ , где } \texttt{i = 1,N} \text{ и } \texttt{j = 1,L.} \qquad (6.1)$$

Рассмотрим более подробно формирование матрицы `C(3,2)` как произведение матриц `A(3,3)` и `B(3,2)`.

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \\ C_{31} & C_{32} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31} & a_{11}b_{12}+a_{12}b_{22}+a_{13}b_{32} \\ a_{21}b_{11}+a_{22}b_{21}+a_{23}b_{31} & a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32} \\ a_{31}b_{11}+a_{32}b_{21}+a_{33}b_{31} & a_{31}b_{12}+a_{32}b_{22}+a_{33}b_{32} \end{pmatrix}.$$

Следует помнить, что $A \cdot B \neq B \cdot A$.

Блок-схема, реализующая расчет каждого элемента матрицы C по

формуле (6.1), приведена на рис. 6.28. Далее приведен текст программы умножения двух матриц с комментариями. Результат работы представлен на рис. 6.29.

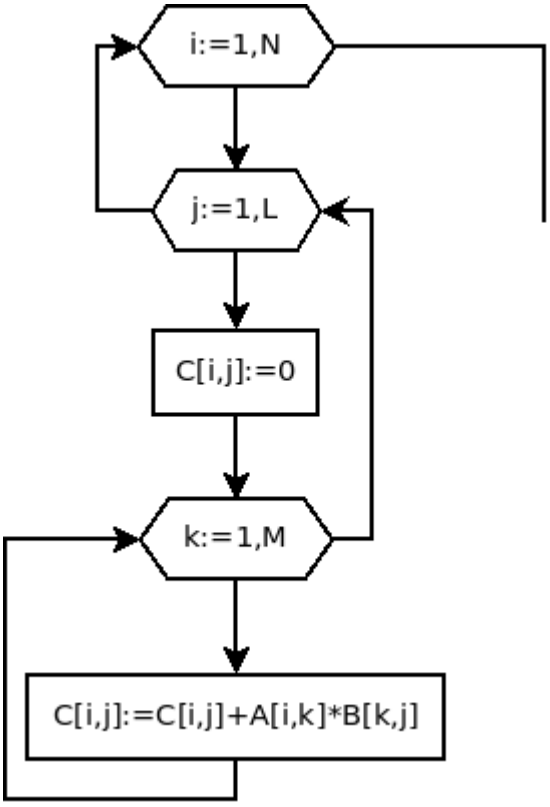

Рисунок 6.28: Блок-схема умножения двух матриц

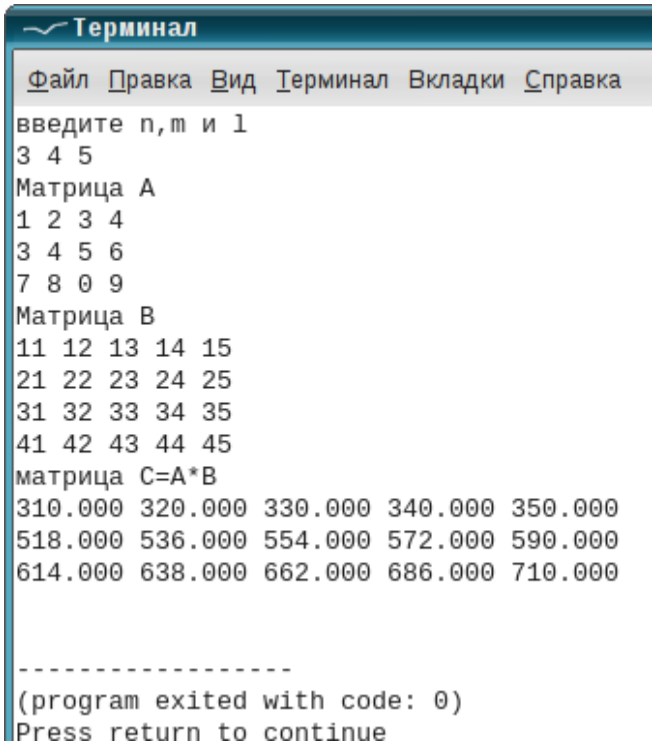

Рисунок 6.29: Результаты работы программы умножения матриц

```
//Умножение двух матриц.
type matrica=array [1..15,1..15] of real;
var  a,b,c:matrica; i,j,M,N,L,k:byte;
begin
 //Ввод размеров матриц.
 writeln('введите n,m и l'); readln(N, M, L);
 writeln('Матрица A');//Ввод матрицы A.
 for i:=1 to N do
     for j:=1 to M do read(a[i,j]);
 writeln('Матрица B');//Ввод матрицы B.
 for i:=1 to M do
     for j:=1 to L do read(b[i,j]);
 for i:=1 to N do //Формирование матрицы C.
     for j:=1 to L do
     begin
```

```
{В C[i,j] хранится результат скалярного произве-
дения i-й строки на j-й столбец.}
            c[i,j]:=0;
            for k:=1 to M do
                c[i,j]:=c[i,j]+a[i,k]*b[k,j];
        end;
    writeln('матрица C=A*B');//Вывод матрицы C=AB.
    for i:=1 to N do
    begin
        for j:=1 to L do
                write(c[i,j]:7:3,' ');
        writeln;
    end;
   end.
```

ЗАДАЧА 6.11 В матрице натуральных чисел A(N,M) найти строки, где находится максимальное из простых чисел. Элементы в них упорядочить по возрастанию. Если в матрице нет простых чисел, то оставить ее без изменений.

Перед решением задачи отметим некоторые ее особенности[72]. В матрице может не быть простых чисел, максимальных значений может быть несколько, и при этом некоторые из них будут находиться в одной строке.

При решении задачи нам понадобятся следующие подпрограммы:

1. Функция `Prostoe`, которая проверяет, является ли число P типа `word` простым. Она возвращает значение `true`, если число `P` – простое и `false` – в противном случае. Заголовок функции имеет вид

```
Function Prostoe (P:word):Boolean;
```

2. Процедура `Udal`, которая из массива чисел `X` удаляет значения, встречающиеся более одного раза. У процедуры два параметра: массив `X` и его размер `N`, оба – параметры переменные. Заголовок процедуры имеет вид:

```
Procedure Udal(var X:massiv; var N:word);
```

Перед описанием процедуры следует описать тип данных massiv (например, `massiv = array [1..200] of word`). Блок-схема процедуры `Udal` представлена на рис. 6.30.

72 Авторы рекомендуют читателям внимательно изучить этот пример, в нем сконцентрированы практически все основные моменты, рассмотренные нами до сих пор.

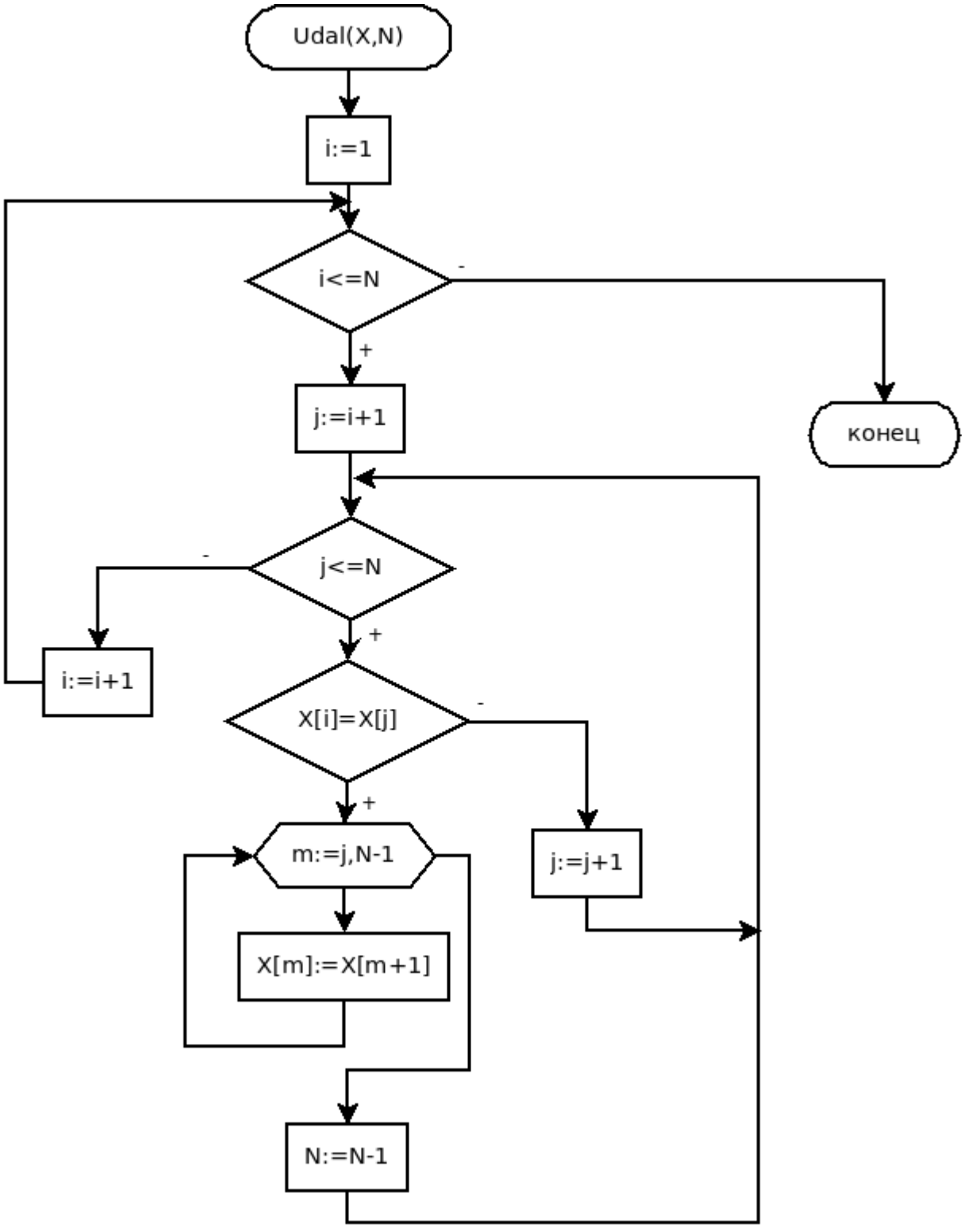

Рисунок 6.30: Блок-схема процедуры Udal

Удаление повторяющихся элементов происходит следующим образом. Просматриваются все элементы, начиная с первого, `i`-й элемент сравнивается со всеми последующими . Если $x_i = x_j$, то встретился повторяющийся элемент, и мы удаляем из массива элемент с но-

мером `j`. Алгоритм удаления был подробно рассмотрен в пятой главе.

3. Функция `Nalichie` возвращает `true`, если число `a` присутствует в массиве `b`, и `false` – в противном случае. Заголовок процедуры имеет вид:

```
Function Nalichie(a:word; b:massiv; N:word);
```

Блок-схема функции представлена на рис. 6.31.

4. Процедура `Vozr` упорядочения массива `x` по возрастанию.

Алгоритмы упорядочения рассматривались в пятой главе. Здесь авторами использовался алгоритм сортировки методом пузырька.

У процедуры `Vozr` два параметра: массив `x` (параметр-переменная) и его размер `N` (параметр-значение). Заголовок процедуры имеет вид:

```
Procedure Vozr(var x:massiv;
                              N:word);
```

Рассмотрим более подробно алгоритм решения задачи 6.11, который приведен на рис. 6.32-6.33. После ввода матрицы (блоки 1-4) предполагаем, что простых чисел нет.

В логическую переменную `Pr` записываем `false`, как только встретится простое число, в переменную Pr запишем `true`.

Количество максимальных значений среди простых чисел равно 0 (`k:=0`) (блок 5).

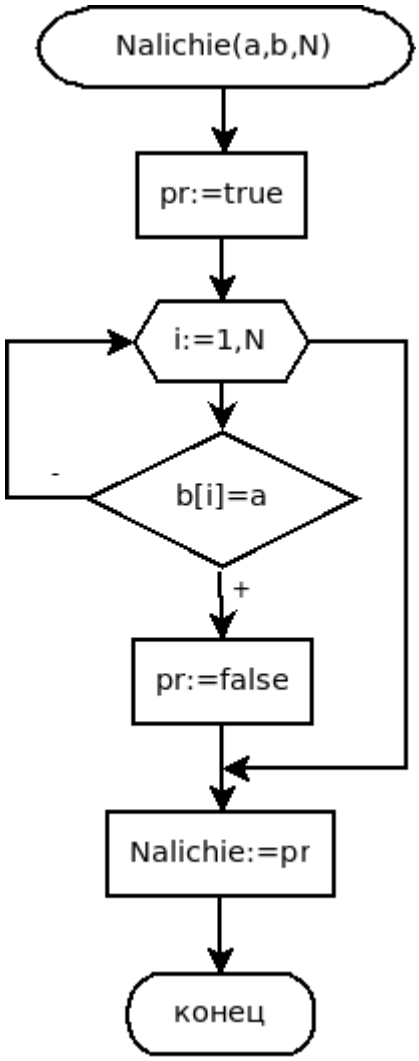

Рисунок 6.31: Блок-схема функции Nalichie

Для проверки, является ли простым числом каждый элемент матрицы, обращаемся к функции `Prostoe`. Если число простое (блок 8), проверяем, первое ли это простое число в матрице (блок 9).

Если это первое простое число, то переписываем его в переменную `max`, в переменную `k` записываем число `1` (количество максимумов равно `1`), номер строки, в которой находится максимум, записы-

ваем в массив mas под номером k[73]. В связи с тем что в матрице есть простые числа, в переменную Pr записываем true (блок 10).

Если это не первое простое число, сравниваем A_{ij} с переменной max. Если $A_{ij} > max$ (блок 11), то в переменную max запишем A_{ij}, в переменную k запишем 1 (есть один максимум), в mas[k] записываем i – номер строки, где находится максимальный элемент (блок 12). Если $A_{ij} = max$ (блок 13), то встретилось число, равное переменной max. В этом случае значение k увеличиваем на 1 и в mas[k] записываем номер строки, где находится элемент, равный max.

В результате двойного цикла обработки всех элементов матрицы (блоки 6-14) в переменной max будет храниться максимальное из простых чисел, в переменной k – количество максимумов, в массиве mas из k элементов будут храниться номера строк, где находятся максимальные значения среди простых чисел матрицы. В переменной Pr хранится true, если в матрице есть простые числа, false – в противном случае.

Если в матрице нет простых чисел (блок 15), выводим соответствующее сообщение (блок 16), в противном случае – с помощью процедуры Udal (блок 17) удаляем из массива mas элементы, встречающиеся более одного раза[74]. Затем просматриваем все строки матрицы (цикл начинается блоком 18), если номер этой строки присутствует в массиве mas (блок 19), то переписываем текущую строку матрицы в массив b (блоки 20-21) и обращаемся к процедуре упорядочивания массива по возрастанию Vozr (блок 22).

Упорядоченный массив b переписываем в i-ю строку матрицы A (блоки 23-24). На последнем этапе выводим на экран матрицу A после преобразования (блоки 25-27).

Ниже приведен листинг всей программы с подробными комментариями.

73 В массиве mas будут храниться номера строк, где находится максимум.

74 Если некоторые максимальные элементы находятся в одной строке, то в массиве mas есть повторяющиеся элементы.

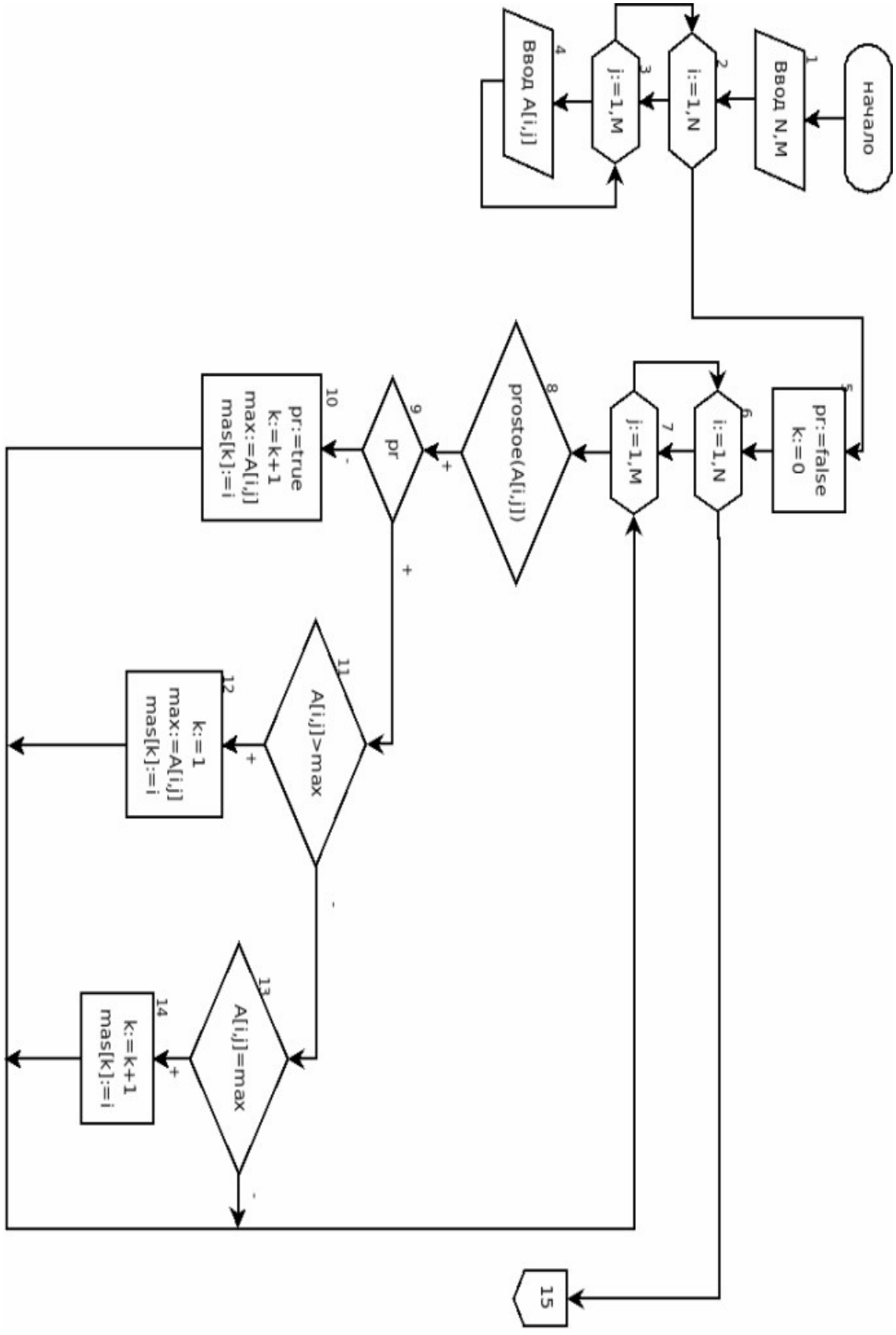

Рисунок 6.32: Блок-схема решение задачи 6.11 (начало)

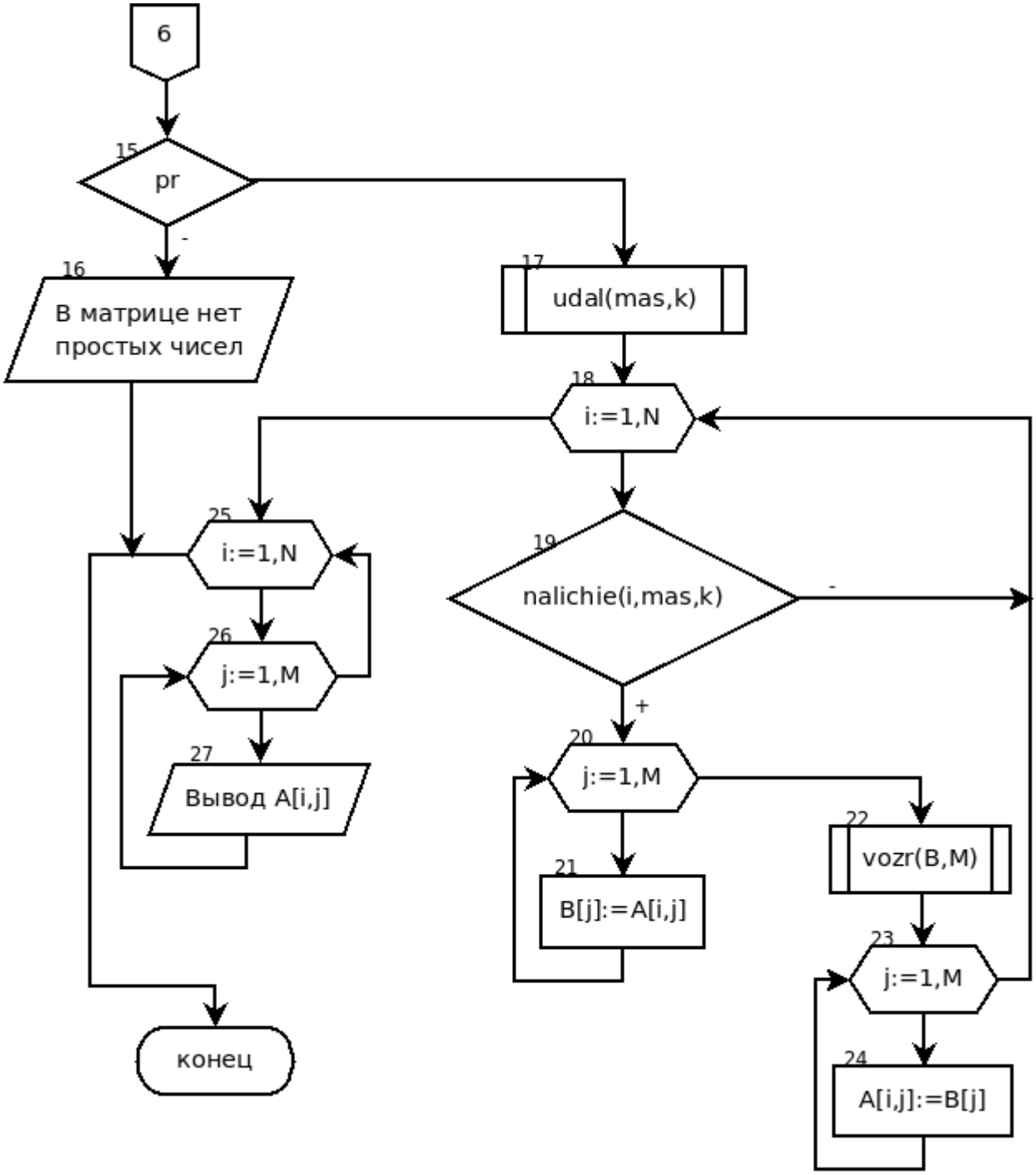

Рисунок 6.33: Блок-схема решения задачи 6.11 (продолжение)

```
{Тип данных massiv будет использоваться
при описании процедур. }
type massiv=array[1..200] of word;
{ Функция prostoe проверяет, является ли
число N простым (true) или нет (false). }
function prostoe(N:word):boolean;
var pr:boolean;i:word;
begin
```

```
    if N>0 then
    begin
   {Предполагаем, что число N – простое (pr=true)}
        pr:=true;
   {Проверяем, делится ли число N на какое-либо
    из чисел от 2 до N/2.}
        for i:=2 to n div 2 do
   { Если встречается число i,
   на которое делится N, то}
        if N mod i =0 then
        begin
   { число N не является простым (pr=false) и }
            pr:=false;
   { выходим из цикла. }
            break;
        end
    end
    else pr:=false;
   { Имени функции присваиваем значение переменной
pr. }
    prostoe:=pr;
   end;
   {Процедура udal удаляет из массива x элементы,
которые встречаются более одного раза. X, N яв-
ляются параметрами-переменными, так как эти значе-
ния возвращаются в головную программу при вызове
процедуры udal.}
   procedure udal(var x:massiv; var n:word);
   var i,j,m:word;
   begin
    i:=1;
   {Просматриваем все элементы, начиная с первого
i=1,2,…,N;i-й элемент сравниваем с последующими
j=i+1,i+2,…,n. }
    while(i<=n) do
    begin
        j:=i+1;
        while(j<=N) do
```

```
   { Если x[i] равно x[j], то встретился повторяю-
щийся элемент}
        if x[i]=x[j] then
        begin
   {и удаляем его (x[j]) из массива. }
            for m:=j to N-1 do x[m]:=x[m+1];
   { После удаления элемента количество элементов
уменьшаем на 1, при этом не переходим к следующему
элементу, так как после удаления под j-м номером
находится уже другой элемент. }
           N:=N-1;
        End
   { Если x[i] не равно x[j], то переходим к сле-
дующему элементу. }
        else j:=j+1;
        i:=i+1;
    end;
   end;
   {Функция nalichie возвращает true, если число a
встречается в массиве b, false – в противном слу-
чае. }
   function
nalichie(a:word;b:massiv;n:word):boolean;
   var
    pr:boolean;
    i:word;
   begin
   {Предполагаем, что в массиве b не встречается
значение a, в pr=false}
    pr:=false;
   { Перебираем все элементы массива. }
    for i:=1 to N do
   { Если очередной элемент массива b равен значе-
нию a, то в pr записываем true}
    if b[i]=a then
    begin
        pr:=true;
   { и выходим из цикла. }
```

```
        break
    end;
   { Имени функции присваиваем значение переменной
pr. }
    nalichie:=pr
   end;
   {Процедура vozr упорядочивает массив x по воз-
растанию. }
   procedure vozr(var x:massiv; n:word);
   { X является параметром-переменной, именно мас-
сив и возвращается в головную программу при вызове
процедуры vozr. }
   var i, j, b :word;
   begin
    for i:=1 to N-1 do
        for j:=1 to N-i do
            if x[j]>x[j+1] then
            begin
                b:=x[j];
                x[j]:=x[j+1];
                x[j+1]:=b;
            end
   end;
   //Начинается основная программа
   var
    N, M, i, j, k, max :word;
    A:array[1..20,1..20] of word;
    pr, L: boolean;
    mas, b:massiv;
   begin
   { Вводим элементы матрицы A. }
    write('N=');readln(N);
    write('M=');readln(M);
   writeln(' Matrica A');
    for i:=1 to N do
    for j:=1 to M do
        read(A[i,j]);
   { Предполагаем, что в матрице нет простых чи-
```

```
сел. }
    Pr:=false;
   { Количество элементов, равных максимальному,
равно 0. }
    k:=0;
   { Перебираем все элементы в матрице. }
    for i:=1 to N do
    for j:=1 to M do
    begin
   { Обращаемся к функции, которая проверяет, яв-
ляется ли число A[I,j] простым. }
        L:=Prostoe(A[i,j]);
   { Если число простое и }
        if L then
   { если простое число встретилось первый раз,}
           if not Pr then
           begin
   { записывем в pr true,}
              Pr:=true;
   { увеличиваем количество максимумов на 1, можно
было просто написать k:=1}
              k:=k+1;
   { Это число записываем в переменную max, это
первое простое число, и предполагаем, что оно мак-
симальное. }
              max:=A[i,j];
   {B mas[k] записываем номер строки, где хранится
число A[I,j]}
              mas[k]:=i
           end
           else
   { Если A[i,j] не первое простое число, то срав-
ниваем max и текущее простое значение матрицы A. }
              if A[i,j]>max then
   { Если A[I,j]> max, то }
              Begin
   { количество максимумов равно 1, т.к. встретил-
ся наибольший в данный момент элемент. }
```

```
                    k:=1;
   { в переменную max записываем A[i,j], }
                    max:=A[i,j];
   { в mas[k] записываем номер строки, где хранит-
ся число A[I,j]}
                    mas[k]:=i
                end
                else
   { Если A[i,j]=max (встретился элемент, равный
максимуму), то }
   if A[i,j]=max then
                    begin
   { количество максимумов увеличиваем на 1. }
                        k:=k+1;
   { в mas[k] записываем номер строки, где хранит-
ся число A[I,j]}
                        mas[k]:=i
                    end
    end;
   {Если в pr осталось значение false,то выводим
сообщение, что в матрице нет простых чисел, }
    if not Pr then writeln('В матрице A нет про-
стых чисел')
    else
    begin
   { иначе удаляем из массива mas, номера строк,
где хранятся максимумы, повторяющиеся элементы. }
        Udal(mas,k);
   {Перебираем все строки матрицы. }
        for i:=1 to N do
        begin
            L:=Nalichie(i,mas,k);
   {Если номер строки присутствует в массиве mas,}
            if L then
            begin
   {то переписываем строку в массив b}
                for j:=1 to M do
                    b[j]:=A[i,j];
```

```
    {упорядочиваем массив b по возрастанию. }
                Vozr(b,M);
    { упорядоченный массив записываем на место i-й
строки матрицы A. }
                for j:=1 to M do
                A[i,j]:=b[j];
            end
        end;
        writeln('Преобразованная матрица A');
        for i:=1 to N do
        begin
            for j:=1 to M do write(A[i,j],' ');
            writeln;
        end
     end
    end.
```

Результаты работы программы приведены на рис. 6.34.

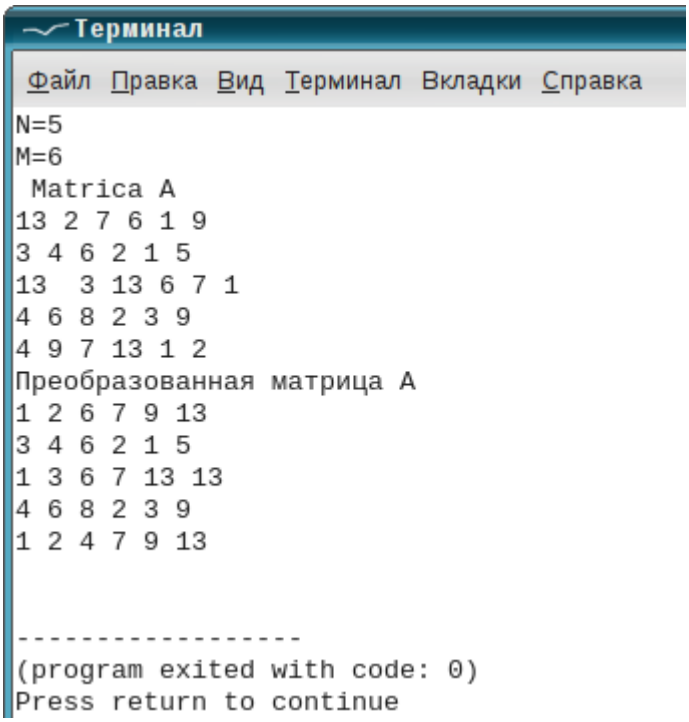

Рисунок 6.34: Результаты решения задачи 6.11

Авторы рекомендуют читателю по рассмотренным алгоритмам и консольным приложениям задач 6.7-6.11 разработать визуальные приложения, аналогичные тем, которые были разработаны для задач 6.2, 6.6.

6.3 Динамические матрицы

Понятие динамического массива можно распространить и на матрицы. Динамическая матрица представляет собой массив указателей, каждый из которых адресует одну строку (или один столбец).

Рассмотрим описание динамической матрицы. Пусть есть типы данных `massiv` и указатель на него `din_massiv`.

```
type massiv=array [1..1000] of real;
din_massiv=^massiv;
```

Динамическая матрица X будет представлять собой массив указателей.

```
Var X: array[1..100] of din_massiv;
```

Работать с матрицей надо следующим образом.

1. Определить ее размеры (пусть `N` – число строк, `M` – число столбцов).

2. Выделить память под матрицу.

```
for i:=1 to N do
getmem(X[i],M*sizeof(real));
```

Каждый элемент статического массива `X[i]` – указатель на динамический массив, состоящий из M элементов типа `real`. В статическом массиве `X` находится `N` указателей.

3. Для обращения к элементу динамической матрицы, расположенному в `i`-й строке и `j`-м столбце, следует использовать конструкцию языка Турбо Паскаль `X[i]^[j]`.

4. После завершения работы с матрицей необходимо освободить память.

```
for i:=1 to N do
     freemem(b[i],M*sizeof(real));
```

Рассмотрим работу с динамической матрицей на следующем примере.

ЗАДАЧА 6.12. В каждой строке матрицы вещественных чисел B(N,M) упорядочить по возрастанию элементы, расположенные между максимальным и минимальным значением.

Алгоритмы упорядочивания рассматривались в пятой главе, основные принципы работы с матрицами – в предыдущих параграфах текущей главы, поэтому в комментариях к тексту программы основное внимание уделено особенностям работы с динамическими матрицами.

```
   { Описываем тип данных massiv как массив 1000
вещественных чисел. }
   type massiv=array [1..1000] of real;
   { Указатель на массив. }
   din_massiv=^massiv;
   { Тип данных matrica – статический массив ука-
зателей, каждые элемент которого является адресом
массива вещественных чисел.}
   matrica=array [1..100] of din_massiv;
   var
   Nmax,Nmin,i,j,n,m,k:word;
   { Описана динамическая матрица b. }
   b:matrica;
   a,max,min:real;
   begin
   { Вводим число строк N и число столбцов M. }
   write('N=');readln(N);
   write('M=');readln(M);
   { Выделяем память под матрицу вещественных чи-
сел размером N на M.}
   for i:=1 to N do
   getmem(b[i],M*sizeof(real));
   { Вводим Матрицу B. }
   writeln('Matrica B');
   for i:=1 to N do
   for j:=1 to M do
   read(b[i]^[j]);
   {В каждой строке находим максимальный, мини-
мальный элементы и их номера, и элементы, располо-
женные между ними упорядочиваем методом
пузырька. }
   for i:=1 to N do
   begin
   { Поиск минимального, максимального элементов в
i-й строке матрицы и их номеров.}
   max:=b[i]^[1];
   Nmax:=1;
   min:=b[i]^[1];
```

```
   Nmin:=1;
   for j:=2 to M do
   begin
   if b[i]^[j]>max then
   begin
   max:=b[i]^[j];
   nmax:=j
   end;
   if b[i]^[j]<min then
   begin
   min:=b[i]^[j];
   nmin:=j
   end;
   end;
   { Если минимальный расположен позже максимального, nmin и nmax меняем местами}
   if nmax<nmin then
   begin
   j:=nmax;
   nmax:=nmin;
   nmin:=j;
   end;
   {В i-той строке упорядочиваем элементы, расположенные между nmin и nmax, методом пузырька. }
   j:=1;
   while nmax-1-j>=nmin+1 do
   begin
   for k:=nmin+1 to nmax-1 -j do
   if b[i]^[k]>b[i]^[k+1] then
   begin
   a:=b[i]^[k];
   b[i]^[k]:=b[i]^[k+1];
   b[i]^[k+1]:=a;
   end;
   j:=j+1;
   end;
   end;
   { Выводим преобразованную матрицу. }
```

```
writeln('Упорядоченная матрица B');
for i:=1 to N do
begin
for j:=1 to M do
write(b[i]^[j]:6:2, ' ');
writeln
end;
{ Освобождаем память. }
for i:=1 to N do
freemem(b[i],M*sizeof(real));
end.
```

Результаты работы программы представлены на рис. 6.35.

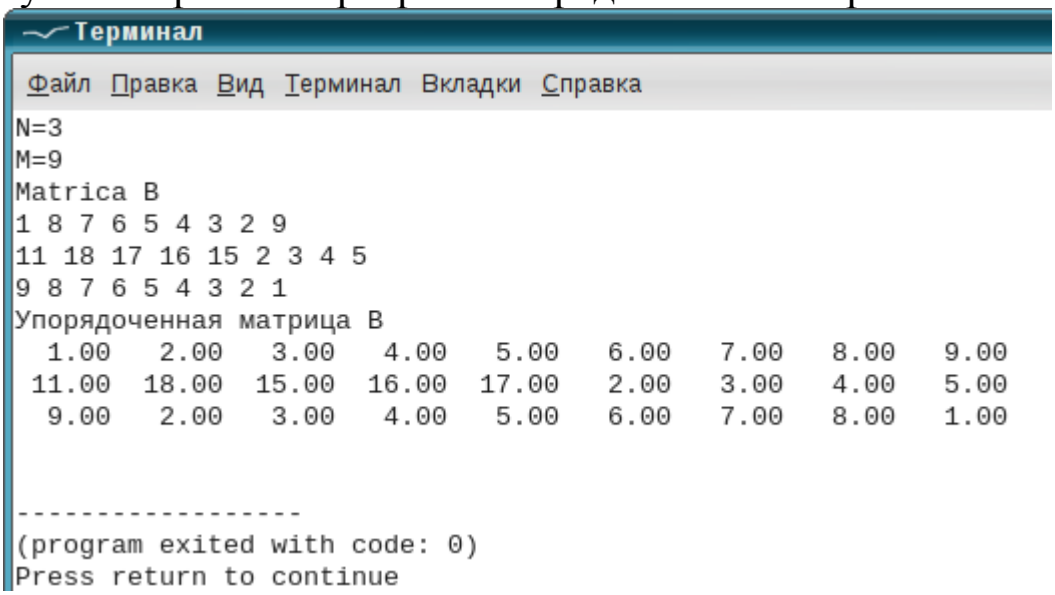

Рисунок 6.35: Результаты решения задачи 6.12

Динамическая матрица может быть достаточно большой, фактически ее размер ограничен только объемом свободной памяти.

6.4 Задачи для самостоятельного решения

1. Определить номера строки и столбца максимального простого числа прямоугольной матрицы `A(n,m)`. Подсчитать количество нулевых элементов матрицы и напечатать их индексы.

2. Найти среднее геометрическое значение элементов квадратной матрицы `X(n,n)`, находящихся по периметру этой матрицы и на ее диагоналях, если это возможно. Если среднее геометрическое вычислить невозможно, то поменять местами максимальный и минимальный элементы матрицы.

3. Сформировать вектор D, каждый элемент которого представляет собой среднее арифметическое значение элементов строк матрицы C(k,m), и вектор G – любой его компонент должен быть равен произведению элементов соответствующего столбца матрицы C.

4. Задана матрица A(n,m), в каждом столбце которой максимальный элемент необходимо заменить произведением отрицательных элементов этого же столбца.

5. Задана матрица A(n,n). Определить максимальный элемент среди элементов матрицы, расположенных выше главной диагонали, и минимальный элемент среди тех, что находятся ниже побочной диагонали. После этого выполнить сортировку каждого столбца матрицы по возрастанию.

6. Заменить строку матрицы P(n,m) с минимальной суммой элементов на строку, где находится максимальный элемент матрицы.

7. Переместить максимальный элемент матрицы F(k,p) в правый верхний угол, а минимальный элемент – в левый нижний.

8. Проверить, является ли матрица A(n,n) диагональной (все элементы нули, кроме главной диагонали), единичной (все элементы нули, на главной диагонали только единицы) или нулевой (все элементы нули).

9. Сформировать из некоторой матрицы A(n,n) верхнетреугольную матрицу B(n,n) (все элементы ниже главной диагонали нулевые), нижнетреугольную матрицу C(n,n) (все элементы выше главной диагонали нулевые) и диагональную матрицу D(n,n) (все элементы нули, кроме главной диагонали) .

10. Заданы матрицы A(m,n) и B(n,m). Найти матрицу $C=(A\cdot B)^4$.

11. Проверить, является ли матрица B(n,n) обратной к A(n,n). Произведением матриц A и B в этом случае должна быть единичная матрица.

12. Определить количество простых чисел, расположенных вне диагоналей матрицы B(n,n).

13. Проверить, лежит ли на главной диагонали максимальный отрицательный элемент матрицы A(n,n).

14. Переписать простые числа из матрицы A в массив B. Массив упорядочить по убыванию.

15. Переписать положительные числа из матрицы целых чисел A в массив B. Из массива B удалить числа, в двоичном представлении которых единиц больше, чем нулей.

16. Даны четыре квадратные матрицы A(n,n), B(n,n), C(n,n), D(n,n), в которых хранятся целые числа. Найти матрицу, в которой находится максимальное простое число.

17. Заданы четыре квадратные матрицы A(n,n), B(n,n), C(n,n), D(n,n), в которых хранятся целые числа. Найти матрицы, в которых на диагоналях есть простые числа.

18. Заданы три прямоугольные матрицы A(n,m), B(r,p), C(k,q). Найти матрицы, в которых по периметру расположены только отрицательные числа.

19. Проверить, лежит ли на побочной диагонали минимальный положительный элемент матрицы A(n,n).

20. Заданы матрицы D(n,n), A(m,n) и B(n,m). Найти матрицу C=(B·A). Проверить, является ли матрица C(n,n) обратной к D(n,n). Произведением матриц C и D в этом случае должна быть единичная матрица.

21. Заданы четыре квадратные матрицы A(n,n), B(n,n), C(n,n), D(n,n), в которых хранятся целые числа. Найти, в какой из матриц на побочной диагонали есть числа, состоящие из восьмерок.

22. Заменить столбец матрицы P(n,m) с максимальной суммой элементов на столбец, где находится максимальное число, состоящее из единиц.

23. Заданы четыре квадратные матрицы A(n,n), B(n,n), C(n,n), D(n,n), в которых хранятся целые числа. Определить, есть ли среди них матрицы, в которых на побочной диагонали находятся только числа, состоящие из единиц и двоек.

24. Переписать простые числа из матрицы целых чисел A в массив B. Из массива B удалить числа, расположенные между максимальным и минимальным элементами.

25. В матрице целых чисел A(n,n) упорядочить те строки, в которых диагональные элементы не содержат семерок.

7 Обработка файлов средствами Free Pascal

В данной главе будут рассмотрены возможности языка FreePascal для работы с файлами. По ходу изложения материала читатель также познакомится с компонентами Lazarus, предназначенными для выбора файла.

Начнем со знакомства с типами файлов.

7.1 Типы файлов

Ввод данных с клавиатуры удобен при обработке небольших объемов информации. В случае обработки массивов, состоящих из сотен элементов, использовать ввод данных с клавиатуры нерационально. В подобных случаях данные удобно хранить в файлах, программа будет их считывать, обрабатывать, выводить результаты на экран или в файл. Рассмотрим, как это можно сделать.

С точки зрения программиста, все файлы можно разделить на три класса:

- типизированные;
- бестиповые;
- текстовые.

Файлы, состоящие из компонентов одного типа (целые, вещественные, массивы и т.д.), число которых заранее не определено и может быть любым, называются *типизированными*. Они заканчиваются специальным символом «*конец файла*», хранятся в двоичном виде, содержимое подобных файлов нельзя просмотреть обычным текстовым редактором, для просмотра подобных файлов нужно писать специальную программу.

В *бестиповых файлах* информация считывается и записывается блоками определенного размера. В подобных файлах хранятся данные любого вида и структуры.

Текстовые файлы состоят из любых символов. При записи информации в текстовый файл все данные преобразуются к символьному типу, в котором и хранятся. Просмотреть данные в подобном файле можно с помощью любого текстового редактора. Информация в текстовом файле хранится построчно. В конце каждой строки хранится специальный символ «*конец строки*». Конец самого файла обозначается символом «*конец файла*».

Для работы с файлами в программе следует описать файловую переменную. Для работы с *текстовым* файлом файловая переменная (например, *f*) описывается с помощью служебного слова *text*.

```
var f:text;
```

Для описания *типизированных*[75] файлов можно описать файловую переменную следующим образом:

```
var f:file of тип;
```
[76]

Бестиповый файл описывается с помощью служебного слова `file`.

Рассмотрим несколько примеров описания файловых переменных.

```
type
massiv=array[1..25]of real;
ff=file of real;
var
a:text; {Файловая переменная a для работы с текстовым файлом}
b:ff; {Файловая переменная f для работы с файлом вещественных чисел}
c:file of integer; {Файловая переменная c для работы с файлом целых чисел}
d:file of massiv; {Файловая переменная d предназначена для работы с типизированным файлом, элементами которого являются массивы из 25 вещественных чисел. }
```

Рассмотрим последовательно работу с файлами каждого типа.

7.2 Работа с типизированными файлами

Знакомство с методами обработки типизированных файлов начнем с подпрограмм, которые являются общими для всех типов файлов.

7.2.1 Процедура *AssignFile*

Для начала работы с файлом необходимо связать файловую переменную в программе с файлом на диске. Для этого используется про-

75 Типизированные файлы иногда называют компонентными.

76 Здесь **f** — имя файловой переменной, **тип** — тип компонентов файла, это могут быть как стандартные типы данных (например, real, integer и т.д.), так и типы, определенные пользователем.

цедура `AssignFile(f,s)`, где `f` – имя файловой переменной, а `s` – полное имя файла на диске (файл должен находиться в текущем каталоге при условии, что к нему специально не указывается путь).

Рассмотрим примеры использования `AssignFile` для различных операционных систем.

```
var
f:file of real;
begin
//Пример процедуры assign для ОС Windows.
AssignFile (f, 'd:\tp\tmp\abc.dat');
//Пример процедуры assign для ОС Linux.
AssignFile(f,'/home/pascal/6/pr1/abc.dat');
```

7.2.2 Процедуры reset, rewrite

После установления связи между файловой переменной и именем файла на диске нужно открыть файл, воспользовавшись процедурами `reset` или `rewrite`.

Процедура `reset(f)` (где `f` – имя файловой переменной) открывает файл, связанный с файловой переменной `f`, после чего становится доступным для чтения первый элемент, хранящийся в файле. Далее можно выполнять чтение и запись данных из файла.

Процедура `rewrite(f)` создает пустой файл (месторасположение файла на диске определяется процедурой `AssignFile`) для последующей записи в него данных.

Внимание!!! Если файл, связанный с файловой переменной *f*, существовал на диске, то вся информация в нем уничтожается.

7.2.3 Процедура CloseFile

Процедура `CloseFile(f)`, где `f` – имя файловой переменной, закрывает файл, который ранее был открыт процедурами `rewrite`, `reset`.

Процедуру `CloseFile(f)` следует *обязательно* использовать при закрытии файла, в который происходила запись данных.

Дело в том, что процедуры записи в файл не обращаются непосредственно к диску, они пишут информацию в специальный участок памяти, называемый буфером файла. После того как буфер заполнится, вся информация из него переносится в файл. При выполнении

процедуры `closefile` сначала происходит запись буфера файла на диск, и только потом файл закрывается. Если его не закрыть вручную, то закрытие произойдет автоматически при завершении работы программы. Однако при автоматическом закрытии файла информация из буфера файла *не переносится* на диск, и как следствие часть информации может пропасть.

Внимание!!! После записи информации в файл его обязательно закрывать с помощью процедуры `CloseFile`. Однако при чтении данных из файла нет необходимости в обязательном его закрытии.

7.2.4 Процедура rename

Переименование файла, связанного с файловой переменной `f`, осуществляется в то время, когда он закрыт, при помощи процедуры `rename(f,s)`, где `f` – файловая переменная, `s` – новое имя файла (строковая переменная).

7.2.5 Процедура erase

Удаление файла, связанного с переменной `f`, выполняется посредством процедуры `erase(f)`, в которой `f` также является именем файловой переменной. Для корректного выполнения этой операции файл должен быть закрыт.

7.2.6 Функция eof

Функция `eof(f)` (end of file), где `f` – имя файловой переменной, принимает значение «истина» (`true`), если достигнут конец файла, иначе – «ложь» (`false`). С помощью этой функции можно проверять, достигнут ли конец файла и можно ли считывать очередную порцию данных.

7.2.7 Чтение и запись данных в файл

Для записи данных в файл можно использовать процедуру `write`:

```
write(f, x1, x2,..., xn);
write(f, x);
```

здесь

`f` — имя файловой переменной,

`x`, `x1`, `x2`, ..., `xn` — имена переменных, значения из которых записываются в файл.

Тип компонентов файла обязательно должен совпадать с типом переменных. При выполнении процедуры `write` значения `x1`, `x2`, ..., `xn` последовательно записываются в файл (начиная с текущей позиции), связанный с файловой переменной `f`.

Для чтения информации из файла, связанного с файловой переменной `f`, можно воспользоваться процедурой `read`:

```
read(f, x1, x2, x3,..., xn);
read(f, x);
```

здесь

`f` — имя файловой переменной,

`x`, `x1`, `x2`, ..., `xn` — имена переменных, в которые считываются значения из файла.

Процедура `read` последовательно считывает компоненты из файла, связанного с файловой переменной `f`, в переменные `x1`, `x2`, ..., `xn`. При считывании очередного значения доступным становится следующее. Следует помнить, что процедура `read` не проверяет, достигнут ли конец файла. За этим нужно следить с помощью функции `eof`.

Для того чтобы записать данные в файл, необходимо выполнить следующее:

1. Описать файловую переменную.
2. Связать ее с физическим файлом (процедура `AssignFile`).
3. Открыть файл для записи (процедура `rewrite`).
4. Записать данные в файл (процедура `write`).
5. Обязательно закрыть файл (процедура `CloseFile`).

Рассмотрим создание компонентного файла на примере решения следующей несложной задачи.

ЗАДАЧА 7.1. Создать типизированный файл и записать туда `n` вещественных чисел.

Алгоритм решения задачи состоит из следующих этапов:

1. Открыть файл для записи с помощью оператора `rewrite`.
2. Ввести значение *n*.
3. В цикле (при *i* меняющемся от `1` до `n`) вводим очередное вещественное число *a*, которое сразу же записываем в файл с помощью процедуры `write`.
4. Закрываем файл с помощью процедуры `closefile`.

Текст консольного приложения, предназначенного для решения

данной задачи приведен ниже.

```
program pr1;
{$mode objfpc}{$H+}
uses
  Classes, SysUtils
  { you can add units after this };
 var f:file of real;
     i,n:integer;
     a:real;
begin
//Связываем файловую переменную с файлом
//на диске.
AssignFile(f,'/home/pascal/6/pr1/abc.dat');
//Открываем пустой файл для записи.
     rewrite(f);
//Определяем количество элементов в файле.
     write('n=');
     readln(n);
//В цикле вводим очередной элемент и
//записываем его в файл.
     for i:=1 to n do
     begin
         write('a=');
         readln(a);
         write(f,a);
     end;
//Закрываем файл.
//При записи данных в файл это делать
//обязательно.
     CloseFile(f)
end.
```

Рассмотрим следующую задачу

ЗАДАЧА 7.2. В папке */home/evgeniy/pascal/6/pr2/* находятся файлы вещественных чисел с расширением dat. В выбранном пользователем файле удалить все числа, меньшие среднего арифметического, расположенные между максимальным и минимальным элементами.

В задаче предполагается выбор пользователем файла для обра-

ботки. Для этого понадобится специальный компонент для выбора файла. Решение задачи 7.2 начнем со знакомства с компонентом `OpenDialog`  . Это компонент предназначен для создания стандартного диалогового окна выбора файла и расположен первым на странице Dialogs (см. рис. 7.1).

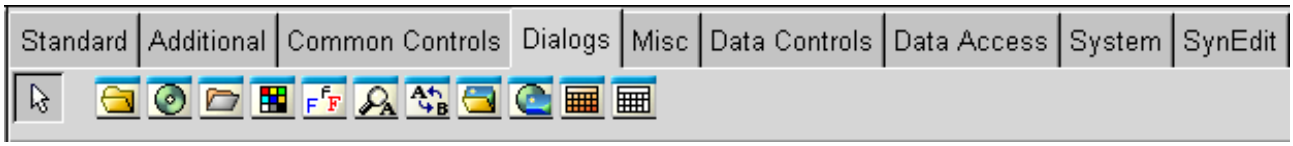

Рисунок 7.1: Компоненты страницы Dialogs

Среди основных свойств этого компонента можно выделить:

FileName: String — полное имя выбранного пользователем файла;

Filter: String — строка, которая возвращает фильтры отбираемых файлов; это свойство можно задать с помощью редактора фильтров или с помощью специальной строки. Для вызова редактора необходимо щелкнуть по кнопке  , после чего появится окно редактора фильтров (см. рис. 7.2).

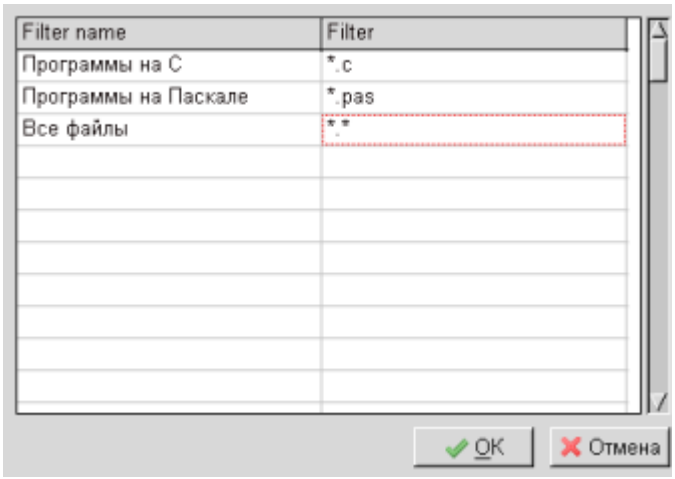

Рисунок 7.2: Редактор фильтров

Окно редактора фильтров состоит из двух столбцов: *Filter name* — имя фильтра (например, все файлы, программы на Паскале); *Filter* — соответствующая имени фильтра маска (фильтру «все файлы» соответствует маска «*.*», фильтру «программы на Паскале» — маска «*.pas».

Специальная строка состоит из последовательных имен фильтров и соответствующих им масок, разделенных символом «|». Специальная строка, соответствующая фильтру, представленному на рис.7.2, имеет вид.

```
OpenDialog1.Filter:=
    'Программы на C|*.c|Программы на Паскале|*.pas|Все файлы|*.*';
```

Первый фильтр в списке является фильтром по умолчанию. В примере на рис. 7.2 фильтр по умолчанию — «Программы на С». `InitialDialog` — имя каталога по умолчанию для выбора файлов.

`DefaultExt` — расширение, добавляемое к имени файла по умолчанию, если пользователь при ручном вводе имени файла не указал расширение.

Основным методом для компонента `OpenDialog` — является логическая функция `Execute`, которая приводит к открытию диалогового окна в соответствии со свойствами компонента. Функция `Execute` возвращает `true`, если пользователь выбрал файл каким-либо методом. Если пользователь нажал в диалоговом окне `Отмена` (`Cancel`), то метод `Execute` возвращает `false`. Имя выбранного файла возвращается в свойстве `FileName`.

Таким образом, вызов диалогового окна можно записать так.

```
Var
s:String;
begin
     if OpenDialog1.Execute then
     s:=OPenDialog1.FileName;
{Имя файла, выбранного пользователем, хранится в переменной s}.
end;
```

Разобравшись с компонентом для выбора файла, вернемся к задаче 7.2. Можно выделить следующие этапы ее решения.

1. Выбор файла.
2. Чтение данных из файла в массив вещественных чисел.
3. В массиве вещественных чисел удалить все числа, меньшие среднего арифметического, расположенные между максимальным и минимальным элементами.
4. Запись преобразованного массива в файл.

Разработку программы начнем с создания графического приложения (**Проект — Создать Проект — Приложение**).

На форме расположим следующие компоненты:

1. Две метки `Label1` и `Label2` для подписи.
2. `Memo1` — компонент, в котором будем хранить содержимое исходного файла.
3. `Memo2` — компонент, в котором хранится преобразованный файл.
4. `OpenDialog1` — компонент для выбора имени обрабатываемого файла.

5. Buttom1 — кнопка для запуска программы.
6. Buttom2 — кнопка для завершения программы.

Установим следующие свойства формы и компонентов (см. табл. 7.1 — 7.8).

Таблица 7.1: Свойства формы

Свойство	Caption	Name
Значение	Преобразование файла	Form_File

Таблица 7.2: Свойства метки label1

Свойство	Caption	Visible
Значение	Файл до преобразования	False

Таблица 7.3: Свойства метки label2

Свойство	Caption	Visible
Значение	Файл после преобразования	False

Таблица 7.4: Свойства компонента Memo1

Свойство	Lines	Visible
Значение	''	False

Таблица 7.5: Свойства компонента Memo2

Свойство	Lines	Visible
Значение	''	False

Таблица 7.6: Свойства компонента OpenDialog1

Свойство	InitDialog	DefaultExt
Значение	/home/evgeniy/pascal/6/pr2/	dat

Таблица 7.7: Свойства кнопки Button1

Свойство	Caption	Name	Visible
Значение	Преобразовать файл	Button_File	True

Таблица 7.8: Свойства кнопки Button2

Свойство	Caption	Name	Visible
Значение	Закрыть	Button_Close	False

Расположим компоненты на форме подобно показанному на рис. 7.3. При запуске программы на выполнение все компоненты, кроме кнопки «Преобразовать файл», будут невидимыми, свойство Visible установлено в False. Окно приложения при запуске пока-

зано на рис. 7.4.

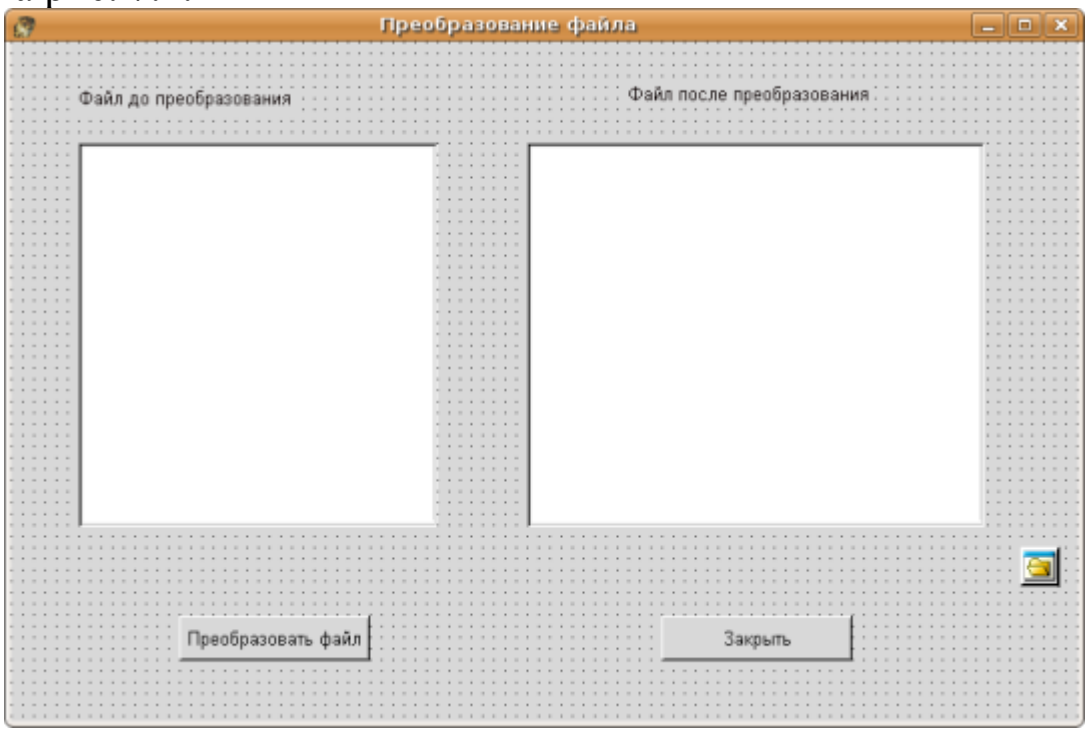

Рисунок 7.3: Форма с расположенными на ней компонентами

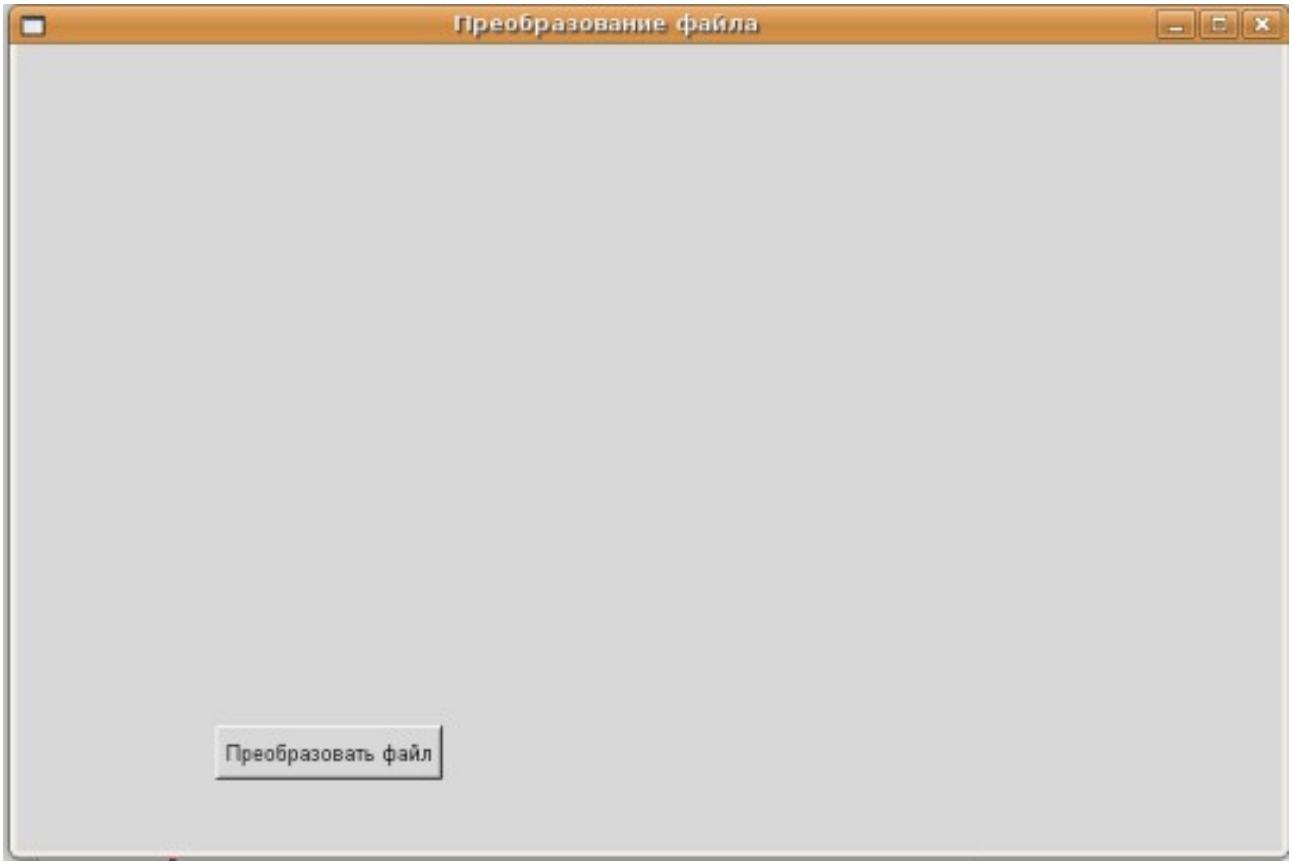

Рисунок 7.4: Приложение при запуске

Проектирование интерфейса приложения завершено. Осталось написать тексты обработчиков событий.

При создании формы установим свойство Filter компонента OpenDialog1. Поэтому подпрограмма FormCreate будет такой.

```
procedure TForm_File.FormCreate(Sender: TObject);
   begin
   OpenDialog1.Filter:=
   'Файлы вещественных чисел|*.dat|Все файлы|*.*';
   end;
```

При щелчке по кнопке «Преобразовать файл» должно происходить следующее:

1. Выбор файла из диалогового окна.
2. Считывание данных из файла в массив вещественных чисел.
3. Установка свойства Visible в True у компонентов label1 и Memo1.
4. Вывод содержимого массива в Memo1.
5. Преобразование массива.
6. Запись преобразованного массива в файл.
7. Установка свойства Visible в True у компонентов label2 и Memo2.
8. Вывод преобразованного массива в Memo2.
9. Установка свойства Visible в True у компонента Button_Close.

Ниже приведен текст обработки события Button_FileClick с подробными комментариями.

```
procedure Tform_File.Button_FileClick(
                                  Sender: TObject);
   var
   f:file of real;
   s:string;
   a:array[1..100] of real;
   nmax,nmin,i,j,n:integer;
   sum,max,min:real;
   begin
   //Открываем диалоговое окно для выбора файла.
    if opendialog1.execute then
    begin
   //Имя выбранного файла считываем в s.
       s:=OpenDialog1.FileName;
```

```
//Связываем файловую переменную
//с файлом на диске.
    AssignFile(f,s);
//Открываем файл для чтения.
    Reset(f);
//Обнуляем счетчик элементов массива.
    n:=0;
//Делаем видимыми первую
//метку и компонент Memo1.
    label1.Visible:=true;
    Memo1.Visible:=true;
//Переменная sum служит для
//нахождения суммы компонентов файла.
    Sum:=0;
//Пока не встретится конец файла,
//будем считывать очередной компонент.
    while not eof(f) do
    begin
//Индекс массива увеличиваем на 1.
        n:=n+1;
//Считываем очередной элемент из
//файла в массив.
        read(f,a[n]);
//Накапливаем сумму элементов массива.
        sum:=sum+a[n];
//Выводим очередной элемент в Memo1.
        Memo1.Lines.Add(FloatToStr(a[n]));
    end;
//Закрываем файл.
    CloseFile(f);
//Находим среднее арифметическое
//элементов массива.
    sum:=sum/n;
//Поиск максимального, минимального
//элементов в массиве и их индексов.
    max:=a[1];min:=max;nmin:=1;nmax:=1;
    for i:=2 to n do
    begin
```

```
        if a[i]>max then
        begin
            max:=a[i]; nmax:=i;
        end;
        if a[i]<min then
        begin
            min:=a[i]; nmin:=i;
        end;
    end;
//Если максимальный элемент расположен
//раньше минимального, то меняем местами
//nmin и nmax.
    if nmax<nmin then
    begin
        i:=nmax;
        nmax:=nmin;
        nmin:=i;
    end;
    i:=nmin+1;
//Цикл для удаления элементов,
//расположенных между максимальным
// и минимальным.
    while(i<nmax) do
    begin
//Если очередной элемент массива
//меньше среднего арифметического, то
        if a[i]<sum then
        begin
//удаляем его.
            for j:=i to n-1 do a[j]:=a[j+1];
//После удаления уменьшаем количество
//элементов и номер максимального на 1.
            n:=n-1;
            nmax:=nmax-1;
        end
//Если очередной элемент массива
//больше среднего арифметического, то
        else
```

```
//переходим к следующему.
            i:=i+1;
    end;
//Делаем видимыми вторую метку
//и компонент Memo2.
    label2.Visible:=true;
    Memo2.Visible:=true;
    rewrite(f);//Открываем файл для записи.
//Преобразованный массив записываем
//в файл и выводим в компонент Memo2.
    for i:=1 to n do
    begin
        write(f,a[i]);
        Memo2.Lines.Add(FloatToStr(a[i]));
    end;
//Закрываем файл, делаем видимой вторую кнопку.
    CloseFile(f);
    Button_Close.Visible:=True;
end;
end;
```

При щелчке по кнопке «**Закрыть**» программа должна завершать свою работу. Поэтому текст обработчика щелчка по кнопке будет очень простым.

```
procedure Tform_File.Button_CloseClick(
                                Sender: TObject);
begin
    Close;
end;
```

Проверим функционирование созданного приложения. После запуска программы появляется окно, подобное представленному на рис. 7.4. Щелкаем по кнопке `Преобразовать файл`, появляется окно, подобное представленному на рис. 7.5. После щелчка по кнопке OK окно программы становится похожим на представленное на рис. 7.6. Щелчок по кнопке `Закрыть` завершает работу приложения. Задача 7.2 решена.

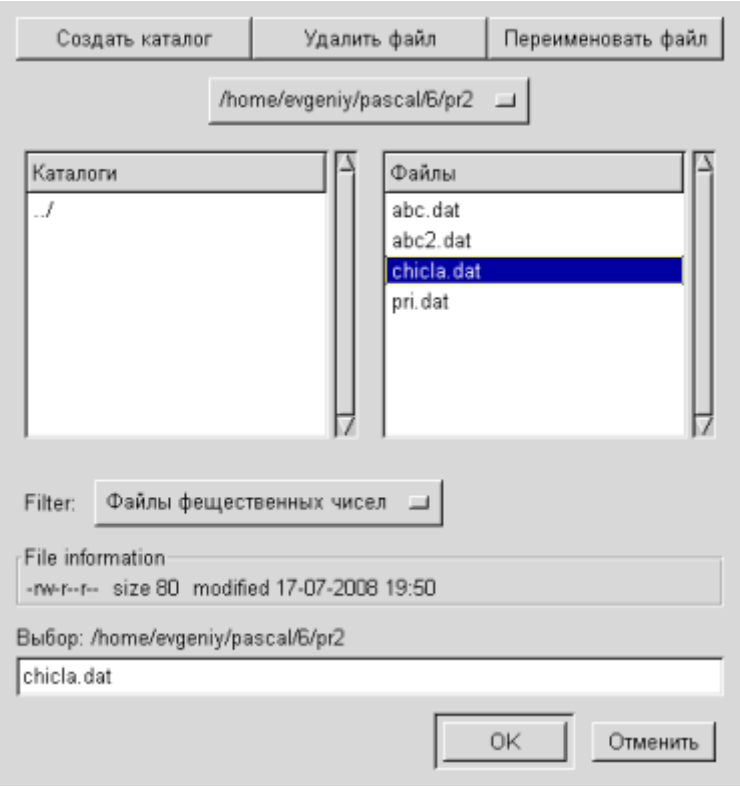

Рисунок 7.5: Диалоговое окно выбора файла

В связи с тем что файл — последовательная структура данных, при решении задачи пришлось содержимое файла целиком считывать в массив, проводить обработку данных в массиве и только затем записывать в файл. Такой метод не всегда удобен. Кроме того, при считывании данных из файла в массив мы ограничены описанием статического массива. Если в файле окажется больше элементов, чем при описании статического массива (в нашем случае >100), то программа не будет правильно работать.

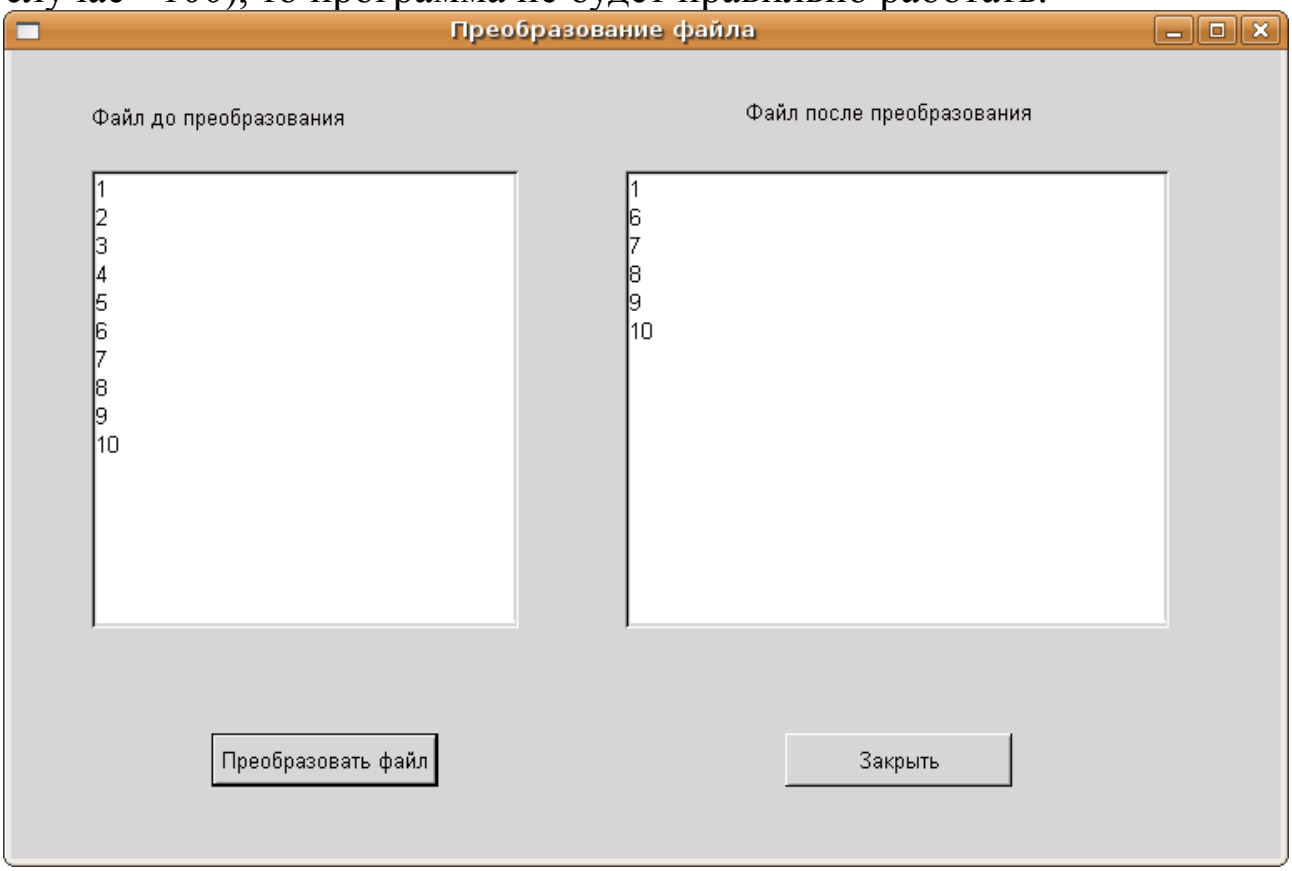

Рисунок 7.6: Окно с результатами работы программы

Использование динамического массива не решит эту проблему, потому что неизвестно количество элементов в массиве. Для решения этой программы в языке FreePascal реализован прямой доступ к файлу с помощью рассмотренных далее подпрограмм работы с файлами.

Одной из проблем при копировании данных из типизированного файла в массив является невозможность корректного выделения памяти под массив. При выделении памяти заранее неизвестно, сколько элементов находится в файле. Для решения этой задачи существует функция `filesize(f)` (`f`– файловая переменная), которая возвращает значение типа `longint` - число реальных компонентов в открытом файле, связанном с файловой переменной `f`. Для пустого файла функция возвращает число `0`.

Применим функцию `filesize` для переписывания вещественных чисел из файла в массив.

ЗАДАЧА 7.3. Переписать содержимое файла вещественных чисел в массив.

```
program Project1;
{$mode objfpc}{$H+}
uses  Classes, SysUtils
            { you can add units after this };
//Massiw - тип данных — массив
//из 1000 элементов.
type
    massiw=array[1..1000] of real;
var
f:file of real;
//a -указатель на массив;
a:^massiw;
n,i:word;
begin
//Связываем файловую переменную
//реальным файлом на диске.
assignfile(f,'/home/pascal/6/pr1/abc.dat');
//Открываем файл для чтения
reset(f);
//Записываем в переменную n - количество
```

```
//элементов в файле f.
n:=filesize(f);
writeln('в файле ',n, ' чисел');
//Выделяем память для динамического массива a
//из n вещественных чисел.
getmem(a,n*sizeof(real));
for i:=1 to n do
begin
//Считываем очередной элемент из файла
//в массив.
read(f,a^[i]);
//Вывод элемента на экран.
write(a^[i]:1:3, ' ')
end;
//Освобождаем память, выделенную
//для динамического массива.
freemem(a,n*sizeof(real));
//Закрываем файл.
closefile(f);
readln;
end.
```

Функция `filepos(f)` возвращает значение типа `longint` – текущую позицию в открытом файле, связанном с файловой переменной `f`. Сразу после открытия файла значение `filepos(f)` равно 0. После прочтения последнего компонента из файла значение `filepos(f)` совпадает со значением `filesize(f)`. Достижение конца файла можно проверить с помощью условия:

```
if filepos(f)=filesize(f) then
writeln('достигнут конца файла');
```

Процедура `seek(f,n)` устанавливает указатель в открытом файле, связанном с файловой переменной `f`, на компонент с номером `n` (нумерация компонентов идет от `0`). Затем значение компонента может быть считано.

Процедура `truncate(f)`, где `f` – имя файловой переменной, отсекает часть открытого файла, начиная с текущего компонента, и подтягивает на его место конец файла.

Использование процедур `seek` и `truncate` рассмотрим на при-

мере решения следующих двух несложных задач по обработке файлов.

ЗАДАЧА 7.4. В файле вещественных чисел */home/pascal/6/pr1/abc.dat* поменять максимальный и минимальный элементы.

Рассмотрим два варианта решения этой задачи.

В первом консольном варианте программы — после считывания компонентов файла в массив происходит поиск минимального и максимального элементов массива и их индексов. Затем максимальное и минимальное значения перезаписываются в файл.

```
program Project1;
{$mode objfpc}{$H+}
uses Classes, SysUtils
{ you can add units after this };
var f:file of real;
i, nmax, nmin :integer;
a:array[0..200] of real;
max,min:real;
begin
assignfile(f,'/home/pascal/6/pr1/abc.dat');
reset(f);
//Считываем компоненты файла в массив a.
for i:=0 to filesize(f)-1 do
begin
read(f,a[i]);
write(a[i]:1:2,' ');
end;
//Начальное присваивание максимального и
//минимального элементов массива и их индексов.
max:=a[0]; nmax:=0;
min:=a[0]; nmin:=0;
//Основной цикл для поиска максимального и
//минимального элементов массива и их индексов.
for i:=1 to filesize(f)-1 do
begin
if a[i]>max then
begin
max:=a[i];
nmax:=i
```

```
end;
if a[i]<min then
begin
min:=a[i];
nmin:=i
end;
end;
//Перезапись максимального и
//минимального значений в файл.
//Передвигаем указатель файла
//к максимальному элементу.
seek(f,nmax);
//Записываем на место максимального
//элемента минимальный.
write(f,min);
//Передвигаем указатель файла
//к минимального элементу.
seek(f,nmin);
//Записываем на место
//максимального элемента минимальный.
write(f,max);
//Обязательно закрываем файл.
closefile(f);readln;
end.
```

Вторую версию программы напишем как полноценное приложение. Использовать массив в программе не будем. Будет организован единственный цикл, в котором очередное значение считывается в переменную `a` и осуществляется поиск минимального и максимального элементов среди компонентов файла и их индексов. Затем происходит перезапись в файл максимального и минимального значений.

Разработку программы начнем с создания шаблона графического приложения (**Проект — Создать Проект — Приложение**).

На форме расположим следующие компоненты:

1. Две метки `Label1` и `Label2` для подписи.
2. `Edit1` — текстовое поле редактирования, в котором будем хранить содержимое исходного файла.
3. `Edit2` — текстовое поле редактирования, в котором хранится файл после преобразования.

4. `OpenDialog1` — компонент для выбора имени обрабатываемого файла.
5. `Button1` — кнопка для запуска программы.
6. `Button2` — кнопка для завершения программы.

Расположим компоненты на форме примерно так, как показано на рис. 7.7.

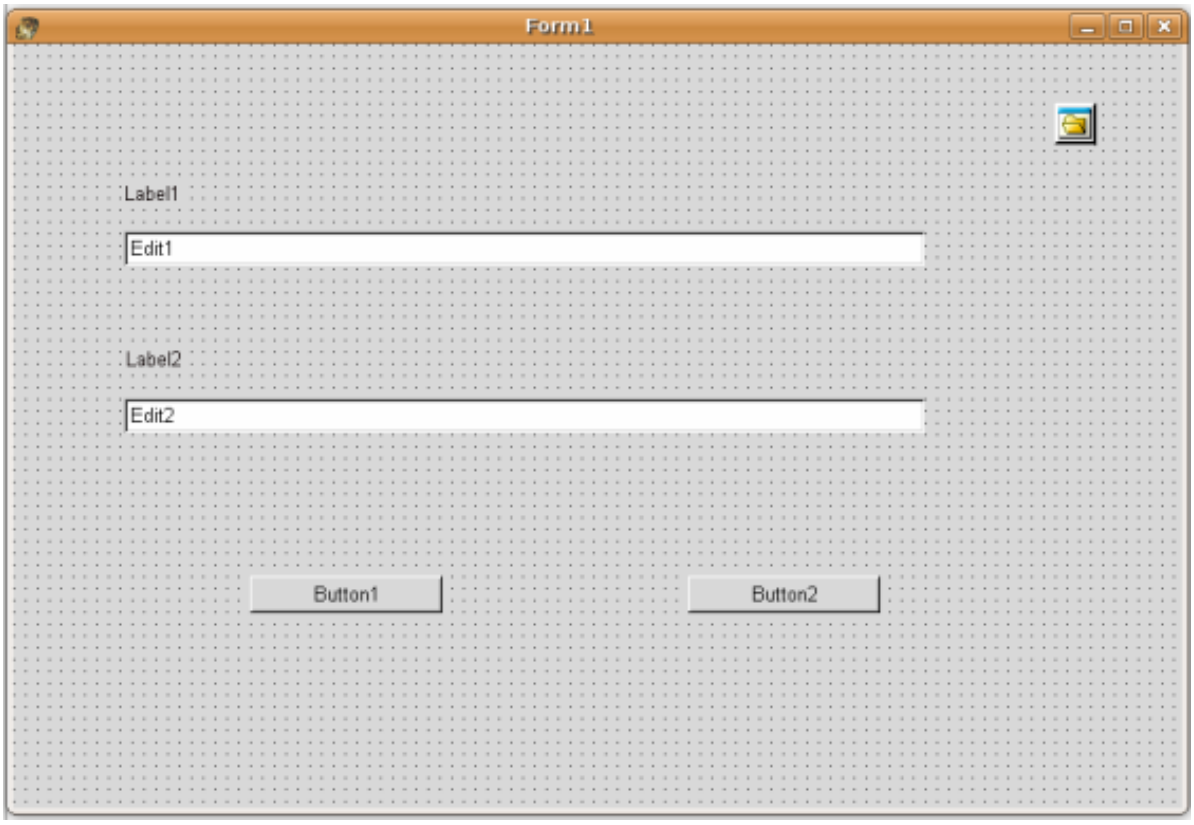

Рисунок 7.7: Окно проекта с компонентами

Основные свойства компонентов будем устанавливать программно при создании формы. Поэтому подпрограмма `FormCreate` будет такой.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Form1.Caption:=
           'Обмен максимального и
           минимального элементов в файле';
    label1.Caption:='Исходный файл';
    label1.Visible:=False;
    label2.Caption:='Файл после
                              преобразования';
```

```
    label2.Visible:=False;
    Edit1.Text:='';
    Edit1.Visible:=false;
    Edit2.Text:='';
    Edit2.Visible:=false;
    OpenDialog1.Filter:=
'Файлы вещественных чисел|*.dat|Все файлы|*.*';
OpenDialog1.InitialDir:='/home/pascal/6/pr1';
    Button1.Caption:='Преобразовать файл';
    Button2.Caption:='Выход';
    Button2.Visible:=False;
end;
```

Авторы надеются, что читателям, дочитавшим книгу до этого момента, текст процедуры `TForm1.FormCreate` понятен без пояснений.

При запуске программы окно формы будет иметь вид, представленный на рис. 7.8.

Основные действия программа будет осуществлять при щелчке по кнопке **Преобразовать файл**, текст соответствующей подпрограммы с комментариями приведен ниже.

```
procedure TForm1.Button1Click(Sender: TObject);
var
f:file of real;
i,nmax,nmin:integer;
a,max,min:real;
s1,s:string;

begin
//Открываем диалоговое окно для выбора файла.
if OpenDialog1.Execute then
begin
//В переменную s записываем полное имя файла.
    s:=OpenDialog1.FileName;
//Связываем файловую переменную
//с файлом на диске.
    assignfile(f,s);
```

Рисунок 7.8: Окно программы решения задачи 7.4 при запуске

```
//Открываем файл в режиме чтения.
    reset(f);
//Делаем видимыми первую метку и поле ввода.
    label1.Visible:=true;
    Edit1.Visible:=true;
//В строке s1 будем формировать
//содержимое файла.
    S1:='';
//Цикл для последовательного чтения
//всех элементов из файла
// вещественных чисел.
    for i:=0 to filesize(f)-1 do
    begin
//Считывание очередного элемента
//массива в переменную a.
        read(f,a);
        if i=0 then
//Начальное присваивание максимального
//и минимального значений и их индексов.
        begin
            max:=a;
```

```
            nmax:=i;
            min:=a;
            nmin:=i;
        end
        else
        begin
//Сравнение текущего значения с
//максимальным (минимальным).
            if max<a then
            begin
                max:=a;
                nmax:=i
            end;
            if min>a then
            begin
                min:=a;
                nmin:=i
            end
        end;
//Добавление очередного значения
//к выходной строке s1.
        s1:=s1+FloatToStr(a)+' ';
    end;
//Вывод содержимого файла
//в первое текстовое поле.
    Edit1.Text:=s1;
//Запрет изменения текстового поля.
    Edit1.ReadOnly:=true;
//Перезапись максимального
//и минимального значений в файл.
//Передвигаем указатель файла
//к максимальному элементу.
    seek(f,nmax);
//Записываем на место
//максимального элемента минимальный.
    write(f,min);
//Передвигаем указатель
//файла к максимальному элементу.
```

```
        seek(f,nmin);
    //Записываем на место
    //минимального элемента максимальный.
        write(f,max);
    //Обязательно закрываем файл.
        closefile(f);
        reset(f);
    //Делаем видимыми вторую метку и поле ввода.
        label2.Visible:=true;
        Edit2.Visible:=true;
    //Считываем данные из преобразованного файла
        s1:='';
        for i:=0 to filesize(f)-1 do
        begin
            read(f,a);
    //Добавление очередного значения
    //из преобразованного файла к строке s1.
            s1:=s1+FloatToStr(a)+' ';
        end;
    //Вывод содержимого преобразованного
    //файла во 2-е текстовое поле.
        Edit2.Text:=s1;
    //Запрет изменения текстового поля.
        Edit2.ReadOnly:=true;
    //Делаем видимой вторую кнопку.
        Button2.Visible:=True;
        CloseFile(f);
        end;
    end;
```

При щелчке по кнопке **Выход** программа должна завершать свою работу. Поэтому текст обработчика щелчка по кнопке будет очень простым.

```
procedure Tform_File.Button2Click(Sender:
                                                TObject);
    begin
        Close;
    end;
```

При щелчке по кнопке `Преобразовать файл`, появляется окно

выбора файла, подобное представленному на рис. 7.5. После выбора файла в файле происходит обмен максимального и минимального элементов и вывод содержимого файла до и после преобразования (см. рис. 7.9).При щелчке по кнопке `Выход` работа программы завершится.

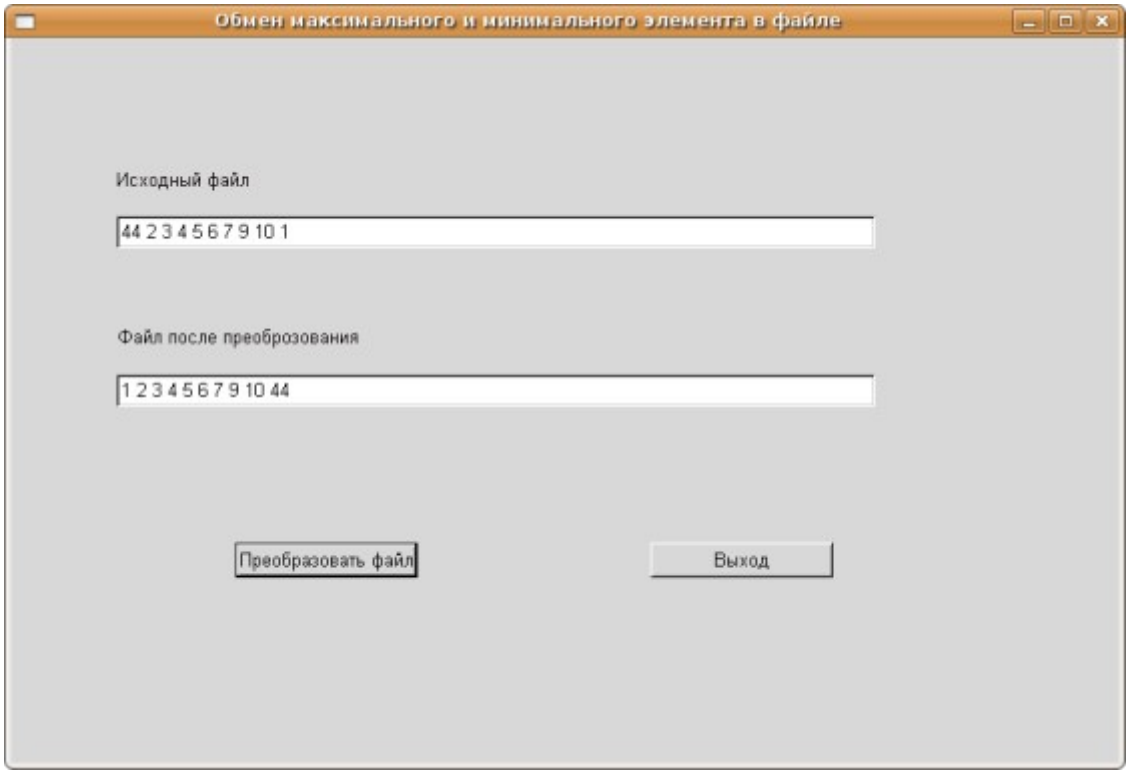

Рисунок 7.9: Окно программы решения задачи 7.4 после преобразования файла

При решении задачи 7.4 была использована процедура `seek`, с помощью которой стало возможным изменение данных в файле непосредственно на диске, без считывания в массив. Однако следует помнить, что большое количество отдельных обращений к файлу занимает больше времени, чем однократное обращение к файлу для обработки большого объема данных.

ЗАДАЧА 7.5. Задан файл вещественных чисел abc.dat. Удалить из него максимальный и минимальный элементы.

Рассмотрим две программы, решающие эту задачу. Алгоритм первой состоит в следующем: считываем компоненты файла в массив, в котором находим максимальный и минимальный элементы и их номера. Открываем файл для записи и вносим в него все элементы, за исключением максимального и минимального.

```
program Project1;
```

```
{$mode objfpc}{$H+}
uses  Classes, SysUtils
{ you can add units after this };
var
f:file of real;
max,min:real;
j, i,nmax,nmin:integer;
a:array [1..300] of real;
begin
assignfile(f,'/home/pascal/6/pr1/abc.dat');
reset(f);
//Запоминаем в переменной j
//количество компонентов в файле.
j:=filesize(f);
//Считываем компоненты файла в массив a.
for i:=1 to j do read(f,a[i]);
for i:=1 to j do write(a[i]:1:2,' ');
writeln;
closefile(f);
//Открываем файл для записи.
rewrite(f);
//Начальное присваивание максимального и
//минимального элементов массива и его индекса.
max:=a[1];min:=a[1];
nmax:=1;nmin:=1;
//Основной цикл для поиска максимального и
//минимального элементов массива и его индекса.
for i:=2 to j do
begin
     if a[i]>max then
     begin
         max:=a[i];
         nmax:=i
     end;
     if a[i]<min then
     begin
         min:=a[i];
         nmin:=i;
```

```
        end;
    end;
    //Перезапись элементов массива в файл,
    //за исключением элементов с
    //номерами nmax и nmin.
    writeln('Преобразованный файл');
    for i:=1 to j do
        if (i<>nmax)and (i<>nmin) then
        begin
            write(f,a[i]);
    //Вывод записанного в файл элемента на экран.
            write(a[i]:1:2,' ');
        end;
    closefile(f);
    readln;
    end.
```

Вторая программа работает следующим образом. Находим максимальный и минимальный элементы и их номера среди компонентов файла. Если nmin>nmax, то меняем содержимое переменных nmin и nmax. Далее элементы, лежащие между минимальным и максимальным (формально между элементами с номерами nmin и nmax), сдвигаем на один порядок влево. Тем самым мы убираем элемент с номером nmin. После этого все компоненты, лежащие после элемента с номером nmax, сдвинем на два порядка влево. Этим мы сотрем максимальный элемент. Затем два последних компонента в файле необходимо удалить.

```
program Project1;
{$mode objfpc}{$H+}
uses  Classes, SysUtils
{ you can add units after this };
var
f:file of real;
a:real;
max,min:real;
i,nmax,nmin:integer;
begin
assign(f,'/home/pascal/6/pr1/abc.dat');
reset (f);
```

```
//Поиск максимального и
//минимального элементов в файле и их индексов.
writeln('Исходный файл');
for i:=0 to filesize(f)-1 do
begin
     read(f,a);
     write(a:1:2,' ');
     if i=0 then
     begin
         max:=a;
         nmax:=i;
         min:=a;
         nmin:=i
     end
     else
     begin
         if a>max then
         begin
             max:=a;
             nmax:=i;
         end;
         if a<min then
         begin
             min:=a;
             nmin:=i;
         end
     end
end;
writeln;
//Сравниваем nmin и nmax.
if nmax<nmin then
begin
     i:=nmax;
     nmax:=nmin;
     nmin:=i
end;
// Сдвигаем элементы, лежащие между
//компонентами с номерами nmin и nmax,
```

```
//на один влево.
for i:=nmin to nmax-2 do
begin
     seek(f,i+1);
     read(f,a);
     seek(f,i);
     write(f,a);
end;
//Сдвигаем элементы, лежащие после компонента
//с номером nmax, на два влево.
for i:=nmax to filesize(f)-3do
begin
     seek(f,i+1);
     read(f,a);
     write(a:1:2);
     seek(f,i-1);
     write(f,a);
     write(a:1:2);
end;
//Отрезаем последние два компонента.
truncate(f);
closefile(f);
reset(f);
//Вывод преобразованного файла.
writeln('Преобразованный файл');
for i:=1 to filesize(f) do
begin
        read(f,a);
//Вывод записанного в файл элемента на экран.
        write(a:1:2,' ');
     end;
closefile(f);
end.
```

В результате программы из файла вещественных чисел удалено наибольшее и наименьшее число.

Кроме типизированных файлов, широкое применение при обработке числовых данных получили бестиповые файлы.

7.3 Бестиповые файлы в языке Free Pascal

Для хранения данных различного типа удобно использовать бестиповые файлы. При открытии бестиповых файлов следует использовать расширенный синтаксис процедур `reset` и `rewrite`.

```
Reset(var f: File; BuferSize:word);
Rewrite(var f: File; BuferSize:word);
```

Необязательный параметр `BufSize` определяет размер блока передачи данных – количество байт, считываемых или записываемых в файл данных за одно обращение к нему. Если этот параметр отсутствует, то используется значение, устанавливаемое по умолчанию (128) при открытии файла. Наибольшей гибкости можно достичь при размере блока 1 байт.

Для записи данных в бестиповый файл используется процедура `BlockWrite`:

```
BlockWrite(var f:file; var X; Count:word;
                          var WriteCount:word);
```

здесь `f` – имя файловой переменной, `X` – имя переменной, из которой данные записываются в файл, `Count` – количество блоков размером `BufSize`, записываемых в файл, необязательный параметр `WriteCount` определяет *реальное количество записанных в файл блоков*.

Процедура `BlockWrite` записывает `Count` блоков в файл, связанный с файловой переменной `f`[77] из переменной `X`. При корректной записи в файл возвращаемое процедурой `BlockWrite` значение `WriteCount` совпадает с `Count`.

При чтении данных из бестипового файла используется процедура `BlockRead`:

```
BlockRead(var f:file; var Y; Count:word;
                          var ReadCount:word);
```

где `f` – имя файловой переменной, `Y` – имя переменной, в которую считываются данные из файла, `Count` – количество блоков размером `BufSize`, считываемых из файла, необязательный параметр `ReadCount` определяет количество блоков размером `BufSize`, реально считанных из файла.

`BlockRead` считывает из файла, связанного с файловой пере-

77 Размер блока определяется при выполнении процедур Reset, Rewrite.

менной f, Count блоков в переменную Y. При корректном чтении из файла возвращаемое процедурой BlockRead значение ReadCount должно совпадать с Count.

Рассмотрим работу с бестиповыми файлами на примере решения следующих задач.

Задача 7.6. В бестиповом файле хранится массив вещественных чисел и количество элементов в нем. Удалить из файла максимальное вещественное число.

Считаем, что файл уже существует, ниже приведен текст консольного приложения создания файла.

```
program Project1;
{$mode objfpc}{$H+}
uses
    Classes, SysUtils
    { you can add units after this };
type
massiv=array[1..100000] of real;
var  i,N:word;
    f:file;
    x:^massiv;
begin
Assignfile(f,'/home/pascal/6/pr_6/new_file.dat');
//Открываем файл для записи,
//размер блока передачи данных 1 байт.
rewrite(f,1);
//Ввод числа N — количества целых
//положительных чисел в файле
write('N=');readln(N);
//Записываем в файл f целое число N,
//а точнее, 2 блока (sizeof(word)=2)
//по одному байту из переменной N.
BlockWrite(f,N,sizeof(word));
//Выделяем память для N элементов массива x
//вещественных чисел.
getmem(x,N*sizeof(real));
//Ввод массива.
writeln('Введите массив');
for i:=1 to N do
```

```
begin
     write('x(',i,')=');
     readln(x^[i]);
end;
for i:=1 to N do
// Записываем в файл f вещественное
//число x^[i], а точнее, 8  блоков
//(sizeof(real)=8) по одному байту
//из переменной x^[i].
BlockWrite(f,x^[i],sizeof(real));
//Запись массива в файл можно осуществить
//и без цикла с помощью следующего обращения
//к функции BlockRead —
//BlockWrite(f,x^,N*sizeof(real));
//Запись массива в бестиповый файл без
//цикла с помощью одного оператора
//BlockWrite(f,x^,N*sizeof(тип))
//кажется авторам предпочтительнее.
//Закрываем файл.
CloseFile(f);
//Освобождаем память.
freemem(x,N*sizeof(real));
readln;
end.
```

Разработку программы решения задачи 7.6 начнем с создания шаблона графического приложения (**Проект — Создать Проект — Приложение**).

На форме расположим следующие компоненты:

1. Две метки `Label1` и `Label2` для подписи.
2. `Edit1` — текстовое поле редактирования, в котором будем хранить содержимое исходного файла.
3. `Edit2` — текстовое поле редактирования, в котором хранится файл после преобразования.
4. `OpenDialog1` — компонент для выбора имени обрабатываемого файла.
5. `Buttom1` — кнопка для преобразования файла.

Расположим компоненты на форме примерно так, как показано на рис. 7.10.

Основные свойства компонентов будем устанавливать программно при создании формы. Программа решения задачи 7.6 будет осуществляться при щелчке по кнопке. Полный текст модуля с необходимыми комментариями.

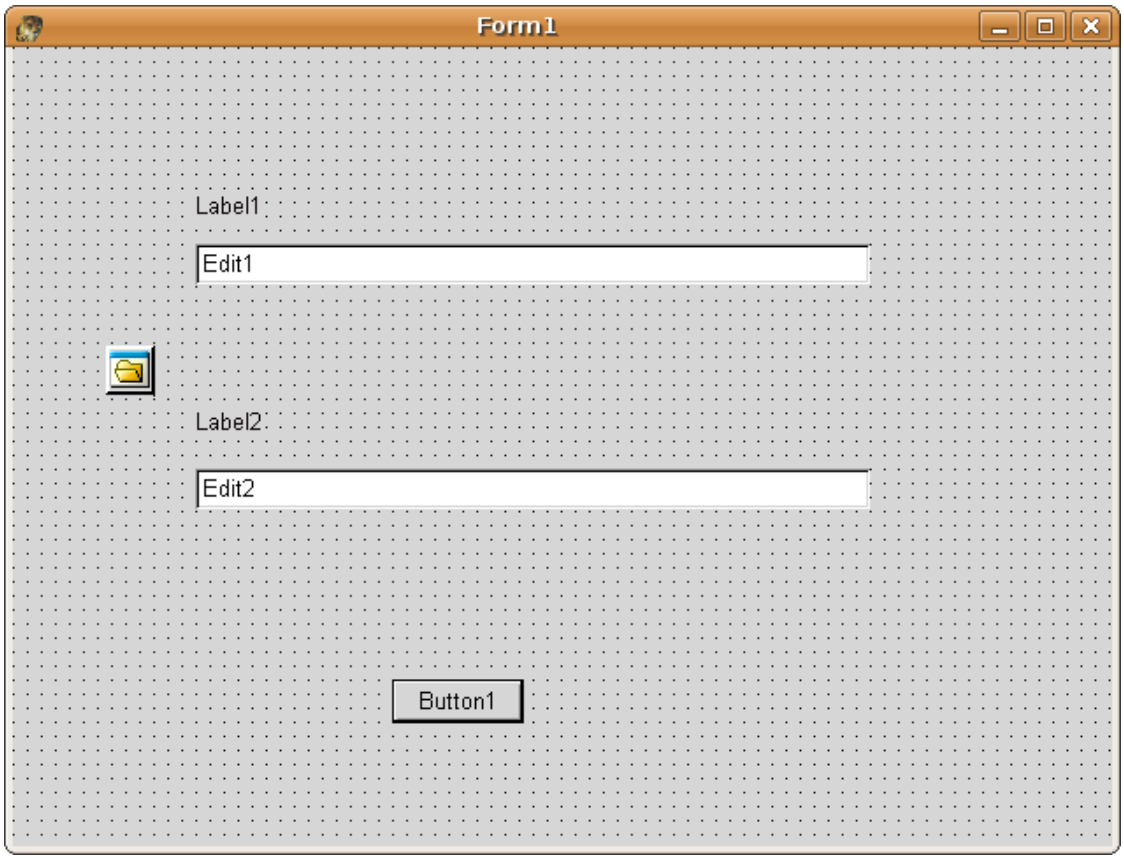

Рисунок 7.10: Окно формы с компонентами

```
   unit Unit1;
   {$mode objfpc}{$H+}
   interface
   uses
   Classes, SysUtils, LResources, Forms, Controls,
Graphics, Dialogs, StdCtrls;
   type
   { TForm1 }
   TForm1 = class(TForm)
   //На форме находятся:
   //Кнопка.
```

```
 Button1: Tbutton;
//Поле редактирования - для вывода
//содержимого исходного файла.
 Edit1: Tedit;
//Поле редактирования - для вывода
//содержимого файла после преобразования.
 Edit2: Tedit;
//Метки для подписи полей редактирования
 Label1: TLabel;
 Label2: Tlabel;
//Компонент для выбора файла
 OpenDialog1: TOpenDialog;
 procedure Button1Click(Sender: TObject);
 procedure FormCreate(Sender: TObject);
private
     private declarations }
public
    { public declarations }
end;
    massiv=array [1..50000] of real;
var
 Form1: TForm1;
 f:file;
implementation
{ TForm1 }

//Текст процедуры обработчика
//события «Щелчок по кнопке»;
procedure Tform1.Button1Click(Sender: TObject);
var
 x:^massiv;
 y:array[1..3000] of real;
 i,N:word;
 max:real;
 nmax:word;
 s:string;
begin
//Выбираем файл, который будем обрабатывать.
```

```
if OpenDialog1.Execute then
begin
//Имя обрабатываемого файла
//записываем в переменную s.
 s:=OpenDialog1.FileName;
//Связываем файловую переменную
//с именем файла, хранящимся в  переменной s.
 AssignFile(f,s);
//Открываем файл для чтения, размер
//блока передачи данных 1 байт.
 Reset(f,1);
//Считываем из файла f целое число (word) N ,
//а точнее, 2 блока (sizeof(word)=2) по одному
//байту из переменной N.
 BlockRead(f,N,sizeof(word));
//Выделяем память для N элементов массива x
//вещественных чисел.
 getmem(x,N*sizeof(real));
//Считываем из файла f N вещественных чисел
//в переменную x.
//Реально считываем N*sizeof(real)
//блоков по 1 байту.
//Заполняем массив X значениями,
//хранящимися в файле.
 BlockRead(f,x^,N*sizeof(real));
//Делаем видимыми первую метку
//и первый компонент Edit1.
 Label1.Visible:=true;
 Edit1.Visible:=true;
 for i:=1 to N do
//Добавление очередного значения из массива x
//к первому текстовому полю редактирования.
//В результате в первом текстом поле будет
//выведен исходный массив вещественных чисел.
Edit1.Text:=Edit1.Text+
        FloatToStrF(x^[i],ffFixed,5,2)+' ';
//Поиск максимального элемента и его номера.
 max:=x^[1];
```

```
 nmax:=1;
 for i:=2 to N do
 if x^[i]>max then
 begin
     max:=x^[i];
     nmax:=i;
 end;
//Удаление максимального элемента из массива.
 for i:=nmax to N-1 do
     x^[i]:=x^[i+1];
//Уменьшение количества элементов в массиве.
 N:=N-1;
//Закрываем файл.
 CloseFile(f);
 AssignFile(f,s);
//Открываем файл для записи.
 Rewrite(f,1);
//Записываем в файл f целое число N,
//а точнее, 2 блока
// (sizeof(word)=2) по одному байту из
//переменной N.
BlockWrite(f,N,sizeof(word));
BlockWrite(f,N,sizeof(word));
//Запись массива в файл с помощью
//обращения к функции BlockRead.
 BlockWrite(f,x^,N*sizeof(real));
//Закрываем файл.
 CloseFile(f);
//Освобождаем память.
 freemem(x,N*sizeof(word));
//Чтение данных из преобразованного файла
 AssignFile(f,s);
 Reset(f,1);
//Считываем из файла f целое число (word) N ,
//а точнее, 2 блока (sizeof(word)=2) по одному
//байту из переменной N.
 BlockRead(f,N,sizeof(word));
//Выделяем память для N элементов массива x
```

```
//вещественных чисел.
 getmem(x,N*sizeof(real));
//Считываем из файла f N вещественных чисел
//в переменную x. Реально считываем
//N*sizeof(real) блоков по 1 байту.
//Заполняем массив X значениями,
//хранящимися в файле.
 BlockRead(f,x^,N*sizeof(real));
//Делаем видимыми вторую метку
//и второй  компонент Edit2.
 Label2.Visible:=true;
 Edit2.Visible:=true;
//Содержимое массива x выводим
//в текстовое поле Edit2.
 for i:=1 to N do
Edit2.Text:=Edit2.Text+
            FloatToStrF(x^[i],ffFixed,5,2)+' ';
//Закрваем файл и освобождаем память.
 CloseFile(f);
 freemem(x,N*sizeof(real));
end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
//При создании формы устанавливаем
//свойства компонентов Edit1,
// Edit2, label1, label2 и кнопки Button1.
//Все компоненты,
//кроме кнопки, делаем невидимыми.
 Form1.Caption:='Работа с бестиповыми файлами';
 Label1.Width:=80;
 Label1.Caption:='Содержимое исходного файла';
 Edit1.Clear;
 Label2.Caption:=
        'Содержимое преобразованного файла';
 Label2.Width:=80;
 Edit2.Clear;
```

```
 Button1.Width:=150;
 Button1.Caption:='Преобразование файла';
 Label1.Visible:=false;
 Edit1.Visible:=false;
 Label2.Visible:=false;
 Edit2.Visible:=false;
end;
initialization
 {$I unit1.lrs}
end.
```

При запуске программы окно приложения будет таким, как показана на рис. 7.11. После щелчка по кнопке **Преобразовать файл** появится окно выбора файла (см. рис. 7.12). Окно программы после преобразования файла представлено на рис. 7.13.

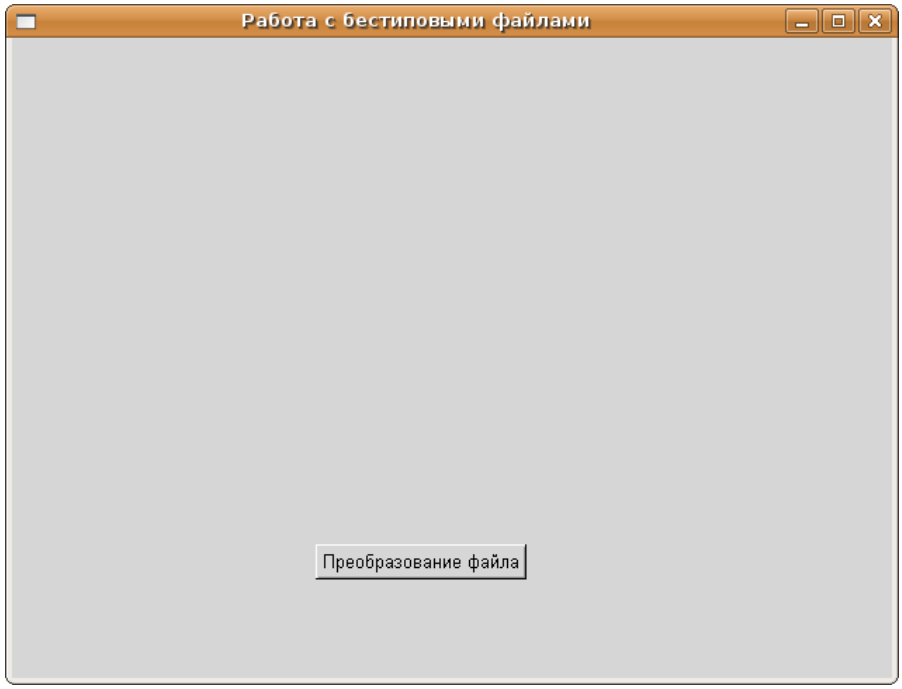

Рисунок 7.11: Окно программы решения задачи 7.6 при запуске

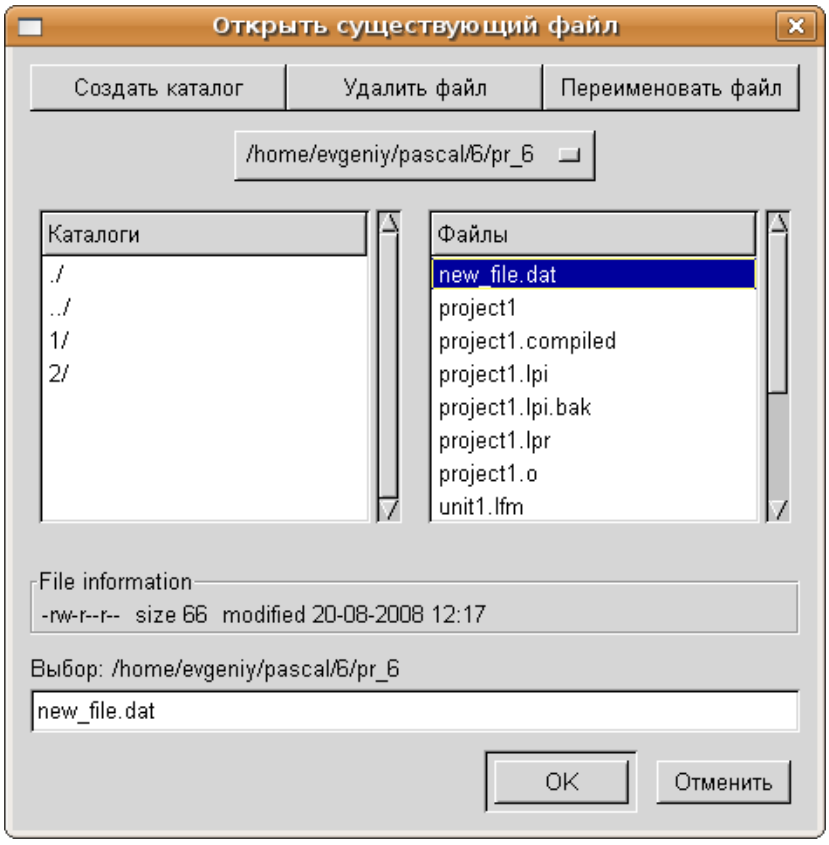

Рисунок 7.12: Окно выбора файла

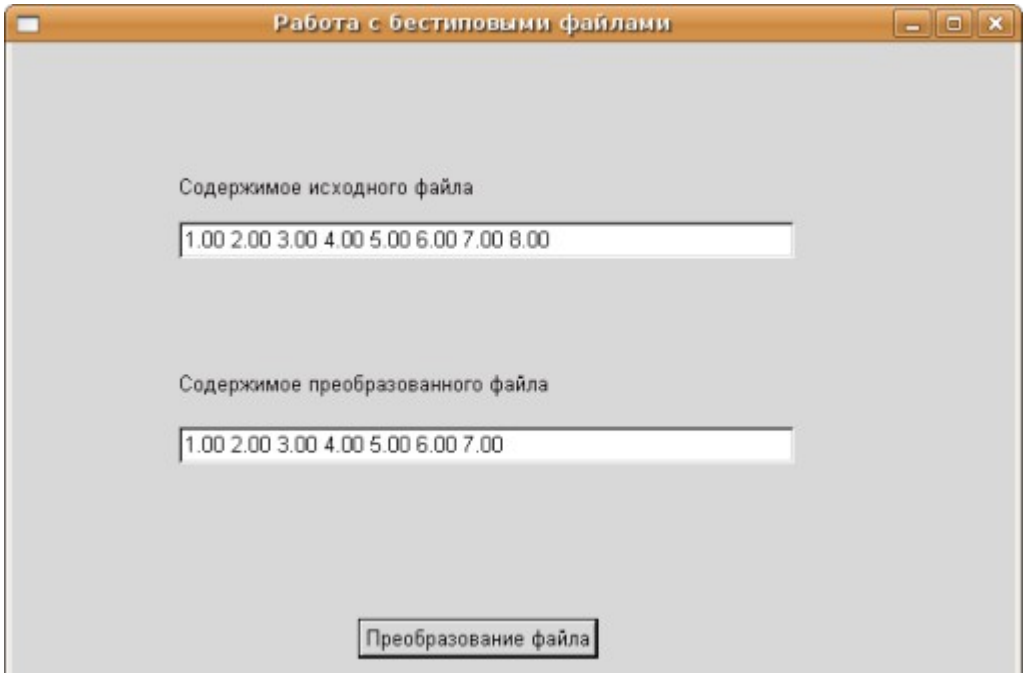

Рисунок 7.13: Окно работающей программы

Задача 7.7. В бестиповом файле хранятся матрицы вещественных чисел A(N,M), B(P,L) и их размеры. Умножить, если это возможно, A на B, результирующую матрицу $C = A \cdot B$ дописать в файл.

Решение задачи начнем с написания консольной программы создания файла. Для этого определимся со структурой файла. Пусть в нем хранится два значения типа `word N` и `M`, затем матрица вещественных чисел `A(N,M)`, потом `P` и `L` типа `word` и матрица `B(P,L)`.

Ниже приведен листинг подпрограммы создания бестипового файла с комментариями.

```
program Project1;
{$mode objfpc}{$H+}
uses
Classes, SysUtils
{ you can add units after this };
var
 f:file;
 i,j,n,m,l,p:word;
 a,b:array[1..20,1..20] of real;
begin
 Assign(f,'Prim.dat');
//Открываем файл для записи, размер
//блока передачи данных 1 байт.
 rewrite(f,1);
 write('N='); readln(N);
 write('M='); readln(M);
//Записываем в файл f целое число N,
//а точнее, 2 блока
//(sizeof(word)=2) по одному байту
//из переменной N.
 BlockWrite(f,N,sizeof(word));
//Записываем в файл f целое число M,
//а точнее, 2 блока (sizeof(word)=2)
//по одному байту из переменной M.
 BlockWrite(f,M,sizeof(word));
writeln('Matrica A');
 for i:=1 to N do
 for j:=1 to M do
 begin
//Вводим очередной элемент матрицы.
    read(A[i,j]);
//Записываем в файл f вещественное
```

```
//число A[i,j], а точнее, sizeof(real) блоков
//по одному байту из переменной A[i,j].
    BlockWrite(f,A[i,j],sizeof(real));
 end;
 write('L='); readln(L);
 write('P='); readln(P);
//Записываем в файл 2 блока по одному байту
//из переменной L.
 BlockWrite(f,L,sizeof(word));
//Записываем в файл 2 блока по одному
//байту из переменной P.
 BlockWrite(f,P,sizeof(word));
 writeln('Matrica B');
 for i:=1 to L do
 for j:=1 to P do
 begin
    read(B[i,j]);
//Записываем в файл sizeof(real) блоков
//по одному байту из переменной B[i,j].
    BlockWrite(f,B[i,j],sizeof(real));
 end;
//Закрываем файл.
 close(f);
end.
```

Теперь напишем программу, которая из бестипового файла считает матрицы A, B, их размеры, вычислит матрицу C как произведение A на B и допишет матрицу C в исходный файл.

```
program Project1;
{$mode objfpc}{$H+}
uses
Classes, SysUtils
{ you can add units after this };
var
 f:file;
 k,i,j,n,m,l,p:word;
 a,b,c:array[1..20,1..20] of real;
begin
//Связываем файловую переменную с файлом
```

```
//на диске.
 Assign(f,'Pr_7_6.dat');
//Открываем файл и устанавливаем размер
//блока в 1 байт.
 reset(f,1);
//Считываем в переменную N из файла f 2
//байта (sizeof(word)=2
//блоков по одному байту).
 BlockRead(f,N,sizeof(word));
//Считываем в переменную M из файла f 2 байта
//( sizeof(word)=2 блоков по одному байту).
 BlockRead(f,M,sizeof(word));
 for i:=1 to N do
 for j:=1 to M do
//Считываем в переменную A[i,j] из
//файла f sizeof(real) байт
//( sizeof(real) блоков по одному байту).
     BlockRead(f,A[i,j],sizeof(real));
//Считываем в переменную L из файла f 2 байта.
 BlockRead(f,L,sizeof(word));
//Считываем в переменную P из файла f 2 байта.
 BlockRead(f,P,sizeof(word));
 for i:=1 to L do
 for j:=1 to P do
//Считываем в переменную B[i,j] из
//файла f sizeof(real) байт.
     BlockRead(f,B[i,j],sizeof(real));
//Вывод матрицы A на экран.
 writeln('Matrica A');
 for i:=1 to N do
 begin
     for j:=1 to M do
         write(A[i,j]:1:2,' ');
     writeln
 end;
//Вывод матрицы B на экран.
 writeln('Matrica B');
 for i:=1 to L do
```

```
begin
    for j:=1 to P do
        write(B[i,j]:1:2,' ');
    writeln
end;
//Проверяем, возможно ли умножение матриц,
//если да,
if M=L then
begin
//то умножаем матрицу A на B
    for i:=1 to N do
        for j:=1 to P do
        begin
            c[i,j]:=0;
            for k:=1 to M do
                c[i,j]:=c[i,j]+a[i,k]*b[k,j]
        end;
//Вывод матрицы C.
    writeln('Matrica C=A*B');
    for i:=1 to N do
    begin
        for j:=1 to P do
            write(C[i,j]:1:2,' ');
        writeln
    end;
//Дописываем в файл f целое число N,  а точнее,
//2 блока по одному байту из переменной N.
    BlockWrite(f,N,sizeof(word));
//Дописываем в файл f целое число P, а точнее,
//2 блока по одному байту из переменной P.
    BlockWrite(f,P,sizeof(word));
    for i:=1 to N do
        for j:=1 to P do
//Записываем в файл f вещественное число
//C[i,j], а точнее, sizeof(real) блоков по
//одному байту из переменной C[i,j].
    BlockWrite(f,c[i,j],sizeof(real));
end
```

```
 else
     writeln('Умножение невозможно');
//Закрываем файл.
 close(f);
end.
```

7.4 Обработка текстовых файлов в языке Free Pascal

При работе с текстовыми файлами следует учесть следующее:

1. Действие процедур `reset`, `rewrite`, `close`, `rename`, `erase` и функции `eof` аналогично их действию при работе с компонентными (типизированными) файлами.

2. Процедуры `seek`, `truncate` и функция `filepos` не работают с текстовыми файлами.

3. При работе с текстовыми файлами можно пользоваться процедурой `append(f)`, где `f` – имя файловой переменной, которая служит для специального открытия файлов для дозаписи. Она применима только к уже физически существующим файлам, открывает и готовит их для добавления информации в конец файла.

4. Запись и чтение в текстовый файл осуществляются с помощью процедур `write`, `writeln`, `read`, `readln` следующей структуры:

```
read(f, x1, x2, x3, ..., xn);
read(f, x);
readln(f, x1, x2, x3, ..., xn);
readln(f, x);
write(f, x1, x2, x3, ..., xn);
write(f, x);
writeln(f, x1, x2, x3, ..., xn);
writeln(f, x);
```

В этих операторах `f` — файловая переменная. В операторах чтения (`read`, `readln`) `x, x1, x2, x3, ..., xn` — переменные, в которые происходит чтение из файла. В операторах записи `write`, `writeln` `x, x1, x2, x3, ..., xn` — переменные или константы, информация из которых записывается в файл.

Есть ряд особенностей при работе операторов `write`, `writeln`,

`read`, `readln` с текстовыми файлами. Имена переменных могут быть целого, вещественного, символьного и строкового типа. Перед записью данных в текстовый файл с помощью процедуры `write` происходит их преобразование в тип `string`. Действие оператора `writeln` отличается тем, что записывает в текстовый файл символ конца строки.

При чтении данных из текстового файла с помощью процедур `read`, `readln` происходит преобразование из строкового типа к нужному типу данных. Если преобразование невозможно, то генерируется код ошибки, значение которого можно узнать, обратившись к функции `IOResult`. Компилятор FreePascal позволяет генерировать код программы в двух режимах: с проверкой корректности ввода-вывода и без нее.

В программу может быть включен ключ режима компиляции. Кроме того, предусмотрен перевод контроля ошибок ввода-вывода из одного состояния в другое:

{$I+} – режим проверки ошибок ввода-вывода включен;

{$I-} – режим проверки ошибок ввода-вывода отключен.

По умолчанию, как правило, действует режим {$I+}. Можно многократно включать и выключать режимы, создавая области с контролем ввода и без него. Все ключи компиляции описаны в приложении.

При включенном режиме проверки ошибка ввода-вывода будет фатальной, программа прервется, выдав номер ошибки.

Если убрать режим проверки, то при возникновении ошибки ввода-вывода программа не будет останавливаться, а продолжит работу со следующего оператора. Результат операции ввода-вывода будет не определен.

Для опроса кода ошибки лучше пользоваться специальной функцией `IOResult`, но необходимо помнить, что опросить ее можно только один раз после каждой операции ввода или вывода: она обнуляет свое значение при каждом вызове. `IOResult` возвращает целое число, соответствующее коду последней ошибки ввода-вывода. Если `IOResult=0`, то при вводе-выводе ошибок не было, иначе `IOResult` возвращает код ошибки. Некоторые коды ошибок приведены в табл. 7.9.

Таблица 7.9: Коды ошибок

Код ошибки	Описание
2	файл не найден
3	путь не найден
4	слишком много открытых файлов
5	отказано в доступе
12	неверный режим доступа
15	неправильный номер диска
16	нельзя удалять текущую директорию
100	ошибка при чтении с диска
101	ошибка при записи на диск
102	не применена процедура Assign
103	файл не открыт
104	файл не открыт для ввода
105	файл не открыт для вывода
106	неверный номер
150	диск защищён от записи

Рассмотрим несколько практических примеров обработки ошибок ввода-вывода:

1. При открытии проверить, существует ли заданный файл и возможно ли чтение данных из него.

```
assign (f, 'abc.dat');
{$I-}
reset(f);
{$I+}
if IOResult<>0 then
writeln ('файл не найден или не читается')
else
begin
read(f,...);
...
close(f);
end;
```

2. Проверить, является ли вводимое с клавиатуры число целым.

```
var i:integer;
begin
{$I-}
repeat
write('введите целое число i');
readln(i);
until (IOResult=0);
{$I+}
{ Этот цикл повторяется до тех пор,
пока не будет введено целое число}
end.
```

При работе с текстовым файлом необходимо помнить специальные правила чтения значений переменных:

- если вводятся числовые значения, то два числа считаются разделенными, если между ними есть хотя бы один пробел, или символ табуляции, или символ конца строки;
- при вводе строк начало текущей строки идет сразу за последним введенным до этого символом. Вводится количество символов, равное объявленной длине строки. Если при чтении встретился символ «конец строки», то работа с этой строкой заканчивается. Сам символ конца строки является разделителем и в переменную никогда не считывается;
- процедура `readln` считывает значения текущей строки файла, курсор переводится в новую строку файла, и дальнейший ввод осуществляется с нее.

В качестве примера работы с текстовыми файлами рассмотрим следующую задачу.

ЗАДАЧА 7.8. В текстовом файле **abc.txt** находятся матрицы `A(N,M)` и `B(N,M)` и их размеры. Найти матрицу C=A+B, которую дописать в файл **abc.txt**.

Сначала создадим текстовый файл **abc.txt** следующей структуры: в первой строке через пробел хранятся размеры матрицы (числа N и M), затем построчно хранятся матрицы `A` и `B`.

На рис. 7.14 приведен пример файла **abc.txt**, в котором хранятся матрицы A(4,5) и B(4,5).

Текст консольного приложения решения задачи 7.8 с комментариями приведен ниже.

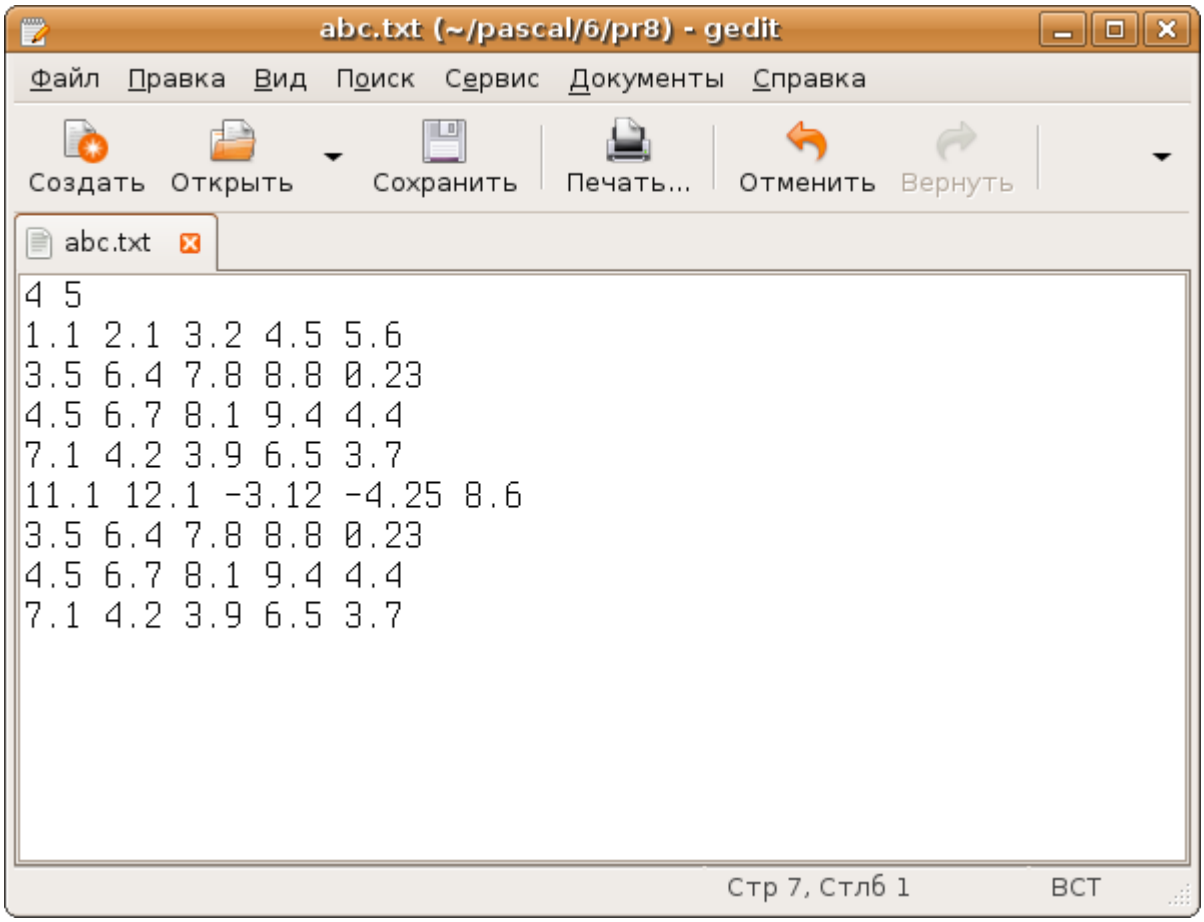

Рисунок 7.14: Содержимое файла ***abc.txt***

```
program project1;
{$mode objfpc}{$H+}
uses
 Classes, SysUtils
 { you can add units after this };
var
 f:Text;
i,j,N,M:word;
 a,b,c:array[1..1000,1..1000] of real;
begin
//Связываем файловую переменную f
//с файлом на диске.
 AssignFile(f,'abc.txt');
//Открываем файл в режиме чтения.
 Reset(f);
//Считываем из первой строки файла abc.txt
```

```
//значения N и M.
 Read(f,N,M);
//Последовательно считываем элементы
//матрицы A из файла.
 for i:=1 to N do
 for j:=1 to M do
 read(f,a[i,j]);
//Последовательно считываем элементы
//матрицы B из файла.
 for i:=1 to N do
 for j:=1 to M do
 read(f,b[i,j]);
//Формируем матрицу C=A+B.
 for i:=1 to N do
 for j:=1 to M do
 c[i,j]:=a[i,j]+b[i][j];
//Закрываем файл f.
 CloseFile(f);
//Открываем файл в режиме дозаписи.
 Append(f);
//Дозапись матрицы C в файл.
 for i:=1 to N do
 begin
     for j:=1 to M do
//Дописываем очередной элемент
//матрицы и пробел в текстовый файл.
     write(f,c[i,j]:1:2,' ');
//По окончании вывода строки матрицы,
//переходим на новую строку в текстовом файле.
     writeln(f);
 end;
//Закрываем файл.
 CloseFile(f);
end.
```

После работы программы файл **abc.txt** будет примерно таким, как показано на рис. 7.15.

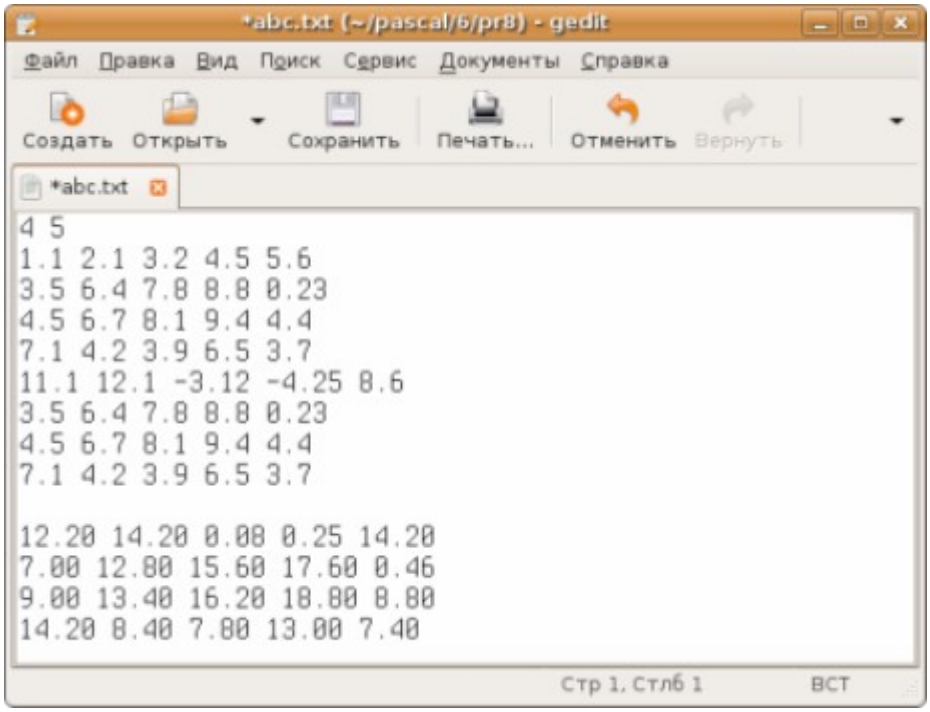

Рисунок 7.15: Файл abc.txt после дозаписи матрицы C

7.5 Задачи для самостоятельного решения

Во всех заданиях составить две программы. Первая должна формировать типизированный файл. Вторая – считать данные из этого файла, выполнить соответствующие вычисления и записать их результаты в текстовый файл.

1. Создать типизированный файл, куда записать `n` целых чисел. Из исходного файла сформировать массивы четных и нечетных чисел. Определить наибольший отрицательный компонент файла и наименьший положительный.

2. Создать типизированный файл, куда записать `n` целых чисел. На основе исходного файла создать массив утроенных четных чисел. Упорядочить его по убыванию элементов.

3. Создать типизированный файл, куда записать `n` целых чисел. Сформировать массив положительных чисел, делящихся на семь без остатка, используя элементы исходного файла. Упорядочить массив по возрастанию элементов.

4. Создать типизированный файл, куда записать `n` вещественных чисел. Из компонентов исходного файла сформировать массивы, из чисел, больших 10 и меньших двух. Вычислить количество нулевых компонентов файла.

5. Создать типизированный файл, куда записать n целых чисел. Из файла создать массив, элементы которого являются простыми числами и расположены после максимального элемента.

6. Создать типизированный файл, куда записать n целых чисел. Из файла целых чисел сформировать массив, записав в него только четные компоненты, находящиеся до минимального элемента.

7. Создать типизированный файл, куда записать n вещественных чисел. Сделать массив из элементов исходного файла, внеся в него числа, превосходящие среднее значение среди положительных значений файла.

8. Создать типизированный файл, куда записать n целых чисел. Из исходного файла сформировать массив, записав в него числа, расположенные в файле до максимального элемента и после минимального.

9. Создать типизированный файл, куда записать n целых чисел. Массив создать из исходного файла. Внести в него простые и совершенные числа, расположенные в файле между минимальным и максимальным элементами.

10. Создать типизированный файл, куда записать n целых чисел. Из исходного файла сформировать массив, в котором вначале расположить четные, а затем нечетные числа. Определить номера наибольшего нечетного и наименьшего четного компонентов.

11. Создать типизированный файл, куда записать n целых чисел. В файле поменять местами минимальный среди положительных элементов и третий по счету простой элемент.

12. Создать типизированный файл, куда записать n целых чисел. Из файла переписать все простые, расположенные после максимального элемента в новый файл.

13. Создать типизированный файл, куда записать n целых чисел. Найти среднее арифметическое среди положительных чисел, расположенных до второго простого числа.

14. Создать типизированный файл, куда записать n целых чисел. Поменять местами последнее совершенное и третье отрицательное числа в файле.

15. Создать типизированный файл, куда записать n целых чисел. Все совершенные и простые числа из исходного файла записать в массив, который упорядочить по возрастанию.

16. Создать типизированный файл, куда записать n целых чисел. Последнюю группу расположенных подряд положительных чисел из исходного файла переписать в текстовый файл.

17. Создать типизированный файл, куда записать n целых чисел. Найти в нем группу подряд расположенных простых элементов наибольшей длины.

18. Создать типизированный файл, куда записать n целых чисел. Из исходного файла сформировать массивы простых и отрицательных чисел. Определить наименьшее простое число в файле и наибольшее совершенное.

19. Создать типизированный файл, куда записать n целых чисел. Из файла создать массив, элементы которого не являются простыми числами и расположены до максимального значения файла.

20. Создать типизированный файл, куда записать n целых чисел. Из файла целых чисел сформировать массив, записав в него только кратные 5 и 7 значения, находящиеся после максимального элемента файла.

21. Создать типизированный файл, куда записать n вещественных чисел. Сделать массив из элементов исходного файла, внеся в него числа, превосходящие среднее значение среди положительных значений файла.

22. Создать типизированный файл, куда записать n вещественных чисел. Поменять местами последнее отрицательное число в файле с четвертым по счету числом.

23. Создать типизированный файл, куда записать n вещественных чисел. Найти сумму третьей группы подряд расположенных отрицательных элементов.

24. Создать типизированный файл, куда записать n целых чисел. Удалить из него четвертую группу, состоящую из подряд расположенных простых чисел.

25. Создать типизированный файл, куда записать n целых чисел. Найти разность между суммой простых чисел, находящихся в файле, и максимальным отрицательным значением файла.

8 Работа со строками и записями

В этой главе мы познакомимся с двумя типами данных – строками и записями. На примерах будет показана работа основных функций обработки строк. Далее читатель может познакомиться с записями (*record*). *Запись* – это сложная структура данных, состоящая из фиксированного количества объектов, называемых *полями*. Поля записи могут быть разного типа.

8. 1 Обработка текста

Часто в задачах программирования приходится обрабатывать текст. Обработка текста в Free Pascal состоит из обработки символов и обработки строк. Понятие строки уже было введено в главе 2 и там же рассмотрены основные функции работы со строками. Напомним основные моменты.

Символ - это буква, цифра или какой-либо знак. Кодовая таблица символов состоит из 256 позиций, т.е. каждый символ имеет свой уникальный код от 0 до 255. Так как код символа представляет собой число не более 255, то очевидно, что в памяти компьютера один символ занимает 1 байт. Для работы с символами существует тип данных `char` (1 байт).

Строка – это набор символов. Текстовая строка имеет определённую длину. Длина строки – это количество символов, которые она содержит. Если один символ занимает 1 байт, то строка из N символов занимает в памяти соответственно N байт. Для работы со строками существует тип данных `string`.

Для задания в программе символьной или строковой переменной используются одинарные кавычки, например:

```
s:='Q'; s1:='Privet';
```

В программе можно ограничить длину текста, хранимого в строковой переменной. Для этого строковую переменную нужно объявить следующим образом (в скобках указывается максимальная длина строки):

```
var str: string[20];
```

Рассмотрим основные операции со строками.

Одной из основных операцией является объединение строк. Во Free Pascal для этого используют операцию «+». Например:

```
var s: string;
begin
 s:='my ' + ' ' + 'text';
...
```

Результатом работы оператора будет строка `my text`.

Строка - это массив символов. Для обращения к элементу строки достаточно указать его номер в квадратных скобках рядом с именем строки. Например:

```
var  s: string; c: char;
begin
 s:='Privet';
 c:=s[4];
 ...
```

Результатом работы будет символ `v`.

Основные функции и процедуры обработки строк, приведены в табл. 2.7. Некоторые из них мы неоднократно применяли при решении задач. В основном это были функции преобразования строк в численные типы данных и наоборот. Рассмотрим другие функции работы со строками из табл. 2.7 на примере:

```
var  Str1, Str2, Str3 , Str4, Str5: String;
     K, L : Integer;
begin
 Str1:='Иванов';    //Первая строка.
 Str2:='Сергей';    //Вторая строка.
 Writeln('Первая строка -', Str1);
 Writeln('Вторая строка -', Str2);
//---------------------------------
//Результат работы операторов:
//Первая строка — Иванов
//Вторая строка — Сергей
//---------------------------------
//Объединение 1-й и 2-й строк.
 Str3:=Str1+' '+Str2;  //Третья строка.
Writeln('Третья строка -', Str3);
//---------------------------------
//Результат работы операторов:
//Третья строка — Иванов Сергей
//---------------------------------
```

```
//Определение длины строки.
 L:= Length(Str3);
Writeln('Длина третьей строки-',L,' символов');
//---------------------------------
//Результат работы операторов:
//Длина третьей строки — 13 символов
//---------------------------------
 Str4:='в'; //Четвертая строка.
//Поиск буквы «в» в третьей строке.
 K:=Pos(Str4, Str3);
 Writeln('Позиция «в» в третьей строке -',K);
//---------------------------------
//Результат работы операторов:
//Позиция «в» в третьей строке - 2
//---------------------------------
//Копирование в переменную Str4 6
//символов, начиная с 8-го, из строки Str3.
 Str4:=copy(Str3,8,6);
 Writeln('Четвертая строка -',Str4);
//---------------------------------
//Результат работы операторов:
//Четвертая строка - Сергей
//---------------------------------
//Выделение первого слова из третьей строки.
//Первое слово - набор букв от первого
//символа до пробела.
 Str5:=Copy(Str3,1,Pos(' ',Str3)-1);
 Writeln('Первое слово в 3-й строке - ',Str5);
//---------------------------------
//Результат работы операторов:
//Первое слово в третьей строке - Иванов
//---------------------------------
//Удаление из третьей строки
//двух символов, начиная с 5-й позиции.
 delete(Str3,5,2);
 Writeln('3-я строка после удаления -',Str3);
//---------------------------------
//Результат работы операторов:
```

```
//Третья строка после удаления — Иван Сергей
//-------------------------------
end.
```

ЗАДАЧА 8.1. Найти сумму элементов числового массива.

Задача очень простая. Ее алгоритм подробно рассмотрен в п. 5.5 . Но так как в данной главе речь идет об обработке строк, предположим, что массив вводится в поле `Edit` формы, то есть представляет собой строку символов. Значит, для решения задачи нужно строку преобразовать в числовой массив. Это можно сделать так. Из исходной строки выделить группу символов (подстроку), расположенных до первого пробела. Полученную подстроку преобразовать в число. Если преобразование прошло успешно, получен первый элемент числового массива. Далее нужно удалить подстроку и следующий за ней пробел из исходной строки. Исходная строка стала короче, и второй элемент массива передвинулся в начало. Выделение подстроки до первого пробела и преобразование ее в число даст второй элемент массива и так далее.

Итак, создадим новый проект. На форме (рис. 8.1) разместим поле `Edit1` для ввода исходных данных, компонент `ListBox1` для вывода результатов и кнопку `Button1` для запуска проекта.

Предположим, что свойство `Text` поля `Edit1` имеет вид:

```
Text =1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

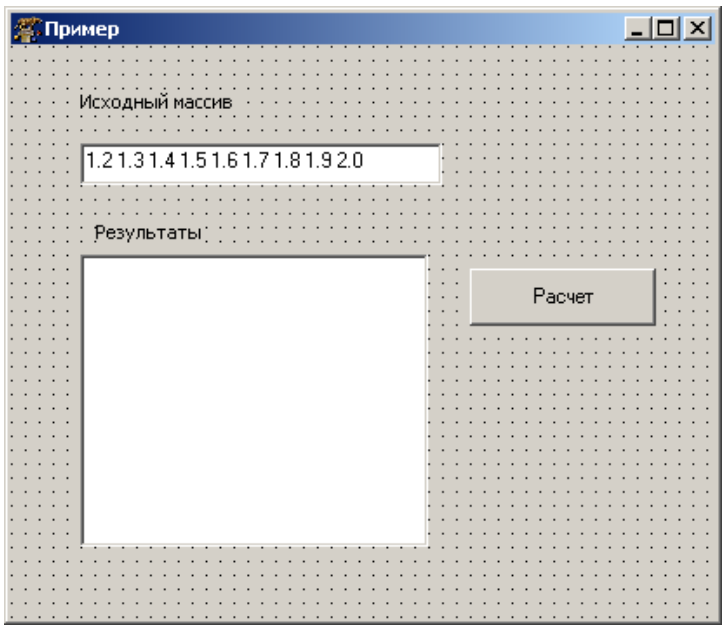

Рисунок 8.1: Окно формы для считывания массива

Ниже приведен листинг программы с комментариями. Результаты работы приложения показаны на рис. 8.2.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes, SysUtils, LResources, Forms,
        Controls, Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
 TForm1 = class(TForm)
 Button1: TButton;
 Edit1: TEdit;
 Label1: TLabel;
 Label2: TLabel;
 ListBox1: TListBox;
 Memo1: TMemo;
 procedure Button1Click(Sender: TObject);
 private
     { private declarations }
 public
     { public declarations }
end;
var
     Form1: TForm1;
implementation
 { TForm1 }
procedure TForm1.Button1Click(Sender: TObject);
var  Str1, Str2, Str3: String;
     i: integer;
     word: String;
     x: array [0..100] of real;
     Sum: real;
begin
//Чтение строки из поля Edit1 диалогового окна.
 Str1:=Edit1.text;
//Вывод строки в поле ListBox1.
 ListBox1.Items.Add(Str1);
```

```
 //Если первый символ – пробел, то удаляем его
 if Str1[1] = ' ' then Delete(Str1,1,1);
 //Если в строке есть двойные пробелы,
//то заменяем их одинарным.
 while Pos('  ',Str1) > 0 do
     Delete(Str1,Pos('  ',Str1),1);
 //Пусть последний символ в строке — пробел.
 if Str1[Length(Str1)] <> ' ' then
     Str1:=Str1+' ';
 i:=0;   Sum:=0;
 //Организуем цикл для работы
 //с элементами строки.
 repeat
 //Выделяем группу символов до пробела.
     Word:=Copy(Str1,1,Pos(' ',Str1)-1);
//Преобразовываем строку в вещественное число,
//записываем его в i-ый элемент массива
     x[i]:=StrToFloat(Word);
 //Вычисляем сумму элементов массива.
     sum:=sum+x[i];
     inc(i);
 //Удаляем из исходной строки подстроку
 //и следующий за ней пробел.
     Delete(Str1,1,Length(Word)+1);
//Выводим в поле ListBox1 подстроку.
     ListBox1.Items.Add(Word);
 //Цикл выполняется до тех пор, пока
 //длина строки не будет равна 0.
 until Length(Str1) = 0;
 //Преобразуем вещественное значение суммы
 //в строковое и формируем строку для
 //вывода в поле ListBox1.
 Str2:='Summa ='+FloatToStr(sum);
 //Преобразуем целочисленное значение
 //количества элементов массива в строковое
 //и формируем строку для вывода
 //в поле ListBox1.
 Str3:='Kolichestvo elementov ='+IntToStr(i);
```

```
//Выводим сумму элементов массива.
 ListBox1.Items.Add(str2);
//Выводим количество элементов массива.
 ListBox1.Items.Add(str3);
end;
initialization
     {$I unit1.lrs}
end.
```

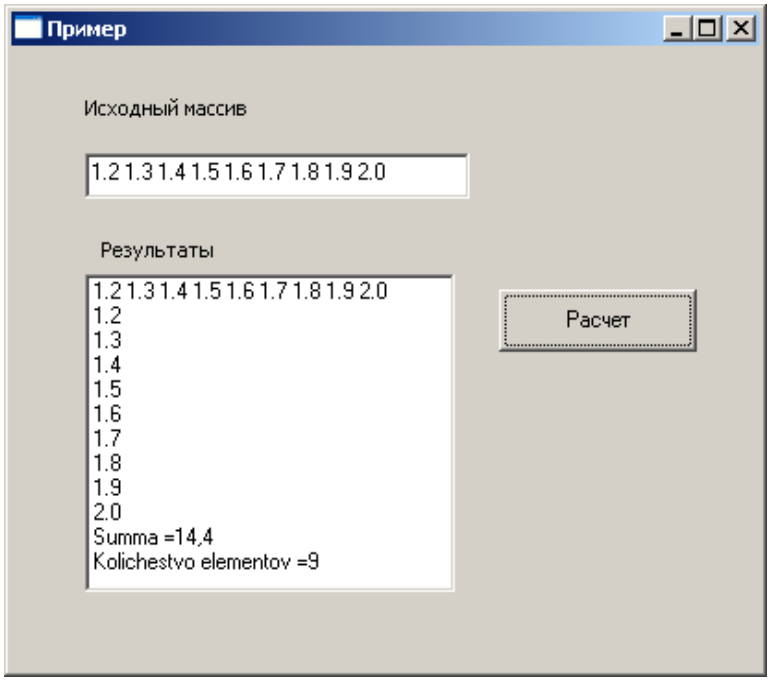

Рисунок 8.2: Результаты работы программы к задаче 8.1.

8.2 Работа с записями

Иногда возникает необходимость объединить в одном типе несколько разных типов данных. В Free Pascal для этого применяется структурный тип данных — `запись`. Запись состоит из фиксированного числа компонентов, называемого полями записи. В общем случае описание записи выглядит так:

```
type
 имя_записи = record
     поле1:  тип;
     поле2 : тип;
     ...
     полеN : тип;
```

```
end;
```

В программе переменная типа записи объявляется следующим образом:

```
var имя_переменной : имя_записи;
```

К каждому из полей записи можно получить доступ, используя составное имя. Для этого вначале пишется имя переменной, затем точка, затем имя поля.

ЗАДАЧА 8.2. Известны длины сторон треугольника *a*, *b* и *c*. Вычислить площадь треугольника *S*.

Подобная задача была рассмотрена во второй главе. В задаче 2.1 подробно описана математическая постановка и алгоритм решения. Рассмотрим поставленную задачу в контексте данной главы. Создадим запись — `Triangle` (треугольник) с тремя полями – `a`, `b` и `c` (стороны треугольника) и вычислим площадь треугольника по формуле Герона.

```
program Project1;
type
            //Запись – треугольник,
            //поля - стороны треугольника
 Triangle= record
     a, b, c: real;
 end;
//Объявление переменной типа Triangle.
var  x: Triangle;
     p, s:real;
begin
     write('a=');
     readln(x.a);   //Ввод значения в поле a.
     write('b=');
     readln(x.b);   //Ввод значения в поле b.
     write('c=');
     readln(x.c);   //Ввод значения в поле c.
     //Вычисление полупериметра
     p:=(x.a+x.b+x.c)/2;
//Проверка существования треугольника -
//подкоренное выражение для формулы Герона
//должно быть положительным.
     if p*(p-x.a)*(p-x.b)*(p-x.c)>0 then
```

```
    begin
        //Вычисление и вывод площади.
        s:=sqrt(p*(p-x.a)*(p-x.b)*(p-x.c));
        writeln('Площадь S=', s:7:2);
    end
    else
        writeln('Треугольник с заданными
                 сторонами не существует.');
end.
```

Элементами записей могут быть как простые, так и структурированные типы. Никаких ограничений на вложение одной структуры в другую не существует.

Например, создадим запись *Student*, которая будет состоять из полей: фамилия, имя, группа, оценки по пяти дисциплинам и адрес. В свою очередь поле «Адрес» также сделаем записью, состоящей из полей: город, улица, дом, квартира.

```
type
adress = record    //Запись «Адрес».
                          //поля:
        city,                 //город,
        street: string;       //улица,
        house,                //дом,
        apartment: integer;   //квартира.
end;
student = record       //Запись «Студент».
                          //поля:
        surname,              //фамилия,
        name: string;         //имя,
        group: string;        //группа,
                              //оценки,
        estimation: array [1..5] of integer;
        residence: adress;    //адрес.
end;
```

После объявления такой записи обращение к полям осуществляется так:

```
var  Ivanov: student;
     x: array [1..100] of student;
begin
```

```
    Ivanov.group:='Ф08';
    Ivanov.residense.city:='Киев';
    //у первого студента 5-я оценка =3
    x[1].estimation[5]:=3;

...
```

С использованием ключевого слова `With` к полям записи можно обращаться без указания префикса каждый раз с названием поля:

```
with <переменная> do <оператор>
```

Например:

```
with stud do
begin
 with residence do
 begin
     city:='Донецк';
     street :='Артема';
     house:=145;
     apartment:=31;
end;
 surname:='Иванов';
 name:='Андрей';
 birthday :='01.11.1990';
 group :='Ф07';
 estimation[1]:=3;  estimation[2]:=5;
 estimation[3]:=4;
 estimation[4]:=3;  estimation[5]:=5;
end;
```

ЗАДАЧА 8.3. Создать базу данных «Сведения о студентах». В программе предусмотреть расчет среднего балла студента, сортировку по алфавиту, вывод результатов в диалоговое окно и в текстовый файл.

Создадим новый проект. На форму (рис. 8.3) поместим двенадцать объектов типа `TEdit` для ввода исходных данных, объект типа `StringGrid` (`ColCount=8`, `RowCount=1`) для вывода результатов и три кнопки типа `Tbutton`:

- `Button1` – для чтения данных из полей ввода и их последующей очистки;

- `Button2` — для сортировки данных по алфавиту;
- `Button3` — для вывода результатов в таблицу `StringGrid1` и записи их в текстовый файл.

Текст программы с комментариями приведен далее.

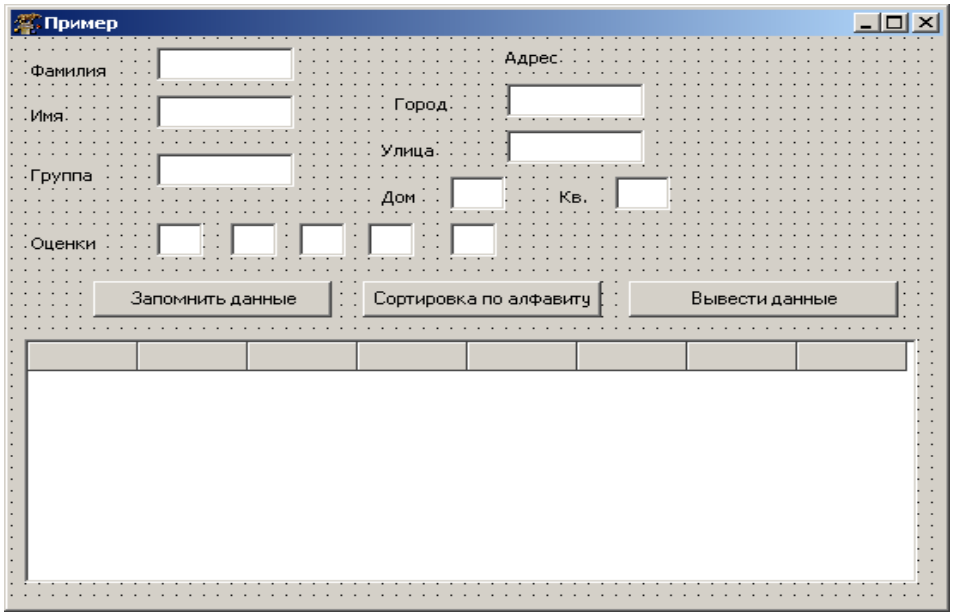

Рисунок 8.3: Пример формы для задачи 8.3

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes, SysUtils, LResources, Forms,
 Controls, Graphics, Dialogs, StdCtrls, Grids;
type
 { TForm1 }
 TForm1 = class(TForm)
     Button1: TButton;
     Button2: TButton;
     Button3: TButton;
     Edit1: TEdit;
     Edit10: TEdit;
     Edit11: TEdit;
     Edit12: TEdit;
     Edit2: TEdit;
     Edit3: TEdit;
     Edit4: TEdit;
     Edit5: TEdit;
```

```
    Edit6: TEdit;
    Edit7: TEdit;
    Edit8: TEdit;
    Edit9: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    Label9: TLabel;
    StringGrid1: TStringGrid;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
 private
    { private declarations }
 public
    { public declarations }
 end;
//Описание записи «Адрес студента».
 adress = record
    city, street : string;    //город, улица,
    house, apartment: integer;//дом, квартира.
 end;
//Описание записи «Сведения о студенте».
 student = record
    surname , name : string;  //фамилия, имя,
    group : string;           //группа,
                              //оценки,
    estimation: array [1..5] of integer;
    residence : adress;       //адрес,
    s_ball: real;             //средний балл.
 end;
var
```

```
 Form1: TForm1;
//Массив переменных типа student.
 x: array [0..100] of student;
//Локальная переменная для подсчета
//количества студентов.
i: integer;
implementation
{ TForm1 }
//Процедура инициализации формы.
procedure TForm1.FormCreate(Sender: TObject);
begin
    i:=0;//количество студентов равно 0.
end;
//Процедура работы кнопки «Запомнить данные».
procedure TForm1.Button1Click(Sender: TObject);
var sum, j:integer;
begin
 //Чтение данных из полей ввода.
    x[i].surname:= Edit1.Text;
    x[i].name:= Edit2.Text;
    x[i].group:= Edit3.Text;
    x[i].residence.city:=Edit5.Text ;
    x[i].residence.street:= Edit6.Text;
    x[i].residence.house:=
                        StrToInt(Edit7.Text);
    x[i].residence.apartment:=
                       StrToInt(Edit8.Text);
    x[i].estimation[1]:=strToInt(Edit4.Text);
    x[i].estimation[2]:=strToInt(Edit9.Text);
    x[i].estimation[3]:=strToInt(Edit10.Text);
    x[i].estimation[4]:=strToInt(Edit11.Text);
    x[i].estimation[5]:=
                       strToInt(Edit12.Text) ;
 //Вычисление среднего балла студента.
    sum:=0;
    for j:=1 to 5   do
        sum:=sum+x[i].estimation[j];
    x[i].s_ball:=sum/5; inc(i);
```

```
 //Очищение полей ввода для следующих данных.
     Edit1.Text:= '';  Edit2.Text:= '';
     Edit3.Text:= '';  Edit4.Text:= '';
     Edit5.Text:= '';  Edit6.Text:= '';
     Edit7.Text:= '';  Edit8.Text:= '';
     Edit9.Text:= '';  Edit10.Text:= '';
     Edit11.Text:= ''; Edit12.Text:= '';
end;
//Процедура работы кнопки «Вывести данные».
procedure TForm1.Button2Click(Sender: TObject);
var  f:textfile; j: integer; s: string;
begin
 //Количество строк в таблице StringGrid
//на один больше, чем количество студентов,
//одна строка для шапки таблицы.
     StringGrid1.RowCount:= i+1;
 //Вывод шапки таблицы.
     StringGrid1.Cells[1,0]:='Фамилия' ;
     StringGrid1.Cells[2,0]:='Имя' ;
     StringGrid1.Cells[3,0]:='Группа' ;
     StringGrid1.Cells[4,0]:='Город' ;
     StringGrid1.Cells[5,0]:='Улица' ;
     StringGrid1.Cells[6,0]:='Дом/кв.' ;
     StringGrid1.Cells[7,0]:='Средний балл' ;
//Вывод сведений о студентах в j-ю строку.
     for j:=1 to i do
     begin
         StringGrid1.Cells[1,j]:=
                           x[j-1].surname ;
         StringGrid1.Cells[2,j]:=x[j-1].name;
         StringGrid1.Cells[3,j]:=x[j-1].group;
         StringGrid1.Cells[4,j]:=
                        x[j-1].residence.city;
         StringGrid1.Cells[5,j]:=
                     x[j-1].residence.street;
         s:=inttostr(x[j-1].residence.house)+
     '/'+inttostr(x[j-1].residence.apartment);
         StringGrid1.Cells[6,j]:=s ;
```

```
            StringGrid1.Cells[7,j]:=
                        floattostr(x[j-1].s_ball);
        end
    //Вывод результатов в текстовый файл.
    assignfile(f,'g:\student.txt');
    rewrite(f);
    for j:=1 to i do
    begin
        writeln(f,x[j-1].surname:20,
                                x[j-1].name:15,
        x[j-1].residence.city:15,',',
        x[j-1].residence.street:15,
        x[j-1].residence.house:4,'/',
        x[j-1].residence.apartment,
        ' Sr_ball=',x[j-1].s_ball:4:1);
    end;
    closefile(f);
end;
   //Процедура работы кнопки
   //«Сортировать по алфавиту».
procedure TForm1.Button3Click(Sender: TObject);
var j, k :integer;
    temp: student; //Переменная для сортировки.
    f:textfile;
Begin
//Сортировка по полю  surname методом пузырька.
    for j:= 0 to  i-1 do
        for k:=j+1 to i-1 do
            if  x[j].surname  >  x[k].surname  then
    begin
        //Строки меняются местами.
        temp:=x[j]; x[j]:=x[k];x[k]:=temp;
    end;
end;
initialization
   {$I unit1.lrs}
end.
```

Результаты работы программы представлены на рис. 8.5 — 8.7.

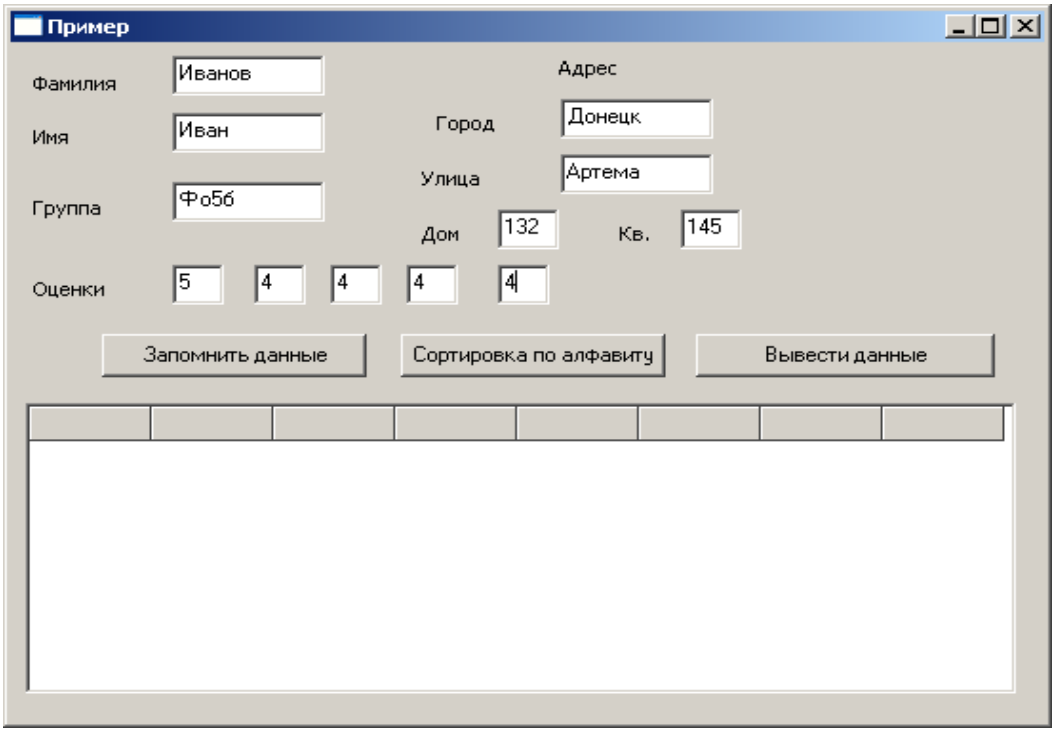

Рисунок 8.5: Окно формы ввода сведений о студенте

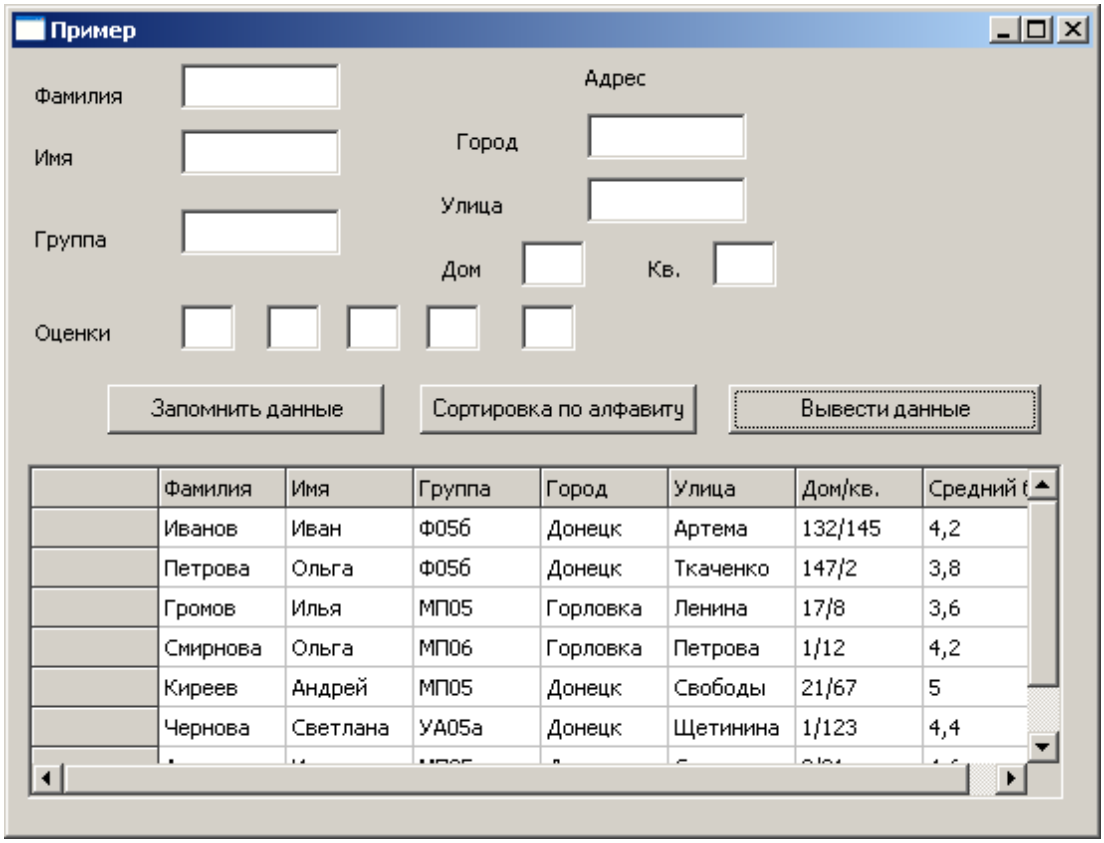

Рисунок 8.6: Окно формы вывода сведений о студентах

При запуске программы и вводе сведений окно формы выглядит так, как показано на рис 8.5. Когда запись введена, следует щелкнуть по кнопке «Запомнить данные», при этом поля ввода очищаются для ввода следующей записи. После щелчка по кнопке «Вывести данные» таблица заполняется введенными сведениями (рис.8.6). После щелчка по кнопке «Сортировать по алфавиту» нужно повторно щелкнуть по кнопке «Вывести данные», чтобы увидеть отсортированный список (рис.8.7).

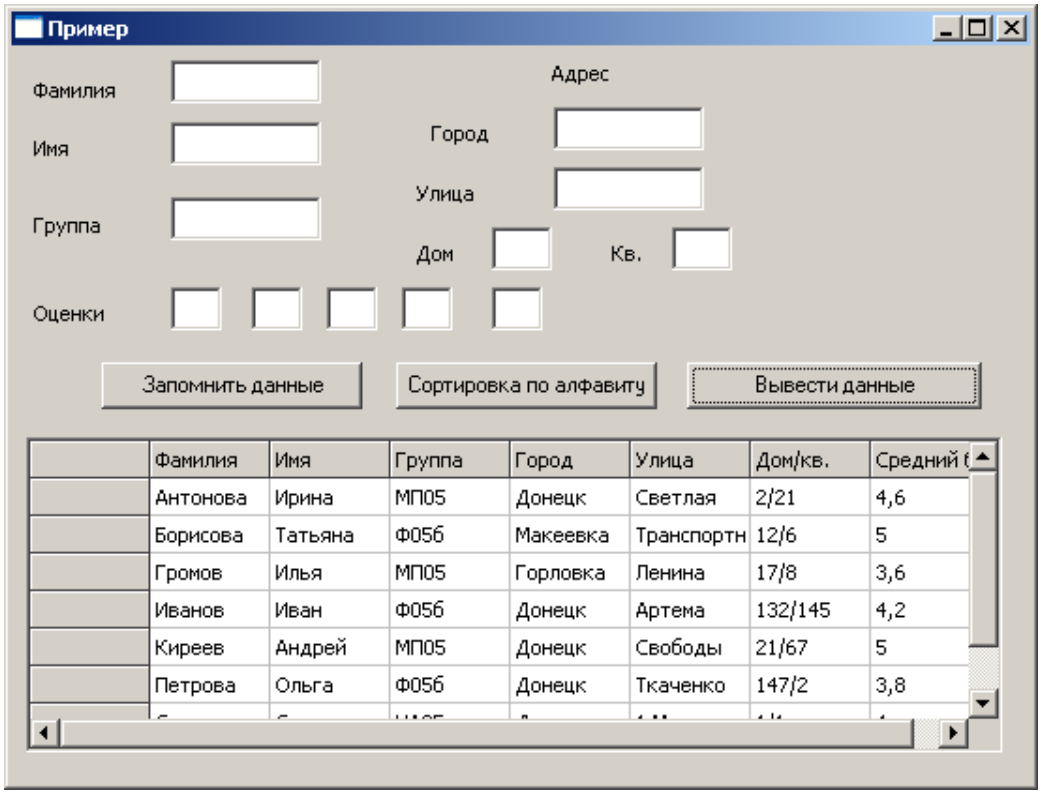

Рисунок 8.7: Окно формы после сортировки записей по полю «Фамилия»

8.3 Задачи для самостоятельного решения по теме «Строки»

Дана строка текста. Выполнить с ней следующие действия:

1. Посчитать количество запятых в строке.
2. Заменить в строке все цифры на пробел. Вывести количество замен.
3. Посчитать в строке количество цифр.
4. Удалить из строки все запятые.
5. Посчитать в строке количество слов.
6. Удалить из строки все слова, начинающиеся на букву «о».
7. После каждого пробела вставить символ *.

8. Найти в строке самое длинное слово.
9. Перед каждым пробелом вставить пробел и символ +.
10. Посчитать сумму всех чисел, которые встречаются в строке.
11. Посчитать в строке количество слов, начинающихся на «Ав».
12. Заменить в строке двойные пробелы на одинарный пробел. Вывести количество замен.
13. Вставить после каждого слова запятую.
14. После каждого слова вставить символ «;».
15. Посчитать в строке количество символов «:» и «;».
16. Удалить из строки все цифры.
17. Посчитать в строке количество слов, заканчивающихся символами «ая».
18. Найти в строке самое короткое слово.
19. Вставить после каждого слова, заканчивающегося на букву «о», слово «Ого».
20. Удалить из строки все слова, состоящие из пяти букв.
21. Найти в строке количество слов, начинающихся на букву «а» и заканчивающихся буквой «т».
22. Удалить из строки второе, третье и пятое слова.
23. Перед каждой цифрой вставить символ №.
24. Посчитать в строке количество гласных букв.
25. Удалить из строки все слова, начинающиеся и заканчивающиеся на «о».

8.4 Задачи для самостоятельного решения по теме «Записи»

1. Создать структуру с данными по таблице 8.1:

Таблица 8.1. Прирост населения в городах

Город	**Прирост населения, тыс. чел.**				
	1999	**2000**	**2001**	**2002**	**2003**
Макеевка	2,5	1,3	-0,2	-0,1	0,6
...					

Добавить и вычислить в структуре поле «Средний прирост». Определить количество городов с отрицательным приростом в 2003 году. Упорядочить записи по возрастанию среднего прироста.

2. Создать структуру с данными по таблице 8.1. Добавить и вычислить в структуре поле «Минимальный прирост». Определить ко-

личество городов с приростом в 2003 году более 2 тыс. чел. Выполнить сортировку записей по полю «Город». Названия городов упорядочить по алфавиту.

3. Создать структуру с данными по таблице 8.2:

Таблица 8.2. Сведения о товаре

Название	Фабрика	Цена	Дата выпуска	Количество
Паровозик	Игрушка	125,00	01.02.2007	
...				

Добавить и вычислить в структуре поле «Цена со скидкой», вводя процент скидки с формы. Найти общее количество игрушек с фабрики «Игрушка». Упорядочить записи по убыванию поля «Цена».

4. Создать структуру с данными по таблице 8.2. Добавить и вычислить в структуре поле «Сумма продажи». Найти количество названий игрушек, у которых цена меньше общей средней цены всех игрушек. Упорядочить записи по названию игрушек.

5. Создать структуру с данными по таблице 8.5:

Таблица 8.5. Сведения о школьнике

Фамилия	Имя	Дата рождения	Школа	Класс
Сергеев	Сергей	05.05.1994	112	9-А
...				

Добавить и вычислить в структуре поле «Возраст», вводя текущую дату с формы. Определить количество школьников с именем Сергей. Упорядочить записи по номеру школы.

6. Создать структуру с данными по таблице 8.5. Добавить и вычислить в структуре поле «Год обучения», убрав из названия класса букву. Найти количество учеников 9-х классов. Выполнить сортировку записей по полю «Фамилия». Фамилии упорядочить по алфавиту.

7. Создать структуру с данными по таблице 8.6:

Таблица 8.6. Сведения о продажах

Принтер	Количество, шт.			Цена, $
	Январь	Февраль	Март	
Samsung CLP-310	25	20	26	550
...				

Добавить и вычислить в структуре поле «Выручка». Найти среднюю цену принтеров. Упорядочить записи по возрастанию поля «Цена».

8. Создать структуру с данными по таблице 8.6. Добавить и вычислить в структуре поле «Среднее количество». Найти среднее количество принтеров в каждом месяце. Упорядочить записи по названию принтера.

9. Создать структуру с данными по таблице 8.6. Добавить и вычислить в структуре поле «Общее количество». Найти общее количество проданных принтеров в каждом месяце. Упорядочить записи по возрастанию поля «Общее количество».

10. Создать структуру с данными по таблице 8.6. Добавить и вычислить в структуре поле «Цена со скидкой», вводя процент скидки с формы. Найти количество принтеров с ценой более 500$. Упорядочить записи по убыванию цены.

11. Создать структуру с данными по таблице 8.7:

Таблица 8.7. Сведения о сотруднике

ФИО	**Дата рождения**	**Должность**	**Стаж**	**Оклад**
Сергеев С. И.	12.03.1966	Менеджер	2	1250
...				

Добавить и вычислить в структуре поле «Премия», рассчитав ее по следующему принципу: 20% от оклада, если стаж более 10 лет, в противном случае 10%. Найти количество сотрудников со стажем более 10 лет. Упорядочить записи по должности.

12. Создать структуру с данными по таблице 8.8. Добавить и вычислить в структуре поле «Возраст», текущую дату вводить с формы. Найти средний оклад всех сотрудников. Упорядочить записи по ФИО.

13. Создать структуру с данными по таблице 8.8. Добавить и вычислить в структуре поле «Возраст», текущую дату вводить с формы. Определить количество молодых специалистов (моложе 25 лет). Упорядочить записи по возрастанию оклада.

14. Создать структуру с данными по таблице 8.8:

Таблица 8.8. Сведения о продажах путевок

Место отдыха	**Количество, шт.**			**Цена, $**
	Июль	**Август**	**Сентябрь**	
Геленджик	255	203	198	510
...				

Добавить и вычислить в структуре поле «Среднее количество». Найти общее количество путевок в каждом месяце. Упорядочить за-

писи по месту отдыха.

15. Создать структуру с данными по таблице 8.8. Добавить и вычислить в структуре поле «Доход от путевок». Найти среднюю цену путевки. Упорядочить записи по возрастанию цены.

16. Создать структуру с данными по таблице 8.9:

Таблица 8.9. Сведения о сотруднике

ФИО	**Дата рождения**	**Должность**	**Пол**	**Оклад**
Сергеев С. И.	12.03.1966	Менеджер	Муж.	1250
...				

Добавить и вычислить в структуре поле «Зарплата», рассчитав ее по следующему принципу: к окладу добавить премию в размере 15% от оклада. Упорядочить записи по ФИО.

17. Создать структуру с данными по таблице 8.9. Добавить и вычислить в структуре поле «Возраст», текущую дату вводить с формы. Определить количество мужчин и женщин. Упорядочить записи по должности.

18. Создать структуру с данными по таблице 8.10:

Таблица 8.10. Сведения о научных сотрудниках

Фамилия	**Инициалы**	**Ученая степень**	**Год рождения**	**Количество статей**
Сергеев	С. А.	Доцент	1971	25
...				

Добавить и вычислить в структуре поле «Активность» по следующему принципу: если количество статей более 10, то в поле записать пробел, в противном случае – фразу «Работать лучше». Упорядочить записи по фамилии.

19. Создать структуру с данными по таблице 8.10. Удалить сотрудника с фамилией, которая вводится с формы. Определить количество доцентов. Упорядочить записи по должности.

20. Создать структуру с данными по таблице 8.11:

Таблица 8.11. Сведения о тираже книг

Название	**Автор**	**Издательство**	**Год издания**	**Цена, $**	**Тираж**
Вий	Гоголь Н. В.	Правда	1971	6,5	15000
...					

Добавить и вычислить в структуре поле «Стоимость тиража». Найти общий тираж книг 2005 года. Упорядочить записи по автору.

21. Создать структуру с данными по таблице 8.11. Удалить все записи книг тиража 2000 года. Найти среднюю цену книг типографии «Правда». Упорядочить записи по году издания.

22. Создать структуру с данными по таблице 8.12:

Таблица 8.12. Сведения о телефонных звонках

ФИО абонента	**Номер**	**Дата звонка**	**Город**	**Стоимость 1 минуты разговора**	**Количество минут**
Моль Р. Ю.	956-25-78	12.05.2003	Казань	3,65	2
...					

Добавить и вычислить в структуре поле «Стоимость звонка». Найти общую стоимость звонков в город, вводимый по запросу. Упорядочить записи по ФИО абонента.

23. Создать структуру с данными по таблице 8.12. Удалить все записи звонков с номерами, начинающимися с цифры 3. Упорядочить записи по названию города.

24. Создать структуру с данными по таблице 8.14:

Таблица 8.14. Сведения о приборах

Название прибора	**Шифр прибора**	**Дата выпуска**	**Количество**	**Гарантийный срок, мес.**
Микроскоп	М12-08	12.06.2006	200	24
...				

Добавить и вычислить в структуре поле «Гарантийное обслуживание» по следующему принципу: записать фразу «1 год», если гарантийный срок более 3 лет, в противном случае – фразу «нет обслуживания». Найти общее количество всех приборов. Упорядочить записи по названию прибора.

25. Создать структуру с данными по таблице 8.14. Удалить все записи с гарантийным сроком менее 6 месяцев. Упорядочить записи по дате выпуска.

9 Объектно-ориентированное программирование

Эта глава посвящена изучению объектно-ориентированного программирования (ООП). ООП – это методика разработки программы, в основе которой лежит понятие объекта как некоторой структуры, описывающей объект реального мира, его поведение и взаимодействие с другими объектами.

9.1 Основные понятия

Основой объектно-ориентированного программирования является объект. Он состоит из трех основных частей:

1. *Имя* (например, автомобиль);

2. *Состояние* или *переменные состояния* (например, марка автомобиля, цвет, масса, число мест и т. д.);

3. *Методы*, или операции, которые выполняют некоторые действия над объектами и определяют, как объект взаимодействует с окружающим миром.

Для работы с объектами во FreePascal введено понятие класса. *Класс* – сложная структура, включающая в себя описание данных, процедуры и функции, которые могут быть выполнены над объектом.

Классом называется составной тип данных, членами (элементами) которого являются функции и переменные (поля). В основу понятия «класс» положен тот факт, что "*над объектами можно совершать различные операции"*. Свойства объектов описываются с помощью переменных (полей) классов, а действия над объектами описываются с помощью подпрограмм, которые называются «методами класса». Объекты называются *экземплярами* класса.

Объектно-ориентированное программирование (ООП) – представляет собой технологию разработки программ с использованием объектов.

В объектно-ориентированных языках есть три основных понятия: инкапсуляция, наследование и полиморфизм. *Инкапсуляцией* называется объединение в классе данных и подпрограмм для их обработки. *Наследование* – это когда любой класс может быть порождён другим классом. Порождённый класс (наследник) автоматически наследует все поля, методы, свойства и события. *Полиморфизм* позволяет ис-

пользовать одинаковые имена для методов, входящих в различные классы.

Для объявления класса используется конструкция:

```
type
<название класса> = class(<имя класса родителя>)
<методы и переменные класса>
private
<поля и методы, доступные только
                                в пределах модуля>
protected
<поля и методы, доступные только
                                в классах-потомках>
public
   <поля и методы, доступные из других модулей>
published
   <поля и методы, видимые в инспекторе объектов>
end;
```

В качестве *имени класса* можно использовать любой допустимый в FreePascal идентификатор. *Имя класса родителя* — имя класса, наследником которого является данный класс, это необязательный параметр, если он не указывается, то это означает, что данный класс является наследником общего из предопределенного класса `TObject`.

Структуры отличаются от классов тем, что поля структуры доступны всегда. При использовании классов могут быть члены, доступные везде — публичные (описатель *public*), и приватные (описатель *private*), доступ к которым возможен только с помощью публичных методов. Это также относится и к методам класса.

Поля, свойства и методы секции *public* не имеют ограничений на видимость. Они доступны из других функций и методов объектов как в данном модуле, так и во всех прочих, ссылающихся на него. При обращении к публичным полям вне класса используется оператор . (точка).

Поля, свойства и методы, находящиеся в секции *private*, доступны только в методах класса и в функциях, содержащихся в том же модуле, что и описываемый класс. Это позволяет полностью скрыть детали внутренней реализации класса. Вызов приватных методов осуществляется из публичных.

Публикуемый (*published*) — это раздел, содержащий свойства, ко-

торые пользователь может устанавливать на этапе проектирования и они доступны в любом модуле.

Защищенный (*protected*) — это раздел, содержащий поля и методы, которые доступны внутри класса, а также любым его классам-потомкам, в том числе и в других модулях.

При программировании с использованием классов программист должен решить, какие члены и методы должны быть объявлены публичными, а какие приватными. Общим принципом является следующее: *"Чем меньше публичных данных о классе используется в программе, тем лучше"*. Уменьшение количества публичных членов и методов позволит минимизировать количество ошибок.

Поля могут быть любого типа, в том числе и классами. Объявление полей осуществляется так же, как и объявление обычных переменных:

```
поле1: тип_данных;
поле2: тип_данных;
...
```

Методы в классе объявляются так же, как и обычные подпрограммы:

```
function метод1 (список параметров): тип результата;
procedure метод2 (список параметров);
```

Описание процедур и функций, реализующих методы, помещается после слова `implementation` того модуля, где объявлен класс, и выглядит так:

```
function имя_класса.метод1(список параметров):
                                        тип результата;
begin
     тело функции;
end;
procedure имя_класса.метод2(список параметров);
begin
     тело процедуры;
end;
```

Объявление переменной типа `class` называется *созданием (инициализацией)* объекта (экземпляра класса). Экземпляр класса объявляется в блоке описания переменных:

```
var имя_переменной : имя_класса;
```

После описания переменной в программе можно обращаться к полям и методам класса аналогично обращению к полям структуры, используя оператор «.». Например:

```
имя_переменной.поле1:=выражение;
имя_переменной.метод1(список параметров);
...
```

Также можно использовать оператор With:

```
With имя_переменной do
begin
 поле1:=выражение;
 метод1(список параметров);
 ...
end;
```

В FreePascal имеется большое количество классов, с помощью которых описывается форма приложения и ее компоненты (кнопки, поля, флажки и т.п.). В процессе конструирования формы в текст программы автоматически добавляются программные объекты. Например, при добавлении на форму компонента формируется описание класса для этого компонента, а при создании подпрограмм обработки событий в описание класса добавляется объявление методов. Рассмотрим это на примере проекта с формой, на которой есть кнопка Button1.

```
unit Unit1;
interface
uses
Classes, SysUtils, LResources, Forms, Controls,
Graphics, Dialogs, StdCtrls;
type
{ TForm1 }
//объявление класса формы TForm1
TForm1 = class(TForm)
//объявление компонента кнопки Button1
Button1: TButton;
//объявление метода обработки события —
//щелчка по кнопке Button1
procedure Button1Click(Sender: TObject);
private
{ private declarations }
```

```
public
{ public declarations }
end;
var
   //описание переменной класса формы TForm1
    Form1: TForm1;
implementation
{ TForm1 }
//описание метода обработки события — щелчка по
кнопке Button1
procedure TForm1.Button1Click(Sender: TObject);
begin
    //Текст процедуры обработки события
end;
initialization
{$I unit1.lrs}
end.
```

Во Free Pascal класс (объект) –это *динамическая структура*. В отличие от статической она содержит не сами данные, а ссылку на них. Поэтому программист должен сам позаботиться о выделении памяти для этих данных.

Конструктор — это специальный метод, создающий и инициализирующий объект. Объявление конструктора имеет вид:

```
constructor Create;
```

Описывают конструктор так же, как и другие методы, после ключевого слова `implemention` того модуля, в котором объявлен класс.

```
constructor имя_класса.Create;
begin
 поле1:=выражение1;
 поле2:=выражение2;
 ...
inherited Create;
end;
```

В результате работы конструктора инициализируются все поля класса, при этом порядковым типам в качестве начальных значений задается 0, а строки задаются пустыми.

Деструктор – это специальный метод, уничтожающий объект и освобождающий занимаемую им память. Объявляется деструктор

следующим образом:

```
destructor Destroy;
```

Если в программе какой-либо объект больше не используется, то оператор

```
имя_переменной_типа_класс.free;
```

с помощью метода `free` вызывает деструктор и освобождает память, занимаемую полями объекта `имя_переменной_типа_класс`.

Рассмотрим все описанное на примере класса — комплексное число[78]. Назовем класс – `Tcomplex`, в классе будут члены класса: `x` – действительная часть комплексного числа, `y` – мнимая часть комплексного числа. Также в классе будут методы:

- конструктор `Create`, который будет записывать в действительную и мнимую части значение 0;
- `Modul()` – функция вычисления модуля комплексного числа;
- `Argument()` – функция вычисления аргумента комплексного числа;
- `ComplexToStr()` - функция, представляющая комплексное число в виде строки для вывода.

Создадим новый проект, на форму поместим кнопку `Button1`, два поля `Edit1` и `Edit2` для ввода действительной и мнимой частей, для вывода результатов разместим компонент `Memo1`. При щелчке по кнопке будет создаваться экземпляр класса «Комплексное число», затем будут вычисляться его модуль и аргумент. В компонент `Memo1` выведем результаты: число в алгебраической форме, его аргумент и модуль. Ниже приведем текст модуля с комментариями, который демонстрирует работу с этим классом. Результат работы программы приведен на рис. 9.1.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes,    SysUtils,    LResources,    Forms,
Controls, Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
```

78 Про комплексные числа можно прочитать на странице http://kvant.mccme.ru./1982/03/komplesnye_chisla.htm

```
 TForm1 = class(TForm)
 Button1: TButton;
 Edit1: TEdit;
 Edit2: TEdit;
 Label1: TLabel;
 Label2: TLabel;
 Label3: TLabel;
 Memo1: TMemo;
 procedure Button1Click(Sender: TObject);
private
 { private declarations }
public
 { public declarations }
end;
type
//Описание класса – комплексное число.
 TComplex = class
 private
     x: real;   //Действительная часть.
     y: real;   //Мнимая часть
 public
     constructor Create;   //Конструктор.
     //Метод вычисления модуля.
     function Modul():real;
     //Метод вычисления аргумента.
     function Argument():real;
     //Метод записи комплексного
     //числа в виде строки.
     function ComplexToStr(): String;
 end;
var
 Form1: TForm1;
//Объявление переменной
//типа класс «комплексное число».
 chislo: TComplex;
implementation
//описание конструктора
constructor TComplex.Create;
```

```
begin
 x:=0; y:=0;
 inherited Create;
end;
//Описание метода вычисления
//модуля комплексного числа.
function TComplex.Modul(): real;
begin
 modul:=sqrt(x*x+y*y);
end;
//Описание метода вычисления
//аргумента комплексного числа.
function TComplex.Argument(): real;
begin
 argument:=arctan(y/x)*180/pi;
end;
//Описание метода записи комплексного
//числа в виде строки.
function TComplex.ComplexToStr(): String;
begin
 if y>=0 then
     ComplexToStr:=FloatToStrF(x,ffFixed,5,2)+
          '+' + FloatTostrF(y,ffFixed,5,2)+ 'i'
 else
ComplexToStr:=FloatTostrF(x,ffFixed,5,2)+
             FloatTostrF(y,ffFixed,5,2)+ 'i'
end;
//Обработчик кнопки: создание экземпляра класса
//«Комплексное число», вычисление его модуля и
//аргумента, вывод числа в алгебраической
//форме, его аргумента и модуля.
procedure TForm1.Button1Click(Sender: TObject);
Var Str1: String;
begin
//Создание объекта (экземпляра класса)
//типа «комплексное число».
 chislo:=TComplex.Create;
//Ввод действительной и мнимой частей
```

```
//комплексного числа.
 chislo.x:=StrToFloat(Edit1.Text);
 chislo.y:=StrToFloat(Edit2.Text);
 Str1:='Kompleksnoe chislo '+
                    chislo.ComplexToStr();
//Вывод на форму в поле Memo
        //построчно комплексного числа,
 Memo1.Lines.Add(Str1) ;
        //его модуля
 Str1:='Modul chisla '+
     FloatToStrF(chislo.Modul(),ffFixed, 5, 2);
 Memo1.Lines.Add(Str1) ;
        //и аргумента.
 Str1:='Argument chisla '+
 FloatToStrF(chislo.Argument(),ffFixed, 5, 2);
 Memo1.Lines.Add(Str1) ;
//Уничтожение объекта.
 chislo.Free;
end;
initialization
  {$I unit1.lrs}
end.
```

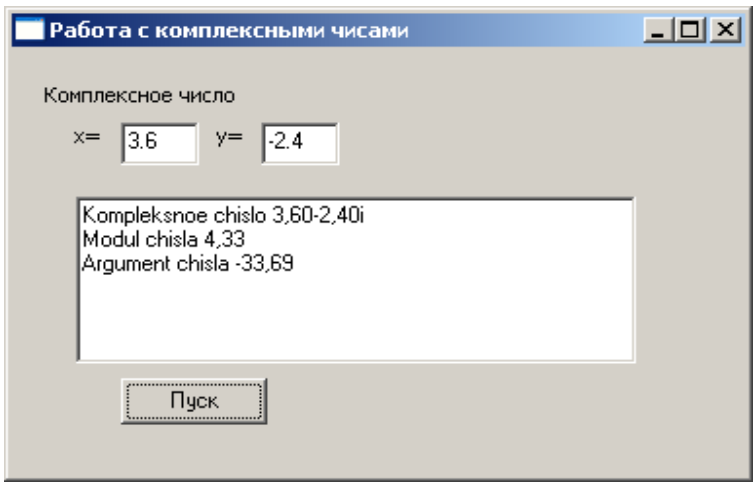

Рисунок 9.1: Результаты программы работы с классом «Комплексное число»

В этом примере был написан конструктор для класса «комплексное число» без параметров. В Free Pascal можно написать конструктор с параметрами, который принимает входные значения и инициализирует поля класса этими значениями. Перепишем предыдущий

пример следующим образом. Действительную и мнимую части будем считывать из полей ввода формы и передавать в конструктор для инициализации объекта типа комплексное число. Листинг программы приведен ниже.

```
   unit Unit1;
   {$mode objfpc}{$H+}
   interface
   uses
    Classes,    SysUtils,    LResources,    Forms,
Controls, Graphics, Dialogs, StdCtrls;
   type
    { TForm1 }
    TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Memo1: TMemo;
    procedure Button1Click(Sender: TObject);
    procedure Memo1Change(Sender: TObject);
   private
    { private declarations }
   public
    { public declarations }
   end;
   type
   TComplex = class
    private x, y: real;
    public
   //Объявление конструктора.
        constructor Create(a,b:real);
        function Modul():real;
        function Argument():real;
        function ComplexToStr(): String;
   end;
   var
```

```
 Form1: TForm1;
 chislo: TComplex;
implementation
//Конструктор, который получает
//в качестве входных параметров
//два вещественных числа и записывает
//их в действительную и мнимую части
//комплексного числа.
constructor TComplex.Create(a,b:real);
begin
     x:=a;   y:=b;
     inherited Create;
end;
function TComplex.Modul(): real;
begin
 modul:=sqrt(x*x+y*y);
end;
function TComplex.Argument(): real;
begin
 argument:=arctan(y/x)*180/pi;
end;
function TComplex.ComplexToStr(): String;
begin
 if y>=0 then
     ComplexToStr:=FloatToStrF(x,ffFixed,5,2)+
         '+' + FloatTostrF(y,ffFixed,5,2)+ 'i'
 else
     ComplexToStr:=FloatTostrF(x,ffFixed,5,2)+
                 FloatTostrF(y,ffFixed,5,2)+ 'i'
end;
procedure TForm1.Button1Click(Sender: TObject);
Var  Str1 : String;
     x1, x2 : real;
begin
 x1:=StrToFloat(Edit1.Text);
 x2:=StrToFloat(Edit2.Text);
 chislo:=TComplex.Create(x1,x2);
 chislo.x:=StrToFloat(Edit1.Text);
```

```
 chislo.y:=StrToFloat(Edit2.Text);
 Str1:='Kompleksnoe chislo '+
                        chislo.ComplexToStr() ;
 Memo1.Lines.Add(Str1) ;
 Str1:='Modul chisla '+
     FloatToStrF(chislo.Modul(),ffFixed, 5, 2);
 Memo1.Lines.Add(Str1) ;
 Str1:='Argument chisla '+
 FloatToStrF(chislo.Argument(),ffFixed, 5, 2);
 Memo1.Lines.Add(Str1) ;
 chislo.Free;
end;
initialization
 {$I unit1.lrs}
end.
```

9.2 Инкапсуляция

Инкапсуляция — один из важнейших механизмов объектно-ориентированного программирования (наряду с наследованием и полиформизмом). Класс представляет собой единство трех сущностей – полей, свойств и методов, что и является инкапсуляцией. Инкапсуляция позволяет создавать класс как нечто целостное, имеющее определённую функциональность. Например, класс `TForm` содержит в себе (инкапсулирует) все необходимое, чтобы создать диалоговое окно.

Основная идея инкапсуляции – защитить поля от несанкционированного доступа. Поэтому целесообразно поля объявлять в разделе `private`. Прямой доступ к полям объекта: чтение и обновление их содержимого должно производиться посредством вызова соответствующих методов. В FreePascal для этого служат свойства класса.

Свойства - это специальный механизм классов, регулирующий доступ к полям. Свойства объявляются с помощью зарезервированных слов `property`, `read` и `write`. Обычно свойство связано с некоторым полем и указывает те методы класса, которые должны использоваться при записи в это поле или при чтении из него. Синтаксис объявления свойств следующий:

```
propety имя_1: тип read имя_чтения write имя_2
```

Зарезервированное слово `read` описывает метод чтения свойств объекта, а слово `write` описывает метод записи свойств объекта. `Имя_1` и `имя_2` – соответственно имена методов, обеспечивающих чтение или запись свойства.

Если необходимо, чтобы свойство было доступно только для чтения или только для записи, следует опустить соответственно часть `write` или `read`.

Рассмотрим следующий пример. Создадим класс – многоугольник, имя класса `TPolygon`. Полями класса будут:

- `K` – количество сторон многоугольника;
- `p` – массив, в котором будут храниться длины сторон многоугольника.

Методами класса будут:

- конструктор `Create`, обнуляющий элементы массива `p`;
- `Perimetr()` - функция вычисления периметра фигуры;
- `Show()` - функция формирования сведений о фигуре (количество сторон и периметр);
- `Set_Input()` - функция проверки исходных данных.

Расположим на форме кнопку и метку. При щелчке по кнопке появляется окно ввода количества сторон многоугольника. Если количество сторон введено корректно, то инициализируется объект «Многоугольник» с количеством сторон, равным введенному, в противном случае количество сторон многоугольника принимается равным по умолчанию 50. После этого вычисляется периметр фигуры и результаты выводятся на форму в метке `Label1`.

Ниже приведен листинг программы с комментариями, результаты работы программы можно увидеть на рис. 9.2.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes,    SysUtils,    LResources,    Forms,
Controls, Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
 TForm1 = class(TForm)
 Button1: TButton;
```

```
 Label1: TLabel;
 procedure Button1Click(Sender: TObject);
 private
     { private declarations }
 public
     { public declarations }
end;
type
//Объявление класса многоугольник Tpolygon.
TPolygon = class
//Закрытые поля.
Private
 K : integer;
 p : array of real;
//Открытые методы.
Public
 constructor Create;       //Конструктор.
 //Метод вычисления периметра.
 function Perimetr():real;
//Метод формирования сведений.
 function Show():String;
Protected      //Защищенные методы.
 //Процедура проверки данных.
 procedure Set_Input(m:integer);
Published
//Объявление свойства n.
//Свойство n оперирует полем K.
//В описании свойства после слова read
//стоит имя поля — K. Это значит, что
//функция чтения отсутствует и пользователь
//может читать непосредственно значение поля.
//Ссылка на функцию  Set_Input после
//зарезервированного слова  write означает,
//что с помощью этой функции в поле K будут
//записываться новые значения.
Property n: integer read K write Set_Input;
end;
var
```

```
 Form1: TForm1;
//Объявление переменной типа
//класс многоугольник.
 Figure: TPolygon;
implementation
//Описание конструктора.
constructor TPolygon.Create;
var i:integer;
begin
 K:=50;//Присваивание начальных значений полям.
//Выделение памяти под массив p.
 SetLength(p,K);
 for i:=0 to K-1 do p[i]:=0;
 inherited Create;
end;
//Функция вычисления периметра.
function TPolygon.Perimetr():real;
var Sum:real; i:integer;
begin
 Sum:=0;
 for i:=0 to K-1 do Sum:=Sum+p[i];
 Perimetr:=Sum;
end;
//Метод формирования сведений о фигуре.
function TPolygon.Show():String;
begin
 Show:='Многоугольник с количеством сторон '+
            IntToStr(K)+chr(13)+'Периметр = '+
                        FloatToStr(Perimetr())
end;
//Метод записи данных в поле K.
procedure TPolygon.Set_Input(m:integer);
begin
//Если введенное значение положительное число,
//то записать его в поле K,
//иначе вернуться к начальному значению.
 if m>1 then K:=m  else K:=50;
end;
```

```
{TForm1 }
//Обработка события.
procedure TForm1.Button1Click(Sender: TObject);
var  i, m:integer;
 s:string;
begin
//Ввод количества сторон многоугольника.
s:=InputBox('Ввод','Введите количество сторон
                          многоугольника','6');
 Val(s,m);
 //Инициализация объекта.
 Figure:=TPolygon.Create;
 with Figure do
 begin
//Метод проверки исходных данных.
     Set_Input(m);
//Формирование массива случайных чисел.
     for i:=0 to K-1 do p[i]:=random(50);
//Обращение к методу вычисления периметра.
         s:=Show();
 end;
 //Вывод результатов в окно формы.
 Label1.Caption:= s;
end;
initialization
{$I unit1.lrs}
end.
```

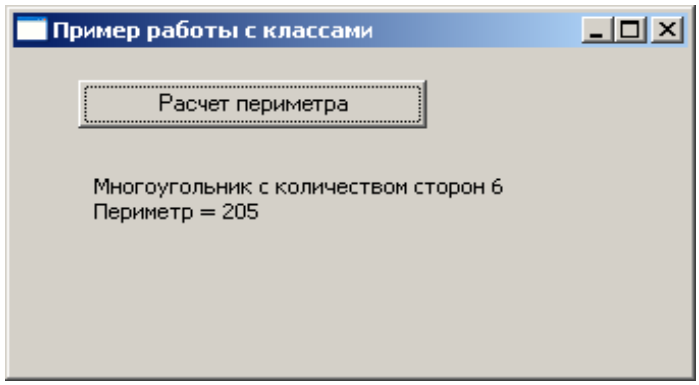

Рисунок 9.2: Результат работы программы с классом многоугольник

9.3 Наследование и полиформизм

Второй основополагающей составляющей объектно-ориентированного программирования является *наследование*. Смысл наследования заключается в следующем: если нужно создать новый класс, лишь немного отличающийся от старого, то нет необходимости в переписывании заново уже существующих полей и методов. В этом случае объявляется новый класс, который является наследником уже имеющегося, и добавляются к нему новые поля, методы и свойства. При создании новый класс является *наследником* членов ранее определенного *базового* класса (родителя). Класс-наследник является *производным* от базового, причем он сам может выступать в качестве базового класса для вновь создаваемых классов.

В Object Pascal все классы являются потомками класса `TObject`. Поэтому если вы создаете дочерний класс прямо от класса `TObject`, то в определении его можно не упоминать.

Производный класс наследует от класса-предка поля и методы; если имеет место совпадение имен методов, то говорят, что они перегружаются. В зависимости от того, какие действия происходят при вызове, методы делятся на следующие группы:

- статические методы;
- виртуальные методы;
- динамические методы.

По умолчанию все методы *статические*. Эти методы полностью перегружаются в классах-потомках при их переопределении. При этом можно полностью изменить объявление метода. Если обращаться к такому методу у объекта базового класса, то будет работать метод класса-родителя. Если обращаться к методу у производного класса, то будет работать новый метод.

Виртуальные и *динамические* методы имеют в базовом и производном классах те же имена и типы. В классах-наследниках эти методы перегружены. В зависимости от того, с каким классом работают, соответственно и вызывается метод этого класса.

Основная разница между виртуальными и динамическими методами — в способе их вызова. Информация о виртуальных методах хранится в таблице виртуальных методов VMT. В VMT хранятся виртуальные методы данного класса и всех его предков. При создании потомка класса вся VMT предка переносится в потомок и там к ней

добавляются новые методы. Поиск нужного метода занимает мало времени, так как класс имеет всю информацию о своих виртуальных методах. Динамические методы не дублируются в таблице динамических методов DMT потомка. DMT класса содержит только методы, объявленные в этом классе. При вызове динамического метода сначала осуществляется поиск в DMT данного класса, и если метод не найден – то в DMT предка класса и т.д. Таким образом использование виртуальных методов требует большего расхода памяти из-за необходимости хранения массивных VMT всех классов, зато они вызываются быстрее.

Изменяя алгоритм того или иного метода в производных классах, программист может придавать этим потомкам отсутствующие у родителя специфические свойства. Для изменения метода необходимо перегрузить его в потомке, т. е. объявить в наследнике одноименный метод и реализовать в нем нужные действия. В результате в объекте-родителе и объекте-потомке будут действовать два одноименных метода, имеющие разную алгоритмическую основу. Это называется *полиморфизмом* объектов.

Виртуальные и динамические методы объявляются так же, как и статические, только в конце описания метода добавляются служебные слова `virtual` или `dynamic`:

```
type
 имя_родителя =class
     ...
     метод;  virtual;
     ...
end;
```

Чтобы перегрузить в классе-наследнике виртуальный метод, нужно после его объявления написать ключевое слово `override`:

```
type
 имя_наследника=class (имя_родителя)
     ...
     метод;  override;
     ...
end;
```

Рассмотрим наследование в классах на следующем примере. Создадим базовый класс `Ttriangle` (*треугольник*) с полями класса –

координаты вершин треугольника. В классе будут следующие методы:

- `Proverka()` – метод проверки существования треугольника (если 3 точки лежат на одной прямой, то треугольник не существует);
- `Perimetr()` – метод вычисления периметра треугольника;
- `Square()` – - метод вычисления площади;
- Методы вычисления длин сторон `a(), b(), c()`;
- `Set_Tr()` – метод получения координат;
- `Show()` – метод формирования сведений о треугольнике.

На основе этого класса создадим производный класс `R_TTriangle` (равносторонний треугольник), который наследует все поля и методы базового класса, но методы проверки и формирования сведений о фигуре перегружаются по другому алгоритму.

На форме поместим метку, кнопку и по 6 компонентов типа `TEdit` для ввода координат вершин для двух треугольников. После щелчка по кнопке создаются два объекта типа «Треугольник» и «Равносторонний треугольник», вычисляются периметр и площадь каждого треугольника, и результаты выводятся ниже в компоненты `Label1 Label2`. Далее приведен листинг программы с комментариями.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes, SysUtils, LResources, Forms,
          Controls, Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
TForm1 = class(TForm)
 Button1: Tbutton;  //кнопка Расчет
 //массивы x1, y1 – координаты вершин
//1-го треугольника частный случай)
 //массивы x1, y1 – координаты вершин
//2-го треугольника (равностороннего)
 Edit1: Tedit;  //для ввода координаты x1[1]
 Edit10: TEdit; //для ввода координаты y2[2]
 Edit11: TEdit; //для ввода координаты x2[3]
```

```
 Edit12: TEdit; //для ввода координаты y2[3]
 Edit2: TEdit;  //для ввода координаты y1[1]
 Edit3: TEdit;  //для ввода координаты x2[2]
 Edit4: TEdit;  //для ввода координаты y1[2]
 Edit5: TEdit;  //для ввода координаты x1[3]
 Edit6: TEdit;  //для ввода координаты y1[3]
 Edit7: TEdit;  //для ввода координаты x2[1]
 Edit8: TEdit;  //для ввода координаты y2[1]
 Edit9: TEdit;  //для ввода координаты x2[2]
 Label1: TLabel;
 Label10: TLabel;
 Label11: TLabel;
 Label12: TLabel;
 Label13: TLabel;
 Label14: TLabel;
 Label15: TLabel;
 Label16: TLabel;
 Label2: TLabel;
 Label3: TLabel;
 Label4: TLabel;
 Label5: TLabel;
 Label6: TLabel;
 Label7: TLabel;
 Label8: TLabel;
 Label9: TLabel;
 procedure Button1Click(Sender: TObject);
private
 { private declarations }
public
 { public declarations }
end;
//Объявление базового класса «треугольник».
type
 TTriangle=class
 Private
//Массивы, в которых хранятся координаты
//вершин треугольника.
    x, y:array[0..2] of real;
```

```
Public
constructor Create;//конструктор
//Метод получения исходных данных.
procedure  Set_Tr(a,b:array of real);
//Методы вычисления сторон треугольника.
function a():real;
function b():real;
function c():real;
//Виртуальный метод проверки существования
//треугольника, который будет перегружен
//в производном классе.
function Proverka():boolean;      virtual;
//Метод вычисления периметра.
function Perimetr():real;
//Метод вычисления площади.
function Square():real;
//Виртуальный метод формирования
//сведений о треугольнике.
function Show():string;        virtual;
end;
//Объявление производного
//класса «равносторонний треугольник».
type
R_TTriangle=class(TTriangle)
public
//Перегружаемые методы проверки, что
//треугольник является равносторонним,
//и формирования сведений о треугольнике.
function Proverka():boolean; override;
function Show():string;        override;
end;
var
 Form1: TForm1;
//Объявление переменной типа
//класс «треугольник».
 Figura1: Ttriangle;
//Объявление переменной типа
//класс «равносторонний треугольник».
```

```
 Figura2: R_TTriangle;
implementation
//Конструктор, который обнуляет
//массивы координат.
constructor TTriangle.Create;
var i:integer;
begin
 for i:=0 to 2 do
 begin
     x[i]:=0;  y[i]:=0;
 end;
end;
//Метод получения координат вершин.
procedure  TTriangle.Set_Tr(a,b:array of real);
var i:integer;
begin
 for i:=0 to 2 do
 begin
     x[i]:=a[i];  y[i]:=b[i];
 end;
end;
//Методы вычисления сторон треугольника a, b, c
function TTriangle.a():real;
begin
 a:=sqrt(sqr(x[1]-x[0])+sqr(y[1]-y[0]));
end;
function TTriangle.b():real;
begin
 b:=sqrt(sqr(x[2]-x[1])+sqr(y[2]-y[1]));
end;
function TTriangle.c():real;
begin
 c:=sqrt(sqr(x[0]-x[2])+sqr(y[0]-y[2]));
end;
//Методы вычисления периметра треугольника.
function TTriangle.Perimetr():real;
begin
 Perimetr:=a()+b()+c();
```

```
   end;
   //Функции вычисления площади треугольника.
   function TTriangle.Square():real;
   var p:real;
   begin
    p:=Perimetr()/2;   //полупериметр
    Squire:=sqrt((p-a())*(p-b())*(p-c()));
   end;
   //Метод проверки существования треугольника:
   //если в уравнение прямой, проходящей через
   //две точки, подставить координаты 3-й точки
   //и при этом получится равенство, значит, три
   //точки лежат на одной прямой и построение
   //треугольника невозможно.
   function TTriangle.Proverka():boolean;
   begin
    if (x[0]-x[1])/(x[0]-x[2])=
                  (y[0]-y[1])/(y[0]-y[2]) then
       Proverka:=false
    else Proverka:=true
   end;
   //Метод формирования строки –
   //сведений о треугольнике.
   function TTriangle.Show():string;
   begin
   //Если треугольник существует,
   //то формируем строку сведений о треугольнике.
    if Proverka() then
       Show:='Tr'+chr(13)+'a='+
       FloatToStrF(a(),ffFixed,5,2)+
chr(13)+'b='+FloatToStrF(b(),ffFixed,5,2)+
chr(13)+'c='+FloatToStrF(c(),ffFixed,5,2)+
chr(13)+'P='+FloatToStrF(Perimetr(),ffFixed,5,2)+
chr(13)+'S='+FloatToStrF(Square(),ffFixed,5,2)
       else
       Show:='Not Triangle';
   end;
```

```
//Метод проверки существования
//равностороннего треугольника.
function R_TTriangle.Proverka():boolean;
begin
 if (a()=b()) and(b()=c()) then
     Proverka:=true
                        else
     Proverka:=false
end;
//Метод формирования сведений
//о равностороннем треугольнике.
function R_TTriangle.Show():string;
begin
//Если треугольник равносторонний,
//то формируем строку сведений.
 if Proverka()=true then
Show:='Tr'+chr(13)+'a='+
FloatToStrF(a(),ffFixed,5,2)+
chr(13)+'P='+
FloatToStrF(Perimetr(),ffFixed,5,2)+chr(13)
+'S='+FloatToStrF(Square(),ffFixed,5,2)
 else
     Show:='Not R_Triangle';
end;
{ TForm1 }
procedure TForm1.Button1Click(Sender: Tobject);
//Массивы x1, y1 – координаты треугольника.
//Массивы x2, y2 – координаты
//равностороннего треугольника
 var x1, y1, x2, y2 :array[1..3] of real;
 s:string;
begin
//Чтение координат треугольников
//из полей ввода диалогового окна.
 x1[1]:=StrToFloat(Edit1.Text);
 y1[1]:=StrToFloat(Edit2.Text);
 x1[2]:=StrToFloat(Edit3.Text);
 y1[2]:=StrToFloat(Edit4.Text);
```

```
 x1[3]:=StrToFloat(Edit5.Text);
 y1[3]:=StrToFloat(Edit6.Text);
 x2[1]:=StrToFloat(Edit7.Text);
 y2[1]:=StrToFloat(Edit8.Text);
 x2[2]:=StrToFloat(Edit9.Text);
 y2[2]:=StrToFloat(Edit10.Text);
 x2[3]:=StrToFloat(Edit11.Text);
 y2[3]:=StrToFloat(Edit12.Text);
//Инициализация объекта класса треугольник.
 Figura1:=TTriangle.Create;
//Инициализация объекта класса
//равносторонний треугольник.
 Figura2:=R_TTriangle.Create;
 Figura1.Set_Tr(x1,y1);
 Figura2.Set_Tr(x2,y2);
//Вызов методов формирования
//сведений и вывод строки на форму.
 s:=Figura1.Show();
 Label15.Caption:= S;
 s:=Figura2.Show();
 Label16.Caption:= S;
//уничтожение объектов
 Figura1.Free;
 Figura2.Free;
end;
initialization
 {$I unit1.lrs}
end.
```

Результаты работы программы представлены на рис. 9.3.

Абстрактный метод – это виртуальный или динамический метод, реализация которого не определена в том классе, где он объявлен. Предполагается, что этот метод будет перегружен в классе-наследнике. Вызывают метод только в тех классах, где он перезагружен. Объявляется абстрактный метод при помощи служебного слова `abstract` после слов `virtual` или `dynamic`, например:

```
        метод1;    virtual;        abstract;
```

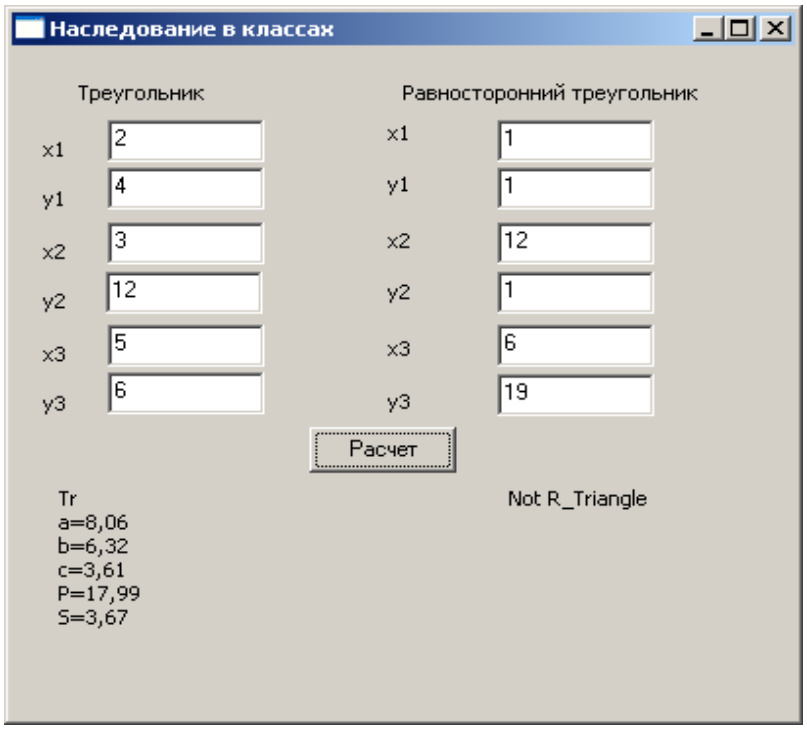

Рисунок 9.3: Результаты работы программы расчета параметров треугольников

Рассмотрим следующий пример. Создадим базовый класс `TFigure` (фигура). На основании этого класса можно построить производные классы для реальных фигур (окружность, четырехугольник и т.д.). В нашем примере рассмотрим два класса-наследника `TCircle` (окружность) и `TRectangle` (прямоугольник). На форму поместим кнопку. При щелчке по кнопке специализируются 2 объекта типа «Окружность» и «Прямоугольник», рассчитываются параметры фигур : периметр (длина окружности), площадь, результаты выводятся в компоненты `Label1` и `label2`. Ниже приведен листинг программы.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes, SysUtils, LResources, Forms,
         Controls, Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
 TForm1 = class(TForm)
 Button1: TButton;
```

```
 Label1: TLabel;
 Label2: TLabel;
 procedure Button1Click(Sender: TObject);
private
 { private declarations }
public
 { public declarations }
end;
//Объявление базового класса «фигура».
type
 TFigure=class
private
 n:integer; //Количество сторон фигуры.
 p:array of real;   //Массив длин сторон фигуры.
public
//Абстрактный конструктор
//в каждом производном классе будет
//перегружаться.
constructor Create;        virtual; abstract;
//Метод вычисления периметра
function Perimetr():real;
//Абстрактный метод вычисления площади,в каждом
//производном классе будет перегружаться
//реальным методом.
function Square():real;virtual; abstract;
//Абстрактный метод формирования
//сведений о фигуре
//в каждом производном классе
//будет перегружаться.
function Show():string;virtual; abstract;
end;
//Объявление производного класса «окружность».
type
TCircle=class(TFigure)
public
constructor Create;        override;
function Perimetr():real;
function Square():real;    override;
```

```
function Show():string;     override;
end;
//Объявление производного
//класса «прямоугольник».
type
TRectangle=class(TFigure)
public
constructor Create;     override;
function Square():real;    override;
function Show():string;     override;
end;
var
 Form1: Tform1;
//Объект типа класса окружность.
 Figura1: Tcircle;
//Объект типа класса прямоугольник.
 Figura2:TRectangle;
implementation
//Описание метода вычисления
//периметра для базового класса.
function  TFigure.Perimetr():real;
var     i:integer;
    s:real;
begin
 s:=0;
 for i:=0 to n-1 do
    s:=s+p[i];
 Perimetr:=s;
end;
//Описание конструктора в классе окружность
//перезагрузка абстрактного
//родительского конструктора.
constructor TCircle.Create;
begin
//Количество сторон для окружности 1.
 n:=1;
//Выделяем память под 1 элемент массива.
 SetLength(p,n);
```

```
 p[0]:=5;   //Сторона - радиус окружности.
end;
//Перезагрузка метода вычисления периметра.
function  TCircle.Perimetr():real;
begin
//Вычисление длины окружности.
 Perimetr:=2*Pi*p[0];
end;
//Перезагрузка метода вычисления площади
function  TCircle.Square():real;
begin
 Square:=Pi*sqr(p[0]);
end;

//Описание метода формирования
//строки сведений об окружности.
//Перезагрузка родительского
//абстрактного метода.
function TCircle.Show():string;
begin
 Show:='Circle'+chr(13)+'r='+
 FloatToStr(p[0])+chr(13)+'P='+
 FloatToStr(Perimetr())+
 chr(13)+'S='+FloatToStr(Square());
end;
//Описание конструктора в классе прямоугольник.
//перезагрузка абстрактного
//родительского конструктора.
constructor TRectangle.Create;
begin
 n:=2;   //Количество сторон — две.
//Выделение памяти под два элемента.
 SetLength(p,n);
 p[0]:=4; p[1]:=2;  //Длины сторон.
End;
//Перезагрузка абстрактного
//родительского метода,
//вычисления площади фигуры.
```

```
function  TRectangle.Square():real;
begin
 Square:=p[0]*p[1];
end;

//Описание метода формирования
//сведений о прямоугольнике.
//Перезагрузка родительского
//абстрактного метода.
function TRectangle.Show():string;
begin
 Show:='Rectangle'+chr(13)+'a='+
 FloatToStr(p[0])+'  b='+
 FloatToStr(p[1])+
chr(13)+'P='+FloatToStr(Perimetr())+
 chr(13)+'S='+FloatToStr(Square());
end;
{ TForm1 }
procedure TForm1.Button1Click(Sender: TObject);
var s:string;
begin
//Инициализация объекта типа окружность.
 Figura1:=TCircle.Create;
//Инициализация объекта типа прямоугольник.
 Figura2:=TRectangle.Create;
//Формирование сведений о фигурах
//и вывод результатов на форму.
 s:=Figura1.Show() ;
 label1.Caption:= s ;
 s:=Figura2.Show() ;
 label2.Caption:= s ;
 Figura1.Free;
 Figura2.Free;
end;
initialization
{$I unit1.lrs}
end.
```

9.4 Перегрузка операций

Во Free Pascal можно перегрузить не только функции, но и операции, например, можно запрограммировать, чтобы операция * при работе с матрицами осуществляла умножение матриц, а при работе с комплексными числами – умножение комплексных чисел.

Для этого в программе нужно написать специальную функцию – метод. Объявление метода перегрузки записывается после объявления класса и выглядит так:

```
operator symbols(параметры:тип)имя_результата:тип;
```

где `operator` – служебное слово;

`symbols` – символ перегружаемой операции;

`параметры` – имена переменных, участвующих в перегрузке оператора;

Описывают метод перегрузки так же, как и другие методы, после ключевого слова `implemention` того модуля, в котором объявлен класс.

Рассмотрим несколько примеров.

Задача 9.1. Создать класс работы с комплексными числами, в котором перегрузить операции сложения и вычитания.

На форме разместим четыре компонента типа `TEdit` для ввода действительной и мнимой частей двух комплексных чисел. Также создадим компонент типа `TMemo` для вывода результатов и кнопку. При щелчке по кнопке создаются четыре объекта типа «комплексное число». Первое и второе числа инициализируются введенными данными. Третье формируется как результат сложения первых двух, а четвертое — как разность этих же чисел. Текст программы с комментариями приведен ниже, на рис. 9.4 показано окно работы программы.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes, SysUtils, LResources, Forms,
          Controls, Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
 TForm1 = class(TForm)
 Button1: Tbutton;
```

```
//Для ввода действительной части первого числа.
 Edit1: Tedit;
//Для ввода мнимой части первого числа.
 Edit2: TEdit;
//Для ввода действительной части второго числа.
 Edit3: TEdit;
//Для ввода мнимой части второго числа.
 Edit4: TEdit;
 Label1: TLabel;
 Label2: TLabel;
 Label3: TLabel;
 Label4: TLabel;
 Label5: TLabel;
 Label6: TLabel;
 Memo1: Tmemo;  //Поле для вывода результатов.
 procedure Button1Click(Sender: TObject);
private
 { private declarations }
public
 { public declarations }
end;
//Объявление класса – комплексное число.
type
 TComplex = class
 private
     x: real;   //Действительная часть.
     y: real;   //Мнимая часть.
 public
     constructor Create;   //Конструктор.
//Функция вычисления модуля комплексного числа.
 function Modul():real;
//Функция вычисления аргумента
//комплексного числа;
 function Argument():real;
//Функция, представляющая комплексное число
//в виде строки для вывода.
 function ComplexToStr(): String;
end;
```

```
//Методы перегрузки оператора «+»
//с комплексными числами.
 operator +(const a, b: TComplex)r: TComplex;
//Методы перегрузки оператора «-»
//с комплексными числами.
 operator -(const a, b: TComplex)r: TComplex;
var
 Form1: TForm1;
//Объявление переменных типа комплексное число.
 chislo1: Tcomplex;
 chislo2: TComplex;
 chislo3: TComplex;
 chislo4: TComplex;
implementation
//Описание конструктора.
constructor TComplex.Create;
begin
     x:=0; y:=0;
     inherited Create;
end;
function TComplex.Modul(): real;
begin
     modul:=sqrt(x*x+y*y);
end;
function TComplex.Argument(): real;
begin
     argument:=arctan(y/x)*180/pi;
end;
//Методы перегрузки оператора «+».
//переменные a и b – комплексные числа,
//которые складываются,
//r – результирующее комплексное число
operator +(const a, b: TComplex)r: TComplex;
begin
 r:=TComplex.Create; //Инициализация объекта.
//Действительная часть нового комплексного
//числа формируется как результат сложения
//действительных частей первого и второго
```

```
//операндов. Мнимая часть нового комплексного
//числа формируется как результат сложения
//мнимых частей первого и второго операндов.
//Операнды - переменные типа комплексное
//число, передаваемые в метод.
 r.x:=a.x+b.x;
 r.y:=a.y+b.y;
end;
//Методы перегрузки оператора «-».
operator -(const a, b: TComplex)r: TComplex;
begin
 r:=TComplex.Create; //Инициализация объекта.
//Действительная часть нового комплексного
//числа формируется как разность действительных
//частей первого и второго операндов. Мнимая
//часть нового комплексного числа формируется
//как разность мнимых частей первого и второго
//операндов. Операнды - переменные типа
//комплексное число, передаваемые в метод.
 r.x:=a.x-b.x;
 r.y:=a.y-b.y;
end;
function TComplex.ComplexToStr(): String;
begin
 if y>=0 then
 ComplexToStr:=FloatToStrF(x,ffFixed,5,2)+'+'+
     FloatTostrF(y,ffFixed,5,2)+ 'i'
         else
 ComplexToStr:=FloatTostrF(x,ffFixed,5,2)+
     FloatTostrF(y,ffFixed,5,2)+ 'i'
end;
procedure TForm1.Button1Click(Sender: TObject);
Var Str1 : String;
begin
//Инициализация объектов.
//chislo1, chislo2 – исходные комплексные
//числа, вводимые с формы.
//chislo3 – Комплексное число,
```

```
//получаемое сложением двух исходных чисел.
//chislo4 – Комплексное число, получаемое
//вычитанием второго числа из первого.
 chislo1:=TComplex.Create;
 chislo2:=TComplex.Create;
 chislo3:=TComplex.Create;
 chislo4:=TComplex.Create;
//Чтение действительной и мнимой частей
//двух исходных комплексных чисел
//из полей ввода формы.
 chislo1.x:=StrToFloat(Edit1.Text);
 chislo1.y:=StrToFloat(Edit2.Text);
 chislo2.x:=StrToFloat(Edit3.Text);
 chislo3.y:=StrToFloat(Edit4.Text);
//Для сложения двух комплексных чисел можно
//использовать операцию +,так как она
//перегружена для этого класса.
 chislo3:=chislo1+chislo2;
//Для вычитания двух комплексных чисел
//можно использовать операцию -,так как она
//перегружена для этого класса.
 chislo4:=chislo1-chislo2;
//Результат сложения двух чисел преобразуется
//в строку для вывода в поле Memo1.
 Str1:='chislo1+chislo2 '+
                chislo3.ComplexToStr() ;
 Memo1.Lines.Add(Str1) ;
//Значение модуля преобразуется
//в строку для вывода в поле Memo1.
 Str1:='Modul chisla '+
  FloatToStrF(chislo3.Modul(),ffFixed, 5, 2);
Memo1.Lines.Add(Str1);
//Значение аргумента преобразуется в строку
//для вывода в поле Memo1.
Str1:='Argument chisla '+
FloatToStrF(chislo3.Argument(), ffFixed, 5, 2);
Memo1.Lines.Add(Str1) ;
//Результат разности двух чисел преобразуется
```

```
//в строку для вывода в поле Memo1.
Str1:='chislo1-chislo2 '+
                    chislo4.ComplexToStr() ;
Memo1.Lines.Add(Str1) ;
Str1:='Modul chisla '+
  FloatToStrF(chislo4.Modul(),ffFixed, 5, 2);
Memo1.Lines.Add(Str1) ;
Str1:='Argument chisla '+
FloatToStrF(chislo4.Argument(), ffFixed,5,2);
Memo1.Lines.Add(Str1) ;
end;
initialization
 {$I unit1.lrs}end.
```

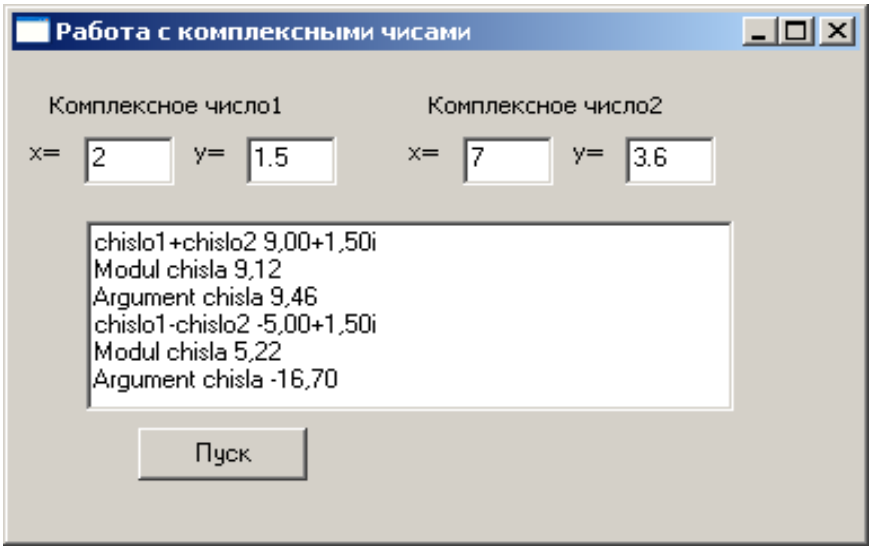

Рисунок 9.4: Пример работы программы к задаче 9.1

Задача 9.2. Создать класс работы с квадратными матрицами, в котором перегрузить операции сложения и умножения. В классе создать метод проверки, что матрица является единичной.

В классе созданы два метода перегрузки операции сложения и два метода перегрузки операции умножения. Это методы сложения и умножения матриц, а также методы добавления к матрице числа и умножения матрицы на число.

На форму поместим кнопку и 8 компонентов типа `TLabel` для вывода результатов. При щелчке по кнопке создаются 6 объектов типа «матрица», все они заполняются случайными числами. Третья матрица затем рассчитывается как сумма первых двух. Четвертая матрица

— результат умножения первой матрицы на вторую. Пятая матрица получается из первой путем добавления к ней числа 10, а шестая — умножением первой матрицы на число 5. Также первая и вторая матрицы проверяются, не являются ли они единичными. Текст программы с комментариями приведен далее.

```
   unit Unit1;
   {$mode objfpc}{$H+}
   interface
   uses
    Classes,    SysUtils,    LResources,    Forms,
Controls, Graphics, Dialogs, StdCtrls;
   type
    { TForm1 }
    TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Memo1Change(Sender: TObject);
   private
    { private declarations }
   public
    { public declarations }
   end;
   //Объявление класса матрица.
   type
    TMatrix = class
    private
        //элементы матрицы
        x: array [0..5,0..5]of real;
    public
        constructor Create;       //Конструктор.
   //Метод формирования строки,
   //выводящей элементы матрицы.
```

```
    function MatrixToStr(): String;
//Функция проверки, является ли
//матрица единичной.
 function Pr_Edin(): boolean;
end;
//Методы перегрузки операции «+» -
//сложение двух матриц.
 operator +(const a, b: TMatrix)r: Tmatrix;
//Методы перегрузки операции «+» -
//добавление к матрице вещественного числа.
operator+(const a:TMatrix;b:real)r:TMatrix;
//Метоы перегрузки операции «*» -
//умножение двух матриц.
 operator *(const a, b: TMatrix)r: Tmatrix;
//Методы перегрузки операции «*» -
//умножение матрицы на вещественное число.
 operator *(const a:TMatrix;b:real)r: TMatrix;
var
 Form1: TForm1;
 matr1: TMatrix;
 matr2: TMatrix;
 matr3: TMatrix;
 matr4: TMatrix;
 matr5: TMatrix;
 matr6: TMatrix;
implementation
//Конструктор в классе матрица, который
//заполняет матрицу случайными целыми числами.
constructor TMatrix.Create;
var i, j : integer;
begin
 for i:=0 to 4 do
     for j:=0 to 4 do
         x[i,j]:=Random(10);
 inherited Create;
end;
//Функция, проверяющая,
//является ли матрица единичной.
```

```
   function TMatrix.Pr_Edin():boolean;
   var i, j : integer;
   begin
   //Предполагаем, что матрица единичная.
    Result:=true;
    for i:=0 to 4 do
        for j:=0 to 4 do
   //Если на главной диагонали элемент не 1
   //или вне главной диагонали элемент не 0,
if ((i=j)and(x[i,j]<>1))or((i<>j)and(x[i,j]<>0))
then
    begin
   //то матрица не является единичной.
            Result:=false;
            break;  //выход из цикла
    end;
   end;
   //Метод перегрузки оператора «+» -
   //сложение двух матриц
   operator +(const a, b: TMatrix)r: TMatrix;
   var i, j: integer;
   begin
    r:=TMatrix.Create;
    for i:=0 to 4 do
        for j:=0 to 4 do
            r.x[i,j]:=a.x[i,i]+b.x[i,j];
    end;
   //Методы перегрузки оператора «*» -
   //перемножение двух матриц.
   operator *(const a, b: TMatrix)r: TMatrix;
   var  i, j, k: integer;
    s: real;
   begin
    r:=TMatrix.Create;
    for i:=0 to 4 do
        for j:=0 to 4 do
            begin
                r.x[i,j]:=0;
```

```
            for k:=0 to 4 do
                r.x[i,j]:=
                r.x[i,j]+a.x[i,k]*b.x[k,j];
        end;
end;
//Методы перегрузки оператора «+» -
//сложение матрицы и вещественного числа.
operator +(const a: TMatrix;b:real)r: TMatrix;
var i, j: integer;
begin
 r:=TMatrix.Create;
 for i:=0 to 4 do
     for j:=0 to 4 do
         r.x[i,j]:=a.x[i,j]+b;
end;
//Методы перегрузки оператора «*» -
//умножение матрицы на вещественное число
operator *(const a: TMatrix;b:real)r: TMatrix;
var i, j: integer;
begin
 r:=TMatrix.Create;
 for i:=0 to 4 do
     for j:=0 to 4 do
         r.x[i,j]:=a.x[i,j]*b;
end;
//Методы формирования строки
//с элементами матрицы.
function TMatrix.MatrixToStr(): String;
var  s:string;
     i, j :integer;
begin
 s:='';//Вначале строка пустая
 for i:=0 to 4 do
     begin
         for j:=0 to 4 do
//Добавление к s строки, которая представляет
//собой элемент матрицы и пробел.
     s:=s+FloatTostrF(x[i,j],ffFixed,5,2)+'  ';
```

```
//Формирование новой строки матрицы.
        s:=s+chr(13);
 end;
 MatrixToStr:=s;
end;
//Обработка кнопки Пуск:
//создание экземпляров класса Матрица,
//вычисление результирующих матриц путем
//сложения, умножения исходных матриц,
//умножение матрицы на число
procedure TForm1.Button1Click(Sender: TObject);
Var Str1 : String;
begin
//Инициализация объектов:
//matr1, matr2 – исходные матрицы,
//заполняемые случайными числами;
//matr3 – матрица, получаемая
//сложением двух исходных матриц;
//matr4 – матрица, получаемая умножением
//двух исходных матриц;
//matr5 – матрица, получаемая добавлением
//к первой матрице числа 10;
//matr6 – матрица, получаемая умножением
//первой матрицы на число 5.
 matr1:=TMatrix.Create;
 matr2:=TMatrix.Create;
 matr3:=TMatrix.Create;
 matr4:=TMatrix.Create;
 matr5:=TMatrix.Create;
 matr6:=TMatrix.Create;
//Вычисление матриц matr3, matr4, matr5, matr6
//с помощью перегрузки операторов.
 matr3:=matr1+matr2;
 matr4:=matr1*matr2;
 matr5:=matr1+10;
 matr6:=matr1*5;
//Формирование строки вывода и ее вывод
//в метку Label1 матрицы matr1.
```

```
 Str1:='Matrix1 '+chr(13)+matr1.MatrixToStr() ;
 Label1.Caption:= Str1;
//Проверка, является ли матрица matr1
//единичной,и вывод соответствующего сообщения.
 if matr1.Pr_Edin()=true then
     Str1:='Matrix1 edinichnaya'
 else
     Str1:='Matrix1 ne edinichnaya';
 Label5.Caption:= Str1 ;
//Проверка, является ли матрица matr2
//единичной,и вывод соответствующего сообщения.
 if matr2.Pr_Edin()=true then
     Str1:='Matrix2 edinichnaya'
 else
     Str1:='Matrix2 ne edinichnaya';
 Label6.Caption:= Str1 ;
 Str1:='Matrix2 '+chr(13)+matr2.MatrixToStr() ;
 Label2.Caption:= Str1 ;
//Вывод элементов матрицы matr3=matr1+matr2.
 Str1:='Matrix1+Matrix2 '+
            chr(13)+matr3.MatrixToStr() ;
 Label3.Caption:= Str1 ;
//Вывод элементов матрицы matr4=matr1*matr2.
 Str1:='Matrix1*Matrix2 '+
            chr(13)+matr4.MatrixToStr() ;
 Label4.Caption:= Str1 ;
//Вывод элементов матрицы matr5=matr1+10.
 Str1:='Matrix1+10 '+
            chr(13)+matr5.MatrixToStr() ;
 Label7.Caption:= Str1 ;
//Вывод элементов матрицы matr5=matr1*5.
 Str1:='Matrix1*5 '+chr(13)+
                  matr6.MatrixToStr() ;
 Label8.Caption:= Str1 ;
end;
initialization
 {$I unit1.lrs}
end.
```

Результат работы программы показан на рис. 9.5 .

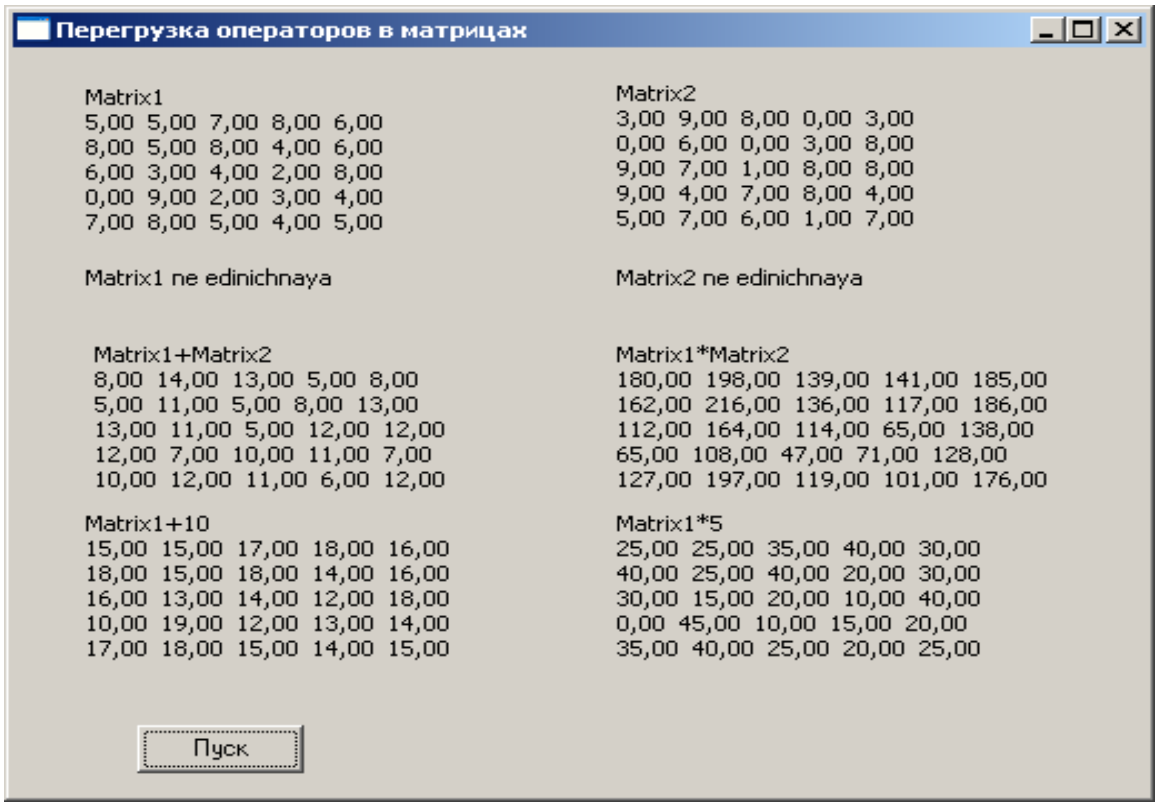

Рисунок 9.5: Результаты работы программы задачи 9.2

ЗАДАЧА 9.3. Создать класс – обыкновенная дробь, в котором перегрузить операции < и >.

На форму поместим четыре компонента `TEdit` для ввода числителей и знаменателей двух обыкновенных дробей и компонент `Memo1` для вывода результатов. Также разместим кнопку, при щелчке по которой инициализируются два объекта типа «обыкновенная дробь», затем они сравниваются и результаты выводятся в Memo1. Ниже приведен листинг программы с комментариями, на рис. 9.6 показаны результаты работы программы.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes,    SysUtils,    LResources,    Forms,
Controls, Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
 TForm1 = class(TForm)
 Button1: Tbutton;
```

```
//Поле для ввода числителя первой дроби.
 Edit1: Tedit;
//Поле для ввода знаменателя первой дроби.
 Edit2: Tedit;
//Поле для ввода числителя второй дроби.
 Edit3: Tedit;
//Поле для ввода знаменателя второй дроби.
 Edit4: Tedit;
 Label1: TLabel;
 Label2: TLabel;
 Label3: TLabel;
 Label4: TLabel;
 Memo1: Tmemo;  //Поле для вывода результатов.
 procedure Button1Click(Sender: TObject);
private
 { private declarations }
public
 { public declarations }
end;
//Объявление класса – обыкновенная дробь.
type
 TOb_Drob = class
 private
     Ch: integer;       //Числитель.
     Zn: integer;       //Знаменатель.
 public
     //Конструктор.
     constructor Create(a, b:integer);
 //Метод формирования строки для вывода дроби.
     function DrobToStr(): String;
end;
//Метод перегрузки оператора «<».
operator <(const a, b: TOb_Drob)r: boolean;
//Метод перегрузки оператора «>».
operator >(const a, b: TOb_Drob)r: boolean;
var
 Form1: Tform1;
//Объявление переменных типа класс
```

```
//обыкновенная дробь.
 d1, d2: TOb_Drob;
implementation
//Конструктор.
constructor TOb_drob.Create(a, b:integer);
begin
 Ch:=a; Zn:=b;
 inherited Create;
end;
//Метод перегрузки оператора «<» -
//сравнение обыкновенных дробей.
operator <(const a, b: TOb_Drob)r: boolean;
begin
 if a.Ch*b.Zn <b.Ch * a.Zn then
        r:=true
 else r:=false ;
end;
//Метод перегрузки оператора «>» -
//сравнение обыкновенных дробей.
operator >(const a, b: TOb_Drob)r: boolean;
begin
 if a.Ch*b.Zn > b.Ch * a.Zn then
        r:=true
 else r:=false ;
end;
//Метод формирования строки
//для вывода дроби на форму.
function TOb_Drob.DrobToStr(): String;
begin
 if Zn<>0 then
     if Ch*Zn>0 then
     DrobToStr:=IntToStr(Ch)+'/'+
                                IntToStr(Zn)
     else
         DrobToStr:='-'+IntToStr(abs(Ch))+'/'+
                            IntTostr(abs(Zn))
 else
        DrobToStr:='Dividing by a zero'
```

```
end;
procedure TForm1.Button1Click(Sender: TObject);
Var  Str1 : String;
     a, b: integer;
begin
//Чтение данных из полей ввода формы:
 a:=StrToInt(Edit1.Text);  //числитель,
 b:=StrToInt(Edit2.Text);  //знаменатель.
//Инициализация первой дроби.
 d1:=TOb_Drob.Create(a,b);
//Чтение данных из полей ввода формы.
 a:=StrToInt(Edit3.Text);
 b:=StrToInt(Edit4.Text);
//Инициализация второй дроби.
 d2:=TOb_Drob.Create(a,b);
//Формирование строки вывода
//и добавление ее в поле Memo1.
//Вывод исходной дроби d1.
 Str1:='Drob 1 '+d1.DrobToStr() ;
 Memo1.Lines.Add(Str1) ;
//Вывод исходной дроби d2.
 Str1:='Drob2 '+d2.DrobToStr() ;
 Memo1.Lines.Add(Str1) ;
//сравнение дробей с помощью перегрузки
//операторов < и > и вывод сообщения.
 if d1<d2   then
     Str1:='Drob1 < Drob2'
            else
         if d1>d2   then
             Str1:='Drob1 > Drob2'
                 else
             Str1:='Drob1 = Drob2';
 Memo1.Lines.Add(Str1) ;
end;
initialization
  {$I unit1.lrs}
end.
```

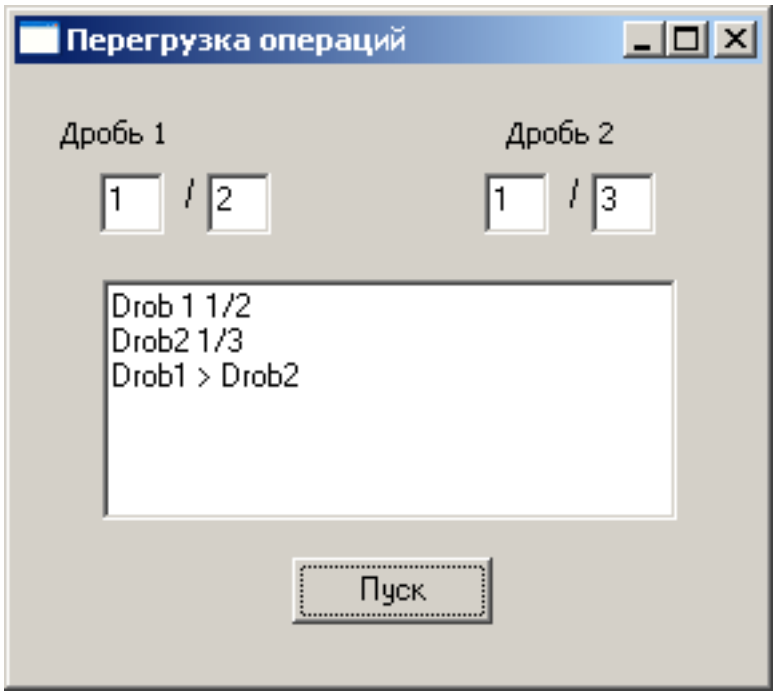

Рисунок 9.6: Результаты работы программы задачи 9.3

9.5 Задачи для самостоятельного решения

1. Создать класс *комплексное число* в алгебраической форме `z=x+y·i`, поля класса – действительная (`x`) и мнимая (`y`) части числа. Методы класса: вычисление корня комплексного числа, вывод комплексного числа. В классе предусмотреть методы перегрузки операций: сложение, вычитание, деление и умножение комплексных чисел.

2. Создать класс *квадратная матрица*, поля класса – размерность и элементы матрицы. Метод класса: вывод матрицы. В классе предусмотреть методы перегрузки операций: сложение, вычитание, умножение матриц, проверку, является ли одна матрица обратной другой ($A \cdot A^{-1} = E$).

3. Создать класс *вектор на плоскости*, поля класса – координаты вектора. Методы класса: вычисление направляющих косинусов вектора, вывод всех характеристик вектора. В классе предусмотреть методы перегрузки операций: сложение, скалярное и векторное произведения векторов.

4. Создать класс *обыкновенная дробь*, поля класса – числитель и знаменатель. Методы класса: сокращение дроби, вывод дроби. В классе предусмотреть методы перегрузки операций: сложение, вычитание,

деление и умножение дробей.

5. Создать класс *квадрат*, член класса – длина стороны. Предусмотреть в классе методы вычисления и вывода сведений о фигуре – периметр, площадь, диагональ. Создать производный класс – *куб*, добавить в класс метод определения объема фигуры, перегрузить методы расчета площади и вывода сведений о фигуре.

6. Создать класс *квадратная матрица*, поля класса – размерность и элементы матрицы. Методы класса: вычисление суммы всех элементов матрицы, вывод матрицы. В классе предусмотреть методы перегрузки операций: сложение, вычитание, умножение матриц, умножение матрицы на число.

7. Создать класс *прямая,* поля класса – координаты двух точек (`x1, y1`) и (`x2, y2`). Метод класса: вывод уравнения прямой вида `y= ax + b`. В классе предусмотреть методы перегрузки операций: проверка параллельности двух прямых, определение угла между двумя прямыми.

8. Создать класс *комплексное число* в тригонометрической форме $a = \rho(\cos\varphi + i\sin\varphi)$, поля класса – модуль (ρ) и аргумент (φ) числа. Методы класса: возведение числа в степень, вывод комплексного числа в алгебраической и тригонометрической формах. В классе предусмотреть методы перегрузки операций: сложение, вычитание, деление и умножение комплексных чисел.

9. Создать класс *вектор на плоскости*, поля класса – координаты вектора. Методы класса: вычисление длины вектора, вывод характеристик вектора. В классе предусмотреть методы перегрузки операций: сложение, скалярное и векторное произведения векторов.

10. Создать класс *обыкновенная дробь*, поля класса – числитель и знаменатель. Методы класса: определение обратной дроби, вывод дроби. В классе предусмотреть методы перегрузки операций: сложение, вычитание, деление и умножение дробей.

11. Создать класс *квадратная матрица*, поля класса – размерность и элементы матрицы. Методы класса: проверка, является ли матрица верхнетреугольной или нижнетреугольной, вывод матрицы. В классе предусмотреть методы перегрузки операций: сложение, вычитание, умножение матриц, умножение матрицы на число.

12. Создать класс *треугольник*, члены класса – длины 3-х сторон. Предусмотреть в классе методы проверки существования треугольни-

ка, вычисления и вывода сведений о фигуре – длины сторон, углы, периметр, площадь. Создать производный класс – *равнобедренный треугольник*, предусмотреть в классе проверку, является ли треугольник равнобедренным.

13. Создать класс *комплексное число* в показательной форме $a = \rho e^{i\varphi}$, поля класса – модуль (ρ) и аргумент (φ) числа. Методы класса: вывод комплексного числа в алгебраической, тригонометрической и показательной формах. В классе предусмотреть методы перегрузки операций: сложение, вычитание, деление и умножение комплексных чисел.

14. Создать класс *прямая,* поля класса – коэффициенты уравнения `y= ax + b`. Методы класса: вывод уравнения прямой, определение точек пересечения с осями. В классе предусмотреть методы перегрузки операций: проверка перпендикулярности двух прямых, определение угла между двумя прямыми.

15. Создать класс *квадратная матрица*, поля класса – размерность и элементы матрицы. Методы класса: проверки, является ли матрица диагональной или нулевой, вывод матрицы. В классе предусмотреть методы перегрузки операций: сложение, вычитание, умножение матриц, добавление к матрице числа.

16. Создать класс *треугольник*, члены класса – координаты 3-х точек. Предусмотреть в классе методы проверки существования треугольника, вычисления и вывода сведений о фигуре – длины сторон, углы, периметр, площадь. Создать производный класс – *прямоугольный треугольник*, предусмотреть в классе проверку, является ли треугольник прямоугольным.

17. Создать класс *комплексное число* в тригонометрической форме $a = \rho(\cos\varphi + i\sin\varphi)$, поля класса – модуль (ρ) и аргумент (φ) числа. Методы класса: извлечение корня из числа, вывод комплексного числа в алгебраической и тригонометрической формах. В классе предусмотреть методы перегрузки операций: сложение, вычитание, деление и умножение комплексных чисел.

18. Создать класс *обыкновенная дробь*, поля класса – числитель и знаменатель. Методы класса: возведение дроби в степень, вывод дроби. В классе предусмотреть методы перегрузки операций: сложение, вычитание, деление и умножение дробей.

19. Создать класс *треугольник*, члены класса – длины 3-х сторон.

Предусмотреть в классе методы проверки существования треугольника, вычисления и вывода сведений о фигуре – длины сторон, углы, периметр, площадь. Создать производный класс – *равносторонний треугольник*, предусмотреть в классе перегрузку метода проверки существования равностороннего треугольника.

20. Создать класс *комплексное число* в алгебраической форме `z=x+ y ·i`, поля класса – действительная (`x`) и мнимая (`y`) части числа. Методы класса: вычисление модуля и аргумента комплексного числа, вывод комплексного числа. В классе предусмотреть методы перегрузки операций: сложение, вычитание комплексных чисел, проверки сопряженности двух комплексных чисел.

21. Создать класс *окружность*, член класса – радиус *R*. Предусмотреть в классе методы вычисления и вывода сведений о фигуре – площадь, длина окружности. Создать производный класс – *круглый прямой цилиндр с высотой h***,** добавить в класс метод определения объема фигуры, перегрузить методы расчета площади и вывода сведений о фигуре.

22. Создать класс *вектор на плоскости*, поля класса – координаты вектора. Методы класса: вычисление длины вектора, вывод характеристик вектора. В классе предусмотреть методы перегрузки операций: сложение, скалярное и векторное произведения векторов, вычисление угла между векторами.

23. Создать класс *квадратная матрица*, поля класса – размерность и элементы матрицы. Методы класса: проверка, является ли матрица симметричной ($A = A^T$), вывод матрицы. В классе предусмотреть методы перегрузки операций: сложение, вычитание, умножение матриц, добавление к матрице числа.

24. Создать класс *обыкновенная дробь*, поля класса – числитель и знаменатель. Метод класса: вывод дроби. В классе предусмотреть методы перегрузки операций: сложение, вычитание, деление и умножение дробей, сравнение дробей.

25. Создать класс *квадрат*, члены класса - длина стороны. Предусмотреть в классе методы вычисления и вывода сведений о фигуре – диагональ, периметр, площадь. Создать производный класс – *правильная квадратная призма с высотой H*, добавить в класс метод определения объема фигуры, перегрузить методы расчета площади и вывода сведений о фигуре.

10 Графика во Free Pascal

Данная глава посвящена графическим средствам Free Pascal. Рассмотрены основные процедуры и функции работы с графикой. Приведен пример построения графика математической зависимости.

10.1 Средства рисования в Lazarus

При разработке оконного приложения Lazarus есть форма, на которой можно рисовать. В распоряжении программиста находится полотно (холст) — свойство `Canvas`, карандаш — свойство `Pen`, и кисть — свойство `Brush`.

Свойством `Canvas` обладают следующие компоненты:

- форма (класс `Tform`);
- таблица (класс `TSringGrid`);
- растровая картинка (класс `Timage`);
- принтер (класс `TPrinter`).

При рисовании компонента, обладающего свойством `Canvas`, сам компонент рассматривается как прямоугольная сетка, состоящая из отдельных точек, называемых пикселями.

Положение пикселя характеризуется его вертикальной `(X)` и горизонтальной `(Y)` координатами. Левый верхний пиксель имеет координаты `(0,0)`. Вертикальная координата возрастает сверху вниз, горизонтальная — слева направо. Общее количество пикселей по вертикали определяется свойством `Height`, а по горизонтали — свойством `Weight`. Каждый пиксель может иметь свой цвет. Для доступа к любой точке полотна используется свойство

```
Pixels[X,Y]:TColor.
```

Это свойство определяет цвет пикселя с координатами `X(integer), Y(integer).`

Изменить цвета любого пикселя полотна можно с помощью следующего оператора присваивания

```
Компонент.Canvas. Pixels[X,Y]:=Color;
```

где `Color` — переменная или константа типа `Tcolor`. Определены следующие константы цветов (тал. 10.1):

Таблица 10.1. Значение свойств Color

Константа	Цвет	Константа	Цвет
clBlack	Черный	clSilve	Серебристый
clMaroon	Каштановый	clRed	Красный
clGreen	Зеленый	clLime	Салатовый
clOlive	Оливковый	clBlue	Синий
clNavy	Темно-синий	clFuchsia	Ярко-розовый
clPurple	Розовый	clAqua	Бирюзовый
clTeal	Лазурный	clWhite	Белый
clGray	Серый		

Цвет любого пикселя можно получить с помощью следующего оператора присваивания:

```
Color:=Компонент.Canvas. Pixels[X,Y];
```

где Color — переменная типа Tcolor.

Класс цвета точки Tcolor определяется как длинное целое longint. Переменные этого типа занимают в памяти четыре байта. Четыре байта переменных этого типа содержат информацию о долях синего (B), зеленого (G) и красного(R) цветов и устроены следующим образом: $00BBGGRR.

Для рисования используются методы класса TCanvas, позволяющие изобразить фигуру (линию, прямоугольник и т.д.) или вывести текст в графическом режиме, и три класса, определяющие инструменты вывода фигур и текстов:

- TFont (шрифты);
- TPen (карандаш, перо);
- TBrush (кисть).

Класс TFONT

Можно выделить следующие свойства объекта Canvas.TFont:

- Name (тип string) — имя используемого шрифта.
- Size (тип integer) — размер шрифта в пунктах (points). Пункт — это единица измерения шрифта, равная 0,353 мм, или 1/72 дюйма.
- Style — стиль начертания символов, который может быть обычным, полужирным (fsBold), курсивным (fsItalic), под-

черкнутым (`fsUnderline`) и перечеркнутым (`fsStrikeOut`). В программе можно комбинировать необходимые стили, например, чтобы установить стиль «полужирный курсив», необходимо написать следующий оператор:

```
Объект.Canvas. Font. Style:=[fsItalic,fsBold]
```

- `Color` (тип `Tcolor`) — цвет символов.
- `Charset` (тип 0..255) — набор символов шрифта. Каждый вид шрифта, определяемый его именем, поддерживает один или более наборов символов. В табл. 10.2 приведены некоторые значения `Charset`.

Таблица 10.2. Значения свойства `Charset`

Константа	Значение	Описание
ANSI_CHARSET	0	Символы ANSI
DEFAULT_CHARSET	1	Задается по умолчанию. Шрифт выбирается только по его имени Name и размеру Size. Если описанный шрифт недоступен в системе, будет заменен другим
SYMBOL_CHARSET	2	Стандартный набор символов
MAC_CHARSET	77	Символы Macintosh
GREEK_CHARSET	161	Греческие символы
RUSSIAN_CHARSET	204	Символы кириллицы
EASTEUROPE_CHARSET	238	Включает диалектические знаки (знаки, добавляемые к буквам и характеризующие их произношение) для восточно-европейских языков

Класс TPEN

Карандаш (перо) используется как инструмент для рисования точек, линий, контуров геометрических фигур. Основные свойства объекта `Canvas.TPen`:

- `Color` (тип Tcolor) – определяет цвет линии;
- `Width` (тип Integer) – задает толщину линии в пикселях;

• `Style` – дает возможность выбрать вид линии. Это свойство может принимать значение, указанное в таблице 10.3.

Таблица 10.3. Виды линий

Значение	Описание
`psSolid`	Сплошная линия
`psDash`	Штриховая линия
`psDot`	Пунктирная линия
`psDashDot`	Штрих-пунктирная линия
`psDashDodDot`	Линия, чередующая шрих и два пунктира
`psClear`	Нет линии

• `Mode` – определяет, каким образом взаимодействуют цвета пера и полотна. Выбор значения этого свойства позволяет получать различные эффекты, возможные значения `Mode` приведены в табл. 10.4. По умолчанию вся линия вычерчивается цветом, определяемым значением `Pen.Color`, но можно определять инверсный цвет линии по отношению к цвету фона. В этом случае независимо от цвета фона, даже если цвет линии и фона одинаков, линия будет видна.

Таблица 10.4. Возможные значения свойства Mode

Режим	Операция	Цвет пикселя
`pmBlack`	`Black`	Всегда черный
`pmWhite`	`White`	Всегда белый
`pmNop`	`-`	Неизменный
`pmNot`	`Not Screen`	Инверсный цвет по отношению к цвету фона
`pmCopy`	`Pen`	Цвет, указанный в свойствах Color пера Pen (это значение принято по умолчанию)
`pmNotCopy`	`Not Pen`	Инверсия цвета пера

Режим	Операция	Цвет пикселя
pmMergePenNot	Pen or Not Pen	Дизъюнкция цвета пера и инверсного цвета фона
pmMaskPenNot	Pen and Not Screen	Конъюнкция цвета пера и инверсного цвета фона
pmMergeNotPen	Not Pen or Screen	Дизъюнкция цвета фона и инверсного цвета пера
PmMaskNotPen	Not Pen and Screen	Конъюнкция цвета фона и инверсного цвета пера
pmMerge	Pen or Screen	Дизъюнкция цвета пера и цвета фона
pmNotMerge	Not (Pen or Screen)	Инверсия режима pmMerge
pmMask	Pen and Screen	Конъюнкция цвета пера и цвета фона
pmNotMask	Not (Pen and Screen)	Инверсия режима pmMask
pmXor	Pen xor Screen	Операция xor над цветом пера и цветом фона
pmNotXor	Not (Pen xor Screen)	Инверсия режима pmXor

Класс TBRUSH

Кисть (`Canvas.Brush`) используется методами, обеспечивающими вычерчивание замкнутых фигур для заливки. Кисть обладает двумя основными свойствами:

- `Color` (тип Tcolor) – цвет закрашивания замкнутой области;
- `Style` – стиль заполнения области.

Класс TCANVAS

Это класс является основным инструментом для рисования гра-

фики. Рассмотрим наиболее часто используемые методы этого класса.

```
Procedure MoveTo(X, Y : Integer);
```

Метод `MoveTo` изменяет текущую позицию пера на позицию, заданную точкой `(X, Y)`. Текущая позиция хранится в переменной `PenPos` типа `Tpoint`. Определение типа `TPoint` следующее:

```
type TPoint =record
 X: Longint;
 Y: Longint;
end;
```

Текущую позицию пера можно считывать с помощью свойства `PenPos` следующим образом:

```
X:=PenPos.X;
Y:=PenPos.Y;
Procedure LineTo(X, Y :Integer);
```

Метод `LineTo` соединяет прямой линией текущую позицию пера и точку с координатами `(X, Y)`. При этом текущая позиция пера перемещается в точку с координатами `(X, Y)`.

Рассмотрим работу процедуры на примере. Расположим на форме кнопку и рассмотрим процедуру обработки события `TForm1.Button1Click`, которая рисует прямые линии:

```
Procedure TfForm1.Button1Click(Sender: TObject)
begin
 Form1.Canvas.LineTo(30,50);
end;
```

В результате щелчка по кнопке на форме возникнет прямая линия, соединяющая точку с координатами (0,0) и точку с координатами (30,50). Теперь перепишем процедуру обработки события следующим образом:

```
Procedure TForm1.Button1Click(Sender: TObject)
begin
Form1.Canvas.LineTo(Canvas.PenPos.x+30,
                                Canvas.PenPos.y+50);
end;
```

При первом щелчке по кнопке на экране прорисуется аналогичная линия. Но при повторном щелчке первая процедура продолжает рисовать линию, соединяющую точки (0,0) и (30,50). Вторая процедура рисует линию, которая соединяет текущую точку с точкой, полу-

чившейся из текущей добавлением к координате X числа 30, а к координате Y – числа 50. Т.е. при повторном щелчке по кнопке процедура соединяет прямой линией точки (30,50) и (60,100). При третьем щелчке по кнопке будут соединяться прямой линией точки (60,100) и (90,150) и т.д.

```
Procedure PolyLine(const Points array of TPoint);
```

Метод `PolyLine` рисует ломаную линию, координаты вершин которой определяются массивом `Points`.

Рассмотрим работу процедуры на примере. Расположим на форме кнопки **Рисовать** и **Выход** и запишем следующие операторы процедур обработки события:

```
procedure TForm1.Button1Click(Sender: TObject);
var temp :array [1..25] of TPoint;
 i : byte;
 j: integer;
begin
 j:=1;
 for i:=1 to 25 do
 begin
//вычисление координат вершин ломаной линии
     temp[i].x:=25+(i-1)*10;
     temp[i].y:=150-j*(i-1)*5;
     j:=-j;
 end;
 Form1.Canvas.Polyline (temp);
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
 Form1.Close;
end;
```

После запуска программы и щелчка по кнопке «Рисовать» окно формы будет выглядеть, как на рисунке 10.1.

```
Procedure Ellipse (X1, Y1, X2, Y2 :Integer);
```

Метод `Ellipse` вычерчивает на холсте эллипс или окружность. `X1, Y1, X2, Y2` — это координаты прямоугольника, внутри которого вычерчивается эллипс. Если прямоугольник является квадратом, то вычерчивается окружность.

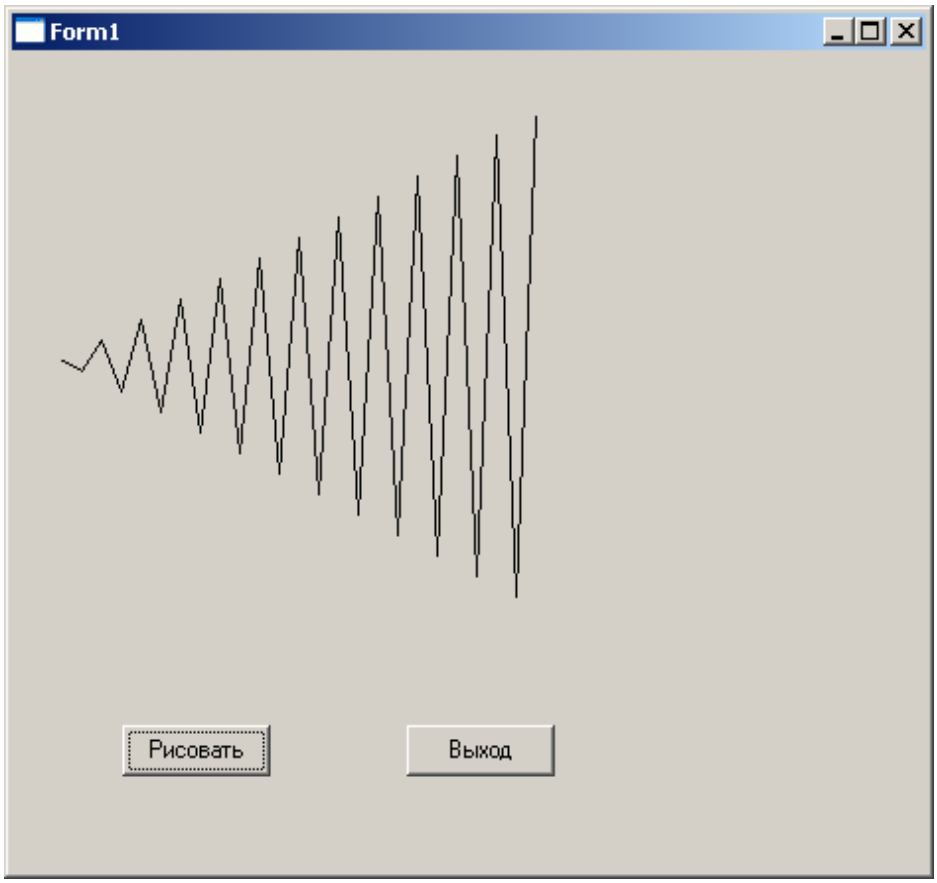

Рисунок 10.1: Пример использования процедуры `PolyLine`

Метод

```
Procedure Arc (X1,Y1,X2,Y2,X3,Y3,X4,Y4:Integer);
```

вычерчивает дугу эллипса. `X1, Y1, X2, Y2` — это координаты, определяющие эллипс, частью которого является дуга. `X3, Y3` — координаты, определяющие начальную точку дуги, `X4, Y4` — координаты, определяющие конечную точку дуги. Дуга рисуется против часовой стрелки.

Метод

```
Procedure Rectangle (X1, Y1, X2, Y2 :Integer);
```

рисует прямоугольник. `X1, Y1, X2, Y2` — координаты верхнего левого и нижнего правого углов прямоугольника.

Метод

```
Procedure RoundRect (X1,Y1,X2,Y2,X3,Y3:Integer);
```

вычерчивает прямоугольник со скругленными углами. `X1, Y1, X2, Y2` — координаты верхнего левого и нижнего правого углов прямоугольника, а `X3, Y3` — размер эллипса, одна четверть которого используется для вычерчивания скругленного угла.

Метод
`Procedure PolyGon(const Points array of TPoint);`
рисует замкнутую фигуру (многоугольник) по множеству угловых точек, заданному массивом `Points`. При этом первая точка соединяется прямой линией с последней. Этим метод `PolyGon` отличается от метода `Poliline`, который не замыкает конечные точки. Рисование осуществляется текущим пером `Pen`, а внутренняя область фигуры закрашивается текущей кистью `Brush`.

Метод
`Procedure Pie (X1,Y1,X2,Y2,X3,Y3,X4,Y4 :Integer);`
рисует замкнутую фигуру — сектор окружности или эллипса с помощью текущих параметров пера `Pen`, внутренняя область закрашивается текущей кистью Brush. Точки `(X1, Y1)` и `(X2, Y2)` задают прямоугольник, описывающий эллипс. Начальная точка дуги определяется пересечением эллипса с прямой, проходящей через его его центр и точку `(X3, Y3)`. Конечная точка дуги определяется пересечением эллипса с прямой, проходящей через его центр и точку `(X4, Y4)`. Дуга рисуется против часовой стрелки от начальной до конечной точки. Рисуются прямые, ограничивающие сегмент и проходящие через центр эллипса и точки `(X3, Y3)` и `(X4, Y4)`.

Создадим форму, установим ей размеры `Heigth` — 500, `Width` — 500. Внизу разместим кнопку, зададим ей свойство `Caption` — «`Рисовать`». При запуске программы и щелчке по этой кнопке на форме прорисуются различные фигуры (см. рис. 10.2). Ниже приведен листинг программы, демонстрирующий работу перечисленных методов. Результат работы программы приведен на рис. 10.2.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes,    SysUtils,    LResources,    Forms,
Controls, Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
 TForm1 = class(TForm)
 Button1: Tbutton;
```

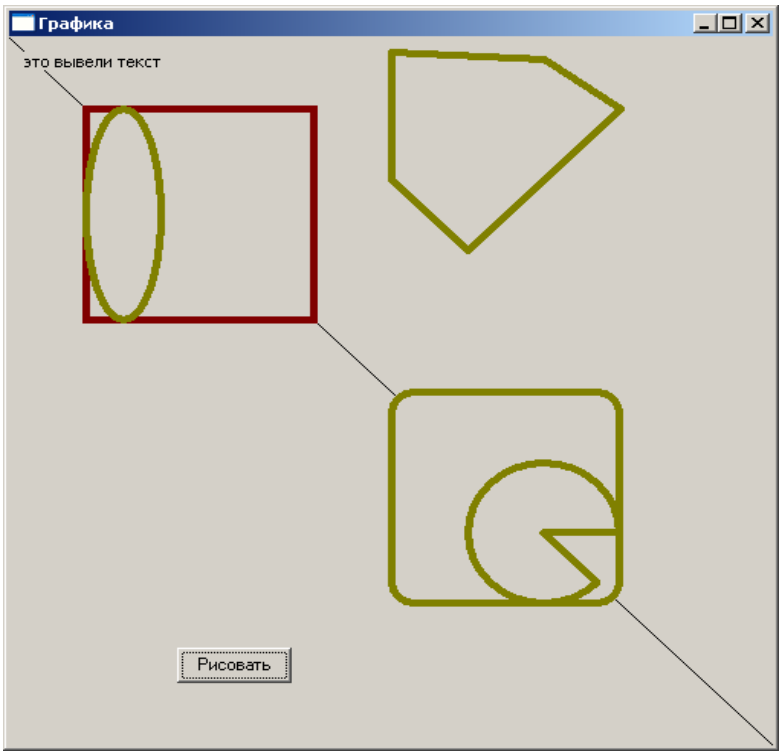

Рисунок 10.2: Пример использования методов рисования фигур

```
 procedure Button1Click(Sender: TObject);
private
 { private declarations }
public
 { public declarations }
end;
var
 Form1: TForm1;
implementation
{ TForm1 }
procedure TForm1.Button1Click(Sender: TObject);
var t: array [1..5] of TPoint;
begin
 Form1.Canvas.LineTo(500,500);//Рисование линии
//Изменение цвета и толщины линии.
 Form1.Canvas.Pen.Color:= clMaroon;
 Form1.Canvas.Pen.Width:= 5;
//Рисование прямоугольника.
 Form1.Canvas.Rectangle(50,50,200,200);
 Form1.Canvas.Pen.Color:= clolive;
```

```
    //Рисование эллипса.
     Form1.Canvas.Ellipse(50,50,100,200);
//Рисование прямоугольника с скругленными углами.
     Form1.Canvas.RoundRect(250,250,400,400,30,30);
//Рисование сектора окружности.
Form1.Canvas.Pie(300,300,400,400,350,350,500,500);
//Массив координат вершин пятиугольника.
     t[1].x:=250; t[1].y:=10;
     t[2].x:=350; t[2].y:=15;
     t[3].x:=400; t[3].y:=50;
     t[4].x:=300; t[4].y:=150;
     t[5].x:=250; t[5].y:=100;
     Form1.Canvas.Polygon (t) ;
    Form1.Canvas.TextOut(10,10,'это вывели текст');
    end;
    initialization
     {$I unit1.lrs}
    end.
```

Функция

```
procedure TextOut(X,Y:Integer;const Text:String);
```

пишет строку текста `Text`, начиная с позиции с координатами `(X, Y)`. Текущая позиция `PenPos` пера `Pen` перемещается на конец выведенного текста. Надпись выводится в соответствии с текущими установками шрифта `Font`, фон надписи определяется установками текущей кисти. Для выравнивания позиции текста на канве можно использовать методы, позволяющие определить высоту и длину текста в пикселях — `TextExtent`, `TextHeight` и `TextWidth`. Рассмотрим эти функции.

Функция

```
function TextExtent (const Text : String): Tsize;
```

возвращает структуру типа `Tsize`, содержащую длину и высоту в пикселях текста `Text`, который предполагается написать на канве текущим шрифтом.

```
    type Tsize = record
     cx:Longint; cy:Longint;
    end;
```

Функция

```
function TextHeight(const Text:String):Integer;
```
возвращает высоту в пикселях текста `Text`, который предполагается написать на канве текущим шрифтом.

Функция
```
function TextWidth(const Text:String):Integer;
```
возвращает длину в пикселях текста `Text`, который предполагается написать на канве текущим шрифтом. Это позволяет перед выводом текста на канву определить размер надписи и расположить ее и другие элементы изображения наилучшим образом.

Если цвет кисти в момент вывода текста отличается от того, которым закрашена канва, то текст будет выведен в цветной прямоугольной рамке, но ее размеры будут точно равны размерам надписи.

Мы рассмотрели основные функции рисования. Прежде чем перейти непосредственно к рисованию, обратите внимание на то, что если вы свернете окно с графикой, а затем его восстановите, то картинка на форме исчезнет. Изменение размеров окна также может испортить графическое изображение в окне. Для решения этой проблемы существуют процедуры обработки событий `Объект.FormPaint` и `Объект.FormResize`.

Процедура `Объект.FormPaint` выполняется после появления формы на экране.

Процедура `Объект.FormResize` выполняется после изменения размера формы.

Следовательно, для того чтобы не пропадала картинка, все операторы рисования нужно помещать внутрь `Объект.FormPaint` и дублировать в процедуре `Объект.FormResize`.

10.2 Построение графиков

Алгоритм построения графика непрерывной функции $y=f(x)$ на отрезке $[a;b]$ состоит в следующем: необходимо построить точки $(x_i, f(x_i))$ в декартовой системе координат и соединить их прямым линиями. Координаты точек определяются по следующим формулам:

$hx=(b-a)/N,$

где N — количество отрезков на отрезке $[a;b]$.

$x_i=a+(i-1)hx; \quad y_i=f(x_i), \quad$ где $i=0,N$

Чем больше точек будет изображено, тем более плавным будет построенный график.

При переносе этого алгоритма на форму или другой компонент Lazarus учитывает размеры и особенности компонента (ось `OX` направлена слева направо, ее координаты лежат в пределах от 0 до `Width`; ось `OY` направлена вниз, ее координаты находятся в пределах от 0 до `Height`). Значения координат *X* и *Y* должны быть целыми.

Необходимо пересчитать все точки из «бумажной» системы координат (*X* изменяется в пределах от a до b, *Y* изменяется от минимального до максимального значения функции) в «компонентную[79]» (в этой системе координат ось абсцисс обозначим буквой U, `0` ≤ U ≤ `Wigth`, а ось ординат — буквой *V*, `0` ≤ `V` ≤ `Height`).

Для преобразования координаты *X* в координату *U* построим линейную функцию *cX+d*, которая переведет точки из интервала *(a;b)* в точки интервала (`X0,Width-Xk`)[80]. Поскольку точка *a* «бумажной» системы координат перейдет в точку *X0* «экранной», а точка *b* — в точку `Width-Xk`, то система линейных уравнений для нахождения коэффициентов *c* и *d* имеет вид:

$$\begin{cases} c \cdot a + d = X0 \\ c \cdot b + d = Width - Xk \end{cases}$$

Решив ее, найдем коэффициенты *c, d*:

$$\begin{cases} d = X0 - c \cdot a \\ c = \dfrac{Width - Xk - X0}{b - a} \end{cases}$$

Для преобразования координаты *Y* в координату *V* построим линейную функцию *V=gY+h.* Точка `min` «бумажной» системы координат перейдет в точку `Height-YK` «компонентную», а точка `max` – в точку *Y0.* Для нахождения коэффициентов *g* и *h* решим систему линейных алгебраических уравнений:

$$\begin{cases} g \cdot min + h = Heignt - Yk \\ g \cdot max + h = Y0 \end{cases}$$

Ее решение позволит найти нам коэффициенты *g* и *h*

$$\begin{cases} h = Y0 - g \cdot max \\ g = \dfrac{Heigth - Yk - Y0}{min - max} \end{cases}$$

Перед описанием алгоритма построения графика давайте

79 Система координат, связанная с конкретным компонентом класса Tform, Timage, Tprinter и т.д.
80 X0, XK, Y0, YK — Отступы от левой, правой, нижней и верхней границы компонента.

уточним формулы для расчета коэффициентов *c, d, g* и *h*. Дело в том, что `Width` – это ширина компонента с учетом рамок слева и справа, а `Height` – полная высота компонента с учетом рамки, а если в качестве компонента будет использоваться форма, то необходимо учесть заголовок окна. Однако, для изображения графика нам нужны вертикальные и горизонтальные размеры компонента без учета рамок и заголовка. Эти размеры хранятся в свойствах `ClientWidth` (ширина клиентской области компонента без учета ширины рамки) и `ClientHeight` (высота клиентской области компонента без учета ширины рамки и ширины заголовка) компонента. Поэтому для расчета коэффициентов *c, d, g* и *h* логичнее использовать следующие формулы:

$$\begin{cases} d = X0 - c \cdot a \\ c = \dfrac{ClientWidth - Xk - X0}{b - a} \end{cases} \qquad (10.1)$$

$$\begin{cases} h = Y0 - g \cdot max \\ c = \dfrac{ClientHeigth - Yk - Y0}{min - max} \end{cases} \qquad (10.2)$$

Алгоритм построения графика на экране дисплея можно разделить на следующие этапы:

1. Определить число отрезков *N*, шаг изменения переменной *X*.
2. Сформировать массивы *X, Y*, вычислить максимальное (`max`) и минимальное (`min`) значения Y.
3. Найти коэффициенты *c, d, g* и *h* по формулам (10.1), (10.2).
4. Создать массивы $U_i = cX_i + d$, $V_i = gY_i + h$.
5. Последовательно соединить соседние точки прямыми линиями с помощью функции `LineTo`.
6. Изобразить систему координат, линий сетки и подписи.

При построении графиков нескольких непрерывных функций вместо массивов *Y* и *V* рационально использовать матрицы размером `k×n`, где `k` – количество функций. Элементы каждой строки матриц являются координатами соответствующего графика в «бумажной» (*Y*) и «компонентной» (*V*) системе координат.

Блок-схема алгоритма изображения графика показана на рис.10.3. Блок-схема построения графиков `k` непрерывных функций представлена на рис. 10.4.

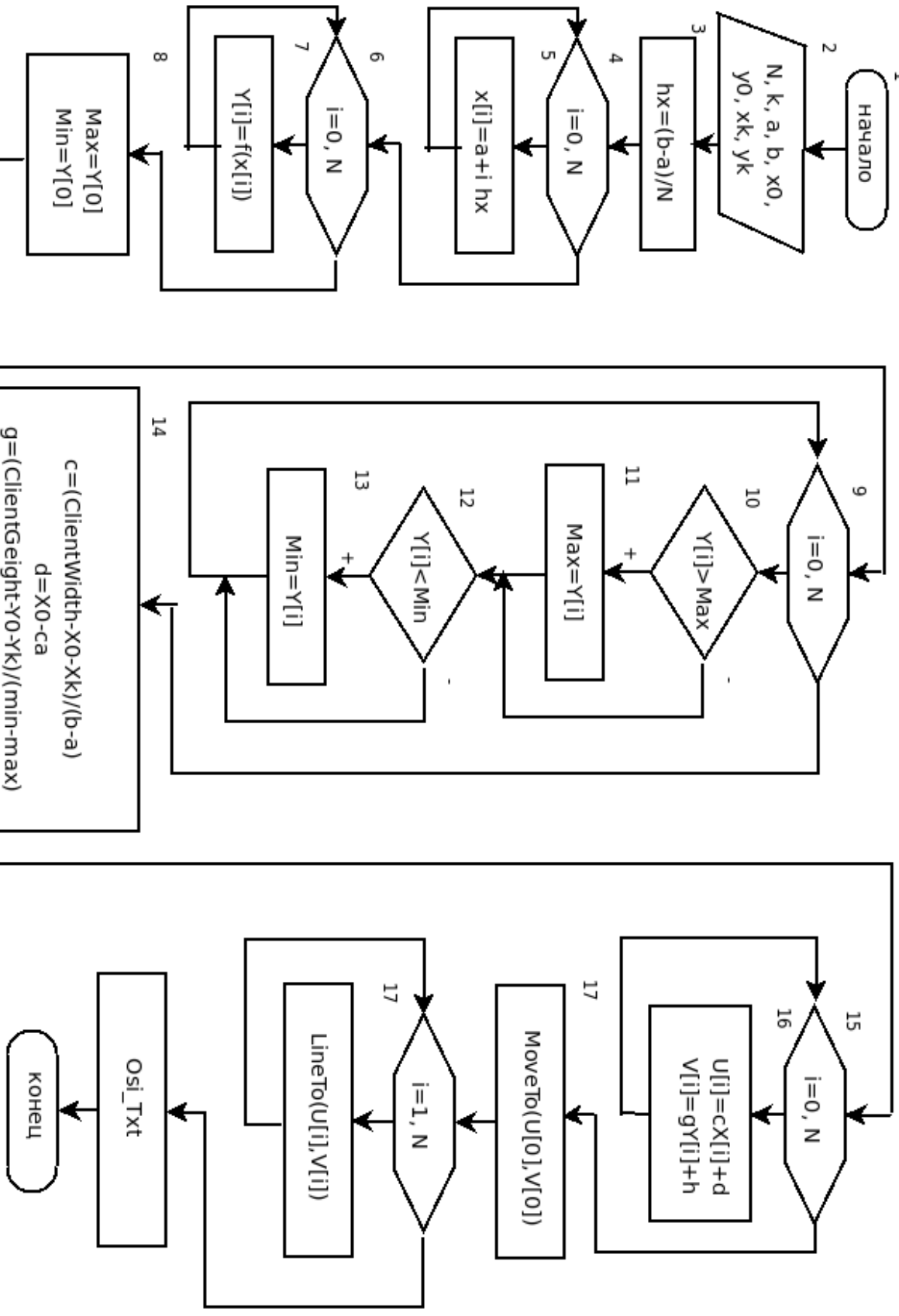

Рисунок 10.3: Блок-схема алгоритма построения графика функции

Рассмотрим поэтапно каждую блок-схему.

Для рис. 10.3 первый этап алгоритма представлен в блоках 2 и 3. Второй этап реализуется в блоках 4 – 13. Коэффициенты третьего этапа рассчитываются в блоке 14. В блоках 15 – 16 формируются массивы значений U и V «компонентной системы координат (этап 4). Блоки 17 – 19 – это вывод графиков на экран. И блок 20 предназначен для шестого этапа.

Для второй схемы рис. 10.4 реализация второго этапа представлена в блоках 4 – 15. В блоке 16 рассчитываются коэффициенты *c, d, g* и *h*. В блоках 17 – 20 формируются массивы четвертого этапа. Блоки 21 – 24 предназначены для вывода графиков, а блок 25 – для построения осей.

ЗАДАЧА 10.1. Построить график функции *f(x)* на интервале *[a, b]*. Функция задана следующим образом:

$$f(x)=\begin{cases}\sin\dfrac{x}{2}, \text{если } x\leqslant 0\\ \sqrt{\dfrac{1+x}{3}}, \text{если } x>0\end{cases}$$

Создадим новый проект, изменим высоту и ширину формы до размеров, достаточных для отображения на ней графика. Например, можно установить следующие свойства: `Width` – 800, `Height` – 700. Разместим на форме кнопку и компонент класса `TImage`. Объект `TImage1` – это растровая картинка, которая будет использоваться для отображения графика после щелчка по кнопке `Button1`. Размеры растровой картинки сделаем чуть меньше размеров формы.

Установим в качестве свойства `Caption` формы сроку «График функции». Чтобы избавиться от проблемы перерисовки будущего графика при изменении размера формы, запретим изменение формы и уберем кнопки минимизации и максимизации окна. Свойство формы `BorderStyle` определяет внешний види поведение рамки вокруг окна формы. Для запрета изменения формы установим значение свойства `BorderStyle` в `bsSingle`, это значение определяет стандартную рамку вокруг формы и запрещает изменение размера формы. Чтобы убрать кнопки минимизации и максимизации формы, установим ее свойства `BolderIcons.BiMaximize` и `BolderIcons.BiMinimize` в `False`.

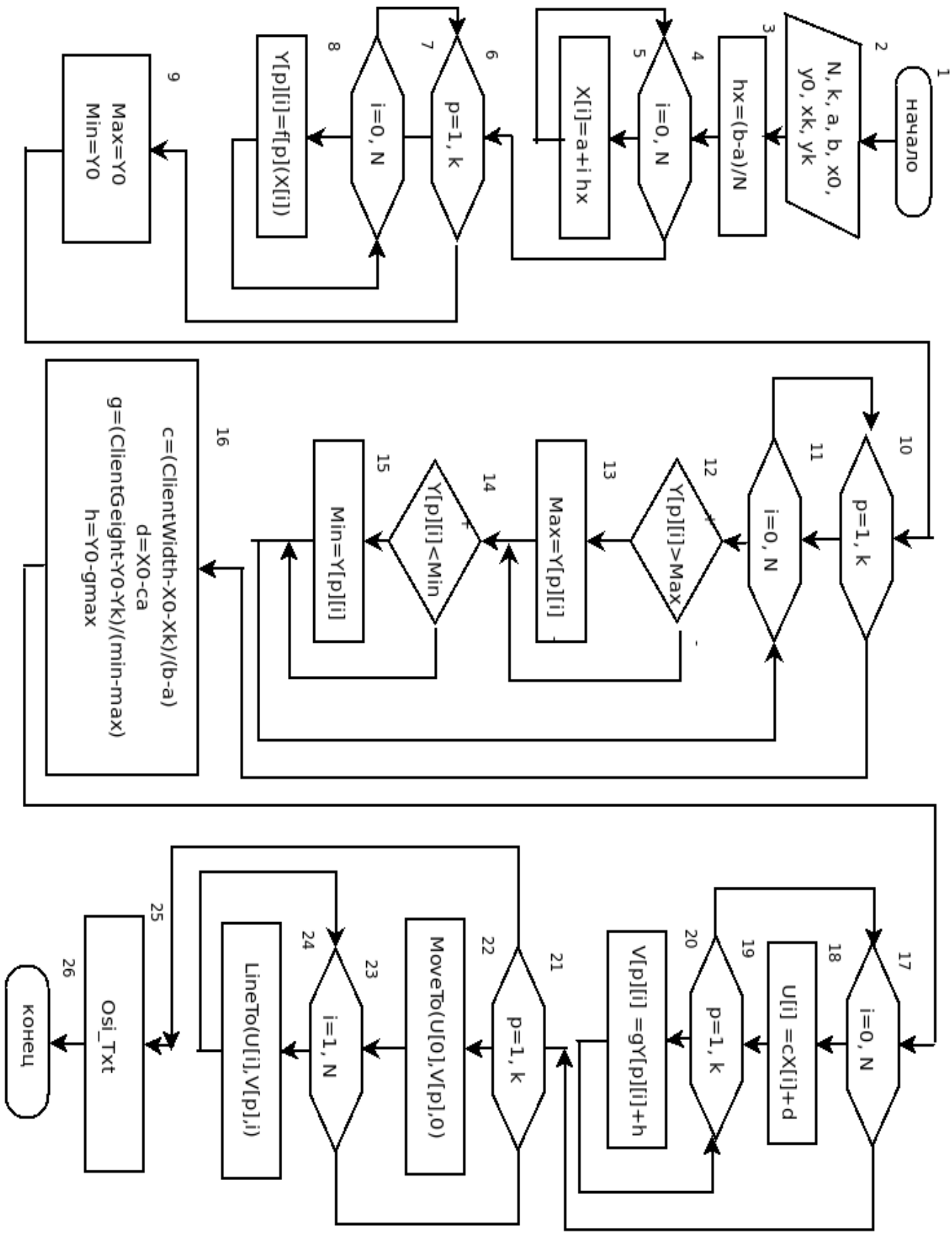

Рисунок 10.4: Блок-схема алгоритма построения графиков нескольких функций

Для кнопки установим свойство `Caption` – фразу «График».

После установки всех описанных свойств окно формы должно стать подобным представленному на рис.10.5.

Рисунок 10.5: Окно формы после установки свойств

Ввод интервала *a* и *b* сделаем с помощью запроса в момент создания формы (в методе инициализации формы `FormCreate`). Ниже приведен листинг модуля проекта рисования графика с комментариями:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
    Classes,    SysUtils,    LResources,    Forms,
Controls, Graphics, Dialogs, ExtCtrls, StdCtrls;
//Функция, которая аналитически определяет график.
function f(x:real):real;
//Функция, которая изображает график.
procedure Graphika(a, b: real);
type
    { TForm1 }
    TForm1 = class(TForm)
```

```
        Button1: TButton;
        Image1: TImage;
        procedure Button1Click(Sender: TObject);
        procedure FormCreate(Sender: TObject);
    private
        { private declarations }
    public
        { public declarations }
    end;
var
    Form1: TForm1;
    //Объявление переменных:
    //x0, xk, y0, yk - отступы от границ
    //компонента слева, справа, сверху и снизу;
    //x, y - массивы, определяющие координаты
    //точек графика в "бумажной"
    //системе координат;
    //u, v - массивы, определяющие координаты
    //точек графика в "компонентной"
    //системе координат;
    //N - количество точек.
    x0, y0, xk, yk, a, b :real;
    x, y : array [0..1000] of real;
    u, v : array [0..1000] of integer;
    N : integer;
implementation
   // функция, которая будет изображена
   //на компоненте Image1
function f (x : real) : real;
begin
    if x<=0 then Result:=sin(x/2)
    else Result:=sqrt((1+x)/3);
end;
//функция, которая рисует график
//заданной функции на компоненте Image1.
procedure Graphika(a, b: real);
//Kx+1 - количество линий сетки,
//перпендикулярных оси OX.
```

```
//Ky+1 - количество линий сетки,
//перпендикулярных оси OY.
const Kx=5;Ky=5;
var dx, dy, c, d, g, h, max, min :real;
    i, tempx, tempy :integer; s : string;
begin
//Вычисление шага изменения по оси X.
    h:=(b-a)/(N-1);
//Формирование массивов x и y.
    x[0]:=a;
    y[0]:=f(x[0]);
    for i:=1 to N do
    begin
        x[i]:=x[i-1]+h;
        y[i]:=f(x[i]);
    end;
//Вычисление максимального
//и минимального значений массива Y.
    max:=y[0]; min:=y[0];
    for i:=1 to N do
    begin
        if y[i]>max then max:=y[i];
            if y[i]<min then min:=y[i];
    end;
//формирование коэффициентов пересчета
//из "бумажной" в "компонентную"
//систему координат.
    c:=(Form1.Image1.ClientWidth-x0-xk)/(b-a);
    d:=x0-c*x[0];
    g:=(Form1.Image1.ClientHeight-y0-yk)/
                                        (min-max);
    h:=yk-g*max;
//Формирование массивов точек
//в экранной системе координат.
    for i:=0 to N do
    begin
        u[i]:=trunc(c*x[i]+d);
        v[i]:=trunc(g*y[i]+h);
```

```
    end;
    Form1.Image1.Canvas.Color:= clGray;
    Form1.Image1.Canvas .Pen .Mode:= pmNot;
//Рисование графика функции на компоненте Image1.
    Form1.Image1.Canvas.MoveTo(u[0],v[0]);
    Form1.Image1.Canvas.Pen.Width:= 2;
    Form1.Image1.Canvas.Pen.Color:= clGreen;
    for i:=1 to N do
        Form1.Image1.Canvas.LineTo(u[i],v[i]);
    Form1.Image1.Canvas.Pen.Width:= 1 ;
    Form1.Image1.Canvas.Pen.Color:= clBlack;
//Рисование осей координат, если они попадают
//в область графика.
Form1.Image1.Canvas.MoveTo(trunc(x0),trunc(h));
    if (trunc(h)>yk) and
    (trunc(h)<trunc(Form1.Image1.ClientHeight-y0))
    then
Form1.Image1.Canvas.LineTo(trunc
        (Form1.Image1.ClientWidth -xk),trunc(h));
Form1.Image1.Canvas.MoveTo(trunc(d),trunc(yk));
    if (trunc(d)>x0) and
    (trunc(d)<trunc(Form1.Image1.ClientWidth-xk))
    then
   Form1.Image1.Canvas.LineTo(trunc(d),
           trunc(Form1.Image1.ClientHeight -y0));
//Рисование линий сетки,
//вычисление расстояния между линиями сетки в
//"компонентной" системе координат,
//перпендикулярных оси OX
    dx:=(Form1.Image1.ClientWidth -x0-xk)/KX;
//Выбираем тип линии для линий сетки,
    for i:=0 to KX do
    begin
//первую и последнюю линии сетки рисуем обычной
//сплошной линией,
        if (i=0) or (i=KX) then
           Form1.Image1.Canvas.Pen.Style:=psSolid
//остальные - рисуем пунктирными линиями.
```

```
        else
            Form1.Image1.Canvas.Pen.Style:=psDash;
//Рисование линии сетки, перпендикулярной оси Ox.
Form1.Image1.Canvas.MoveTo(trunc(x0+i*dx),
                                    trunc(yk));
Form1.Image1.Canvas.LineTo(trunc(x0+i*dx),
            trunc(Form1.Image1.ClientHeight -y0));
    end;
//Вычисление расстояния между линиями сетки,
//перпендикулярными оси OY,
//в "компонентной" системе координат.
    dy:=(Form1.Image1.ClientHeight  -y0-yk)/KY;
    for i:=0 to KY do
    begin
//Первую и последнюю линии сетки
//рисуем обычной сплошной линией,
    if (i=0) or (i=KY) then
        Form1.Image1.Canvas.Pen.Style:=psSolid
//остальные - рисуем пунктирными линиями.
    else
        Form1.Image1.Canvas.Pen.Style:=psDash;
//Рисование линии сетки, перпендикулярной оси OY.
    Form1.Image1.Canvas.MoveTo(trunc(x0),
                                  trunc(yk+i*dy));
Form1.Image1.Canvas.LineTo(trunc
    (Form1.Image1.ClientWidth-xk),trunc(yk+i*dy));
    end;
    Form1.Image1.Canvas.Pen.Style:=psSolid;
//Вывод подписей под осями,
//определяем dx - расстояние между выводимыми
//под осью OX значениями.
    dx:=(b-a)/KX;
    tempy:=
        trunc(Form1.Image1.ClientHeight -y0+10);
    for i:=0 to KX do
    begin
//Преобразование выводимого значения в строку.
        Str(a+i*dx:5:2,s);
```

```
//Вычисление x-координаты выводимого
//под осью OX значения в "компонентной" системе.
    tempx:=trunc(x0+i*(Form1.Image1.ClientWidth -
                                    x0-xk)/KX)-10;
//Вывод значения под осью OX
    Form1.Image1.Canvas.TextOut(tempx,tempy,s);
    end;
    if (trunc(d)>x0) and
     (trunc(d)<Form1.Image1.ClientWidth-xk) then
Form1.Image1.Canvas.TextOut(trunc(d)-5,tempy,'0');
//Определяем dy - расстояние между
//выводимыми левее оси OY значениями.
    dy:=(max-min)/KY;
    tempx:=5;
    for i:=0 to KY do
    begin
//Преобразование выводимого значения в строку.
        Str(max-i*dy:5:2,s);
//Вычисление y-координаты выводимого левее оси OY
//значения в "компонентной" системе.
        tempy:=trunc(yk-5+
        i*(Form1.Image1.ClientHeight- y0-yk)/KY);
//Вывод значения левее оси OY.

    Form1.Image1.Canvas.TextOut(tempx,tempy,s);
    end;
    if (trunc(h)>yk) and (trunc(h)<
        Form1.Image1.ClientHeight-y0) then
        Form1.Image1.Canvas.TextOut(tempx+10,
                                    runc(h)-5,'0');
    tempx:=trunc(x0+i*(Form1.Image1.ClientWidth -
                                    x0-xk)/2);
    Form1.Image1.Canvas.TextOut(tempx,10,
                                'График функции');
end;
{ TForm1 }
procedure TForm1.FormCreate(Sender: TObject);
var s : string; kod : integer;
```

```
begin
    N:=100;
    x0:=40;
    xk:=40;
    y0:=40;
    yk:=40;
    repeat
        s:=InputBox('График непрерывной функции',
                    'Введите левую границу','-10');
        Val(s,a,kod);
    until kod=0;
    repeat
        s:=InputBox('График непрерывной функции',
                    'Введите правую границу','10');
        Val(s,b,kod);
    until kod=0;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
    Graphika(a, b);
end;
initialization
   {$I unit1.lrs}
end.
```

При запуске проекта появится запрос ввода левой (рис. 10.6) и правой (рис. 10.7) границ интервала. После этого появится окно формы с кнопкой `График`. После щелчка по кнопке на форме прорисуется график функции (рис. 10.8).

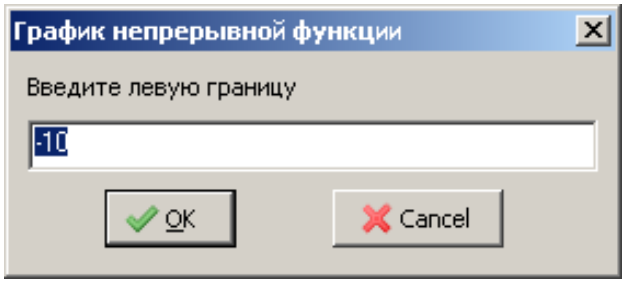

Рисунок 10.6: Окно ввода левой границы

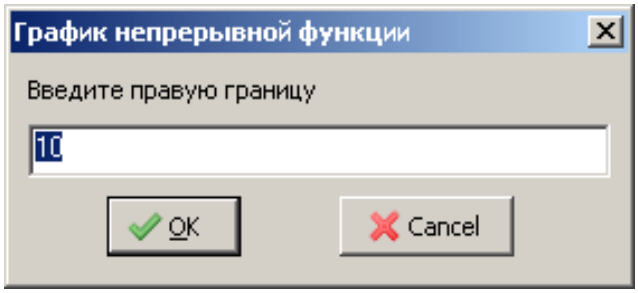

Рисунок 10.7: Окно ввода правой границы интервала

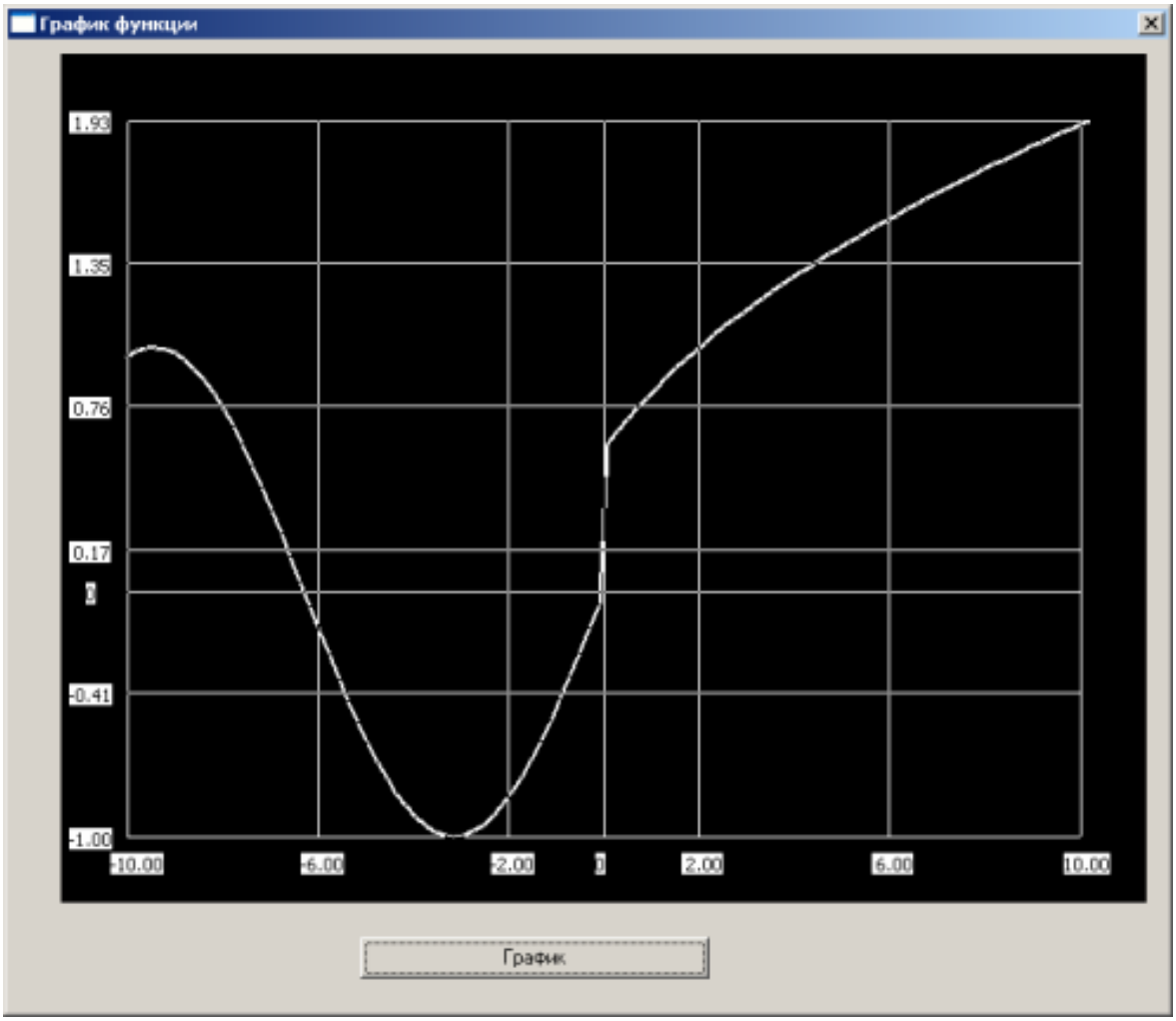

Рисунок 10.8: График функции

10.3 Задачи для самостоятельного решения

Построить график функции *f(x)* на интервале *[a;b]*. Функции заданы в табл. 10.5.

Таблица 10.5. Варианты заданий

№ варианта	f(x)	№ варианта	f(x)
1	$\begin{cases} \sin\dfrac{x^3+2}{3}, если\ x \leqslant 5 \\ \dfrac{\sqrt[5]{1+x}}{3}, если\ x > 5 \end{cases}$	9	$\begin{cases} \dfrac{x^2-x+2}{x^3}, если\ x \geq 1 \\ \sqrt[5]{\dfrac{e^x}{2}+x^2}, если\ x < 1 \end{cases}$
2	$\begin{cases} \tan(\pi+x), если\ x \leqslant -1 \\ \sqrt[3]{e^{x+1}}, если\ x > -1 \end{cases}$	10	$\begin{cases} \sin\dfrac{x+3,6}{x^3}, если\ x < 0 \\ \sqrt[3]{1+x}, если\ x \geq 0 \end{cases}$

№ варианта	f(x)	№ варианта	f(x)
3	$\begin{cases}\log(\frac{1+x}{3}), если\ x>2\\ \sqrt[3]{1+x^3}, если\ x\le 2\end{cases}$	11	$\begin{cases}\frac{x^3+3x^2}{3}, если\ x\leqslant -2\\ \sqrt[3]{\log(1{,}5+x^2)}, если\ x>-2\end{cases}$
4	$\begin{cases}\frac{\sqrt[2]{\cos(x+2)^3}}{3{,}5}, если\ x>2\\ \sin(x+2)^2, если\ x\le 2\end{cases}$	12	$\begin{cases}e^{\frac{x}{4}}, если\ x>-2{,}5\\ \sqrt[5]{x^2}, если\ x\le -2{,}5\end{cases}$
5	$\begin{cases}\cos\frac{x-6}{x-3}, если\ x>5\\ \sqrt{1+x^4}, если\ x\le 5\end{cases}$	13	$\begin{cases}\cos\frac{x+2}{x^3}, если\ x\ge 4\\ \sqrt[3]{e^x+x^2}, если\ x<4\end{cases}$
6	$\begin{cases}x^3+3x^2, если\ x\leqslant -1\\ \sqrt[3]{\ln(10{,}5+x)}, если\ x>-1\end{cases}$	14	$\begin{cases}x^2-x+2, если\ x\ge -1\\ \sqrt[5]{e^x+7}, если\ x<-1\end{cases}$
7	$\begin{cases}\cos\frac{x^2+2}{x}, если\ x>3\\ \sqrt{12+x^2}, если\ x\le 3\end{cases}$	15	$\begin{cases}\log(\frac{1+x^3}{2}), если\ x>2{,}5\\ \sqrt[3]{1+2x^2-x^3}, если\ x\le 2{,}5\end{cases}$
8	$\begin{cases}x^3\cdot\sin(x), если\ x>-3{,}5\\ \sqrt[5]{\left\|\frac{x}{2}\right\|}, если\ x\leqslant -3{,}5\end{cases}$	16	$\begin{cases}x^3+3(x+2)^2, если\ x\leqslant 3\\ \sqrt[5]{(\sin(10{,}5+x))^2}, если\ x>3\end{cases}$

Построить графики функций *f1(x), f2(x), f2(x)* в одной системе координат на интервале *[a;b]*. Функции заданы в табл. 10.6.

Таблица 10.6. Варианты заданий

№ варианта	f1(x)	f2(x)	f3(x)
17	$\cos\frac{x-6}{x^2+3}$	$x^3\cdot\sin(x)$	$\sqrt{(5x^3)}\cdot\sin(x^2)$
18	$1+2x^2-x^3$	$e^{\frac{x}{2}}+7$	$\sin\left(\frac{x}{3}\right)$
19	$\sqrt[3]{1+x^3}$	$14+2x^2-3x^3$	$\cos(\sin(x))$
20	$\sin\left(\frac{x}{3}+e^x\right)$	$\sqrt{7+2x^4}$	$\sqrt[3]{(3+5x^2-x^3)^2}$

№ варианта	f1(x)	f2(x)	f3(x)
21	$\sqrt[5]{7-x^3}$	$\cos\left(\frac{x}{2}+\pi\right)$	$\cos(\sin(x^2))$
22	$\sqrt[3]{\ln(12+x^2)}$	$e^{\frac{x}{5}}$	$\cos\left(\frac{x}{\pi}\right)$
23	$\cos\left(\frac{x+2}{\pi}\right)$	$\sqrt[4]{1+x^6+2x^2}$	$3x^4-5x^2+7x-2$
24	$\sqrt[3]{(1+x^2-x^3)^2}$	$2\cdot\sin\left(\frac{x}{2}+\pi\right)$	$5x^2-3x^3$
25	$\frac{x}{2}+\sin(2\cdot x\cdot\pi)$	$\sqrt[3]{(1+x)(x^3-4)^2}$	$e^{\frac{x}{7}}+4$

Вместо заключения

Перевернута последняя страница книги. Что теперь? Авторы надеются, что знакомство с языком Free Pascal будет только первым этапом в изучении программирования. Желание читателя что-то исправить в книге — переписать приведенные в книге программы, предложить более простые и быстро работающие алгоритмы, написать свои программы и модули — будет лучшей благодарностью авторам. Если у вас, читатели, появилось подобное желание, то мы выполнили свою задачу — научили вас основам программирования.

Следующим этапом в освоении программирования будет разработка ваших алгоритмов и написание реально работающих программ для различных операционных систем.

Алфавитный указатель

Литература

1. Алексеев Е.Р., Чеснокова О.В. Турбо Паскаль 7.0. - М.:НТ Пресс, 2006. - 320 с.:ил. - (Полная версия).

2. Алексеев Е.Р. Учимся программировать на Microsoft Visual C++ и Turbo C++ Explorer (под общей редакцией Чесноковой О.В.)/Алексеев Е.Р. - М.:НТ Пресс, 2007. - 352 с.:ил. - (Самоучитель).

3. Фаронов В.В. Delphi. Программирование на языке высокого уровня: Учебник для вузов. - Спб.:Питер, 2005.-640 с.:ил.

4. Чеснокова О.В. Delphi 2007. Алгоритмы и программы. Учимся программировать на Delphi 2007/Чеснокова О.В. Под общ. ред. Алексеева Е.Р.- М.:НТ Пресс, 2008. - 368 с.:ил.

5. Бронштейн И.Н., Семендяев К.А. Справочник по математике для инженеров и учащихся вузов.- М.:Наука, 1981. - 720 с.

6. *GNU Pascal — Википедия.* URL: .http://ru.wikipedia.org/wiki/GNU_Pascal (дата обращения 03.08.2009).

7. *Free Pascal - Advanced open source Pascal compiler for Pascal and Object Pascal - Home Page.* URL: http://www.freepascal.org (дата обращения 03.08.2009).

8. *GNU Pascal.* URL: http://www.gnu-pascal.de/gpc/h-index.html (дата обращения 03.08.2009).

9. Main Page/ru – Free Pascal wiki. URL: http://wiki.freepascal.org/Main_Page/ru (дата обращения 03.11.2009).

10. Free Pascal.ru - Информационный портал для разработчиков на Free Pascal & Lazarus & MSE. URL http://www.freepascal.ru (дата обращения: 03.11.2009).

11. Lazarus – News. URL: http://www.lazarus.freepascal.org (дата обращения 03.11.2009).

12. Lazarus – Википедия. URL: http://ru.wikipedia.org/wiki/Lazarus (дата обращения 03.11.2009).

ISBN 978-966-8248-26-9

А 47 **Алексеев Е.Р., Чеснокова О.В., Кучер Т.В.** Самоучитель по программированию на Free Pascal и Lazarus. - Донецк.: ДонНТУ, Технопарк ДонНТУ УНИТЕХ, 2009. - 503 с.

ISBN 978-966-8248-26-9 **ООО** «Технопарк ДонНТУ УНИТЕХ»

Издательство: ООО «Технопарк ДонНТУ УНИТЕХ»

Свидетельство о внесении субъекта издательского дела в государственный реестр издателей, изготовителей и распространителей издательской продукции: Дк №1017 от 21.08.2002.

83000, г. Донецк, ул. Артема, 58, к.1.311

Тел. : (062) 304 90 19

Технический редактор: Аноприенко А.Я.

Дизайн обложки: Сорокина Л.С.

Корректор: Молодых Э.В.

Подписано к печати 8.12.2009 г.
Формат 60x90 1/16
Усл.печ.л. 36.04
Печать лазерная.
Заказ №
Тираж 200 экз.

Отпечатано в типографии «Друк-Инфо».
Адрес: 83000, г. Донецк, ул. Артема, 58.
Тел.: (062) 335 64 55.