

Программирование на Free Pascal и Lazarus

Алексеев Евгений Ростиславович, Чеснокова Оксана Витальевна, Кучер Татьяна Викторовна

Содержание

- Лекция 1. [Введение. Средства разработки программ на языке Free Pascal](#)
- [О свободном программном обеспечении для образования](#)
- [Введение](#)
- [Средства разработки программ на языке Free Pascal](#)
- [1.1 Процесс разработки программы](#)
- [1.2 Среда программирования Free Pascal](#)
- [1.2.1 Работа в текстовом редакторе Free Pascal](#)
- [1.2.2 Запуск программы в среде Free Pascal и просмотр результатов](#)
- [1.3 Среда разработки Geany](#)
- [1.4 Среда визуального программирования Lazarus](#)
- [1.4.1 Установка Lazarus в ОС Linux](#)
- [1.4.2 Установка Lazarus под управлением ОС Windows](#)
- [1.4.3 Среда Lazarus](#)
- [1.4.4 Главное меню Lazarus](#)
- [1.4.5 Форма приложения](#)
- [1.4.6 Окно редактора Lazarus](#)
- [1.4.7 Панель компонентов](#)
- [1.4.8 Инспектор объектов](#)
- [1.4.9 Первая программа в Lazarus](#)
- [1.4.10 Полезная программа](#)
- [1.4.11 Консольное приложение среды Lazarus](#)
- [1.4.12 Операторы ввода — вывода данных](#)
- Лекция 2. [Общие сведения о языке программирования Free Pascal](#)
- [2.1 Структура проекта Lazarus](#)
- [2.2 Структура консольного приложения](#)
- [2.3 Элементы языка](#)
- [2.4 Данные в языке Free Pascal](#)
- [2.4.1 Символьный тип данных](#)
- [2.4.2 Целочисленный тип данных](#)
- [2.4.3 Вещественный тип данных](#)
- [2.4.4 Тип дата-время](#)
- [2.4.5 Логический тип данных](#)
- [2.4.6 Создание новых типов данных](#)
- [2.4.7 Перечислимый тип данных](#)
- [2.4.8 Интервальный тип](#)
- [2.4.9 Структурированные типы](#)
- [2.4.10 Указатели](#)
- [2.5 Операции и выражения](#)
- [2.5.1 Арифметические операции](#)
- [2.5.2 Операции отношения](#)
- [2.5.3 Логические операции](#)
- [2.5.4 Операции над указателями](#)
- [2.6 Стандартные функции](#)
- Лекция 3. [Операторы управления](#)
- [3.1 Основные конструкции алгоритма](#)
- [3.2 Оператор присваивания](#)
- [3.3 Составной оператор](#)
- [3.4 Условные операторы](#)
- [3.4.1 Условный оператор if..then..else](#)
- [3.4.2 Оператор варианта case](#)
- [3.4.3 Обработка ошибок. Вывод сообщений в среде Lazarus](#)
- [3.5 Операторы цикла](#)
- [3.5.1 Оператор цикла с предусловием while..do](#)
- [3.5.2 Оператор цикла с постусловием repeat..until](#)
- [3.5.3 Оператор цикла for..do](#)
- [3.5.4 Операторы передачи управления](#)
- [3.5.5 Решение задач с использованием циклов](#)
- [3.5.6 Ввод данных из диалогового окна в среде Lazarus](#)
- Лекция 4. [Подпрограммы](#)
- [4.1 Общие сведения о подпрограммах. Локальные и глобальные переменные](#)

- [4.2 Формальные и фактические параметры. Передача параметров в подпрограмму](#)
- [4.3 Процедуры](#)
- [4.4 Функции](#)
- [4.5 Решение задач с использованием подпрограмм](#)
- [4.6 Рекурсивные функции](#)
- [4.7 Особенности работы с подпрограммами](#)
- [4.7.1 Параметры-константы](#)
- [4.7.2 Процедурные типы](#)
- [4.8 Разработка модулей](#)
- Лекция 5. [Использование языка Free Pascal для обработки массивов](#)
 - [5.1 Общие сведения о массивах](#)
 - [5.2 Описание массивов](#)
 - [5.3 Операции над массивами](#)
 - [5.4 Ввод-вывод элементов массива](#)
 - [5.4.1 Организация ввода-вывода в консольных приложениях](#)
 - [5.4.2 Организация ввода-вывода в визуальных приложениях](#)
 - [5.5 Вычисление суммы и произведения элементов массива](#)
 - [5.6 Поиск максимального элемента в массиве и его номера](#)
 - [5.7 Сортировка элементов в массиве](#)
 - [5.7.1 Сортировка методом "пузырька"](#)
 - [5.7.2 Сортировка выбором](#)
 - [5.8 Удаление элемента из массива](#)
 - [5.9 Вставка элемента в массив](#)
 - [5.10 Использование подпрограмм для работы с массивами](#)
 - [5.11 Использование указателей для работы с динамическими массивами](#)
 - [5.11.1 Работа с динамическими переменными и указателями](#)
 - [5.11.2 Работа с динамическими массивами с помощью процедур `getmem` и `freemem`](#)
 - [5.12 Примеры программ](#)
- Лекция 6. [Обработка матриц в Паскале](#)
 - [6.1 Ввод-вывод матриц](#)
 - [6.2 Алгоритмы и программы работы с матрицами](#)
 - [6.3 Динамические матрицы](#)
- Лекция 7. [Обработка файлов средствами Free Pascal](#)
 - [7.1 Типы файлов в Free Pascal](#)
 - [7.2 Работа с типизированными файлами](#)
 - [7.2.1 Процедура `AssignFile`](#)
 - [7.2.2 Процедуры `reset`, `rewrite`](#)
 - [7.2.3 Процедура `CloseFile`](#)
 - [7.2.4 Процедура `rename`](#)
 - [7.2.5 Процедура `erase`](#)
 - [7.2.6 Функция `eof`](#)
 - [7.2.7 Чтение и запись данных в файл](#)
 - [7.2.8 Функция `filesize`](#)
 - [7.2.9 Функция `filepos`](#)
 - [7.2.10 Процедура `seek`](#)
 - [7.2.11 Процедура `truncate`](#)
 - [7.3 Бестиповые файлы в языке Free Pascal](#)
 - [7.4 Обработка текстовых файлов в языке Free Pascal](#)
- Лекция 8. [Работа со строками и записями](#)
 - [8.1 Обработка текста](#)
 - [8.2 Работа с записями](#)
- Лекция 9. [Объектно-ориентированное программирование](#)
 - [9.1 Основные понятия](#)
 - [9.2 Инкапсуляция](#)
 - [9.3 Наследование и полиморфизм](#)
 - [9.4 Перегрузка операций](#)
- Лекция 10. [Графика в Lazarus](#)
 - [10.1 Средства рисования в Lazarus](#)
 - [10.2 Построение графиков](#)
 - [Вместо заключения](#)
 - [Дополнения](#)
 - [Литература](#)

Лекция 1. Введение. Средства разработки программ на языке Free Pascal

В этой главе мы начинаем знакомство с программированием на языке Free Pascal. Язык программирования Free Pascal ведёт своё начало от классического языка Pascal, который был разработан в конце 60-х годов XX века Никлаусом Виртом. Н. Вирт 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 расчёты, разрабатывать программы для проведения сложных инженерных и экономических вычислений.

О свободном программном обеспечении для образования

Эта книга — не просто учебник по программированию на давно уже ставшем традиционным для преподавания языке Паскаль. Это первое издание на русском языке, посвященное свободной реализации этого языка — Free Pascal и свободной же интегрированной среде разработки Lazarus. Как и любые свободные программы, компилятор Free Pascal compiler и среда Lazarus могут быть установлены на любое количество компьютеров бесплатно (без лицензионных отчислений), могут использоваться без ограничений, их исходные тексты полностью доступны для изучения и модификации. А это означает, что они идеально подходят для задач и условий обучения программированию, не накладывая на учащихся дополнительных расходов и давая им возможность максимально полно разобраться в предмете (вплоть до анализа исходных текстов компилятора).

Преимущества свободного программного обеспечения для образования были признаны в России на самом высоком уровне, и осенью 2007 года состоялся первый открытый конкурс на разработку и поставку пакета свободного программного обеспечения. Его победителем стала группа "Армада". Соисполнителем по этому конкурсу стала компания ALT Linux, работающая с 2001 года и являющаяся российским лидером в разработке свободного ПО и дистрибутивов на базе Linux.

В рамках проекта сотрудниками ALT Linux был разработан и протестирован [пакет свободного программного обеспечения \(ПСПО\)](#), который был внедрен более чем в 1000 школ трех pilotных регионов России — Республики Татарстан, Пермского края и Томской области. В этих регионах была отработана не только технология внедрения, но и технология организации техподдержки, что очень важно для успеха проекта.

Несмотря на то что pilotных регионов только три, любая школа России могла добровольно присоединиться к проекту и бесплатно получить комплект ПСПО. Для популяризации этой возможности и проекта в целом был проведен ряд семинаров в разных уголках страны. По итогам 2008 года, в проекте внедрения участвует 2151 школа.

В 2010 году компания ALT Linux выпустила новый комплект образовательных дистрибутивов, [Альт Линукс 5.0 Школьный](#), который является авторским развитием пакета свободного программного обеспечения (ПСПО) 2007—2008 годов.

Комплект включает в себя дистрибутивы операционной системы ALT Linux, содержащие все необходимые программы для подготовки уроков и обучения информатике, значительный набор программ по математике, физике, черчению, астрономии и другим предметам, обучающие и развивающие игры, стандартный набор офисных программ. Альт Линукс Школьный может быть установлен практически на любой компьютер, так как включает в себя версии, адаптированные под различные системные требования аппаратного обеспечения.

Дистрибутивы комплекта Альт Линукс 5.0 Школьный включают Free Pascal и Lazarus и идеально подходят для освоения материала книги и организации преподавания на основе языка Free Pascal.

Введение

Авторы книги давно хотели написать учебник по программированию, который был бы полезен пользователям различных операционных систем. Благодаря компании ALT Linux, мы попытались это сделать. В качестве языка программирования был выбран язык Free Pascal, который представляется нам ясным, логичным и гибким языком и приучает к хорошему стилю программирования. Свободно распространяемые компиляторы 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. В настоящее время существует множество подходов к изучению программирования. По мнению авторов, нельзя изучать программирование на каком-либо языке, не изучив методы разработки алгоритмов. Одним из наиболее наглядных методов составления алгоритмов является язык блок-схем. Об этом свидетельствует и многолетний опыт авторов преподавания программирования. 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, приведены примеры простейших 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 блок-схем алгоритмов и программ различной сложности. Авторы рекомендуют внимательно разобрать все примеры и выполнить упражнения этой главы, и только после этого приступить к изучению последующих глав книги.

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

Пятая и шестая главы посвящены изучению алгоритмов 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 ОС 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. Приведённые 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 рассмотрены графические возможности Lazarus, изложено подробное описание алгоритма 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 сайте Евгения Ростиславовича Алексеева — <http://www.teacher.dn-ua.com>. Авторы благодарят компанию ALT Linux и лично Кирилла Маслинского за возможность издать эту книгу. Авторы выражают благодарность своим родным за помощь и понимание. Алексеев Е. Р., Чеснокова О. В., Кучер Т. В. Донецк, январь 2009 г.

Средства разработки программ на языке Free Pascal

В этой главе мы начинаем знакомство с программированием на языке Free Pascal. Язык программирования Free Pascal ведёт своё начало от классического языка Pascal, который был разработан в конце 60-х годов XX века Никлаусом Виртом. Н. Вирт 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 расчёты, 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. В этом случае программа выдаёт не те результаты, которые ожидает разработчик или пользователь. Логические ошибки исправить сложнее, чем синтаксические, иногда для этого придётся переписывать отдельные участки программы, а иногда и перерабатывать весь алгоритм.
4. Тестирование программы. Тестирование программы — процесс выявления ошибок в работе программы.

Процессы отладки и тестирования сопровождаются неоднократным запуском программы на выполнение. Процесс запуска программы может быть осуществлён только после того, как введённая в компьютер программа на алгоритмическом языке Pascal¹ 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 перевод всей программы прошёл 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 компиляторы².

Рассмотрим основные этапы обработки компилятором программы на языке Pascal.

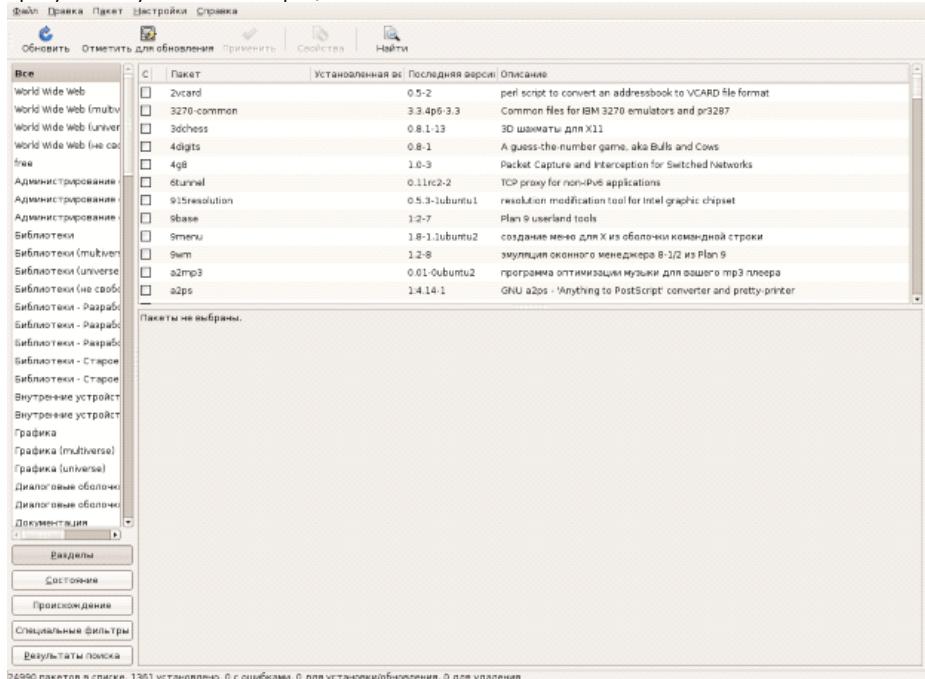
1. Компилятор анализирует, какие внешние библиотеки² нужно подключить, разбирает текст программы на составляющие элементы, проверяет синтаксические ошибки и в случае их отсутствия формирует объектный код (в Windows — файл с расширением .obj, в Linux — файл с расширением.o). Получаемый на этом этапе двоичный 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 программы можно запускать на выполнение.

На сегодняшний день существует множество компиляторов языка Pascal, среди которых можно выделить Borland Pascal, Delphi, а также свободно распространяемый кроссплатформенный компилятор языка Free Pascal и среду визуального программирования Lazarus.

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. В разных дистрибутивах Linux используются различные менеджеры пакетов, например, в ALT Linux можно воспользоваться программой Synaptic. Okno Synaptic представлено на [рис. 1.1](#). В школьной линейке дистрибутивов ALT Linux Free Pascal и Lazarus присутствуют

сразу после установки операционной системы.



увеличить изображение

Рис. 1.1. Менеджер пакетов Synaptic

Обратите внимание, что для установки программ необходимо установить список источников программ (список репозиториев⁴).

Для установки Free Pascal в окне Synaptic ([см. рис. 1.1](#)) необходимо щёлкнуть по кнопке **Найти** и в открывшемся окне ввести **fpc** ([см. рис. 1.2](#)). Менеджер программ находит программу FreePascal, после чего в окне Synaptic необходимо отметить программы fpc (Free Pascal Compiler Meta Package) для установки (с помощью контекстного меню или с помощью кнопки **Отметить для обновления**) и начать установку, щёлкнув по кнопке **Применить**. После этого начнётся процесс загрузки 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 **fpc**. На [рис. 1.3](#) представлено окно среды разработки программ на языке Free Pascal в ОС Linux.

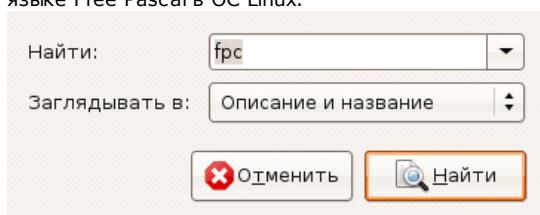


Рис. 1.2. Окно поиска компилятора Free Pascal в Synaptic

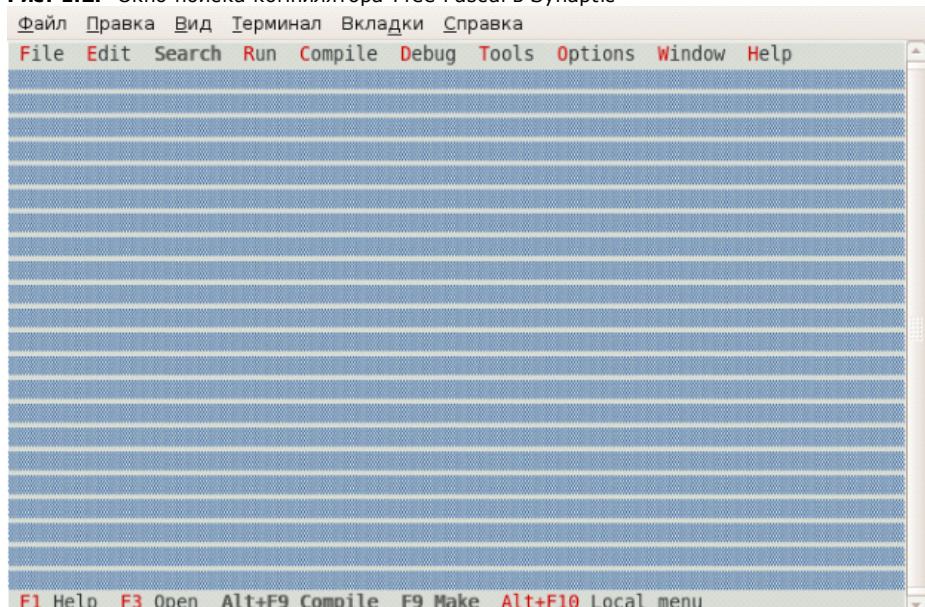


Рис. 1.3. Среда программирования Free Pascal в ОС Linux

Для установки Free Pascal в ОС Windows необходимо запустить скачанный со страницы [загрузки](#) инсталляционный файл. Первое диалоговое окно сообщит о начале процесса установки Free Pascal на компьютер. Для продолжения 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 С. Для выбора другого пути установки можно воспользоваться кнопкой **Browse...** Кроме того, в этом окне выводится информация о количестве свободного места на диске. В следующих четырёх 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 установки Free Pascal, нажав кнопку **Install**. Контролировать процесс установки можно с помощью линейного индикатора.

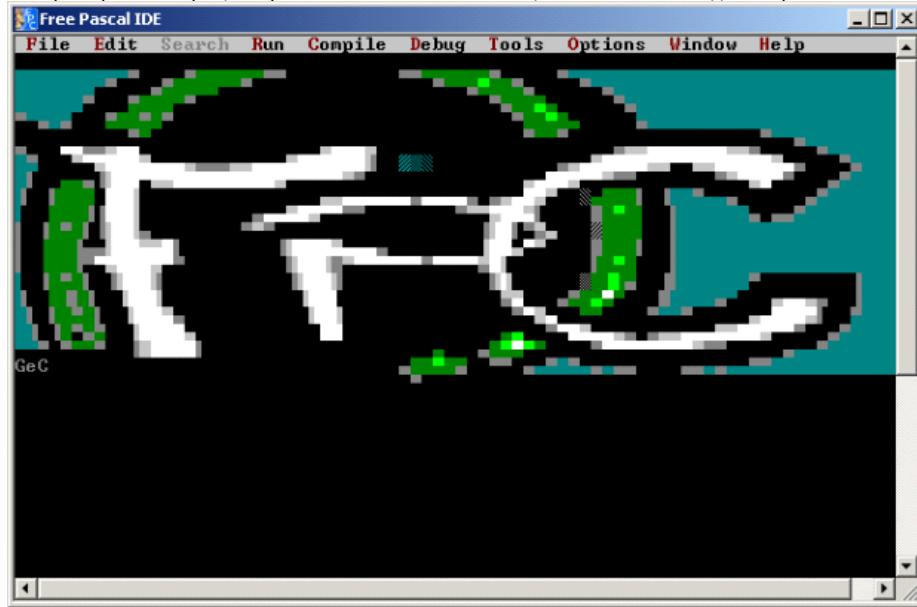


Рис. 1.4. Окно компилятора Free Pascal

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 терминале запускаем компилятор, после исправления синтаксических ошибок запускаем исполняемый файл. При такой технологии работы с компилятором, необходимо не забывать сохранять текст программы, иначе при запуске компилятора будет компилироваться старая версия текста программы.

Однако среда программирования позволяет значительно упростить процесс разработки программ. В состав среды программирования Free Pascal 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**) мы находимся в режиме редактирования, что подтверждается наличием 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. Выделить фрагмент текста можно с помощью клавиши **Shift** и клавиш перемещения курсора (стрелок).

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** применяются только к выделенным фрагментам текста. 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, таковы:

- **Ctrl+K+B** — пометить начало блока;
- **Ctrl+K+E** — пометить конец блока;
- **Ctrl+K+T** — пометить в качестве блока слово слева от курсора;
- **Ctrl+K+Y** — стереть блок;
- **Ctrl+K+C** — скопировать 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 **Ctrl+K+H** 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 **NONAME00.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 сохранён, 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. Переключаться между окнами можно двумя способами:

- 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 список активных окон, в котором будет необходимо выбрать нужное и нажать **Enter**.

1.2.2 Запуск программы в среде Free Pascal и просмотр результатов

После того как текст программы набран, его следует перевести в машинный код. Для этого необходимо вызвать транслятор с помощью команды **Compile** —

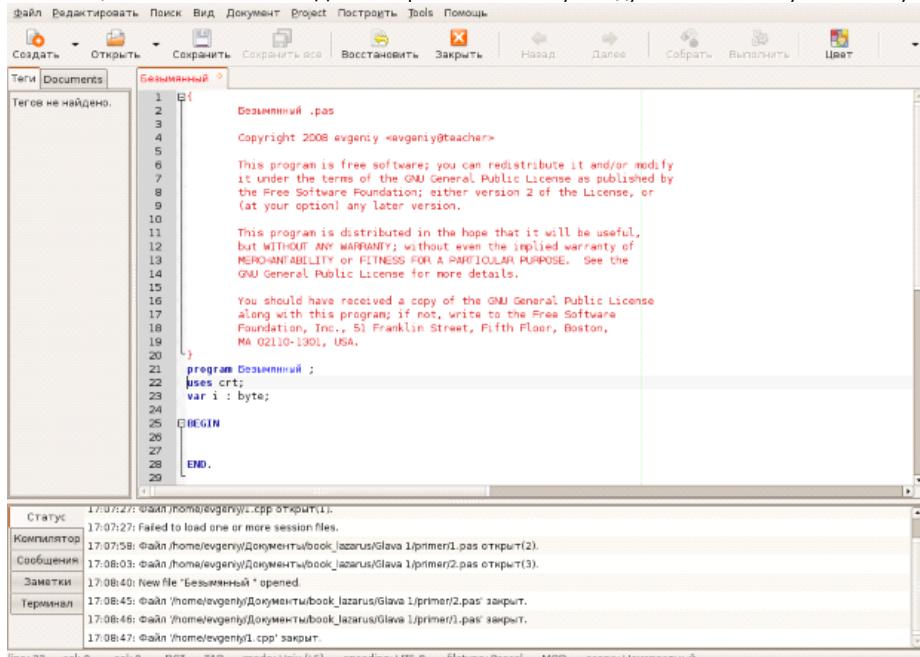
Compile (комбинация клавиш **Alt+F9**). На первом этапе транслятор проверяет наличие синтаксических ошибок. Если в программе нет синтаксических ошибок, то на экране сообщается о количестве строк транслированной программы и объёме доступной оперативной памяти.

Если на каком-либо этапе транслятор обнаружит ошибку, то в окне редактора курсор укажет ту строку программы, в которой ошибка была обнаружена. При этом в верхней строке редактора появится 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 вновь появляется 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.



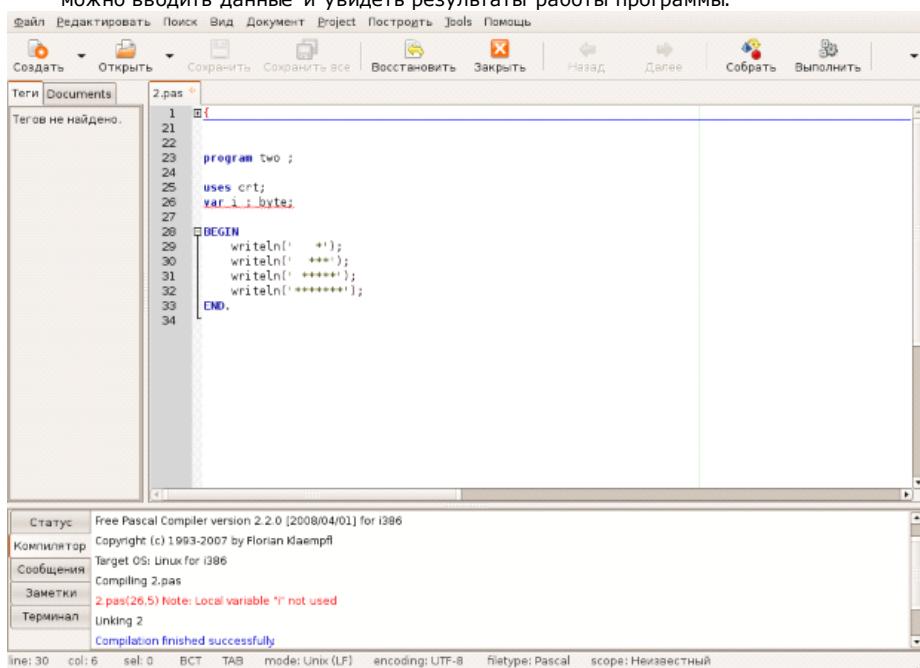
[увеличить изображение](#)

Рис. 1.5. Окно Geany с шаблоном программы на Free Pascal
Однако при использовании среды программирования 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 Среда разработки Geany

Среда разработки (её также иногда называют текстовым редактором) Geany есть в репозитории большинства современных дистрибутивов Linux. Разработка программ с использованием Geany довольно удобна. Geany можно установить стандартным для вашего дистрибутива способом, например, 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 (см. рис. 1.5), в котором необходимо ввести текст программы и сохранить его (см. 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 будет создан файл с объектным кодом программы и исполняемый файл. После компиляции программы ниже окна с программой будет представлен подробный отчёт (см. рис. 1.6) о результатах компиляции. Этот отчёт следует внимательно изучить, поскольку он позволяет быстро исправлять синтаксические ошибки. В сообщении об ошибке указана строка, где она найдена, и её краткое описание на английском языке. В отчётах по результатам компиляции могут быть ошибки (error) и сообщения (warning). Сообщения — это обнаруженные компилятором неточности, при которых, однако, возможно создание исполняемого кода программы.
3. Для запуска программы следует выполнить команду **Построить — Выполнить (F5)**. После чего на экране появится окно терминала (см. рис. 1.7), в котором можно вводить данные и увидеть результаты работы программы.



[увеличить изображение](#)

Рис. 1.6. Окно Geany с текстом программы на языке Free Pascal
В Geany можно настроить команды вызова компиляции, компоновки и запуска. Для этого 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.

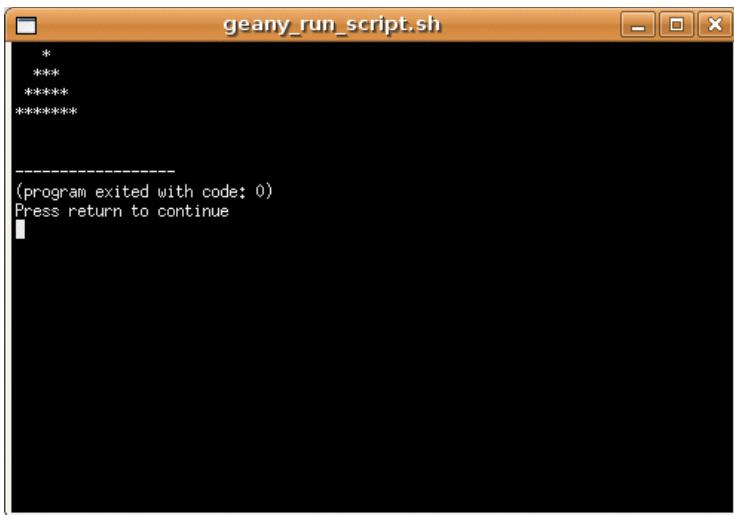


Рис. 1.7. Окно терминала с результатами работы программы

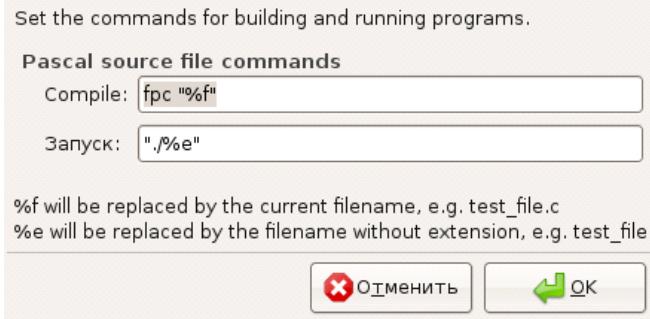


Рис. 1.8. Окно Установить включения и аргументы для Free Pascal

Выбор среды для разработки консольных программ на Free Pascal — дело пользователя. Авторы советуют читателю под управлением ОС Linux использовать Geany⁴¹, хотя для набора текста программы можно использовать и 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 интерфейсы⁴² будущей программы из специальных компонентов, реализующих нужные свойства. Количество таких компонентов достаточно велико. Каждый компонент содержит готовый программный код и все необходимые для работы данные, что избавляет программиста от создания того, что уже создано ранее. Подобный подход во много раз сокращает время написания программы. Кроме того, быстрота создания программного кода в Lazarus достигается за счёт того, что значительная часть текста формируется автоматически.

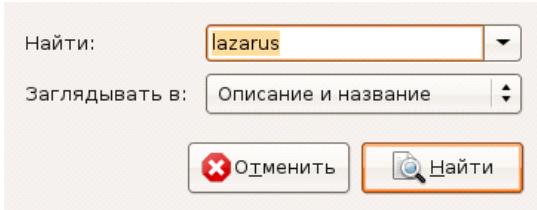


Рис. 1.9. Окно поиска пакета Lazarus для установки

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 щёлкнуть по кнопке Найти. В появившемся окне поиска (см. рис. 1.9) вводим имена необходимых программ (lazarus, fp, fpc, fpc-source) и щёлкаем по кнопке Найти. Менеджер программ находит программы Lazarus и FreePascal, после чего в окне Synaptic необходимо отметить программы Lazarus, fp, fpc, fpc-ide для установки (с помощью контекстного меню или с помощью кнопки Отметить для обновления) и начать установку, щёлкнув по кнопке Применить. После этого Synaptic предложит установить ещё несколько пакетов, которые необходимы для нормального функционирования Lazarus. С этим предложением нужно согласиться. После этого начнётся процесс загрузки файлов пакетов и установки Lazarus на компьютер. После установки запуск программы осуществляется с помощью команды меню Разработка — Lazarus⁴³.

Для установки Lazarus можно не использовать менеджер пакетов Synaptic, а самостоятельно загрузить все необходимые пакеты с сайта [проекта](#) и затем вручную их установить. Подробное описание процесса ручной установки можно найти в [Интернете](#).

1.4.2 Установка Lazarus под управлением ОС Windows

Рассмотрим особенности установки среды визуального программирования La-za-rus для операционной системы Windows. Перед установкой необходимо скачать установочный файл со страницы [загрузки](#). Установка Lazarus на компьютер осуществляется с помощью Мастера установки. Для того чтобы Мастер 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 **OK** приведёт к появлению следующего окна.

Использование кнопки **Отмена** в этом и во всех последующих окнах приведёт к прерыванию работы Мастера установки.

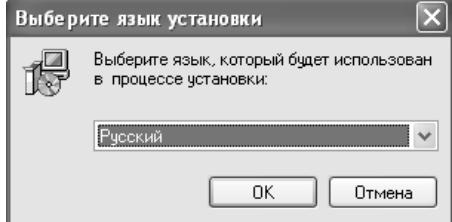


Рис. 1.10. Окно Мастера установки. Выбор языка.

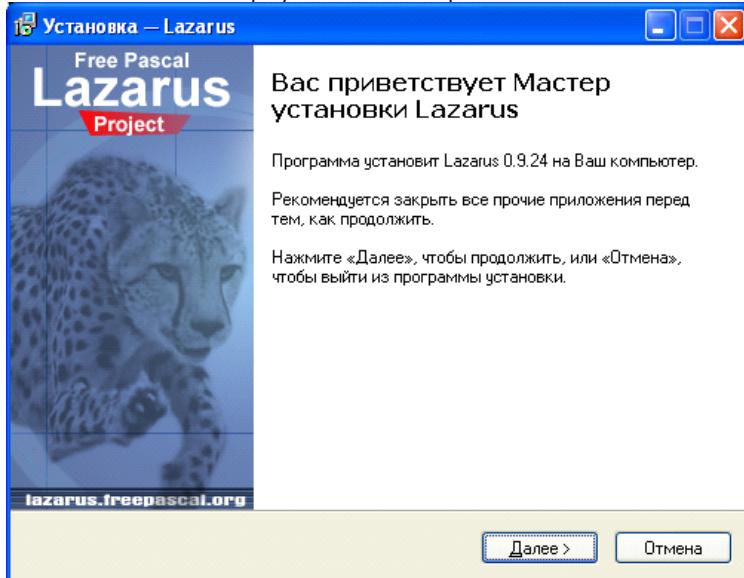


Рис. 1.11. Начало процесса установки Lazarus

Следующее окно ([рис. 1.11](#)) — информационное. Оно сообщает пользователю о начале процесса установки. Нажатие кнопки **Далее** приведёт к следующему шагу Мастера установки.

На следующем этапе ([рис. 1.12](#)) необходимо выбрать путь для установки Lazarus. По умолчанию программа будет установлена на диск С. Для выбора иного пути для установки следует воспользоваться кнопкой **Обзор**. В этом и во всех последующих окнах с помощью кнопки **Назад** можно вернуться к предыдущему шагу Мастера установки.

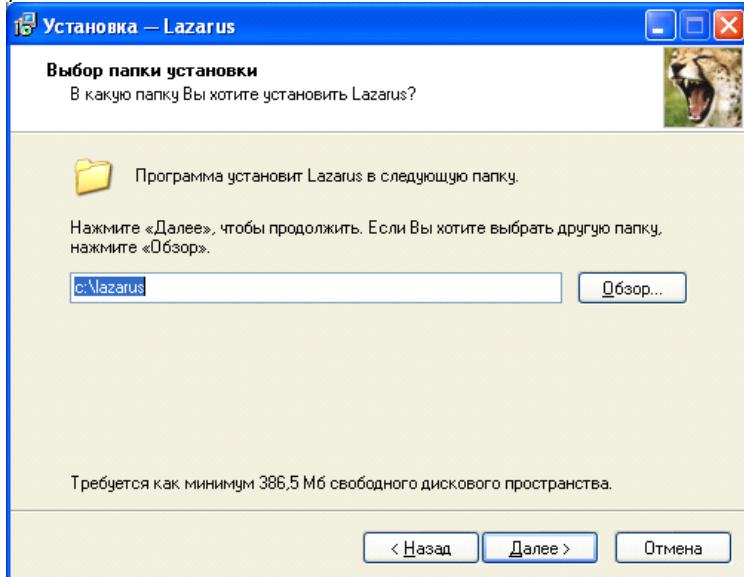


Рис. 1.12. Окно выбора пути для установки Lazarus

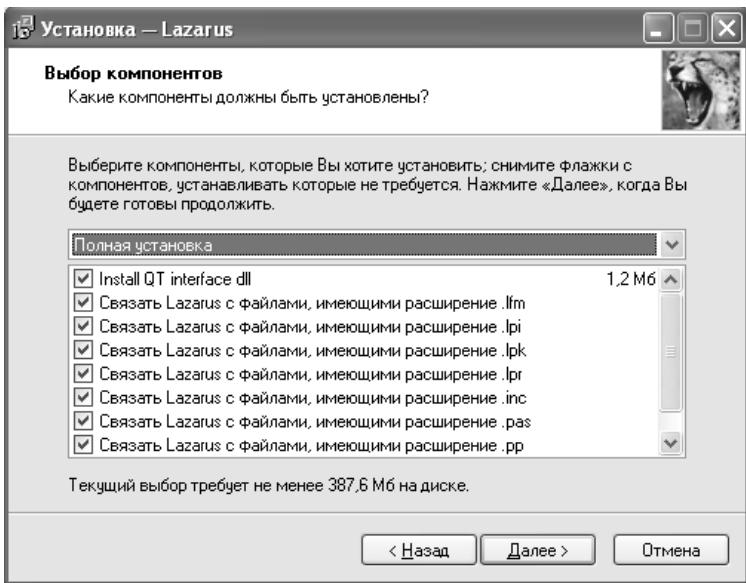
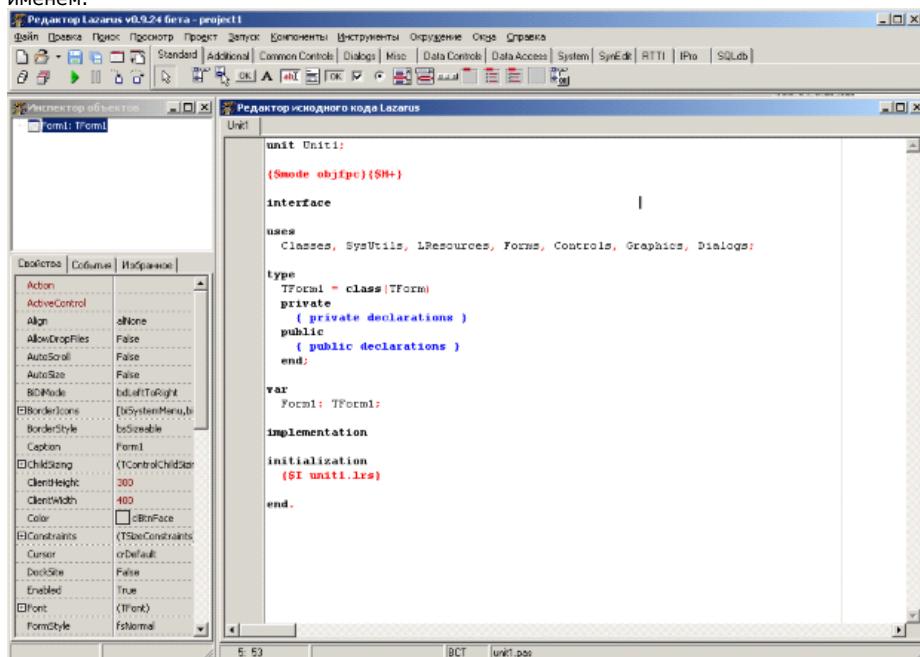


Рис. 1.13. Окно выбора компонентов для установки приложения

Следующий шаг — выбор из полного перечня средств системы необходимых компонентов ([рис. 1.13](#)). По умолчанию будут установлены все компоненты системы. Отменить установку компонента можно, если щелчком мыши убрать метку рядом с его именем.



[увеличить изображение](#)

Рис. 1.14. Среда визуального программирования Lazarus

В следующих окнах Мастера установки пользователь может выбрать папку меню Пуск, в которой будет создан ярлык для устанавливаемого приложения [2](#), и создать на рабочем столе ярлык для приложения Lazarus [2](#). Далее Мастер установки сообщит о том, куда будет установлено приложение Lazarus, коков тип установки, какие компоненты системы были выбраны, где будет создан ярлык и будет ли создан значок на рабочем столе. После этого начнётся процесс установки приложения на компьютер.

Контролировать установку 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**.

1.4.3 Среда Lazarus

На [рис. 1.14](#) 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. 1.15](#).

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. Окно инспектора объектов и окно формы нужны для создания интерфейса программы, а редактор исходного кода — для работы с её текстом. Файлы, из которых в результате получается программа, называют проектом.

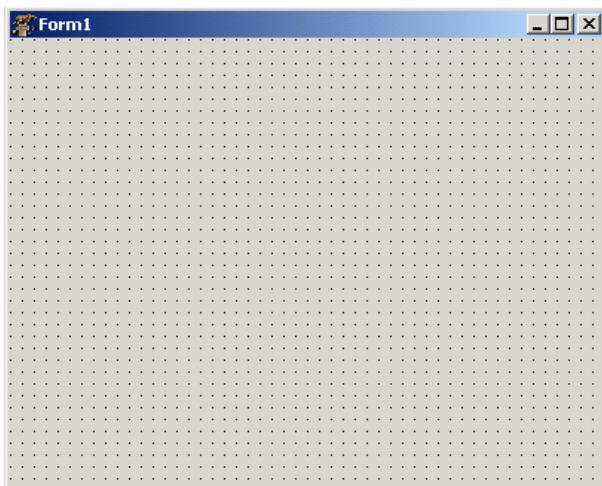


Рис. 1.15. Окно формы

1.4.4 Главное меню Lazarus

Все команды, необходимые для работы в среде визуального программирования Lazarus, содержаться 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...**, **Вернуть**, **Открыть недавнее**;
- сохранение файлов — **Сохранить**, **Сохранить как...**, **Сохранить все**;
- закрытие файлов — **Закрыть**, **Закрыть все файлы редактора**;
- удаление файлов из каталога — **Очистить каталог...**;
- 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**.

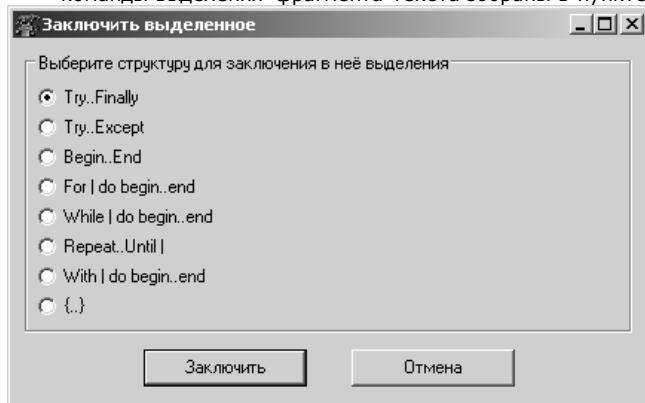


Рис. 1.16. Выбор конструкции языка для заключения в неё выделенного фрагмента программного кода

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 комментарий², 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 заключён выделенный фрагмент текста.

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

Команды меню **Поиск** также можно разделить на группы. Первая группа — это непосредственно команды поиска и замены, вторая — команды перехода, а третья — работа с закладкой. В четвёртой 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 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. 1.4.6);
- **Обозреватель кода** — содержит общую информацию о проекте;
- **Редактор LazDoc** — редактор шаблонов;
- **Браузер кода** — окно проводника проекта.

Следующая группа команд пункта меню **Просмотр** также открывает диалоговые окна.

Эти окна носят информационный характер. Так,

команды **Модуль...** и **Форма...** выводят 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**. Установливая 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 проектом²¹:

- 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 всё** — скомпоновать все файлы проекта;
- **Быстрая компиляция** — компиляция файлов программы;
- **Запуск** — запуск проекта с помощью отладчика (компиляция, компоновка и исполнение);
- **Пауза** — приостановка выполнения программы до нажатия любой клавиши;
- **Шаг со входом** — режим пошагового отладочного выполнения программы со 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, помеченному 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** — открытие окна, в котором 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.

Работа с компонентами²¹ организована при помощи команд пункта меню **Компоненты**.

Пункты меню **Инструменты** и Окружение¹⁰ применяются для настройки свойств среды программирования.

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 её настройку.

1.4.5 Форма приложения

Форма приложения ([рис. 1.15](#)) — 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 сетки²². 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...**

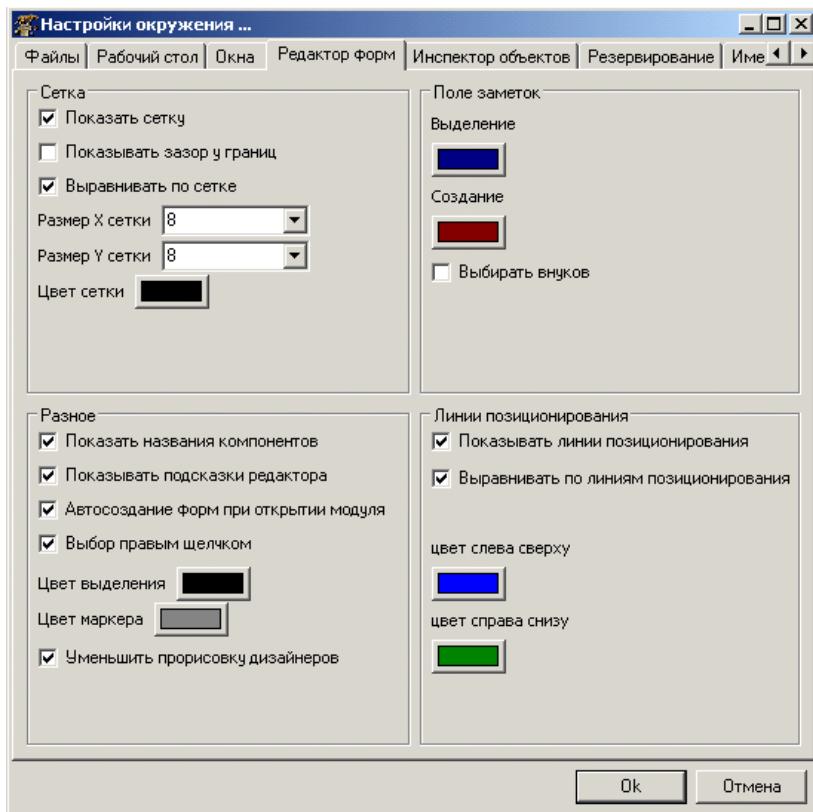


Рис. 1.17. Настройка окна формы

1.4.6 Окно редактора Lazarus

Окно редактора (рис. 1.14) тесно связано с формой приложения (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 её 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 создаваемой программы.

Обратите внимание, что при загрузке Lazarus автоматически загружается последний проект, с которым работал пользователь. Происходит это благодаря отмеченному пункту **Открывать последний проект при запуске** (рис. 1.19), который 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.

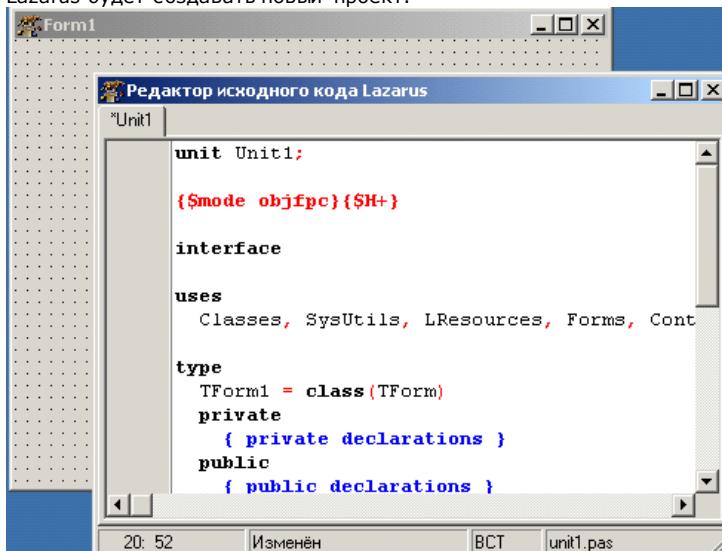


Рис. 1.18. Окно формы и окно редактора

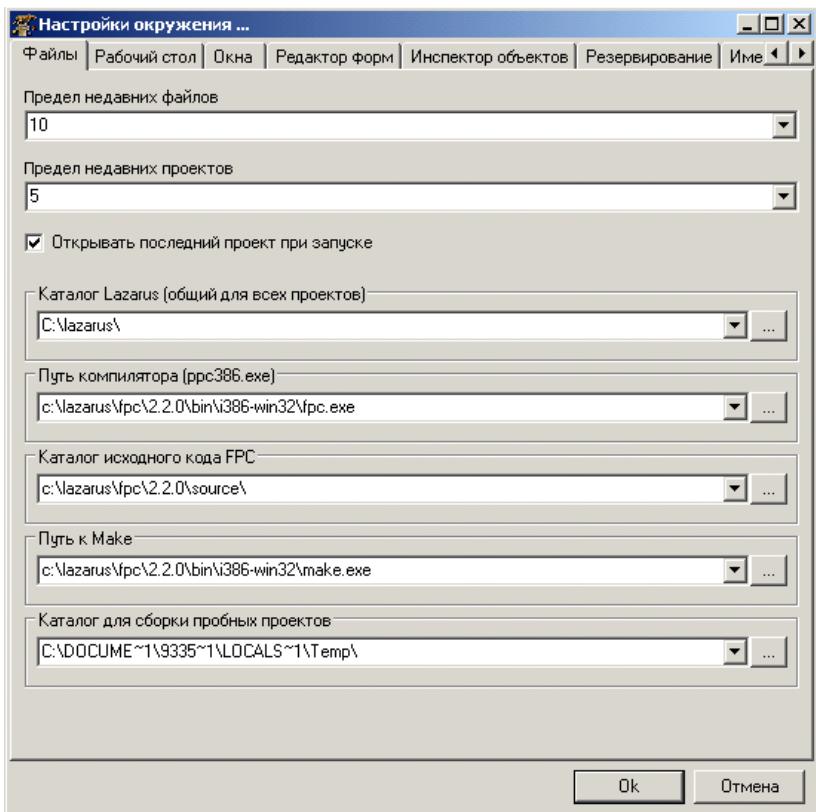


Рис. 1.19. Окно настройки файлов в среде Windows

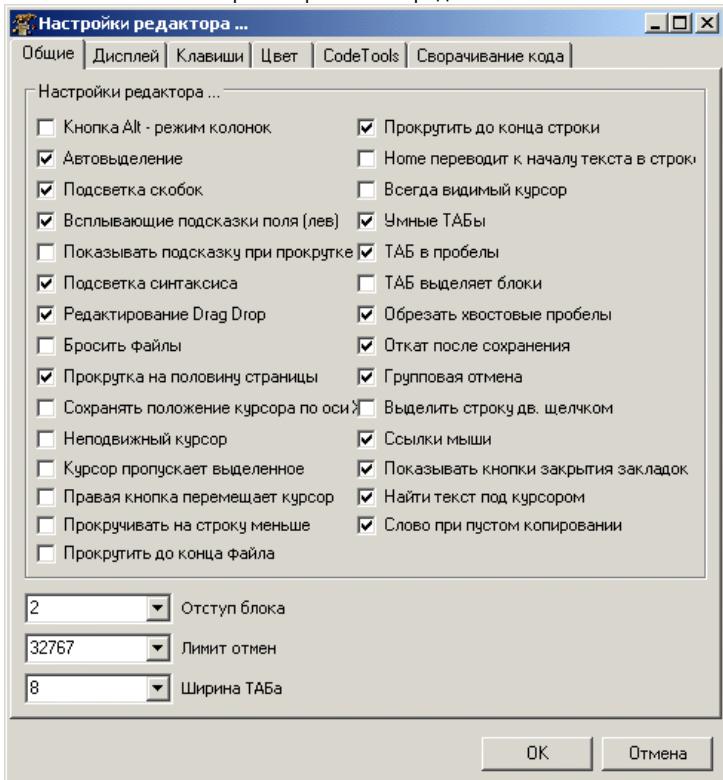


Рис. 1.20. Настройки окна редактора программного кода

Настройки окна редактора можно с помощью вкладки **Общие** меню **Окружение — Настройки редактора...** (рис. 1.20). Для того чтобы установить (отменить) ту или иную настройку, достаточно установить (убрать) маркер рядом с её названием и подтвердить изменения нажатием кнопки **Ok**.

На вкладке **Дисплей** меню **Окружение — Настройки редактора...** (рис. 1.21) можно изменить шрифт текста программного кода.

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. 1.22), которое появится после выполнения команды **Окружение — Шаблоны кода...**. Например, чтобы заработал рассмотренный шаблон, нужно выбрать (щёлкнуть 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...** выбрать

слово **пробела**. Если среди команд автозавершения выбрать **конец строки**, то шаблон будет появляться в тексте программы после ввода символа **b** и нажатия клавиши **Enter**.

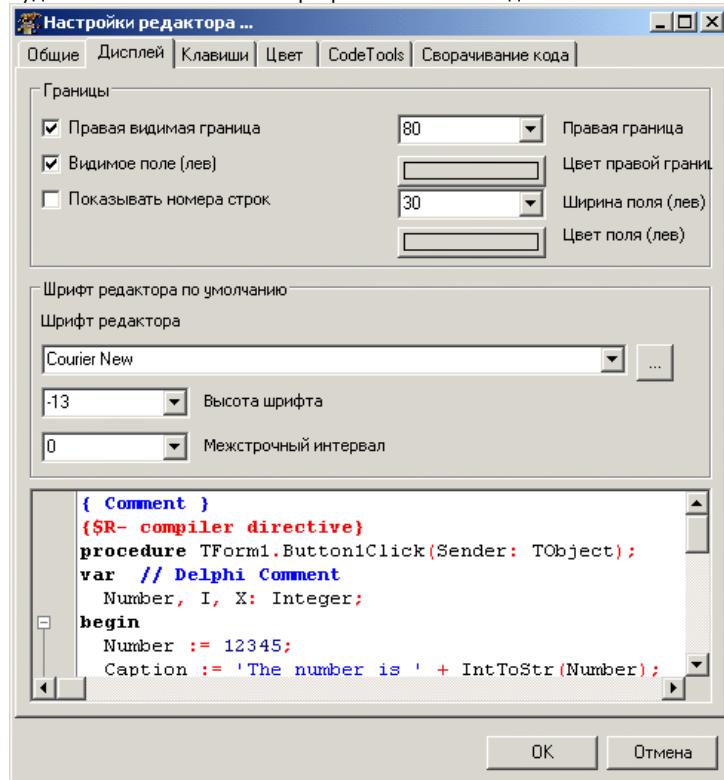


Рис. 1.21. Окно настройки редактора программного кода, вкладка Дисплей

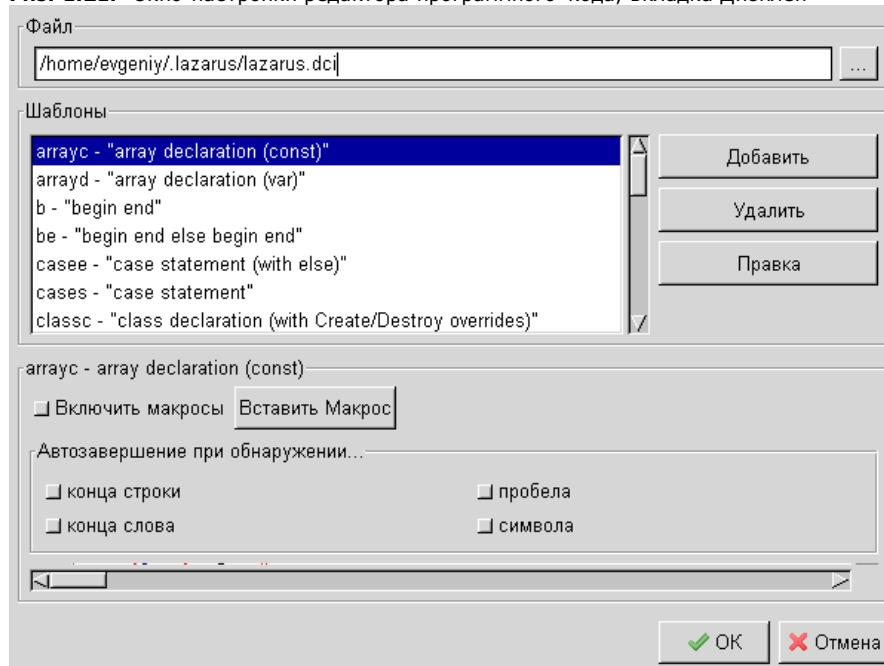


Рис. 1.22. Окно настройки шаблонов редактора программного кода в среде Linux

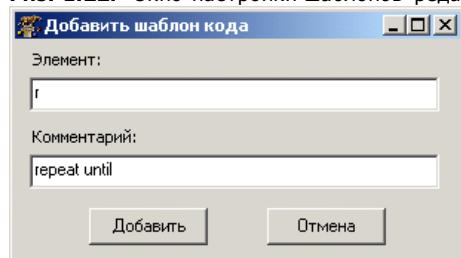


Рис. 1.23. Окно создания шаблона программного кода

Шаблоны можно добавлять, удалять и править. Для этого в окне **Шаблоны кода** предусмотрены специальные кнопки. В нижней части окна, над кнопками **Ok** и **Отмена**, расположено поле ввода и редактирования шаблонов с активным курсором. Если выделить любой шаблон из списка, щёлкнув по нему мышкой, то в поле ввода появится соответствующая конструкция. Если воспользоваться кнопкой **Добавить**, то появится диалоговое окно (рис. 1.23) 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. Для этого надо выделить вновь созданный шаблон и в поле ввода, расположеннном в нижней части окна **Шаблоны кода**, ввести соответствующий текст.

Например, создадим шаблон для конструкции `repeat..until`, которая будет добавлена в текст, если пользователь введёт символ `r`. Выполним следующую последовательность действий:

- щёлкнем по кнопке **Добавить** в окне **Шаблоны кода** ([см. рис. 1.22](#));
- в появившемся диалоговом окне **Добавить шаблон кода** ([рис. 1.23](#)) в поле **Элемент** введём символ `r`, а в поле **Комментарий** фразу `repeat until`, и нажмём кнопку **Добавить**;
- в окне **Шаблоны кода** ([рис. 1.24–1.25](#)) выделим в списке шаблонов строку `r – repeat until`;
- введём в поле ввода, расположенном в нижней части окна, конструкцию языка `repeat until`;
- нажмём 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.

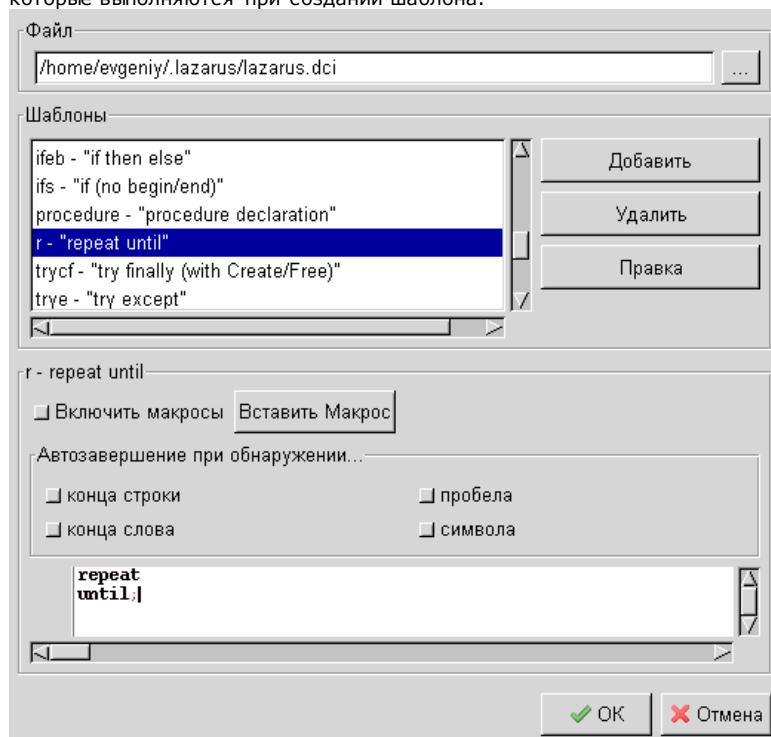


Рис. 1.24. Создание нового шаблона в Linux

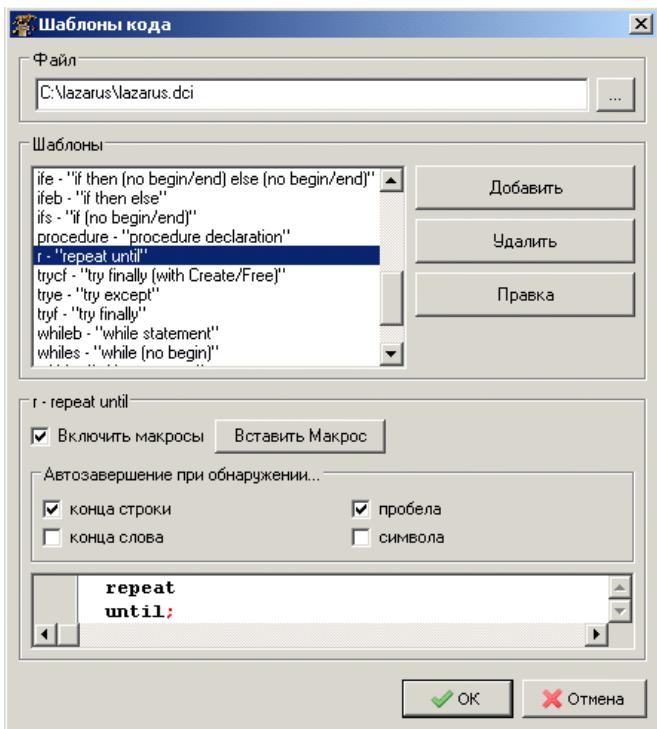


Рис. 1.25. Создание нового шаблона в среде Windows

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

1.4.7 Панель компонентов

Панель компонентов расположена под главным меню (рис. 1.14). Она состоит из большого числа групп, в которых располагаются соответствующие компоненты (рис. 1.26).



Рис. 1.26. Панель компонентов

Компонент — это некий функциональный элемент интерфейса, обладающий определенными свойствами. Размещая компоненты на форме, программист создаёт элементы внешнего вида своей будущей программы — окна, кнопки, переключатели, поля ввода и т. п.

Компоненты на панели 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.

Для того чтобы расположить новый компонент на форме нужно сделать два щелчка мышью:

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

1.4.8 Инспектор объектов

Окно инспектора объектов располагается слева от окна редактирования. Как правило, оно содержит информацию о выделенном объекте. На рис. 1.27 представлен инспектор объектов с информацией о вновь созданной форме.

Окно инспектора объектов имеет три вкладки: **Свойства, События, Избранное**, которые 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.

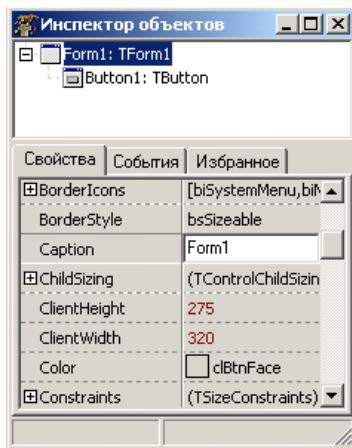


Рис. 1.27. Окно инспектора объектов

Чтобы выбрать свойство или событие, необходимо щёлкнуть левой кнопкой мыши по соответствующей строке.

Свойства, отображённые в таблице, могут быть простыми или сложными. Простые свойства определены единственным значением. Например, свойство **Caption** (Заголовок) определяется строкой символов, свойства **Height** (Высота) и **Width** (Ширина) — числом, свойство **Enabled** (Допустимость) — 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 **многоточия**, либо кнопка со стрелкой, направленной вниз. Щелчок по **многоточию** откроет диалоговое окно для установки значений сложных свойств (например **Font**). Обращение к стрелке приведёт 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:

- формирования внешнего вида программы, её интерфейса;
- написания 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.

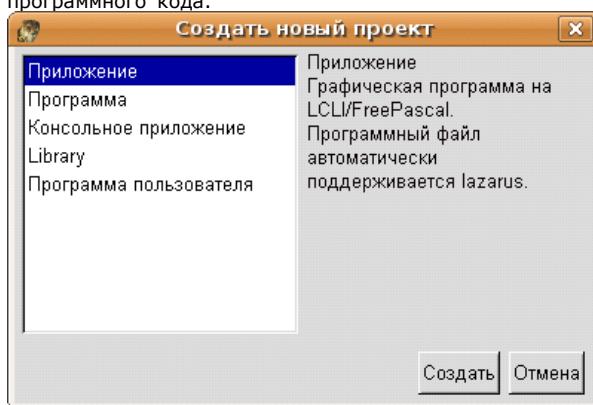


Рис. 1.28. Создание нового проекта в Linux

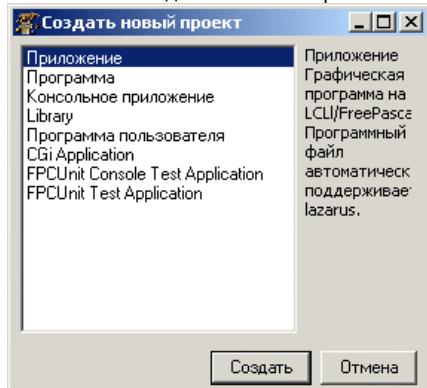


Рис. 1.29. Создание нового проекта в среде Windows

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

Начнём с создания нового проекта. Для этого выполним команду главного меню **Проект — Создать проект...** В появившемся диалоговом окне (рис. 1.28—1.29) выберем из списка слово **Приложение** и нажмём кнопку **Создать**. Результатом этих действий будет появление окна формы и окна редактора программного кода. Сохраним созданный проект, воспользовавшись командой **Проект — Сохранить проект как...** Откроется окно сохранения программного кода **Сохранить Unit1** (рис. 1.30—1.31).

Создадим в нём новую папку **Primer_1** (можно воспользоваться кнопкой **Создание новой папки (Создание каталога)**), откроем её и щёлкнем по кнопке **Сохранить**. Тем самым мы сохраним файл **Unit1.pas**, содержащий текст программы. Сразу же откроется окно **Сохранить проект** (рис. 1.32—1.33), в котором также необходимо щёлкнуть по кнопке **Сохранить**. Теперь мы сохранили файл **Project1**, содержащий общие сведения о проекте.

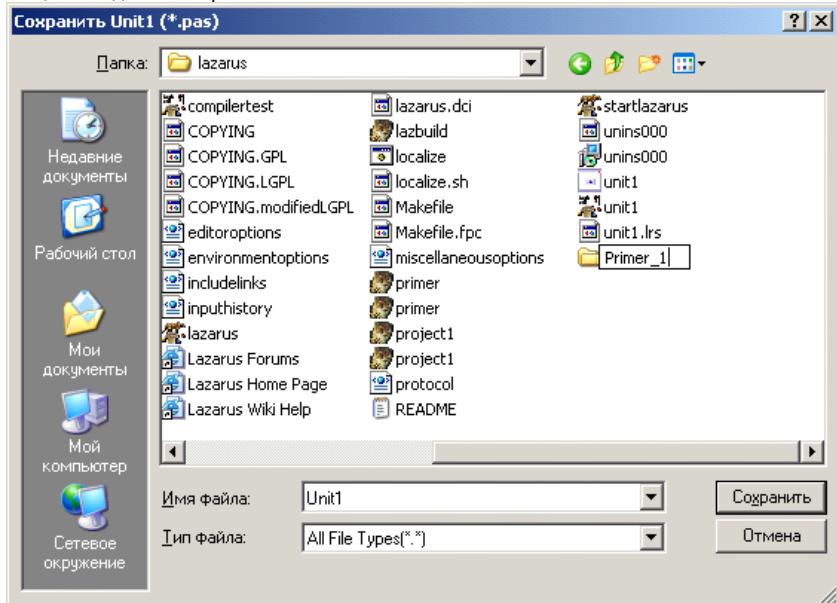


Рис. 1.30. Окно сохранения модуля в Windows

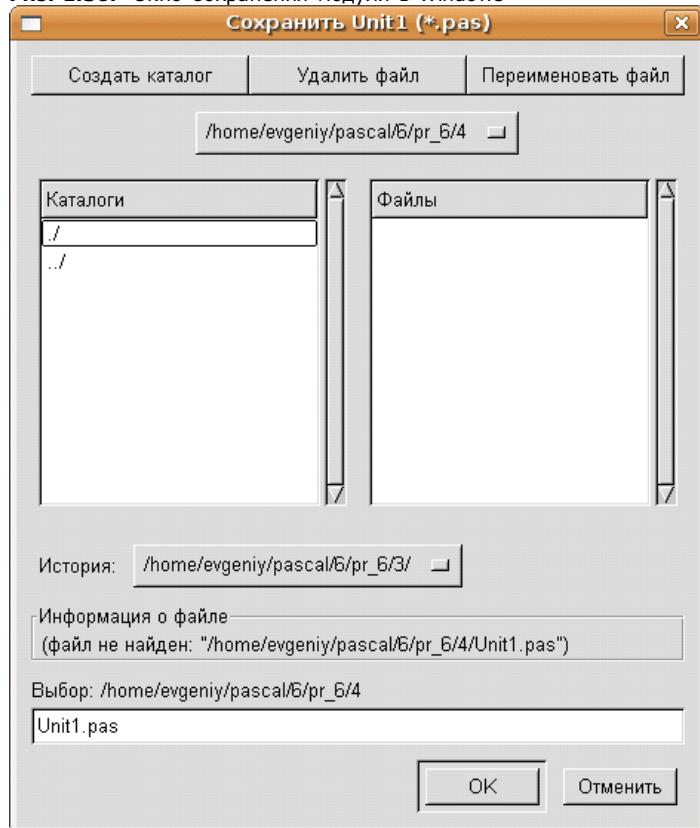


Рис. 1.31. Окно сохранения модуля в Linux

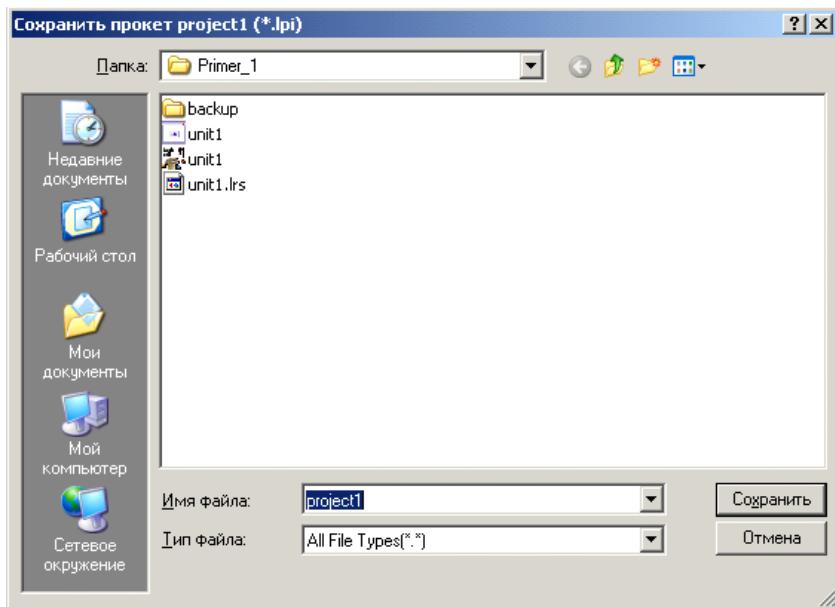


Рис. 1.32. Окно сохранения проекта в Windows

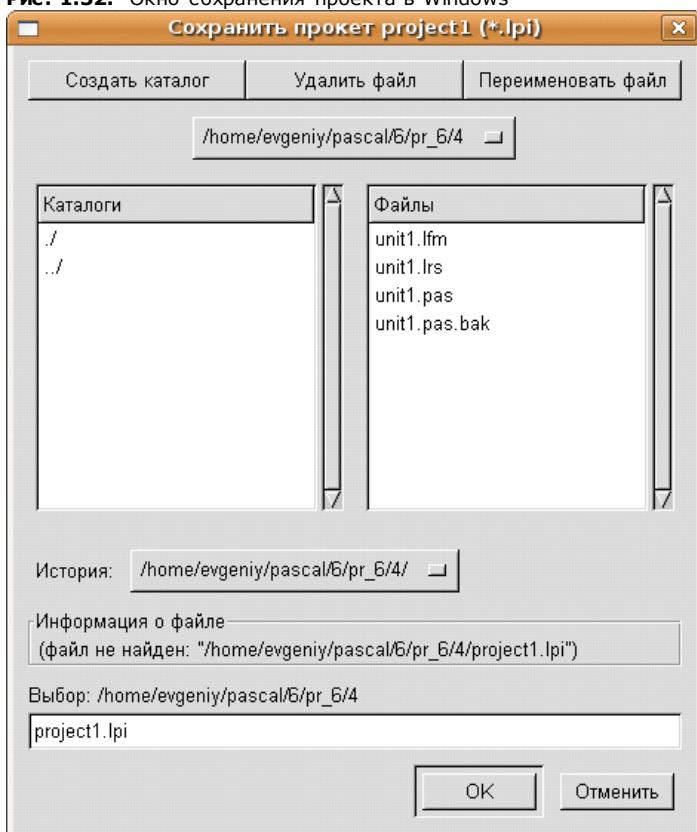


Рис. 1.33. Окно сохранения проекта в Linux

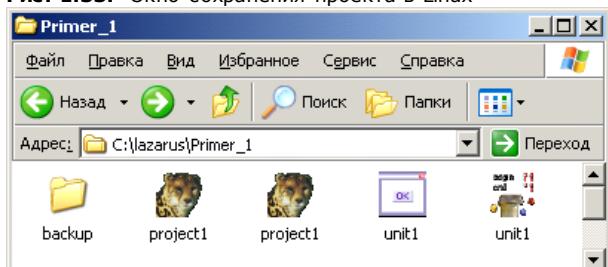


Рис. 1.34. Файлы проекта в Windows

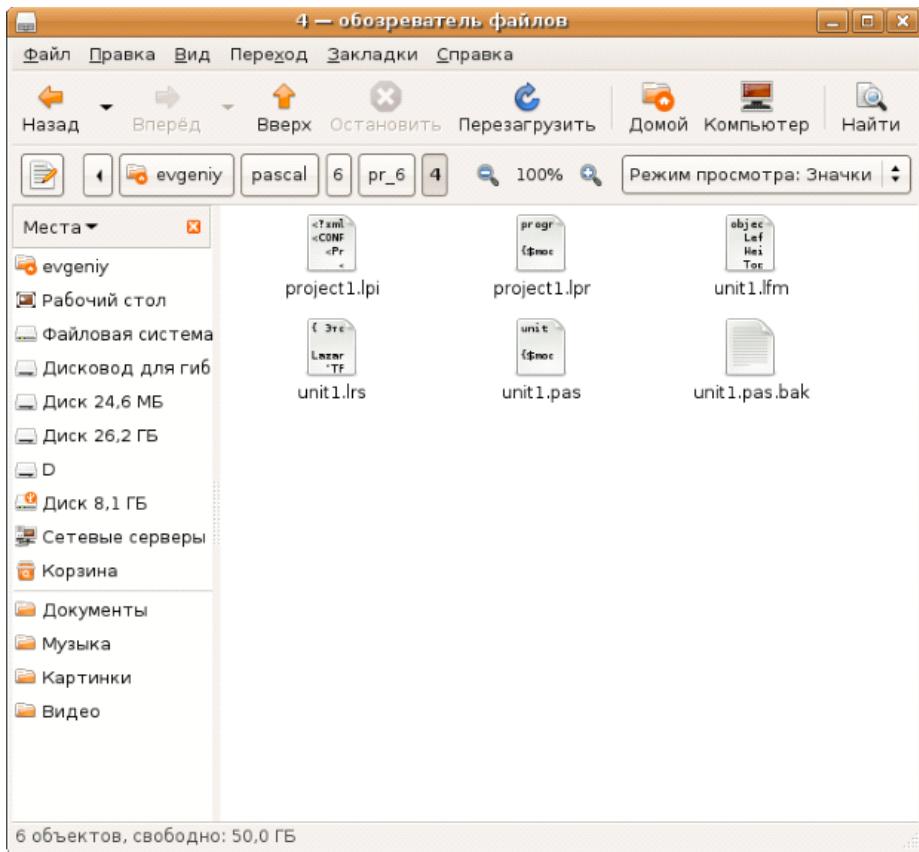


Рис. 1.35. Файлы проекта в Linux

На самом деле все наши манипуляции привели к сохранению более чем двух файлов. Теперь в каталоге **Primer_1** ([рис. 1.34—1.35](#)) хранится файл с текстом программы **unit1.pas**, файл **unit1.lfm** со сведениями о форме **Form1**, а также файлы **project1.lpn** и **project1.lpi**, содержащие настройки системы программирования и параметры конфигурации проекта.

Итак, проект сохранён. 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. Перейдём в окно инспектора объектов и найдем в нём свойство **Caption** (Заголовок). Свойство формы **Caption** отвечает за текст, выводимый в заголовке формы. По умолчанию оно имеет значение **Form1**. Изменим его на слово **ПРИМЕР 1** ([рис. 1.36](#)). Это изменение сразу же отразится на форме — в поле заголовка появится надпись "ПРИМЕР 1" ([рис. 1.38](#)). Аналогично можно изменить размеры формы, присвоив новые значения свойствам **Height** (Высота) и **Width** (Ширина), ещё проще это сделать, изменив положение границы формы с помощью кнопки мыши, как это делается со всеми стандартными окнами.

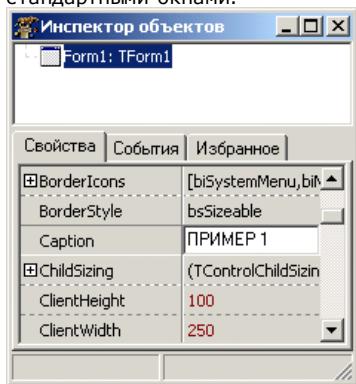


Рис. 1.36. Изменение свойства формы **Caption**

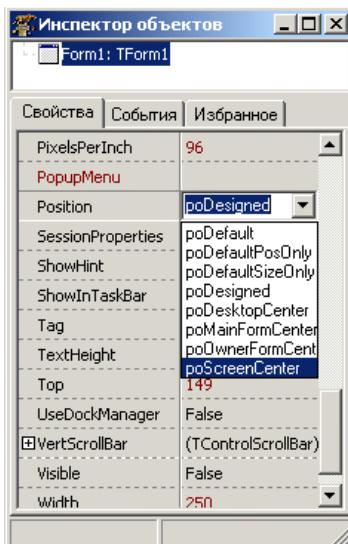


Рис. 1.37. Изменение свойства формы Position

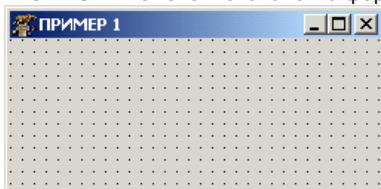


Рис. 1.38. Изменение заголовка формы

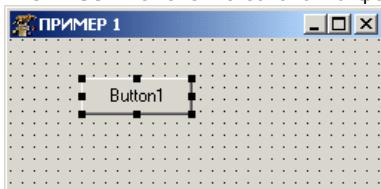


Рис. 1.39. Размещение кнопки на форме

Изменим ещё одно свойство формы — **Position** (положение формы на экране). Значения этого свойства не вводятся вручную, а выбираются из списка ([рис. 1.37](#)). Если выбрать значение **poScreenCenter**, то форма всегда будет появляться в центре экрана.

Напомним, что мы пишем программу про кнопку. Пришло время поместить её на форму. Для этого обратимся к панели компонентов ([рис. 1.26](#)) и найдём там

компонент **Tbutton**. Чтобы разместить компонент на форме, нужно сделать два щелчка мышью: первый — по компоненту, второй — по окну формы. В результате форма примет вид, представленный на [рис. 1.39](#).

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** (высота формы), **Width** (ширина формы). В заголовке напишем фразу "Щёлкни мышкой!!!", установим ширину 135 пикселей, а высоту — 25 пикселей.

Сохраним проект ([Проект — Сохранить проект](#)).



Рис. 1.40. Кнопка Запуск на панели инструментов Lazarus

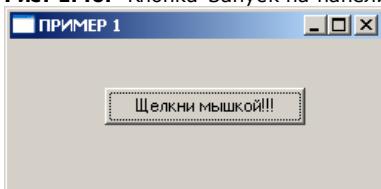


Рис. 1.41. Результат работы программы

Для того чтобы посмотреть, как работает наша программа, её необходимо запустить на выполнение. Сделать это можно командой **Запуск — Запуск**, функциональной клавишей **F9** или кнопкой **Запуск** на панели инструментов ([рис. 1.40](#)). Созданное окно с кнопкой должно выглядеть, как на [рис. 1.41](#).

Теперь щёлкнем по кнопке и убедимся, что ничего не произойдет. В этом нет ничего удивительного, ведь мы ещё 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. Вернёмся 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 "Щёлкни мышкой!!!". Предположим, что если мы щёлкнем 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 перейдём в режим редактирования. Перейдём к инспектору объектов, выделим кнопку **Button1** и обратимся к вкладке **События** инспектора объектов. Напомним, что здесь расположены описания событий. Выберем событие **OnClick** — обработка щелчка мыши и дважды щёлкнем в поле справа от названия ([рис. 1.42](#)).

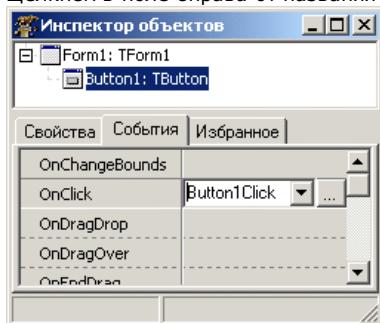


Рис. 1.42. Выбор события OnClick

В результате описанных действий в окне редактора появится следующий текст:

```

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

```

На данном этапе изложения материала данный текст — это фрагмент программного кода, именуемый подпрограммой. О назначении этой подпрограммы можно догадаться по её 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 (щелчка мыши).

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;

```

Сохраним созданную программу, запустим её на выполнение и убедимся, что кнопка реагирует на щелчок мыши ([рис. 1.43](#)). Закрыть проект можно командой **Проект — Закрыть проект**.

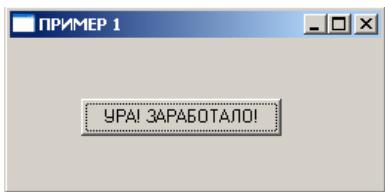


Рис. 1.43. Окончательный результат работы программы

1.4.10 Полезная программа

Итак, первая программа написана. Она не несёт 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...**

Воспользуемсяся 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 односторонних надписей); два поля ввода **Edit** (это редактируемое текстовое поле, предназначенное для ввода, отображения или редактирования одной текстовой строки); одну кнопку **Button** (этот компонент обычно используют для реализации некоторой команды). Компоненты должны располагаться на форме примерно так, как показано на [рис. 1.44](#).

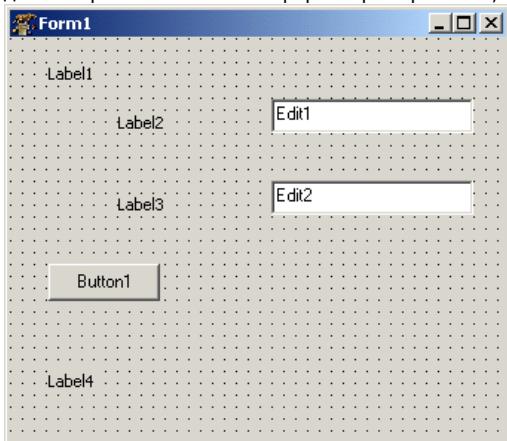


Рис. 1.44. Конструирование формы



Рис. 1.45. Изменение свойств формы и ее компонентов

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 произойдёт при щелчке по кнопке **Button1** и будет выведено в **Label4**. Изменим свойства формы и размещенных на ней компонентов в соответствии с [табл. 1.1](#), [табл. 1.2](#), [табл. 1.3](#), [табл. 1.4](#). После всех изменений форма будет иметь вид, представленный на [рис. 1.45](#).

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 1.1. Значение свойств формы

Свойство	Label1	Label2	Label3	Label4	Описание свойства
Caption	Меры веса	пуды	фунты	В кг:	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 1.2. Свойства компонентов типа надпись

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 1.3. Свойства компонентов поле ввода

Свойство	Значение	Описание свойства
Caption	ВЫЧИСЛИТЬ	Текст на кнопке
Height	25	Высота компонента
Left	200	Ширина компонента
*Top ¹¹	150	Координата верхнего края
Width	80	Координата левого края

Таблица 1.4. Свойства компонента кнопка

Интерфейс программы готов. Займёмся вычислительной частью задачи. Для этого создадим для кнопки **Button1** обработчик события **OnClick**, для чего дважды щёлкнем по объекту **Button1** левой кнопкой мыши. Между словами **begin** и **end** созданной подпрограммы запишем несколько команд, чтобы получился следующий программный код:

```
procedure TForm1.Button1Click ( Sender : TObject );
var
pud, fuit : integer;
kg : real;
begin
    pud:= StrToInt ( Edit1. Text );
    fuit := StrToInt ( Edit2. Text );
    kg:=pud * 16.38+ fuit * 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 такие имеются.

Строка 2. Открывает блок описания переменных (**var** — от **variable**, переменная).

Строка 3. Описывает переменные **pud** и **fuit**. Компьютер понимает эту запись как команду выделить в памяти место для хранения двух целых чисел¹².

Строка 4. Описывает одну вещественную переменную **kg**.

Строка 5. Начинает программный блок (**begin** — начало).

Строка 6. Команда, выполняющая следующие действия. Из свойства **Text** поля ввода **Edit1** считывает введённую туда информацию. Эта информация воспринимается компьютером как строка текста (например, "360" — это строка из трёх символов), а поскольку нам для вычислений нужны числовые значения, то

функция **StrToInt** преобразовывает её 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 **fuit**. Фактически команды, 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 чётко указывать все операции. Нельзя, например, опустить знак умножения, как это принято в математике.

The screenshot shows the Lazarus IDE interface. The main window displays a Pascal program in the Unit1 unit:

```

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;

```

The line `kg=pud*16.38+funt*16.38/40;` is highlighted in orange. Below the code editor is a status bar showing '43: 31' and 'ВСТ'. The bottom part of the interface is a 'Сообщения' (Messages) window containing the following error output:

- Unit1.pas(43,7) Warning: Local variable "kg" does not seem to be initialized
- Unit1.pas(43,31) Error: Illegal expression
- Unit1.pas(52) Fatal: There were 1 errors compiling module, stopping

Рис. 1.46. Сообщение об ошибке в программе

Строка 9. Изменяет свойство **Caption** объекта **Label4**. В него будет помещён 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 машинный код. Для компиляции нужно выполнить команду **Запуск — Быстрая компиляция**. Во время компиляции проверяется наличие синтаксических ошибок, и 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. 1.46](#).

Строка с ошибкой выделена красным цветом, а под окном редактора появилось окно Сообщение³¹, в котором описана допущенная ошибка. Ошибку нужно исправить, а программу ещё 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 исполняемый файл, имя которого совпадает с именем проекта.

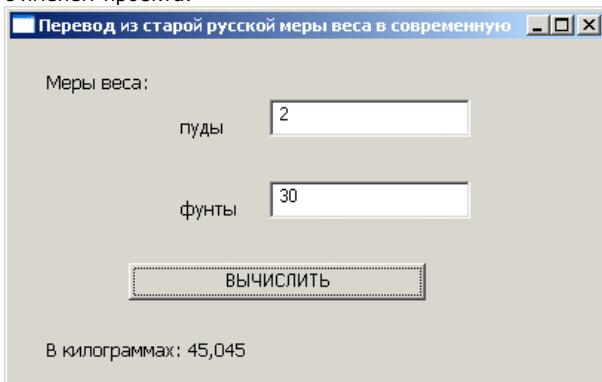


Рис. 1.47. Результат работы программы

Вообще говоря, если команду **Запуск — Быстрая компиляция** опустить, а сразу выполнить команду **Запуск — Собрать**, то произойдёт 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: в Linux — **./project1**, в Windows — **projec1.exe**. На [рис. 1.47](#) показан результат работы нашей 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, не сложно. Нужно выполнить команду **Проект — Открыть проект...** (**Ctrl+F11**), перейти к папке с проектом и выбрать из списка файл с именем **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. 1.28—1.29](#)) выбрать

фразу **Программа пользователя** и нажать кнопку **Создать**. На экране появится окно редактора программного кода ([рис. 1.48](#)), в котором уже представлена общая структура программы на языке Free Pascal.

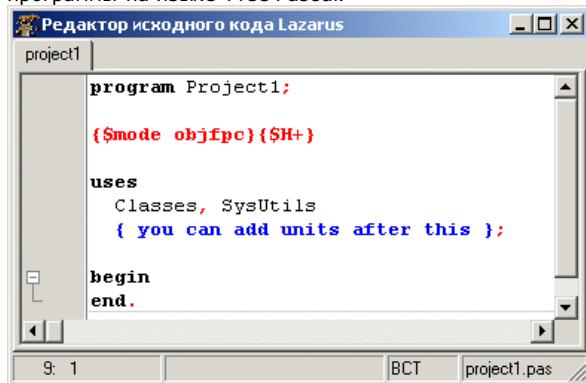


Рис. 1.48. Окно редактора в консольном приложении

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. 1.48](#)) 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. 1.48](#) 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, заключённый в фигурные скобки или начинающийся с двух наклонных черт. Этот текст не является программным кодом, а носит информационный характер. Например, в нашем случае текст, заключённый 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 какогонибудь угла по данной градусной мере, нужно умножить число градусов на $\pi/180$, число минут на $\pi/(180 * 60)$ 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 и в визуальном проекте. Результаты работы нашей программы будет иметь вид:

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

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

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

и тело программы

```
begin //Начало тела программы.
  write ( ' gradus= ' ); //Вывод на экран символов gradus=.
```

```

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

или

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**, её действие приостанавливается до тех пор, пока не будут введены исходные данные. При вводе числовых значений два числа считаются разделёнными, 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** переведёт 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);

или

writeln (x1, x2,..., xn);

где **x1, x2,..., xn** представляют собой список выводимых переменных, констант, выражений. Если элемент списка — текстовая информация, её 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

(мантизму числа), умножить на 10, возведённое в степень, значение которой указано после числа

E

(порядок числа). Например,

0.35469000000E - 01 = 0.35469000000 * 10⁻¹ = 0.035469,

- 5.43286710000E + 02 = - 5.43286710000 * 10² = - 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 (всё число занимает пять позиций и три из них после запятой).

Лекция 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 создаётся 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 создаётся единый выполняемый файл с расширением **.exe**, 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 [▲](#).

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 модул1, модул2,..., модулn;  
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;
- 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 жёстко установлен, во Free Pascal такого строгого требования нет. В программе может быть несколько разделов описания констант, переменных и т. д. Более подробно структуру консольной программы на языке Free Pascal можно представить следующим образом:

```
program имя_программы;  
uses модул1, модул2,..., модулn;  
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). Особо отметим символ ";" — он служит для отделения одного оператора от другого. Можно сказать, что ";" ставится между операторами. 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 модул1, модул2,..., модулn;
const описания_констант;
type описания_типов;
var описания_переменных;
begin
    оператор_1;
    оператор_2;
    ...
    оператор_n
end.
```

Приведём 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 служебные слова языка, которые имеют специальное значение для компилятора и используются только в том смысле, в котором они определены (операторы языка, типы данных и т. п.). Идентификатор — совокупность букв, цифр и символа подчёркивания. Идентификатор начинается с буквы или символа подчёркивания. Идентификаторы служат для именования различных объектов (констант, переменных, меток, типов данных, процедур, функций, модулей, классов) языка. 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. Если текст начинается с двух символов "косая черта" // и заканчивается символом перехода на новую строку, заключён в фигурные скобки {} или располагается между парами символов (* и *), то компилятор его игнорирует. Например:

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

Комментарии удобно использовать как для пояснений к программе, так и для временного исключения фрагментов программы при отладке.

2.4 Данные в языке Free Pascal

Для решения задачи в любой программе выполняется обработка каких-либо данных. Данные хранятся в памяти компьютера и могут быть самых различных типов: целые и вещественные числа, символы, строки, массивы. 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. Участок памяти (ячейка), в котором хранится значение определённого 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 на 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_2,..., переменная_N: тип_переменных;
Например:
```

```
var
ha : integer; //Объявлена целочисленная переменная.
```

```
hb, c : real; //Объявлены две вещественные переменные.
```

Константа — это величина, которая не изменяет своего значения в процессе выполнения программы. Описание константы имеет вид:

const имя_константы = значение;

Например:

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

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 любого из 256 символов клавиатуры.

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 байтов. Диапазоны значений данных целочисленного типа представлены в [табл. 2.1](#).

Тип	Диапазон	Размер, байт
Byte	0... 255	1
Word	0... 65535	2
LongWord	0... 4294967295	4
ShortInt	-128... 127	1
integer	-2147483648... 2147483647	4
LongInt	-2147483648... 2147483647	4
Smallint	-32768... 32767	2
Int64	-2 ⁶³ ... 2 ⁶³	8
Cardinal	0... 4294967295	4

Таблица 2.1. Целочисленные типы данных

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

mE ± p

, где

m

— мантисса (целое или дробное число с десятичной точкой),

p

— порядок (целое число). Чтобы перейти от экспоненциальной формы к обычному представлению числа, необходимо мантиссу умножить на десять в степени порядка.

Например:

$$-36.142E + 2 = -36.142 * 10^2 = 3614.2; 7.25E - 5 = 7.25 * 10^{-5} = 0.0000725.$$

Вещественное число в Pascal может занимать от четырёх до десяти байтов. Диапазоны значений вещественного типа представлены в [табл. 2.2](#).

Тип	Диапазон	Кол-во знач-х цифр	Размер, байт
Single	1.5E45... 3.4E + 38	7–8	4
Real	2.9E - 39... 1.7E + 38	15–16	8
Double	5.0E - 324... 1.7E + 308	15–16	8
Extended	3.4E - 4932... 3.4E + 4932	19–20	10
Comp	-2 ⁶³ ... 2 ⁶³	19–20	8
Currency	-922337203685477.5808... 922337203685477.5807	19–20	8

Таблица 2.2. Вещественные типы данных

Примеры описания вещественных переменных:

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

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 определён лишь один логический тип

данных — `boolean`. Логические типы данных, определённые в языке Free Pascal, представлены в [табл. 2.3](#).

Тип Размер, байт

<code>Boolean</code>	1
<code>ByteBool</code>	1
<code>WordBool</code>	2
<code>LongBool</code>	4

Таблица 2.3. Логические типы данных

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 задаётся непосредственным перечислением значений, которые он может принимать:

```
var имя_переменной: (значение_1, значение_2,..., значение_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:

```
type //Создание нового типа данных — времена года.
```

```
year_seasons = ( winter, spring, summer, autumn );
```

```
var //Описание соответствующей переменной.
```

```
yt : year_seasons;
```

2.4.8 Интервальный тип

Интервальный тип задаётся границами своих значений внутри базового типа:

```
var
```

```
имя_переменной: минимальное_значение.. максимальное_значение;
```

Обратите внимание, что в данной записи два символа точки рассматриваются как один, поэтому пробел между ними не допускается. Кроме того, левая граница диапазона не должна превышать правую. Например:

```
var
date : 1.. 3 1;
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;
```

```
var //Описание соответствующей переменной days — рабочие дни.
```

```
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 [4](#). Число элементов массива фиксируется при описании типа и в процессе выполнения программы не изменяется.

Для описания массива используют ключевые слова `array...of`:

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

где:

- имя_массива — 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_нач.. индекс1_кон, индекс2_нач.. индекс2_кон,...]

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;
```

Ещё один способ описать массив — создать новый тип данных. Например, так:

```
type
```

```
//Объявляется новый тип данных — трёхмерный массив целых чисел.
```

```
massiv=array [ 0.. 4, 1.. 2, 3.. 5 ] of word;
```

```
var //Описывается переменная соответствующего типа.
```

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 приведённом примере строки Str1 и Str2 одинаковой длины.

Запись — 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

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 — 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 ёмкостью диска, на котором он хранится. В Lazarus можно объявить типизированный файл:

имя_файловой_переменой = file of тип_данных;

нетипизированный файл:

имя_файловой_переменой = file;

и текстовый файл:

имя_файловой_переменой = 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

компилятор автоматически выделяет память под переменную имя_переменной в

соответствии с указанным типом. Доступ к объявленной переменной осуществляется по её имени. При этом все обращения к переменной заменяются адресом ячейки памяти, в которой хранится её 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 — это переменная, значением которой является адрес памяти, где хранится объект определённого 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 **типовизированными**. 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 меняются в процессе выполнения программы, то есть динамически.

2.5 Операции и выражения

Выражение задаёт 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))**. В [табл. 2.4](#) представлены основные операции языка программирования Free Pascal.

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

Таблица 2.4. Основные операции языка Free Pascal

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.

Перейдём к подробному рассмотрению основных операций языка.

2.5.1 Арифметические операции

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

Операции целочисленной арифметики (применяются 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 = 3`.

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`)²¹. Оба операнда переводятся в двоичную систему, затем над ними происходит логическое поразрядное умножение operandов 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 —

$A = 000000000001110$

и

$B = 0000000000011000$

. В результате логического умножения A and B получим `000000000001000` или 8 в десятичной системе счисления ([рис. 2.1](#)). Таким образом,

$A \& B = 14 \& 24 = 8$

$A =$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	14
0	0	0	0	0	0	0	0	0	0	0	1	1	1	0			
и																	
$B =$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	24
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0			
A and $B =$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	8
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0			

Рис. 2.1. Пример логического умножения A and B

Арифметическое ИЛИ (or)²². Здесь оба операнда также переводятся в двоичную систему, после чего над ними происходит логическое поразрядное сложение operandов 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](#)).

$A =$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	14
0	0	0	0	0	0	0	0	0	0	0	1	1	1	0			
или																	
$B =$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	24
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0			
A or $B =$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	30
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1			

Рис. 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 operandом.

Применение операции not вызывает побитную инверсию²³ двоичного представления числа ([рис. 2.3](#)).

A=  13

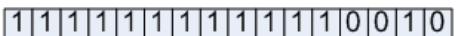
not A=  not 13=-14

Рис. 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 111000. В десятичной системе это двоичное число равно 120. Итак, **15 shl 3 = 120** ([рис. 2.4](#)).

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**.



00000000000001111 15
← Сдвиг на три позиции
00000000001111000 15 shl 3=120

Рис. 2.4. Сдвиг влево 15 shl 3



00000000000001111 15
→ Сдвиг на одну позицию
00000000000000111 15 shr 1=7

Рис. 2.5. Сдвиг вправо 15 shr 1

2.5.2 Операции отношения

Операции отношения применяются к двум operandам и возвращают в качестве результата логическое значение. Таких операций семь: **>**, **>=**, **<**, **<=**, **=**, **<>**, **in**.

Результат операции отношения — логическое значение **true** (истина) или **false** (ложь). Назначение операций **>**, **>=**, **<**, **<=**, **=**, **<>** понятно. Результат операции **2 > 3** — ложь, **5 ≥ 4** — истина. Результат операции

x ≤ y

зависит от конкретных значений

x

и

y

. Поясним, как работает операция **in**. Первым operandом этой операции должно быть любое выражение, вторым — множество, состоящее из элементов того же типа.

Результат операции **true** (истина), если левый operand 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.

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

Таблица 2.5. Логические операции

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

...

adr_a:=@a; //Оператор записывает в переменную adr_a

//адрес переменной 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
...
a:=adr_a ^; //Оператор записывает в переменную a
//значение, хранящееся по адресу adr_a.
```

2.6 Стандартные функции

В языке Free Pascal определены стандартные функции над арифметическими операндами ([табл. 2.6](#)).

Обозначение	Тип результата	Тип аргументов	Действие
<code>abs(x)</code>	целый/вещественный	целый/вещественный	модуль числа
<code>sin(x)</code>	вещественный	вещественный	синус
<code>cos(x)</code>	вещественный	вещественный	косинус
<code>arctan(x)</code>	вещественный	вещественный	арктангенс
<code>pi</code>	без аргумента	вещественный	число π
<code>exp(x)</code>	вещественный	вещественный	экспонента e^x
<code>ln(x)</code>	вещественный	вещественный	натуральный логарифм
<code>sqr(x)</code>	вещественный	вещественный	квадрат числа
<code>sqrt(x)</code>	вещественный	вещественный	корень квадратный
<code>int(x)</code>	вещественный	вещественный	целая часть числа
<code>frac(x)</code>	вещественный	вещественный	дробная часть числа
<code>round(x)</code>	вещественный	целый	округление числа
<code>trunc(x)</code>	вещественный	целый	отсечение дробной части числа
<code>random(n)</code>	целый	целый	случайное число от 0 до n

Функции, определённые в программном модуле `Math`

<code>arcos(x)</code>	вещественный	вещественный	арккосинус
<code>arcsin(x)</code>	вещественный	вещественный	арксинус
<code>arccot(x)</code>	вещественный	вещественный	арккотангенс
<code>arctan2(y,x)</code>	вещественный	вещественный	арктангенс y/x
<code>cosecant(x)</code>	вещественный	вещественный	косеканс
<code>sec(x)</code>	вещественный	вещественный	секанс
<code>cot(x)</code>	вещественный	вещественный	котангенс
<code>tan(x)</code>	вещественный	вещественный	тангенс
<code>lnXP1(x)</code>	вещественный	вещественный	логарифм натуральный от $x + 1$
<code>log10(x)</code>	вещественный	вещественный	десятичный логарифм
<code>log2(x)</code>	вещественный	вещественный	логарифм по основанию два
<code>logN(n,x)</code>	вещественный	вещественный	Логарифм от x по основанию n

Таблица 2.6. Некоторые арифметические функции

Определённую проблему представляет возведение

x

в степень

n

. Если значение степени

n

— целое положительное число, то можно

n

раз перемножить

x

(что даёт более точный результат и при целом

n

предпочтительней) или воспользоваться формулой

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

которая программируется с помощью стандартных функций языка:

- `exp(n*ln(x))` — для положительного ;

	$\cdot -\exp(n * \ln(\text{abs}(x)))$ — для отрицательного .		
Данную же формулу можно использовать для возведения			
x	в дробную степень		
n	, где		
k/l	— обыкновенная правильная дробь вида		
l	, а знаменатель		
l	— нечётный. Если знаменатель		
l	— чётный, это означает извлечение корня чётной степени, следовательно, есть ограничения на выполнение операции ($x > 0$).		
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.			
Обозначение	Тип аргументов	Тип результата	Действие
Работа со строками			
<code>length(S)</code>	строка	целое	текущая длина строки S
<code>concat(S1, S2,...)</code>	строки	строка	объединение строк $S1, S2, \dots$
<code>copy(S, n, m)</code>	строка, целое, целое	строка	копирование n символов строки S начиная с m -й позиции
<code>delete(S, n, m)</code>	строка, целое, целое	строка	удаление n символов из строки S начиная с m -й позиции
<code>insert(S, n, m)</code>	строка, целое, целое	строка	вставка n символов в строку S начиная с m -й позиции
<code>pos(S1, S2)</code>	строки	целое	номер позиции, с которой начинается вхождение $S2$ в $S1$
<code>chr(x)</code>	целое	символ	возвращает символ с кодом x
<code>ord(c)</code>	символ	целое	возвращает код символа c
Преобразование строк в другие типы			
<code>StrToDateTame(S)</code>	строка	дата и время	преобразует символы из строки s в дату и время
<code>StrToFloat(S)</code>	строка	вещественное	преобразует символы из строки s в вещественное число
<code>StrToInt(S)</code>	строка	целое	преобразует символы из строки s в вещественное число
<code>Val(S,X,Kod)</code>	строка		преобразует строку символов S во внутреннее представление целой или вещественной переменной X , если преобразование прошло успешно, $Kod=0$
Обратное преобразование			
<code>DateTimeToStr(V)</code>	дата и время	строка	преобразует дату и время в строку.
<code>FloatToStr(V)</code>	вещественное	строка	преобразует вещественное число в строку
<code>IntToStr(V)</code>	целое	строка	преобразует целочисленное число в строку
<code>FloatToStrF(V,F,P,D)</code>	вещественное	строка	преобразует вещественное число V в строку символов с учётом формата F и параметров P, D

Таблица 2.7. Функции обработки строк

Поясним назначение функции `FloatToStrF(V,F,P,D)`. Обычно её используют для форматированного вывода вещественного числа. Значения параметра `F` представлены в [табл. 2.8](#), `P` — количество символов в формируемой из числа строке, `D` — количество символов в дробной части числа.

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

Таблица 2.8. Параметры функции `FloatToStrF`

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 4 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;
```

Его результатом 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.

Если в процессе работы над программой возникает необходимость в создании переменной, размер которой неизвестен заранее, используют динамическую память²¹. В языке Free Pascal операции распределения памяти осуществляются по средством представленными в [табл. 2.10](#).

Обозначение Тип аргументов Тип результата Действие

<code>date</code>	без аргумента	дата-время	возвращает текущую дату
<code>now</code>	без аргумента	дата-время	возвращает текущую дату и время
<code>time</code>	без аргумента	дата-время	возвращает текущее время

Таблица 2.9. Функции для работы с датой и временем

Обозначение	Действие
<code>now</code>	

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

Таблица 2.10. Функции для работы с памятью

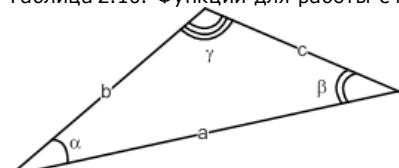


Рис. 2.6. Иллюстрация к задаче 2.1

С подробным описанием приведённых 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).

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

$$r = (a + b + c)/2$$

— полупериметр; один из углов найдём по теореме косинусов:

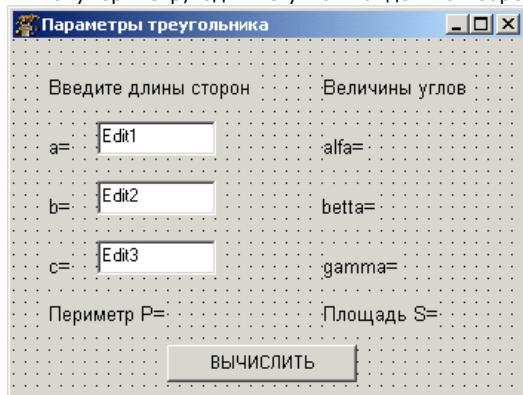


Рис. 2.7. Проект формы к задаче 2.1

Компонент Свойство Caption

Form1 Параметры треугольника

Label1 Введите длины сторон

Label2 $a =$

Label3 $b =$

Label4 $c =$

Label5 Величины углов

Label6 $\alpha =$

Label7 $\beta =$

Label8 $\gamma =$

Label9 Периметр $P =$

Label10 Площадь $S =$

Button1 ВЫЧИСЛИТЬ

Таблица 2.11. Заголовки компонентов формы, представленной на рис. 2.7

$$\cos(\alpha) = (b^2 + c^2 - a^2) / (2 \cdot b \cdot c)$$

; второй по теореме синусов:

$$\sin(\beta) = b/a \cdot \sin(\alpha)$$

; третий по формуле:

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

Решение задачи можно разбить на следующие этапы:

1. Определение значений a и b (ввод величин в память компьютера).

2. Расчёт значений S , P , α , β и γ и по приведённым выше формулам.

3. Вывод значений S , P , α , β и γ и .

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 заголовки формы и меток (свойство Caption) в соответствии с табл. 2.11. В качестве свойства Text компонентов Edit1, Edit2, Edit3 установите пробел. В результате форма должна выглядеть так, как показано на рис. 2.7.

Итак, проект формы готов. В окне программного кода среды Lazarus автоматически сформировал структуру модуля, перечислив названия основных разделов. Двойной щелчок по кнопке Вычислить приведёт к созданию процедуры TForm1.Button1Click в разделе implementation:

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

и её описание 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, beta, gamma - углы треугольника;
// S - площадь треугольника;
// r - полупериметр треугольника
//Все переменные вещественного типа.
var a, b, c, alfa, beta, gamma, S, r : real;
begin
//Из полей ввода Edit1, Edit2, Edit3 считаются 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:
// acos(x) - арккосинус x;
// sqr(x) - возведение x в квадрат.
alfa := arccos ( ( sqr ( b)+sqr ( c)-sqr ( a ) ) / 2 / b/ c );
//Вычисление значения угла beta в радианах.
//Для вычисления применяем функции:
// asin(x) - арксинус x;
beta := arcsin ( b/a * sin ( alfa ) );
//Вычисление значения угла gamma в радианах.
//Математическая постоянная определена функцией без аргумента pi.
gamma:= pi -( alfa+beta );
//Перевод из радиан в градусы.
alfa := alfa * 180/ pi;
beta := beta * 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 := 'beta= '+FloatToStrF ( beta, 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 позиции 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 были написаны всего десять команд, предназначенных для решения поставленной задачи. Остальной текст в окне редактора создаётся автоматически. В результате весь программный код (без комментариев) 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, beta, 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 );
  beta := arcsin ( b/a * sin ( alfa ) );
  gamma:= pi -( alfa+beta );
  alfa := alfa * 180/ pi;
  beta := beta * 180/ pi;
  gamma:=gamma * 180/ pi;
  Label6. Caption := 'альфа=' +FloatToStrF ( alfa, fffFixed, 3, 0 );
  Label7. Caption := 'бета=' +FloatToStrF ( beta, fffFixed, 3, 0 );
  Label8. Caption := 'гамма=' +FloatToStrF ( gamma, fffFixed, 3, 0 );
  Label9. Caption := 'Периметр Р=' +FloatToStrF ( 2 * r, fffFixed, 5, 2 );
  Label10. Caption := 'Площадь S=' +FloatToStrF ( S, fffFixed, 5, 2 );
end;
initialization
  {$I unit1.lrs}
end.

```

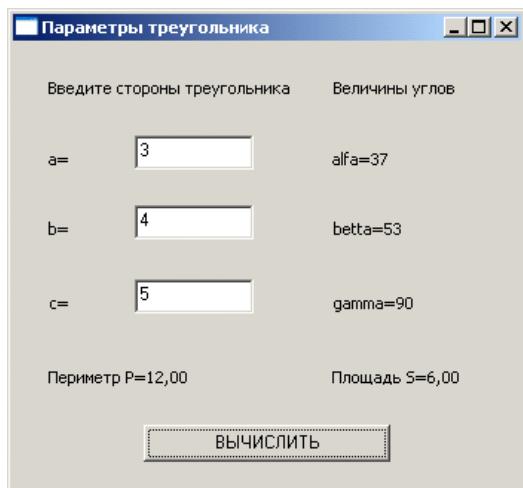


Рис. 2.8. Результат работы программы к задаче 2.1
На [рис. 2.8](#) представлено диалоговое окно, которое появится, если запустить эту программу, щёлкнув по кнопке **Вычислить**.
Теперь напишем консольное приложение для решения этой задачи. Для этого запустим текстовый редактор Geany. Выполним команду **Файл — New with Template — Pascal Source File**. В открывшемся окне реактора введём 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, beta, gamma : real;
begin
//Ввод сторон треугольника.
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:
//acos(x) — арккосинус x;
//sqr(x) — возвведение x в квадрат.
alfa := arccos ( ( sqr ( b)+ sqr ( c ) - sqr ( a ) ) / 2 / b/ c );
//Вычисление значения угла beta в радианах.
//Для вычисления применяем функции:
//asin(x) — арксинус x;

```

```

beta := arcsin( b/a * sin( alfa ) );
//Вычисление значения угла gamma в радианах.
//Математическая постоянная определена
//функцией без аргумента pi.
gamma:= pi - ( alfa+beta );
//Перевод из радиан в градусы.
alfa := alfa * 180/ pi;
beta := beta * 180/ pi;
gamma:=gamma * 180/ pi;
//Вывод площади и сторон треугольника.
writeln( 'Площадь треугольника = ', S : 6 : 2 );
writeln( ' alf= ', alfa : 6 : 2, ' bet= ', beta : 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.
Введите стороны треугольника

3 4 5

Площадь треугольника = 6.00

alf= 36.87 bet= 53.13 gam= 90.00

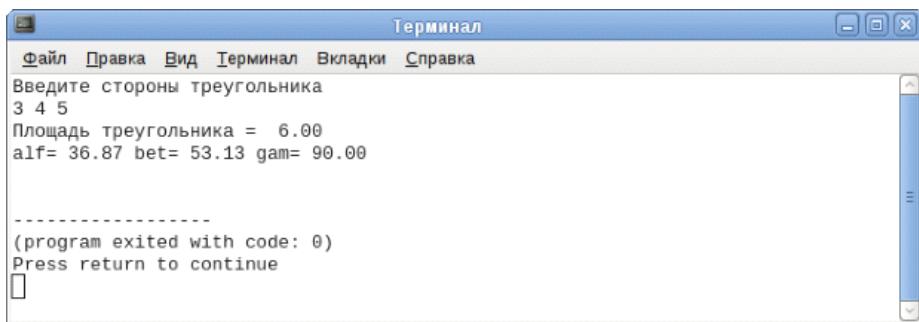


Рис. 2.9. Результаты работы консольного приложения решения задачи 2.1

Лекция 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 алгоритма¹¹. Алгоритм — это чёткое 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 составлении блок-схемы все этапы решения задачи изображаются с помощью различных геометрических фигур. Эти фигуры называют блоками и, как правило, сопровождаются текстом. Последовательность выполнения этапов указываются при помощи стрелок, соединяющих эти блоки. Типичные этапы решения задачи изображаются следующими геометрическими фигурами:

- блок начала (конца) ([рис. 3.1](#)). Текст внутри блока: "начало" ("конец");



Рис. 3.1. Блок начала (конца) алгоритма

- блок ввода (вывода) данных ([рис. 3.2](#)). Текст внутри блока: "ввод" ("вывод" или "печать") и список вводимых (выводимых) переменных;



Рис. 3.2. Блок ввода (вывода)

- блок решения, или арифметический ([рис. 3.3](#)). Внутри блока записывается действие, вычислительная операция или группа операций;

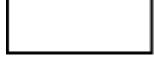


Рис. 3.3. Арифметический блок

- условный блок ([рис. 3.4](#)). Логическое условие записывается внутри блока. В результате проверки условия осуществляется выбор одного из возможных путей (ветвей) вычислительного процесса.

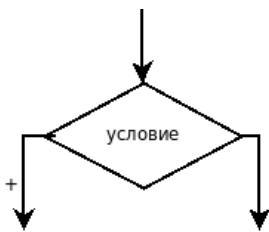


Рис. 3.4. Условный блок

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

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

Разветвляющийся процесс задаёт выполнение одного или другого оператора в зависимости от выполнения условия ([рис. 3.6](#)).

Циклический процесс задаёт многократное выполнение оператора или группы операторов ([рис. 3.7](#)).

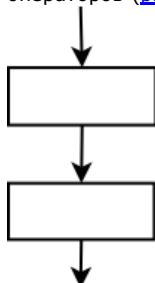


Рис. 3.5. Линейный процесс

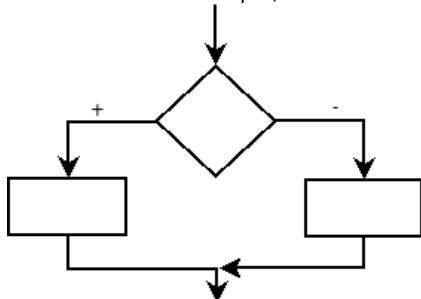


Рис. 3.6. Разветвляющийся процесс

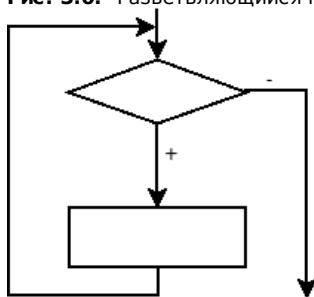


Рис. 3.7. Циклический процесс

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.

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 выражения **b**.

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

Оператор присваивания, как и любой другой оператор в языке Free Pascal, отделяется от других точкой с запятой.

3.3 Составной оператор

Составной оператор — группа операторов, отделённых 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 определённых 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:

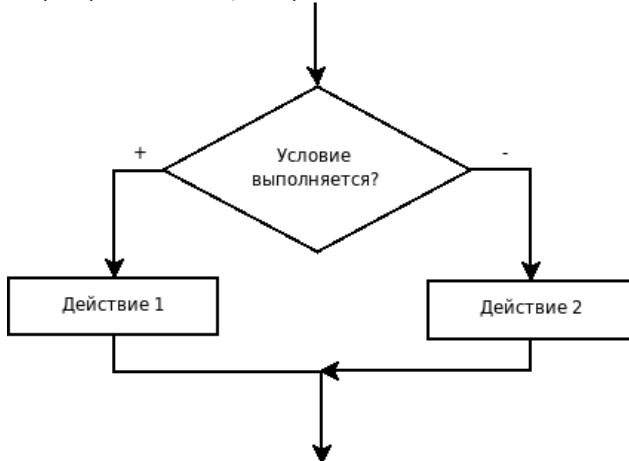


Рис. 3.8. Алгоритм условного оператора `if..then..else`

`if` условие `then` оператор_1 `else` оператор_2;
где `if..then..else` — зарезервированные слова, `условие` — выражение логического типа, `оператор_1` и `оператор_2` — любые операторы языка Free Pascal.

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 передаётся `оператору_2`.

Алгоритм, который реализован в условном операторе `if..then..else`, представлен на [рис. 3.8](#).

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, необходимо использовать составной оператор:

```
if условие then  
begin  
    оператор_1;  
    оператор_2;  
    ...  
    оператор_n;  
end  
else  
begin  
    оператор_1;  
    оператор_2;  
    ...  
    оператор_n;  
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 "усечённом" 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](#).

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

```

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

```

Условные операторы могут быть вложены друг в друга. При вложениих условных операторов всегда действует правило: альтернатива `else` считается принадлежащей ближайшему `if`, имеющему ветвь `else`. Например, в записи он принадлежит оператору `if` с [условием_1](#).



Рис. 3.9. Алгоритм условного оператора `if` без альтернативной ветви `else`

```

if условие_1 then
  if условие_2 then
    оператор_A
  else оператор_Б;

```

оператор_Б относится к [условию_2](#), а в конструкции

```

if условие_1 then
  begin
    if условие_2 then
      оператор_A;
    end
  else оператор_Б;

```

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

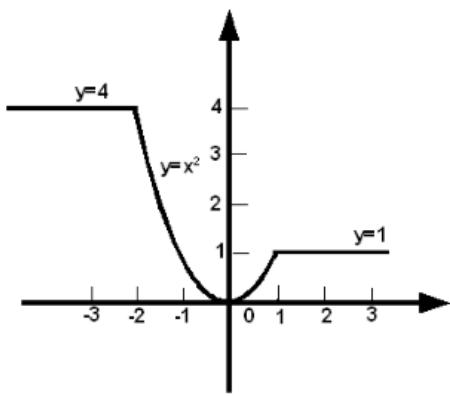
```

if x>=a and x<=b then...

```

неверна, так как фактически будет вычисляться значение выражения

$x>=(a \text{ and } x)<=b$.



[увеличить изображение](#)

Рис. 3.10. Графическое представление задачи 3.1
Рассмотрим использование оператора **if** на примерах [3.1](#).

ЗАДАЧА 3.1. Дано вещественное число

x. Для функции, график которой приведён на [рис. 3.10](#), вычислить
 $y = f(x)$

Аналитически функцию, представленную на [рис. 3.10](#), можно записать так:

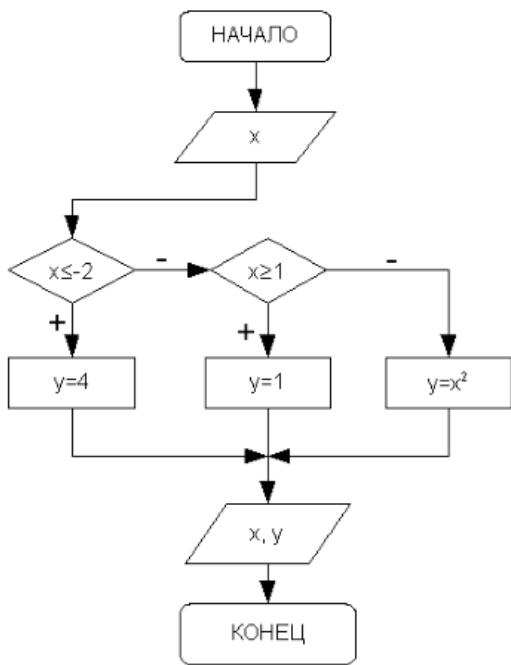
$$y(x) = \begin{cases} 4, & x \leq -2, \\ x^2, & -2 < x < 1, \\ 1, & x \geq 1. \end{cases}$$

Составим словесный алгоритм решения этой задачи:

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

Блок-схема, соответствующая описанному алгоритму, представлена на [рисунке 3.11](#).

Как видно, блок-схема нагляднее и проще для восприятия, чем словесное описание алгоритма. В дальнейшем для описания алгоритма мы часто будем использовать именно блок-схемы.



[увеличить изображение](#)

Рис. 3.11. Блок-схема алгоритма решения задачи 3.1
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.
    
```

Эту программу можно ввести в текстовом редакторе Geany (или в текстовом редакторе Free Pascal) и запустить на выполнение.

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](#), область ограничена линиями

$x = -1, x = 3, y = -2$

и

$y = 4$

. Значит, точка с координатами

$(x; y)$

будет принадлежать этой области, если будут выполняться следующие условия:

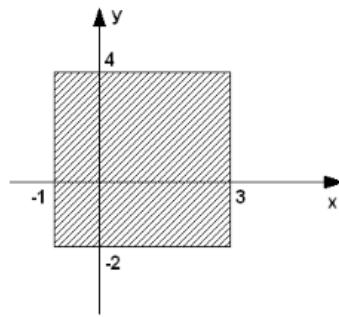
$x \geq -1, x \leq 3, y \geq -2$

и

$y \leq 4$

. Иначе точка лежит за пределами области.

Блок-схема, описывающая алгоритм решения данной задачи, представлена на [рис. 3.13](#).



[увеличить изображение](#)

Рис. 3.12. Графическое представление задачи 3.2



[увеличить изображение](#)

Рис. 3.13. Алгоритм решения задачи 3.2

Текст программы к задаче 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 области' )
    else
        writeln ( 'Точка не принадлежит области' );
end.
    
```

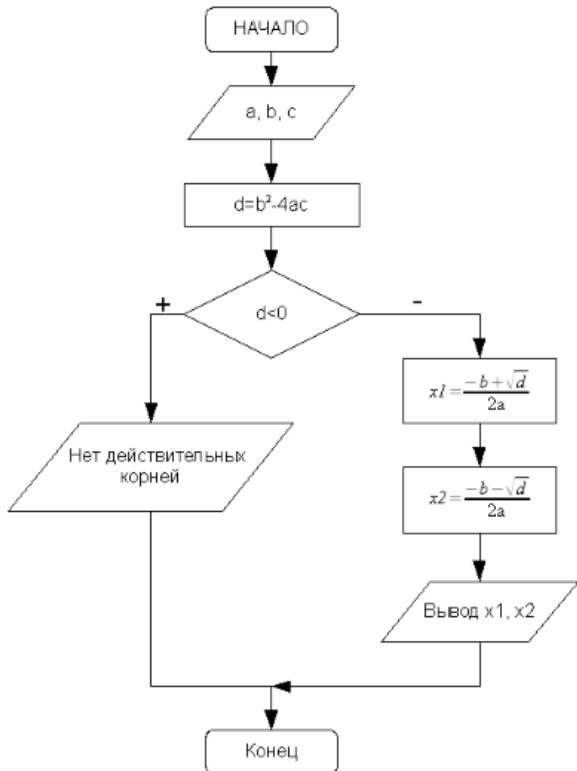
ЗАДАЧА 3.3. Написать программу решения квадратного уравнения $ax^2 + bx + c = 0$.
Исходные данные: вещественные числа

a, b

и

c

— коэффициенты квадратного уравнения.



[увеличить изображение](#)

Рис. 3.14. Алгоритм решения квадратного уравнения

Результаты работы программы: вещественные числа

x_1
и
 x_2

— 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 и b .
3. Вычисление значения дискриминанта по формуле $d = b^2 - 4ac$.
4. Если $d < 0$, то переход к п. 5, иначе переход к п. 6.
5. Вывод сообщения "Действительных корней нет" и переход к п. 8.
6. Вычисление корней x_1 и x_2 .
7. Вывод значений x_1 и x_2 на экран.
8. Конец алгоритма.

Блок-схема, соответствующая этому описанию, представлена на [рис. 3.14](#).

Текст программы, которая реализует решение квадратного уравнения:

```

{Описание переменных.}
var a, b, c, d, x1, x2 : real;
begin
  {Ввод значения коэффициентов квадратного уравнения.}
  writeln ('Введите коэффициенты квадратного 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 ('Действительных 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

x_1

и

x_2

— действительные корни квадратного уравнения — либо

x_1

и

x_2

— действительная и мнимая части комплексного числа.

Вспомогательные переменные: вещественная переменная

d

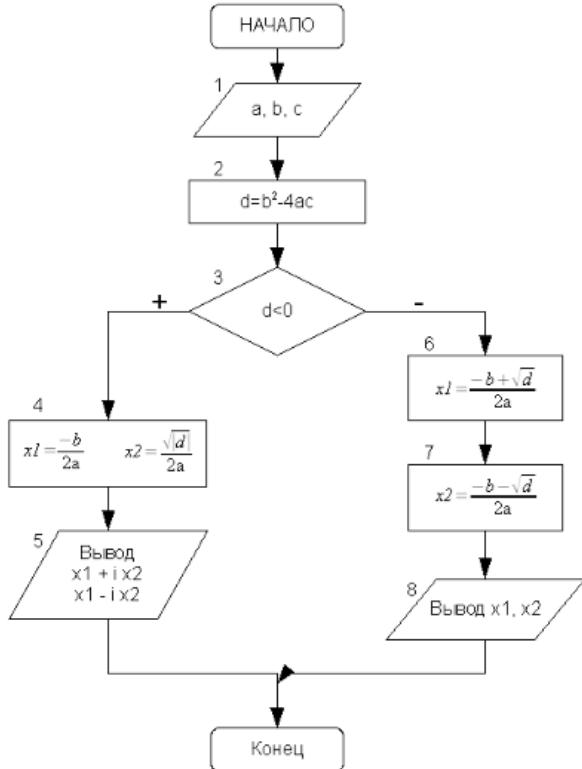
, в которой будет храниться дискриминант квадратного уравнения.

Можно выделить следующие этапы решения задачи:

1. Ввод коэффициентов квадратного уравнения и .

2. Вычисление дискриминанта по формуле .

3. Проверка знака дискриминанта. Если , то вычисление действительных корней: и вывод их на экран. При отрицательном дискриминанте выводится сообщение о том, что действительных корней нет, и вычисляются комплексные корни²¹



[увеличить изображение](#)

Рис. 3.15. Алгоритм решения задачи 3.4

$$\frac{-b}{2a} + i \frac{\sqrt{|d|}}{2a}, \frac{-b}{2a} - i \frac{\sqrt{|d|}}{2a}.$$

У обоих комплексных корней действительные части одинаковые, а мнимые отличаются знаком. Поэтому можно в переменной x_1 хранить действительную часть числа

$$-b/(2a)$$

, в переменной x_2 — модуль мнимой части

$$\sqrt{|d|}/(2a)$$

, а в качестве корней вывести x_1+ix_2 и x_1-ix_2 .

На [рис. 3.15](#) изображена блок-схема решения задачи. 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 расчёт происходит в блоке 4 (действительная часть корня записывается в переменную x_1 , модуль мнимой — в переменную x_2), а вывод — в блоке 5 (первый корень x_1+ix_2 , второй — x_1-ix_2). 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 квадратного 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

$$ax^3 + bx^2 + cx + d = 0 \quad (3.1)$$

После деления на

a уравнение (3.1) принимает канонический вид:

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

где

$$r = b/a, s = c/a, t = d/a$$

. В уравнении (3.2) сделаем замену

$$x = y - r/3$$

и получим приведённое уравнение:

$$y^3 + py + q = 0, \quad (3.3)$$

где

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

Число действительных корней приведённого уравнения (3.3) зависит от знака дискриминанта

$$D = (p/3)^3 + (q/2)^2$$

приведённого кубического уравнения ([табл. 3.1](#)).

Дискриминант	Количество действительных корней	Количество комплексных корней
$D > 0$	1	2
$D < 0$	3	—

Таблица 3.1. Количество корней кубического уравнения

Корни приведённого уравнения могут быть рассчитаны по формулам Кардано:

$$y_1 = u + v \quad (3.4)$$

$$y_2 = \frac{-u + v}{2} + \frac{u - v}{2}i\sqrt{3}$$

$$y_3 = \frac{-u + v}{2} - \frac{u - v}{2}i\sqrt{3}$$

где

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

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:

$$y_1 = 2\sqrt[3]{\rho} \cos\left(\frac{\varphi}{3}\right), \quad (3.5)$$

$$y_2 = 2\sqrt[3]{\rho} \cos\left(\frac{\varphi}{3} + \frac{2\pi}{3}\right),$$

$$y_3 = 2\sqrt[3]{\rho} \cos\left(\frac{\varphi}{3} + \frac{4\pi}{3}\right),$$

где

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

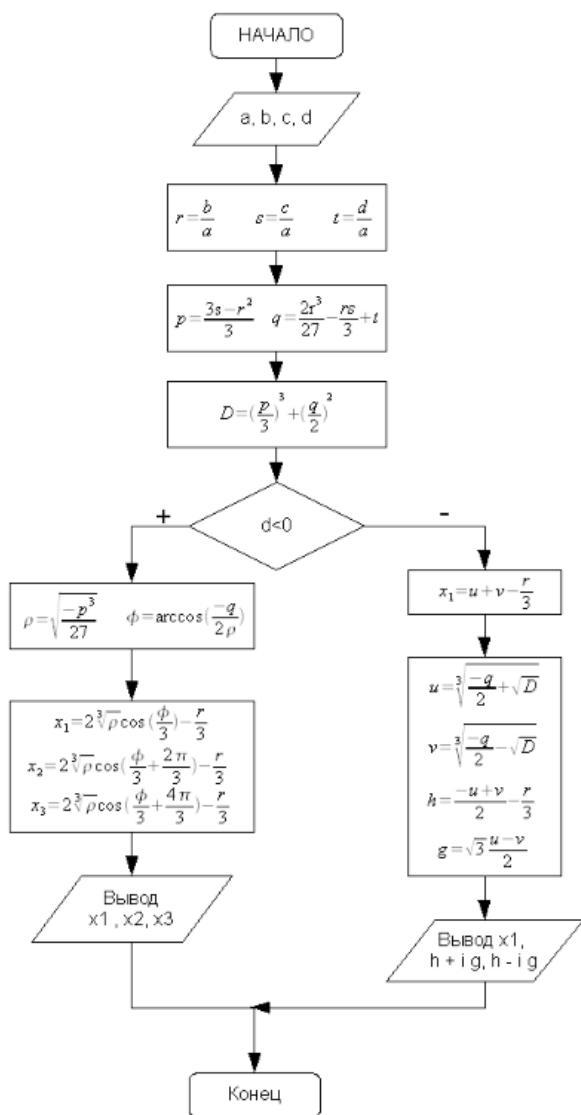
Таким образом, при положительном дискриминанте кубического уравнения (3.3) расчёт корней будем вести по формулам (3.4), а при отрицательном — по формулам (3.5).

После расчёта корней приведённого уравнения (3.3) по формулам (3.4) или (3.5) необходимо по формулам

$$x_k = y_k - \frac{r}{3}, k = 1, 2, 3 \dots$$

перейти к корням заданного кубического уравнения (3.1).

Блок-схема решения кубического уравнения представлена на [рис. 3.16](#).



[увеличить изображение](#)

Рис. 3.16. Алгоритм решения кубического уравнения

Описание блок-схемы. В блоке 1 вводятся коэффициенты кубического уравнения, в блоках 2–3 рассчитываются коэффициенты канонического и приведённого 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 расчёт идет по формулам (3.4) (блок 9, 10). Блоки 8 и 11 предназначены для вывода результатов

на экран.

Текст программы с комментариями приведён ниже 5.2.

```
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 );
    //Расчёт коэффициентов канонического уравнения по 3.2.
    r := b/a; s := c/a; t := d/a;
    //Вычисление коэффициентов приведённого 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) * sqrt( p / 3 ) + sqrt( 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 );
            //Следующие два оператора реализуют расчёт 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 части 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.
end.
```

ЗАДАЧА 3.6. Заданы коэффициенты a , b и с биквадратного уравнения

$$ax^4 + bx^2 + c = 0$$

. Найти все его действительные корни.

Входные данные: a , b , c . Выходные данные: x_1 , x_2 , x_3 , x_4 .

Для решения биквадратного уравнения необходимо заменой

$$y = x^2$$

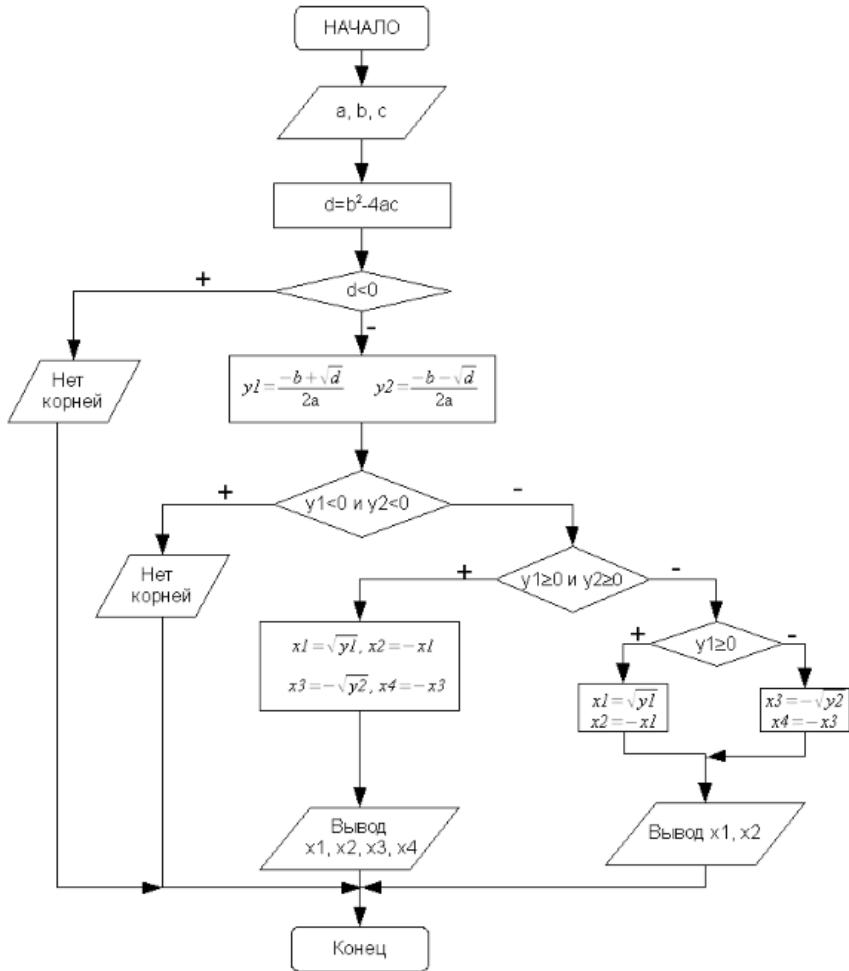
привести его к квадратному уравнению

$$ay^2 + by + c = 0$$

и решить это уравнение.

Опишем алгоритм решения этой задачи (рис. 3.17):

1. Ввод коэффициентов биквадратного уравнения и (блок 1).
2. Вычисление дискриминанта уравнения (блок 2).
3. Если (блок 3), вывод сообщения, что корней нет (блок 4), а иначе определяются корни соответствующего квадратного уравнения и (блок 5).
4. Если и (блок 6), то вывод сообщения, что корней нет (блок 7).
5. Если и (блок 8), то вычисляются четыре корня по формулам (блок 9) и выводятся значения корней (блок 10).
6. Если условия 4) и 5) не выполняются, то необходимо проверить знак . Если (блок 11), то вычисляются два корня по формуле (блок 12), иначе (если) вычисляются два корня по формуле (блок 13).
7. Вывод вычисленных значений корней (блок 14).



[увеличить изображение](#)

Рис. 3.17. Алгоритм решения биквадратного уравнения

Текст программы на языке Free Pascal с комментариями:

```

//Описание переменных:
//a,b,c - коэффициенты биквадратного уравнения,
//d - дискриминант,
//x1,x2,x3,x4 - корни биквадратного уравнения,
//y1,y2 - корни квадратного уравнения ay^2+by+c=0.
var
    a, b, c, d, x1, x2, x3, x4, y1, y2 : real;
begin
    //Ввод коэффициентов уравнения.
    writeln ('Введите коэффициенты биквадратного 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
        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 биквадратного уравнения.
                    x1:= sqrt ( y1 );
                    x2:=-x1;
                    x3:=-sqrt ( y2 );
                    x4:=x3;
                end;
        end;
end.
    
```

```

x3:= sqrt ( y2 );
x4:=-sqrt ( y2 );
//Вывод корней биквадратного 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 выражение of
    значение_1 : оператор_1;
    значение_2 : оператор_2;
    ...
    значение_N : оператор_N
else
    альтернативный_оператор
end;

```

Здесь **выражение** — переменная перечислимого типа (включая **char** и **boolean**), **значение_1**, **значение_2**,..., **значение_N** — 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. Оператор работает следующим образом. Вычисляется значение **выражения**. Затем выполняется **оператор**, помеченный **значением**, совпадающим со значением **выражения**. То есть, если **выражение** принимает **значение_1**, то выполняется **оператор_1**. Если выражение принимает **значение_2**, то выполняется **оператор_2** и так далее. Если **выражение** не принимает ни одно из **значений**, то выполняется **альтернативный_оператор**, расположенный после ключевого слова **else**.

Альтернативная ветвь **else** может отсутствовать, тогда оператор имеет вид:

```

case выражение of
    значение_1 : оператор_1;
    значение_2 : оператор_2;
    ...
    значение_N : оператор_N;
end;

```

Кроме того, в операторе case допустимо использование составного оператора.

Например:

```

case выражение of
    значение_1 : begin оператор_A; оператор_B; end;
    значение_2 : begin оператор_C; оператор_D; оператор_E; end;
    ...
    значение_N : оператор_N;
end;

```

Рассмотрим применение оператора варианта на следующих примерах.

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 первое 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 на [рис. 3.18](#).

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

```

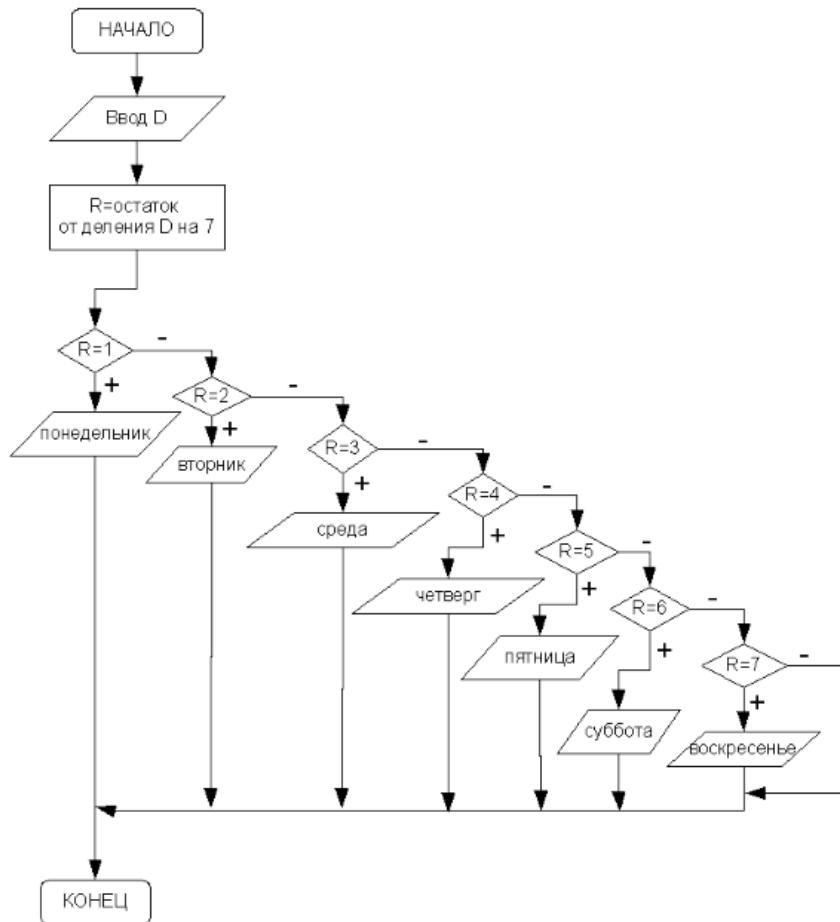
var d : byte;
begin
    write ( 'Введите число D= ' );
    readln ( D );
    case D mod 7 of {Вычисляется остаток от деления D на 7.}

```

```

{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.

```



[увеличить изображение](#)

Рис. 3.18. Алгоритм решения задачи 3.7

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 четырёх 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 ветвь `else`.

```

var m: byte;
begin
  write ( 'Введите номер месяца m= ' ); readln (m);
  //Проверка области допустимых значений переменной m.
  case m of
    //В зависимости от значения m на печать
    //выводится название времени года.
    1..2, 12 : 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.
```

3.4.3 Обработка ошибок. Вывод сообщений в среде Lazarus

Понятно, что чем меньше в программе ошибок, тем она лучше. В очень хороший 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 отображён 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.

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

Таблица 3.2. Тип окна сообщения

Константа	Кнопка
<code>mbYes</code>	Да
<code>mbNo</code>	Нет
<code>mbOk</code>	Ок
<code>mbCancel</code>	Отмена
<code>mbAbort</code>	Прервать
<code>mbRetry</code>	Повторить
<code>mbIgnore</code>	Пропустить
<code>mbHelp</code>	Помощь

Таблица 3.3. Тип кнопки в окне сообщения



Рис. 3.19. Конструирование формы для решения квадратного уравнения

Вернёмся 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 проект²⁴([рис. 3.19](#)).

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`²⁵. 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 нулю²⁶.

Для того чтобы не допустить подобные ошибки, необходимо контролировать данные, вводимые пользователем. Применим для этой цели встроенную процедуру `Val(S,X,Kod)`, которая преобразовывает строку **S** в число. Тип этого числа (целое или вещественное) зависит от типа переменной **X**. Если преобразование 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 приведён 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 квадратного 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 квадратного 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.

```

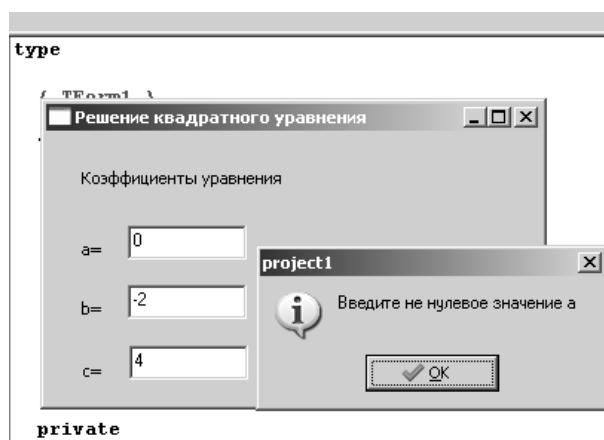


Рис. 3.20. Обработка ошибки ввода данных — коэффициент а равен 0
 Решение квадратного уравнения
 $3x^2 + 4x + 5 = 0$

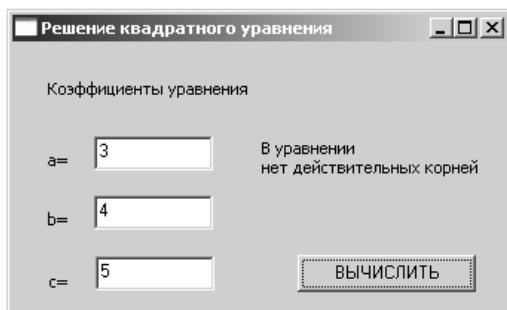


Рис. 3.21. Решение квадратного уравнения(корней нет)

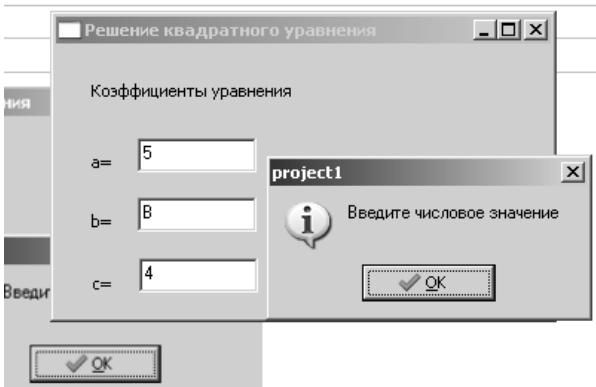


Рис. 3.22. Обработка ошибки ввода данных — в поле ввода строка, которую невозможно преобразовать в число

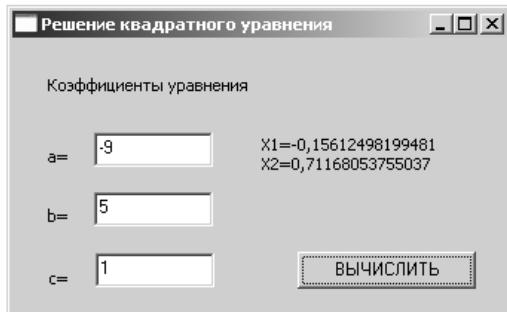


Рис. 3.23. Вычисление корней квадратного уравнения

Результаты работы программы показаны на [рис. 3.20](#), [рис. 3.21](#), [рис. 3.22](#), [рис. 3.23](#). 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 введённый 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 щёлкнуть возле события нажатия клавиши OnKeyPress, то будет создана процедура обработки этого события. Текст процедуры с комментариями:

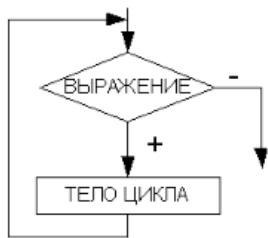
```
//Обработка события ввод символа в поле Edit2.
procedure TForm1.Edit2KeyPress ( Sender : TObject;
var Key : Char );
begin //Если символ не принадлежит
      //множеству допустимых символов,то
if not ( Key in [#8, ',', '_','0 .. '9 '] ) then
begin //выдать сообщение и
      MessageDlg ( 'Введите числовое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

3.5 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. Переменные, которые изменяются внутри цикла и влияют на его окончание, называются параметрами цикла. 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, определённых ещё до первого выполнения тела цикла. В языке Free Pascal для удобства программиста предусмотрены три оператора, реализующих циклический процесс: **while**, **repeat..until** и **for**.

3.5.1 Оператор цикла с предусловием **while..do**

На [рис. 3.24](#) изображена блок-схема алгоритма цикла с предусловием. Оператор, реализующий этот алгоритм, в языке Free Pascal имеет вид:
while выражение **do** оператор;



[увеличить изображение](#)

Рис. 3.24. Алгоритм циклической структуры с предусловием

Выражение должно быть логической константой, переменной или логическим выражением.

Работает оператор `while` следующим образом. Вычисляется значение выражения.

Если оно истинно (`True`), выполняется оператор. Выполнение цикла заканчивается, если условие ложно, после этого 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.

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

$$[0; \pi]$$

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.
```

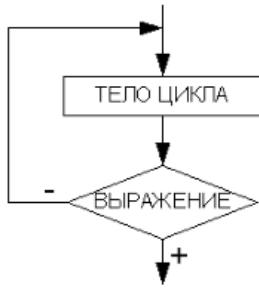
В результате работы данного фрагмента программы на экран последовательно будут выводиться сообщения со значениями переменных `x` и `y`:

```

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

3.5.2 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. 3.25](#)). Сначала выполняются операторы, являющиеся телом цикла, после чего проверяется условие, если последнее ложно, то цикл повторяется. Выполнение цикла прекратится, если условие станет истинным.



[увеличить изображение](#)

Рис. 3.25. Алгоритм циклической структуры с постусловием
В языке Free Pascal цикл с постусловием реализован конструкцией

repeat
 оператор;
 until выражение;

или, если тело цикла состоит более чем из одного оператора
repeat

```

    оператор_1;
    оператор_2;
    ...
    оператор_N;
until выражение;

```

Выражение должно быть логическим.

Работает цикл следующим образом. В начале выполняется оператор, представляющий собой тело цикла. Затем вычисляется значение **выражения**. Если оно ложно (**False**), оператор тела цикла выполняется ещё 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 значения параметра цикла.}
    until x > pi; {Закончить работу цикла,}
    {когда параметр превысит конечное значение.}
end.

```

3.5.3 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 параметр_цикла:=начальное_значение to конечное_значение do
    оператор;
for параметр_цикла:=конечное_значение downto начальное_значение do
    оператор;

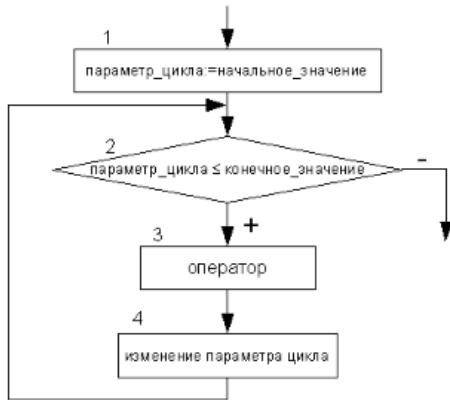
```

где оператор — любой оператор языка, параметр_цикла — имя переменной целочисленного или перечислимого

типов, начальное_значение и конечное_значение должны быть того же типа, что и параметр_цикла.

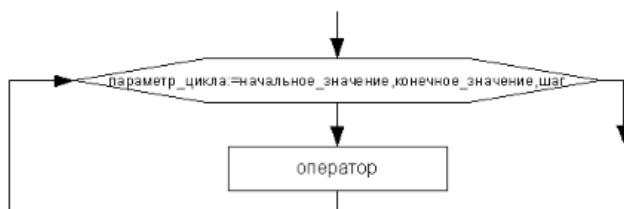
Шаг изменения цикла **for** всегда постоянен и равен интервалу между двумя ближайшими значениями типа параметра цикла (при целочисленном значении параметра цикла шаг равен 1).

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



[увеличить изображение](#)

Рис. 3.26. Алгоритм работы цикла for..do



[увеличить изображение](#)

Рис. 3.27. Представление цикла for..do с помощью блок-схемы
`for параметр_цикла:=начальное_значение to конечное_значение do
begin
 оператор_1;
 оператор_2;
 ...
 оператор_N;
end;`

Опишем алгоритм работы цикла for..do (рис. 3.26).

- Параметру_цикла присваивается начальное_значение.
- Если значение параметра_цикла превосходит конечное_значение, то цикл завершает свою работу. В противном случае выполняется п. 3.
- Выполняется оператор.
- Значение параметра_цикла изменяется на соответствующий шаг и осуществляется переход к п. 2, и т. д.

Понятно, что этот алгоритм представляет собой цикл с предусловием.

В дальнейшем, чтобы избежать создания слишком громоздких алгоритмов, в блок-схемах цикл for будем изображать так, как показано на рис. 3.27 и.

Фрагмент подпрограммы, приведённый далее, демонстрирует применение цикла for:

```

vari : integer; c : char;
begin
    {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 а до r.}
    for c := 'a' to 'r' do
        writeln ( c );
end.

```

Вернёмся к задаче вывода значений функции

$$y = e^{\sin(x)} \cos(x)$$

на отрезке

$$[0; \pi]$$

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

x_n

до
 x_k
, изменяясь с шагом
 dx
, тогда количество повторений тела цикла можно определить по формуле:

$$n = \left(\frac{x_k - x_n}{dx} \right) + 1 \quad (3.6)$$

округлив результат деления до целого числа. Следовательно, фрагмент программы вывода значений функции

$$y = e^{\sin(x)} \cos(x)$$

на отрезке

$$[0; \pi]$$

с шагом 0.1 будет иметь вид:

```
var i, n : integer; x, y : real;
begin
  n:=round (( pi - 0)/0.1)+1; {Количество повторений цикла.}
  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.}
      y:=exp ( sin ( x ) ) * cos ( x ); {Вычисление значения функции }
      {для соответствующего значения аргумента.}
      writeln ('x= ', x, ' y= ', y );
      x:=x + 0.1; {Вычисление нового}
      {значения аргумента.}
    end; {Конец цикла.}
end.
```

3.5.4 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 передаёт управление оператору с меткой.

`метка : оператор;`

Обычно применение оператора `goto` приводит к усложнению программы и затрудняет отладку. Использование оператора нарушает принцип структурного программирования, согласно которому все блоки, составляющие программу, должны иметь только один вход и один выход. В большинстве алгоритмов применения оператора `goto` можно избежать, в нашей книге мы будем его использовать, но читатель должен знать о наличии этого оператора в языке.

Операторы `break` и `continue` используют только внутри циклов.

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.

3.5.5 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](#).

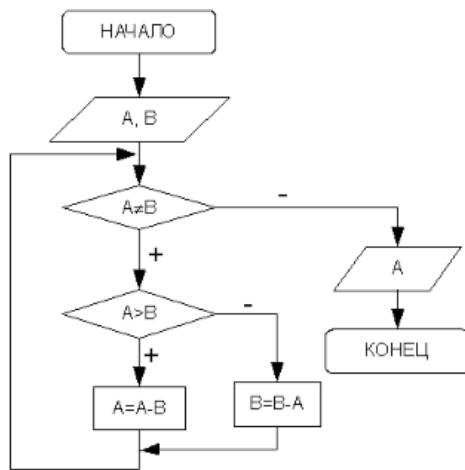
В блок-схеме решения задачи, представленной на [рис. 3.28](#), для решения поставленной задачи используется цикл с предусловием, то есть тело цикла повторяется до тех пор, пока A не равно B. Следовательно, при создании программы воспользуемся циклом `while..do`.

Исходные данные Первый шаг Второй шаг Третий шаг НОД(A,B)=5

A=25	A=10	A=10	A=5
------	------	------	-----

B=15	B=15	B=5	B=5
------	------	-----	-----

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



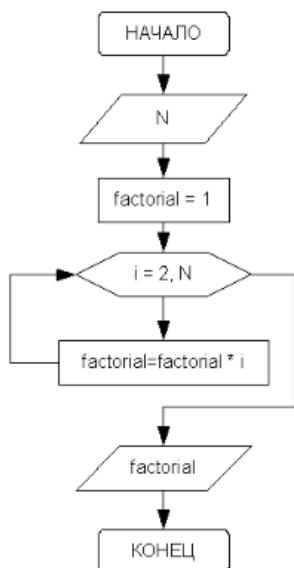
[увеличить изображение](#)

Рис. 3.28. Алгоритм поиска наибольшего общего делителя двух чисел
Программа на языке Free Pascal, реализующая поставленную задачу:

```

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 А больше, чем В, то уменьшить его значение на В,}
  if a>b then
    a:=a-b
  {иначе уменьшить значение числа В на А.}
  else
    b:=b-a;
  writeln ( 'НОД= ',a);
end.

```



[увеличить изображение](#)

Рис. 3.29. Алгоритм вычисления факториала
ЗАДАЧА 3.10. Вычислить факториал числа

$$N(N! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot N).$$

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. 3.29](#).

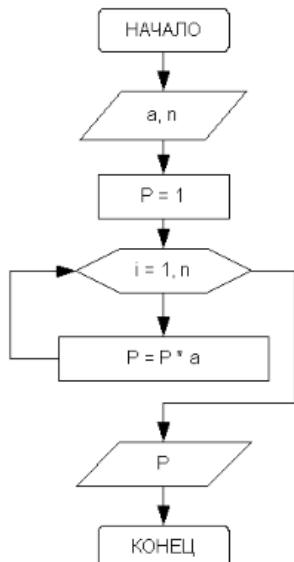
Итак, вводится число **N**. Переменной **factorial**, предназначеннной для хранения значения произведения последовательности чисел, присваивается начальное значение, равное единице. Затем организуется цикл, параметром которого выступает переменная **i**. Если значение параметра цикла меньше или равно **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 на печать выводится значение переменной **factorial**, которая была вычислена в теле цикла.

Ниже представлен 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.

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. 3.30](#).



[увеличить изображение](#)

Рис. 3.30. Алгоритм возведения вещественного числа в целую степень

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](#) отображён 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.

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

Таблица 3.5. Процесс возведения числа **a** в степень **n**

Далее приведён 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

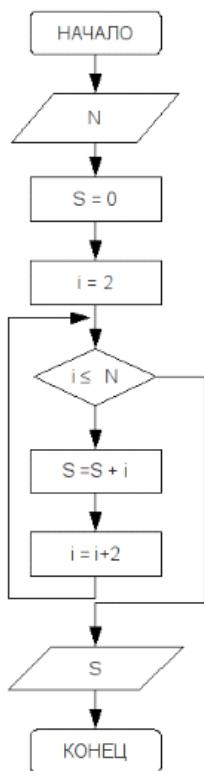
Входные данные: **N** — целое число.

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 определённом 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, предназначеннной для накапливания суммы, необходимо присвоить значение нуль (**s=0**). Блок-схема решения этой задачи представлена на [рис. 3.31](#).

Так как параметр цикла **i** изменяется с шагом 2, в блок-схеме, построенной для решения данной задачи ([рис. 3.31](#)), использован цикл с предусловием, который реализуется при составлении программы с помощью оператора **while..do**:



[увеличить изображение](#)

Рис. 3.31. Алгоритм вычисления суммы чётных натуральных чисел

```

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.

```

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 чётное, следовательно,}
    {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 $n=7$. Несложно заметить, что при нечётных значениях параметра цикла значение переменной, предназначеннной для накапливания суммы, не изменяется.

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

Таблица 3.6. Суммирование чётных чисел

ЗАДАЧА 3.13. Дано натуральное число

N

. Определить

K

— количество делителей этого числа, меньших его (например, для

$N = 12$

делители 1, 2, 3, 4, 6. Количество

$K = 5$

).

Входные данные: N — целое число.

Выходные данные: целое число K — количество делителей N .

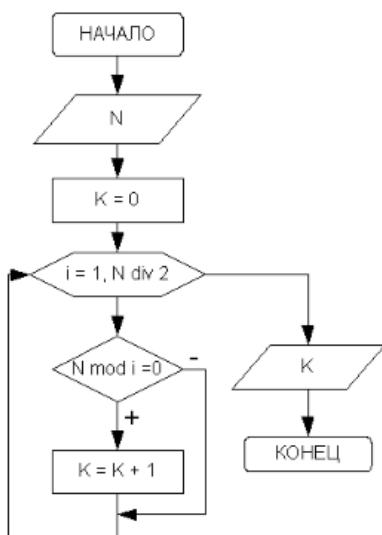
Промежуточные переменные: i — параметр цикла, возможные делители числа N .

В блок-схеме, изображённой на [рис. 3.32](#), реализован следующий алгоритм: в переменную K , предназначенную для подсчёта 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.

В [таблице 3.7](#) отображены результаты тестирования алгоритма при определении делителей числа $N=12$.

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

Таблица 3.7. Определение количества делителей числа N



[увеличить изображение](#)

Рис. 3.32. Алгоритм подсчёта делителей натурального числа
Текст программы, соответствующий описанному алгоритму:

```

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
    k:=k+1; {увеличить счетчик на единицу.}
writeln (' K= ',k);
end.

```

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

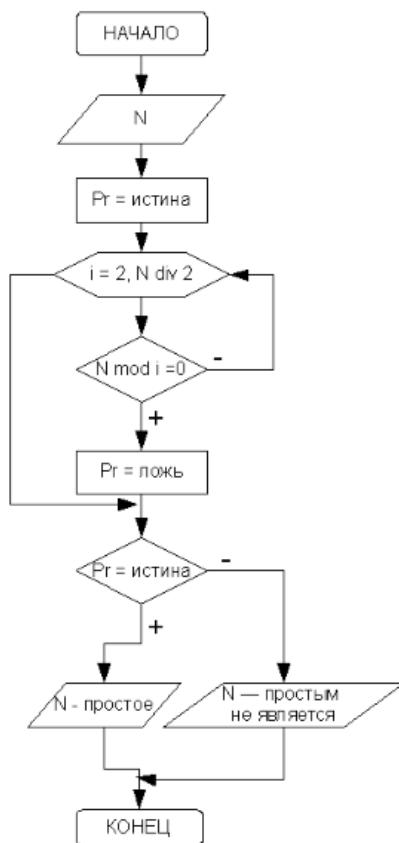
N. Число 13 — простое, так как делится только на 1 и 13, **N** = 12 не является простым, так как делится на 1, 2, 3, 4, 6 и 12.

Входные данные: **N** — целое число.

Выходные данные: сообщение.

Промежуточные данные: **i** — параметр цикла, возможные делители числа **N**.

Алгоритм решения этой задачи ([рис. 3.33](#)) заключается в том, что необходимо определить, есть ли у числа **N** делители среди чисел от 2 до **N/2**. Если делителей нет — число простое. 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, аварийно покидаем цикл.



[увеличить изображение](#)

Рис. 3.33. Алгоритм определения простого числа

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**:

```

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 и}
    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

$N \leq M$

. Алгоритм решения данной задачи представлен на [рис. 3.34](#).

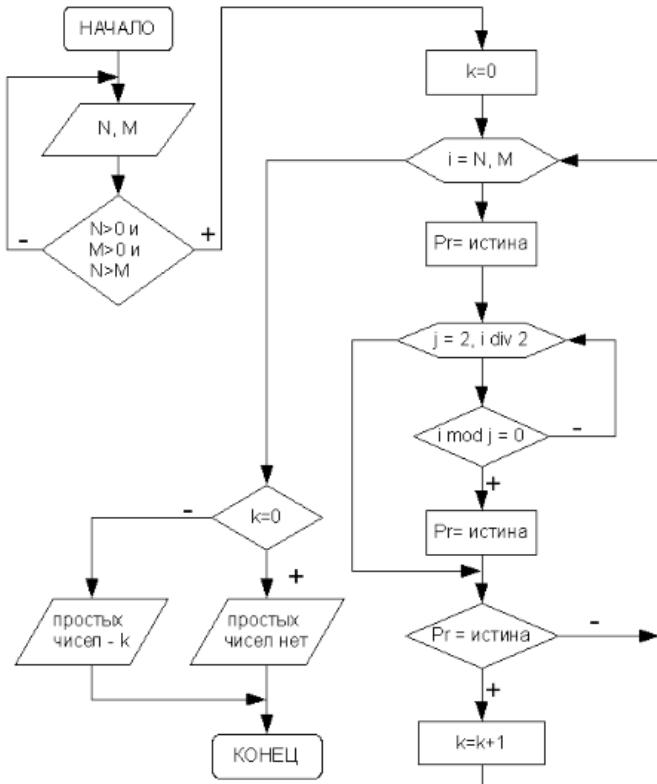
Обратите внимание, что здесь осуществляется проверка корректности ввода исходных данных. Если границы интервала не положительны, или значение N превышает M , ввод данных повторяется в цикле с постусловием до тех пор, пока не будут введены корректные исходные данные. Далее для каждого числа из указанного интервала (параметр i принимает значения от N до M) происходит проверка. Если число является простым, то переменная k увеличивается на единицу. Подробно определение простого числа описано в задаче 3.14.

Программа на языке Free Pascal, реализующая алгоритм подсчёта 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.}
for i := N to M do {Параметр i принимает значения от N до M.}
begin
    {Определение простого числа.}
    Pr:= true;
    for j := 2 to i div 2 do
        if i mod j = 0 then
            begin
                Pr:= false;
                break;
            end;
    {Если число простое, увеличиваем количество на 1.}
    if Pr then
        k:=k+1;
    end;
if k=0 then writeln ( 'Простых чисел в диапазоне нет ' )
else writeln ( 'Простых чисел в диапазоне ', k );
end.

```



[увеличить изображение](#)

Рис. 3.34. Алгоритм определения простых чисел в заданном интервале
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](#).

Алгоритм определения количества цифр в числе представлен на [рис. 3.35](#).

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

Таблица 3.8. Определение количества цифр числа

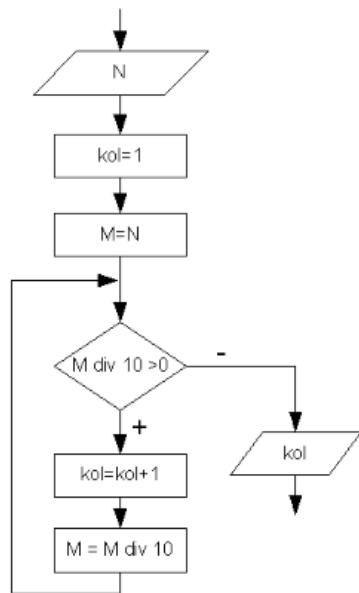
Текст программы, реализующей данную задачу, можно записать так:

```

var
  M,N: longint;
  kol : word;
begin
  {Так как речь идёт 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 нацело на 10.}
  begin
    kol := kol +1; {Счётчик количества цифр.}
    M:=M div 10; {Изменение числа.}
  end;
end;

```

```
writeln ('kol= ', kol);
end.
```



[увеличить изображение](#)

Рис. 3.35. 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 1 101 111 011 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 12¹².

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, нужно вывести на экран номер разряда, который занимает эта цифра. Процесс определения текущей цифры числа $N=120405$ представлен в [таблице 3.9](#).

Блок-схема алгоритма решения данной задачи показана на [рис. 3.36](#).

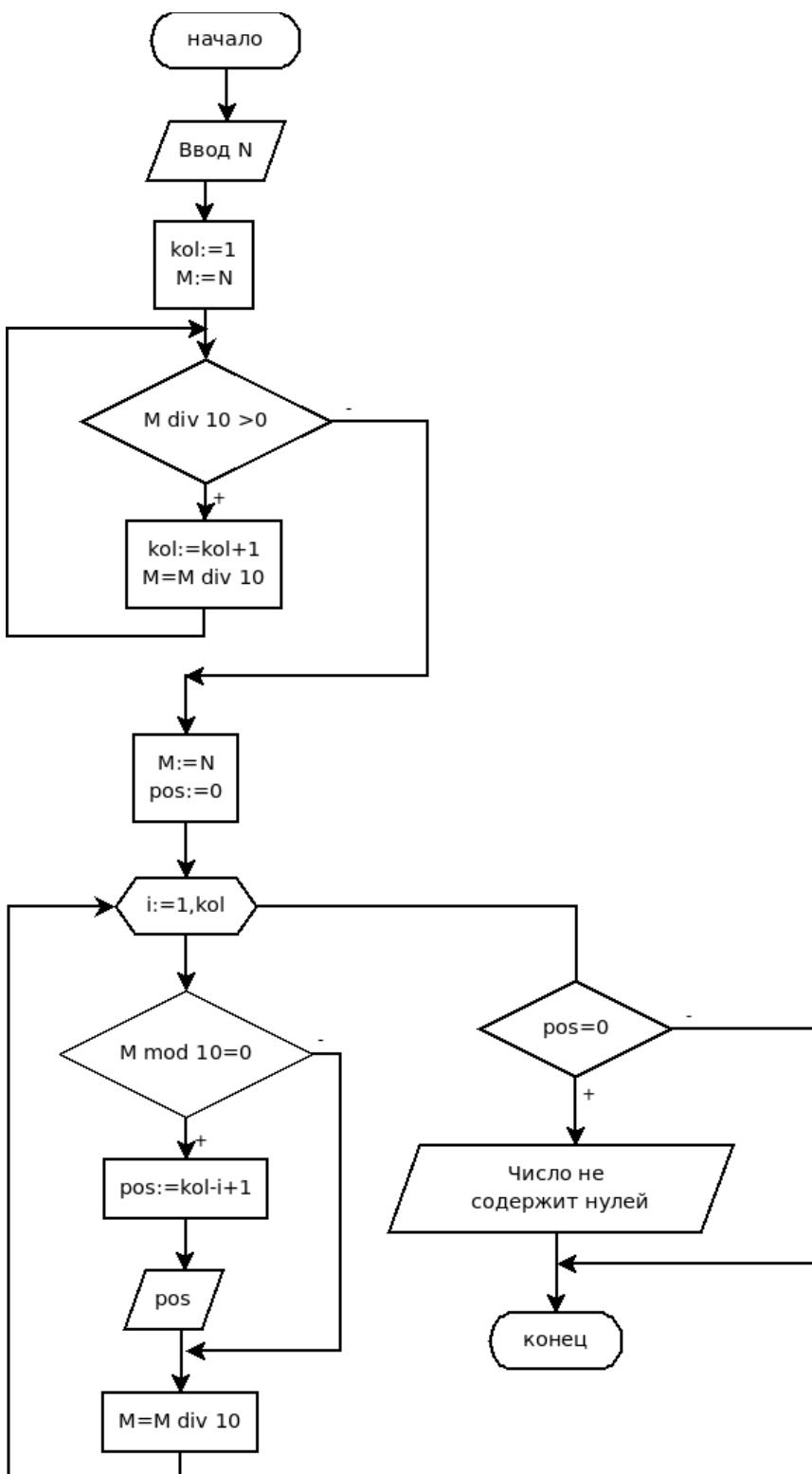


Рис. 3.36. Алгоритм решения задачи 3.17

i	Число M	Цифра	Номер позиции
1	120405 div 10=12040	120405 mod 10=5	6
2	12040 div 10=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 10=0	1 mod 10=1	1

Таблица 3.9. Определение текущей цифры числа
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; {Счётчик количества цифр.}
    M:=M div 10; {Изменение числа.}
end;
writeln ( ' kol= ', kol );
M:=N;
pos := 0; {Пусть в числе нет нулей.}
for i :=1 to kol do
begin
    {Выделение цифры из числа и сравнение её с нулем.}
    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.

```

3.5.6 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. 3.37](#).

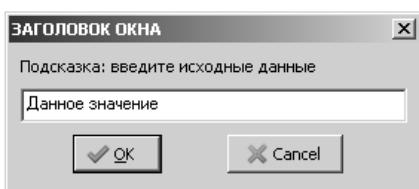


Рис. 3.37. Окно ввода

У пользователя есть возможность изменять текст в поле ввода. Щелчок по кнопке **OK** приведёт 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.

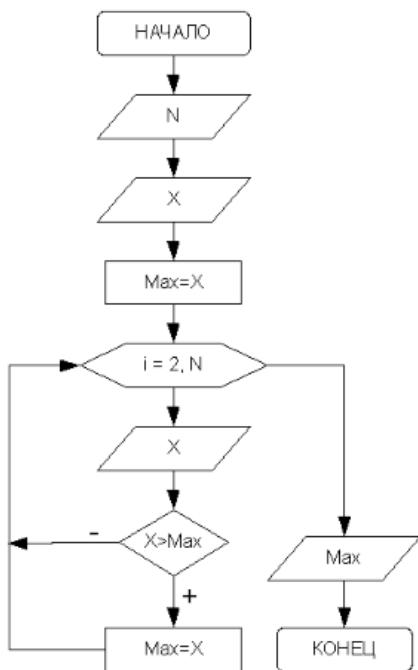
Промежуточные переменные: **i** — параметр цикла, номер вводимого элемента последовательности.

Алгоритм поиска наибольшего элемента в последовательности следующий ([рис. 3.38](#)).

В памяти компьютера отводится ячейка, например, с именем **Max**, в которой будет храниться наибольший элемент последовательности — максимум. 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 предпринимаем. Потом переходим к вводу следующего элемента последовательности, и алгоритм повторяется с начала. В результате, в ячейке **Max** будет храниться элемент последовательности с наибольшим значением¹¹.

Разместим на форме объект типа надпись **Label1** и кнопку **Button1** ([см. рисунок 3.39](#)).

Щелчок по кнопке приведёт к выполнению следующей процедуры:



[увеличить изображение](#)

Рис. 3.38. Алгоритм поиска наибольшего числа в последовательности

```

procedure TForm1.Button1Click ( Sender : TObject );
var i,N: integer; max,X: real; S : string; begin
    //Ввод количества элементов последовательности.
    S:= InputBox ( 'Ввод ',
    'Введите_количество элементов в последовательности.', '0' );
    N:= StrToFloat ( S );
    //Ввод первого элемента последовательности.
    S:= InputBox ( 'Ввод элементов последовательности',
    'Введите число.', '0' );
    X:= StrToFloat ( S );
    //Предположим, что первый элемент максимальный — Max=X.
    max:=X;
    //Параметр цикла принимает стартовое значение i=2,
    //и 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 найдётся элемент, превышающий максимум,
//записать его в ячейку Max —
//теперь он предполагаемый максимум.
if X>max then max:=X;
end;
//Вывод наибольшего элемента последовательности.
MessageDlg ( 'Значение наибольшего элемента - ' +
+FloatToStr ( max ), MtInformation, [ mbOk ], 0 );
end;

```

Результаты работы программы представлены на [рис. 3.39, 3.40, 3.41, 3.42.](#)

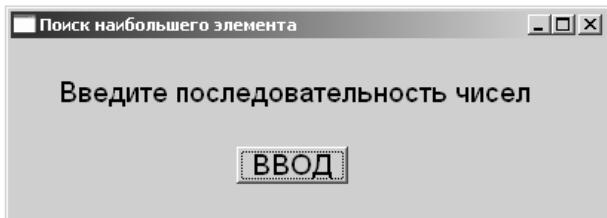


Рис. 3.39. Первое окно диалога к задаче 3.18

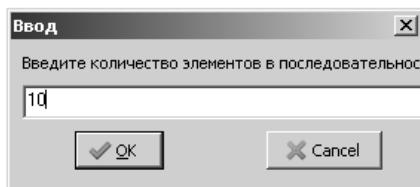


Рис. 3.40. Второе окно диалога к задаче 3.18

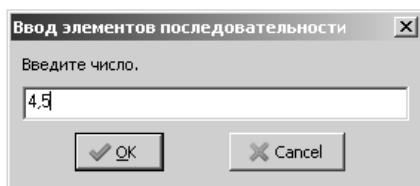


Рис. 3.41. Третье окно диалога к задаче 3.18

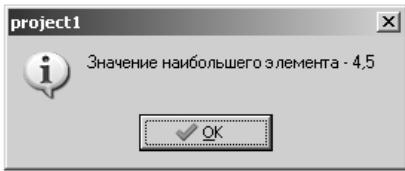
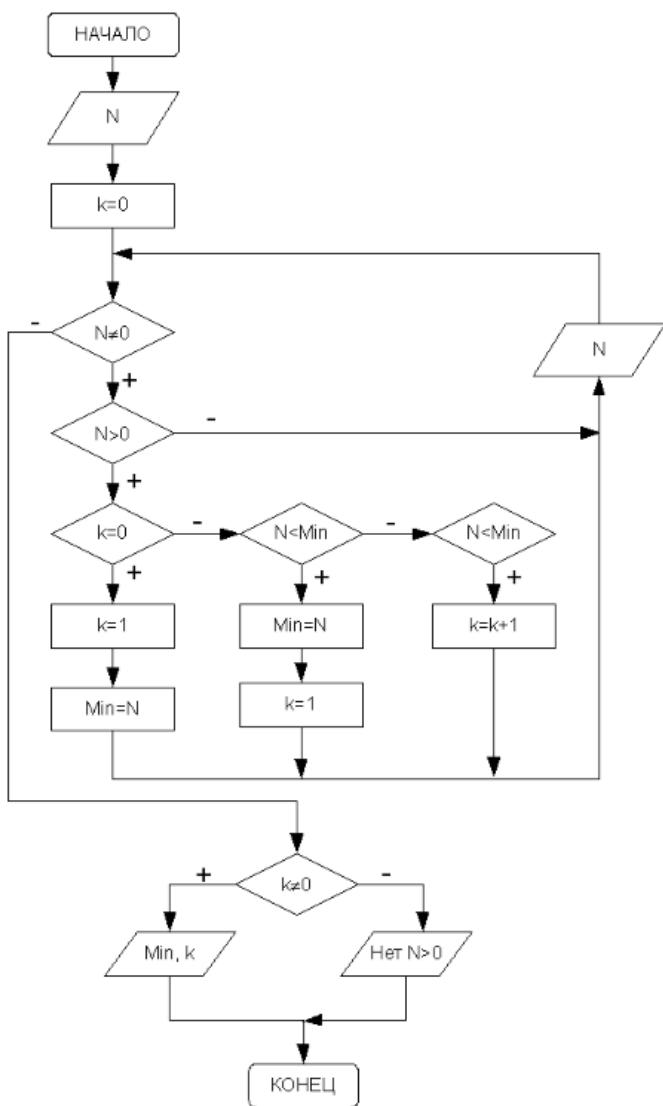


Рис. 3.42. Результат работы программы к задаче 3.18

ЗАДАЧА 3.19. Вводится последовательность целых чисел, 0 — конец последовательности. Найти наименьшее число среди положительных, если таких значений несколько, определить, сколько их. Блок-схема решения задачи приведена на [рис. 3.43](#).



[увеличить изображение](#)

Рис. 3.43. Алгоритм поиска минимального положительного числа в последовательности

Далее приведён текст подпрограммы с подробными комментариями². Подпрограмма

выполняется при обращении к кнопке Button1, предварительно размещённой 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 );
//Предположим, что положительных чисел нет — K=0. В переменной
//K будет храниться количество минимумов среди положительных
//чисел.
  k := 0;
  //Пока введённое число не равно нулю, выполнять тело цикла.
  while N<>0 do
  begin
    //Проверяем, является ли введённое 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 и обнуляем счетчик.
      else if N<min then
      begin
        min:=N;
        k := 1;
      end
      //Если элемент равен минимуму,
      //увеличиваем количество минимумов на 1.
      else if N=min then k:=k+1;
    end;
    //Ввод следующего элемента последовательности.
    S:= InputBox ( 'Ввод элементов последовательности',
      'введите число. 0 - конец последовательности', '0' );
    N:= StrToInt ( S );
  end; //Конец цикла.
  if k<>0 then
    //Если значение счётчика 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* хранится текущий элемент последовательности, а в переменной *B* — предыдущий. Введём *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 последовательности (

i ≠ 1

), то проверяем знак произведения

A · B

(текущего и предыдущего элемента последовательности). Если произведение отрицательно, то счётчик *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, приведён далее.

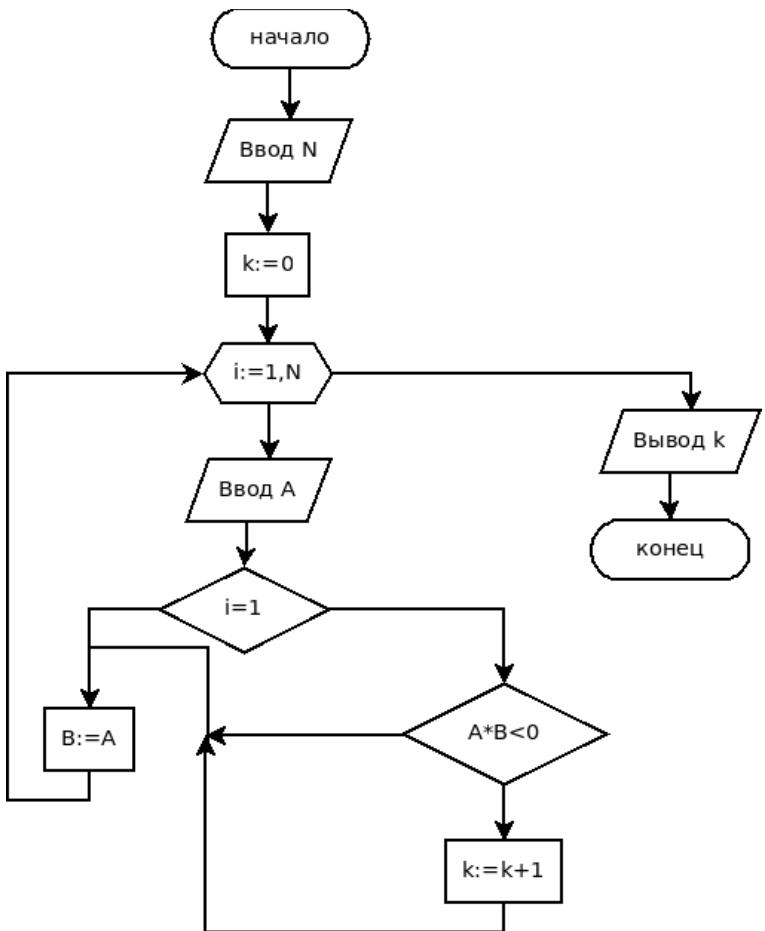


Рис. 3.44. Алгоритм к задаче 3.20

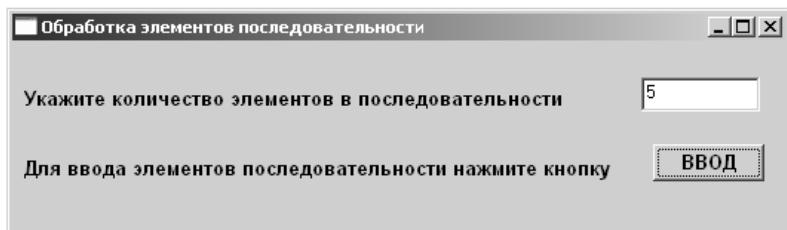


Рис. 3.45. Форма программы решения задачи 3.20

```

procedure TForm1.Button1Click ( Sender : TObject );
var N, A, B, i, k : integer; S : string;
begin
  N:= StrToInt ( Edit1.Text );
  S:= InputBox ( 'Ввод элементов последовательности',
                 'Введите число. 0 - конец_последовательности', '0' );
  A:= StrToInt ( S );
  k := 0;
  for i := 1 to N do
  begin
    S:= InputBox ( 'Ввод элементов последовательности',
                   'введите число. 0 - конец последовательности', '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](#).

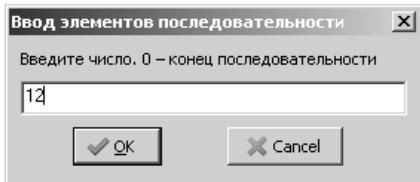


Рис. 3.46. Второе окно диалога к задаче 3.20



Рис. 3.47. Результат работы программы к задаче 3.20

Лекция 4. Подпрограммы

В практике программирования часто складываются ситуации, когда одну и ту же группу операторов, реализующих определённую 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 должна быть обязательно описана в разделе описаний. Описание подпрограммы состоит из заголовка и тела. В заголовке объявляется имя подпрограммы, и в круглых скобках её 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;
- одноимённые 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

её 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 списке в заголовке функции, называются формальными параметрами, или просто параметрами подпрограммы. Все переменные из этого списка 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 [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 передадим к практическому использованию подпрограмм при решении задач. Изучение подпрограмм начнем с процедур.

4.3 Процедуры

Описание процедуры имеет вид:

```
procedure имя_процедуры(список_формальных_параметров);
    label
        список_меток;
    const
        список_констант;
    type
        список_типов;
    var
        список_переменных;
    begin
        //Тело процедуры.
end;
```

Описание начинается с заголовка процедуры, где **procedure** — ключевое слово языка, **имя_процедуры** — любой допустимый в языке Free Pasacal идентификатор, **список_формальных_параметров** — имена формальных параметров и их типы, разделённые 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 необязателен и может отсутствовать:

```
procedure name_3;
```

Если в заголовке процедуры будут применяться параметры-переменные, то перед ними необходимо указывать служебное слово **var**:

```
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 [\(константы, типы, переменные, процедуры и функции, используемые в процедуре\)](#) 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 введёт

a = 0

, то уравнение из квадратного превратится в линейное. Алгоритм решения линейного уравнения тривиален:

$$x = -c/b$$

, при условии, что

$$b \neq 0$$

. Чтобы не усложнять уже составленный алгоритм решения квадратного уравнения, запишем его в виде подпрограммы-процедуры. Далее приведён 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:
    //a,b,c - коэффициенты квадратного уравнения;
    //Выходные параметры процедуры:
    //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 квадратного уравнения ax^2 + bx + c = 0.
    begin
        korni ( a_, b_, c_, x1_, x2_, pr_ ); //Вызов процедуры.
        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
        write ( 'X= ' ); readln (X); //Элемент последовательности.
        if X>0 then //Если элемент положительный, то
        begin
            max_min(X,X_max, X_min ); //вызов процедуры.
            //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(список_формальных_параметров ) :
    тип_результата;
label
    список_меток;
const
    список_констант;
type
    список_типов;
var
    список_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, разделённые точкой с запятой — **список_формальных_параметров**, тип возвращаемого функцией значения — **тип_результата** (функции могут возвращать скалярные значения целого, вещественного, логического, символьного или ссылочного типов).

Примеры описания функций:

```

function fun_1 ( x : real ) : real;
function fun_2 ( a, b : integer ) : real;

```

Тело функции состоит из раздела описаний (константы, типы, переменные, процедуры и функции, используемые в процедуре) и операторов языка, реализующих её 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:

имя_функции (список_фактических_параметров);

Например:

```

y:=fun_1 ( 1. 2 8 );
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 в последовательности.
    write ('K= '); readln (K);
    Sum := 0; //Переменная для накапливания суммы.
    Pro := 1; //Переменная для вычисления произведения.
    kol_p := 0; //Счётчик простых чисел.
    kol_s := 0; //Счётчик совершенных чисел.
    for i := 1 to K do
begin
    //Ввод элемента последовательности.
    Writeln ('X= '); readln (X);
    if prostoe (X) then //Если число простое,
begin
        //выполнить операцию умножения,
        Pro:=Pro *X;
        //увеличить счётчик простых чисел.
        kol_p:=kol_p +1;
    end;
    if soversh (X) then //Если число совершенное,
begin
        //выполнить операцию умножения,
        Sum:=Sum+X;
        //увеличить счётчик совершенных чисел.
        kol_s := kol_s +1;
    end;
end;
    //Если были найдены совершенные числа,
    if kol_s<> 0 then
begin
    Sum:=Sum/ kol_s; //вычислить среднее арифметическое.
    writeln ('Среднее арифметическое совершенных чисел ',
            Sum : 5 : 2 );
end
else //иначе вывести сообщение:
    writeln ('Совершенных чисел в последовательности нет. ');
if kol_p<>0 then //Если были найдены простые числа,
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, 123 454 321, 678 876 — 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 неё число 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
        palindrom := false; //то, число не палиндром.
        break; //Досрочный выход из цикла.
      end;
    //Изменение числа
    M := M - (M div j) * j; //Удаление первой цифры числа.
    M := M div 10; //Удаление последней цифры числа.
    j := j div 100; //Уменьшение разрядности.
  end;
end;
//Основная программа.
var X: longint; pr : boolean;
begin
  write ( 'X= ' ); readln (X); //Ввод элемента последовательности.
  pr := false; //Пусть в последовательности нет палиндромов.
  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. Если введёт в радианах, ответ получит в градусах и, соответственно, введёт в градусах — ответ получит в радианах.

С точки зрения математика задача не вызывает сложности:

- чтобы найти радианную меру какого-нибудь угла по данной градусной мере, нужно помножить число градусов на , число минут на , число секунд на и найденные произведения сложить;
- чтобы найти градусную меру угла по заданной радианной, нужно помножить число радиан на ; если из полученной дроби выделить целую часть, то получим градусы; если из числа, полученного умножением оставшейся дробной части 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. 4.1](#).

С компонентами **Edit**, **Label** и **Button** мы уже хорошо знакомы. 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: включён

первый компонент — будет осуществлён перевод из градусной меры в радианную, включен второй — наоборот. Двойной щелчок по компоненту `TForm1.RadioButton1` приведёт к созданию процедуры `TForm1.RadioButton1Click` обработки события щелчка мыши по кнопке переключателю. В тексте процедуры следует указать команды, которые будут выполняться, если пользователь включил или выключил компонент.

Свойство	Значение	Описание свойства
Caption	Величина угла	Заголовок формы
Height	265	Высота формы
Width	325	Ширина формы
BorderIcons.BiMaximize	false	Кнопка развертывания окна недоступна
BorderStyle	bdDialog	Стиль рамки — диалоговое окно, не изменяет размеры
Position	poScreenCenter	Окно появится в центре экрана

Таблица 4.1. Свойства формы

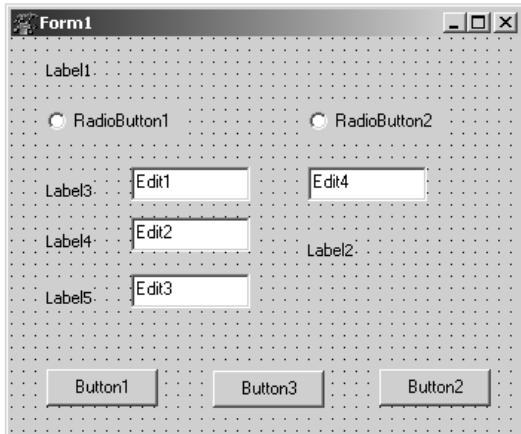


Рис. 4.1. Настройка формы к задаче 4.5

Нам уже известно, что свойства компонентов могут изменяться как в окне конструирования формы, так и непосредственно в программе. Если дважды щелкнуть по форме, вне размещённых на ней компонентов, то будет создана процедура `TForm1.FormCreate` обработки события открытия формы. На вкладке **События** инспектора объектов это событие носит название `OnCreate`. Процедура `OnCreate` выполняется в момент создания формы. Все действия, включённые в неё, будут выполнены перед первым выводом формы на экран. Здесь 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 приведён ниже.

```
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; //Радианская мера угла.
kod_g, kod_m, kod_s, kod_r : integer; //Контроль ввода.
begin
if RadioButton1. Checked then //Если первый переключатель вкл.
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
        Label2. Visible := true; //сделать видимым компонент Label2
        //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;
if RadioButton2. Checked then //Если второй переключатель вкл.
begin
    Val ( Edit4. Text, rad, kod_r ); //Ввод радианной меры угла.
    if ( kod_r=0) then //Если нет ошибок при вводе, то
    begin
        Label2. Visible := true; //сделать видимым компонент Label2
        //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;
//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; end.

```

Результаты работы программы представлены на [рис. 4.2](#) и [рис. 4.3](#).

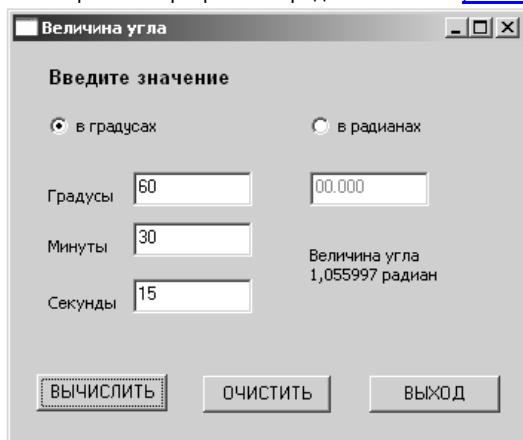


Рис. 4.2. Перевод значений из градусной меры в радианную

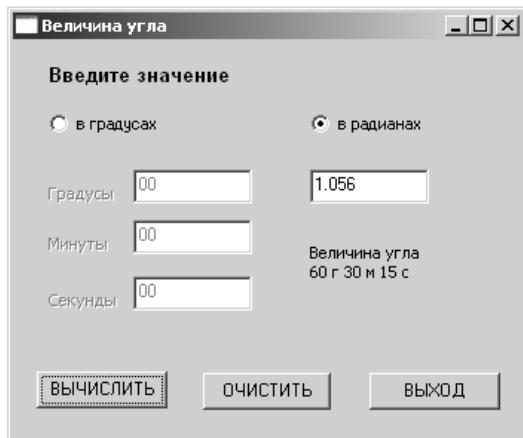


Рис. 4.3. Перевод значений из радианной меры в градусную
ЗАДАЧА 4.6. Создать программу для решения уравнений:

- линейное ;
- квадратное ;
- кубическое $ax^3 + bx^2 + cx + d$.

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. Свойства формы настроим по [табл. 4.1](#), за исключением свойства **Caption**, которому присвоим значение **Решение уравнения**. Компоненты на форме разместим так, как показано на [рис. 4.4](#).

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 указать свое решение: да или нет. Установлен флажок или нет, определяет свойство **Checked** (**true**, **false**). В составе диалогового окна может быть несколько таких компонентов, причём состояние любого из них не зависит от состояния остальных.

Компонент группы переключателей **RadioGroup** объединяет в себе несколько переключателей. Каждый размещённый в нём переключатель помещается в специальный список **Items** и доступен по номеру, установленному в свойстве **ItemIndex**. После размещения на форме компонент пуст. Чтобы создать в нём 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](#).

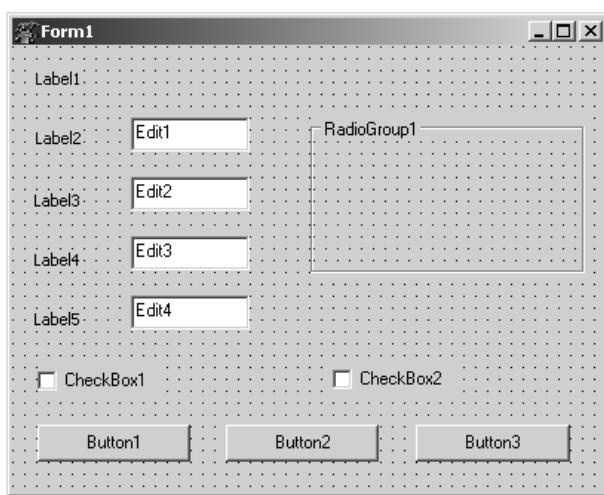


Рис. 4.4. Процесс создания формы к задаче 4.6

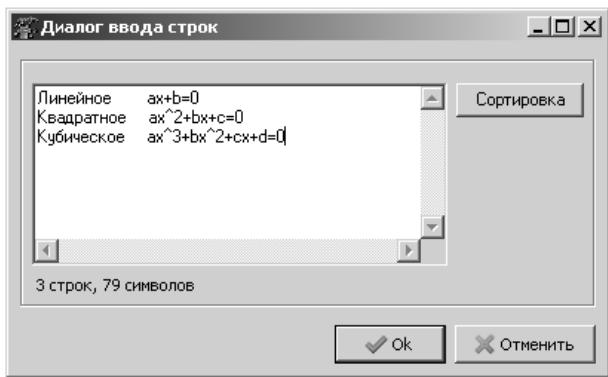


Рис. 4.5. Окно редактора списка

После создания компонента группа переключателей, его свойство номер переключателя **ItemIndex** по умолчанию равно **_1**. Это означает, что ни один компонент в группе не установлен. 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, размещёнными на форме, пользователь уже знаком.

На [рис. 4.6](#), [рис. 4.7](#), [рис. 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.

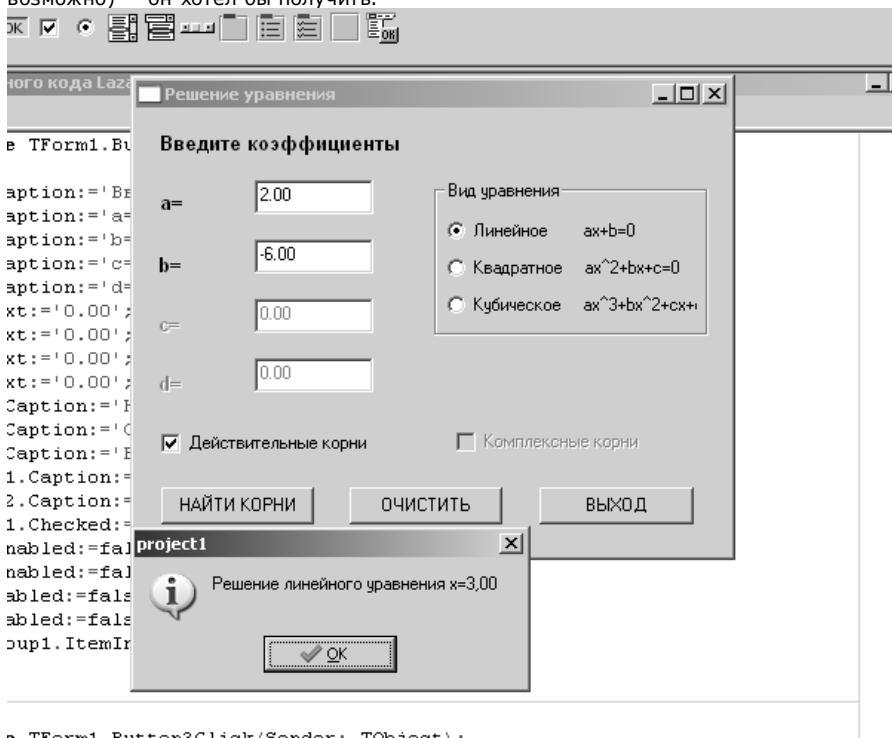


Рис. 4.6. Решение линейного уравнения

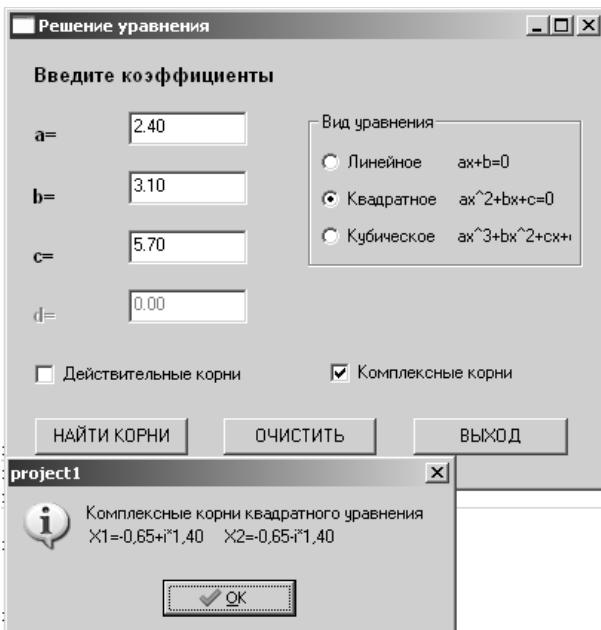


Рис. 4.7. Решение квадратного уравнения

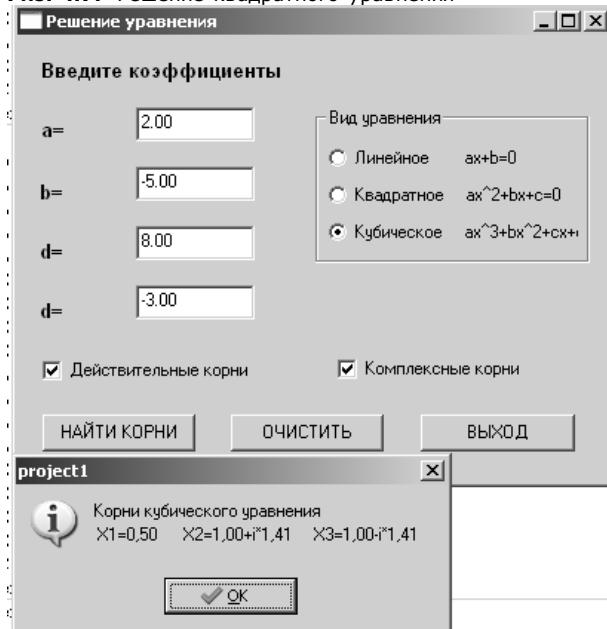


Рис. 4.8. Решение кубического уравнения

Далее приведён 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.
    procedure korni_1 ( a, b : real; var x_ : string );
        var x : real;
begin
    x:=-b/a;
    x_:= FloatToStrF ( x, ffFixed, 5, 2 );
end;
//Решение квадратного уравнения.
procedure korni_2 ( a, b, c : real; var x1_, x2_ : string;
                    var pr : byte );
    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.
procedure korni_3 ( a, b, c, d : real; var x1_, x2_, x3_ : string;
                    var pr : byte );
    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 / 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;
    x1:=u+v-r / 3; //Вычисление действительного корня.
    //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; //Один действительный и два 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 первый переключатель из группы.
    begin
      //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
          if a_<>0 then //Первый коэффициент не ноль.
            begin
              //Решение линейного уравнения.
              korni_1( a_, b_, _x );
              //Вывод найденного значения.
              MessageDlg ( 'Решение линейного уравнения x= '
+ _x, mtInformation, [ mbOk ], 0 );
            end
          else //Первый коэффициент равен нулю,
            //вывод соответствующего сообщения.
            MessageDlg ( 'Нет корней ! ', mtInformation,
[mbOk ], 0 );
        end
      else //Некорректный ввод данных.
        MessageDlg ( 'Ошибка при вводе коэффициентов ! ',
mtInformation, [ mbOk ], 0 );
    end;
    1 : //Пользователь выбрал второй переключатель из группы.
    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
          if a_<>0 then //Первый коэффициент не ноль.
            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_квадратного_уравнения '
+ chr(13)+ ' X1= '+_x1+' X2= '+_x2,
mtInformation, [ mbOk ], 0 );
              //Установлен первый флагок,
              //корни действительные.
              if CheckBox1. Checked and
                ( CheckBox2. Checked=false ) and ( _pr=1) then
                  MessageDlg ( 'Действительные корни ',
'квадратного уравнения '+ chr(13)+ ' X1= '+
_x1+' X2= '+_x2, mtInformation, [ mbOk ], 0 );
              //Установлен второй флагок,
              //корни действительные.
              if ( CheckBox1. Checked=false ) and
                CheckBox2. Checked and ( _pr=1) then
                  MessageDlg ( 'Уравнение имеет только ',
'действительные корни.', mtInformation, [ mbOk ], 0 );
              //Установлен первый флагок, корни комплексные.
              if CheckBox1. Checked and
                ( CheckBox2. Checked=false ) and ( _pr=2) then
                  MessageDlg ( 'В уравнении действительных ',
'корней нет', mtInformation, [ mbOk ], 0 );
              //Установлен второй флагок, корни комплексные.
              if ( CheckBox1. Checked=false ) and

```

```

    CheckBox2. Checked and ( _pr=2) then
    MessageDlg ( 'Комплексные корни квадратного',
    'уравнения '+ chr (13)+ ' X1= '+_x1+ ' X2= '+_x2,
    mtInformation, [ mbOk ], 0 );
end
else //Первый коэффициент равен нулю.
    MessageDlg ( 'Первый коэффициент не должен быть',
    'нулевым ! ', mtInformation, [ mbOk ], 0 );
end //Некорректный ввод данных.
else MessageDlg ( 'Ошибка при вводе коэффициентов ! ',
    mtInformation, [ mbOk ], 0 );
end;
2 : //Пользователь выбрал третий переключатель из группы.
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
    if a_<>0 then //Первый коэффициент не ноль.
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 флашка не установлены.
    if ( CheckBox1. Checked=false ) and
    ( CheckBox2. Checked=false ) then
        MessageDlg ( 'Выберите тип решения ',
        mtInformation, [ mbOk ], 0 );
    //Оба флашка 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 флашок,
    //корни действительные.
    if CheckBox1. Checked and
        ( CheckBox2. Checked=false ) and ( _pr=1) then
            MessageDlg ( 'Действительные корни ',
            'кубического уравнения '+chr (13)+_
            ' X1= '+_x1+ ' X2= '+_x2+ ' X3= '+_x3,
            mtInformation, [ mbOk ], 0 );
    //Установлен первый флашок, корни комплексные.
    if CheckBox1. Checked and
        ( CheckBox2. Checked=false ) and ( _pr=2) then
            MessageDlg ( 'Действительные корни ',
            'кубического_уравнения '+chr (13)+_
            ' X1= '+_x1, mtInformation, [ mbOk ], 0 );
    //Установлен второй флашок, корни комплексные.
    if ( CheckBox1. Checked=false ) and
        CheckBox2. Checked and ( _pr=2) then
            MessageDlg ( 'Комплексные корни кубического_',
            'уравнения '+ chr (13)+ ' X1= '+_x2+ ' X2= '+_x3,
            mtInformation, [ mbOk ], 0 );
    //Установлен второй флашок,
    //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 должен быть',
    'нулевым ! ', mtInformation, [ mbOk ], 0 );
end
else //Некорректный ввод данных.
    MessageDlg ( 'Ошибка при вводе коэффициентов ! ',
    mtInformation, [ mbOk ], 0 );
end;
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;
        end;
    end;

```

```

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. 0 0';
    Edit2. Text := '0. 0 0';
    Edit3. Text := '0. 0 0';
    Edit4. Text := '0. 0 0';
    CheckBox2. Enabled := true;
end;
end;
end;
end.

```

4.6 Рекурсивные функции

Под рекурсией в программировании понимают подпрограмму, которая вызывает сама себя. Рекурсивные функции чаще всего используют для компактной реализации рекурсивных алгоритмов. Классическими 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.

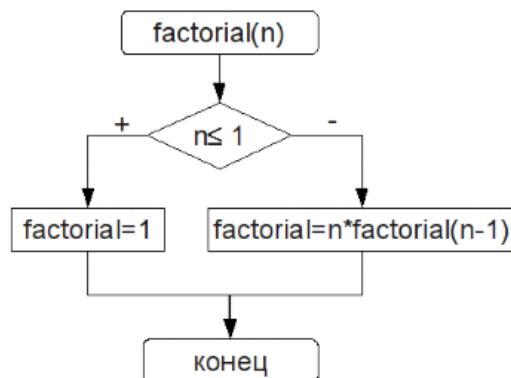
В рекурсивной функции необходимо обязательно предусмотреть завершение рекурсивного вызова. Иначе функция никогда не завершит свою работу.

Рассмотрим применение рекурсии на примерах.

ЗАДАЧА 4.7. Вычислить факториал числа

n

Вычисление факториала подробно рассмотрено в задаче 3.10 (рисунок 3.29). Для решения этой задачи с применением рекурсии создадим функцию **factoial**, алгоритм которой представлен на [рисунке 4.9](#).



[увеличить изображение](#)

Рис. 4.9. Рекурсивный алгоритм вычисления факториала

Текст подпрограммы с применением рекурсии на языке Free Pascal:

```

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 а (

n — целое число).

Результатом возведения числа а 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

$$a^{-n} = 1/a^n$$

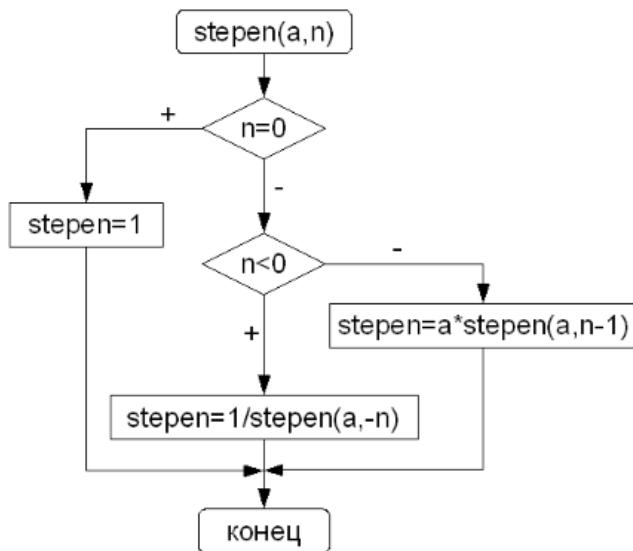
. В случае если

$$n = 0$$

, то

$$a^0 = 1$$

Для решения задачи создадим рекурсивную функцию **stepen**, алгоритм которой представлен на [рис. 4.10](#).



[увеличить изображение](#)

Рис. 4.10. Рекурсивный алгоритм вычисления степени числа

Фрагмент программы с применением рекурсии:

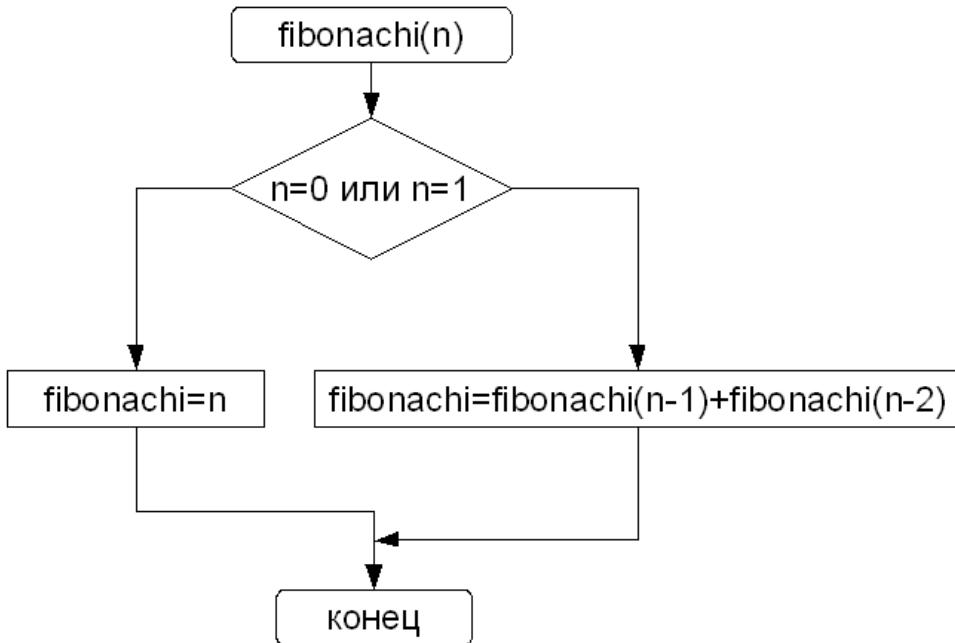
```
function stepen ( a : real; n : word ) : real;
begin
  if n=0 then
    stepen:=1
  else
    if n<0 then
      stepen:=1/ stepen ( a, -n )
    else
      stepen :=a * stepen ( a, n - 1 );
end;
var x : real; k : word;
begin
  writeln ( 'x=' );
  readln ( x );
  writeln ( 'k=' );
  readln ( k );
  writeln ( 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 **fibonacci** изображен на [рис. 4.11](#).



[увеличить изображение](#)

Рис. 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 ещё 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 передаётся 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 её имя. Существует два процедурных типа: тип-процедура и тип-функция:

type

тип_процедура=procedure (список_формальных_параметров);
 тип_функция=function (список_формальных_параметров) : 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 адресу¹¹:

@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`. Её 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;

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

{Вывов Vivodfunc с функцией f в качестве параметра.}

VivodFunc (0, 1, 7, @f);

writeln;

{Вывов Vivodfunc с функцией g в качестве параметра.}

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 в интервале (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.}

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;
    {Вызов процедуры.}
    VivodFunc ( 0, 1, 7, fff, 2 );
end.

```

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 при разработке визуальных приложений в Lazarus. Теперь рассмотрим, как создавать личный модуль.

Модуль имеет следующую структуру:

UNIT имя модуля;

INTERFACE

интерфейсная часть

IMPLEMENTATION

исполнляемая часть

BEGIN

инициализирующая часть

end.

Заголовок модуля состоит из служебного слова **UNIT** и следующего за ним имени.

Причём 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 инициализирующей части размещаются исполняемые операторы, содержащие некоторый фрагмент программы. Эти программы исполняются до передачи управления основной программе и обычно используются для подготовки её работы. Инициализирующая часть может отсутствовать вместе с начинающим её словом **begin**.

В качестве примера создадим модуль работы с комплексными числами, в котором будут представлены подпрограммы, реализующие основные операции¹¹ 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
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, введём 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**, и с расширением **.ppu**. Для использования нужен именно файл с расширением **.o**. Теперь скомпилированный файл модуля с расширением **.o** должен находиться 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 приведён ниже.

```

program test;
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](#).

```

Введите первое комплексное число
3 4
3.000+4.000i
Введите второе комплексное число
8 -9
3.000+4.000i
Сумма чисел
11.000-5.000i
Разность чисел
-5.000+13.000i
Произведение чисел
60.000+5.000i
Частное чисел
-0.083+0.407i

```

```

Терминал
Файл Правка Вид Терминал Вкладки Справка
Введите первое комплексное число
3 4
3.000+4.000i
Введите второе комплексное число
8 -9
3.000+4.000i
Сумма чисел
11.000-5.000i
Разность чисел
-5.000+13.000i
Произведение чисел
60.000+5.000i
Частное чисел
-0.083+0.407i

-----
(program exited with code: 0)
Press return to continue

```

Рис. 4.12. Операции с комплексными числами

Лекция 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 течении 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 ([см. табл. 5.1](#)).

№ элемента	1	2	3	4	...	364	365
<code>temperature</code>	-1.5	-3	-6.7	1	...	2	-3

Таблица 5.1. Массив значений температур

Таким образом, можно дать следующее определение.

Массив — структурированный тип данных, состоящий из фиксированного числа элементов одного типа.

Массив, представленный в [таблице 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 определённом элементе массива, необходимо указать имя массива и порядковый номер этого элемента, называемый индексом.

Если возникает необходимость хранения данных в виде таблиц, в формате строк и столбцов, то необходимо использовать многомерные массивы.

Элементы массива

1	2	3	4	5	6	7
-1.5	-3.913	13.672	-1.56	45.89	4.008	-3.61

Таблица 5.2. Одномерный массив из 7 вещественных чисел

В [таблице 5.3](#) приведён пример массива, состоящего из трёх строк и четырёх столбцов. Это двумерный массив (матрица). Строки в нём можно 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
	2	1.1	4.7	8.12
	3	-2.4	-6.2	11.23

Таблица 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.

Первый — 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 определённый тип данных, например:

```
type
massiv=array [ 0..12 ] of real;
//Тип данных massiv из 13 элементов, которые нумеруются от 0
//до 12.
dabc=array [ -3..6 ] of integer;
//Тип данных dabc из 10 элементов, которые 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_переменных;
```

Например:

```
var
z, x : array [ 1..25 ] of word;
//Массивы z и x из 25 значений типа word, которые нумеруются
//от 1 до 25.
g : array [ -3..7 ] of real;
//Массив g из 11 значений типа real, которые нумеруются от -3
//до 7.
```

Для описания массива можно использовать предварительно определённые 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 трёх определениях мы получали матрицу вещественных чисел, состоящую из 300 строк и 200 столбцов.

Аналогично можно ввести трёхмерный 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 единственным 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.

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,...,

-й ([см. рис. 5.1–5.2](#)). Для обработки элементов массива удобно использовать цикл `for..do`.

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 и 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 ([см. рис.](#)

5.3—5.4). Для обращения к элементу массива необходимо указать имя массива и в квадратных скобках номер элемента, например **X[5], b[2]** и т. д.

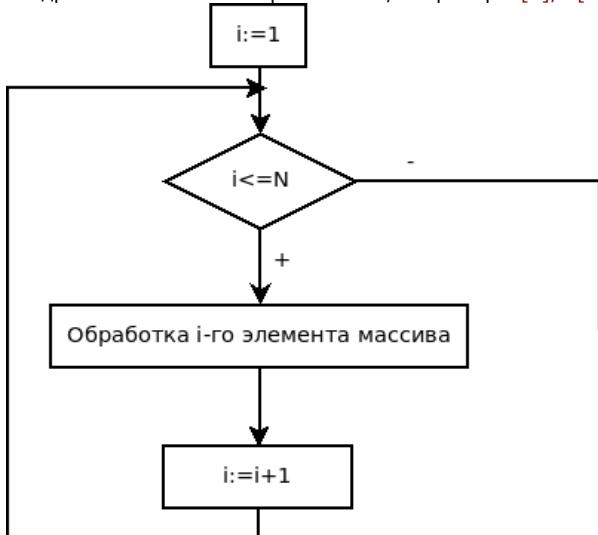


Рис. 5.1. Блок-схема обработки элементов массива с использованием цикла с предусловием

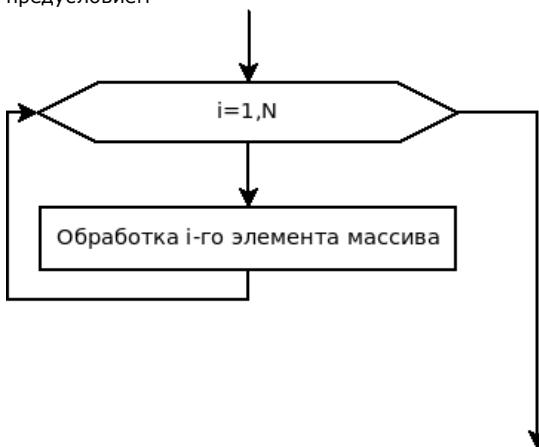


Рис. 5.2. Блок-схема обработки элементов массива с использованием цикла `for`

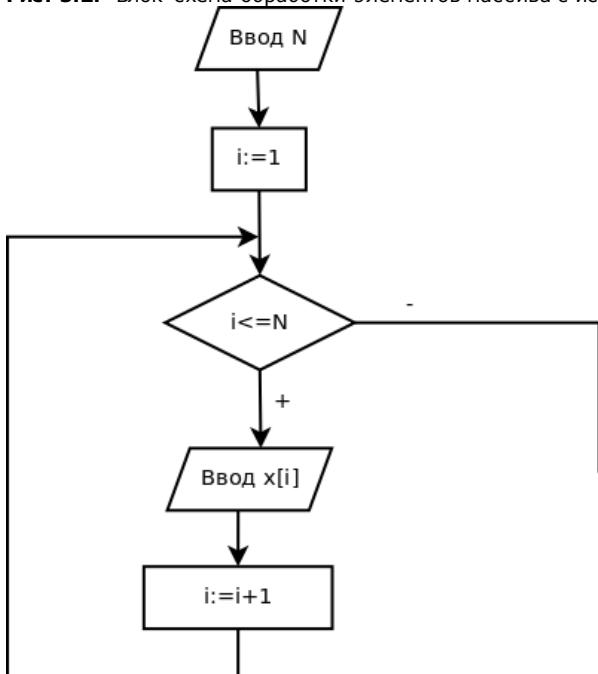


Рис. 5.3. Алгоритм ввода массива X с использованием цикла с предусловием

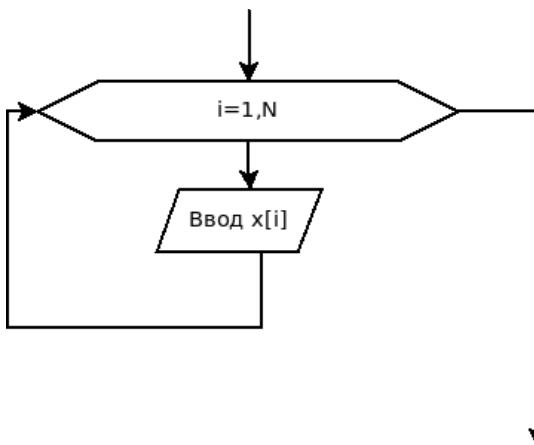


Рис. 5.4. Алгоритм ввода массива X с использованием блока модификации

5.4.1 Организация ввода-вывода в консольных приложениях

Рассмотрим реализацию этих алгоритмов 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
x : array [ 1.. 1 0 0 ] of real;
i, n : integer;
begin
readln ( N );
for i :=1 to N do
begin
    write ( ' x ( ', i, ' )=' ); readln ( x [ i ] )
end;
end.

```

Цикл `for..do` удобнее использовать для обработки всего массива, и в дальнейшем при выполнении подобных операций с массивами мы 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 [рисунке 5.5](#).

Обратите внимание, что между числами в данном варианте отсутствует пробел.

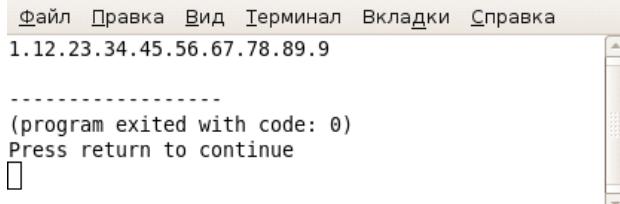


Рис. 5.5. Результат первого варианта вывода массива

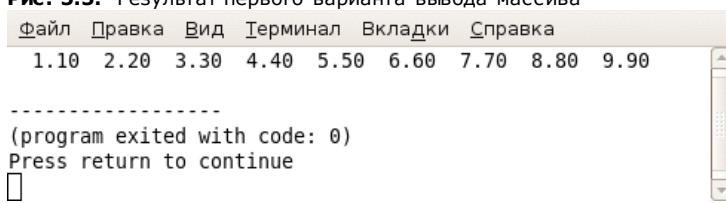


Рис. 5.6. Результат второго варианта вывода массива

```
Файл Правка Вид Терминал Вкладки Справка
1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9

-----
(program exited with code: 0)
Press return to continue
□
```

Рис. 5.7. Результат третьего 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 [рисунке 5.6](#).

```
//Вариант 3
for i := 1 to n do write ( a [ i ] : 3 : 1, ' ' );
```

Результат третьего варианта вывода массива на экран представлен на [рисунке 5.7](#).

```
//Вариант 4
writeln ( 'массив А' );
for i := 1 to n do
writeln ( a [ i ] : 6 : 2 );
```

Результат четвёртого варианта вывода массива на экран представлен на [рисунке 5.8](#).

```
//Вариант 5
for i := 1 to n do write ( ' a ( ', i, ' )= ', a [ i ] : 3 : 1, ' ' ); }
```

Результат пятого варианта вывода массива на экран представлен на [рисунке 5.9](#).

```
Файл Правка Вид Терминал Вкладки Справка
Массив А
1.10
2.20
3.30
4.40
5.50
6.60
7.70
8.80
9.90

-----
(program exited with code: 0)
Press return to continue
□
```

Рис. 5.8. Результат четвёртого варианта вывода массива

```
Файл Правка Вид Терминал Вкладки Справка
a(1)=1.1 a(2)=2.2 a(3)=3.3 a(4)=4.4 a(5)=5.5 a(6)=6.6 a(7)=7.7 a(8)=8.8 a(9)=9.9
```

```
-----
(program exited with code: 0)
Press return to continue
□
```

Рис. 5.9. Результат пятого варианта вывода массива

```

Файл Правка Вид Терминал Вкладки Справка
a(2)= 2.20
a(3)= 3.30
a(4)= 4.40
a(5)= 5.50
a(6)= 6.60
a(7)= 7.70
a(8)= 8.80
a(9)= 9.90

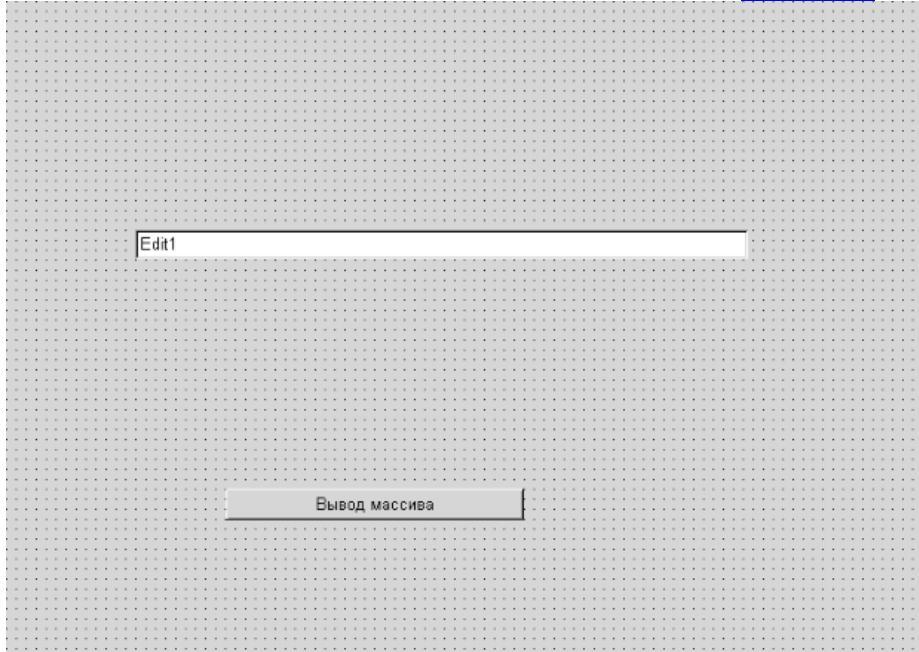
-----
(program exited with code: 0)
Press return to continue

```

Рис. 5.10. Результат шестого 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 [рисунке 5.10](#).



[увеличить изображение](#)

Рис. 5.11. Форма для задачи вывода массива вещественных чисел

Свойство	Name1	Text	Height	Left	Top	Width	ReadOnly
Значение	Edit1	''	23	103	184	492	True

Таблица 5.4. Свойства компонента Edit1

Свойство	Name1	Caption	Height	Left	Top	Width
Значение	label1	Button1	25	177	392	239

Таблица 5.5. Свойства компонента Button1

5.4.2 Организация ввода-вывода 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** ([рис. 5.11](#)).

В [табл. 5.4—5.5](#) приведены свойства компонентов типа **TButton** и **TEdit**.

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 а.
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;

```

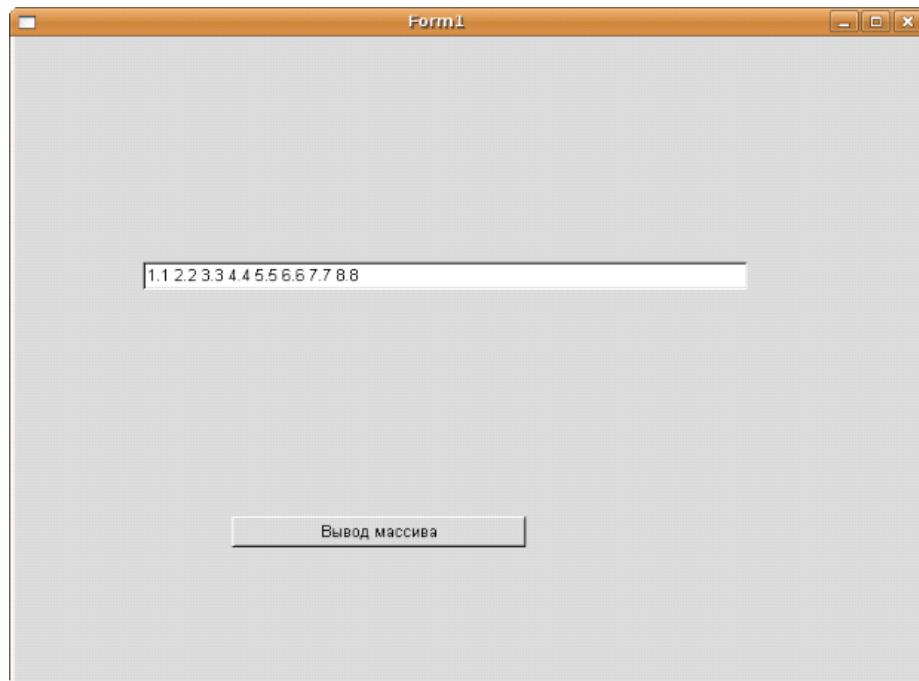
После щелчка по кнопке Вывод массива окно приложения станет подобным тому, которое представлено на [рис. 5.12](#).

Для ввода массивов в Lazarus в простейшем случае можно воспользоваться функцией `InputDialog`. 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;

```



[увеличить изображение](#)

Рис. 5.12. Вывод массива в поле ввода

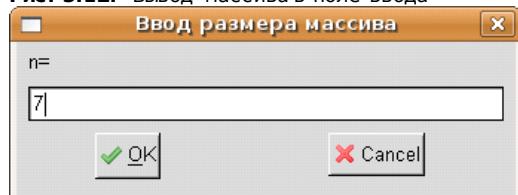


Рис. 5.13. Ввод размера массива

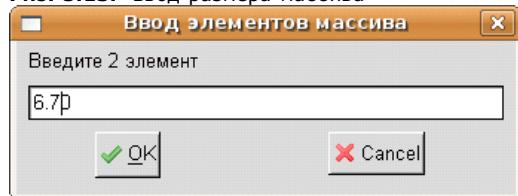


Рис. 5.14. Ввод второго элемента массива

При щелчке по кнопке на экране появится окно для ввода размера массива ([см. рис. 5.13](#)).

После корректного ввода размера массива, последовательно будут появляться

диалоговые окна для ввода очередного элемента, подобные представленному на [рис. 5.14](#).

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. 5.15](#)).

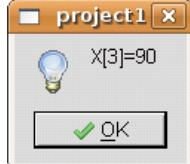


Рис. 5.15. Вывод третьего элемента массива

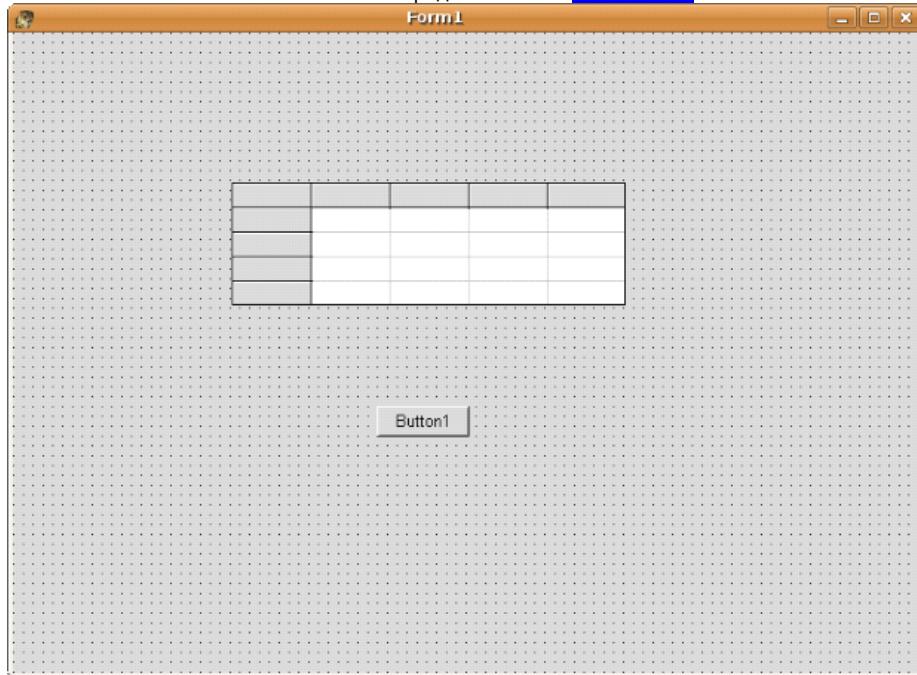
Чтобы у пользователя была возможность просматривать элементы массива одновременно, можно сформировать из них строку, а затем вывести её, например, на форму в виде метки или в окне сообщения:

```
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)+ ' ';
  //Полученную строку можно вывести в виде метки
  Label2. Caption :=S; //Вывод строки на форму в виде метки.
  //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"). Ознакомимся с этим компонентом подробнее.

На [рис. 5.16](#) представлен проект формы, на которой расположен компонент типа `TStringGrid` (таблица строк). Ячейки таблицы строк предназначены для чтения или редактирования данных. Этот компонент может служить как для ввода, так и для вывода массива.

Основные свойства этого компонента представлены в [таблице 5.6](#).



[увеличить изображение](#)

Рис. 5.16. Форма с компонентом "таблица"

Свойств Описание

о

Name	Имя компонента
ColCount	Количество столбцов таблицы
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 нумеруется 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:
	<ul style="list-style-type: none"> • ssNone — отсутствие полос прокрутки (в этом случае пользователь может перемещаться по таблице 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.g_oEditing	Логическая переменная, которая определяет, может пользователь (True) или нет (False) редактировать содержимое ячеек таблицы
Options.g_oTab	Логическая переменная, которая разрешает (True) или запрещает (False) использование клавиши Tab для перемещения курсора в следующую ячейку таблицы
DefaultColWidth	Ширина колонок таблицы
DefaultRowHeight	Высота строк таблицы
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

Таблица 5.6. Основные свойства компонента типа **TStringGrid**

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. 5.7](#).

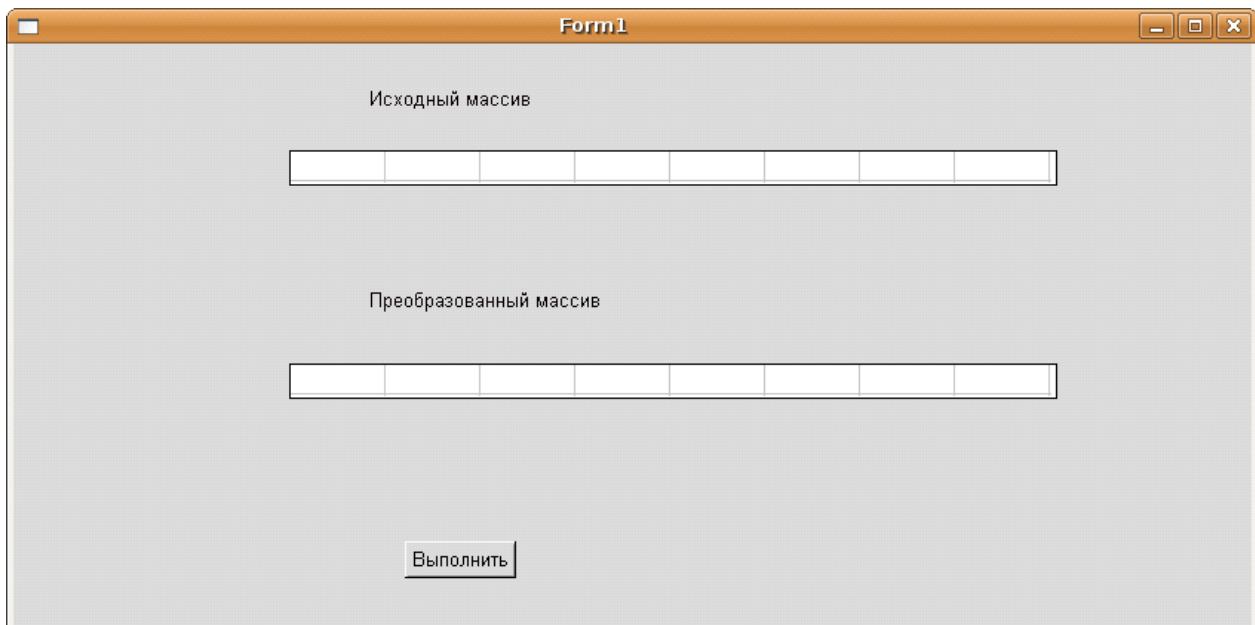
Свойство	StringGrid1	StringGrid2	Описание
ColCount	8	8	Количество столбцов таблицы
RowCount	1	1	Количество строк таблицы
FixedCols	0	0	Количество зафиксированных слева столбцов таблицы
FixedRows	0	0	Количество зафиксированных сверху строк таблицы
Options.go_Editing	True	False	Логическая переменная, которая определяет возможность редактирования содержимого ячеек таблицы пользователем
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

Таблица 5.7. Основные свойства компонента типа **TStringGrid**

Окно формы приложения ввода-вывода массива будет подобно представленному на [рис. 5.17](#).

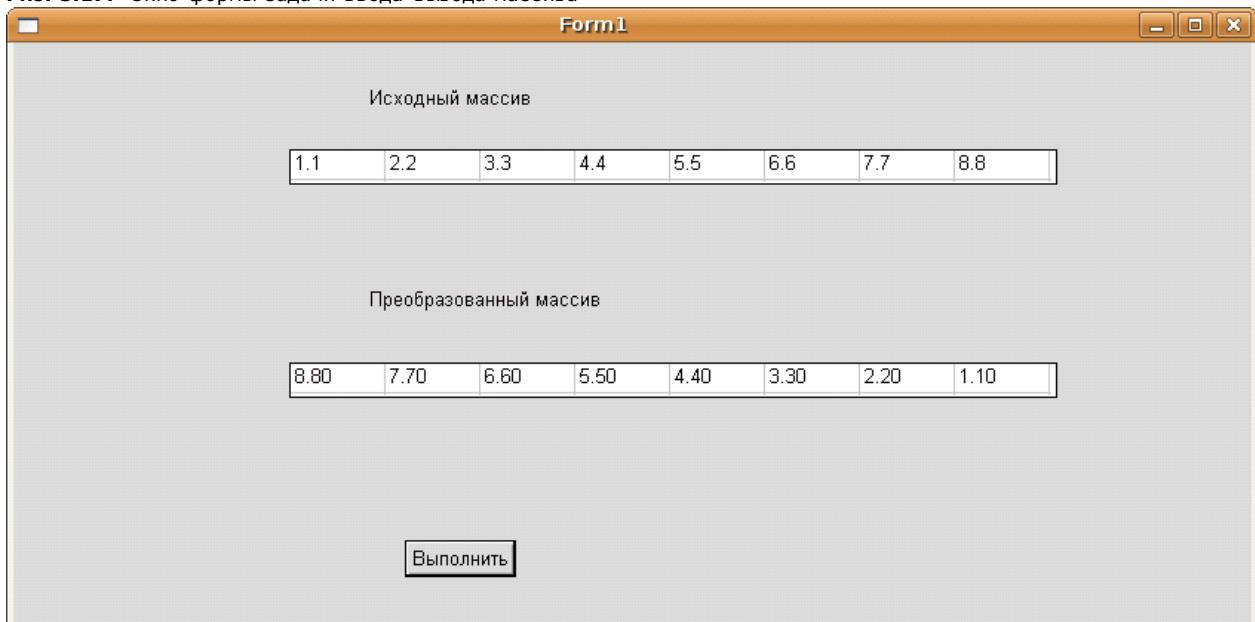
В первую таблицу будем вводить элементы массива, во вторую — преобразованный массив. Щелчок по кнопке **Выполнить** вызовет следующую подпрограмму:

```
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 // Выход массива.
// Элемент массива преобразовывается в строку и помещается в
// поле таблицы.
StringGrid2.Cells [ i, 0 ] := FloatToStrF ( a[7 - i], fffFixed, 5, 2 );
end;
```



[увеличить изображение](#)

Рис. 5.17. Окно формы задачи ввода-вывода массива



[увеличить изображение](#)

Рис. 5.18. Окно программы ввода-вывода массива

При запуске программы на выполнение появляется окно приложения, подобное представленному на [рис. 5.17](#), пользователь вводит исходный массив, щёлкает по кнопке **Выполнить**, после чего окно приложения принимает вид, как на [рис. 5.18](#). В параграфе 5.4 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 перейдём к рассмотрению основных алгоритмов обработки одномерных массивов, многие из которых аналогичны соответствующим алгоритмам обработки последовательностей (вычисление суммы, произведения, поиск элементов по определённому признаку, выборки и т. д.). Отличие заключается в том, что в массиве одновременно доступны все его компоненты, поэтому с массивами становятся возможны более сложные действия (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 аналогичны 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 суммируются элементы массива **X**. Блок-схема алгоритма расчёта суммы приведена на [рис. 5.19](#).

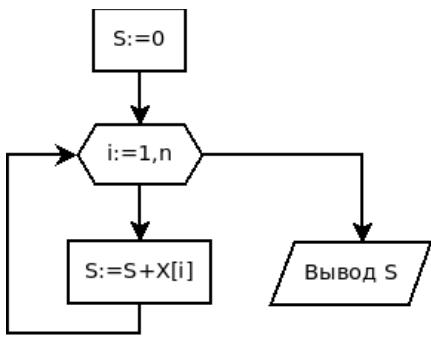


Рис. 5.19. Алгоритм нахождения суммы элементов массива
Соответствующий алгоритму фрагмент программы будет иметь вид:

```
s := 0;
for i := 1 to n do
  s := s + x [ i ];
writeln ( 's= ', s : 7 : 3 );
```

Найдём 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. 5.20](#).

Соответствующий фрагмент программы будет иметь вид:

```
p := 1;
for i := 1 to n do
  p := p * x [ i ];
writeln ( 'P= ', P : 7 : 3 );
```

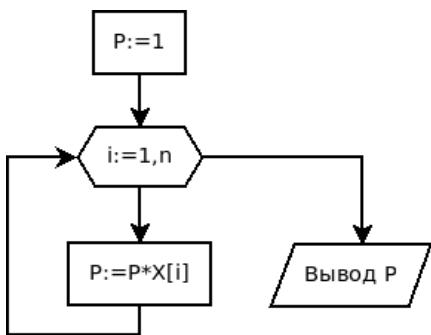


Рис. 5.20. Алгоритм нахождения произведения элементов массива

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). 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. 5.21](#).

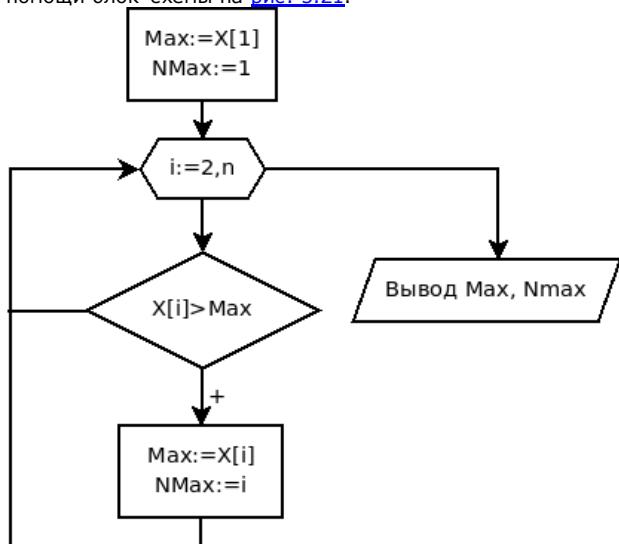


Рис. 5.21. Алгоритм поиска максимального элемента массива и его номера
Соответствующий фрагмент программы имеет вид:

```

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 приведённого 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

$X[1] \leq X[2] \leq \dots \leq X[n]$,

и в порядке убывания, если

$X[1] \geq X[2] \geq \dots \geq X[n]$.

Многие алгоритмы сортировки основаны на том факте, что переставлять два элемента надо 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. Рассмотрим некоторые из алгоритмов.

5.7.1 Сортировка методом "пузырька"

Наиболее известным методом сортировки является сортировка массивов пузырьковым методом. Её популярность объясняется запоминающимся названием 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, и если второй окажется больше третьего, то поменяем их местами. Далее сравниваем третий и четвёртый, и если третий большего четвёртого, их также меняем местами. После трёх этих сравнений самым большим элементом станет элемент с номером 4. Если продолжить сравнение соседних элементов: сравнить четвёртый с пятым, пятый с 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.

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
Четвёртый просмотр	2	3	4	5	7

Таблица 5.8. Процесс упорядочения элементов в массиве по возрастанию

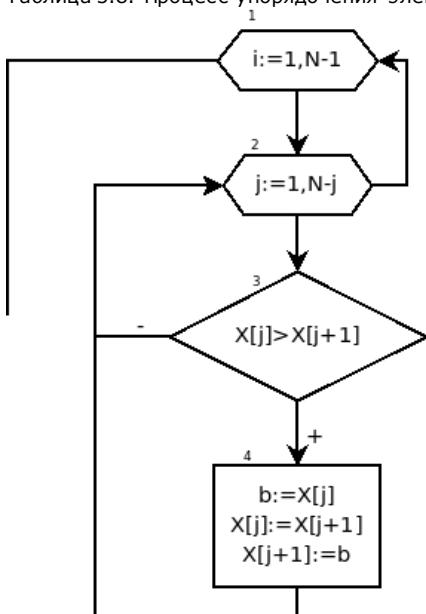


Рис. 5.22. Алгоритм упорядочения по возрастанию методом "пузырька"
Теперь повторим данный алгоритм сначала с 1-го до

$n - 1$

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 своё (

n - 1

-е) место. Так повторяем до тех пор, пока не упорядочим весь массив.

В [табл. 5.8](#) подробно показан процесс упорядочения 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. 5.22](#). Для обмена двух элементов в массиве (блок 4) используется буферная переменная

b

, в которой временно хранится значение элемента, подлежащего замене.

Ниже приведён текст консольного приложения, предназначенного для упорядочения массива по возрастанию методом пузырька.

```
program upor_massiv;
var i, j, n : byte;
    X: array [ 1.. 100 ] of real;
    b : 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.
```

На [рис. 5.23](#) приведены результаты работы этой программы.

Для упорядочения элементов в массиве по убыванию их значений необходимо при сравнении элементов массива заменить знак > на < (см. блок 3 на [рис. 5.22](#)).

Рассмотрим следующий алгоритм сортировки.

5.7.2 Сортировка выбором

Для сортировки элементов массива по возрастанию (по убыванию) можно воспользоваться алгоритмом сортировки выбора максимального (минимального) элемента. Алгоритм выбором приведён в виде блок-схемы на [рис. 5.24](#).

```
Терминал
Файл Правка Вид Терминал Вкладки Справка
Введите размер массива
5
X[1]=7
X[2]=3
X[3]=5
X[4]=4
X[5]=2
массив X
7.00 3.00 5.00 4.00 2.00
упорядоченный массив
2.00 3.00 4.00 5.00 7.00

-----
(program exited with code: 0)
Press return to continue

```

[увеличить изображение](#)

Рис. 5.23. Результат программы упорядочения массива по возрастанию

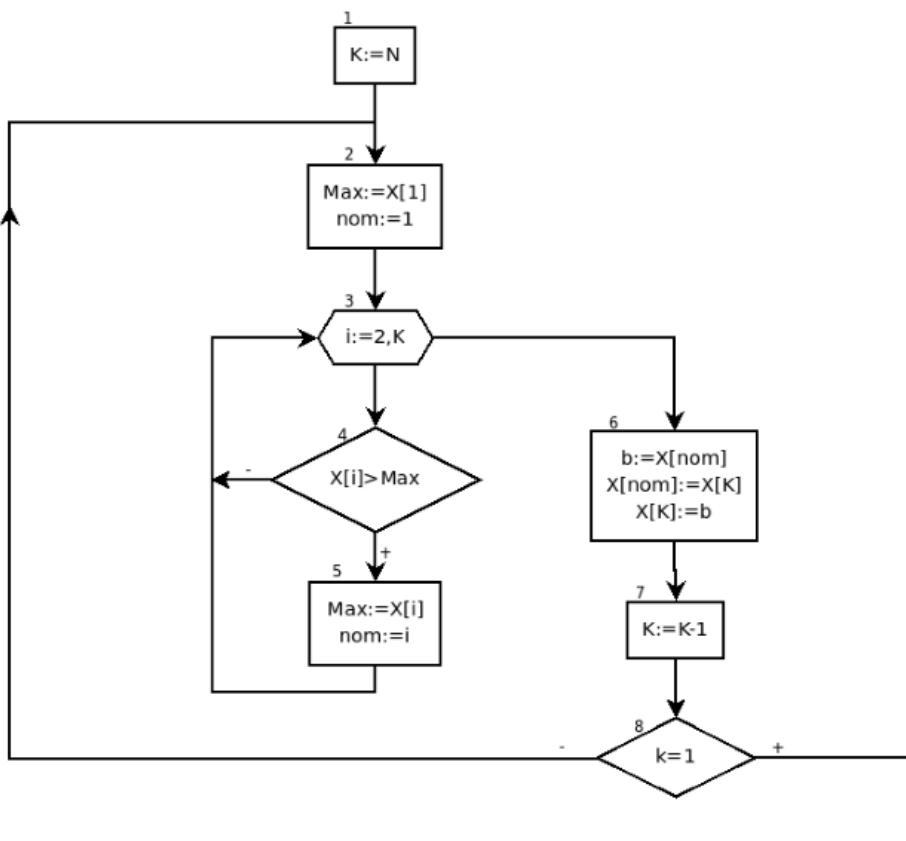


Рис. 5.24. Сортировка массива по возрастанию выбором наибольшего элемента
Найдём в массиве самый большой элемент (блоки 2–5) и поменяем его местами с последним элементом (блок 6). После этого максимальный элемент встанет на своё место. Теперь надо повторять эти действия (блоки 2–6), уменьшив количество 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 упорядочении массива по убыванию необходимо перемещать минимальный элемент. Для этого в алгоритме (рис. 5.24) в блоке 4 достаточно поменять знак **>** на знак **<**.

Ниже приведён фрагмент программы упорядочения массива по возрастанию, используя сортировку выбором максимального элемента.

```

k:=n;
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;

```

Следующим важным алгоритмом обработки массивов является алгоритм удаления элемента из массива.

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 четвёртый, на место четвёртого — 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. 5.25.

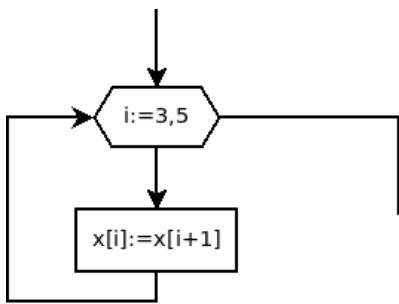


Рис. 5.25. Алгоритм удаления 3-го элемента из массива

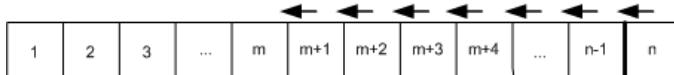


Рис. 5.26. Процесс удаления элемента из массива

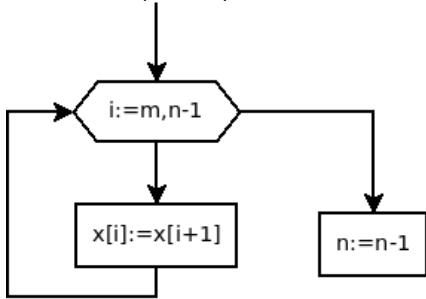


Рис. 5.27. Алгоритм удаления m -го элемента из массива из n элементов

Теперь рассмотрим более общую задачу: необходимо удалить m -й элемент из массива X , состоящего из n элементов. Для этого достаточно записать элемент $(m+1)$ -й 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. 5.26](#).

Алгоритм удаления из массива X размерностью n элемента с номером m приведён на [рис. 5.27](#).

После удаления элемента из массива изменится количество элементов в массиве (уменьшится на один), и у части элементов изменится индекс. Если элемент удалён, то на место него приходит следующий, передвигаясь к которому (путём увеличения индекса на один) нет необходимости. Следующий элемент сам сдвинулся влево после удаления.

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 путём сдвига всех последующих на один влево.

Единственное, о чём стоит помнить, — что после удаления элемента не надо переходить к следующему для последующей обработки, он сам сдвигается на место текущего. Блок-схема решения задачи 5.1 представлена на [рис. 5.28](#).

Ниже представлен текст программы с комментариями.

```

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;
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 надо переходить к следующему элементу массива.}
            n:=n -1;
        end
    else
{Если элемент не удалялся, то переходим к следующему элементу массива.}

```

```

i := i +1;
writeln ('Изменённый массив ');
for i :=1 to n do {Вывод преобразованного массива.}
write (X[ i ] : 5 : 2, ' '); writeln; end.

```

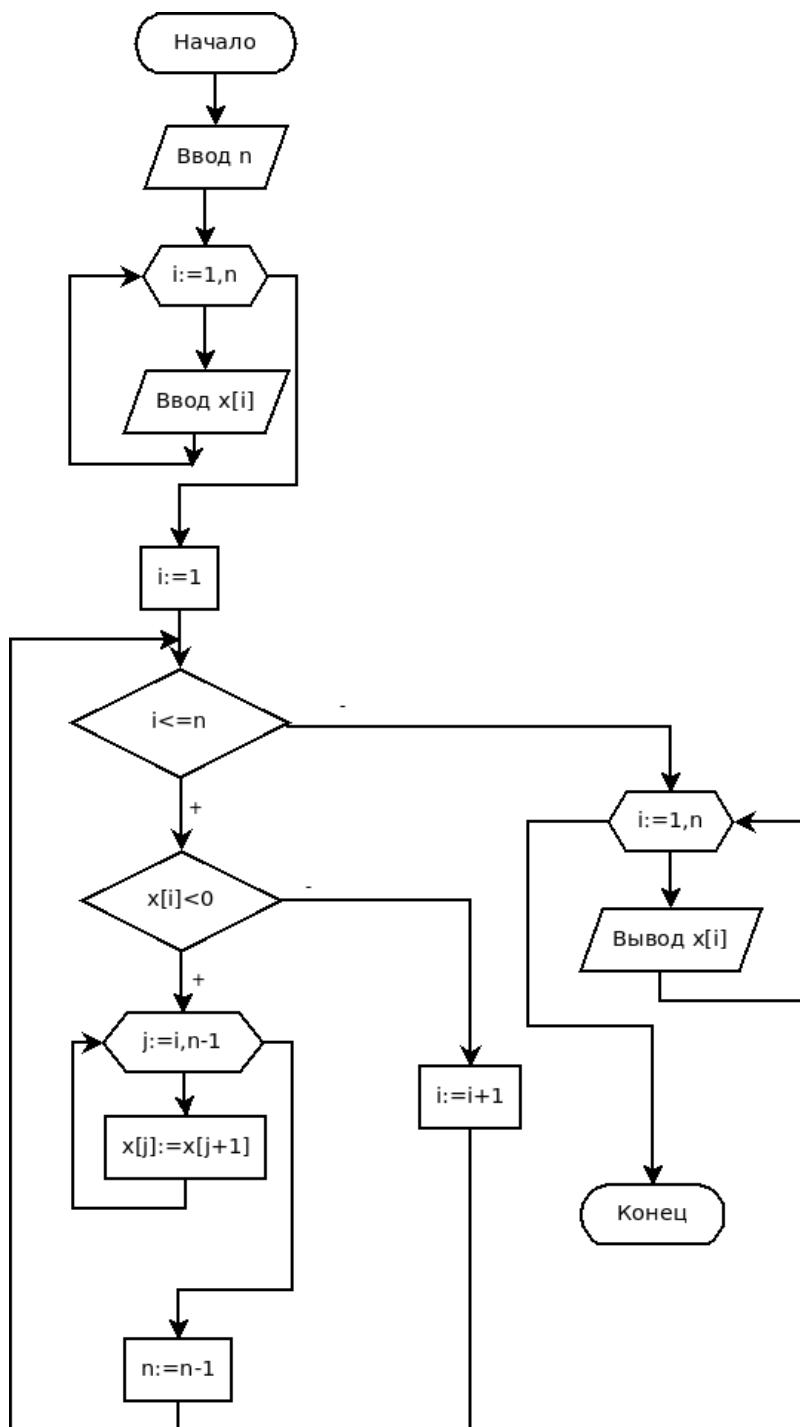


Рис. 5.28. Блок-схема решения задачи 5.1
Результаты работы программы представлены на [рис. 5.29](#).

```

Файл Правка Вид Терминал Вкладки Справка
Ввeдите размер массива
7
X[1]=1
X[2]=-2
X[3]=3
X[4]=-4
X[5]=5
X[6]=-6
X[7]=7
Массив X
1.00 -2.00 3.00 -4.00 5.00 -6.00 7.00
Измененный массив
1.00 3.00 5.00 7.00
-----
(program exited with code: 0)
Press return to continue

```

Рис. 5.29. 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 четвёртым элементами.

Для решения этой задачи необходимо все элементы массива, начиная со четвёртого, сдвинуть вправо на один элемент. Затем в четвёртый элемент массива нужно будет записать b ($X[4]:=b$). Но чтобы не потерять соседнее значение, сдвигать на один вправо нужно сначала десятый элемент, затем девятый, восьмой и т. д. до четвёртого. Блок-схема алгоритма вставки приведена на [рис. 5.30](#).

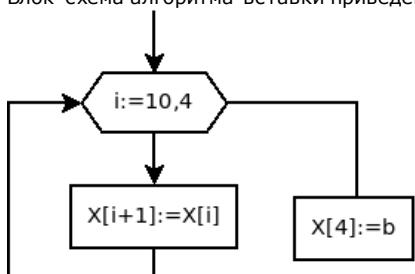


Рис. 5.30. Вставка числа b между третьим и четвёртым элементов массива X
В общем случае блок-схема вставки числа b в массив $X(N)$, между элементами с номерами m и $m+1$ представлена на [рис. 5.31](#).

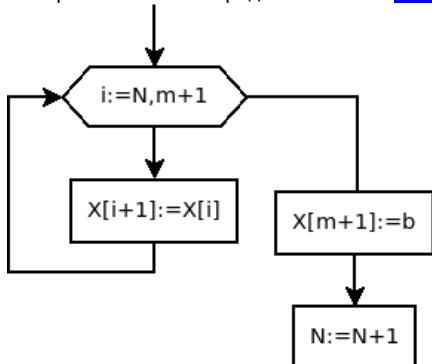


Рис. 5.31. Вставка числа b в массив X
Ниже представлен фрагмент программы, реализующий этот алгоритм²¹.

```

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, ' ' );

```

```

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 ('Изменённый 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 тип²¹, а затем объявлять процедуру:

```

type
типа_массива = array [ список_индексов ] of тип;
procedure
имя_процедуры(имя_массива : типа_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 массивом²¹, должна осуществляться аналогично:

```

type
типа_строки = string [ длина_строки ];
procedure
имя_процедуры(имя_строки : типа_строки );
...
Например:
type
stroka_5:string [ 5 ];
stroka_10:string [ 10 ];
function fun ( S t r : stroka_5 ) : stroka_10;

```

Массивы в подпрограмму можно передавать, используя понятие открытого массива. Открытый массив — это массив²¹, при описании которого указывается тип элементов, из которых он состоит, 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_открытого_массива, список_границ_индексов);

Для освобождения выделенной памяти нужно выполнить оператор:

имя_открытого_массива:=**NIL**;

Нижняя граница открытого массива (минимальное значение номера элемента) всегда равна нулю. Верхняя граница (максимальное значение номера элемента) возвращается стандартной функцией:

high (имя_открытого_массива)

Открытые массивы можно использовать при обычной обработке массивов в языке Free Pascal. Рассмотрим использование открытого массива на примере простейшей задачи нахождения суммы элементов массива.

```

var x : array of real; //Описание открытого массива.
    s : real;
    i, n : integer;
begin
write ( ' n= ' );
readln ( n );
//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_процедуры(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 А 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 С 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:

```

//Описание типа: массив целых чисел,
//пронумерованных от 0 до 10.
type massiv=array [ 0.. 10 ] of integer;
//Процедура предназначена для вывода на экран
//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 умолчанию этот размер определяется всей доступной памятью ПК.

Динамическое размещение данных осуществляется компилятором непосредственно в процессе выполнения программы. При динамическом размещении заранее не известно 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.

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 бы 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 в Паскале 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 `heapptr`.

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 `heapptr`. Сам `heapptr` увеличивает своё значение на два, так как длина внутреннего представления типа `integer`, связанного с указателем `i`, составляет 4 байта. Оператор `new(r)` вызывает ещё одно смещение указателя `heapptr`, но уже на 8 байт, 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 * p i;
```

Таким образом, значение, на которое указывает указатель, то есть собственно данные, размещённые в куче, обозначаются значком `^`, который ставится сразу за указателем. Если за последним этот значок отсутствует, то имеется в виду адрес, по которому размещаются данные. Динамически размещённые 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 серьёзным ошибкам!

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 значение кучи `heapptr` (обращение с помощью `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 освобождённому участку памяти возникает ошибка.

После рассмотрения основных принципов и процедур работы с указателями возникает вопрос: а зачем всё 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 её запуске, 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_mas3`).

```
program 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.
```

```
program din_mas3;
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;
```

```

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 5.32](#) видно, что первый отрицательный элемент хранится в массиве **A** под номером три, второй и третий под номерами пять и шесть соответственно, а четвёртый 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, приведённой на [рисунке 5.33](#) 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 5) последовательно перебираем все элементы **A**, если очередной элемент массива **A** отрицателен, 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 **m** чисел.

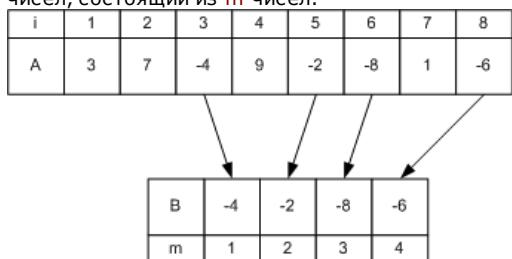


Рис. 5.32. Процесс формирование массива **B** из отрицательных элементов массива **A**

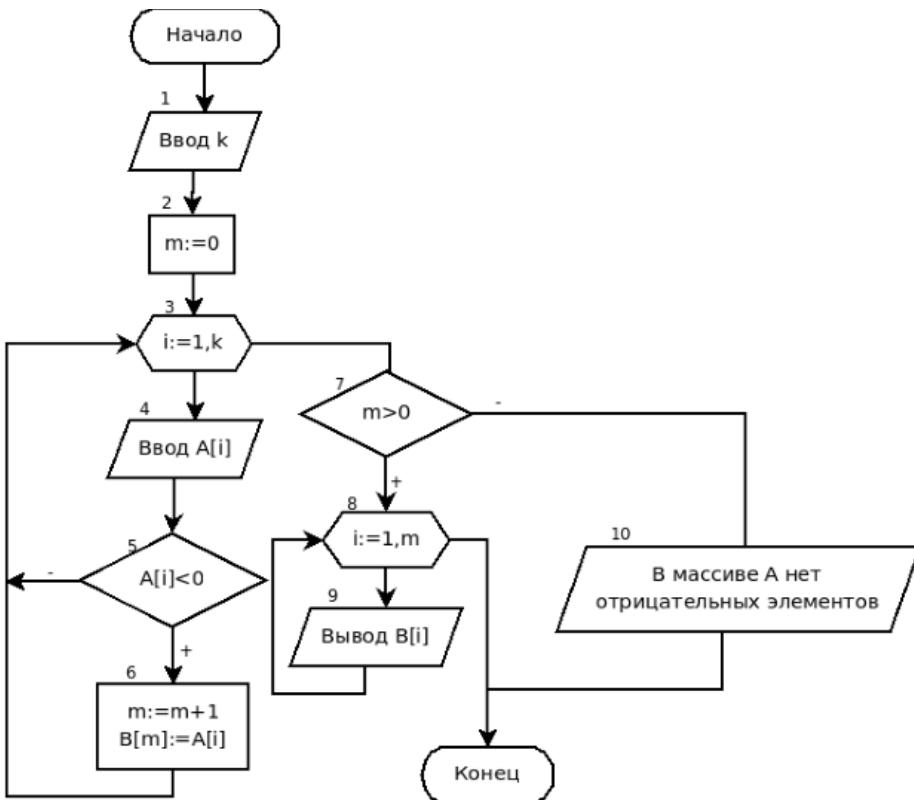


Рис. 5.33. Блок-схема формирования массива В из отрицательных элементов массива А

Приведённая ниже программа реализует описанный алгоритм.

```

var a, b : array [ 1.. 200 ] of word;
k,m, i : byte;
begin
write ( 'введите размерность массива k= ' );
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
z
целых чисел. Сформировать массив

y
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. 5.34](#).

Текст программы с комментариями приведён ниже.

```

program mas_four;
var y, z : array [ 1.. 50 ] of integer;
i, k, n : 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;
//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.

```

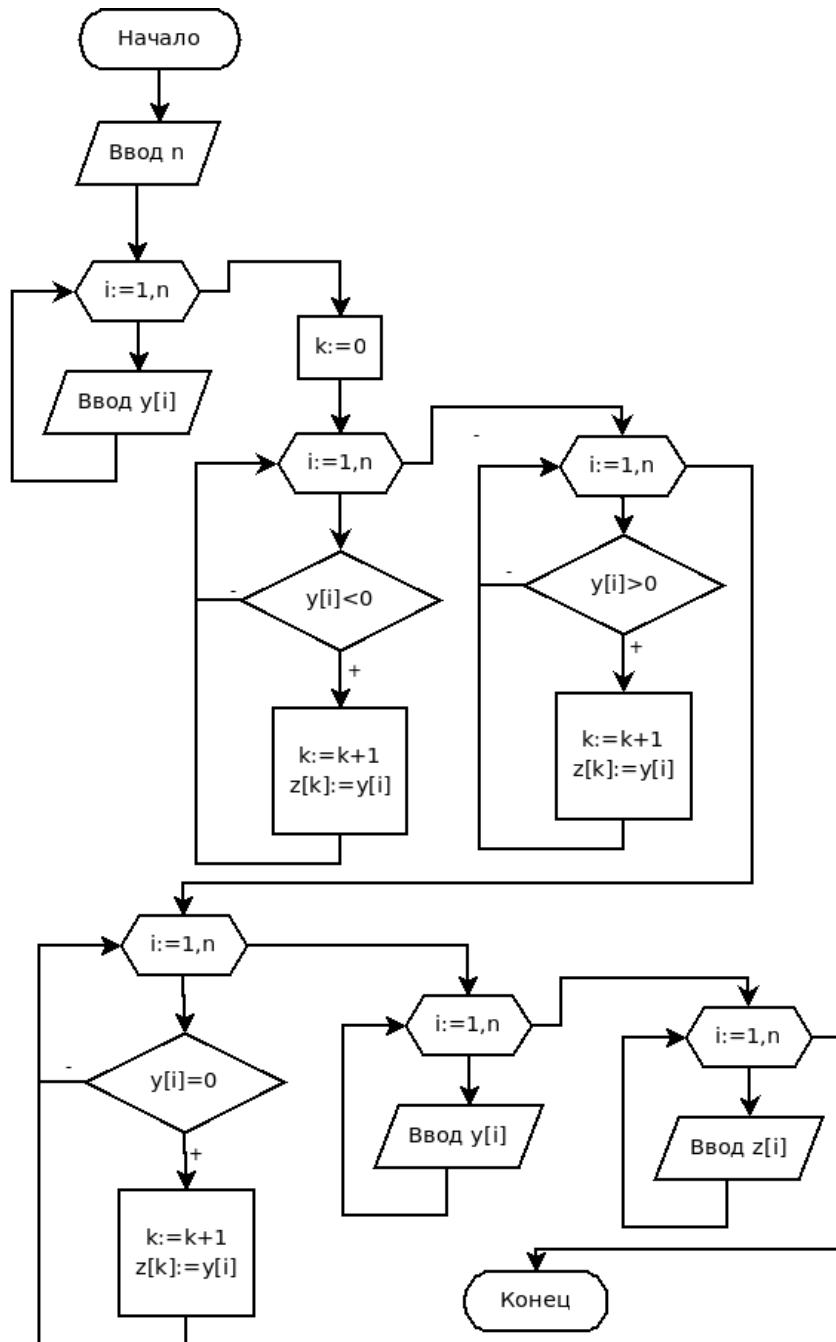


Рис. 5.34. Блок-схема решения задачи 5.4

ЗАДАЧА 5.5. Переписать элементы в массиве

X

в обратном порядке.

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 поменять с элементом **n+1-i**. Блок-схема обмена элементов в массиве представлена на [рис. 5.35](#).

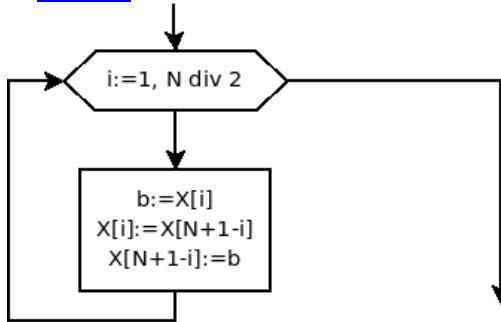


Рис. 5.35. Фрагмент блок-схемы к задаче 5.5

Ниже приведён текст консольного приложения задачи 5.5.

```

program mas_five;
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 местами
//элементы 1-й с n-м, 2-й с (n-1)... i-й с (n+1-i)-м.
for i :=1 to n div 2 do
begin
b:=x [ n+1 - i ];
x [ n+1 - i ] : = x [ i ];
x [ i ] : = b;
end;
//Вывод 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. 5.36](#).

Текст программы с комментариями приведён 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
    if x [ j ]=0 then {Если нашли нулевой элемент, то}
begin
        k:=k+1; {посчитать его номер}
        if k>4 then break
        {Если k превышает 4, то выйти из цикла.}
        Else {Иначе удаляем j-й элемент из массива.}
            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.}
for i :=1 to n-k do
write (X[ i ], ' ');
end.
```

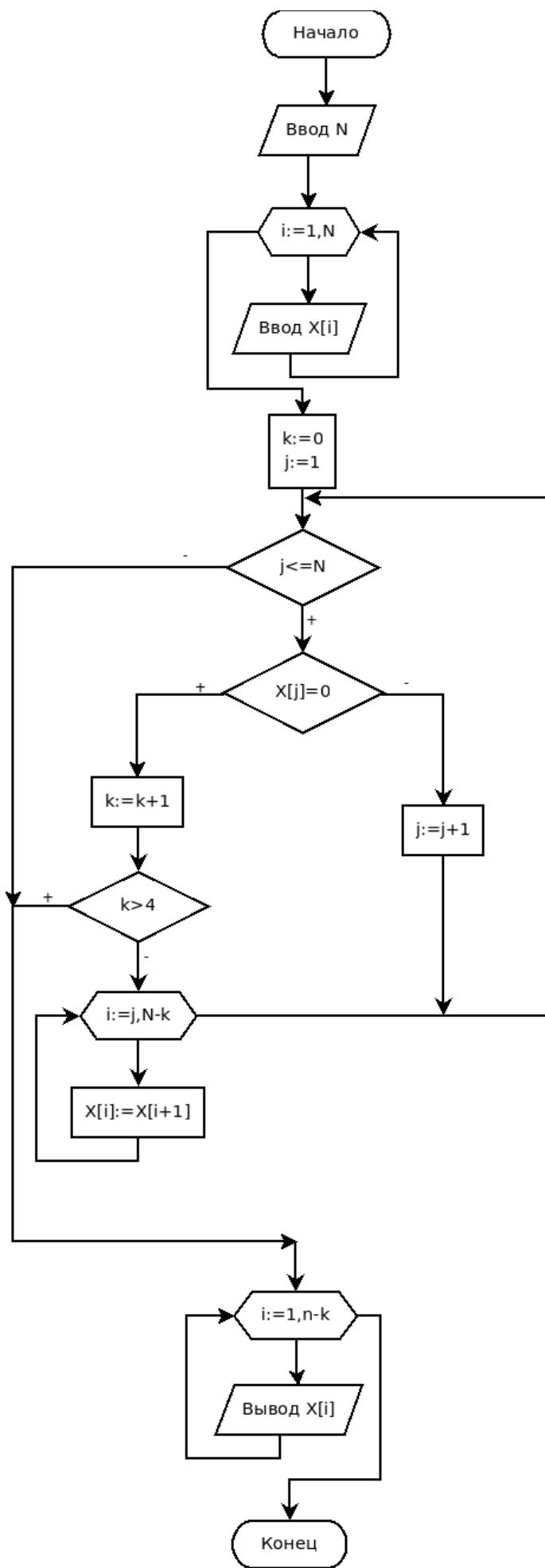


Рис. 5.36. Алгоритм решения задачи 5.6
ЗАДАЧА 5.7. Найти сумму простых чисел целочисленного массива

C(N)

Идея алгоритма состоит в следующем. Вначале 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`. Блок-схема этой функции представлена на [рис. 5.37](#).

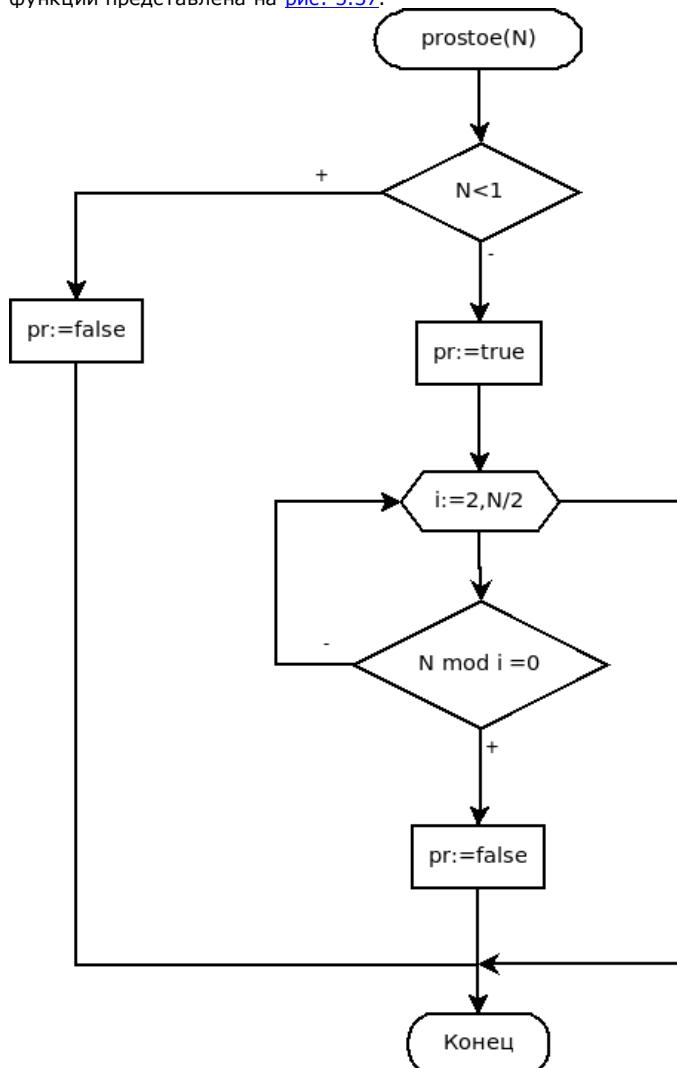


Рис. 5.37. Блок-схема функции `prostoe`

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`.

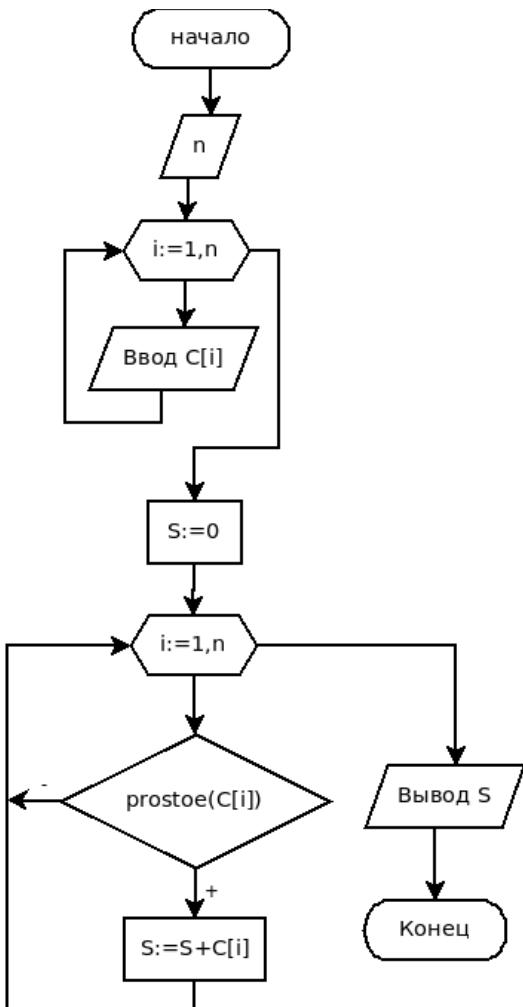


Рис. 5.38. Блок-схема решения задачи 5.7

Блок-схема решения задачи 5.7 изображена на рис. 5.38.

Ниже приведён текст программы, реализующей этот алгоритм, с комментариями.

```

program mas7;
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;
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, ' - 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. 5.39). Если есть, то вывести на экран количество таких серий.

1	2	3	-1	4	-7	-9	-1	3	-5	-1	-1	4	-2	6	1
↑ Серия из 4-х элементов					↑ Серия из 3-х элементов					↑ Серия из 4-х элементов					

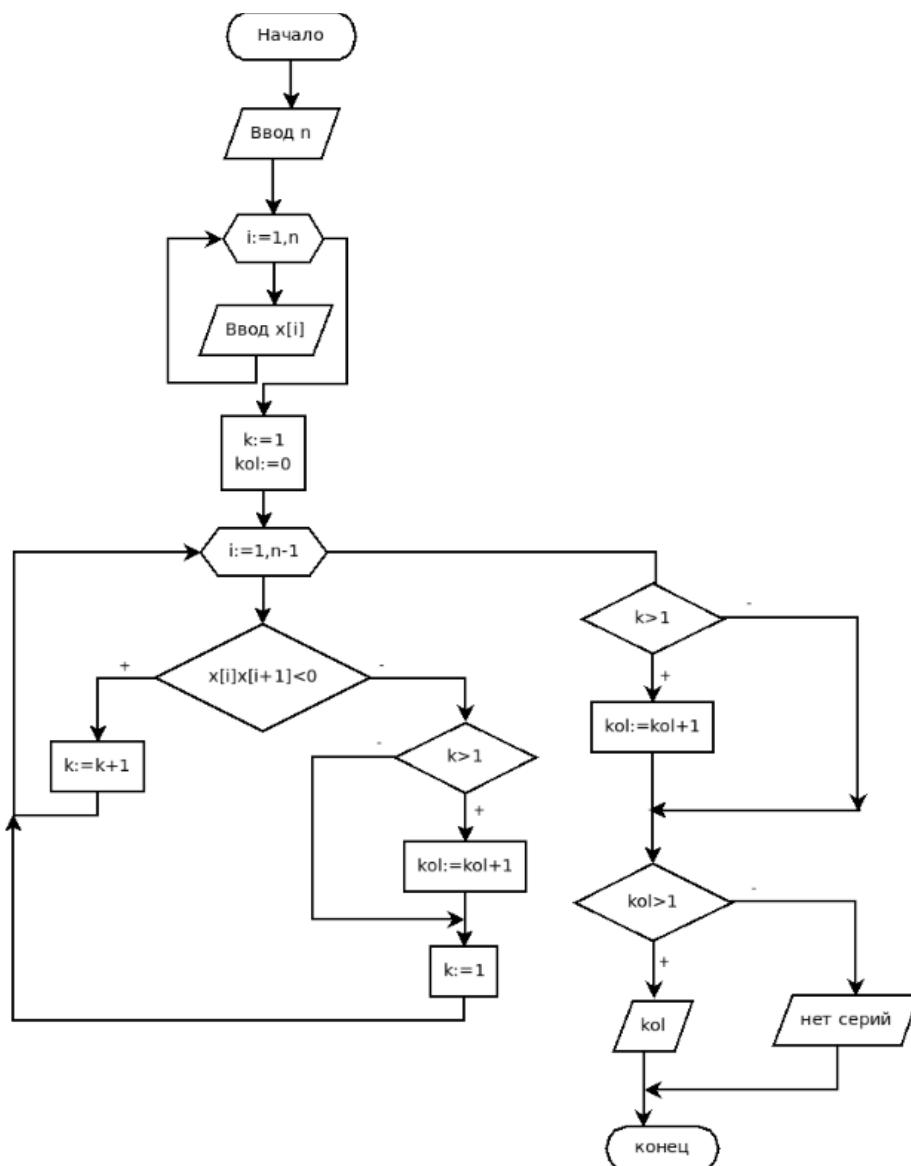
Рис. 5.39. Массив с тремя сериями знакочередующихся элементов

Идея алгоритма состоит в следующем. Вначале количество серий (*kol*) равно нулю, а длина серии (*k*) равна единице¹¹. Последовательно в цикле от 1 до *n-1* сравниваются соседние элементы (первый и второй, второй и третий,..., предпоследний и последний). Если произведение соседних элементов отрицательно, то количество элементов в серии¹² увеличиваем на 1. Если произведение неотрицательно, то возможны два варианта: либо сейчас оборвалась серия из знакочередующихся элементов, либо встретилась очередная пара элементов одного знака. Выбор из этих двух вариантов можно осуществить, сравнив переменную *k* с единицей. Если *k>1*, то это означает, что серия знакочередующихся элементов оборвалась и количество серий *kol* надо увеличить на 1, и после обрыва серии количество элементов в ней положить равным одному (*k=1*). После выхода из цикла проверяем, не было ли в конце массива серии из знакочередующихся элементов. Если такая серия была (*k>1*), то количество серий (*kol*) опять увеличиваем на 1. В завершении выводим количество серий из знакочередующихся элементов — переменную *kol*. Блок-схема решения задачи 5.8 представлена на [рис. 5.40](#).

Ниже приведён 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.}
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 счётчик подсчёта количества}
{серий.}
if k>1 then kol := kol +1;
{Подготовить показатель продолжения серии}
{k 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.
```

Далее рассмотрим чуть более сложную задачу на серии.



[увеличить изображение](#)

Рис. 5.40. Блок-схема решения задачи 5.8

ЗАДАЧА 5.9. В заданном массиве найти самую длинную серию элементов, состоящую из единиц.

Для максимальной серии будем хранить её длину (**max**) и 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 трёх параметров: первый элемент серии, последний элемент серии, длина серии. В связи с тем, что мы фиксируем серию в момент её окончания, то в качестве параметров серии будем использовать последний элемент серии (**kon**) и её длину (**k**).

Алгоритм решения этой задачи следующий. Вначале количество серий (**kol**) и её длина (**k**) равны нулю. Перебираем последовательно все элементы, если текущий элемент равен 1, то количество элементов в серии **k** увеличиваем на 1. Если текущий элемент не равен 1, то возможны два варианта: либо сейчас оборвалась серия из единиц, либо встретился очередной неравный 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**) и длину серии (**dlina:=k**). После этого необходимо проверить порядковый номер серии. 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 нулю 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 её так же, как и серию, которая встретилась в цикле.

Блок-схема решения задачи приведена на [рис. 5.41](#).

Ниже приведён текст консольного приложения решения задачи.

```

var x : array [.. 50] of integer;
n, i, k, kol, kon, max, kon_max, dlina : integer;
begin

```

```

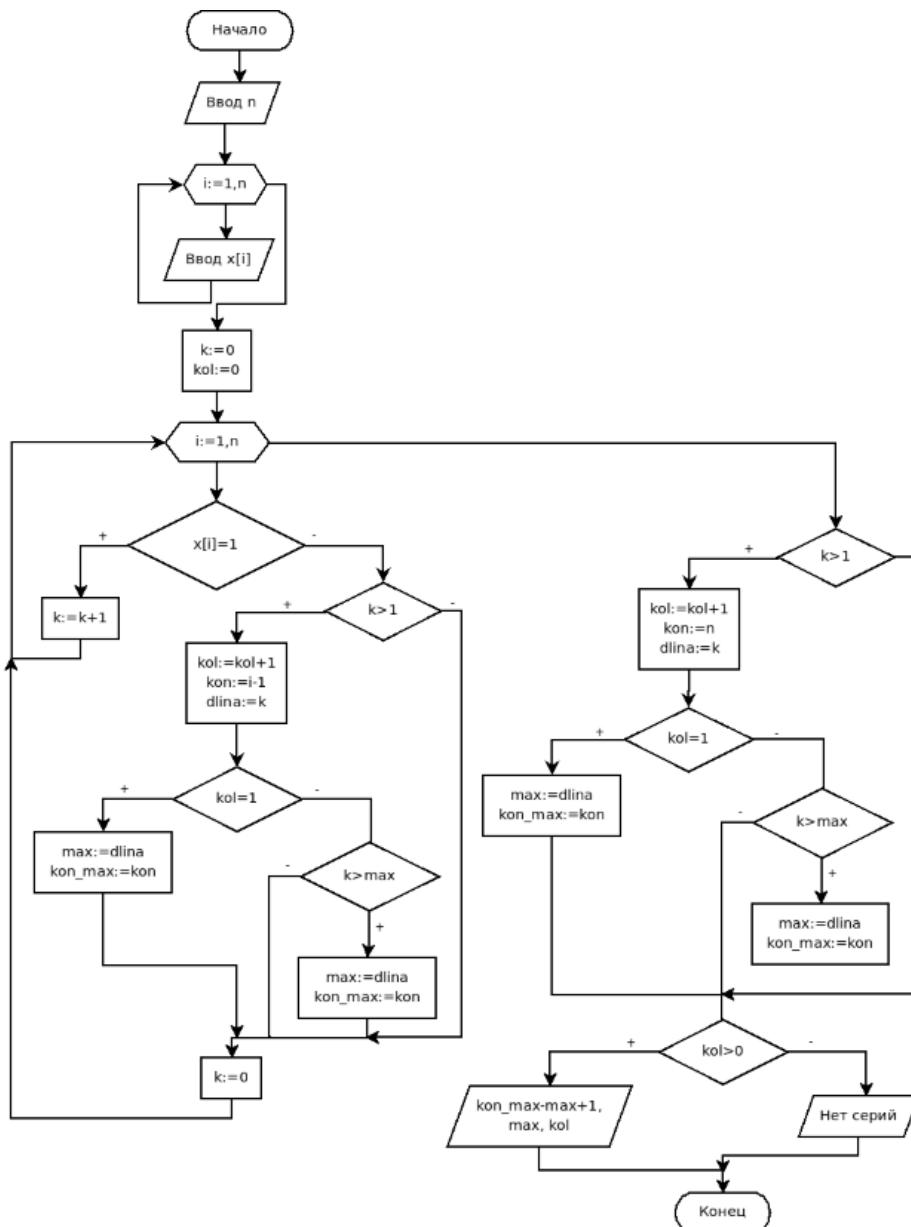
{Ввод размера массива.}
write( ' n= ' );
readln ( n );
{Ввод массива}
writeln ( 'Массив X ' );
for i :=1 to n do
read ( x [ i ] );
{Начальное присваивание длины серии и количества серий}
k : = 0; {Длина серии.}
kol : = 0; {Количество серий в массиве.}
{Перебираем все элементы в массиве}
for i :=1 to 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 её 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 0, 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 счётчик ещё 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,
        ', её длина равна ', max)
    end

```

```

{Вывод информации об отсутствии серий.}
else
writeln ( 'Нет серий, состоящих из единиц' )
end.

```



[увеличить изображение](#)

Рис. 5.41. Блок-схема решения задачи 5.9

ЗАДАЧА 5.10. Задан массив вещественных чисел. Перевести все элементы массива в *p*-ричную систему счисления.

Перед решением всей задачи давайте разберёмся с алгоритмом перевода вещественного числа из десятичной 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 другую систему счисления

Умножить дробную часть числа на основание системы счисления. В полученном произведении выделить целую часть числа, это будет старший разряд числа, который необходимо будет умножить на 10^1 .

. Дробную часть опять умножить на основание системы счисления. В произведении

целая часть будет очередным разрядом (его надо будет умножить на 10^{-2}), а дробную часть необходимо опять умножить на основание системы счисления до получения необходимого количества разрядов исходного числа.

Блок-схема функции перевода вещественного числа

N

из десятичной системы счисления в другую систему представлена на [рис. 5.42](#).

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 с

0

, для формирования степеней десяти вводится переменная

q

, которая вначале равна

1

, а затем в цикле последовательно умножается на

10

. При переводе дробной части числа последовательно нужны отрицательные степени

$10 : 10^{-1}, 10^{-2}, \dots$

Поэтому при формировании дробной части числа переменная

$q := 0.1$

, которая в цикле последовательно делится на

10

. Ниже приведён текст консольной программы решения задачи 5.10 с комментариями.

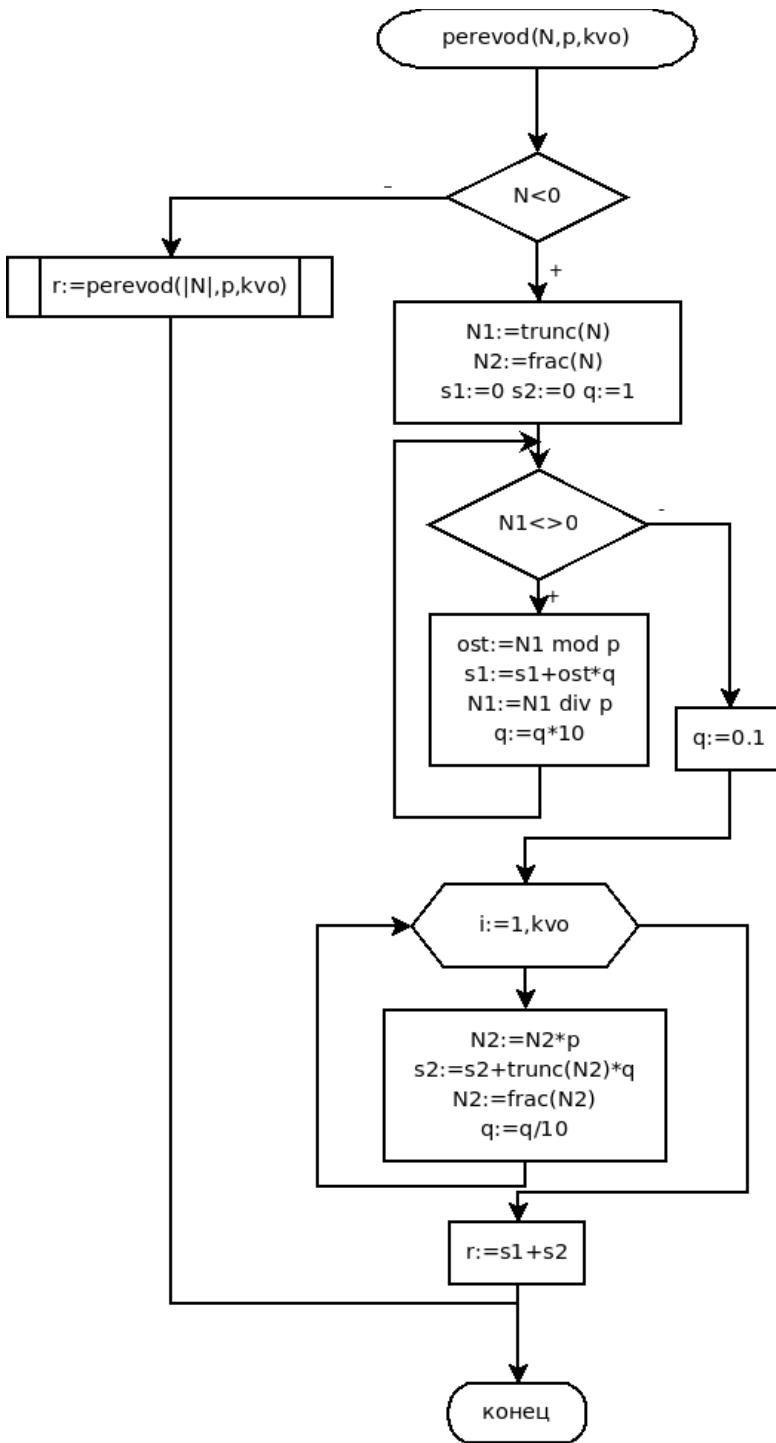


Рис. 5.42. Блок-схема функции перевода вещественного числа в р-ричную систему счисления

{Функция перевода вещественного числа в р-ричную систему счисления.}
 {Входные параметры функции: вещественное число N, основание системы}
 {счисления - целое число p, kvo - количество разрядов в дробной 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 вещественного 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 в цикле будем}
 {последовательно умножать q на 10.}
 q := 1;

```

{Перевод целой части числа, пока число не станет 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 * 1 0;
    end;
{В переменной q будем последовательно хранить отрицательные степени}
{десяти, вначале туда записываем 0.1 - десять в минус первой}
{степени, а затем в цикле будем последовательно 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 часть 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 систему 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.

```

```

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

```

[увеличить изображение](#)

Рис. 5.43. 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 десятичной 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 приведён 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
{Достаём 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 числа N в}
{восьмеричном представлении не образуют убывающую последовательность}
{((pr:=false) - аварийно покидаем цикл.}
                if tsifra <=tsifra_st then
                    begin
                        pr := false;
                        break;
                    end;
                tsifra_st:= tsifra;
            end;
        vosem:= pr; end;

```

Алгоритм решения задачи 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 ≤ 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. 5.44](#).

Свойства основных компонентов представлены в [таблицах 5.9—5.10](#).

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

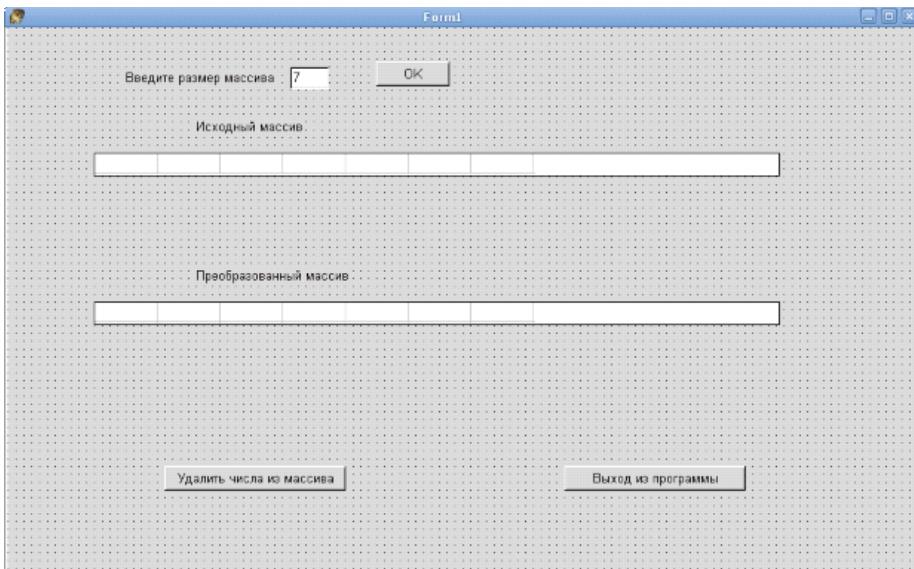
Таблица 5.9. Свойства меток, кнопок и текстового поля

Name	ColCount	RowCount	Visible	FixedCols	FixedRows	Options.goEditing
StringGrid1	7	1	false	0	0	true
StringGrid2	7	1	false	0	0	false

Таблица 5.10. Свойства таблиц строк

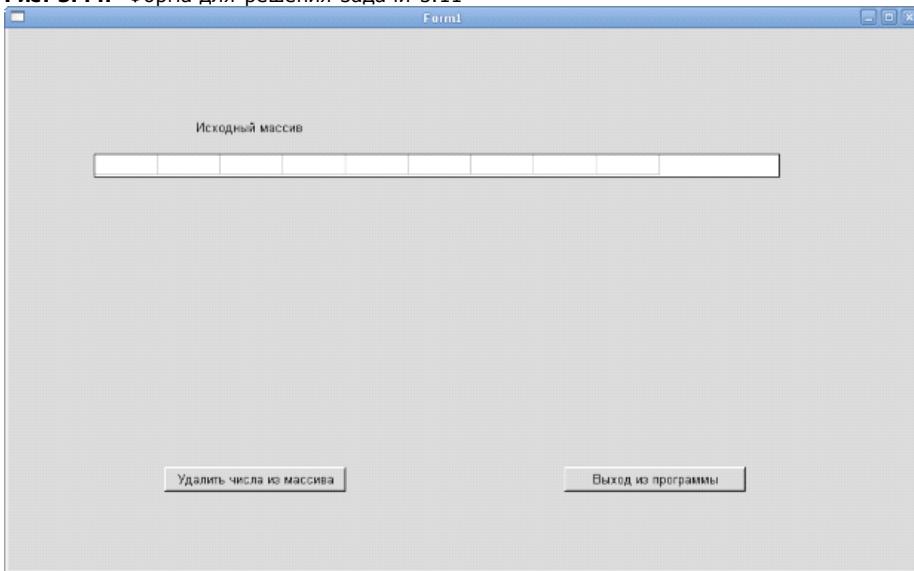
При запуске приложения видимы метка **Label1**, поле для ввода размера массива **Edit1** и кнопка **Button1**.

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. 5.45](#).



[увеличить изображение](#)

Рис. 5.44. Форма для решения задачи 5.11



[увеличить изображение](#)

Рис. 5.45. Окно приложения после щелчка по кнопке ОК

При щелчке по кнопке **Удалить числа из массива** происходят следующие действия:

- считывание массива из **StringGrid1**;
- удаление из массива 3-х последних чисел, цифры которых в восьмеричном представлении образуют убывающую последовательность;
- становятся видимыми метка **label3** и таблица строк **StringGrid2** для вывода элементов преобразованного массива.

Ниже приведён текст модуля с необходимыми комментариями.

```
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.. 1 0 0 ] of word;
var
    Form1 : TForm1;
    N: word;
    X: massiv;
implementation
{TForm1}
{Функция vosem проверки, образуют ли цифры числа N в восьмеричном}
{представлении убывающую последовательность.}
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 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 числа N в}
                {восьмеричном представлении не образуют убывающую последовательность}
                {(pr:=false) - аварийно покидаем цикл.}
                if t s if r a <=t s if r a _ s t 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 );
vari : 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 и 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;
{Обработчик 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 порядке.}
    for i :=N -1 downto 0 do
        {Если цифры очередного элемента массива в восьмеричном представлении}
        {образуют убывающую последовательность,}
        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 массива.}
    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.

```

В результате работы программы решения задачи 5.11 окно приложения примет вид, представленный на [рис. 5.46.](#)



[увеличить изображение](#)

Рис. 5.46. Окно с результатами решения задачи 5.11
Этой задачей мы заканчиваем раздел, посвящённый обработке массивов, и предлагаем читателю самостоятельно решить несколько задач.

Лекция 6. Обработка матриц в Паскале

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 integer;**

Описана матрица целых 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;

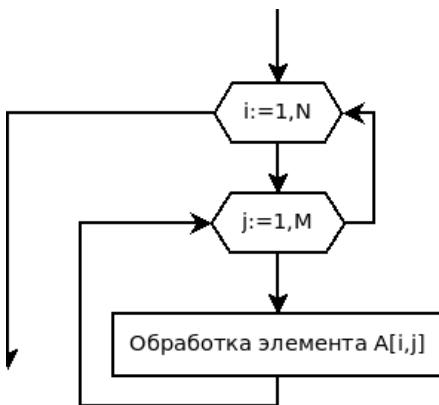


Рис. 6.1. Построчная обработка матрицы

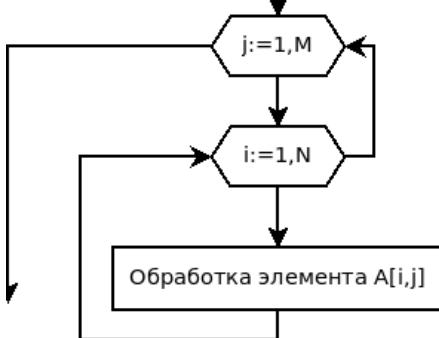


Рис. 6.2. Алгоритм обработки матрицы по столбцам

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 16 строк и 14 столбцов.

Для обращения к элементу матрицы необходимо указать её имя и в квадратных скобках через запятую номер строки и номер столбца:

имя [номер_строки, номер_столбца]

или

имя [номер_строки] [номер_столбца]

Например, **h[2,4]** — элемент матрицы **h**, находящийся в строке под номером два и столбце под номером четыре.

Для обработки всех элементов матрицы необходимо использовать два цикла. Если матрица обрабатывается построчно, то во внешнем цикле последовательно перебираются строки от первой до последней, затем во внутреннем — все (первый, второй, третий и т. д.) элементы текущей строки. При обработке элементов матрицы по столбцам внешний цикл будет перебирать столбцы, внутренний — строки. 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**.

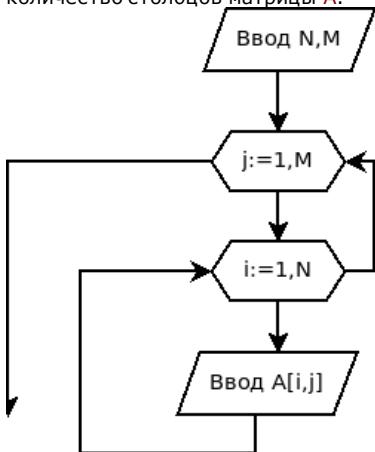


Рис. 6.3. Блок-схема ввода элементов матрицы

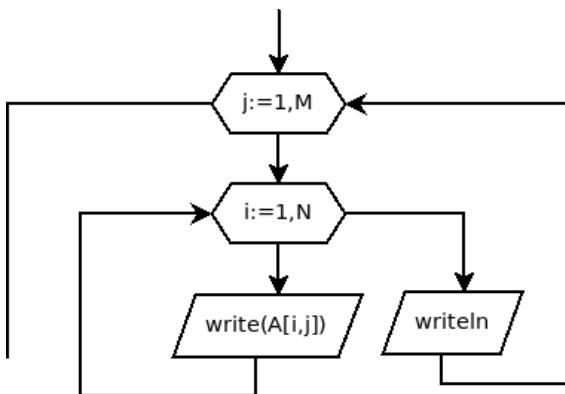


Рис. 6.4. Построчный вывод матрицы

Рассмотрим основные операции, выполняемые над матрицами при решении задач.

6.1 Ввод-вывод матриц

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 приведён на [рис. 6.4](#).

Об описании матриц на языке Паскаль было рассказано в разделе 5.2 главы 5, 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 её на экран монитора.

Ниже приведён пример консольного приложения ввода-вывода матрицы.

```
var
  a : array [ 1.. 2 0, 1.. 2 0 ] 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, ' ' ); {Печатается строка.}
    writeln {Переход на новую строку.}
end;
```

На [рис. 6.5](#) представлены результаты работы программы.

```

Файл Правка Вид Терминал Вкладки Справка
Введите количество строк и столбцов матрицы А
3 4
Введите матрицу
1 2 3 4
5 6 7 8
10 12 17 16
матрица А
 1.000    2.000    3.000    4.000
  5.000    6.000    7.000    8.000
 10.000   12.000   17.000   16.000

-----
(program exited with code: 0)
Press return to continue

```

Рис. 6.5. Результаты работы программы решения задачи 6.1
Ввод матрицы также можно организовать с помощью следующего цикла.

```

for i := 1 to N do
  for j := 1 to m do
begin
  write ('A( ', i, ', ', j, ')= ');
  readln (A[ i, j ] )
end;

```

Авторы предлагают читателю самостоятельно разобраться, в чём будет отличие ввода матрицы в этом случае.

Для ввода-вывода матриц можно использовать компонент типа TStringGrid, с которым мы познакомились в главе 5.

В качестве примера рассмотрим следующую задачу.

ЗАДАЧА 6.2. Составить программу транспонирования² матрицы

A

Блок-схема транспонирования матрицы приведена на [рис. 6.6](#). При транспонировании матрицы

A(N, M)

получается матрица *B*

(*M, N*)

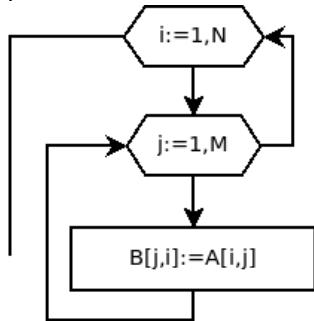


Рис. 6.6. Блок-схема транспонирования матрицы А

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.

Окно формы приложения представлено на [рис. 6.7](#).

Ниже приведён текст подпрограммы с комментариями, которая будет выполняться, если пользователь щёлкнет по кнопке Транспонирование матрицы.

Свойство	StringGrid 1	StringGrid 2	Описание свойства
Top	30	30	Расстояние от верхней границы таблицы до верхнего края формы
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

Таблица 6.1. Свойства компонентов StringGrid1, StringGrid2.

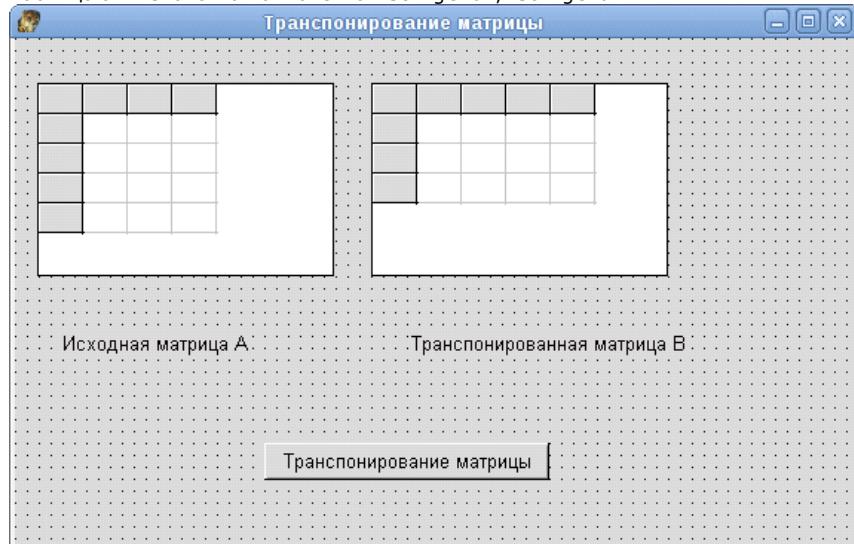


Рис. 6.7. Форма приложения транспонирования матрицы

```

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.
A: array [ 1.. n, 1..m] of integer; //Исходная матрица.
B: array [ 1.. m, 1.. n ] of integer; //Транспонированная матрица.
begin
//Исходные данныечитываются из ячеек таблицы на форме,
//и их значения записываются в двумерный массив A.
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 ] );
//Формирование транспонированной матрицы B, см. блок-схему на
//рис. 6.6.
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 ] );
end;

```

Результаты работы программы представлены на [рис. 6.8](#).

Транспонирование матрицы

1	2	3
4	5	6
7	8	9
10	11	12

Исходная матрица A

1	4	7	10
2	5	8	11
3	6	9	12

Транспонированная матрица B

Транспонирование матрицы

Рис. 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 так, как показано на [рис. 6.9.](#)

Транспонирование матрицы

Введите размерность матрицы:

N: M:

Ввод **Очистить**

Исходная матрица A :

Транспонирование

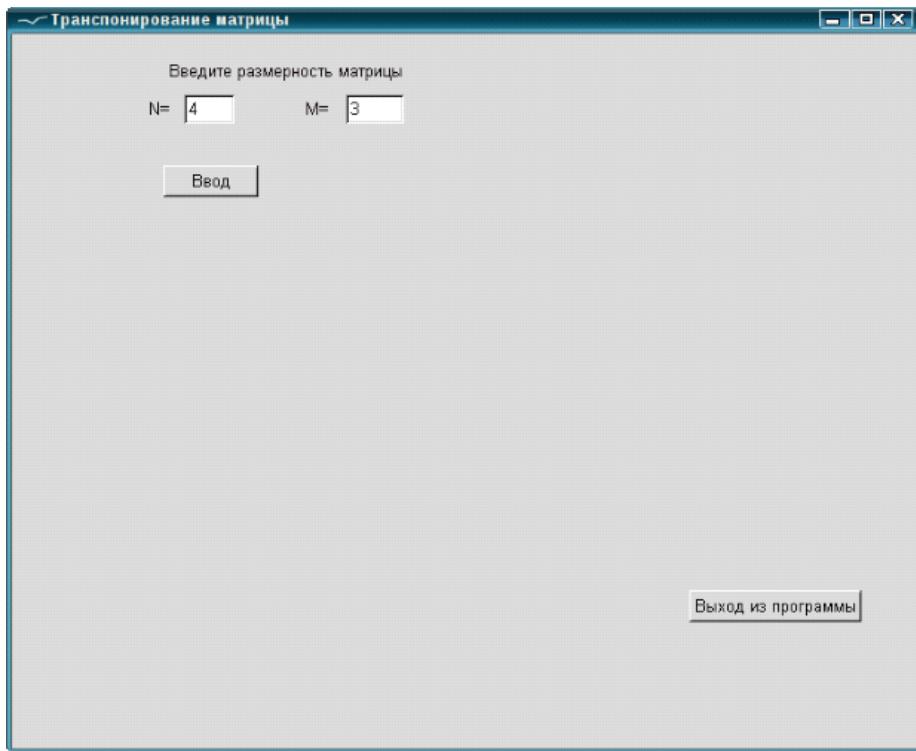
Транспонированная матрица B :

Выход из программы

Form1: TForm

[увеличить изображение](#)

Рис. 6.9. Окно формы решения задачи транспонирования матрицы **A(N, M)**



[увеличить изображение](#)

Рис. 6.10. Стартовое окно программы транспонирования матрицы A(N, M)
Установим свойство видимости (Visible) в False у компонентов метки label4, label5, StringGrid1, StringGrid2, кнопки Button2 и Button3. После этого при запуске программы будут видны только компоненты, отвечающие за ввод размеров матрицы, и кнопка Выход из программы ([см. рис. 6.10](#)). Матрицы A и B, их размеры N, M объявим глобально.

```
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;
    Label4. Visible := true; //соответствующая ей надпись,
    Button2. Visible := true; //кнопки "Очистить"
    Button3. Visible := true; //и "Транспонирование".
    with StringGrid1 do
    begin
//Определяем число строк (RowCount) и столбцов(ColCount) в
//компоненте StringGrid1.
    ColCount :=M+1;
    RowCount:=N+1;
//и нумеруем 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 выдаётся 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;

```

Теперь напишем обработчик кнопки Транспонирование. При щелчке по этой кнопке становится видимым компонент **StringGrid2**, 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 приведён 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 );
for i :=1 to ColCount -1 do //и столбцы компонента StringGrid2, в
Cells [ i, 0 ] := IntToStr ( i ); //котором отображается матрица B.
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 ] );
Butoon1.Visible := False;
end;

```

Осталось написать обработчик события при нажатии на кнопку Очистить. При щелчке по этой кнопке должно происходить следующее:

- очистка содержимого компонентов **StringGrid1**, **StringGrid2**;
- компоненты **StringGrid1**, **StringGrid2** и соответствующие им метки **label4** и **label5**, а также кнопки Транспонировать и Очистить становятся 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,
//label4, 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;

```

Мы получили работающую программу для транспонирования матрицы. На [рис. 6.11](#) представлены результаты транспонирования матрицы A(2,4).

Обратите внимание на использование оператора присоединения

with имя_компоненты **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 ] : = '';

```

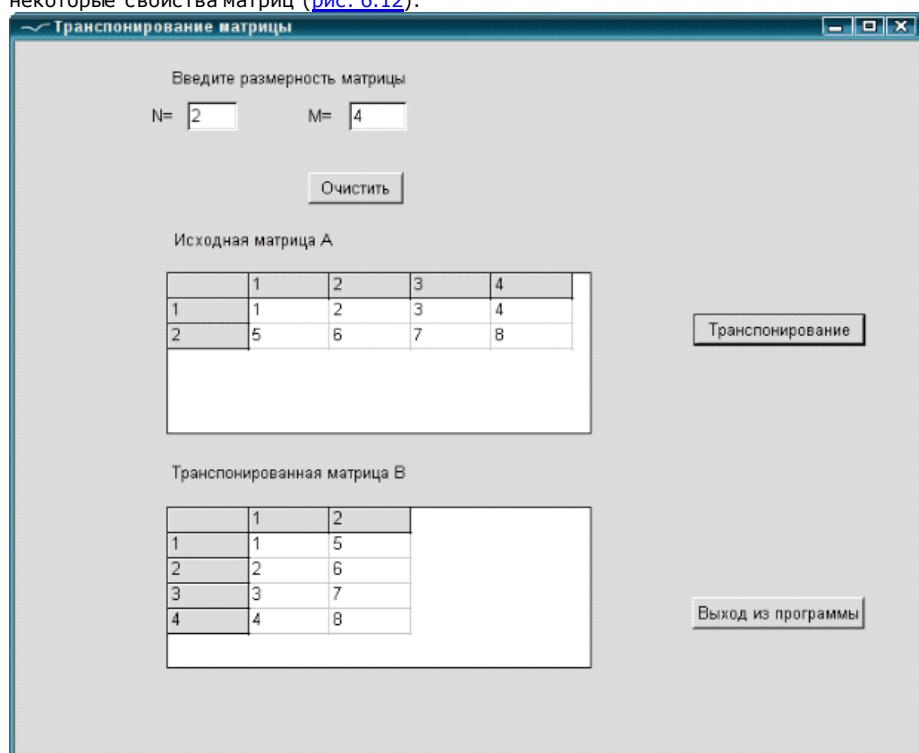
был использован оператор

```

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](#)):



[увеличить изображение](#)

Рис. 6.11. Транспонирование матрицы A(2,4)

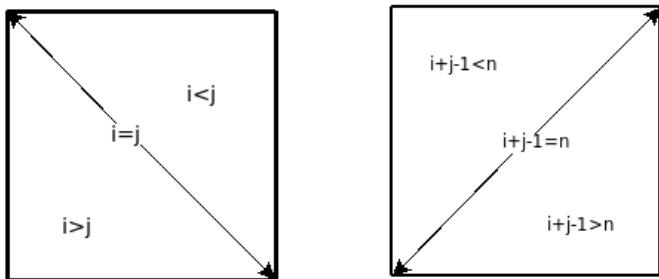


Рис. 6.12. Свойства элементов матрицы

- если номер строки элемента совпадает с номером столбца (), это означает, что элемент лежит на главной диагонали матрицы;
- если номер строки превышает номер столбца (), то элемент находится ниже главной диагонали;
- если номер столбца больше номера строки (), то элемент находится выше главной диагонали;
- элемент лежит на побочной диагонали, если его индексы удовлетворяют равенству ;
- неравенство характерно для элемента, находящегося выше побочной диагонали;
- соответственно, элементу лежащему ниже побочной диагонали, соответствует выражение .

6.2 Алгоритмы и программы работы с матрицами

Рассмотрим несколько примеров решения задач обработки матриц.

ЗАДАЧА 6.3. Найти сумму элементов матрицы, лежащих выше главной диагонали ([см. рис. 6.13](#)).

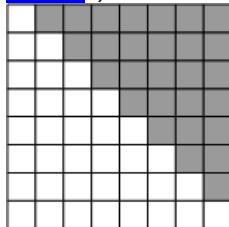


Рис. 6.13. Рисунок к задаче 6.3

Рассмотрим два алгоритма решения данной задачи.

Первый алгоритм решения данной задачи ([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**).

Текст консольного приложения с комментариями приведён ниже.

```

var
a : array [ 1.. 15, 1.. 10 ] of real;
i, j, n,m: integer; s : real;
begin
writeln ( 'введите размеры матрицы ' );
writeln ( 'n - количество строк, m - количество столбцов ' );
readln ( n,m);
writeln ( 'Введите матрицу 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
if j>i then {Если элемент лежит выше главной диагонали, то}
s := s+a [ i, j ]; {наращиваем сумму.}
writeln ( 'матрица A ' );
for i :=1 to n do
begin
for j :=1 to m do
{Здесь важен формат, особенно общая ширина поля!}
write ( a [ i, j ] : 8 : 3, ' ' );
writeln
end;
writeln ( 'сумма элементов матрицы ', s : 8 : 3 );
end.

```

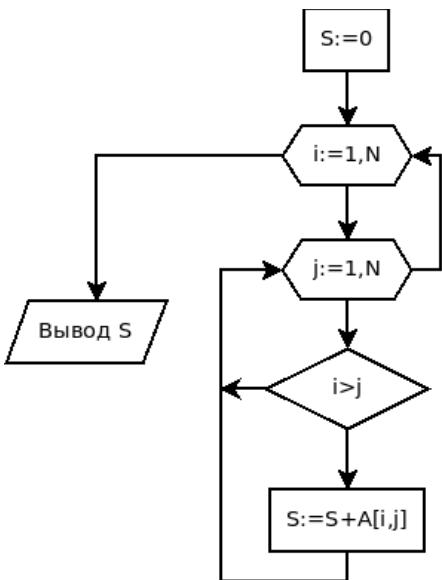


Рис. 6.14. Блок-схема задачи 6.3 (алгоритм 1)

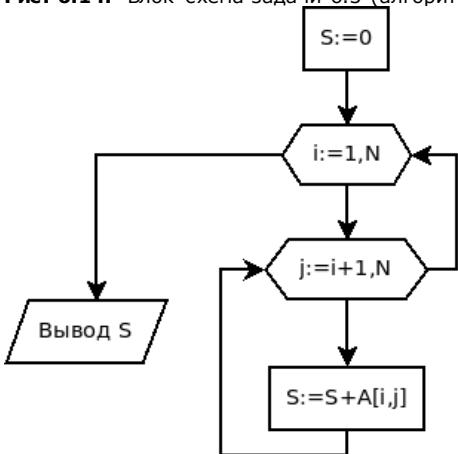


Рис. 6.15. Блок-схема задачи 6.3 (алгоритм 2)

Результаты работы программы представлены на [рис. 6.16](#).

Второй алгоритм решения этой задачи представлен на [рис. 6.15](#).

В нём проверка условия $i < j$ не выполняется, но, тем не менее, в нём также суммируются элементы матрицы, находящиеся выше главной диагонали. Для пояснения функционирования алгоритма обратимся к [рисунку 6.13](#). В первой строке заданной матрицы необходимо сложить все элементы, начиная со второго. Во второй — все, начиная с третьего, в i -й строке процесс суммирования начнётся 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.

Терминал

```

Файл Правка Вид Терминал Вкладки Справка
введите размеры матрицы
п - количество строк, м - количество столбцов
3 4
Введите матрицу A
1 2 3 4
5 6 7 8
9 11 12 13
матрица A
 1.000   2.000   3.000   4.000
 5.000   6.000   7.000   8.000
 9.000  11.000  12.000  13.000
сумма элементов матрицы 37.000

-----
(program exited with code: 0)
Press return to continue

```

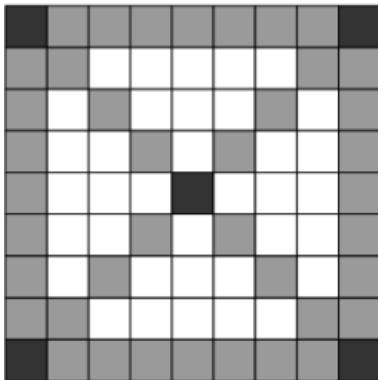
Рис. 6.16. Результаты работы программы решения задачи 6.3

ЗАДАЧА 6.4. Вычислить количество положительных элементов квадратной матрицы

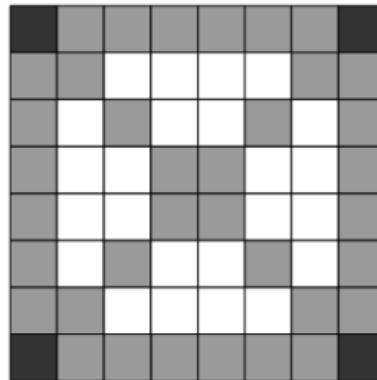
A

, расположенных по её периметру и на диагоналях.

Матрица из N строк и N столбцов



N- нечетное



N - четное

Рис. 6.17. Рисунок к условию задачи 6.4

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](#), причём чёрным 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 нечётное (на [рисунке 6.17](#) эта матрица расположена слева), то существует элемент с индексом ($N \text{ div } 2 + 1, N \text{ div } 2 + 1$), который находится на пересечении главной и побочной диагоналей. При чётном значении

N

(матрица справа на [рис. 6.17](#)) диагонали не пересекаются.

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

$A_{i,i}$

— элемент главной диагонали. Воспользовавшись свойством, характерным для элементов побочной диагонали, получим:

$$i + j - 1 = N \rightarrow j = N - i + 1$$

, следовательно, для строк $i=1,2,\dots,N$ 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 ($i=1,2,\dots,N$),

$A_{N,i}$

— элементы, расположенные в последней строке ($i=1,2,\dots,N$) и, соответственно,

$A_{i,1}$

— элементы, расположенные в первом столбце ($i=1,2,\dots,N$),

$A_{i,N}$

— в последнем столбце ($i=1,2,\dots,N$).

Алгоритм обработки построим следующим образом: сначала обработаем элементы, расположенные на диагоналях квадратной матрицы. Для этого необходимо в каждой строке ($i=1,2,\dots,N$) 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 угловые элементы матрицы уже учтены при проверке диагональных элементов, при обработке элементов, расположенных по периметру матрицы, их учитывать уже не нужно. Поэтому надо перебрать элементы со второго до предпоследнего в первой и последней строке, в первом и последнем столбце.

```

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 нужно проверить, не был ли элемент, находящийся на пересечении диагоналей, подсчитан дважды. Это могло произойти только в том случае, если **N** — нечётно и элемент, расположенный на пересечении диагоналей **4**, положителен.

```

if (N mod 2 <> 0) and ( a [ n div 2 + 1, n div 2 + 1 ] > 0) then
k:=k -1;

```

Ниже приведён полный текст консольного приложения решения задачи 6.4 с комментариями.

```

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: ' );
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 диагоналей, подсчитан дважды,}
{то уменьшить вычисленное 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.

```

На [рис. 6.18](#) представлены результаты работы программы решения задачи 6.4.

```

~ Терминал
Файл Правка Вид Терминал Вкладки Справка
N=5
Введите матрицу A
11 22 33 44 -5
-6 77 33 -8 99
22 33 44 55 66
17 -8 -9 99 37
64 76 -7 17 41
Была введена матрица A:
11 22 33 44 -5
-6 77 33 -8 99
22 33 44 55 66
17 -8 -9 99 37
64 76 -7 17 41
k=16

-----
(program exited with code: 0)
Press return to continue

```

Рис. 6.18. Результаты решения задачи 6.5

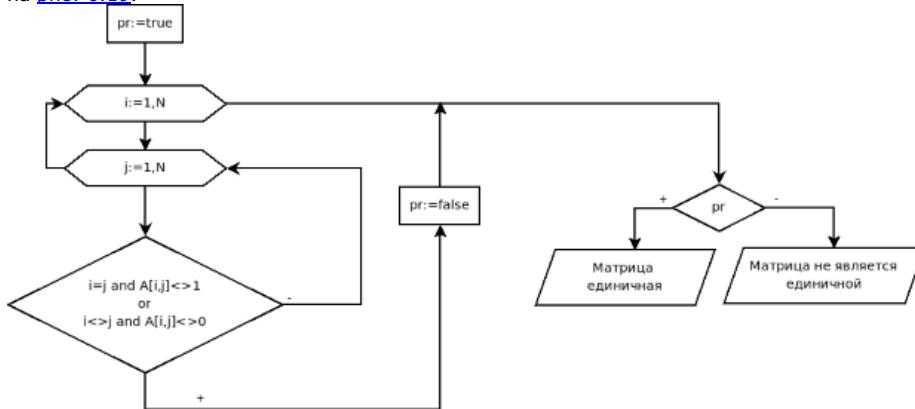
ЗАДАЧА 6.5. Проверить, является ли заданная квадратная матрица единичной. Единичной называют матрицу, у которой элементы главной диагонали — единицы, а все остальные — нули. Например,

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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 нулю²⁴, то в логическую переменную `pr` записываем значение `false` и прекращаем проверку (аварийно покидаем цикл). После цикла проверяем значение `pr`. Если переменная `pr` по прежнему равна `true`, то матрица единичная; в противном случае она такойой не является. Блок-схема алгоритма решения задачи представлена на [рис. 6.19](#).



[увеличить изображение](#)

Рис. 6.19. Блок-схема алгоритма решения задачи 6.5

```

program pr_6_5;
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
        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 действительно 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 диагонали и не равен нулю, то}
    begin
    {логической переменной присвоить значение "ложь"}
    {это будет означать, что матрица единичной не является,}
    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 заменён разностью минимального и максимального элемента в этом же столбце.

Для решения данной задачи необходимо найти в каждом столбце максимальный и минимальный элементы, после чего в последний элемент столбца записать их разность.

Блок-схема алгоритма решения приведена на [рис. 6.20](#).

Ниже приведён текст консольного приложения с комментариями.

```

program pr_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.

```

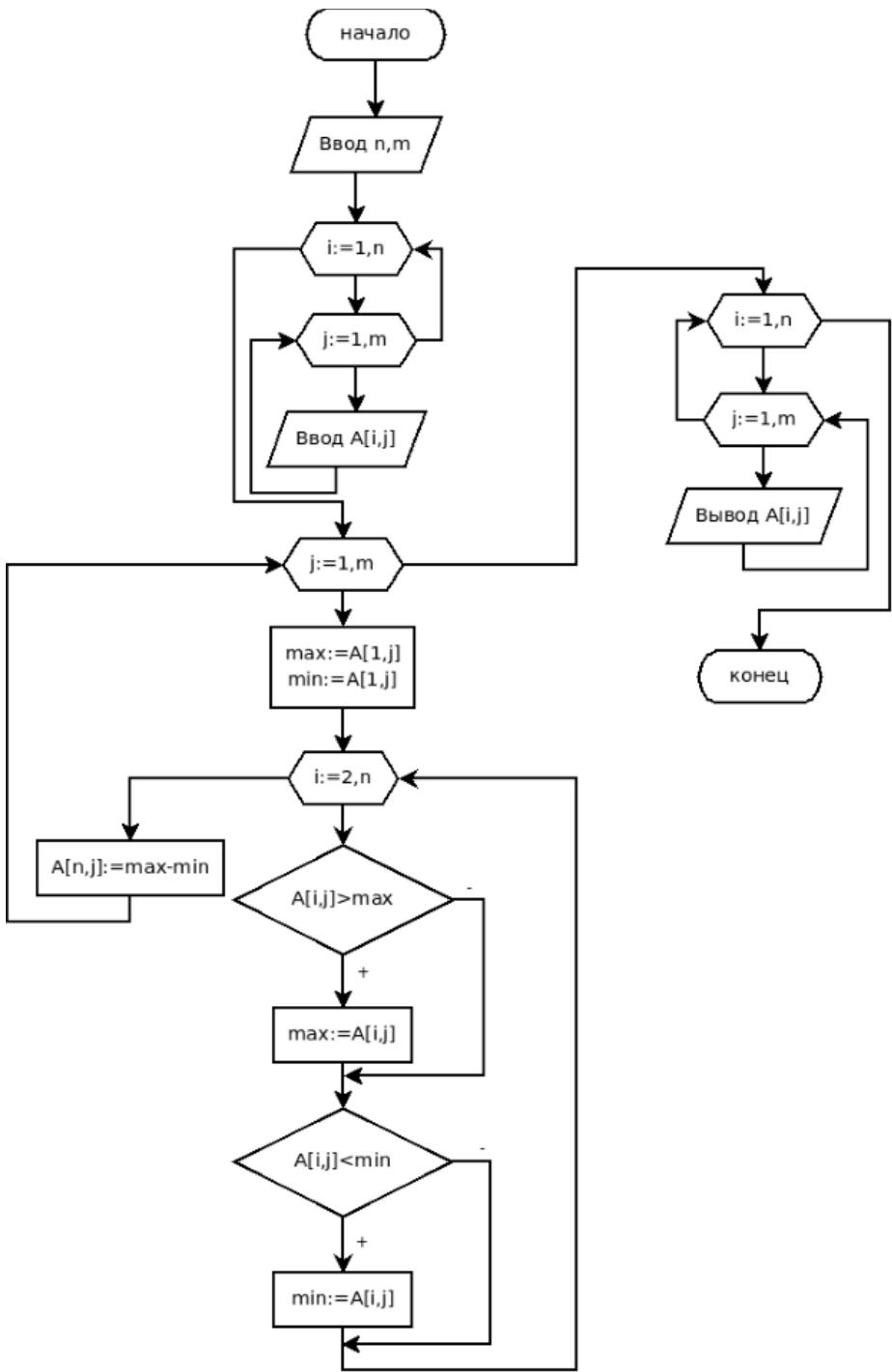
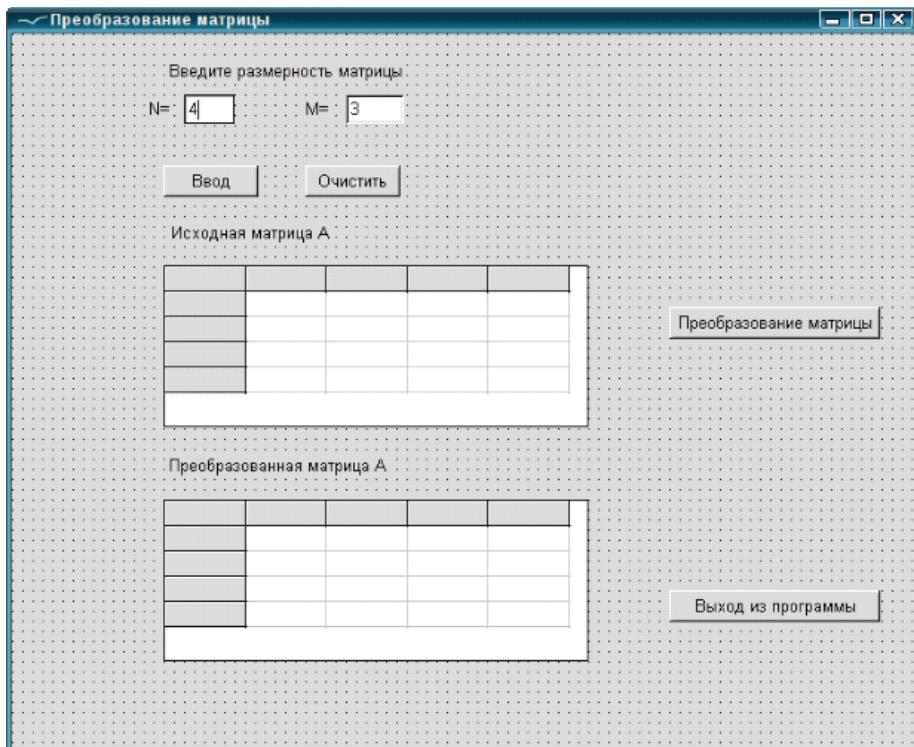


Рис. 6.20. Блок-схема алгоритма решения задачи 6.6

Теперь давайте создадим визуальное приложение, реализующее рассмотренный алгоритм. За основу возьмём форму ([см. рис. 6.9](#)) и проект транспонирования матрицы $A(N,M)$, разработанные для задачи 6.2. Окно формы несколько изменим ([рис. 6.9](#)). Кнопку Транспонирование переименуем в Преобразование матрицы, а свойством **Caption** метки **label5** установим Преобразованная матрица A . Кроме того, изменим свойство **Caption** формы ([см. рис. 6.21](#)). Окно изменённой формы представлено на [рис. 6.21](#).

Обработчики кнопок Ввод, Очистить, Выход из программы изменятся мало. Рассмотрим алгоритм работы обработчика кнопки Преобразование матрицы. В этом обработчике будет производиться считывание матрицы из компонента **StringGrid1**, преобразование матрицы A по алгоритму, представленному на [рис. 6.20](#), вывод преобразованной матрицы в компонент **StringGrid2**.



[увеличить изображение](#)

Рис. 6.21. Окно формы решения задачи 6.6

Текст модуля визуального приложения решения задачи 6.6 с комментариями приведён 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.. 2 5, 1.. 2 5 ] 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 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;
    //кнопки "Очистить"
        Button2. Visible := true;
    //и "Транспонирование"
        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 неверно! ',
        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 //и нумеруются
                Cells [ 0, i ] := IntToStr ( i ); //строки и
            for i :=1 to ColCount -1 do //столбцы второй матрицы.
                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 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;
//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 );
begin
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,
//label4, 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;
initialization
{$I unit1.lrs}
end.

```

На [рис. 6.22](#) представлено окно с результатами решения задачи.

ЗАДАЧА 6.7. Поменять местами

n

-й и

r

-й столбцы матрицы

A(K, M)

.
Задача сводится к обмену

n

-го и

r

-го элемента во всех строках матрицы.

Блок-схема приведена на [рис. 6.23](#).

Ниже приведён листинг консольного приложения с комментариями.

```

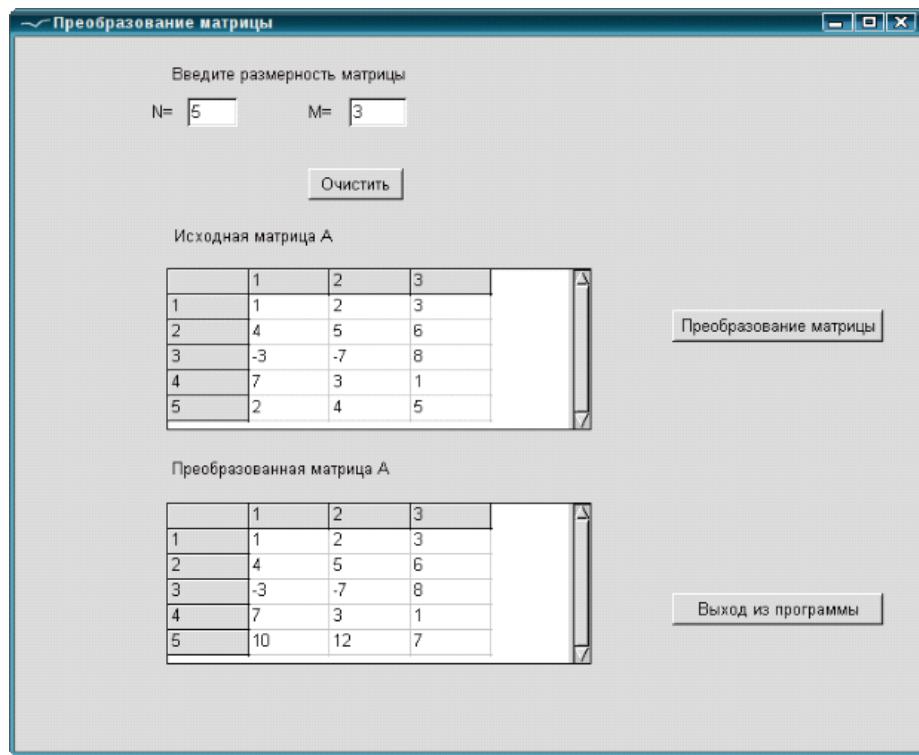
type matrica=array [ 1.. 15, 1.. 15 ] of real;
var
a : matrica;
i, j, k,m, n, r : byte; b : real;
begin
//Ввод размеров матрицы.
write ( 'k= ' ); readln ( k );
write ( 'm= ' ); readln ( m );
//Ввод матрицы A.
writeln ( 'Матрица A' );
for i :=1 to k do
for j :=1 to m do
read ( a [ i, j ] );
//Ввод номеров 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 );
{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.

```



[увеличить изображение](#)

Рис. 6.22. Результаты решения задачи 6.6

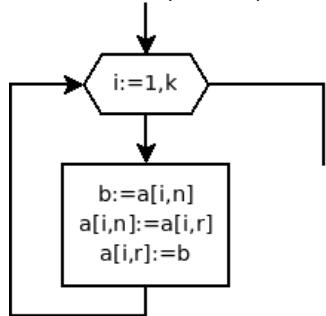


Рис. 6.23. Блок-схема алгоритма решения задачи 6.7
Результаты работы программы приведены на [рис. 6.24](#).

```

Файл Правка Вид Терминал Вкладки Справка
k=4
m=5
Матрица A
1 2 3 4 5
6 7 8 9 3
7 5 3 1 2
5 4 3 2 1
n=2
r=5
Преобразованная матрица A
1.000 5.000 3.000 4.000 2.000
6.000 3.000 8.000 9.000 7.000
7.000 2.000 3.000 1.000 5.000
5.000 1.000 3.000 2.000 4.000

-----
(program exited with code: 0)
Press return to continue

```

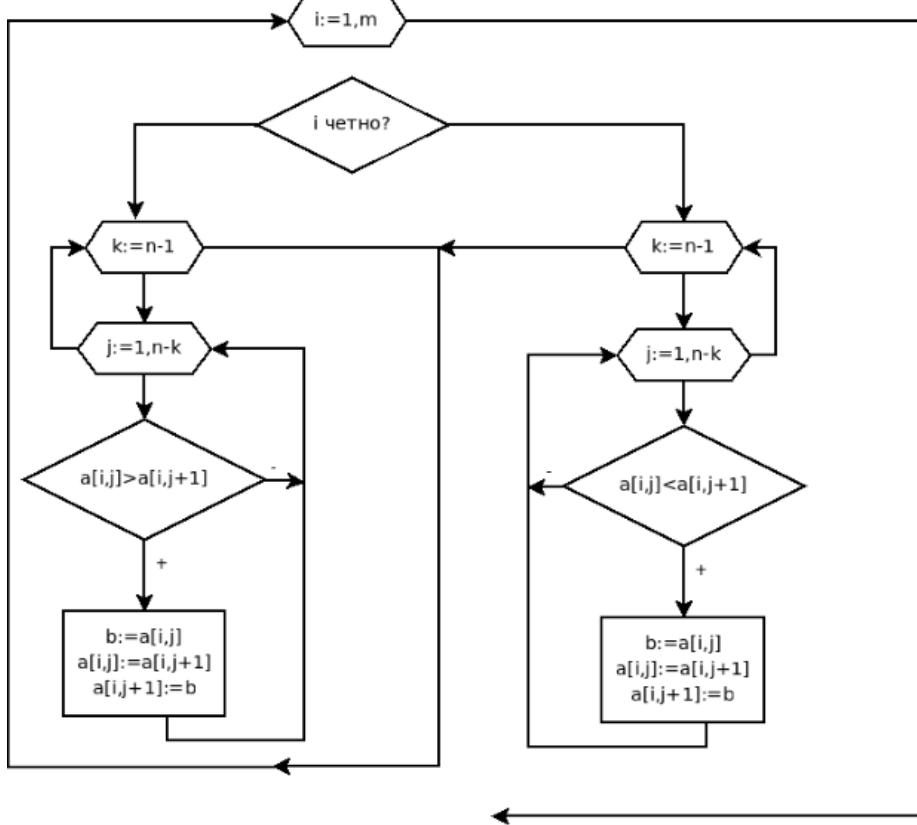
Рис. 6.24. Результаты решения задачи 6.7

ЗАДАЧА 6.8. Преобразовать матрицу

A(m, n)

так, чтобы строки с нечётными индексами были упорядочены по убыванию, с чётными — по возрастанию.

Каждая строка матрицы является одномерным массивом. Поэтому для упорядочения строки или столбца можно использовать обычные алгоритмы сортировки массивов. При решении задачи необходимо последовательно просматривать все строки матрицы, если номер строки нечётный, то сортируем строку методом пузырька по убыванию, иначе — по возрастанию. Блок-схема этого алгоритма представлена на [рис. 6.25](#).



[увеличить изображение](#)

Рис. 6.25. Блок-схема алгоритма решения задачи 6.8

Ниже представлено консольное предложение для решения этой задачи с комментариями.

```

var a : array [.. 15, .. 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.}
                {Упорядочение 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.}
                    {Упорядочение 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 ];
                                end
                            end
                        end
                    end
                end
            end
        end
    end

```

```

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.

```

На [рис. 6.26](#) приведены результаты работы программы.

```

~\~ Терминал
Файл Правка Вид Терминал Вкладки Справка
введите m и n
5 6
Матрица A
1 2 3 6 5 4
2 3 4 7 6 5
3 4 5 8 7 6
4 5 6 1 2 3
9 8 7 4 5 6
преобразованная матрица A
6.000 5.000 4.000 3.000 2.000 1.000
2.000 3.000 4.000 5.000 6.000 7.000
8.000 7.000 6.000 5.000 4.000 3.000
1.000 2.000 3.000 4.000 5.000 6.000
9.000 8.000 7.000 6.000 5.000 4.000

-----
(program exited with code: 0)
Press return to continue

```

Рис. 6.26. Результаты решения задачи 6.8

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. Алгоритм проверки, является ли число простым, уже неоднократно рассматривался в книге. Функция этой проверки подробно рассматривалась в главе 5 при решении задачи 5.7. Поэтому здесь просто приведем её текст.

```

function prostoe( N: integer ) : boolean;
vari : 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;

```

В этой же главе 5 мы рассматривали алгоритм (рис. 5.42) 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 приведён 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;
vars1, i, q, ost : word;
begin
{В переменной s1 мы будем собирать число в четверичной системе}
{счисления.}
s1 := 0;

```

```

{В переменной q будем последовательно хранить степени десяти; вначале}
{туда записываем 1 - десять в 0 степени, а затем в цикле будем}
{последовательно 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 счисления).)}
    ost := N mod 4;
{Очередной разряд числа умножаем на 10 в степени i и добавляем к}
{формируемому числу s1.}
    s1 := s1+ost * q;
{Уменьшаем число N в 4 раза путем целочисленного деления на 4.}
    N1:=N1 div 4;
{Формируем следующую степень десятки.}
    q:=q * 10;
  end;
//Возвращаем число в четверичной системе счисления.
  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 ($i := 1, 2, \dots, n$) выполнить следующее: обнулить сумму S ($S := 0$), организовать цикл по элементам строки ($j := 1, 2, \dots, m$), 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 ($P[i] := \text{perevod4}(S)$).

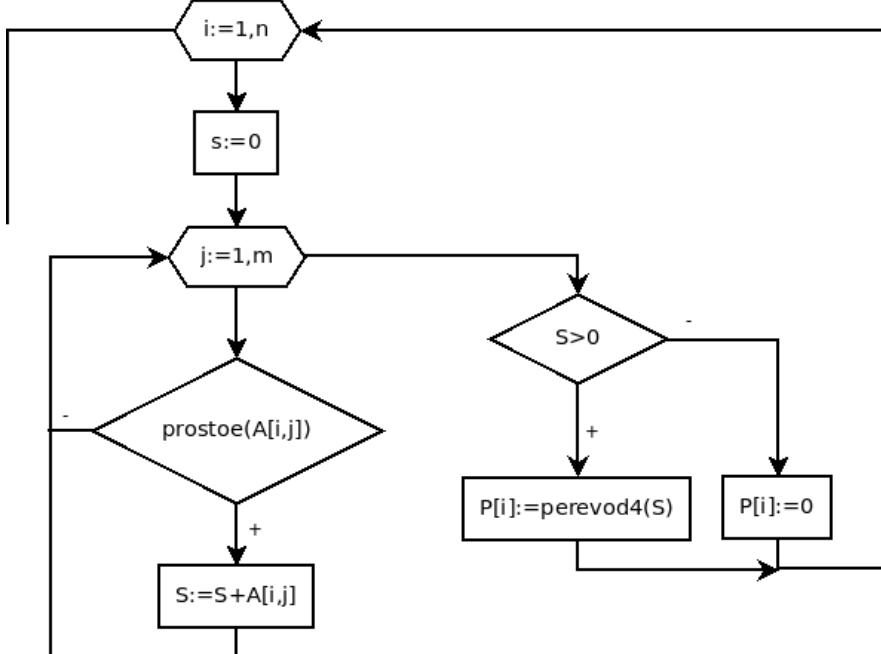


Рис. 6.27. Блок-схема алгоритма решения задачи 6.9

Блок-схема алгоритма приведена на [рис. 6.27](#).

Полный текст консольного приложения приведён ниже.

```

program pr_6_9;
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 четверичной системе}
{счисления.}
  s1 := 0;

```

```

{В переменной q будем последовательно хранить степени десяти, вначале}
{туда записываем 1 - десять в 0 степени, а затем в цикле будем}
{последовательно 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 счисления}.}
        ost :=N mod 4;
{Очередной разряд числа умножаем на 10 в степени i и добавляем к}
{формируемому числу s1.}
        s1 := s1+ost * q;
{Уменьшаем число N в 4 раза путем целочисленного деления на 4.}
        N:=N div 4;
{Формируем следующую степень десятки.}
        q:=q * 10;
    end;
//Возвращаем число в четверичной 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);
//Ввод матрицы.
    writeln ( 'Введите матрицу A ' );
    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 строки.}
    for i :=1 to n do
        begin
//Вначале сумма равна нулю.
        S := 0;
//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 записываем 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. 6.28.](#)

```

Терминал
Файл Правка Вид Терминал Вкладки Справка
Введите размеры матрицы
4 5
Введите матрицу A
2 4 6 8 13
6 8 10 12 14
13 11 6 10 11
34 56 76 80 5
Массив P
33 0 203 11

-----
(program exited with code: 0)
Press return to continue

```

Рис. 6.28. 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:

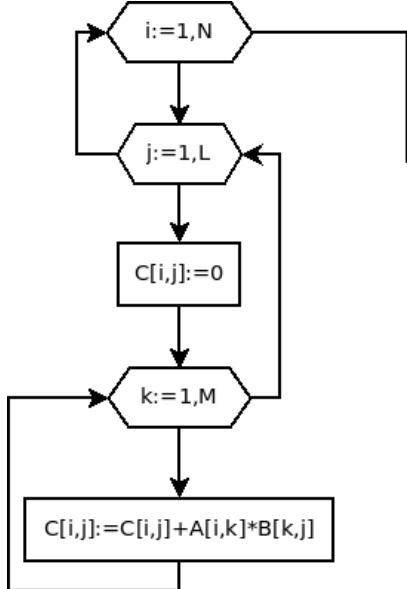


Рис. 6.29. Блок-схема умножения двух матриц

$$c_{i,j} = \sum_{k=1}^M a_{ik} b_{kj}, \quad (6.1)$$

где

$i = 1, \dots, N$

и

$j = 1, \dots, L$

Рассмотрим более подробно формирование матрицы $C(3,2)$ как произведения 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 умножение каждого элемента матрицы C по формуле (6.1), приведена на рис. 6.29.

Ниже приведён текст программы умножения двух матриц с комментариями.

```

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 );
  //Ввод матрицы A.
  writeln ( 'Матрица A' );
  for i := 1 to N do
    for j := 1 to M do
      read ( a [ i, j ] );
  //Ввод матрицы B.
  writeln ( 'Матрица B' );
  for i := 1 to M do
    for j := 1 to L do
      read ( b [ i, j ] );
  //Формирование матрицы C.

```

```

for i :=1 to N do
    for j :=1 to L do
        begin
{В C[i,j] будет храниться результат скалярного}
{умножения 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;
//Вывод матрицы C=AB.
writeln ( 'матрица C=A*B ' );
for i :=1 to N do
begin
    for j :=1 to L do
        write ( c [ i, j ] : 7 : 3, ' ' );
    writeln;
end;
end.

```

Результат работы представлен на [рис. 6.30](#).

```

Терминал
Файл Правка Вид Терминал Вкладки Справка
введите n, m и l
3 4 5
Матрица А
1 2 3 4
3 4 5 6
7 8 0 9
Матрица В
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
41 42 43 44 45
матрица C=A*B
310.000 320.000 330.000 340.000 350.000
518.000 536.000 554.000 572.000 590.000
614.000 638.000 662.000 686.000 710.000

-----
(program exited with code: 0)
Press return to continue

```

Рис. 6.30. Результаты работы программы умножения двух матриц

ЗАДАЧА 6.11. В матрице натуральных чисел

A(N, M)

найти строки, в которых находится максимальное из простых чисел. Элементы в них упорядочить по возрастанию. Если в матрице нет простых чисел, то оставить её без изменений.

Перед решением задачи отметим её некоторые особенности. В матрице может не быть простых чисел, максимальных значений может быть несколько, и при этом некоторые из них могут 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. 6.31](#). Удаление повторяющихся элементов происходит следующим образом. Просматриваются все элементы, начиная с первого, *i*-й элемент сравнивается со всеми последующими. Если, то встретился повторяющийся элемент, и мы удаляем из массива элемент с номером *j*. Алгоритм удаления был подробно рассмотрен в главе 5.

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. 6.32](#).

4. Процедура *Vozr* упорядочения массива *x* по возрастанию. Алгоритмы упорядочения рассматривались в главе 5. 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);

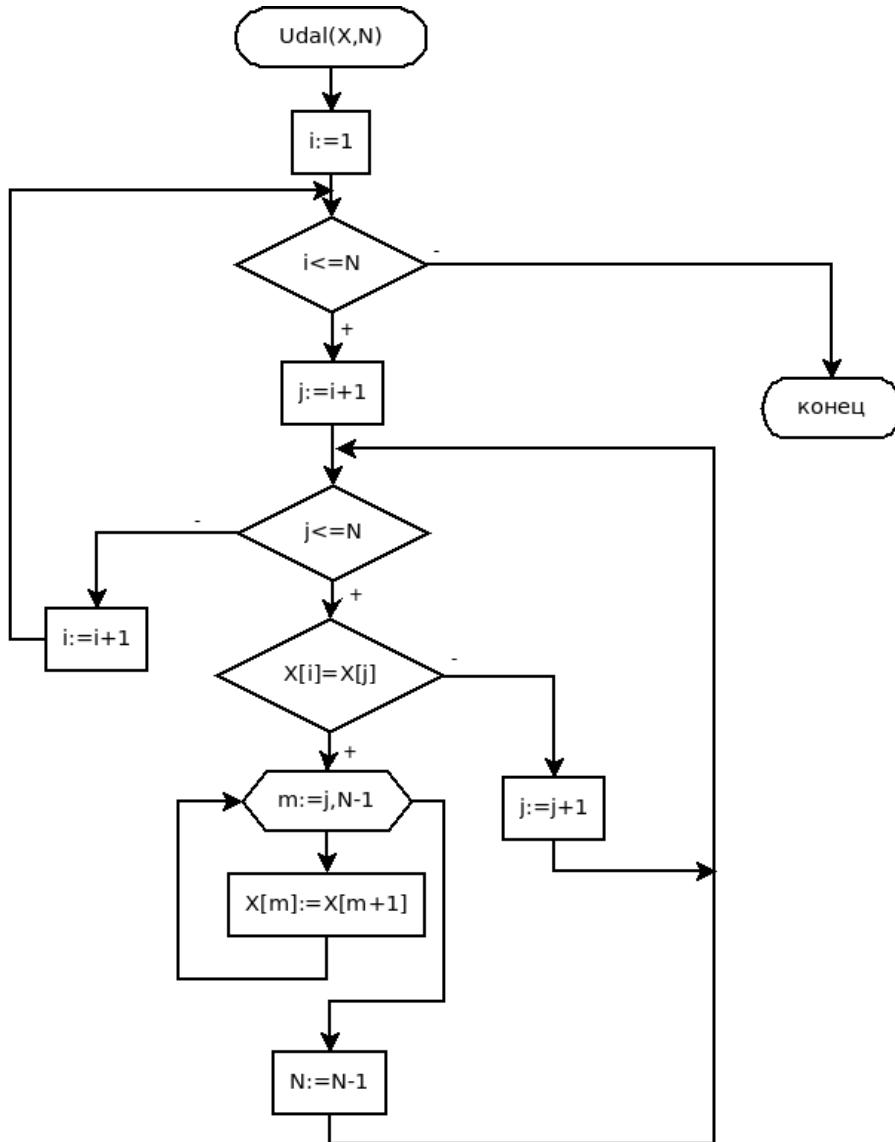


Рис. 6.31. Блок-схема процедуры `Udal`

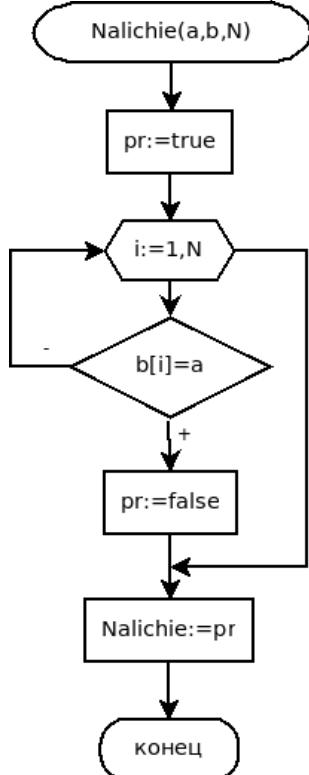
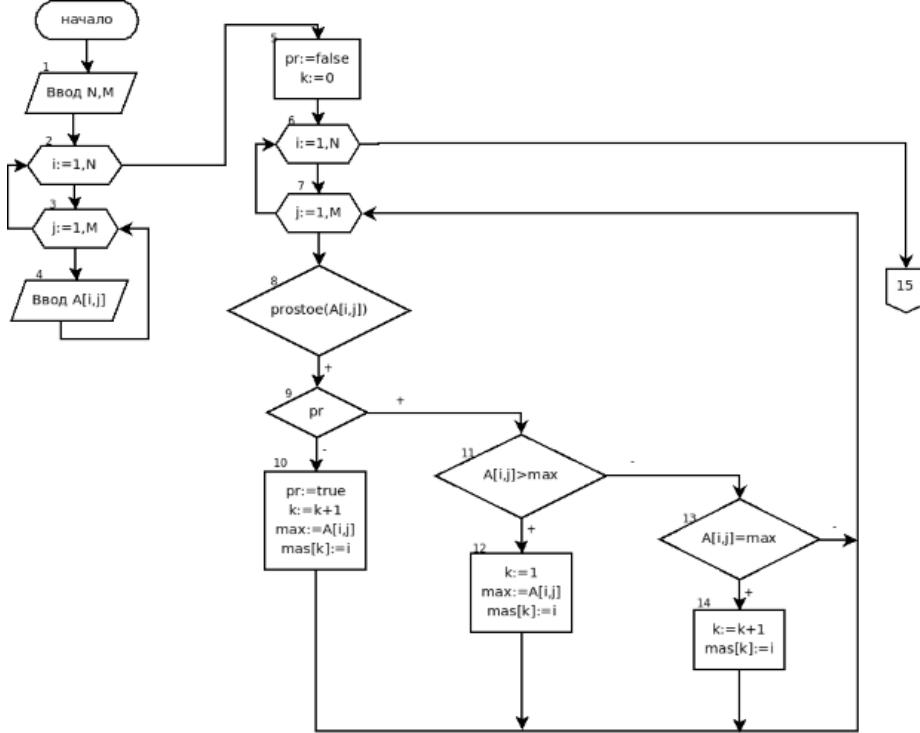


Рис. 6.32. Блок-схема функции `Nalichie`

Рассмотрим более подробно алгоритм решения задачи 6.11, который приведён на [рис. 6.33–6.34](#).

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). 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, записываем в массив `mas` под номером `ka`. В связи с тем, что в матрице есть простые числа, в переменную `Pr` записываем `true` (блок 10). Если это не первое простое число, сравниваем $A[i,j]$ с переменной `max`. Если $A[i,j] > max$ (блок 11), то в переменную `max` запишем $A[i,j]$, в переменную `k` запишем 1 (есть один максимум), в `mas[k]` записываем `i` — номер строки, где находится максимальный элемент (блок 12). Если $A[i,j] = max$ (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 раза²¹. 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 упорядочения 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).



[увеличить изображение](#)

Рис. 6.33. Блок-схема решения задачи 6.11 (начало)

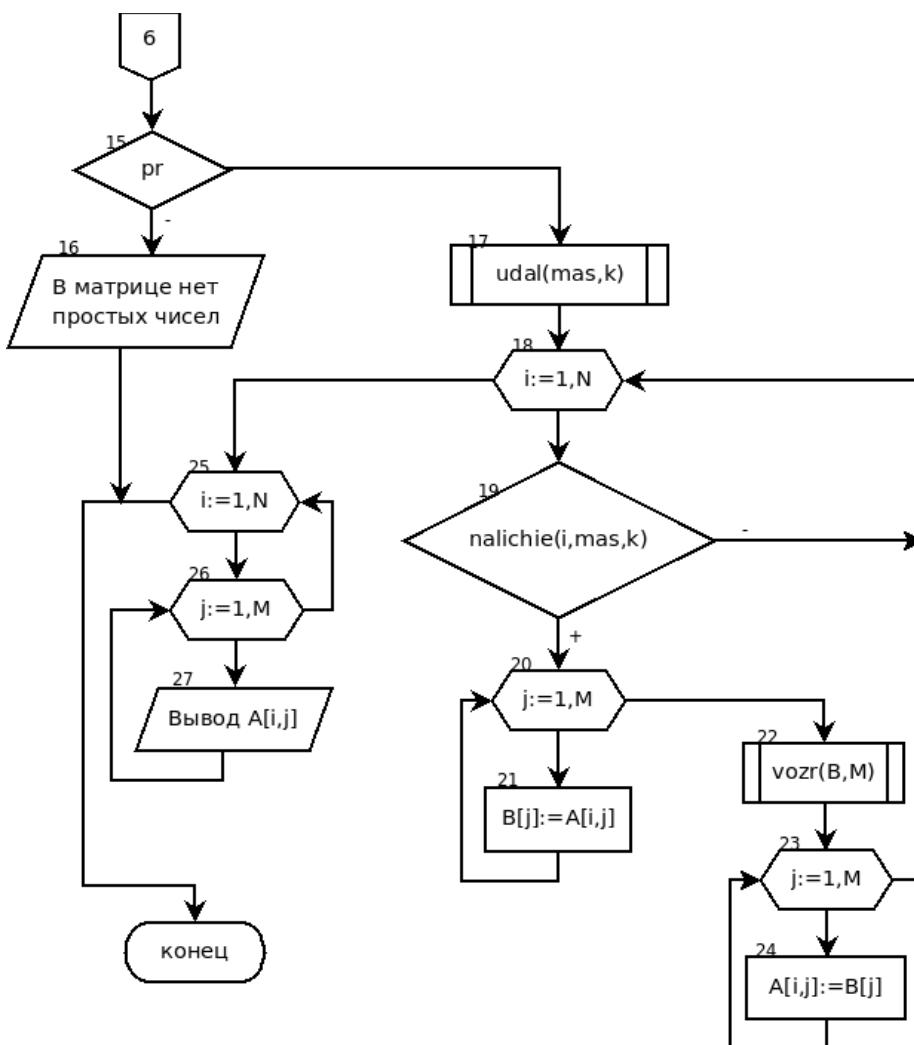


Рис. 6.34. Блок-схема решения задачи 6.11 (продолжение)

Ниже приведён листинг всей программы с подробными комментариями.

```

{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 являются параметрами-переменными, так как эти}
  {значения 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 повторяющийся элемент - }
        if x [ i ]=x [ j ] then
        begin
{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 следующему элементу.}
            else j := j +1;
            i := i +1;
        end;
    end;
{Функция nalichie возвращает true, если число a встречается в массиве}
{b, false - в противном случае.}
function nalichie ( a : word; b : massiv; n : word ) : boolean;
var
    pr : boolean;
    i : word;
begin
{Предполагаем, что в массиве b не встречается значение a - pr=false}
    pr := false;
{Перебираем все элементы массива.}
    for i :=1 to N do
{Если очередной элемент массива b равен значению 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 возрастанию.}
procedure vozr ( var x : massiv; n : word );
{X является параметром-переменной, именно массив 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 чисел.}
    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, является 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
{записываем 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 максимальное.}
    max:=A[ i, j ];
{В 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 сравниваем 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. встретился 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 хранится 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 хранится 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 простых 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;
    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. 6.35](#).

```

Терминал
Файл Правка Вид Терминал Вкладки Справка
N=5
M=6
Matrica A
13 2 7 6 1 9
3 4 6 2 1 5
13 3 13 6 7 1
4 6 8 2 3 9
4 9 7 13 1 2
Преобразованная матрица A
1 2 6 7 9 13
3 4 6 2 1 5
1 3 6 7 13 13
4 6 8 2 3 9
1 2 4 7 9 13

-----
(program exited with code: 0)
Press return to continue

```

Рис. 6.35. 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 и 6.6.

В завершении этой главы рассмотрим "динамические матрицы".

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 её 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.

1. Для обращения к элементу динамической матрицы, расположенному в `i`-й строке и `j`-м столбце, следует использовать конструкцию языка Турбо Паскаль `X[i]^j`.
2. 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 значениями.

Алгоритмы упорядочения рассматривались в главе 5, 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 указателей, каждый 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 чисел 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, минимальный элементы и их номера}
{и элементы, расположенные между ними, упорядочиваем "методом пузырька".}
for i :=1 to N do
begin
{Поиск минимального, максимального элемента 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 элемент расположен позже максимального, nmin и nmax}
{меняю местами.}
if nmax<nmin then
begin
j :=nmax;
nmax:=nmin;
nmin:= j;
end;
{В i-й строке упорядочиваем элементы, расположенные между nmin и nmax}
{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. 6.36](#).
Динамическая матрица может быть достаточно большой, фактически её размер ограничен только объемом свободной памяти.
В заключении главы приведём задачи для самостоятельного решения.

```

Терминал
Файл Правка Вид Терминал Вкладки Справка
N=3
M=9
Matrica B
1 8 7 6 5 4 3 2 9
11 18 17 16 15 2 3 4 5
9 8 7 6 5 4 3 2 1
Упорядоченная матрица B
 1.00   2.00   3.00   4.00   5.00   6.00   7.00   8.00   9.00
11.00  18.00  15.00  16.00  17.00  2.00   3.00   4.00   5.00
 9.00   2.00   3.00   4.00   5.00   6.00   7.00   8.00   1.00

-----
(program exited with code: 0)
Press return to continue

```

Рис. 6.36. Результаты решения задачи 6.12

Лекция 7. Обработка файлов средствами Free Pascal

В данной главе будут рассмотрены возможности языка Free Pascal для работы с файлами. По ходу изложения материала читатель также познакомится с компонентами Lazarus, предназначенными для выбора файла.

Предварительно давайте познакомимся с типами файлов.

7.1 Типы файлов в Free Pascal

Ввод данных с клавиатуры удобен при обработке небольших объёмов 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 и 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 их просмотра нужно писать специальную программу.

В бестиповых файлах информация считывается и записывается блоками определённого 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 файлов можно описать файловую переменную следующим образом:
```

```
var f : file of тип;
```

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 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 начнём 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 процедура `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/ evgeniy/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)` создаёт 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 после выполнения процедуры `rewrite` вся информация в нём уничтожается.

7.2.3 Процедура `CloseFile`

Процедура `CloseFile(f)`, где `f` — имя файловой переменной, закрывает файл, который ранее был открыт процедурами `rewrite` или `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 записывают 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 произойдёт 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 закрытии файла.

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 проверить, 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 считаются 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 её 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`) ввести очередное вещественное число `a`, которое сразу же записать в файл с помощью процедуры `writeln`.
4. Закрыть файл с помощью процедуры `closefile`.

Текст консольного приложения Lazarus, предназначенного для решения данной задачи, приведён 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/ evgeniy/pascal /6/ pr1 / abc. dat ' );
//Открываем пустой файл для записи.
  rewrite ( f );
//Определяем количество элементов в файле.
  writeln ( ' n= ' ); readln ( n );
//В цикле вводим очередной элемент и записываем его в файл.
  for i :=1 to n do
    begin
      write ( ' a= ' ); readln ( a );
      writeln ( f, a );
    end;
//Закрываем файл.
//При записи данных в файл это делать обязательно!!!
  CloseFile ( f )
end.
```

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 обработки. Для этого понадобится специальный компонент для выбора файла. Решение задачи 7.2 начнём со знакомства с компонентом `OpenDialog`. Этот компонент предназначен для создания стандартного диалогового окна выбора файла и расположен первым на странице `Dialogs` (см. рис. 7.1).

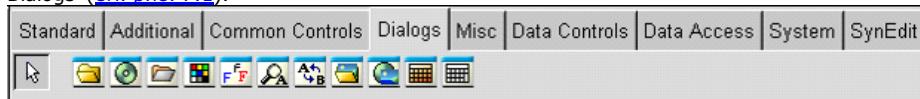


Рис. 7.1. Компоненты страницы `Dialogs`

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 щёлкнуть по кнопке (свойства `Filter`), после чего появится окно редактора фильтров (см. рис. 7.2).

Окно редактора фильтров состоит из двух столбцов: `Filter name` — имя фильтра (например, все файлы — программы 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, разделённых символом `|`. Специальная строка, соответствующая фильтру, представленному на рис. 7.2, имеет вид:

`OpenDialog1. Filter :=`

`'Программы_на_C|*.c|Программы_на_Паскале|*.pas|Все_файлы|*.*';`

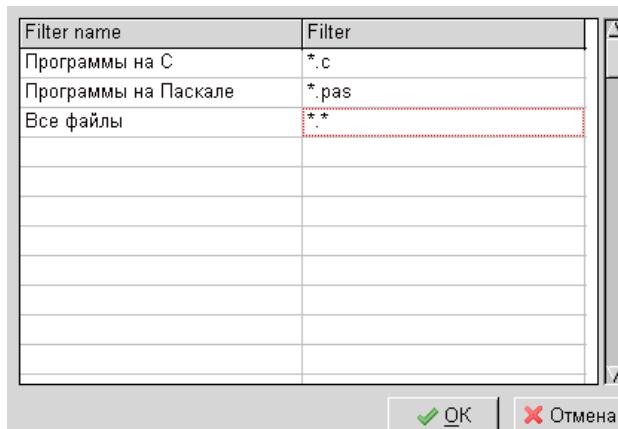


Рис. 7.2. Редактор фильтров

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 какимлибо 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 решения задачи.

1. Выбор файла.
2. Чтение данных из файла в массив вещественных чисел.
3. Удаление в массиве вещественных чисел всех чисел, меньших среднего арифметического, расположенных между максимальным и минимальным элементами.
4. Запись преобразованного массива в файл.

Разработку программы начнём 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 будет храниться преобразованный файл.
4. **OpenDialog1** — компонент для выбора имени обрабатываемого файла.
5. **Button1** — кнопка для запуска программы.
6. **Button2** — кнопка для завершения программы.

Установим следующие свойства формы и компонентов ([см. табл. 7.1](#), [см. табл. 7.2](#), [см. табл. 7.3](#), [см. табл. 7.4](#), [см. табл. 7.5](#), [см. табл. 7.6](#), [см. табл. 7.7](#), [см. табл. 7.8](#)).

Свойство Caption **Name**

Значение Преобразование файла Form_File

Таблица 7.1. Свойства формы

Свойство Caption Visible

Значение Файл до преобразования False

Таблица 7.2. Свойства метки label1

Свойство Caption Visible

Значение Файл до преобразования False

Таблица 7.3. Свойства метки label2

Свойство Lines Visible

Значение " False

Таблица 7.4. Свойства компонента Memo1

Свойство Lines Visible

Значение " False

Таблица 7.5. Свойства компонента Memo2

Свойство InitDialog DefaultExt

Значение /home/evgeniy/pascal/6/pr2/ dat

Таблица 7.6. Свойства компонента OpenDialog1

Свойство Caption Name Visible

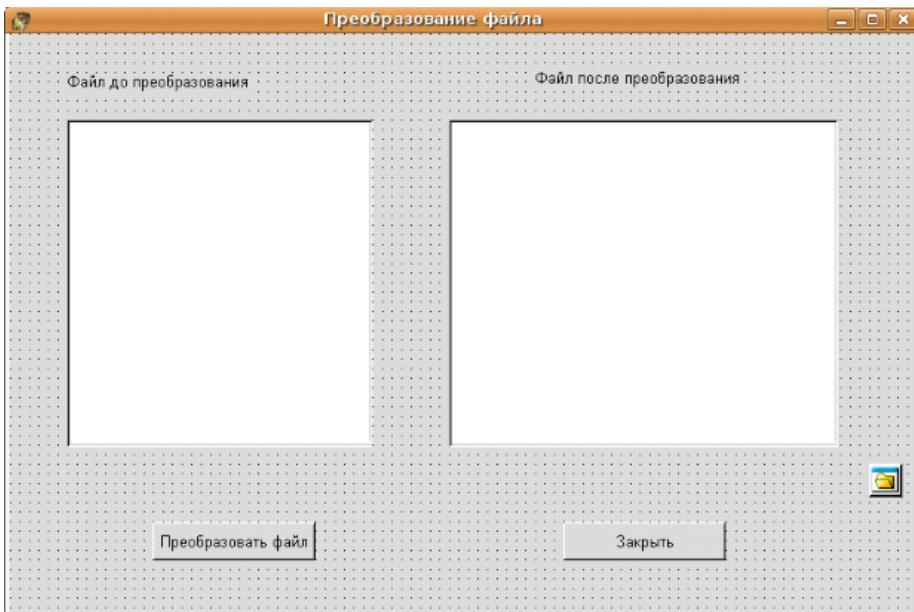
Значение Преобразовать файл Button_File True

Таблица 7.7. Свойства кнопки Button1

Свойство Caption Name Visible

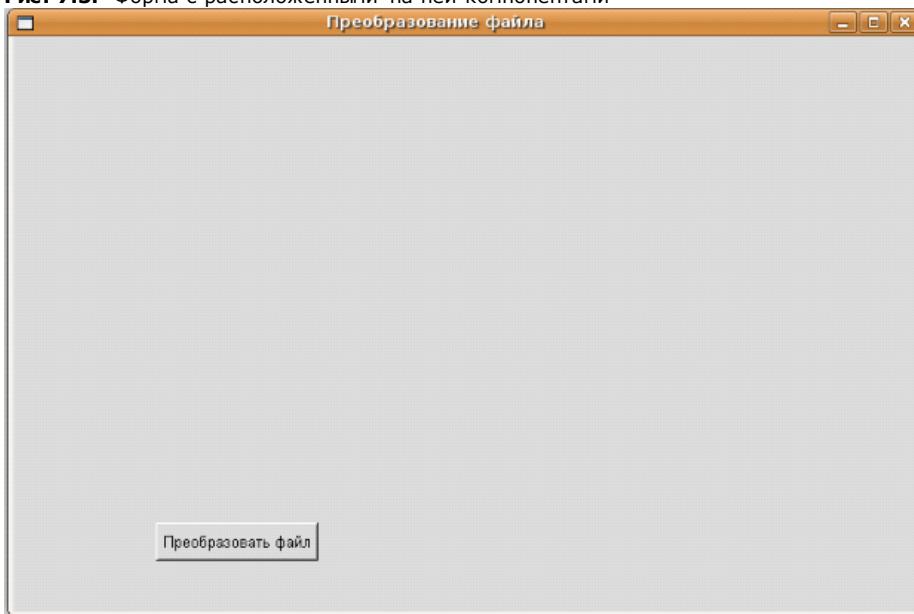
Значение Закрыть Button_Close False

Таблица 7.8. Свойства кнопки Button2



[увеличить изображение](#)

Рис. 7.3. Форма с расположенными на ней компонентами



[увеличить изображение](#)

Рис. 7.4. Приложение при запуске

Расположим компоненты на форме подобно тому, как показано 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 показано на [рис. 7.4](#).

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:

- Выбор файла из диалогового окна.
- Считывание данных из файла в массив вещественных чисел.
- Установка свойства **Visible** в **true** у компонентов **label1** и **Memo1**.
- Вывод содержимого массива в **Memo1**.
- Преобразование массива.
- Запись преобразованного массива в файл.
- Установка свойства **Visible** в **true** у компонентов **label2** и **Memo2**.
- Вывод преобразованного массива в **Memo2**.
- Установка свойства **Visible** в **True** у компонента **Button_Close**.

Ниже приведён текст обработки события **Button_Click** с подробными комментариями.

```

procedure TForm_File. Button_Click ( 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 переменную 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;
//Выполняем 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;
//Открываем файл для записи.
rewrite ( f );
//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.

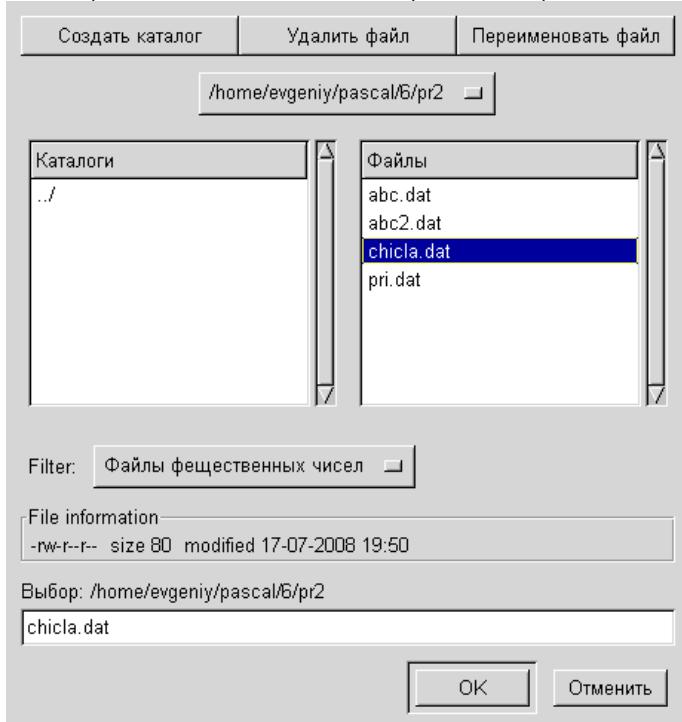
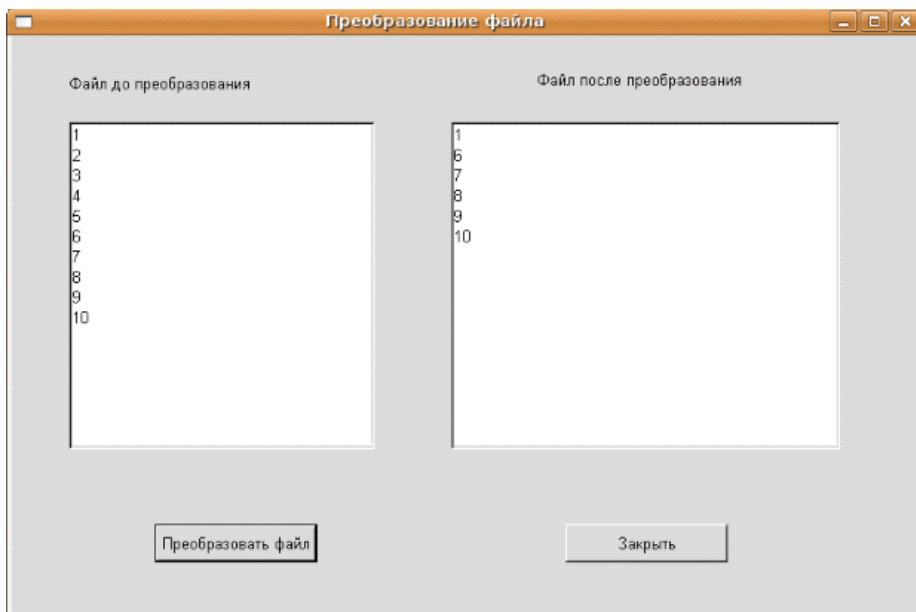


Рис. 7.5. Диалоговое окно выбора файла
procedure TForm_File. Button_CloseClick (Sender : TObject);
begin

 Close;

end;

Проверим функционирование созданного приложения. После запуска программы появляется окно, подобное представленному на [рисунке 7.4](#). Щелчком по кнопке Преобразовать файл открывается диалоговое окно выбора файла ([рисунок 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, разработано функционирующее приложение.
В связи с тем, что файл — последовательная структура данных, при решении задачи пришлось целиком считывать содержимое файла 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. Использование динамического массива не решит эту проблему, потому что неизвестно количество элементов в массиве.



[увеличить изображение](#)

Рис. 7.6. Окно с результатами работы программы

Для решения этой проблемы в языке Free Pascal реализован прямой доступ к файлу с помощью рассмотренных далее подпрограмм работы с файлами.

7.2.8 Функция filesize

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.

Рассмотрим использование функции `filesize` при переписывании чисел из файла вещественных чисел 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 с реальным файлом на диске.
assign file ( f, '/home/ evgeniy/pascal /6/ pr1 / abc. dat ' );
//Открываем файл для чтения.
reset ( f );
//Записываем в переменную n количество элементов в файле f.
n:= file size ( 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.
close file ( f );
readln;
end.
```

7.2.9 Функция filepos

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)`. Достигние конца файла можно проверить с помощью условия:

```
if filepos ( f )= file size ( f ) then  
writeln ( 'достигнут конца файла ' );
```

7.2.10 Процедура seek

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.

7.2.11 Процедура truncate

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 примере решения следующих двух несложных задач по обработке файлов.

ЗАДАЧА 7.4. В файле `/home/evgeniy/pascal/6/pr1/abc.dat` вещественных чисел 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.

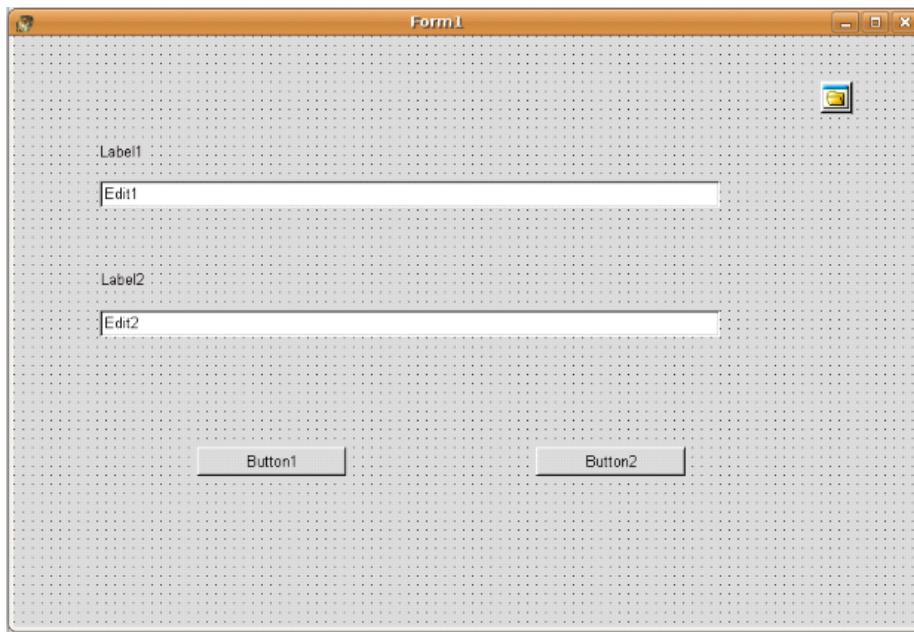
```
program Project1;  
{$mode objfpc}{$H+}  
uses Classes, SysUtils  
{ you can add units after this };  
var f : file of real;  
i, nmax, nmin : integer;  
a : array [ 0.. 200 ] of real;  
max, min : real;  
begin  
assignfile ( f, '/home/ evgeniy/pascal /6/ pr1 / abc. dat' );  
reset ( f );  
//Считываем компоненты файла в массив a.  
for i := 0 to file size ( 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 b  
//минимального элементов массива и их индексов.  
for i :=1 to file size ( 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 будет храниться файл после преобразования.
4. `OpenDialog1` — компонент для выбора имени обрабатываемого файла.
5. `Button1` — кнопка для запуска программы.
6. `Button2` — кнопка для завершения программы.



[увеличить изображение](#)

Рис. 7.7. Окно проекта с компонентами

Расположим компоненты на форме примерно так, показано на [рис. 7.7](#).

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 элемента 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/ evgeniy/pascal /6/ pr1';
  Button1. Caption := 'Преобразовать файл';
  Button2. Caption := 'Выход';
  Button2. Visible := False;
end;
```



[увеличить изображение](#)

Рис. 7.8. Окно программы решения задачи 7.4 при запуске

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 программы будут осуществляться при щелчке по кнопке **Преобразовать файл**, текст соответствующей подпрограммы с комментариями приведён 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 );
//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 file size ( 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
//редактирования.
    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 file size ( f ) -1 do
begin
  read ( f, a );
//Добавление очередного значения из преобразованного файла
//к выходной строке s1.
  s1 := s1+FloatToStr ( a)+ ' ';
end;
//Вывод содержимого преобразованного файла во второе текстовое

```

```

//pole редактирования.
Edit2. Text := s1;
//Запрет изменения текстового поля.
Edit2. ReadOnly:= true;
//Делаем видимой вторую кнопку.
Button2. Visible :=True;
CloseFile ( f );
end;
end;

```

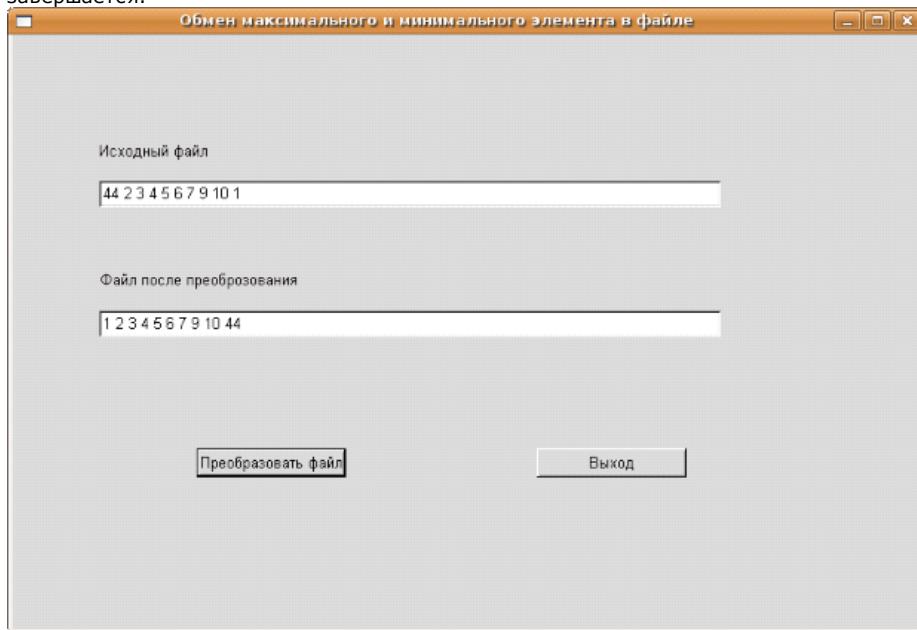
При щелчке по кнопке Выход программа должна завершать свою работу. Поэтому текст обработчика щелчка по кнопке будет очень простым.

```

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 нём 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 завершается.



[увеличить изображение](#)

Рис. 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.

ЗАДАЧА 7.5. Задан файл вещественных чисел `abc.dat`. Удалить из него максимальный и минимальный элементы.

Рассмотрим две программы, решающие эту задачу. Программы будем создавать как консольные приложения в Lazarus. 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/ evgeniy/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 их индексов.

```

```

max:=a [ 1 ]; min:=a [ 1 ];
nmax := 1; nmin := 1;
//Основной цикл для поиска максимального и
//минимального элементов массива и их индексов.
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`, сдвигаем на два порядка влево. Этим мы сотрём 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
assignfile ( f, '/home/ evgeniy/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 ) -3 do
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 file size ( f ) do
begin
  read ( f, a );
//Вывод записанного в файл элемента на экран.
  write ( a : 1 : 2, '_');
  end;
closefile ( f );
readln;
end.

```

В результате работы 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 широкое применение при обработке числовых данных получили бестиповые файлы.

Reset (var f : File; BuferSize : word);
Rewrite (var f : File; BuferSize : word);

Необязательный параметр **BuferSize** 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 при открытии файла (128). 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 **BuferSize**, 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**, 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 считаются данные из файла, **Count** — количество блоков размером **BuferSize**, считываемых из файла, необязательный параметр **ReadCount** определяет количество блоков размером **BuferSize**, реально считанных из файла.

BlockRead считывает из файла, связанного с файловой переменной **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 нескольких задач.

ЗАДАЧА 7.6. В бестиповом файле хранится массив вещественных чисел и количество элементов в нём. Удалить из файла максимальное вещественное число.

Напишем консольное приложение для 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/ evgeniy/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 бестиповой файл без цикла с помощью одного
//оператора BlockWrite(f,x^,N*sizeof(типа)) кажется авторам
//предпочтительнее.
//Закрываем файл.
CloseFile ( f );
//Освобождаем память.
freemem ( x,N*sizeof ( real ) );
readln;
end.

```

Разработку программы решения задачи 7.6 начнём 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 будет храниться файл после преобразования.
4. **OpenDialog1** — компонент для выбора имени обрабатываемого файла.
5. **Button1** — кнопка для преобразования файла.

Расположим компоненты на форме примерно так, показано на [рис. 7.10](#).

Основные свойства компонентов будем устанавливать программно при создании формы. Программа решения задачи 7.6 будет выполняться при щелчке по кнопке.

Полный текст модуля с необходимыми комментариями:

```

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 текстовом 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 ], fffFixed,5, 2)+ ' ';
//Закрываем 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.

```

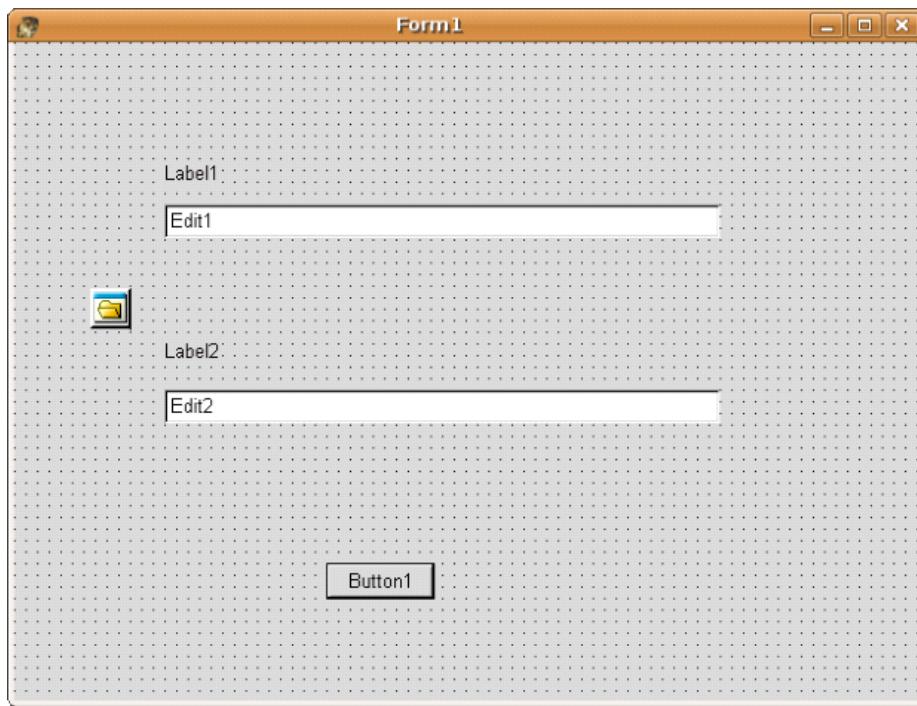


Рис. 7.10. Окно формы с компонентами

При запуске программы окно приложения будет таким, как 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](#).



Рис. 7.11. Окно программы решения задачи 7.6 при запуске

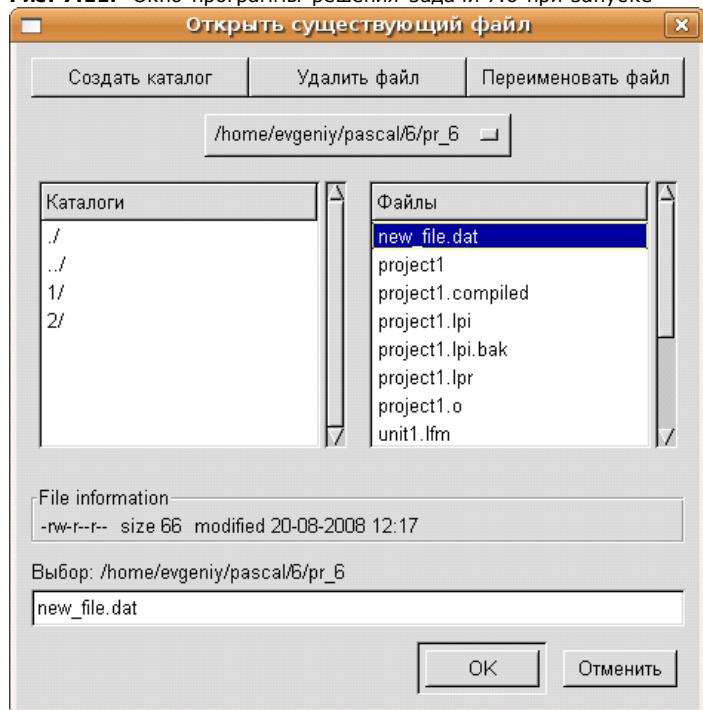


Рис. 7.12. Окно выбора файла

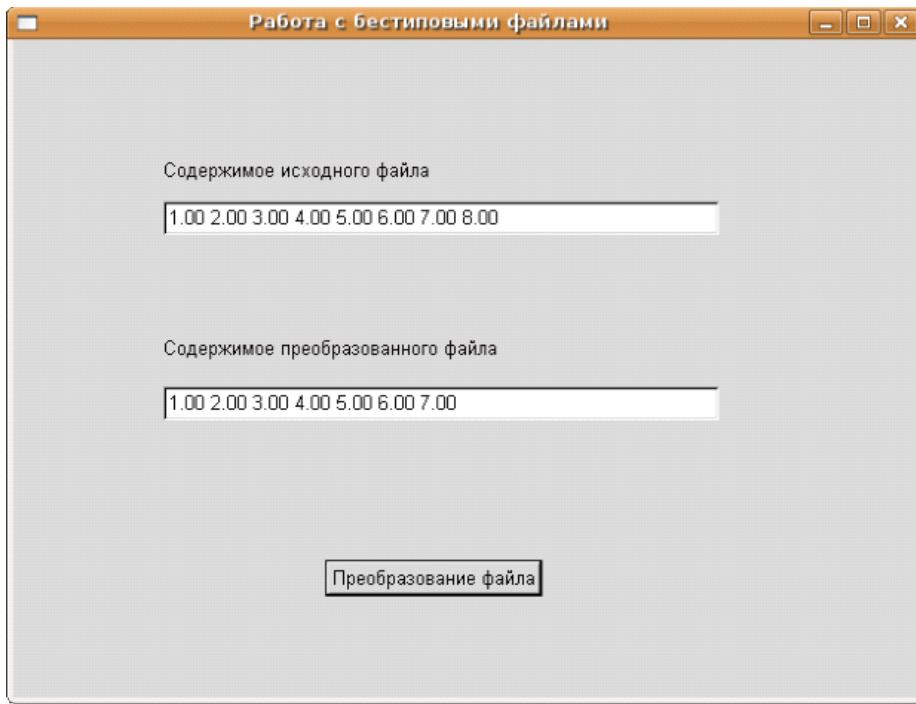


Рис. 7.13. Окно работающей программы

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 начнём с написания консольной программы создания файла. Для этого определимся со структурой файла. Пусть в нём хранятся два значения типа **word**:

N и **M**, затем матрица вещественных чисел $A(N, M)$, затем значения **P** и **L** типа **word** и матрица $B(P, L)$.

Ниже приведён листинг программы 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
  AssignFile ( 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.
  CloseFile ( f );
end.

Теперь напишем консольное приложение Lazarus, с помощью которого из бестипового
файла считаем матрицы A, B, их размеры, после чего вычислим матрицу C как
произведение A и B.

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.
  AssignFile ( 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.
CloseFile ( 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 (типованными) файлами.
2. Процедуры `seek`, `truncate` и функция `filepos` не работают с текстовыми файлами.
3. Можно пользоваться процедурой открытия текстового файла `append(f)`, где `f` — имя файловой переменной. Эта процедура служит для открытия файла в режиме дозаписи в конец файла. 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 после указанных переменных и констант в файл записывается символ "конец 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 Free Pascal позволяет генерировать код программы в двух режимах: с проверкой корректности ввода-вывода и без неё.

В программу может быть включен ключ режима компиляции. Кроме того, предусмотрен перевод контроля ошибок ввода-вывода из одного состояния в другое:

- `{$I+}` — режим проверки ошибок ввода-вывода включён;
- `{$I-}` — режим проверки ошибок ввода-вывода отключён.

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 включённом режиме проверки любая ошибка ввода-вывода будет фатальной, программа прервётся, 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 неопределён.

Для опроса кода ошибки лучше пользоваться специальной функцией `IOResult`, но необходимо помнить, что опросить её можно только один раз после каждой операции ввода или вывода: она обнуляет своё 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](#).

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 <code>AssignFile</code>
103	файл не открыт
104	файл не открыт для ввода
105	файл не открыт для вывода

106 неверный номер
150 диск защищён от записи

Таблица 7.9. Коды ошибок операций ввода-вывода

Рассмотрим несколько практических примеров обработки ошибок ввода-вывода:

- 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;
```

- 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:

- Когда вводятся числовые значения, два числа считаются разделёнными, если между ними есть хотя бы один пробел, или символ табуляции, или символ конца строки.
- При вводе строк начало текущей строки идёт сразу за последним, введённым 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 неё.

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**.

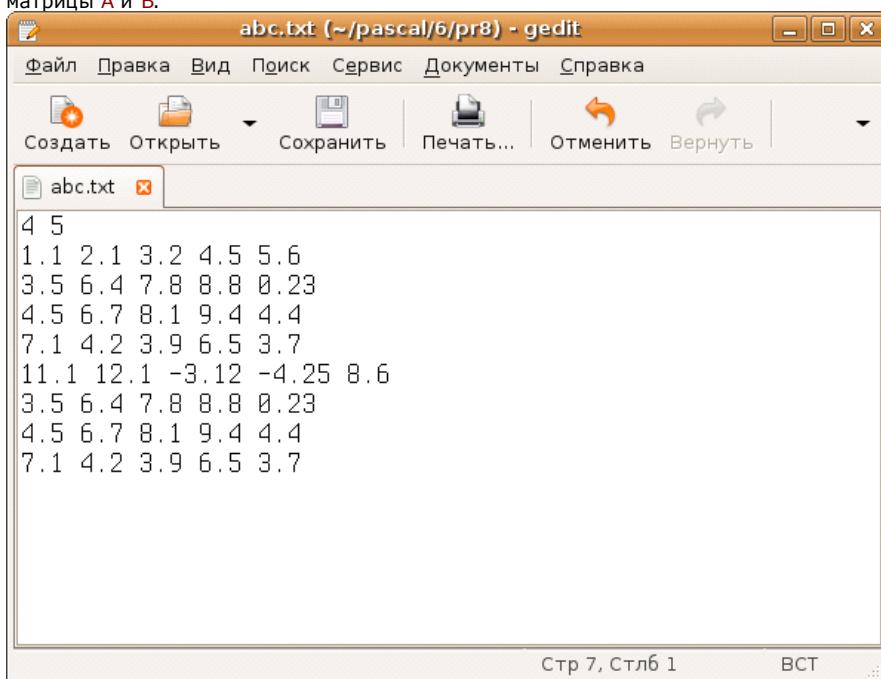


Рис. 7.14. Файл abc.txt

На [рис. 7.14](#) приведён пример файла `abc.txt`, в котором хранятся матрицы $A(4,5)$ и $B(4,5)$.
Текст консольного приложения решения задачи 7.8 с комментариями приведён ниже.

```
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. t x t' );
//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 в файл 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](#).

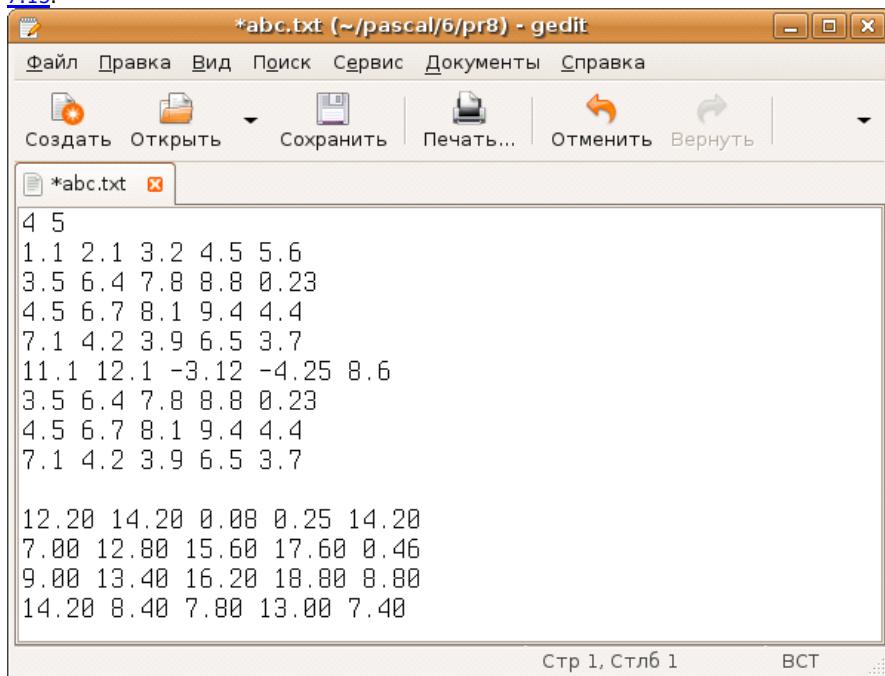


Рис. 7.15. Файл `abc.txt` после дозаписи матрицы С

Лекция 8. Работа со строками и записями

В этой главе мы познакомимся с двумя типами данных — строками и за-писями. Работа

основных функций обработки строк будет показана на примерах, после чего читатель может познакомиться с записями. Запись (record) — это сложная структура данных, состоящая из фиксированного количества объектов, называемых полями записи. В отличие от массивов, поля записи могут быть разного типа.

8.1 Обработка текста

Часто в задачах программирования приходится обрабатывать текст. Обработка текста в FreePascal 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 операций со строками является их объединение. Для этого их можно сложить подобно числам. Например:

```
var s : string;
begin
s := 'text' + ' ' + '*' * 1';
...
```

Результатом работы оператора будет строка `text_*1`.

Строка — это массив символов, поэтому из строки можно получить любой символ, указав его номер в квадратных скобках рядом с именем переменной.

Например:

```
var s : string; c : char;
begin
s := 'Privet ';
c := s [ 4 ];
...
```

Результатом работы будет символ `v`.

Теперь рассмотрим основные функции и процедуры обработки строк, приведённые в табл. 2.7, на примерах.

Ниже приведён листинг программы с комментариями. Результаты её работы можно увидеть на [рис. 8.1](#).

```
program Project1;
var
    Str1, Str2, Str3, Str4 : String;
    word : String;
    K, L : integer;
begin
    Str1 := 'Ivanov ';
    Str2 := 'Sergey ';
    Writeln (' Stroka Str1 - ', Str1 );
    Writeln (' Stroka Str2 - ', Str2 );
//объединение 1-й и 2-й строк
    Str3 := Str1+ '_'+Str2;
//определение длины строки
    L:= Length ( Str3 );
    Writeln (' Stroka Str1+Str2 = ', Str3 );
    Writeln (' Dlina stroki= ', L );
    Str4 := 'v ';
//поиск вхождения буквы v в строке 3
    K:=Pos ( Str4, Str3 );
    Writeln (' 1 raz bukva V na pozicii ', K );
//копирование в переменную Str4 6 символов, начиная с 8-го, из строки
//Str3
    Str4 :=copy ( Str3, 8, 6 );
    Writeln (' Str4= ', Str4 );
//поиск первого слова - копирование в переменную word символов
//из строки Str3, начиная с первого символа и до пробела
    word:=Copy( Str3, 1, Pos ( ' ', Str3 ) - 1);
    Writeln (' 1_e_slovo ', word );
//удаление из строки Str3 2-х символов, начиная с 5-й позиции
    delete ( Str3, 5, 2 );
    Writeln (' Str3 posle udaleniya simvolov ', Str3 );
    Readln;
end.
```

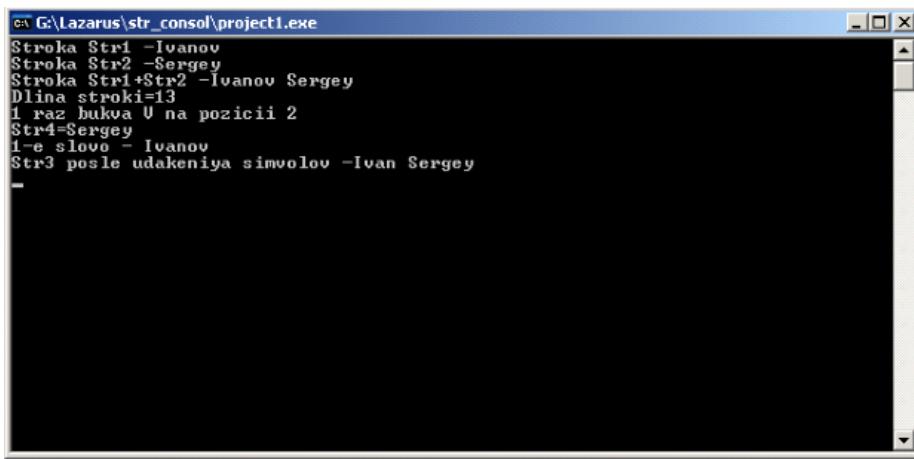


Рис. 8.1. Результаты работы программы обработки строк

В следующем примере мы прочитаем из поля Edit формы массив чисел и найдём их сумму. Для этого создадим новый проект, на форме разместим поле Edit1, компонент ListBox1 для вывода результатов и кнопку Button1, как на [рис. 8.2](#). Установим в качестве свойства Text компонента Edit1 следующую строку: 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0, в этом случае при запуске программы этот массив уже будет введён (при записи элементов массива между элементами можно вставлять любое количество пробелов, в дальнейшем в программе количество пробелов между элементами будет сокращено до одного).

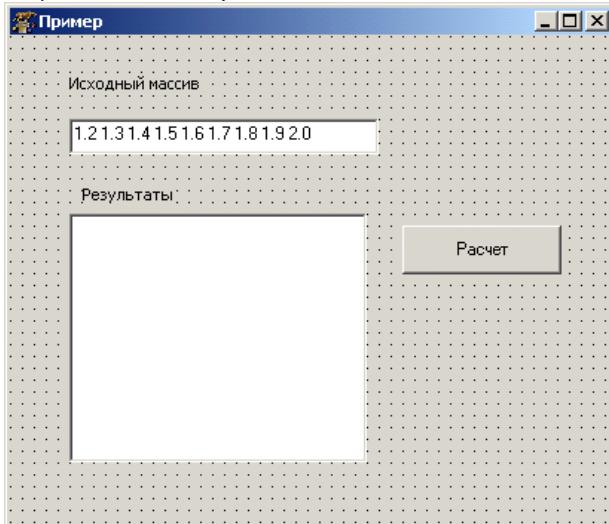


Рис. 8.2. Окно формы для считывания массива

Ниже приведён листинг программы с комментариями, результаты работы программы показаны на [рис. 8.3](#).

```

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 с диалогового окна из поля Edit1
Str1 := Edit1.Text;
//вывод исходной строки в диалоговое окно
Str2 := 'Исходная строка';
ListBox1.Items.Add(Str2);
ListBox1.Items.Add(Str1);
//строку Str1 разобьем на подстроки, в которых
//хранятся считанные строки-числа
//если первый символ - пробел, то удаляем его
if Str1[1] = ' ' then Delete(Str1, 1, 1);
//проход по строке и замена двойного пробела одинарным
//для этого организовывается следующий цикл:
//пока в строке Str1 есть вхождение подстроки, состоящей из
//двух пробелов, удаляется один символ с начальной позиции
//вхождения подстроки, т. е. удаляется первый пробел
while Pos(' ', Str1) > 0 do
  Delete(Str1, Pos(' ', Str1), 1);
//для корректной обработки текста сделаем последним
//символом пробел
if Str1[Length(Str1)] <> ' ' then
  Str1 := Str1 + ' ';
//вывод в диалоговое окно изменённой строки
//с одним пробелом между числами
Str2 := 'Строка после удаления пробелов';
ListBox1.Items.Add(Str2);
ListBox1.Items.Add(Str1);
i := 0; Sum := 0;
//организуем цикл, в котором выделяем подстроку с числом
//затем запоминаем число в массив,
//а подстроку из строки удаляем
repeat
  //выделение строки-числа (до пробела)
  word := Copy(Str1, 1, Pos(' ', Str1) - 1);
  //преобразование строки в вещественное число и запись его
  //в i-ый элемент массива
  x[i] := StrToFloat(word);
  sum := sum + x[i];
  inc(i);
//удаление из строки Str1 найденной подстроки
Delete(Str1, 1, Length(word) + 1);
//вывод в ListBox1 строки-числа
ListBox1.Items.Add(word);
//делаем до тех пор, пока длина строки не будет равна 0
until Length(Str1) = 0;
//вывод рассчитанной суммы элементов массива и количества
Str2 := 'Сумма = ' + FloatToStr(sum);
Str3 := 'Количество элементов = ' + IntToStr(i);
ListBox1.Items.Add(Str2);
ListBox1.Items.Add(Str3);
end;
initialization
{$I unit1.lrs}
end.

```

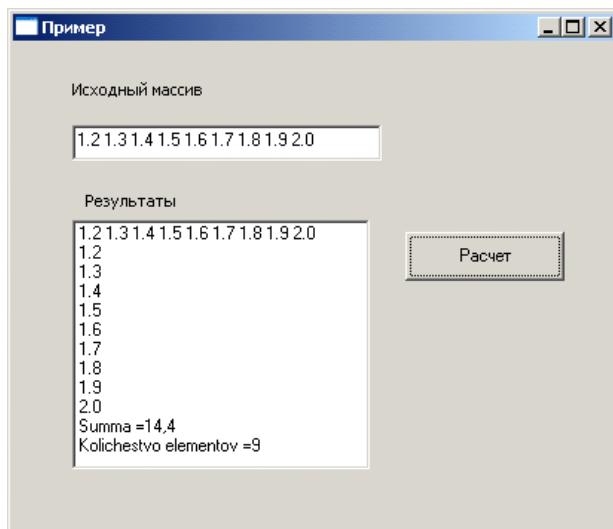


Рис. 8.3. Результаты работы программы считывания массива из поля Edit

8.2 Работа с записями

В большинстве случаев при написании программ применяются простые типы данных (числа, строки). Но часто 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, называемых полями

записи. Общий синтаксис объявления 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 компонентов записи можно получить доступ, используя составное имя.
Для этого вначале пишется имя переменной, затем точка, затем имя поля.
Рассмотрим следующий пример. Создадим запись — треугольник с тремя полями —
сторонами треугольника. В программе произведен расчёт площади треугольника по
формуле Герона.
program Project1;
type
Triangle= record
a, b, c : real; //стороны треугольника
end;
var x : Triangle; //объявление переменной типа записи Triangle
p, s : real;
begin
write ( ' a=' );
readln ( x. a ); //чтение поля a
write ( ' b=' );
readln ( x. b ); //чтение поля b
write ( ' c=' );
readln ( x. c ); //чтение поля c
p:=( x. a+x. b+x. c ) / 2; //вычисление полупериметра
//проверка существования треугольника - подкоренное выражение
//для формулы Герона должно быть положительным
if ( 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 ( ' Square= ', s : 7 : 2 );
end
else
writeln ( ' Not triangle ' );
readln;
end.
```

Элементами записей могут быть как простые, так и структурные типы. Никаких ограничений по уровню вложенности структур не существует.

Теперь для примера создадим запись Student, которая будет состоять из полей: фамилия, имя, группа, оценки по пяти дисциплинам и прописка. В свою очередь прописку тоже сделаем записью, состоящей из полей: город, улица, дом, квартира.

```
type
adress = record
city, street : string; //город, улица
house, apartment : integer; //дом, квартира
end;
student = record
surname, name : string; //фамилия, имя
group : string; //группа
estimation : array [ 1.. 5 ] of integer; //оценки
residence : adress; //прописка
end;
```

При объявлении такой записи обращение к полям осуществляется так:

```
var Ivanov : student;
x : array [ 1.. 100 ] of student;
begin
Ivanov. group := 'Ф08 ';
Ivanov. residence. city := 'Киев ';
x [ 1 ]. estimation [ 5 ] := 3; //у первого студента 5-я оценка =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 := 'Артёма ';
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 8.1. Создать базу данных, содержащую сведения о студентах. В программе предусмотреть расчёт среднего балла студента, сортировку по алфавиту, вывод результатов в диалоговое окно и в текстовый файл.

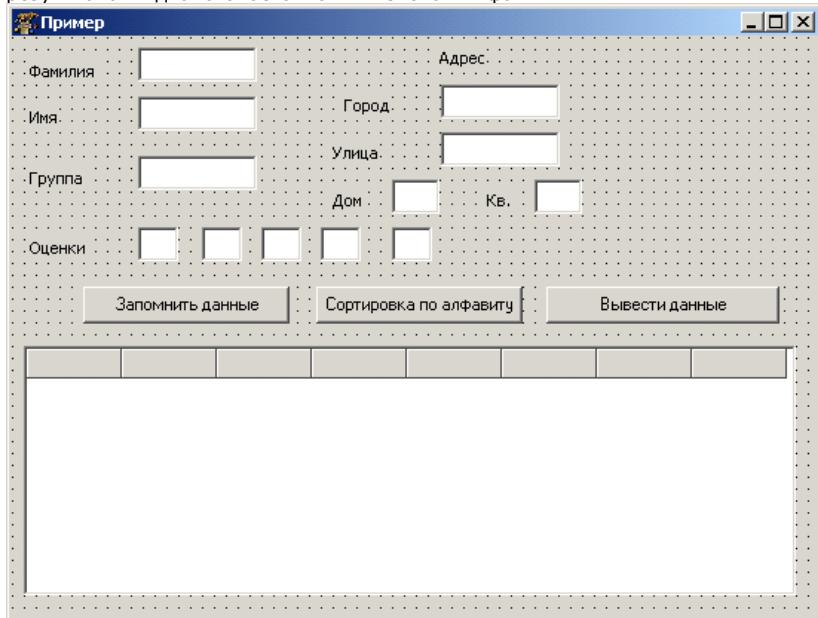


Рис. 8.4. Пример формы для задачи 8.1

Создадим новый проект. На форму (рис. 8.4) поместим необходимое количество объектов типа **TEdit** для ввода исходных данных и объект **StringGrid** для вывода результатов. Для объекта **StringGrid1** зададим свойства **ColCount=8** и **RowCount=1**.

Также создадим три кнопки:

- Запомнить данные — после ввода данных оничитываются, и поля ввода очищаются для дальнейшего ввода;
- Сортировка по алфавиту — после нажатия этой кнопки записи сортируются по алфавиту фамилий;
- Вывести данные — результаты выводятся в таблицу ниже и записываются в текстовый файл.

```

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;
    //Процедура-обработчик кнопки "Запомнить данные"
    procedure Button1Click ( Sender : TObject );
    //Процедура-обработчик кнопки "Вывести данные"
    procedure Button2Click ( Sender : TObject );
    //Процедура-обработчик кнопки "Сортировать по алфавиту"
  end;

```

```

procedure Button3Click ( Sender : TObject );
//Процедура инициализации формы
procedure FormCreate ( Sender : TObject );
private
{ private declarations }
public
{ public declarations }
end;
//объявление записи адреса студента
adress = record
city, street : string; //город, улица
house, apartment : integer; //дом, квартира
end;
//объявление записи сведений о студенте
student = record
surname, name : string; //фамилия, имя
group : string; //группа
estimation : array [ 1.. 5 ] of integer; //оценки
residence : adress; //прописка
s _ ball : real; //средний балл
end;
var
Form1 : TForm1;
//массив переменных типа "студент"
x : array [ 0.. 100 ] of student;
//переменная для подсчета количества студентов
i : integer;
implementation
{TForm1}
//процедура инициализации формы
procedure TForm1. FormCreate ( Sender : TObject );
begin
i := 0; //количество студентов вначале равно 0
end;
//процедура работы кнопки "Запомнить данные"
procedure TForm1. Button1Click ( Sender : TObject );
var sum, j : integer;
begin
//чтение данных с формы
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 );
//расчет 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 := ''; E
dit9. 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 будет на 1 больше,
//чем количество студентов (плюс 1 строка для шапки)
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;
f: textfile; //временная переменная для сортировки
begin
for j := 0 to i -1 do
  for k:= j +1 to i -1 do
    if x [ j ]. surname > x [ k ]. surname then
    begin
      temp:=x [ j ];
      x [ j ]:= x [ k ];
      x [ k ]:= temp;
    end;
end;
initialization
{$I unit1.lrs}
end.

```

При запуске программы и вводе сведений окно формы выглядит, как на [рис. 8.5](#). Когда запись введена, следует щёлкнуть по кнопке Запомнить данные, при этом поля ввода очищаются для ввода следующей записи. После щелчка по кнопке Вывести данные таблица заполняется введёнными сведениями ([рис. 8.6](#)).

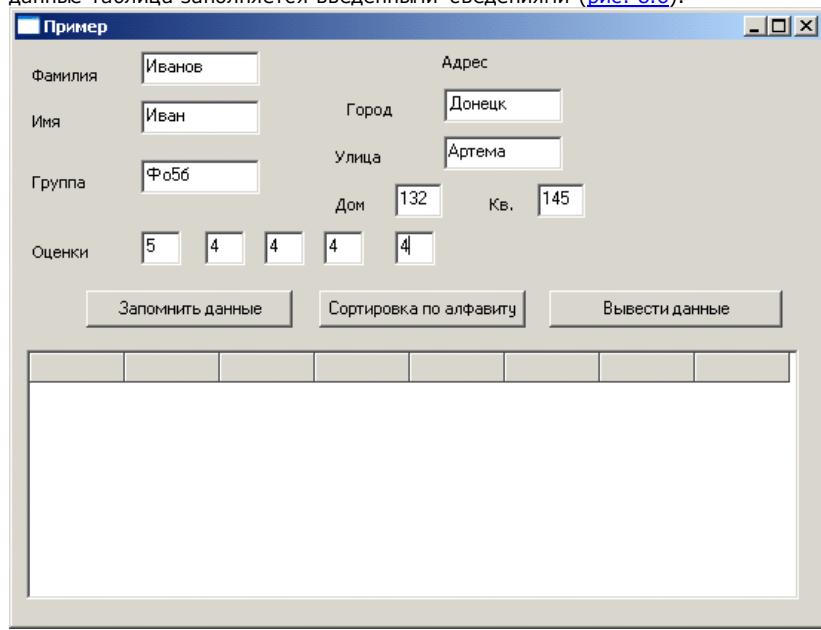


Рис. 8.5. Окно формы ввода сведений о студенте

Пример

Фамилия	<input type="text"/>	Адрес	
Имя	<input type="text"/>	Город	<input type="text"/>
Группа	<input type="text"/>	Улица	
Оценки	<input type="text"/> <input type="text"/> <input type="text"/> <input type="text"/> <input type="text"/>	Дом	<input type="text"/>
		Кв.	<input type="text"/>

Запомнить данные **Сортировка по алфавиту** **Вывести данные**

	Фамилия	Имя	Группа	Город	Улица	Дом/кв.	Средний
1	Иванов	Иван	Ф056	Донецк	Артема	132/145	4,2
2	Петрова	Ольга	Ф056	Донецк	Ткаченко	147/2	3,8
3	Громов	Илья	МП05	Горловка	Ленина	17/8	3,6
4	Смирнова	Ольга	МП06	Горловка	Петрова	1/12	4,2
5	Киреев	Андрей	МП05	Донецк	Свободы	21/67	5
6	Чернова	Светлана	УА05а	Донецк	Щетинина	1/123	4,4
7							

Рис. 8.6. Окно формы вывода сведений о студентах

Пример

Фамилия	<input type="text"/>	Адрес	
Имя	<input type="text"/>	Город	<input type="text"/>
Группа	<input type="text"/>	Улица	
Оценки	<input type="text"/> <input type="text"/> <input type="text"/> <input type="text"/> <input type="text"/>	Дом	<input type="text"/>
		Кв.	<input type="text"/>

Запомнить данные **Сортировка по алфавиту** **Вывести данные**

	Фамилия	Имя	Группа	Город	Улица	Дом/кв.	Средний
1	Антонова	Ирина	МП05	Донецк	Светлая	2/21	4,6
2	Борисова	Татьяна	Ф056	Макеевка	Транспортн	12/6	5
3	Громов	Илья	МП05	Горловка	Ленина	17/8	3,6
4	Иванов	Иван	Ф056	Донецк	Артема	132/145	4,2
5	Киреев	Андрей	МП05	Донецк	Свободы	21/67	5
6	Петрова	Ольга	Ф056	Донецк	Ткаченко	147/2	3,8
7							

Рис. 8.7. Окно формы после сортировки записей по алфавиту фамилии

После щелчка по кнопке Сортировка по алфавиту нужно повторно щелкнуть по кнопке Вывести данные, чтобы увидеть отсортированный список (рисунок 8.7).

Лекция 9. Объектно-ориентированное программирование

Эта глава посвящена изучению объектно-ориентированного программирования (ООП). ООП — это методика разработки программ, 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. Объект состоит из трёх 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 использовать одинаковые имена для методов, входящих в различные классы.

Для объявления класса используется конструкция:

```
type
  <название класса> = class (<имя класса родителя>)
    <поля и методы 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 предопределённого 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, которые пользователь может устанавливать на этапе проектирования и которые доступны для редактирования в инспекторе объектов.

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. Поля и методы, описанные до указания имени секции, являются публичными (**public**). 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 функции;

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 Free Pascal имеется большое количество классов, с помощью которых описывается форма приложения и её компоненты (кнопки, поля, флагги 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 **implementation** 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 задаётся 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_класса. free;

с помощью метода **free** вызывает деструктор и освобождает память, занимаемую полями объекта **имя_переменной_типа_класса**.

Рассмотрим всё описанное на примере класса "комплексное число"¹¹. Назовём этот класс **Tcomplex**. В нём будут члены класса: **x** — действительная часть комплексного числа — и **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 будет вычисляться его модуль и аргумент. В компонент **Memo1** выведем результаты: число в алгебраической форме, его аргумент и модуль. Ниже приведём текст модуля с комментариями, который демонстрирует работу с этим классом.

```

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 );
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
  function Modul () : real; //метод вычисления модуля
  function Argument () : real; //метод вычисления аргумента
  //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;
//Вызов родительского конструктора
  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:= StrToInt ( Edit1. Text );
  chislo. y:= StrToInt ( Edit2. Text );
  Str1 := 'Kompleksnoe chislo '+chislo. ComplexToStr ();
//вывод на форму в поле Memo1 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.

```

Результат работы программы приведён на [рис. 9.1](#).

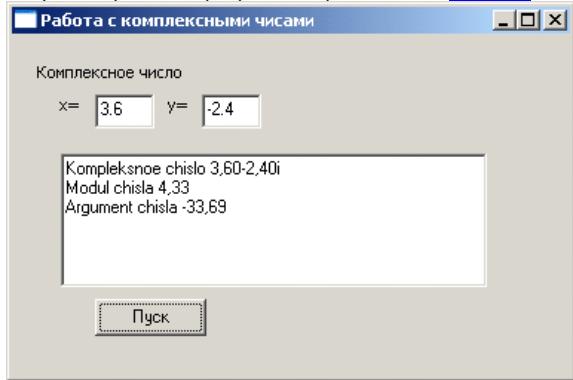


Рис. 9.1. Результаты работы программы работы с классом "Комплексное число". В этом примере был написан конструктор для класса "комплексное число" без параметров. В FreePascal можно писать конструктор с параметрами, который принимает входные значения и инициализирует поля класса этими значениями. Перепишем предыдущий пример следующим образом. Действительную и мнимую часть будем считывать из полей ввода формы и передавать в конструктор для инициализации объекта типа "комплексное число". Листинг программы приведён 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 часть 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 полиморфизмом). Класс представляет собой единство трёх сущностей: полей, свойств и методов, что и представляет собой инкапсуляцию. Инкапсуляция позволяет создавать класс как нечто целостное, имеющее определённую функциональность. Например, класс **TForm** содержит в себе (инкапсулирует) всё необходимое, чтобы создать диалоговое окно.

Основная идея инкапсуляции — защитить поля от несанкционированного доступа. Поэтому поля целесообразно объявлять в разделе **private**. Прямой доступ к полям объекта — чтение и обновление их содержимого — должны 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:

property имя_свойства : тип **read имя_чтения **write** имя_записи**

Зарезервированное слово **read** описывает метод чтения свойств объекта, а слово **write** описывает метод записи свойств объекта. **Имя_чтения** и **имя_записи** — 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", задав ему имя **TPolygon**. Полями класса будут:

- К — количество сторон многоугольника;
- р — 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 введённому, в противном случае количество сторон многоугольника по умолчанию принимается равным 50. После этого вычисляется периметр фигуры, и результаты выводятся на форму в метке **Label1**.

Ниже приведён 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
    function Perimetr () : real; //метод вычисления периметра
    function Show () : String; //метод
//защищенные методы
Protected
    procedure Set_Input (m: integer ); //процедура проверки данных
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 р
    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 введённое 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);
    Figure :=TPolygon. Create; //инициализация объекта
    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;
Label1. Caption := s; // вывод результатов в окно формы
end;
initialization
{$I unit1.lrs}
end.

```

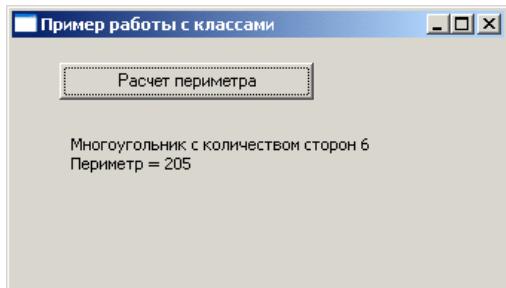


Рис. 9.2. Результат работы программы с классом "многоугольник"

9.3 Наследование и полиморфизм

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 к нему добавляются новые поля, методы и свойства. При создании нового класса является наследником членов ранее определённого базового класса (родителя). Класс-наследник является производным от базового, причём он сам может выступать в качестве базового класса для вновь создаваемых классов.

В Object Pascal все классы являются потомками класса **TObject**. Поэтому если вы создаёте дочерний класс непосредственно от класса **TObject**, то в определении его можно не упоминать.

Производный класс наследует от класса-предка поля и методы. Если имеет место совпадение имён методов, то говорят, что они перекрываются. 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 перекрываются 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 классе 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 больший расход 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 одноимённый метод и реализовать в нём нужные действия. В результате в объекте-родителе и объекте-потомке будут действовать два одноимённых метода, имеющих 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_класса_родителя = class
...
метод; virtual;
...
end;

```

Чтобы перегрузить в классе-наследнике виртуальный метод, нужно после его объявления написать ключевое слово **override**:

```

type
имя_класса_наследника = class (имя_класса_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 компонент 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 **Label15** и **Label16**. Далее приведён 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 "Расчёт"
    //массивы x1, y1 - координаты вершин
    //первого треугольника (частный случай)
    //массивы x1, y1 - координаты вершин
    //второго 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;
    function Perimetr () : real; //метод вычисления периметра
```

```

function Square ( ) : real; //метод вычисления площади
//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;
//Метод 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 третьей 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 );
//вывов 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. Вывывают метод только в тех классах, где он перезагружен. Абстрактный метод объявляется при помощи служебного слова `abstract` после слов `virtual` или `dynaminc`, например:

`метод1; virtual; abstract;`

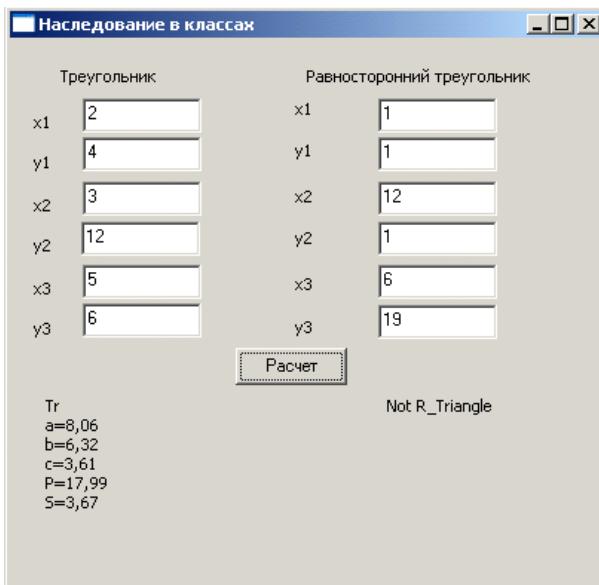


Рис. 9.3. Окно формы работы программы расчёта параметров треугольников
Рассмотрим следующий пример. Создадим базовый класс **TFigure** (фигура). На основании этого класса можно построить производные классы для реальных фигур (окружность, четырёхугольник и т. д.). В нашем примере рассмотрим два класса-наследника **TCircle** (окружность) и **TRectangle** (прямоугольник). На форму поместим кнопку. При щелчке по кнопке инициализируются 2 объекта типа "окружность" и "прямоугольник", рассчитываются параметры фигур: периметр (длина окружности) и площадь. Результаты выводятся в компоненты **Label1** и **label2**. Ниже приведён 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
    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;
    function Perimetr () : real; //метод вычисления периметра
    //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;
  end;

```

```

function Show ( ) : string; override;
end;
var
  Form1 : TForm1;
  Figura1 : Tcircle; //объект типа класса "окружность"
  Figura2 : Trectangle; //объект типа класса "прямоугольник"
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
  Perimetr := 2 * Pi * p [ 0 ]; //вычисление длины окружности
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
  SetLength ( p, n ); //выделение памяти под два элемента
  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

В FreePascal 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_переменной_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 **implementation** 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 четвёртое — как разность этих же чисел. Текст программы с комментариями приведён 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
    function Modul ( ) : real;
    function Argument ( ) : real;
    function ComplexToStr ( ) : String;
  end;
//объявление метода перегрузки
//оператора "+" с комплексными числами
operator +(const a, b : TComplex ) r : TComplex;
//объявление метода 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;
//Метода 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;
//метод перегрузки оператора "-"
operator -( const a, b : TComplex ) r : TComplex;
begin
  r :=TComplex. Create;
  r. x:=a. x-b. x;
  r. y:=a. y-b. y;
end;
function TComplex. ComplexToStr ( ) : String;
//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.
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:=TComplex. Create;
  chislo2:=TComplex. Create;
  chislo3:=TComplex. Create;
  chislo4:=TComplex. Create;
//чтение действительной и мнимой частей двух исходных
//комплексных чисел с формы
  chislo1.x:= StrToFloat ( Edit1. Text );
  chislo1.y:= StrToFloat ( Edit2. Text );
  chislo2.x:= StrToFloat ( Edit3. Text );
  chislo2.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;
  Str1 := ' chislo1+chislo2 '+chislo3. ComplexToStr ( );
  Memo1. Lines.Add( Str1 );

```

```

Str1 := 'Modul chisla '+
FloatToStrF( chislo3. Modul ( ), ffFixed, 5, 2 );
Memo1.Lines.Add( Str1 );
Str1 := 'Argument chisla '+
FloatToStrF( chislo3.Argument ( ), ffFixed, 5, 2 );
Memo1. Lines.Add( Str1 );
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.

```

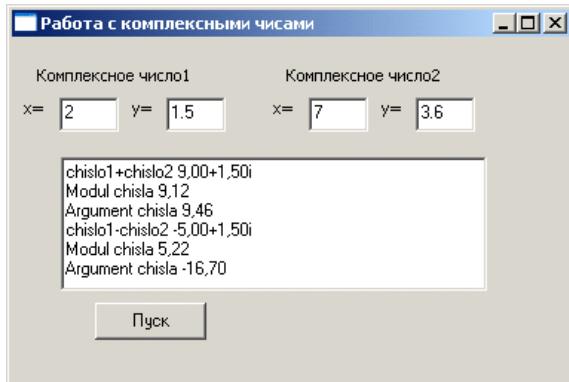


Рис. 9.4. Пример работы программы к задаче 9.1

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 по два метода перегрузки операций сложения и умножения. В первом случае складываются и перемножаются две матрицы. Во втором случае к матрице прибавляется вещественное число, и матрица умножается на вещественное число. На форму поместим кнопку и 8 компонент 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. Четвёртая 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. Первая и вторая матрицы также проверяются, не являются ли они единичными. Текст программы с комментариями приведён ниже, на [рис.](#)

9.5 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;
    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
    x : array [ 0.. 5, 0.. 5 ] of real; //элементы матрицы
  public
    constructor Create; //конструктор
    //метод формирования строки, выводящей элементы матрицы
    function MatrixToStr ( ) : String;
    //функции проверки, является ли матрица единичной
    function Pr_Edin ( ) : boolean;
  end;

```

```

//объявление метода перегрузки операции "+" с
//классом "матрица"
//в данном случае складываются две матрицы
operator +(const a, b : TMatrix ) r : Tmatrix;
//объявление метода перегрузки операции "+" с
//классом "матрица"
//в данном случае к матрице добавляется вещественное число
operator +(const a : TMatrix; b : real ) r : TMatrix;
//объявление метода перегрузки операции "*"
//с классом "матрица"
operator * ( const a, b : TMatrix ) r : Tmatrix;
//объявление метода перегрузки операции "*" с
//классом "матрица"
//в данном случае матрица умножается 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 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( 1 0 );
    inherited Create;
end;
//Функции, проверяющая,
//является ли матрица единичной.
function TMatrix. Pr_Edin ( ) : boolean;
var i, j : integer;
begin
//предварительно решаем, 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 в этом случае 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;
//метод 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;
//метод 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;

```

```

//метод 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;
//Метод 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
                s := s + FloatToStrF ( x [ i, j ], ffFixed, 5, 2 ) + ' ';
        end;
    MatrixToStr := s;
end;
//Обработка кнопки Пуск:
//создание экземпляров класса Матрица,
//вычисление результирующих матриц путём
//сложения, умножения исходных матриц,
//умножение матрица на число
procedure TForm1. Button1Click ( Sender : TObject );
var Str1 : String;
begin
    //Инициализация объектов:
    //матр1, матр2 - исходные матрицы,
    //заполняемые случайными числами;
    //матр3 - матрица, получаемая
    //сложением двух исходных матриц;
    //матр4 - матрица, получаемая умножением
    //двух исходных матриц;
    //матр5 - матрица, получаемая добавлением
    //к первой матрице числа 10;
    //матр6 - матрица, получаемая умножением
    //первой матрицы на число 5.
    матр1 := TMatrix. Create;
    матр2 := TMatrix. Create;
    матр3 := TMatrix. Create;
    матр4 := TMatrix. Create;
    матр5 := TMatrix. Create;
    матр6 := TMatrix. Create;
    //Вычисление матриц матр3, матр4, матр5, матр6
    //с помощью перегрузки операторов.
    матр3 := матр1 + матр2;
    матр4 := матр1 * матр2;
    матр5 := матр1 + 10;
    матр6 := матр1 * 5;
    //Формирование строки вывода и её вывод
    //в метку Label1 матрицы матр1.
    Str1 := ' Matrix1 ' + Chr ( 13 ) + матр1. MatrixToStr ( );
    Label1. Caption := Str1;
    //Проверка, является ли матрица матр1 единичной
    //и вывод соответствующего сообщения.
    if матр1. Pr_Edin () = true then
        Str1 := ' Matrix1 edinichnaya '
        else
            Str1 := ' Matrix1 ne edinichnaya ';
    Label5. Caption := Str1;
    //Проверка, является ли матрица матр2 единичной
    //и вывод соответствующего сообщения.
    if матр2. Pr_Edin () = true then
        Str1 := ' Matrix2 edinichnaya '
        else
            Str1 := ' Matrix2 ne edinichnaya ';
    Label6. Caption := Str1;
    Str1 := ' Matrix2 ' + Chr ( 13 ) + матр2. MatrixToStr ( );
    Label2. Caption := Str1;
    //вывод элементов матрицы матр3=матр1+матр2
    Str1 := ' Matrix1+Matrix2 ' + Chr ( 13 ) + матр3. MatrixToStr ( );
    Label3. Caption := Str1;
    //вывод элементов матрицы матр4=матр1*матр2
    Str1 := ' Matrix1 * Matrix2 ' + Chr ( 13 ) + матр4. MatrixToStr ( );
    Label4. Caption := Str1;
    //вывод элементов матрицы матр5=матр1+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.

```

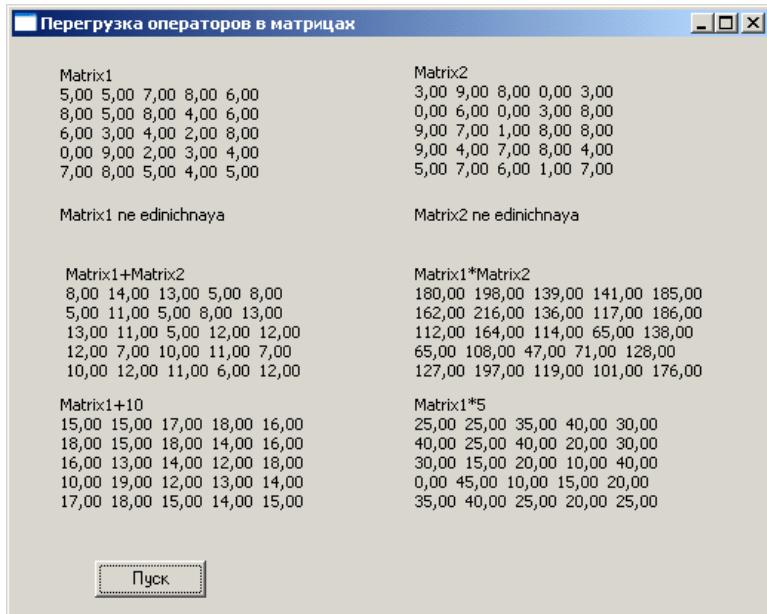


Рис. 9.5. Результаты работы программы задачи 9.2

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 приведён 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
    constructor Create ( a, b : integer ); //конструктор
    //метод формирования строки для вывода дроби
    function DrobToStr ( ) : String;
  end;

```

```

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 класса "обыкновенная дробь"
  d1, d2 : TOb_Drob;
implementation
//конструктор
constructor TOb_drob. Create ( a, b : integer );
begin
  Ch:=a; Zn:=b;
  inherited Create;
end;
//описание метода перегрузки оператора < сравнения
//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;
//описание метода перегрузки оператора > сравнения
//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;
//описание метода 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.

```

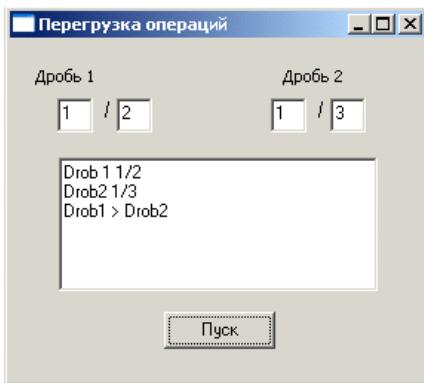


Рис. 9.6. Результаты работы программы задачи 9.3

Лекция 10. Графика в Lazarus

Данная глава посвящена графическим средствам языка. Рассмотрены основные процедуры и функции работы с графикой. Приведён пример построения графика функции.

10.1 Средства рисования в Lazarus

При разработке проекта, в котором 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 **Width**. 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**.

Константа	Цвет	Константа	Цвет
clBlack	Чёрный	clSilver	Серебристый
clMaroon	Каштановый	clRed	Красный
clGreen	Зелёный	clLime	Салатовый
clOlive	Оливковый	clBlue	Синий
clNavy	Тёмно-синий	clFuchsia	Ярко-розовый
clPurple	Розовый	clAqua	Бирюзовый
clTeal	Лазурный	clWhite	Белый
clGray	Серый		

Таблица 10.1. Значение свойств **Color**

Определены следующие константы цветов (табл. 10.1).

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), зелёного (G) и красного (R) цветов и устроены следующим образом: **\$00BBGRR**.

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 соответствующего 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**), подчёркнутым (**fsUnderline**) и перечёркнутым (**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**.

Константа	Значение	Описание
<code>ANSI_CHARSET</code>	0	Символы ANSI
<code>DEFAULT_CHARSET</code>	1	Задаётся по умолчанию. Шрифт выбирается только по его имени Name и размеру Size. Если описанный шрифт недоступен в системе, он будет заменён другим
<code>SYMBOL_CHARSET</code>	2	Стандартный набор символов
<code>MAC_CHARSET</code>	77	Символы Macintosh
<code>GREEK_CHARSET</code>	161	Греческие символы
<code>RUSSIAN_CHARSET</code>	204	Символы кириллицы
<code>EAST_EUROPE_CHARSET</code>	238	Включает диалектические знаки (знаки, добавляемые к буквам и характеризующие их произношение) для восточно-европейских языков

Таблица 10.2. Значения свойства Charset

Класс TPen. Карандаш (перо) используется как инструмент для рисования точек, линий, контуров геометрических фигур. Основные свойства объекта **Canvas.TPen**:

- **Color** (тип `TColor`) — определяет тип линии;
- **Width** (тип `Integer`) — задаёт толщину линии в пикселях;
- **Style** — даёт возможность выбрать вид линии. Это свойство может принимать значение, указанное в [таблице 10.3](#).

Значение	Описание
<code>psSolid</code>	Сплошная линия
<code>psDash</code>	Штриховая линия
<code>psDot</code>	Пунктирная линия
<code>psDashDot</code>	Штрихпунктирная линия
<code>psDashDodDot</code>	Линия, чередующая штрих и два пунктира
<code>psClear</code>	Нет линии

Таблица 10.3. Виды линий

- **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.

Режим	Операция	Цвет пикселя
<code>pmBlack</code>	<code>Black</code>	Всегда чёрный
<code>pmWhite</code>	<code>White</code>	Всегда белый
<code>pmNop</code>	—	Неизменный
<code>pmNot</code>	<code>Not Screen</code>	Инверсный цвет по отношению к цвету фона
<code>pmCopy</code>	<code>Pen</code>	Цвет, указанный в свойствах Color пера Pen (это значение принято по умолчанию)
<code>pmNotCopy</code>	<code>Not Pen</code>	Инверсия цвета пера
<code>pmMergePen</code>	<code>Pen or Not Pen</code>	Дизъюнкция цвета пера и инверсного цвета фона
<code>pmNot</code>		
<code>pmMaskPen</code>	<code>Pen and Not Screen</code>	Конъюнкция цвета пера и инверсного цвета фона
<code>pmMergeNot</code>	<code>Not Pen or Screen</code>	Дизъюнкция цвета фона и инверсного цвета пера
<code>PmMaskNotPen</code>	<code>Not Pen and Screen</code>	Конъюнкция цвета фона и инверсного цвета пера
<code>pmMerge</code>	<code>Pen or Screen</code>	Дизъюнкция цвета пера и цвета фона
<code>pmNotMerge</code>	<code>Not (Pen or Screen)</code>	Инверсия режима <code>pmMerge</code>
<code>pmMask</code>	<code>Pen and Screen</code>	Конъюнкция цвета пера и цвета фона
<code>pmNotMask</code>	<code>Not (Pen and Screen)</code>	Инверсия режима <code>pmMask</code>
<code>pmXor</code>	<code>Pen xor Screen</code>	Операция xor над цветом пера и цветом фона
<code>pmNotXor</code>	<code>Not (Pen xor Screen)</code>	Инверсия режима <code>pmXor</code>

Таблица 10.4. Возможные значения свойства Mode

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** (тип **TBrushStyle**) — определяет стиль заполнения области (**bsSolid** — сплошное заполнение, **bsClear** — прозрачное, **bsHorizontal** — горизонтальные линии, **bsVertical** — вертикальные линии, **bsFDiagonal**, **bsBDiagonal** — диагональные линии, **bsCross** — решётка, **bsDiagCross** — диагональная решётка).

Класс TCANVAS. Этот класс является основным инструментом для рисования графики. Рассмотрим его наиболее часто используемые методы.

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 будет нарисована прямая линия, соединяющая точку с координатами **(0,0)** и точку с координатами **(30,50)**. При повторном щелчке по кнопке процедура продолжит рисовать эту же линию.

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 процедура рисует линию, которая соединяет текущую точку с точкой, получившейся 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.. 2 5 ] 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.

procedure Arc (X1, Y1, X2, Y2, X3, Y3, X4, Y4 : integer);

Метод **Arc** 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.

procedure Rectangle (X1, Y1, X2, Y2 : integer);

Метод **Rectangle** рисует прямоугольник. **X1, Y1, X2, Y2** — координаты верхнего левого и

нижнего правого углов прямоугольника.

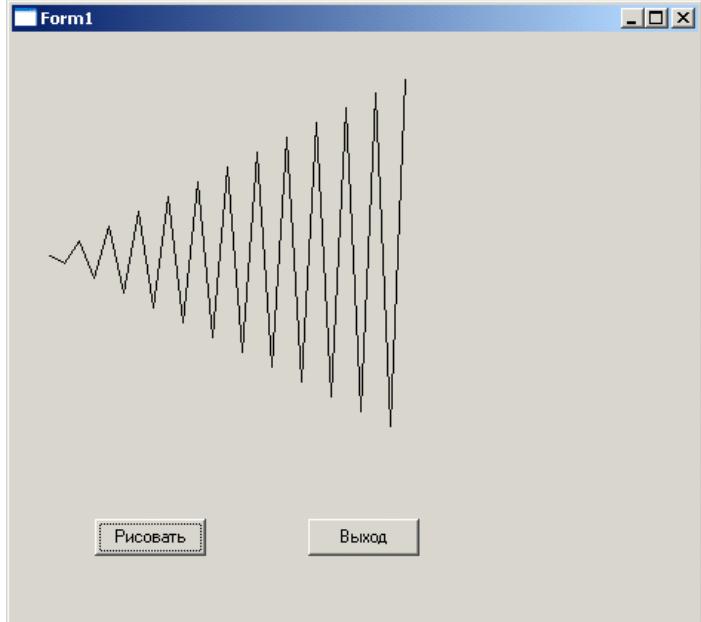


Рис. 10.1. Пример использования процедуры PolyLine

procedure RoundRect (X1, Y1, X2, Y2, X3, Y3 : integer);

Это метод вычерчивания прямоугольника со скруглёнными 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.

procedure PolyGon (const Points array of TPoint);

Метод **PolyGon** 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 **Polyline**, который не замыкает конечные точки. Рисование осуществляется текущим пером **Pen**, а внутренняя область фигуры закрашивается текущей кистью **Brush**.

procedure Pie (X1, Y1, X2, Y2, X3, Y3, X4, Y4 : integer);

Метод **Pie** 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 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)**.

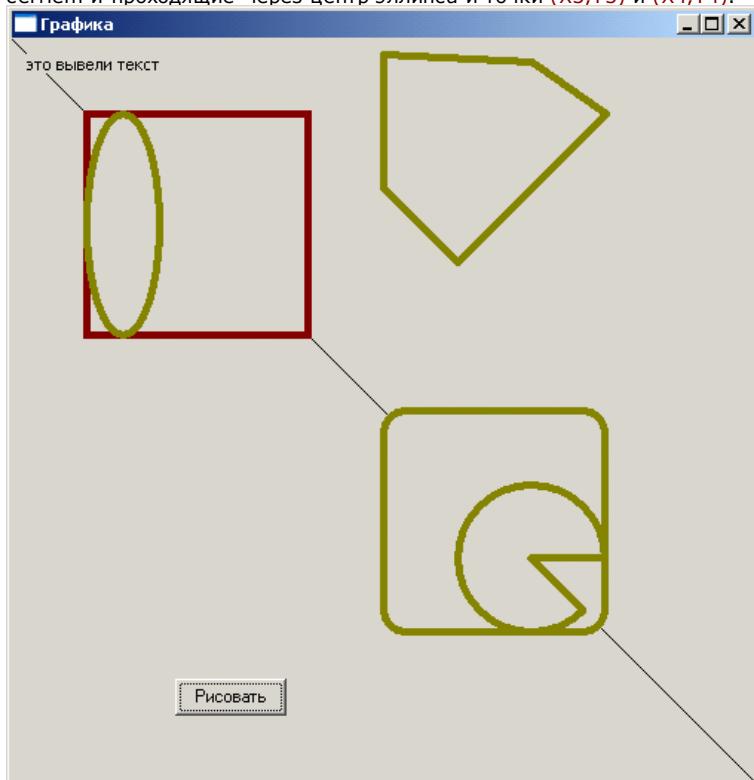


Рис. 10.2. Пример использования методов рисования фигур

Создадим форму, установим ей размеры **Height** — 500, **Weight** — 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 приведён 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;
    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
  //рисование линии
  Form1. Canvas. LineTo ( 500, 500 );
  //изменяем цвет и толщину линии
  Form1. Canvas. Pen. Color := clMaroon;
  Form1. Canvas. Pen. Width:= 5;
  //рисование прямоугольника
  Form1. Canvas. Rect ( 50, 50, 200, 200 );
  Form1. Canvas. Pen. Color := clOlive;
  //рисование эллипса
  Form1. Canvas. Ellipse ( 50, 50, 100, 200 );
  //рисование прямоугольника со скруглёнными углами
  Form1. Canvas. RoundRect ( 250, 250, 400, 400, 30, 30 );
  //рисование сектора окружности
  Form1. Canvas. Pie ( 300, 300, 400, 400, 350, 500, 500 );
  //формирование массива 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.
```

procedure TextOut (X, Y : **integer**; **const** Text : **String**);

Эта функция пишет строку текста **Text**, начиная с позиции с координатами **(X,Y)**. Текущая позиция **PenPos** пера **Pen** перемещается в 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.

function TextExtent (**const** Text : **String**) : **Tsize**;

Эта функция 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;

function TextHeight (**const** Text : **String**) : **integer**;

Функция возвращает высоту в пикселях текста **Text**, который предполагается написать на канве текущим шрифтом.

function TextWidth (**const** Text : **String**) : **integer**;

Функция 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 её 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 её размеры будут точно равны размерам надписи.

Мы рассмотрели основные функции рисования. Теперь можно перейти непосредственно к рисованию. Но перед этим следует заметить, что если вы свернёте окно с графикой, а затем его восстановите, то картинка на форме исчезнет. Изменение размеров окна также может испортить графическое изображение в нём. Для решения этой проблемы существуют процедуры обработки событий **Объект.FormPaint** и **Объект.FormResize**.

Процедура **Объект.FormPaint** выполняется после появления формы на экране, а процедура **Объект.FormResize** — после изменения размера формы. Следовательно, 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 прямыми линиями. Координаты точек определяются по следующим формулам:

$$hx = (b - a)/N,$$

где

$$N$$

— количество отрезков на отрезке

$$[a; b]$$

$$x_i = a + (i - 1)hx; y_i = f(x_i),$$

где

$$i = 0, \dots, N$$

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, её координаты лежат в пределах от 0 до **Width**; ось **OY** направлена вниз, её 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"¹¹

(в этой системе координат ось абсцисс обозначим буквой

$$U, 0 \leq U \leq Width$$

, а ось ординат — буквой

$$V, 0 \leq V \leq Height$$

).

Для преобразования координаты

$$X$$

в координату

$$U$$

построим линейную функцию

$$cX + d$$

, которая переведёт точки из интервала

$$(a; b)$$

в точки интервала

$$(X_0, Width - X_k)$$

¹¹. Поскольку точка

$$a$$

"бумажной" системы координат перейдет в точку

$$X_0$$

"экранной", а точка

$$b$$

— в точку

$$Width - X_k$$

, то система линейных уравнений для нахождения коэффициентов

$$c$$

и

$$d$$

имеет вид:

$$\begin{cases} c \times a + d = X_0 \\ c \times b + d = Width - X_k \end{cases}$$

Решив её, найдём коэффициенты

$$c, d$$

:

$$\begin{cases} d = X_0 - c \times a \\ c = (Width - X_k - X_0)/(b - a) \end{cases}$$

Для преобразования координаты

$$Y$$

в координату

$$V$$

построим линейную функцию

$$V = gY + h$$

. Точка

min

"бумажной" системы координат перейдет в точку

Height - Y_k

"компонентной", а точка

max

— в точку

Y₀

. Для нахождения коэффициентов

g

и

h

решим систему линейных алгебраических уравнений:

$$\begin{cases} g \times \text{min} + h = \text{Height} - Y_k \\ g \times \text{max} + h = Y_0 \end{cases}$$

Её решение позволит найти нам коэффициенты

g

и

h

$$\begin{cases} h = Y_0 - g \times \text{max} \\ g = (\text{Height} - Y_k - Y_0) / (\text{min} - \text{max}) \end{cases}$$

Перед описанием алгоритма построения графика давайте уточним формулы для расчёта коэффициентов

c, d, g

и

h

. Дело в том, что *Width* — это ширина компонента с учётом рамок слева и справа, а *Height* — полная высота компонента с учётом рамки, а если в качестве компонента будет использоваться форма, то необходимо учесть заголовок окна. Однако для изображения графика нам нужны вертикальные и горизонтальные размеры компонент без учёта рамок и заголовка. Эти размеры хранятся в свойствах *ClientWidth* (ширина клиентской области компонента без учёта ширины рамки) и *ClientHeight* (высота клиентской области компонента, без учёта ширины рамки и ширины заголовка) компонента. Поэтому для расчёта коэффициентов

c, d, g

и

h

логичнее использовать следующие формулы:

$$\begin{cases} d = X_0 - c \times a \\ c = (\text{ClientWidth} - X_k - X_0) / (b - a) \end{cases} \quad (10.2)$$

$$\begin{cases} h = Y_0 - g \times \text{max} \\ g = (\text{ClientHeight} - Y_k - Y_0) / (\text{min} - \text{max}) \end{cases} \quad (10.3)$$

Алгоритм построения графика на экране монитора можно разделить на следующие этапы:

1. Определить число отрезков , шаг изменения переменной .
2. Сформировать массивы , вычислить максимальное (*max*) и минимальное (*min*) значения .
3. Найти коэффициенты и по формулам (10.1), (10.2).
4. Создать массивы .
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

rationально использовать матрицы размером

k × *n*

, где

k

— количество функций. Элементы каждой строки матриц являются координатами соответствующего графика в "бумажной" (

Y

) и "компонентной" (

V

) системах координат.

Блок-схема алгоритма изображения графика представлена на [рис. 10.3](#), блок-схема построения графиков к непрерывных функций на [рис. 10.4](#).

Рассмотрим поэтапно каждую 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

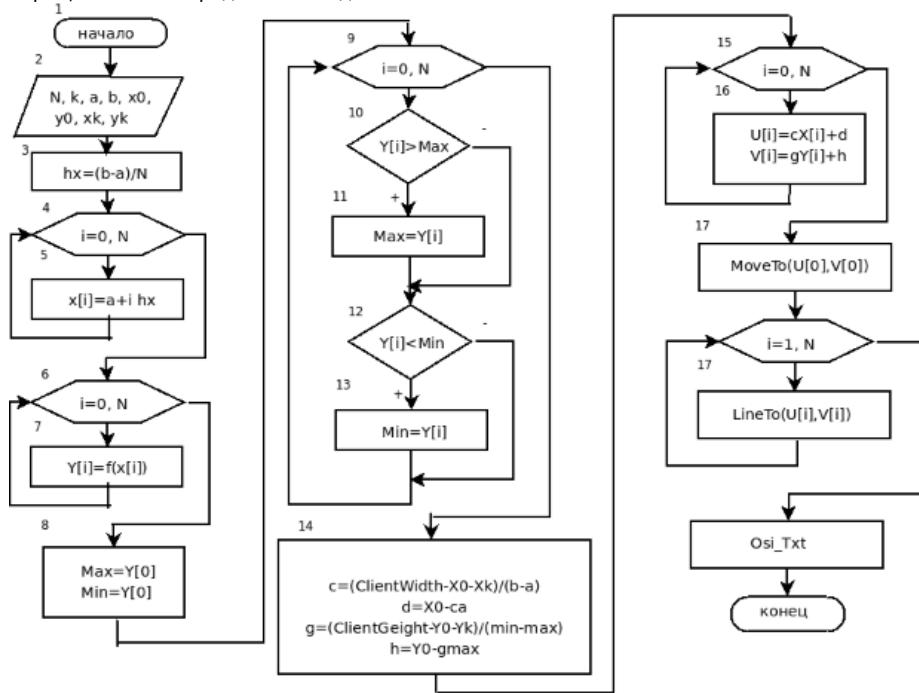
U

и

V

"компонентной" системы координат (этап 4). Блоки 17–19 — это вывод графиков на

экран, и блок 20 предназначен для шестого этапа.



[увеличить изображение](#)

Рис. 10.3. Блок-схема алгоритма построения графика функции

Для второй схемы ([рис. 10.4](#)) реализация второго этапа представлена в блоках 4–15. В блоке 16 рассчитываются коэффициенты

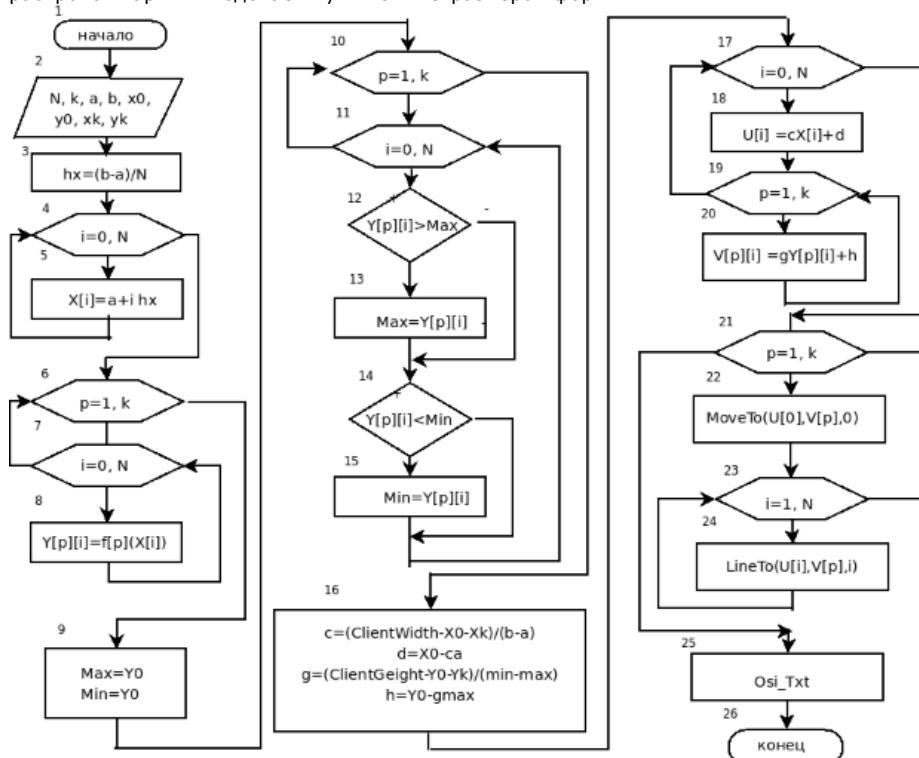
c, d, g
и
h

. В блоках 17–20 формируются массивы четвёртого 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:

$$f(x) = \begin{cases} \sin(x/2), & \text{если } x \leq 0 \\ \sqrt{(1+x)/3}, & \text{если } x > 0 \end{cases}$$

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.



[увеличить изображение](#)

Рис. 10.4. Блок-схема алгоритма построения графика нескольких функций
Установим в качестве свойства формы **Caption** строку "График функции".
Чтобы избавиться от проблемы перерисовки будущего графика при изменении размера
формы, запретим изменение формы и уберём кнопки минимизации и максимизации
окна. Свойство формы **BorderStyle** определяет внешний вид и 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 её
свойства **BorderIcons.BiMaximize** и **BorderIcons.BiMinimize** в **False**.
Для кнопки установим свойство **Caption** — фразу "Построить график".

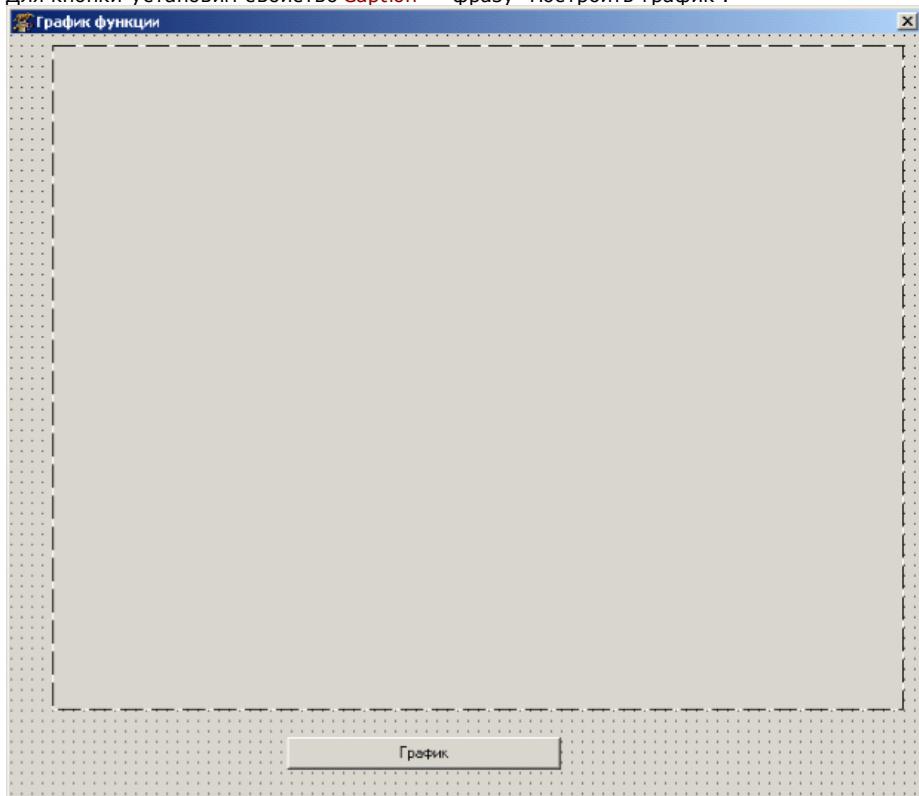


Рис. 10.5. Окно формы после установки свойств
После установки всех описанных свойств окно формы должно стать подобным
представленному на [рис. 10.5](#).

Ввод интервала

a

и

b

выполним с помощью запроса в момент создания формы (в методе инициализации
формы)

FormCreate

). Ниже приведён листинг модуля проекта рисования графика с комментариями:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls, Graphics,
  Dialogs, ExtCtrls, StdCtrls;
//объявление функции, которая задаётся
//математически для построения
function f ( x : real ) : real;
//объявление функции, которая изображает
//график функции на компоненте
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 ОХ
//Ky+1 - количество линий сетки, перпендикулярных оси ОУ
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;
  //нахождение 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 пересчёта 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
  begin
    Form1. Image1. Canvas. LineTo ( u [ i ], v [ i ] );
    Form1. Image1. Canvas. Pen. Width:= 1;
    Form1. Image1. Canvas. Pen. Color := clBlack;
  end;
  //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,перпендикулярными оси ОХ
  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 ОХ
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 в "компонентной" системе
// координат, перпендикулярными оси ОY
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 ОY
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 ОХ 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 х-координаты выводимого под осью ОХ значения
// в "компонентной" системе
tempx:= trunc ( x0+i * ( Form1. Image1. ClientWidth -x0-xk )/KX ) -10;
// вывод значения под осью ОХ
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 ОY 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 ОY
// значения в "компонентной" системе
tempy:= trunc ( yk-5+i * ( Form1. Image1. ClientHeight
-y0-yk )/KY );
// вывод значения левее оси ОY
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, trunc ( 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 );
vars : 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 ([рисунок 10.6](#)) и правой ([рисунок 10.7](#)) границ интервала.

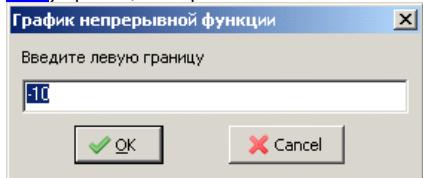


Рис. 10.6. Окно ввода левой границы интервала

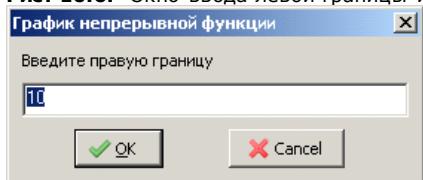
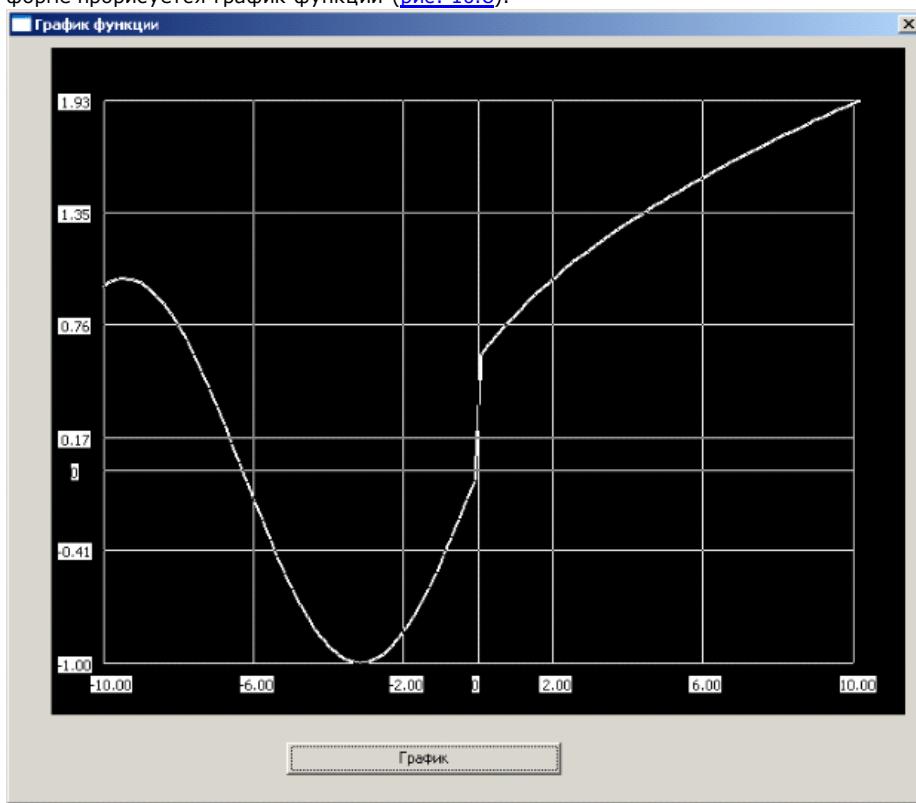


Рис. 10.7. Окно ввода правой границы интервала

После этого появится окно формы с кнопкой **График**. После щелчка по кнопке на форме прорисуется график функции ([рис. 10.8](#)).



[увеличить изображение](#)

Рис. 10.8. Окно с прорисованным графиком

Вместо заключения

Перевёрнута 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 приведённые в ней программы, предложить более простые и быстро работающие алгоритмы, написать свои программы и модули, будет лучшей благодарностью авторам. Если у Вас, читатель, появилось подобное желание, то мы выполнили свою задачу — научили Вас основам программирования.

Следующим этапом в освоении программирования будет разработка своих алгоритмов и написание реально работающих программ для различных операционных систем.

Мы надеемся, что это — только первая книга, посвящённая программированию.
Думаем, что наша книга положит начало новой серии книг, посвящённой
программированию в Linux. Следующей может стать книга, посвящённая решению
более сложных задач на Free Pascal.

Дополнения

Литература

Источник <<http://intuit.vavrkl.ru/course-708/index.html>>