Мансуров К.Т.

Информатика для программиста

введение в специальность

Мансуров К.Т.

Информатика для программиста

введение в специальность

Издание второе,
переработанное и дополненное

Ош-2011

ББК 32.97
М - 23

Рецензенты:
доктор физико-математических наук, профессор Сопуев А.С.
доктор физико-математических наук, профессор Сатыбаев А.С.

М-23 Мансуров К.Т. **Информатика для программиста. Введение в специальность.** Учебник для вузов. – 2-е изд., перераб. и доп. – Ошский Технологический Университет, г. Ош, 2011. – 875 с.: ил.

ISBN 9967-21-148-2

Настоящий учебник предназначен для студентов младших курсов высших учебных заведений, обучающихся по направлению подготовки дипломированных специалистов "Информатика и вычислительная техника", специальностям "Программное обеспечение вычислительной техники и автоматизированных систем", "Вычислительные машины, комплексы, системы и сети", хотя многие главы этой книги могут с успехом использовать студенты и других специальностей при изучении информатики.

Книга является введением в специальность для программистов. Она вводит читателя в круг тех идей, понятий, принципов и методов, на которых зиждется современное программирование и информационные технологии.

Приведены многочисленные задачи и их решения, а также сведения по численным методам, алгоритмам их реализации и пакетах программ.

Во втором издании материал книги значительно переработан. Добавлены главы об объектно-ориентированном программировании, типовых алгоритмах и программирование приложений с графическим интерфейсом.

Изложение языка Паскаль ведется применительно к компилятору Free Pascal и ее графической среде Lazarus.

Учебник и все материалы, входящие в него распространяются на условиях лицензии GNU FDL.

М 2404000000-02 ББК 32.97
ISBN 9967-21-148-2 © Мансуров К.Т., 2011

Содержание

Предисловие ко второму изданию 9
Предисловие к первому изданию 12
Введение 15
Информация и информатика. 16
Формы и виды представления информации 17
Глава I Теоретические основы информатики 19
1.1. Системы счисления 19
1.1.1. Перевод чисел из одной системы счисления в другую. 23
1.1.2. Представление чисел в памяти компьютера 25
1.2. Кодирование чисел 30
1.3. Алгебраическое сложение чисел 34
1.4. Умножение и деление чисел 39
1.5. Логические основы вычислительной техники 43
1.5.1. Законы алгебры логики 50
Глава II Технические средства вычислительной техники 53
2.1. История развития средств вычислительной техники 53
2.2. Логические элементы вычислительной техники 56
2.3. Функциональные элементы 61
2.4. Типовые функциональные схемы 62
2.5. Принципы работы компьютера 65
2.5.1. Что такое "компьютер"? 65
2.5.2. Простейшая структура компьютера 66
2.5.3. Принципы фон - Неймана 68
2.6. Классификация компьютеров 72
2.6.1. Классификация по назначению 72
2.6.2. Другие системы классификации компьютеров 76
2.7. Аппаратное обеспечение 78
Глава III Программное обеспечение компьютеров 93
3.1. Структура программного обеспечения 93
3.2. Основы операционной системы Windows 96
3.2.1. Графический интерфейс Windows 96
3.2.2. Файловая система Windows 106
Глава IV. Основы программирования 114
4.1. Понятие алгоритма. 114
4.1.1 Алгоритм Евклида. 116
4.1.2. Задача о поездах и мухе 121
4.1.3. "Вместо лирического отступления" 130
4.2. Этапы подготовки задачи для решения на компьютере 132
4.3. Примеры разработки алгоритмов 136
4.3.1. Решение квадратного уравнения 136

4.3.2. Вычисление интегралов ... 138
4.3.3. Обработка результатов эксперимента ... 140
4.3.4. Решение системы линейных алгебраических уравнений ... 142
Глава V. Введение в язык программирования Pascal ... 152
5.1. Основные элементы языка ... 152
5.1.1. Переменные. Стандартные типы. ... 154
5.1.2. Операции отношения ... 155
5.1.3. Раздел описаний переменных ... 156
5.1.4. Выражения. Порядок выполнения операций. ... 156
5.1.5. Константы ... 157
5.1.6. Комментарии в программе ... 158
5.1.7. Операторы ... 159
5.1.7.1. Оператор присваивания ... 160
5.1.7.2. Операторы ввода/вывода ... 160
5.1.7.3. Операторы инкремента и декремента ... 162
5.1.8. Среда разработки Lazarus ... 163
5.1.9. Русский язык в консольных приложениях ... 175
5.1.10. Первая программа ... 176
5.1.11. Открытие существующего проекта ... 193
5.1.12. Другие способы создания консольных приложений ... 198
5.1.13. Типовой пустой проект ... 200
5.1.14. Операции с целыми числами ... 202
5.1.15. "Вместо лирического отступления 2" ... 205
5.1.16. Стандартные функции с целыми аргументами ... 207
5.1.17. Операции с вещественными числами (тип real). ... 208
5.1.18. Форматирование вывода ... 209
5.1.19. Одновременное использование вещественных и целых чисел. ... 210
5.1.20. Другие стандартные функции с вещественными аргументами ... 211
5.1.21. Булевы переменные ... 212
5.1.22. Условные операторы. ... 214
5.1.22.1 Оператор if …. then ... 214
5.1.22.2. Оператор if …then ... else ... 215
5.1.23. Операторы цикла ... 220
5.1.23.1. Оператор цикла с предусловием ... 220
5.1.23.2. Оператор цикла с постусловием ... 221
5.1.23.3. Оператор цикла с параметром ... 227
5.1.23.4. Второй вариант оператора цикла с параметром ... 229
5.1.24. Оператор выбора case ... 231
5.1.25. Организация простейшего контроля ввода данных. ... 233
5.1.26. Вычисление сумм сходящихся рядов ... 238
5.2. Реализация некоторых алгоритмов главы 4. ... 243
5.2.1. Программа решения задачи о поездах и мухе ... 243
5.2.2. Программа вычисления определенного интеграла ... 244
Глава VI Более сложные элементы языка ... 248

6.1. Общая структура Паскаль – программы 248
6.1.1. Процедуры и функции 249
6.1.1.1 Структура процедуры 249
6.1.1.2. Структура функции 250
6.1.1.3 Глобальные и локальные переменные 251
6.1.1.4 Способы передачи параметров 262
6.1.1.5 Процедуры завершения 266
6.2. Еще раз о типах данных 266
6.2.1. Классификация типов данных 267
6.2.1.1 Целый тип 267
6.2.1.2. Интервальный тип 268
6.2.1.3. Перечислимый тип 269
6.2.1.4. Множества 269
6.2.1.5. Логический тип 270
6.2.1.6. Вещественный тип 271
6.2.1.7. Указатели 271
6.3. Обработка символьной информации в Паскале 273
6.3.1. Символьные и строковые типы данных. 273
6.3.1.1 Тип Char 278
6.3.1.2. Функции для работы с символами 278
6.3.1.3. Тип String 278
6.3.1.4. Строковые процедуры и функции 284
6.4. Массивы 298
6.4.1. Динамические массивы 306
6.4.2. Программа решения системы линейных алгебраических уравнений методом Гаусса 311
6.4.1.1. Вариант 1 – с `goto` 312
6.4.1.2. Вариант 2 – без `goto` 315
6.4.1.3. Вариант 3 – наилучшая реализация 317
6.5. Модули в Паскале 321
6.5.1. Структура модуля 322
6.5.2. Системные модули 327
6.5.2.1. Модуль CRT 328
6.6. Файлы 333
6.6.1. Тип данных – запись 333
6.6.2. Файловые типы 335
6.6.3. Процедуры для работы с файлами 337
6.6.3.1. Общие процедуры для работы с файлами всех типов 337
6.6.3.2. Процедуры для работы с текстовыми файлами 339
6.6.3.3. Процедуры для работы с типизированными файлами 346
6.6.3.4. Процедуры для работы с нетипизированными файлами 357
6.6.3.5. Организация контроля ввода/вывода при работе файлами 363
6.6.3.6. Создание простой базы данных с типизированными файлами. Программа "Студент" 366
Глава VII. Типовые алгоритмы обработки информации 381
7.1. Алгоритмы сортировки 381
7.1.1. Обменная сортировка (метод "пузырька") 382

7.1.2. Сортировка выбором ... 388
7.1.3. Сортировка вставками ... 395
7.1.4. Метод быстрой сортировки ... 409
7.2. Алгоритмы поиска ... 420
7.2.1. Поиск в массивах ... 420
7.2.2. Вставка и удаление элементов в упорядоченном массиве ... 431
7.3. Динамические структуры данных ... 440
7.3.1. Представление в памяти компьютера динамических структур. ... 446
7.3.2. Реализация стека с помощью массивов ... 449
7.3.3. Представление двоичного дерева в виде массива и реализация алгоритма обхода двоичного дерева слева. ... 458
7.3.4. Указатели ... 470
7.3.5. Стандартные операции с линейными списками ... 474
7.3.6. Реализация динамических структур линейными списками ... 481
7.3.6.1. Реализация стека ... 481
7.3.6.2.Реализация очереди с помощью линейного списка ... 485
7.3.6.3. Реализация двоичного дерева с помощью линейного списка ... 489
7.3.6. Сортировка и поиск с помощью двоичного дерева ... 497
Глава VIII. Основы объектно-ориентированного программирования ... 505
8.1 Три источника и три составные части ООП. ... 505
8.2. Классы и объекты. ... 507
8.2.1. Обращение к членам класса. ... 510
8.3. Инкапсуляция ... 515
8.3.1. Спецификаторы доступа ... 520
8.3.2. Свойства. ... 526
8.4. Наследование ... 535
8.5. Полиморфизм ... 545
8.5.1. Раннее связывание. ... 547
8.5.2. Позднее связывание. ... 552
8.5.3. Конструкторы и деструкторы. ... 558
Глава IX. Программирование приложений с графическим интерфейсом ... 568
9.1 Различия между консольными и графическими приложениями ... 569
9.2. Визуальное программирование в среде Lazarus ... 572
9.2.1. Создание графического приложения ... 572
9.2.2. Форма и ее основные свойства ... 578
9.2.3. Компоненты ... 584
9.2.4. Обработчики событий ... 584
9.2.5. Простейшие компоненты ... 587
9.2.5.1. Компонент `TLabel` ... 587
9.2.5.2. Кнопки TButton, TBitBtn и TSpeedButton ... 603
9.2.6. Организация ввода данных. Однострочные редакторы TEdit, TLabeledEdit ... 607
9.2.6.1. Компонент `TEdit` ... 608
9.2.6.2. Компонент `TLabeledEdit` ... 616

9.2.7. Обработка исключений. Компонент TMaskEdit. Организация контроля ввода данных 621
9.2.7.1. Компонент TMaskEdit 632
9.2.8. Специальные компоненты для ввода чисел TSpinEdit, TFloatSpinEdit и TUpDown 650
9.2.9. Тестирование и отладка программы 653
9.2.10. Компоненты отображения и выбора данных 657
9.2.10.1. Компонент TMemo 657
9.2.10.2. Компонент TStringGrid 711
9.2.10.3. Компоненты выбора 721
9.2.10.4. Компоненты отображения структурированных данных 748
9.2.11. Организация меню. Механизм действий - Actions 822
9.2.11.1. Компонент TMainMenu 822
9.2.11.2. Компонент TToolBar 841
9.2.11.3. Компонент TActionList 845
9.2.11.4. Создание приложений с изменяемыми размерами окон 867
Литература 870
Алфавитный указатель 871

Предисловие ко второму изданию

Первое издание книги "Информатика для программиста. Введение в специальность" увидело свет в 2002 году. За это время книга прошла апробацию на факультете кибернетики и информационных технологий Ошского технологического университета студентами специальностей ПОВТАС и ВМКСС.

В большинстве своем отзывы студентов об этой книге положительны. В то же время автор получил многочисленные просьбы и пожелания расширить и дополнить материал книги. В основном это касалось главы V, посвященной программированию на языке Паскаль. Согласно общему замыслу книги, эта глава была написана в упрощенном виде, для первого знакомства с языком. Однако, как оказалось, студенты пожелали, чтобы материал этой главы был изложен более развернуто и глубоко.

Следуя этим пожеланиям, автором было принято решение дополнить и расширить главу V. Кроме того, было принято решение главу III изъять, поскольку литературы по текстовому редактору Word более чем предостаточно.

Укажу на наиболее существенные изменения и дополнения.

Во-первых, материал главы V, которая в этом издании имеет номер IV, был существенно переработан, дополнен и расширен и изложен теперь в виде трех глав. Кроме того добавлены еще три новые главы, в частности глава посвященная типовым алгоритмам. В этой главе рассмотрены типичные алгоритмы сортировки и поиска, а также алгоритмы работы с динамическими структурами данных.

Добавлена глава об объектно-ориентированном программировании, которая, на взгляд автора, имеет огромное значение. Поскольку современное программирование зиждется именно на ООП и знание и умение применять принципы ООП является неотъемлемой составляющей в подготовке специалистов в области программного обеспечения.

И, наконец, добавлена глава о программировании приложений с графическим интерфейсом. Эта глава является наиболее существенным дополнением к настоящему изданию, поскольку подавляющее большинство приложений разрабатывается на основе графического интерфейса.

Следует отметить еще одну важную особенность настоящей книги.

В предыдущем издании изложение языка Паскаль было ориентировано на компилятор Турбо-Паскаль 7.0. фирмы Borland. Хотя Турбо-Паскаль 7.0. ныне уже не поддерживаются, тем не менее, он являются лицензионными. Тем более правопреемником Borland в настоящее время является компания Embarcadero Technologies.

Несмотря на то, что многие ведущие разработчики программного обеспечения имеют специальные предложения для учебных заведений с существенными скидками, в настоящее время, к сожалению, практически все ВУЗы страны не в состоянии приобретать лицензионные средства разработки программ, например, такие как Embarcadero RAD Studio 2010, Microsoft Visual Studio и многие другие.

На кафедре ПОВТАС ОшТУ в настоящее время принята стратегия на постепенный переход и применение в учебном процессе свободного программного обеспечения. Это позволит снять все проблемы с использованием нелегального лицензионного программного обеспечения.

Достоинством свободного ПО является бесплатность и общедоступность. Кроме того, свободное ПО подразумевает открытый исходный код. Анализ и изучение лучших образцов исходных кодов свободного ПО без сомнения принесет огромную пользу тому, кто решил стать программистом.

Таким образом, планируется переход на операционные системы семейства Linux.

Для изучения языков программирования имеются немало бесплатных компиляторов. В частности, для языка Паскаль имеется компилятор Free Pascal Compiler.

Free Pascal Compiler (часто применяется сокращение FPC) это свободно распространяемый компилятор языка Паскаль с открытыми исходными кодами. Распространяется на условиях GNU – General Public License (GNU GPL). Он совместим с Borland Pascal 7.0. и Object Pascal – Delphi, но при этом обладает рядом дополнительных возможностей, например, поддерживает перегрузку операторов. FPC — кроссплатформенный инструмент, поддерживающий огромное количество платформ. Среди них — AmigaOS, DOS, Linux, *BSD, OS/2, MacOS(X) и Win32.

Кроме того, имеется бесплатная графическая среда для компилятора FPC, которая называется Lazarus.

Lazarus представляет собой среду с графическим интерфейсом для быстрой разработки программ, аналогичную Delphi, и базируется на оригинальной кроссплатформенной библиотеке визуальных компонентов LCL (Lazarus Component Library), совместимых с VCL Delphi.

Как FPC, так и Lazarus можно свободно скачать из Интернета.

Поэтому изложение языка Паскаль в настоящем издании ведется применительно к вышеуказанному компилятору и ее графической среде.

Мансуров К.Т.

Предисловие к первому изданию

Мы вступаем в этап, когда стоящие перед нами проблемы невозможно решить без применения компьютеров.
Я не испытываю страха перед компьютерами. Меня страшит их отсутствие.

Айзек Азимов

Настоящая книга возникла в результате переработки лекций, которые я читал на протяжении ряда лет студентам Ошского технологического университета.

Она предназначена в первую очередь для студентов младших курсов специальности "Программное обеспечение вычислительной техники и автоматизированных систем", а также специальности "Вычислительные машины, комплексы, системы и сети", хотя многие главы этой книги могут с успехом использовать студенты и других специальностей при изучении информатики.

Книга является "как бы" введением в специальность для программистов. Она вводит читателя в круг тех идей, понятий, принципов и методов, на которых зиждется современное программирование и информационные технологии.

Изучив эту книгу, читатель будет вполне подготовлен для усвоения более специальных дисциплин по избранной специальности, в частности таких как:

- алгоритмические языки и программирование;
- технология программирования;

- системное программирование;
- структуры данных;
- системы управления базами данных;
- операционные системы;
- электроника и системотехника;
- организация ЭВМ и комплексов и др.

Автор помнит, как в свое время он очень нуждался в подобной книге. Представьте себе первокурсника, семнадцатилетнего юношу (ах, где моя молодость!), который хочет узнать "сразу обо – всём". Как работает компьютер? Что у него "внутри" происходит, когда он работает? Почему компьютер никогда не ошибается? Как "он понимает", что нужно делать? И много других почему! Хотелось иметь книгу, в которой просто, доходчиво и понятно, почти "на пальцах" объяснялись бы подобного рода вопросы, излагались бы основные идеи и приемы разработки программ, приводились бы примеры и т.д.

Конечно, истинное и глубокое понимание придет к читателю на старших курсах, после прочтения уймы книг, изучения вышеупомянутых дисциплин, но все – таки хотелось бы уже сейчас иметь некоторое представление обо всем этом.

В течение многих лет я вынашивал идею написания подобной книги и наконец "созрел"! Не знаю, удалось ли в полной мере автору реализовать свой замысел (об этом судить читателю!), но основная цель книги заключается в том, чтобы помочь пытливому (именно пытливому!) студенту усвоить азы своей будущей профессии.

Конечно, объять необъятное невозможно, поэтому материал книги подвергся тщательному отбору и здесь представлен тот необходимый минимум, который должен освоить студент. Это та база, опираясь на которую можно в дальнейшем расширять и углублять свои знания, слушая лекции и используя специальную литературу.

К сожалению, современные молодые люди очень мало читают, довольст-

вуясь лишь посещением занятий. На мой взгляд, это тенденция и очень опасная тенденция! Но стоп! Боюсь показаться в глазах читателя эдаким брюзжащим стариком, который вечно ругает молодежь (вот в наше время …!). Как говорится, это философский вопрос, а моя задача в другом. Ограничусь просто советом, читайте больше!

Например, на формирование моих взглядов как программиста огромное влияние оказали труды таких ученых как Лебедев В.Н., Глушков В.М, Ершов А.П. Из зарубежных авторов сильнейшее влияние на меня оказали Дейкстра Э., Донован Дж., Мэдник С., Вирт Н. и, конечно же, Кнут Д. Не будучи знакомым с их книгами, я не состоялся бы как специалист! Так что читайте, читайте и ещё раз читайте, не жалейте на это ни сил, ни времени, ни средств!

Теперь немного о том, как и в какой последовательности читать эту книгу. Студентам специальности ПОВТАС и ВМКСС рекомендую прочитать всю книгу, от начала до конца. Студентам других специальностей ("чайникам"!) главу I, главу II (§2.1, §2.2) можно пропустить, поскольку они носят теоретический и немного специальный характер.

В заключение хочу выразить благодарность моим коллегам, преподавателям кафедры "Вычислительная техника" ОшТУ, с которыми я неоднократно обсуждал содержание книги, в частности Джуманову Д., Игамбердиеву Р., Худайбердиеву Д., которые проверили все программы этой книги на компьютере.

Спасибо моим детям, которые безропотно уступали мне компьютер, когда я работал над этой книгой.

Спасибо моей жене, которая мужественно терпела мои ночные бдения за компьютером, вместо того, чтобы заниматься более "полезным" (с её точки зрения) делом!

Спасибо моим многочисленным студентам, которые были "подопытными кроликами" во время моих "методических экспериментов" на лекциях.

Мансуров К.Т.

Введение

Кончай разговоры, пошли к лошадям.

Английская поговорка

Становление информатики как научной дисциплины относится к 60-годам прошлого столетия. В самом общем смысле под информатикой понимают фундаментальную естественную науку, изучающую процессы передачи, накопления и обработки информации.

Термин "Informatique" был введен французами на рубеже 60-70-х годов. Но еще раньше американцами был введен в употребление термин "Computer Science" для обозначения науки о преобразовании информации, базирующейся на вычислительной технике.

В настоящее время термины "Informatique" и "Computer Science" употребляются в эквивалентном смысле.

В нашей стране для обозначения новой научной дисциплины применялись термины "Вычислительные науки" или " Вычислительное дело", использование же термина "Информатика" сдерживалось употреблением его в области, связанной с документалистикой. Новое толкование этого термина относится к 1976г., когда появился русский перевод книги Ф. Бауэра и Г.Гооза " Информатика".

Известно широкое определение информатики, данное Международным конгрессом в Японии в 1978г.: "Понятие информатики охватывает области, связанные с разработкой, созданием, использованием и материально- техническим обслуживанием систем обработки информации, включая машины, оборудование, математическое обеспечение, организационные аспекты, а также комплекс промышленного, коммерческого,

административного, социального и политического воздействия".
В этом определении можно выделить то главное, что и составляет основу современного содержания информатики, а именно компьютеры и компьютерную обработку информации.

Исходя из такого толкования ядра информатики, можно сказать, что его содержание определяют три неразрывно связанные между собой части: алгоритмические, программные и технические средства. Элементы этих составных частей информатики и рассматриваются в настоящем учебнике.

Информация и информатика.

Информация об окружающем мире необходима человеку так же, как воздух, вода, тепло. Представьте себе, что Вы оказались в замкнутом пространстве без света и звука, с абсолютно гладкими поверхностями. Вы почувствуете себя очень неуютно. Отсутствие информации о времени, пространстве, окружающей действительности противоестественно для человека. Человек видит, слышит, осязает, чувствует запахи. Органы чувств служат человеку для получения информации.

Информация – это сообщение о состоянии и свойствах объекта, явления, процесса. Информация преобразуется уникальным устройством- человеческим мозгом. Ребенок, впервые увидевший огонь, не боится его. Но обжегшись, запоминает на всю жизнь – с огнем играть нельзя. Значит, мозг человека не только преобразует информацию, но и запоминает – хранит ее. Человечество за тысячелетия своего существования накопило огромное количество информации. Мозг человека не в состоянии хранить такой объем ее и без искажения передавать другим людям. Поэтому для хранения и накопления информации с незапамятных времен использовались природные средства: рисунки на стенах пещер, скалах. Носители информации непрерывно совершенствовались, так постепенно появились: пергамент, папирус,

береста, бумага, перфорационные носители информации (перфокарты, перфоленты), фотопленка, магнитные носители (ленты, карты, диски и т.д.).

Первобытный человек, создав первые примитивные орудия труда, положил начало эпохе механизмов, увеличивающих физическую силу человека. В середине XX столетия были созданы первые электронные вычислительные машины, предназначенные для увеличения интеллектуальной мощи человека. Появление ЭВМ не случайно. Лавинообразно растут информационные потоки. Особенно это характерно для промышленности, управления и науки. Общая сумма человеческих знаний в XVIII веке удваивалась каждые 50 лет, к 1950г. –каждые 10 лет, к 1970г. – 5 лет, к 1985г. –каждые 2-3 года. До XX века основным предметом труда были материальные объекты, экономическая мощь государства измерялась его материальными ресурсами, теперь основным предметом труда становится информация. Человек не в состоянии без электронных помощников навести порядок в информационном хозяйстве.

Возросшая роль информации и уровень развития средств обработки информации привели к появлению понятия "информационные ресурсы". Информационные ресурсы – информация, используемая на производстве, в технике, в управлении обществом, специально организованная и обрабатываемая на компьютере. Процесс компьютеризации общества привел к появлению нового термина – пользователь. Пользователем будем называть человека любой профессии, который непосредственно взаимодействует с компьютером.

Формы и виды представления информации

Выделяют две формы представления информации – непрерывную и дискретную. Непрерывная форма – это величина, характеризующая процесс, не имеющий перерывов или промежутков, например: температура человека, ско-

рость перемещения автомобиля за определенное время на участке пути без остановок.

Дискретная форма представления информации – это последовательность символов, характеризующая прерывистую, изменяющуюся величину. Например, рабочий за первый час смены обработал на станке 40 деталей, за второй – 45, за третий 38 и т.д.

Часто возникает необходимость представить информацию в форме отличной от привычной. Так, для передачи информации на расстояние изобретен телеграфный код Морзе, в которой буквы и цифры закодированы с помощью коротких и длинных импульсов (точка, тире). Почтовый индекс – закодированный адрес.

Электронные элементы, применяемые в компьютерах, имеют два состояния – есть импульс или нет импульса, поэтому вся информация для компьютеров кодируется в двоичной системе счисления. Любой символ (цифра, буква, знак) получает кодированное обозначение с помощью цифр 1 и 0, составляющих основу двоичной системы счисления.

Рассмотрим основные единицы измерения информации. Наименьшей единицей информации является бит (bit), принимающий только значение 1 или 0. Более крупная единица информации – байт (byte). Он состоит из восьми бит. Байт – основная единица количества информации, используется для образования еще более крупных единиц информации.

Глава I

Теоретические основы информатики

Записи были сделаны… римскими цифрами, отнюдь не арабскими, которые считались легкомысленным нововведением, неприличным в деловой книге.

Неизвестный автор, 1494 г.

1.1. Системы счисления

Система счисления – совокупность правил и приемов для записи чисел с помощью различных знаков или символов.

Системы счисления делятся на позиционные и непозиционные.

Непозиционная система счисления – система, для которой значение символа не зависит от его положения в числе. Примером непозиционной системы счисления может служить римская система счисления. Используется набор знаков I, V, X, L, C, D, M, …

Например: числа IV и VI Значение символа V одно и то же и равно пяти.

Позиционная система счисления – система, которая использует фиксированный набор знаков (символов) для записи числа и значение этого знака (символа) зависит от его положения в числе.

Например, 654 это не одно и то же, что 645

Число единиц

Число десятков

Число сотен

Любая позиционная система счисления характеризуется основанием или базисом.

Базисом (основанием) q позиционной системы счисления является коли-

чество знаков или символов используемых для записи числа в данной системе счисления.

Например, в десятичной системе счисления q=10, используются знаки или цифры 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Для любой позиционной системы счисления имеет место равенство:

$$A_{(q)} = a_n q^n + a_{n-1} q^{n-1} + \ldots + a_1 q^1 + a_0 q^0 + a_{-1} q^{-1} + \ldots + a_{-m} q^{-m} \quad (1.1.1)$$

где $A_{(q)}$ - произвольное число в q-ичной системе счисления, q - базис данной системы счисления, a_i - знаки или символы данной системы счисления.

Для десятичной системы имеем:

q=10, a_i={0,1,2,3,4,5,6,7,8,9}

$$A_{(10)} = a_n 10^n + a_{n-1} 10^{n-1} + \ldots + a_1 10^1 + a_0 10^0 + a_{-1} 10^{-1} + \ldots + a_{-m} 10^{-m} \quad (1.1.2)$$

или в сокращенной записи:

$$A_{(10)} = \sum_{i=-m}^{n} a_i 10^i \quad (1.1.3)$$

На практике используют условную, сокращенную запись числа:

$$A_{(q)} = a_n a_{n-1} \ldots a_1 a_0, a_{-1} a_{-2} \ldots a_{-m} \quad (1.1.4)$$

Пример:

$$A_{(10)} = 654 = 6 \cdot 10^2 + 5 \cdot 10^1 + 4 \cdot 10^0$$

$$A_{(10)} = 358{,}124 = 3 \cdot 10^2 + 5 \cdot 10^1 + 8 \cdot 10^0 + 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 4 \cdot 10^{-3}$$

Существуют и другие системы счисления:

1. **<u>Двоичная система счисления</u>**

$$q = 2, \quad a_i = \{0,1\}$$

$$A_{(2)} = \sum_{i=-m}^{n} a_i 10^i \quad (1.1.5)$$

Правила образования чисел как в десятичной системе: вес k-й позиции числа равен q^k. Для двоичной системы 2^k.

$$0_{(10)} = 0_{(2)}$$
$$1_{(10)} = 1_{(2)}$$
$$2_{(10)} = 10_{(2)}$$
$$3_{(10)} = 11_{(2)}$$
$$4_{(10)} = 100_{(2)}$$

Число $14_{(10)} = 1110_{(2)} = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 8 + 4 + 2 + 0 = 14_{(10)}$

2. **<u>Восьмеричная система счисления</u>**

$$q = 8, \quad a_i = \{0,1,2,3,4,5,6,7\}$$

$$A_{(10)} = 118{,}375_{(10)} = 166{,}3_{(8)} = 1 \cdot 8^2 + 6 \cdot 8^1 + 6 \cdot 8^0 + 3 \cdot 8^{-1} = 64 + 48 + 6 + \frac{3}{8} = 118{,}375_{(10)}$$

3. **<u>16 – ричная система счисления</u>**

$$q = 16, \quad a_i = \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$$

$$100_{(10)} = 64_{(16)} = 6 \cdot 16^1 + 4 \cdot 16^0 = 100_{(10)}$$

Разрядом называется позиция одной цифры в числе. Вес разряда p_i в позиционной системе счисления это отношение $p_i = \frac{q^i}{q^0} = q^i$, где i - номер разряда справа налево.

Если разряд p_k имеет вес q^k, то следующий старший разряд (слева) имеет вес $p_{k+1} = q^{k+1}$, а следующий младший разряд имеет вес $p_{k-1} = q^{k-1}$.

Такая взаимосвязь между разрядами приводит к необходимости передачи информации между разрядами. Если в текущем разряде накопилось единиц больше или равное q, должна происходить передача единицы в соседний старший разряд. Этот процесс называется *переносом*. Передача единицы из старшего разряда в младший называется *заемом*.

Длиной числа называется количество разрядов необходимым для записи числа. В техническом аспекте длина числа интерпретируется как длина разрядной сетки процессора. В большинстве компьютеров длина разрядной сетки фиксирована. Если длина разрядной сетки задана, то это ограничивает максимальное и минимальное по модулю число, представимое в компьютере.

Пусть длина разрядной сетки равна n ($n > 0$, целое). Тогда

$$A_{(q)\max} = q^n - 1;\ A_{(q)\min} = -(q^n - 1) \qquad (1.1.6)$$

Отсюда диапазон представления чисел в заданной системе счисления и для заданной разрядной сетке это интервал числовой оси, заключенный между $A_{(q)\max}$ и $A_{(q)\min}$, т.е.

$$A_{(q)\min} \le \text{ДП} \le A_{(q)\max} \qquad (1.1.7)$$

Таблица соответствия десятичных чисел в других системах счисления

Таблица 1.1

Десятичные числа	Их эквиваленты в других системах		
	q =2	q =8	q =16
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

1.1.1. Перевод чисел из одной системы счисления в другую.

Пусть необходимо перевести число в системе счисления с основанием q в число в системе счисления с основанием r.

$$\begin{aligned} A_{(q)} &= a_n q^n + a_{n-1} q^{n-1} + \ldots + a_1 q^1 + a_0 q^0 + a_{-1} q^{-1} + \ldots + a_{-m} q^{-m} = \\ &= b_k r^k + \cdots + b_1 r^1 + b_0 r^0 + b_{-1} r^{-1} + \cdots + b_{-s} r^{-s} = A_{(r)} \end{aligned} \qquad (1.1.8)$$

Задачу перевода числа из системы счисления q в систему счисления r можно представить в общем виде как задачу определения коэффициентов нового ряда b_j

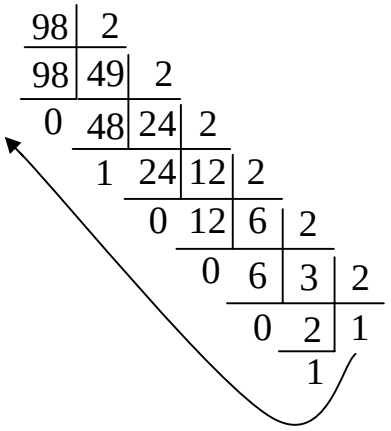
$$\begin{aligned} 0 \le a_i \le q-1, \qquad i \in [-m, n] \\ 0 \le b_j \le r-1, \qquad j \in [-s, k] \end{aligned} \qquad (1.1.9)$$

Например,

$$56_{(10)} = 5 \cdot 10^1 + 6 \cdot 10^0 = 111000_{(2)} = 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 32+16+8 = 56_{(10)}$$

<u>Правило перевода целых чисел</u>

Для перевода целого числа в системе счисления с основанием q в число в системе с основанием r необходимо последовательно делить это число и получаемые затем целые части частных на основание r новой системы счисления до тех пор, пока очередная полученная целая часть частного не станет меньше r. Эта вновь полученная целая часть определит старшую цифру числа в новой системе счисления, а все предыдущие остатки – младшие цифры вновь полученного числа. Пример: перевести в двоичную систему счисления число $X_{(10)} = 98_{(10)}$



Получим $X_{(10)} = 98_{(10)} = 1100010_{(2)} = X_{(2)}$

<u>Правило перевода правильных дробей.</u>

Для перевода правильной дроби из одной системы счисления в другую надо её и получающиеся затем дробные части произведений умножать на основание новой системы счисления до тех пор, пока в новой дроби не будет нужного количества цифр, которая определяется требуемой точностью представления дроби. Правильная дробь в новой системе счисления записывается из целых частей произведений, получающихся при последовательном умножении, причем первая целая часть будет старшей цифрой новой дроби.

Пример: $X_{(10)} = 0{,}375$ перевести в двоичную систему с 4-мя знаками после запятой.

$$
\begin{array}{rr}
 & 0{,}375 \\
\times & 2 \\
\hline
 & 0{,}750 \\
\times & 2 \\
\hline
 & 1{,}500 \\
\times & 2 \\
\hline
 & 1{,}000 \\
\times & 2 \\
\hline
 & 0{,}000
\end{array}
$$

$X_{(2)} = 0{,}0110_{(2)} = 0{,}375_{(10)}$

<u>Правило перевода неправильных дробей.</u>

При переводе неправильных дробей отдельно переводят целую и дробную части числа по правилам, приведенными выше.

$$X_{(10)} = 98{,}375_{(10)} = 1100010{,}0110_{(2)}$$

1.1.2. Представление чисел в памяти компьютера

Существуют две формы представления чисел: естественная и нормальная.

Естественной называется такая форма записи числа, которая в неявном виде реализует формулу:

$$X = \sum_{i=-m}^{n} a_i q^i \qquad (1.1.10)$$

При этом число записывают только при помощи набора цифр a_i, без явного указания их весов и знаков сложения между ними.

Отсчет весов ведется от запятой, которая фиксируется между целой и дробной частью числа. Запись числа имеет вид:

$$a_n a_{n-1} \cdots a_1 a_0, a_{-1} \cdots a_{-m} \qquad (125 = 1 \cdot 10^2 + 2 \cdot 10^1 + 5 \cdot 10^0)$$

Это формой мы обычно пользуется.

Нормальной называется такая форма записи числа, которая в неявном виде реализует формулу:

$$X = \left(\sum_{i=-m}^{n} a_i q^{i-p} \right) q^p \qquad (1.1.11)$$

когда число представляется в виде произведения правильной дроби и некоторой целой степени основания системы счисления.

При этом правильная дробь называется мантиссой, а показатель степени – порядком. Число 125 в нормальной форме представляется в виде:

$$0{,}125 \cdot 10^3$$

Так как в естественной форме положение запятой в числе строго зафиксирована между целой и дробной частью, то часто числа в такой форме называют числами с фиксированной запятой. Следует отметить, что во многих западных странах для отделения целой и дробной частей числа используют точку вместо запятой. Поэтому часто числа в такой форме называют ещё числами с фиксированной точкой.

В дальнейшем мы будем пользоваться в основном термином "с фиксированной точкой" как наиболее распространенной и часто используемой в литературе.

В определении нормальной формы не наложено никаких ограничений на величину мантиссы, лишь бы она была правильной дробью. Поэтому положение запятой может меняться при изменении порядка. Поэтому числа в нормальной форме называют числами с плавающей точкой или с плавающей запятой.

Недостатки чисел с плавающей точкой:

1. Сложная конструкция аппаратуры процессора для реализации различных операций над числами из-за неоднозначности записи чисел и необходимости вследствие этого нормализации числа.
2. Ещё большее усложнение аппаратуры, во – первых из – за сложности структуры числа, состоящей из двух частей: мантиссы и порядка и, во – вторых вследствие того, что приходится производить действия с обеими частями числа.

Представление с фиксированной точкой не имеет указанных недостатков, но, к сожалению, у этой формы есть свои недостатки:

1. Малый диапазон чисел, представимых в компьютере.

Пусть имеется некий гипотетический компьютер с 9-ти разрядной сеткой для записи десятичных чисел. Пусть также задано, что компьютер может работать в двух режимах: с фиксированной и с плавающей точкой, причем на мантиссу отведено 5 разрядов и на дробную часть отведено тоже 5 разрядов.

Форма с фиксированной точкой:

1	2	3	4	5	6	7	8	9
знак	целая часть			дробная часть				

Форма с плавающей точкой:

1	2	3	4	5	6	7	8	9
знак мантиссы	мантисса					знак порядка	порядок	

Знак числа имеет двоичную природу, так как может принимать лишь два значения (+) и (-). Знак (+) принято изображать цифрой 0, а знак (-) цифрой 1

Режим с фиксированной точкой.	Режим с плавающей точкой.
$+\lvert X\rvert_{\max} = +999{,}99999$	$+\lvert X\rvert_{\max} = +0{,}99999 \cdot 10^{+99}$
$+\lvert X\rvert_{\min} = +000{,}00000$	$+\lvert X\rvert_{\min} = +0{,}10000 \cdot 10^{-99}$
$-\lvert X\rvert_{\min} = -000{,}00000$	$-\lvert X\rvert_{\min} = -0{,}10000 \cdot 10^{-99}$
$-\lvert X\rvert_{\max} = -999{,}99999$	$-\lvert X\rvert_{\max} = -0{,}99999 \cdot 10^{+99}$

Отсюда мы можем получить оценку диапазонов представления чисел в обоих формах:

$$\begin{aligned} -10^{+3} &< X_{ф.т.} < +10^{+3} \\ -10^{+99} &< X_{п.т.} < +10^{+99} \end{aligned} \qquad (1.1.12)$$

2. Меньшая точность.

Рассмотрим для двоичной системы. Абсолютная погрешность для чисел с фиксированной точкой составляет:

$$\Delta_{ф.т.} = \frac{1}{2} 2^{-n} = const \qquad (1.1.13)$$

Абсолютная погрешность для чисел с плавающей точкой составляет:

$$\Delta_{п.т.} = \frac{1}{2} 2^{-n} 2^{p} \qquad (1.1.14)$$

$\Delta_{п.т.}$ зависит от p и, следовательно, меняется от

$$\left|\Delta_{п.т.}\right|_{\min} = \frac{1}{2} 2^{-n} 2^{-|P_{\max}|} \qquad (1.1.15)$$

до

$$\left|\Delta_{п.т.}\right|_{\max} = \frac{1}{2} 2^{-n} 2^{+|P_{\max}|} \qquad (1.1.16)$$

Относительная погрешность для чисел с фиксированной точкой определяется следующим образом:

$$\delta_{ф.т.\min} = \frac{\Delta_{ф.т.}}{\left|X_{ф.т.\max}\right|} \qquad (1.1.17)$$

$$\delta_{ф.т.\max} = \frac{\Delta_{ф.т.}}{\left|X_{ф.т.\min}\right|} \qquad (1.1.18)$$

где

$$\begin{aligned} \left|X_{ф.т.\max}\right| &= \overbrace{0{,}11\cdots1}^{n} = 1 - 2^{-n} \\ \left|X_{ф.т.\min}\right| &= 0{,}00\cdots1 = 2^{-n} \end{aligned} \qquad (1.1.19)$$

Теперь можно получить оценку относительной погрешности для чисел с фиксированной точкой:

$$\delta_{ф.т.\min} = \frac{\frac{1}{2} 2^{-n}}{1 - 2^{-n}} \cong 2^{-(n+1)} \qquad (1.1.20)$$

если $1 >> 2^{-n}$

$$\delta_{ф.т.\max} = \frac{\frac{1}{2}2^{-n}}{2^{-n}} = \frac{1}{2} = 50\% \qquad (1.1.21)$$

То есть максимальная относительная погрешность может достигать 50%, что конечно во многих случаях неприемлемо!

Рассмотрим теперь относительную погрешность для чисел с плавающей точкой:

$$\delta_{n.т.} = \frac{\Delta_{n.т.}}{|X_{n.т.}|}$$

$$\delta_{n.т.\min} = \frac{|\Delta_{n.т.\max}|}{|X_{n.т.\max}|} = \frac{\frac{1}{2}2^{-n}2^{+|P_{\max}|}}{(1-2^{-n})2^{+|P_{\max}|}} \cong 2^{-(n+1)} \qquad (1.1.22)$$

при $1 >> 2^{-n}$

$$\delta_{n.т.\max} = \frac{|\Delta_{n.т.\min}|}{|X_{n.т.\min}|} = \frac{\frac{1}{2}2^{-n}2^{-|P_{\max}|}}{\frac{1}{2}2^{-|P_{\max}|}} = 2^{-n} \qquad (1.1.23)$$

Отсюда видно, что относительная погрешность $\delta_{n.т.}$ не зависит от величины p, а зависит от n.

Главный вывод в том, что величина ошибки для чисел с плавающей точкой изменяется в очень малом диапазоне и очень мала ($2^{-n} << 1$), если числа нормализованы.

1.2. Кодирование чисел

Писатель пользуется известными привилегиями, в благодетельности которых, надеюсь нет никаких оснований сомневаться. Так, встретив у меня непонятное место, читатель должен предположить, что под ним кроется нечто весьма полезное и глубокомысленное.

Джонатан Свифт

Вопрос о кодировании чисел возникает по той причине, что в компьютер либо нельзя, либо нерационально вводить числа в том виде, каком они изображаются человеком – вычислителем на бумаге.

Во-первых, нужно кодировать знак числа. Во-вторых, приходится иногда и само число кодировать. В частности, при выполнении операций с фиксированной точкой все числа приводятся к виду правильных дробей.

Простейший машинный код, называемый прямым, получается в том случае, когда знак двоичной дроби кодируется 0 (+) и 1 (-), а цифровая часть остается без изменений.

Более строго прямой код можно определить следующим образом:

Прямым кодом отрицательной двоичной дроби называется её обычное изображение в естественной форме, у которого в знаковом разряде проставляется единица. Прямой код положительной двоичной дроби совпадает с её обычным изображением в естественной форме. Аналитически:

$$[X]_{пр} = \begin{cases} X, \ X \geq 0 \\ 1 - X, X < 0 \end{cases} \qquad (1.2.1)$$

$$X_1 = +0{,}110011_{(2)} \qquad [X_1]_{пр} = 0{,}110011$$
$$X_2 = -0{,}110011_{(2)} \qquad [X_2]_{пр} = 1{,}110011$$

Так как операция сложения значительно проще реализуется, чем операция вычитания, то возникает вопрос: нельзя ли каким либо способом свести алгебраическое сложение к арифметическому, т.е. попросту говоря заменить операцию вычитания сложением.

Идея замены операции вычитания сложением основана на применении некоторых вспомогательных положительных чисел, однозначно связанных с исходными отрицательными числами. Эти вспомогательные числа находятся как дополнения (по абсолютной величине) заданных отрицательных чисел до некоторого $X_{гр}$:

$$X_{гр} = |X_{\max}| + 1_{м.р.} = 10^n \qquad (1.2.2)$$

где $1_{м.р.}$ означает единицу младшего разряда.

Пусть имеются двухразрядные десятичные числа.

$$0 \le X_{тек} \le X_{\max} = 99_{(10)}$$

Здесь $X_{гр} = X_{\max} + 1 = 99 + 1 = 100 = 10^2$

Пусть нужно алгебраически сложить

$$X_1 = 84 \quad \text{и} \quad X_2 = -63$$

X_2 согласно вышесказанному, мы заменим его дополнением (по абсолютной величине) до $X_{гр}$ (до ста), т.е.

$$[X_2]_{доп} = X_2 + X_{гр} = X_2 + 100 = -63 + 100 = 37$$

Далее складываем X_1 и $[X_2]_{доп}$

$$X_1 + [X_2]_{доп} = 84 + 37 = \underbrace{1}_{\text{отбрасывается}}21$$

Прямое вычитание $84 - 63 = 21$ показывает правильность наших действий.

Нахождение $[X_2]_{доп}$ производят без вычитания. Выпишем цифры от 0 до 9 сначала в прямом порядке, затем в обратном:

$$\begin{array}{l} 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9 \\ \qquad\qquad\qquad\ \updownarrow\ \updownarrow\ \updownarrow\ \updownarrow\ \updownarrow \\ 9,\ 8,\ 7,\ 6,\ 5,\ 4,\ 3,\ 2,\ 1,\ 0 \end{array} \quad \Leftarrow \quad \text{взаимно обратные}$$

Отсюда видно, что замена в некотором отрицательном числе всех цифр на взаимно обратные равносильно сложению исходного числа с $|X|_{\max}$.

Правило нахождения дополнения отрицательных чисел без использования операции вычитания:

для получения дополнения отрицательного числа следует все цифры исходного числа заменить на взаимно обратные и к полученному результату добавить единицу младшего разряда.

Действительно:

$$[X_2]_{доп} = X_2 + X_{гр} = X_2 + |X_{\max}| + 1 = -63 + 99 + 1 = 36 + 1 = 37$$

Аналогично и вместе с тем проще для двоичной системы счисления.

$$\begin{array}{l} 0\ 1 \\ 1\ 0 \end{array} \text{ взаимно обратные цифры}$$

Дополнение здесь ищется до двух.

Дополнительным кодом отрицательной двоичной дроби называется её дополнение по модулю до двух.

Правило:

Для получения дополнительного кода отрицательной двоичной дроби нужно в знаковом разряде записать 1, а во всех других разрядах цифры заменить на взаимно обратные, т.е. 0 на 1, а 1 на 0, после чего к младшему разряду числа добавляется 1.

Дополнительный код положительной двоичной дроби совпадает с её пря-

мым кодом. Аналитически это можно выразить следующим образом:

$$[X]_{доп} = \begin{cases} X, \ X \geq 0 \\ X + 10_{(2)}, X < 0 \end{cases} \qquad (1.2.3)$$

$$X_1 = +0{,}110011 \qquad [X_1]_{доп} = 0{,}110011$$

$$X_2 = -0{,}110011 \qquad [X_2]_{доп} = 1{,}001100$$

$$\begin{array}{r} + \qquad 1 \\ \hline 1{,}001101 \end{array}$$

Переход от дополнительного кода отрицательного числа к прямому коду производится следующим образом:

Заменяют все цифры числа, за исключением знакового на взаимно обратные. После чего добавляют единицу к младшему разряду.

$$[X]_{доп} = 1{,}001101$$

$$[X]_{пр} = 1{,}110010$$

$$\begin{array}{r} + \qquad 1 \\ \hline 1{,}110011 \end{array}$$

Обратный код чисел.

Обратным кодом отрицательной двоичной дроби называется её дополнение по модулю до двух без единицы младшего разряда.

Он получается по правилу: в знаковом разряде числа проставляется 1, а во всех остальных разрядах цифры заменяются на взаимно обратные.

Обратный код положительной двоичной дроби совпадает с прямым.

Аналитически:

$$[X]_{обр} = \begin{cases} X, \quad X \geq 0 \\ X + 10_{(2)} - 10_{(2)}^{-n}, \ X < 0 \end{cases} \qquad (1.2.4)$$

$$X_1 = +0{,}110011 \qquad [X_1]_{обр} = 0{,}110011$$

$$X_2 = -0{,}110011 \qquad [X_2]_{обр} = 1{,}001100$$

1.3. Алгебраическое сложение чисел

Алгебраическое сложение всегда производится либо в дополнительном, либо в обратном коде. Сумма, естественно, также получается в одном из этих кодов соответственно. Правильный знак суммы получается автоматически.

В случае возникновения единицы переноса из знакового разряда суммы, её нужно отбросить при производстве сложения в дополнительном коде и прибавить к младшему разряду суммы при сложении в обратном коде.

Необходимо учитывать, что в поразрядном сложении участвуют не две, а три цифры. Третьим слагаемым будет единица переноса.

Таблица сложения

Таблица 1.2

Текущий разряд первого слагаемого	Текущий разряд второго слагаемого	Цифра переноса из предыдущего разряда	Результат (текущий разряд и цифра переноса)
0	0	0	00
0	0	1	01
0	1	0	01
1	0	0	01
1	0	1	10
1	1	0	10
0	1	1	10
1	1	1	11

Сложение с фиксированной точкой

При алгебраическом сложении возможны 4 различных случая:

1. $X_1 > 0;\ X_2 > 0;\ X_1 + X_2 = X_3 > 0$

$X_1 = +0{,}1100$

$X_2 = +0{,}0010$

$[X_1]_{пр} = 0{,}1100;\quad [X_1]_{доп} = [X_1]_{пр} \qquad 0{,}1100$

$[X_2]_{пр} = 0{,}0010;\quad [X_2]_{доп} = [X_2]_{пр} \qquad \dfrac{0{,}0010}{0{,}1110}$

$X_3 = +0{,}1110$

2. $X_1 > 0;\ X_2 < 0;\ X_1 + X_2 = X_3 > 0$

$X_1 = +0{,}1100$

$X_2 = -0{,}0010$

$[X_1]_{пр} = 0{,}1100;\quad [X_1]_{доп} = [X_1]_{пр}$ *в доп.коде* *в обр.коде*

$[X_2]_{пр} = 1{,}0010;\quad [X_2]_{доп} = 1{,}1110 \qquad 0{,}1100 \qquad 0{,}1100$

$[X_2]_{обр} = 1{,}1101 \qquad \dfrac{1{,}1110}{1|\,0{,}1010} \qquad \dfrac{1{,}1101}{1|\,0{,}1001}$

$\dfrac{1}{0{,}1010}$

$X_3 = +0{,}1010$

3. $X_1 < 0;\ X_2 > 0;\ X_1 + X_2 = X_3 < 0$

$X_1 = -0{,}1100$

$X_2 = +0{,}0010$ *в доп. коде* *в обр.коде*

$[X_1]_{пр} = 1{,}1100;\quad [X_1]_{обр} = 1{,}0011 \qquad 1{,}0011 \qquad 1{,}0100$

$[X_1]_{доп} = 1{,}0100 \qquad \dfrac{0{,}0010}{1{,}0101} \qquad \dfrac{0{,}0010}{1{,}0110}$

$[X_2]_{пр} = 0{,}0010 = [X_2]_{обр} = [X_2]_{доп}$

$[X_3]_{обр} = 1{,}0101 \qquad [X_3]_{доп} = 1{,}0110$

$[X_3]_{пр} = 1{,}1010 \qquad [X_3]_{пр} = 1{,}1001$

$\dfrac{1}{1{,}1010}$

$X_3 = -0{,}1010$

4. $X_1 < 0;\ X_2 < 0;\ X_1 + X_2 = X_3 < 0$

$X_1 = -0{,}1100$

$X_2 = -0{,}0010$

		в доп. коде	в обр.коде
$[X_1]_{пр} = 1{,}1100;$	$[X_1]_{обр} = 1{,}0011$	1,0100	1,0011
	$[X_1]_{доп} = 1{,}0100$	0,1110	1,1101
		1|1,0010	1|1,0000
			1
			1,0001

$[X_2]_{пр} = 1{,}0010$ $\quad [X_2]_{обр} = 1{,}1101$

$[X_2]_{доп} = 1{,}1110$

$[X_3]_{обр} = 1{,}0001$ $\quad [X_3]_{доп} = 1{,}0010$

$[X_3]_{пр} = 1{,}1110$ $\quad [X_3]_{пр} = 1{,}1101$

1

1,1110

$X_3 = -0{,}1110$

При сложении любых положительных чисел, сумма которых > 1, результат их машинного суммирования будет иметь вид отрицательного числа, хотя никаких ошибок не было. В этом случае происходит переполнение разрядной сетки.

Необходимо, чтобы компьютер научился фиксировать такие ситуации. Для этого используются так называемые модифицированные коды. Модифицированные коды отличаются от обыкновенных тем, что в них знак числа изображаются двумя разрядами.

Знак + кодируется 00, а знак – кодируется 11. Комбинации 01 и 10 являются запрещенными.

Пусть имеются $[X_1]_{пр} = 0{,}10101$ и $[X_2]_{пр} = 1{,}10101$. Перевести их в модифицированный обратный код.

$$[X_1]_{обр}^{мод} = 00{,}10101$$

$$[X_2]_{обр}^{мод} = 11{,}01010$$

Пусть требуется сложить числа:

$X_1 = +0{,}1100$

$X_2 = +0{,}1010$

Переведем их в модифицированный дополнительный код и выполним требуемую операцию:

$$[X_1]_{доп}^{мод} = 00{,}1100 \qquad \begin{array}{r} 00{,}1100 \\ \underline{00{,}1010} \\ 01{,}0110 \end{array}$$

$$[X_2]_{доп}^{мод} = 00{,}1010$$

Произошло переполнение разрядной сетки (здесь мы предполагаем, что длина разрядной сетки равно 4).

Пусть теперь необходимо сложить числа:

$X_1 = +0{,}1100$

$X_2 = -0{,}1010$

Имеем:

$$[X_1]_{доп}^{мод} = 00{,}1100 \qquad \begin{array}{r} 00{,}1100 \\ \underline{11{,}1110} \\ 1\,|\,00{,}1010 \end{array}$$

$$[X_2]_{доп}^{мод} = 11{,}1110$$

$$X_3 = +0{,}1010$$

т.е. знак результата получился автоматически.

Сложение чисел с плавающей точкой.

Правило:

Уравниваются порядки слагаемых. Меньший порядок увеличивается до большего, а мантисса этого числа сдвигается вправо на соответствующее количество разрядов.

1) Производится преобразование мантисс в один из модифицированных кодов, дополнительный или обратный.
2) Производится сложение мантисс по правилам сложения чисел с фиксированной точкой.

3) Полученная сумма мантисс переводится в прямой код. К ней приписывается общий порядок слагаемых, производится нормализация суммы (в случае необходимости) и округление мантиссы.

Округление производится следующим образом: пусть используется n разрядов. В результате сложения мантисс получилось, предположим, $n+k$ разрядов. Прежде чем отбросить последние k разрядов, к $n+1$ разряду добавляется 1 (напомним, что речь идет о двоичном сложении), затем разряды с $n+1$ до $n+k$ отбрасываются.

Пример:

$$\begin{array}{r} 0{,}01011 \\ 1 \\ \hline 0{,}0110\,|\,0 \end{array} \qquad \begin{array}{r} 0{,}01010 \\ 1 \\ \hline 0{,}0101\,|\,0 \end{array}$$

Для лучшего уяснения рассмотрим операцию сложения с плавающей точкой сначала для десятичных чисел:

$$X_1 = 0{,}125 \cdot 10^3$$

$$X_2 = 0{,}625 \cdot 10^2 = 0{,}0625 \cdot 10^3$$

$$\begin{array}{r} 0{,}125 \\ +\ 0{,}0625 \\ \hline 0{,}1875 \end{array}$$

$$0{,}1875 \cdot 10^3 \cong 0{,}188 \cdot 10^3$$

Пример нормализации:

$$0{,}0256 \cdot 10^3 = 0{,}256 \cdot 10^2$$

$$\begin{array}{l} 0{,}625 \cdot 10^3 \\ + \\ \underline{0{,}600 \cdot 10^3} \\ 1{,}225 \cdot 10^3 \end{array}$$

$$1{,}225 \cdot 10^3 = 0{,}1225 \cdot 10^4$$

Теперь рассмотрим в двоичной системе счисления:

$$[X_1]_{пр} = 0\ 1100\ 0\ 0100 = 0\ 001100\ 0\ 110$$

$$[X_2]_{пр} = 0\ 0010\ 0\ 110$$

$$[m_1]_{доп}^{мод} = 00\ 001100 \quad (m - мантисса)$$

$$[m_2]_{доп}^{мод} = 00\ 001000$$

$$[m_1]_{доп}^{мод} + [m_2]_{доп}^{мод} = \begin{array}{r} 00\ 001100 \\ \underline{00\ 001000} \\ 00\ 010100 \end{array}$$

$$[m_3]_{доп}^{мод} = 00\ 010100 = [m_3]_{пр}^{мод}$$

нормализация:

$$[X_3]_{пр} = 0\ 010100\ 0\ 0110 = 0\ 10100\ 0\ 0101$$

округление:

$$\begin{array}{r} 0{,}10100 \\ \underline{1} \\ 0{,}10101 \end{array}$$

$$[X_3]_{пр} = 0\ 1010\ 0\ 0101$$

1.4. Умножение и деление чисел

Умножение чисел с фиксированной точкой производится в два этапа:

1) определяется знак произведения при помощи сложения по модулю 2 (mod 2). Суммирование по mod 2 производится согласно правилам двоичной арифметики, но без учета единицы переноса. Правомерность такого определения знака доказывается с помощью таблицы:

Таблица 1.3

$(+)\cdot(+)=(+)$	$0\oplus 0=0$
$(+)\cdot(-)=(-)$	$0\oplus 1=1$
$(-)\cdot(+)=(-)$	$1\oplus 0=1$
$(-)\cdot(-)=(+)$	$1\oplus 1=0$

2) производится перемножение модулей сомножителей, затем в случае необходимости округление и к полученному результату приписывается знак.

Умножение производится по обычным правилам арифметики согласно двоичной таблице умножения. Пример:

$$[X_1]_{пр} = 0{,}1010$$
$$[X_2]_{пр} = 1{,}1101$$

1. определяем знак произведения

$$0 \oplus 1 = 1$$

2. выполняем умножение

$$\begin{array}{r} 0{,}1010 \\ \underline{0{,}1101} \\ 1010 \\ 0000 \\ 1010 \\ \underline{1010} \\ 0{,}10000010 \end{array}$$

$$[X_3]_{пр} = 1{,}1000$$

Умножение чисел с плавающей точкой

$$X_1 = m_1 \cdot 10^{P_1}$$
$$X_2 = m_2 \cdot 10^{P_2}$$
$$X_3 = X_1 \cdot X_2 = (m_1 \cdot m_2) \cdot 10^{P_1 + P_2}$$

Отсюда 4 этапа для умножения чисел с плавающей точкой:

1. определение знака произведения путем сложения знаковых разрядов по mod 2.
2. перемножение модулей мантисс сомножителей по правилам умножения чисел с фиксированной точкой.
3. сложение порядков по правилам алгебраического сложения чисел с фиксированной точкой.

4. округление полученного результата и приписывание соответствующего знака результату.

Пример:

$[X_1]_{пр} = 1\ 1010\ \ 0\ 101$

$[X_2]_{пр} = 0\ 1011\ 1\ 011$

1) $1 \oplus 0 = 1$

2)

$$\begin{array}{r} 0{,}1010 \\ \underline{0{,}1011} \\ 1010 \\ 1010 \\ 0000 \\ \underline{1010} \\ 0{,}01101110 \end{array}$$

3) $[P_1]_{пр}^{мод} = 00{,}101$

$[P_2]_{пр}^{мод} = 11{,}011$

$[P_2]_{доп}^{мод} = 11{,}101$

$$\begin{array}{r} 00\ 101 \\ \underline{11\ 101} \\ 1|\ 00\ 010 \end{array}$$

$[P_3]_{пр} = 0\ \ 010$

4) $[X_3]_{пр} = 1{,}01101110\ \ 0\ \ 010$

округление:

$[X_3]_{пр} = 1{,}0111\ \ 0\ \ 010$

Деление чисел с фиксированной точкой

Деление чисел с фиксированной точкой осуществляется в два этапа:

1. определение знака
2. деление

$X_1 = +0{,}101$

$X_2 = -0{,}110$

$[X_1]_{пр} = 0{,}101$

$[X_2]_{пр} = 1{,}110$

1) $0 \oplus 1 = 1$

2)

$$
\begin{array}{r|l}
0{,}1010 & 0{,}110 \\ \cline{2-2}
110 & 0{,}1100 \cong 0{,}111 \\ \cline{1-1}
1000 & \\
110 & \\ \cline{1-1}
1000 & \\
110 & \\ \cline{1-1}
10 &
\end{array}
$$

$[X_3]_{пр} = 1{,}111$

Деление чисел с плавающей точкой

$X_1 = m_1 \cdot 10^{P_1}$

$X_2 = m_2 \cdot 10^{P_2}$

$X_3 = X_1 : X_2 = (m_1 : m_2) \cdot 10^{P_1 - P_2}$

Правила деления:

1) определение знака частного посредством сложения знаковых цифр мантисс операндов по mod 2.
2) деление модуля мантиссы делимого на модуль мантиссы делителя по правилам деления чисел с фиксированной точкой.

3) нахождение порядка частного путем вычитания из порядка делимого порядка делителя с учетом их знаков.
4) нормализация результата, если это необходимо.

$X_1 = 0{,}101\ 0\ 110$

$X_2 = 1{,}110\ 0\ 011$

1) $0 \oplus 1 = 1$

$\boxed{2)}$

$$
\begin{array}{r|l}
0{,}1010 & 0{,}110 \\ \cline{2-2}
110 & 0{,}1100 \cong 0{,}111 \\ \cline{1-1}
1000 & \\
110 & \\ \cline{1-1}
1000 & \\
110 & \\ \cline{1-1}
10 &
\end{array}
$$

$|m_3| = 0{,}111$

3) $[P_1]_{доп}^{мод} = 00{,}110$

$[P_2]_{доп}^{мод} = 11{,}101$

$$
\begin{array}{r}
00{,}110 \\
11{,}101 \\ \hline
1\,|\,00{,}011
\end{array}
$$

$[P_3]_{пр} = 0{,}011$

$[X_3]_{пр} = 1\ 111\ 0\ 011$

1.5. Логические основы вычислительной техники

Как известно, логика – это наука о законах и формах мышления. Математическая логика занимается изучением возможностей применения формальных математических методов для решения логических задач. В вычислительной технике используется главным образом начальный раздел математической логики – исчисление высказываний или алгебра логики. Иногда её ещё называют

булевой алгеброй по имени английского математика Джорджа Буля, который ещё в XIX веке разработал основные положения исчисления высказываний.

Однако сам Буль не видел путей практической реализации своих исследований. Но уже в 40-х годах XX века алгебра логики нашла широкое применение при проектировании и синтезе сложных переключательных схем в автоматических телефонных станциях, а затем и в вычислительной технике.

Начальным понятием алгебры логики является высказывание. Под высказыванием понимается любое утверждение, которое оценивается только с точки зрения его истинности или ложности. То есть высказывание может быть либо истинным, либо ложным.

В соответствии с такой двоичной природой высказываний условились называть их логическими двоичными переменными. Любое высказывание можно обозначить символом x и считать, что x=1, если высказывание истинно, x=0, если высказывание ложно.

Таким образом, логическая (булевая) переменная – это такая величина x, которая может принимать только два значения (0 или 1):

x={0, 1}

Высказывание абсолютно истинно, если соответствующая ей логическая величина принимает значение x=1 при любых условиях.

Высказывание абсолютно ложно, если соответствующая ей логическая переменная принимает значение 0 при любых условиях.

Сложным называется высказывание, значение истинности которого зависит от значения истинности других высказываний. Следовательно, любое сложное высказывание можно считать функцией некоторых двоичных аргументов – простых высказываний, входящих в его состав.

Логическая функция $f(x_1, x_2, \ldots, x_n)$ (функция алгебры логики) – функция принимающая значение 0 или 1 на наборе логических переменных $x_1, x_2, \ldots, x_n$. Причем $x_1, x_2, \ldots, x_n$ могут быть в свою очередь логическими функциями. Отсюда

следует, что можно построить функцию любой наперед заданной сложности, пользуясь принципом суперпозиции.

Ограниченный набор функций, обеспечивающий на основе суперпозиции построение функции любой наперед заданной сложности, называется функционально полной системой функций.

Из множества функционально полных систем рассмотрим один, получивший наибольшее распространение и вследствие этого названного основным.

В его состав входят всего 3 функции:

функция отрицания (функция НЕ, инверсия).

Отрицанием называется такая функция y, которая истинна тогда и только тогда, когда ложен аргумент x и наоборот, y ложна тогда и только тогда, когда аргумент x истинен.

В алгебре логики любые функции удобно изображать в виде таблицы соответствия всех возможных комбинаций значений истинности входных переменных (аргументов) и выходной логической функции.

Таблица 1.4

аргумент x	0	1
значение функции y	1	0

При помощи логико-математической символики функция отрицания записывается в виде:

$y = \overline{x}, \rceil x$, NOT

Эта функция обладает свойством $\overline{\overline{x}} = x$

Конъюнкция (логическое умножение, функция "И")

Конъюнкцией n высказываний называется такое сложное высказывание y, которое истинно тогда и только тогда, когда истинны все исходные высказывания; во всех остальных случаях y ложно. Таблица истинности для функции двух переменных:

Таблица 1.5

x_1	0	0	1	1
x_2	0	1	0	1
y	0	0	0	1

Обозначения:

$y = x_1 \cdot x_2$

$y = x_1 \& x_2$

AND

Конъюнкция обладает 4-мя очевидными свойствами:

$x \cdot x \cdot \cdots \cdot x = x$

результат многократного логического умножения аргумента x на себя равен исходному.

$x \cdot 1 = x$

логическое произведение двух аргументов, один из которых абсолютно истинен, равно второму аргументу.

$x \cdot 0 = 0$

логическое произведение двух аргументов, один из которых абсолютно ложен, равно 0.

$x \cdot \overline{x} = 0$

логическое произведение некоторого аргумента и его отрицания равно 0.

дизъюнкция (логическое сложение, функция "ИЛИ")

Дизъюнкцией n высказываний называется такое сложное высказывание y, которое ложно тогда и только тогда, когда ложны все исходные высказывания. Если же истинно хотя бы одно исходное высказывание, функция y тоже истинно.

Таблица истинности для функции двух переменных:

Таблица 1.6

x_1	0	0	1	1
x_2	0	1	0	1
y	0	1	1	1

Обозначения:

$y = x_1 + x_2$

$y = x_1 \vee x_2$

OR

Дизъюнкция обладает 4-мя очевидными свойствами:

$x + x + \cdots + x = x$

результат многократного логического сложения аргумента *x* на себя равен исходному.

$x + 1 = 1$

логическое сложение двух аргументов, один из которых абсолютно истинен, равно 1.

$x + 0 = x$

логическое сложение двух аргументов, один из которых абсолютно ложен, равно второму аргументу.

$x + \bar{x} = 1$

логическое сложение некоторого аргумента и его отрицания равно 1.

Рассмотренные выше функции можно элементарным образом реализовать с помощью электрических схем.

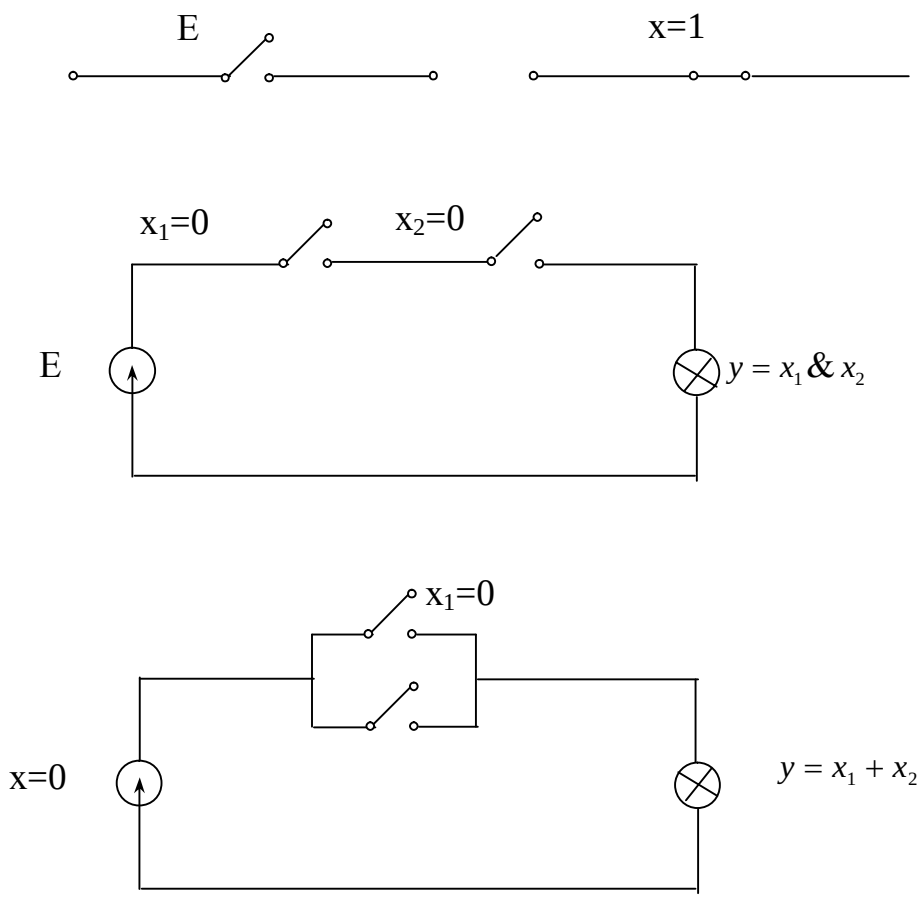


Рис. 1.1.

Абсолютно истинную функцию называют константа 1, абсолютно ложную – константой 0.

Функция $f(x) = x$, т.е. функция, повторяющая значение аргумента, называется тождественной.

Существуют и другие функции от двух логических переменных. Их всего 16. См. таблицу 1.7.

Функция $f_9 = \overline{x_1 + x_2}$ называется функцией Пирса. Это функция, которая истинна тогда и только тогда, когда оба x_1 и x_2 ложны.

Функция $f_{15} = \overline{x_1 \cdot x_2}$ называется функцией Шеффера. Это функция, которая ложна тогда и только тогда, когда оба x_1 и x_2 истинны.

$f_9 = x_1 \downarrow x_2$ - стрелка Пирса

$f_{15} = x_1 | x_2$ - штрих Шеффера

Таблица 1.7

Функция	Значения x_1 и x_2				Аналитическое представление функции
	00	01	10	11	
f_1	0	0	0	0	0
f_2	0	0	0	1	$x_1 \cdot x_2$
f_3	0	0	1	0	$x_1 \cdot \overline{x_2}$
f_4	0	0	1	1	x_1
f_5	0	1	0	0	$\overline{x_1} \cdot x_2$
f_6	0	1	0	1	$\overline{x_1} \cdot x_2 + x_1 \cdot x_2 = x_2$
f_7	0	1	1	0	$\overline{x_1} \cdot x_2 + x_1 \cdot \overline{x_2} = x_1 \oplus x_2$
f_8	0	1	1	1	$x_1 + x_2$
f_9	1	0	0	0	$\overline{x_1 + x_2}$
f_{10}	1	0	0	1	$x_1 \equiv x_2$
f_{11}	1	0	1	0	$\overline{x_1} \cdot \overline{x_2} + x_1 \cdot \overline{x_2}$
f_{12}	1	0	1	1	$\overline{x_1} \cdot \overline{x_2} + x_1 \cdot \overline{x_2} + x_1 \cdot x_2$
f_{13}	1	1	0	0	$\overline{x_1} \cdot \overline{x_2} + \overline{x_1} \cdot x_2 = \overline{x_1}$
f_{14}	1	1	0	1	$x_1 \to x_2$
f_{15}	1	1	1	0	$\overline{x_1 \cdot x_2}$
f_{16}	1	1	1	1	1

1.5.1. Законы алгебры логики

В алгебре логики имеются 4 основных закона, регламентирующие порядок выполнения операций НЕ, И, ИЛИ в любом логическом выражении:

переместительный (коммутативный);

сочетательный (ассоциативный);

распределительный (дистрибутивный);

инверсии (правило Де Моргана).

Рассмотрим эти законы:

Переместительный закон

Справедлив как для операции И, так и для операции ИЛИ.

1. $x_1 + x_2 = x_2 + x_1$
2. $x_1 \cdot x_2 = x_2 \cdot x_1$

Сочетательный закон

Также справедлив как для операции И, так и для операции ИЛИ.

1. $x_1 + x_2 + x_3 = x_1 + (x_2 + x_3) = x_2 + (x_1 + x_3) = x_3 + (x_1 + x_2)$
2. $x_1 \cdot x_2 \cdot x_3 = x_1 \cdot (x_2 \cdot x_3) = x_2 \cdot (x_1 \cdot x_3) = x_3 \cdot (x_1 \cdot x_2)$

Распределительный закон

Распределительный закон первого рода

Произведение суммы нескольких аргументов на какую-либо переменную равно сумме произведений.

$$(x_1 + x_2) \cdot x_3 = x_1 \cdot x_3 + x_2 \cdot x_3$$

Распределительный закон второго рода

Сумма произведения нескольких аргументов и какой-либо переменной равна произведению суммы каждого сомножителя и этой переменной.

$$x_2 \cdot x_3 + x_1 = (x_1 + x_2) \cdot (x_1 + x_3)$$

Этот закон выглядит несколько парадоксальным. Докажем его:

$$(x_1 + x_2)\cdot(x_1 + x_3) = x_1 \cdot x_1 + x_2 \cdot x_1 + x_1 \cdot x_3 + x_2 \cdot x_3 =$$
$$= x_1 + x_1 \cdot x_2 + x_1 \cdot x_3 + x_2 \cdot x_3 = x_1 \cdot (1 + x_2 + x_3) + x_2 \cdot x_3 =$$
$$= x_1 \cdot 1 + x_2 \cdot x_3 = x_1 + x_2 \cdot x_3 = x_2 \cdot x_3 + x_1$$

Законы инверсии

Отрицание логической суммы нескольких аргументов равно логическому произведению отрицаний этих же аргументов.

$$\overline{x_1 + x_2 + \ldots + x_n} = \overline{x_1} \cdot \overline{x_1} \cdot \ldots \cdot \overline{x_n}$$

Отрицание логического произведения нескольких аргументов равно логической сумме отрицаний этих же аргументов.

$$\overline{x_1 \cdot x_2 \cdot \ldots \cdot x_n} = \overline{x_1} + \overline{x_1} + \ldots + \overline{x_n}$$

Следствия из законов алгебры логики

Логическим выражением называют всякую совокупность логических переменных и знаков операций, указывающих какие действия над этими переменными нужно выполнить. Будем считать отрицание логическим действием первой ступени (старшей логической операцией), конъюнкцию – логическим действием второй ступени, дизъюнкцию – логическим действием третьей ступени.

Тогда можно определить правило выполнения логических операций:

Если в логическом выражении встречаются только операции одной и той же ступени, то их нужно выполнять в том порядке, в каком они записаны (т.е. слева направо).

Если же в логическом выражении встречаются действия различных ступеней, то сначала выполняются действия первой ступени, затем второй и только после этого третьей.

Всякое отклонение от этого порядка должно регламентироваться скобками.

Пример. Вычислить значение логического выражения

$$y = x_1 \cdot \overline{x_2} + x_3 \cdot \overline{\overline{x_4} + x_5}$$

при $x_1 = 1,\ x_2 = 0,\ x_3 = 1,\ x_4 = 0,\ x_5 = 1$

$$y = 1 \cdot 1 + 1 \cdot \overline{1 + 1} = 1 + 1 \cdot 0 = 1 + 0 = 1$$

<u>Контрольные вопросы:</u>

1. Формы представления информации.
2. Что такое система счисления?
3. В чем различие между позиционной и непозиционной системами счисления?
4. Перечислите наиболее распространенные системы счисления.
5. Правила перевода чисел из одной системы счисления в другую.
6. Формы представления чисел в памяти компьютера.
7. Дополнительный и обратный коды двоичных чисел.
8. Алгебраическое сложение двоичных чисел.
9. Умножение двоичных чисел.
10. Деление двоичных чисел.
11. Логические переменные. Логические функции.
12. Основная система логических функций ("И","ИЛИ", "НЕ").
13. Законы алгебры логики.
14. Правило выполнения логических операций.

Глава II

Технические средства вычислительной техники

2.1. История развития средств вычислительной техники

Многие математики и механики прошлого пытались сконструировать различные вычислительные устройства, приборы и машины для облегчения вычислений. Во многих странах с древнейших времен были известны счеты.

В 1617 году Джон Непер описал специальные палочки для быстрого умножения и деления на счетах.

В 1642 году Блез Паскаль изобрел суммирующую машину.

В 1694 году Г. Лейбниц создал машину, которая могла не только складывать, но и умножать.

Идею построения вычислительной машины, похожей на современную, впервые высказал ещё в XIX веке английский математик Чарльз Бэббидж. В 1833 году им был предложен проект аналитической машины, которая должна была выполнять элементарные математические операции, запоминать исходные, промежуточные и окончательные результаты. Его машина состояла из "склада" для хранения чисел и "мельницы" для выполнения действий над числами. Из–за невысокого уровня развития тогдашней техники, Ч. Бэббидж не смог собрать эту машину.

Хотя идеи Ч. Бэббиджа и не были реализованы, для его машины была создана первая в мире программа. В 1843 году Ада Лавлейс, дочь английского поэта Д. Байрона, опубликовала статью, в которой были заложены основы современного программирования. Ею же была составлена программа для вычисления так называемых чисел Фибоначчи.

В 1937 г. американский ученый Г. Айкен заново разработал принципы построения цифровых вычислительных машин с автоматическим ходом вычислений, не зная о том, что это давно было сделано Ч. Бэббиджем. В 1944 г. вступила в строй первая релейно-механическая цифровая вычислительная машина "МАРК – I". Постройка следующей машины "МАРК – II" на реле улучшенных конструкций показала, что надежность релейных машин крайне низка.

В начале 40-х годов Клод Шеннон предложил для синтеза релейных схем использовать аппарат математической логики (булевой алгебры). Это дало возможность формальным способом строить экономичные схемы. Вследствие недостатков электромеханических реле конструкторы решают использовать электронные реле – триггеры. Это во много раз повысило надежность вычислительных машин.

Первая электронно-цифровая вычислительная машина была разработана в США в 1946 году и называлась "ЭНИАК". Она содержала 19000 электронных ламп. Скорость 5000 сложений в секунду.

Этот момент явился точкой отсчета пути, по которому пошло развитие вычислительной техники.

Развитие вычислительной техники определялось развитием электроники, появлением новых элементов и принципов их действия, т.е. развитием элементной базы. В настоящее время можно выделить 4 поколения компьютеров. Причем компьютеры четвертого поколения отличаются от предшествующих не только элементной базой, но и структурой построения, а также способом общения с человеком и эта тенденция сохранится и в будущем.

Под поколением компьютеров понимаются все типы и модели компьютеров, разработанные, хотя и различными конструкторскими коллективами, но построенные на одних и тех же научных принципах и элементной базе. Появление каждого нового поколения определялось тем, что появлялись новые электронные элементы, технология изготовления которых принципиально отличалась от предыдущего поколения.

Первое поколение (1946 – середина 50-х годов).

Элементная база – электронные лампы. В ЭВМ "ЭНИАК" было 19000 ламп, из которых ежемесячно заменяли 5000.

В Советском Союзе первая вычислительная машина была создана в 1951 году и называлась МЭСМ.

Второе поколение (середина 50-х – середина 60-х годов)

Элементная база – полупроводниковые приборы (транзисторы, диоды). Значительно улучшились характеристики компьютеров (быстродействие, надежность, габариты, условия эксплуатации).

Третье поколение (середина 60-х – середина 70-х годов)

В качестве элементной базы используются интегральные схемы (ИС). ИС выполняет те же функции, что и аналогичная схема на полупроводниках, но при этом существенно меньше по размерам, увеличилось надежность и быстродействие. Более точно, ИС – это электронная схема, реализующая определенную функцию, технически выполненная на одном кристалле кремния площадью всего несколько квадратных миллиметров.

Первый компьютер на ИС – широко известная IBM/360.

Четвертое поколение (середина 70-х – по настоящее время)

Компьютеры создаются на основе БИС (больших интегральных схем) со степенью интеграции от сотен тысяч до нескольких миллионов элементов на одном кристалле.

В виде БИС удалось создать процессор компьютера, который получил название микропроцессор (МП). МП повлияли на структуру и конструкцию компьютеров. Размеры компьютеров значительно уменьшились, появились персональные компьютеры (ПК).

2.2. Логические элементы вычислительной техники

Логическими элементами называются такие электронные устройства, в которых между входными и выходными сигналами реализуются те или иные логические операции. Логическим элементом называется устройство, реализующее только одну булеву функцию.

Схема отрицания (элемент НЕ, инвертор)

Как следует из определения функции НЕ, единичный сигнал на выходе схемы отрицания должен появиться при наличии на её входе нулевого сигнала и наоборот, на выходе схемы должен появиться нулевой сигнал, если на входе действует сигнал, соответствующий логической единице.

Функциональная схема имеет вид:

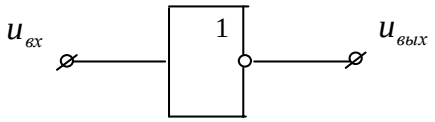


Рис. 2.1. Функциональная схема элемента НЕ

Слева вход, справа выход. На выходной стороне прямоугольника проставляется ○ означающий инверсию. На практике чаще всего элемент НЕ реализуется при помощи однокаскадного усилителя постоянного тока, собранного на транзисторе. Схема электрическая принципиальная имеет вид:

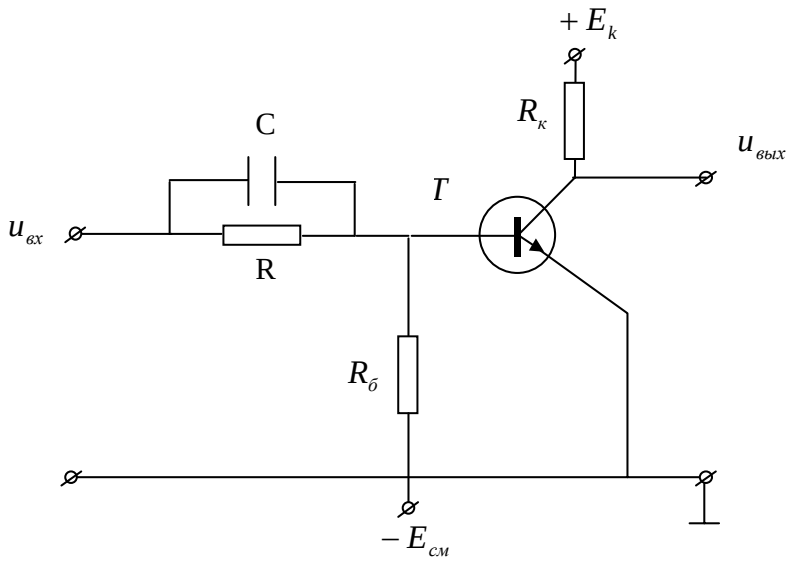


Рис. 2.2. Схема электрическая принципиальная

При этом используется свойство любого линейного усилительного каскада изменять у входного переменного сигнала фазу на 180^0. В логических схемах все электронные приборы (транзистор в нашем случае) работают в ключевом режиме. При поступлении на вход схемы низкого уровня напряжения, транзистор (n-p-n типа) остается закрытым и на выходе будет $u_{вых} = +E_k$ означающий "1". Если на вход поступает высокий уровень сигнала ("1"), то транзистор откроется и на выходе будет низкий уровень напряжения $u_{вых} = E_k - I_k R_k$, означающий логический "0".

О том, как работает транзистор и другие элементы этой схемы вы сможете узнать из соответствующих курсов физики, ТОЭ и электроники. Здесь за неимением места мы эти подробности опускаем. Для целей нашего курса достаточно уяснить, что между входным и выходным напряжениями схемы существует такая взаимосвязь, которая и реализует логическую функцию НЕ.

Схема совпадения (схема "И")

Единичный сигнал на выходе схемы должен появляться только тогда, когда на всех входах "1".

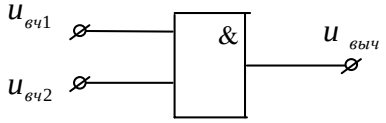


Рис. 2.3. Функциональная схема

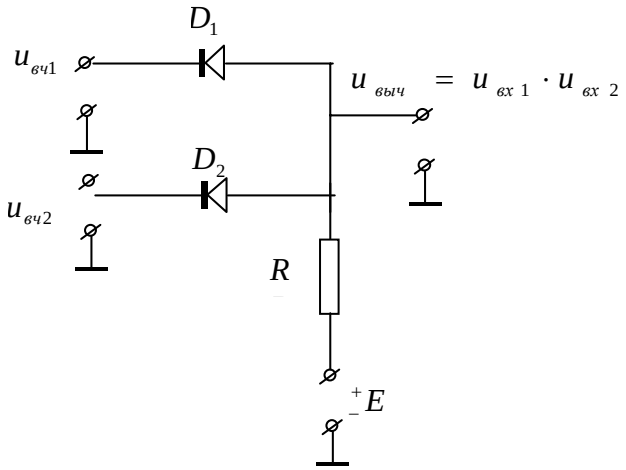


Рис. 2.4. Схема электрическая принципиальная

Электрическая схема реализована на полупроводниковых диодах, работающих в ключевом режиме (т.е. сопротивление открытого диода $r_{д.о.} \to 0$, а сопротивление закрытого диода $r_{д.з.} \to \infty$). Резистор R выбирается из условия:

$$r_{д.о.} << R << r_{д.з.}$$

Выходное напряжение снимается с двух параллельно включенных диодов, сопротивления которых совместно с резистором R образует обыкновенный делитель напряжения. Возможны 3 случая:

1) Оба входных напряжения низкие

$$u_{вх1} = u_{вх2} = 0$$

и следовательно оба диода открыты. Тогда общее сопротивление диодов:

$$r = r_{д.о1} \| r_{д.о1} = 0$$
$$R >> r$$

Практически все напряжение E падает на R, а на выходе низкий потенциал $u_{вых} = 0$.

2) Один $u_{вх}$ низкий, а другой высокий. Пусть $u_{вх1}$ низкий. Следовательно:

$$r_{д.о1} = 0, \quad r_{д.о2} \to \infty$$
$$r = r_{д.о1} \| r_{д.о2} = 0$$
$$R >> r$$

Все напряжение E падает на R. На выходе низкий потенциал.

3) Оба входных сигнала высокие. Тогда:

$$r_{д.з1} \to \infty, \quad r_{д.з2} \to \infty$$
$$r \to \infty$$
$$R << r$$

Теперь все напряжение будет падать на r и на выходе будет высокий потенциал $u_{вых} = E$.

Схема собирания (схема "ИЛИ")

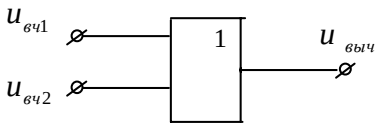


Рис. 2.5. Функциональная схема

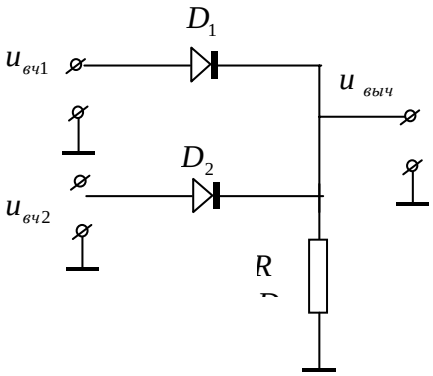


Рис. 2.6. Схема электрическая принципиальная

$$r_{д.о.} << R << r_{д.з.}$$

Анализ работы схемы производится аналогично предыдущей схеме.

Схема И – НЕ

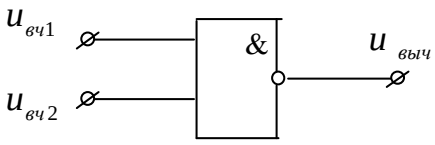


Рис. 2.7

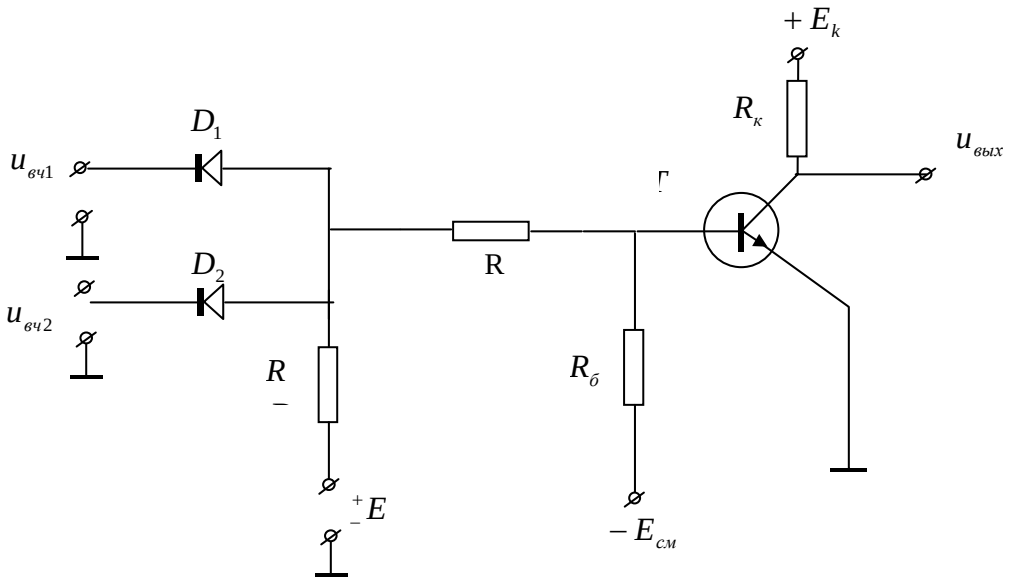


Рис. 2.8 Схема электрическая принципиальная

Схема ИЛИ – НЕ

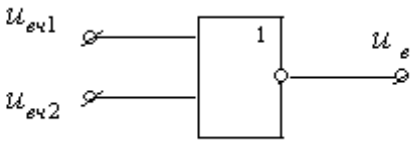


Рис. 2.9

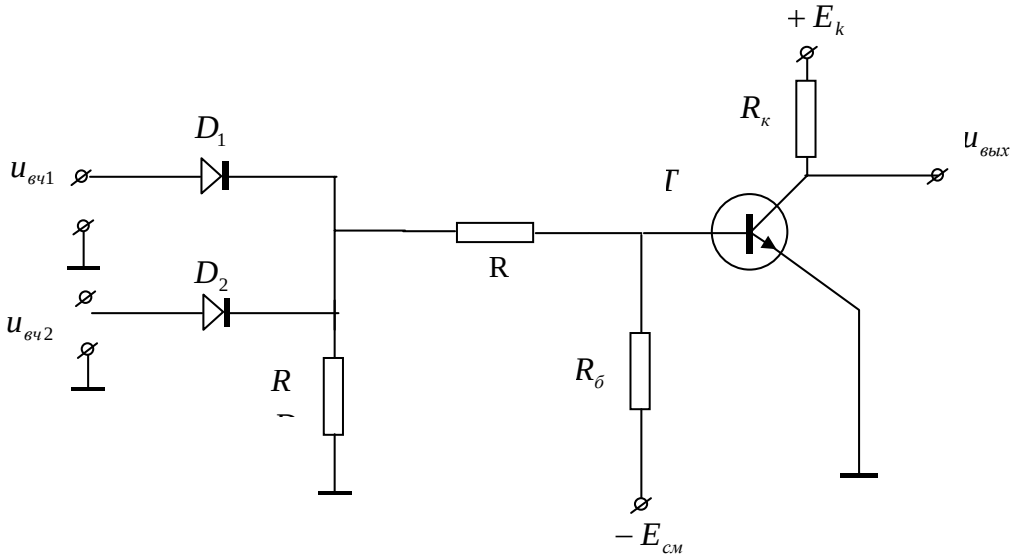


Рис. 2.10. Схема электрическая принципиальная

2.3. Функциональные элементы

Функциональным элементом называется элемент, реализующий несколько булевых функций.

Рассмотрим только один функциональный элемент – триггер. В переводе с английского языка слово триггер переводится как защелка. Триггером называется элементарный автомат, имеющий два устойчивых состояния. Триггер используется для хранения двоичной информации. Рассмотрим структурную схему триггера на элементах ИЛИ – НЕ:

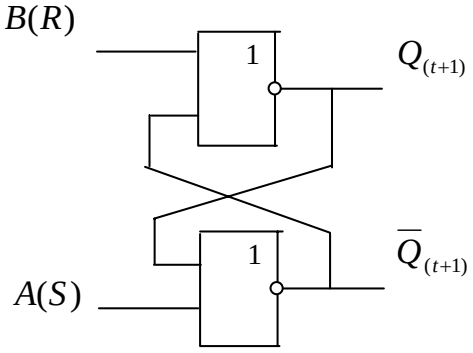


Рис. 2.11

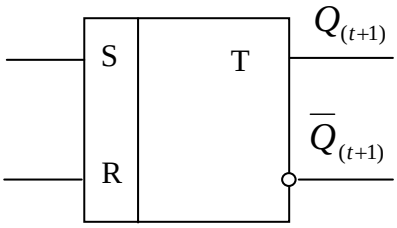


Рис. 2.12

Таблица переходов RS – триггера

Таблица 2.1

S_t	R_t	Q_t	Q_{t+1}	$\overline{Q}_{t+1}$
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	-	-
1	1	1	-	-

Принято считать, что триггер перешел в единичное состояние, если на выходе Q появился потенциальный сигнал низкого уровня. Соответственно за переход в нулевое состояние триггера принято считать появление на выходе Q сигнала высокого уровня. Для передачи единичного сигнала из триггера необ ходимо изменить его состояние с 1 на 0 и получить на выходе Q изменение потенциального сигнала с низкого на высокий.

2.4. Типовые функциональные схемы

Типовые функциональные схемы строятся на основе логических и функциональных элементов. К ним относятся:

Регистры.

Регистром называется устройство, предназначенное для временного хранения двоичной информации. Каждый разряд регистра используется для ввода, вывода и хранения одного разряда двоичного числа. Рассмотрим, например,

схему 3-х разрядного регистра:

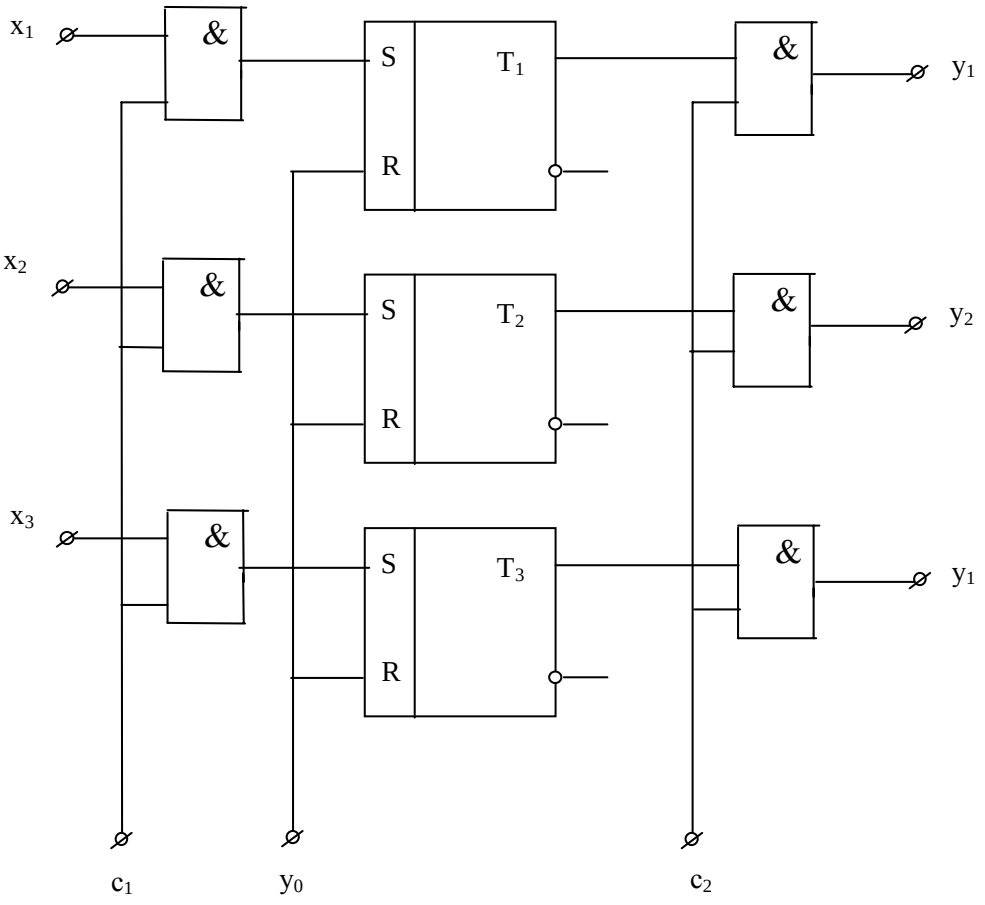


Рис. 2.13. Схема 3-х разрядного регистра

В начале работы все триггеры устанавливаются в 0 сигналом y_0. Конъюнкторы на входе пропустят двоичную информацию на входы S триггеров только при наличии синхроимпульса c_1. Для передачи двоичного числа из регистра на входы конъюнкторов необходимо на R входы триггеров подать импульс y_0 который перебросит триггеры из 1→0. При переходе из 1→0 на входы конъюнкторов поступят сигналы передачи 1. При этом они пройдут в другие устройства только при наличии синхроимпульса c_2.

Счетчики.

Назначением счетчиков является:

- подсчет числа импульсов, поступающих на вход счетчика
- временное хранение каждого состояния

- преобразование непрерывного сигнала, заданного последовательностью импульсов в двоичный код.

Счетчики бывают прямого счета (суммирующие), вычитающие, реверсивные.

Дешифраторы

Дешифратором называется устройство, позволяющее преобразовать сочетание входных сигналов (входной код) в сигнал только на одном из выходов.

С помощью дешифратора двоичный код числа преобразуется в сигнал, управляющий выбором соответствующего блока, схемы или устройства. Схема дешифратора представляет собой матрицу логических элементов и содержит в общем случае n входов и 2^n выходов. Каждой комбинации входных сигналов соответствует появление сигнала только на одном из выходов.

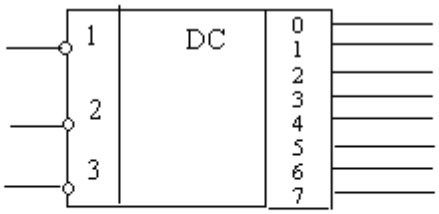


Рис. 2.14. Функциональная схема дешифратора

2.5. Принципы работы компьютера

Компьютер компьютеру рознь!
Один компьютер бывалоча все чего-то вычисляет, рисует, играет и ещё невесть что делает… А другой – бдит и студентов мучает!

Из рассказов бывалого студента,
прошедшего компьютерное тестирование

2.5.1. Что такое "компьютер"?

Слово компьютер произошло от английского compute – вычислять. Ранее в нашей стране применялся термин "электронно-вычислительная машина" (ЭВМ), что, на мой взгляд, более точно отражает суть дела. Дело в том, что в основе любого современного компьютера лежат электронные устройства, о которых мы говорили выше.

Автоматизация работ с информацией (данными) имеет свои особенности и отличия от автоматизации других видов работ. Для этого класса задач используют особые устройства, большинство из которых являются электронными приборами и устройствами. Совокупность устройств, предназначенных для автоматической или автоматизированной обработки данных, называют *вычислительной техникой*.

Конкретный набор взаимодействующих между собой устройств и программ называют *вычислительной системой*. Центральным устройством любой вычислительной системы является *компьютер*.

Итак, компьютер – это электронный прибор, предназначенный для автоматизации создания, хранения, обработки и передачи данных.

2.5.2. Простейшая структура компьютера

Принципы работы всех компьютеров одинаковы и носят название *принципов фон – Неймана*.

Общность принципов работы компьютеров и определяет общность структур компьютеров. Рассмотрим простейшую структуру компьютера лишь с целью уяснения основных принципов работы компьютера.

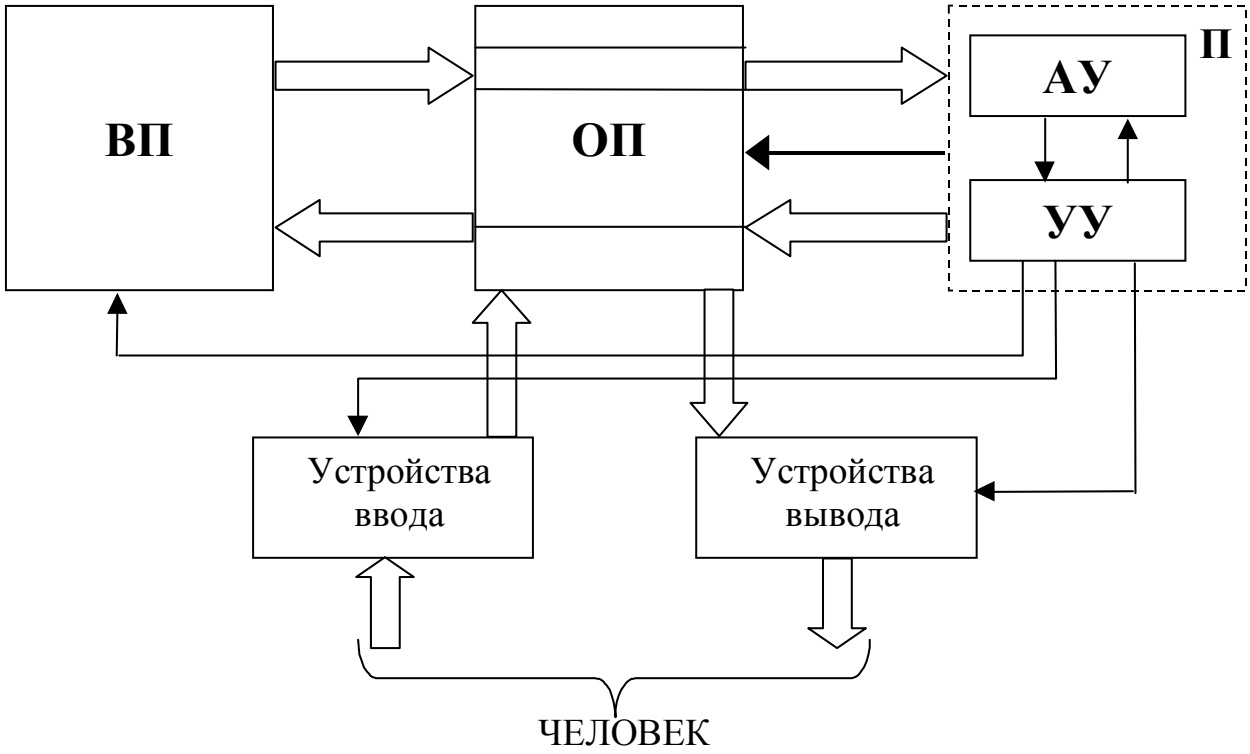


Рис. 2.15.

Здесь двойные стрелки означают передачу информации (данных), одинарные стрелки означают управляющие сигналы процессора. На рисунке процессор обозначен буквой **П**. Процессор – это устройство, которое управляет всеми остальными устройствами компьютера и выполняет арифметические и логические операции над данными, находящимися в памяти.

Ту часть процессора, которая предназначена для непосредственного вы

полнения операций называют *арифметическим устройством* (**АУ**). Остальные функции процессора, как уже было сказано, – это управление работой различных устройств. Поэтому часть процессора, выполняющий эти функции, называют *устройством управления* (**УУ**).

Память компьютера предназначена для хранения информации и команд программы. Информация, хранящаяся в памяти компьютера, представляет собой закодированные с помощью всего двух цифр 0 и 1 числа, буквы, символы, слова.

Команды, хранящиеся в памяти – это закодированные с помощью 0 и 1 инструкции компьютеру по выполнению определенных действий. Например, командой может служить предписание компьютеру сложить два числа, выдать сообщение на экран и т.п. Совокупность команд образует программу, которая и определяет последовательность действий компьютера.

Стрелка от **ОП** к **П** отражает два факта:

1) Процессор в своей работе руководствуется командами программы, выбираемых из памяти.

2) Аргументы выполняемых операций тоже выбираются из памяти.

Стрелка **П** → **ОП** отражает тот факт, что получаемые в процессоре результаты отправляются на запоминание в память.

Память компьютера состоит из отдельных ячеек – байтов, каждый из которых имеет свой номер, называемый адресом. Из байтов образуются более сложные структуры памяти, например, полуслово (два байта), слово (четыре байта), двойное слово (восемь байтов). Однако, внутри этих структур, компьютер может обращаться к отдельным байтам. Таким образом, минимальной адресуемой единицей памяти компьютера является байт.

Память компьютера делится на *основную* (**ОП**) и *внешнюю* (**ВП**).

Основная память – это память высокого быстродействия, по скорости приближающаяся к скорости процессора, но ограниченной емкости. Конструктивно он выполнен в одном корпусе с процессором и является центральной частью

компьютера. **ОП** состоит в свою очередь из *оперативной* и *постоянной* памяти. Оперативная память служит для хранения оперативной (нужной в данный момент) информации, предназначенной для решения данной задачи. При решении другой задачи в оперативной памяти будет храниться уже другая информация для другой задачи.

При отключении компьютера вся информация в оперативной памяти теряется.

Постоянная память (ПЗУ – постоянное запоминающее устройство) предназначена для хранения постоянной (но часто нужной) информации, которая не зависит от того, какая задача решается. Чаще всего в ней хранится служебная информация, необходимая процессору. При отключении компьютера информация в постоянной памяти сохраняется. Кроме того, информацию в постоянной памяти нельзя стирать, изменять, дополнять.

Внешняя память (**ВП**) предназначена для долговременного хранения информации независимо от того, работает компьютер или нет и характеризуется очень большим объемом. Информацию во внешней памяти можно изменять, дополнять, уничтожать и т.д.

Для ввода данных с внешних носителей в ОП служат устройства ввода. Результаты решения задачи запоминаются в памяти компьютера.

Для выдачи результатов, хранящихся в памяти служат устройства вывода.

2.5.3. Принципы фон - Неймана

1. Принцип произвольного доступа к ячейкам памяти.

Компьютер на каждом такте своей работы может с одинаковым успехом обращаться к любым ячейкам памяти – как для чтения, так и для записи. Для обеспечения такого произвольного доступа к памяти каждой ячейке дается индивидуальное имя – эти имена и используются для указания нужных ячеек. Как правило, все ячейки пронумерованы от 0 до N-1, где N – общее число

ячеек памяти и в качестве имени ячейки принимается ее порядковый номер, который называют адресом ячейки, а также адресом ее содержимого (по аналогии с обычным почтовым адресом). Этот принцип обеспечивает большую свободу и простоту использования информации в памяти компьютера.

2. Принцип хранимой программы.

Компьютер выполняет программу, которая представлена в цифровой (двоичной) форме и хранится в той же памяти, что и данные, подлежащие переработке по этой программе. Перед началом решения задачи программа вводится (записывается) в отведенные для нее ячейки памяти и хранится там во время выполнения этой программы компьютером.

Принцип хранимой программы, прежде всего, делает компьютер универсальным в том смысле, что на одном и том же компьютере можно решать самые разные задачи, поскольку для решения каждой из них можно составить программу, ввести ее в память и заставить компьютер выполнить эту программу.

Факт представления программы в такой же форме, что и другие данные (например, числа) и хранения программы в памяти открывает целый ряд и других возможностей. Например, программа в процессе своего выполнения также может подвергаться переработке, что позволяет задавать в самой программе правила получения некоторых ее частей и тем самым получать весьма компактные программы. Более того, одна программа может являться результатом работы другой программы, что открывает принципиальную возможность автоматизировать процесс составления программы, поручив эту работу самой машине путем выполнения специальных программ.

Машинные операции.

Каждый тип компьютера имеет свой набор машинных операций, т.е. элементарных действий (с точки зрения программиста) по преобразованию

информации. Например, имеется операция сложения двух чисел. Причем сложение с фиксированной точкой и сложение с плавающей представляют собой две разные операции. Решение любой задачи должно быть сведено к выполнению последовательности операций из этого набора.

С точки зрения удобства использования желательно, чтобы набор операций был возможно шире и, чтобы он включал достаточно содержательные операции. С точки же зрения простоты реализации желательно, чтобы этот набор был невелик и, чтобы в него входили лишь простейшие операции, т.к. это позволяет сделать компьютер более простым, дешевым и надежным. Эти два требования противоречивы и на практике выбирается некоторый компромиссный вариант.

В каждой ячейке памяти хранится упорядоченная последовательность символов, которая называется *машинным словом* или просто словом.

Машинное слово может, например, представляет собой отдельное число, изображенное с достаточно высокой степенью точности, так что за одно обращение к памяти можно выбрать из нее сразу все число, являющийся одним из аргументов арифметической операции или запомнить результат выполнения этой операции.

Машинное слово может представлять собой и отдельную команду программы. Команда – это предписание компьютеру выполнить одну операцию из ее набора операций. Компьютер предназначен для выполнения не какого-нибудь одного, а многих разных алгоритмов.

Это обстоятельство требует наличия специального языка, который был бы понятен машине. Такой язык называется *машинным языком*, запись алгоритма на машинном языке называется *программой*, а самостоятельная фраза языка, являющаяся некоторой операцией, называется *командой*. Таким образом, программа представляет собой некоторую последовательность команд, а выполнение компьютерной программы сводится к последовательному выполнению отдельных ее команд.

В общем случае команда должна содержать информацию в том, какую именно операцию нужно выполнить, информацию об ее операндах (аргументах и результате) и информацию о том, какую команду надо выполнить вслед за данной командой.

Информация об операндах и следующей команде, подлежащей выполнению, задается путем указания их адресов. На каждом типе компьютеров фиксируется определенный формат команд, позиции (разряды) машинного слова, представляющего собой команду, объединяются в определенные поля, и каждое поле служит для задания информации определенного назначения. Одно из полей, называемое полем кода операции, служит для указания вида операции.

Для этого все операции, входящие в набор машинных операций компьютера, нумеруются (кодируются) и в поле операции команды указывается номер (код) требуемой операции.

Например, команда может иметь следующий формат:

θ	A1	A2	A3	A4
Поле кода операции	Поле первого адреса	Поле 2-го адреса	Поле 3-го адреса	Поле адреса следующей команды

Например:

0010	0100	0110	1000	1100

Такая команда называется *четырехадресной*. Она обеспечивает принудительный порядок выполнения команд в компьютере, т.к. из памяти выбирается и выполняется та команда, адрес которой записан в четвертом поле адресной части команды. В настоящее время подавляющее распространение получили компьютеры с естественным порядком выполнения команд, при котором команды нумеруются при программировании и в этом порядке записываются в

память, затем в этом же порядке выбираются из памяти и выполняются. При естественном порядке надобность в четвертом адресе
отпадает, и команда становится 3-х адресной.

θ	A1	A2	A3

Это структура наиболее точно отвечает формуле

Z=X θ Y,

что представляет известные удобства при программировании. Но эта структура с точки зрения технической реализации не является оптимальной. Структуру команды можно значительно упростить, если результат операции оставлять в сумматоре АУ или заносить в специальную ячейку, например в А1.

Тогда надобность в 3-м адресе отпадает.

θ	A1	A2

И, наконец, структура команды становится наиболее простой, если одним из двух операндов считать результат предыдущей операций, зафиксированным в сумматоре.

θ	A

2.6. Классификация компьютеров

Существуют различные системы классификации компьютеров. Рассмотрим некоторые из них.

2.6.1. Классификация по назначению

Классификация по назначению – одна из наиболее ранних систем класси-

фикации. Он связан с тем, как компьютер применяется. По этому принципу различают большие ЭВМ, мини – ЭВМ, микро – ЭВМ и персональные компьютеры, которые в свою очередь, подразделяются на массовые, деловые, портативные, развлекательные и рабочие станции.

Большие ЭВМ

Это самые мощные компьютеры. Их применяют для очень сложных расчетов, например в космических исследованиях, ядерной физике, в военных целях и др. За рубежом компьютеры этого класса называют мэйнфреймами (mainframe). Для обслуживания таких компьютеров необходимо несколько десятков человек. На базе таких суперкомпьютеров создаются вычислительные центры, включающие в себя несколько отделов или групп.

Мини – ЭВМ

От больших ЭВМ компьютеры этой группы отличаются уменьшенными размерами и, соответственно, меньшей производительностью и стоимостью. Такие компьютеры используются крупными предприятиями, научными учреждениями и некоторыми высшими учебными заведениями, сочетающими учебную деятельность с научной.

Мини – ЭВМ часто применяются для управления производственными процессами. Для обслуживания мини – ЭВМ также требуется создание вычислительного центра, хотя и не такой многочисленный, как для больших ЭВМ.

Микро – ЭВМ

Компьютеры данного класса доступны многим предприятиям и организациям. Для обслуживания этих компьютеров обычно не создаются вычислительные центры, т.к. для этого достаточно всего несколько человек. Микро – ЭВМ чаще всего используются как вспомогательные компьютеры для больших ЭВМ. Например, они используются для предварительной подготовки данных, перед

тем как их будет обрабатывать большая ЭВМ.

Персональные компьютеры (ПК)

Эта категория компьютеров получила особо бурное развитие в течение последних двадцати лет. Из названия видно, что такой компьютер предназначен для обслуживания одного рабочего места. Как правило, с персональным компьютером работает один человек. Несмотря на свои небольшие размеры и относительно невысокую стоимость, современные персональные компьютеры обладают немалой производительностью. Последние модели персональных компьютеров превосходят по производительности и особенно по своим возможностям большие ЭВМ 70-х годов, мини – ЭВМ 80-х годов и микро – ЭВМ первой половины 90-х годов 20 века.

Особенно широкую популярность персональные компьютеры приобрели после 1995 года в связи с бурным развитием Интернета. Возможностей персонального компьютера вполне достаточно для использования всемирной сети в качестве источника научной, справочной, учебной, культурной и развлекательной информации. Персональные компьютеры являются удобным средством автоматизации учебного процесса по любым дисциплинам, средством организации дистанционного (заочного) обучения и средством организации досуга.

До последнего времени персональные компьютеры условно рассматривали в двух категориях: бытовые ПК и профессиональные ПК. Бытовые модели имели, как правило, меньшую производительность, но в них были приняты особые меры для работы с цветной графикой и звуком, чего не требовалось для профессиональных моделей. В связи с достигнутым в последние годы резким удешевлением средств вычислительной техники, границы между профессиональными и бытовыми моделями практически стерлись.

Начиная с 1999 года в области персональных компьютеров действует международный классификационный стандарт – спецификация PC99. Он регламентирует принципы классификации персональных компьютеров и оговаривает

минимальные и рекомендуемые требования к каждой категории персональных компьютеров. Новый стандарт устанавливает следующие категории персональных компьютеров:

Consumer Personal Computer (PC) (массовый ПК);

Office PC (деловой ПК);

Mobile PC (портативный ПК);

Workstation PC (рабочая станция);

Entertainment PC (развлекательный ПК).

Согласно спецификации PC99 большинство персональных компьютеров попадают в категорию массовых ПК. Для деловых ПК минимизированы требования к средствам воспроизведения графики, а к средствам работы со звуковыми данными требования вообще не предъявляются. Для портативных ПК обязательным является наличие средств для создания соединений удаленного доступа, то есть средств компьютерной связи. В категории рабочих станций повышены требования к устройствам хранения данных, а категории развлекательных ПК – требования к средствам воспроизведения графики и звука.

Однако жизнь не стоит на месте. Классификация PC99 уже устарела, и мы стоим на пороге новой классификации. Грани между указанными категориями персональных компьютеров также практически стерлись. Сейчас уже невозможно представить себе офисный компьютер без соответствующих мультимедийных средств, а на развлекательном ПК можно производить достаточно сложные вычисления и расчеты.

*Под термином **мультимедиа** подразумевается сочетание нескольких видов данных в одном документе (текстовые, графические, музыкальные и видеоданные) или совокупность устройств для воспроизведения этих данных.*

Персональные компьютеры получили столь широкое и повсеместное распространение, их применение настолько разнообразное, что сейчас уже не говорят персональные компьютеры, а говорят просто компьютеры. В дальнейшем мы тоже не будем подчеркивать "персональность" компьютеров и будем гово-

рить просто о компьютерах.

2.6.2. Другие системы классификации компьютеров

Классификация по уровню специализации. По уровню специализации компьютеры делят на *универсальные и специализированные*. На базе универсальных компьютеров можно собирать вычислительные системы произвольной конфигурации. Так, например, один и тот же компьютер можно использовать для работы с текстами, музыкой, графикой, фото и видеоматериалами.

Специализированные компьютеры предназначены для решения конкретного круга задач. К таким компьютерам относятся, например, бортовые компьютеры автомобилей, судов, самолетов, космических аппаратов. Бортовые компьютеры решают только специальные задачи, на которые они запрограммированы на заводе изготовителе (например, оптимизацию расхода топлива двигателя в зависимости от конкретных условий движения или управление средствами ориентации и навигации и т.д.).

Специализированные компьютеры, ориентированные на работу с графикой, называют *графическими станциями*. Специализированные компьютеры, объединяющие компьютеры какого-нибудь предприятия в одну сеть, называют *файловыми серверами*. Компьютеры, обеспечивающие передачу информации между различными участками Интернета, называют *сетевыми серверами*.

Во многих случаях с задачами специализированных компьютеров могут успешно справляться и обычные универсальные компьютеры, и так часто на практике и происходит, хотя считается, что использование специализированных компьютеров все – таки эффективнее. Здесь опять таки идет тенденция к сближению обоих категорий компьютеров.

Классификация по типоразмерам. Различают настольные (desktop), портативные (notebook), карманные (palmtop).

Настольные модели распространены наиболее широко. Они являются при-

надлежностью рабочего места. Эти модели отличаются простотой изменения конфигурации за счет несложного подключения дополнительных внешних устройств или установки дополнительных внутренних компонентов. Достаточные размеры корпуса в настольном исполнении позволяют выполнять большинство подобных работ без привлечения специалистов, а это позволяет настраивать компьютерную систему оптимально для решения именно тех задач, для которых она была приобретена.

Портативные модели удобны для транспортировки. Их используют бизнесмены, коммерсанты, руководители предприятий и организаций, проводящие много времени в командировках и переездах. Особая привлекательность портативных компьютеров связана с тем, что их можно использовать в качестве средства связи.

Карманные модели выполняют функции "интеллектуальных записных книжек". Они позволяют хранить оперативные данные и получать к ним быстрый доступ. Некоторые карманные модели имеют жестко встроенное программное обеспечение, что облегчает непосредственную работу, но снижает гибкость в выборе прикладных программ.

Классификация по совместимости. В мире существуют множество различных видов и типов компьютеров. Они выпускаются разными производителями, работают с разными программами. При этом очень важным вопросом становится совместимость различных компьютеров между собой. От совместимости зависит возможность переноса программ с одного компьютера на другой и возможность совместной работы разных типов компьютеров с одними и теми же данными.

Аппаратная совместимость. По аппаратной совместимости различают так называемые *аппаратные платформы*. В области персональных компьютеров на сегодняшний день наиболее широко распространены две аппаратные платформы – *IBM PC* и *Apple Macintosh*. Принадлежность компьютеров к одной платформе повышает совместимость между собой, принадлежность к разным

платформам соответственно понижает совместимость, но, тем не менее, говорить о полной несовместимости этих двух платформ нельзя.

Кроме аппаратной совместимости существуют и другие виды совместимо сти: совместимость на уровне операционной системы, программная совместимость, совместимость на уровне данных.

2.7. Аппаратное обеспечение

В 2.3. мы на элементарном уровне рассмотрели принципы работы компьютера и основные его устройства. Рассмотрим теперь более подробно и на современном уровне устройства, из которых состоит компьютер.

Состав вычислительной системы называется *конфигурацией.* Аппаратные и программные средства вычислительной техники принято рассматривать отдельно. Соответственно, отдельно рассматривают *аппаратную конфигурацию* вычислительных систем и их *программную конфигурацию.*

К *аппаратному обеспечению* вычислительных систем относятся устройства и приборы, образующие аппаратную конфигурацию. Современные компьютеры и вычислительные комплексы имеют блочно – модульную конструкцию. Аппаратную конфигурацию, необходимую для выполнения конкретных видов работ, можно собирать из готовых узлов и блоков.

По способу расположения устройств относительно центрального *процессорного устройства* (ЦПУ – *Central Processing Unit, CPU*) различают внутренние и внешние устройства. Внешними, как правило, являются большинство устройств ввода/вывода данных (их также часто называют *периферийными устройствами*).

Согласование между отдельными узлами и блоками выполняют с помощью переходных аппаратно – логических устройств, называемых *аппаратными интерфейсами.* Стандарты на аппаратные интерфейсы в вычислительной тех-

нике называют *протоколами*.

Таким образом, протокол – это совокупность технических условий, которые должны быть обеспечены разработчиками устройств для успешного согласования их работы с другими устройствами.

Многочисленные интерфейсы, присутствующие в архитектуре любой вычислительной системы, можно условно разбить на две большие группы: *последовательные* и *параллельные*. Через последовательный интерфейс данные передаются последовательно, бит за битом, а через параллельный – одновременно группами битов. Количество битов, участвующих в одной посылке, определяется разрядностью интерфейса. Например, восьмиразрядные параллельные интерфейсы передают 8 бит (один байт) за один цикл.

Параллельные интерфейсы обычно имеют более сложное устройство, чем последовательные, но обеспечивают более высокую производительность. Их применяют там, где важна скорость передачи данных. Производительность параллельных интерфейсов измеряют *байтами в секунду* (байт/с; Кбайт/с; Мбайт/с).

В последовательных интерфейсах, как правило, не надо синхронизировать работу передающего и принимающего устройства (поэтому их часто называют *асинхронными интерфейсами*), но пропускная способность их меньше и коэффициент полезного действия ниже, так как из-за отсутствия синхронизации полезные данные предваряются и завершаются посылками служебных данных. Т.е. на один байт полезных данных могут приходиться 1 – 3 служебных бита (состав и структуру посылки определяет конкретный протокол).

Поскольку обмен данными через последовательные устройства производится не байтами, а битами, их производительность измеряют битами в секунду (бит/с; Кбит/с; Мбит/с). Несмотря на кажущую простоту перевода единиц измерения скорости последовательной передачи в единицы измерения скорости параллельной передачи путем механического деления на 8, такой пересчет не выполняют, поскольку он некорректен из-за наличия служебных данных.

Последовательные интерфейсы применяют для подключения "медленных" устройств (устройств печати, устройств ввода/вывода знаковой и сигнальной информации, например клавиатуры, контрольных датчиков, малопроизводительных устройств связи и т.п.).

Базовая аппаратная конфигурация

Компьютер – универсальная техническая система. Его *конфигурацию* (состав оборудования) можно гибко изменять по мере необходимости. Тем не менее, существует понятие *базовой конфигурации*, которую считают типовой. В таком комплекте компьютер обычно поставляется. В настоящее время в базовой конфигурации рассматривают четыре устройства:

системный блок;

монитор;

клавиатура;

мышь.

Системный блок

Системный блок представляет собой основной узел, внутри которого установлены наиболее важные компоненты. Устройства, находящиеся в системном блоке, называются *внутренними*, а устройства, подключаемые к нему снаружи, называются *внешними*.

По внешнему виду системные блоки отличаются формой корпуса. Корпуса выпускаются в горизонтальном (*desktop*) и вертикальном (*tower*) исполнении. Корпуса, имеющие вертикальное исполнение, различают по габаритам: *полноразмерный (big tower), среднеразмерный (midi tower) и малоразмерный (mini tower).* Среди корпусов, имеющих горизонтальное исполнение, выделяют *плоские* и *особо плоские (slim).*

Кроме формы, для корпуса важен параметр, называемый *форм – фактором*. В настоящее время в основном используются корпуса двух форм – факто-

ров: AT и ATX. Форм – фактор корпуса должен быть обязательно согласован с форм – фактором главной (системной) платы компьютера, так называемой *материнской платы*.

Корпуса компьютеров поставляются вместе с блоком питания и, таким образом, мощность блока питания также является одним из параметров корпуса. Для массовых моделей достаточной является мощность блока питания 200 – 250 Вт.

Одним из основных устройств, находящихся в системном блоке является материнская плата. На ней размещаются:

процессор – основная микросхема, выполняющая математические и логические операции;

микропроцессорный комплект (чипсет) – набор микросхем, управляющих работой внутренних устройств компьютера и определяющая основные функциональные возможности материнской платы;

шины – наборы проводников, по которым происходит обмен сигналами между устройствами компьютера;

оперативная память – набор микросхем, предназначенных для временного хранения данных, когда компьютер включен;

ПЗУ (постоянное запоминающее устройство) – микросхема, предназначенная для длительного хранения данных, в том числе и когда компьютер выключен;

разъемы для подключения дополнительных устройств (*слоты)*.

Рассмотрим эти устройства более подробно.

Оперативная память

Оперативная память (RAM – Random Access Memory) – это массив кристаллических ячеек, способных хранить данные. Существуют много различных типов оперативной памяти, но с точки зрения физического принципа действия различают *динамическую память (DRAM) и статическую память (SRAM).*

Ячейки динамической памяти (DRAM) можно представить в виде микро-

конденсаторов, способных накапливать заряд на своих обкладках. Это наиболее распространенный и экономически доступный тип памяти. Недостатки этого типа связаны, во-первых, с тем, что как при заряде, так и при разряде конденсаторов неизбежны переходные процессы, то есть запись данных происходит сравнительно медленно. Второй важный недостаток связан с тем, что заряды ячеек имеют свойство рассеиваться в пространстве, причем весьма быстро. Если оперативную память постоянно не "подзаряжать", утрата данных происходит через несколько сотых долей секунды. Для борьбы с этим явлением в компьютере происходит постоянная *регенерация (освежение, подзарядка)* ячеек оперативной памяти. Регенерация происходит несколько десятков раз в секунду и вызывает непроизводительный расход ресурсов компьютера.

Ячейки статической памяти (SRAM) можно представить как электронные микроэлементы – *триггеры*, состоящие из нескольких транзисторов. В триггере хранится не заряд, а состояние (включен/выключен), поэтому этот тип памяти обеспечивает более высокое быстродействие, хотя технологически он сложнее и, соответственно, дороже.

Микросхемы динамической памяти используют в качестве основной оперативной памяти компьютера. Микросхемы статической памяти используют в качестве вспомогательной памяти (так называемой *кэш-памяти*), предназначенной для оптимизации работы процессора.

Каждая ячейка памяти имеет свой адрес, который выражается числом. В настоящее время в процессорах Intel Pentium и некоторых других принята 32-разрядная адресация, а это означает, что всего независимых адресов может быть 2^{32}. Таким образом, в современных компьютерах возможна непосредственная адресация к полю памяти размером 2^{32}=4 294 967 296 байт (4,3 Гбайт). Однако это отнюдь не означает, что именно столько оперативной памяти непременно должно быть к компьютере. Предельный размер поля оперативной памяти, установленной в компьютере, определяется микропроцессорным комплектом (чипсетом) материнской платы и обычно составляет несколько сот

Мбайт. Одна адресуемая ячейка содержит восемь двоичных ячеек, в которых можно сохранить 8 бит, то есть один байт данных. Таким образом, адрес любой ячейки памяти можно выразить четырьмя байтами.

Представление о том, сколько оперативной памяти должно быть в типовом компьютере, непрерывно меняется. В середине 80-х годов поле памяти размером в 1 Мбайт казалось огромным, в начале 90-х достаточным считался объем 4 Мбайт, к середине 90-х годов он увеличился до 8 Мбайт, а затем и до 16 Мбайт. Сегодня типичным считается размер "оперативки" 64 Мбайт, но очень скоро эта величина будет превышена в несколько раз даже для моделей массового потребления.

Оперативная память в компьютере размещается на стандартных панельках, называемых *модулями*. Модули оперативной памяти вставляют в соответствующие разъемы на материнской плате. Конструктивно модули памяти имеют два исполнения – однорядные (*SIMM-модули)* и двухрядные (*DIMM-модули)*. На компьютерах с процессором Pentium однорядные модули можно применять только парами (количество разъемов для их установки на материнской плате всегда четное), а DIMM-модули можно устанавливать по одному. Многие модели материнских плат имеют разъемы как того, так и другого типа, но комбинировать на одной плате модули разных типов нельзя.

Основными характеристиками модулей оперативной памяти являются объем памяти и время доступа. SIMM-модули поставляются объемами 4, 8, 16, 32 Мбайт, а DIMM-модули 16, 32, 64, 128 Мбайт и более. Время доступа показывает, сколько времени необходимо для обращения к ячейкам памяти – чем оно меньше, тем лучше. Время доступа измеряется в миллиардных долях секунды (*наносекундах, нс*). Типичное время доступа к оперативной памяти для SIMM-модулей – 50-70 нс. Для современных DIMM-модулей оно составляет 7 – 10 нс.

Процессор

Процессор – основная микросхема компьютера, в которой и производятся все вычисления. Конструктивно процессор состоит из ячеек, похожих на ячейки оперативной памяти, но в этих ячейках данные могут не только храниться, но и изменяться. Внутренние ячейки процессора называются *регистрами*. Важно также отметить, что данные, попавшие в некоторые регистры, рассматриваются не как данные, а как команды, управляющие обработкой данных в других регистрах. Среди регистров процессора есть и такие, которые в зависимости от своего содержания способны модифицировать исполнение команд. Таким образом, управляя засылкой данных в разные регистры процессора, можно управлять обработкой данных. На этом и основано исполнение программ.

С остальными устройствами компьютера, и в первую очередь с оперативной памятью процессор связан несколькими группами проводников, называемых *шинами*. Основных шин три: *шина данных*, *адресная шина* и *командная шина*.

Адресная шина. У процессоров Intel Pentium адресная шина 32-разрядная, то есть состоит из 32 параллельных линий. В зависимости от того, есть напряжение на какой-то линии или нет, говорят, что на этой линии выставлена единица или ноль. Комбинация из 32 нулей и единиц образует 32-разрядный адрес, указывающих на одну из ячеек оперативной памяти. К ней и обращается процессор для копирования содержимого этой ячейки в один из своих регистров либо для записи содержимого регистра в эту ячейку.

Шина данных. По этой шине непосредственно и происходит процесс копирования данных из оперативной памяти в регистры процессора и обратно. В компьютерах с процессором Intel Pentium шина данных 64-разрядная, т.е. состоит из 64 линий, по которым за один раз (такт или цикл) на обработку поступает сразу 8 байтов.

Шина команд. Для того, чтобы процессор мог обрабатывать данные, ему нужны команды. Он должен знать, что следует сделать с теми байтами, которые хранятся в его регистрах. Эти команды в соответствии со вторым принципом фон Неймана поступают в процессор тоже из оперативной памяти, но не из тех областей, где хранятся данные, а оттуда, где хранятся программы. Команды тоже представлены в виде байтов. Самые простые команды укладываются в один байт, однако есть и такие, для которых нужно два, три и более байтов. В большинстве современных процессоров шина команд 32-разрядная (например, в процессоре Intel Pentium), хотя существуют 64-разрядные и даже 128-разрядные.

Система команд процессора. Совокупность всех возможных команд, которые может выполнить процессор над данными, образует так называемую *систему команд процессора*. Процессоры, относящиеся к одному семейству, имеют одинаковые или близкие системы команд. Процессоры, относящиеся к разным семействам, различаются по системе команд и не взаимозаменяемы.

Процессоры с расширенной и сокращенной системой команд. Чем шире набор системных команд процессора, тем сложнее его архитектура, тем длиннее формальная запись команды (в байтах), тем выше средняя продолжительность исполнения одной команды, измеренная в тактах работы процессора. Так, например, система команд процессоров Intel Pentium в настоящее время насчитывает более тысячи команд. Такие процессоры называют *процессорами с расширенной системой команд – CISC-процессорами (CISC – Complex Instruction Set Computing).*

В противоположность CISC-процессорам в середине 80-х годов появились процессоры архитектуры RISC с сокращенной системой команд (*RISC – Reduced Instruction Set Computing).* При такой архитектуре количество команд в системе намного меньше, и каждая из них выполняется намного быстрее. Таким образом, программы, состоящие из простейших команд, выполняются этими процессорами много быстрее. Оборотная сторона сокращенного набора ко-

манд состоит в том, что сложные операции приходится эмулировать далеко не эффективной последовательностью простейших команд сокращенного набора.

В результате конкуренции между двумя подходами к архитектуре процессоров сложилось следующее распределение их сфер применения:

CISC-процессоры используют в универсальных вычислительных системах;

RISC-процессоры используют в специализированных вычислительных системах или устройствах, ориентированных на выполнение единообразных операций.

Для компьютеров платформы IBM PC долгое время выпускались только CISC-процессоры, к которым относятся и все процессоры семейства Intel Pentium. Однако в последнее время компания AMD приступила к выпуску процессоров семейства AMD-K6, в основе которых лежит внутреннее ядро, выполненное по RISC-архитектуре, и внешняя структура, выполненная по архитектуре CISC. Таким образом, сегодня появились процессоры, имеющие гибридную структуру.

Совместимость процессоров. Если два процессора имеют одинаковую систему команд, то они полностью совместимы на программном уровне. Это означает, что программа, написанная для одного процессора, может исполняться и другим процессором. Процессоры, имеющие разные системы команд, как правило, несовместимы или ограниченно совместимы на программном уровне.

Группы процессоров, имеющих ограниченную совместимость, рассматривают как *семейство процессоров*. Так, например, все процессоры Intel Pentium относятся к так называемому семейству x86. Родоначальником этого семейства был 16-разрядный процессор Intel 8086, на базе которого собиралась первая модель компьютера IBM PC. Впоследствии выпускались процессоры Intel 80286, Intel 80386, Intel 80486, Intel Pentium 60, 66, 75, 90, 100, 133; несколько моделей процессоров Intel Pentium MMX, модели Intel Pentium Pro, Intel Pentium II, Intel Celeron, Intel Xeon, Intel Pentium III и другие. Все эти модели, и не только они, а также многие модели процессоров компаний AMD и Cyrix относятся к семей-

ству x86 и обладают совместимостью по принципу "сверху вниз".

Принцип совместимости "сверху вниз" – это пример неполной совместимости, когда каждый новый процессор "понимает" все команды своих предшественников, но не наоборот. Это естественно, поскольку двадцать лет назад разработчики процессоров не могли предусмотреть систему команд, нужную для современных программ.

Благодаря такой совместимости на современном компьютере можно выполнять любые программы, созданные в последние десятилетия для любого из предшествующих компьютеров, принадлежащего той же аппаратной платформе.

Монитор

Монитор – устройство визуального представления данных. Это не единственно возможное, но главное устройство вывода. Его основными потребительскими параметрами являются: размер и шаг маски экрана, максимальная частота регенерации изображения, класс защиты.

Размер экрана монитора измеряется между противоположными углами трубки кинескопа по диагонали. Единица измерения – дюймы. Стандартные размеры: 14"; 15"; 17"; 19"; 20"; 21". В настоящее время наиболее универсальными являются мониторы размером 15 и 17 дюймов, а для операций с графикой желательны мониторы размером 19 – 21 дюйм.

Изображение на экране монитора получается в результате облучения люминофорного покрытия остронаправленным пучком электронов, разогнанных в вакуумной колбе. Для получения цветного изображения люминофорное покрытие имеет точки или полоски трех типов, светящиеся красным, зеленым и синим цветом. Чтобы на экране все три луча сходились строго в одну точку и изображение было четким, перед люминофором ставят маску – панель с регулярно расположенными отверстиями или щелями. Часть мониторов оснащена

маской из вертикальных проволочек, что усиливает яркость и насыщенность изображения. Чем меньше шаг между отверстиями или щелями (*шаг маски*), тем четче и точнее полученное изображение. Шаг маски измеряют в долях миллиметра. В настоящее время наиболее распространены мониторы с шагом маски 0,25 – 0,27 мм. Устаревшие мониторы могут иметь шаг до 0,43 мм, что негативно сказывается на органах зрения при работе с компьютером. Модели повышенной стоимости могут иметь шаг маски менее 0,25 мм.

Частота регенерации (обновления) изображения показывает, сколько раз в течении секунды монитор может полностью сменить изображение (поэтому её также называют *частотой кадров*). Частоту регенерации измеряют в герцах (Гц). Чем она выше, тем четче и устойчивее изображение, тем меньше утомление глаз, тем больше времени можно работать на компьютере без перерыва. При частоте регенерации порядка 60 Гц мелкое мерцание изображения заметно невооруженным глазом. Сегодня такое значение считается недопустимым. Минимальным считают значение 75 Гц, нормативным – 85 Гц, а комфортным – 100 Гц и более.

Класс защиты монитора определяется стандартом, которому соответствует монитор с точки зрения техники безопасности. В настоящее время общепризнанными считаются следующие международные стандарты: *MPR-II, TCO-92, TCO-95, TCO-99* (приведены в хронологическом порядке). Стандарт MPR-II ограничил уровни электромагнитного излучения пределами, безопасными для человека. В стандарте TCO-92 эти нормы были сохранены, а в стандартах TCO-95 и TCO-99 ужесточены. Эргономические и экологические нормы впервые появились в стандарте TCO-95, а стандарт TCO-99 установил самые жесткие нормы по параметрам, определяющим качество изображения (яркость, контрастность, мерцание, антибликовые свойства покрытия).

Клавиатура

Клавиатура – это клавишное устройство управления компьютером. Она служит для ввода *алфавитно-цифровых (знаковых)* данных, а также команд управления. Клавиатура и монитор обеспечивают простейший *интерфейс пользователя.* С помощью клавиатуры управляют компьютером, а с помощью монитора получают от него отклик.

Клавиатура относится к стандартным средствам персонального компьютера. Ее основные функции не нуждаются в поддержке специальными системными программами (драйверами). Необходимое программное обеспечение для начала работы с компьютером уже имеется в микросхеме ПЗУ в составе базовой системы ввода/вывода (*BIOS – Basic Input Output System*), и потому компьютер реагирует на нажатие клавиш сразу после включения компьютера. Принцип действия клавиатуры заключается в следующем:

При нажатии на клавишу (или комбинацию клавиш) специальная микросхема, встроенная в клавиатуру, выдает так называемый *скан-код.*

Скан-код поступает в микросхему, выполняющая функции *порта* клавиатуры.

Порты *– специальные аппаратно – логические устройства, отвечающие за связь процессора с другими устройствами.*

Порт клавиатуры выдает процессору прерывание с фиксированным номером. Для клавиатуры номер прерывания – 9 (Interrupt 9, Int 9).

Получив прерывание, процессор откладывает текущую работу и по номеру прерывания обращается в специальную область оперативной памяти, в которой находится так называемый *вектор прерываний.* Вектор прерываний – это список адресных данных с фиксированной длиной записи. Каждая запись содержит адрес программы, которая должна обслужить прерывание с номером, совпадающим с номером записи.

Определив адрес начала программы – обработчика возникшего прерывания, процессор переходит к ее исполнению. Простейшая программа обработки клавиатурного прерывания "зашита" в микросхему ПЗУ, но программисты могут "подставить" вместо нее свою программу обработки данного прерывания, если изменят данные в векторе прерываний.

Программа-обработчик прерывания направляет процессор к порту клавиатуры, где он находит скан-код, загружает его в свои регистры, потом под управлением обработчика определяет, какой код символа соответствует данному скан-коду.

Далее обработчик прерываний направляет полученный код символа в небольшую область памяти, известную как *буфер клавиатуры* и прекращает свою работу, известив об этом процессор.

Процессор прекращает обработку данного прерывания и возвращается к отложенной задаче.

Введенный символ хранится в буфере клавиатуры до тех пор, пока его не заберет оттуда та программа, для которой он и предназначался, например, текстовый редактор.

Мышь

Мышь – устройство управления манипуляторного типа. Перемещение мыши по плоской поверхности синхронизировано с перемещением графического объекта (*указателя мыши*) на экране монитора.

В отличие от клавиатуры, мышь не является стандартным органом управления и компьютер не имеет для нее выделенного порта. Для мыши нет и постоянного выделенного прерывания, а базовые средства ввода и вывода (BIOS) компьютера, размещенные в постоянном запоминающем устройстве (ПЗУ), не содержат программных средств для обработки прерываний мыши.

В связи с этим в первый момент после включения компьютера мышь не

работает. Она нуждается в поддержке специальной системной программы – *драйвера мыши.*

Драйвер устанавливается либо при первом подключении мыши, либо при установке операционной системы компьютера. Хотя мышь и не имеет выделенного порта на материнской плате, для работы с ней используют один из стандартных портов, средства для работы с которыми имеются в составе BIOS. Драйвер мыши предназначен для интерпретации сигналов, поступающих через порт. Кроме того, он обеспечивает механизм передачи информации о состоянии и положении мыши операционной системе и работающим программам.

Компьютером управляют перемещением мыши по плоскости и кратковременными нажатиями правой или левой кнопок (клавиш). Эти нажатия называются *щелчками (click*). В отличие от клавиатуры мышь не может напрямую использоваться для ввода знаковой информации – ее принцип управления является событийным. Перемещения мыши и щелчки ее кнопок являются *событиями* с точки зрения ее программы – драйвера. Анализируя эти события, драйвер устанавливает, когда произошло событие и в каком месте экрана в этот момент находился указатель. Эти данные передаются в прикладную программу, с которой работает пользователь в данный момент. По ним программа может определить команду, которую имел в виду пользователь и приступить к ее исполнению.

Комбинация монитора и мыши обеспечивает наиболее современный тип интерфейса пользователя, который называется *графическим*. Пользователь наблюдает на экране графические *объекты* и *элементы управления*. С помощью мыши пользователь изменяет свойства объектов и приводит в действие элементы управления компьютером, а с помощью монитора получает отклик в графическом виде.

Контрольные вопросы:

1. Краткая история развития средств вычислительной техники.
2. Логические элементы вычислительной техники.
3. Функциональные элементы и типовые функциональные схемы.
4. Принципы фон – Неймана.
5. Классификация компьютеров.
6. Базовая аппаратная конфигурация компьютеров.

Глава III

Программное обеспечение компьютеров

3.1. Структура программного обеспечения

Программное обеспечение составляет вторую неотъемлемую компоненту современных компьютеров, без которого немыслимо их функционирование.

Программное обеспечение – это совокупность программ, позволяющих организовать решение задач пользователя. По назначению программное обеспечение можно условно разбить на три класса, рис. 3.1:

Рис. 3.1.

Системное программное обеспечение организует процесс обработки информации в компьютере. По сути – это программное устройство управления компьютером. Основу системного программного обеспечения составляют операционные системы (ОС).

Основное назначение ОС – управление процессом обработки информации и организация связи пользователя с компьютером. ОС – является неотъемлемой

частью компьютера, обеспечивая управление всеми аппаратными средствами, а человеку (пользователю) позволяет управлять работой компьютера с помощью соответствующего графического интерфейса и (или) команд.

Ядро ОС дополняется набором сервисных программ, которые служат разным целям. С их помощью производится например, начальная разметка дисков, установка параметров внешних устройств, выдача информации на печать, тестирование ОП и других устройств, стыковка с другими компьютерами (локальная сеть) и многое другое.

В состав системного программного обеспечения входят также средства диагностики и контроля. Эти программы обеспечивают автоматический поиск неисправностей в компьютере. Средства диагностики и контроля могут входить в состав ОС, но могут существовать и автономно.

Системы программирования (СП) позволяют разрабатывать программы на удобном для восприятия человеком языке, а не в машинных кодах. В былые времена программисты вынуждены были писать программы прямо в машинных кодах. Это было крайне неудобно, т.к. приходилось оперировать с двоичными командами, адресами и числами. Автор помнит, как в свое время ему приходилось выписывать длинные ряды нулей и единиц при программировании на ЭВМ "Минск-32". Очень муторное занятие, я вам скажу. Как известно, основной движущей силой прогресса является обычная человеческая лень. Именно "ленивые" программисты стали задумываться, как же облегчить процесс программирования. Так были созданы первые языки программирования, так называемые ассемблеры. Ассемблеры ненамного отличались от машинных языков. В них использовались мнемонические имена для команд и адресов.

Со временем были придуманы языки программирования высокого уровня. Ассемблеры и машинные языки относятся к языкам программирования низкого уровня в том смысле, что они максимально приближены к архитектуре того или иного процессора. Языки высокого уровня независимы от архитектуры компьютеров, т.е. они могут быть выполнены на любом компьютере. В СП входят

также компиляторы – комплекс программ, обеспечивающих автоматический перевод программ написанных на символическом языке на машинный язык.

Системы программирования – это особая категория программных средств. С их помощью создаются другие программы, в том числе и прикладные программы. Следовательно, эта категория программных средств совершенно аналогична средствам производства в промышленности – таким как станки, инструменты, сырье и т.д. При этом роль сырья играет информация – текстовые и числовые данные, закодированные сообщения, графическая информация и др.

Прикладное программное обеспечение составляют те программы, ради которых и существуют компьютеры. Это различные пакеты программ, предназначенные для пользователей различных категорий. Например, это могут быть различные текстовые редакторы, СУБД, табличные процессоры и интегрированные программные средства.

Прикладные программы составляют категорию программных средств, обращенных к пользователям компьютера – людям, которые не обязаны уметь программировать или знать устройство компьютера. Их цель заключается либо в том, чтобы с помощью компьютера решать свои повседневные задачи, включая профессиональные, либо учиться определенным навыкам (не обязательно относящихся к компьютеру), либо проводить свой досуг, играя в компьютерные игры.

Прикладные системы могут иметь общий характер, например, обеспечивать составление документов и их печать или хранение и выдачу различных справок.

Другие классы прикладных программ ориентированы на автоматизацию конкретных видов деятельности, например, обучение определенным предметам в школе, проектирование зданий и сооружений, анализ электрокардиограмм, проведение бухгалтерских расчетов и т.д. и т.п.

3.2. Основы операционной системы Windows

Вопрос: "Что такое слон?"

Ответ: "Это мышь с операционной системой"

Программистский фольклор

3.2.1. Графический интерфейс Windows

Операционная система Windows, разработанная фирмой, обеспечивает большое количество возможностей и удобств для пользователей и программистов. Прежде всего, Windows обеспечивает удобный и наглядный графический интерфейс пользователя (так называемый стандарт GUI – Graphical User Interface), а также обеспечивает многозадачный режим работы, что означает возможность одновременно запускать и работать с несколькими программами. Каждая программа работает в своем собственном окне. Управление окнами в Windows стандартизовано. Это значительно облегчает изучение самых разных программных средств. Рассмотрим некоторые элементы графического интерфейса:

Рабочий стол Windows

На рисунке 3.2 изображен вид рабочего стола сразу после включения компьютера. Рабочий стол можно ассоциировать с настоящим рабочим столом в Вашем офисе или дома. На нем можно расположить всё, что нужно Вам для работы. На рабочем столе расположены значки папок и значки ярлыков наиболее часто используемых программ. Количество значков на рабочем столе зависит

от способа установки Windows и от Вашей настройки рабочего стола. Значки ярлыков отличаются от значков папок наличием в левом нижнем углу небольшой стрелочки. (См. рис. 3.3)

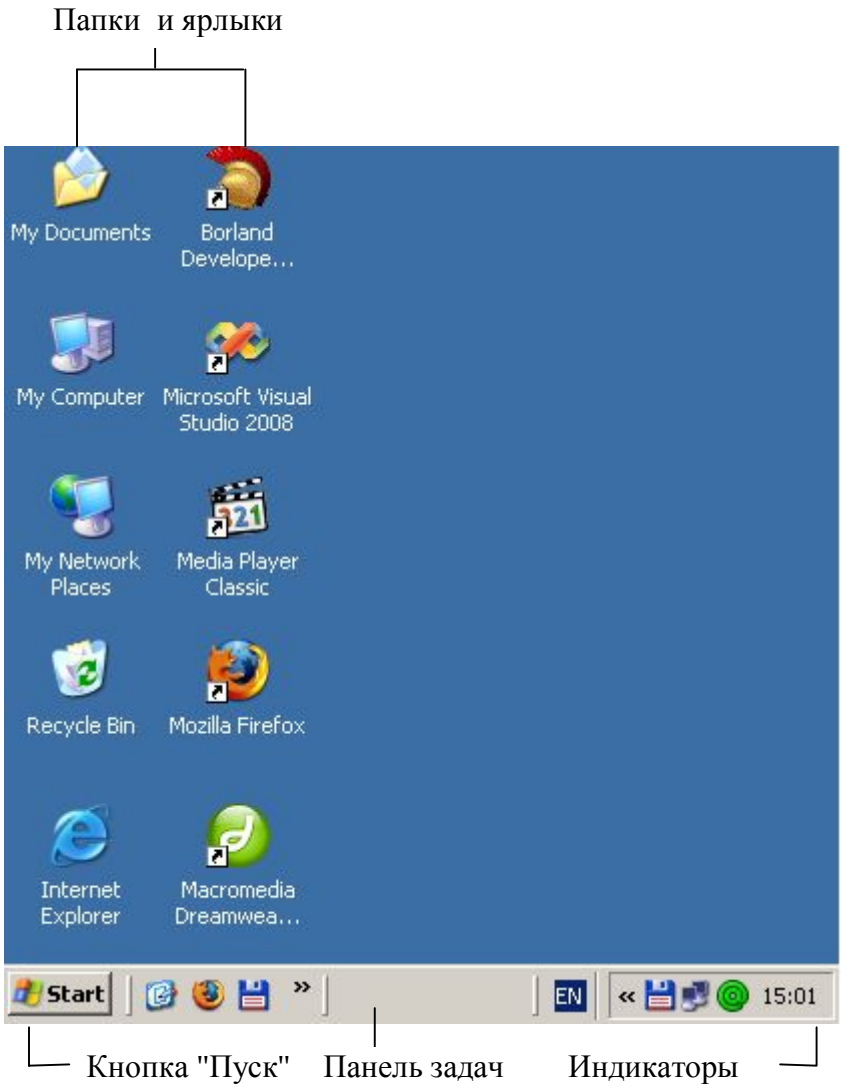


Рис. 3.2

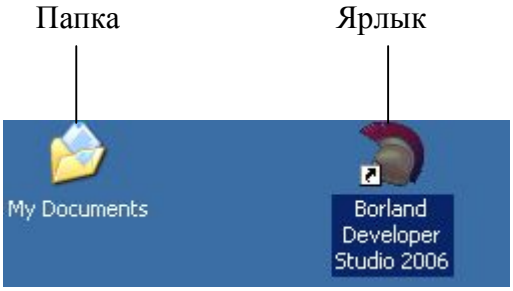


Рис. 3.3

Ярлык относится к какому-либо приложению или документу. (Обратите внимание, что понятия программа и приложение означают одно и тоже). В дальнейшем мы будем использовать оба этих термина.

Чтобы запустить приложение или документ достаточно дважды щелкнуть по его ярлыку на рабочем столе, при этом, если вы открываете документ, то автоматически запускается приложение, которое обрабатывает этот документ.

Существует и другой способ запуска программ. Для этого надо воспользоваться кнопкой "Пуск". Если нажать мышью на эту кнопку, то появится ***меню***.

Меню это такой механизм, который предоставляет пользователю возможность выбора дальнейших действий. Каждое меню состоит из отдельных пунктов (которые в свою очередь могут также быть меню – так называемые вложенные меню, говорят ещё подменю) или команд.

Итак, чтобы запустить приложение необходимо нажать кнопку "Пуск" и пропутешествовав по нужным пунктам меню и подменю добраться до нужной программы и нажать левую кнопку мыши или клавишу Enter.

На рисунке 3.4 показан пример запуска программы Word.

Внимание! При выключении компьютера всегда следует использовать кнопку "Пуск". Если выключить компьютер "просто так", то работа Windows завершится некорректно, могут быть потеряны некоторые данные и может нарушиться работоспособность Windows!

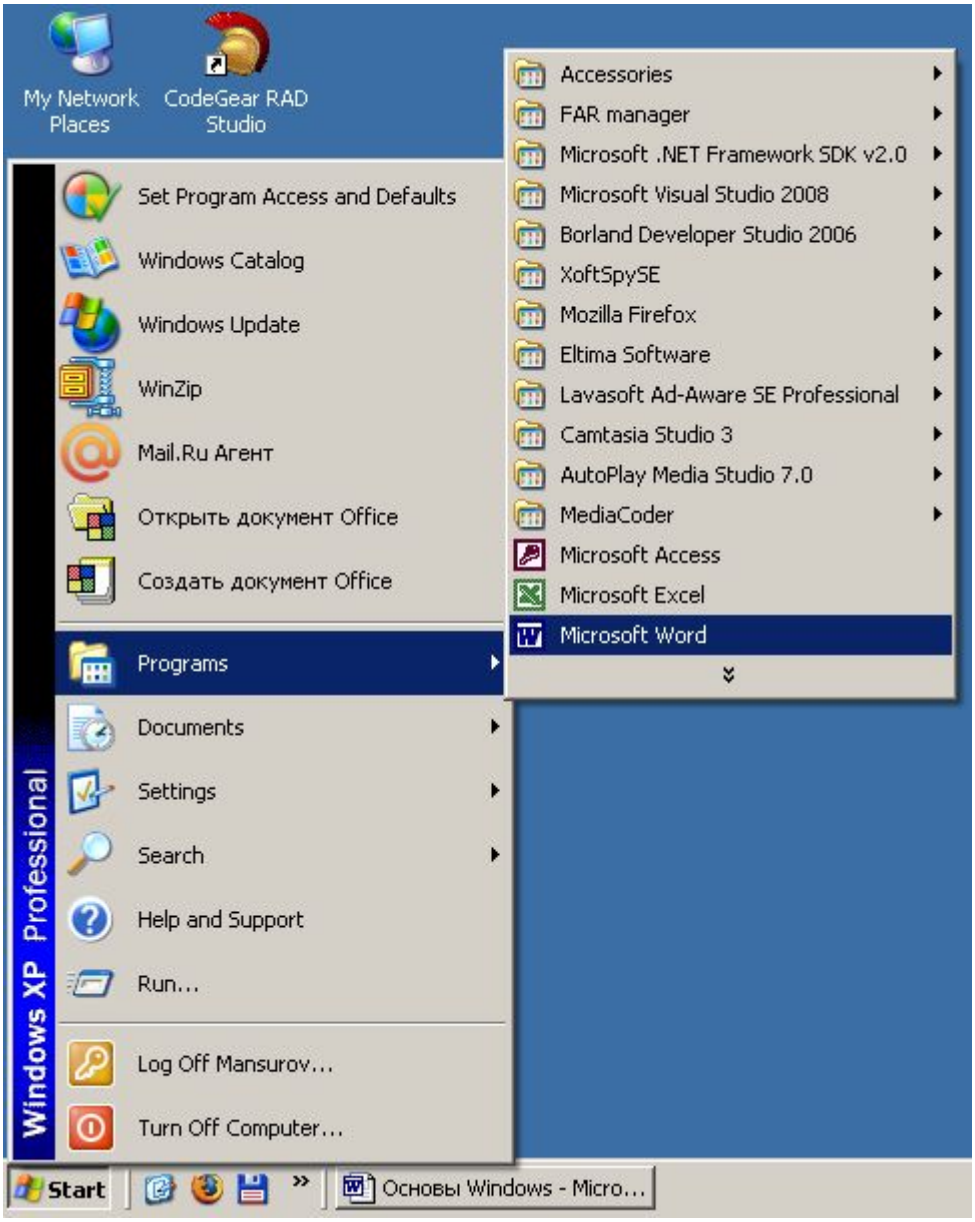


Рис. 3.4

Панель задач

Панель задач отображает значки работающих в данный момент приложений и позволяет быстро переключаться с одной задачи на другую. Для этого надо просто щелкнуть по соответствующему значку программы на панели задач. После закрытия приложения его значок также исчезает с панели задач.

Индикаторы.

Индикаторы дают некоторую дополнительную информацию для пользователя. Например, в правом нижнем углу отображаются часы. Рядом слева находится индикатор используемого языка.

En (English) – означает, что в данный момент используется английский язык.

Ru (Russian) – означает, что в данный момент используется русский язык.

Для переключения с одного языка на другой используются клавиши Shift + Alt (знак + означает, что эти клавиши нужно нажимать одновременно). Если индикатор языка виден на экране, то можно просто щелкнуть по индикатору.

Элементы окон Windows

Как уже отмечалось, управление окнами в Windows стандартизовано. Практически все окна имеют одни и те же элементы управления. Рассмотрим их на рисунке 3.5.

Любое окно имеет ***строку заголовка*** в котором указывается имя программы и имя документа.

При нажатии кнопки "Свернуть" активное окно уменьшается до размеров значка и помещается на панель задач. Следует иметь в виду, что программа в этом случае продолжает работать, но в свернутом виде. При нажатии кнопки "Развернуть" окно разворачивается до максимально возможных размеров. При этом кнопка "Развернуть" заменяется на кнопку "Восстановить"

Если щелкнуть по этой кнопке, то окно восстановит свой первоначальный размер. Чтобы закрыть окно, щелкните по кнопке "Закрыть".

Вторая строка окна называется ***строкой меню***. Если щелкнуть по какому – либо пункту, то появится ***подменю*** – вертикальное меню в котором можно выбрать соответствующие команды.

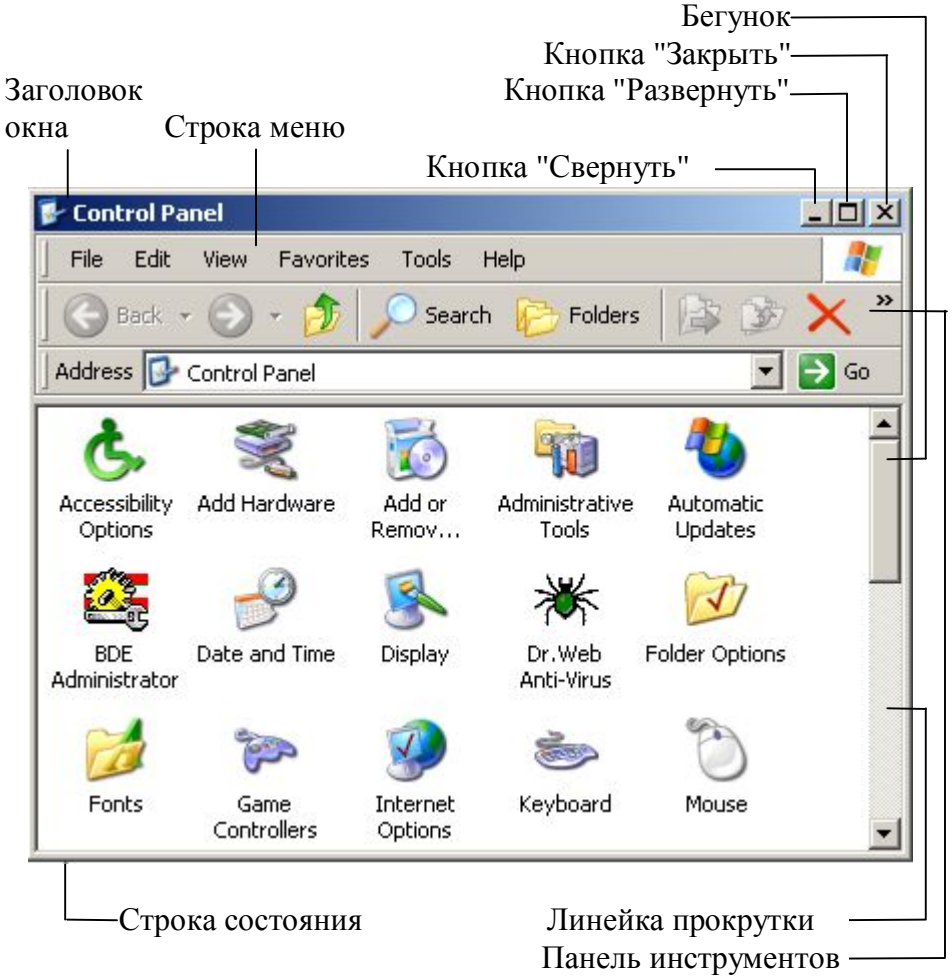


Рис. 3.5

Панель инструментов позволяет выполнять некоторые действия быстрее, используя кнопки на панели и не обращаясь к пунктам меню.

Кнопки на панели инструментов дублируют некоторые, наиболее часто используемые команды меню.

Линейка прокрутки используется в том случае, когда вся информация не помещается в окне. Если нажать на кнопки , то информация в окне будет сдвигаться вверх или вниз.

Бегунок показывает относительное положение в тексте. Например, если бегунок находится вверху линейки прокрутки, то вы находится ближе к началу документа, если внизу, то ближе к концу документа. Можно прокручивать текст и с помощью бегунка. Для этого надо нажать на бегунок мышью и, не отпуская двигать бегунок в нужном направлении. Линейки прокрутки бывают

вертикальные и горизонтальные. Методы работы с горизонтальными линейками прокрутки абсолютно идентичны. В тех случаях, когда информация полностью помещается в окне, линейки прокрутки не выводятся.

Строка состояния отображает состояние различных элементов в данном окне. В дальнейшем мы познакомимся с соответствующими примерами, работая в Word.

Меню

В окне каждой программы может находиться строка меню. Это так называемое горизонтальное меню. Каждый пункт этого меню состоит в свою очередь из соответствующих подменю – вертикальных меню. Для того чтобы открыть подменю надо щелкнуть по соответствующему пункту горизонтального меню. Каждый пункт меню содержит либо команды с помощью которых можно выполнить какие – то действия, либо режимы (опции), которые можно активизировать или дезактивизировать, либо содержат ещё дополнительные подменю (вложенные меню).

Соглашения по обозначениям в меню

Таблица 3.8

Соглашения	Значение соглашения
Треугольник (►) после команды	Означает, что этот пункт содержит вложенное меню (подменю).
Многоточие (**…**) после команды	После выбора этого пункта появится диалоговое окно. Оно содержит необходимые для выполнения команды параметры, которые нужно выбрать или ввести.
Комбинация клавиш после	Эта комбинация клавиш является "быстрыми" клавишами для команды, т.е. для выполнения

команды	этой команды можно нажать на указанную комбинацию клавиш, не открывая меню.
Блеклый (или невидимый) пункт	В данный момент этот пункт (команду) нельзя использовать
Галочка (√) перед командой	Означает, что данная команда или режим активизирована. Повторное нажатие отключает команду или режим. Значок √ при этом исчезает.

Диалоговые окна

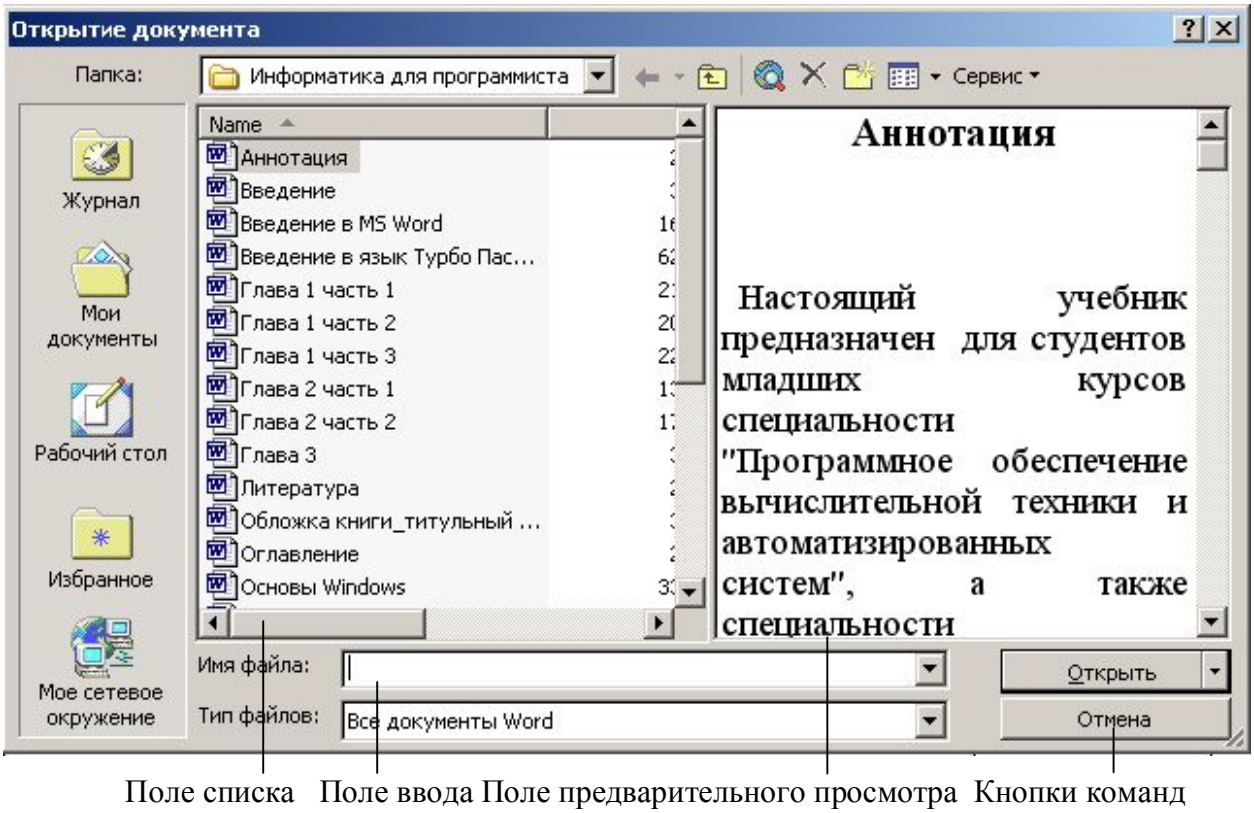
Для выполнения некоторых команд необходимо задать один или несколько параметров. Для этого используются ***диалоговые окна***. Если после команды в меню стоит многоточие, то после её выбора открывается диалоговое окно. (См. рис. 3.6, 3.7).

Диалоговые окна содержат специальные области (называемые элементами управления или параметрами).

Значения параметров необходимо выбрать из некоторого списка или ввести с клавиатуры.

Типы параметров диалоговых окон.

Кнопки команд – используются для выполнения действий. Все диалоговые окна обязательно имеют кнопки **ОК** и **Отмена**. При нажатии кнопки ОК диалог завершается и команда с выбранными вами параметрами выполняется. При нажатии кнопки Отмена окно диалога просто закрывается и команда не исполняется. Наличие остальных кнопок зависит от вида команды.



Поле списка Поле ввода Поле предварительного просмотра Кнопки команд

Рис. 3.6

Поле ввода – используется для ввода значения параметра с клавиатуры.

Поле списка – требуемый параметр выбирается из списка. Выбор осуществляется простым щелчком по нужному элементу списка. При необходимости можно воспользоваться линейкой прокрутки.

Раскрывающийся список – используется в целях экономии места на экране. Чтобы раскрыть этот список нажмите на треугольник. В дальнейшем работа с этим списком полностью аналогична работе с обычным списком.

Флажок (индикатор) – это поле может иметь только два значения: включен или выключен. С помощью флажков можно выбрать (включить) некоторые режимы или какие-то опции. Соглашения здесь таковы:

- выключено
- включено

Чтобы включить или отключить флажок нужно просто щелкнуть по флажку. Если имеются несколько флажков (индикаторов), то каждый из них может быть включен/выключен независимо от других флажков.

Переключатель – говорят еще радиокнопка, также имеет только два значения: включен/выключен. Имеет вид кружочка с точкой или без. Если переключатель включен, то он отмечается кружочком с точкой внутри. Если имеются несколько переключателей, то может быть включен только один переключатель. Таким образом, этот механизм позволяет выбрать только один из возможных режимов (опций). В этом различие переключателей от флажков.

Также в литературе вы можете встретить понятия "группа независимых переключателей" – это в нашей терминологии флажки и "группа зависимых переключателей" – по-нашему просто переключатели.

Вкладки – некоторые команды используют очень большое количество различных параметров. В этом случае диалоговые окна оформляются в виде вкладок или подшивок. Чтобы выбрать некоторую группу параметров нужно нажать на требуемую вкладку. Рисунок 3.7.

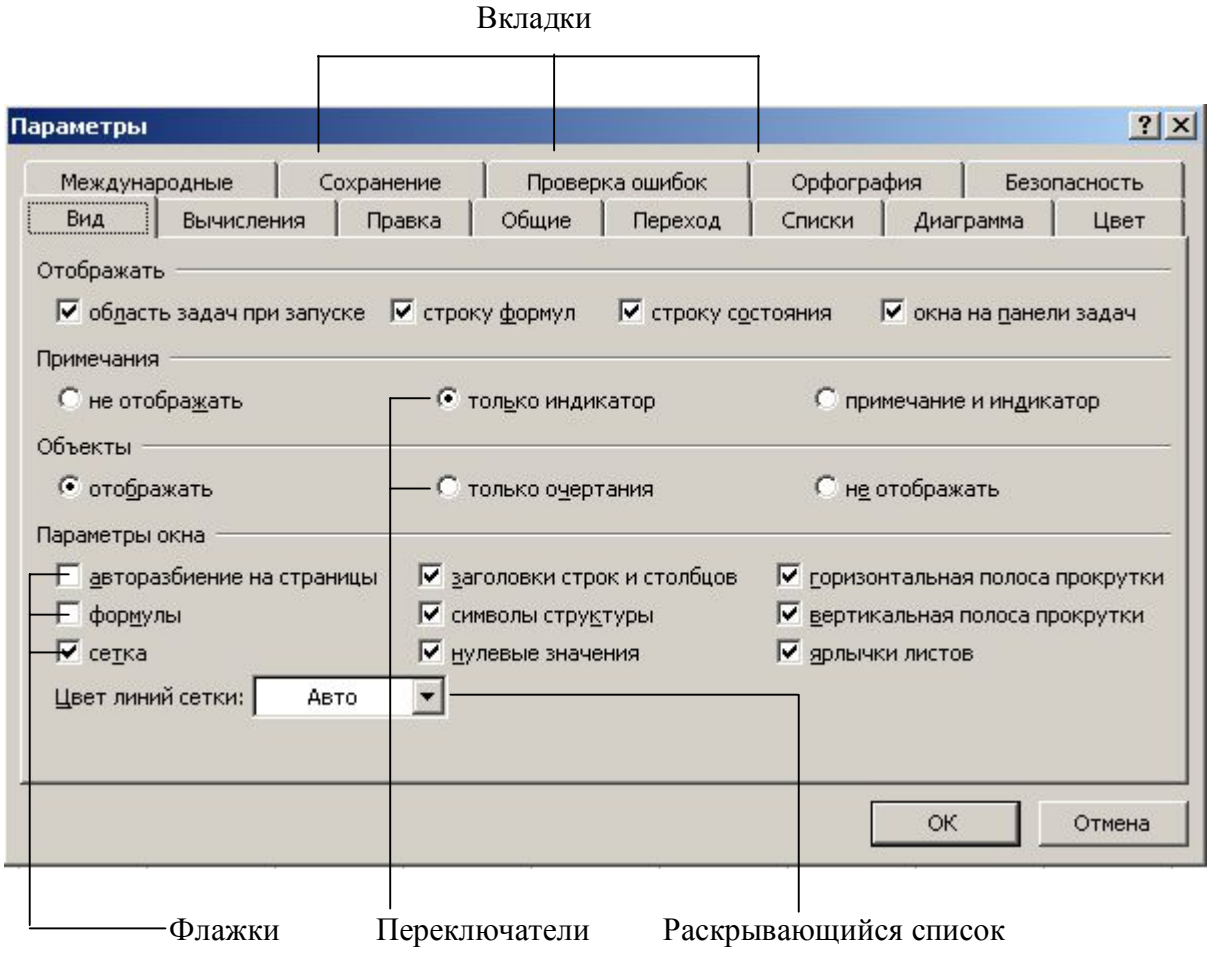


Рис.3.7

3.2.2. Файловая система Windows

Диски

Любая информация в компьютере хранится на диске. Диск вращается в устройстве подобно компакт – диску в проигрывателе, что позволяет записывать и считывать информацию с диска. Устройство в котором находится диск называется накопителем или дисководом. Диски бывают гибкие и жесткие, а также компакт – диски (CD – ROM). Жесткие диски находятся внутри компьютера в так называемом системном блоке.

Каждый диск имеет однобуквенное обозначение (имя). Дисковод для гибких дисков имеет обозначение A. Жесткий диск имеет обозначение C. И, наконец, CD-ROM имеет обозначение D.

Большая часть информации в компьютере хранится на жестком диске. Гибкие диски обычно используются для копирования с жесткого диска (или, наоборот, на жесткий диск) информации, например, для перенесения информации с одного компьютера на другой. На жестком диске помещается столько информации, сколько помещается на сотнях (и даже на тысячах) гибких дисках. CD – ROM также имеет очень большой объем. Обычно на CD – ROM поставляется программное обеспечение, игры и т.д.

Файлы, папки, документы

Файл – это область памяти на диске предназначенный для хранения информации. Информацией может быть программа, написанная на каком – либо языке программирования (например, Бейсик, Паскаль, Си), программа готовая к исполнению (в машинном коде), данные, необходимые для работы программы, текстовые документы. Кроме того, в файлах может содержаться: числовые данные, закодированная табличная, графическая и любая другая информация.

Каждый файл имеет ***имя*** и в зависимости от характера содержащейся в файле информации ***тип***.

Имя файла – это последовательность букв, цифр и некоторых знаков, расположенных в произвольном порядке.

Тип (говорят ещё расширение имени файла) – это последовательность из трех букв, цифр или знаков. Между именем файла и типом всегда ставится точка. Например, Годовой отчет.doc

Здесь: Годовой отчет – имя файла, doc – тип файла.

В большинстве случаев пользователю не обязательно присваивать файлу тип. Приложение, в котором обрабатывается данный файл, автоматически присвоит ему нужный тип.

Основной единицей информации, которую обрабатывают приложения Windows является ***документ***. Каждый документ хранится на диске в виде файла. Программы также хранятся на диске в виде файлов.

Обычно на диске хранятся сотни и тысячи документов. Для упорядочения их используют ***папки***.

Папка – это группа документов, объединенных по какому – либо признаку. Папки можно ассоциировать с выдвижными ящичками стола, в каждом из которых хранятся документы.

Каждая папка должна иметь имя и может, в свою очередь, входить в другую папку как единое целое (вложенные папки). Так образуется иерархическая файловая система Windows. На каждом диске имеется так называемый корневой каталог. В нем регистрируются файлы и папки 1-го уровня. В последних регистрируются файлы и папки следующего 2-го уровня и т.д.

При такой сложной структуре файловой системы для нахождения нужного файла (документа) необходимо указать не только имя документа, но и всю цепочку вложенных папок. Такая цепочка называется ***путем***. При записи пути каждый из вложенных папок отделятся от другого знаком \. Пример записи пути:

C:\Windows\Бухгалтерские документы\Отчеты\Годовой отчет.doc

Здесь C: - имя диска;

Windows – папка верхнего (первого) уровня;

Бухгалтерские документы – вложенная папка (второго уровня);

Отчеты – вложенная папка (третьего уровня);

Годовой отчет.doc – имя документа;

Из примера видно, что папки не имеют расширения имени.

Проводник Windows

С помощью Проводника можно просматривать содержимое дисков и управлять файлами. Под управлением файлами понимается выполнение различных операций над ними как целыми неделимыми объектами (копирование, перемещение, удаление, создание новых папок, удаление ненужных и пр.).

Чтобы открыть окно Проводника выполните одно из следующих действий:

Если ярлык для Проводника создан и виден на Рабочем столе, щелкните по этому ярлыку;

Щелкните по кнопке Пуск, выберите пункт Программы, а затем щелкните по пункту Проводник;

Щелкните правой кнопкой по значку Мой компьютер, затем выберите пункт Проводник;

Щелкните правой кнопкой по кнопке Пуск, затем выберите пункт Проводник;

Окно Проводника выглядит так, как показано на рисунке 3.8.

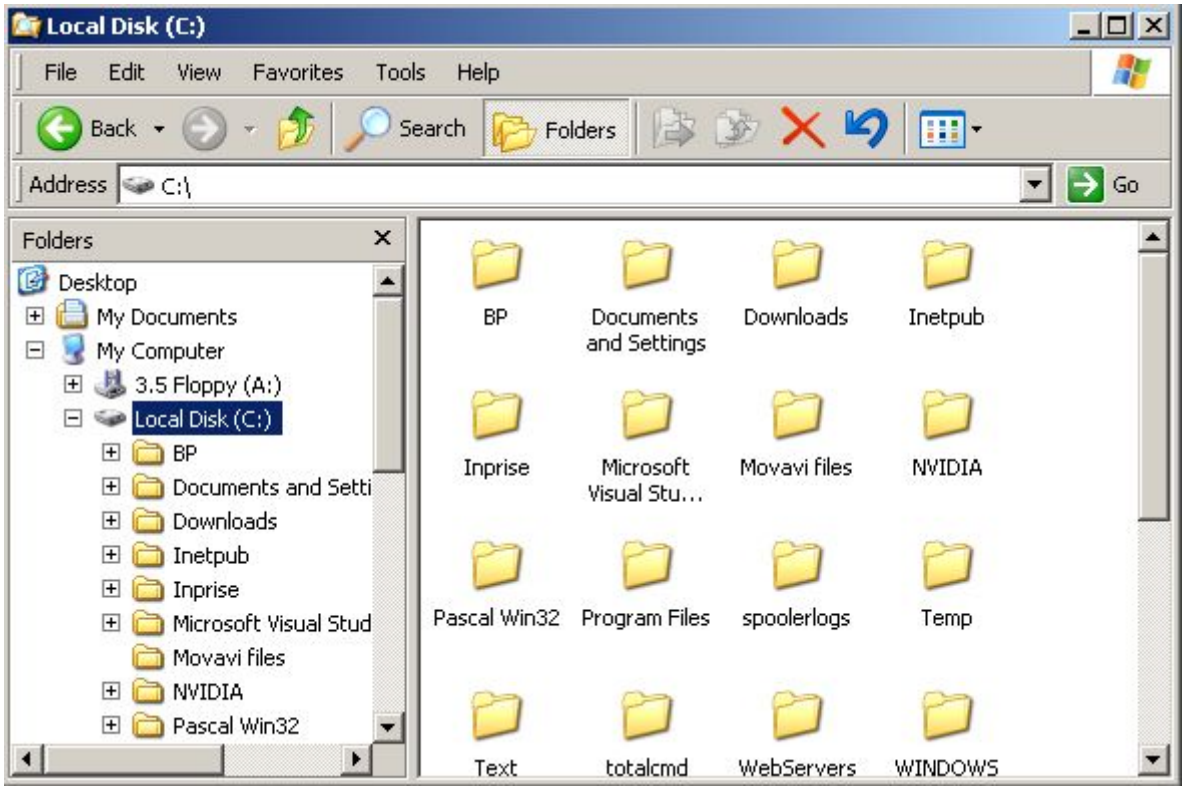


Рис. 3.8. Окно Проводника

Окно Проводника разбито на две части. Левое окно называется **Все папки**, а правое окно – **Содержимое папки**. В верхней части окна под строкой меню имеется раскрывающийся список, который называется **Переход к другой папке**. В той же строке находится панель инструментов Проводника.

Выбор файла или папки

Чтобы выбрать папку или файл просто щелкните по имени файла или папки. Выделенный объект будет подсвечен. Если вы выделили папку в левом окне, то он откроется автоматически.

Чтобы открыть документ или папку в правом окне нужно дважды щелкнуть по имени документа или папки.

Чтобы перейти на один уровень вверх в иерархии папок щелкните по кнопке с изображением желтой папки с маленькой стрелкой вверх на панели инструментов.

Создание новой папки

Документы близкие по смыслу и одинаковое предназначение удобнее хранить в одной папке (например, документы с отчетами). Для создания новой папки необходимо:

Открыть папку, в которую нужно вложить новую папку.

В окне ***Содержимое папки*** (правое окно) щелкнуть в свободном месте правой кнопкой.

Из появившегося контекстного меню выбрать пункт **Создать**, а затем пункт **Папку.** В списке файлов и папок появится новая папка с именем, присвоенным ей по умолчанию.

Чтобы изменить имя папки, просто наберите его в строке ввода и нажмите Enter.

Копирование и перемещение файлов и папок

Для выполнения этих действий:

Выделите нужный файл или папку.

Щелкните правой клавишей по имени выбранного файла или папки. Выберите команду **Копировать** (или, если хотите переместить, выберите команду **Вырезать**).

Перейдите в ту папку, в которую хотите копировать или переместить. В свободном месте окна щелкните правой клавишей. Выберите команду **Вставить**

При копировании файл или папка остается на прежнем месте (копия отправляется на новое место). При перемещении файл или папка физически перемещается на новое место.

Удаление файла или папки

Если Вам некоторые файлы и (или) папки не нужны, то можно их удалить, чтобы они не занимали лишнего места на диске.

Для удаления файла:

Открыть папку, в которой находится удаляемый файл.

Выделить нужный файл. Нажать на правую кнопку мыши

Из появившегося контекстного меню выбрать пункт **Удалить**.

На экран выведется диалоговое окно **Подтверждение удаления файла**. Если вы уверены, что хотите действительно удалить этот файл, нажмите кнопку **Да**

Для удаления папки:

Выделить нужную папку. Нажать правую кнопку мыши.

Из контекстного меню выбрать пункт **Удалить.**

На экран выведется диалоговое окно Подтверждение удаления папки. Если вы уверены, что действительно хотите удалить эту папку –нажмите кнопку **Да**

К Вашему сведению:

1. При удалении папки удаляются и все вложенные в нее папки и файлы.
2. Все удаляемые папки и файлы помещаются в так называемую ***корзину***.
3. Если Вы все же передумали, то ещё можете восстановить папки и файлы, взяв их обратно из корзины. Для этого на рабочем столе откройте корзину, дважды щелкнув по значку корзины. Выберите нужные папки и файлы. В меню **Файл** выберите пункт **Восстановить**.
4. Время от времени не забывайте очищать корзину от "мусора". Для этого в меню **Файл** выберите команду **Очистить**

Переименование файла или папки

Чтобы переименовать файл или папку надо щелкнуть по имени правой клавишей и выбрать команду **Переименовать**

Выделение группы файлов или папок

Чтобы выделить группу файлов или папок:

Если файлы или папки расположены подряд, то нажмите клавишу Shift и, не отпуская её, щелкните по имени первого файла или папки, затем последнего. После чего отпустите клавишу Shift.

Если файлы или папки расположены не подряд, то нужно нажать клавишу Ctrl и, не отпуская её щелкать по именам нужных файлов и папок.

С выделенной группой файлов и папок можно выполнять операции, так как будто это один файл. Однако это могут быть только операции копирования, перемещения и удаления.

Создание ярлыков

Ярлыки – это одно из удобств, которые предоставляет Windows. Они позволяют быстро запустить программу или быстро открыть документ простым двойным щелчком по ярлыку на рабочем столе, без долгих поисков нужного файла и копания в папках. Чтобы создать ярлык, надо щелкнуть по имени файла правой клавишей и из контекстного меню выбрать команду **Создать ярлык**.

Присвойте ярлыку имя. Для того чтобы переместить ярлык на рабочий стол нажмите на ярлык мышью и, не отпуская клавишу, перенесите ярлык на рабочий стол, затем отпустите клавишу.

Если вы хотите, чтобы ярлык оставался и в той папке, где он был создан, то прежде чем переносить его на рабочий стол нажмите на клавишу Ctrl и, не

отпуская его, нажмите мышью на ярлык и "тащите" его на рабочий стол.

Кстати, это ещё один способ копирования и перемещения – так называемый метод "перенести – оставить".

Контрольные вопросы:

1. Перечислите и объясните значение элементов управления окнами в Windows;
2. Где находится панель задач и что она отображает?
3. Что такое меню?
4. Как переключиться с русского языка на английский и наоборот?
5. Чем отличается флажок от переключателя?
6. Чем отличается сворачивание окна от её закрытия?
7. Что такое файл?
8. Что такое путь?
9. Как создать новую папку?
10. Чем отличается перемещение от копирования?

Глава IV. Основы программирования

Да благословит высшая мудрость нашу беседу и ниспошлет тебе дар понимания, а мне – дар ясности и глубины в моих словах.

Л. Соловьев "Повесть о Ходже Насреддине"

4.1. Понятие алгоритма.

Компьютер - это устройство для решения задач. Не обязательно задач чисто математического характера. Это могут быть и задачи управления станками или ракетами, и задачи планирования производства, и задачи информационно-справочного обслуживания, и задачи обработки гипертекстовой информации и мультимедиа, т.е. обработки звуковой и видеоинформации. Чтобы решить какую - либо задачу на компьютере необходимо сначала придумать как ее вообще решить, т.е. придумать алгоритм ее решении. Алгоритм – является одним из краеугольных понятий информатики и программирования.

Итак, что же понимается под алгоритмом?

Алгоритм – это строгая и четкая, конечная система правил, которая определяет последовательность действий над некоторыми объектами и после конечного числа шагов приводит к достижению поставленной цели.

Из определения алгоритма следует, что он должен удовлетворять следующим требованием:

1) конечность (финитность)

Алгоритм всегда должен заканчиваться после конечного числа шагов.

Процедуру, обладающую всеми характеристиками алгоритма, за исключением конечности, вызывают вычислительным методом.

2) определенность (детерминированность)

Каждый шаг алгоритма должен быть строго определен. Действия, которые необходимо произвести, должны быть строго и недвусмысленно определены в каждом возможном случае, так чтобы если дать алгоритм нескольким людям, то они, действуя по этому алгоритму, получали один и тот же результат. Поскольку обычный язык полон двусмысленностей, то чтобы преодолеть это затруднение, для описания алгоритмов разработаны формально определенные языки программирования, или машинные языки, в которых каждое утверждение имеет абсолютно точный смысл.

Запись алгоритма на языке программирования называется программой.

3) Алгоритм должен иметь некоторое число входных данных, т.е. величин, объектов заданных ему до начала работы. Эти данные берутся из некоего конкретного множества объектов.

4) Алгоритм имеет одну или несколько выходных величин, т.е. величин, имеющих вполне определенное отношение к входным данным.

5) Эффективность

От алгоритма требуют, чтобы он был эффективным. Это означает, что все операции, которые необходимо произвести в алгоритме, должны быть достаточно простыми, чтобы их в принципе можно было выполнить точно и за конечный отрезок времени с помощью карандаша и бумаги.

Следует отметить, что для практических целей "финитность" является наиболее важным требованием – используемый алгоритм должен иметь не просто конечное, а предельно конечное, разумное число шагов. Например, в принципе имеется алгоритм, определяющий, является ли начальное положение в игре в шахматы форсированно выигранным для белых или нет. Но для выполнения этого алгоритма требуется фантастически огромный промежуток

времени. Пусть имеется компьютер, обладающий быстродействием 100 млн. операций в секунду. Тогда этот компьютер будет выполнять алгоритм в течение 10^{23} лет. Для сравнения укажем, что период времени с начала возникновения жизни на земле и до наших дней намного меньше 10^{23} лет.

Пример алгоритма.

4.1.1 Алгоритм Евклида.

Алгоритм Евклида нахождения наибольшего общего делителя двух целых чисел, т.е. наибольшее целое число, которое делит нацело заданные числа.

1. Рассмотреть А как первое число и В как второе число. Перейти к п.2.

2. Сравнить первое и второе числа. Если они равны, то перейти к п.5. Если нет, то перейти к п.3.

3. Если первое число меньше второго, то переставить их местами. Перейти к п.4.

4. Вычесть из первого числа второе и рассмотреть полученную разность как новое первое число. Перейти к п.2.

5. Рассмотреть первое число как результат.

Стоп.

Этот набор правил является алгоритмом, т.к. следуя ему, любой человек умеющий вычитать, может получить наибольший общий делитель для любой пары чисел. Следуя этому алгоритму, найдем НОД чисел 544 и 119.

$A = 544 \quad B = 119$

1)
$$\begin{array}{r} -\ 544 \\ 119 \\ \hline 425 \end{array}$$

$A = 425 \quad B = 119$

2)
$$\begin{array}{r} -\ 425 \\ 119 \\ \hline 306 \end{array}$$

$A = 306 \quad B = 119$

3)
$$\begin{array}{r} -\ 306 \\ 119 \\ \hline 187 \end{array}$$

$A = 187 \quad B = 119$

4)
$$\begin{array}{r} -\ 187 \\ 119 \\ \hline 68 \end{array}$$

5) $A = 68 \quad B = 119$

$A < B$

Меняем местами

$A = 119 \quad B = 68$

6)
$$\begin{array}{r} -\ 119 \\ 68 \\ \hline 51 \end{array}$$

7) $A = 51 \quad B = 68$

$A < B$

$A = 68 \quad B = 51$

8) $$\begin{array}{r} -\ 68 \\ 51 \\ \hline 17 \end{array}$$

9) $A = 17 \quad B = 51$

$A < B$

$A = 51 \quad B = 17$

10) $$\begin{array}{r} -\ 51 \\ 17 \\ \hline 34 \end{array}$$

$A = 34 \quad B = 17$

11) $$\begin{array}{r} -\ 34 \\ 17 \\ \hline 17 \end{array}$$

А=В=17=НОД

Данный способ записи алгоритмов возможен, но неудобен. Во-первых, нет наглядности, во-вторых, "многословен". Одним из способов записи алгоритма являются блок- схемы. Блок- схемой называется такое графическое изображение структуры алгоритма, в котором каждый этап или шаг процесса переработки данных представляется в виде прямоугольника, ромба, овала или другой геометрической фигуры, называемой блоком.

Эти фигуры соединяются между собой линиями со стрелками, отображающими последовательность выполнения алгоритма. Внутри каждой фигуры разрешается писать произвольный текст, в котором на понятном человеку языке сообщаются о нужных вычислениях в соответствующей части программы.

Приняты определенные стандарты графических обозначений. Так, прямоугольник обозначает вычислительные действия в результате которых изменяются значения данных. Ромбом обозначают этап разветвления алгоритма. Вы-

бор одного из двух возможных направлений дальнейшего счета производится в зависимости от выполнения условия, записанного в ромбе. Овалом обозначают начало и конец алгоритма. В параллелограммах записывают процедуры ввода и вывода данных. Запишем алгоритм Евклида в виде блок- схемы:

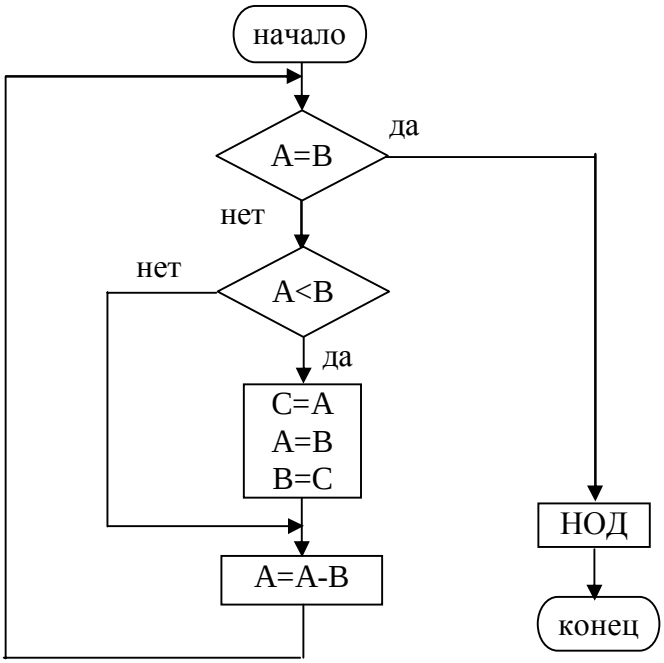


Рис. 4.1. Алгоритм Евклида

Существуют и другие способы записи алгоритмов. В частности, достаточно распространены записи алгоритмов на так называемых псевдоязыках. Псевдоязык похож на обычный алгоритмический язык программирования, но в нем можно использовать и естественный язык для более ясного выражения своих мыслей. Некоторые псевдоязыки более близки к языкам высокого уровня, так называемые алголоподобные языки (в прошлом существовали языки программирования ALGOL-60, ALGOL-68, которые в настоящее время не используются). Примером такого псевдоязыка служит язык описания алгоритмов, применяемым в школах. Некоторые псевдоязыки более близки к машинным языкам, так называемые ассемблеры. Примером такого языка описания алгоритмов служит язык, предложенный Д. Кнутом в его знаменитой книге "Искусство программирования" [9].

Итак, чтобы решить какую либо задачу, нужно придумать алгоритм ее решения и записать его в том или ином виде. Этот процесс называется алгоритмизацией. Но в таком виде компьютер алгоритм не сможет выполнить. Следует представить этот алгоритм в таком виде, чтобы компьютер мог его выполнить. Для этого нужно, во-первых, разбить алгоритм на элементарные операции, называемые инструкциями или командами, которые умеет выполнять компьютер, и, во- вторых, записать каждую такую операцию на языке, понятом компьютеру. Такая запись алгоритма на языке компьютера называется программой. Процесс разработки программы называется программированием. А человек, выполняющий эту работу - программистом. При этом следует различать программу в машинных кодах, говорят еще исполнимая программа и программу, написанную на каком-либо языке программирования. Обычно человек пишет программу на языке высокого уровня, в крайнем случае, на ассемблере. Компилятор переводит ее в программу на машинном языке, которую и исполняет компьютер.

Программирование – это научная дисциплина. Если бы процессы программирования разных задач не имели между собой ничего общего, то программирование, как таковое, не было бы научной дисциплиной. Но дело обстоит не так. Существуют общие методы, которые позволяют, постепенно расчленяя задачи на подзадачи, сводить их решение, в конечном счете, к некоторым элементарным операциям (чаще всего к элементарным арифметическим операциям), подобно тому, как разбирая совершенно непохожие сложные механизмы, мы обнаруживаем, что они состоят из одинаковых деталей и узлов, только по разному соединенных (напр. подшипники, болты, гайки и т.д.). Из этого, конечно, не следует, что процесс программирования не содержит в себе элементов творчества. Составление программы, такой же творческий процесс, что и разработка сложного механизма из заданных наборов деталей.

Основное назначение алгоритмов заключается в их фактическом вы-

полнении тем или иным исполнителем. Т.е. алгоритм составляется для того, чтобы он был выполнен для решения какой либо задачи. В качестве исполнителя может выступать кто угодно – человек, станок с ЧПУ, компьютер и т.д. В силу своей природы, компьютеры стали наиболее распространенными исполнителями алгоритмов. Собственно они и были придуманы для того, чтобы с их помощью исполнять алгоритмы.

Назначение компьютеров, таким образом, состоит в фактическом исполнении алгоритмов, разработанных человеком.

Важно понять, что компьютер сам по себе ничего не делает. Он лишь слепо выполняет то, что укажет ему человек. Поэтому, когда говорят, что компьютер управляет станками, ракетами, сочиняет музыку, играет в шахматы "как бы самостоятельно", это неверно!

Это человек придумывает алгоритм, т.е. способ управления станками, ракетами, сочинения музыки, игры в шахматы, а компьютер лишь исполняет этот алгоритм. Таким образом, компьютер это помощник, инструмент человека для решения различных задач, но инструмент очень мощный, разноплановый. С помощью компьютера человек может решать самые разнообразные задачи.

Рассмотрим пример разработки алгоритма.

4.1.2. Задача о поездах и мухе

Рассмотрим задачу, которую мы позаимствовали из [11] и рассмотрим на этом примере основные приемы и способы составления алгоритмов. Пусть два города А и В удалены на расстояние d=600 км. Одновременно из каждого города отходят поезда в направлении друг другу. Поезд вышедший из города А имеет скорость v_1=40 км/ч, а из В – имеет скорость v_2=60 км/ч. Одновременно из пункта А вылетает исключительно быстрая муха со скоростью v=200 км/ч и летит на встречу поезду вышедшего из пункта В. При встрече с ним муха дела-

ет поворот и летит обратно навстречу с поездом, идущему из А. Когда она его встретит, муха снова делает поворот и летит в обратном направлении. Так продолжается до тех пор, пока поезда не встретятся.

Задание.

1) определить длину различных отрезков пути, которые пролетит муха
2) определить общее расстояние, которое пролетит муха

На второй вопрос легко ответить: поезда встретятся через 600/(40+60)=6 часов, а муха за это же время пролетит расстояние 200*6=1200 км. Но оставим в стороне эту хитрость, и будем решать задачу в «лоб».

Прежде всего, нужно составить алгоритм, т.е. придумать, как решить задачу с помощью элементарных шагов. Алгоритм будем записывать в виде блок-схемы. С чего нужно составлять блок-схему? Как правило, блок-схема начинается с ввода исходных данных и начальных установок, т.е. определение начальных значений некоторых вспомогательных переменных. Но в начале бывает трудно сразу определить начальные значения, поэтому рисуют сначала центральную часть блок-схемы (ядро), в которой определены те действия, которые решают задачу, причем сначала рисуют укрупненную блок-схему, чтобы как можно яснее представить себе весь процесс решения.

Какие идеи сразу приходят в голову?

Самая простая идея заключается в том, чтобы производить действия следующим образом:

1) вычислить длину первого отрезка пути, который пролетит муха
2) вычислить длину второго отрезка и прибавить к предыдущему результату
3) вычислить длину третьего отрезка и сложить с ранее полученной суммой и т.д.

Мы можем этот процесс представить в виде

Рис. 4.2. Первый вариант алгоритма

Тут же возникает вопрос: т.к. схема должна иметь конечную длину, то нужно средствами рисунка выразить понятие повторения, кроме того, надо указать какие величины нужно вывести на экран. Отсюда схему перечерчиваем в виде:

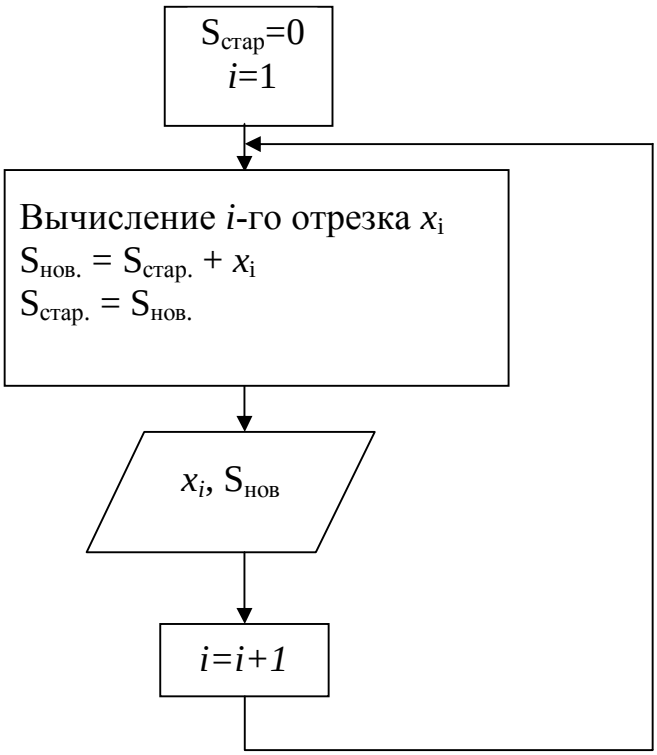


Рис. 4.3. Второй вариант алгоритма

Случается, что когда некоторая последовательность операций закончена, необходимо бывает вернуться и повторить эти операции. Это называется циклом. На этой схеме виден серьезный недостаток - вычислительный процесс по этой схеме никогда не закончится. Необходимо придумать критерий окончания алгоритма. Пока не совсем все ясно, будем считать, что нам задано число повторений n, а также заданы все исходные данные, тогда блок-схема примет вид:

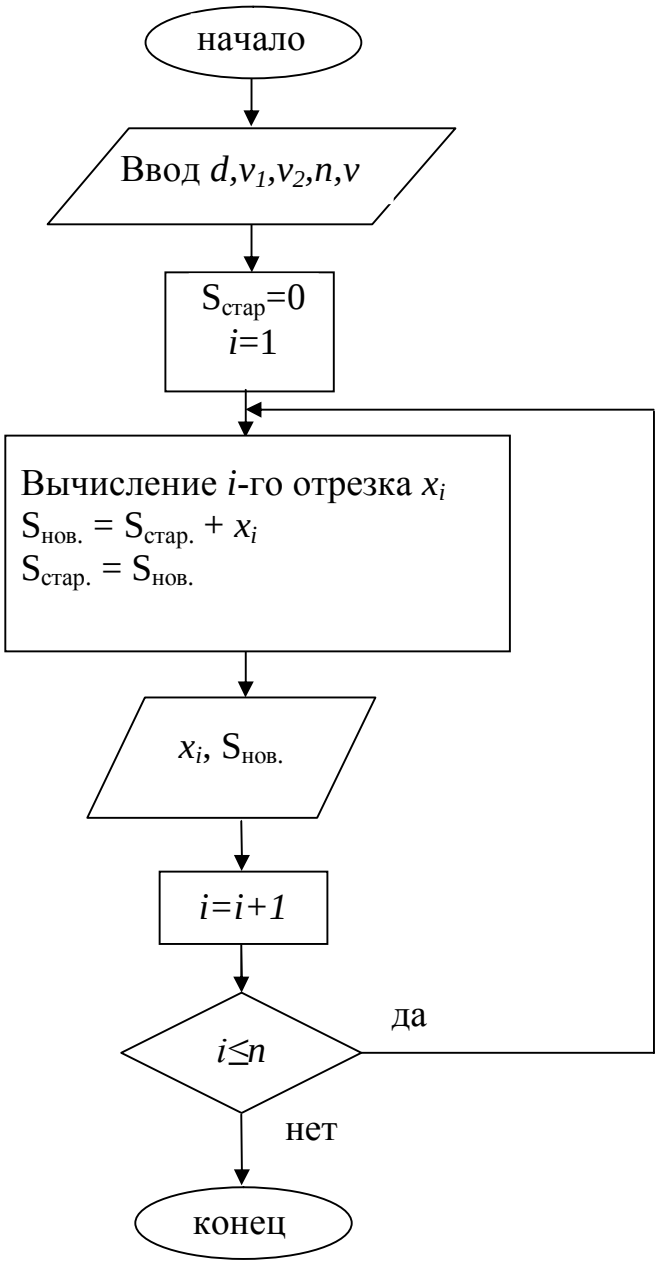


Рис. 4.4. Третий вариант алгоритма

Подробный анализ отрезков пути.

Мы написали "вычисление отрезка x_i". Но как все-таки вычислять эти отрезки? Вот тут многие "застревают" не в силах придумать что-либо. Единого рецепта как дальше придумывать алгоритм не существует. Ответ один – надо "просто" думать, искать, пробовать! Помните, что придумывать, разрабатывать алгоритмы это такой же творческий процесс, как, например, сочинять стихи, музыку, писать картины и т.д. Многие быстро придумывают алгоритмы, другие – с трудом и долго, ну а третьи вообще не могут придумать даже простейшие алгоритмы. Так что, уважаемый читатель, не всякий может стать программистом! Для этого тоже нужны определенные способности к творчеству, если хотите – талант! Программист – это творческая профессия! Разумеется, как и во всякой другой творческой профессии, знания тоже играют немаловажную роль!

Вернемся к нашей задаче. Нарисуем график для облегчения.

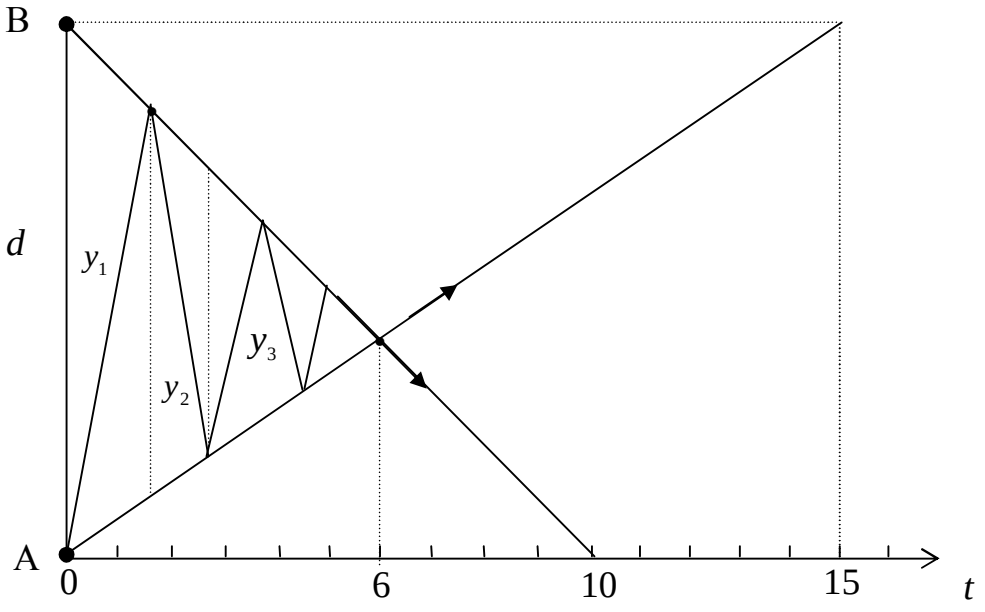


Рис. 4.5. График движения поездов

Будем рассуждать, используя элементарные законы физики о прямолинейном равномерном движении тел, которые изучаются в школе.

Когда муха в первый раз полетит в направлении к поезду из пункта А, то до встречи с этим поездом пройдет время:

$$t_1 = \frac{d}{(v + v_2)}; \tag{4.1}$$

В момент встречи мухи и поезда В (будем для краткости называть поезд, вышедший из пункта В поездом В, а поезд, вышедший из пункта А, поездом А) расстояние между поездами составит:

$$y_1 = d - t_1(v_1 + v_2); \tag{4.2}$$

а муха пролетит расстояние:

$$x = t_1 v; \tag{4.3}$$

И соответственно, при полете мухи в обратном направлении имеем:

$$t_2 = \frac{y_1}{(v + v_1)}, \qquad y_2 = y_1 - t_2(v_1 + v_2), \qquad x = t_2 v; \tag{4.4}$$

Отсюда можно вывести общую формулу:

$$t = \frac{y}{v + v_2}; \quad x = tv \text{ при полете мухи из А к В.} \tag{4.5}$$

$$t = \frac{y}{v + v_1}; \quad x = tv \text{ при полете мухи из В к А.} \tag{4.6}$$

$$y = y - t(v_1 + v_2) \tag{4.7}$$

Нарисуем блок-схему с учетом полученных формул**:**

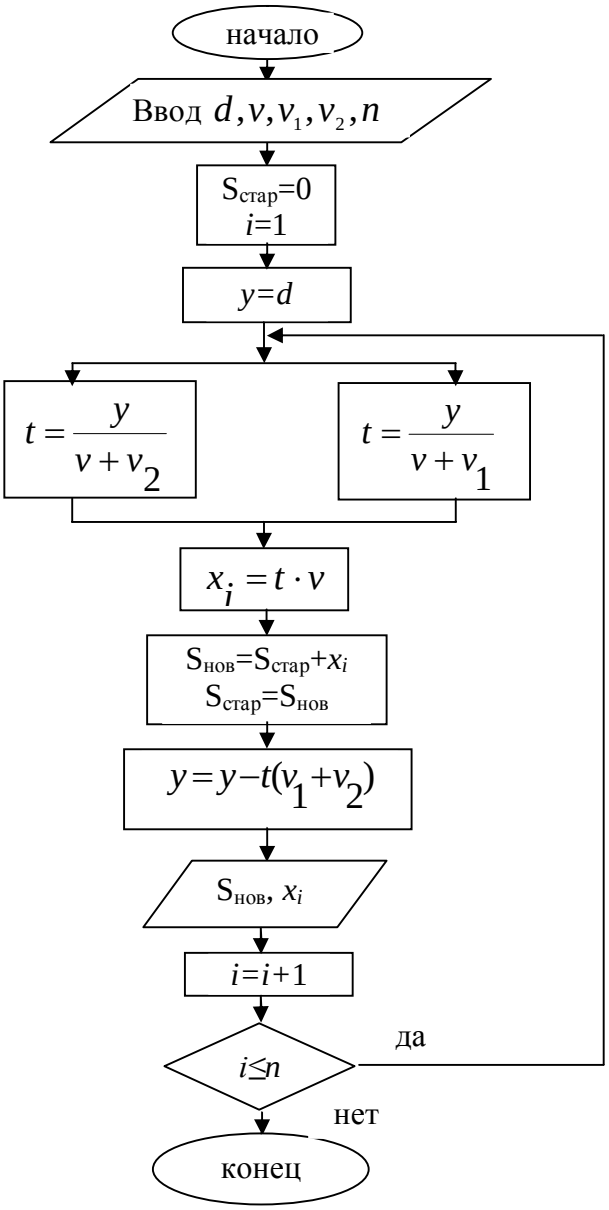


Рис. 4.6. Четвертый вариант алгоритма

Здесь мы видим, что алгоритм должен разветвляться на две ветви (когда муха летит к поезду B и когда летит к поезду A). Как компьютеру сообщить, что нужно попеременно проходить через эти ветви? Используется прием, который широко известен в программировании и называется метод "флажков" или "семафора". Будем считать что, если флажок поднят, то нужно идти по левой веточке, если опущен, то по правой. В качестве флажка принято использовать либо целочисленную переменную, либо булевую переменную, которая может принимать только два значения:

`0` – означает, что флажок опущен, `1` – означает, что флажок поднят, если это переменная целого типа и `false` – флажок опущен, `true` – флажок поднят, если это булевая переменная.

Перерисуем блок-схему

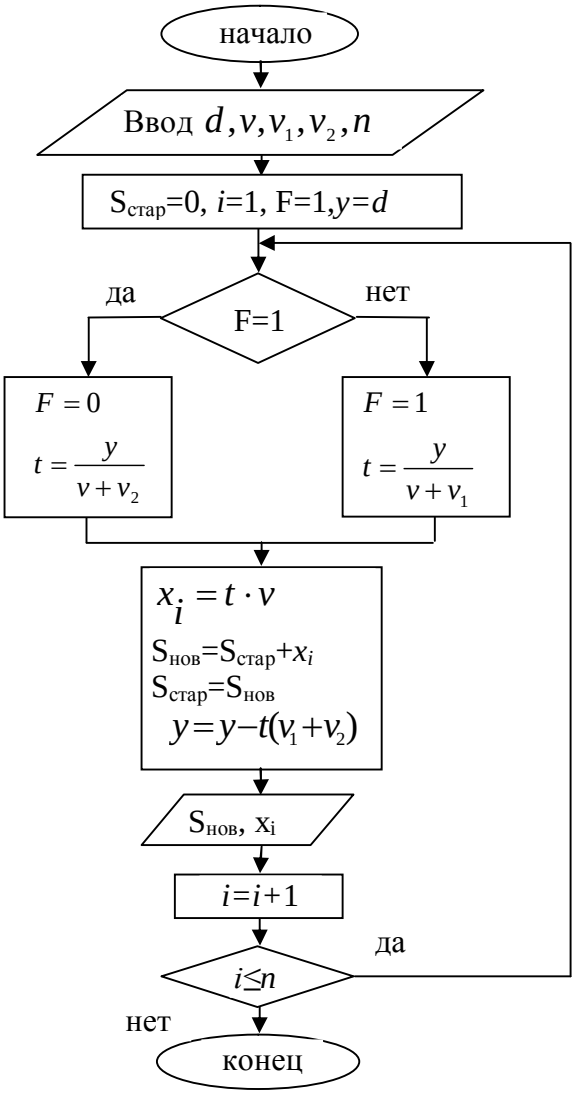


Рис. 4.7. Пятый вариант алгоритма

Построив один вариант блок-схемы всегда нужно посмотреть нельзя ли ее упростить?

Анализируя блок-схему, видим, что мы попеременно используем $S_{стар}$, $S_{нов}$, причем после вывода на экран $S_{нов}$, его значение нам не нужно, оно все равно изменяется. Отсюда можно использовать только одну переменную S.

$S = S + x_i$

Далее: критерий окончания алгоритма мы определили не совсем хорошо. Действительно не ясно, чему равно n. Может 100, а может 1000. Допустим, мы приняли n=100, а на самом деле число отрезков оказалось равным 10, тогда 90 раз алгоритм будет работать «впустую», т.к. полученные результаты будут бессмысленными. Как быть? Не лучше ли определить конец алгоритма по y. Действительно из рисунка видно, что $y \to 0$. Будем считать, что поезда встретились, если $y \le 10^{-2}$. Кроме того, мы видим, что и значение очередного отрезка x_i после вывода его на экран, нам не нужно, т.е. параметр i можно совсем убрать. Окончательно получаем:

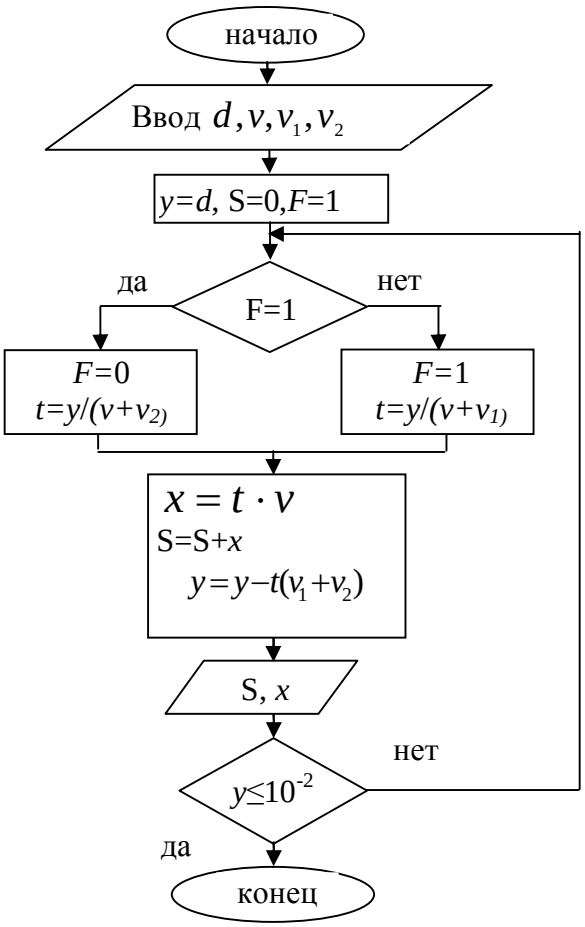


Рис. 4.8. Окончательная блок-схема алгоритма

Будем считать, что алгоритм больше не упростить. В следующей главе, когда будем изучать язык программирования `Pascal`, мы напишем программу для этого алгоритма (см. главу 5, раздел 5.2).

4.1.3. "Вместо лирического отступления"

Какие выводы можно сделать из анализа этой задачи с точки зрения разработки алгоритма или, как еще говорят, алгоритмизации данной задачи?

Во-первых, крайне редко и для очень простых задач удается сразу и безошибочно записать алгоритм. В большинстве случаев приходится, как мы видели, начинать с самой общей и несколько "расплывчатой" формулировки алгоритма. В любом случае стоит даже этот алгоритм записать в виде блок схемы (рис. 4.2). Потому что одно из несомненных достоинств записи алгоритма в виде блок-схемы заключается в том, что это позволяет увидеть структуру алгоритма.

Постепенно мы детализировали алгоритм, с каждым шагом улучшая и совершенствуя его. Кстати, метод, которым мы воспользовались, так и называется *метод пошаговой детализации*.

Во-вторых, многие молодые, особенно начинающие, программисты часто вообще игнорируют этап составления алгоритма, и едва успев прочесть условие задачи, сразу садятся за компьютер и начинают писать программу. Встречаются, конечно, сверх одаренные люди, которые могут позволить себе это, и то – не всегда. Но таких – единицы! Большинству "смертных" приходится пошагово разрабатывать алгоритмы и записывать их в том или ином виде.

В-третьих, автор многократно наблюдал такую картину – студент получает задание, записывает условие задачи себе в тетрадь и … застывает как сфинкс! Проходит минут пять, десять, двадцать, … много проходит времени, а он даже не шелохнется, не в силах предложить что-нибудь путное! Потом, когда спрашиваешь у него – о чем ты думал в это время, ответы неопределенные, чаще всего типа "не знал с чего начать".

Помимо стандартного ответа – "если не знаешь с чего начать, начни с начала", что еще можно посоветовать:

1. Еще более стандартно – больше читайте книг, скрупулезно разбирайте

примеры, приведенные в них. "Горе тому, кто читает только одну книгу" (Джордж Герберт).

2. Одного чтения и "понимания" примеров в книгах недостаточно. Надо самостоятельно тренироваться в составлении алгоритмов для самых разных задач. Если можно так выразиться – тренируйте "соображалку"! Лучший способ такой. Если вы видите в книге подробно разобранный пример, не спешите читать дальше. Попробуйте составить алгоритм самостоятельно! Это может отнять у вас довольно много времени. Не жалейте его! И только после полного завершения составления алгоритма можете свериться с книгой. На первых порах у вас может ничего не получаться. Не отчаивайтесь, со временем придет умение и опыт.

3. Не стесняйтесь спрашивать, перенимайте опыт у других. В сети огромное количество форумов по программированию. Зайдите в любой из них, посмотрите, какие вопросы там задаются и, опять, постарайтесь сначала сами ответить на этот вопрос и лишь после этого можете смотреть, как на этот вопрос отвечают другие. В конце книги приведены адреса нескольких сайтов, в которых имеются форумы по программированию, в частности форум, посвященный программированию в среде Lazarus.

4. У вас должна быть создана (со временем, конечно!) своя коллекция алгоритмов, собственных наработок, типовых приемов и даже готовых кусков кода, которые вы будете вставлять в будущие свои программы.

И, наконец, сплошь и рядом встречаются ситуации, когда не помогает ни опыт, ни знания. Как быть? Готовых рецептов не существует. И здесь снова хочу подчеркнуть, что программирование это творчество. А в творчестве нет, и не может быть никаких готовых рецептов. Здесь, как говорится, дело в "искре божьей"! Уместно вспомнить и высказывание Остапа Бендера – "блондин играет в шахматы хорошо, брюнет играет плохо и никакие лекции не изменят этого соотношения сил"!

4.2. Этапы подготовки задачи для решения на компьютере

Процесс решения задачи на компьютере состоит из ряда этапов, включающих как подготовку задачи к решению, так и собственно решение.

Эти этапы включают:

1. постановку задачи;
2. построение модели изучаемого процесса или явления;
3. выбор метода решения;
4. разработка алгоритма решения задачи;
5. составление программы (кодирование);
6. отладка и тестирование программы;
7. собственно вычисления;
8. анализ полученных результатов.

Рассмотрим эти этапы.

Постановка задачи.

Под постановкой задачи подразумевается определение цели или целей, которых необходимо достичь в результате решения данной задачи. В постановку задачи входит определение необходимой информации (что дано), результата решения задачи (что требуется определить) и выработка общего подхода к решению задачи.

Построение соответствующей модели изучаемого процесса или явления.

На этом этапе производится выбор из всего множества зависимостей и связей основных, определяющих тот или иной реальный процесс, явление и формирование гипотез, позволяющих представить реальный, обычно достаточно сложный процесс, в виде уже известных процессов. Моделирование предусмат-

ривает некоторую разумную абстракцию, дающую возможность с достаточной точностью представить себе реальные физические, информационные или экономические процессы.

Для адекватного отражения сути изучаемого процесса или явления приходится разрабатывать различные модели. Чаще всего используются ***информационные*** и ***математические*** модели. В информационной модели указываются наиболее значимые характеристики объекта, имеющих существенное значение для данной задачи. Например, если создается база данных таксопарка, то наиболее значимыми характеристиками такого объекта как автомобиль являются марка автомобиля, год выпуска, государственный и др.

Под математической формулировкой задачи или как ее иногда называют математической моделью, подразумевается любое математическое описание изучаемого процесса или явления в виде уравнений или неравенств. В качестве уравнений могут быть алгебраические и трансцендентные уравнения, системы линейных алгебраических уравнений, дифференциальные уравнения, интегральные уравнения и т.д.

К математическому описанию предъявляются, в общем, противоречивые требования. С одной стороны математическое описание должно быть полным, с другой стороны желательно, чтобы математические зависимости были проще.

Выбор метода решения.

Выбор метода решения зависит от вида модели, постановки задачи и возможностей имеющихся средств вычислительной техники. Многие задачи можно решить разными методами и способами, поэтому актуальным становится выбор оптимального метода. Причем критерии оптимальности даже для одной и той же задачи могут быть разными.

Например, разработать базу данных таксопарка с максимальным количеством сервисных функций и процедур, причем руководство таксопарка готово закупить и установить столько компьютеров, сколько необходимо для успешного функционирования этой базы.

Или за неимением достаточных средств, руководство таксопарка согласно установить только один компьютер в планово-экономическом отделе. Ясно, что базы данных в первом и втором случаях будут существенно разниться по своим функциональным возможностям и по методам, использованными разработчиками для реализации этих баз данных.

Поскольку компьютеры оперируют с числами, то в качестве методов решения математических моделей обычно применяются численные методы. Преобразование математических выражений, характерное для классической математики, не является типичным при применении компьютеров. Хотя в последние годы появились мощные программные системы типа MAPLE, которые позволяют производить и символьные вычисления. В то же время численные методы часто позволяют решать задачи, которые методами классической математики обычно неразрешимы.

Разработка алгоритма решения задачи.

Характер работы на этом этапе существенно зависит от предыдущих этапов. Кроме того, имеет значение и размер задачи. Если это достаточно простая задача, с которой может справиться один человек, то работа может идти примерно так, как в примере, рассмотренном в предыдущем разделе (задача о поездах и мухе).

Для средних и крупных проектов, для реализации которых могут потребоваться группа или даже целый коллектив программистов, возникают проблемы определения методологии проектирования, планирования и, распределения работ между членами группы, их взаимодействия и пр. Важное значение имеет выбор средств проектирования и разработки ПО. В настоящее время широкое распространение получили так называемые RAD-системы (Rapid Application Development – быстрая разработка приложений). В качестве примера приведу такие системы так Microsoft Visual Studio 2008, Code Gear RAD Studio 2009, Embarcadero RAD Studio 2010 в которых имеются развитые средства для разработки ПО в коллективе.

Составление программы (кодирование).

Под этим этапом подразумевается непосредственная запись полученных ранее алгоритмов на выбранном языке программирования. Современные системы программирования позволяют значительно облегчить этот процесс, хотя этот этап по-прежнему остается одним из самых трудоемких. Разумеется, этот этап предполагает хорошее знание того языка программирования, на котором ведутся работы.

В последующих разделах книги мы собственно и займемся изучением одного из самых популярных языков программирования, каким является язык `Pascal`.

Отладка и тестирование программы.

Под этим понимается поиск и исправление ошибок в программе. Причем под отладкой понимается исправление ошибок непосредственно в процессе кодирования. Огромную помощь программисту в этом деле оказывает компилятор, который указывает программисту место возникновения ошибки и характер ошибки. Однако факт того, что компилятор не сообщил об ошибке и программа стала работать, еще не гарантирует от отсутствия ошибок. Это так называемые логические ошибки или ошибки времени исполнения. Для выявления таких ошибок разрабатываются система тестов – специальным образом подобранные контрольные примеры, для которых решение задачи известно.

В крупных проектах программы подразделяются на версии. Альфа-версия это первая работоспособная версия программы. Бета-версия это версия или версии, которые передаются заказчику для дополнительного тестирования в уже реальных условиях функционирования программы.

4.3. Примеры разработки алгоритмов

4.3.1. Решение квадратного уравнения.

Найти корни квадратного уравнения $AX^2+BX+C=0$, коэффициенты A, B, C заданы и вводятся с клавиатуры.

Из элементарной математики известна формула для нахождения корней этого уравнения:

$$X_{1,2} = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}, \qquad (4.8)$$

Однако эта формула применима только для случая действительных корней. Но мы считаем, что коэффициенты A, B, C могут быть произвольными, поэтому необходимо произвести анализ задачи и определить возможные варианты вычислений. Анализ задачи и определение возможных ситуаций, возникающих в ходе, вычислений является одной из важнейших функций программиста. Попытка запрограммировать только формулу (4.8) может привести к неопределенной ситуации, если A=0, или B^2-$4AC$<0. Именно программист должен предусмотреть возможность возникновения таких ситуаций и явным образом указать порядок вычислений в каждом конкретном случае.

Если A=0, это означает, что исходное уравнение выродилось в линейное $BX+C$=0. В этом случае решением его будет

$$X = -\frac{B}{C}, \qquad (4.9.)$$

Если дискриминант B^2-$4AC$<0, уравнение будет иметь комплексные сопряженные корни. Каждое комплексное число можно представить парой действительных чисел, одно из которых изображает действительную часть, другое - мнимую часть комплексного числа.

Действительные части обоих корней равны.

$$\mathrm{Re}_{x1} = \mathrm{Re}_{x2} = -\frac{B}{2A}\ , \qquad (4.10)$$

А мнимые будут иметь разные знаки, и вычисляться по формуле

$$\mathrm{Im}_{x1} = \frac{\sqrt{-(B^2 - 4AC)}}{2A}\ , \qquad \mathrm{Im}_{x2} = -\mathrm{Im}_{x1}\ , \qquad (4.11)$$

Исходя из этих рассуждений, нетрудно составить блок-схему алгоритма вычисления корней квадратного уравнения:

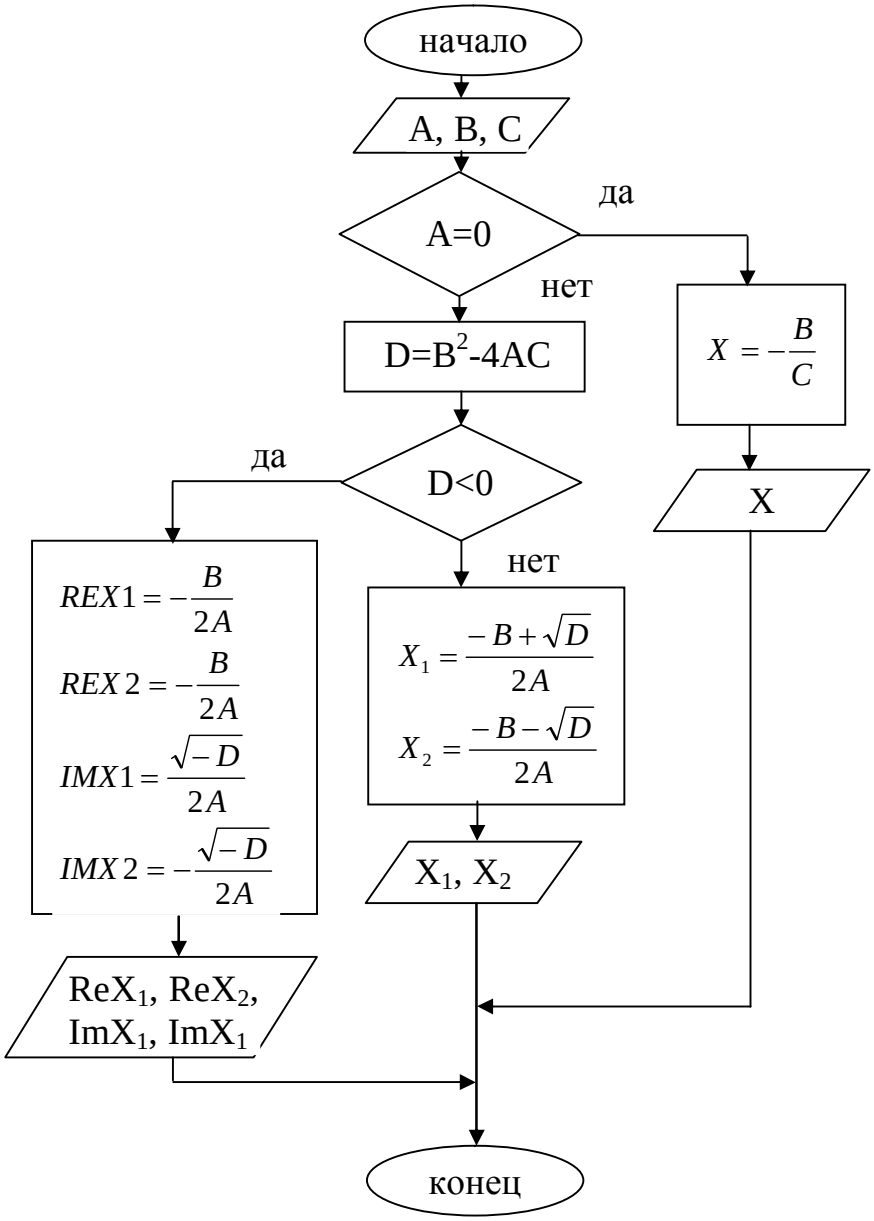


Рис. 4.9. Алгоритм вычисления корней квадратного уравнения

4.3.2. Вычисление интегралов

Вычислить интеграл $\int_a^b f(x)dx$ по формуле Симпсона с точностью $\varepsilon = 10^{-5}$.

Формула Симпсона, как известно, имеет вид [1, 2]:

$$\int_a^b f(x)dx \cong \frac{b-a}{n \cdot 3}(y_0 + y_n + 2(y_2 + y_n + ... + y_{n-2}) + 4(y_1 + y_3 + ... + y_{n-1})) \ , \qquad (4.12)$$

Для достижения требуемой точности применим метод двойного пересчета, суть которого заключается в следующем. Пусть n=4 – число точек разбиения интервала (a, b).

Вычисляем интеграл I_4. Затем увеличиваем n в два раза, (n=8) и вычисляем I_8.

Если $|I_4-I_8| \le \varepsilon$,то требуемая точность достигнута. В качестве результата берем I_8. Если же $|I_4-I_8| > \varepsilon$, то снова увеличиваем n в два раза (n=16) вычисляем I_{16}, затем если $|I_8-I_{16}| \le \varepsilon$, то точность достигнута. Если нет, то повторяем вышеуказанный процесс до достижения требуемой точности. Блок-схема алгоритма вычисления интеграла по формуле Симпсона методом двойного пересчета будет выглядеть следующим образом:

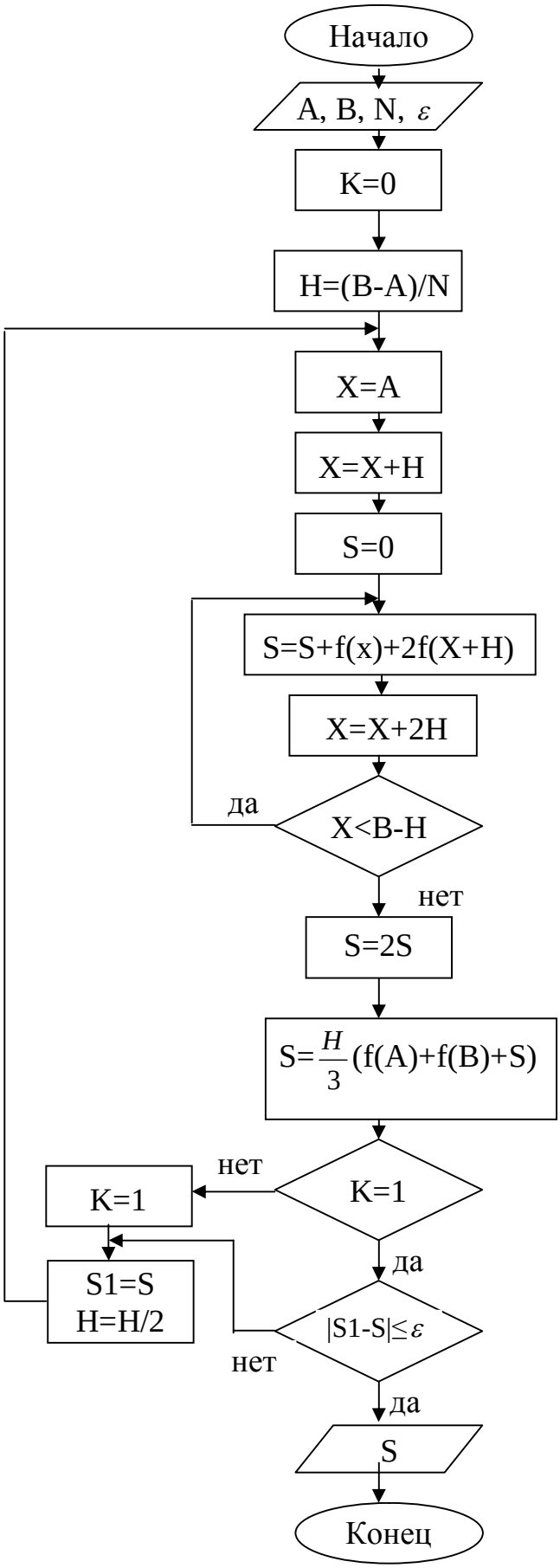


Рис. 4.10. Алгоритм вычисления определенного интеграла по формуле Симпсона

4.3.3. Обработка результатов эксперимента

При решении инженерных и экономических задач часто возникает необходимость в получении математических зависимостей между различными параметрами, характерными для данной задачи. Исходной информацией для установления этих зависимостей является физический эксперимент или экономические показатели. Как в том, так и другом случае мы располагаем либо табличными данными, либо точками на графике. Пусть имеется зависимость P_i, полученная при дискретных значениях Z_i. Значения P_i получены из эксперимента с некоторыми погрешностями. Требуется найти зависимость $P = f(Z)$.

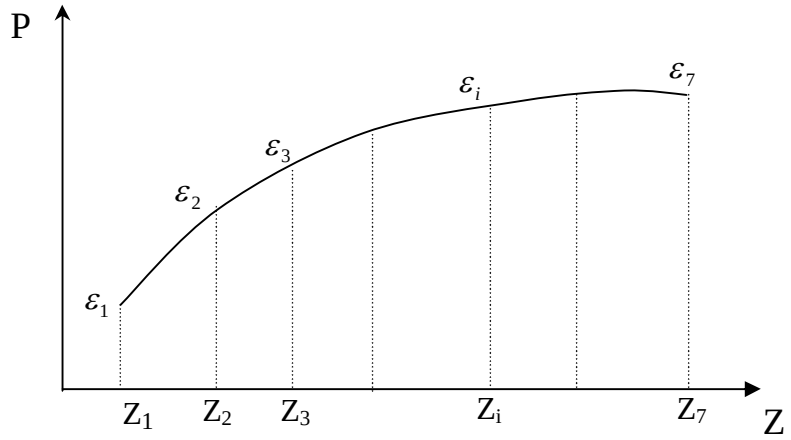


Рис. 4.11 График функции

Учитывая, что $P = f(Z)$ имеет явно выраженную нелинейную зависимость, запишем уравнение кривой второго порядка.

$$P = X_0 + X_1 Z + X_2 Z^2 \tag{4.13}$$

В этом уравнении X_0, X_1, X_2 неизвестные пока коэффициенты. Для нахождения этих коэффициентов запишем для всех имеющихся значений P_i зависимость вида (4.13).

$$P_1 = X_0 + X_1 Z_1 + X_2 Z_1^2$$

$$P_2 = X_0 + X_1 Z_2 + X_2 Z_2^2$$

$$
\begin{aligned}
&\ldots\ldots\ldots\ldots\ldots\ldots \\
&P_i = X_0 + X_1 Z_i + X_2 Z_i^{\,2} \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
&P_7 = X_0 + X_1 Z_7 + X_2 Z_7^{\,2}
\end{aligned}
\qquad (4.14
$$

Получена система из 7 уравнений с 3 неизвестными. Необходимо таким методом найти X_0, X_1, X_2, чтобы зависимость (4.13) лучшим способом описала результаты, представленные на графике.

Для нахождения трех неизвестных предстоит решить систему из 7 уравнений. Если мы отбросим какие-либо 4 лишних уравнений, мы найдем значения неизвестных без учета этих отброшенных уравнений. С другой стороны, система (4.14) может быть несовместной, т.е. при ее решении мы можем не получить тождества и при подстановке найденных значений неизвестных в уравнения системы получим разницу между левой и правой частями.

Обозначим эти разницы в соответствии с номерами уравнений через $\varepsilon_1, \varepsilon_2 \ldots \varepsilon_i, \ldots, \varepsilon_7$ и будем называть их невязками. Невязка представляет собой разницу между аналитической зависимостью и значениями P_i, заданными в качестве исходной информации в дискретных точках Z_i.

Для того чтобы аналитическая зависимость наиболее полно отражала результаты эксперимента, будем минимизировать величину:

$$S = \sum_{i=1}^{7} \varepsilon_i^2 \qquad (4.15)$$

Невязки взяты в квадрат для того, чтобы любая невязка получалась с одним положительным знаком, при этом соотношения малых и больших невязок увеличатся. Минимизация S будет выражать наилучшее приближение аналитической зависимости к экспериментальным точкам (при заданной степени полинома). Рассмотренный нами метод носит название метода наименьших квадратов [3].

Общая формулировка задачи:

необходимо решить систему n-линейных уравнений с m неизвестными.

$$\begin{aligned}
&a_{11}x_1+a_{12}x_2+\ldots+a_{1j}x_j+\ldots a_{1m}x_m=b_1\\
&a_{21}x_1+a_{22}x_2+\ldots+a_{2j}x_j+\ldots a_{2m}x_m=b_2\\
&\\
&a_{i1}x_1+ a_{i2}x_2+\ldots+a_{ij}x_j +\ldots a_{im}x_m =b_i\\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\
&a_{n1}x_1+a_{n2}x_2+\ldots+a_{nj}x_j+\ldots a_{nm}x_m=b_n
\end{aligned} \tag{4.16}$$

Запишем i-тое уравнение в более компактном виде:

$$\sum_{j=1}^{m} a_{ij}x_j = b_i \tag{4.17}$$

тогда $$S = \sum_{i=1}^{n} \varepsilon_i^2 = \sum_{i=1}^{n} (\sum_{j=1}^{m} a_{ij} x_j - b_i)^2 \tag{4.18}$$

Для минимизации S возьмем от этой величины частные производные по каждой переменной x_j и приравняем к 0.

$$\frac{\partial s}{\partial x_j} = 2\sum_{i=1}^{n} (\sum_{j=1}^{m} a_{ij} x_j - b_i) a_{ij} \ , \tag{4.19}$$

$\frac{\partial s}{\partial x_j} = 0$, отсюда:

$$\sum_{i=1}^{n} (\sum_{j=1}^{m} a_{ij} x_j - b_i) a_{ij} = 0, \tag{4.20}$$

Таких уравнений будет столько, сколько неизвестных x_j и получим систему n-линейных алгебраических уравнений с n неизвестными, которые решаются методом исключения с выделением главного элемента.

4.3.4. Решение системы линейных алгебраических уравнений

Будем рассматривать систему из n уравнений с n неизвестными. Методы

численного решения систем линейных уравнений подразделяются на две группы: прямые (конечные) и итерационные (бесконечные). Естественно, никакой практический метод решения не может быть бесконечным. Мы имеем в виду только то, что прямые методы могут в принципе (с точностью до ошибок округления) дать такое решение, если оно существует, с помощью конечного числа арифметических операций. С другой стороны, при использовании итерационных методов, для получения точного решения теоретически требуется бесконечное число арифметических операций. Значит, при практическом исследовании итерационных методов появляются ошибки ограничения. Это не значит, что прямые методы лучше, т.к. ошибки округления, появляющиеся в прямых методах, играют большую роль. В некоторых случаях из-за ошибок округления могут быть получены бессмысленные результаты. Несмотря на неизбежную ошибку ограничения, итерационные методы могут оказаться наиболее удобными, т.к. при его использовании ошибки округления не накапливаются.

Рассмотрим один из прямых методов называемых методом исключения (метод Гаусса).

Для иллюстрации метода рассмотрим систему из 3 уравнений с 3 неизвестными:

$$\begin{aligned} &a_{11}x_1+a_{12}x_2+a_{13}x_3=b_1 \ (1)\\ &a_{21}x_1+a_{22}x_2+a_{23}x_3=b_2 \ (2)\\ &a_{31}x_1+a_{32}x_2+a_{33}x_3=b_3 \ (3) \end{aligned} \qquad (4.21)$$

Здесь хотя бы один из коэффициентов a_{11}, a_{21}, a_{31} должен быть отличен от 0, иначе мы бы имели дело с 3 уравнениями с двумя неизвестными. Пусть $a_{11}\neq 0$, если, это не так мы можем переставить местами уравнения, так чтобы коэффициент при x_1 в первом уравнении был отличен от 0. Перестановка уравнений систему не изменит. Теперь введем множитель:

$$m_2 = \frac{a_{21}}{a_{11}} \qquad (4.22)$$

Умножим 1-е уравнение (4.21) на m_2 и вычтем его из 2-го уравнения (4.21). Имеем:

$$(a_{21} - m_2 a_{11})x_1 + (a_{22} - m_2 a_{12})x_2 + (a_{23} - m_2 a_{13})x_3 = b_2 - m_2 b_1 \qquad (4.23)$$

Но $$a_{21} - m_2 a_{11} = a_{21} - \frac{a_{21}}{a_{11}} a_{11} = 0 \qquad (4.24)$$

Обозначим

$$\begin{aligned} a'_{22} &= a_{22} - m_2 a_{12} \\ a'_{23} &= a_{23} - m_2 a_{13} \\ b'_2 &= b_2 - m_2 b_1 \end{aligned} \qquad (4.25)$$

Тогда 2-е уравнение (4.22) приобретет вид:

$$a'_{22} x_2 + a'_{23} x_3 = b'_2 \qquad (4.26)$$

Заменим это уравнение в (4.21) уравнением (4.26), получим систему:

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 && (1) \\ &a'_{22} x_2 + a'_{23} x_3 = b'_2 && (4) \\ &a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 && (3) \end{aligned} \qquad (4.27)$$

Умножим теперь (1) в (4.27) на $m_3 = \frac{a_{31}}{a_{11}}$ и вычтем из (3)

$$\begin{aligned} a'_{32} &= a_{32} - m_3 a_{12} \\ a'_{33} &= a_{33} - m_3 a_{13} \\ b'_3 &= b_3 - m_3 b_1 \end{aligned}$$

Уравнение (3) приобретает вид:

$$a'_{32}x_2 + a'_{33}x_3 = b'_3 \qquad (5)$$

И исходная система (4.21) теперь имеет вид:

$$\begin{aligned} &a_{11}x_1+a_{12}x_2+a_{13}x_3=b_1 && (1) \\ &a'_{22}x_2 + a'_{23}x_3 = b'_2 && (4) \\ &a'_{32}x_2 + a'_{33}x_3 = b'_3 && (5) \end{aligned} \qquad (4.28)$$

Эти новые уравнения эквивалентны исходным, с тем преимуществом, что x_1 не входит ни во второе, ни в третье уравнение системы.

Попытаемся теперь исключить x_2 из уравнений (4) и (5).

Если $a'_{22} = 0$, то мы вновь снова переставим местами уравнения, так чтобы $a'_{22} = 0$. Если же $a'_{22} = 0$ и $a'_{32} = 0$, то система вырождена и либо не имеет решения, либо имеет бесконечное множество решений. Введем новый множитель $m_3' = \frac{a'_{32}}{a'_{22}}$. Умножим его на (4) и вычтем его из (5)

$$(a'_{32} - m'_3 a'_{22})x_2 + (a'_{33} - m'_3 a'_{23})x_3 = b'_3 - b'_2 m'_3 \qquad (4.29)$$

В силу выбора m'_3

$$\begin{aligned} &a'_{32} - m'_3 a'_{22} = 0 \\ &a''_{33} = a'_{33} - m'_3 a'_{23} \\ &b''_3 = b'_3 - b'_2 m'_3 \end{aligned} \qquad (4.30)$$

Уравнение (4.29) запишется в виде

$$a''_{33}x_3 = b''_3 \qquad (4.31)$$

Уравнение (4.29) можно заменить уравнением (4.31).

Система (4.21) приобретает вид:

$$\begin{aligned} a_{11}x_1+a_{12}x_2+a_{13}x_3=b_1 \quad (8) \\ a'_{22}x_2 + a'_{23}x_3 = b'_2 \quad (9) \\ a''_{33}x_3 = b''_3 \quad (10) \end{aligned} \tag{4.32}$$

Решение этой системы совершенно очевидно.

$$\begin{aligned} x_3 &= \frac{b''_3}{a''_{33}} \\ x_2 &= \frac{b_2 - a'_{23}x_3}{a'_{22}} \\ x_1 &= \frac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} \end{aligned} \tag{4.33}$$

Для чего мы всегда переставляем уравнения таким образом, чтобы a_{11}, a'_{22}, a'_{33} были не равны 0? Чтобы не было деления на 0!

Пример:

$$\begin{cases} x + y + z = 4 \\ 2x + 3y + z = 9 \\ x - y - z = -2 \end{cases} \tag{4.34}$$

Умножим первое уравнение (4.34) на 2 и вычтем из 2-го уравнения. Затем первое уравнение умножим на 1 и вычтем из 3-го. Получим систему, эквивалентную (4.34).

$$\begin{cases} x + y + z = 4 \\ y - z = 1 \\ -2y - 2z = -6 \end{cases} \tag{4.35}$$

Умножив второе уравнение (4.35) на (-2) и вычтя его из 3-го уравнения получаем

$$\begin{cases} x + y + z = 4 \\ y - z = 1 \\ -4z = -4 \end{cases} \tag{4.36}$$

Отсюда решением этой системы будет:

$x = 1$

$y = 2$

$z = 1$

Обобщим этот метод на случай системы из *n* уравнений с *n* неизвестными

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n \end{aligned} \tag{4.37}$$

Предполагается в силу расположения уравнений $a_{11} \neq 0$. Введем *n-1* множителей:

$$m_i = \frac{a_{i1}}{a_{11}}, \qquad i = 1, 2, 3, \ldots n \tag{4.38}$$

И вычтем из каждого *i*-го уравнения первое, умноженное на m_i. Обозначим

$$\begin{aligned} a'_{ij} &= a_{ij} - m_i a_{1j}, \\ b'_i &= b_i - m_i b_1, \end{aligned} \tag{4.39}$$

$$i=2, 3, \ldots, n, \; j=1, 2, \ldots, n$$

Для всех уравнений, начиная со 2-го $a'_{i1}=0, \; i=2, 3, \ldots n$

Получим систему

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ 0 \; + a'_{22}x_2 + \ldots + a'_{2n}x_n &= b'_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ 0 \; + a'_{n2}x_2 + \ldots + a'_{nn}x_n &= b'_n \end{aligned} \tag{4.40}$$

Продолжая таким образом, мы можем исключить x_2 из последних n-2 уравнений, x_3 из последних n-3 уравнений и т.д. На некотором k-ом этапе мы исключим x_k с помощью множителей.

$$m_i^{(k-1)} = \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}, \qquad i=k+1,\ldots n \tag{4.41}$$

Причем

$$a_{kk}^{(k-1)} \neq 0,$$
$$a_{ij}^{(k)} = a_{ij}^{(k-1)} m_i^{(k-1)} a_{kj}^{(k-1)}$$
$$b_i^{(k)} = b_i^{(k-1)} - m_i^{(k-1)} b_k^{(k-1)}$$

где $i = k+1, k+2,\ldots, n; \quad j = k,\ldots, n; \quad k = 1,\ldots, n-1$

Окончательно треугольная система уравнений записывается следующим образом.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a'_{22}x_2 + \ldots + a'_{2n}x_n &= b'_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{nn}^{(n-1)}x_n &= b_n^{(n-1)} \end{aligned} \tag{4.42}$$

Обратная подстановка для нахождения значений неизвестных задается формулами:

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$$

$$x_{n-1} = \frac{b_{n-1}^{(n-2)} - a_{n-1,n}^{(n-2)} \cdot x_n}{a_{n-1,n-1}^{(n-2)}} \qquad (4.43)$$

$$x_j = \frac{b_j^{(j-1)} - a_{j,n}^{(j-1)} \cdot x_j - \ldots - a_{j,j+1}^{(j-1)} \cdot x_{j+1}}{a_{jj}^{(j-1)}}, \quad j=n-2,\ n-3,\ldots,1$$

Блок-схема алгоритма показана на рис. 4.12. Здесь, чтобы не загромождать блок-схему, мы предположили, что коэффициенты системы уже введены. В этой блок-схеме неясно только одно – что значит переставить уравнение, как это сделать?

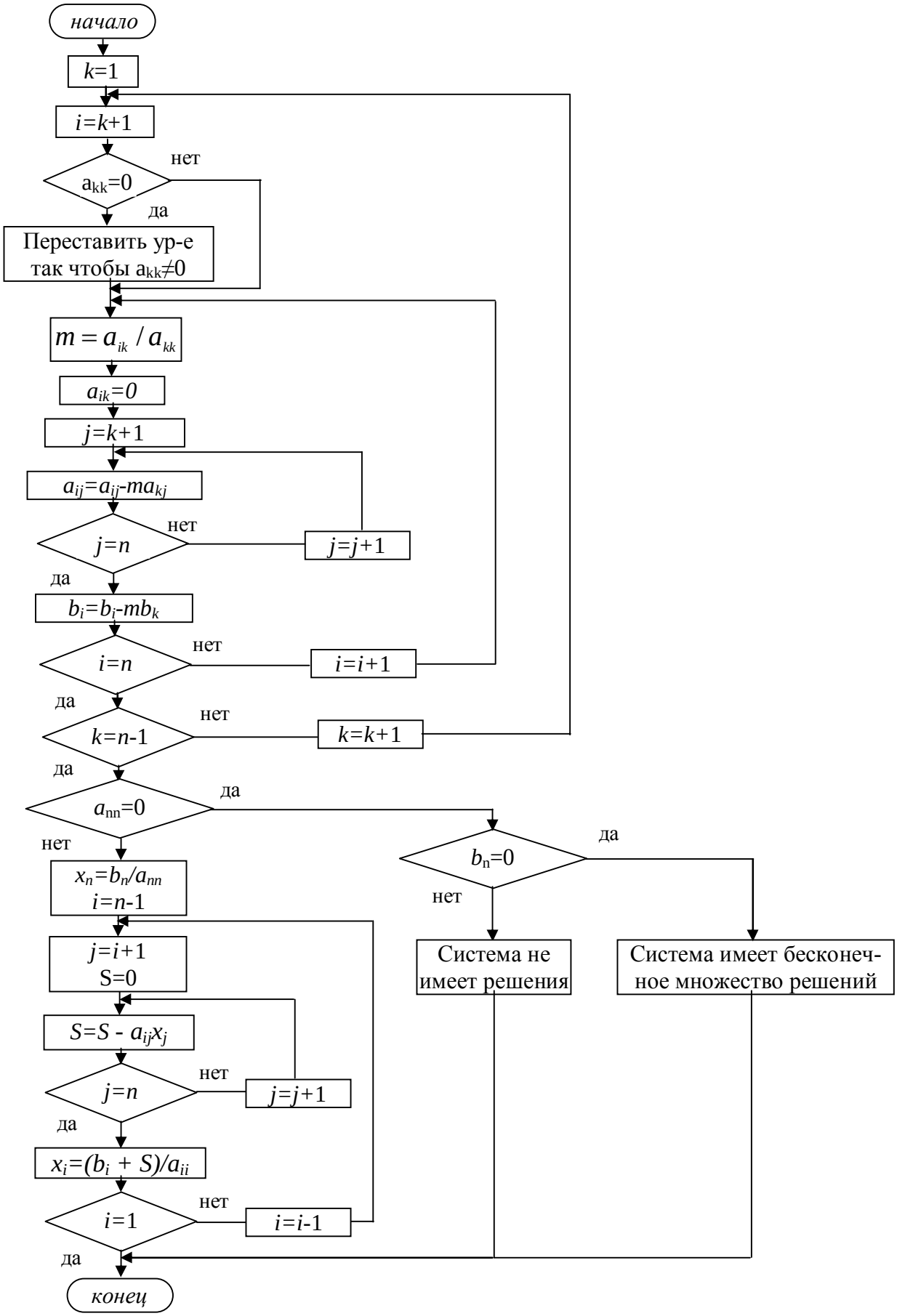


Рис. 4.12 Алгоритм Гаусса

Оказывается, что если переставить уравнения таким образом, чтобы коэффициент при x_k был наибольшим, то ошибки округления будут минимальными. Этот коэффициент называется главным элементом. И перестановка уравнений с выбором главного элемента называется методом главных элементов.

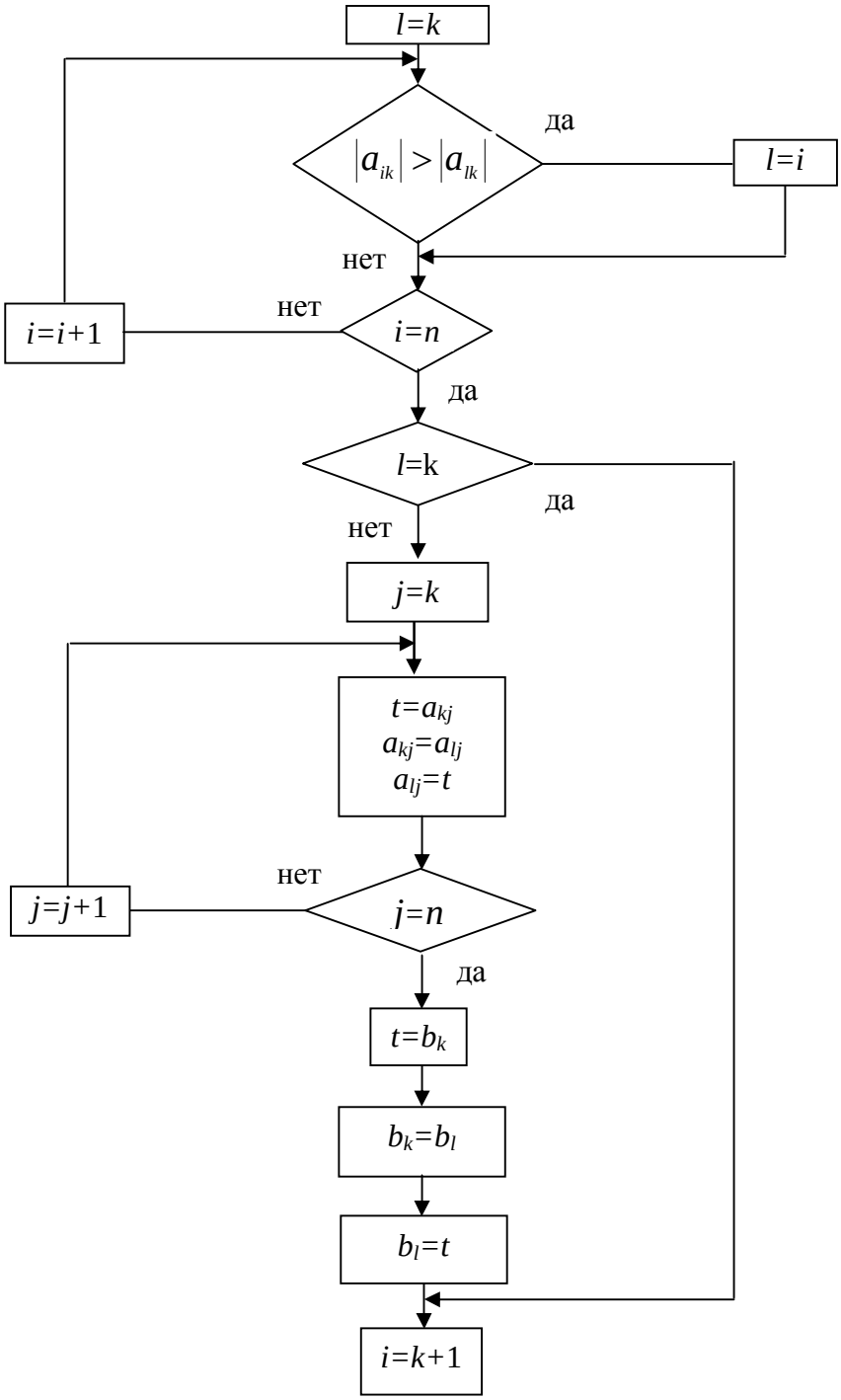


Рис. 4.13 Алгоритм перестановки уравнений

Глава V. Введение в язык программирования Pascal

Привилегия быть программистом в XX веке эквивалентна привилегии быть грамотным в XVII веке.

А.П. Ершов

5.1. Основные элементы языка

Язык `Pascal` `Паскаль`), изобретенный в начале 70-х годов 20-го века Н. Виртом и названный в честь французского математика и философа Блеза Паскаля, является одним из наиболее распространенных языков программирования. От других языков он выгодно отличается возможностью более ясно и логично записывать программы.

Программа на языке Паскаль состоит из двух частей: описание действий, которые должны быть выполнены и описание данных, над которыми они выполняются. В тексте программы описание данных предшествует описанию действий. В этом выражается общее правило языка – каждый встречающийся в программе объект должен быть предварительно описан.

Описание данных состоит из описания переменных. Операторами называются действия над данными. В общем виде любая Паскаль – программа имеет вид:

```
заголовок программы
раздел описания переменных
раздел операторов
```

Заголовок программы имеет вид:

`program имя программы;`

Здесь слово "`program`" – это так называемое ключевое (или служебное или еще говорят зарезервированное) слово. Оно должно записываться именно так (без кавычек), а не иначе. Допускается использовать как строчные, так и прописные буквы. Записи `PROGRAM`, `Program`, `ProgRam` – разрешены и означают одно и то же.

Так начинаются все программы, написанные на языке Паскаль. Здесь нечего понимать, просто так принято разработчиком языка.

В принципе допускается не использовать заголовок `program`, но лучше все-таки начинать программу именно с заголовка!

Имя программы – это любая последовательность букв, цифр и некоторых знаков, начинающихся с буквы. Такие последовательности называются ***идентификаторами***. Идентификатор состоит из 1-127 символов – букв, цифр или знаков подчёркивания, при этом первым должна быть буква или знак подчёркивания. Идентификатор не должен совпадать ни с одним из ключевых слов. В идентификаторе не должно быть (`.`) – точки, (`,`) – запятой, самих скобок `(  )`, а также пробелов и знаков операций.

Примеры правильных идентификаторов:

`X3`

`Summa`

`VOLVO`

`Select_screen_color`

Примеры неправильных идентификаторов:

`3x` начинается с цифры

`Sum.ma` внутри идентификатора есть точка

`VOL VO` есть пробел

5.1.1. Переменные. Стандартные типы.

Каждая переменная имеет имя и тип. Имя переменной – это произвольный идентификатор. В дальнейшем будем говорить "переменная `x`", вместо "переменная с именем `x`".

Тип переменной определяет множество её возможных значений, набор допустимых операций над переменной и размер занимаемой памяти.

В Паскале существуют следующие стандартные типы переменных:

`integer` (целый), `real` (вещественный), `boolean` (логический), `char` (символьный), `string` (строковый).

Значениями переменных целого типа являются целые (и только!) числа.

Примеры целых чисел:

25 +150 -200 10000

Операции над целыми числами таковы:

`+` (сложение), `-` (вычитание), `*` (умножение), div (деление нацело), mod (остаток от деления двух целых чисел).

Значениями переменных вещественного типа являются вещественные числа. Определены следующие операции над вещественными числами:

`+` (сложение), `-` (вычитание), `*` (умножение), `/` (деление).

Запись вещественных чисел похожа на общепринятую, только вместо запятой используется точка и вместо степени 10 используется буква `E`.

Пример:

Таблица 5.1

Общепринятая	на Паскале
5,30	5.30
-1,0	-1.0
41000	41000 или 4.1E4
$-0{,}73 \cdot 10^{-2}$	-0.73E-2

Значениями переменных логического типа является true (истина), false (ложь). Определены операции: not (не), and (и), or (или), xor (исключающее или).

Значения переменных символьного типа – одиночные символы. Для представления символов в памяти компьютера используются специальные таблицы кодирования, о которых речь пойдет позже.

Значения переменных строкового типа – цепочка символов. При записи констант символьного и строкового типа используют одиночные кавычки.

Пример.

`'A'` - это символ A

`'Это цепочка символов'`

5.1.2. Операции отношения

Существуют следующие операции отношения:

= равно, <> не равно, < меньше, > больше, <= меньше или равно, >= больше или равно.

Результатом этих операций являются логические значения true или false.

5.1.3. Раздел описаний переменных

Этот раздел имеет вид:

`var` описание 1; описание 2; …; описание n;

`var` – ключевое слово (от английского variable – переменная)

описание имеет вид:

переменная 1, переменная 2, …, переменная m: тип;

переменная 1, переменная 2, …, переменная к: тип;

……

переменная 1, переменная 2, …, переменная s: тип;

тип – одно из ключевых слов: `integer, real, boolean, char, string`

Пример раздела описаний:

```
var a, b, c, x, y: real;
    i, j, k, m, n: integer;
    FLAG: boolean;
    symbol: char;
```

5.1.4. Выражения. Порядок выполнения операций.

Совокупность переменных и констант, соединенных знаками операций и скобками, называется ***выражением***.

Пример.

```
(a+b)/c
((n+q1)*dx+(i+j)*dy)/(x1-2)*(y1-2)
```

Правила выполнения операций в Паскале:

1. Умножение и деление выполняются раньше, чем сложение и вычитание. Говорят также, что умножение и деление имеют более высокий приоритет, чем сложение и вычитание.
2. Операции, имеющие одинаковый приоритет выполняются слева направо.

Умножение и деление имеют одинаковый приоритет, сложение и вычитание имеют также одинаковый приоритет.

Исходя из этих правил выражение

```
4/8*5
```

будет вычисляться следующим образом:

Сначала будет вычислено `4/8 (=0.5)`, а затем результат будет умножен на `5`. Получится `0.5*5=2.5`

Всякое отклонение от этих правил должно регламентироваться скобками, т.к. действия над переменными стоящими в скобках выполняются в первую очередь.

5.1.5. Константы

В Паскале есть возможность присвоить константе имя, при этом в последующем тексте программы всюду вместо этой константы можно использовать её имя. Все определения констант перечисляются в специальном разделе – разделе описания констант, имеющем вид:

```
const
    имя 1 = значение 1;
    имя 2 = значение 2;
    ..............................................
```

```
имя n = значение n;
```

Пример.

```
const
    r = 1.87E+5;
    g = 981E-2;
    atmosphere = 760;
    pi = 3.14159;
```

Давая имена константам, мы делаем программу более понятной. Запись `2*pi*r` гораздо понятнее и информативнее, нежели запись `2*3.14159*1.87E+5`

Кроме того, при внесении изменений в программу нам будет достаточно изменить только значение константы. Следует иметь в виду, что память для констант не отводится. Компилятор вставляет их значения прямо в двоичный код программы в нужные места в виде литералов.

5.1.6. Комментарии в программе

Для того чтобы программистам было легче читать и разбираться в программах, в языке предусмотрены средства для комментирования фрагментов программного кода. *Комментарием* называется некоторый пояснительный текст на обычном человеческом языке, поясняющий те или иные действия программиста.

Комментарии бывают *однострочные* и *многострочные*. Однострочный комментарий начинается с символов // и размещается только в одной строке. Например:

```
// переменная i используется как индекс
```

```
// в операторах цикла
i:= 0;  // Инициализация переменной
```

Как мы видим, комментарий можно располагать в одной строке с оператором!

Многострочный комментарий, как явствует из названия, позволяет размещать комментарии в нескольких строках. Многострочный комментарий начинается с символов (* и заканчивается символами *). В качестве ограничителей комментария используются также фигурные открывающие и закрывающие скобки {}. Например:

```
(* переменная i используется как индекс
 в операторах цикла *)
i:= 0;  (* Инициализация переменной *)
```

или

```
{ переменная i используется как индекс
 в операторах цикла }
i:= 0;  { Инициализация переменной }
```

Программисты чаще всего используют однострочный комментарий и многострочный комментарий с фигурными скобками.

5.1.7. Операторы

Основная часть программы на Паскале – раздел операторов. Он начинается ключевым словом `begin` и заканчивается ключевым словом `end`, за которым следует точка. Операторы отделяются друг от друга точкой с запятой (;). Рассмотрим основные операторы:

5.1.7.1. Оператор присваивания

Элементарное действие над переменной – изменение её значения. Для этого применяется оператор присваивания, имеющий вид:

имя переменной:= выражение;

В нем переменная и выражение должны быть одного типа.

Пример.

Пусть x – переменная целого типа. Запишем следующий оператор присваивания:

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

В левой части оператора x обозначает переменную, а в правой части – число, являющееся её текущим значением. Выполнение этого оператора приводит к увеличению значения переменной x на единицу.

5.1.7.2. Операторы ввода/вывода

Во время исполнения программы она обменивается информацией с "внешним миром". Например, она может выдавать информацию на экран или получать информацию с клавиатуры. Для этого используются операторы ввода и вывода.

Оператор вывода имеет вид:

```
write(выражение);
```

или

```
writeln(выражение);
```

В результате выполнения этого оператора значение соответствующего выражения будет выведено на экран. Выражение может быть любым из указанных выше типов.

Пример.

`write(2 + 2);` будет выведено на экран 4

`write(x = y);` будет выведено `true` или `false` в зависимости от значений `x, y`

Оператор `writeln` отличается от оператора `write` тем, что выведет значение выражения с начала новой строки, а оператор write с той позиции строки, где находится курсор.

В операторе вывода можно указывать несколько выражений, разделяя их запятыми, а также любой текст, заключенный в одинарные кавычки.

Пусть значение A=5. Тогда при выполнении оператора

```
writeln('Значение A=', A);
```

будет выведено на экран с новой строки

Значение `A=5`

Оператор ввода имеет вид:

`read(`имя переменной 1, имя переменной 2,…, имя переменной n);

`readln(`имя переменной 1, имя переменной 2,…, имя переменной n);

В результате его выполнения переменной (переменным) присваивается считанное с клавиатуры значение. Вводимое значение должно записываться при вводе так, как описана переменная, т.е. если, например, переменная целого типа, то вводимое число должно быть целым. При этом если используется `read`, то значение вводится в то место на экране, где в данный момент находится курсор, если же используется `readln,` то с новой строки и с первой колонки экрана. Если в одном операторе `readln` или `read` вводятся значения нескольких переменных, то при выполнении программы значения, водимые с

клавиатуры можно разделять пробелом или символом табуляции (клавиша **Tab)**. После ввода значения последней переменной необходимо нажать клавишу **Enter**.

На практике предпочтительнее использовать оператор `readln` т.к. во-первых, курсор будет располагаться всегда в начале строки, что позволит пользователю легче ориентироваться при вводе большого числа значений. Во-вторых, при этом после окончания ввода буфер ввода с клавиатуры полностью очищается. Дело в том, что все вводимые с клавиатуры символы сначала накапливаются в специальной временной области памяти – *буфере* и лишь после нажатия клавиши **Enter** присваиваются соответствующим переменным. При использовании оператора `read` в буфере остается код клавиши **Enter**. В некоторых случаях это может привести к неправильной работе следующего оператора ввода.

Вы, конечно, должны понимать, что при вводе чисел с клавиатуры они вводятся в виде строки символов. После нажатия клавиши **Enter** они переводятся во внутреннее представление числа (вспомните главу I).

5.1.7.3. Операторы инкремента и декремента

Кроме оператора присваивания существуют оператор инкремента и декремента, которые также позволяют изменять значения переменных целого типа.

Оператор `inc(x, n)` увеличивает значение переменной *x* целого типа на *n*. Параметр *n* может быть опущен, тогда значение *x* увеличится на единицу.

Пример.

```
inc(x, 10);
```

увеличит значение *x* на 10, а оператор

```
inc(x);
```

увеличит значение *x* на 1. Записи

```
x:= x + 10;
```

и

```
inc(x, 10);
```

совершенно идентичны по своему результату.

Оператор `dec(x, n)` уменьшает значение переменной *x* на *n*, а оператор `dec(x)` уменьшает *x* на единицу.

Программисты чаще всего используют короткие формы операторов инкремента и декремента для увеличения или уменьшения значений целочисленных переменных на единицу, т.е. могут использовать

`inc(i)` вместо `i:= i + 1` и `dec(i)` вместо `i:= i - 1`

5.1.8. Среда разработки Lazarus

Уже этих полученных знаний нам достаточно, чтобы написать простейшую программу. Для того чтобы писать и выполнять программы, нам понадобится компилятор и среда разработки. Существует довольно много компиляторов для языка *Pascal*. Мы с вами будем использовать компилятор *Free Pascal Compiler* версии 2.2.4.

Free Pascal Compiler (часто применяется сокращение *FPC*) это свободно распространяемый компилятор языка Паскаль с открытыми исходными кодами, распространяется на условиях *GNU General Public License* (*GNU GPL*). Он совместим с *Borland Pascal 7* и *Object Pascal – Delphi*, но при этом обладает рядом дополнительных возможностей, например, поддерживает перегрузку операторов. *FPC* — кроссплатформенный инструмент, поддерживающий огромное количество платформ. Среди них — *AmigaOS*, *DOS*, *Linux*, **BSD*, *OS/2*, *MacOS*(*X*) и *Win32*.

`Free Pascal` поддерживает компиляцию в нескольких режимах, обеспечивающих совместимость с различными диалектами и реализациями языка:

- **TP** — режим совместимости с `Turbo Pascal`: совместимость практически полная, за исключением нескольких моментов, связанных с тем, что `FPC`

компилирует программы для защищённого режима процессора, где невозможно прямое обращение к памяти, портам и т. д.

- **FPC** — собственный диалект: соответствует предыдущему, расширенному дополнительными возможностями, такими как, например, перегрузка операторов.
- **DELPHI** — режим совместимости с `Delphi`: включает поддержку классов и интерфейсов.
- **OBJFPC** — совмещает объектно-ориентированные возможности Delphi и собственные расширения языка.
- **MACPAS** — режим совместимости с `Mac Pascal`.
- **GNU** — режим частичной совместимости с `GNU Pascal`.

`Free Pascal Compiler` имеет свою собственную интегрированную среду разработки. Применяется также аббревиатура *IDE* (*Integrated Development Environment*). Среда имеет текстовый интерфейс очень похожий на интерфейс *Turbo Pascal 7.0*.

Но времена изменились! Текстовые интерфейсы практически полностью вытеснены так называемыми графическими интерфейсами, работать в которых значительно удобнее и приятнее.

В 1999 г. три человека - Cliff Baeseman, Shane Miller и Michael A. Hess. предприняли попытку написать бесплатную графическую среду для бесплатного компилятора *FPC*. Проект получает название **Lazarus**. На сегодняшний день следует признать, что идея оказалась весьма плодотворной потому, что среда существует и развивается и поныне.

Lazarus это бесплатный инструмент разработки с открытым кодом. IDE Lazarus представляет собой среду с графическим интерфейсом для быстрой разработки программ, аналогичную *Delphi*, и базируется на оригинальной кроссплатформенной библиотеке визуальных компонентов *LCL* (Lazarus Component Library), совместимых с *VCL Delphi*. В состав IDE входят и не визу-

альные компоненты — из FCL (Free Pascal Component Library). В принципе такого набора достаточно для создания программ с графическим интерфейсом и приложений, работающих с базами данных и Интернетом.

В среде Lazarus используются собственный формат управления пакетами и свои файлы проектов.

Lazarus это стабильная богатая возможностями среда разработки для создания самостоятельных графических и консольных приложений. В настоящее время она работает на Linux, FreeBSD и Windows и предоставляет настраиваемый редактор кода и визуальную среду создания форм вместе с менеджером пакетов, отладчиком и графическим интерфейсом, полностью интегрированным с компилятором FreePascal.

Почему для этой книги был выбран именно этот компилятор и его среда быстрой разработки? Потому что они бесплатны и распространяются по свободной лицензии GNU GPL. Кроме того, компилятор Free Pascal позволяет создавать кроссплатформенные приложения, т.е. приложения, которые могут выполняться на различных платформах. Поэтому в этой книге изложение ведется применительно и к Linux, и к Windows.

Рассмотрим основные элементы среды разработки Lazarus. Если вы еще не установили его, то предварительно установите. Дистрибутивы находятся на прилагаемом DVD диске отдельно для Linux и Windows.

Также вы можете совершенно бесплатно скачать свежие дистрибутивы по адресу http://sourceforge.net/projects/lazarus/files/

Следует заметить, что у тех, у кого `Alt Linux Master School -` Lazarus уже должен быть автоматически установлен.

Запускать Lazarus можно несколькими способами. Расскажу о некоторых, часто используемых.

В Linux в `Главном меню` должен быть пункт `Lazarus`. В зависимости от вашего дистрибутива он может находиться либо в меню `Разработка->Среды разработки->Lazarus` (Mandriva) или

`Образование->Разработка->Lazarus` (Alt Linux) или `Программирование->Lazarus` (Ubuntu).

Проще всего этот ярлык перетащить на рабочий стол. Кроме того, можно перейти в каталог установки Lazarus (чаще всего это каталог `/usr/lib/lazarus`) и дважды щелкнуть по имени файла `lazarus` или `startlazarus`, если вы используете файловый менеджер или дать команду `./lazarus` (`./startlazarus`), если вы используете консоль.

В Windows во время установки можно задать опцию "Создать значок на рабочем столе". Если вы этого не сделали, то можете запускать из меню `Пуск->Программы->Lazarus`. Вы также можете скопировать этот ярлык на рабочий стол. Наконец, можно перейти в папку установки (чаще всего C:\lazarus) и дважды щелкнуть по имени файла `lazarus.exe` или `startlazarus.exe`.

Итак, IDE Lazarus имеет вид, показанный на рис. 5.1, 5.2.

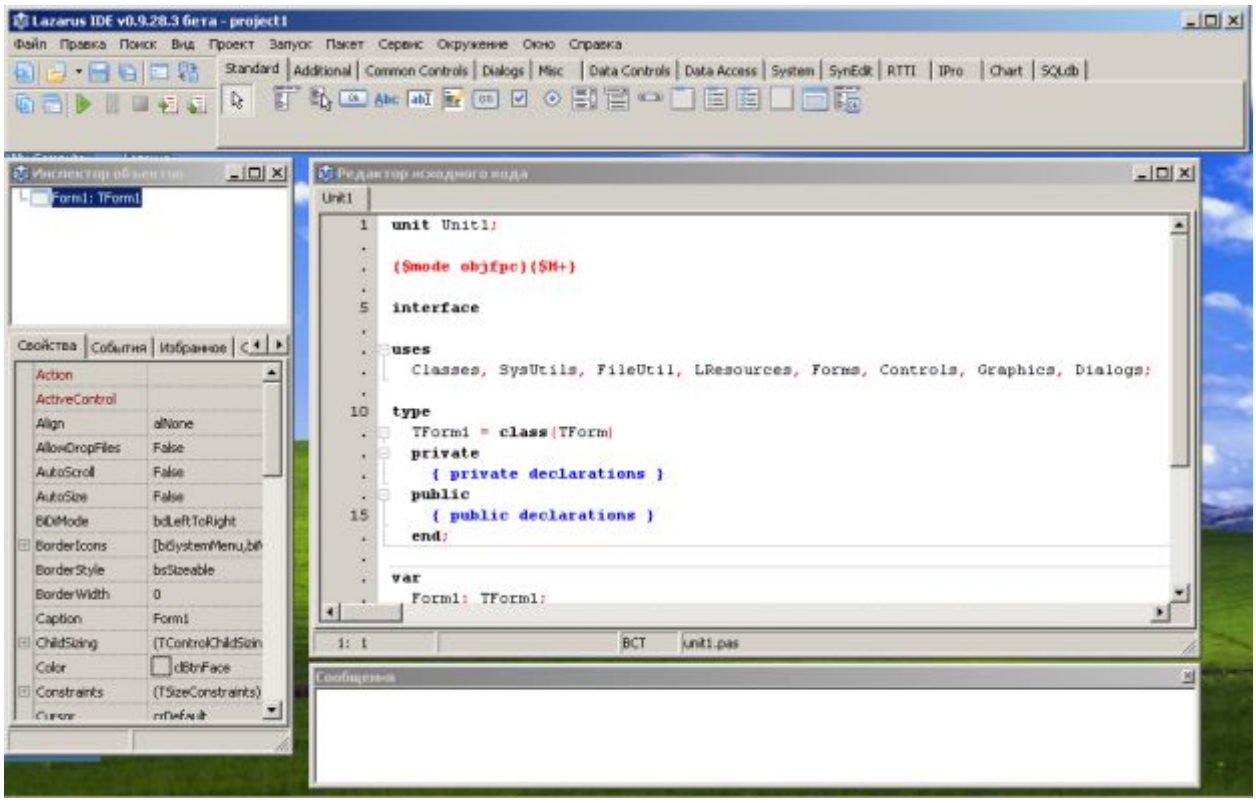


Рис. 5.1 Вид IDE Lazarus в Windows

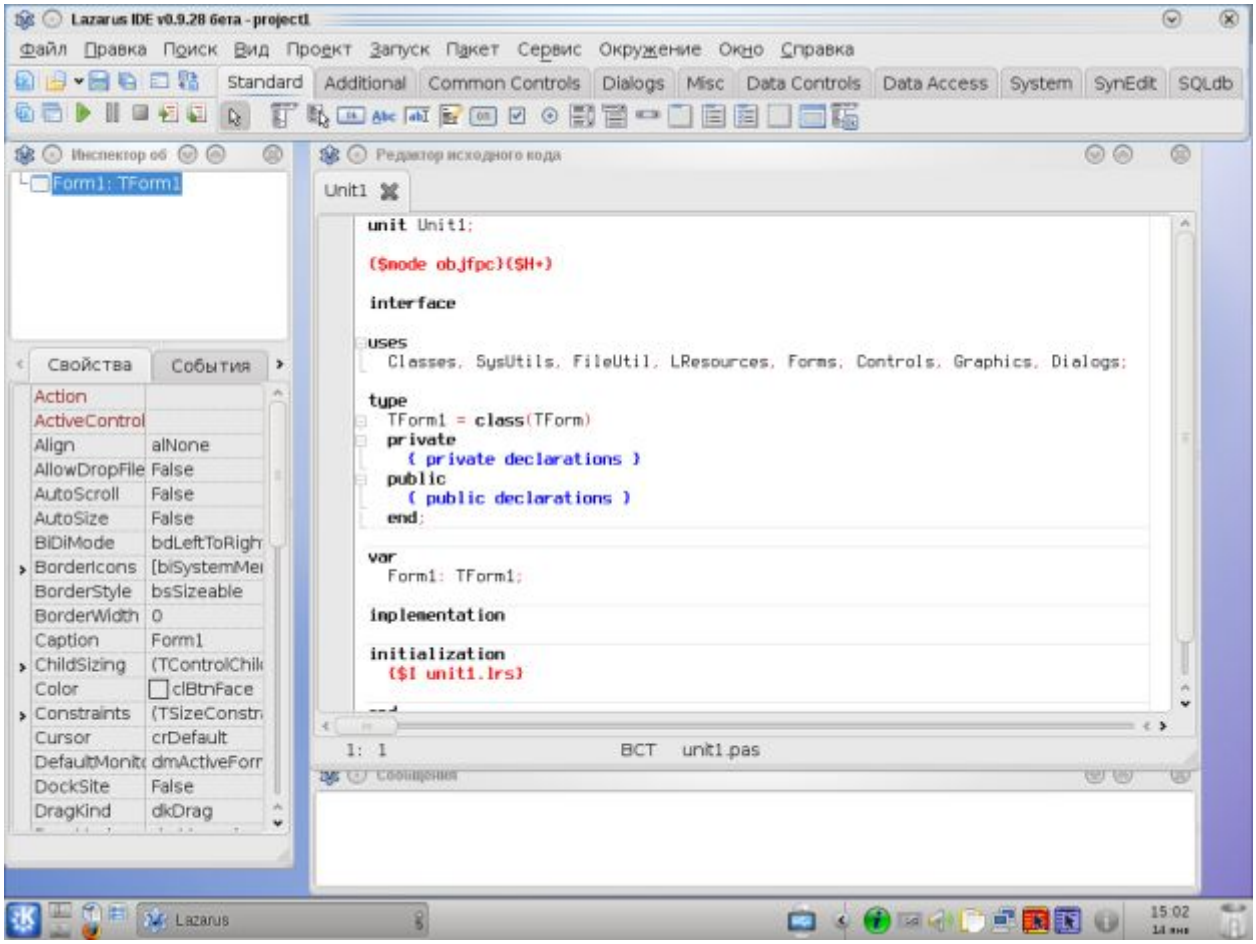


Рис. 5.2 Вид IDE Lazarus в Linux, рабочий стол KDE

Как видите, IDE Lazarus выглядит одинаково в обеих операционных системах, только цветовое оформление окон чуть разнится.

Среда Lazarus состоит из нескольких, вообще говоря, не связанных окон. Рассмотрим их по порядку.

1. **Главное окно**, рис. 5.3.

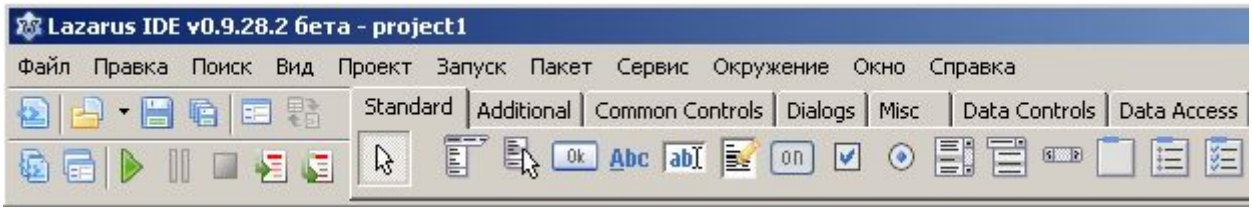


Рис. 5.3 Главное окно IDE Lazarus

С помощью этого окна можно управлять процессом разработки приложения. В нем предусмотрены команды управления файлами, компиляцией, редактированием, окнами и т.д. Окно разбито на три функциональных блока:

- **Главное меню.** В нём расположены команды управления файлами, команды управления компиляцией и свойствами всего приложения, команды управления окнами и настройками среды и многое другое. Меню располагается в верхней части основного окна.

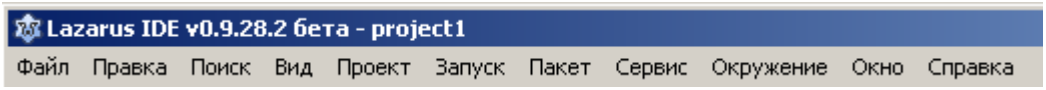


Рис. 5.4 Главное меню

- **Панель инструментов.** Панель инструментов предоставляет быстрый доступ к основным командам главного меню. Она расположена в левой части главного окна, под главным меню.

Рис. 5.5 Панель инструментов

- **Палитра компонентов.** Предоставляет доступ к основным компонентам среды разработки, например: поле ввода, надпись, меню, кнопка и т.п.

Рис. 5.6 Палитра компонентов

Инспектор объектов, рис. 5.7.

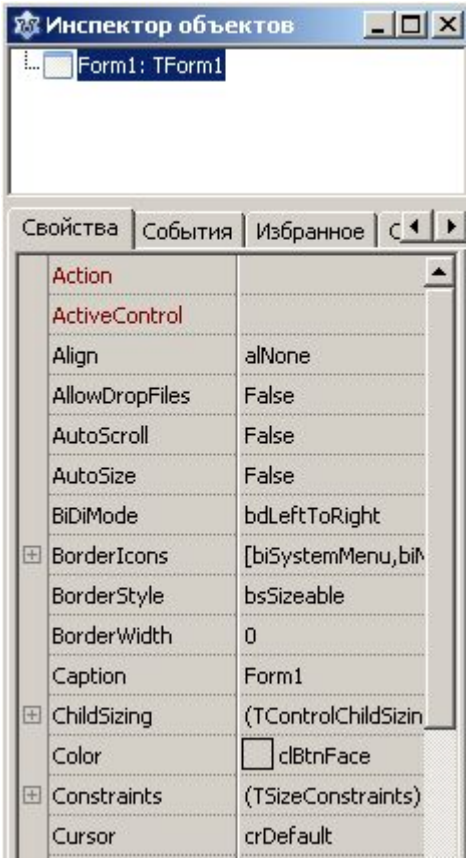


Рис. 5.7 Инспектор объектов

В верхней части окна показывается иерархия объектов, а снизу, расположены три вкладки: **Свойства**, **События**, **Избранное**. Назначение инспектора объекта – это просмотр всех свойств и методов объектов. На вкладке **Свойства** перечисляются все свойства выбранного объекта. На вкладке **События** перечисляются все события для объекта. На вкладке **Избранное** избранные свойства и методы. Подробнее об этом будет сказано в главе 9.

2. Редактор исходного кода Lazarus

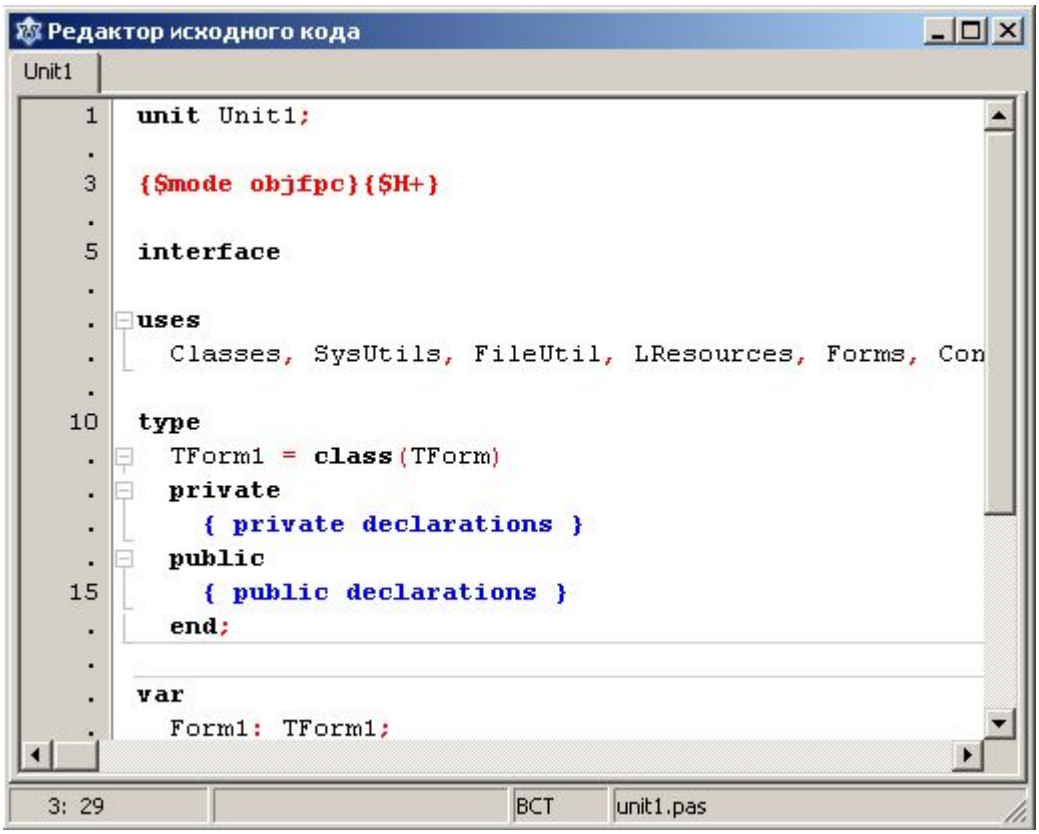


Рис. 5.8 Редактор исходного кода

Именно в этом окне мы будем набирать тексты своих программ. Многие функции и возможности этого редактора совпадают с возможностями обычных текстовых редакторов, например **Блокнота**. Текст в редакторе можно выделять, копировать, вырезать, вставлять. Кроме того, в редакторе можно осуществлять поиск заданного фрагмента текста, выполнять вставку и замену. Но, конечно, этот редактор исходных текстов Lazarus обладает еще рядом дополнительных возможностей для комфортной работы применительно к разработке программ. Основное преимущество редактора заключается в том, что он обладает возможностями подсветки синтаксиса, причём не только `Pascal`, но и других языков, а также рядом других удобств. В частности, выделенный фрагмент текста можно сдвигать вправо или влево на количество позиций, указанных в настройках Окружение ->Параметры…->Редактор -> Общие -> Отступ блока, что

очень удобно для форматирования с целью структурирования кода. Выделенный фрагмент можно закомментировать или раскомментировать, перевести в верхний или нижний регистр и т.д.

Все возможные операции в редакторе собраны в меню **Правка** и **Поиск** главного меню Lazarus, рис. 5.9, 5.10.

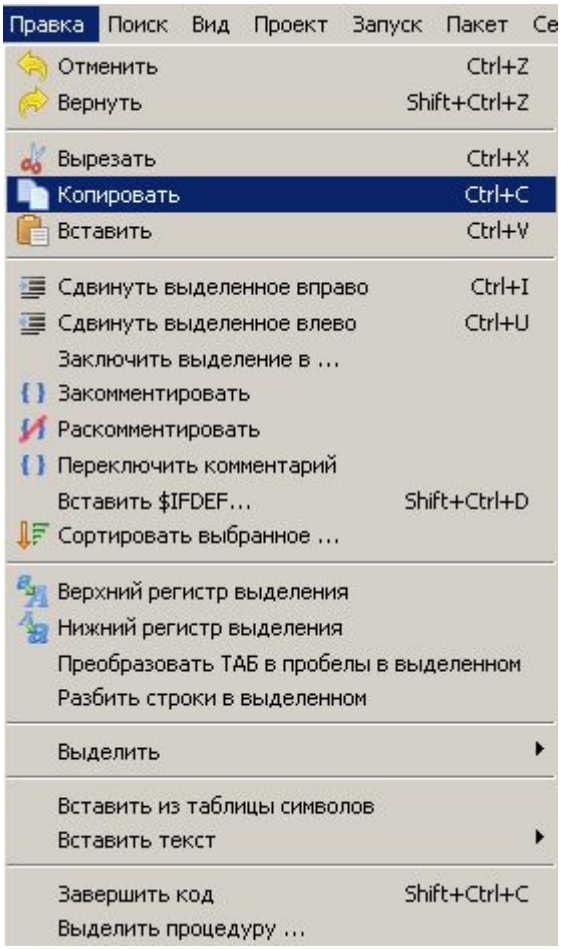


Рис. 5.9 Меню "Справка"

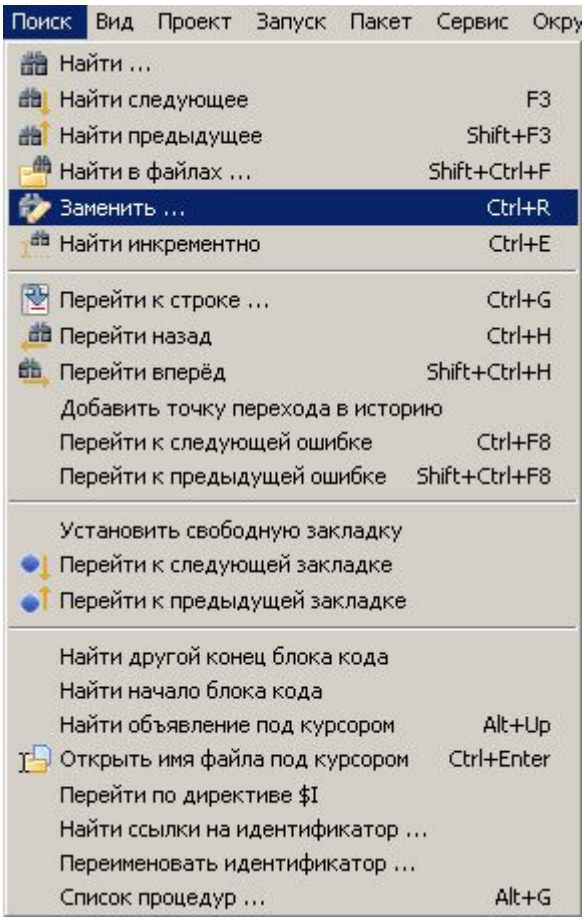


Рис. 5.10 Меню "Поиск"

3. **Окно сообщений**

В этом окне выводятся сообщения компилятора, компоновщика и отладчика.

На этом мы закончим наш краткий обзор среды Lazarus. Мы рассмотрели далеко не все виды окон IDE, да и те, что рассмотрели, мы рассмотрели бегло,

лишь для того, чтобы получить первое представление о среде Lazarus. Заниматься нудным рассказом обо всех пунктах, опциях и возможностях Lazarus я сейчас не буду. Ведь не все будет понятно, да и … скучно! Ведь вам, уважаемый читатель, "не терпится в бой"! Поэтому будем рассматривать только те элементы, которые будут нам нужны на первых порах. А, остальное мы будем изучать по мере необходимости и в нужных местах. Так, мне кажется, будет лучше!

Сначала настроим IDE так, как нам будет удобнее работать. Как уже говорилось, при первом запуске окна IDE Lazarus не связаны и представляют собой "плавающие" окна. Советую вам их соединить так, как показано на рис. 5.11. путем изменения размеров окон, чтобы Lazarus занимал весь экран.

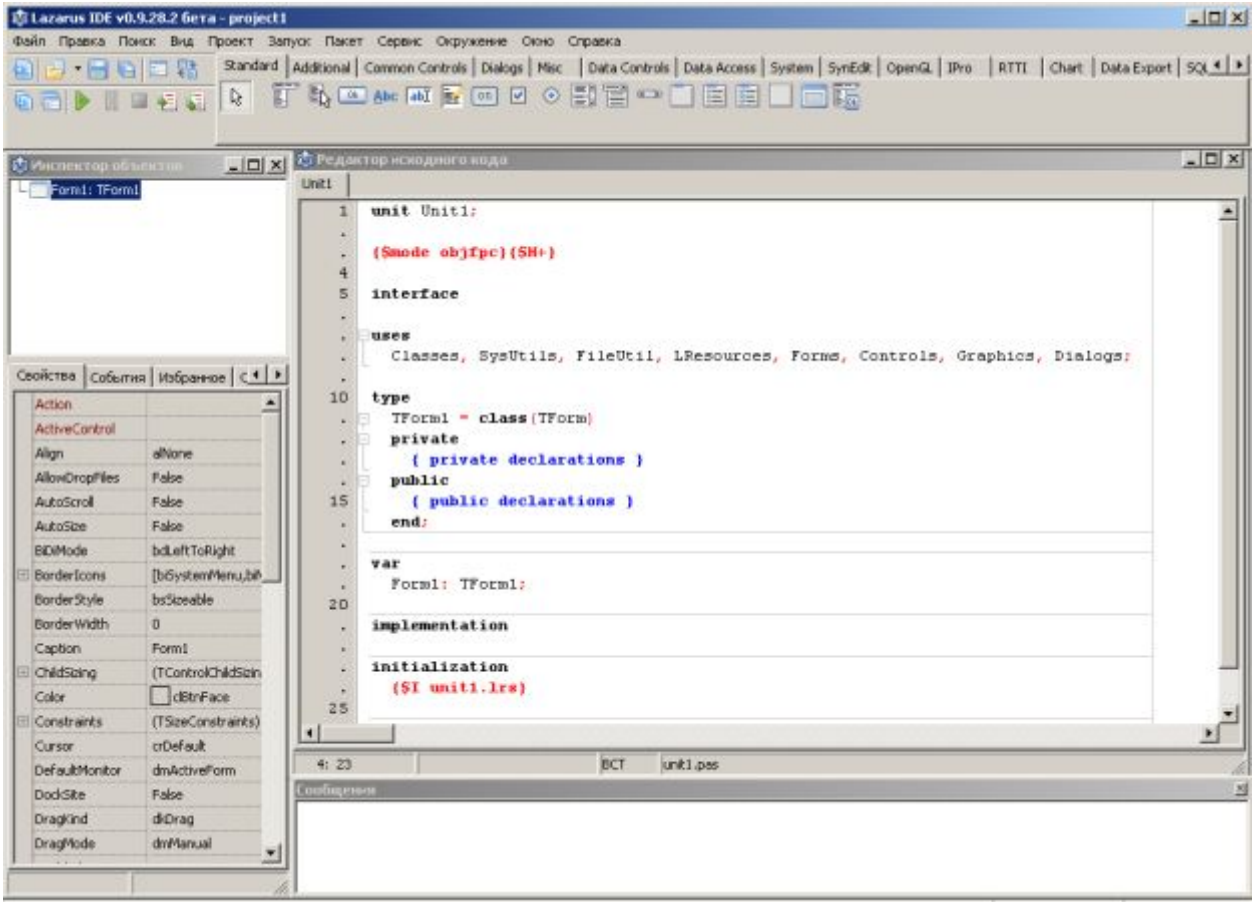


Рис. 5.11 Вид IDE Lazarus после изменения размеров и положения окон

После этого в `Главном меню` зайдите в меню `Окружение->Параметры...` Откроется окно `Параметры IDE`. В этом окне во вкладке `Окружение` выберите пункт `Окно`. Далее для каждого окна установите опцию "**Пользовательская позиция**" и нажмите кнопку "**Применить**", рис. 5.12.

В Windows все окна Lazarus после этого жестко скрепляются, поэтому, если, например, у вас Lazarus был свернут, то после разворачивания будут видны все окна. В Linux не так. Размеры и положение окон сохраняются, но вы можете открывать окна по отдельности, рис. 5.13.

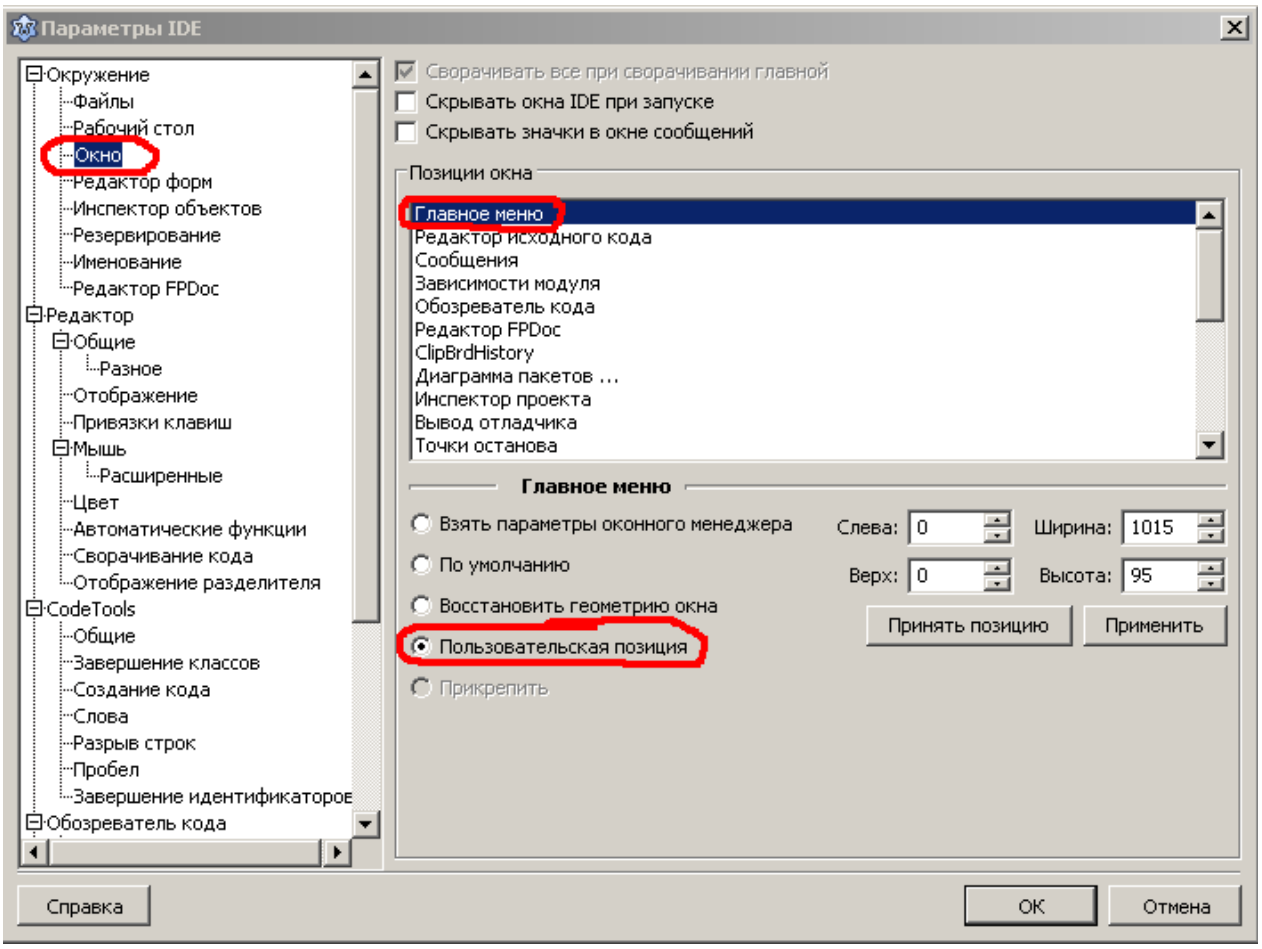


Рис. 5.12 Вкладка "Окно" меню "Окружение"

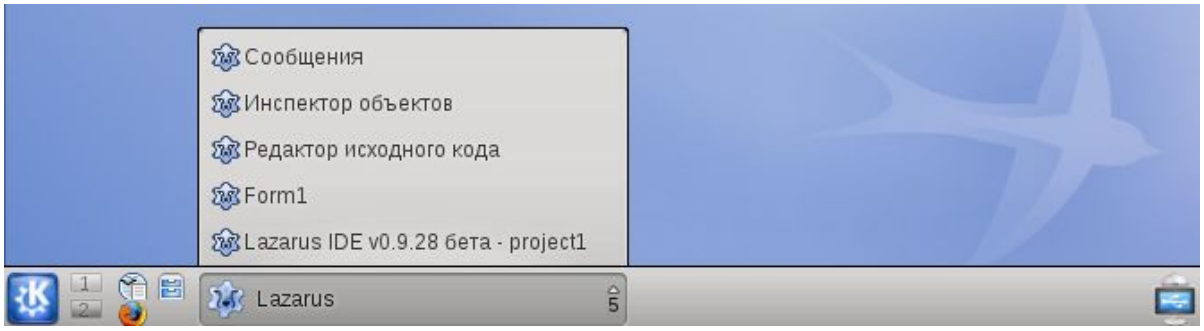


Рис. 5.13 Окна IDE Lazarus в Linux

Далее в том же окне `Параметры IDE` выберите пункт `Отладчик` и снимите галочку с опции "Показывать сообщение при остановке". Этим мы избавимся от надоедливого сообщения "Выполнение остановлено" при каждом завершении наших программ, рис. 5.14.

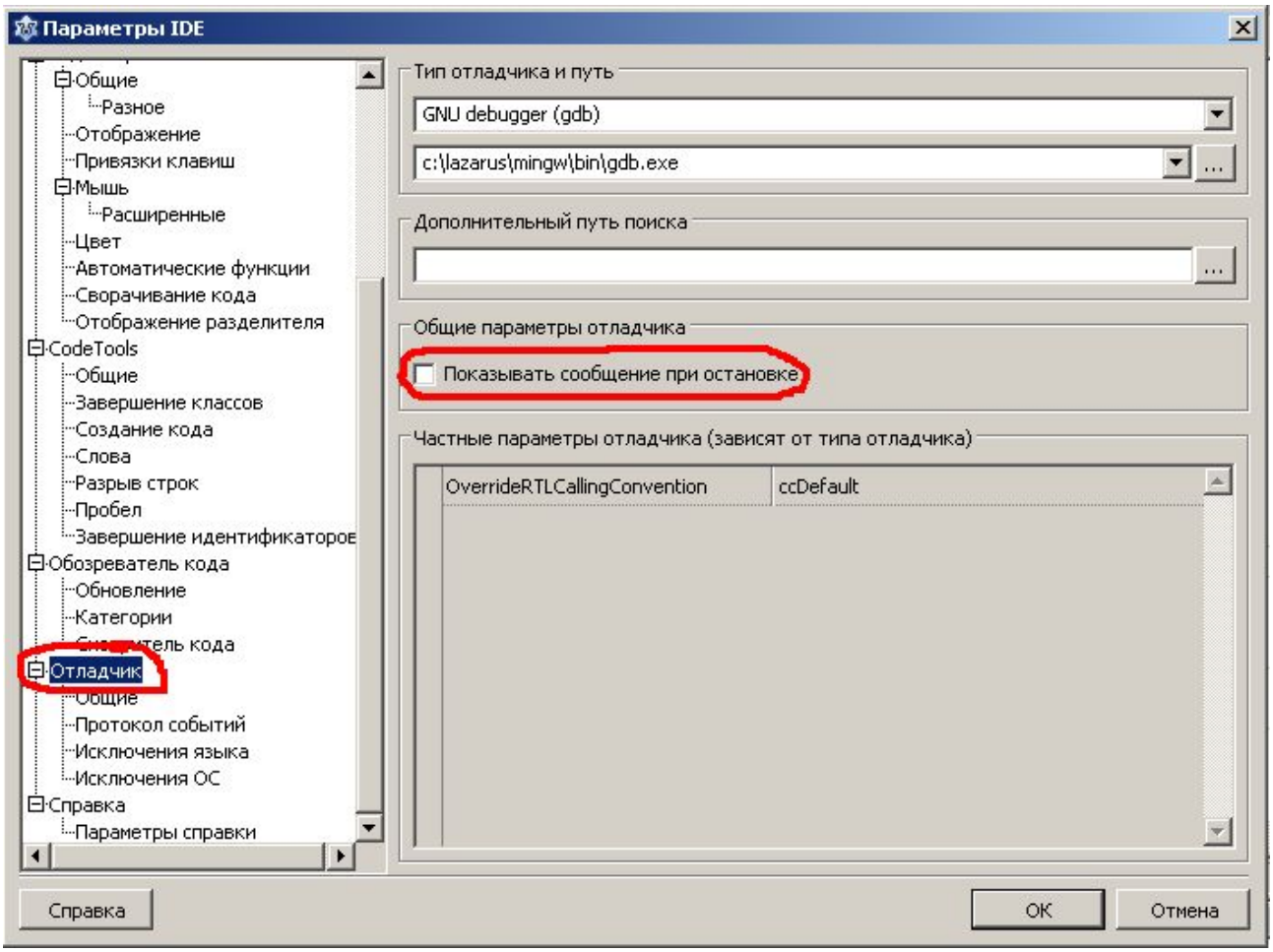


Рис. 5.14 Окно настройки параметров отладчика

Изучение языка лучше всего начинать с консольных приложений. Консольное приложение — программа, которая не имеет графического интерфейса и выполняется в текстовом режиме в консоли. В Windows консоли обычно соответствует окно командной строки. В Linux консоли соответствует окно терминала. Для таких программ устройством ввода является клавиатура, а устройством вывода — монитор, работающий в текстовом режиме отображения символьной информации (буквы, цифры и специальные знаки).

Консольное приложение позволяет сосредоточиться на существе той или иной конструкции языка, поэтому мы начнем с консольных приложений. Однако нельзя считать, что консольные приложения предназначены лишь для изучения языка. Можно и с помощью консольных приложений разрабатывать очень сложные программы. Например, компилятор Free Pascal является консольным приложением, то есть может быть запущен в консоли, из командных файлов или из IDE Lazarus.

Для создания консольного приложения необходим только текстовый редактор и компилятор Free Pascal. В принципе для этого Lazarus не нужен. Однако мы все же будем использовать Lazarus, поскольку IDE Lazarus позволяет создавать в том числе и консольные приложения. А его мощный текстовый редактор с подсветкой синтаксиса и другими широкими возможностями значительно облегчит нам написание программ.

5.1.9. Русский язык в консольных приложениях

В консольных приложениях под Windows, к сожалению, возникают проблемы с выводом на экран русских букв. Это вызвано различием в кодировке символов в Windows, которая работает в графическом режиме и ее консоли, которая работает в текстовом режиме. Не вдаваясь пока в тонкости, скажу лишь, что мы в своих программах будем использовать специальную функцию `UTF8ToConsole()`, которая позволит нам корректно отображать русские бу-

квы на экране в консольных приложениях для платформы Windows в операторах `writeln` и `write`.

Используйте функцию `UTF8ToConsole()` вот таким образом:

```
writeln(UTF8ToConsole('Русский текст'));
```

вместо

```
writeln('Русский текст');
```

В Linux таких проблем нет, поэтому можно писать просто

```
writeln('Русский текст');
```

Но мы и в Linux будем писать

```
writeln(UTF8ToConsole('Русский текст'));
```

Это позволит нам без проблем переносить программы из Linux в Windows, т.е. наши программы без каких-либо переделок будут безошибочно компилироваться и выполняться как на платформе Linux, так и на платформе Windows.

5.1.10. Первая программа

Запустите Lazarus. Выберите пункт меню **Проект, Создать проект…** (рис. 5.15).

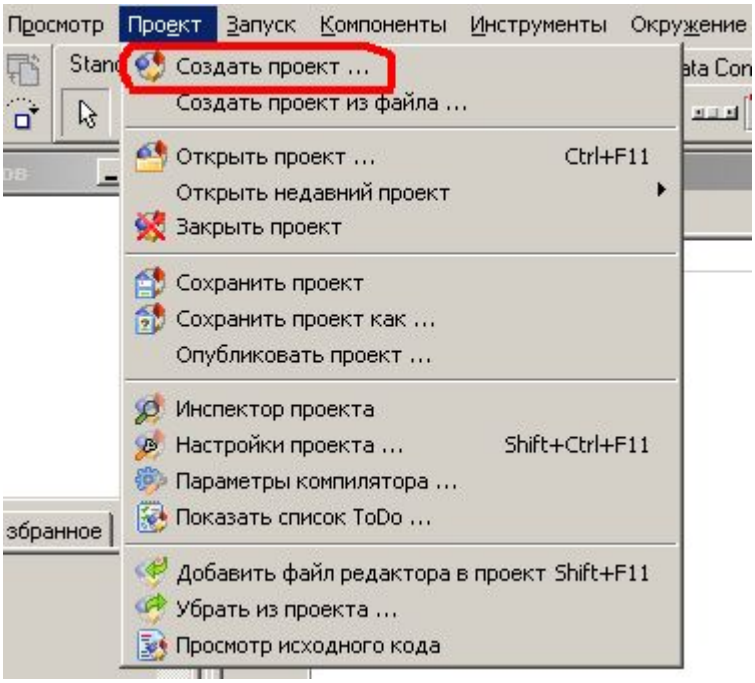


Рис. 5.15 Меню "Проект"

Создайте консольное приложение (рис. 5.16). Для этого выберите **Консольное приложение** и нажмите **Создать**.

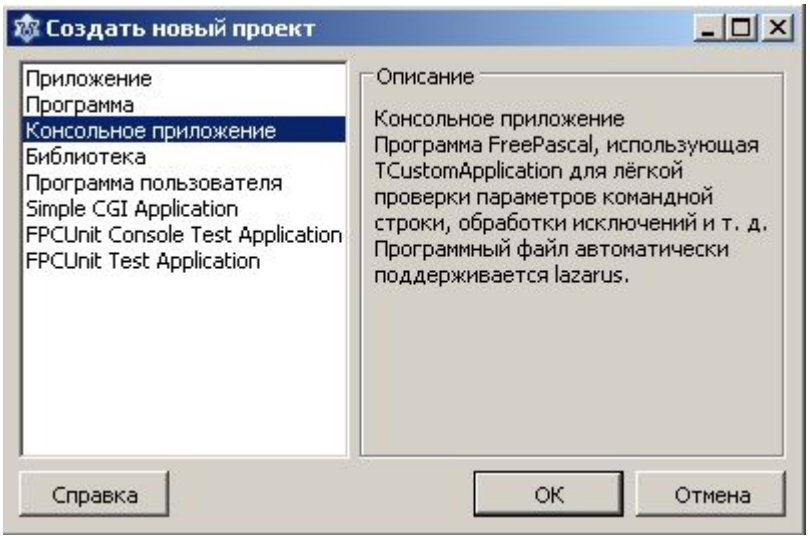


Рис. 5.16 Окно создания нового проекта

Возьмите себе за правило сразу же сохранять только что созданный проект, даже если он пока пустой. Это должно стать вашей хорошей привычкой, такой же, как чистить зубы утром и вечером! Дело в том, что во время сохране-

ния, вы можете создать папку для своего проекта, и все ваши файлы текущего проекта будут сохранены в отдельной папке. Это поможет вам структурировать ваши проекты и не запутаться в них, если их будет много.

Для сохранения проекта проще всего воспользоваться кнопками сохранения на панели инструментов, рис. 5.17.

Рис. 5.17 Кнопки сохранения проекта

В открывшемся диалоговом окне сохранения проекта создайте новую папку в нужном месте, укажите имя проекта и нажмите **Сохранить**, рис. 5.18, 5.19.

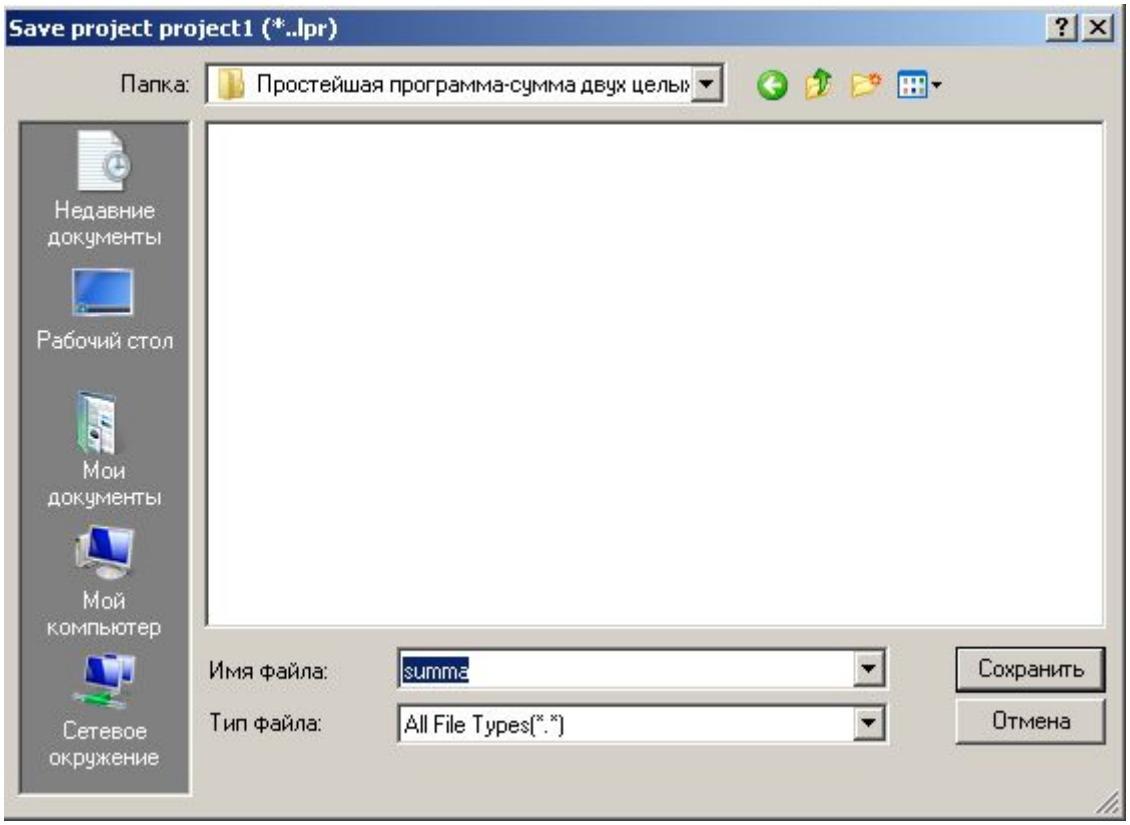


Рис. 5.18 Стандартное диалоговое окно сохранения в Windows

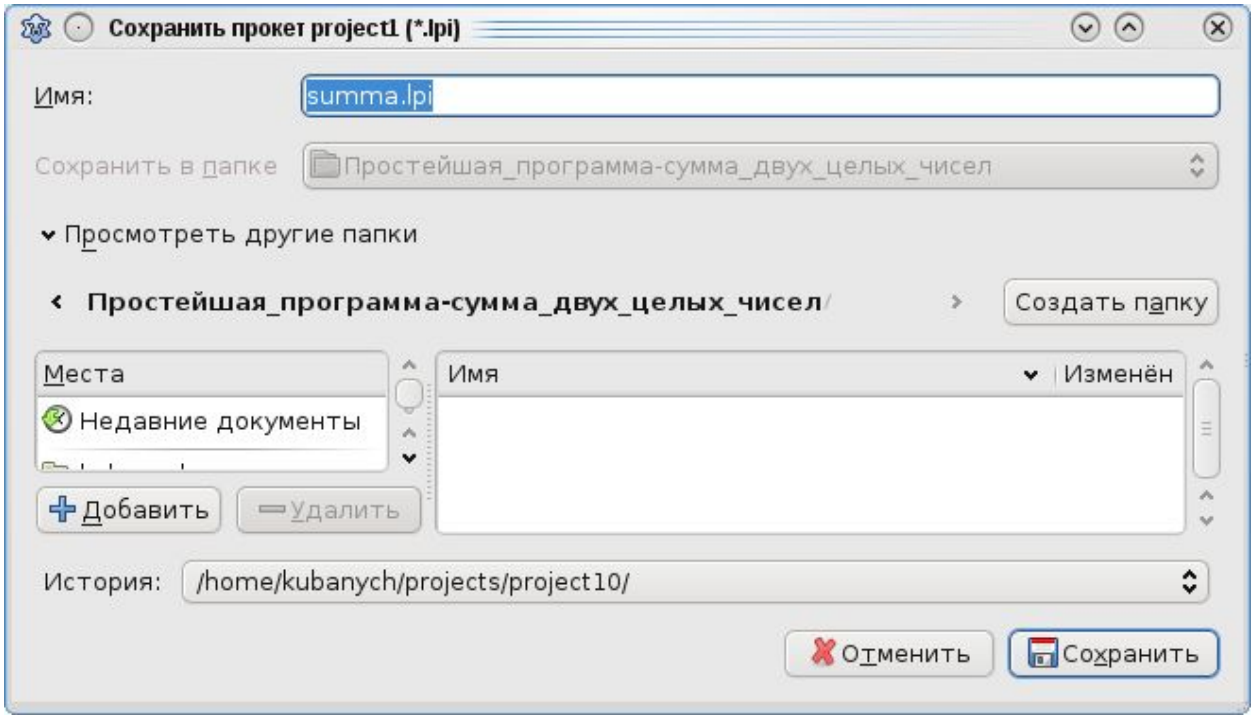


Рис. 5.19 Стандартное диалоговое окно сохранения в Linux

При задании имени папки и имени проекта старайтесь, чтобы имена отражали суть проекта. Это поможет вам легко ориентироваться в своих проектах, особенно когда их накопится достаточно много.

Также помните, что если вы даете имя, состоящее из нескольких слов, то в Linux нельзя ставить пробелы между словами. В этом случае Lazarus не сможет открыть ваш проект, рис. 5.20.

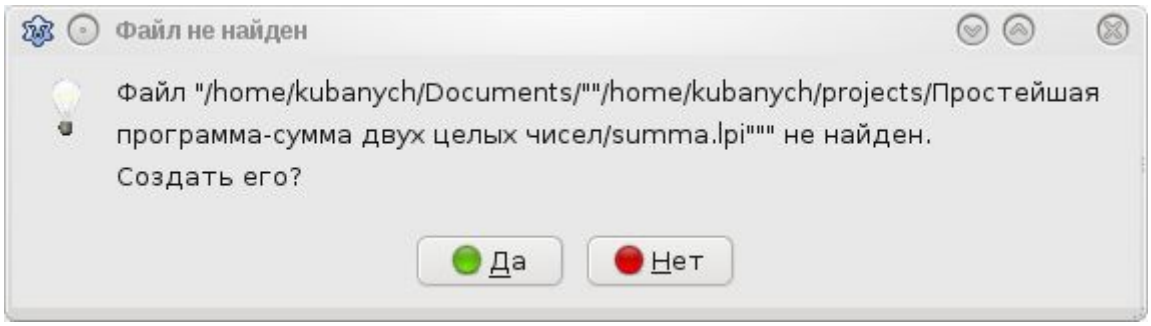


Рис. 5.20 Окно сообщения "Файл не найден"

Имя проекта всегда задавайте в нижнем регистре.

После сохранения в папке с проектом появятся несколько файлов, которые

мы рассмотрим позже.

В окне редактора исходного кода вы увидите текст. Это заготовка кода для консольного приложения, автоматически вставляемого Lazarus (рис. 5.21).

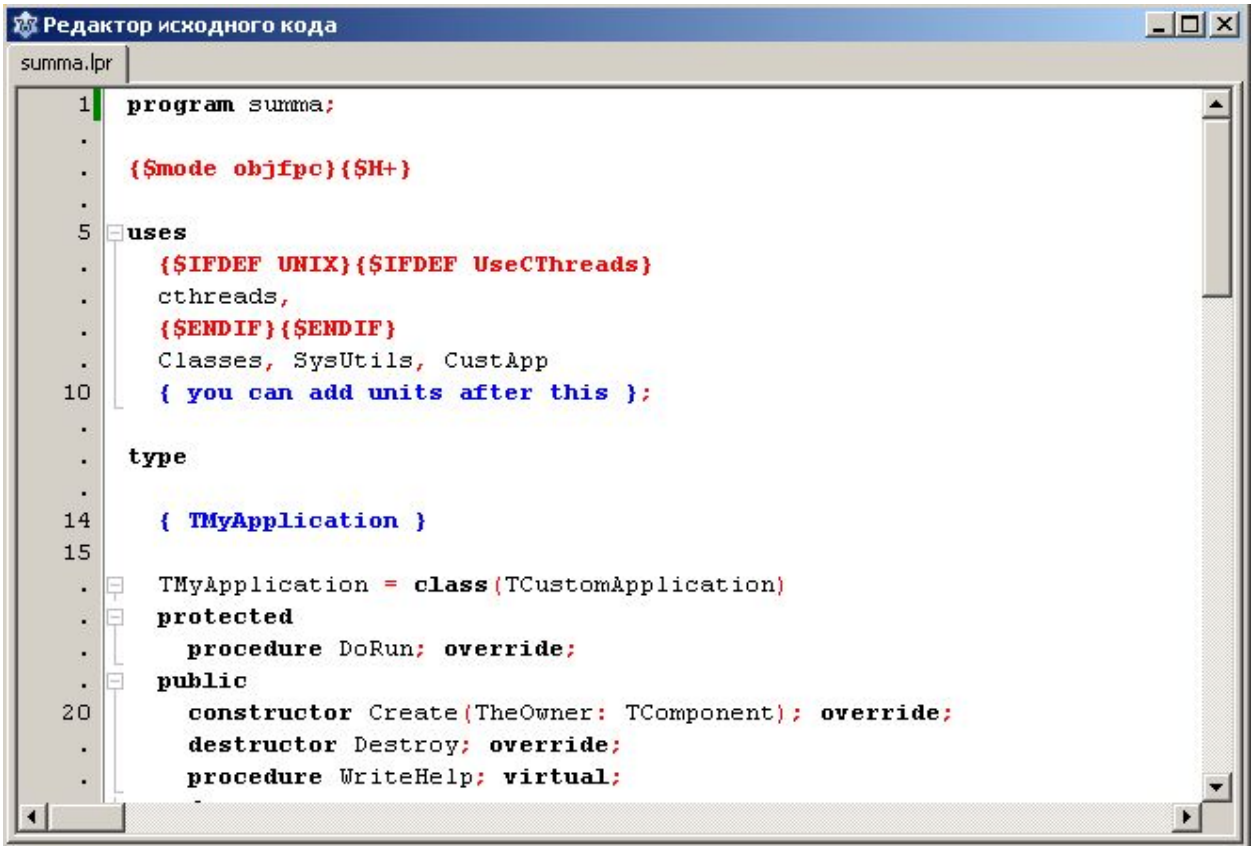


Рис. 5.21 Заготовка кода для консольного приложения, автоматически вставляемого Lazarus

Мы не будем сейчас обращать внимание на этот код и разбирать его, поскольку у нас для этого пока недостаточно знаний. Просто удалите этот код. Для этого установите курсор в любое место окна редактора исходного текста и нажмите **Ctrl+A**. Весь текст в окне выделится. Нажмите клавишу **Delete.**

Введите следующий код программы:

```
program summa;
{$mode objfpc}{$H+}
uses
  FileUtil; {Подключение модуля FileUtil для корректного
```

```
отображения русских букв с помощью функции UTF8ToConsole  }
var
  result, A, B: integer;
begin
  readln(A, B);
  result:=A + B;
  writeln(UTF8ToConsole('1-е введенное число= '), A);
  writeln(UTF8ToConsole('2-е введенное число= '), B);
  writeln(UTF8ToConsole('Сумма двух чисел = '), result);
end.
```

Окно редактора исходного кода в Windows будет иметь вид, рис. 5.22:

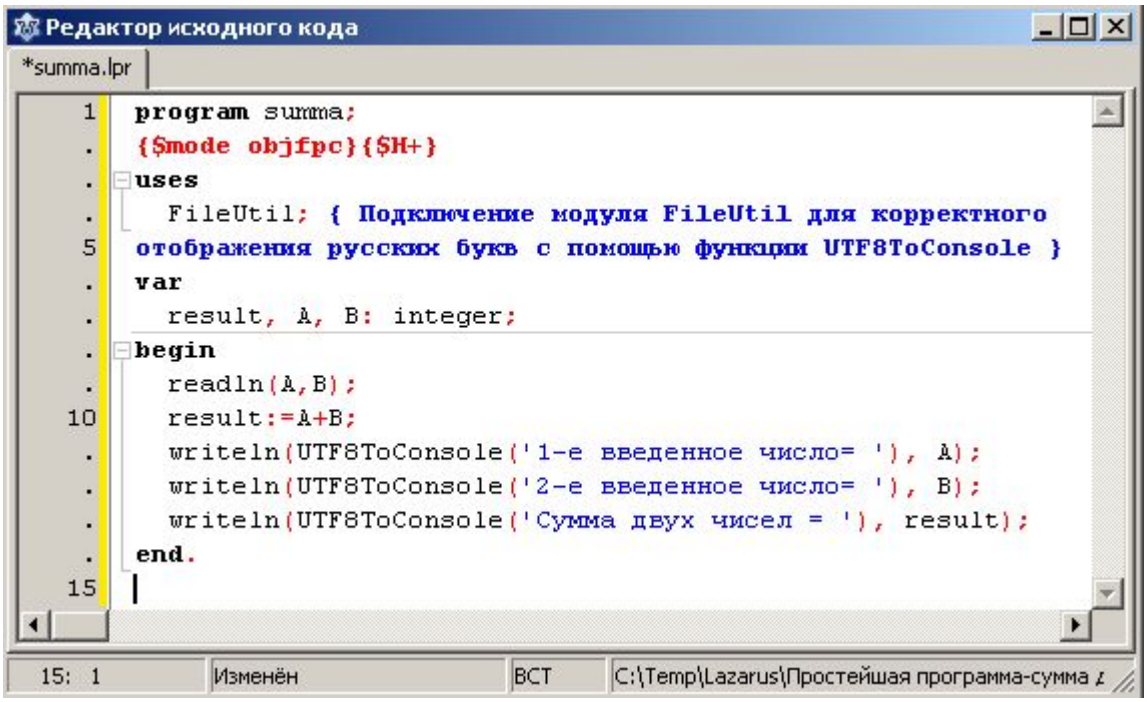


Рис. 5.22 Окно редактора исходного кода в Windows

В Linux это же окно будет иметь вид, рис. 5.23:

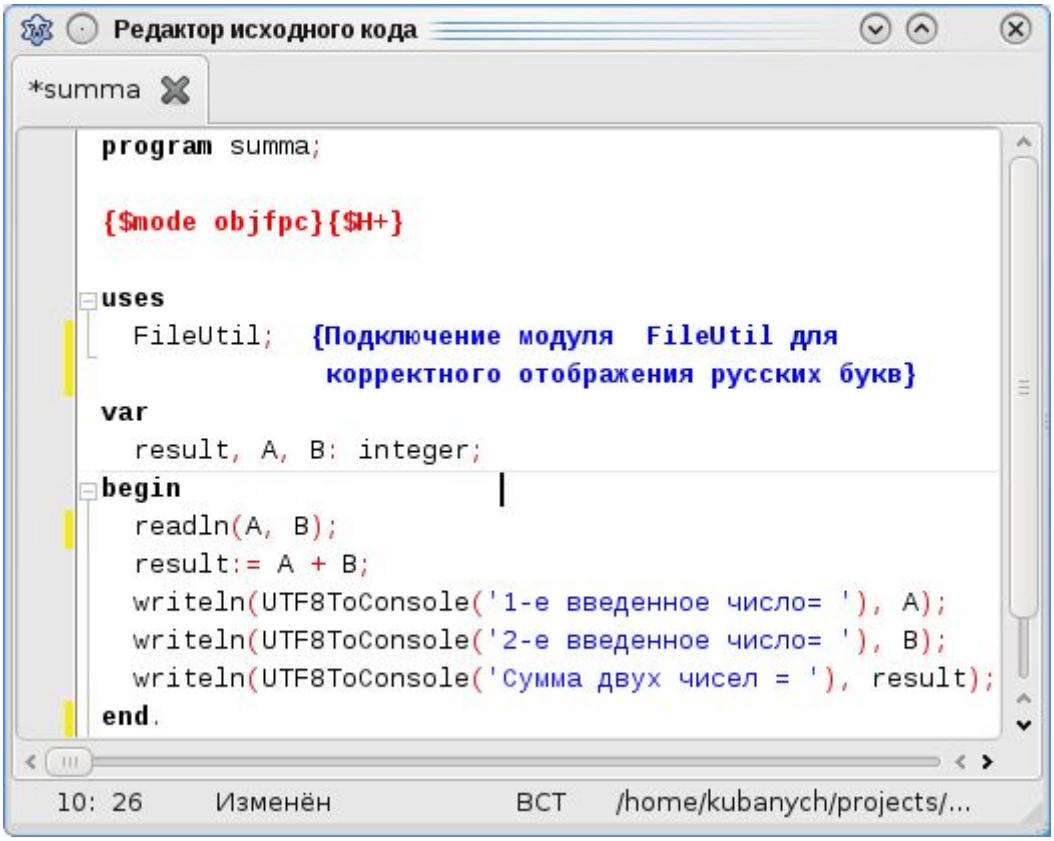


Рис. 5.23 Окно редактора исходного кода в Linux

Обратите внимание на объявление

```
uses
    FileUtil;
```

Этим объявлением мы подключаем модуль `FileUtil` в котором определена функция `UTF8ToConsole()`.

Если вас смущает что значит модуль и функция в Паскале, то немного потерпите. В главе 6 мы подробно рассмотрим все эти вопросы. Напоминаю, что мы вынуждены это делать, чтобы в Windows в окне DOS при работе вашей программы корректно отображался русский шрифт. Также пока примите на веру и проделайте следующее.

Откройте меню `Проект->Инспектор проекта` и нажмите на кнопку

со значком "+", рис. 5.24.

Рис. 5.24 Окно инспектора проекта

В появившемся окне "Добавить к проекту" нажмите на кнопку "Новое требование", рис. 5.25.

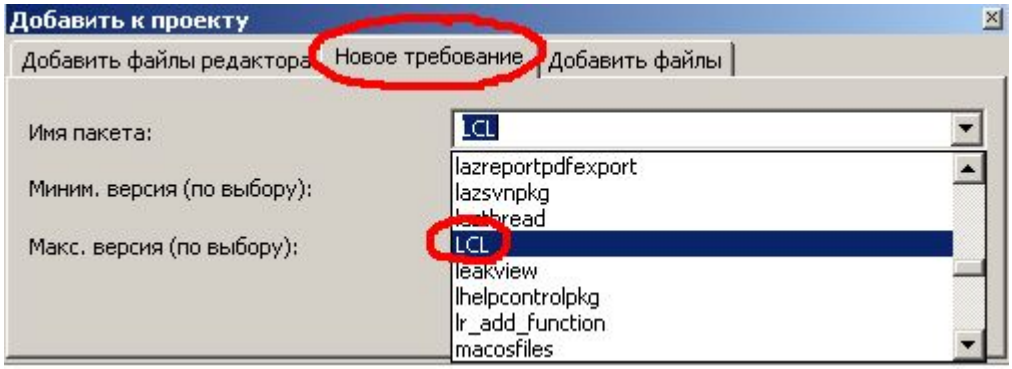


Рис. 5.25 Добавление нового требования

В раскрывающемся списке "Имя пакета" найдите и выберите пакет `LCL`.

Нажмите клавиши **Ctrl+F9**. Начнется компиляция и сборка программы. Если вы ввели текст программы без ошибок в точности как приведено выше, то компиляция завершится успешно. В окне **Сообщения** вы увидите сообщение *Проект "summa" успешно собран*.

В папке проекта появятся, в дополнение к уже существующим, еще несколько файлов. В частности, готовый к исполнению файл. В Windows это будет файл с расширением `exe`, в Linux файл без расширения.

Чтобы запустить программу на выполнение прямо из среды Lazarus нажмите клавишу F9 или кнопку "Запуск" (зеленый треугольник) на панели инструментов или меню `Запуск->Запуск`, рис. 5.26.

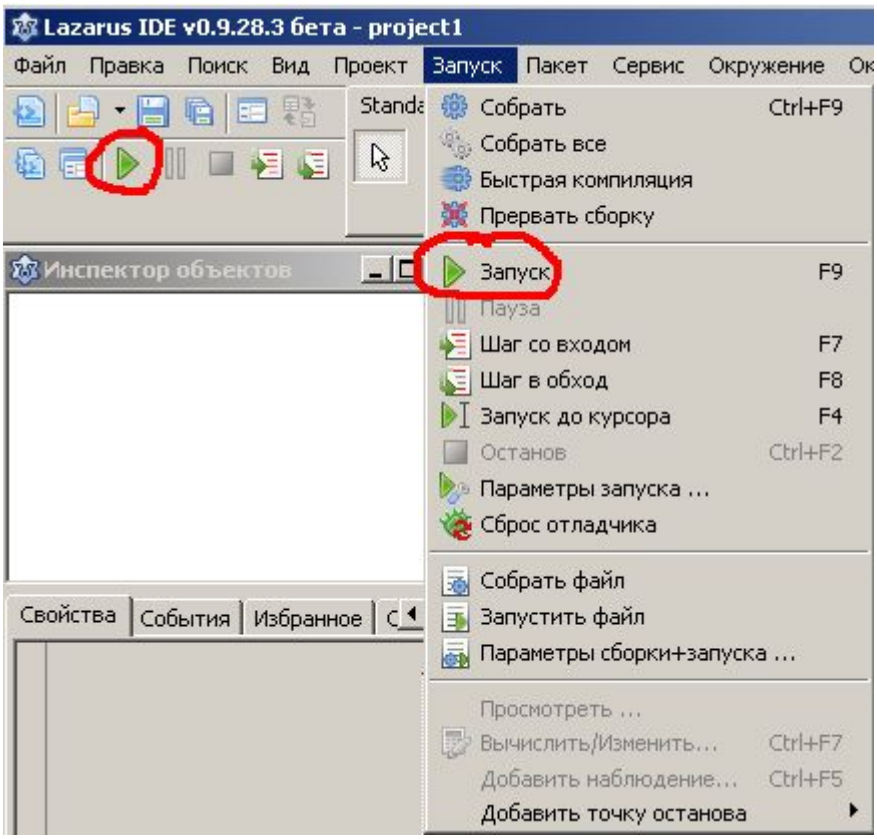


Рис. 5.26 Способы запуска программы

Пользователям Linux для того, чтобы запускать программы из среды Lazarus в терминале необходимо в меню `Запуск->Параметры запуска` установить флажок "Использовать приложение для запуска", рис. 5.27, 5.28.

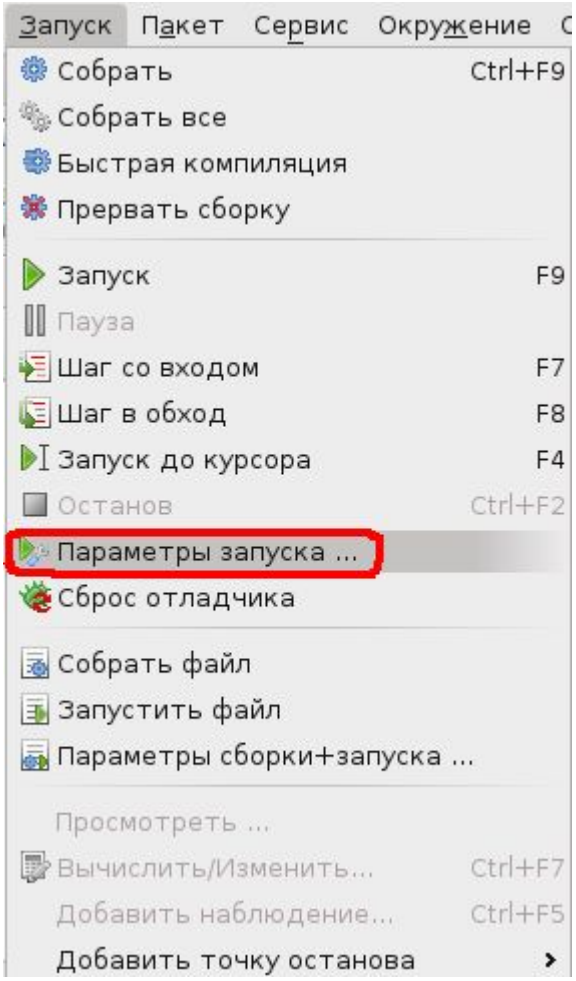


Рис. 5.27 Меню "Запуск"

При этом для некоторых дистрибутивов Linux надо заменить строку

/usr/X11R6/bin/xterm -T 'Lazarus Run Output' -e $(LazarusDir)/tools/runwait.sh $(TargetCmdLine)

на

/usr/bin/xterm -T 'Lazarus Run Output' -e $(LazarusDir)/tools/runwait.sh $(TargetCmdLine)

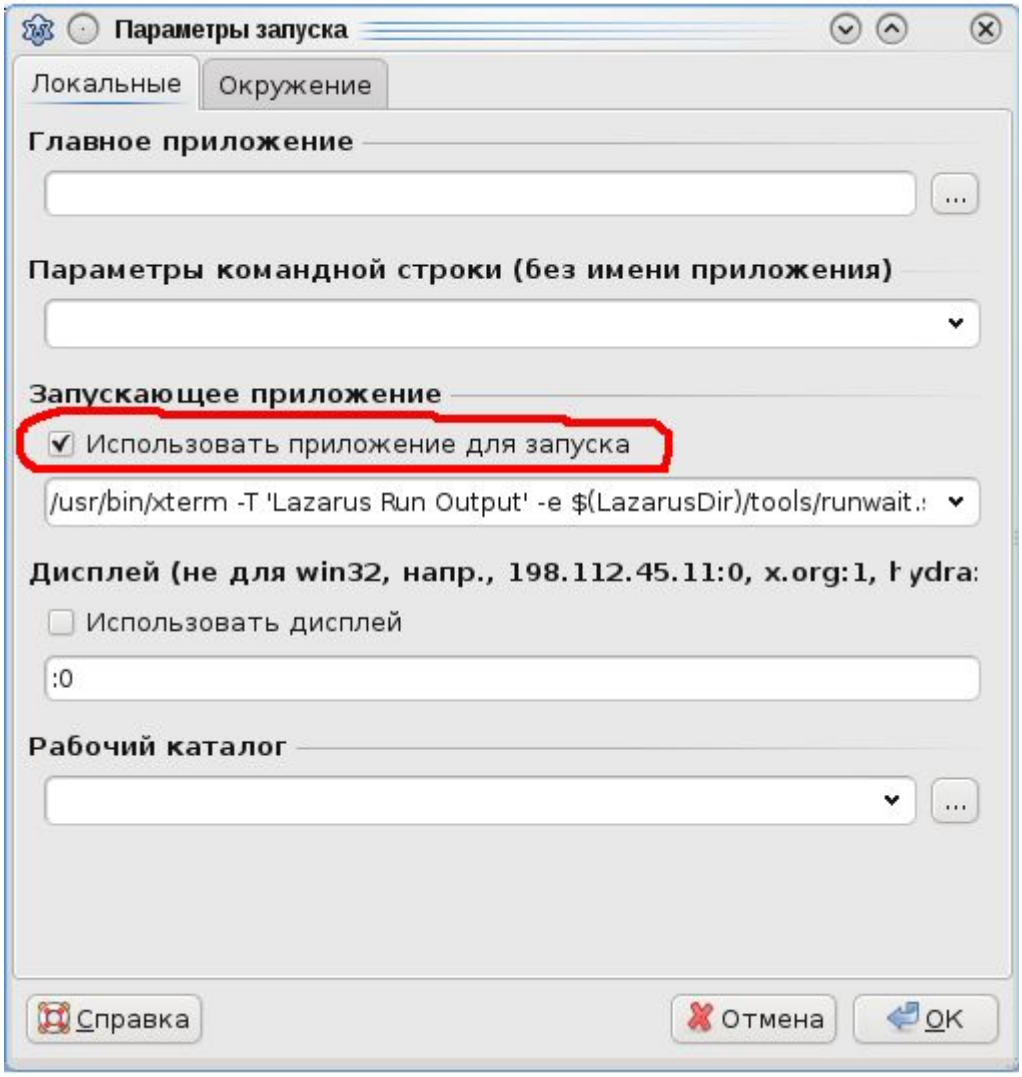


Рис. 5.28 Настройка проекта для запуска в терминале

Для запуска программы вне среды Lazarus в Windows достаточно дважды щелкнуть по имени исполняемого exe-файла.

В Linux выдать команду <путь к файлу> ./<имя исполняемого файла>

В дальнейшем, для единообразия в изложении, будем предполагать, что все примеры в книге запускаются из среды Lazarus.

После запуска программы у вас появится окно вида, рис. 5.29 (Windows) и рис. 5.30 (Linux).

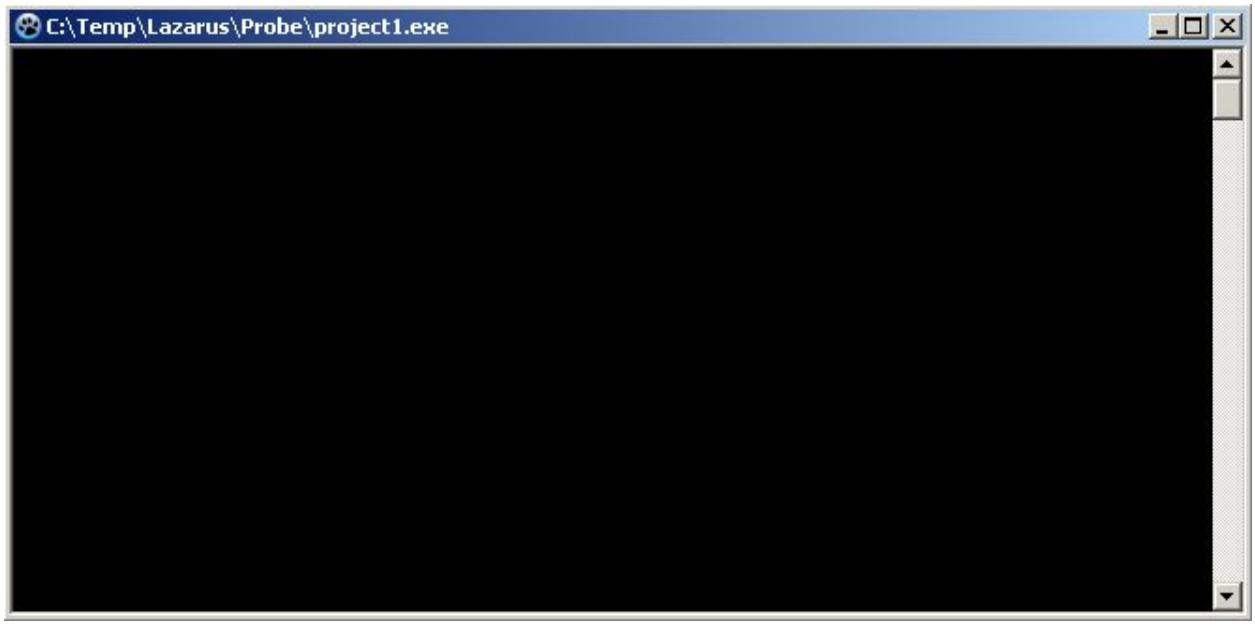


Рис. 5.29 Окно исполняемой программы в Windows

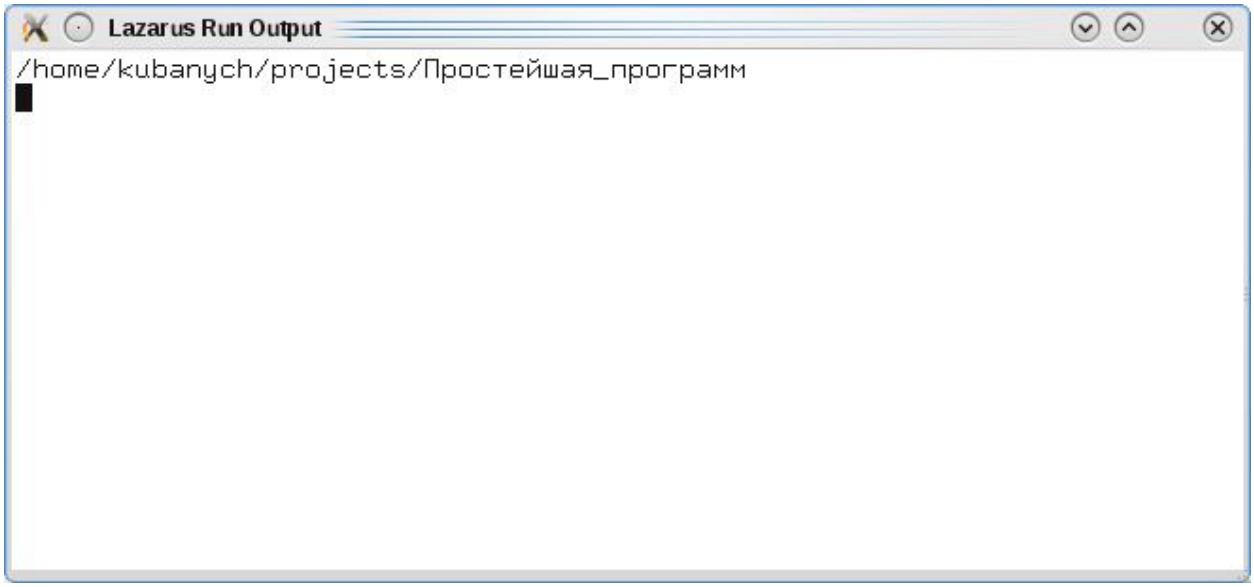


Рис. 5.30 Окно исполняемой программы в Linux

Введите значения переменных A и B. Это должны быть целые числа. Для завершения ввода числа нажмите **Enter.**

Окно вывода программы в Linux будет иметь вид, рис. 5.31:

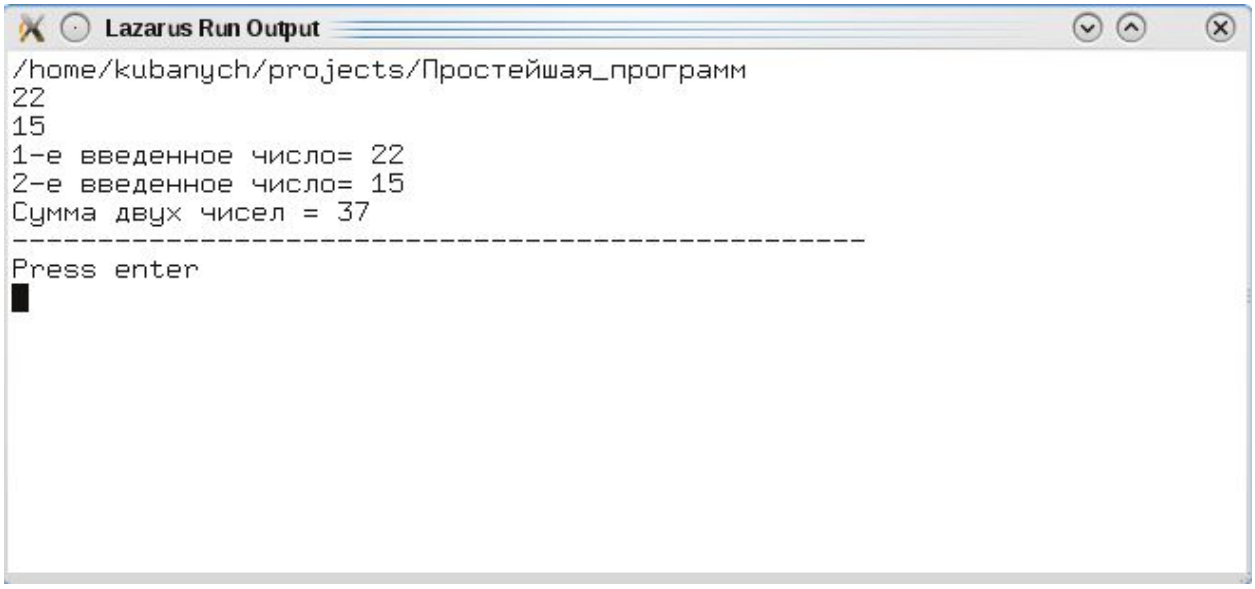


Рис. 5.31 Окно вывода программы в Linux

В Windows после ввода второго числа в окне что-то промелькнет и окно закроется. Как сделать так, чтобы окно не закрывалось, и мы могли увидеть результаты выполнения программы?

Для этого можно вставить в код программы оператор `readln` без параметров перед завершающим оператором `end` с точкой. Снова нажмите клавишу F9. На этот раз после ввода чисел окно не закроется и вы сможете увидеть результаты работы программы, рис. 5.32.

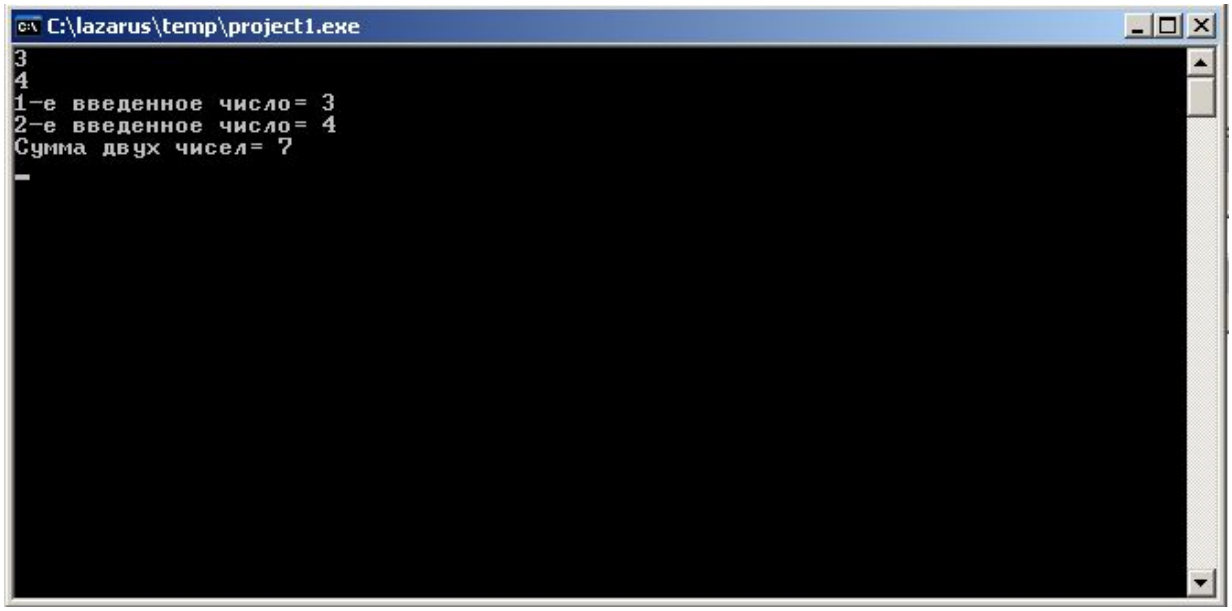


Рис. 5.32 Окно вывода программы в Windows

Чтобы закрыть окно программы нажмите Enter.

Проанализируем нашу программу. Вроде программа работает, сумма вычисляется правильно. И все же даже в этой простейшей программе видны существенные недостатки. Недостатки с точки зрения организации взаимосвязи с пользователем или, как говорят, `интерфейса` связи с пользователем.

Во-первых, сразу после запуска нашей программы появляется пустое окно. Это сразу же вызывает негативные ассоциации. Что-то случилось с компьютером? Он завис?

Ну, мы-то с вами знаем, что нужно ввести два числа, но пользователь может и не знать об этом! Как исправить это? А очень просто! Вставьте оператор

`writeln(UTF8ToConsole('`Введите два целых числа`'));`

перед оператором ввода `readln(A, B);`

Окно программы будет выглядеть так, рис. 5.33, 5.34:

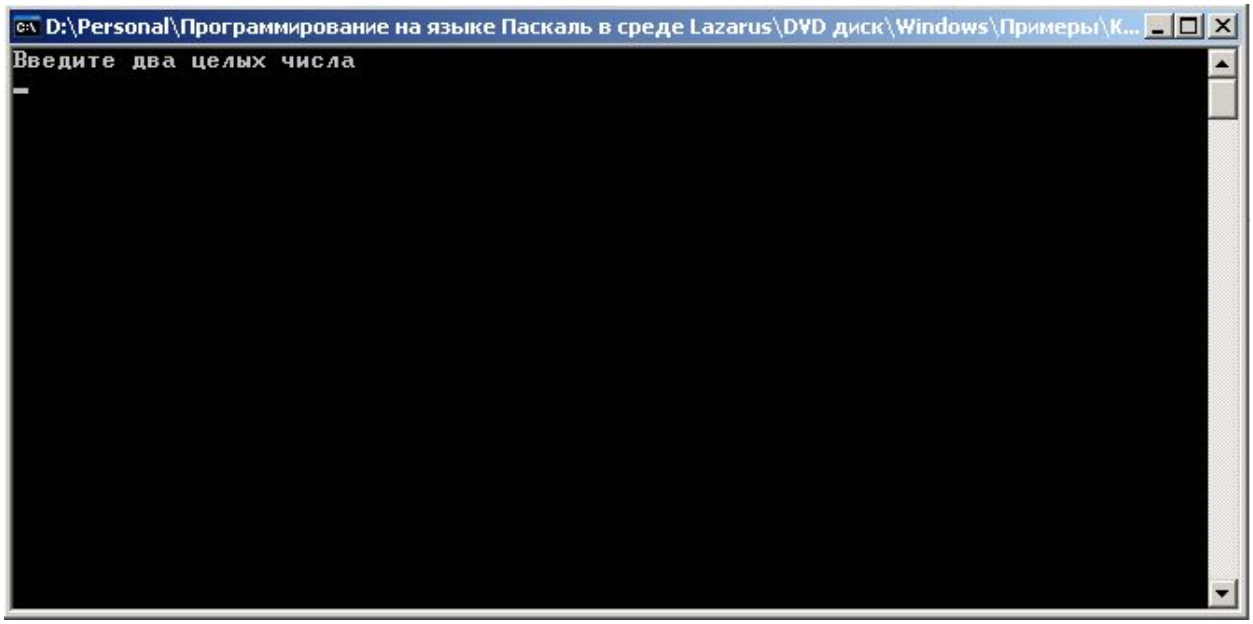


Рис. 5.33 Окно программы в Windows

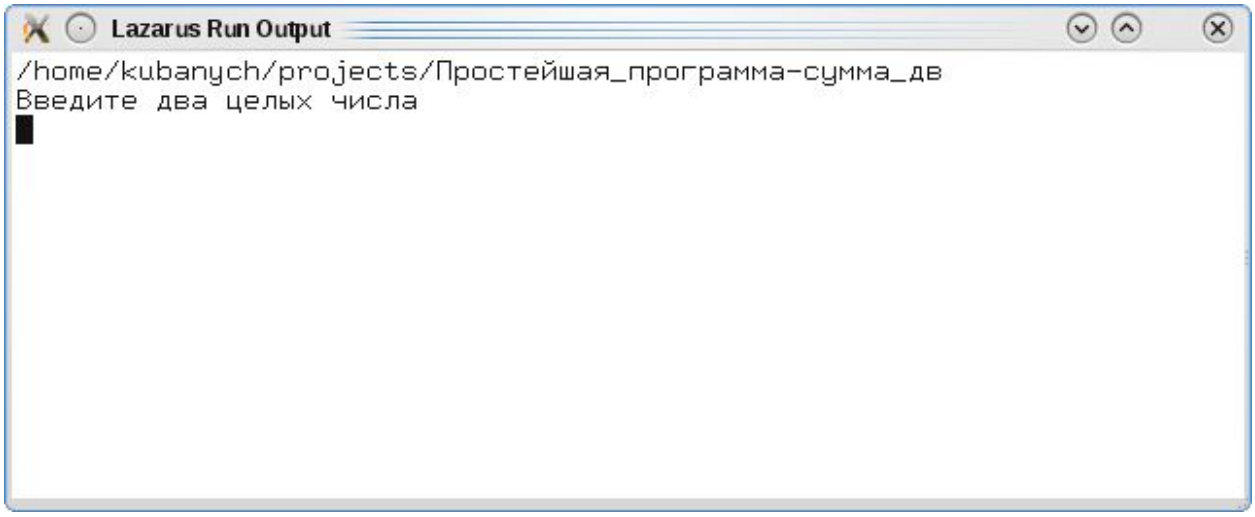


Рис. 5.34 Окно программы в Linux

Теперь даже ежу понятно, что нужно ввести два числа!

Во-вторых, (это касается опять же Windows) неплохо было бы дать пользователю инструкцию как выйти их программы. Окно программы будет закрыто только после нажатия клавиши **Enter**. Нажатие любых других клавиш к закрытию окна не приведет! Можно, конечно, воспользоваться кнопкой закрытия окна. Но желательно, чтобы программа сама закрывала свое собственное окно. И опять, пользователь ведь не знает, что для выхода из программы нужно нажать клавишу **Enter**. Есть два варианта выхода из этой ситуации:

1. В конце программы, но перед `readln` вставить оператор

```
writeln(UTF8ToConsole ('Для выхода нажмите Enter'));
```

2. Воспользоваться функцией `readkey` без параметров. Окно программы будет закрываться при нажатии любой клавиши. Этот вариант предпочтительней, только для этого необходимо использовать объявление `uses Crt`, чтобы включить модуль **CRT** о котором речь пойдет позже и в котором и есть эта функция. Но, даже в этом случае, будет правильнее, если вставить оператор (разумеется, перед `readkey`)

```
writeln(UTF8ToConsole('Для выхода нажмите любую клавишу'));
```

Окончательный текст программы будет выглядеть следующим образом:

```
program summa;
{$mode objfpc}{$H+}
uses
  Crt, FileUtil; {Подключение модуля CRT для использования функ-
ции readkey и модуля FileUtil для корректного отображения русских букв
с помощью функции UTF8ToConsole}
var
  result, A, B: integer;
begin
  writeln(UTF8ToConsole('Введите два числа'));
  readln(A, B);
  result:=A + B;
  writeln(UTF8ToConsole('1-е введенное число= '), A);
  writeln(UTF8ToConsole('2-е введенное число= '), B);
  writeln(UTF8ToConsole('Сумма двух чисел = '), result);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Результаты работы программы приведены на рис. 5.35, 5.36.

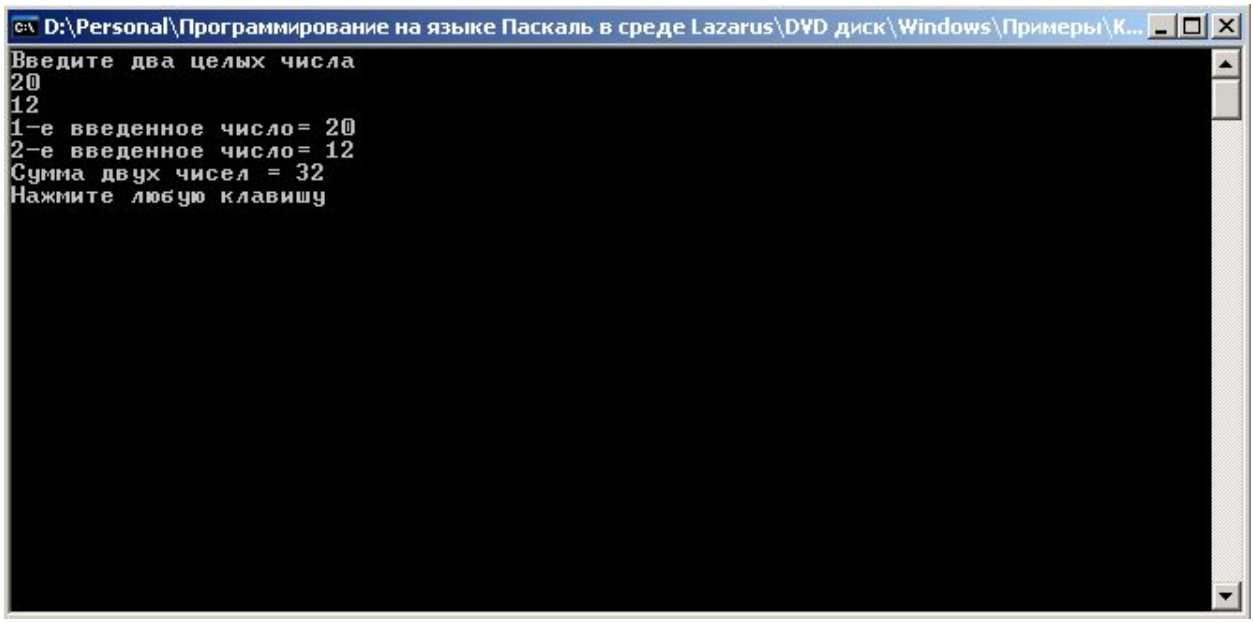


Рис. 5.35 Результаты работы программы в Windows

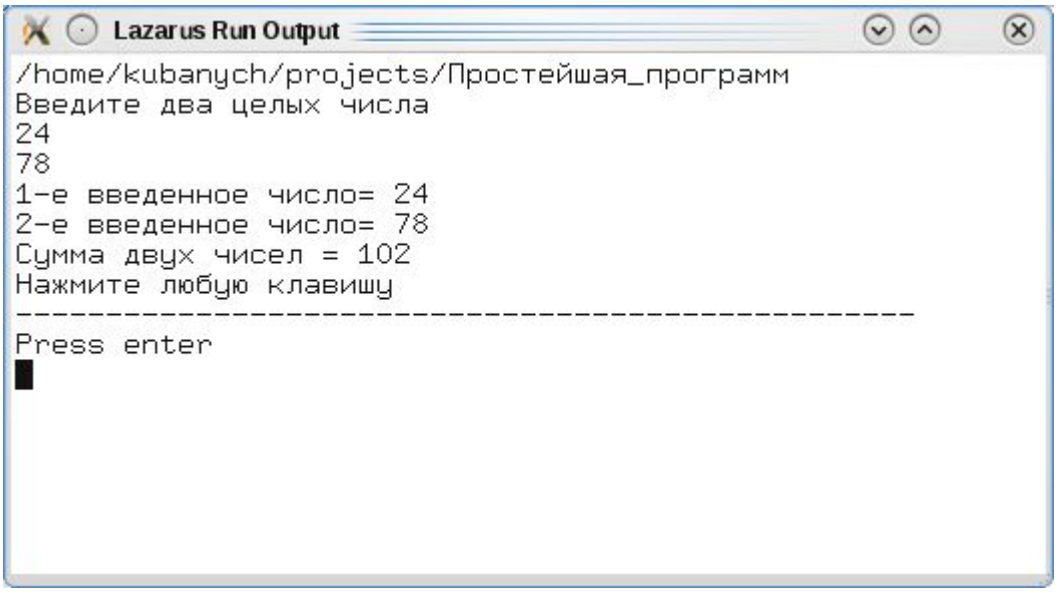


Рис. 5.36 Результаты работы программы в Linux

Пользователи Linux обратите внимание, что если вы выполняете программу в терминале, то выдается еще дополнительно сообщение "Press enter". Все же операторы

```
writeln(UTF8ToConsole('Нажмите любую клавишу'));
readkey;
```

мы оставим, так как книга рассчитана не только для "линуксоидов", но и для "оконщиков"! А в Windows, как мы уже видели, эти операторы нужны. Ну, а пользователям Linux оставляем право выбирать, использовать эти операторы или нет, равно как и применять функцию `UTF8ToConsole()`.

5.1.11. Открытие существующего проекта

Открыть существующий проект из среды Lazarus можно, воспользовавшись кнопкой открытия в панели инструментов, рис. 5.37.

Рис. 5.37 Кнопка открытия существующего проекта

или в меню **Файл** выбрать пункт **Открыть**... или нажать клавиши **`Ctrl`**+`O`. Будет открыт стандартный диалог открытия файла, рис. 5.38, 5.39.

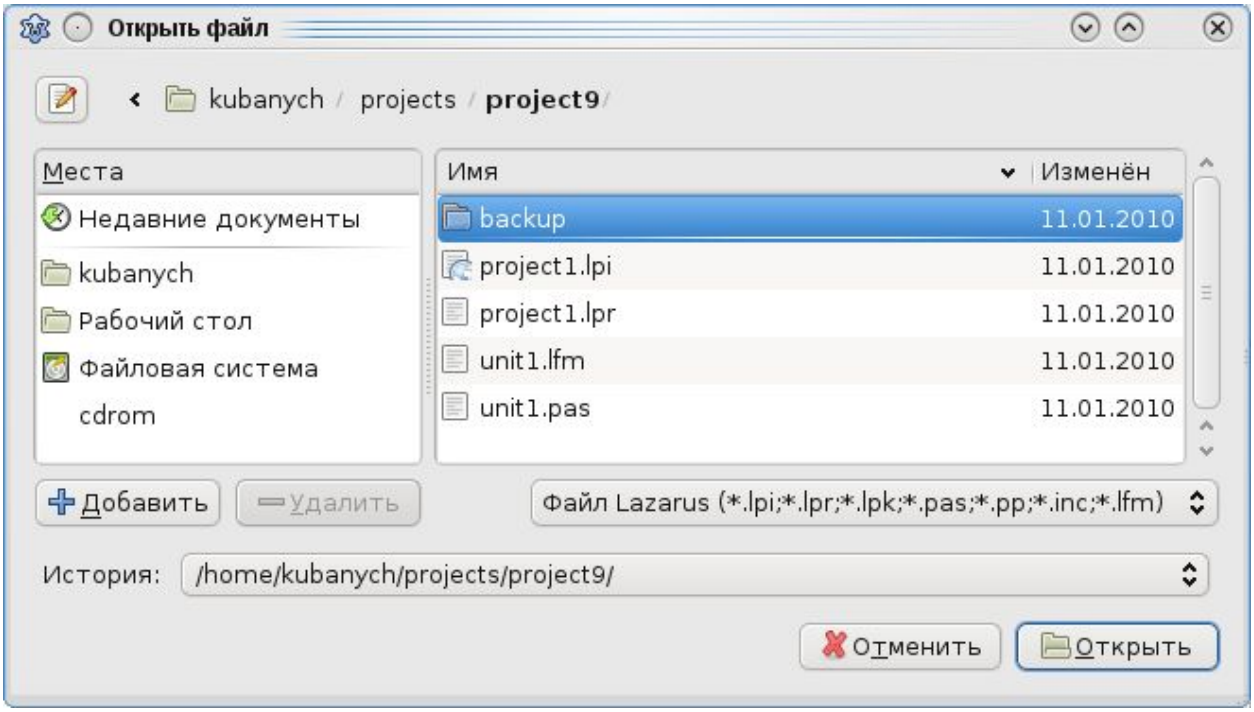


Рис. 5.38 Стандартный диалог открытия проекта в Linux

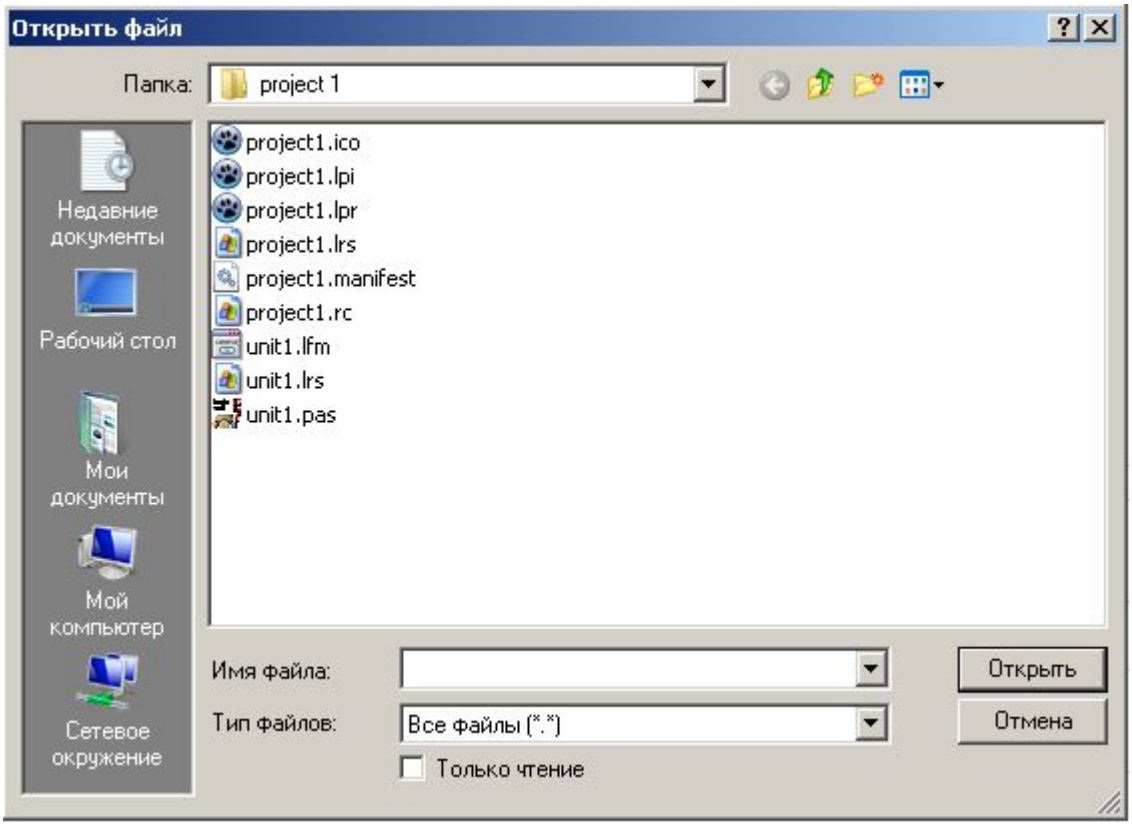


Рис. 5.39 Стандартный диалог открытия проекта в Windows

Если у вас Lazarus не запущен, то можно открыть существующий проект дважды щелкнув по имени файла с расширением *.lpi или *.lpr. Файл с расширением lpi (Lazarus Project Information) это основной файл проекта Lazarus, сохраняется в XML формате.

Файл с расширением lpr – исходный код основной программы. Не смотря на специфичное для Lazarus расширение на самом деле это обычный Pascal-код.

В Linux, если вместо Lazarus будет запускаться другая программа (чаще всего Konqueror или KWrite), то надо настроить файловый менеджер, чтобы он открывал Lazarus по двойному щелчку по имени файла.

Щелкните по имени файла проекта правой клавишей мыши. В открывшемся контекстном меню выберите пункт "Свойства…", рис. 5.40. Далее в открывшемся окне "Свойства" нажмите на кнопку с изображением гаечного ключа, рис. 5.41. В окне "Изменить тип файла" нажмите на кнопку "Добавить…", рис. 5.42. Откроется окно "Выбор приложения…", рис. 5.43. В этом окне введите

имя приложения с путем к нему, например

```
/usr/lib/lazarus/lazarus
```

или

```
/usr/lib/lazarus/startlazarus
```

или нажмите на кнопку выбора файла и в окне "Редактор типов файла" найдите папку, где установлен Lazarus, рис. 5.44.

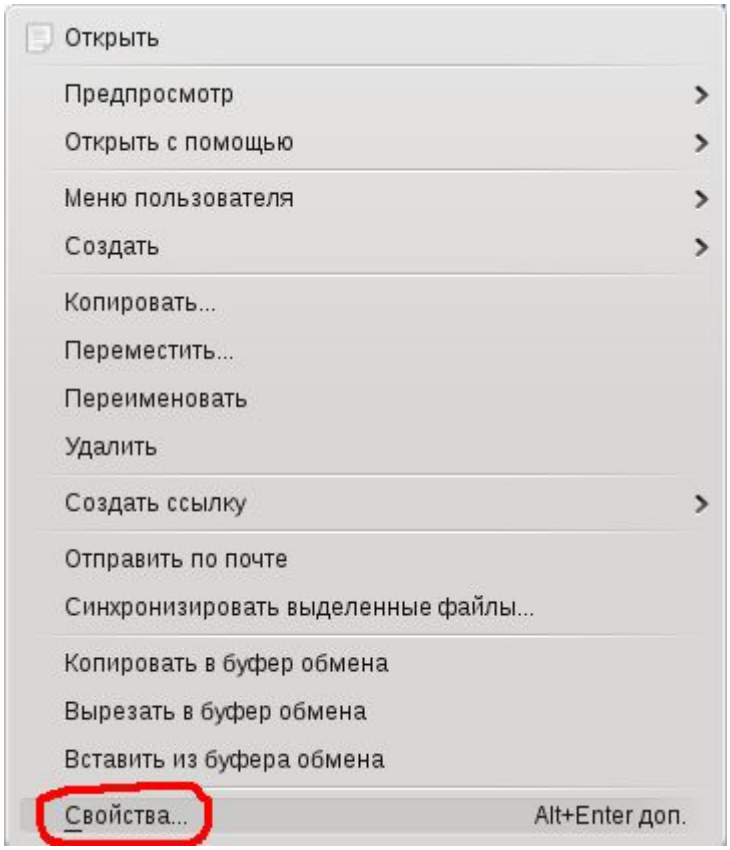


Рис. 5.40 Контекстное меню

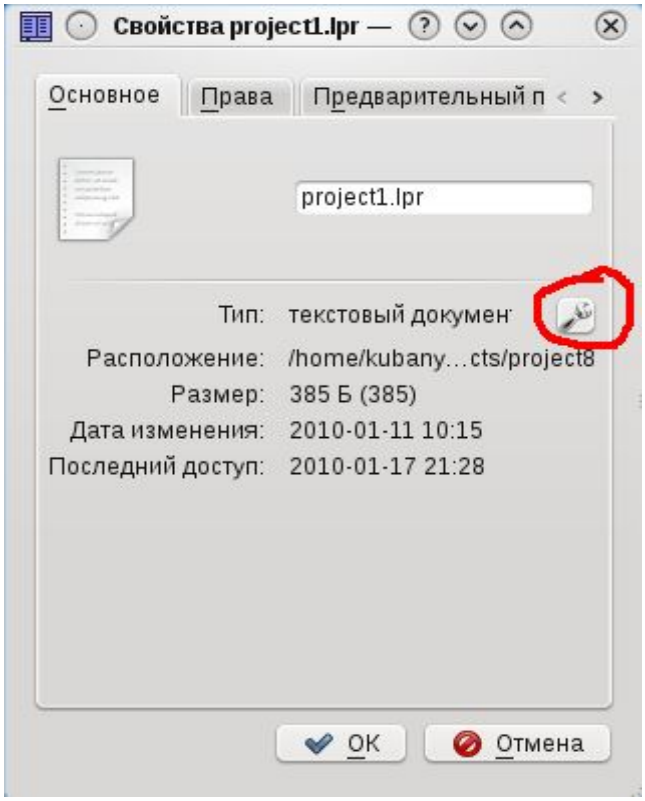


Рис. 5.41 Окно свойств файла

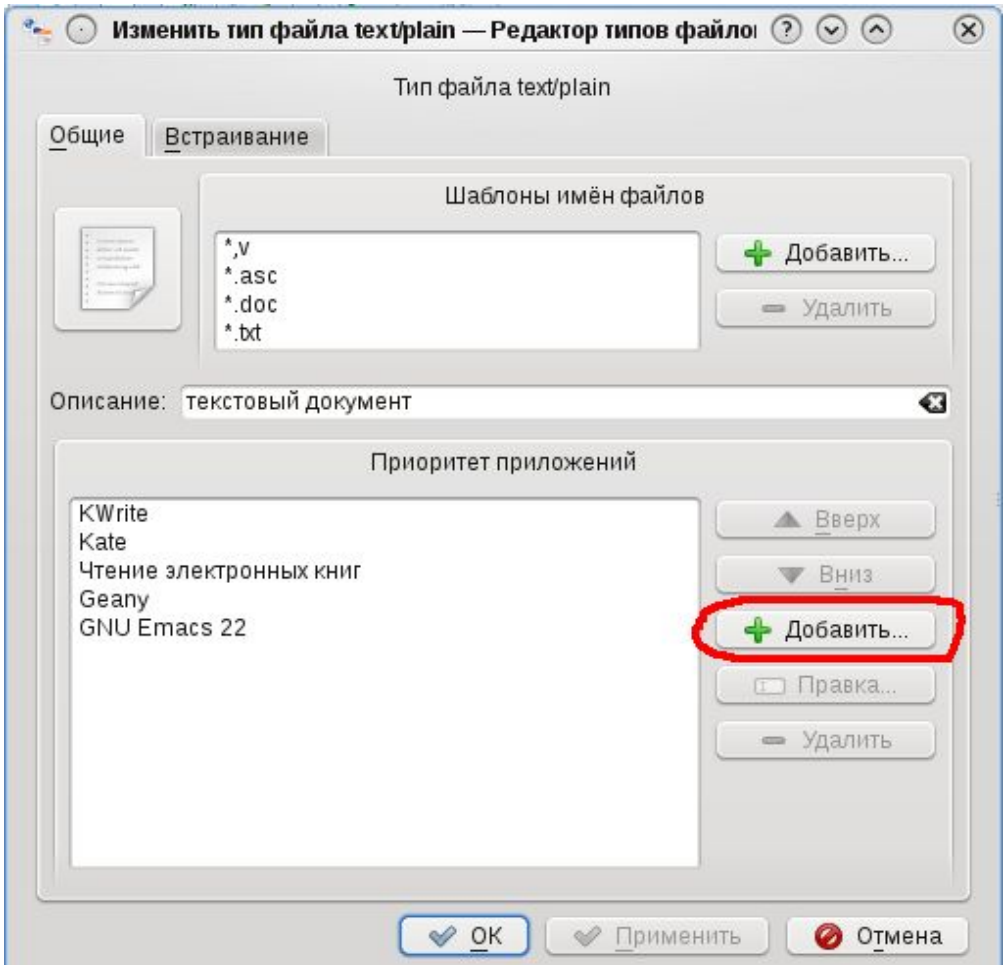


Рис. 5.42 Изменение типа файла

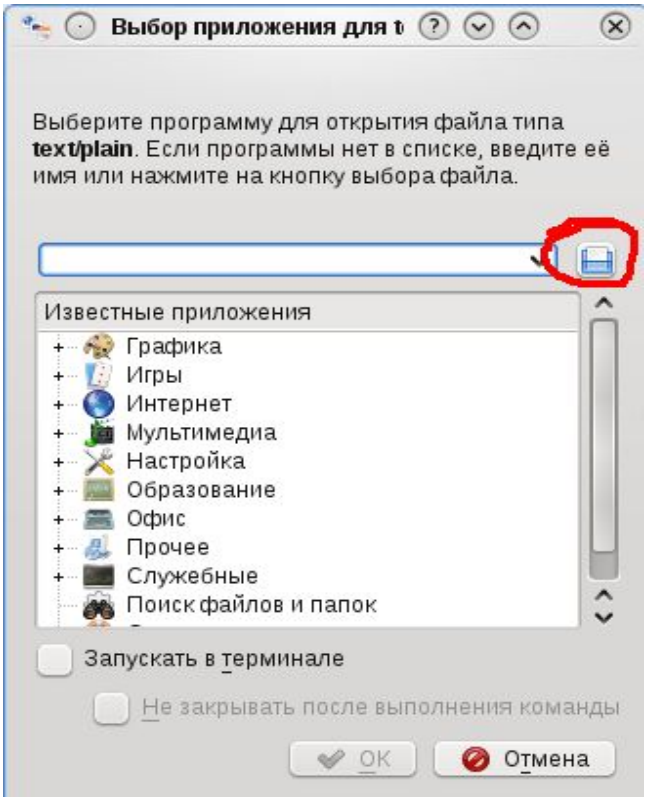


Рис. 5.43 Окно выбора приложения для открытия файла

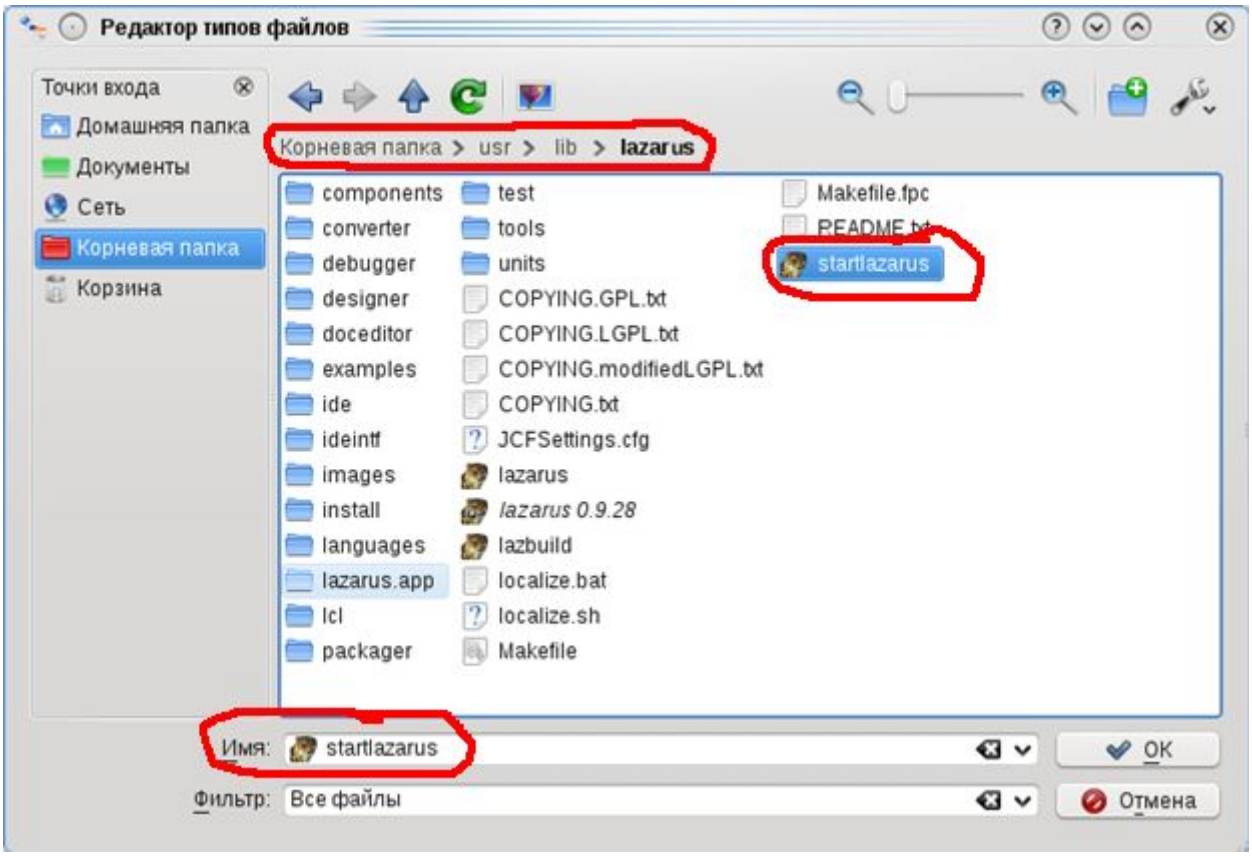


Рис. 5.44 Выбор Lazarus для открытия файла проекта

5.1.12. Другие способы создания консольных приложений

Lazarus предоставляет и другие способы создания консольных приложений. Рассмотрим снова меню **Проект**-> **Создать проект**…, рис. 5.45.

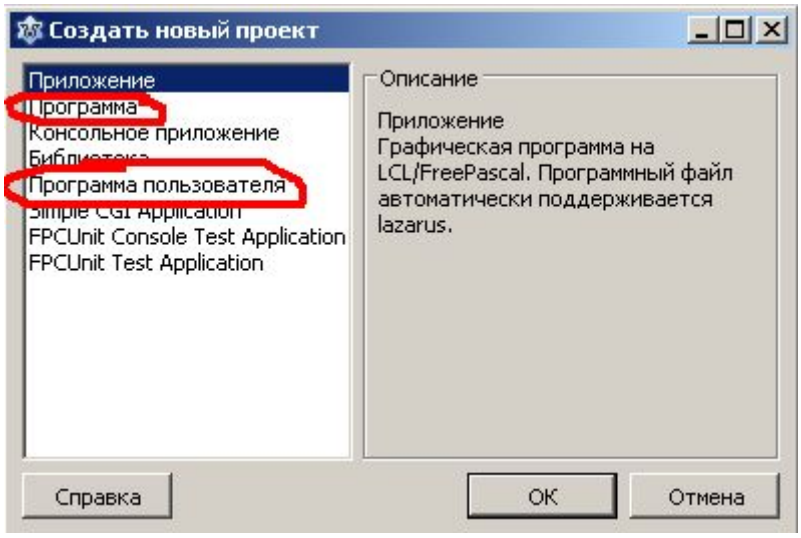


Рис. 5.45 Другие способы создания консольных приложений

Консольное приложение можно создать, выбрав пункты "**Программа**" и "**Программа пользователя**".

При выборе "**Программа**" Lazarus сделает следующую заготовку:

```
program Project1;
{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };
{$IFDEF WINDOWS}{$R project1.rc}{$ENDIF}
begin
end.
```

Исходный код программы будет автоматически поддерживаться Lazarus и будет сохранен в файле с расширением .lpr.

При выборе "**Программа пользователя**" Lazarus сделает следующую заготовку:

```
program Project1;
{$mode objfpc}{$H+}
uses
  Classes, SysUtils
  { you can add units after this };
{$IFDEF WINDOWS}{$R project1.rc}{$ENDIF}
begin
end.
```

При этом исходный код программы будет сохранен в файле с расширением .pas.

Сравнивая эти три способа создания консольных приложений, можно сделать вывод, что самым "продвинутым" способом будет первый способ (каким мы и воспользовались). Правда, пока мы только "продемонстрировали" свои намерения. Ведь мы полностью заменили код, который нам предлагал Lazarus. Дело в том, что заготовка программы, предложенная Lazarus, предполагает, что мы знаем объектно-ориентированное программирование (ООП) и еще много чего, например, обработку исключений. Нам до этого еще далеко!

Но мы, выбрав пункт "Консольное приложение" подтверждаем свою решимость "идти до конца" – изучить язык, изучить ООП и затем создавать консольные приложения "как положено"!

А пока, в нашей книге, мы не будем делать различий в способах создания консольных приложений. Вы можете выбирать любой способ, какой вам понравится.

В заключение отметим, что создавать новые консольные проекты, а также множество других видов проектов и программных объектов можно через меню **Файл** -> **Создать**…, рис. 5.46.

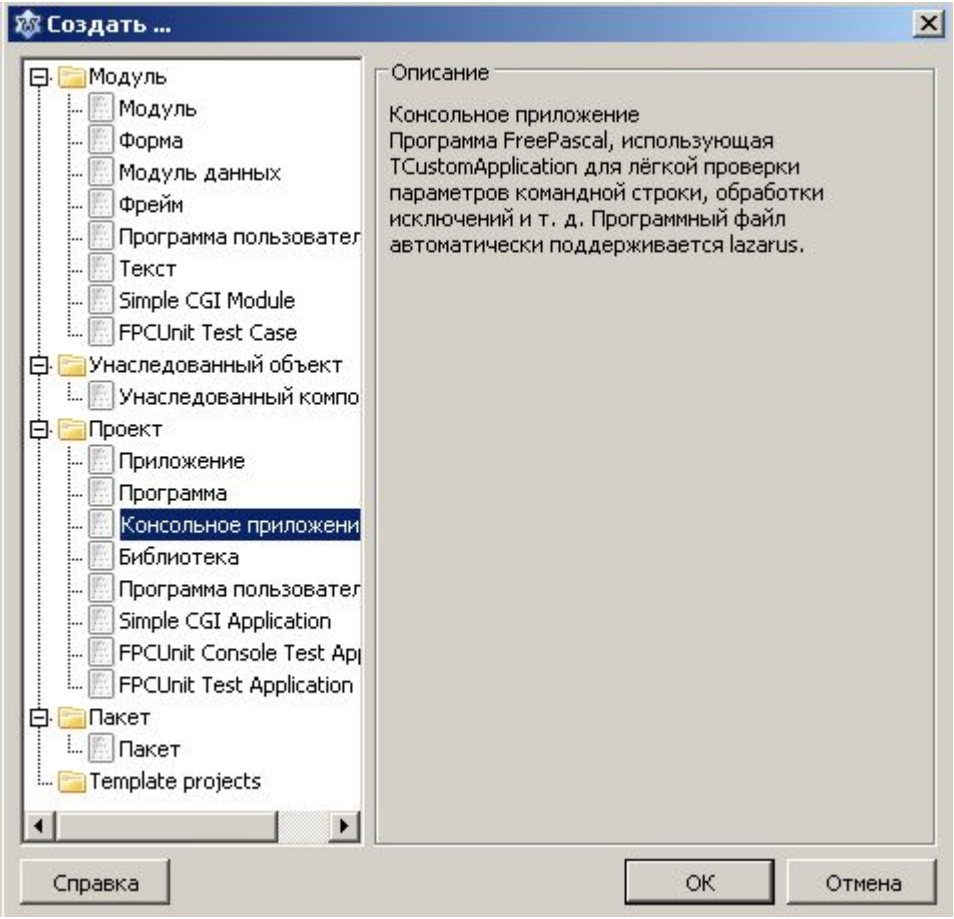


Рис. 5.46 Меню "Создать"

5.1.13. Типовой пустой проект

Как вы видели из 5.1.10 после создания нового проекта, его необходимо настраивать. В частности, включить в проект пакет LCL (см. рис. 5.24, 5.25), а в Linux настроить параметры запуска (см. рис. 5.27, 5.28). При большом числе создаваемых проектов это напрягает. В нашей книге мы будем создавать довольно много консольных приложений. Автор очень надеется, что вы, уважаемый читатель, все примеры, приведенные в книге, будете скрупулезно выполнять на компьютере. Ведь только так можно научиться программировать! Даже

простой набор текстов программ в редакторе исходного кода Lazarus позволит вам намного быстрее освоить и запомнить многие синтаксические конструкции языка Паскаль. Как было сказано в 4.1.3 одного чтения и "понимания" примеров книги будет совершенно недостаточно!

Поэтому удобнее всего для выполнения примеров создать пустой проект со всеми настройками, о которых говорилось выше и всегда, когда необходимо создавать новый проект, использовать уже готовый пустой проект. Давайте поступим следующим образом:

1. Создайте консольное приложение любым удобным для вас способом.

2. Вместо заготовки Lazarus введите в окне редактора исходного кода следующий текст:

```
program project1;

{$mode objfpc}{$H+}

uses
  CRT, FileUtil, SysUtils;
begin
  {Вставьте сюда исходный код вашей программы}

  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

3. Установите необходимые свойства проекта, т.е. включите в проект пакет LCL, в Linux настройте параметры запуска.

4. Сохраните проект в папке "Типовой пустой проект для консольных приложений". Не забывайте, что в Linux в названиях папок не допускаются пробе-

лы, поэтому присвойте этой папке имя без пробелов, например такое:

"Типовой_пустой_проект_для_консольных_приложений".

Теперь, если вам необходимо создать новый проект, откройте этот пустой проект и тут же сохраните его в другой папке с помощью меню **Файл**-> **Сохранить как**… При этом можно и нужно сохранить проект под другим именем, отражающим характер решаемой задачи. Ваш новый проект будет уже иметь необходимые нам настройки.

5.1.14. Операции с целыми числами

До сих пор мы рассматривали лишь одну операцию с целыми числами – сложение. Естественно в Паскале разрешены и другие операции. Рассмотрим программу, показывающую все возможные операции с целыми числами:

Пример.

```
program int_operations;
uses Crt, FileUtil;
var
  A, B, C: integer;
begin
  writeln(UTF8ToConsole('Введите два числа'));
  readln(A, B);
  writeln('A= ',A,' B= ',B);
  C:=A + B;
  writeln(UTF8ToConsole('Демонстрация сложения, C= '),C);
  C:=A * B;
  writeln(UTF8ToConsole('Демонстрация умножения, C= '),C);
  C:=A div B;
  writeln(UTF8ToConsole('Демонстрация деления нацело, C= '),C);
```

```
  C:=A mod B;
  writeln(UTF8ToConsole('Остаток от деления, C= '),C);
  C:=A - B;
  writeln(UTF8ToConsole('Демонстрация вычитания, C= '),C);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Из этой программы видно, что знак умножения * (а не буква x). Тем самым избегают возможности спутать его с буквой x.

Далее с целыми числами определены операции:

`div` (от английского `divide` – делить) – деление нацело.

Пусть `A = 17`, `B = 3`, отсюда `17:3 = 5*3+2` и

`A div B` дает результат 5

`mod` (от английского `modulo` – определить остаток) – определение остатка от деления нацело.

`A mod B` дает результат 2

Откомпилируйте и выполните свою программу (клавиша F9).

Все нормально? Программа работает? У вас еще не выходило такое? (рис. 5.47, 5.48)

Рис. 5.47. Сообщение об ошибке

Редактор исходного кода

int_operations.lpr

```
1   program int_operations;
    {$mode objfpc}{$H+}
3   uses
      CRT, FileUtil;
5   var
      A, B, C: integer;
    begin
      writeln(UTF8ToConsole('Введите два числа'));
      readln(A, B);
10    writeln('A= ', A, ' B= ', B);
      C:= A + B;
      writeln(UTF8ToConsole('Демонстрация сложения, C= '), C);
      C:= A * B;
      writeln(UTF8ToConsole('Демонстрация умножения, C= '), C);
15    C:= A div B;
      writeln(UTF8ToConsole('Демонстрация деления нацело, C= '), C);
      C:= A mod B;
      writeln(UTF8ToConsole('Остаток от деления, C= '), C);
      C:= A - B;
20    writeln(UTF8ToConsole('Демонстрация вычитания, C= '), C);
      writeln(UTF8ToConsole('Нажмите любую клавишу'));
      readkey;
    end.
```

Рис. 5.48. Оператор, при выполнении которого произошла ошибка

В Linux окно вывода будет таким:

Lazarus Run Output

```
/media/share/Примеры/К_главе_2/2.1.Основные_элементы_языка/Операции_с_целыми_чис
лами/int_operations
Введите два числа
5
0
A= 5 B= 0
Демонстрация сложения, C= 5
Демонстрация умножения, C= 0
An unhandled exception occurred at $08048768 :
                                                EDivByZero : Division by zero
                                                                                $08
048768  main,  line 14 of int_operations.lpr

                                              ------------------------------------
--------------
Press enter
```

Рис. 5.49 Сообщение об ошибке в терминале

Или, если вы запускаете программу в консоли, то окно будет иметь вид:

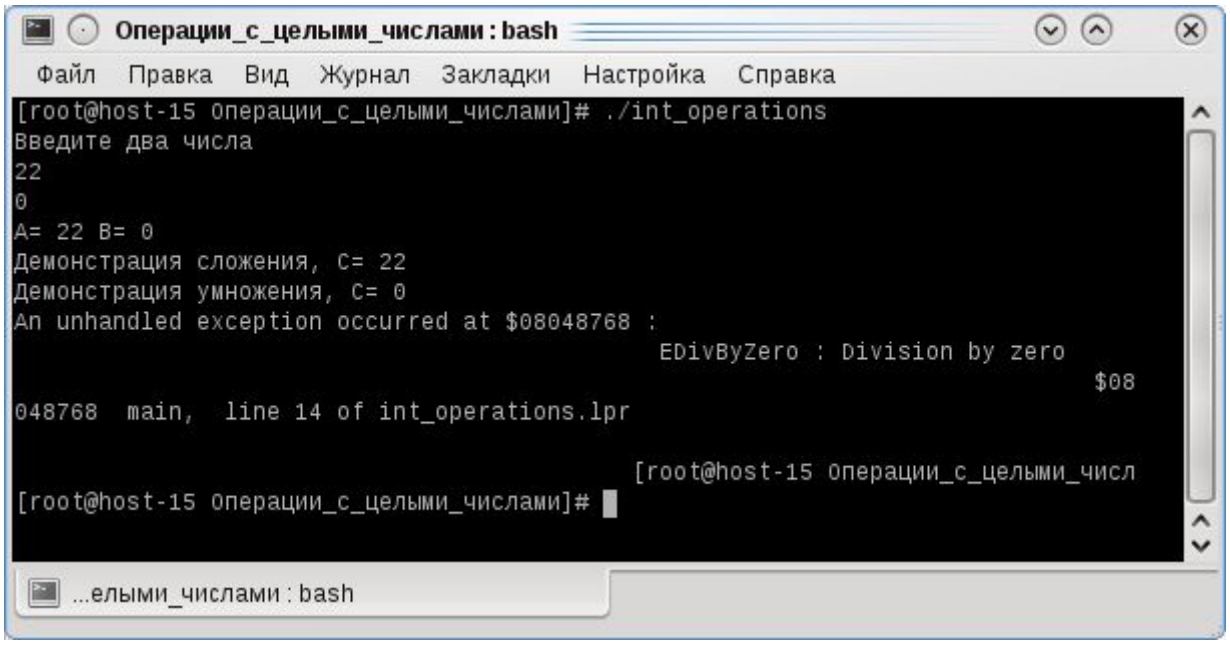


Рис. 5.50. Сообщение об ошибке в консоли

Даже если такого сообщения не выходило, многие читатели, наверное, уже догадались, что мы в этой программе не учли. Да-да, вы абсолютно правы, в программе имеется операция деления. А ведь пользователь мог в качестве второго числа ввести ноль. Но делить на ноль, как известно, нельзя!

Здесь мне бы хотелось снова отступить от конкретики и порассуждать на "общие" темы.

5.1.15. "Вместо лирического отступления 2"

В процессе разработки программы многие начинающие программисты совершенно не обращают внимания на такие, казалось бы, мелочи. А зря! Ваши программы должны быть защищены от любых мыслимых и немыслимых ошибок или непреднамеренных действий пользователя. Казалось бы, в совершенно очевидных ситуациях, когда пользователь ни при каких обстоятельствах не

должен был бы ввести число ноль, из тысячи пользователей найдется хотя бы один, который обязательно введет 0! Даже если вы выведете яркое, написанное крупным шрифтом предупреждение, что нельзя вводить здесь число 0. Тому есть миллион причин. Пользователь мог в это время задуматься о чем-то другом, его могли в этот момент чем-то отвлечь, он мог просто не обратить внимания на ваше предупреждение или забыть о нем. В конце концов, он мог нажать не на ту клавишу! Например, цифра 9 на клавиатуре расположена рядом с цифрой 0. И, наконец, обязательно найдутся такие пользователи, которые захотят посмотреть, а что будет, если я все-таки введу ноль!

Ясно, что если в ваших программах будут встречаться такие "казусы", то вашему престижу как программиста будет нанесен невосполнимый урон, особенно если вы пишете программы на коммерческой основе, т.е. продаете их. Это также отразится на количестве продаж вашей программы.

Таким образом, контроль за такими "не предусмотренными" действиями пользователя лежит на программисте! Есть такое понятие в программировании– писать программы, рассчитанные на "дурака" (fool-tolerance).

Поэтому любите своего пользователя, уважайте его, заботьтесь о том, чтобы ему было легко и комфортно работать с вашей программой (даже есть такое понятие *дружественность* программы), но пишите свои программы так, чтобы они были защищены от любых непреднамеренных, неумелых и даже "невозможных" действий пользователя.

Чаще всего такого рода ошибки возникают при вводе пользователем каких-то данных. Со способами защиты своей программы от таких непреднамеренных действий пользователя мы познакомимся позже (см. раздел 5.1.25. и 9.2.7.). Здесь я просто заострил ваше внимание на этой проблеме, чтобы вы всегда помнили об этом в процессе написания своих программ.

5.1.16. Стандартные функции с целыми аргументами

Рассмотрим программу:

```
program functions;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  a, b, c: integer;
begin
  a:=-2;
  b:= abs(a);
  writeln(UTF8ToConsole('Абсолютная величина числа  a= '), b);
  c:= sqr(b);
  writeln(UTF8ToConsole('Квадрат числа  b= '), c);
  c:= sqr(b + b);
  writeln(UTF8ToConsole('Квадрат числа  (b + b)= '), c);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Оператор `b:= abs(a);` присваивает переменной `b` абсолютное значение числа `a`.

Под абсолютным значением числа понимается значение этого числа, если отбросить знак.

```
abs(-2)= 2
abs(-10)= 10
abs(5)= 5
```

Оператор `c:= sqr(b)`- присваивает переменной c квадрат числа b, т.е.

$c = b^2$, `sqr` (от английского square – квадрат). Число в скобках называется аргументом функции. В качестве аргумента может быть выражение, например, $\sqrt{b^2 - 4ac}$ запишется на Паскале следующим образом:

$$sqrt(sqr(b) - 4 * a * c);$$

где *sqrt* – функция извлечения квадратного корня. Как уже отмечалось, в операторах `write` и `writeln` можно использовать любые допустимые выражения, т.е. можно записывать так:

```
writeln('Квадрат числа (b + b)= ', sqr(b + b));
```

5.1.17. Операции с вещественными числами (тип real).

С вещественными числами можно выполнять различные операции. Все возможные операции иллюстрируются следующей программой:

```
program real_numbers;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  a, b, c: real;
begin
  a:= 17.3;
  b:= 3.4;
  c:= a * b;
  writeln(UTF8ToConsole('Умножение вещественных чисел c = '), c);
  c:= a / b;
  writeln(UTF8ToConsole('Деление вещественных чисел c = '), c);
  c:= a + b;
  writeln(UTF8ToConsole('Сложение вещественных чисел c = '), c);
```

```
  c:= a - b;
  writeln(UTF8ToConsole('Вычитание вещественных чисел c = '), c);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

5.1.18. Форматирование вывода

При выводе значений вещественного типа используется так называемое экспоненциальное представление, в котором используется степень десяти. Вы это могли видеть при выполнении предыдущего примера. Такой вид чисел на экране часто неудобен. В операторах вывода можно использовать *форматирование* для указания *ширины поля вывода*. Вид оператора вывода в этом случае будет таким:

```
write(переменная_1:w:d, … переменная_n:w:d);
writeln(переменная_1:w:d, … переменная_n:w:d);
```

где `w` – общая ширина поля вывода, `d` – количество знаков после запятой, т.е. дробной части числа. `w` и `d` должны быть константами или выражениями целого типа. Для того чтобы общая ширина поля вывода определялась автоматически, указывайте `w = 0`. Например:

```
writeln('a * b = ',c:0:2);
```

В этом случае на экран будет выведено

```
a * b = 58.82
```

вместо

```
a * b = 5.8820000000000000E+001
```

5.1.19. Одновременное использование вещественных и целых чисел.

В программе могут встречаться переменные разных типов:

```
program int_real;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  n, k: integer;
  a, b: real;
begin
  a:= 3.6;
  n:= 4;
  b:= n;
  writeln(UTF8ToConsole('Вещественная переменная b='), b);
  n:= trunc(a);
  writeln(UTF8ToConsole('Операция truncate n= '), n);
  k:= round(a);
  writeln(UTF8ToConsole('Операция round k= '), k);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

В программе мы видим запись `b:= n;` где вещественной переменной `b` присваивается значение целой переменной `n.` Кроме того, в таких записях как `b:= n + 4.6;` или `b:= 3 * 7.2 + n;` встречаются вещественные и целые числа, стоящие в правой части выражения. Такие записи разрешены. Компилятор автоматически преобразует выражение в правой части оператора при-

сваивания к вещественному типу. И наоборот, присвоение вещественных значений целым переменным просто запрещены. Т.е. если написать оператор присваивания

```
n:= 3.14;
```

компилятор выдаст ошибку.

Для этого используются стандартные функции `trunc` и `round`. С помощью функции `trunc` производится преобразование вещественного числа в целое путем отбрасывания всех цифр, стоящих после десятичной точки (`truncate` – усекать), `round` – позволяет преобразовать вещественное число в целое путем округления (`round` – округлять).

5.1.20. Другие стандартные функции с вещественными аргументами

Таблица 5.2

Функция	Запись на Паскале
x^2	`sqr(x)`
$\sqrt{x}$	`sqrt(x)`
$\sin x$	`sin(x)`
$\cos x$	`cos(x)`
$arctgx$	`arctan(x)`
$\ln x$	`ln(x)`
e^x	`exp(x)`

Еще раз напоминаю, что аргументом функции (т.е. то, что стоит в скобках) может быть выражение. Например,

```
s:=sin(a+b*2/5.2-sqr(a));
```

5.1.21. Булевы переменные

Булевы переменные (логические переменные) – это переменные, имеющие только два значения `false` (ложь) и `true` (истина). Над булевыми переменными определены логические операции `not` (логическое отрицание, "НЕ"), `and` (логическое "И"), `or` (логическое "ИЛИ"), `xor` (исключающее "ИЛИ").

Кроме того, определены следующие операции отношения:

`=` равно, `<>` не равно,

`<` меньше, `>` больше

`<=` меньше или равно, `>=` больше или равно

Результатом этих операций отношения являются логические `false` или `true`.

Рассмотрим следующее выражение:

```
x > 3
```

В зависимости от значения `x` это выражение будет либо истинным (`true`), либо ложным (`false`).

Пример.

```
program logic;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x: integer;
  flag: boolean;
begin
  x:= 4;
  flag:= x > 3;
```

```
  writeln('flag = ', flag);
  flag:= x < 3;
  writeln('flag = ', flag);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Допускается размещать справа и слева от знаков отношений арифметические выражения: `X + 6.5 < X + 5` такие выражения называются *логическими* или *булевыми* выражениями.

Рассмотрим программу, где используются логические функции:

```
program logic_1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x: integer;
  L1, L2, Result: boolean;
begin
  x:= 4;
  L1:= x > 3;
  L2:= x < 3;
  writeln(UTF8ToConsole('Булевая переменная L1= '), L1);
  writeln(UTF8ToConsole('Булевая переменная L2= '), L2);
  Result := L1 AND L2;
  writeln(UTF8ToConsole('L1 AND L2 равно '), Result);
  Result := L1 OR L2;
  writeln (UTF8ToConsole('L1 OR L2 равно '), Result);
```

```
  Result := NOT Result;
  writeln (UTF8ToConsole('NOT Result равно '), Result);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

5.1.22. Условные операторы.

Условные операторы – это такие операторы, с помощью которых можно изменять последовательность выполнения операторов программы.

5.1.22.1 Оператор if …. then

Этот оператор имеет вид:

```
if   условие then
  оператор;
```

где оператор – любой оператор Паскаля, условие – логическое выражение.

Если это условие выполняется (т.е. это условие дает значение `TRUE`), то будет выполнен оператор стоящий после слова `then`.

Рассмотрим пример:

```
if X < 3 then
  writeln (X);
```

Здесь записано: "Если X < 3 , то вывести на экран значение X".

Оператор `if…then` можно представить в виде структурной схемы, с помощью которой показывается ход выполнения программы:

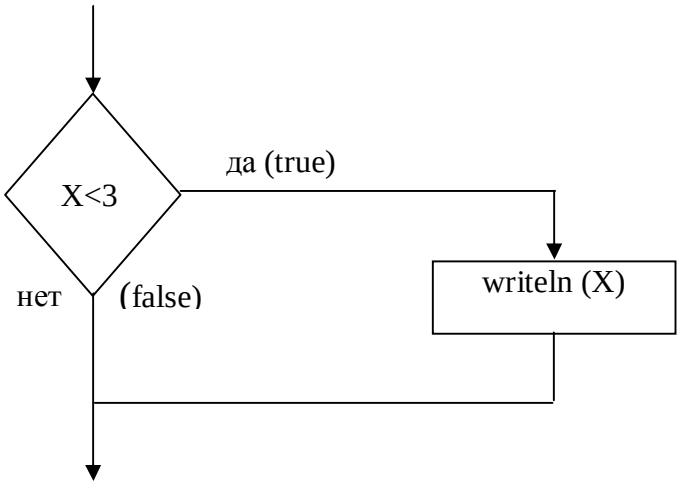


Рис.5 51 Блок-схема выполнения условного оператора if…then

5.1.22.2. Оператор if …then ... else

Этот оператор записывается следующим образом:

```
if  условие then
   оператор 1
else
   оператор 2;
```

Обратите внимание, что перед `else` точка с запятой (;) не ставится. В этом операторе, в зависимости от значения условия, будет выполняться либо оператор 1, либо оператор 2 , что иллюстрирует структурная схема.

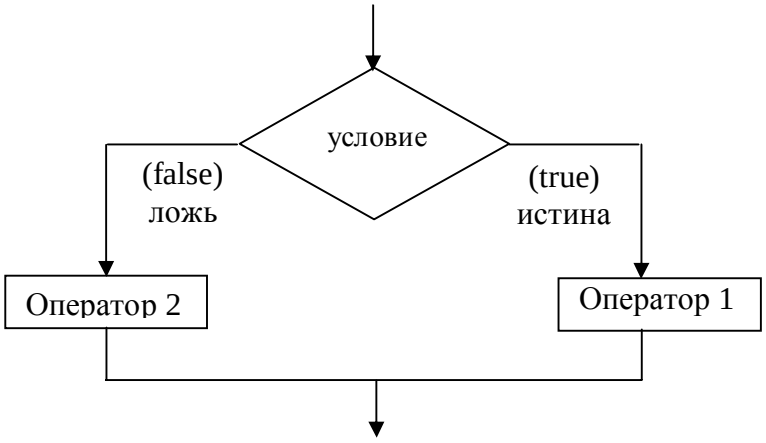


Рис. 5.52 Блок-схема выполнения условного оператора if…then…else

Из этой структурной схемы видно, что выбирается либо один, либо другой вариант продолжения программы.

Например, после выполнения оператора:

```
if X < 2 then
  X1:= 0
else
  X1:= 1;
```

переменной `X1` будет присвоено значение 0, если `X` меньше 2 или 1, если `X` больше или равно 2.

<u>Пример.</u> Вычислить значение функции:

$$y = \begin{cases} x, \ x > 0 \\ 0, \ x = 0 \\ x^2, x < 0 \end{cases},$$

значение x ввести с клавиатуры.

В приведенных далее примерах мы не будем составлять блок-схемы в тайной надежде, что читатель сам их составит в случае необходимости.

```
program ex1;{Вариант 1 - использование оператора if…then }
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  X, Y: real;
begin
  writeln(UTF8ToConsole('Введите значение X' ));
  readln(X);
  if X > 0 then
```

```
    Y:= X;
  if X = 0 then
    Y:= 0;
  if X < 0 then
    Y:= SQR(X);
  writeln('X= ', X:0:2, '; Y= ', Y:0:2);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
program ex2;{Вариант 2 - использование оператора if…then…else }
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  X, Y: real;
begin
  writeln(UTF8ToConsole('Введите значение X'));
  readln(X);
  if X > 0 then
    Y:= X
  else
  if X = 0 then
    Y:= 0
  else
    Y:= sqr(X);
  writeln('X= ', X:0:2, '; Y= ', Y:0:2);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Часто бывает необходимо, чтобы выполнялись или не выполнялись группа из нескольких операторов. Для этого эти операторы объединяются в блок, в котором перед первым оператором ставится слово `begin`, а после последнего слово `end`.

Все операторы между `begin` и `end` образуют так называемый составной оператор, а `begin` и `end` как бы выполняют роль скобок. Их часто так и называют – операторные скобки.

Пример:

```
program demo; {Демонстрация применения составного оператора}
{$mode objfpc}{$H+}
uses CRT, FileUtil;
var X: integer;
begin
  writeln(UTF8ToConsole ('Введите значение X' ));
  readln(X);
  if X < 2 then
  begin     // начало составного оператора
    writeln(UTF8ToConsole('Выполнение программы по условию true'));
    writeln('X = ', X);
  end       // конец составного оператора
{ составной оператор считается как бы одним оператором, поэтому перед else ;
не ставится}
  else
  begin     // начало составного оператора
    writeln(UTF8ToConsole('Выполнение программы по условию false'));
    writeln('X = ', X) ;
  end;      // конец составного оператора
```

```
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Иллюстрация в общем виде: составной оператор записывается внутри служебных слов `begin` и `end`:

```
begin
  s1;
  s2;
  s3;
  s4;
end;
```

где `S1, S2, S3, S4` - операторы Паскаля, в том числе и составные операторы, т.е. составные операторы могут быть вложены друг в друга. Схематично это выглядит так:

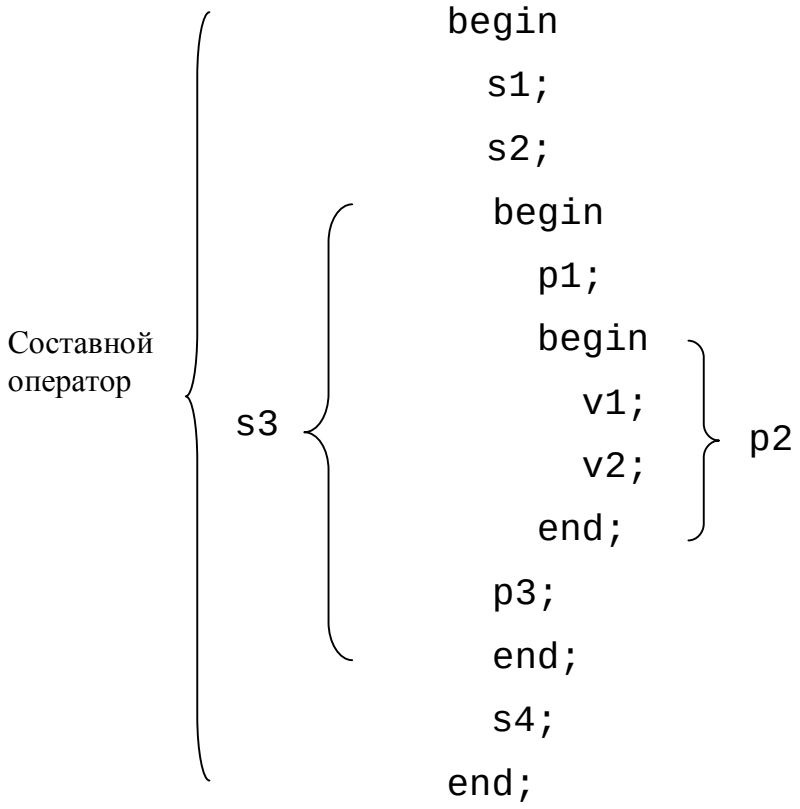


Рис. 5.53 Вложенные составные операторы

Здесь `s3` и `p2` также составные операторы.

Условные операторы также могут быть вложены друг в друга.

```
if x < 2 then
  if y > 0
  then  s:= s + 1
  else  s:= s - 1;
```

Соответствующая блок-схема приведена на рис. 5.54.

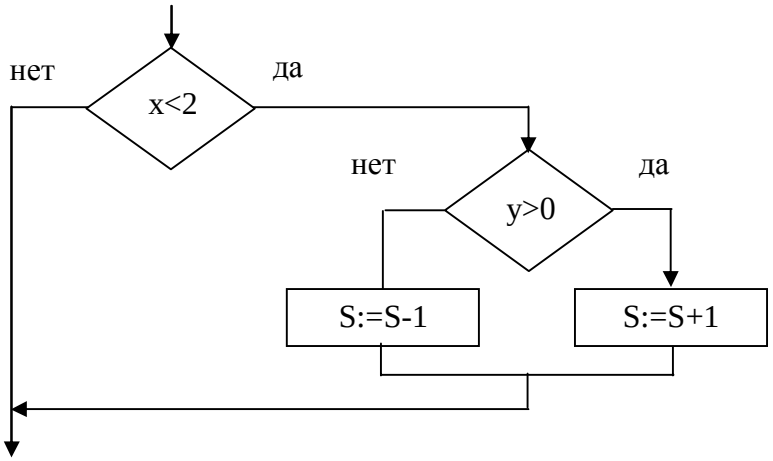


Рис. 5.54 Блок-схема выполнения вложенных условных операторов

5.1.23. Операторы цикла

Операторы цикла используются для организации многократного повторения выполнения одних и тех же операторов. В языке Паскаль существует три типа операторов цикла.

5.1.23.1. Оператор цикла с предусловием

Этот оператор имеет вид:

`while` условие `do` оператор;

где условие – булевское выражение, оператор – любой оператор Паскаля, в частности может быть и составным оператором. Слова `while` и `do` являются

служебными словами, а оператор после `do` часто называют *телом* цикла. Выполняется этот оператор следующим образом: сначала вычисляется значение булевого выражения. Если это значение есть `true`, то выполняется оператор после слова `do` и снова происходит возврат к вычислению булевого выражения. Так повторяется, пока булевое выражение имеет значение `true`. Как только значение булевского выражения станет `false`, то происходит выход из цикла, т. е. оператор после служебного слова `do` уже не выполняется, а будет выполняться следующий после оператора цикла оператор.

В данном операторе вычисление выражения происходит раньше, чем будет выполняться оператор после `do`, поэтому он и называется оператор цикла с предусловием. Может так случиться, что оператор после `do` не будет выполнен вообще, если значение условия с первого раза будет `false`.

Структурная схема цикла с предусловием

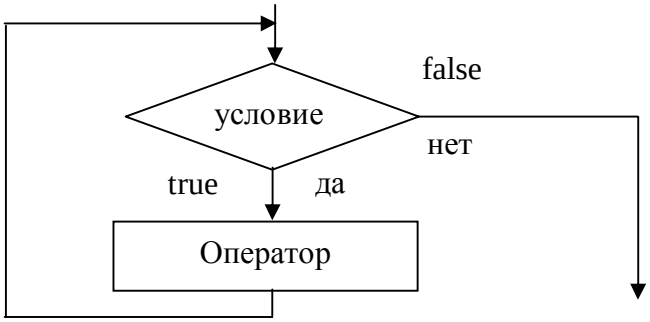


Рис. 5.55 Структурная схема цикла с предусловием

5.1.23.2. Оператор цикла с постусловием

Этот оператор имеет вид:

```
repeat оператор
until  условие;
```

где оператор – любой оператор Паскаля, в том числе и составной, условие – булевское выражение.

`repeat` и `until`-служебные слова.

Этот оператор выполняется следующим образом: сначала выполняется оператор следующий за служебным словом `repeat`, затем вычисляется значение булевского выражения (условия). Если значение условия `false`, то происходит возврат к выполнению оператора и после этого снова вычисляется значение булевского выражения. Так повторяется до тех пор, пока значение булевского выражения `false`. Как только условие станет `true`, выполнение оператора цикла прекращается.

Структурная схема оператора

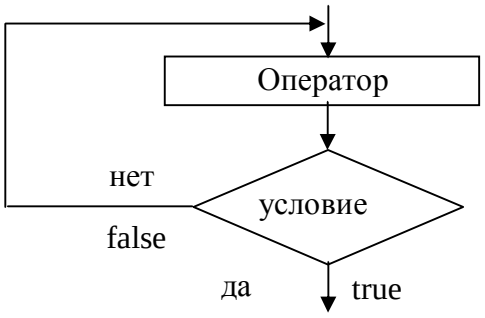


Рис. 5.56 Структурная схема цикла с постусловием

В отличие от оператора цикла `while-do` здесь оператор будет выполнен хотя бы один раз, независимо от значения условий.

Предупреждение! Чтобы рассмотренные выше операторы цикла выполнялись конечное число раз, при построении цикла необходимо предусмотреть, чтобы среди выполняемых операторов обязательно был оператор, который изменял бы значение условия, таким образом, чтобы когда-нибудь значение условия принимало бы `false(`для оператора `while-do)`или `true(`для оператора `repeat- until).`

В противном случае цикл будет повторяться бесконечное число раз и программа "зациклится". Ответственность за правильное применение этих операторов цикла несет на себе программист!

Пример: Вычислить $S = \sum_{x=1}^{100} x$

```
program sum_1; {Вариант 1 цикл с предусловием}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var x, Sum: integer;
begin
  Sum:= 0; //  в этой переменной накапливается сумма
  x:= 1;
  while x <= 100 do
  begin
    Sum:= Sum + x;
    x:= x + 1;
  end;
  writeln('Sum= ',Sum);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.

program sum_2; {Вариант 2 цикл с постусловием}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var x, Sum: integer;
begin
  Sum:= 0; //  в этой переменной накапливается сумма
  x:= 1;
  repeat
```

```
    Sum:= Sum + x;
    x:= x + 1;
  until x > 100;
  writeln('Sum= ',Sum);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Пример: Вычислить функцию:

$$y = \begin{cases} x^2 + 1\ , x > 0 \\ 0\ , x = 0 \\ x^2 - 1\ , x < 0 \end{cases} \quad \text{для } x \in [-10,\ 10] \text{ с шагом } 1$$

Напишем программу вычисления функции с использованием оператора `if…then` и оператора цикла `while…do`:

```
program fun_1;{Вариант 1}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: integer;
begin
  x:=-10;
  while x <= 10 do
  begin
    if x > 0 then
      y:= sqr(x) + 1;
    if x = 0 then
      y:= 0;
```

```
    if x < 0 then
      y:= sqr(x) - 1;
    writeln('x= ', x, '  y= ', y);
    x:= x + 1;
  end;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Напишем программу вычисления этой же функции с использованием оператора `if…then…else` и оператора цикла `while…do`:

```
program fun_2;{Вариант 2}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: integer;
begin
  x:=-10;
  while x <= 10 do
  begin
    if x > 0 then
      y:= sqr(x) + 1
    else
    if x = 0 then
      y:= 0
    else
      y:= sqr(x) - 1;
    writeln('x= ', x ,'  y= ', y);
```

```
    x:= x + 1;
  end;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

В программах fun_1 и fun_2, как вы могли заметить, использовались составные операторы.

```
program fun_3;{Вариант 3 с использованием оператора цикла с постусловием}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: integer;
begin
  x:=-10;
  repeat
  if x > 0 then
    y:= sqr(x) + 1;
  if x = 0 then
    y:= 0;
  if x < 0 then
    y:= sqr(x) - 1;
  writeln('x= ', x, '  y= ', y);
  x:= x + 1;
  until x > 10;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
```

```
  readkey;
end.
```

Заметим, что в операторе цикла с постусловием, если после слова `repeat` используется не один, а несколько операторов, то не обязательно использовать операторные скобки `begin` и `end`, поскольку служебные слова `repeat` и `until` сами выполняют роль операторных скобок.

5.1.23.3. Оператор цикла с параметром.

Иногда заранее точно известно, сколько раз должно быть выполнено определенное действие. Для задач такого типа в языке Паскаль, имеется оператор цикла с параметром. Этот оператор имеет вид:

`for` переменная:= выражение 1 `to` выражение 2 `do` оператор;

где `for`, `to`, `do` – служебные слова; переменная- переменная целого типа, называемая индексом или параметром цикла.

Выражение 1, выражение 2 – арифметические выражения целого типа, т.е. значения выражений должны быть целыми;

оператор – простой или составной оператор.

Для того чтобы оператор цикла выполнялся хотя бы один раз, значение выражения 1 должно быть меньше или равно значению выражения 2 (на практике значения выражения 1 всегда меньше значения выражения 2). Оператор работает следующим образом: вначале переменной (параметру цикла) присваивается значение выражения 1, затем сравнивается значение параметра цикла и значение выражения 2. Если параметр цикла меньше значения выражения 2, то выполняется оператор после слова `do`. Затем параметр цикла увеличивается на 1, после этого вновь сравнивается значение параметра цикла и выражение 2, если параметр цикла меньше, то вновь выполняется оператор после слова

do. И так продолжается до тех пор, пока параметр цикла не станет больше выражения 2. Как только это происходит, оператор цикла заканчивается.

Пример: давайте вычислим снова значения функции

$$y = \begin{cases} x^2 + 1\ , x > 0 \\ 0\ , x = 0 \\ x^2 - 1\ , x < 0 \end{cases}$$ для x $\in$[-10, 10] с шагом 1

```
program fun_4; {оператор цикла с параметром for...to}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: integer;
begin
  for x:=-10 to 10 do
  begin
    if x > 0 then
      y:=sqr(x) + 1
    else
    if x= 0 then
      y:= 0
    else
      y:=sqr(x)-1;
    writeln('x= ', x, '  y= ', y);
  end;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
   end.
```

5.1.23.4. Второй вариант оператора цикла с параметром

Оператор цикла с параметром может быть записан и в таком виде:

`for` переменная:= выражение 1 `downto` выражение 2 `do` оператор;

В этом варианте параметр цикла (переменная) после каждого повторения не увеличивается, а уменьшается на 1. Значение выражения 1 больше или равно (на практике всегда больше) значения выражения 2.

Оператор цикла заканчивается как только параметр цикла станет меньше выражения 2.

Пример:

```
program fun_5; {оператор цикла с параметром for...downto}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: integer;
begin
  for x:= 10 downto - 10 do
  begin
    if x > 0 then
      y:=sqr(x) + 1
    else
    if x = 0 then
      y:= 0
    else
      y:= sqr(x) - 1;
    writeln('x= ', x, '  y= ', y);
  end;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
```

```
  readkey;
end.
```

Пример: Вычислить $S = \sum_{x=1}^{10} x,$

```
program summa_1; {Вариант 1 используется оператор цикла for}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, s: integer;
begin
  s:= 0; // в этой переменной накапливается сумма
  for x:= 1 to 10 do
    s:= s + x;
  writeln('x= ', x, UTF8ToConsole(' сумма s= '), s);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
program summa_2;{Вариант 2 используется оператор цикла while…do}
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, s: integer;
begin
  s:= 0; // в этой переменной накапливается сумма
  x:= 1;
  while x <= 10 do
  begin
```

```
    s:= s + x;
    x:= x + 1;
  end;
  writeln('x= ', x, UTF8ToConsole(' сумма s= '), s);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey; end.
```

Как мы видим, при использовании оператора `for` запись цикла становится короче. Поэтому, если заранее точно известно число повторений цикла или его можно вычислить перед выполнением цикла, то предпочтительней использовать цикл `for`.

5.1.24. Оператор выбора case

Для программирования разветвлений в алгоритме чаще всего используется условный оператор `if…then` или `if…then…else`. Однако если путей выбора много, то запись алгоритма с помощью условного оператора становится громоздкой и труднообозримой. В таких случаях намного удобнее использовать оператор выбора `case`. Этот оператор имеет следующий синтаксис:

```
case <выражение> of
значение 1: оператор 1;
значение 2: оператор 2;
……————————————————————————————
значение n: оператор n;
else
  оператор;
end;
```

В этой конструкции операторы могут быть составными, <выражение> должно быть порядкового типа, т.е. `integer`, `char`, `boolean`, перечислимого

или интервального типа. Более подробно о типах данных мы поговорим в 6.2.

Тип <значение> должен совпадать с типом <выражение>, может быть одно или несколько, разделенных запятыми, а также может представлять собой некоторый диапазон значений. Вся конструкция должна завершаться ключевым словом `end`.

Ветвь `else` вместе с оператором может отсутствовать.

Оператор работает следующим образом:

1. вычисляется значение <выражения>.

2. выполняется оператор, метка которого <значение> совпадает со значением <выражение>.

3. Если ни одно<значение> не совпадает со значением <выражение>, выполняется оператор после `else`.

Рассмотрим применение оператора выбора case на примере создания меню в консольных приложениях.

Пример.

```
program case_menu;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  choose: integer;
begin
  {Формирование меню}
  writeln(UTF8ToConsole('Выберите нужный режим работы    :'));
  writeln(UTF8ToConsole('Создание файла                  1'));
  writeln(UTF8ToConsole('Вывод содержимого файла         2'));
  writeln(UTF8ToConsole('Поиск записей по заданным полям 3'));
```

```
  writeln(UTF8ToConsole('Добавление записей в файл          4'));
  writeln(UTF8ToConsole('Удаление записей из файла          5'));
  writeln(UTF8ToConsole('Корректировка записей в файле      6'));
  writeln(UTF8ToConsole('Выход из программы                 7'));
  repeat
    readln(choose);
    case choose of
    {choose - значение для выбора режима работы}
      1: writeln(UTF8ToConsole('Вы выбрали пункт меню 1'));
      2: writeln(UTF8ToConsole('Вы выбрали пункт меню 2'));
      3: writeln(UTF8ToConsole('Вы выбрали пункт меню 3'));
      4: writeln(UTF8ToConsole('Вы выбрали пункт меню 4'));
      5: writeln(UTF8ToConsole('Вы выбрали пункт меню 5'));
      6: writeln(UTF8ToConsole('Вы выбрали пункт меню 6'));
      7: writeln(UTF8ToConsole('Вы выбрали пункт меню 7' ));
    else
      writeln(UTF8ToConsole('Такого режима нет'));
    end; { end of case }
  until choose = 7;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

5.1.25. Организация простейшего контроля ввода данных.

Вернемся, теперь, к примеру, разобранному в разделе 5.1.14. "Операции с целыми числами", где мы обсуждали вопрос о контроле за вводимыми пользователем данных. Сейчас у нас уже достаточно знаний, чтобы организовать простейший контроль вводимых данных. Для этого необходимо организовать цикл

и, в случае ввода ошибочных данных, дать возможность пользователю повторно ввести необходимые данные. Программа в этом случае будет иметь вид:

```
program int_operations_control;
{$mode objfpc}{$H+}
uses Crt, FileUtil;
var
  A, B, C: integer;
  zero: boolean;
begin
  zero:= true;
  writeln(UTF8ToConsole('Введите два числа'));
  readln(A);
  repeat
    readln(B);
{Проверка введенного числа B на ноль}
    if B = 0 then
    begin
      writeln(UTF8ToConsole('Второе число не может равняться 0'));
      writeln(UTF8ToConsole('Введите другое число не равное 0'));
    end
    else
    zero:= false;
  until not zero;
  writeln('A= ',A,' B= ',B);
  C:= A + B;
  writeln(UTF8ToConsole('Демонстрация сложения, C= '),C);
  C:= A * B;
  writeln(UTF8ToConsole('Демонстрация умножения, C= '),C);
```

```
  C:= A div B;
  writeln(UTF8ToConsole('Демонстрация деления нацело, C= '),C);
  C:= A mod B;
  writeln(UTF8ToConsole('Остаток от деления, C= '),C);
  C:= A - B;
  writeln(UTF8ToConsole('Демонстрация вычитания, C= '),C);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Иногда при выполнении цикла бывает необходимо принудительно завершить цикл, даже если условие выхода из цикла не приняло нужного значения (для циклов `while…do` и `repeat…until)` или параметр цикла не достиг нужной величины (для цикла `for`). Для этого используют инструкцию

```
break;
```

Также достаточно часто встречаются ситуации, когда необходимо начать цикл с начала, даже если не все операторы тела цикла выполнены. В этом случае используют оператор

```
continue;
```

Усовершенствуем нашу программу. Дело в том, что пользователь может в качестве второго числа ввести не только число 0, а вообще ввести недопустимые символы. Например, знаки `+` и `-` для целых чисел допустимы, а остальные символы, в том числе и точка, недопустимы. То же самое относится и для первого вводимого числа.

Для организации контроля за вводом воспользуемся специальной функци-

ей IOResult. Для этого нужно директивой компилятора отключить стандартный контроль ввода/вывода. Эта директива имеет вид {$I-}. Для восстановления стандартного режима контроля ввода/вывода необходимо использовать директиву {$I+}.

Функция IOResult возвращает 0, если операция ввода/вывода завершилась успешно. Программа теперь будет иметь вид:

```
program int_operations;
{$mode objfpc}{$H+}
{$I-} // отключение стандартного режима контроля ввода
uses
  CRT, FileUtil;
var
  A, B, C: integer;
  error: boolean; {булевая переменная для контроля ошибок ввода}
 begin
  error:= false; {ошибок при вводе первого числа нет}
  writeln(UTF8ToConsole('Введите два числа'));
  {Проверка на ошибочный ввод числа A}
  repeat
    readln(A);
    error:= (IOResult <> 0);
    if error then  { если error=true, значит произошла ошибка при вводе}
    writeln(UTF8ToConsole('Ошибка! Введите целое число'));
  until not error;
  error:= false; {ошибок при вводе второго числа нет }
  {Проверка на ошибочный ввод числа B}
```

```
repeat
  readln(B);
  error:= (IOResult <> 0);
  if error then
  begin
    writeln(UTF8ToConsole('Ошибка! Введите целое число'));
    continue; { при выполнении этого оператора последующая
    проверка числа на ноль произведена не будет, выполнение цикла
    начнется с начала, т.е. с оператора readln(B) }
  end;
{Проверка введенного пользователем числа на ноль}
  if B = 0 then
  begin
    writeln(UTF8ToConsole('Второе число не может равняться 0'));
    writeln(UTF8ToConsole('Введите другое число не равное 0'));
  end
until (B <> 0) and (not error);
{$I+}  { восстановление стандартного режима контроля ввода/вывода }
writeln('A= ',A,' B= ', B);
C:= A + B;
writeln(UTF8ToConsole('Демонстрация сложения, C= '), C);
C:= A * B;
writeln(UTF8ToConsole('Демонстрация умножения, C= '), C);
C:= A div B;
writeln(UTF8ToConsole('Демонстрация деления нацело, C= '), C);
C:= A mod B;
writeln(UTF8ToConsole('Остаток от деления, C= '), C);
C:= A - B;
writeln(UTF8ToConsole('Демонстрация вычитания, C= '),C);
```

```
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Разумеется, рассмотренный метод контроля ввода/вывода не является единственным. Существуют более надежные и общие методы контроля, о которых мы будем говорить в главе 9, частности в разделе, посвященном исключениям.

5.1.26. Вычисление сумм сходящихся рядов

Вычислить сумму ряда $S = \frac{1}{x} + \frac{1}{x^2} + \frac{1}{x^3} + ... + \frac{1}{x^n} + ...,$

x∈[1.5, 2] с шагом h=0.5, точность $\varepsilon = 10^{-5}$

Составим блок-схему алгоритма:

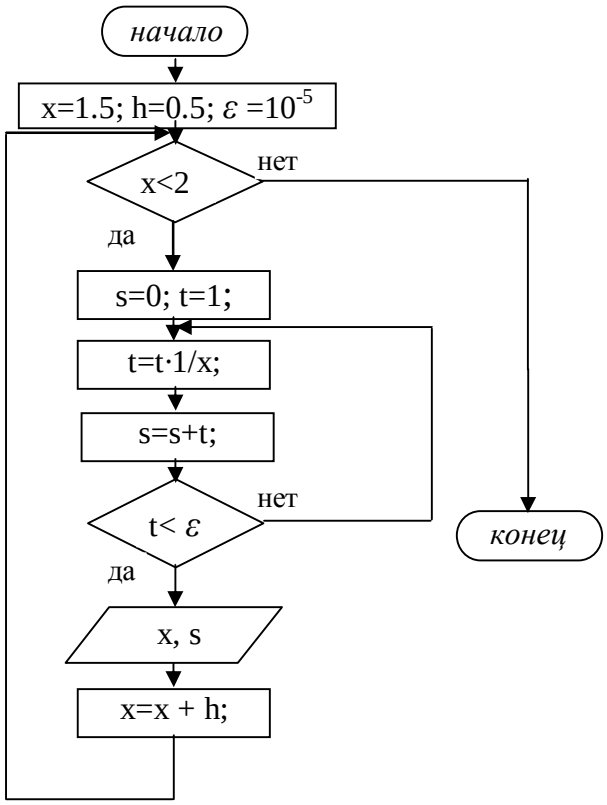


Рис. 5.57 Алгоритм вычисления суммы ряда

Из этой блок-схемы мы видим, что циклы могут быть вложены друг в дру-

га, так называемый цикл в цикле. Внутренний цикл вычисляет сумму ряда для текущего значения переменной *x*. Внешний цикл меняет значение *x*, после этого снова начинает свою работу внутренний цикл, который вычисляет сумму ряда уже для другого значения *x*. Так происходит до тех пор, пока не будут перебраны все значения *x*.

Запись $s=s+t$ следует рассматривать в "динамике", т.е. к текущему значению суммы *s* прибавляется очередной член ряда *t* и полученное значение снова запоминается в переменной *s*. Таким образом в переменной *s* накапливается сумма ряда.

Теперь что означает вычислить сумму ряда с точностью ε. Это значит, что мы округляем вычисленную сумму ряда с числом знаков после запятой указанной в ε. То есть, если очередной член ряда окажется меньше ε (вообще говоря, по абсолютной величине), то добавление этого члена в сумму не окажет влияния на результат и не скажется на итоговой сумме, поскольку мы его округляем с заданной точностью ε.

Все эти рассуждения верны для сходящихся рядов, вычисление суммы расходящегося ряда не имеет смысла.

Рассмотрим в блок-схеме фрагмент, где вычисляется очередной член ряда *t*. Многие здесь допускают типичную ошибку. Для вычисления x^n используют формулу $x^n = e^{n\ln(x)}$. Так можно вычислять степень, но при таком способе ваша программа будет выполняться по времени на порядок дольше, чем при способе, примененном нами. Ведь используя вышеприведенную формулу, вы заставляете компьютер вычислять x^n при каждом новом *n* как бы заново, с нуля. Но присмотритесь внимательно к ряду. Чтобы вычислить очередной член ряда надо предыдущий член ряда умножить на $\frac{1}{x}$. И все!

Хотя на современных компьютерах разница во времени выполнения для таких простых программ практически не заметна, приучайтесь писать программы с оптимальным кодом! В данном случае оптимальность заключается в

уменьшении количества арифметических операций для вычисления очередного члена ряда, а значит выигрыш во времени выполнения программы.

```
program summ;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, s, h, eps, t: real;
begin
  x:= 1.5;
  h:= 0.5;
  eps:= 0.0000000001;
  while x <= 2 do
  begin
    s:= 0;
    t:= 1;
    repeat
      t:= t * 1/x;
      s:= s + t;
    until t <= eps;
    writeln('X= ', X:0:2, ' S= ', S:0:5);
    x:= x + h;
  end;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Вычислить значения функции `sinx` используя его разложение в ряд Тейлора:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

x∈[0,1], значения *x* заданы в радианах, h=0.2, точность $\varepsilon = 10^{-7}$

Вывести также "точное" значение функции, вычисленное с помощью стандартной встроенной функции Паскаля `sin(x)`

Составление блок-схемы алгоритма предоставляется самому читателю. Здесь сложности может вызвать только процесс организации вычислений для определения очередного члена ряда по уже вычисленному предыдущему члену. Внимательно проанализируйте ряд, включите "соображалку" и вы без труда придумаете, как это сделать. Рекомендую также не спешить смотреть программу, а попытаться самому написать и отладить ее!

```
program sinus;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y, s, h, eps, t: real;
  n: integer;
begin
  x:= 0;
  h:= 0.2;
  eps:= 1e-7;
  while x <= 1 do
  begin
    s:= x;
    t:= x;
    n:= 2;
    repeat
```

```
      t:=-t * sqr(x)/(n * (n + 1));
      s:= s + t;
      n:= n + 2;
    until abs(t) < eps;
    y:= sin(x);
    writeln('x= ', x:0:2,
           UTF8ToConsole('Приближенное значение синуса = '),
           s:0:7, UTF8ToConsole('Точное значение синуса = '),
           y:0:7);
    x:= x + h;
  end;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Как проверить, правильные результаты выдает программа или нет? Можно использовать так называемые *модельные исходные данные*, при которых априори известно, какими должны быть результаты. Для синуса, например, известно, что $\sin 0^\circ = 0$, $\sin\frac{\pi}{2} = 1$. Если в этих точках ваша программа выдает правильные результаты, можно с определенной долей уверенности считать, что ваша программа работает правильно.

Есть и другой способ – сравнить результаты, выдаваемые вашей программой с другой программой, которая решает ту же задачу.

В нашем случае мы использовали встроенную функцию `sin(x)`. Важно понимать, что эту функцию писали тоже люди, программисты, разработавшие `Free Pascal`. Скорее всего, они тоже использовали разложение функции `sinx` в ряд. И у нас нет никаких оснований не доверять им! Поэтому будем считать, что встроенная функция `sin(x)` "точная" и если наша программа выдает те же результаты, то, можно считать, что программа работает правильно.

5.2. Реализация некоторых алгоритмов главы 4.

Настало время для реализации алгоритмов разобранных в главе 4, в разделах 4.1 и 4.3, кроме программы решения квадратного уравнения, которое, надеюсь, вы уже давно сами написали.

5.2.1. Программа решения задачи о поездах и мухе

```
program mukha;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var d, v, v1, v2, x, y, s, t: real;
    F: boolean;
begin
   {Блок определения исходных данных}
   {Можно заменить вводом их с клавиатуры.
 Тогда программа станет более универсальной,
 в том смысле, что можно задавать разные расстояния и скорости}
  d:= 600; v:= 200;
  v1:= 40; v2:= 60;
  y:= d;
  s:= 0;
  F:= false; //  сначала идем по правой ветке алгоритма
  while y > 1e-2 do
  begin
    if not F then
```

```
    begin  // это правая ветка алгоритма
      F:= true;
      t:= y/(v + v1);
    end
    else
    begin  // это левая ветка
      F:= false;
      t:= y/(v + v2);
    end;
    x:= t * v;
    s:= s + x;
    y:= y – t * (v1 + v2);
    writeln('x= ', x:0:4, '   s= ', s:0:2);
  end;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

5.2.2. Программа вычисления определенного интеграла

Напишем программу вычисления интеграла

$$\int_0^{\frac{\pi}{2}} \sin x dx$$

по формуле Симпсона методом двойного пересчета. Напомним, что блок-схему вычисления интеграла мы рассматривали в 4.3.2.

```
program integral;
```

```
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  a, b, h, x, s, s1, eps: real;
  n, k: integer;
begin
 {задаем интервал, на котором вычисляется интеграл}
  a:= 0;
  b:= pi/2;
  k:= 0; // при первом вычислении интеграла k=0
  eps:= 1e-5; // заданная точность вычисления интеграла
  n:= 4; // начальное число точек разбиения интервала (a, b)
  h:= (b - a)/n; // шаг вычисления подынтегральной функции
  while true do
  begin
    x:= a;
    x:= x + h;
    s:= 0;
    while x < (b - h) do
    begin
      s:= s + sin(x) + 2 * sin(x + h);
      x:= x + 2 * h;
    end;
    s:= 2 * s;
    s:= (h/3) * (sin(a) + 2 * sin(b) + s);
    if k = 0
    then
    begin
```

```
      k:= 1;
      s1:= s;
      h:= h/2;
      continue;
    end
    else
    if abs(s1 - s) > eps then
    begin
      s1:= s;
      h:= h/2;
      continue;
    end
    else
      break;
  end;
  writeln(UTF8ToConsole('Значение интеграла s= '), s:0:4);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

При заданных пределах интегрирования значение интеграла s= 1.0000

Проверим правильность полученных результатов прямым вычислением интеграла.

$$\int_0^{\frac{\pi}{2}} \sin x dx = -\cos\Big|_0^{\frac{\pi}{2}} = -(\cos(\frac{\pi}{2}) - \cos(0)) = -(0-1) = 1$$

Изменим верхний предел интегрирования на $b = \pi$, для этого в программе измените оператор

```
b:= pi/2;
```

на

```
b:= pi;
```

Осуществите новый прогон программы. Получим ответ s= 2.0000

Снова проверим прямым вычислением интеграла:

$$\int_0^{\pi} \sin x dx = -\cos\Big|_0^{\pi} = -(\cos(\pi) - \cos(0)) = -(-1-1) = 2$$

И, наконец, попробуем сделать верхний предел интегрирования $b=\frac{\pi}{4}$

Программа выдаст ответ s=0.2929, снова проверим:

$$\int_0^{\frac{\pi}{4}} \sin x dx = -\cos\Big|_0^{\frac{\pi}{4}} = -(\cos(\frac{\pi}{4}) - \cos(0)) = -(-\frac{\sqrt{2}}{2} - 1) = 0.70710 - 1 = 0.2929$$

Таким образом, мы можем смело утверждать, что наша программа правильно вычисляет данный интеграл.

Глава VI Более сложные элементы языка

6.1. Общая структура Паскаль – программы

Общая структура Паскаль – программы имеет вид:

```
program < Имя программы >;
label
   < Метки >;
const
   < Константы >;
type
   < Новые типы данных >;
var
   < Описание переменных >;
   < Процедуры >;
   < Функции >;
begin
   < Тело главной программы >;
end.
```

С основными элементами этой структуры мы уже знакомы. Нами остались не рассмотренными <Метки>,<Процедуры>,<Функции>.

Метки нужны для организации перехода на какой – либо оператор. Для этого оператор помечается меткой. Оператор перехода имеет вид:

```
goto <Метка>;
```

и позволяет передать управление оператору с меткой <Метка>.

Программа, содержащая метки обычно плохо читается и, как правило, является не структурированной. Вполне можно писать программы вообще не используя метки. Итак, забыли про метки и поехали дальше!

6.1.1. Процедуры и функции

При разработке больших программ почти всегда появляются часто повторяющиеся фрагменты кода. Чтобы не повторять эти фрагменты в разных частях программы, их можно записать один раз, присвоить им какие-нибудь имена и использовать их в нужных местах программы. Такие именованные фрагменты кода называются подпрограммами.

Подпрограммы делятся на процедуры и функции. Текст процедуры или функции записывается в разделе описаний, после описания переменных. В дальнейшем для того, чтобы использовать эту процедуру или функцию достаточно указать ее имя. В некоторых случаях процедура (функция) использует некоторые значения, которые должны передаваться из главной программы или из других процедур (функций). Эти значения называются параметрами. Параметры указываются в заголовке процедуры (функции) в скобках. Указывается имя переменной и через двоеточие тип переменной. Если переменных одного типа несколько, то они разделяются запятыми. Параметры разных типов разделяются точкой с запятой.

Отличие функции от процедуры в том, что функция обязательно должна возвратить некоторое вычисленное значение вызывающей программе.

6.1.1.1 Структура процедуры

```
procedure <имя процедуры>[(параметры)];
label
   < Метки >;
const
```

```
   < Константы >;
type
   < Новые типы данных >;
var
   < Описание локальных переменных >;
   < Описание внутренних процедур >;
   < Описание внутренних функций >;
begin
   < Тело процедуры >;
end;
```

Параметры, указанные в заголовке называются формальными параметрами. Для вызова процедуры необходимо просто указать ее имя и параметры (если они имеются). Параметры, указанные при вызове процедуры называются фактическими параметрами.

6.1.1.2. Структура функции

```
function <имя функции>[(параметры)]:тип функции;
label
   < Метки >;
const
   < Константы >;
type
   < Новые типы данных >;
var
   < Описание локальных переменных >;
   < Описание внутренних процедур >;
   < Описание внутренних функций >;
begin
```

```
    < Тело функции >;
end;
```

В заголовке функции указывается ее имя, параметры (если они есть) и тип значения, которую функция должна возвратить. Для вызова функции необходимо указать ее имя и фактические параметры (при их наличии). Причем имя функции можно указывать в выражениях!

В теле функции должен присутствовать хотя бы один оператор, который присваивает имени функции некоторое значение. В противном случае значение функции остается неопределенным, что может привести к непредсказуемым последствиям. При вычислении выражений, имя функции замещается вычисленным значением.

В последнее время входит в моду присваивать значение внутри функции не имени функции, а системной переменной `Result`.

Функция или процедура может в свою очередь вызывать другие функции или процедуры, причем может вызывать и саму себя! Уровень вложенности при этом не ограничен. Хотя на практике стараются избегать слишком глубоко вложенных функций (процедур), так как это увеличивает вероятность появления трудно находимых ошибок и, кроме того, ухудшает читабельность и понимание логики работы программы.

6.1.1.3 Глобальные и локальные переменные

Обычно любая программа, функция или процедура использует какие-то переменные, константы, функции и процедуры. В программе любой объект перед ее использованием должен быть описан в разделе описаний. При этом у каждой переменной, а также константы, функции или процедуры есть своя область видимости, где они могут использоваться. Если объект описан в подпрограмме, то доступ к ней из вызывающей программы невозможен. Этот объект является локальным по отношению к подпрограмме, где он описан, т.е. "не видим" в вызывающей программе. Такие переменные, константы, функции и про-

цедуры называются локальными. В то же время, объекты, описанные в вызывающей программе, доступны вызываемым подпрограммам, т.е. "видимы" им. Такие объекты называются глобальными.

Рассмотрим пример, чтобы глубже понять вышесказанное:

```
program console_app;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x: integer; // Глобальная переменная
procedure local_global; // процедура без параметров
begin
  x:= 25; // Глобальная переменная x доступна процедуре
  writeln(UTF8ToConsole('Значение переменной x'));
  writeln(UTF8ToConsole('внутри процедуры = '), x);
end; // конец процедуры
begin  // начало главной программы
  x := 1;
  writeln(UTF8ToConsole('Глобальная переменная x'));
  writeln(UTF8ToConsole('до вызова процедуры = '), x);
  local_global; // вызов процедуры
  writeln(UTF8ToConsole('Глобальная переменная x'));
  writeln(UTF8ToConsole('после вызова процедуры = '), x);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Как видим, процедура изменила значение глобальной переменной. Следует иметь в виду, что переменная, являющаяся локальной в данной подпрограмме, становится глобальной для всех подпрограмм низшего уровня. Рассмотрим пример:

```
program console_app;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x: integer; // глобальная переменная
procedure local_global; // процедура без параметров
var
  y: integer; // Локальная переменная
procedure A; // Процедура внутри процедуры local_global
begin
  y:= 100; // Внутри процедуры A переменная y является глобальной
  writeln(UTF8ToConsole('В процедуре A y= '), y);
end; // конец вложенной процедуры
begin
  x:= 25;  y:= 0;
  A; // вызов процедуры низшего уровня
  writeln(UTF8ToConsole('После вызова процедуры A y= '), y);
end; // конец процедуры
begin  // начало главной программы
  x:= 1;
  writeln(UTF8ToConsole('Глобальная переменная x'));
  writeln(UTF8ToConsole('до вызова процедуры равна '), x);
```

```
  local_global; // вызов процедуры
  // y:=1; если убрать знак комментария, компилятор выдаст ошибку,
  // A; т.к. локальные объекты невидимы для главной программы
  writeln(UTF8ToConsole('Глобальная переменная x'));
  writeln(UTF8ToConsole('после вызова процедуры равна '), x);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Переменная y является локальной в процедуре `local_global`, в то же время она доступна процедуре A, т.е. для нее она является глобальной.

Обратите внимание, что локальные объекты недоступны вызывающей программе. Таким образом, из главной программы нельзя вызвать процедуру `A` и нельзя изменить значение переменной `y`. Главная программа просто не знает об их существовании! Процедура `local_global` выступает для главной программы "черным ящиком". Для обмена данными между вызывающей программой и подпрограммой не стоит использовать глобальные переменные. Для этого используется механизм передачи параметров, о котором пойдет речь в следующем разделе.

Использование глобальных переменных может привести к трудноуловимым ошибкам, поскольку, когда подпрограмма изменяет значение глобальной переменной, главная программа об этом может и "не знать". Подпрограмма должна быть независима и самодостаточна ("черный ящик"!) в том смысле, что должна решать свою задачу исключительно "своими силами", лишь при необходимости получая от вызывающей программы данные через параметры и также через параметры передавая результаты своей работы. Если это функция, то она должна возвращать единственное значение через свое имя.

Почему локальные переменные "невидимы" вызывающей программе? Дело в том, что локальным переменным память распределяется по-другому, чем

глобальным переменным. Память для глобальных переменных выделяется статически компилятором в области памяти, называемой сегментом данных или статической памятью. Глобальные переменные "живут" в статической памяти от запуска главной программы и вплоть до ее закрытия. Поэтому они доступны всем подпрограммам. Для локальных переменных создается специальная область памяти, называемая стеком памяти. Параметры подпрограммы также размещаются в стеке. После завершения работы подпрограммы, память, отведенная для нее, автоматически освобождается и может быть распределена уже для другой подпрограммы.

Есть еще один тип переменных, которые по существу являются локальными, но память им выделяется в сегменте данных. Это так называемые статические переменные. Тот факт, что они находятся в сегменте данных означает, что такие переменные после выхода из подпрограммы не уничтожаются. Таким образом, если подпрограмма будет вызвана повторно, то значение статической переменной сохранится и ее можно будет использовать.

В Паскале статической переменной служит так называемая типизированная константа. Ее описание имеет вид:

```
const
  Константа: Тип = Значение;
```

Отличие от обычной константы в том, что указывается тип и для таких констант выделяется память. Значение типизированной константы в подпрограмме можно изменять (только не в Delphi! Там типизированная константа является "настоящей" константой).

Пример.

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
```

```
procedure static_var; // процедура
const x: integer = 0;
begin
  writeln(UTF8ToConsole('До изменения x= '), x);
  x:= x + 25;
  writeln(UTF8ToConsole('После изменения x= '), x);
end; // конец процедуры
begin
  writeln(UTF8ToConsole('Изменение статической переменной x'));
  writeln(UTF8ToConsole('внутри процедуры после первого вызова'));
  static_var;
  writeln(UTF8ToConsole('Изменение статической переменной x'));
  writeln(UTF8ToConsole('внутри процедуры после второго вызова'));
  static_var;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

После первого вызова процедуры значение переменной x становится равным 25. После второго вызова x=50, т.е. статическая переменная сохранила свое значение 25 после первого вызова процедуры.

Имена локальных переменных в подпрограмме могут совпадать с глобальными именами вызывающей программы. В этом случае при входе в подпрограмму доступна только локальная переменная, а глобальная становится недоступной, например:

```
program console_app;
{$mode objfpc}{$H+}
uses
```

```
  CRT, FileUtil;
var
x: integer; // Глобальная переменная
procedure local_global; // процедура без параметров
var
  x: integer; // Локальная переменная с тем же именем
begin
  x:= 25;
  writeln(UTF8ToConsole('Локальная переменная x= '), x);
end; // конец процедуры
begin  // начало главной программы
  x:= 1;
  writeln(UTF8ToConsole('Значение глобальной переменной x));
  writeln(UTF8ToConsole(' до вызова процедуры '), x);
  local_global; // вызов процедуры
  writeln(UTF8ToConsole('Значение глобальной переменной x));
  writeln(UTF8ToConsole(' после  вызова процедуры '), x);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Мы видим, что значение глобальной переменной не изменилось. Локальная переменная “перекрыла” глобальную и оператор присваивания

x:= 25;

в процедуре присвоило значение 25 совсем другой переменной, хотя и с тем же именем x.

Таким образом, одноименные глобальные и локальные переменные - это разные переменные. Любое обращение к таким переменным в теле подпрограммы трактуется как обращение к локальным переменным, т. е. глобальные переменные в этом случае попросту недоступны.

Пример использования функции.

Вычислить значение $y = \sin x$ путем разложения $\sin x$ в ряд Тейлора с точностью $\varepsilon = 10^{-3}$, $x \in [0,1]$, $\Delta x = 0.1$ Вычисление оформить в виде функции. Вычисленное значение $\sin x$ для каждого x сравнить с "точным" значением путем применения стандартной функции Паскаля sin(x).

Напомню, что мы уже писали эту программу (см. 5.1.25), но без применения функций. Разложение $\sin x$ в ряд Тейлора имеет вид:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots.$$

Функцию, которая вычисляет *sinx* по данному ряду, назовем `No_standard_sin(x)`.

```
program sinus_fun;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y, y1, dx:real;// глобальные переменные, доступны функции
function No_standard_sin (x: real): real;
var
  eps, s, t: real; // локальные переменные, недоступны
  n: integer;     // вызывающей программе
begin
  s:= x;
  t:= x;
```

```
  n:= 2;
  eps:= 1e-7;
  repeat
    t:= -t * sqr(x)/(n * (n + 1));
    s:= s + t;
    n:= n + 2;
  until abs(t) < eps;
  No_standard_sin:= s; // возврат значения функции
  //Result:= s; // можно и так
end;
begin
  x:= 0;
  dx:= 0.2;
  writeln(UTF8ToConsole('Значения синуса'));
  writeln(UTF8ToConsole('моя функции,  стандартная функции'));
  while x <= 1 do
  begin
    y:= No_standard_sin(x);// вызов моей функции
    y1:= sin(x); // вызов стандартной функции
    writeln('  ', y:0:7,'           ', y1:0:7);
    x:= x + dx;
  end;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Пример.

Вспомним пример из 5.2.2. вычисления интеграла по формуле Симпсона. Перепишем программу, причем оформим вычисление подынтегральной функ-

ции в виде функции, а вычисление интеграла в виде процедуры:

```
program integral;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  a, b, eps: real;

function Fx(x: real): real; // Подынтегральная функция
{Если потребуется вычислить интеграл для другой функции,
достаточно поменять только один оператор}
begin
  Fx:= sin(x); // Для другой функции меняем здесь
end;

{ Процедура вычисления интеграла по формуле Симпсона
 методом двойного пересчета}
procedure Simpson(a, b, eps: real);
var
  x, h, s, s1: real;
  n, k: integer;
begin
  k:= 0; // при первом вычислении интеграла k=0
  n:= 4; // начальное число точек разбиения интервала (a,b)
  h:= (b - a)/n; // шаг вычисления подынтегральной функции
  while true do
  begin
    x:= a;
```

```
    x:= x + h;
    s:= 0;
    while x < (b-h) do
    begin
      s:= s + Fx(x) + 2 * Fx(x + h);
      x:= x + 2 * h;
    end;
    s:= 2 * s;
    s:= (h/3) * (Fx(a) + 2 * Fx(b) + s);
    if k = 0
    then
    begin
      k:= 1;
      s1:= s;
      h:= h/2;
      continue;
    end
    else
    if abs(s1 - s) > eps then
    begin
      s1:= s;
      h:= h/2;
      continue;
    end
    else
      break;
  end;
  writeln(UTF8ToConsole('Значение интеграла s= '), s:0:4);
end; // конец процедуры
```

```
begin // начало основной программы
 {задаем интервал на котором вычисляется интеграл}
  a:= 0;
  b:= pi/4;
  eps:=  1e-5; // заданная точность вычисления интеграла
  Simpson(a, b, eps); // вызов процедуры вычисления интеграла
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

6.1.1.4 Способы передачи параметров

Передача параметров по значению

При таком способе передачи параметров в функцию или процедуру передается копия переменной. Внутри функции или процедуры можно менять значения переданных параметров, однако в вызывающей программе значения параметров остаются неизменными.

Рассмотрим пример:

```
program parameters;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: real;
  n: integer;
{Объявление процедуры с параметром}
procedure example(x, y: real; n: integer);
begin
  x:= 1.5;
```

```
  y:= 2.8;
  n:= 10;
end;
begin
  x:= 1;
  y:= 1;
  n:= 1;
  writeln('x= ', x:0:2, ' y= ', y:0:2, ' n= ', n);
  example(x, y, n);
  writeln('x= ', x:0:2, ' y= ', y:0:2, ' n= ', n);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Как видите, после вызова процедуры значения переменных `x, y, n` не изменились.

В предыдущих двух примерах мы использовали передачу параметров по значению.

Передача параметров по ссылке

Другая ситуация при передаче параметров по ссылке. В этом случае изменение параметра внутри функции (процедуры) влечет за собой и изменение значения переменной в вызывающей программе. Для передачи параметра по ссылке нужно перед именем параметра в заголовке указать ключевое слово `var`. Рассмотрим предыдущий пример, но передадим параметры `x` и `n` по ссылке.

```
program parameters;
```

```
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: real;
  n: integer;
{Объявление процедуры с параметром}
procedure example(var x: real; y: real; var n: integer);
begin
  x:= 1.5;
  y:= 2.8;
  n:= 10;
end;
begin
  x:= 1;
  y:= 1;
  n:= 1;
  writeln('x= ', x:0:2, ' y= ', y:0:2, ' n= ', n);
  example(x, y, n);
  writeln('x= ', x:0:2, ' y= ', y:0:2, ' n= ', n);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Здесь уже значения переменных x и n изменились!

Передача параметров-констант

Если при передаче параметров по значению внутри функции (процедуры) их значения можно изменять, то при передаче параметров как констант внутри функции или процедуры их вообще невозможно изменить. При попытке их изменения компилятор выдаст ошибку. Для передачи параметра как константы нужно перед именем параметра задать ключевое слово `const`.

```
program parameters;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  x, y: real;
  n: integer;
{Объявление процедуры с параметром}
procedure example(var x: real; y: real;
                  const n: integer);
begin
  x:= 1.5;
  y:= 2.8;
  {Попытка изменить параметр-константу}
  //n:= 10; //  здесь, если убрать комментарий, компилятор укажет на ошибку
end;
begin
  x:= 1;
  y:= 1;
  n:= 1;
  writeln('x=  ',  x:0:2,  '  y=  ',  y:0:2,  '  n=  ',  n);
```

```
  example(x, y, n);
  writeln('x= ', x:0:2, ' y= ', y:0:2, ' n= ', n);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

6.1.1.5 Процедуры завершения

Для досрочного завершения функции или процедуры, а также основной программы применяются специальные процедуры `exit` и `halt`. Если `exit` выполняется в функции (процедуре), то ее выполнение немедленно прекращается, даже если не все операторы были выполнены и управление передается вызывающей программе (функции, процедуре). Далее будет выполняться следующий за вызовом этой функции (процедуры) оператор. Если `exit` выполняется в основной программе, то работа программы завершается и управление передается операционной системе.

Процедура `halt` сразу завершает работу программы и передает управление операционной системе. Чаще всего `halt` используют для аварийного завершения программы.

6.2. Еще раз о типах данных

До сих пор при составлении программ мы использовали всего три типа данных – целый, вещественный и логический, к тому же использовали не все их возможности. Однако в Паскале имеется очень широкий набор типов данных, причем язык предусматривает создание пользовательских типов данных. Здесь мы дадим классификацию типов данных в Паскале и рассмотрим некоторые из них.

6.2.1. Классификация типов данных

Все типы в Паскале подразделяются на стандартные и пользовательские типы. Пользовательские типы создаются на основе стандартных. При описании пользовательских типов используется ключевое слово `type`.

К стандартным типам относятся:

- целый;
- вещественный;
- символьный;
- логический.

Кроме того, все типы можно разделить на категории:

- простые;
- строковые;
- даты и времени;
- структурированные;
- указатели;
- объекты;
- классы;
- варианты.

Простые типы, в свою очередь, делятся на порядковые и вещественные. Порядковый тип характеризуется тем, что каждому его значению можно поставить в соответствие целое число – его порядковый номер в совокупности значений. Например, для целого типа само значение числа является его порядковым номером. Для логического типа значению `true` соответствует 1, а значению `false` соответствует 0.

6.2.1.1 Целый тип

Кроме уже знакомого нам типа `integer`, в Паскале имеются и другие це-

лочисленные типы. Они различаются диапазоном представления целого числа в памяти компьютера (размером выделяемой памяти) и способом представления числа (со знаком или без знака). В таблице 6.1 приведены важнейшие характеристики целочисленных типов данных (напоминаю, что мы рассматриваем язык применительно к компилятору Free Pascal).

Таблица 6.1

Тип	Диапазон значений	Размер памяти (байты)	Формат
Byte	0..255	1	без знака
ShortInt	-128..+127	1	со знаком
Word	0..65535	2	без знака
SmallInt	-32768..+32767	2	со знаком
Integer	-2147483648..+2147483647	4	со знаком
LongInt	-2147483648..+2147483647	4	со знаком
LongWord	0..4294967295	4	без знака
Cardinal	0..4294967295	4	без знака
Int64	$-2^{63}..+2^{63}-1$	8	со знаком

Как видите, количество целочисленных типов достаточно велико, однако наиболее часто используемыми являются типы `integer` и `cardinal`. Эти два типа обеспечивают максимальную производительность на 32-битных платформах. Заметим, что указанный диапазон, соответствующий LongInt, верен только для режимов компиляции OBJFPC и DELPHI. В остальных случаях (в т.ч. и по умолчанию) Integer соответствует SmallInt (2 байта).

6.2.1.2. Интервальный тип

Интервальный тип определяется на основе порядкового типа и позволяет ограничить диапазон допустимых значений в виде некоторого интервала вида:

```
Минимальное значение..Максимальное значение
```

При этом символы "`..`" между минимальным и максимальным значениями считаются одним символом, т.е. пробелы между этими точками недопустимы. Естественно, максимальное значение должно быть больше минимального. Описание переменной интервального типа имеет вид:

var <Переменная>: Минимальное значение..Максимальное значение;

Например:

```
var day: 1..31;
    month: 1..12;
    year: 2000..2008;
```

6.2.1.3. Перечислимый тип

В этом типе, как следует из названия, значения задаются простым перечислением через запятую, причем весь список заключается в скобки, например:

```
var education: (student, bachelor, graduate, doctor);
    course: (first, second, third, fourth, fifth);
```

Порядковый номер элемента списка начинается с 0, таким образом student имеет порядковый номер 0, bachelor порядковый номер 1, graduate номер 2, doctor номер 3.

6.2.1.4. Множества

Множество в Object Pascal представляют собой группу элементов, с которыми можно сравнивать другие элементы с целью определения входят эти элементы в состав множества или нет. Множества предоставляют программисту возможность достаточно просто представить коллекцию символов, чисел или других перечислимых типов данных. Объявление множества имеет вид:

Set of <Базовый тип>

Например:

```
var
  num: set of 1..10;
  num1: set of 20..100;
```

Здесь num – множество, состоящее из десяти целых чисел.

num1 – множество состоящее из целых чисел от 20 до 100. Значением пе-

ременной типа множество является набор значений или интервалов порядкового типа, заключенных в квадратные скобки. Такая форма определения множества называется конструктором множества.

Присвоение значения переменным:

```
num:=[1, 3, 5, 7, 9];
num1:=[21..75, 81, 82..95];
```

Чтобы определить принадлежит ли переменная множеству, используется оператор `in`:

```
if 81 in num1 then
 <Какие-то действия >
```

Можно множество заранее не определять, а сразу использовать конструктор множества в операторе `in`:

```
if range in [1..50, 75..100] then
 <Какие-то действия >
```

6.2.1.5. Логический тип

Логический тип имеет только два значения `true` (истина, да) и `false` (ложь, нет). Причем, логическому `true` соответствует порядковое число 1, а `false` 0. Таким образом `true` "больше" `false`!

Возможные логические типы представлены в таблице 6.2.

Таблица 6.2

Тип	Размер памяти (байты)
Boolean	1
ByteBool	1
WordBool	2
LongBool	4

Рекомендуется использовать тип `boolean`, остальные типы введены для совместимости с другими языками.

6.2.1.6. Вещественный тип

Вещественные числа представляются в памяти компьютера в форме с плавающей точкой и позволяют производить вычисления с большой точностью и значительно большим диапазоном значений чисел, в том числе и дробных.

Основные вещественные типы представлены в таблице 6.3.

Таблица 6.3

Тип	Диапазон значений	Число значащих разрядов	Размер памяти (байты)
Real48	$\pm 2.9 \cdot 10^{-39}$ .. $\pm 1.7 \cdot 10^{38}$	11 – 12	6
Real	$\pm 5.0 \cdot 10^{-324}$ .. $\pm 1.7 \cdot 10^{308}$	15 – 16	8
Single	$\pm 1.5 \cdot 10^{-45}$ .. $\pm 3.4 \cdot 10^{38}$	7 – 8	4
Double	$\pm 5.0 \cdot 10^{-324}$ .. $\pm 1.7 \cdot 10^{308}$	15 – 16	8
Extended	$\pm 3.6 \cdot 10^{-4932}$ .. $\pm 1.1 \cdot 10^{4932}$	19 – 20	10
Comp	-2^{63} .. 2^{63}	19 – 20	8
Currency	-922337203685477.5808 +922337203685477.5807	19 – 20	8

Максимальную производительность и точность обеспечивает тип `Extended`. Тип `Currency` минимизирует ошибки округления и его целесообразно применять для денежных расчетов. Тип Comp на самом деле целое 64-х разрядное число, но оно обрабатывается так же как и вещественные типы, т.е. в выражениях полностью совместим с вещественными типами.

6.2.1.7. Указатели

Указатель это переменная особенного типа, в которой содержатся не сами данные, а адрес памяти, где эти данные хранятся. Точнее адрес первого байта этих данных. Таким образом, указатель как бы ссылается на данные посредством своего значения (адреса). Примером указателя в обычной жизни может служить номер телефона. Само по себе это число ничего не значит, но если вы наберете этот номер в своем мобильном телефоне, вы "попадете" к нужному

абоненту.

Указатели бывают *типизированные* и *нетипизированные*. При объявлении типизированного указателя всегда указывается тип данных, на которые ссылается указатель. Описание указателя имеет вид:

`var` имя переменной: `^тип;`

Например:

```
var px: ^integer; // указатель на данные целого типа
    py: ^real;  // указатель на данные вещественного типа
    pname: ^string;  // указатель на данные типа строка
    pphone: ^string[7];
```

Компилятор строго следит за правильностью использования указателя и типом данных, на которые ссылается этот указатель.

Для нетипизированных указателей тип данных не указывается. В этом случае программист сам должен заботиться о правильном использовании указателя и типов данных. При его описании используется ключевое слово `pointer`. Пример описания нетипизированного указателя:

```
var p: pointer;
```

Более подробно мы будем изучать указатели в главе 7.

В дальнейшем мы рассмотрим еще типы данных, в частности, в следующем разделе познакомимся с типами данных, позволяющих обрабатывать символьную информацию.

6.3. Обработка символьной информации в Паскале

6.3.1. Символьные и строковые типы данных.

Первые применения ЭВМ были в основном для решения так называемых вычислительных задач, т.е. задач возникающих в математике, в научно – технических задачах, где требуется решение различных уравнений, вычисление значений функций и т.д. Но применение компьютеров для вычислительных задач составляет всего 20 – 25% по сравнению с их применением в других областях. К таким задачам относятся задачи, возникающие в лингвистике, логике, психологии, теории игр и т.д.

Компьютеры широко применяются для набора различных текстов и документов, перевода текстов. Компьютеры сочиняют музыку, пишут стихи, играют в шахматы. Конечно, при решении таких задач вычисления производятся, но не в таком объеме, как при решении вычислительных задач. В основном в нечисловых задачах компьютер оперирует с символьной информацией.

Необходимо понимать, что вся информация, хранящаяся в памяти компьютера, представлена в виде двоичных чисел. Все дело в том, что под этими двоичными числами понимается, действительно ли это какие-то числа или что-либо другое. Так вот под "другое" имеются в виду символы или совокупность символов. Т.е. каждый символ кодируется в виде двоичных чисел. Только обрабатываются эти числа совершенно по-другому. Программист знает, когда он работает с числами, когда с символами и поэтому предусматривает для каждого случая соответствующие способы работы с этими числами. А в памяти компьютера двоичное представление символа или группы символов может совпадать с каким-нибудь "настоящим" числом.

Примечательно, что возможность применения компьютеров для решения

нечисловых задач понимали еще тогда, когда и компьютеров то вообще не было! Вот что писала знаменитая Ада Лавлейс еще в 1843 году: "Многие не сведущие в математике люди думают, что поскольку назначение аналитической машины Бэббиджа – выдавать результаты в численном виде, то природа происходящих в ней процессов должна быть арифметической и численной, а не алгебраической и аналитической. Но они ошибаются. Машина может упорядочивать и комбинировать числовые значения так же, как и буквы или любые другие символы общего характера. В сущности, при выполнении соответствующих условий она могла бы выдавать результаты и в алгебраическом виде".

В 1963 г. американская организация по стандартизации American Standards Association (ASA) предложила для представления символов, т.е. цифр, букв и других знаков специальный семибитный код, который стал называться кодовой таблицей ASCII (American Standard Code for Information Interchange). Номер, который символ имеет в таблице ASCII, называется кодом этого символа. Символ можно представить, указав его в кавычках, а можно использовать значок #, за которым следует код символа. Например, буква 'A', в таблице `ASCII` имеет номер 65, т е. его код равен 65, тогда можно указать # 65 и это будет означать букву 'A'.

Однако эта кодовая таблица содержала кроме цифр и знаков только буквы английского алфавита. Поэтому был принят стандарт на 8-битную таблицу ASCII, в которой первые 128 символов оставались те же, что и в 7-битной таблице, а символы с 128 по 255 отводились для не английских символов. Позднее Microsoft расширила таблицу ASCII и она была переименована и стала называться ANSI (American National Standards Institute). В таблице 6.4. приведена первая половина (с кодами 0…127) этого стандарта.

Кодировка символов в соответствии со стандартом ANSI

Таблица 6.4.

Код	Символ	Код	Символ	Код	Символ	Код	Символ
0	NUL	32	BL	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	“	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	‘	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DEL	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
25	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
28	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	□

Как мы видим, первая половина таблицы содержит все буквы латинского алфавита, цифры от 0 до 9, а также все наиболее употребимые знаки, такие как знак +, -, /, *, скобки и т.д.

Вторая половина символов с кодами 128…255 меняется для различных на-

циональных алфавитов. С появлением национальных локализаций для второй половины таблицы ASCII было введено понятие «кодовая страница» (code page, CP). Для кодирования русских букв в MS DOS стали применять кодировку CP866, ранее известную как альтернативная кодировка ВЦ Академии Наук СССР.

В Windows для представления кириллицы используется кодовая страница CP-1251. Стандартные Windows-шрифты Arial Cyr, Courier New Cyr и Times New Roman Cyr для представления символов кириллицы (без букв `'ё'` и `'Ё'`) используют последние 64 кода (от 192 до 256): `'А'`…`'Я'` кодируются значениями 192…223, `'а'`…`'я'` – 224…255.

А в консоли Windows используется кодировка CP866. Именно этим объясняются проблемы при выводе русских букв на экран в консольных приложениях.

Существуют и другие стандарты кодировки символов. В частности, для представления букв некоторых языков, таких как китайский, японский, корейский и др. 8-ми разрядов не хватает. Поэтому разработан специальный стандарт `Unicode`.

Юникод, или Уникод (Unicode) — стандарт кодирования символов, позволяющий представить знаки практически всех письменных языков.

Стандарт предложен в 1991 году некоммерческой организацией «Консорциум Юникода» (Unicode Consortium, Unicode Inc.). Применение этого стандарта позволяет закодировать очень большое число символов из разных письменностей: в документах Unicode могут соседствовать китайские иероглифы, математические символы, буквы греческого алфавита, латиницы и кириллицы, при этом становятся ненужными кодовые страницы.

Стандарт состоит из двух основных разделов: универсальный набор символов (UCS, Universal Character Set) и семейство кодировок (UTF, Unicode Transformation Format).

Универсальный набор символов задаёт однозначное соответствие симво-

лов кодам — элементам кодового пространства, представляющим неотрицательные целые числа.

Семейство кодировок определяет машинное представление последовательности кодов UCS.

Коды в стандарте Юникод разделены на несколько областей. Область с кодами от U+0000 до U+007F содержит символы набора ASCII с соответствующими кодами. Далее расположены области знаков различных письменностей, знаки пунктуации и технические символы. Часть кодов зарезервирована для использования в будущем.

В Lazarus по умолчанию используется кодировка UTF-8. В UTF-8 все символы разделены на несколько групп. Символы с кодами менее 128 кодируются одним байтом, первый бит которого равен нулю, а последующие 7 бит в точности соответствуют первым 128 символам 7-битной таблицы ASCII, следующие 1920 символов – кодируются двумя байтами. Последующие символы кодируются тремя и четырьмя байтами.

Для нас важным является тот факт, что символы кириллицы кодируются в UTF-8 в точности двумя байтами.

Как уже отмечалось в 2.1.8. Lazarus представляет собой среду с графическим интерфейсом для быстрой разработки программ и базируется на оригинальной кроссплатформенной библиотеке визуальных компонент *LCL* (Lazarus Component Library). Разработчиками Lazarus предложено использовать UTF-8 в LCL в качестве универсальной кодировки на всех платформах. Поэтому LCL содержит, кроме визуальных компонентов, функции преобразования UTF-8 в кодировку, с которой работает консоль на каждой из платформ. Функция UTF8ToConsole() объявлена в модуле FileUtil, которая является частью LCL. Вот почему мы должны были в наши консольные проекты добавлять библиотеку LCL.

6.3.1.1 Тип Char

Для обозначения типа "символ" в Паскале используется зарезервированное слово `char`. Для хранения переменной типа "символ" требуется один байт памяти, т.е. значением переменной типа `char` является один символ.

Напоминаю, что хотя компьютер обрабатывает символы, тем не менее, в действительности он оперирует с числами, а точнее с кодами этих символов из кодовой таблицы. Таким образом, один символ может быть "больше", чем другой или "меньше". Это зависит от места расположения символов в таблице. Например, символ (буква) 'B' "больше" буквы 'A', поскольку код 'B' (номер в кодовой таблице) равен 66, а код буквы 'A' равен 65 (буквы английского алфавита, см. табл. 6.4.). В силу этого, над символьными переменными определены операции отношения: =, <, >, <>, <=, >=.

6.3.1.2. Функции для работы с символами

Для символьных переменных существуют следующие функции:

`chr(x)` – возвращает значение символа по его коду;

`ord(ch)` – возвращает код символа ch;

`pred(ch)` – возвращает предыдущий символ из кодовой таблицы;

`succ(ch)` – возвращает следующий символ кодовой таблицы;

`upcase(ch)` – преобразует строчную букву в заглавную. Работает только для букв английского алфавита.

6.3.1.3. Тип String

Строкой называется некоторая последовательность символов: 'AABBCC', 'Вася-это кот'.

Строка может состоять из нескольких символов, из одного символа, а может быть совсем пустой. Максимальная длина строковой переменной зависит от директивы компилятора $H. Если включена директива {$H-}, то максималь-

ная длина строки 255 байтов, если же включена директива {$H+}, то длина строки практически неограниченна и может достигать до ~2 Гб.

Тип строковой переменной указывается зарезервированным словом `string`. Если заранее точно известна длина строки, например 30 символов, то можно указать `string`[30].

Строковую переменную можно рассматривать как одно целое, а можно как массив символов, причем нумерация символов начинается с 0.

Пример описания строк и символов.

```
var Symbol: char;
    Message: string;
    Name: string[30];
```

Какие операции допустимы для строк и символов? Только операция сложения. При этом в результате получается строка.

Пример:

```
program ch_str;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  Symbol: Char;
  Mum, Str: string;
begin
  Symbol:='A'; // это буква А латинского алфавита
  Str:= 'Ш';
  Str:= Str + Symbol;
  Mum:= 'МАМА';
```

```
  Mum:= Mum + Str;
  {теперь  Mum = 'МАМАША '}
  writeln(UTF8ToConsole(Mum));
  Str:= '  ПАПАША = ПРЕДКИ';
  Mum:= Mum + ' +' + Str;
  {теперь  Mum = 'МАМАША  +  ПАПАША = ПРЕДКИ '}
  writeln(UTF8ToConsole(Mum));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Обратите внимание, если вы в операторе

```
Symbol:='А';
```

укажете русскую букву 'А', то компилятор выдаст ошибку!

Как уже отмечалось, в Lazarus и FPC используется кодировка UTF8. В этой кодировке для представления кириллических символов используются два байта. Таким образом, переменной типа `char` нельзя присвоить значение кириллицы. Если вы все же хотите использовать символы кириллицы в символьных переменных, то используйте тип TUTF8Char, как в следующем примере.

Пример:

```
program ch_str;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, LCLType;
var
  Symbol: TUTF8Char; //  новый  тип, введенный в FPC
  Mum, Str: string;
```

```
begin
  Symbol:='А'; //  это русская буква А, занимает в памяти два байта
  Str:= 'Ш';
  Str:= Str + Symbol;
  Mum:= 'МАМА';
  Mum:= Mum + Str;
  {теперь  Mum = 'МАМАША '}
  writeln(UTF8ToConsole(Mum));
  Str:= '  ПАПАША = ПРЕДКИ';
  Mum:= Mum + ' +' + Str;
  {теперь  Mum = 'МАМАША  +  ПАПАША = ПРЕДКИ '}
  writeln(UTF8ToConsole(Mum));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Заметьте, что в строковых переменных (типа string) под символы латиницы (коды которых меньше 128 в таблице ASCII) отводится один байт, а под символы кириллицы отводится два байта!

Пример:

Следующая программа имитирует процесс авторизации пользователя для входа в систему. Для этого пользователь должен ввести правильный пароль. Если после трех попыток правильный пароль не будет введен, пользователю будет запрещен доступ в систему.

```
program password;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, LConvEncoding;
var
```

```
  answ, passw: string;
  n: integer;
begin
 passw:= 'абвг';
 n:= 1;
 repeat
   writeln(UTF8ToConsole('Введите пароль'));
   readln(answ);
   {$IFDEF WINDOWS}
      answ:= CP866ToUTF8(answ); // преобразование введенной
                                // строки к UTF8
   {$ENDIF}
   if answ <> passw then
   begin
     if n = 1 then
     begin
       writeln(UTF8ToConsole('Вы не пользователь'));
       inc(n);
     end
     else
     begin
       if n = 3 then
       begin
         writeln(UTF8ToConsole('Вам отказано в доступе'));
         break;
       end
       else
       begin
         writeln(UTF8ToConsole('Вы не пользователь'));
```

```
          writeln(UTF8ToConsole('Вы '), n,
          UTF8ToConsole('-раз ввели неправильный пароль'));
          writeln(UTF8ToConsole('После 3-й попытки вам '));
          writeln(UTF8ToConsole('будет отказано в доступе'));
          inc(n);
        end
      end
    end
    else
    writeln(UTF8ToConsole('Здравствуйте, вы вошли в систему!'));
  until answ = passw;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

В этой программе мы использовали функцию `CP866ToUTF8()` для преобразования введенной с клавиатуры строки к UTF8. Разумеется, это касается только пользователей Windows, так как мы уже неоднократно отмечали, что в консоли Windows используется кодировка CP866. Функция `CP866ToUTF8()` описана в модуле `LConvEncoding`.

Обратите внимание на директивы компилятора для условной компиляции

```
{$IFDEF WINDOWS}
   answ:= CP866ToUTF8(answ);
{$ENDIF}
```

Директива `{$IFDEF <условие>}` предписывает компилятору компилировать операторы, находящиеся после нее и до `{$ENDIF}` если `<условие>` выполняется. В противном случае эти операторы пропускаются. В данном слу-

чае оператор

```
answ:= CP866ToUTF8(answ);
```

будет включен в компиляцию в операционной системе Windows и будет пропущен в Linux. Полная форма этой директивы:

```
{$IFDEF <условие>}
    <операторы> // компилируются, если <условие> выполнено
{$ELSE}
    <операторы> // компилируются, если <условие> не выполнено
{$ENDIF}
```

6.3.1.4. Строковые процедуры и функции

Функция `Concat` – выполняет конкатенацию последовательности строк, т. е. объединение нескольких строк.

Описание: `Concat(S1[,S2,…,Sn]: string): string;`

`S1,S2,…,Sn` – формальные параметры, имеют тип `string`. Квадратные скобки указывают, что параметры `S2,…,Sn` могут отсутствовать. Результат функции строка символов, т.е. имеет тип также `string`.

Пример:

```
program function_concat;
{$mode objfpc}{$H+}
uses
  Crt, FileUtil;
var
```

```
  a, c: string;
begin
  a:= 'Коля + Оля';
  writeln(UTF8ToConsole('Первая строка: '),
          UTF8ToConsole(a));
  c:= ' = любовь';
  writeln(UTF8ToConsole('Вторая строка: '),
          UTF8ToConsole(c));
  c:= Concat(a, c);
  writeln(UTF8ToConsole('Результирующая строка: '),
          UTF8ToConsole(c));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Функция `Length` - возвращает динамическую длину строки.

Описание: `Length(S: string): integer;`

Пример:

```
program function_length;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  S: string;
  i: integer;
begin
  S:= 'Ivanov';
  i:= Length(S);
```

```
  writeln(UTF8ToConsole('Строка S: '), UTF8ToConsole(S));
  writeln(UTF8ToConsole('Длина этой строки = '), i);
  S:= '';
  i:= Length(S);
  writeln(UTF8ToConsole('Строка S: '), UTF8ToConsole(S));
  writeln(UTF8ToConsole('Теперь длина строки стала = '), i);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Следует помнить, что функция `Length()`возвращает количество байтов, занимаемой строкой! В этом легко убедиться, немного видоизменив предыдущую программу.

```
program function_length;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  S, S1: string;
  i: integer;
begin
  S:= 'Ivanov';
  i:= Length(S);
  writeln(UTF8ToConsole('Строка S: '), UTF8ToConsole(S));
  writeln(UTF8ToConsole('Длина этой строки = '), i);
  S1:= 'Иванов';
  i:= Length(S1);
  writeln(UTF8ToConsole('Строка S1: '), UTF8ToConsole(S1));
```

```
  writeln(UTF8ToConsole('Длина этой строки = '), i);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Длина строки `S` равна 6, длина строки `S1` равна 12! Это не ошибка, просто функция `Length()` возвращает не количество символов в строке, а количество байтов, которую занимает строка в памяти. Для латиницы это получается одно и то же, поскольку в UTF8 символы латиницы кодируются одним байтом, а для кириллицы в два раза больше, так как русские буквы кодируются двумя байтами. Если вы хотите получить количество символов в строке с кириллицей, можно воспользоваться функцией `UTF8Length()`. Эта функция объявлена в модуле `LCLProc`.

Замените в предыдущем примере оператор

```
i:= Length(S1);
```

на оператор

```
i:= UTF8Length(S1);
```

и добавьте модуль `LCLProc` в объявление `uses` и запустите заново программу. Теперь длина строки `S1` также стала равна 6.

Попробуйте заменить оператор

```
i:= Length(S);
```

на оператор

```
i:= UTF8Length(S);
```

Вы увидите, что длина строки `S` по-прежнему равна 6. Т.е. функция `UTF8Length()` для строк с латиницей работает идентично функции `Length()`. Отсюда можно сделать вывод, что удобнее применять функцию `UTF8Length()`, особенно, если вам заранее неизвестно содержимое строки.

Процедура `SetLength` - устанавливает длину строки.

Описание:

```
SetLength(S: string, n: integer);
```

`S` – строка, `n` – новая длина строки. Если строка состоит из кириллицы, то необходимо `n` умножать на 2. А если строка содержит символы и кириллицы и латиницы, то придется высчитывать, сколько символов кириллицы содержится в строке. Так что, без особой необходимости применять эту функцию не стоит.

Функция `Copy` – возвращает подстроку строки.

Описание:

```
Copy(S: string, index, count: integer): string;
```

`S`- исходная строка,

`index` - номер символа, начиная с которого выделяется подстрока из `S`,

`count` - количество символов в подстроке.

В модуле `LCLProc` имеется аналог этой функции `UTF8Copy()`.

Пример:

```
program function_copy;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, LCLProc;
var
  a, c: string;
begin
  a:= 'Head&Shoulders';
  writeln(UTF8ToConsole('Исходная строка: '),
```

```
          UTF8ToConsole(a));
  c:= Copy(a, 6, 9);
  c:= UTF8Copy(a, 6, 9); // можно и так
  writeln(UTF8ToConsole('Результирующая строка: '),
          UTF8ToConsole(c));
  a:= 'Меню:Окружение';
  writeln(UTF8ToConsole('Исходная строка: '),
          UTF8ToConsole(a));
  //c:= Copy(a, 6, 9); // это неправильно!
  //c:= Copy(a, 10, 18); // но можно так
  c:= UTF8Copy(a, 6, 9); // лучше всего так
  writeln(UTF8ToConsole('Результирующая строка: '),
          UTF8ToConsole(c));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Процедура `Delete-` удаляет из строки подстроку

Описание:

`Delete(var S:string, index:integer,count:integer);`

`S` – исходная строка, `index` – номер символа, начиная с которого удаляется подстрока из `S`, `count-` количество символов в подстроке. Также имеется аналогичная процедура `UTF8Delete()`.

Пример:

```
program procedure_delete;
{$mode objfpc}{$H+}
```

```
uses
  CRT, FileUtil, LCLProc;
var
  a: string;
begin
  a:= '123XYZ';
  writeln(UTF8ToConsole('Исходная строка: '),
          UTF8ToConsole(a));
  UTF8Delete(a, 4, 3);
  writeln(UTF8ToConsole(' Результирующая строка: '),
          UTF8ToConsole(a));
  a:= '123АБВ';
   writeln(UTF8ToConsole('Исходная строка: '),
          UTF8ToConsole(a));
  UTF8Delete(a, 4, 3);
  writeln(UTF8ToConsole(' Результирующая строка: '),
          UTF8ToConsole(a));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Процедура `Insert` – добавляет подстроку в строку.

Описание:

`Insert(Source: string, S: string, index: integer);`

`Source -` исходная строка

`S-` добавляемая строка

`index -` номер символа строки `Source,` начиная с которого добавляется подстрока `S.`

Пример:

```
program procedure_insert;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, LCLProc;
var
  S: string;
begin
  S:= '1236789';
  writeln(UTF8ToConsole('Исходная строка: '),
          UTF8ToConsole(S));
  Insert('45', S, 4);
  writeln(UTF8ToConsole(' Результирующая строка: '),
          UTF8ToConsole(S));
  S:= 'абвеёжзиклмн';
  writeln(UTF8ToConsole('Исходная строка: '),
          UTF8ToConsole(S));
  UTF8Insert('гд', S, 4);
  writeln(UTF8ToConsole(' Результирующая строка: '),
          UTF8ToConsole(S));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Функция `Pos`- производит поиск подстроки в строке.

Описание:

`Pos(Substr, S: string): Byte;`

`Substr` –подстрока, поиск которой производится в строке `S`. Функция

`Pos` возвращает номер символа, начиная с которого подстрока `Substr` найдена в `S`. Если такой подстроки нет, то возвращается `0`. В модуле LCLProc имеется аналогичная функция `UTF8Pos()`.

Пример:

```
program function_pos_1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  S: string;
  i: integer;
begin
  S:= 'МАМАША';
  i:= Pos('ША', S);
  writeln(UTF8ToConsole('Строка: '), UTF8ToConsole(S));
  writeln(UTF8ToConsole('Номер символа, с которого'));
  writeln(UTF8ToConsole('начинается  подстрока  "ША"= '),i);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Пример: Дана строка символов. Определить, входит ли в эту строку подстрока `'ША'`. Строку ввести с клавиатуры.

```
program function_pos_2;
{$mode objfpc}{$H+}
uses
```

```
  CRT, FileUtil, LConvEncoding , LCLProc;
var
  S: string;
  Substr: string;
  i: integer;
begin
  writeln(UTF8ToConsole('Введите строку символов'));
  readln(S);
  {$IFDEF WINDOWS}
    S:= CP866ToUTF8(S);
  {$ENDIF}
  Substr:= 'ША';
  i:= UTF8Pos(Substr, S);
  if i = 0 then
    writeln(UTF8ToConsole('В данной строке ') +
           UTF8ToConsole(S) + UTF8ToConsole(' подстрока '),
           UTF8ToConsole(Substr) +
           UTF8ToConsole(' отсутствует'))
  else
    writeln(UTF8ToConsole('подстрока ') +
            UTF8ToConsole(Substr) +
    UTF8ToConsole(' входит в данную строку с номера символа '), i);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Процедура `Str` – преобразует численное значение в его строковое представление. Описание:

```
Str(X[: width[: decimals]],S: string);
```

X- числовая переменная (типа integer или real)

width - длина строки S; decimals - количество знаков после запятой.

Пример:

```
program function_str;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  S: string;
  X: real;
begin
  X:=123.565;
  Str(X:6:2, S);
{теперь S='123.56'}
  writeln(S);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Процедура Val преобразует строковое значение в его численное представление.

Описание:

Val(S: string, v, code: integer);

S- исходная строка символов

v- числовая переменная – результат преобразования S

code- целая переменная

Если невозможно преобразовать S в число, то после выполнения процеду-

ры `Val code≠0,` если преобразование произошло удачно, то `code=0.`

Пример:

```
program function_val;
uses
  CRT, FileUtil;
var
  S: string;
  c: integer;
  X: real;
begin
  S:='234.12';
  Val(S, X, c);
  writeln(X);
  writeln(X:0:2);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Процедуру удобно использовать для организации контроля при вводе числовых данных. Для этого сначала необходимо вводить числа в символьном формате, затем преобразовывать в число процедурой `Val` и проверять значение `code`.

Функция `LowerCase` преобразует заглавные буквы в строчные (не работает с кириллицей). Описание:

`LowerCase(const S:String): String;`

Функция `Uppercase` преобразует строчные буквы в заглавные (не работает с кириллицей). Описание:

```
Uppercase (const S:String): String;
```

Удобнее для этих же целей использовать функции `UTF8LowerCase()` и `UTF8UpperCase()` которые поддерживают кириллицу.

```
program project1;
{$mode objfpc}{$H+}
uses
  Crt, FileUtil, LCLProc, LConvEncoding;
var
  str: string;
begin
  writeln(UTF8ToConsole('Введите строку'));
  readln(str);
  {$IFDEF WINDOWS}
    str:= CP866ToUtf8(str);
  {$ENDIF}
  str:= UTF8UpperCase(str);
  writeln(UTF8ToConsole('Строка в верхнем регистре: '),
          UTF8ToConsole(str));
  str:= UTF8LowerCase(str);
  writeln(UTF8ToConsole('Строка в нижнем регистре: '),
          UTF8ToConsole(str));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Рассмотрим вкратце еще несколько функций работы со строками:

- `CompareStr(const S1, S2: string): Integer` — выполняет сравнение двух строк, делая различие между строчными и заглавными буквами;

не учитывает местный язык. Возвращаемое значение меньше нуля, если S1 < S2, равно нулю, если S1 = S2, и больше нуля, если S1 > S2.

- `CompareText(const S1, S2: string): Integer` — выполняет сравнение двух строк, не делая различий между строчными и заглавными буквами; не учитывает местный язык. Возвращаемое значение меньше нуля, если S1 < S2, равно нулю, если S1 = S2, и больше нуля, если S1 > S2.
- `DateTimeToStr(const DateTime: TDateTime): string` — преобразует значение даты и времени в строку.
- `DateToStr(const DateTime: TDateTime): string` — преобразует числовое значение даты в строку.
- `IntToStr(Value: Integer): string` — преобразует целое число Value в строку.
- `StrToDate(const S: string): TDateTime` — преобразует строку со значением даты в числовой формат даты и времени.
- `StrToDateTime(const S: string): TDateTime` — преобразует строку в числовое значение даты и времени.
- `StrToInt(const S: string): Integer` — преобразует строку в целое число. Если строка не может быть преобразована в целое число, функция генерирует исключительную ситуацию класса EConvertError (обработка исключительных ситуаций рассматривается в главе 9).
- `StrToTime(const S: string): TDateTime` — преобразует строку в числовой формат времени. Если строка не может быть преобразована в числовой формат времени, функция генерирует исключительную ситуацию класса EConvertError.
- `TimeToStr(Time: TDateTime): string` — преобразует числовое значение времени в строку.

- `Trim(const S: string): string` — возвращает часть строки S без лидирующих и завершающих пробелов и управляющих символов.
- `TrimLeft(const S: string): string` — возвращает часть строки S без лидирующих пробелов и управляющих символов.
- `TrimRight(const S: string): string` — возвращает часть строки S без завершающих пробелов и управляющих символов.
- `AnsiToUTF 8(const S: string): UTF8String` — преобразует строку в формат UTF8.
- `UTF8Decode(const S: UTF8String): WideString` — преобразует строку формата UTF-8 к строке формата Unicode.
- `UTF8Encode(const WS: WideString): UTF8String` — преобразует строку формата Unicode к строке формата UTF-8.
- `UTF8ToAnsi(const S: UTF8String): string` — преобразует строку формата UTF-8 к стандартной строке.

6.4. Массивы

В некоторых случаях приходится сталкиваться с ситуацией, когда должно использоваться относительно много переменных одного типа.

Представим себе программу, которая позволяет вычислить определенные метеорологические данные и в которой всегда будет не меньше 365 переменных. Таким образом можно присваивать отдельным переменным метеорологические данные по отдельным дням всего года.

В принципе сейчас нет никакой проблемы, чтобы объявить тип `real` у 365 переменных, используя запись.

`var day1, day2, day3` и т.д. `day365:real;`

Однако в Паскале нельзя писать и "т.д.". Нужно перечислить все переменные. Для этого нужна по крайней мере страница. Если теперь нужно подсчи-

тать среднее арифметическое на один день, то нужно записать:

```
sred:=(day1 + day2 + и т.д. + day365)/ 365;
```

и опять как быть с "и т.д."?

Для таких случаев Паскаль предоставляет возможность введения большого числа переменных одного и того же типа, используя простые выражения.

Эта возможность в Паскале реализуется с помощью так называемых массивов. Используется служебное слово `array`, которая указывает на ряд переменных одного и того же типа и имеющих одно имя.

Например:

```
var
  day: array[1..365] of real;
```

С помощью этого описания определяется массив `day` состоящий из 365 элементов (переменных) вещественного типа. Каждый элемент массива может использоваться как отдельная переменная. Можно записать:

```
day[1]:= 1.25;
day[2]:= 0.34;
```

Другими словами, каждый элемент массива определяется именем массива и номером элемента в квадратных скобках. Говорят еще индекс массива. Причем в качестве индекса можно использовать арифметическое выражение целого типа.

В описании массива после имени в квадратных скобках указывается минимальное значение индекса, затем две точки без пробелов и максимальное значение индекса. В нашем случае массив `day` состоит из 365 элементов, нумерация идет от 1 до 365 т.е. шаг нумерации по умолчанию всегда равен 1.

При использовании массивов запись программы становится намного короче и наглядней.

Пусть, например, в начале, все элементы массива `day` должны быть приравнены 0.

Вместо того, чтобы писать

```
day[1]:= 0;
day[2]:= 0;
```

и т.д.

```
day[365]:= 0;
```

мы можем поступить следующим образом:

```
var
  day:array[1..365] of real;
  k : integer;
 begin
   for k:= 1 to 365 do
     day[k]:= 0;
```

Пример.

Пусть имеются 2 массива типа `string`. В одном массиве содержатся фамилии людей, а в другом номера их домашних телефонов. Написать программу, которая в цикле по номеру телефона выводит на экран фамилию этого абонента.

```
program phone_1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  name: array[1..50] of string[30];
  tel: array[1..50] of string[3];
  k: integer;
  phone: string[3];
begin
  for k:=1 to 50 do
```

```
  begin
    writeln(UTF8ToConsole('Введите фамилию'));
    readln(name[k]);
    writeln(UTF8ToConsole('Введите номер телефона'));
    readln(tel[k]);
  {Удаляем ведущие и ведомые ("хвостовые") пробелы}
    tel[k]:= Trim(tel[k]);
  end;
  writeln(UTF8ToConsole('Ввод данных закончен'));
  writeln(UTF8ToConsole('Для поиска абонента введите'));
  writeln(UTF8ToConsole('номер его телефона'));
  writeln(UTF8ToConsole('Для выхода из программы'));
  writeln(UTF8ToConsole('введите "***"'));
  phone:= '';
  while phone <> '***' do
  begin
    writeln(UTF8ToConsole('Введите номер телефона'));
    readln(phone);
    for  k:= 1 to 50 do
      if tel[k] = phone then
        writeln(UTF8ToConsole('Фамилия этого  абонента '),
                name[k]);
  end;
 end.
```

Недостаток этой программы в том, что если указанный номер телефона в массиве не существует, то никаких сообщений на этот счет на экран не выводится. Кроме того, после того как найден абонент, например для k=2, программа ещё 48 раз прокрутит цикл. Как модифицировать программу?

```
program phone_2;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  name: array[1..50] of string[30];
  tel: array[1..50] of string[7];
  k: integer;
  phone: string[7];
  found: boolean;
begin
  for k:= 1 to 50 do
  begin
    writeln(UTF8ToConsole('Введите фамилию'));
    readln (name[k]);
    writeln(UTF8ToConsole('Введите номер телефона'));
    readln(tel[k]);
    {Удаляем ведущие и ведомые ("хвостовые") пробелы}
    tel[k]:= Trim(tel[k]);
  end;
  writeln(UTF8ToConsole('Ввод данных закончен'));
  writeln(UTF8ToConsole('Для поиска абонента введите'));
  writeln(UTF8ToConsole('номер его телефона'));
  writeln(UTF8ToConsole('Для выхода из программы'));
  writeln(UTF8ToConsole('введите "***"'));
  phone:= '';
  while phone <> '***' do
  begin
```

```
    found:= false;
    writeln(UTF8ToConsole('Введите номер телефона'));
    readln(phone);
    for  k:= 1 to 50 do
      if tel[k]= phone then
      begin
        writeln(UTF8ToConsole('Фамилия этого  абонента '),
                name[k]);
        found:= true;
        break;
      end;
    if not found then
      writeln(UTF8ToConsole('Абонента с таким номером нет'));
    end;
 end.
```

Здесь мы ввели булевую переменную `found`, которой присваиваем значение `true`, если абонент с указанным номером найден. Затем мы "досрочно" выходим из цикла `for` командой `break`. Во внешнем цикле, если абонент не найден, выводим соответствующее сообщение на экран.

Если вы несколько раз запускали программу, то могли видеть, что каждый раз приходится заново вводить фамилии и телефоны. Вводить их при каждом новом прогоне программы явно неудобно. Единственный выход – сохранить введенные данные в файле. Но, поскольку, мы еще использование файлов в Паскале не изучали, введем признак окончания ввода. Договоримся, что если вместо очередной фамилии мы введем строку '***', то это означает, что ввод данных закончен. Теперь мы можем вводить лишь несколько фамилий и телефонов только для того, чтобы убедиться в работоспособности программы.

```
program phone_3;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  name: array[1..50] of string[30];
  tel: array[1..50]  of string[7];
  fam: string[30];
  k, n: integer;
  phone: string[7];
  found: boolean;
begin
  n:= 0;
  writeln(UTF8ToConsole('Введите фамилию'));
  writeln(UTF8ToConsole('Чтобы закончить ввод введите "***"'));
  for k:= 1 to 50 do
  begin
    readln (fam);
    if fam = '***' then break;
    name[k]:= fam;
    n:= n + 1;
    writeln(UTF8ToConsole('Введите номер телефона'));
    readln(tel[k]);
    {Удаляем ведущие и ведомые ("хвостовые") пробелы}
    tel[k]:= Trim(tel[k]);
    writeln(UTF8ToConsole('Введите фамилию'));
  end;
  if n = 0 then exit;
  writeln(UTF8ToConsole('Ввод данных закончен'));
```

```
  writeln(UTF8ToConsole('Для поиска абонента введите'));
  writeln(UTF8ToConsole('номер его телефона'));
  writeln(UTF8ToConsole('Для выхода из программы'));
  writeln(UTF8ToConsole('введите "***"'));
  phone:= ' ';
  while phone <> '***' do
  begin
    found:= false;
    writeln(UTF8ToConsole('Введите номер телефона'));
    readln(phone);
    for  k:= 1 to n do
      if tel[k] = phone then
      begin
        writeln(UTF8ToConsole('Фамилия этого  абонента '),
                name[k]);
        found:= true;
        break;
      end;
    if not found then
      writeln(UTF8ToConsole('Абонента с таким номером нет'));
  end;
 end.
```

И опять не останавливаемся на достигнутом, ищем недостатки в программе, улучшаем и совершенствуем ее. Собственно так и поступают все разработчики программ. Они распространяют свои программные продукты версиями, улучшая и совершенствуя свою программу от версии к версии.

В нашем случае мы научились вводить меньше данных, чем зарезервировали в массивах `name` и `tel`, но компилятор распределит память для всего объ-

ема массивов. Значит, часть памяти останется неиспользованной. Налицо нерациональное использование памяти. Хотя для современных компьютеров проблемы с объемами доступной памяти не столь остры, как в недавнем прошлом, все же ни один профессиональный программист не пойдет на резервирование памяти, так сказать, "про запас", использует ровно столько памяти, сколько ему нужно. Для исправления этого недостатка в нашей программе воспользуемся так называемыми *динамическими массивами*.

6.4.1. Динамические массивы

Динамический массив – это массив, память для которого выделяется динамически во время выполнения программы. А в момент компиляции его реальный размер неизвестен. Компилятор просто выделяет память для указателя на этот массив (4 байта).

При описании динамического массива границы не указываются, а указывается только его тип, например:

```
var
  name array of string[30];
  tel array of string[7];
```

Перед использованием такого массива необходимо установить размер массива с помощью процедуры `SetLength`(имя массива, размер), например:

`SetLength`(`name`, 50);

В этом случае для массива `name` будет выделено (динамически!) память для размещения 50 элементов.

Можно использовать и многомерные динамические массивы. Вот как, например, описывается 3-х мерный динамический массив:

```
a: array of array of array of integer;
```

Выделим под этот массив память:

`SetLength`(`a`, 50, 50, 50);

После использования динамических массивов, память, распределенная для них, будет автоматически освобождена. Но можно и "вручную" освободить память, занимаемую динамическим массивом. Для этого достаточно присвоить массиву значение `nil`.

```
a:=nil;
```

Поскольку для реализации динамических массивов используется механизм указателей, применение оператора присваивания

```
b:=a;
```

где a и b динамические массивы не приведет к созданию нового массива b, а будут созданы два указателя, ссылающиеся на одни и те же участки памяти. Т.е. изменение элемента массива b приведет к изменению соответствующего элемента массива a, поскольку фактически это одно и то же данное, только с разными именами (в нашем случае `a` и `b`).

Для того чтобы создать другой массив идентичный по содержанию необходимо использовать процедуру `Copy()`, например:

```
b:=Copy(a);// полная копия массива a
```

Можно скопировать часть массива:

```
b:=Copy(a,5,5);
```

В новый массив b будет скопировано 5 элементов массива a, начиная с пятого элемента.

Необходимо помнить, что нумерация элементов в динамических массивах начинается с нуля.

При передаче динамических массивов в качестве параметров в функции и процедуры узнать реальный размер массива можно с помощью функции `high`(имя массива), но можно просто передать реальный размер массива через дополнительный параметр. К сожалению, в функцию или процедуру можно передавать только одномерные массивы.

Теперь мы можем написать нашу многострадальную программу. В ней мы использовали динамические массивы, а также реализовали ввод данных в виде

процедуры и функцию поиска. В функции для определения фактического размера массива использовали функцию `high()`, а в процедуру передали фактический размер массива через параметр.

```
program phone_4;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  name: array of string[30];
  tel: array  of string[7];
  n: integer;
  phone: string[7];
{Эта функция осуществляет поиск абонента по его номеру телефона}
function find_data (var name: array of string[30];
                    var tel: array  of string[7];
                        phone:string[7]): boolean;
var
  k: integer;
begin
  find_data:= false;
  for  k:= 0 to high(tel) do
  if (tel[k] = phone) and (phone <> '') then
  begin
    writeln(UTF8ToConsole('Фамилия этого  абонента '),
            name[k]);
    find_data:= true;
    break;
  end;
```

```
end;
{Процедура ввода фамилий и номеров телефонов}
procedure data_input(var name: array of string[30];
                     var tel: array  of string[7];
                     var n: integer);
var
  k: integer;
  fam: string[30];
begin
  writeln(UTF8ToConsole('Введите фамилию'));
  writeln(UTF8ToConsole('Чтобы закончить ввод введите "***"'));
  for k:= 0 to n - 1 do
  begin
    // запомним текущий индекс в переменной n
    n:= k; // по его значению мы потом
    // установим фактический размер массивов
    readln (fam);
    if fam = '***' then break;
    name[k]:= fam;
    writeln(UTF8ToConsole('Введите номер телефона'));
    readln(tel[k]);
    {Удаляем ведущие и ведомые ("хвостовые") пробелы}
    tel[k]:= Trim(tel[k]);
    writeln(UTF8ToConsole('Введите фамилию'));
  end;
  writeln(UTF8ToConsole('Ввод данных закончен'));
end;
begin
  n:= 50;
```

```
  {сначала устанавливаем максимальный размер массивов}
  SetLength(name, 50);
  SetLength(tel, 50);
  data_input(name, tel, n); {ввод фамилий и номеров телефонов}
  {теперь устанавливаем фактический размер массивов}
  SetLength(name, n);
  SetLength(tel, n);
  if n = 0 then exit;
  writeln(UTF8ToConsole('Для поиска абонента введите'));
  writeln(UTF8ToConsole('номер его телефона'));
  writeln(UTF8ToConsole('Для выхода из программы'));
  writeln(UTF8ToConsole('введите "***"'));
  phone:= '';
  while phone <> '***' do
  begin
    writeln(UTF8ToConsole('Введите номер телефона'));
    readln(phone);
    if (not find_data(name, tel, phone)) and (phone <>
'***') then
      writeln(UTF8ToConsole('Абонента с таким номером нет'));
  end;
end.
```

Как вы думаете, есть ли в программе еще недостатки? Вообще говоря, можно, конечно, придраться и к столбу, но, справедливости ради, отмечу, что в программе есть еще существенные недостатки.

Во-первых, что будет, если при вводе будут введены одинаковые фамилии и/или телефоны? Будут выведены та фамилия и/или тот телефон, которые были введены первыми. Но в массиве дублирующие данные останутся. Значит,

предварительно необходимо просмотреть уже сформированный массив и только, если фамилия и/или телефон не совпадают, лишь тогда добавлять очередного абонента в массив. Второй значительный недостаток – при вводе телефона не осуществляется никакого контроля. В номере телефона могут быть только цифры, ну, может быть еще знак тире.

Однако разбор этой программы занимает уже чуть ли не половину книги. Поэтому попробуйте сами покопаться. Идеи я вам подкинул. Уверен, что вы справитесь с этой задачей! А самостоятельное решение окажет вам неоценимую пользу.

6.4.2. Программа решения системы линейных алгебраических уравнений методом Гаусса

Напишем программу, реализующую алгоритм Гаусса с выбором главного элемента для решения системы линейных алгебраических уравнений. Сам алгоритм был нами рассмотрен в 4.3.4.

Как уже повелось у нас с вами, прежде чем смотреть программу, напишите его сами по блок схеме, приведенной в разделе 4.3.4. Используя массивы, реализовать этот алгоритм для вас не составит труда. И лишь после этого сопоставьте свою программу с приведенной в книге. Думайте, анализируйте, сравнивайте коды, ищите лучшие решения, нещадно критикуйте меня! Вполне возможно, что вы напишете программу лучше. Лучше не в смысле получаемых результатов (они должны совпадать!), а с точки зрения реализации. Может быть, вы напишете более эффективную программу или реализуете отдельные фрагменты алгоритма более просто, ну и т.д. И имейте в виду, что и сами алгоритмы, решающие одну и ту же задачу, можно составлять по-разному! Так что попробуйте и алгоритм придумать свой.

Для сравнения я приведу три варианта программы, написанных тремя моими студентами. Первый, назовем его "плохим программистом", реализовал алгоритм, как говорится в "лоб". Как записано в блок-схеме, так и реализовано

в программе. На защите своей программы он признался мне, что так и не смог написать эту программу без оператора `goto`. Кроме того, его программа не умела определять существует решение или нет, а также не был организован ввод коэффициентов расширенной матрицы. Его программа умела решать только систему из трех уравнений с тремя неизвестными.

Второй студент, назовем его "средним программистом", сумел написать программу без `goto`, но также действовал в "лоб". Правда его программа уже "умела" вводить коэффициенты расширенной матрицы.

И, наконец, третий студент, назовем его "хорошим программистом", сумел написать очень изящную по реализации программу. Его программа оказалась намного короче по количеству операторов в тексте программы. В программе он использовал динамические массивы, что позволило реализовать алгоритм метода Гаусса для любого числа уравнений. Кроме того, часть кода, где непосредственно реализуется алгоритм метода Гаусса, организовал в виде процедуры. При этом он использовал одну маленькую хитрость. Поскольку в качестве параметров функции или процедуры передавать многомерные массивы нельзя, он предварительно преобразовал матрицу коэффициентов при неизвестных в одномерный массив!

В качестве модельного примера выбрана система:

$$\left.\begin{aligned} 2x_1 + 6x_2 - x_3 &= -12 \\ 5x_1 \quad - x_2 + 2x_3 &= 29 \\ -3x_1 - 4x_2 + x_3 &= \quad 5 \end{aligned}\right\} \text{ ее решением является } \begin{aligned} x_1 &= \ \ 3; \\ x_2 &= -2; \\ x_3 &= \ \ 6 \end{aligned}$$

6.4.1.1. Вариант 1 – с `goto`

```
program Gauss_console_app;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
label
  L1, L2, L3, L4, L5, L6, L7;
var
```

```
  a: array[1..3, 1..3] of real;
  b: array[1..3] of real;
  x: array[1..3] of real;
  i, j, k, p, n: integer;
  m, S, t: real;
begin
  {Определение коэффициентов расширенной матрицы}
  n:= 3;
  a[1,1]:= 2; a[1,2]:= 6; a[1,3]:=-1; b[1]:=-12;
  a[2,1]:= 5; a[2,2]:=-1; a[2,3]:= 2; b[2]:=29;
  a[3,1]:=-3; a[3,2]:=-4; a[3,3]:= 1; b[3]:=5;


  {Основная часть программы}
  k:= 1;
  L1: i:= k + 1;
  if (a[k, k] = 0) then
  begin
    {перестановка уравнений}
    p:= k; // в алгоритме используется буква l, но она
похожа на 1
    // Поэтому используем идентификатор p
    L6: if abs(a[i, k]) > abs(a[p, k]) then p:= i;
    if not( i = n) then
    begin
      i:= i + 1;
      goto L6;
    end;
    if p = k then i:= k + 1
    else
    begin
      j:= k;
      L7: t:= a[k, j];
      a[k, j]:= a[p, j];
      a[p, j]:= t;
      if not(j = n) then
      begin
        j:= j + 1;
        goto L7;
      end;
      t:= b[k];
      b[k]:= b[p];
      b[p]:= t;
    end;
```

```
  end;  // конец блока перестановки уравнений
  L2: m:= a[i, k] / a[k, k];
  a[i, k]:= 0;
  j:= k + 1;
  L3: a[i, j]:= a[i, j] - m * a[k, j];
  if not(j = n) then
  begin
    j:= j + 1;
    goto L3;
  end;
  b[i]:= b[i] - m * b[k];
  if not(i = n) then
  begin
    i:= i + 1;
    goto L2;
  end;
  if not( k= n - 1) then
  begin
    k:= k + 1;
    goto L1;
  end;
  x[n]:= b[n] / a[n, n];
  i:= n - 1;
  L4: j:= i + 1;
  S:= 0;
  L5: S:= S - a[i, j] * x[j];
  if not(j = n) then
  begin
    j:= j + 1;
    goto L5;
  end;
  x[i]:= (b[i] + S) / a[i, i];
  if not(i = 1) then
  begin
    i:= i - 1;
    goto L4;
  end;
  for i:= 1 to n do
  writeln('x', i, '= ', x[i]:0:4);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

6.4.1.2. Вариант 2 – без goto

```
program Gauss_console_app;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  a: array[1..3, 1..3] of real;
  b: array[1..3] of real;
  x: array[1..3] of real;
  i, j, k, p, n: integer;
  m, S, t: real;
begin
  {Ввод коэффициентов расширенной матрицы}
  n:= 3;
  for i:=1 to n do
  begin
    for j:=1 to n do
    begin
      writeln(UTF8ToConsole('Введите a'), i, j);
      readln (a[i, j]);
    end;
    writeln(UTF8ToConsole('Введите b'), i);
    readln(b[i]);
  end;


  {Основная часть программы}
  k:= 1;
  while true do
  begin
    i:= k + 1;
    if (a[k, k] = 0) then
    begin
      {перестановка уравнений}
      p:= k; //  в алгоритме используется буква l, но она похожа на 1
      //  Поэтому используем идентификатор p
      while true do
      begin
        if abs(a[i, k]) > abs(a[p, k]) then p:= i;
        if i = n then break;
        i:= i + 1;
        continue;
```

```
      end;
      if p= k then i:= k + 1
      else
      begin
        j:= k;
        while true do
        begin
          t:= a[k, j];
          a[k, j]:= a[p, j];
          a[p, j]:= t;
          if j = n then break;
          j:= j + 1;
          continue;
        end;
        t:= b[k];
        b[k]:= b[p];
        b[p]:= t;
      end;
    end;  // конец блока перестановки уравнений
    while true do
    begin
      m:=a[i, k] / a[k, k];
      a[i, k]:= 0;
      j:= k + 1;
      while true do
      begin
        a[i, j]:= a[i, j] - m * a[k, j];
        if j = n then break;
        j:= j + 1;
        continue;
      end;
      b[i]:= b[i] - m * b[k];
      if i = n then break;
      i:= i + 1;
      continue;
    end;
    if k= n - 1 then break;
    k:= k + 1;
    continue;
  end;
  {Проверка существования решения}
  if a[n, n] <> 0  then
  begin
```

```
    x[n]:= b[n] / a[n, n];
    i:= n - 1;
    while true do
    begin
      j:= i + 1;
      S:= 0;
      while true do
      begin
        S:= S - a[i, j] * x[j];
        if j = n then break;
        j:= j + 1;
        continue;
      end;
      x[i]:= (b[i] + S) / a[i, i];
      if i = 1 then break;
      i:= i - 1;
      continue;
    end;
    for i:= 1 to n do
    writeln('x', i, '=', x[i]:0:4);
  end
  else
  if b[n] = 0 then
    writeln(UTF8ToConsole('Система уравнений' +
            ' не имеет решения.'))
  else
    writeln(UTF8ToConsole('Система уравнений'+
            '  имеет бесконечное множество решений.'));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

6.4.1.3. Вариант 3 – наилучшая реализация

```
program Gauss_console_app;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  a: array of array of real; {матрица коэффициентов системы,
 двумерный динамический массив}
  vector: array of real; {преобразованный одномерный
```

```
  динамический массив}
  b: array of real;
  x: array of real;
  i, j, k, n: integer;
procedure gauss(var vector: array of real;
                var b: array of real;
                var x: array of real;
                var n: integer);
var
  a: array of array of real; {матрица коэффициентов системы,
 двумерный динамический массив}
  i, j, k, p, r: integer;
  m, s, t: real;
begin
  SetLength(a, n, n); // установка фактического размера массива
    {Преобразование одномерного массива в двумерный}
  k:=1;
  for i:=0 to n-1 do
  for j:=0 to n-1 do
  begin
    a[i,j]:= vector[k];
    k:=k+1;
  end;
  for k:=0 to n-2 do
  begin
    for i:=k+1 to n-1 do
    begin
      if (a[k,k]=0) then
      begin
        {перестановка уравнений}
        p:=k; // в алгоритме используется буква l, но она похожа на 1
              // Поэтому используем идентификатор p
        for r:=i to n-1 do
        begin
          if abs(a[r,k]) > abs(a[p,k]) then p:=r;
        end;
        if p<>k then
        begin
          for j:= k to n-1 do
          begin
            t:=a[k,j];
            a[k,j]:=a[p,j];
            a[p,j]:=t;
```

```
        end;
        t:=b[k];
        b[k]:=b[p];
        b[p]:=t;
      end;
    end; // конец блока перестановки уравнений
    m:=a[i,k]/a[k,k];
    a[i,k]:=0;
    for j:=k+1 to n-1 do
    begin
      a[i,j]:=a[i,j]-m*a[k,j];
    end;
    b[i]:= b[i]-m*b[k];
  end;
end;
{Проверка существования решения}
if a[n-1,n-1] <> 0  then
begin
  x[n-1]:=b[n-1]/a[n-1,n-1];
  for i:=n-2 downto 0 do
  begin
    s:=0;
    for j:=i+1 to n-1 do
    begin
      s:=s-a[i,j]*x[j];
    end;
    x[i]:=(b[i] + s)/a[i,i];
  end;
  writeln('');
  writeln(UTF8ToConsole('Решение:'));
  writeln('');
  for i:=0 to n-1 do
  writeln('x', i+1, '= ', x[i]:0:4);
end
else
if b[n-1] = 0 then
  writeln(UTF8ToConsole('Система не имеет решения.'))
else
  writeln(UTF8ToConsole('Система уравнений'+
  ' имеет бесконечное множество решений.'));
writeln('');
{освобождение памяти,
распределенной для динамического массива}
```

```
  a:=nil;
end;
  {Начало основной программы}
begin
  {Ввод коэффициентов расширенной матрицы}
  writeln(UTF8ToConsole('Введите количество неизвестных'));
  readln(n);
  {Установка реальных размеров динамических массивов}
  SetLength(a, n, n);
  SetLength(vector, n*n);
  SetLength(b, n);
  SetLength(x, n);
   {в динамических массивах индексы начинаются с нуля}
  for i:=0 to n-1 do
  begin
    for j:=0 to n-1 do
    begin
      writeln(UTF8ToConsole('Введите a'), i+1, j+1);
      readln(a[i,j]);
    end;
    writeln(UTF8ToConsole('Введите b'), i+1);
    readln(b[i]);
  end;
  {Преобразование двумерного массива в одномерный}
  k:=1;
  for i:=0 to n-1 do
  for j:=0 to n-1 do
  begin
    vector[k]:=a[i,j];
    k:=k+1;
  end;
  {Вызов процедуры решения системы линейных
  алгебраических уравнений методом Гаусса}
  gauss(vector, b, x, n);
  {освобождение памяти,
  распределенной для динамических массивов}
  a:=nil;
  vector:=nil;
  x:=nil;
  b:=nil;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey; end.
```

Обратите внимание, что массив "a" в главной программе и массив "a" в процедуре это разные массивы, в том смысле, что они будут при выполнении программы занимать совершенно разные участки памяти. Хотя по смыслу задачи это одна и та же матрица коэффициентов, поэтому им присвоено одинаковое имя. Перед передачей матрицы коэффициентов в процедуру двумерный массив преобразуется в одномерный, который и передается, а внутри процедуры одномерный массив преобразуется обратно в двумерный массив с именем "a".

В принципе, можно реализовать алгоритм и с одномерным массивом, главное не запутаться в индексах. Предлагаю вам самим написать такую программу. Организуйте ввод коэффициентов системы сразу в одномерный динамический массив и реализуйте с ним алгоритм метода Гаусса.

Общим недостатком всех трех программ является отсутствие контроля при вводе коэффициентов системы. Однако это сделано намеренно, чтобы не отвлекаться от сути задачи и не загромождать программы излишними деталями. Программы и так получаются достаточно большими. Вы можете самостоятельно вставить проверку при вводе, используя способы, изложенные в 5.1.25. Более продвинутые и профессиональные способы контроля мы будем рассматривать в главе 9.

6.5. Модули в Паскале

Модуль это такая программная единица, которая может и, чаще всего, компилируется отдельно и независимо от главной программы. В модуле могут содержаться определения и реализации часто используемых функций и процедур, а также константы, переменные, объявленные типы. Это позволяет разделить работу между программистами при разработке больших и сложных про грамм. Например, один программист может разрабатывать один модуль, второй программист – другой модуль, а третий главную программу.

Распределением работы, в том числе какие функции и процедуры должны содержаться в том или ином модуле, что они должны делать, какие результаты они должны выдавать, их интерфейс, т.е. список и типы формальных параметров, как правило, занимается руководитель проекта.

При этом каждый программист пишет и отлаживает свой модуль независимо от других.

Текст модуля на языке программирования называется исходным модулем. После компиляции создается так называемый объектный модуль. Объектный модуль это такой модуль, который уже переведен на внутренний машинный язык. На этапе компоновки (сборки) необходимые модули включаются в программу (компонуются) для обеспечения правильных вызовов функций и процедур. Например, главная программа вызывает некую функцию из модуля ***A***, в модуле ***B*** происходит вызов процедуры из модуля ***C*** и т.д. Такой процесс называется разрешением связей. В итоге собирается исполняемая программа.

6.5.1. Структура модуля

Структура модуля имеет вид:

```
unit <Имя модуля>;
interface          //    раздел интерфейса
  <раздел открытых описаний>
implementation   //    раздел реализации
  <раздел закрытых описаний>
initialization  //  раздел инициализации
finalization    //  раздел завершения
end.
```

Модуль начинается со служебного слова `unit`, за которым следует имя модуля. В случае если данный модуль использует другие модули, после слова `interface` необходимо поместить служебное слово `uses` и список исполь-

зуемых модулей.

Интерфейсный раздел модуля начинается со служебного слова `interface`. В этом разделе можно определять константы, типы данных, переменные, процедуры и функции, которые доступны для всех программ и модулей, использующих данный модуль. Глобальные переменные, помещенные в интерфейсной секции, могут быть использованы в основной программе. Интерфейсный раздел модуля начинается со служебного слова `interface`. В этом разделе можно определять константы, типы данных, переменные, процедуры и функции, которые доступны для всех программ и модулей, использующих данный модуль. Глобальные переменные, помещенные в интерфейсной секции, могут быть использованы в основной программе.

Раздел реализации модуля начинается служебным словом `implementation`. В секции реализации могут находиться свои описания, невидимые для программ и модулей, использующих данный модуль. Описанные в секции интерфейса константы, типы данных, переменные, процедуры и функции являются видимыми в секции реализации.

Те процедуры и функции, которые описаны в интерфейсной секции, описываются еще раз в секции реализации, причем их заголовок должен быть точно таким же, как тот, который указан в секции интерфейса.

В секции инициализации помещаются операторы, выполняющиеся только один раз при обращении к данному модулю основной программы. Эти операторы обычно используются для подготовительных операций. Например, в секции инициализации могут инициализироваться переменные, открываться нужные файлы. При выполнении программы, использующей некоторый модуль, секция инициализации этого модуля вызывается перед запуском основного тела программы. При использовании нескольких модулей, их секции инициализации вызываются в порядке, указанном в `uses`. Секция инициализации является необязательной и может вообще отсутствовать.

Раздел `finalization` также является необязательным. В нем выполня-

ются различные действия перед закрытием программы, например закрытие файлов баз данных, освобождение динамически распределенной памяти и т.д.

С целью уменьшения размера основной программы, улучшения читабельности готовые подпрограммы рекомендуется оформлять в виде модуля. Со временем у вас появится своя собственная библиотека модулей.

Имя библиотечного модуля должно совпадать с именем файла, под которым хранится текст модуля на диске. Исходный текст библиотечного модуля имеет расширение *.pas (<имя модуля>.pas).

В качестве примера оформим программу вычисления синуса из раздела 6.1.1.3 в виде модуля.

В меню **Файл** выберите пункт **Создать модуль**. В окне редактора исходного кода появится заготовка кода для модуля. Очистите окно редактора и введите текст модуля:

```
unit my_module;
interface
   function No_standard_sin(x:real):real;
implementation
   {повторяем заголовок функции точно таким
   что и в разделе interface}
   function No_standard_sin(x:real):real;
   var eps,s,t: real;
    n: integer;
begin
  s:= x; t:= x; n:= 2; eps:= 1e-7;
  repeat
    t:= -t * (sqr(x)/(n * (n + 1)));
    s:= s + t;
    n:= n + 2;
```

```
  until abs(t)< eps;
  No_standard_sin:= s;
end;
end.
```

Сохраните модуль обязательно с тем же именем, что указан в заголовке модуля, т.е. my_module.pas в какой-нибудь папке. Откомпилируйте модуль нажав клавиши *Ctrl+F9* или меню *Запуск-> Собрать*. Для каждого модуля создаются два файла: двоичный файл описания модуля с расширением (.ppu) и объектный с расширением (.o). Таким образом, в вашей папке создадутся два файла my_module.ppu и my_module.o

Создайте консольное приложение и введите текст программы:

```
program project_with_my_module;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil,SysUtils,  my_module; // указываем имя модуля
var x, y, dx:real;
begin
  x:= 0;
  dx:= 0.1;
  while x <= 1 do
    begin
      y:=No_standard_sin(x);
      writeln('x= ', x:0:1, ' y= ', y:0:7,
              ' sin(x)= ', sin(x):0:7);
      x:= x + dx;
    end;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
```

```
end.
```

Сохраните проект в той же папке, где вы сохранили модуль или скопируйте файлы my_module.ppu и my_module.o в папку с текущим проектом. Нажмите клавишу F9. Начнется компиляция и сборка (компоновка) программы. На этапе сборки ваш модуль будет автоматически добавлен в исполняемую программу. После выполнения программы вы получите те же самые результаты, что и в разделе 6.1.1.3.

Если вы хотите, чтобы ваш модуль был доступен и в других программах, то проще всего поступить следующим образом:

1. Создайте папку, например с именем my-units
2. Перенесите в эту папку файлы my_module.ppu и my_module.o;
3. Скопируйте саму папку my-units в системный каталог "C:\lazarus\fpc\2.2.4\units\i386-win32" если вы работаете в Windows или в каталог "/usr/lib/fpc/2.2.4/units/i386-linux", если вы работаете в Linux.

Теперь ваш модуль будет автоматически подключаться в ваши программы (разумеется, надо прописать имена модулей в объявлении `uses`). Почему именно в этот каталог? Потому что компилятор если не найдет модуль в папке с вашим проектом, он будет искать его по умолчанию в этой системной папке.

Если же вы разместили папку my-units в другом месте, то вам придется прописывать путь к этой папке, либо в инспекторе проекта, либо непосредственно в объявлении `uses` вот таким образом:

`uses my_module in '`<путь к папке с исходным кодом модуля>`'`

Другой способ – это добавить путь к модулю в файле `fpc.cfg`. Файл находится в папке \lazarus\fpc\2.2.4\bin\i386-win32 в Windows или в папке /etc в Linux. Найдите в этом файле строку

searchpath for units and other system dependent things

И добавьте запись вида:

`-Fu`путь к модулю (именно без пробела!), например

-Fuc:\my-projects\my-units\ (Windows)

-Fu/home/user/my-projects/my-units (Linux)

Но этот файл системный, "лезть" в него без особой необходимости не рекомендую.

В предложенном мною способе "ничего не надо делать"! Надо просто скопировать папку my-units в системный каталог. Путь к нему уже прописан в файле fpc.cfg. Только чтобы не "замусоривать" системный каталог, вам нужно все свои модули, сколько бы их ни было, помещать только в одну папку, в нашем случае мы будем размещать все наши общие модули в папку my-units.

6.5.2. Системные модули

Системные модули Паскаля содержат ряд полезных процедур, функций и констант, которые дают возможность использовать почти всю мощь компьютеров. Рассмотрим вкратце некоторые системные модули:

System – содержит стандартные функции и процедуры "классического" Паскаля, например, вычисления функции cos, sin, ln, exp и др. Модуль System автоматически доступен всем программам. Фактически мы широко пользовались этим модулем.

DOS- содержит процедуры и функции, позволяющие использовать средства операционной системы MS-DOS.

CRT – набор процедур для работы с экраном, клавиатурой.

Graph – содержит обширный набор программ, который позволяет использовать графические возможности компьютера.

В настоящее время этот модуль устарел и практически не используется. Логика его работы слабо совместима с современными системами и проблем с ним уйма. Начинающие часто находят какие-то примеры с его использованием и разочаровываются - часть не работает вообще, в Linux возникают проблемы с

библиотеками, правами доступа и т.д.

Для использования модуля достаточно в программе после строки заголовка вставить строку.

`uses` имя модуля;

Если используется несколько модулей, то пишется так:

`uses` имя модуля 1 , имя модуля 2, ….., имя модуля `N`;

Просмотреть тот или иной системный модуль можно, нажав клавишу `Ctrl` и одновременно подведя указатель мыши к имени модуля. Когда имя модуля заменится на гиперссылку, нажмите на левую клавишу мыши.

В частности, если вы посмотрите модули `SysUtils`, `FileUtil`, `LCLProc` и др., то увидите там объявления и реализации тех функций и процедур, которыми мы широко пользовались.

Рекомендую почаще заглядывать в эти и другие системные модули. Вы найдете для себя немало интересного и поучительного. Заодно посмотрите, как пишут программы профессионалы высокого уровня.

6.5.2.1. Модуль CRT

Модуль `CRT`, как уже говорилось, содержит ряд процедур и функции для работы с экраном в текстовом режиме, клавиатурой и динамиком. Замечу, что этот модуль оставлен для совместимости с Turbo Pascal. В настоящее время программисты чаще используют другие средства. Рассмотрим экран компьютера в текстовом режиме. Обычно на нем размещается 25 строк текста и 80 символов в каждой строке. Для того чтобы полностью владеть экраном, нужно знать координаты курсора, менять цвет символов, текста и фона. Для этого существуют следующие процедуры:

- `GoToXY(i,j)-` позиционирует курсор в `i` строку и `j` столбец экрана.
- `write(s) -` выводит строку `s` начиная с текущей позиции курсора.
- процедура `TextColor(Color)-` устанавливает цвет выводимого текста.

Существуют следующие константы цветов:

`Black` - черный;

`Blue` - синий;

`Green` - зеленый;

`Cyan` - бирюзовый;

`Red` - красный;

`Magenta` - малиновый;

`Brown` - коричневый;

`LightGray` - светло-серый;

`DarkGray` - темно-серый;

`LightBlue` - ярко-голубой;

`LightGreen` - ярко- зеленый;

`LightCyan` - ярко-бирюзовый;

`LightRed` - ярко- красный;

`LightMagenta` - ярко- малиновый;

`Yellow` - желтый;

`White` - белый;

`Blink` - мерцание (мигание);

- Процедура `TextBackGround(Color)` - устанавливает цвет фона.
- Процедура `ClrScr` - очищает экран, одновременно окрашивая его в цвет установленного фона.
- Процедура `Window(x1, y1, x2, y2)`- определяет на экране окно, где `x1, y1`- координаты левого верхнего угла, а `x2, y2`- координаты правого нижнего угла окна

Функция `KeyPressed` - возвращает значение `true,` если клавиша нажата и `false` – в противном случае.

Функция `readkey` – считывает символ с клавиатуры.

Пример: Написать программу, которая очищает экран и устанавливает синий фон. Затем выводит текст желтым цветом.

```
program project1;
uses
  CRT, FileUtil;
begin
  TextBackGround(Blue);
  ClrScr;
  GoToXY(6, 6);
  TextColor(Yellow);
  writeln(UTF8ToConsole('Привет, Вася!'));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Дальнейшие "игры" с цветами и окнами предоставляю вам самому, уважаемый читатель, а сейчас давайте напишем более серьезную программу.

Пример. В некой компании по продаже компьютеров имеются менеджеры, фамилии которых, хранятся в массиве `Name`. В другом массиве имеются данные о количестве проданных компьютеров каждым менеджером за месяц. Написать программу, которая выводит таблицу, в которой следующие графы: фамилия, количество проданных компьютеров, сумма выручки, сумма премии. Итоговая строка должна содержать общее количество проданных компанией компьютеров за месяц, общую сумму выручки, сумму премиальных. Стоимость компьютера принять равной 1000$, размер премии за каждый проданный компьютер 25$

```
program manager;
```

```
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  name: array of string[18];
  kol: array of integer;
  sum, cost, prem, i, k, n: integer;
  progprem: integer;
begin
  writeln(UTF8ToConsole('Введите количество менеджеров'));
  readln(n);
  SetLength(name, n);
  SetLength(kol, n);
  for k:= 0 to n - 1 do
  begin
    writeln(UTF8ToConsole('Введите фамилию'));
    readln(name[k]);
    writeln(UTF8ToConsole('Введите количество компьютеров'));
    readln(kol[k]);
  end;
  ClrScr;
  GoToXY(6,1);
  write(UTF8ToConsole('Сведения о реализации компьютеров'));
  GoToXY(14,2);
  write(UTF8ToConsole('за январь 2010 г.'));
  GoToXY(1,3);
  write('---------------------------------------------');
  GoToXY(1,4);
  write(UTF8ToConsole('Фамилия   Количество Выручка Премия'));
```

```
GoToXY(1,5);
write('-------------------------------------------');
cost:= 1000;
progprem:= 25;
for k:= 0 to n - 1 do
begin
  sum:= cost * kol[k];
  prem:= progprem * kol[k];
  writeln;
  writeln(name[k]);
  GoToXY(24, k + 6);
  write(kol[k]);
  GoToXY(32, k + 6);
  write(sum);
  GoToXY(40, k + 6);
  write(prem);
end;
sum:= 0;
i:= 0;
for k:= 0 to n - 1 do
begin
  i:= i + kol[k];
  sum:= i * cost; {Сумма выручки}
  prem:=i * progprem;
end;
GoToXY(1, n + 6);
```

```
  write('---------------------------------------------');
  GoToXY(17, n + 7);
  writeln(UTF8ToConsole('Итого:'));

  GoToXY(24, n + 7);
  write(i);
  GoToXY(32, n + 7);
  write(sum);
  GoToXY(40, n + 7);
  write(prem);
  GoToXY(1, n + 9);
  write(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

6.6. Файлы

Файл – некоторый участок на диске, где хранятся данные. Каждому файлу присваивается уникальное имя, благодаря которому становится возможным обращаться к различным данным на диске. Файлы состоят из отдельных элементов одного типа. Размер файла зависит от объема того физического устройства, на котором он хранится.

6.6.1. Тип данных – запись

Запись – совокупность данных, имеющих свои имена и тип, объединенных одним именем. Запись - комбинированный тип данных. Каждая запись состоит из отдельных полей, которые могут иметь разные типы.

Общий вид объявления записи:

```
type
   <имя записи>= record
   <имя1>: <тип1>;
   <имя2>: <тип2>;
   ..............
   <имяN>: <типN>;
  end;
```

После этого можно объявить переменные с типом запись:

var <имя переменной>: <имя записи>;

Пример. Пусть имеется таблица следующей структуры:

Фамилия	Имя	Отчество	Группа	Год рождения

Тогда можно объявить запись следующего вида:

```
type
  student = record
  family, name, otch, group: string[20];
  birthday: integer;
end;
var
  a, b: student;
```

Обращение к элементам записи выполняется через уточненное (составное) имя.

```
a.family:= 'Иванов';
b.group:= 'ПОВТАС-1/08';
```

Чтобы не удлинять слишком текст программы, в Паскале предусмотрен оператор присоединения `with`. Это дает возможность определить кусок программы, внутри которого можно просто указывать требуемые поля записи, не

указывая каждый раз при этом имя самой записи.

Синтаксис оператора:

```
with <список имен записей> do
begin
  … {в этом блоке при обращении к полям записи не обязательно каждый раз указывать имя записи}
  ……
end;
```

Пример:

```
type
   student= record
   family, name, otch, group:  string[20];
end;
var
  a, b: student;
begin
  with b do
  begin
    family:= 'Иванов';
    group:= 'ПОВТАС-1/08';
  end;
end.
```

6.6.2. Файловые типы

В программе для работы с файлами вводятся файловые переменные. Переменные файловых типов необходимы тем программам, которым требуется чи-

тать данные с диска или записывать данные на диск. Синтаксис описания файловой переменной:

var <имя файловой переменной>: <тип файла>;

Различают три типа файлов:

- текстовый файл;
- типизированный файл;
- нетипизированный файл

Текстовый файл состоит из последовательности любых символов переменной длины, поделенных на строки. Каждая строка заканчивается специальным признаком "конец строки" – eoln (end of line).

Описание текстового файла имеет вид:

var <имя файловой переменной>: TextFile;

Типизированный файл состоит из элементов одного типа. Однако программист может создать свой тип данных – запись с разнородными типами полей данных. Таким образом, вообще говоря, в типизированном файле можно хранить данные различных типов, но только в структурированном виде – в виде записей.

Описание типизированного файла имеет вид:

var <имя файловой переменной>: File of <тип компонентов>;

И, наконец, в нетипизированных файлах хранятся данные любого типа и структуры. Описание нетипизированного файла имеет вид:

var <имя файловой переменной>: File;

Пример.

```
var
  name: TextFile; // Текстовый файл
  kol: File of integer; // Типизированный файл
  buf: File; // Нетипизированный файл
```

Файл `name` состоит из последовательности строк любых символов переменной длины, файл `kol` из целых чисел, файл `buf` содержит данные любого типа и структуры, т.е. интерпретируются независимо от типа и структуры содержащихся в нем данных.

Для всех типов файлов определяется специальный признак "конец файла" – `eof` (end of file).

6.6.3. Процедуры для работы с файлами

6.6.3.1. Общие процедуры для работы с файлами всех типов

Для связи файловой переменной с файлом во внешней памяти используется процедура:

```
AssignFile(f, fname);
```

где `f` - файловая переменная, `fname` - переменная типа `string`, содержащая имя файла. Процедура `AssignFile` – связывает имя внешнего файла на диске с файловой переменной.

Общий вид имени файла:

<диск>:\<имя каталога>\<имя подкаталога>\...\<имя файла> (Windows)

/<имя каталога>/<имя подкаталога>/.../<имя файла> (Linux)

Если указано только имя файла, то принимается текущий каталог и текущее логическое устройство. Только после выполнения `AssignFile` можно обращаться к другим процедурам по обработке файлов.

```
AssignFile(name, 'Name.dat')
AssignFile(kol, 'C:\Work\Kol.dat')
AssignFile(data, '/home/user/data.dat')
```

Процедура Rewrite – создает и открывает новый файл. Описание:

```
Rewrite(f)
```

f - является файловой переменной. Перед использованием Rewrite переменная f должна быть связана с дисковым файлом с помощью процедуры AssignFile. Если файл с таким именем уже существует, то он удаляется, а на его месте создается новый пустой файл.

Процедура Reset - открывает существующий файл для чтения или записи. Описание:

```
Reset(f)
```

где f - файловая переменная. Перед использованием Reset переменная f должна быть связана с файлом на диске с помощью процедуры AssignFile.

После создания или открытия любого файла создается специальный указатель, с помощью которого отслеживается местоположение текущего элемента в файле. После чтения очередного элемента файла указатель автоматически перемещается к следующему элементу. После записи указатель автоматически перемещается к только что записанному в файл на диске элементу.

Процедура CloseFile(f)- закрывает открытый файл.

Функция Eof() в случае, если достигнут конец файла, возвращает значение true и false , если конец файла еще не достигнут.

Процедура Rename(f, newname); позволяет переименовать файл; newname - переменная типа string, содержащая новое имя файла;

Процедура Erase(f);- удаляет файл.

Функция IOResult – позволяет определить результат последней операции ввода/вывода. Возвращает 0, если ввод/вывод прошел успешно. Если операция ввода/вывода прошла неудачно, возвращает код ошибки.

Функция FileExists(fname) – позволяет определить существует ли

файл fname на диске. Здесь fname - переменная типа string, в ней задается имя файла на внешнем устройстве, чаще всего на диске. Допускается указывать имя файла вместе с путем к файлу, например

FileExists('C:\Work\Data\Student.dat') – возвращает true, если файл существует и false, если файл с этим именем в указанной папке не существует.

6.6.3.2. Процедуры для работы с текстовыми файлами

Текстовые файлы являются файлами с последовательным доступом. Доступ к каждой строке возможен лишь последовательно, начиная с первой. Если нужно, например, прочитать строку n, то сначала нужно прочитать все n-1 строк.

Процедура Reset(f) открывает текстовый файл только для чтения.

Процедура Rewrite(f) создает новый файл (если файл с тем же именем уже существовал, то он уничтожается) и он открывается только для записи.

Процедура Append(f) открывает файл для записи и позволяет добавлять новые элементы в конец файла.

Процедура Read(f,v)- считывает в переменную v очередной элемент текстового файла, соответствующего текущему положению указателя. Если производится считывание сразу в несколько переменных, то они разделяются запятыми, например:

```
Read(f,v1, v2, v3);
```

При этом после считывания значения элемента файла в последнюю переменную списка (v3) указатель остается на месте. Допускается использование переменных разных типов.

Процедура Readln(f,v)- считывает в переменную v очередной элемент текстового файла, соответствующего текущему положению указателя. Если производится считывание сразу в несколько переменных, то они разделяются

запятыми, например:

```
Readln(f,v1, v2, v3);
```

При этом после считывания значения элемента файла в последнюю переменную списка (v3), оставшиеся элементы вплоть до символа eoln пропускаются, а указатель сдвигается на следующую строку. В этом заключается отличие Readln от Read.

Поэтому, если в текстовом файле имеется несколько строк, процедурой Read можно прочитать только одну строку, а процедурой Readln несколько строк, но для этого надо процедуру Readln вызывать столько раз, сколько это необходимо.

Вызов процедуры в виде Readln(f) позволяет пропустить строку в файле, не считывая ее содержимого.

При использовании процедур Read и Readln происходит автоматическое преобразование прочитанных символов в типы тех переменных, которые указаны в списке ввода. При чтении переменных типа char выполняется чтение одного символа и присваивание считанного значения переменной. При чтении из файла значения строковой переменной, длина которой явно задана в ее объявлении, считывается столько символов, сколько указано в объявлении, но не больше, чем в текущей строке.

При чтении из файла значения строковой переменной, длина которой явно не задана в объявлении переменной, значением переменной становится оставшаяся после последнего чтения часть текущей строки.

Если одной инструкцией Readln осуществляется ввод нескольких, например, двух переменных, то первая переменная будет содержать столько символов, сколько указано в ее объявлении или, если длина не указана, всю строку файла. Вторая переменная будет содержать оставшиеся символы текущей строки или, если таких символов нет, не будет содержать ни одного символа (длина строки равна нулю).

При чтении числовых значений алгоритм работы процедур ввода меняется. Начиная с текущего положения указателя выделяются значащие символы до первого пробела, знака табуляции или признака конца строки. Полученная последовательность символов считается символьным представлением числа и делается попытка преобразования его во внутреннее представление числа (целого или вещественного, в зависимости от типа вводимой переменной). Если преобразование прошло успешно, соответствующей переменной присваивается значение этого числа. В противном случае фиксируется ошибка, т.е. это означает, что в символьном представлении числа встречается недопустимый символ. Такие ошибки можно обработать программно. Способы фиксации и обработки ошибок ввода/вывода мы рассмотрим в разделе 6.6.3.5.

Процедура `Write(f,v)`- записывает значение переменной `v` в файл. Здесь `f` по-прежнему файловая переменная. Если производится запись значений сразу нескольких переменных, то они разделяются запятыми, например:

```
Write(f,v1, v2, v3);
```

Процедура `Writeln(f,v)`- записывает значение переменной `v` в файл и добавляет символ конца строки `eoln`. Если производится запись значений сразу нескольких переменных, то они разделяются запятыми, например:

```
Writeln(f,v1, v2, v3);
```

В текстовых файлах при записи происходит автоматическое преобразование типов переменных в списке вывода в их символьное представление.

Процедура `Writeln(f)` позволяет записать в файл пустую строку.

Чтобы проверить не является ли текущий элемент символом конца строки `eoln` можно использовать функции `EoLn(f)` и `SeekEoLn(f)`. Функция `EoLn(f)` проверяет только текущую позицию указателя на признак конца строки, а `SeekEoLn(f)` сначала пропускает пробелы и символы табуляции и лишь затем проверяет на признак конца строки.

Точно также функция `SeekEof(f)` сначала пропускает пробелы и симво-

лы табуляции и лишь затем проверяет файл на признак конца.

Рассмотрим снова программу из раздела 6.5.2.1. С этой программой, было неудобно работать, т.к. постоянная информация (фамилии, количество проданных компьютеров) не сохранялась и при каждом запуске программы приходилось заново их вводить. Проблема решается, если постоянную информацию сохранить на диске.

Напишем программу создания файлов на диске и добавления в них данных:

```
program create_files;
{$mode objfpc}
uses
  {В модуле SysUtils содержится функция FileExists}
  CRT, FileUtil, SysUtils;
var
  name: TextFile; // Файловая переменная
  kol: TextFile;  // Файловая переменная
  comp, k, n:integer;
  fam: string[18];
begin
  AssignFile(name,'Name.txt');
  AssignFile(kol,'Kol.txt');
  {Проверка существования файлов. Если файлы
   существуют, то они открываются для дозаписи.
   Если нет, то создаются новые пустые файлы}
  if not FileExists('Name.txt') then
    Rewrite(name)
  else
    Append(name);
```

```
  if not FileExists('Kol.txt') then
    Rewrite(kol)
  else
    Append(kol);
  writeln(UTF8ToConsole('Введите количество менеджеров ));
  readln(n);
  for k:=1 to n do
  begin
    writeln(UTF8ToConsole('Введите фамилию'));
    readln(fam);
    Writeln(name, fam); // запись в файл
    writeln(UTF8ToConsole('Введите количество компьютеров'));
    readln(comp);
    Writeln(kol, comp); // запись в файл
  end;
  writeln(UTF8ToConsole('Информация на диск записана'));
  CloseFile(name);
  CloseFile(kol);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Напишем программу обработки файлов:

```
program manager;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  name: TextFile;
```

```
  kol: TextFile;
  sum, cost, prem, k: integer;
  comp, sumc, sumv, sump, sum1: integer;
  fam: string[18];
begin
  {При необходимости укажите полный путь
  к файлам или скопируйте эти файлы в папку
  с данным проектом или сохраните сам проект
  в папке, где создан проект create_files.lpr}
  if (not FileExists('Name.txt')) or
     (not FileExists('Kol.txt')) then
  begin
    writeln(UTF8ToConsole('Файлы не существуют'));
    writeln(UTF8ToConsole('Сначала создайте их'));
    writeln(UTF8ToConsole('Нажмите любую клавишу'));
    readkey;
    exit;
  end;
  begin
    AssignFile(name,'Name.txt');
    AssignFile(kol,'Kol.txt');
    // файлы открываются для чтения
    Reset(name);
    Reset(kol);
  end;
  ClrScr;
  GoToXY(6, 1);
  write(UTF8ToConsole('Сведения о реализации компьютеров'));
  GoToXY(14, 2);
```

```
write(UTF8ToConsole('за январь 2010 г.'));
GoToXY(1, 3);
write('---------------------------------------------');
GoToXY(1, 4);
write(UTF8ToConsole(' Фамилия  Количество Выручка Премия'));
GoToXY(1, 5);
write('---------------------------------------------');
cost:= 1000; //  стоимость компьютера
prem:= 25;   //  размер премии
sumc:= 0;    //  Общее количество компьютеров
sumv:= 0;    //  Общая сумма выручки
sump:= 0;    //  Общая сумма премии
k:= 0;       //  Отслеживает текущую строку на экране
while not Eof(name) do
begin
  Readln(name, fam);
  Readln(kol, comp);
  sum:=comp*cost;  //  сумма выручки для одного менеджера
  sum1:=comp*prem; //  сумма премиальных для одного менеджера
  sumc:=sumc+comp; //  всего продано компьютеров
  sumv:=sumv+sum;  //  выручка от продажи всех компьютеров
  sump:=sump+sum1; //  общая сумма премиальных
  writeln;
  writeln(fam);
  GoToXY(24, k + 6);
  write(comp);
  GoToXY(32, k + 6);
  write(sum);
```

```
    GoToXY(40, k + 6);
    write(sum1);
    k:= k + 1;
  end;
  GoToXY(1, k + 6);
  write('-------------------------------------------');
  GoToXY(17, k + 7);
  write(UTF8ToConsole('Итого:'));
  GoToXY(24, k + 7);
  write(sumc);
  GoToXY(32, k + 7);
  write( sumv);
  GoToXY(40, k + 7);
  write(sump);
  GoToXY(1, k + 9);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  CloseFile(name);
  CloseFile(kol);
  readkey;
end.
```

Почему мы разделили эти программы на две самостоятельные? Чтобы отделить процесс ввода от непосредственно обработки. Так часто делают на практике. Ведь вводом данных может заниматься, например, оператор. Причем данные должны вводиться, по идее, каждый день. А программа выдачи сводных данных запускается один раз в месяц.

6.6.3.3. Процедуры для работы с типизированными файлами

Процедура `Read(f, v)`- считывает в переменную `v` очередной элемент файла. Тип переменной `v` должен соответствовать файловой переменной `f`.

Процедура `Write(f, v)-` записывает переменную `v` в файл. `f-` файловая переменная, `v -` переменная того, же типа, что и элемент файла `f.`

Для типизированных файлов применение процедур `Writeln` и `Readln` запрещены.

Типизированные файлы в отличие от текстовых являются файлами прямого доступа, т.е. позволяют обратиться "сразу" к нужному элементу по его номеру. Таким образом, в типизированных файлах все его элементы имеют свой номер. Нумерация элементов файла начинается с 0.

Для перемещения по типизированному файлу применяется процедура `Seek`. Формат вызова процедуры:

```
Seek(f, n);
```

Где `f` – файловая переменная, `n` – номер элемента в файле. Процедура `Seek` просто устанавливает указатель на элемент с номером `n`.

Функция `FilePos(f)` возвращает текущую позицию указателя, т.е. номер того элемента на который установлен указатель.

Функция `FileSize(f)` возвращает число элементов в файле.

Напишем программу предыдущего раздела с использованием типизированных файлов. В том примере мы создали два файла, каждый элемент которых состоял только из одного поля.

Файл Name

Иванов
Петров
……
Сидоров
Яковлев

Файл Kol

2
10
15
0

Как мы уже отмечали, типизированные файлы позволяют организовать

хранение данных более сложной структуры, т.е. состоящие из нескольких полей, имеющих разные типы. Создадим один файл с именем "File_manager.dat", имеющий следующую структуру записи:

name	comp
Иванов	2
Петров	10
……	
Сидоров	15
Яковлев	0

Для этого сначала создадим тип данных запись следующей структуры:

```
type
    manager = record
    name:string[18];
    comp: integer;
 end;
```

Программа создания и ввода данных в типизированный файл будет выглядеть следующим образом:

```
program create_files;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
type
  manager = record
  name: string[18];
  comp: integer;
end;
```

```
var
  company: manager;
  fmanager: File of manager; // Файловая переменная
  k: integer;
  n: integer;
begin
  AssignFile(fmanager,'File_manager.dat');
  {Проверка существования файлов. Если файлы
  существуют, то они открываются для дозаписи.
  Если нет, то создаются новые пустые файлы}
  if not FileExists('File_manager.dat') then
    Rewrite(fmanager)
  else
  Reset(fmanager);
  Seek(fmanager, System.FileSize(fmanager));
  while not Eof(fmanager) do
     Read(fmanager, company);
  writeln(UTF8ToConsole('Введите количество менеджеров'));
  readln(n);
  with company do
  begin
    for k:= 1 to n do
    begin
      writeln(UTF8ToConsole('Введите фамилию'));
      readln(name);
      writeln(UTF8ToConsole('Введите количество компьютеров'));
      readln(comp);
      Write(fmanager, company); // запись в файл
    end;
```

```
  end;
  writeln(UTF8ToConsole('Информация на диск записана'));
  CloseFile(fmanager);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Обратите внимание, процедура

```
Seek(fmanager, System.FileSize(fmanager));
```

переводит указатель на конец файла, что позволяет производить дозапись элементов в конец файла. Также обратите внимание на вызов функции `FileSize()`. Имейте в виду, что существуют две версии функции `FileSize()`:

- `function FileSize(const Filename: string): int64;`
- `function FileSize(var f: File): int64;`

Первая функция описана в модуле `FileUtil`, она возвращает полный объем файла в байтах. А вторая описана в модуле `System`, она возвращает количество записей в файле. Нам нужна именно вторая функция, поэтому мы перед именем файла указываем имя модуля `System`, а затем через точку имя функции.

Напишем программу выдачи сводных данных. Поскольку мы собираемся в дальнейшем использовать созданный типизированный файл менеджеров и программу вывода сводной ведомости на экран, оформим эту программу в виде модуля. Тогда программа будем выглядеть таким образом:

```
unit OutScr;
interface
uses
  CRT, Classes, FileUtil, SysUtils, LCLProc;
procedure output_to_screen(var name_file: string);
implementation
procedure output_to_screen(var name_file: string);
type
    manager= record
    name: string[18];
    comp: integer;
end;
var
  company: manager;
  sum, cost, prem, k: integer;
  sumc, sumv, sump, sum1: integer;
  fmanager: File of manager;
begin
  Assign(fmanager, name_file);
  Reset(fmanager);
  ClrScr;
  GoToXY(6, 1);
  write(UTF8ToConsole('Сведения о реализации компьютеров'));
  GoToXY(14, 2);
  write(UTF8ToConsole('за январь 2010 г.'));
  GoToXY(1, 3);
  write('---------------------------------------------');
  GoToXY(1, 4);
  write(UTF8ToConsole(' Фамилия  Количество Выручка Премия'));
```

```
GoToXY(1, 5);
write('---------------------------------------------');
cost:= 1000; // стоимость компьютера
prem:= 25;   // размер премии
sumc:= 0;    // Общее количество компьютеров
sumv:= 0;    // Общая сумма выручки
sump:= 0;    // Общая сумма премии
k:= 0;       // Отслеживает текущую строку на экране
with company do
begin
  while not Eof(fmanager) do
  begin
    Read(fmanager, company);
    sum:= comp * cost;  // сумма выручки для одного менеджера
    sum1:= comp * prem; // сумма премиальных одного менеджера
    sumc:= sumc + comp;
    sumv:= sumv + sum;
    sump:= sump + sum1;
    writeln;
    writeln(name);
    GoToXY(24, k + 6);
    write(comp);
    GoToXY(32, k + 6);
    write(sum);
    GoToXY(40, k + 6);
    write(sum1);
    k:= k + 1;
  end;
end;
```

```
  GoToXY(1, k + 6);
  write('-------------------------------------------');
  GoToXY(17, k + 7);
  write(UTF8ToConsole('Итого:'));
  GoToXY(24, k + 7);
  write(sumc);
  GoToXY(32, k + 7);
  write( sumv);
  GoToXY(40, k + 7);
  write(sump);
  GoToXY(1, k + 9);
  CloseFile(fmanager);
end;
end.
```

Скопируйте полученные после компиляции объектные модули (с расширением `OutScr.o` и `OutScr.ppu`) в папку, где находятся ваши часто используемые модули, например в папку `my-units` (см. раздел 6.5.1).

Программа, использующая этот модуль, будет такой:

```
program manager_computer;
uses
  CRT, SysUtils, FileUtil, OutScr;
var
  name_file: string;
begin
   {При необходимости укажите полный путь
 к файлам или скопируйте эти файлы в папку
 с данным проектом или сохраните сам проект
 в папке, где создан проект create_files.lpr}
```

```
  name_file:='File_manager.dat';
  if not FileExists(name_file) then
  begin
    writeln(UTF8ToConsole('Файлы не существуют'));
    writeln(UTF8ToConsole('Сначала создайте их'));
    writeln(UTF8ToConsole('Нажмите любую клавишу'));
    readkey;
    exit;
  end;
  output_to_screen(name_file);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Проанализировав нашу программу, мы вынуждены констатировать, что при создании файла отсутствует контроль вводимых данных. Исправим этот недостаток. Воспользуемся следующим способом. Будем вводить количество менеджеров и количество компьютеров сначала в символьные переменные. Затем воспользуемся функцией `Val()` для преобразования строки в число. Как вы знаете, эта функция имеет специальный параметр и если при преобразовании произошла ошибка, функция передает через этот параметр код ошибки. Если преобразование прошло успешно, параметр равен нулю. Контролировать ввод будем в цикле для того, чтобы в случае ошибки при вводе дать возможность пользователю повторить ввод.

```
program create_files;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
```

```
type
  manager=record
  name:string[18];
  comp: integer;
 end;
var
  company:manager;
  fmanager: File of manager; // Файловая переменная
  k: integer;
  n: integer;
  code: integer;
  str_n, str_comp: string;
begin
  AssignFile(fmanager,'File_manager.dat');
  {Проверка существования файлов. Если файлы
  существуют, то они открываются для дозаписи.
  Если нет, то создаются новые пустые файлы}
  if not FileExists('File_manager.dat') then
    Rewrite(fmanager)
  else
  Reset(fmanager);
  Seek(fmanager, System.FileSize(fmanager));
  while not Eof(fmanager) do
     Read(fmanager, company);
  while true do
  begin
    writeln(UTF8ToConsole('Введите количество менеджеров'));
    readln(str_n);
    Val(str_n, n, code);
```

```
    if code = 0 then
      break
    else
      writeln(UTF8ToConsole('Ошибка! Повторите ввод'));
  end;
  with company do
  begin
    for k:= 1 to n do
    begin
      writeln(UTF8ToConsole('Введите фамилию'));
      readln(name);
      while true do
      begin
      writeln(UTF8ToConsole('Введите количество компьютеров'));
      readln(str_comp);
      Val(str_comp, comp, code);
      if code = 0 then
        break
      else
        writeln(UTF8ToConsole('Ошибка! Повторите ввод'));
      end;
      Write(fmanager, company); // запись в файл
    end;
  end;
  writeln(UTF8ToConsole('Информация на диск записана'));
  CloseFile(fmanager);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

6.6.3.4. Процедуры для работы с нетипизированными файлами

В файлах любого типа чтение и запись данных производится порциями определенной длины, называемых блоками или записями. В текстовых и типизированных файлах размер записей определяется автоматически. За этим следит операционная система, точнее, ее часть, называемая файловой системой. Для нетипизированных файлов программист сам должен следить за размером вводимых и выводимых записей. Обратите внимание, что в данном случае понятие "запись" и тип данных в Паскале "запись" – это совершенно разные вещи. При операциях с файлами под записью понимается размер физического блока данных, которые записываются или считываются с внешнего устройства.

Нетипизированные файлы также являются файлами прямого доступа. К ним применимы все те процедуры, которые используются для типизированных файлов, за исключением процедур ввода/вывода.

Ввод/вывод в нетипизированных файлах осуществляется с помощью процедур, имеющих следующий формат:

```
BlockRead(f, buf, count[, fact_count]);
BlockWrite(f, buf, count[, fact_count]);
```

Процедура `BlockRead` осуществляет чтение данных из дискового устройства, а процедура `BlockWrite` запись данных.

Здесь `f` – файловая переменная, `buf` – переменная, содержимое которой либо записывается в файл, либо данные из файла считываются в эту переменную, `count` – переменная целого типа, содержит количество записей, которые необходимо прочитать или записать, `fact_count` – необязательный параметр, переменная целого типа, в нем содержится фактическое количество прочитанных или записанных блоков данных.

Создать или открыть файл можно процедурами `Rewrite` или `Reset`, при этом в параметры процедур можно добавить необязательный параметр – размер записей, например:

```
Rewrite(f, 256);
Reset(f1, 1);
```

Этими процедурами создается файл `f` с длиной записи 256 байтов и открывается файл `f1` размер записи которой составляет 1 байт.

Если второй параметр не указан, то размер записи по умолчанию принимается равным 128 байтам. Для обеспечения максимальной скорости обмена данными можно указывать размер записи кратной величине кластера дискового накопителя. Однако на практике пользуются размером записи всего в один байт, что позволяет обмениваться данными блоками любой длины.

Часто требуется определить размер памяти, занимаемый тем или иным объектом. Для этого используется функция `SizeOf(x)`, которая возвращает количество байт, занимаемых аргументом x в памяти.

При написании нашей любимой программы – решения системы линейных алгебраических уравнений методом Гаусса введенные коэффициенты не сохранялись. Перепишем программу так, чтобы введенные коэффициенты расширенной матрицы сохранялись на диске. Программа записывает в нетипизированный файл количество уравнений системы и коэффициенты расширенной матрицы системы.

```
program Input_coeff;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  matrix: File;
  temp: real;
  i, j, n, count: integer;
begin
  AssignFile(matrix, 'Coeff.dat');
  {Создается нетипизированный файл. Длина блока 1 байт}
  Rewrite(matrix, 1);
  writeln(UTF8ToConsole('Введите количество неизвестных'));
  readln(n);
```

```
  {На диск будет записано count блоков длиной по 1 байту.
  Поскольку целый тип занимает 4 байта, будет записано 4 блока}
  count:= SizeOf(integer);
  BlockWrite(matrix, n, count);
  {Поскольку вещественный тип real занимает 8 байт,
  будет записано 8 блоков по 1 байту}
  count:= SizeOf(real);
  {Ввод коэффициентов расширенной матрицы}
  for i:=1 to n do
  begin
    for j:=1 to n do
    begin
      writeln(UTF8ToConsole('Введите a'), i, j);
      readln(temp);
      BlockWrite(matrix, temp, count);
    end;
    writeln(UTF8ToConsole('Введите b'), i);
    readln(temp);
    {Можно сразу записать функцию SizeOf в качестве
    3-го параметра процедуры записи/чтения}
    BlockWrite(matrix, temp, SizeOf(real));
  end;
  writeln(UTF8ToConsole('Информация на диск записана'));
  CloseFile(matrix);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Программа решения системы линейных алгебраических уравнений методом Гаусса, коэффициенты расширенной матрицы вводятся из нетипизированного файла:

```
program Gauss_File;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
var
  matrix: File;
  a: array of array of real; {матрица коэффициентов системы,
  двумерный динамический массив}
```

```
  vector: array of real; {преобразованный одномерный
  динамический массив}
  b: array of real;
  x: array of real;
  temp: real;
  i, j, k, n: integer;
{Процедура остается без изменений}
procedure gauss(var vector: array of real;
                var b: array of real;
                var x: array of real;
                var n: integer);
var
  a: array of array of real; {матрица коэффициентов системы,
  двумерный динамический массив}
  i, j, k, p, r: integer;
  m, s, t: real;
begin
  SetLength(a, n, n); // установка фактического размера массива

  {Преобразование одномерного массива в двумерный}
  k:=1;
  for i:=0 to n-1 do
    for j:=0 to n-1 do
    begin
      a[i,j]:= vector[k];
      k:=k+1;
    end;
  for k:=0 to n-2 do
  begin
    for i:=k+1 to n-1 do
    begin
      if (a[k,k]=0) then
      begin
        {перестановка уравнений}
        p:=k; // в алгоритме используется буква l, но она похожа на 1
        // Поэтому используем идентификатор p
        for r:=i to n-1 do
        begin
          if abs(a[r,k]) > abs(a[p,k]) then p:=r;
        end;
        if p<>k then
        begin
          for j:= k to n-1 do
```

```
        begin
          t:=a[k,j];
          a[k,j]:=a[p,j];
          a[p,j]:=t;
        end;
        t:=b[k];
        b[k]:=b[p];
        b[p]:=t;
      end;
    end; // конец блока перестановки уравнений
    m:=a[i,k]/a[k,k];
    a[i,k]:=0;
    for j:=k+1 to n-1 do
    begin
      a[i,j]:=a[i,j]-m*a[k,j];
    end;
    b[i]:= b[i]-m*b[k];
  end;
end;
{Проверка существования решения}
if a[n-1,n-1] <> 0  then
begin
  x[n-1]:=b[n-1]/a[n-1,n-1];
  for i:=n-2 downto 0 do
  begin
    s:=0;
    for j:=i+1 to n-1 do
    begin
      s:=s-a[i,j]*x[j];
    end;
    x[i]:=(b[i] + s)/a[i,i];
  end;
  writeln('');
  writeln(UTF8ToConsole('Решение:'));
  writeln('');
  for i:=0 to n-1 do
    writeln('x', i+1, '= ', x[i]:0:4);
end
else
if b[n-1] = 0 then
  writeln(UTF8ToConsole('Система не имеет решения.'))
else
  writeln(UTF8ToConsole('Система уравнений'+
```

```
    ' имеет бесконечное множество решений.'));
  writeln('');
  {освобождение памяти,
 распределенной для динамического массива}
  a:=nil;
end;
{Начало основной программы}
begin
  AssignFile(matrix, 'Coeff.dat');
  Reset(matrix, 1);
  {Чтение количества уравнений системы из файла}
  BlockRead(matrix, n, SizeOf(integer));
  {Установка реальных размеров динамических массивов}
  SetLength(a, n, n);
  SetLength(vector, n*n);
  SetLength(b, n);
  SetLength(x, n);
  {Ввод коэффициентов расширенной матрицы}
  for i:=1 to n do
  begin
    for j:=1 to n do
    begin
      BlockRead(matrix, temp, SizeOf(real));
      a[i-1, j-1]:= temp;
    end;
    BlockRead(matrix, temp, SizeOf (real));
    b[i-1]:= temp;
  end;
  {Преобразование двумерного массива в одномерный}
  k:=1;
  for i:=0 to n-1 do
    for j:=0 to n-1 do
    begin
      vector[k]:=a[i,j];
     k:=k+1;
    end;
  CloseFile(matrix);
  {Вызов процедуры решения системы линейных
  алгебраических уравнений методом Гаусса}
  gauss(vector, b, x, n);
  {освобождение памяти, распределенной
  для динамических массивов}
  a:=nil;
```

```
vector:=nil;
x:=nil;
b:=nil;
writeln(UTF8ToConsole('Нажмите любую клавишу'));
readkey; end.
```

6.6.3.5. Организация контроля ввода/вывода при работе файлами

При работе с файлами часто возникают непредвиденные ситуации, которые приводят к ошибкам ввода/вывода. Такие ситуации возникают, например, если указанный файл не существует на диске или диск не готов к работе или файл по каким-либо причинам был запорчен и прочитать данные невозможно и т.п. В таких случаях, если не предусмотреть соответствующих действий, программа аварийно завершается. Паскаль предоставляет возможность ввести в программу контроль операций ввода/вывода. Это дает возможность даже если не исправить сразу возникшую ошибку, то хотя бы локализовать ее, вывести какое-то сообщение или предупреждение и продолжить работу с программой.

Для организации контроля за операциями ввода/вывода применяется функция `IOResult`. Перед вызовом этой функции необходимо отключить автоматический контроль операций ввода/вывода директивой компилятора `{$I-}`. Этой директивой программа как бы сообщает компилятору и операционной системе, что контроль последующих операций ввода/вывода она берет на себя. Только после этого функция `IOResult` становится доступной. После завершения очередной операции ввода/вывода функция возвращает 0, если операция прошла успешно. В противном случае функция возвращает код ошибки. Необходимо анализировать значение функции `IOResult` сразу после ввода/вывода. В противном случае функция сбрасывает свое значение в 0. После завершения опасного участка программы, автоматический контроль ввода/вывода нужно восстановить директивой компилятора `{$I+}`.

Рассмотрим пример. Пусть имеется текстовый файл `Data.txt`. В программе моделируется ситуация, когда пользователь записывает в файл последо-

вательности целых чисел в их символьном представлении. Затем программа читает эти числа в переменную целого типа. В первый раз пользователь вводит строку `'1234'`, во второй раз пользователь случайно ввел вместо цифры недопустимый для числа символ, например вводит строку `'12#4'`, т.е. нажал на клавишу с цифрой 3 в верхнем регистре.

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  F: TextFile;
  code: word;
  number:integer;
  s:string;
begin
  AssignFile(F, 'Data.txt');
  Rewrite(F);
  s:='1234';
  Writeln(F, s);
  s:='12#4';
  Writeln(F, s);
  Reset(F);
  while not Eof(F) do
  begin
    {$I-}
    Readln(F, number);
    {$I+}
    code:= IOResult;
    if code<>0 then
```

```
    begin
      writeln(UTF8ToConsole('Ошибка преобразования типов,  '));
      writeln(UTF8ToConsole('код ошибки '), code);
      break
    end;
    writeln('number= ', number);
  end;
  CloseFile(F);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Как видим, первая строка файла прочитана правильно. Выводится значение целого числа `number=1234`. При чтении второй строки файла происходит ошибка преобразования строки символов в целое число, о чем программа и сообщает пользователю.

Для проверки существования файла удобнее пользоваться функцией

`FileExists(fname)`, где `fname` имя файла – строка символов. В следующем фрагменте программе проверяется существование файла на диске. Если файл существует, он открывается для чтения, в противном случае создается новый пустой файл с тем же именем:

```
    if not FileExists('My_file.txt') then
        Rewrite(kol)
      else
        Reset(kol);
```

6.6.3.6. Создание простой базы данных с типизированными файлами. Программа "Студент".

Рассмотрим пример разработки достаточно большой и сложной программы. При ее написании мы будем использовать все полученные нами знания по языку Паскаль, изложенные в предыдущих главах. В частности, методы работы с типизированными файлами, в которой используются записи сложной структуры, процедуры и функции, контроль операций ввода/вывода. Освоим технику создания и работы с меню. Фактически создается небольшая и простенькая база данных. Чтобы не загромождать текст программы многочисленными проверками в программу введены процедуры `Reset_file`, `Read_File` и `Write_File` внутри которых и производится контроль правильности операций ввода/вывода. Структура записи, используемая в программе имеет вид:

Фамилия	Группа	Предмет	Оценка

```
program Database_Student;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils, LConvEncoding, LCLType;
type
  student = record   {тип запись}
  fio: string[24];   // фамилия
  predmet: string[32]; // предмет
  gruppa: string[24]; // группа
  ocenka: integer;
end;

  fstud = File of student;

var
  fam: string[24]; // фамилия
  sub: string[32]; // предмет
```

```
  gr: string[24]; // группа
  answ: TUTF8Char; // символ для приема ответа пользователя
  f, v: fstud; {f,v - файловые переменные,
  где f - имя основного файла;
  v - имя вспомогательного файла}
  new_student: student;
  choice,choose, ocen: integer; {переменные,
  предназначенные для выбора режима}
  fname: string; {строковая переменная,
  имя файла на диске без расширения}
  full_fname: string; {строковая переменная,
  полное имя файла на диске с расширением}
  code_error: integer; // код ошибки ввода/вывода

// опережающее объявление функций и процедур
procedure Reset_File( var f: fstud); forward;
procedure Read_File(var f: fstud;
                    var st: student); forward;
procedure Write_File(var f: fstud;
                     var st: student); forward;
procedure check_file; forward;

{ ================ Ввод данных ================ }
procedure input_data;
begin
  with new_student do
  begin
    writeln(UTF8ToConsole(' Фамилия '));
    readln(fio);
    writeln(UTF8ToConsole(' Группа '));
    readln(gruppa);
    writeln(UTF8ToConsole(' Предмет '));
    readln(predmet);
    writeln(UTF8ToConsole(' Оценка '));
    repeat
      {$I-}
      readln(ocenka);
      {$I+}
      if (IOResult <> 0) or ((ocenka > 5)
       or (ocenka < 1)) then
      begin
        writeln(UTF8ToConsole('Введите целое число от 1 до 5'));
```

```
      end;
    until (ocenka >= 1) and (ocenka <= 5);
  end;
end;

{ ================ Создание файла ================ }
procedure create_data;
begin
  check_file;
  Rewrite(f);
  writeln(UTF8ToConsole(' Введите данные '));
  repeat
  input_data;
  Write(f, new_student);
  writeln(UTF8ToConsole(' Продолжить?, ответ - д/н (y/n) '));
  readln(answ);
  {$IFDEF WINDOWS}
     answ:= CP866ToUTF8(answ);
  {$ENDIF}
  until (answ='N') or (answ='n') or (answ='н') or (answ= 'Н');
end;

procedure check_file;
var
answ: TUTF8char;
begin
  if FileExists(full_fname) then
  begin
    Assign(f, full_fname);
    Reset(f);
    writeln(UTF8ToConsole('Текущий файл будет уничтожен!!'));
    writeln(UTF8ToConsole('Чтобы стереть существующий  '));
    writeln(UTF8ToConsole('файл, нажмите клавишу Esc,'));
    writeln(UTF8ToConsole('иначе нажмите любую клавишу.'));
    repeat
      answ:= readkey;
        if answ= #27 then
        begin
          writeln(UTF8ToConsole('Вы уверены? Нажмите '));
          writeln(UTF8ToConsole('еще раз клавишу Esc'));
          writeln(UTF8ToConsole('Для отмены нажмите '));
          writeln(UTF8ToConsole('любую клавишу.'));
          answ:= readkey;
```

```
          if answ = #27 then
            break;
        end;
        writeln(UTF8ToConsole('Введите другое имя файла'));
        CloseFile(f);
        readln(fname);
        Assign(f, fname +' .dat');
        break;
    until answ = #27;
  end;
end;

{ ================ Вывод содержимого файла ================ }
procedure out_to_screen;
var j: integer;
begin
  Reset_File(f);
  ClrScr;
  GoToXY(1, 5);
  j:= 0;
  writeln(UTF8ToConsole('*фамилия * группа * предмет*оценка *'));

writeln('================================================');
  while not Eof(f) do
  begin
    read(f,new_student);
    j:= j + 1;
    GoToXY(2, 6 + j);
    writeln(new_student.fio);
    GoToXY(15, 6 + j);
    writeln(new_student.gruppa);
    GoToXY(28, 6 + j);
    writeln(new_student.predmet);
    GoToXY(48, 6 + j);
    writeln(new_student.ocenka);
  end;

writeln('================================================');
  writeln(UTF8ToConsole(' Число студентов='),j:2);
  writeln(UTF8ToConsole('Нажмите любую клавишу '));
  readkey;
end;
```

```
{ ========= Поиск записей по заданным полям ============ }
procedure select_data;
begin
  repeat
  Reset_File(f);
  ClrScr;
  GoToXY(10, 10);
  write  (UTF8ToConsole('Выбор информации по:'));
  GoToXY(10, 11);
  write  (UTF8ToConsole(' группе      -1'));
  GoToXY(10, 12);
  write  (UTF8ToConsole(' предмету     -2'));
  GoToXY(10, 13);
  write  (UTF8ToConsole(' оценке      -3'));
  GoToXY(10, 14);
  writeln(UTF8ToConsole(' выход из режима  -4'));
  readln(choice);
  ClrScr;
  case choice of
    1: begin
         write(UTF8ToConsole(' Группа -'));
         readln(gr);
         writeln(UTF8ToConsole(' Сведения по группе '),
                 UTF8ToConsole(gr):5);
       end;
    2: begin
         write(UTF8ToConsole(' Предмет -'));
         readln(sub);
         writeln(UTF8ToConsole(' Сведения по предмету '),
                 UTF8ToConsole(sub):15);
       end;
    3: begin
         write(UTF8ToConsole(' Оценка ='));
         readln(ocen);
         writeln(UTF8ToConsole(' Сведения по оценке  '),
                 ocen:1);
       end;
  else
    exit;
  end; { end of case }
  while not eof(f) do
  begin
```

```
    Read_File(f,new_student);
    case choice of
      1: if new_student.gruppa=gr then
           writeln(new_student.fio:15,
           ' ',new_student.predmet:15,
           ' ',new_student.ocenka:1);
      2: if new_student.predmet=sub then
           writeln(new_student.fio:15,
           ' ',new_student.gruppa:15,
           ' ',new_student.ocenka:1);
      3: if new_student.ocenka=ocen then
           writeln(new_student.fio:15,
           ' ',new_student.predmet:15,
           ' ',new_student.gruppa:5);
    end; { end of case }
  end; { end of while }
  GoToXY(5, 24);
  writeln(UTF8ToConsole('Нажмите любую клавишу '));
  readkey;
  until choice = 4;
end;

{ ======== Восстановление файла под основное имя f ========== }
procedure restorefile;
begin
  CloseFile(f);
  CloseFile(v);
  Erase(f);
  Rewrite(f);
  Reset(v);
  while not Eof(v) do
  begin
    Read_File(v, new_student);
    Write_File(f, new_student);
  end;
  CloseFile(f); CloseFile(v); Erase(v);
    {удален вспомогательный файл v под внешним именем s.dat}
end;

{================ Добавление записей в файл ================}
procedure add_data;
begin
  Assign(v, 's.dat');
```

```
  Rewrite(v);
  { "s.dat" -  имя вспомогательного файла}
  Reset_File(f);

  {  копирование содержимого файла f в файл v  }
  while not Eof(f) do
  begin
    Read_File(f, new_student);
    Write_File(v, new_student);
  end;
  writeln(UTF8ToConsole(' Вводите информацию '));
  { записи добавляются в конце файла }
  repeat
    input_data;
    Write_File(v, new_student);
    writeln(UTF8ToConsole('   Продолжить?,  ответ - д/н (y/n) '));
    readln(answ);
    {$IFDEF WINDOWS}
      answ:= CP866ToUTF8(answ);
    {$ENDIF}
  until (answ='N') or (answ='n') or (answ='н') or (answ= 'Н');
  restorefile;
end;

{================ Удаление записей из файла ================}
procedure delete_data;
begin
  Assign(v, 's.dat'); Rewrite(v); Reset(f);
  ClrScr;
  GoToXY(10, 10);
  writeln(UTF8ToConsole('Удаление информации по:'));
  GoToXY(10, 11);
  writeln(UTF8ToConsole(' группе      -1'));
  GoToXY(10, 12);
  writeln(UTF8ToConsole(' фамилии      -2'));
  GoToXY(10, 13);
  writeln(UTF8ToConsole(' предмету     -3'));
  GoToXY(10, 14);
  writeln(UTF8ToConsole(' оценке      -4'));
  GoToXY(10, 15);
  writeln(UTF8ToConsole(' выход из режима  -5'));
  GoToXY(10, 16);
  write(UTF8ToConsole(' выбор режима ='));
```

```
  readln(choice);
  case choice of
    1: begin
         write(UTF8ToConsole(' Группа - '));
         readln(gr);
       end;
    2: begin
         write(UTF8ToConsole(' Фамилия - '));
         readln(fam);
       end;
    3: begin
         write(UTF8ToConsole(' Предмет - '));
         readln(sub);
       end;
    4: begin
         write(UTF8ToConsole(' Оценка - '));
         readln(ocen);
       end;
    5: exit; { выход в основную программу }
  end; { end of case }

{ ========= поиск записи для удаления =========== }
  while not Eof(f) do
  begin
    Read_File(f, new_student);
    case choice of
      1: if new_student.gruppa<>gr then
           Write_File(v,new_student);
      2: if new_student.fio<>fam then
           Write_File(v,new_student);
      3: if new_student.predmet<>sub then
           Write_File(v,new_student);
      4: if new_student.ocenka<>ocen then
           Write_File(v,new_student);
         else
         begin
           writeln(UTF8ToConsole(' Ошибка при вводе '));
           writeln(UTF8ToConsole('Нажмите любую клавишу '));
           readkey;
         end;
    end; { end of case }
  end; { end of while }
  restorefile;
```

```
end;
{======== процедура открытия файла с контролем операции ======== }
procedure Reset_File( var f:fstud);
begin
  {$I-}
  Reset(f);
  {$I+}
  code_error:= IOResult;
  if code_error <> 0 then
  begin
    writeln(UTF8ToConsole('Файл не существует, код ошибки '),
code_error);
    writeln(UTF8ToConsole('Нажмите любую клавишу'));
    readkey;
    Halt;
  end;
end;

{========== процедура чтения с контролем операции ============}
procedure Read_File(var f: fstud; var st: student);
begin
  {$I-}
  Read(f, st);
  {$I+}
  code_error:= IOResult;
  if code_error <> 0 then
  begin
    writeln(UTF8ToConsole('Ошибка чтения, код ошибки '),
code_error);
    writeln(UTF8ToConsole('Нажмите любую клавишу'));
    readkey;
    Halt;
  end;
end;
{========== процедура записи с контролем операции ============}
procedure Write_File(var f: fstud; var st: student);
begin
  {$I-}
  Write(f, st);
  {$I+}
  code_error:= IOResult;
  if code_error <> 0 then
  begin
```

```
    writeln(UTF8ToConsole('Ошибка записи в файл, код ошибки '),
code_error);
    writeln(UTF8ToConsole('Нажмите любую клавишу'));
    readkey;
    Halt;
  end;
end;
{ ============== корректировка записей в файле ============== }
procedure find_data;
var r: student;
begin
  Reset_File(f);  Assign(v,'s.dat'); Rewrite(v);
  ClrScr;
  GoToXY(10, 9);
  writeln(UTF8ToConsole('Укажите ключ (поле) для поиска'));
  GoToXY(10, 10);
  writeln(UTF8ToConsole('корректируемой записи - по:'));
  GoToXY(10, 11);
  writeln(UTF8ToConsole(' группе      -1'));
  GoToXY(10, 12);
  writeln(UTF8ToConsole(' фамилии      -2'));
  GoToXY(10, 13);
  writeln(UTF8ToConsole(' предмету      -3'));
  GoToXY(10, 14);
  writeln(UTF8ToConsole(' оценке      -4'));
  GoToXY(10, 15);
  writeln(UTF8ToConsole(' выход из режима  -5'));
  GoToXY(10, 16);
  write(UTF8ToConsole(' выбор режима ='));
  readln(choice); ClrScr;
  GoToXY(10, 9);
  writeln(UTF8ToConsole(' Замена информации  '));
  case choice of   { поиск записи }
    1: begin
         GoToXY(10, 10);
         write(UTF8ToConsole('группа='));
         readln(gr);
         input_data;
       end;
    2: begin
        GoToXY(10, 10);
        write(UTF8ToConsole('фамилия='));
```

```
        readln(fam);
        input_data;
      end;
    3: begin
        GoToXY(10, 10);
        write(UTF8ToConsole('предмет='));
        readln(sub);
        input_data;
      end;
    4: begin
        GoToXY(10, 10);
        write(UTF8ToConsole('оценка='));
        readln(ocen);
        input_data;
      end;
    5: exit; { выход в основную программу }
  end; { end of case }
  while not Eof(f) do
  begin
    Read_File(f, r);
    case choice of
      1: begin
           if gr=r.gruppa then
             Write_File(v,new_student)
           else
             Write_File(v,r)
         end;
      2: begin
           if fam=r.fio then
             Write_File(v,new_student)
           else
             Write_File(v,r)
         end;
      3: begin
           if sub=r.predmet then
             Write_File(v,new_student)
           else
             Write_File(v,r)
         end;
      4: begin
           if ocen=r.ocenka then
             Write_File(v,new_student)
           else
```

```
            Write_File(v,r)
        end;
    end; { end of case }
  end; { end of while }
  restorefile;
end;
   { ============== основная программа  ===================}
begin
  writeln(UTF8ToConsole('Введите имя файла:'));
  readln(fname);
  {$IFDEF WINDOWS}
    fname:=CP866ToUTF8(fname);
    fname:=UTF8ToAnsi(fname);
  {$ENDIF}
  full_fname:=fname + '.dat';
  Assign(f,full_fname);
  repeat
    ClrScr;
{ Формирование меню работы с основным файлом  f }
    GoToXY(10, 7);
    writeln(UTF8ToConsole('Выберите нужный режим работы :'));
    GoToXY(10, 8);
    writeln(UTF8ToConsole('Создание файла                  1'));
    GoToXY(10, 9);
    writeln(UTF8ToConsole('Вывод содержимого файла       2'));
    GoToXY(10, 10);
    writeln(UTF8ToConsole('Поиск по заданным полям       3'));
    GoToXY(10, 11);
    writeln(UTF8ToConsole('Добавление записей в файл     4'));
    GoToXY(10, 12);
    writeln(UTF8ToConsole('Удаление записей из файла     5'));
    GoToXY(10, 13);
    writeln(UTF8ToConsole('Корректировка записей в файле   6'));
    GoToXY(10, 14);
    writeln(UTF8ToConsole('Выход из программы              7'));
    readln(choose);
    case choose of
  { choose - значение для выбора режима работы с файлом  f }
      1: create_data;
      2: out_to_screen;
      3: select_data;
      4: add_data;
```

```
      5: delete_data;
      6: find_data;
    end; { end of case }
  until choose=7;
end.
```

В этой программе новым для нас является только опережающее объявление функций и процедур. В Паскале строго соблюдается правило – каждый объект перед использованием должен быть описан. Что касается функций и процедур, то здесь сделаны некоторые "поблажки". Пусть имеются две процедуры, одна из которых вызывает другую.

```
procedure A (param: integer;)
begin
  . . . . .
  B(i);
   . . . . .
end;
procedure B(param: integer);
begin
  . . . . .
end;
```

При таком расположении и таком описании процедур компилятор выдаст ошибку "Identifier not found "B"". Можно конечно переставить местами эти процедуры, но можно сделать так называемое `опережающее описание` процедуры таким вот образом:

```
procedure B(param: integer); forward;
procedure A (param: integer;)
begin
```

```
      . . . . .
      B(i);
       . . . . .
    end;
    procedure B(param: integer);
    begin
      . . . . .
    end;
```

Как видим, опережающее описание заключается в том, что объявляется лишь заголовок процедуры `B`, а тело процедуры заменяется директивой `forward`. Теперь в процедуре `A` можно использовать обращение к процедуре `B`, поскольку она уже описана, точнее, известны ее формальные параметры, и компилятор может правильным образом организовать ее вызов.

Рассмотрим другую ситуацию. Предположим, что процедуры `A` и `B` вызывают друг друга:

```
procedure A (param: integer;)
begin
  . . . . .
  B(i);
   . . . . .
end;
procedure B(param: integer);
begin
  . . . . .
  A(i);
   . . . . .
end;
```

Теперь не поможет и перестановка местами процедур, так как они "зациклены" друг на друга. Только использование опережающего объявления процедуры `B` позволяет разрешить эту проблему.

```
procedure B(param: integer); forward;
procedure A (param: integer;)
begin
  . . . . .
  B(i);
   . . . . .
end;
procedure B(param: integer);
begin
  . . . . .
  A(i);
   . . . . .
end;
```

Глава VII. Типовые алгоритмы обработки информации

К типовым алгоритмам я отношу алгоритмы сортировки, поиска и алгоритмы работы с динамическими структурами. Можно, конечно, рассмотреть и множество других алгоритмов, отнеся их к типовым. Но мы данной главе рассмотрим именно эти алгоритмы.

7.1. Алгоритмы сортировки

Сортировка - процесс упорядочения элементов какого-либо объекта по возрастанию или убыванию значений выбранного признака. По окончании сортировки элементы располагаются либо по возрастанию, либо по убыванию. Если среди элементов имеются одинаковые, то говорят сортировка по неубыванию или по невозрастанию. На сортировку информации тратится 25-50% машинного времени при работе со сложными программными комплексами (различные информационно-поисковые системы на базе СУБД, компиляторы и другие программные средства).

Сортировать можно данные, для которых определены операции сравнения. Чаще всего сортировке подлежат числа. Однако сортировать можно не только числа как таковые. Мы с вами уже знаем, что символы представляются в памяти в виде некоторых кодов. Таким образом, вполне можно сортировать символы, например, фамилии можно отсортировать по алфавиту. Сортировать можно не только объекты, содержащие одиночные элементы, но и объекты достаточно сложной структуры, например записи, содержащие несколько элементов. Файлы, как правило, содержат записи сложной структуры. Таким образом и в основном, сортировке подлежат файлы. В программе предыдущего раздела

структура записи файла состояла из нескольких полей, таких как "Фамилия", "Группа", "Предмет", "Оценка". Вполне можно осуществить сортировку этого файла, например по группе, а внутри группы отсортировать фамилии студентов по алфавиту. Поле, по которому осуществляется сортировка называется ключом. В нашем случае ключ это сначала поле "Группа", затем поле "Фамилия" внутри отсортированной части файла, соответствующей той или иной группе.

Эффективность методов сортировки оценивается по числу сравнений элементов между собой и числу перестановок элементов для упорядочения массива или файла. Существуют огромное количество алгоритмов сортировки. Если сортируемый файл полностью помещается в оперативной памяти, то сортировка называется внутренней. Сортировка файлов на диске называется внешней сортировкой. Кроме того, все алгоритмы сортировки можно условно разделить на "простые", но "медленные" и "сложные", но "быстрые". Взятие в кавычки здесь не случайно. В некоторых случаях "сложные", но "быстрые" алгоритмы оказываются медленнее "простых". Чаще всего так происходит для небольших файлов. Для подавляющего числа случаев "простые" алгоритмы вполне подходят по скорости работы и нет смысла "извращаться", пытаясь реализовать сложные и хитроумные алгоритмы. Необходимо помнить, что чем проще алгоритм (и это относится не только к алгоритмам сортировки), тем надежнее будет работать ваша программа. И только для очень больших файлов, когда именно на сортировке ваша программа "застревает", стоит поискать более сложные и быстрые алгоритмы.

Рассмотрим несколько алгоритмов сортировки. В примерах мы будем рассматривать сортировку массивов целых чисел.

7.1.1. Обменная сортировка (метод "пузырька")

Свое название алгоритм получил благодаря тому, что в процессе сортировки меньшие (большие) числа как бы всплывают вверх как пузырьки воздуха в

воде. Если идет сортировка по возрастанию, "всплывают" меньшие числа, если по убыванию, то большие. Рассмотрим пример.

Пусть имеется массив, состоящий из четырех чисел: 4, 3, 2, 1. Необходимо расположить элементы массива по возрастанию. Сравниваются, начиная с конца массива пары соседних чисел. Если они расположены в неправильном порядке – меняем их местами. В результате первого прохода наименьший элемент (число 1) оказывается "наверху", на месте нулевого элемента, т.е. самый "легкий" элемент "всплыл на поверхность", рис. 7.1.

4		4		4		1
3		3		1		4
2	→	1	→	3	→	3
1		2		2		2

Рис. 7.1. Состояние массива во время первого прохода.

Как видно из рисунка, на первом шаге текущего прохода меняются местами числа "1" и "2". На втором шаге меняются "1" и "3" и на четвертом шаге меняются местами "1" и "4".

Далее осуществляется второй проход, в котором просматриваются элементы массива, кроме первого, который уже "встал" на свое место. Состояние массива во время второго прохода, рис. 7.2:

1		1		1
4		4		2
3	→	2	→	4
2		3		3

Рис. 7.2. Состояние массива во время второго прохода.

На рисунке элемент, который уже не участвует в просмотре, выделен.

На следующем (последнем для данного массива) проходе поменяются местами числа "4" и "3". В итоге получаем отсортированный массив:

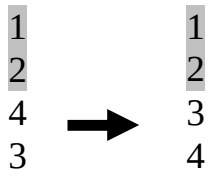


Рис. 7.3. После третьего прохода получаем отсортированный массив.

Таким образом, проходы делаются по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию. Блок-схема алгоритма сортировки методом "пузырька" приведена на рисунке 7.4.

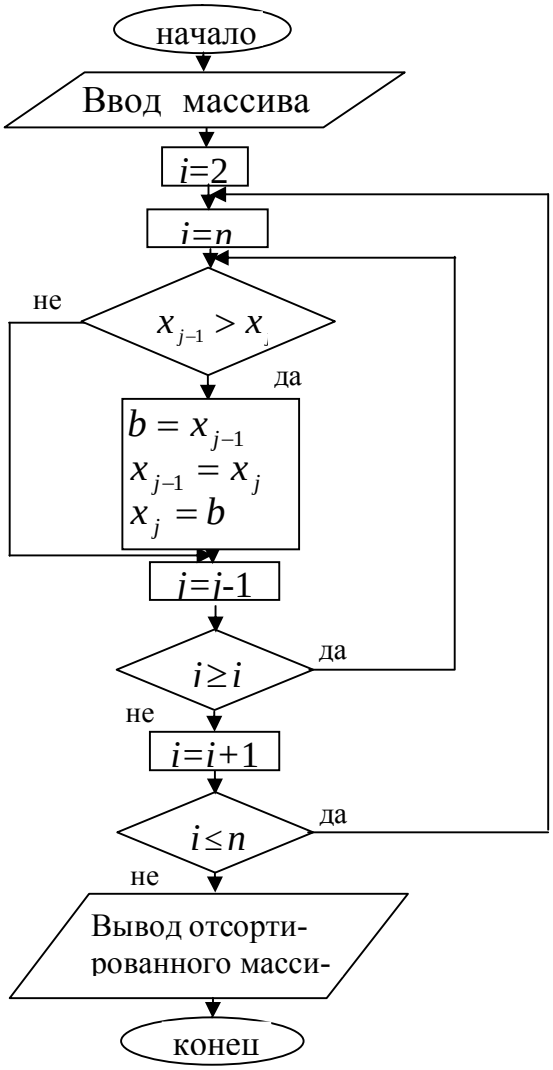


Рис. 7.4. Блок-схема алгоритма сортировки методом "пузырька".

Составим программу с использованием динамических массивов. При разборе программы не забудьте, что индексация элементов в таких массивах начинается с 0!

```
program bubble_sort;
uses
  CRT, FileUtil;
var
  i, n: integer;
  vector: array of integer;

{ ============= Сортировка методом "пузырька" ======== }
procedure bubble(var vector: array of integer);
var
  temp: integer;
  i, j, count: integer;
begin
  count:= high(vector);
  for i:= 1 to count   do
  for j:= count downto i do
  if vector[j - 1] > vector[j] then
  begin
    temp:= vector[j - 1];
    vector[j - 1]:= vector[j];
    vector[j]:= temp;
  end
end;
{ ================================================== }
begin
```

```
  writeln(UTF8ToConsole('Введите количество элементов массива'));
  readln(n);
  SetLength(vector, n );
  writeln(UTF8ToConsole('Введите '), n);
  writeln(UTF8ToConsole('значений элементов массива'));
  for i:= 0 to n - 1  do read(vector[i]);
  bubble(vector);
  writeln;
  writeln(UTF8ToConsole('Отсортированный массив'));
  for i:= 0 to n - 1  do write(vector[i], ' ');
  writeln;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Алгоритм пузырьковой сортировки является одним из самых медленных. В реализации алгоритма имеются два цикла. При первом выполнении внутреннего цикла будет выполнено n-1 сравнений, при втором n-2 сравнений и т.д. Всего будет выполнено n-1 таких циклов. Таким образом, всего будет выполнено

$$(n-1) + (n-2) + \ldots + 1$$

сравнений. Выражение можно упростить:

$$n(n-1)/2 \quad \text{или} \quad (n^2 - n)/2$$

Таким образом, пузырьковая сортировка требует $O(n^2)$ сравнений. Количество перестановок для самого худшего случая, когда элементы массива отсортированы в обратном порядке равно количеству сравнений, т.е. $O(n^2)$.

Имеется возможность небольшого улучшения алгоритма. Если во внутреннем цикле не происходит ни одной перестановки, то это означает, что массив уже отсортирован и дальнейшее выполнение алгоритма можно прекратить. В наилучшем случае, когда входной массив уже отсортирован, алгоритму тре-

буется всего (n-1) сравнений и ни одной перестановки. К сожалению, в наихудшем случае, когда все элементы массива расположены в обратном порядке, выигрыша во времени исполнения алгоритма не происходит. Приведем все-таки реализацию этой модификации метода:

```
program modify_bubble_sort;
uses
  CRT, FileUtil;
var
  i, n: integer;
  vector: array of integer;

{ ============= Сортировка методом "пузырька" ======== }
procedure bubble(var vector: array of integer);
var
  temp: integer;
  i, j, count: integer;
  perestanovka: boolean = false;
begin
  count:= high(vector);
  for i:= 1 to count   do
  begin
    for j:= count downto i do
    if vector[j - 1] > vector[j] then
    begin
      perestanovka:= true;
      temp:= vector[j - 1];
      vector[j - 1]:=vector[j];
      vector[j]:= temp;
```

```
    end;
    if not perestanovka then break;
  end;
end;
      {=========================================}
begin
  writeln(UTF8ToConsole('Введите количество элементов массива'));
  readln(n);
  SetLength(vector, n );
  writeln(UTF8ToConsole('Введите '), n);
  writeln(UTF8ToConsole('значений элементов массива'));
  for i:= 0 to n - 1  do read(vector[i]);
  bubble(vector);
  writeln;
  writeln(UTF8ToConsole('Отсортированный массив'));
  for i:= 0 to n - 1  do write(vector[i], ' ');
  writeln;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Существуют еще несколько модификаций и улучшений этого алгоритма. При желании вы можете ознакомиться с ними в специальной литературе.

7.1.2. Сортировка выбором

Алгоритм сортировки выбором работает следующим образом: находим наименьший элемент в массиве и обмениваем его с элементом находящимся на первом месте. Затем ищем минимальный элемент без учета первого элемента и найденный минимальный элемент обмениваем со вторым элементом и так да-

лее. На i-м шаге выбираем наименьший из элементов $a[i]$, ..., $a[n]$ и меняем его местами с $a[i]$. После шага i, последовательность $a[0]$,..., $a[i]$ будет уже упорядоченной. Теперь ищем минимальный элемент среди $a[i+1]$, ..., $a[n]$ и меняем его с $a[i+1]$. Таким образом, на $(n-1)$-м шаге вся последовательность, кроме $a[n]$ оказывается отсортированной, а $a[n]$ оказывается как раз там, где он и должен стоять, так как все меньшие элементы уже "ушли" влево. Этот метод называется *сортировкой выбором,* поскольку он на каждом следующем шаге алгоритма находит наименьший из оставшихся элементов массива и переставляет его сразу в нужное место в массиве.

Количество сравнений для первого прохода равно n, для второго n-1 и т.д. Общее количество сравнений равно $n(n+1)/2 - 1$, т.е. данный алгоритм требует $O(n^2)$ сравнений. Количество же перестановок в этом алгоритме меньше, так как в каждом проходе он переставляет элементы только один раз и число перестановок составляет $O(n)$. Таким образом, алгоритм выбора несколько эффективнее пузырькового метода. К тому же, алгоритм выбора является устойчивым. Что это означает? Если среди элементов сортируемого объекта имеются одинаковые, то алгоритм не нарушает их взаимного расположения в исходном объекте. Пусть имеются две записи с одинаковыми ключами

1 A

2 B

1 C

Рис. 7.5. Записи с одинаковыми ключами

Устойчивый алгоритм отсортирует записи в таком виде:

1 A

1 C

2 B

Рис. 7.6. Результат сортировки устойчивым алгоритмом

а неустойчивый может упорядочить записи в таком виде:

1 C

1 A

2 B

Рис. 7.7. Результат сортировки неустойчивым алгоритмом

т.е. порядок следования записей с одинаковыми ключами по сравнению с исходным файлом может нарушиться.

Блок-схему алгоритма составить не представляет труда. Предлагаю это сделать вам самим.

```
program select_sort;
uses
  CRT, FileUtil;
var
  vector: array of integer;
  i, n: integer;

{ ============= Сортировка выбором ======== }
procedure select (var vector: array of integer);
var
  i, j, count, min, t: integer;
begin
  count:= high(vector);
  for i:= 0 to count - 1 do
  begin
   min:= i;
   for j:= i + 1 to count do
   if vector[j] < vector[min] then min:= j;
```

```
    t:= vector[min];
    vector[min]:= vector[i];
    vector[i]:= t;
   end;
end;
{ ================================================ }

begin
writeln(UTF8ToConsole('Введите количество элементов массива'));
   readln(n);
   SetLength(vector, n );
   writeln(UTF8ToConsole('Введите '), n);
   writeln(UTF8ToConsole('значений массива'));
   for i:= 0 to n - 1  do read(vector[i]);
   select(vector);
   writeln;
   writeln(UTF8ToConsole('Отсортированный массив'));
   for i:= 0 to n - 1  do write(vector[i], ' ');
   writeln;
   writeln(UTF8ToConsole('Нажмите любую клавишу'));
   readkey;
end.
```

Разберем случай сортировки файлов, используя алгоритм выбора. Для этого воспользуемся файлом менеджеров созданный в программе раздела 6.6.3.3. Для того чтобы вам проще было сравнивать результаты, программа создает новый отсортированный файл, а старый не отсортированный оставляет без изменений, хотя в реальных программах, как правило, отсортированный файл замещает собой исходный. Для небольших файлов более эффективным является

внутренняя сортировка, т.е. весь файл сначала считывается в некоторый массив, сортируется и затем записывается на диск на место исходного не отсортированного файла. Листинг программы сортировки типизированного файла выглядит следующим образом:

```
program select_sort_file;
uses
  CRT, FileUtil, SysUtils, OutScr;
type
  manager= record
  name: string[18];
  comp: integer;
 end;
var
  company: manager;
     {Файловые переменные}
  f_not_sorted, f_sorted: File of manager;
  vector: array of manager;
  i, n: integer;
  name_file: string;

{ Сортировка выбором, файла по фамилиям менеджеров }
procedure select (var vector: array of manager);
var
  i, j, count, min: integer;
  t: manager;
begin
  count:= high(vector);
  for i:= 0 to count - 1 do
```

```
  begin
   min:= i;
   for j:= i + 1 to count do
   if vector[j].name < vector[min].name then min:= j;
   t:= vector[min];
   vector[min]:= vector[i];
   vector[i]:= t;
  end;
end;
{ ================================================ }

begin
  {При необходимости укажите полный путь к файлу
   или скопируйте файл в папку с данным проектом}
  if not FileExists('File_not_sorted.dat') then
  begin
    writeln(UTF8ToConsole('Файлы не существуют'));
    writeln(UTF8ToConsole('Сначала создайте их'));
    writeln(UTF8ToConsole('Нажмите любую клавишу'));
    readkey;
    exit;
  end;
  AssignFile(f_not_sorted, 'File_not_sorted.dat');
  Reset(f_not_sorted);
  //  Определение количества записей в файле
  n:= System.FileSize(f_not_sorted);
  SetLength(vector, n);
  {Подготовка к внутренней сортировке,
   считывание записей файла в массив,
```

```
  в процедуру передается массив, а не файл.}
  i:= 0;
  while not Eof(f_not_sorted) do
  begin
     Read(f_not_sorted, company);

     // массив, который будет сортироваться
     vector[i]:= company;
     i:= i + 1;
  end;
  select(vector); // вызов процедуры сортировки методом выбора
  CloseFile(f_not_sorted);
  AssignFile(f_sorted, 'File_sorted.dat');
  Rewrite(f_sorted);
  for i:= 0 to n - 1 do
  Write(f_sorted, vector[i]);
  CloseFile(f_sorted);
  name_file:= 'File_sorted.dat';
  {Вызов процедуры для вывода на экран сводной ведомости.
   Процедура находится в модуле OutScr}
  output_to_screen(name_file);
  write(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Отсортируем файл менеджеров по количеству проданных компьютеров. Для этого достаточно в процедуре сортировки `select` изменить оператор

```
if vector[j].name[1] < vector[min].name[1] then min:= j;
```

на

```
if vector[j].comp < vector[min].comp then min:= j;
```

7.1.3. Сортировка вставками

Алгоритм этого метода следующий: берем первые два элемента и располагаем их в правильном порядке. Затем берем следующий элемент и вставляем его в нужное место среди тех, что мы уже обработали. Рассматриваемый элемент вставляется в позицию посредством передвижения большего элемента на одну позицию вправо и затем размещением меньшего элемента в освободившуюся позицию. На каждом шаге i элемент $a[i]$ помещается в подходящую позицию среди элементов $a[1]$, ..., $a[i-1]$. Теперь элементы $a[1]$, ..., $a[i]$ являются упорядоченными, но не "совсем", поскольку среди оставшихся элементов $a[i+1]$, ..., $a[n]$ на каком-нибудь k-м шаге может найтись элемент меньший, чем $a[1]$, ..., $a[i]$, $a[i+1]$, … $a[k-1]$ и он будет вставлен в подходящее место среди этих элементов. Процесс завершится когда последний n-й элемент будет помещен в подходящее для него место. Проиллюстрируем сказанное:

4 3 2 1

Рис. 7.8. Исходный массив

3 4 2 1

Рис. 7.9. Шаг первый, меняются первые два элемента

3 2 4 1

2 3 4 1

Рис. 7. 10. Шаг второй, элемент "2" занял свое место, при этом "3" и "4" сдвинулись вправо.

2 3 1 4

2 1 3 4

1 2 3 4

Рис. 7. 11. Шаг третий, элемент "1" занял свое место,
при этом "2", "3" и "4" сдвинулись вправо.

Алгоритм будет содержать два цикла. Во внешнем цикле указатель *i* будет двигаться направо (увеличиваться) начиная с 2, до *n*, где *n* количество элементов. Во внутреннем цикле указатель *j* сдвигается влево начиная с *i*-1. На каждом шаге *j*-м шаге происходит сдвиг элементов вправо до тех пор, пока *a*[*i*] не окажется меньше *a*[*j*]. Тогда *a*[*i*] останется на этом месте и внутренний цикл завершится. Внутренний цикл также может завершиться, когда будет достигнуто начало списка. Рассмотрим программу сортировки строки символов по алфавиту методом вставок:

```
program insertion_sort_string;
uses
  CRT, FileUtil;
var
  str: string;

{Процедура сортировки методом вставок}
procedure insert(var str: string);
var
  i, j: integer;
  v: char;
begin
```

```
  for i:= 2 to Length(str) do
  begin
    v:= str[i]; j:= i - 1;
    {Условие выхода из цикла: найдено подходящее место
     для текущего элемента или достигнуто начало строки}
    while (str[j] > v) and (j > 0) do
    begin
      str[j+1]:= str[j];
      dec(j);
    end;
    str[j+1]:= v;
  end;
end;

begin
  str:= 'hgfedcba';
  writeln(UTF8ToConsole('Исходная строка: '),
          UTF8ToConsole(str));
  insert(str);
  writeln(UTF8ToConsole('Отсортированная строка: '),
          UTF8ToConsole(str));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

В этой программе мы рассматривали строку как массив символов. Для строки, состоящей из символов кириллицы, алгоритм необходимо несколько видоизменить. Внешний цикл будет начинаться с 3, так как в кодировке UTF8 первый байт второго символа строки будет иметь номер 3.

```
program insertion_sort_string;
uses
  CRT, FileUtil;
var
  str: string;

{Процедура сортировки методом вставок кириллицы}
procedure insert_kyr(var str: string);
var
  i, j: integer;
  v, v1: string[2];
begin
  for i:= 3 to Length(str) do
  begin
    v:= Copy( str, i, 2);
    j:= i - 2;
  {Условие выхода из цикла: найдено подходящее место
  для текущего элемента или достигнуто начало строки}
    while (Copy(str, j, 2) > v) and (j > 0) do
    begin
      v1:= Copy(str, j, 2);
      str[j+2]:= v1[1];
      str[j+3]:= v1[2];
      dec(j, 2);
    end;
    str[j+2]:= v[1];
    str[j+3]:= v[2];
  end;
end;
```

```
begin
  str:= 'зжедгвба';
  writeln(UTF8ToConsole('Исходная строка: '),
          UTF8ToConsole(str));
  insert_kyr(str);
  writeln(UTF8ToConsole('Отсортированная строка: '),
          UTF8ToConsole(str));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Что будет, если строка "смешанная"? Т.е. содержит и символы кириллицы и символы латиницы. Проще всего поступить следующим образом. Просмотреть всю строку и сформировать две новые строки. Одна будет содержать только кириллицу, а другая только латиницу. Затем "скормить" процедуре insert_kyr строку с кириллицей, процедуре insert строку с латиницей. И в завершение объединить отсортированные строки функцией Concat.

```
program insertion_sort_string;
uses
  CRT, FileUtil;
var
  str, str_lat, str_kyr: string;
  i: integer;
{Процедура сортировки методом вставок латиницы}
procedure insert(var str: string);
var
  i, j: integer;
```

```
  v: char;
begin
  for i:= 2 to Length(str) do
  begin
    v:= str[i]; j:= i - 1;
  {Условие выхода из цикла: найдено подходящее место
  для текущего элемента или достигнуто начало строки}
    while (str[j] > v) and (j > 0) do
    begin
      str[j+1]:= str[j];
      dec(j);
    end;
    str[j+1]:= v;
  end;
end;

{Процедура сортировки методом вставок кирилицы}
procedure insert_kyr(var str: string);
var
  i, j: integer;
  v, v1: string[2];
begin
  for i:= 3 to Length(str) do
  begin
    v:= Copy( str, i, 2);
    j:= i - 2;
  {Условие выхода из цикла: найдено подходящее место
  для текущего элемента или достигнуто начало строки}
    while (Copy(str, j, 2) > v) and (j > 0) do
```

```
    begin
      v1:= Copy(str, j, 2);
      str[j+2]:= v1[1];
      str[j+3]:= v1[2];
      dec(j, 2);
    end;
    str[j+2]:= v[1];
    str[j+3]:= v[2];
  end;
end;

begin
  str:= 'зжедгвбahgfedcbаиклмнijklmn';
  i:= 1;
  str_lat:= '';
  str_kyr:= '';
  while i <= Length(str) do
  begin
    if ord(str[i]) < 128
    then
    begin
      str_lat:= str_lat + str[i];
      inc(i);
    end
    else
    begin
      str_kyr:= str_kyr + Copy(str, i, 2);
      inc(i, 2);
    end;
```

```
  end;
  writeln(UTF8ToConsole('Исходная строка: '),
          UTF8ToConsole(str));
  insert(str_lat); // сортировка латиницы
  insert_kyr(str_kyr); // сортировка кириллицы
  str:= Concat(str_lat, str_kyr);
  writeln(UTF8ToConsole('Отсортированная строка: '),
          UTF8ToConsole(str));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Напишем программу сортировки массива строк методом вставок. В программе строки сортируются по первому символу строки. Например, если задан массив из 4 строк:

Яковлев

Петров

Сидоров

Алексеев

то программа выдаст массив в следующем виде:

Алексеев

Петров

Сидоров

Яковлев

```
program insertion_sort_array;
uses
```

```
  CRT, FileUtil;
var
  str: array of string;
  i, n: integer;

{Процедура сортировки методом вставок}
procedure insert(var str: array of string);
var
  i, j: integer;
  stroka: string;
begin
  for i:= 1 to High(str) do
  begin
    stroka:= str[i]; j:= i - 1;
    while (str[j][1] > stroka[1]) and (j >= 0) do
    begin
      str[j+1]:= str[j];
      dec(j);
    end;
    str[j+1]:= stroka;
  end;
end;

begin
  writeln(UTF8ToConsole('Введите количество строк в массиве'));
  readln(n);
  SetLength(str, n);
  writeln(UTF8ToConsole('Введите '), n);
  writeln(UTF8ToConsole('строк(и) символов'));
```

```
  for i:= 0 to n - 1 do
  readln(str[i]);
  writeln(UTF8ToConsole('Исходный массив строк'));
  for i:= 0 to n - 1 do
  writeln(str[i]);
  insert(str);
  writeln(UTF8ToConsole('Отсортированный массив строк: '));
  for i:= 0 to n - 1 do
  writeln(str[i]);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Отсортируем массив менеджеров (раздел 6.6.3.3. и 7.1.2) методом вставок:

```
program insertion_sort_file;
uses
  CRT, SysUtils, OutScr, FileUtil;
type
  manager= record
  name: string[18];
  comp: integer;
 end;
var
  company: manager;
  f_not_sorted, f_sorted: File of manager;
  vector: array of manager;
  i, n: integer;
  name_file: string;
```

```
{Процедура сортировки методом вставок файла по фамилиям менеджеров}
procedure insert(var vector: array of manager);
var
  i, j, count: integer;
  t: manager;
begin
  count:= high(vector);
  for i:= 0 to count do
  begin
    t:= vector[i];
    j:= i - 1;
    while (vector[j].name > t.name) and (j >= 0) do
    begin
      vector[j + 1] := vector[j];
      dec(j);
    end;
    vector[j + 1] := t;
  end;
end;
{ ================================================ }
begin
  {При необходимости укажите полный путь к файлу
   или скопируйте файл в папку с данным проектом}
  if not FileExists('File_not_sorted.dat') then
  begin
    writeln(UTF8ToConsole('Файлы не существуют'));
    writeln(UTF8ToConsole('Сначала создайте их'));
    writeln(UTF8ToConsole('Нажмите любую клавишу'));
    readkey;
```

```
  exit;
end;
AssignFile(f_not_sorted,'File_not_sorted.dat');
Reset(f_not_sorted);
// Определение количества записей в файле
n:= System.FileSize(f_not_sorted);
SetLength(vector, n);
{Подготовка к внутренней сортировке,
 считывание записей файла в массив,
 в процедуру передается массив, а не файл.}
i:= 0;
while not Eof(f_not_sorted) do
begin
   Read(f_not_sorted, company);
   vector[i]:= company;  // сортируемый массив
   i:= i + 1;
end;
insert(vector); // вызов процедуры сортировки методом вставок
CloseFile(f_not_sorted);
AssignFile(f_sorted, 'File_sorted.dat');
Rewrite(f_sorted);
for i:= 0 to n - 1 do
Write(f_sorted, vector[i]);
CloseFile(f_sorted);
name_file:= 'File_sorted.dat';
{Вызов процедуры для вывода на экран сводной ведомости.
 Процедура находится в модуле OutScr}
output_to_screen(name_file);
write(UTF8ToConsole('Нажмите любую клавишу'));
```

```
  readkey;
end.
```

Во внутреннем цикле метода вставок осуществляется две проверки. Одна для нахождения подходящего места текущему элементу, а другая (`j>0`, если нумерация индексов начинается с 1 и `j>=0`, если нумерация индексов начинается с 0) для предотвращения выхода за пределы левого края массива (строку символов можно рассматривать как массив, элементами которого являются отдельные символы строки).

Однако эта дополнительная проверка, как показывают многочисленные эксперименты, замедляют работу алгоритма почти до 7%. Выходом из этой ситуации является помещение в первый элемент массива так называемого "сторожевого" элемента, который являлся бы минимальным элементом массива. Но заранее (до начала работы алгоритма) значение минимального элемента, как правило, неизвестно. Следовательно, необходимо предварительно просмотреть весь массив, найти минимальный элемент и переместить его в начало массива (фактически это выполнение первого цикла сортировки методом выбора). Теперь, когда первый элемент уже находится в требуемой позиции, можно запускать алгоритм сортировки методом вставок без проверки на выход за пределы начала массива. Программа для улучшенного метода вставок выглядит следующим образом:

```
program modify_insertion_sort_string;
uses
  CRT, FileUtil;
var
  str: string;

{Процедура, реализующая улучшенный алгоритм
 сортировки методом вставок}
```

```
procedure insert(var str:string);
var
  i, j: integer;
  v: char;
  min: integer;
begin
  {Поиск минимального элемента в строке}
  min:= 1;
  for i:= 2 to length(str) do
  if str[i] < str[min] then min:= i;
  {Меняем местами найденный символ с первым
   символом в строке}
  v:= str[1];
  str[1]:= str[min];
  str[min]:= v;
  {Реализация собственно метода вставок}
  for i:= 2 to length(str) do
  begin
    v:= str[i]; j:= i - 1;
    // теперь в цикле проверяется только одно условие
    while str[j] > v do
    begin
      str[j+1]:= str[j];
      dec(j);
    end;
    str[j+1]:= v;
  end;
end;
begin
```

```
  str:= 'dkfhytcba';
  writeln(UTF8ToConsole('Исходная строка: '),
          UTF8ToConsole(str));
  insert(str);
  writeln(UTF8ToConsole('Отсортированная строка: '),
          UTF8ToConsole(str));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

В качестве упражнения напишите программы сортировки строки с кириллицей, массива строк и файла менеджеров улучшенным методом вставок.

Как и предыдущие алгоритмы, сортировка методом вставок принадлежит классу $O(n^2)$. Если же список частично отсортирован, алгоритм метода вставок работает очень быстро и имеет порядок $O(n)$. Немаловажно и то, что алгоритм является устойчивым.

7.1.4. Метод быстрой сортировки

Алгоритм быстрой сортировки был разработан К. Хоаром в 1960 году. Быструю сортировку называют еще сортировкой с разделением или просто алгоритмом сортировки Хоара. Суть метода заключается в следующем: выбирается какой-нибудь элемент списка, называемый базовым или опорным. После этого сортируемый список делится на две части: в левую помещаются все элементы меньшие базового элемента, в правую элементы, большие базового элемента, а сам базовый элемент при этом окажется на нужном месте в списке. Далее полученные два списка сортируются таким же образом, т.е. вновь выбираются базовые элементы (уже внутри подсписков), подсписки делятся на два, в левую помещаются элементы меньшие базового элемента, в правую элементы,

большие базового элемента и т.д. При реализации алгоритма процедура, выполняющая основные действия по сортировке вызывает саму себя. Такой вызов функции или процедуры самой себя называется рекурсией. Рассмотрим алгоритм подробнее.

Сначала поговорим о выборе базового элемента. Идеальным случаем было бы выбор в качестве базового элемента среднего по *значению* элемента списка. Но для этого нужно предварительно просмотреть весь список, вычислить среднее значение, затем снова просмотреть весь список – имеется ли это среднее значение среди элементов списка и все это рекурсивно! По количеству операций это равносильно самой сортировке.

Если в качестве базового выбирать минимальный или максимальный элемент списка, то, во-первых, для этого все равно нужен предварительный просмотр всего списка (подсписков в дальнейшем), во-вторых, что еще хуже, при разделении списка на два, один из подсписков окажется пустым, поскольку все элементы списка будут находиться по одну сторону от базового. При числе элементов списка *n*, будут выполнены *n* рекурсивных вызовов. При большом числе *n* может не хватить стековой памяти, к тому же алгоритм может просто зациклиться.

Не вдаваясь в дальнейшие тонкости, скажем, что чаще всего в качестве базового элемента берут элемент средний по *месту нахождения* элемента в списке. Итак, алгоритм:

Заводятся два указателя (индекса) *i* и *j*. В начале $i = 1$ и $j = n$, где n - число сортируемых записей. На каждом шаге выполняется разбиение записей на две подгруппы так, чтобы

левая подгруппа	правая подгруппа
---	---
$R_i < R_{base}$	$R_j > R_{base}$

R_i, R_j - текущие записи (элементы); R_{base} - базовый элемент рассматри-

ваемой группы из n записей (элементов). Имеются два внутренних цикла. Первый цикл делает просмотр элементов правой подгруппы справа. Сравниваются R_{base} с R_j, $j = n, n-1, \cdots$, до тех пор, пока не встретится $R_j < R_{base}$. Затем второй внутренний цикл выполняет просмотр элементов левой подгруппы слева. Сравниваются R_{base} с R_i, $i = 1, 2, \cdots$ до тех пор, пока не встретится $R_i > R_{base}$. Теперь возможны два случая:

1. $i < j$, это означает, что два элемента, на которые указывают индексы расположены в неправильном порядке. Действительно, элемент, который находится в левом подсписке больше, чем базовый элемент, а элемент, который находится в правом подсписке, меньше базового элемента.

Меняем местами элементы и продолжаем выполнение внутренних циклов.

2. $i >= j$, это означает, что текущая подгруппа успешно разделена. Выполнение внутренних циклов завершается.

Внешний цикл начинает свою работу в новь с определения базового элемента, теперь уже для текущей подгруппы.

Теоретические расчеты эффективности метода показывают в среднем $n\log_2(n)$ операций сравнений.

Напишем программу быстрой сортировки для массива с произвольным числом элементов. В качестве элементов возьмем целые числа.

```
program quick_sort;
uses
  CRT, FileUtil;
type
  vector= array of integer;
var
  i, n: integer;
  sorted_array: vector; // сортируемый массив
```

```
    { ============= Метод быстрой сортировки ======== }
procedure QuickSort(var sorted_array: vector);

  { rec_sort - рекурсивная процедура }
procedure rec_sort(first, last: integer;
                   var sorted_array: vector);
 { first, last - первый и последний элементы текущей группы }
var
  i: integer; // указатель для левого списка
  j: integer;  // указатель для правого списка
  middle: integer; // средний элемент списка
  temp: integer;
begin
  while (first < last) do
  begin
    middle:= sorted_array[(first + last) div 2];
    i:= pred(first);
    j:= succ(last);
    while true do
    begin
      repeat dec(j);
      until sorted_array[j] <= middle;
      repeat inc(i);
      until sorted_array[i] >= middle;
      if i >= j then break;
      temp:= sorted_array[i];
      sorted_array[i]:= sorted_array[j];
      sorted_array[j]:= temp;
      end;
```

```
  {рекурсивный вызов процедуры}
  if first < j then rec_sort(first, j, sorted_array);
  first:= succ(j);
  end;
end;
{ ======= конец процедуры rec_sort ======== }
begin
  { первый вызов рекурсивной процедуры}
  rec_sort(0, n - 1, sorted_array);
end;
{ ======= конец процедуры QuickSort ======== }
begin
  writeln(UTF8ToConsole('Введите количество элементов массива'));
  readln(n);
  SetLength(sorted_array, n);
  writeln(UTF8ToConsole('Введите элементы массива'));
  for i:= 0 to n - 1 do
    read(sorted_array[i]);
  QuickSort(sorted_array);
  writeln(UTF8ToConsole('Отсортированный массив:'));
  writeln;
  for i:= 0 to n - 1 do
    write(sorted_array[i], ' ');
  writeln;
  writeln;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

В программе использованы функции `pred()` и `succ()`.

Функция `succ` увеличивает (инкрементирует) значение порядковой переменной. Можно инкрементировать:

- Символы;
- Невещественные числовые типы;
- Тип перечисления;
- Указатели.

Функция `pred` уменьшает значение порядковой переменной.

Выше мы неоднократно отмечали, что один и тот же алгоритм можно реализовывать по разному. Вот пример другой реализации того же алгоритма.

```
program quick_sort;
uses
  CRT, FileUtil;
type
  vector= array of integer;
var
  i, n: integer;
  sorted_array: vector; // сортируемый массив

  { ============ Метод быстрой сортировки ======= }
procedure QuickSort(var sorted_array: vector);

  { rec_sort - рекурсивная процедура }
procedure rec_sort(first, last: integer;
                   var sorted_array: vector);
 { first, last - первый и последний элементы текущей группы }
```

```
var
  i: integer; // указатель для левого списка
  j: integer;  // указатель для правого списка
  middle: integer; // средний элемент списка
  temp: integer;
begin
  i:= first; // первый элемент текущего списка
  j:= last;  // последний элемент текущего списка
  middle:= sorted_array[(first + last) div 2];
  repeat
    while sorted_array[i] < middle do
      i:= i + 1;
    while middle < sorted_array[j] do
      j:= j - 1;
    if i <= j then
    begin
      temp:= sorted_array[i];
      sorted_array[i]:= sorted_array[j];
      sorted_array[j]:= temp;
      i:= i + 1;
      j:= j - 1;
    end;
  until i > j;
  {рекурсивные вызовы процедуры для левых и правых подсписков}
  if first < j then rec_sort(first, j, sorted_array);
  if i < last then rec_sort(i, last, sorted_array);
end;
 { ========================================================= }
begin
```

```
  {первый вызов рекурсивной процедуры}
  rec_sort(0, n - 1, sorted_array);
end;
      {=========================================}
begin
  writeln(UTF8ToConsole('Введите количество элементов массива'));
  readln(n);
  SetLength(sorted_array, n);
  writeln(UTF8ToConsole('Введите элементы массива'));
  for i:= 0 to n - 1 do
    read(sorted_array[i]);
  QuickSort(sorted_array);
  writeln(UTF8ToConsole('Отсортированный массив:'));
  writeln;
  for i:= 0 to n - 1 do
    write(sorted_array[i], ' ');
  writeln;
  writeln;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
   end.
```

Напишем программу сортировка файла алгоритмом Хоара.

```
  program quick_sort_file;
uses
  CRT, FileUtil, SysUtils, OutScr;
type
  manager= record
```

```
  name: string[18];
  comp: integer;
 end;
var
  company: manager;
  f_not_sorted, f_sorted: File of manager;
  vector: array of manager;
  i, n: integer;
  name_file: string;

  { ========= Метод быстрой сортировки ======== }
procedure QuickSort(var sorted_array: array of manager);

  { rec_sort - рекурсивная процедура }
procedure rec_sort(first, last: integer;
                   var sorted_array: array of manager);
 { first, last - первый и последний элементы текущей группы }
var
  i: integer; // указатель для левого списка
  j: integer;  // указатель для правого списка
  middle: manager; // средний элемент списка
  temp: manager;
begin
  while (first < last) do
  begin
    middle:= sorted_array[(first + last) div 2];
    i:= Pred(first);
    j:= Succ(last);
    while true do
```

```
    begin
      repeat dec(j);
      until sorted_array[j].name <= middle.name;
      repeat inc(i);
      until sorted_array[i].name >= middle.name;
      if i >= j then break;
      temp:= sorted_array[i];
      sorted_array[i]:= sorted_array[j];
      sorted_array[j]:= temp;
      end;
  {рекурсивный вызов процедуры}
  if first < j then rec_sort(first, j, sorted_array);
  first:= Succ(j);
  end;
end;
 { ========================================= }
begin
  {первый вызов рекурсивной процедуры}
  rec_sort(0, n - 1, sorted_array);
end;
{ ========================================= }

begin
  {При необходимости укажите полный путь к файлу
   или скопируйте файл в папку с данным проектом}
  if not FileExists('File_not_sorted.dat') then
  begin
    writeln(UTF8ToConsole('Файлы не существуют'));
    writeln(UTF8ToConsole('Сначала создайте их'));
```

```
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  exit;
end;
AssignFile(f_not_sorted, 'File_not_sorted.dat');
Reset(f_not_sorted);
// Определение количества записей в файле
n:= System.FileSize(f_not_sorted);
SetLength(vector, n);
{Подготовка к внутренней сортировке,
 считывание записей файла в массив,
 в процедуру передается массив, а не файл.}
i:= 0;
while not Eof(f_not_sorted) do
begin
   Read(f_not_sorted, company);
   vector[i]:= company;  // сортируемый массив
   i:= i + 1;
end;
QuickSort(vector); // вызов процедуры быстрой сортировки
CloseFile(f_not_sorted);
AssignFile(f_sorted, 'File_sorted.dat');
Rewrite(f_sorted);
for i:= 0 to n - 1 do
Write(f_sorted, vector[i]);
CloseFile(f_sorted);
name_file:= 'File_sorted.dat';
{Вызов процедуры для вывода на экран сводной ведомости.
```

```
  Процедура находится в модуле OutScr}
  output_to_screen(name_file);
  write(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Резюмируя все вышесказанное относительно методов сортировки можно сказать, что наиболее часто используются метод вставок и алгоритм быстрой сортировки Хоара. Что касается других алгоритмов сортировки, то вы можете их найти в специальной литературе [9, 10, 12].

7.2. Алгоритмы поиска

Поиск каких-либо данных, наряду с сортировкой, является одной из наиболее распространенных задач при разработке сколько-нибудь сложных программных проектов, в частности при разработке информационно-поисковых и информационно-справочных систем. В любой базе данных присутствуют задачи сортировки и поиска. Алгоритмы поиска являются не менее сложными и интересными, чем алгоритмы сортировки.

Рассмотрим некоторые алгоритмы поиска в массивах.

7.2.1. Поиск в массивах

Пусть нам дан какой-нибудь массив. Требуется найти элемент массива по заданному ключу. Первое, что приходит на ум, чтобы решить эту задачу, это последовательно сравнивать элементы массива с заданным ключом, пока не будет найден искомый элемент. Если элемент найден, алгоритм должен выдать номер индекса этого элемента в массиве. Если же такого элемента в массиве нет, то алгоритм должен нам каким-то образом сообщить об этом. Чаще всего в

таком случае алгоритм возвращает значение, которого заведомо нет в массиве, а точнее такого индекса. Такой алгоритм называется алгоритмом линейного поиска. Напишем соответствующую функцию в предположении, что поиск ведется в массиве целых чисел.

```
function LinearSearch(var a: array of integer;
                      key: integer): integer;
var
  i: integer;
begin
  for i:=0 to High(a) do
  begin
    if key = a[i] then
    begin
      LinearSearch := i;
      exit;
    end;
  end;
LinearSearch:= -1;
end;
```

Итак, эта функция возвращает нам номер индекса в массиве, где находится искомый элемент. Возвращать само значение не имеет смысла, мы и так его знаем (значение `key`). В случае отсутствия искомого элемента функция возвращает -1. Обратите внимание, в самом массиве может быть и есть элемент со значением -1, но функция возвращает индекс элемента в массиве. Как мы знаем, индекс не может быть отрицательным, возврат -1 и говорит нам о том, что искомого элемента в массиве нет.

Программа линейного поиска для целочисленного массива:

```
program search_in_array;
uses
  CRT, FileUtil;
const SizeOfFile= 9;
var
  a:array[0..SizeOfFile] of integer =
  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
  n: integer;
  key: integer;

  { ============= Линейный поиск ======== }
function LinearSearch(var a:array of integer;
                      key: integer): integer;
var
  i: integer;
begin
  for i:=0 to SizeOfFile do
  begin
    if key = a[i] then
    begin
      LinearSearch := i;
      exit;
    end;
  end;
LinearSearch:= -1;
end;
{ ========================================= }
```

```
begin
  writeln(UTF8ToConsole('Введите ключ для поиска'));
  readln(key);
  {вызов функции поиска}
  n:= LinearSearch(a, key);
  if n= -1 then
     writeln(UTF8ToConsole('Такого элемента в массиве нет'))
  else
     writeln(UTF8ToConsole('Элемент найден, его номер в массиве '),
n);
  write(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Напишем программу поиска менеджера по фамилии. Для удобства проверки результатов работы программы, одновременно выводится весь файл менеджеров и указывается номер искомой записи.

```
program search_in_file;
uses
  CRT, FileUtil, SysUtils, OutScr;
type
    manager= record
    name: string[18];
    comp: integer;
 end;
var
  company: manager;
  f_not_sorted: File of manager; // Файловая переменная
```

```
  vector: array of manager;
  i, n: integer;
  name_file: string;
  name_manager: string;
  { ========= Линейный поиск ======== }
function LinearSearch(var not_sorted_array:
                      array of manager;
                      name_manager: string): integer;
var
  i: integer;
begin
  for i:= 0 to High(not_sorted_array) do
  begin
    if name_manager = not_sorted_array[i].name then
    begin
      LinearSearch := i;
      exit;
    end;
  end;
LinearSearch:= -1;
end;
{ ================================== }

begin
  {При необходимости укажите полный путь к файлу
   или скопируйте файл в папку с данным проектом}
  if not FileExists('File_not_sorted.dat') then
  begin
    writeln(UTF8ToConsole('Файл не существует'));
```

```
  writeln(UTF8ToConsole('Сначала создайте его'));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  exit;
end;
AssignFile(f_not_sorted,'File_not_sorted.dat');
Reset(f_not_sorted);
// Определение количества записей в файле
n:= System.FileSize(f_not_sorted);
SetLength(vector, n);
{Подготовка поиску,
 считывание записей файла в массив,
 в функцию передается массив, а не файл.}
i:= 0;
while not Eof(f_not_sorted) do
begin
   Read(f_not_sorted, company);
   vector[i]:= company; // массив, в которым будет вестись поиск
   i:= i + 1;
end;
CloseFile(f_not_sorted);
writeln(UTF8ToConsole('Введите фамилию менеджера'));
readln(name_manager);
{вызов функции поиска}
n:= LinearSearch(vector, name_manager);
name_file:= 'File_not_sorted.dat';
{Вызов процедуры для вывода на экран списка менеджеров.
 Процедура находится в модуле OutScr}
output_to_screen(name_file);
```

```
  if n= -1 then
     writeln(UTF8ToConsole('Такого менеджера нет'))
  else
     writeln(UTF8ToConsole('Менеджер найден, его номер '),
n+1);
  write(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Очевидно, что время поиска по этому алгоритму зависит от размера массива – количества его элементов *n*. Т.е. этот алгоритм принадлежит классу O(*n*).

Если массив неупорядочен, то единственный способ найти какой-либо элемент, это линейный поиск. Если же массив упорядочен, то существуют более быстрые и эффективные алгоритмы. Наиболее часто используемым является алгоритм двоичного (бинарного) поиска. Его также часто называют метод деления пополам.

Суть метода в том, что, пользуясь тем, что массив отсортирован, вместо просмотра подряд всех элементов массива, находим элемент, находящийся посередине массива. Назовем его средним элементом (не по значению, а по месту в массиве). Затем сравниваем искомый элемент с этим средним элементом. Если искомый элемент меньше среднего, то искать теперь его следует только среди тех, которые меньше среднего. Таким образом, диапазон поиска уменьшается ровно наполовину. Берем ту половину массива, где следует искать наш элемент и опять находим средний элемент, опять его сравниваем с искомым элементом и вновь суживаем зону поиска наполовину и т.д. продолжаем процесс до тех пор, пока в рассматриваемых подмассивах не останется один элемент. Если он равен искомому, значит элемент найден, если не равен, значит такого элемента в массиве нет.

При выполнении алгоритма в каждом следующем цикле размер массива

уменьшается в два раза, отсюда оценка скорости работы алгоритма O($\log_2 n$).

Напишем программу бинарного поиска в массиве целых чисел:

```
program search_in_array;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
const SizeOfFile= 9;
var
  a: array[0..SizeOfFile] of integer =
  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
  n: integer;
  key: integer;

  { ======== Бинарный поиск ======== }
function BinarySearch(var a: array of integer;
                      key: integer): integer;
var
  left, right, middle: integer;
begin
  left:= Low(a);
  right:= High(a);
  repeat
  middle:= (left + right) div 2;
  if key < a[middle] then
    right:= middle - 1
  else
    if key > a[middle] then
      left:= middle + 1
```

```
    else
    begin
      BinarySearch:= middle;
      exit;
    end;
  until left > right;
  BinarySearch:= -1;
end;
{ ========================================== }

begin
  writeln(UTF8ToConsole('Введите ключ для поиска'));
  readln(key);
  {вызов функции поиска}
  n:= BinarySearch(a, key);
  if n= -1 then
     writeln(UTF8ToConsole('Такого элемента в массиве нет'))
  else
     writeln(UTF8ToConsole('Элемент найден, его номер '), n);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Напишем программу поиска в файле менеджеров по фамилии бинарным методом:

```
program search_in_file;
uses
  CRT, FileUtil, SysUtils, OutScr;
```

```
type
    manager= record
    name: string[18];
    comp: integer;
 end;
var
  company: manager;
  f_not_sorted, f_sorted: File of manager;
  vector: array of manager;
  i, n: integer;
  name_file: string;
  name_manager: string;
  { ======= Бинарный поиск ======== }
function BinarySearch(var a: array of manager;
                      name_manager: string): integer;
var
  left, right, middle: integer;
begin
  left:= Low(a);
  right:= High(a);
  repeat
  middle:= (left + right) div 2;
  if name_manager < a[middle].name then
    right:= middle - 1
  else
    if name_manager > a[middle].name then
      left:= middle + 1
    else
    begin
```

```
      BinarySearch:= middle;
      exit;
    end;
  until left > right;
  BinarySearch:= -1;
end;

{  ========================================== }
begin
  {При необходимости укажите полный путь к файлу
   или скопируйте файл в папку с данным проектом}
  if not FileExists('File_not_sorted.dat') then
  begin
    writeln(UTF8ToConsole('Файл не существует'));
    writeln(UTF8ToConsole('Сначала создайте его'));
    writeln(UTF8ToConsole('Нажмите любую клавишу'));
    readkey;
    exit;
  end;
  AssignFile(f_sorted,'File_sorted.dat');
  Reset(f_sorted);
  n:= System.FileSize(f_sorted);
  SetLength(vector, n);
  {Подготовка к поиску,
   считывание записей файла в массив,
   в функцию передается массив, а не файл.}
  i:= 0;
  while not Eof(f_sorted) do
  begin
```

```
      Read(f_sorted, company);
      vector[i]:= company; // массив, который будет сортироваться
      i:= i + 1;
  end;
  CloseFile(f_sorted);
  writeln(UTF8ToConsole('Введите фамилию менеджера'));
  readln(name_manager);
  {вызов функции поиска}
  n:= BinarySearch(vector, name_manager);
  name_file:= 'File_sorted.dat';
  {Вызов процедуры для вывода на экран сводной ведомости.
   Процедура находится в модуле OutScr}
  output_to_screen(name_file);
   if n= -1 then
      writeln(UTF8ToConsole('Такого менеджера нет'))
  else
      writeln(UTF8ToConsole('Менеджер найден, его номер '),
n+1);
  write(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

7.2.2. Вставка и удаление элементов в упорядоченном массиве

Часто бывает необходимо вставлять новые элементы в упорядоченный массив, а также удалять элементы из массива. Такие задачи часто возникают в системах управления базами данных, где пользователи вводят большое количество данных. "Приличная" система должна давать пользователям возможность корректирования введенных данных, т.е. вставки новых данных, изменения уже

введенных данных и удаления данных. Задача эта не столь тривиальна, как может показаться на первый взгляд. Пусть необходимо вставить новый элемент x в упорядоченный по возрастанию массив $a[1..n]$ в место с номером k, где $k<n$. Если просто присвоить

$a[k]:=x$;

то элемент с индексом k, который находился там ранее, будет потерян. Следовательно, предварительно необходимо сдвинуть все элементы $a[k]$, $a[k+1]$, …, $a[n]$ на одну позицию вправо. Это можно сделать с помощью цикла:

```
for i:= n downto k do
    a[i+1]:= a[i];
    inc(n);
```

То же самое при удалении элемента из массива. Если удаляется элемент $a[k]$, то все оставшиеся элементы $a[k+1]$, …, $a[n]$ надо сдвинуть на одну позицию влево, чтобы закрыть возникшую "дыру" в k-й позиции массива.

Код для сдвига элементов массива при удалении элемента $a[k]$ будет таким:

```
for i:= k + 1 to n do
    a[i-1]:=a[i];
    dec(n);
```

Напишем программу вставки нового элемента в массив. Предположим, что массив состоит из целых чисел 1, 2, 3, 4. Моделируем ситуацию, когда пользователь при вводе массива допустил ошибку, пропустил число 3. Поскольку оператор `readln` "не реагирует" на ввод пустой строки, была введена следующая последовательность чисел: 1, 2, 4, 5. Необходимо дать пользователю возможность откорректировать данные, введенные им в массив.

```
program project1;
uses
  CRT, FileUtil;
var
  i, n: integer;
  a: array of integer;
  NewElement: integer;
  IndexOfNewElement: integer;
  SizeOfArray: integer;
  answ: char;
begin
  writeln(UTF8ToConsole('Введите размер массива'));
  readln(SizeOfArray);
  SetLength(a, SizeOfArray);
  writeln(UTF8ToConsole('Введите элементы массива'));
  for i:= 0 to SizeOfArray - 1 do
  read(a[i]);
  writeln(UTF8ToConsole('Введен массив:'));
  for i:= 0 to SizeOfArray - 1 do
  write(a[i], ' '); writeln;
  writeln(UTF8ToConsole('Откорректируйте данные'));
  writeln(UTF8ToConsole('Если все правильно, нажмите Enter,'));
  writeln(UTF8ToConsole('иначе - любую клавишу'));
  answ:= readkey;
  if answ = #13 then exit;
  writeln(UTF8ToConsole('Введите новый элемент'));
  readln(NewElement);
  writeln(UTF8ToConsole('Введите индекс в массиве,'));
```

```
  writeln(UTF8ToConsole('куда нужно вставить новый элемент'));
  readln(IndexOfNewElement);
  for i:= SizeOfArray - 1 downto IndexOfNewElement do
    a[i+1]:= a[i];
  a[IndexOfNewElement]:= NewElement;
  if IndexOfNewElement= SizeOfArray then
  inc(SizeOfArray);
  writeln(UTF8ToConsole('Массив после вставки нового элемента'));
  for i:= 0 to SizeOfArray - 1 do
  write(a[i], ' '); writeln;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Как всегда, ищем возможность улучшения программы. В предыдущей программе пользователь сам вводил индекс в массиве, куда нужно вставить новый элемент. Оказывается, алгоритм бинарного поиска позволяет автоматически вставлять новый элемент в нужное место. Каким образом? Рассмотрим на конкретном примере. Пусть имеется массив из 5 элементов. Необходимо вставить число 3. Вспомним алгоритм бинарного поиска. Первоначальное расположение указателей будет, так как показано на рисунке 7.12.

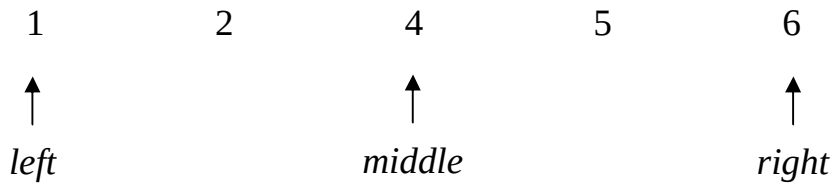


Рис. 7.12. Начальное расположение указателей

При этом значения переменных будут равны:

key = 3

left = 0 (не забывайте, нумерация в динамических массивах с нуля!)

right = 4

middle = (*left*+*right*) div 2 = (0 + 4) div 2 = 2

a[*middle*] = *a*[2] = 4

key = 3 < *a*[2] = 4, т.е. значение ключа меньше значения среднего элемента. Согласно алгоритму, переменной *right* будет присвоено значение:

right = *middle*-1 = 2 – 1 = 1

middle = (*left*+*right*) div 2 = (0 + 1) div 2 = 1

Новое расположение указателей показано на рисунке 7.13.

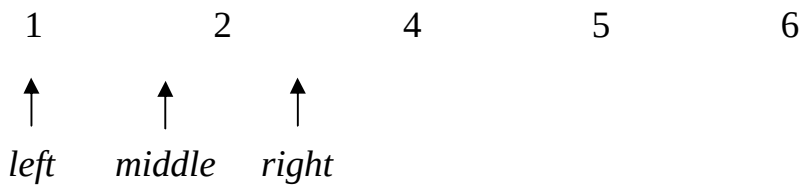


Рис. 7. 13 Новое расположение указателей

a[*middle*] = *a*[1] = 2

key = 3 > *a*[1] = 2, т.е. значение ключа больше значения среднего элемента, следовательно, переменной *left* будет присвоено значение

left = *middle* + 1 = 1 + 1 = 2

Значение *left* оказывается больше *right*:

left = 2 > *right* = 1

Алгоритм поиска заканчивает свою работу, искомый элемент не найден. Но, посмотрите чему равно значение *left*. Оно равно 2, т.е. как раз тому индексу в массиве, куда должен быть помещен новый элемент, т.е. *key* равный 3!

В алгоритме достаточно изменить один оператор, в том месте, где функция BinarySearch возвращает -1 (элемент не найден!), т.е. вместо

```
BinarySearch:= -1;
```

необходимо записать

```
    BinarySearch:= left;
```

Итак, улучшенная программа предыдущего примера. В этой программе вставка нового элемента оформлена в виде процедуры. Кроме того, введены проверки на правильность ввода размера массива

```
program modify_insert;
uses
  CRT, FileUtil;
var
  i: integer;
  a: array of integer;
  NewElement: integer;
  IndexOfNewElement: integer;
  SizeOfArray: integer;
  answ: char;

  { ========= Бинарный поиск ======== }

function BinarySearch(var a: array of integer;
                      key: integer): integer;
var
  left, right, middle: integer;
begin
  left:= Low(a);
  right:= High(a);
  repeat
  middle:= (left + right) div 2;
  if key < a[middle] then
```

```
    right:= middle - 1
  else
    if key > a[middle] then
      left:= middle + 1
    else
    begin
      BinarySearch:= middle;
      exit;
    end;
  until left > right;
  BinarySearch:= left;
end;

{ ============================================== }
procedure insert_new_element(var a: array of integer;
                             var NewElement: integer;
                             var IndexOfNewElement: integer);
begin
  for i:= SizeOfArray - 1 downto IndexOfNewElement do
  a[i + 1]:= a[i];
  a[IndexOfNewElement]:= NewElement;
  if IndexOfNewElement= SizeOfArray then
  inc(SizeOfArray);
end;
begin
  writeln(UTF8ToConsole('Введите размер массива'));
  while true do
  begin
    readln(SizeOfArray);
```

```
    if SizeOfArray = 0 then
    begin
      writeln(UTF8ToConsole('Размер массива не может быть = 0'));
      writeln(UTF8ToConsole('Введите размер массива'));
    end
    else
    if SizeOfArray < 0 then
    begin
      writeln(UTF8ToConsole('Размер массива не может быть < 0'));
      writeln(UTF8ToConsole('Введите размер массива'));
    end
    else break;
  end;
  SetLength(a, SizeOfArray);
  writeln(UTF8ToConsole('Введите элементы массива'));
  for i:= 0 to SizeOfArray - 1 do
  read(a[i]);
  writeln(UTF8ToConsole('Введен массив:'));
  for i:= 0 to SizeOfArray - 1 do
  write(a[i], ' '); writeln;
  writeln(UTF8ToConsole('Откорректируйте данные'));
  writeln(UTF8ToConsole('Если все правильно, нажмите Enter,'));
  writeln(UTF8ToConsole('иначе - любую клавишу'));
  answ:= readkey;
  if answ = #13 then exit;
  writeln(UTF8ToConsole('Введите новый элемент'));
  readln(NewElement);
  IndexOfNewElement:= BinarySearch(a, NewElement);
  insert_new_element(a, NewElement, IndexOfNewElement);
```

```
  writeln(UTF8ToConsole('Массив после вставки нового элемента'));
  for i:= 0 to SizeOfArray - 1 do
  write(a[i], ' '); writeln;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

В программе предполагается, что первоначальный размер массива не должен изменяться. Если вы помните, в примере мы моделировали ситуацию, когда пользователю необходимо было откорректировать ошибочно введенные данные. Если же вам нужно, чтобы вставляемый элемент расширял исходный массив, вам нужно в процедуре `insert_new_element` заменить условный оператор

```
if IndexOfNewElement = SizeOfArray then
  inc(SizeOfArray);
```

на просто оператор инкремента

```
  inc(SizeOfArray);
```

Предлагаю вам самостоятельно написать процедуру вставки в массив только уникального элемента, т.е. если в исходном массиве уже имеется такой элемент, то вставка не производится.

7.3. Динамические структуры данных

При описании некоторых задач применяются абстрактные структуры данных, в частности графы.

Ориентированный граф – это система из двух множеств $G=(X,U)$, где X - множество элементов, называемых вершинами, а U - определенное на множестве X отношение (т.е. $U \supseteq X \times X$), элементы которого называются дугами или ребрами. Если a и b - вершины, то (a,b) - ребро. Говорят, что ребро направлено от a к b.

Неориентированный граф – это такой граф, в котором отсутствует ориентация ребер, т.е. $(a,b)=(b,a)$.

Если каждому ребру поставить в соответствие некоторое число, то это взвешенный граф.

Вершины графа или его ребра (или те и другие) могут быть помечены. В качестве меток могут использоваться символы или числа.

Рис. 7.14 Примеры графов

Путем в графе называется последовательность вершин, связанных между собой ребрами. Две вершины связаны между собой, если существует путь от одной вершины до другой.

Граф называется связным, если все пары вершин связаны. Будем говорить, что граф пуст, если в нем нет вершин (а, следовательно, и ребер).

Рассмотрим теперь графы специального вида, называемыми деревьями. Деревом называется связный граф, в котором:

1) имеется единственная особая вершина, называется корнем, в которую не заходит ни одно ребро;
2) во все остальные вершины (иначе называемых листьями, а также узлами) заходит только одно ребро, а исходит сколько угодно ребер;
3) нет циклов (т.е. замкнутых петель)

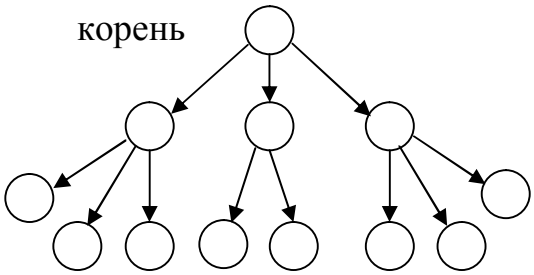


Рис. 7.15 Пример дерева

На рисунках корень указывают с помощью наглядного расположения, когда корень изображается самой верхней вершиной. С помощью дерева мы можем представить родословную некоторого человека.

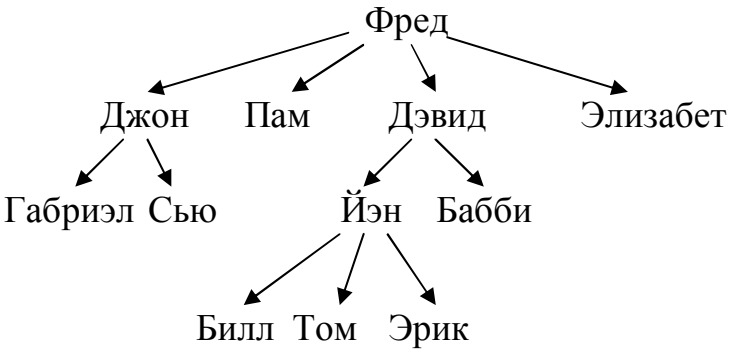


Рис. 7.16 Родословная человека по имени Фред

Этой родословной соответствует дерево вида:

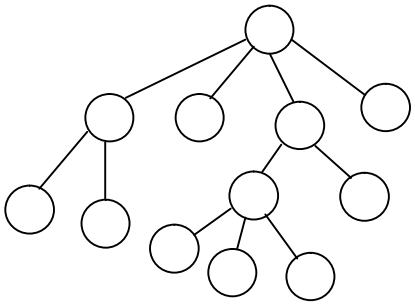


Рис. 7.17 Дерево, соответствующее родословной Фреда

Поэтому вершины нижних уровней называют еще потомками или сыновьями.

Одно из первых применений деревьев в программировании это трансляция арифметических выражений. Пусть имеется некоторое арифметическое выражение: X+Y*Z-A*B+C/D

При переводе этого выражения на внутренний язык компилятор строит дерево вида:

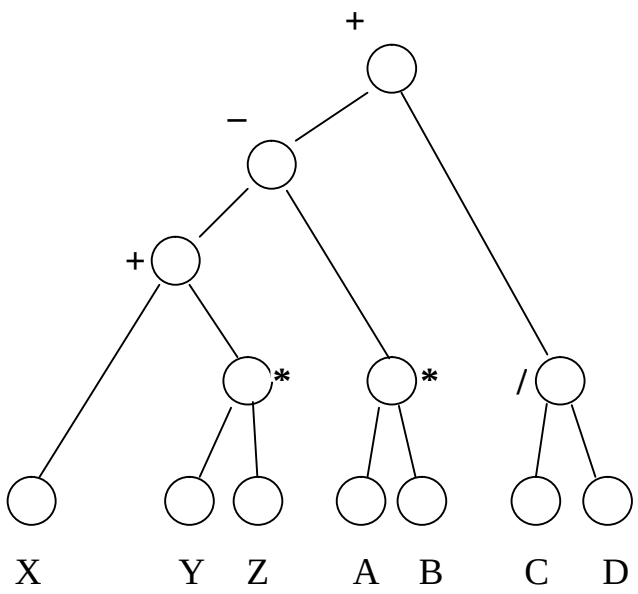


Рис. 7.18 Дерево, построенное компилятором при разборе выражения

Любая иерархическая структура может быть представлена в виде дерева. Рассмотрим упрощенную структуру типичного университета.

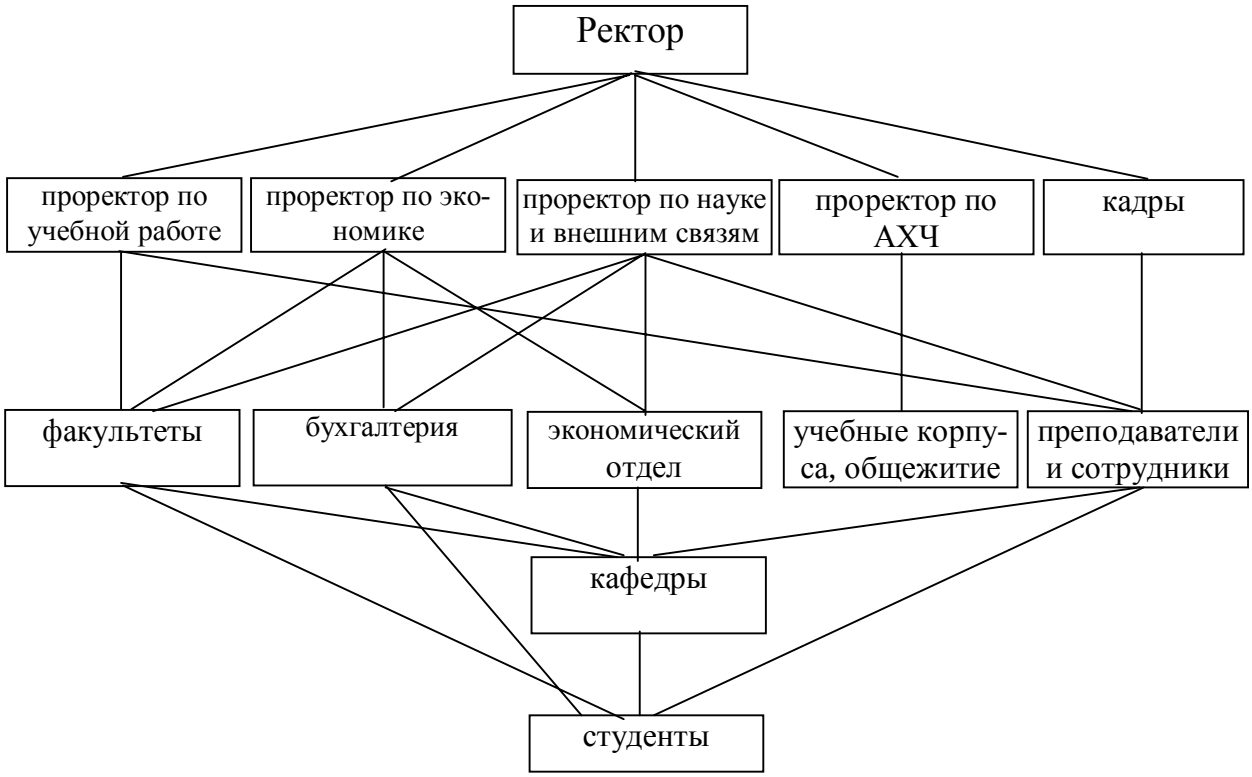


Рис. 7.19 Структура типичного университета

Нарисовать соответствующее дерево предоставляется самому читателю.

Количество ребер выходящих из любой вершины называется степенью узла или степенью вершины. Если из вершины не выходит ни одного ребра, степень такой вершины равна нулю.

Важнейшим классом деревьев, чаще всего используемых в программировании являются так называемые двоичные или бинарные деревья. Двоичным деревом называется такое дерево, из каждой вершины которого выходит не более двух ребер, т.е. в степень двоичного дерева не превышает двух. Строгое бинарное дерево состоит только из узлов, имеющих степень два или степень ноль. Нестрогое бинарное дерево содержит узлы со степенью равной 0, 1 или 2.

В строгом бинарном дереве на каждом уровне n может быть не более 2^{n-1} вершин.

Бинарное дерево называется полным, если в строгом бинарном дереве на каждом n-м уровне содержатся все 2^{n-1} вершин, рис. 7.20.

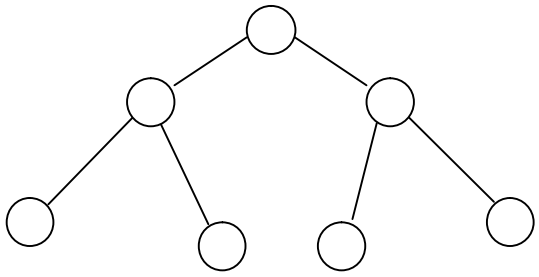


Рис. 7.20. Бинарное (двоичное) дерево

Наиболее часто используемые действия над деревьями – это обход дерева. Обходя дерево, мы можем что-то делать с вершиной, которую обходим, например, печатать номер вершины. Различают 3 способа обхода дерева:

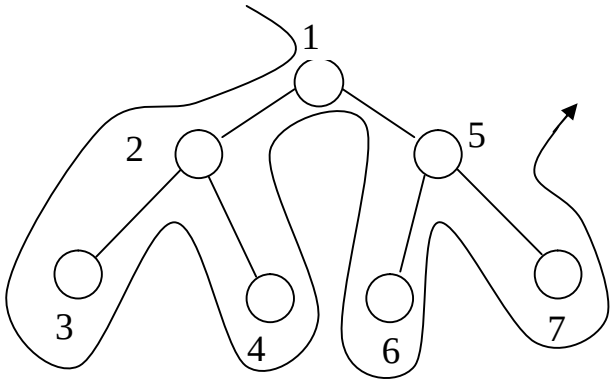


Рис. 7.21 Обход дерева

1) обход сверху, при этом сначала обрабатывают корень, затем левое поддерево, и затем правое поддерево. Тогда будут напечатаны 1, 2, 3, 4, 5, 6, 7;
2) обход слева. Здесь сначала обрабатывают левое поддерево, затем корень, затем правое поддерево. Будут напечатаны 3, 2, 4, 1, 6, 5, 7;
3) обход снизу. Здесь обрабатываются сначала левое поддерево, затем правое поддерево, затем корень. Будут напечатаны 3, 4, 2, 6, 7, 5, 1.

Обход слева часто используется в алгоритмах сортировки, при работе с таблицами.

Рассмотрим еще одну информационную структуру – стек.

Стек это динамическая структура, приспособленная для того, чтобы добавлять элементы в стек и извлекать их оттуда по определенным правилам – очередной элемент в стек можно поместить или взять только через специальное место, называемое верхушкой стека. В верхушке стека находится всегда эле-

мент, помещенный в него последним. Стек работает по принципу «последним пришел, первым ушел» ("Last In – First Out", LIFO).

Лицам мужского пола, возможно, будет легче понять принцип работы стека, если они представят себе обойму пистолета. Патрон, помещенный в обойму последним, первым уйдет в цель. Особам женского пола более приятно будет ассоциировать стек со стопкой тарелок. Тарелка, помещенная в стопку последней, первой пойдет в "дело".

Рассмотрим процесс помещения в стек трех элементов a, b, c. Первоначально стек пусть будет пустым. На рисунке 7.22 показаны последовательные состояния стека при добавлении элементов a, b, c.

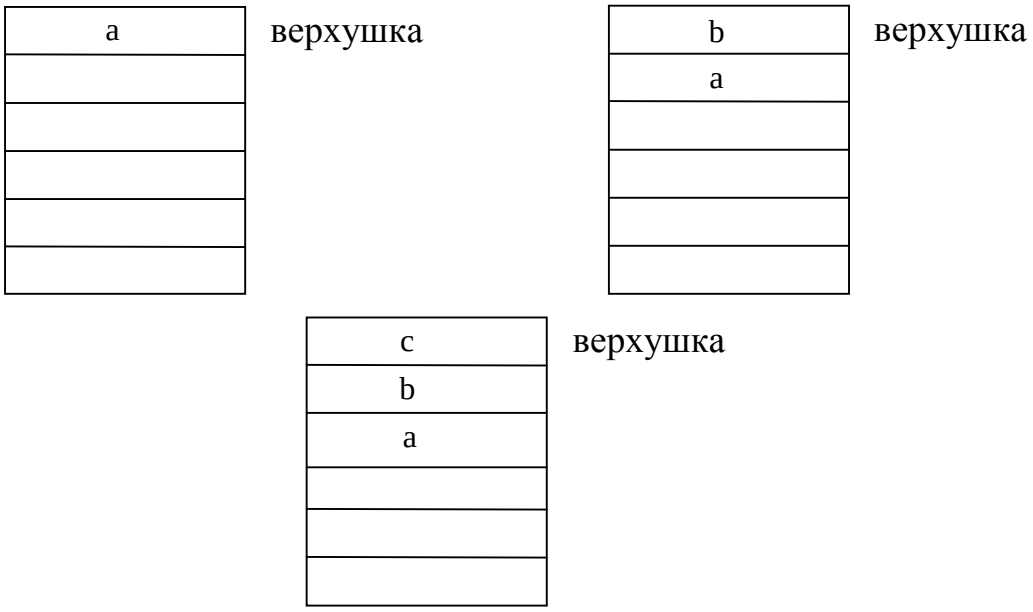


Рис. 7.22. Последовательные состояния стека при добавлении элементов

Таким образом, последний добавленный в стек элемент *c* оказывается в верхушке стека. Применение стека рассмотрим на примере составления алгоритма обхода двоичного дерева слева.

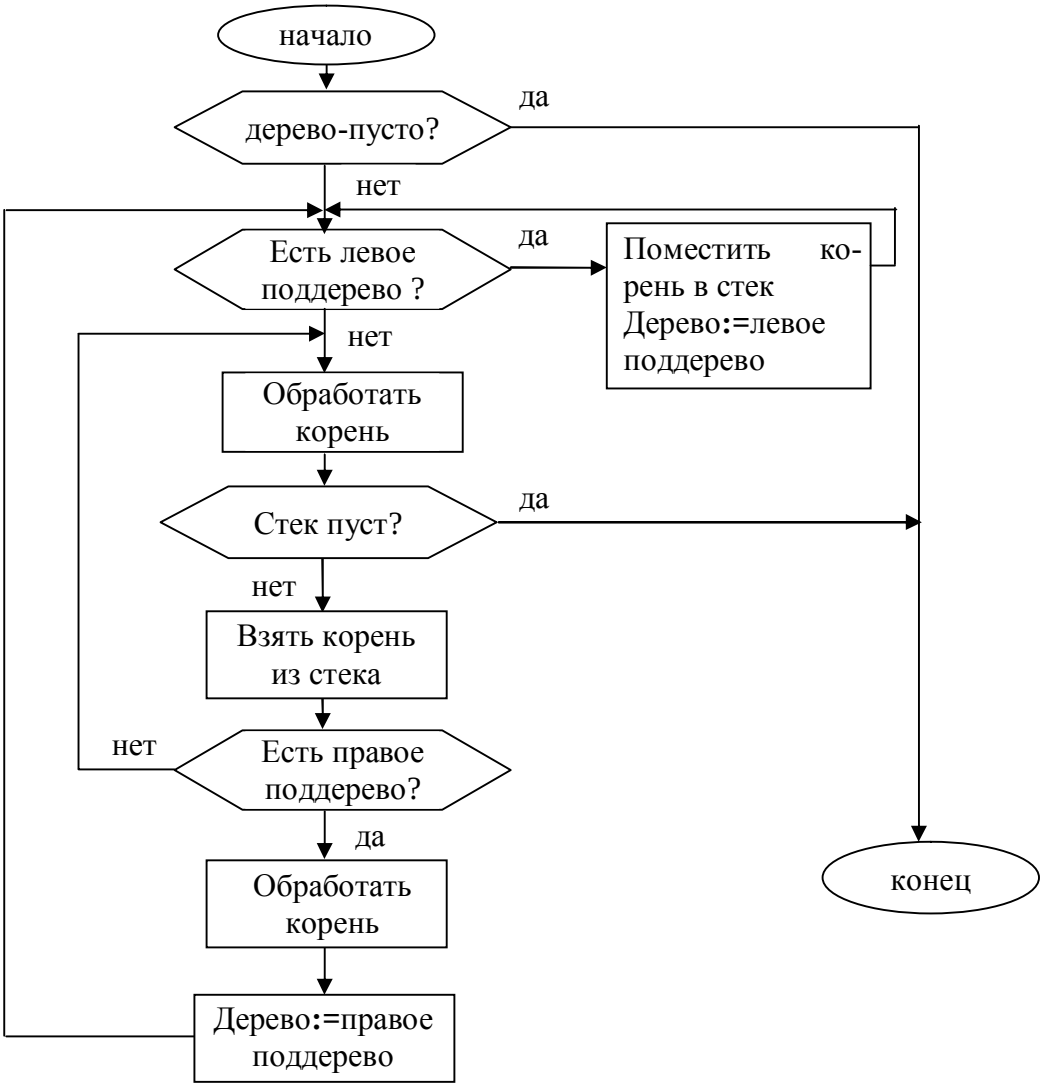


Рис. 7.23. Алгоритм обхода двоичного дерева слева

7.3.1. Представление в памяти компьютера динамических структур.

Структура памяти компьютера нами была рассмотрена в главе II. Она состоит из ячеек, называемых байтами. Каждый байт имеет свой номер или иначе адрес. Нумерация идет от 0 до N-1. Где N объем оперативной памяти. Если мы хотим записать в память компьютера значение некоторой переменной, то мы не указываем конкретный адрес, куда должно быть записано значение переменной. Мы просто даем этой переменной имя, т.е. обозначаем ее, например `A`. Компилятор отведет этой переменной нужное количество байтов. Каждая пе-

ременная в памяти занимает в зависимости от ее типа определенное количество байтов, расположенных подряд. При этом, адресом переменной считается адрес ее первого байта. Таким образом, под `A` мы подразумеваем не само значение переменной, а ее адрес (адрес первого байта). Хотя в программе мы пишем, например `A:=28;` на самом деле мы говорим компьютеру: запиши в байты памяти начиная с номера `A` число 28. Переменной `A` компилятор отведет четыре подряд расположенных байта, поскольку число 28 является целым.

Если мы хотим представить в памяти некоторый массив, то под этот массив компилятор также отведет группу подряд расположенных байтов. Пусть этому массиву присвоено имя `M`, количество элементов массива 20, тип массива – целый, т.е. каждый его элемент это целые числа.

Тогда компилятор отведет под этот массив 20*4=80 байтов. Причем для обращения к отдельным элементам массива используется индекс `M[1]`, `M[I],B[I+K]`, т.е. в этом случае адрес состоит из двух частей: одна часть указывает на начало всей совокупности, отводимой для массива, а другая – адрес байта относительно начала совокупности. Обычно адрес начала совокупности называют базовым адресом, а адрес байта относительно начала совокупности – смещением.

Таким образом:

```
Полный адрес = базовый адрес + смещение
```

Такая совокупность называется вектором памяти. В языке `Pascal` массивы (как одномерные, так и многомерные) представляются векторами.

Рис. 7.24 Схема распределения памяти переменной A и массиву M

Но можно использовать и другой способ представления информации.

Как известно, последовательности байтов памяти мы можем интерпретировать как угодно, в частности их можно интерпретировать как адрес некоторой другой совокупности байтов. Например, значение переменной A:

<A> = 2000← можно считать адресом

Переменная, значением которой является некоторый адрес памяти, называется указателем, а адрес, на который указывает указатель, называется звеном. Если имеется некоторая последовательность указателей и звеньев, то ее называют цепочкой или сцеплением звеньев.

Рассмотрим простейшую структуру сцепления – связанный список. Представим, например, с помощью связанного списка строку символов "МАМА".

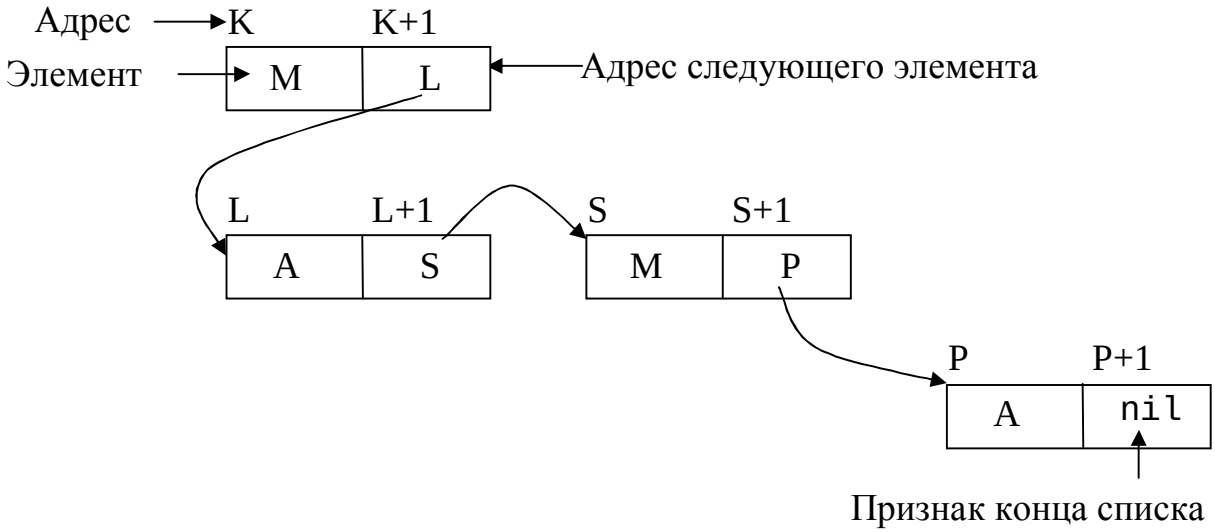


Рис. 7.25 Связанный список

Как видно из рисунка связанный список состоит из двух полей. Первое поле содержит непосредственно сам элемент списка. Второе поле – указатель на следующий элемент списка. Причем поле указателя последнего элемента списка содержит специальный признак конца списка – `nil`.

Тип `nil` означает "пустой" адрес, т.е. этот адрес не может быть адресом ни одной переменной, он является "ничейным", фиктивным адресом. В списках `nil` обозначает конец списка.

Связанные списки еще называют линейными списками.

Стек и деревья можно реализовать как с помощью векторов памяти (массивов), так и с помощью линейных списков, используя указатели.

7.3.2. Реализация стека с помощью массивов

Напишем программу работы со стеком. Существует всего две операции со стеком – поместить элемент в стек и извлечь элемент из стека. Используются также термины втолкнуть (затолкнуть) элемент в стек и вытолкнуть элемент из стека. Процедуру, реализующую операцию помещения элемента в стек назовем `Put_Stack`, а процедуру, реализующую процесс извлечения элемента из стека – `Take_Stack`. Вполне можно было бы назвать эти процедуры и Push_Stack,

Pop_Stack. Сначала реализуем стек с помощью массива, рисунок 7.26.

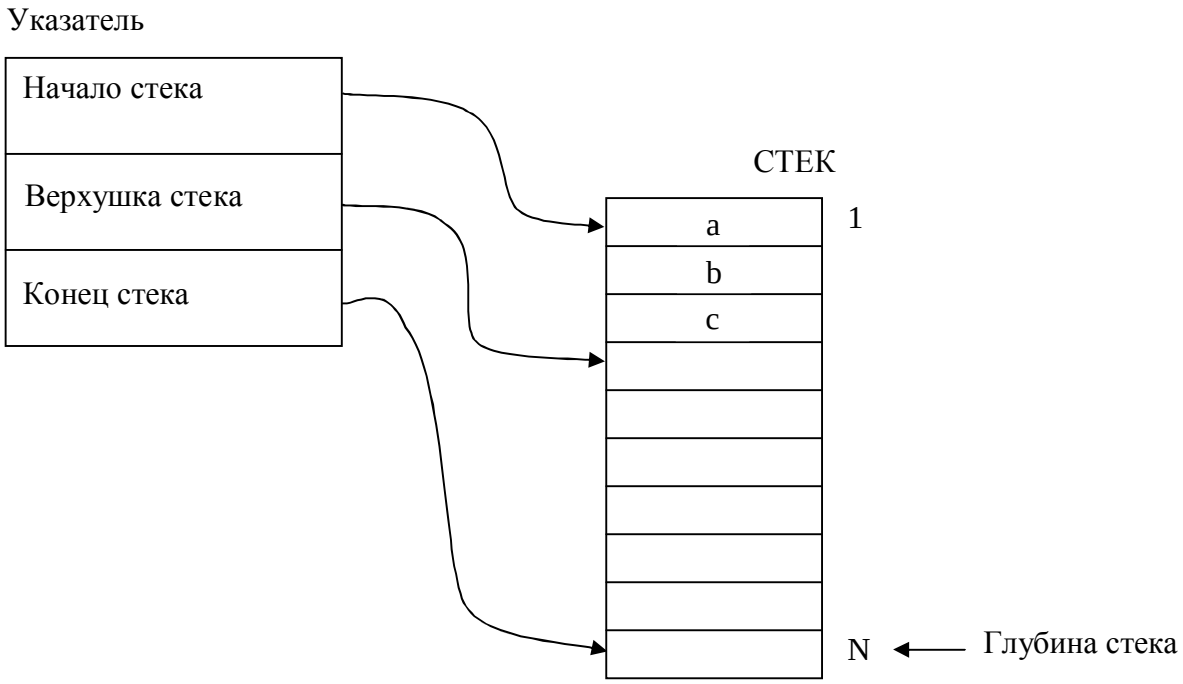


Рис. 7.26. Реализация стека на основе массивов.

Из рисунка видно, что для реализации стека нам нужны два массива, массив-указатель, назовем его `ukaz[1..3]` и массив-тело стека, назовем его `СТЕК`. Для определенности пусть он состоит из 100 элементов, т.е. глубина стека равна 100 элементам. Пусть, также для определенности, элементами стека являются просто целые числа. На практике элементами стека могут являться и более сложные структуры, например, записи. В массиве `ukaz` первый элемент, т.е. `ukaz[1]` содержит индекс массива `СТЕК`, равный началу стека, обычно это 1. Элемент `ukaz[3]` содержит индекс массива `СТЕК`, равный концу стека, т.е. заданной глубине стека N = 100 и обычно совпадает с максимальным размером массива `СТЕК`. Элемент массива `ukaz[2]` – это индекс в массиве `СТЕК`, куда будет помещен новый элемент стека, а `ukaz[2]-1` это индекс, по которому из `СТЕК` можно будет взять последний помещенный в него элемент.

Программа будет выводить на экран меню, с помощью которого можно помещать новый элемент в стек, брать элемент из верхушки стека и просматри-

вать весь массив стека.

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
const
  SizeOfArray = 100;
type
  c= array[1..SizeOfArray] of integer;
  u= array[1..3] of integer;
var
  choose: integer;
  EMPTY: boolean;
  ERROR: boolean;
  CTEK: c;
  ukaz: u;
  ELEM: integer;
{ ============================================= }
procedure Put_Stack(var ELEM: integer; var CTEK: c;
                    var ukaz: u; var ERROR: Boolean);
{ ============================================= }

var k: integer;
begin
  if ukaz[2] > ukaz[3] then
      ERROR:= true
  else
  begin
```

```
    k:= ukaz[2];
    CTEK[k]:= ELEM;
    ukaz[2]:= k + 1;
  end;
end;
{ ============================================= }
procedure Take_Stack(var ELEM: integer; var CTEK: c;
                     var ukaz: u; var EMPTY: Boolean);
{ ============================================= }
var k: integer;
begin
  if ukaz[2] <= ukaz[1] then
    EMPTY:= true
  else
  begin
    k:= ukaz[2] - 1;
    ELEM:= CTEK[k];
    ukaz[2]:= k;
  end;
end;
procedure View_Stack(var CTEK: c; var ukaz: u);
var
  i: integer;
begin
  writeln(UTF8ToConsole('Массив стека'));
  for i:= 1 to ukaz[2] - 1 do
    write(CTEK[i], ' ');
  writeln;
end;
```

```
begin
  {Инициализация стека}
  ukaz[1]:= 1;
  ukaz[2]:= 1;
  ukaz[3]:= SizeOfArray;
  repeat
    EMPTY:= false;
    ERROR:= false;
    writeln(UTF8ToConsole('Выберите нужный режим работы :'));
    writeln(UTF8ToConsole('Поместить элемент в стек        1'));
    writeln(UTF8ToConsole('Взять элемент из стека          2'));
    writeln(UTF8ToConsole('Просмотреть содержимое стека  3'));
    writeln(UTF8ToConsole('Выход из программы              4'));
    readln(choose);
    case choose of
    1: begin
         writeln(UTF8ToConsole('Введите новый элемент стека'));
         readln(ELEM);
         Put_Stack(ELEM, CTEK, ukaz, ERROR);
         if ERROR then
         begin
           writeln(UTF8ToConsole('Переполнение стека.'));
           writeln(UTF8ToConsole('Увеличьте размер массива'));
           writeln(UTF8ToConsole('Нажмите любую клавишу'));
           readkey;
         end;
       end;
    2: begin
         Take_Stack(ELEM, CTEK, ukaz, EMPTY);
```

```
        if EMPTY then
          writeln(UTF8ToConsole('Стек пуст'))
        else
          writeln(UTF8ToConsole('Из стека взят элемент '),
                  ELEM);
      end;
    3: View_Stack(CTEK, ukaz);
    end; { end of case }
  until choose = 4;
end.
```

Процедура `Put_Stack` просто помещает новый элемент в массив СТЕК по индексу `ukaz[2]` и затем увеличивает значение этого индекса на единицу. Перед этим процедура проверяет стек на переполнение, т.е. не перешли ли мы за пределы массива `СТЕК`.

Процедура `Take_Stack` возвращает элемент массива `СТЕК` с индексом `ukaz[2]-1` и уменьшает значение индекса на единицу. Перед этим она проверяет не пуст ли уже стек.

Если внимательно присмотреться к процедурам `Put_Stack` и `Take_Stack`, то возникает вполне резонный вопрос, а зачем, собственно говоря, здесь нужен массив-указатель `ukaz`? И мы будем абсолютно правы. При реализации стека вполне можно обходиться одним массивом, только для размещения элементов стека. А в качестве указателя использовать простую целочисленную переменную, которую назовем также `ukaz`. Вот листинг другой, улучшенной реализации стека:

```
program modify_stack;
{$mode objfpc}{$H+}
```

```
uses
  CRT, FileUtil;
const
  SizeOfArray = 100;
type
  c= array[1..SizeOfArray] of integer;

var
  choose: integer;
  EMPTY: boolean;
  ERROR: boolean;
  CTEK: c;
  ukaz: integer;
  ELEM: integer;
{ ================================================== }
procedure Put_Stack(var ELEM: integer; var CTEK: c;
                var ukaz: integer; var ERROR: Boolean);
{ ================================================== }

begin
  if ukaz > SizeOfArray then
      ERROR:= true
  else
  begin
    CTEK[ukaz]:= ELEM;
    ukaz:= ukaz + 1;
  end;
end;
```

```
{ ================================================== }
procedure Take_Stack(var ELEM: integer; var CTEK: c;
               var ukaz: integer; var EMPTY: Boolean);
{ ================================================== }

begin
  if ukaz = 1 then
    EMPTY:= true
  else
  begin
    ukaz:= ukaz - 1;
    ELEM:= CTEK[ukaz];
  end;
end;
procedure View_Stack(var CTEK: c; var ukaz: integer);
var
  i: integer;
begin
  writeln(UTF8ToConsole('Массив стека'));
  for i:= 1 to ukaz - 1 do
    write(CTEK[i], ' ');
  writeln;
end;

begin
  {Инициализация указателя стека (начального индекса в массиве)}
  ukaz:= 1;
  repeat
    EMPTY:= false;
```

```
    ERROR:= false;
    writeln(UTF8ToConsole('Выберите нужный режим работы :'));
    writeln(UTF8ToConsole('Поместить элемент в стек        1'));
    writeln(UTF8ToConsole('Взять элемент из стека          2'));
    writeln(UTF8ToConsole('Просмотреть содержимое стека  3'));
    writeln(UTF8ToConsole('Выход из программы              4'));
    readln(choose);
    case choose of
    1: begin
         writeln(UTF8ToConsole('Введите новый элемент стека'));
         readln(ELEM);
         Put_Stack(ELEM, CTEK, ukaz, ERROR);
         if ERROR then
         begin
           writeln(UTF8ToConsole('Переполнение стека.'));
           writeln(UTF8ToConsole('Увеличьте размер массива'));
           writeln(UTF8ToConsole('Нажмите любую клавишу'));
           readkey;
         end;
       end;
    2: begin
         Take_Stack(ELEM, CTEK, ukaz, EMPTY);
         if EMPTY then
           writeln(UTF8ToConsole('Стек пуст'))
         else
           writeln(UTF8ToConsole('Из стека взят элемент '),
ELEM);
       end;
    3: View_Stack(CTEK, ukaz);
```

```
    end; { end of case }
  until choose = 4;
end.
```

В этой реализации переменная ukaz является индексом (указателем) верхушки стека и одновременно показывает количество элементов в стеке, т.е. определяет фактический размер массива. Для того чтобы поместить элемент в стек, мы записываем его в массив СТЕК по индексу ukaz и увеличиваем ukaz на единицу. Для того чтобы взять элемент из стека процедура Take_Stack должна уменьшить значение ukaz на единицу и возвратить элемент с индексом ukaz. Таким образом, указатель (индекс в массиве) расположен так, как показано на рисунке 7.26.

Существенным недостатком представления стека векторным способом является необходимость задания максимальной глубины стека, поскольку не всегда известна точная глубина стека. Отсюда необходимость задания глубины стека с "запасом", т.е. память используется нерационально. Достоинство – относительная простота работы со стеком.

7.3.3. Представление двоичного дерева в виде массива и реализация алгоритма обхода двоичного дерева слева.

Деревья наиболее важные структуры, используемые в программировании. Поэтому крайне необходимо освоить технику работы с ними. Здесь мы рассмотрим представление дерева с помощью массива.

Пусть имеется дерево, рис. 7.27.

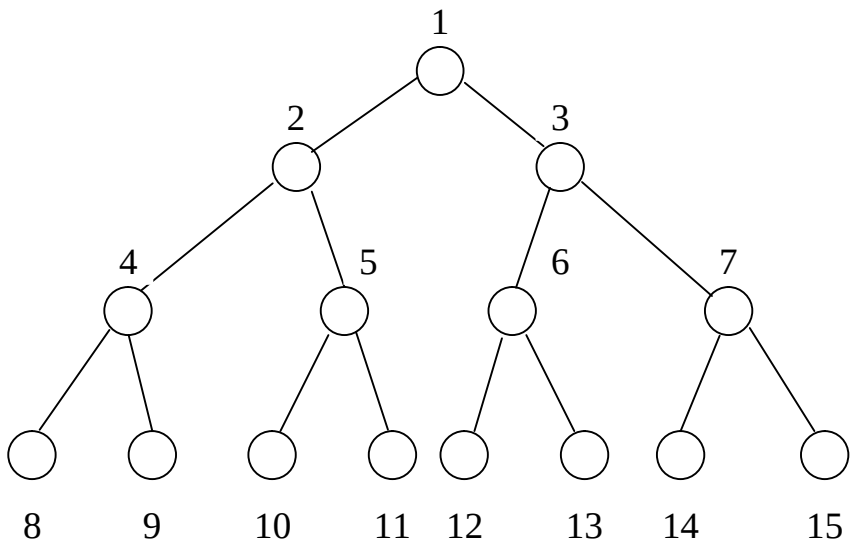


Рис. 7.27 Двоичное дерево. В качестве вершин выступают целые числа

В этом дереве вершинами являются просто целые числа, хотя в качестве вершин могут выступать любые объекты. Представим наше дерево в виде, рис. 7.28. Отсюда возникает идея хранить в массиве не только саму вершину, но и индексы левого и правого поддерева. Таким образом, каждый элемент массива представляет собой тройку чисел – корень и индексы в массиве левого и правого поддерева. Если левое или правое поддеревья пустые, то индекс равен -1 (аналог `nil`).

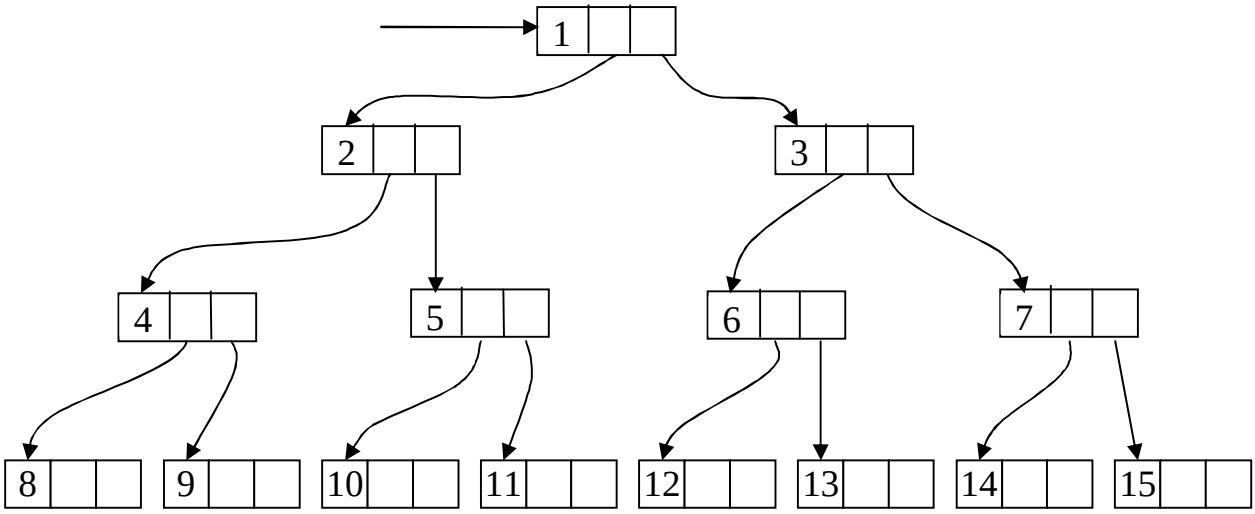


Рис. 7.28 Схема представления двоичного дерева с помощью массива

Внимательно присмотритесь к рисунку. Фактически это похоже на связанный список, только вместо указателей мы используем индексы массива. Представление этого же дерева с помощью "настоящего" связанного списка и указа-

телей мы рассмотрим в следующем разделе. Для того чтобы записать индексы на левые и правые поддеревья в массиве поступим следующим образом: если вершина – корень имеет номер индекса *n*, то левое и правое поддерево – соответственно 2**n* и 2**n*+1. Нумерация начинается с *n*=1. Ячейка со значением -1 обозначает отсутствие вершины. Сначала сделаем вручную распределение индексов в массиве для нашего дерева, табл. 7.1.

Таблица 7.1

индекс	корень	левое поддерево (индекс в массиве)	правое поддерево (индекс в массиве)
1	1	2	3
2	2	4	5
3	3	6	7
4	4	8	9
5	5	10	11
6	6	12	13
7	7	14	15
8	8	-1	-1
9	9	-1	-1
10	10	-1	-1
11	11	-1	-1
12	12	-1	-1
13	13	-1	-1
14	14	-1	-1
15	15	-1	-1
16	-1	-1	-1

Визуально массив, реализующий наше исходное двоичное дерево можно представить следующим образом, рис. 7.29.

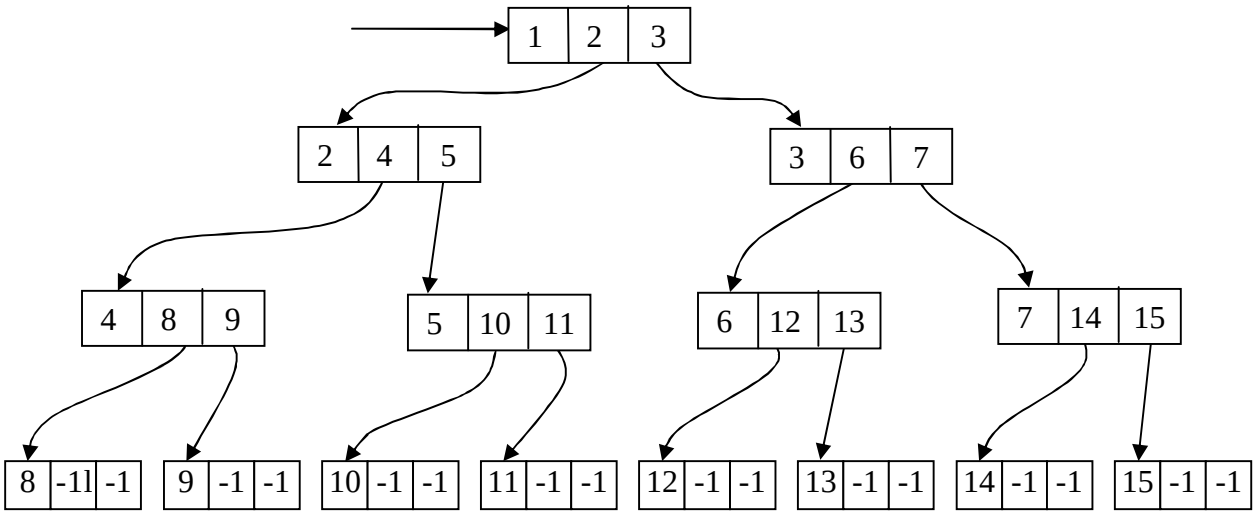


Рис. 7.29 Окончательная схема представления двоичного дерева с помощью массива

При обходе этого двоичного дерева слева последовательность обработки вершин будет следующей:

8, 4, 9, 2, 10, 5, 11, 1, 12, 6, 13, 3, 14, 7, 15

Блок схему алгоритма обхода мы с вами уже рассматривали. Последовательность состояний стека для данного алгоритма будет следующей, рис. 7.30:

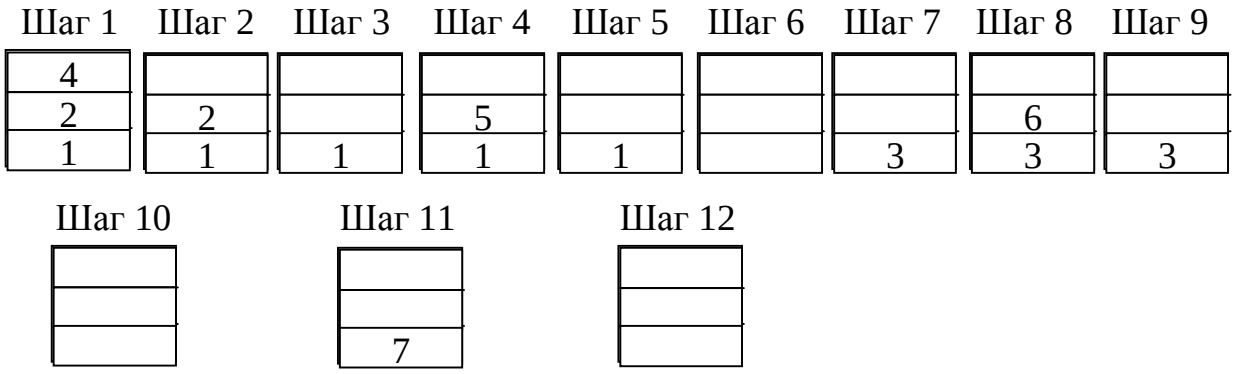


Рис. 7.30 Состояния стека при обходе двоичного дерева слева

Напишем программу обхода двоичного дерева слева. Сначала отладим программу для уже готового дерева. Далее мы научимся организовывать ввод дерева в диалоговом режиме с клавиатуры. Для этого подготовьте в текстовом редакторе "Блокнот" (NotePad) файл представления двоичного дерева в виде

массива приведенного в таблице 7.1 с именем "derevo.dat" и разместите его в папке с проектом. Можете взять готовый файл из прилагаемого к книге DVD диска. В соответствии со структурой массива будем использовать запись вида:

```
type
  T = record
  TElem: integer;
  TLeft: integer;
  TRight: integer;
  end;
```

Где `T` – элемент массива типа запись, `TElem` вершина дерева, `TLeft` индекс в массиве левого поддерева, `TRight` индекс в массиве правого поддерева.

Чтобы не загромождать текст основной программы, процедуры `Put_Stack` и `Take_Stack` оформим в виде отдельного модуля STACK.

Текст модуля:

```
unit STACK;
interface
const
  SizeOfArrayCTEK= 100;
  SizeOfArrayDER= 100;
type
  T = record
  TElem: integer;
  TLeft: integer;
  TRight: integer;
end;
```

```
procedure Put_Stack(var ELEM: integer;
                    var CTEK: array of T;
                    var ukaz: integer;
                    var ERROR: Boolean);

procedure Take_Stack(var ELEM: integer;
                     var CTEK: array of T;
                     var ukaz: integer;
                     var EMPTY: Boolean);

implementation
{ ========================================= }
procedure Put_Stack(var ELEM: integer;
                    var CTEK: array of T;
                    var ukaz: integer;
                    var ERROR: Boolean);
{ ========================================= }

begin
  if ukaz > SizeOfArrayCTEK then
      ERROR:= true
  else
  begin
    CTEK[ukaz].TElem:= ELEM;
    ukaz:= ukaz + 1;
  end;
end;
{ ======================================= }
procedure Take_Stack(var ELEM: integer;
```

```
                    var CTEK: array of T;
                    var ukaz: integer;
                    var EMPTY: Boolean);
{ ======================================== }
begin
  if ukaz = 1 then
    EMPTY:= true
  else
  begin
    ukaz:= ukaz - 1;
    ELEM:= CTEK[ukaz].TElem;
  end;
end;
end.
```

Листинг программы обхода двоичного дерева слева:

```
program Derevo_Left;
{$mode objfpc}{$H+}
{Процедуры работы со стеком мы оформили в виде модуля!}
uses
  CRT, FileUtil, STACK;
var
  CTEK:array [1..SizeOfArrayCTEK] of T;
  DER:array[1..SizeOfArrayDER] of T;
  ukaz: integer;
  i ,ELEM: integer;
  ERROR, EMPTY: Boolean;
  fder: TextFile;
```

```
begin
 {Инициализация указателя стека (начального индекса в массиве)}
  ukaz:= 1;
  {Чтение из файла дерева}
  Assign(fder, 'derevo.dat');
  Reset(fder);
  i:= 1;
  while not Eof(fder) do
  begin
    Read(fder, DER[i].TElem);
    Read(fder, DER[i].TLeft);
    Read(fder, DER[i].TRight);
    inc(i);
  end;
  Close(fder);
  ERROR:= false;
  EMPTY:= false;
  i:= 1;
  writeln(UTF8ToConsole('Обход  двоичного дерева слева'));
  while DER[i].TElem <> -1 do
  begin
    if DER[i].TLeft <> -1 then
    begin
      ELEM:= i;
      Put_Stack(ELEM, CTEK, ukaz, ERROR);
      if ERROR = true then
      begin
        writeln(UTF8ToConsole('Ошибка! Переполнение стека.'));
        writeln(UTF8ToConsole('Увеличьте размер массива'));
```

```
        writeln(UTF8ToConsole('Нажмите любую клавишу'));
        readkey;
        exit;
      end;
      i:= DER[i].TLeft;
    end
    else
    begin
      repeat
        writeln(DER[i].TElem);
        Take_Stack(ELEM,CTEK,ukaz,EMPTY);
        if EMPTY = true  then break;
        i:= ELEM;
      until DER[i].TRight <> -1;
      if EMPTY = true then break;
      writeln(DER[i].TElem);
      i:= DER[i].TRight;
    end;
  end;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Теперь напишем программу, где ввод дерева производится с клавиатуры.

```
program Derevo_Left;
{$mode objfpc}{$H+}
{Процедуры работы со стеком мы оформили в виде модуля!}
uses
```

```
  CRT, FileUtil, STACK;
var
  CTEK: array [1..SizeOfArrayCTEK] of T;
  DER: array[1..SizeOfArrayDER] of T;
  ukaz: integer;
  i,ELEM:integer;
  ERROR, EMPTY: Boolean;
  k: integer;
  answ: char;
begin
  {Ввод дерева}
  writeln(UTF8ToConsole('Вводите дерево строго'));
  writeln(UTF8ToConsole('сверху вниз и слева направо'));
  writeln(UTF8ToConsole('Чтобы закончить ввод введите -1'));
  i:= 1;
  while true do
  begin
    writeln(UTF8ToConsole('Введите корень.'));
    {$i-} //  отключение стандартного режима контроля ввода
    repeat
      ERROR:= false;
      readln(k);
      ERROR:= (IoResult <> 0);
      if ERROR then   {если ERROR=true, значит произошла ошибка при
вводе}
      writeln(UTF8ToConsole('Ошибка! Введите номер вершины'));
    until not ERROR;
    {$+} //  восстановление стандартного режима контроля ввода/вывода
    DER[i].TElem:= k;
```

```
    if k = -1 then
    begin
      DER[i].TLeft:= -1;
      DER[i].TRight:= -1;
      break;
    end;
    writeln(UTF8ToConsole('Текущая вершина имеет поддеревья?'));
    writeln(UTF8ToConsole('Ответ (y/n)'));
    repeat
      readln(answ);
      if (answ = 'y') then
      begin
        DER[i].TLeft:= 2 * i;
        DER[i].TRight:= 2 * i + 1;
      end
      else
      begin
        DER[i].TLeft:= -1;
        DER[i].TRight:= -1;
      end;
    until (answ = 'n') or (answ = 'y');
    inc(i);
  end;
{Обход заданного двоичного дерева слева}
{Инициализация указателя стека (начального индекса в массиве)}
  ukaz:= 1;
  ERROR:=false;
  EMPTY:=false;
  i:= 1;
```

```
writeln(UTF8ToConsole('Обход  двоичного дерева слева'));
while DER[i].TElem <> -1 do
begin
  if DER[i].TLeft <> -1 then
  begin
    ELEM:= i;
    Put_Stack(ELEM,CTEK,ukaz,ERROR);
    if ERROR = true then
    begin
      writeln(UTF8ToConsole('Ошибка! Переполнение стека.'));
      writeln(UTF8ToConsole('Увеличьте размер массива'));
      writeln(UTF8ToConsole('Нажмите любую клавишу'));
      readkey; exit;
    end;
    i:= DER[i].TLeft;
  end
  else
  begin
    repeat
      writeln(DER[i].TElem);
      Take_Stack(ELEM, CTEK, ukaz, EMPTY);
      if EMPTY = true  then break;
      i:= ELEM;
    until DER[i].TRight <> -1;
    if EMPTY=true then break;
    writeln(DER[i].TElem);
    i:= DER[i].TRight;
  end;
end;
```

```
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Намучились с вводом дерева? Действительно, муторное это занятие. К счастью, в реальных задачах деревья никто и никогда не вводит. Они формируются программно в зависимости от задачи. Пример автоматического создания дерева мы рассмотрим в разделе сортировка и поиск с помощью двоичного дерева.

Теперь попробуйте самостоятельно реализовать другие способы обхода двоичного дерева, т.е. обход сверху и снизу.

Достоинства представления деревьев в виде массивов – относительная простота реализации. Для небольших деревьев, особенно если количество вершин заранее известно, использование массивов может оказаться более эффективным решением. Но отсюда вытекает и недостаток, если количество вершин дерева заранее неизвестно, то, как и в случае со стеком, приходится резервировать память по максимуму. А каков этот максимум? Определенных критериев нет. Память расходуется непродуктивно. Еще один существенный недостаток – порядок расположения поддеревьев в массиве не поддается формализации.

Паскаль предоставляет значительно более удобный механизм для представления списков и, следовательно, для реализации стека и деревьев. Рассмотрим этот механизм.

7.3.4. Указатели

Существует ряд задач, где статические структуры данных неэффективны для реализации алгоритма, поэтому в Паскале предусмотрена возможность работы с динамическими типами структур. С некоторыми из них, в частности

стеками и связанными списками, мы познакомились в предыдущих разделах. Также мы увидели, что стек можно реализовать, используя массивы.

Эффективным средством построения связанных списков являются указатели.

Об указателях мы уже вели разговор в разделе 6.2.1.6, где шла речь о типе данных – указатель.

Напомним синтаксис объявления типизированных указателей:

Напомним синтаксис объявления типизированных указателей:

```
type
   Pint = ^integer;
var
  p: ^integer;  {указатель на переменную целого типа}
  p1: ^ string;  {указатель на строку символов}
  p2: Pint;
```

Нетипизированный указатель описывается следующим образом:

```
var
  ptr: pointer;   {нетипизированный указатель}
```

Описание указателей `p` и `p2` эквивалентны, т.е. можно применять тот и другой способ.

Указатель фактически является адресом памяти, по которому осуществляется доступ к значениям динамической переменной.

Над указателями допустимы операции проверки на равенство (=) и неравенство (<>), а также возможно выполнение оператора присваивания := . Если список пуст, то значение переменной-указателя равно `nil`.

Память для динамических структур, реализуемых при помощи указателей,

распределяется в специальной области оперативной памяти компьютера и называется "кучей" (от англ. heap – куча) или динамически распределяемой памятью (ДРП).

Для работы с динамическими переменными в Паскале предусмотрены процедуры:

`New`(*p*) – выделить в динамически распределяемой памяти (ДРП) необходимую память для переменной заданного типа с типизированным указателем *p*;

`Dispose`(*p*) – освободить участок ДРП, занятый переменной с типизированным указателем p.

`Getmem`(*p*, `size`) – второй способ выделить память переменной с указателем *p* размером `size` байтов. Этой процедурой можно выделить память как для типизированных, так и для нетипизированных указателей.

`Freemem`(*p*, `size`)) – освободить память распределенной переменной с указателем *p* размером `size` байтов.

Пример:

```
New(p);{Выделяется память для переменной целого типа}
New(p1); {Выделяется память для строки символов}
Getmem(p2, 1000); {Выделяется память размером 1000 байт для переменной с указателем p2}
```

Как вы видите, динамические переменные не имеют собственного имени и доступ к значению такой переменной осуществляется через операцию *разыменования* указателя. Для этого используется тот же символ "^", что использовался при описании указателей.

Пример:

```
p^:= 25;
p1^:= 'Это строка символов';
p2^:= 44;
```

При выделении памяти процедурой `Getmem` для типизированных указате-

лей и определении размера выделяемой памяти, лучше всего использовать процедуру `SizeOf` так как, размеры памяти, отводимой под тот или иной тип, могут разниться в различных версиях компиляторов, например:

```
Getmem(p2, SizeOf(integer));
```

Обязательно освобождайте память после использования! Имейте в виду только, что после освобождения памяти значения указателей не изменятся, что может привести к ошибкам. Необходимо присваивать значениям указателей `nil` после освобождения памяти. Кроме того, нельзя смешивать методы выделения и освобождения памяти. Так, если память была выделена процедурой `New()`, то нельзя освобождать память процедурой `Freemem()` и, наоборот, если память была выделена процедурой `Getmem()`, то нельзя освобождать процедурой `Dispose()`.

Теперь давайте построим связанный список, используя указатели. Для этого достаточно определить тип данных – запись и указатель на него. Структура записи будет следующей:

```
type
  PMyList = ^TMyList;
  TMyList = record
  data: integer;
  next: PMyList;
  end;
var
  mylist: PMyList;
```

В поле `data` находится содержательная часть элемента списка, причем `data` может быть, в свою очередь, записью достаточно сложной структуры.

Поле `next` – указатель. В нем содержится ссылка на следующий элемент списка. Благодаря этому указателю можно передвигаться к следующему элементу списка. В качестве признака окончания списка используют пустой указатель `nil`. Если необходимо двигаться по списку в прямом и обратном направлении, то в структуру записи добавляют еще одно поле – указатель, в котором указывают адрес предыдущего элемента списка. Такие списки с двумя указателями называются двусвязными или двунаправленными.

Если в конце списка вместо пустого указателя `nil` поставить ссылку на начало списка, т.е. на первый элемент списка, то получим так называемый кольцевой список.

Для всех видов списков можно определить стандартные операции, которые производятся над списками и которые должен уметь реализовывать программист. К таким операциям относятся добавление нового элемента в список, удаление элемента из списка, поиск нужного элемента в списке. Рассмотрим их.

7.3.5. Стандартные операции с линейными списками

Запишем стандартные операции с линейными списками в виде готовых процедур и функций.

Процедура добавления (вставки) нового элемента в список:

```
procedure Insert_MyList(Elem: integer;
                        var pHead, pCurrent: PMyList);
{pHead - указатель на первый элемент списка ("голову" списка)
 pCurrent - указатель на текущий элемент списка}
var
  p: PMyList; // рабочий указатель
begin
```

```
  new(p);  // заводим новый элемент
  p^.data:=  Elem; // записываем в него данные
  {Если список был пустой, то головой списка становится  p}
  if pHead = nil then
  begin
    p^.next:= nil;
    pHead:= p;
  end
  else
  begin
    p^.next:= pCurrent^.next; {  в новом элементе делаем ссылку
    на следующий элемент, который был до вставки}
    pCurrent^.next:= p; {в текущем элементе делаем
    ссылку на новый}
  end;
  pCurrent:= p; //  текущим становится новый элемент
end;
```

Функция поиска элемента в списке просто проходит все элементы списка по ссылкам от начала до тех пор, пока не будет найден искомый элемент.

```
function Search_MyList(Elem: integer;
                       var pHead: PMyList): boolean;
var
  p: PMyList;
begin
  if pHead <> nil then
    p:= pHead
  else
```

```
  begin
    writeln(UTF8ToConsole('Список пуст'));
    exit;
  end;
  Search_MyList:= false;
  while true do
  begin
    if p^.data = Elem then
    begin
      Search_MyList:= true;
      break;
    end
    else
    if p^.next = nil then break;
    p:= p^.next;
  end;
end;
```

Функция возвращает `true`, если требуемый элемент найден.

Процедура удаления элемента из списка:

```
procedure Delete_MyList(Elem: integer;
                        var pHead, pCurrent: PMyList);
var
  p: PMyList;
  pPrev: PMyList; // предыдущий элемент списка
  find: boolean;
begin
   if pHead <> nil then
```

```
  p:= pHead
else
begin
  writeln(UTF8ToConsole('Список пуст'));
  exit;
end;
find:= false;
while true do
begin
  if p^.data = Elem then
  begin
    find:= true;
    break;
  end
  else
  if p^.next = nil  then break;
  pPrev:= p;
  p:= p^.next;
end;
if find then
begin
  if p = pHead then // если удаляется первый элемент
  begin
    p:= pHead^.next; // запоминаем ссылку на следующий элемент
    Dispose(pHead);  // удаляем первый элемент списка
    pHead:= p;       // головой списка становится p
  end
  else
  begin  // если удаляется не первый элемент
```

```
      pPrev^.next:= p^.next; { в предыдущем элементе заменяем
      ссылку на следующий элемент после удаляемого }
      Dispose(p);
      pCurrent:= pPrev; // текущим делаем предыдущий элемент
    end;
    writeln(UTF8ToConsole('Элемент успешно удален'));
  end
  else writeln(UTF8ToConsole('Элемент не найден'));
end;
```

Процедура, прежде чем удалить элемент, должна его найти. Эта часть совпадает с функцией поиска. Поскольку код функции поиска небольшой, мы вставили в процедуру непосредственно сам код. В принципе, в процедуре можно вызвать функцию поиска. Для этого необходимо видоизменить функцию таким образом, чтобы она возвращала ссылку на найденный элемент. Если элемента нет, она должна возвращать *nil*. Предоставляем изменить функцию поиска самому читателю.

Напишем главную программу работы со списком (не забудьте включить в программу процедуры и функции, которые мы только что разобрали):

```
program operation_list;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type
  PMyList = ^TMyList;
  TMyList = record
  data: integer;
  next: PMyList;
```

```
  end;
var
  pHead, pCurrent: PMyList;
  Elem, choose: integer;
{Сюда добавьте процедуры и функции, реализующие
 стандартные операции с линейными списками }
begin
  pHead:= nil;
  pCurrent:= nil;
  repeat
    writeln(UTF8ToConsole('Выберите нужное действие:'));
    writeln(UTF8ToConsole('1-ввод элемента списка'));
    writeln(UTF8ToConsole('2-поиск элемента списка'));
    writeln(UTF8ToConsole('3-удаление элемента списка'));
    writeln(UTF8ToConsole('4-просмотр всего списка'));
    writeln(UTF8ToConsole('5-выход из программы'));
    readln(choose);
    case choose of
    1: begin {ввод элемента списка}
         writeln(UTF8ToConsole('Введите элемент списка'));
         readln(Elem);
         Insert_MyList(Elem, pHead, pCurrent);
       end;
    2: begin {поиск элемента списка}
         writeln(UTF8ToConsole('введите искомый элемент'));
         readln(Elem);
         if Search_MyList(Elem, pHead)then
           writeln(UTF8ToConsole('Элемент найден'))
```

```
        else
          writeln(UTF8ToConsole('Элемент не найден'));
      end;
    3: begin {удаление элемента списка}
        writeln(UTF8ToConsole('введите элемент'));
        readln(Elem);
        Delete_MyList(Elem, pHead, pCurrent);
      end;
    4: begin {Вывод списка на экран}
        writeln(UTF8ToConsole('Элементы списка:'));
        writeln;
        view_MyList (pHead);
        writeln;
      end;
    end; { end of case }
  until choose = 5;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Кроме того, часто встречаются особые списки, где добавление или удаление элементов производится только в начале или в конце списка. Какие это списки:

- линейный список, в котором добавление или удаление элементов производится только в одном конце. Читатель, наверное, уже догадался, что это есть не что иное, как стек!
- линейный список, в котором добавление элементов производится на одном конце, а удаление на противоположном конце списка. Если внимательно подумать, то работа такого списка будет напоминать живую оче-

редь людей в магазине. Поэтому такой список так и называют – очередь. Часто очередь называют структурой FIFO (First In - First Out, т.е. "первый пришел, первый ушел").

- линейный список, в котором все добавления и удаления производятся с обоих концов списка. Такой список носит название дек.

Реализуем рассмотренные динамические структуры с помощью линейных списков.

7.3.6. Реализация динамических структур линейными списками

7.3.6.1. Реализация стека

Для того чтобы поместить элемент в стек, достаточно написать следующий код:

```
New(p); // Резервируем память для нового элемента
p ^.data:= ELEM; //Записываем данные
p^.next:= CTEK;// устанавливаем ссылку на текущую вершину стека
CTEK:=  p; // Теперь новый элемент стал верхушкой стека
```

Для того чтобы взять элемент из верхушки стека, необходим код:

```
if CTEK <> nil then // Если стек не пуст
  begin
    ELEM:= CTEK^.data; //  Взять элемент
    p:= CTEK^.next; //  Указатель на следующий элемент
    Dispose(CTEK);//  Удалить текущий элемент из вершины стека
    CTEK:= p; //Предыдущий элемент сделать вершиной стека
```

Приведем полный листинг программы работы со стеком:

```
program stack;
```

```
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type
  c = ^MyStack; // указатель
  MyStack = record // структура записи
  data: integer; // содержательная часть стека
  next: c; // ссылка на следующий элемент
  end;
var
  choose: integer;
  EMPTY: boolean;
  CTEK: c; // указатель, указывает на вершину стека
  ukaz: c; // рабочий указатель
  ELEM: integer; // содержательный элемент стека
{ ================================================ }
procedure Put_Stack(var ELEM: integer; var CTEK: c);
{ ================================================ }

begin
  New(ukaz); // создаем новый элемент стека
  ukaz^.data:= ELEM; // В поле data записи записываем значение
  ukaz^.next:= CTEK; // ссылка на следующий элемент
  CTEK:= ukaz; // вершиной стека становится новый элемент
end;

{ ================================================ }
procedure Take_Stack(var ELEM: integer; var CTEK: c;
                     var EMPTY:Boolean);
```

```
{ ================================================ }
var ukaz: c;
begin
  if CTEK <> nil then
  begin
    ELEM:= CTEK^.data; // Берем элемент из вершины стека
    ukaz:= CTEK^.next; // Указатель на следующий элемент
    Dispose(CTEK); // уничтожение текущей вершины
    CTEK:= ukaz;  // вершиной стека становится следующий элемент
  end
  else EMPTY:= true;
end;
procedure View_Stack(var CTEK: c);
var ukaz: c;
begin
  writeln(UTF8ToConsole('Содержимое стека'));
  ukaz:= CTEK; // Берем вершину стека
  {В цикле проходим все элементы стека и печатаем их}
  while ukaz <> nil do
  begin
    write(ukaz^.data, ' ');
    ukaz:= ukaz^.next;
  end;
  writeln;
end;

begin
  {Инициализация стека}
  CTEK:= nil;
```

```
repeat
  EMPTY:= false;
  writeln(UTF8ToConsole('Выберите нужный режим работы :'));
  writeln(UTF8ToConsole('Поместить элемент в стек    1'));
  writeln(UTF8ToConsole('Взять элемент из стека      2'));
  writeln(UTF8ToConsole('Просмотреть содержимое стека 3'));
  writeln(UTF8ToConsole('Выход из программы        4'));
  readln(choose);
  case choose of
  1: begin
       writeln(UTF8ToConsole('Введите новый элемент стека'));
       readln(ELEM);
       Put_Stack(ELEM, CTEK);
     end;
  2: begin
       Take_Stack(ELEM, CTEK, EMPTY);
       if EMPTY then
       begin
         writeln(UTF8ToConsole('Стек пуст'));
       end
       else
       begin
         writeln(UTF8ToConsole('Из стека взят элемент '),
                 ELEM);
       end;
     end;
  3: begin
      View_Stack(CTEK);
    end;
```

```
    end; { end of case }
  until choose = 4;
end.
```

7.3.6.2.Реализация очереди с помощью линейного списка

Проще всего для этого завести два указателя. Указатель на первый элемент очереди `first` и указатель на последний элемент очереди `last`. Листинг программы:

```
program queue; // queue означает по английски очередь
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type
  PQueue = ^Element; // указатель
  Element = record // структура записи
  data : integer; // содержательная часть очереди
  next : PQueue; // ссылка на следующий элемент
  end;
  TQueue = record // структура для реализации очереди
  first: PQueue; // указатель, указывает на первый элемент очереди
  last: PQueue; // указатель, указывает на последний элемент очереди
  end;

{ ============================================= }
procedure Put_Que(ELEM: integer; var Que: TQueue);
{ ============================================= }
```

```
var ukaz: PQueue; // временный указатель
begin
  if (Que.last = nil) then
  {Если очередь пуста, то указатели на первый
   и последний элемент будут совпадать}
  begin
    {Выделяем память для нового элемента}
    Que.last:= New(PQueue);
    Que.first:= Que.last;  // Приравниваем указатели
    Que.last^.data:= ELEM; // записываем данные
    Que.last^.next:= nil; // Следующий элемент пока пустой
  end
  else
  begin
    ukaz:= New(PQueue);
    ukaz^.data:= ELEM;
    ukaz^.next:= nil;
    Que.last^.next:= ukaz; // ссылка на следующий элемент
    Que.last:= ukaz; // Сдвигаем указатель на последний элемент
  end;
end;

{ ================================================== }
procedure Take_Que(var ELEM: integer; var Que: TQueue;
                   var EMPTY:Boolean);
{ ================================================== }

var ukaz: PQueue;
begin
```

```
  if (Que.first = nil) then
  begin
    EMPTY:= true;
  end
  else
  begin
    ELEM:= Que.first^.data; {Первый элемент очереди}
    ukaz:= Que.first; // Запомним ссылку
    Que.first:=  Que.first^.next;  {переход на следующий
элемент}
    Dispose(ukaz); {уничтожение текущего элемента}
  end;
end;

{ ================================= }
procedure View_Que(var Que: TQueue);
{ ================================= }

var
  ukaz: PQueue;
begin
  writeln(UTF8ToConsole('Содержимое очереди'));
  ukaz:= Que.first; // Берем первый элемент очереди
  {В цикле проходим все элементы очереди и печатаем их}
  while ukaz <> nil do
  begin
    write(ukaz^.data, ' ');
    ukaz:= ukaz^.next;
  end;
```

```
  writeln;
end;

var
  choose: integer;
  EMPTY: boolean;
  Que: TQueue;
  ELEM: integer; // содержательный элемент очереди
begin
  {Инициализация очереди}
  Que.last:= nil;
  repeat
    EMPTY:= false;
    writeln(UTF8ToConsole('Выберите нужный режим работы :'));
    writeln(UTF8ToConsole('Поместить элемент в очередь   1'));
    writeln(UTF8ToConsole('Взять элемент из очереди      2'));
    writeln(UTF8ToConsole('Просмотреть содержимое очереди 3'));
    writeln(UTF8ToConsole('Выход из программы        4'));
    readln(choose);
    case choose of
    1: begin
         writeln(UTF8ToConsole('Введите новый элемент'));
         readln(ELEM);
         Put_Que(ELEM, Que);
       end;
    2: begin
         Take_Que(ELEM, Que, EMPTY);
         if EMPTY then
         begin
```

```
          writeln(UTF8ToConsole('Очередь пуста'));
        end
        else
        begin
          writeln(UTF8ToConsole('Из очереди взят элемент '),
                  ELEM);
        end;
      end;
    3: begin
        View_Que(Que);
      end;
    end; { end of case }
  until choose = 4;
end.
```

Реализация дека теперь для вас не представляет такой уж сложной задачей. Попробуйте реализовать его самостоятельно.

7.3.6.3. Реализация двоичного дерева с помощью линейного списка

Двоичное дерево можно представить с помощью двухсвязного списка следующим образом:

```
type
  PTree= ^Tree;   // Указатель на дерево
  Tree= record    // Само дерево, имеет тип - запись
  node: string;   // значение вершины (узла) дерева
  left: PTree; // Ссылка на левое поддерево
  right: PTree; // Ссылка на правое поддерево
  end;
```

Здесь для разнообразия в качестве значений вершин мы взяли строки символов. Напишем рекурсивную процедуру поиска вершины по его значению:

```
procedure search_node(Elem: string;
                      ptr_tree: PTree;
                      var current_tree: PTree);
var
  p: ^Tree; // рабочий указатель
begin
  p:= ptr_tree;
  if not (p^.node = Elem) then
    begin
      if p^.left <> nil then
        search_node (Elem, p^.left, current_tree);
      if p^.right <> nil then
        search_node (Elem, p^.right, current_tree);
    end
  else current_tree:= p;
end;
```

В процедуру передаются значение искомого узла `Elem`, само дерево `ptr_tree`. Процедура возвращает ссылку на найденный узел `current_tree`.

Рекурсивная процедура удаления текущего поддерева:

```
procedure dispose_tree (ptr_tree: PTree);
var
  p: ^Tree;
begin
  if ptr_tree <> nil then
  begin
    p:= ptr_tree;
```

```
    if p^.left <> nil then
    begin
      dispose_tree(p^.left);
    end;
    if p^.right <> nil then
    begin
      dispose_tree(p^.right);
    end;
    dispose(p);
  end
end;
```

Рекурсивная процедура обхода двоичного дерева слева запишется на удивление просто:

```
procedure obhod(p: PTree);
begin
   if p<>nil then
   begin
      obhod(p^.left);
      write(p^.node, ' ');
      obhod(p^.right);
   end;
end;
```

Реализуйте самостоятельно обход двоичного дерева слева по не рекурсивному алгоритму 7.30!

Напишем программу ввода двоичного дерева с клавиатуры и его обхода слева:

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type
  PTree= ^Tree;   // Указатель на дерево
  Tree= record    // Само дерево, имеет тип - запись
  node: string;   // значение вершины (узла) дерева
  left: PTree;    // Ссылка на левое поддерево
  right: PTree;   // Ссылка на правое поддерево
  end;
var
  ptr_tree, current_tree, root: ^Tree;
  p, current: ^Tree;
  s: string;
  choose: integer;
{Процедура поиска узла}
procedure search_node(Elem: string;
                      ptr_tree:PTree;
                      var current_tree:PTree);
var
  p: ^Tree;
begin
  p:= ptr_tree;
  writeln(p^.node);
  if not (p^.node = Elem) then
    begin
      if p^.left <> nil then
        search_node (Elem, p^.left, current_tree);
```

```
      if p^.right <> nil then
        search_node (Elem, p^.right, current_tree);
    end
  else current_tree:= p;
end;
```

{Процедура вывода дерева на экран}

```
procedure view_tree (ptr_tree: PTree);
var
  p: ^Tree;
begin
  p:= ptr_tree;
  writeln(p^.node);
  if p^.left <> nil then
    view_tree(p^.left);
  if p^.right <> nil then
    view_tree(p^.right);
end;
```

{Процедура удаления текущего поддерева}

```
procedure dispose_tree (ptr_tree:PTree);
var
  p: ^Tree;
begin
  if ptr_tree <> nil then
  begin
    p:= ptr_tree;
    writeln(p^.node);
    if p^.left <> nil then
```

```
    begin
      dispose_tree(p^.left);
    end;
    if p^.right <> nil then
    begin
      dispose_tree(p^.right);
    end;
    dispose(p);
  end
end;
procedure obhod(p: PTree);
begin
   if p <> nil then
   begin
      obhod(p^.left);
      write(p^.node, ' ');
      obhod(p^.right);
   end;
end;

begin
  writeln(UTF8ToConsole('введите номер или имя вершины'));
  readln(s);
  new(current);
  root:= current;
  current^.node:= s;
  current^.left:= nil;
  current^.right:= nil;
  repeat
```

```
writeln;
writeln(UTF8ToConsole('Корень текущего подерева: '),
        current^.node);
writeln(UTF8ToConsole('Выберите нужное действие:'));
writeln(UTF8ToConsole('1-ввод левого поддерева'));
writeln(UTF8ToConsole('2-ввод правого поддерева'));
writeln(UTF8ToConsole('3-сделать корень поддерева текущим'));
writeln(UTF8ToConsole('4-просмотреть дерево'));
writeln(UTF8ToConsole('5-удалить текущее поддерево'));
writeln(UTF8ToConsole('6-обход дерева слева'));
writeln(UTF8ToConsole('7-выход из программы'));
readln(choose);
case choose of
1: begin {Создание левого поддерева}
     if current^.left= nil then
       new(p)
     else
       p:= current^.left;
     writeln(UTF8ToConsole('введите номер или имя вершины'));
     readln(s);
     p^.node:= s;
     p^.left:= nil;
     p^.right:= nil;
     current^.left:= p;
   end;
2: begin {Создание правого поддерева}
     if current^.right= nil then
       new(p)
     else
```

```
        p:= current^.right;
      writeln(UTF8ToConsole('введите номер или имя вершины'));
      readln(s);
      p^.node:= s;
      p^.left:= nil;
      p^.right:= nil;
      current^.right:= p;
    end;
3: begin {Поиск нужной вершины}
      writeln(UTF8ToConsole('введите номер или имя вершины'));
      readln(s);
      current_tree:= nil;
      ptr_tree:= root;
      search_node (s, ptr_tree, current_tree);
      if current_tree <> nil then
        current:= current_tree;
    end;
4: begin {Вывод введенного дерева на экран}
      ptr_tree:= root;
      view_tree(ptr_tree);
    end;
5: begin {Удаление поддерева}
      writeln(UTF8ToConsole('введите букву L для'));
      writeln(UTF8ToConsole('удаления левого поддерева'));
      writeln(UTF8ToConsole('или любой символ для'));
      writeln(UTF8ToConsole('удаления правого поддерева'));
      readln(s);
      if (s= 'l') or (s= 'L') then
      begin {Удаление левого поддерева}
```

```
          ptr_tree:= current^.left;
          current^.left:= nil;
          dispose_tree(ptr_tree);
        end
        else
        begin {Удаление правого поддерева}
          ptr_tree:= current^.right;
          current^.right:= nil;
          dispose_tree(ptr_tree);
        end;
      end;
    6: begin
        writeln(UTF8ToConsole('Обход двоичного дерева слева:'));
        writeln;
        obhod(root);
        writeln;
      end;
    end; { end of case }
  until choose = 7
end.
```

7.3.6. Сортировка и поиск с помощью двоичного дерева

Двоичные деревья чаще всего применяются для сортировки и поиска. Пусть для определенности сортируется массив, состоящий из целых чисел. Для этого сначала строится двоичное дерево по следующему алгоритму: в качестве корня двоичного дерева берется первый элемент массива. Следующие элементы массива становятся либо левыми, либо правыми потомками в зависимости от значения элемента. Если следующий элемент меньше корня, то он вставля-

ется в левое поддерево, если больше корня, то в правое поддерево, причем вставляется в нужное место в зависимости от значения текущего элемента. После построения двоичного дерева осуществляется его обход слева. Легко видеть, что если двоичное дерево построено так, как описано выше, то при его обходе слева мы получим упорядоченный по возрастанию массив.

Такие двоичные деревья называются двоичными деревьями поиска или деревья бинарного поиска (`binary search tree`), поскольку при поиске используется упорядоченность двоичного дерева. Поиск происходит следующим образом. В качестве текущего узла принимаем корень. Затем сравниваем значение искомого элемента со значением текущего узла. Если они равны, то требуемый элемент найден и алгоритм заканчивает свою работу. В противном случае, если искомый элемент меньше значения текущего узла, то текущим делаем левое поддерево, если больше, то текущим становится правое поддерево. Снова сравниваем наш элемент с текущим узлом и так далее до тех пор, пока искомый элемент не будет найден, либо не будет достигнут пустой узел, что будет означать, что искомого элемента в массиве нет.

Выше мы уже отмечали, что деревья обычно не вводятся, а формируются программно. Напишем процедуру формирования двоичного дерева по заданному массиву целых чисел:

```
procedure insert_node(Elem: integer; var root:PTree);
var
  p: ^Tree; // текущий узел
  newTree: ^Tree; // новый узел
begin
  p:= root; // начинаем с корня и проходим до нужного узла
  while (Elem > p^.node) and (p^.right <> nil) or
        (Elem < p^.node) and (p^.left <> nil) do
  if Elem < p^.node then
```

```
    p:= p^.left
  else
    p:= p^.right;
  New(newTree); {Создание нового узла}
  newTree^.left:= nil;
  newTree^.right:= nil;
  newTree^.node:= Elem;
  {В зависимости от значения Elem новый
   узел добавляется либо справа, либо слева}
  if Elem > p^.node then
    p^.right:= newTree
  else
    p^.left:= newTree;
end;
```

Процедуру поиска мы уже разрабатывали в последнем примере предыдущего раздела. На этот раз оформим поиск в виде функции:

```
function Search_Elem(Elem: integer;
                     var root:PTree): boolean;
var
  p: PTree;
begin
  Search_Elem:= false;
  if root = nil then exit;
  p:= root;
  if p^.node = Elem then
    Search_Elem:= true  // элемент найден
  else
    if Elem < p^.node then
```

```
        Search_Elem:= Search_Elem(Elem, p^.left)
    else
        Search_Elem:= Search_Elem(Elem, p^.right);
end;
```

Теперь давайте напишем программу сортировки массива целых чисел по возрастанию и поиска в дереве бинарного поиска нужного элемента. Все нужные процедуры у нас уже имеются.

```
program in_order;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type
  vector = array of integer;
  PTree= ^Tree;  // Указатель на дерево
  Tree= record   // Само дерево, имеет тип - запись
  node: integer;  // значение вершины (узла) дерева
  left: PTree; // Ссылка на левое поддерево
  right: PTree; // Ссылка на правое поддерево
  end;
var
  i, n, choose: integer;
  Elem: integer;
  sorted_array: vector; // сортируемый массив
  root: PTree;

{Процедура обхода двоичного дерева слева}
procedure obhod(p: PTree);
```

```
begin
   if p <> nil then
   begin
      obhod(p^.left);
      write(p^.node, ' ');
      obhod(p^.right);
   end;
end;

{Процедура поиска узла для вставки нового узла}
procedure insert_node(Elem: integer; var root: PTree);
var
  p:^Tree; // текущий узел
  newTree:^Tree; // новый узел
begin
  p:=root; // начинаем с корня и проходим до нужного узла
  while (Elem > p^.node) and (p^.right <> nil) or
        (Elem < p^.node) and (p^.left <> nil) do
  if Elem < p^.node then
    p:= p^.left
  else
    p:= p^.right;
  {Создание нового узла}
  New(newTree);
  newTree^.left:= nil;
  newTree^.right:= nil;
  newTree^.node:= Elem;
  {В зависимости от значения Elem новый
узел добавляется либо справа, либо слева}
```

```
  if Elem > p^.node then
    p^.right:= newTree
  else
    p^.left:= newTree;
end;

  { ========= Сортировка двоичным деревом поиска ======== }
procedure Tree_Sort(var sorted_array: vector);
var
  Elem: integer;
begin
  Elem:= sorted_array[0];
  New(root);
  root^.left:= nil;
  root^.right:= nil;
  root^.node:= Elem;
  for i:= 1 to High(sorted_array) do
  begin
    Elem:= sorted_array[i];
    insert_node(Elem, root);
  end;
  obhod(root); // Обход полученного дерева слева
end;

{Функция поиска в бинарном дереве}
function Search_Elem(Elem: integer;
                     var root: PTree): boolean;
var
  p: PTree;
```

```
begin
  Search_Elem:= false;
  if root = nil then exit;
  p:= root;
  if p^.node = Elem then
    Search_Elem:= true  // элемент найден
  else
    if Elem < p^.node then
       Search_Elem:= Search_Elem(Elem, p^.left)
    else
       Search_Elem:= Search_Elem(Elem, p^.right);
end;
begin
  writeln(UTF8ToConsole('Введите количество элементов массива'));
  readln(n);
  SetLength(sorted_array, n);
  writeln(UTF8ToConsole('Введите элементы массива'));
  for i:= 0 to n - 1 do
    read(sorted_array[i]);
  repeat
    writeln(UTF8ToConsole('Выберите нужное действие:'));
    writeln(UTF8ToConsole('1-сортировка массива'));
    writeln(UTF8ToConsole('2-поиск элемента массива'));
    writeln(UTF8ToConsole('3-выход из программы'));
    readln(choose);
    case choose of
    1: begin {Сортировка}
      {Вызов процедуры сортировки массива бинарным деревом поиска}
         Tree_Sort(sorted_array); writeln;
```

```
      end;
    2: begin {поиск}
         writeln(UTF8ToConsole('введите искомый элемент'));
         readln(Elem);
         if Search_Elem(Elem, root) then
           writeln(UTF8ToConsole('Элемент найден'))
         else
           writeln(UTF8ToConsole('Элемент не найден'));
      end;
    end; { end of case }
  until choose = 3;
end.
```

При работе с программой обратите внимание на то, что, прежде чем вызывать функцию поиска, необходимо отсортировать массив, т.е. построить соответствующее двоичное дерево.

Глава VIII. Основы объектно-ориентированного программирования

8.1 Три источника и три составные части ООП.

Аббревиатура ООП расшифровывается как объектно-ориентированное программирование. Рассмотрим три источника и три составные части марксизма-ленинизма, ой простите (куда это меня занесло?!), конечно же, ООП!

Три источника – это объекты, абстракция и классификация.

Основная идея ООП заключается в объединении данных, с которыми работает программа и процедур, которые эти данные обрабатывают в единое целое – объект. Такая организация программы позволила максимально приблизить к естественному восприятию человеком окружающих его предметов, сущностей и понятий. Ведь человек воспринимает окружающий его мир, предметы и явления в совокупности свойств, составляющих их элементов и их поведения.

Программист при решении задачи из какой-либо предметной области, выделяет отдельные объекты, исходя из особенностей задачи. Этот процесс называется объектной декомпозицией [5]. Объекты состоят из данных, описывающих свойства этих объектов и процедур, обрабатывающих эти данные. Например, при создании, скажем, базы данных студентов некоторого университета, можно выделить объект "Студент". Данными (свойствами) для этого объекта могут выступать фамилия и имя студента, курс, группа, его оценки и т.д. При этом можно определить некоторые процедуры для обработки этих данных, например процедуру вычисления средней оценки за семестр (эту процедуру можно в дальнейшем использовать для установления размера стипендии), процедуру перевода с курса на курс, процедуру отчисления (к сожалению) из университета и т.д.

При рассмотрении объектов очень важен уровень абстракции (или уровень

детализации) с которой мы рассматриваем объект. В нашем случае нас совершенно не интересуют такие характеристики объекта как рост, цвет глаз, размер обуви и т.д. Мы абстрагируемся от этих свойств и выделяем только те свойства, которые позволяют нам решать поставленную задачу.

В то же время важно понимать, что конкретный "экземпляр" студента, например по фамилии Иванов является представителем целого класса студентов. Эту классификацию можно продолжать и развивать как говорится "в разные стороны". Вообще студент, студент какой-либо группы, студент какого-нибудь ВУЗа. Следовательно, мы можем ввести в рассмотрение объект "Группа", объект "ВУЗ". Наконец, студент без всякого сомнения является человеком!! Отсюда мы можем говорить о таком объекте, как "Человек".

Три составные части – это инкапсуляция, наследование и полиморфизм.

Объединение данных и обрабатывающих их функций и процедур в виде отдельных объектов называется инкапсуляцией. Основная сложность ООП заключается в искусстве выделить объекты, используя абстрагирование и классификацию, которые как можно точнее описывали бы решаемую задачу и, кроме того, позволяли бы их повторное использование. Внутреннее "устройство" объекта может быть достаточно сложным, но оно "скрыто от чужих глаз". Для связи с "внешним миром" используются лишь небольшие объемы данных, причем количество и тип этих данных строго под контролем. Это существенно повышает надежность программы.

Наследование одно из важнейших свойств ООП. Создание новых объектов путем использования уже существующих дает программисту ряд преимуществ:

- Не нужно повторно разрабатывать код. Весь код для существующих объектов автоматически может быть использован для новых объектов;
- Вероятность ошибок резко снижается. Если код для уже существующих объектов был уже отлажен и проверен, то любые возникшие ошибки следует искать в кодах, которые были добавлены для новых объектов и, наоборот, если найдены и исправлены ошибки в кодах для исходных объектов, то они будут

автоматически исправлены и в кодах для новых объектов, созданных путем наследования;

- Код программы становится более понятным. Для того чтобы понять как работают объекты, созданные путем наследования достаточно понять как работают добавленные данные и функции.

Под полиморфизмом понимается возможность одних и тех же функций или процедур по разному обрабатывать данные, принадлежащие разным объектам. Например, пусть у вас имеется объект "Геометрическая фигура" и пусть у нее имеется функция вычисления площади. Если необходимо вычислить площадь прямоугольника, то функция будет вычислять ее по одному алгоритму, а если это треугольник, то ее площадь функция будет вычислять по совсем другому алгоритму.

8.2. Классы и объекты.

Основным понятием ООП является класс. Класс представляет собой структуру, состоящую из описания полей данных, функций и процедур, обрабатывающих поля данных и свойств. Функции и процедуры класса принято называть методами. Поля, методы и свойства называются элементами класса или членами класса.

Поле это обычная переменная языка Паскаль. Допускаются любые разрешенные в языке типы переменных, в том числе поле может иметь тип класса. С помощью полей описываются состояние объекта. С помощью методов – поведение объекта. Свойство это специфический способ для организации связи с данными класса.

Таким образом, класс описывает состояние и поведение объекта. Под объектом понимается конкретный экземпляр (сущность) класса. В программе объект описывается как переменная типа класс. В языке Free Pascal как и в боль-

шинстве современных объектно-ориентированных языков программирования переменная типа класс содержит не сами данные, а ссылку на них, т.е. является указателем, а сам объект размещается в динамической памяти. Однако операция разыменования для классовых переменных в Free Pascal не применяется. Перед использованием объекта необходимо выделить для него память путем создания нового экземпляра класса или присвоением существующего экземпляра переменной типа класс. После завершения работы с этим экземпляром класса (объектом) память необходимо освободить. Объявление класса в простейшем случае производится следующим образом:

```
type
  <Имя класса> = class
  <объявление полей класса>
  <объявление методов класса>
end;
```

Рассмотрим пример объявления класса:

```
type
  THuman = class      // объявление класса
  name: string;  // поля класса
  fam: string;
  function GetData: string; // объявление метода класса
end;
```

Здесь `THuman` имя класса. Имена классов принято начинать с буквы T (от слова `Type`), хотя это и не обязательно. Полями класса являются `name` и `fam`, а `GetData` – метод класса (в данном случае это функция).

Реализация метода (тело функции или процедуры) помещается после объявления класса (ключевого слова `end`). Если класс создается внутри модуля (в большинстве случаев именно так и происходит), то реализацию метода следует помещать после ключевого слова `implementation`. При этом перед именем метода указывается имя класса через точку:

```
function THuman.GetData: string; // реализация метода
begin
  Result:= name + ' ' + fam;
end;
```

В данном случае мы создали класс `THuman` (человек) с полями `name` (имя), `fam` (фамилия) и методом `GetData`, который просто возвращает имя и фамилию человека.

Обратите внимание, что все поля класса доступны методам класса и их не надо передавать в качестве формальных параметров. Если вы укажете имена полей класса в качестве формальных параметров метода класса, например таким образом

```
function GetData(name, fam: string): string;
```

то компилятор выдаст ошибку. Точно так же в теле метода нельзя описывать переменные полей класса, например таким образом:

```
function GetData: string;
var
  name: string;
```

8.2.1. Обращение к членам класса.

После того как класс объявлен необходимо создать экземпляр класса путем объявления переменной типа класс:

```
var
  Person: THuman;
```

После описания переменной можно обращаться к членам класса. Обращение производится аналогично обращению к полям записи, т.е. через точку, например:

```
Person.name:= 'Виталий';
Person.fam:= 'Петров';
fname:= Person.GetData;
```

или с помощью оператора `with`:

```
with Person do
begin
  name:= 'Алексей';
  fam:= 'Морозов';
  fname:= GetData;
end;
```

Напишем программу работы с только что созданным классом:

```
program project1;
{$mode objfpc}{$H+}
```

```
uses
  CRT, FileUtil;
type

  THuman = class      // объявление класса
      name: string;   // поля класса
      fam: string;
      function GetData: string; // объявление метода класса
  end;

function THuman.GetData: string; // реализация метода
begin
  Result:= name + ' ' + fam;
end;

var
  Person: THuman;   // объявление переменной типа класс
  fname: string;

begin
  Person:= THuman.Create; // создание объекта
  Person.name:= 'Виталий'; // присвоение значений полям
  Person.fam:= 'Петров';
  fname:= Person.GetData; // вызов метода
  writeln(UTF8ToConsole('Это: ' + fname));
  with Person do // другой вариант обращения к членам класса
  begin
    name:= 'Алексей';
```

```
    fam:= 'Морозов';
    fname:= GetData;
  end;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;   // освобождение памяти (уничтожение объекта)
end.
```

В этой программе нами остался не рассмотренным оператор

```
    Person:= THuman.Create;
```

С помощью этого оператора в момент выполнения программы объект реально создается (конструируется), т.е. объект размещается в памяти, поля объекта инициализируются необходимыми значениями, указанными в специальном методе, называемым конструктором (о конструкторах мы поговорим позже), адрес этого объекта заносится в `Person` (не забывайте, что это указатель!).

В конце программы память, выделенная объекту, освобождается, т.е. объект уничтожается.

В программе можно использовать сколько угодно экземпляров класса. В следующем примере создается массив объектов. Имена и фамилии вводятся с клавиатуры.

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type
```

```
  THuman = class
      name: string;
      fam: string;
      function GetData: string;
      procedure SetData(var f_name, s_name: string);
  end;

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;

procedure THuman.SetData(var f_name, s_name: string);
begin
  name:= f_name;
  fam:= s_name;
end;

function GetName(var f_name, s_name: string): boolean;
begin
  Result:= true;
  writeln(UTF8ToConsole('Введите имя'));
  readln(f_name);
  if f_name = '***' then
  begin
    Result:= false;
    exit;
  end;
  writeln(UTF8ToConsole('Введите фамилию'));
```

```
  readln(s_name);
end;
const kolHuman = 25;
var
  Person: array [1..kolHuman] of THuman;
  i, kolperson: integer;
  fname, sname: string;
begin
  kolperson:= 0;
  for i:= 1 to kolHuman do
  begin
    if not GetName(fname, sname) then break;
    Person[i]:= THuman.Create;
    Person[i].SetData(fname, sname);
    inc(kolperson);
  end;
  for i:= 1 to kolperson do
  begin
    writeln(UTF8ToConsole('Это: '), Person[i].GetData);
    Person[i].Free;
  end;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Функция `GetName` не является членом класса. Это обычная функция, которая организует ввод данных. В качестве параметров передаются строковые переменные, в которые вводятся имя и фамилия. После ввода необходимых

данных создается новый объект. В класс `THuman` добавлен новый метод `SetData` с помощью которого введенные данные заносятся в нужные поля класса. Признаком окончания ввода служит строка '***'. Получив такую строку с клавиатуры, функция `GetName` "сигнализирует" вызывающей программе о том, что ввод окончен. При этом новый объект, естественно, создаваться не будет. Переменная `kolperson` это счетчик, которая отслеживает количество реально созданных объектов.

8.3. Инкапсуляция

Выше мы отмечали, что под инкапсуляцией понимается объединение данных и логики их обработки в одно целое – объект. Помимо этого, инкапсуляция подразумевает сокрытие внутренней структуры и ограничение доступа. Связь с "внешним миром" осуществляется через так называемый интерфейс класса, т.е. набором правил, регламентирующих доступ к членам класса. В известном смысле с классом можно ассоциировать некий "черный ящик". Нам известно что он ("черный ящик") может делать. Управлять им мы можем только через его интерфейс. А вот как он это делает нам неизвестно.

Точно так же, нам важно знать что делает и умеет делать класс. А как он это делает нас, вообще говоря, не должно занимать. Этим должен заниматься программист – разработчик класса. Также для нас важно, как этим классом пользоваться (какой у этого класса интерфейс), т.е. какие у этого класса имеются доступные свойства и методы.

Возьмем, для примера, телевизор. Что происходит у него внутри, когда мы его включаем известно "одному аллаху", но нам это и не нужно знать. Возможно, кому-то интересно как устроен телевизор, но для большинства людей важно то, что у телевизора имеются соответствующие кнопки или пульт с кнопками (интерфейс) с помощью которых мы можем смотреть футбол или хоккей (ну,

или кому что нравится смотреть).

Принцип сокрытия информации – один из важнейших принципов ООП. Скрывая внутреннее устройство класса, мы абстрагируемся от ненужных деталей. Кроме того, это позволяет защитить члены класса от несанкционированного доступа извне.

В первой программе раздела 8.2.1. мы обращались к членам класса напрямую, например оператором

```
Person.name:= 'Виталий';
```

Вообще говоря, прямое обращение к полям не рекомендуется. Это может стать источником ошибок. Напишем для иллюстрации следующий пример:

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type

  THuman = class
      name: string;
      fam: string;
      birthday: integer;
      function GetData: string;
  end;

function THuman.GetData: string;
var
  s: string;
begin
```

```
  str(birthday, s);
  Result:= name + ' ' + fam + ' ' + s;
end;

var
  Person: THuman;

begin
  Person:= THuman.Create;
  writeln(UTF8ToConsole('Введите имя'));
  readln(Person.name);
  writeln(UTF8ToConsole('Введите фамилию'));
  readln(Person.fam);
  writeln(UTF8ToConsole('Введите год рождения'));
  readln(Person.birthday);
  writeln(UTF8ToConsole('Это: '), Person.GetData);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;
end.
```

В класс `THuman` мы добавили новое поле `birthday` (год рождения). Кроме того, немного видоизменили метод `GetData`. Поскольку эта функция возвращает строку, необходимо содержимое поля `birthday`, имеющий тип `integer`, преобразовать в строку. Теперь предположим, при вводе года рождения был введен недопустимый символ. Ваша программа аварийно завершится. Поэтому для изменения значения поля лучше использовать метод (процедуру или функцию). Во второй программе раздела 8.2.1. с массивом объектов мы собственно так и поступили. В методе вы можете организовать контроль над

вводимыми данными. Перепишем предыдущую программу следующим образом:

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type

  THuman = class
      name: string;
      fam: string;
      birthday: integer;
      function GetData: string;
      procedure SetData(var f_name, s_name,
                        b_year: string);
  end;

function THuman.GetData: string;
var
  s: string;
begin
  str(birthday, s);
  Result:= name + ' ' + fam + ' ' + s;
end;

procedure THuman.SetData(var f_name, s_name,
                         b_year: string);
var
```

```
  code, year: integer;
begin
  name:= f_name;
  fam:= s_name;
  val(b_year, year, code);
  if code = 0 then
    birthday:= year;
end;

var
  Person: THuman;
  fname, sname, year: string;
begin
  Person:= THuman.Create;
  writeln(UTF8ToConsole('Введите имя'));
  readln(fname);
  writeln(UTF8ToConsole('Введите фамилию'));
  readln(sname);
  writeln(UTF8ToConsole('Введите год рождения'));
  readln(year);
  Person.SetData(fname, sname, year);
  writeln(UTF8ToConsole('Это: '), Person.GetData);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;
end.
```

Здесь мы вводим данные в некоторые вспомогательные переменные. Лишь после окончания ввода методом `SetData` записываем значения в поля объекта

`Person`. Причем в `SetData` осуществляем контроль значения года рождения путем преобразования строки в число функцией `val`. Как вам уже известно, эта функция возвращает ненулевое значение в параметре `code`, если строка не может быть преобразована в число.

Однако, в большинстве случаев, целесообразно вообще запретить доступ к некоторым членам класса извне. Для этого применяются спецификаторы доступа.

8.3.1. Спецификаторы доступа.

Для реализации инкапсуляции основанной на классах имеются следующие спецификаторы (директивы), управляющие видимостью (доступностью) членов класса:

- `private` (частный, говорят еще приватный) – поля и методы класса недоступны из других модулей. Это позволяет полностью скрыть всю "кухню" реализации класса. Однако они доступны в пределах того модуля, где описан данный класс. Более того, если в одном модуле определены несколько классов, то они "видят" приватные разделы друг друга. Это сделано для удобства разработчика данного класса (классов) в этом модуле. Согласитесь, глупо ограничивать в доступе к "внутренностям" телевизора самого изготовителя.
- `protected` (защищенный) – поля и методы класса имеют ограниченную видимость. Они видны в самом классе, во всех классах наследниках этого класса (том числе и в других модулях) и в программном коде, расположенном в том же модуле, что и данный класс.
- `public` (публичный) – свободно доступны из любого места программы, в том числе и из других модулей.

Как правило, поля класса объявляются как `private`, а методы – `public`. Хотя те методы, которые нужны только для внутреннего использования, вполне можно поместить в раздел `private` или `protected`.

Напишем следующую программу:

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type

  THuman = class
    private
      name: string;
      fam: string;
    public
      function GetData: string;
  end;

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;

var
  Person: THuman;
  fname: string;

begin
  Person:= THuman.Create;
  Person.name:= 'Виталий';
  Person.fam:= 'Петров';
```

```
  fname:= Person.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;
end.
```

Эта программа отличается от первой программы раздела 5.2.1. тем, что мы ввели в описание класса спецификаторы доступа. Здесь поля `name` и `fam` по-прежнему останутся доступны программе, несмотря на то, что они помещены в раздел `private`.

Теперь создайте модуль (меню **Файл->Создать модуль**) и переместите описание класса в созданный модуль (раздел `interface`), а реализацию метода `GetData` в раздел `implementation`. Остальной код программы оставьте без изменений. Обратите внимание, Lazarus автоматически добавил в объявление `uses` имя вновь созданного модуля.

```
unit Unit1;
{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;
type

  THuman = class
    private
      name: string;
```

```
      fam: string;
    public
      function GetData: string;
  end;

implementation

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
var
  Person: THuman;
  fname: string;

begin
  Person:= THuman.Create;
  Person.name:= 'Виталий';
  Person.fam:= 'Петров';
  fname:= Person.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
```

```
  Person.Free;
end.
```

При попытке компиляции компилятор выдаст ошибку на операторах:

```
  Person.name:= 'Виталий';
  Person.fam:= 'Петров';
```

Для записи значений в поля `name` и `fam` необходимо создать метод и поместить его объявление в раздел `public`.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type
  { THuman }
  THuman = class
    private
      name: string;
      fam: string;
    public
      function GetData: string;
      procedure SetData(const f_name, s_name: string);
  end;

implementation
```

```
function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;

procedure THuman.SetData(const f_name, s_name: string);
begin
  name:= f_name;
  fam:= s_name;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
var
  Person: THuman;
  fname: string;
begin
  Person:= THuman.Create;
  Person.SetData('Виталий', 'Петров');
  fname:= Person.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;
end.
```

При разработке классов используйте функцию "Автозавершение кода". После того, как вы написали объявление метода, нажмите `Ctrl+Shift+C` и редактор исходного кода Lazarus автоматически сформирует заготовку тела вашего метода. Например, в описании класса наберите

```
procedure SetData(const f_name, s_name: string);
```

и нажмите `Ctrl+Shift+C`. Lazarus автоматически создаст следующий код:

```
procedure THuman.SetData(const f_name, s_name: string);
begin

end;
```

8.3.2. Свойства.

В предыдущей программе для доступа к частным (`private`) полям мы использовали методы, которые поместили в раздел `public`. Фактически мы организовали обмен данными между классом и внешней средой. Но для доступа к полям данных в классе лучше использовать свойства. Свойства описывают состояние объекта, поскольку определяются полями класса и снабжены двумя специальными методами для записи и чтения значения поля. Таким образом, свойство это специальный механизм для организации доступа к полю. Определяется тремя составляющими: поле, метод чтения, метод записи. Преимущество свойства в том, что методы чтения и записи полностью скрыты. Это позволяет вносить изменения в код класса не изменяя использующий его внешний код.

Объявление свойства имеет вид:

`property` имя 1: тип `<read` имя 2`>` `<write` имя 3`>;`

где `property` (свойство) – ключевое слово;

имя 1 – имя свойства, любой разрешенный идентификатор ;

тип – тип поля;

имя 2 – имя метода чтения или непосредственно имя поля;

имя 3 – имя метода записи или непосредственно имя поля;

Угловые скобки означают, что данный параметр может отсутствовать. В случае отсутствия параметра `write` свойство становится свойством "только для чтения". Изменить значение свойства тогда будет невозможно. Отсутствие параметра `read` приводит к тому, что свойство становится свойством "только для записи". Однако это лишено смысла, поскольку значение этого свойства (для чтения) будет недоступно.

Наиболее логично использовать свойство со следующими параметрами:

`property` имя свойства: тип `read` имя поля `write` имя метода записи;

В этом случае для обращения к значению свойства программа производит чтение непосредственно защищенного поля, а для записи значения свойства будет вызываться метод записи. В методе перед записью можно организовать контроль данных на соответствие каким-либо критериям в зависимости от решаемой задачи.

Объявления свойств следует располагать в разделе `public`. При записи свойства вы также можете воспользоваться функцией "Автозавершение кода" редактора исходного кода. Введите

`property` имя свойства: тип

и нажмите `Ctrl+Shift+C`. Lazarus автоматически завершит объявление свойства, а также заготовку тела метода записи. Кроме того, добавит в секцию

`private` поле с именем <Fимя свойства> с соответствующим типом и процедуру записи со стандартным именем <Setимя свойства>. Имена полей принято начинать с буквы F, поэтому Lazarus формирует такое имя. Но это не обязательно. Если вас не устраивают имена, предложенные Lazarus, вы можете просто их переименовать. Например, начните набирать описание класса:

```
type
  THuman = class
    private
      name: string;
    public
      property name: string
  end;
```

Нажмите `Ctrl+Shift+C`. Автоматически будет создан следующий код:

```
type

  { THuman }

  THuman = class
    private
      Fname: string;
      name: string;
      procedure Setname(const AValue: string);
    public
      property name: string read Fname write Setname;
  end;
```

```
implementation

procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
  Fname:= AValue;
end;
```

Поскольку при использовании функции "Автозавершение кода" Lazarus всегда добавляет новое поле в секцию `private`, логичнее начинать описание класса со спецификатора `public` и определения свойств. В этом случае секция `private` также будет автоматически создана и вы получите описание полей и свойств и готовые шаблоны реализации методов записи. В данном случае мы получили описание поля `Fname` и заготовку процедуры `Setname`.

При использовании свойства нет необходимости явно вызывать процедуру записи. Достаточно записать:

```
Person:= THuman.Create;
Person.name:=  AValue;
```

Таким образом, для пользователя класса свойство будет выглядеть как обычное поле.

Если вы хотите (при использовании функции "Автозавершение кода") для чтения также использовать метод, то вам достаточно перед именем поля в описании свойства добавить слово `Get` и вновь нажать на комбинацию клавиш `Ctrl+Shift+C`. Вы получите заготовку процедуры чтения, например:

```
property name: string read GetFname write Setname;
```

```
   implementation
   function THuman.GetFname: string;
   begin

   end;
```

Рассмотрим примеры работы со свойствами.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type
  { THuman }
  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setfam(const AValue: string);
      procedure Setname(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      function GetData: string;
   end;

implementation
```

```
{ THuman }
procedure THuman.Setname(const AValue: string);
begin
  if Fname = AValue then exit;
  Fname:= AValue;
end;
procedure THuman.Setfam(const AValue: string);
begin
  if Ffam = AValue then exit;
  Ffam:= AValue;
end;
function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
var
  Person: THuman;
  fname: string;
begin
  Person:= THuman.Create;
  Person.name:= 'Виталий';
  Person.fam:= 'Петров';
  fname:= Person.GetData;
```

```
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;
end.
```

В этой программе мы определили два свойства `name` и `fam`. Воспользовавшись функцией автозавершения, мы получили методы записи данных в поля. В программе в явном виде мы эти методы не вызывали. Мы записали просто

```
  Person.name:= 'Виталий';
  Person.fam:= 'Петров';
```

Всю остальную работу (вызов методов записи) за нас выполнил компилятор. Обратите внимание, по сравнению с предыдущей программой `name` и `fam` это не имена полей, а имена свойств. В следующем примере данные считываются с клавиатуры. Добавлен контроль ввода данных.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type
  { THuman }

  THuman = class
    private
```

```
      Fname: string;
      Ffam: string;
      Fbirthday: integer;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
      procedure Setbirthday(const AValue: integer);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      property birthday: integer read Fbirthday write
                                      Setbirthday;
      function GetData: string;
   end;

implementation

{ THuman }

procedure THuman.Setname(const AValue: string);
begin
  if Fname = AValue then exit;
  Fname:= AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam = AValue then exit;
  Ffam:= AValue;
end;
```

```
procedure THuman.Setbirthday(const AValue: integer);
begin
  if Fbirthday = AValue then exit;
  Fbirthday:= AValue;
end;
function THuman.GetData: string;
var
  s: string;
begin
  str(birthday, s);
  Result:= name + ' ' + fam + ' ' + s;
end;
end.

program project1;

{$mode objfpc}{$H+}

uses
  CRT, FileUtil, Unit1;
var
  Person: THuman;
  fname, sname, s_year: string;
  code, year: integer;
begin
  Person:= THuman.Create;
  writeln(UTF8ToConsole('Введите имя'));
  readln(fname);
```

```
  Person.name:= fname;
  writeln(UTF8ToConsole('Введите фамилию'));
  readln(sname);
  Person.fam:= sname;
  writeln(UTF8ToConsole('Введите год рождения'));
  readln(s_year);
  val(s_year, year, code);  // контроль ввода
  if code = 0 then
    Person.birthday:= year
  else
    Person.birthday:= 0;
  fname:= Person.GetData;
  writeln(UTF8ToConsole('Это: '), fname);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;
end.
```

8.4. Наследование

Представьте себе, что имеется класс, который вас в принципе удовлетворяет, но чего-то в нем чуть-чуть не хватает. Что придется заново разрабатывать класс? Конечно же, заново создавать класс не нужно. Достаточно создать класс на основе существующего и просто добавить в него недостающие члены (поля, методы и свойства). Это и есть наследование. Новый класс называется наследником или потомком или дочерним классом. А старый класс называется родительским классом или предком или базовым классом. Объявляется класс наследник следующим образом:

```
type
<Имя класса наследника> = class(Имя класса родителя)
            <Описание полей, методов и свойств,
        характерных только для класса наследника>
end;
<Реализация методов>
```

Если имя класса родителя не указано, то по умолчанию класс наследуется от самого общего класса `TObject`. Наш класс `THuman` из предыдущих примеров является наследником `TObject`.

Класс наследник получает (наследует) все поля, методы и свойства класса родителя. В то же время созданный класс, в свою очередь может выступать в качестве родителя. В `Free Pascal` существует так называемая иерархия классов, на вершине которого и находится класс `TObject`. Таким образом, класс `TObject` является прародителем всех остальных классов.

Например, создадим класс `TStudent` (студент). Можно создать этот класс с нуля, но, поскольку студент является человеком, то логично будет создать класс на основе класса `THuman`.

```
program project1;

{$mode objfpc}{$H+}

uses
  CRT, FileUtil;
type

  THuman = class      // объявление базового класса
    private
```

```
      name: string;
      fam: string;
    public
      function GetData: string;
  end;
function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;

type

  TStudent = class(THuman) // объявление класса – наследника
    private
      group: string;
  end;

var

  Student: TStudent;
  fname: string;

begin

  Student:= TStudent.Create;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
  Student.group:= 'ПОВТАС-1/09';
  fname:= Student.GetData;
```

```
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Student.Free;
end.
```

Здесь в класс TStudent мы ввели поле group, характерное только для студента. В данном случае это группа. В то же время, благодаря наследованию, будут доступны поля name (имя) и fam (фамилия), а также метод GetData родительского класса. И класс THuman и класс TStudent являются потомками класса TObject, поэтому для класса TStudent доступен метод (конструктор) Create класса TObject (более подробно о конструкторах смотрите ниже). Обратите внимание, если вы собираетесь использовать только объект класса TStudent, то создавать экземпляр класса THuman вовсе не обязательно. Более того, вы можете создавать класс наследник в другом модуле, нежели где описан базовый класс.

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

type

  THuman = class      // объявление класса
```

```
    protected
      name: string;
      fam: string;
    public
      function GetData: string;
  end;
implementation

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
type

  TStudent = class(THuman) //  объявление класса - наследника
    private
      group: string;
  end;

var

  Student: TStudent;
  fname: string;
```

```
begin
  Student:=TStudent.Create;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
  Student.group:= 'ПОВТАС-1/09';
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Student.Free;
end.
```

Здесь мы описали родительский класс в отдельном модуле `Unit1`. Обратите внимание, чтобы поля родительского класса были доступны классу наследнику, мы в классе `THuman` объявили поля со спецификатором `protected`.

Если поля родительского класса защитить спецификатором `private`, то дочерний класс может получить доступ к его полям с помощью его свойств (разумеется, в родительском классе должны быть описаны соответствующие свойства).

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;
```

```
type

  { THuman }

  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      function GetData: string;
   end;

implementation

{ THuman }

procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
  Fname:=AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam=AValue then exit;
```

```
  Ffam:=AValue;
end;

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;
end.
program project1;
{$mode objfpc}{$H+}

uses
  CRT, FileUtil, Unit1;

type

  TStudent = class(THuman) // объявление класса - наследника
    private
      group: string;
  end;

var
  Student: TStudent;
  fname: string;

begin
  Student:= TStudent.Create;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
```

```
  Student.group:= 'ПОВТАС-1/09';
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Student.Free;
end.
```

Напишем класс `TProfessor` (преподаватель). Преподаватель тоже является человеком (или у вас есть сомнения на этот счет?!), поэтому совершенно естественно, что этот класс будет также наследоваться от класса `THuman`. Так же как студенты "кучкуются" в группы, так и преподаватели объединяются в кафедры. Поэтому для класса введем поле `kafedra`. В следующем примере создаются сразу два класса наследника. Приведу только код основной программы. Класс `THuman` (модуль `Unit1`, см. предыдущий пример) не претерпит никаких изменений.

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;

type
  TStudent = class(THuman) // объявление класса - наследника
    private
      group: string;
  end;
```

```
type
    TProfessor = class(THuman) // объявление класса - наследника
      private
        kafedra: string;
    end;
var
  Student: TStudent;
  Professor: TProfessor;
  fname: string;
begin
  Student:= TStudent.Create;
  Professor:= TProfessor.Create;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
  Student.group:= 'ПОВТАС-1/09';
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  Professor.name:= 'Иван';
  Professor.fam:= 'Иванов';
  Professor.kafedra:='Программирование';
  fname:= Professor.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Student.Free;
end.
```

8.5. Полиморфизм

Во всех примерах раздела 8.4. для чтения полей мы использовали метод `GetData` родительского класса. Но он возвращает только значения полей `name` и `fam`. Поэтому, несмотря на то, что мы в программе указывали значения полей `group` и `kafedra`, они на экран не выводились. Как же вывести значения этих полей? Решение заключается в написании соответствующего метода в дочернем классе. Причем мы вольны присваивать этому методу любое имя. Например, мы можем написать методы:

```
function TStudent.A: string;
begin
  Result:= name + ' ' + fam + ', группа ' + group;
end;

function TProfessor.B: string;
begin
  Result:= name + ' ' + fam + ', кафедра ' + kafedra;
end;
```

Но, давайте вспомним, что методы как мы отмечали выше, характеризуют некоторые действия по обработке данных класса. В этом контексте мы можем ввести в рассмотрение действие "Получить сведения об объекте". В одном случае это будет "Получить сведения о человеке", в другом случае "Получить сведения о студенте", в третьем – "Получить сведения о преподавателе". Во всех трех случаях это в принципе однотипные действия, хотя и несколько различающиеся по сути. Мы можем написать совсем коротко – "Получить сведения".

Так вот, возвращаясь "к нашим баранам", мы можем констатировать, что функция `GetData` это и есть действие "Получить сведения". Поэтому мы можем использовать в дочерних классах методы с таким же именем, что и в родительском классе. Конечно, реализации этих методов различаются и даже могут различаться кардинально, но по сути, если вы хотите реализовать однотипные аспекты поведения объектов, вы можете и должны давать одинаковые имена их методам. Давая им различные имена, вы можете очень скоро запутаться, особенно если производных классов достаточно много. Отсюда, мы можем написать:

```
function TStudent.GetData: string;
begin
  Result:= name + ' ' + fam + ', группа ' + group;
end;

function TProfessor.GetData: string;
begin
  Result:= name + ' ' + fam + ', кафедра ' + kafedra;
end;
```

Такое явление, когда методы класса родителя и методы дочерних классов имеют одинаковые имена, но различные реализации называется полиморфизмом. Функции `THuman.GetData`, `TStudent.GetData` и `TProfessor.GetData` несомненно различаются, поскольку возвращают строки с разным содержанием. В одном случае просто имя и фамилию, в другом имя, фамилию и наименование группы, а в третьем случае – имя, фамилию и название кафедры. Можно еще более подчеркнуть их различие, если в классе `TStudent` описать тип поля `group` как `integer`. В этом случае будет возвращаться не название группы, а его номер. Реализация этого метода:

```
function TStudent.GetData: string;
var
  s: string;
begin
  str(group, s);
  Result:= name + ' ' + fam + ', группа ' + s;
end;
```

8.5.1. Раннее связывание.

Одноименные методы, определенные таким образом, называются статическими. При вызове метода, помимо явно описанных параметров функции или процедуры, неявно передается еще один параметр `self` – указатель на объект, вызвавший метод. Таким образом, компилятор легко определяет объект какого класса вызвал метод и организует вызов нужного метода. Это так называемое раннее связывание. Метод дочернего класса как бы подменяет метод родительского класса с тем же именем. Гораздо чаще применяются термины перекрытие или переопределение. Можно было сказать – метод дочернего класса перекрывает (переопределяет) метод родительского класса с тем же именем. При этом количество и типы параметров нового метода дочернего класса и переопределяемого метода могут не совпадать. Рассмотрим пример, где используется механизм раннего связывания.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type
```

```
  { THuman }

  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      function GetData: string;
   end;

implementation

{ THuman }

procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
  Fname:=AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam=AValue then exit;
  Ffam:=AValue;
end;
```

```
function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;
end.

program project1;

{$mode objfpc}{$H+}

uses
  CRT, FileUtil, Unit1;

type

  TStudent = class(THuman)
    private
      group: string;
    public
      function GetData: string;
  end;

function TStudent.GetData: string;
begin
  Result:= name + ' ' + fam + ', группа ' + group;
end;

  type
```

```
    TProfessor = class(THuman)
      private
        kafedra: string;
      public
        function GetData: string;
    end;

function TProfessor.GetData: string;
begin
  Result:= name + ' ' + fam + ', кафедра ' + kafedra;
end;

var
  Student: TStudent;
  Professor: TProfessor;
  fname: string;

begin
  Student:= TStudent.Create;
  Professor:= TProfessor.Create;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
  Student.group:= '"ПОВТАС-1/09"';
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  Professor.name:= 'Иван';
  Professor.fam:= 'Иванов';
  Professor.kafedra:='"Программирование"';
  fname:= Professor.GetData;
```

```
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Student.Free;
  Professor.Free;
end.
```

Посмотрите на реализации методов `GetData`. В чем-то они похожи. Особенно если мы перепишем их в виде:

```
function TStudent.GetData: string;
begin
  Result:= name + ' ' + fam;
  Result:= Result + ', группа ' + group;
end;

function TProfessor.GetData: string;
begin
  Result:= name + ' ' + fam;
  Result:= Result + ', кафедра ' + kafedra;
end;
```

Первые операторы этих функций совпадают с методом родительского класса. Это в наших примерах методы очень простые. В реальности методы могут быть очень сложными и насчитывать не один десяток, а то и сотен строк кода. Что, надо копировать весь похожий код в каждый из дочерних методов? Конечно же, нет! Оказывается можно вызывать метод родительского класса с помощью ключевого слова `inherited`. Например:

```
function TStudent.GetData: string;
begin
  Result:= inherited GetData;
  Result:= Result + ', группа ' + group;
end;
```

Здесь сначала вызывается метод родительского класса, а затем добавляется новый код.

8.5.2. Позднее связывание.

Выше мы видели, что для статических методов разрешение связей, т.е. определение того, какой именно метод следует вызывать, происходит на этапе компиляции. Но бывают ситуации, когда компилятор не может определить, к какому объекту (экземпляру класса) относится метод. Рассмотрим пример. Пусть нам необходимо, чтобы на экран выводилась кроме прежней информации еще и статус человека, т.е. кто он – студент или преподаватель. Для этого введем новый метод `Status`. Этот метод будет вызываться из метода `GetData`. Во время компиляции неизвестно, объект какого класса вызывает метод `Status`. Это определяется на этапе выполнения, когда точно известен активный в данный момент объект. Разрешение связей на этапе выполнения называется поздним связыванием. Для реализации механизма позднего связывания применяются ключевые слова `virtual` и `override`. Метод родительского класса помечается ключевым словом `virtual`. Методы дочерних классов, перекрывающих метод родительского класса помечаются ключевым словом `override`, а все эти методы, включая и родительский, называются виртуальными методами. Причем в отличие от статических методов, количество, тип и порядок следования параметров в виртуальных методах должны совпадать. Рассмотрим пример.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type

  { THuman }

  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      function GetData: string;
      function Status: string; virtual;
   end;

implementation

{ THuman }

procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
```

```
  Fname:=AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam=AValue then exit;
  Ffam:=AValue;
end;

function THuman.Status: string;
begin

end;

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam + Status;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
type

  TStudent = class(THuman)
    private
      group: string;
```

```
    public
      function GetData: string;
      function Status: string; override;
  end;

function TStudent.Status: string;
begin
  Result:= ' - студент';
end;

function TStudent.GetData: string;
begin
  Result:= inherited GetData + ', группа ' + group;
end;

type

  TProfessor = class(THuman)
    private
      kafedra: string;
    public
      function GetData: string;
      function Status: string; override;
  end;

function TProfessor.Status: string;
begin
  Result:= ' - преподаватель';
end;
```

```
function TProfessor.GetData: string;
begin
  Result:= inherited GetData + ', кафедра ' + kafedra;
end;

var
  Student: TStudent;
  Professor: TProfessor;
  fname: string;

begin
  Student:= TStudent.Create;
  Professor:= TProfessor.Create;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
  Student.group:= '"ПОВТАС-1/09"';
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  Professor.name:= 'Иван';
  Professor.fam:= 'Иванов';
  Professor.kafedra:= '"Программирование"';
  fname:= Professor.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Student.Free;
  Professor.Free;
end.
```

Кроме виртуальных методов могут быть объявлены и так называемые динамические методы. Они объявляются с помощью ключевого слова `dynamic`. А в классах наследниках все того же ключевого слова `override`. С точки зрения наследования, методы этих двух видов одинаковы: они могут быть перекрыты в дочернем классе только одноименными методами, имеющими тот же тип. Разница заключается в реализации механизма позднего связывания. Для виртуальных методов используется специальная таблица виртуальных методов (ТВМ), а для динамических методов таблица динамических методов (ТДМ).

В ТВМ хранятся адреса всех виртуальных методов класса, включая и родительских классов, даже если они не перекрыты в данном классе. Поэтому вызовы виртуальных методов происходят достаточно быстро, однако ТВМ требует больше памяти.

С другой стороны, каждому динамическому методу присваивается уникальный индекс. В ТДМ хранятся индексы и адреса только тех динамических методов, которые описаны в данном классе, поэтому ТДМ занимает меньше памяти. При вызове динамического метода, происходит поиск в ТДМ данного класса. Если поиск не дал результатов, поиск продолжается в ТДМ всех классов-родителей в порядке иерархии. Чем больше глубина иерархии классов, тем медленнее будут работать вызовы динамических методов.

Чтобы реализовать динамические методы для нашего примера, достаточно заменить слово `virtual`, на `dynamic`.

Посмотрим теперь на код метода `Status` в родительском классе. Фактически в этом методе ничего не делается, функция возвращает пустую строку. В таких случаях удобнее объявить метод как абстрактный с помощью ключевого слова `abstract`. Реализация метода, который объявлен как абстрактный, в данном классе не производится, но обязательно должен быть переопределен в дочерних классах. Обратите внимание, только виртуальные методы могут быть объявлены абстрактными. Для нашего примера чтобы объявить абстрактный метод добавьте в классе `THuman` в объявление метода `Status` после слова

virtual через точку с запятой ключевое слово abstract и удалите код реализации метода.

8.5.3. Конструкторы и деструкторы.

Для создания объектов мы применяли метод Create. Это так называемый конструктор, т.е. специальный метод класса, который предназначен для выделения памяти и размещения в памяти экземпляра класса. Кроме того, в задачу конструктора входит инициализация значений полей экземпляра класса. Стандартный конструктор устанавливает все данные нового экземпляра класса в ноль. В результате все числовые поля и поля порядкового типа приобретают нулевые значения, строковые поля становятся пустыми, а поля, содержащие указатели и объекты получают значение nil. Под стандартным конструктором имеется в виду конструктор, унаследованный классом от класса TObject.

Если нужно инициализировать данные экземпляра класса определенными значениями, то необходимо написать свой собственный конструктор. Допускается использование нескольких конструкторов. Желательно, чтобы все конструкторы имели стандартное имя Create. Это позволяет переопределять конструкторы, включая и стандартный конструктор, для выполнения каких-либо полезных действий. Описание конструктора производится с помощью ключевого слова constructor. Давайте в последнем примере предыдущего раздела в класс THuman добавим конструктор. Код программы будет следующим:

```
unit Unit1;

{$mode objfpc}{$H+}

interface
```

```
uses
  Classes, SysUtils;
type

  { THuman }

  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      constructor Create; // конструктор
      function GetData: string;
      function Status: string; virtual; abstract;
   end;

implementation

{ THuman }

constructor THuman.Create; // реализация конструктора
begin
  Fname:= 'Андрей';
  Ffam:= 'Аршавин';
end;
```

```
procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
  Fname:=AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam=AValue then exit;
  Ffam:=AValue;
end;

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam + Status;
end;
end.

program project1;

{$mode objfpc}{$H+}

uses
  CRT, FileUtil, Unit1;

type

  TStudent = class(THuman)
    private
```

```
      group: string;
    public
      function GetData: string;
      function Status: string; override;
  end;
function TStudent.Status: string;
begin
  Result:= ' - студент';
end;
function TStudent.GetData: string;
begin
  Result:= inherited GetData + ', группа ' + group;
end;

type

  TProfessor = class(THuman)
    private
      kafedra: string;
    public
      function GetData: string;
      function Status: string; override;
  end;
function TProfessor.Status: string;
begin
  Result:= ' - преподаватель';
end;

function TProfessor.GetData: string;
```

```
begin
  Result:= inherited GetData + ', кафедра ' + kafedra;
end;
var
  Student: TStudent;
  Professor: TProfessor;
  fname: string;

begin
  Student:= TStudent.Create;
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  Professor:= TProfessor.Create;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
  Student.group:= '"ПОВТАС-1/09"';
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  Professor.name:= 'Иван';
  Professor.fam:= 'Иванов';
  Professor.kafedra:= '"Программирование"';
  fname:= Professor.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Student.Free;
  Professor.Free;
end.
```

При выполнении оператора

```
Student:= TStudent.Create;
```

сработает конструктор, определенный нами в классе `THuman`, поскольку он перекрывает стандартный конструктор. В результате поля `Fname` и `Ffam` будут инициализированы значениями `'Андрей'` и `'Аршавин'`. После выполнения операторов

```
fname:= Student.GetData;
writeln(UTF8ToConsole('Это: ' + fname));
```

на экран будет выведено

"Это: Андрей Аршавин – студент, группа".

Обратите внимание, название группы выведено не будет. Для инициализации значения поля `group` необходимо написать конструктор в классе `TStudent`, например таким образом:

```
constructor TStudent.Create;
begin
  group:= '"Арсенал"';
end;
```

Теперь при выполнении оператора

```
Student:= TStudent.Create;
```

будет вызван конструктор, определенный в классе `TStudent` и на экран будет выведено:

"Это: – студент, группа "Арсенал"".

Мы видим, что имя и фамилия отсутствуют. Это опять происходит из-за того, что конструктор `Create` дочернего класса `TStudent` перекрывает конструктор родительского класса `THuman`. Чтобы вызвать конструктор родительского класса, необходимо использовать ключевое слово `inherited`. Таким образом, реализацию конструктора класса `TStudent` необходимо записать в виде:

```
constructor TStudent.Create;
begin
  inherited Create;
  group:= '"Арсенал"';
end;
```

Теперь на экран будет выведена строка так, как мы хотели:

"Это: Андрей Аршавин – студент, группа "Арсенал"".

Вполне возможно использовать конструкторы с параметрами. Вот как, например, может выглядеть конструктор с параметрами класса `THuman`:

```
public
      constructor Create(n, f: string);
.....................................................................
end;
implementation
```

```
{ THuman }

constructor THuman.Create(n, f: string);
begin
  Fname:= n;
  Ffam:= f;
end;
```

Конструктор класса `TStudent`:

```
public
      constructor Create(gr: string);
.................................................................................
end;

constructor TStudent.Create(gr: string);
begin
  inherited Create('Андрей', 'Аршавин');
  group:= gr;
end;
```

Если вы теперь вызовете конструктор в виде

```
Student:= TStudent.Create;
```

то компилятор будет "сильно ругаться"! Необходимо вызывать конструктор таким образом:

```
Student:= TStudent.Create('"Арсенал"');
```

Обратите внимание и на вызов конструктора родительского класса

```
inherited Create('Андрей', 'Аршавин');
```

Поскольку, мы с вами отлично знаем, что Андрей Аршавин не студент, видоизмените описания классов, чтобы на экран выводилось, ну что-то типа:
"Андрей Аршавин – футболист команды Арсенал".

В заключение отметим, что конструктор создаёт новый объект только в том случае, если перед его именем указано имя класса. Если указать имя уже существующего объекта, он поведёт себя по-другому: не создаст новый объект, а только выполнит код, содержащийся в теле конструктора.

Деструктор имеет стандартное имя `Destroy` и предназначен для уничтожения объекта:

```
Student.Destroy;
```

В теле деструктора обычно должны уничтожаться встроенные объекты и динамические данные, как правило, созданные конструктором. Как и обычные методы, деструктор может иметь параметры, но эта возможность используется крайне редко. Для объявления деструктора используется ключевое слово `destructor`.

`Destroy` – это виртуальный деструктор класса `TObject`. Поэтому при переопределении деструктора его необходимо объявлять с ключевым словом `override`, например:

```
destructor Destroy; override;
```

В теле самого деструктора необходимо вызывать родительский деструктор (`inherited Destroy`). Этим обеспечивается формирование цепочки дест-

рукторов, восходящей к деструктору класса `TObject`, который и осуществляет освобождение памяти, занятой объектом.

Рекомендуется вместо метода `Destroy` использовать метод `Free`. Этот метод вначале проверяет, не является ли текущий объект `nil` и лишь, затем вызывает метод `Destroy`. Если объекты создаются в начале работы программы, а уничтожаются в самом конце, то применение метода `Free` является самым оптимальным.

На этом мы закончим наш краткий экскурс в объектно-ориентированное программирование. В следующей главе мы еще коснемся некоторых аспектов ООП применительно к созданию программ с графическим интерфейсом.

Более подробные сведения о ООП вы можете почерпнуть из [5] и [6]. Хотя в этих книгах материал излагается применительно к `Turbo Pascal` и `Delphi`, тем не менее, вы можете использовать их, особенно [6].

Если вы помните, при создании нашего самого первого консольного приложения (см. 5.1.10) Lazarus предложил нам заготовку кода, который мы с вами просто удалили. Тогда нам этот код был совершенно непонятен. Ну а теперь…, теперь вы можете без труда разобраться с этим кодом! Да-да, Lazarus создал для нас заготовку класса с конструктором и деструктором!

Глава IX. Программирование приложений с графическим интерфейсом

В предыдущих главах мы с вами создавали только консольные приложения. При изучении основ языка программирования (не только Паскаля, но и любого другого языка), консольные приложения являются наиболее удобными, так как позволяют сосредоточиться на существе той или иной конструкции или возможности языка.

Однако, как вы неоднократно имели возможность видеть, интерфейс пользователя был довольно примитивен и скуден. В основном удавалось построить лишь простое меню. Можно, конечно, ценой значительных усилий, создать в консольном приложении и более "приличный" интерфейс. Примером может служить IDE самого компилятора FPC.

Но в настоящее время наибольшую и заслуженную популярность завоевала идея графического интерфейса. Согласно этой идее каждая программа работает в своем собственном окне. Управление окнами стандартизовано. Стандартный интерфейс очень удобен для пользователей и значительно облегчает изучение самых разных программных средств. Запустите, например, Word и Excel в Windows или текстовый процессор OpenOffice.org и электронные таблицы OpenOffice.org в Linux. Вы увидите много похожих элементов управления окнами (кнопки, переключатели, меню и т.д.).

Графический интерфейс пользователя, иначе *GUI* (*Graphics User Interface*) является обязательным компонентом большинства современных программных продуктов, ориентированных на работу конечного пользователя. Появление графического интерфейса - одно из важнейших достижений в области разработки программного обеспечения последних лет. В дальнейшем мы будем использовать как термин "графический интерфейс пользователя", так и термин "приложение с GUI" или "GUI-приложение".

Некоторые элементы графического интерфейса Windows мы с вами рассматривали в главе III.

Стандарты GUI для различных операционных систем и платформ несколько различаются, но в целом намечается тенденция к их сближению и унификации.

Все элементы графического интерфейса можно реализовать в среде Lazarus с помощью специальной библиотеки LCL (Lazarus Component Library). Эта библиотека предоставляет программисту целую палитру так называемых визуальных компонентов, с помощью которых и можно разрабатывать интерфейс программы. При этом программист освобождается от мелких рутинных операций и может сосредоточиться именно на проектировании интерфейса своей будущей программы. Для этого он выбирает необходимые ему визуальные компоненты, настраивает их под собственные нужды и таким образом строит внешний вид (интерфейс) своей программы. При этом он еще до компиляции своего приложения видит результаты проектирования интерфейса пользователя.

Поскольку элементы библиотеки LCL доступны для всех поддерживаемых платформ, то GUI-приложения созданные на одной платформе (например Windows), могут быть без изменения скомпилированы на другой платформе (например OS X или Linux). При этом следует помнить, что графический интерфейс в различных операционных системах реализован совершенно по-другому. Так, в Windows GUI основан на функциях WinAPI, а в Linux на библиотеках GTK-2 или QT.

9.1 Различия между консольными и графическими приложениями

Итак, приложение с GUI отличается от консольного наличием графического интерфейса (напомню, что консольное приложение использует экран в тек

стовом режиме, а приложение с GUI в графическом режиме). Но между кон сольным и GUI-приложением существует еще одно, более существенное разли чие. Консольное приложение, получив управление, далее полностью само контролирует вычислительный процесс. По мере необходимости запрашивает данные для работы (оператор read), выводит результаты вычислений на экран (оператор `writeln`) и завершает работу по достижению последнего оператора программы.

Совершенно по-другому работают GUI-программы. Они реагируют на `события`. Событие может вызвать действие пользователя, например, нажатие мышью на какую-нибудь кнопку в приложении. Событие может вызвать операционная система или же событие может породить само приложение. Все события отслеживаются операционной системой, которая формирует на каждый тип события соответствующее сообщение. Это сообщение передается в приложение, на которое приложение должно реагировать в соответствии с алгоритмом своей работы. Т.е. GUI-приложение должно уметь обрабатывать сообщения операционной системы и уметь генерировать и посылать свои собственные сообщения.

Приложение, после запуска, создает свое окно и запускает так называемый цикл обработки сообщений. Грубо говоря, оно ждет сообщений от операционной системы. Роль программиста заключается в разработке кода по обработке сообщений при возникновении какого-либо события. Все сообщения имеют свои уникальный номер (код сообщения).

Сообщения передаются операционной системой именно тому приложению, которому оно предназначено. Таким образом реализуется мультипрограммность операционной системы, т.е. возможность одновременно запускать и работать с несколькими приложениями.

В Lazarus обработка сообщений заменена на обработку событий. При этом Lazarus берет на себя всю рутинную работу по расшифровке многочисленных типов сообщений и их не менее многочисленных параметров. Таким образом,

работа программиста значительно облегчается. Программисту достаточно выбрать те события, на которые будет реагировать его приложение и написать процедуру по обработке соответствующего события.

В таблице 9.1 приведены некоторые события и условия, при которых они возникают.

Табл. 9.1

№ п/п	Событие	Когда возникает
1.	`OnCreate`	Событие возникает при создании окна и только один раз.
2.	`OnShow`	Это событие генерируется, когда окно станет видимым.
3.	`OnActivate`	Это событие генерируется, когда окно становится активным, то есть когда получает фокус ввода.
4.	`OnPaint`	Это событие возникает каждый раз при перерисовке окна.
5.	`OnReSize`	Это событие возникает каждый раз при изменении размеров окна.
6.	`OnClose`	Событие возникает при закрытии окна.
7.	`OnKeyDown`	Событие наступает при нажатии любой клавиши.
8.	`OnKeyPress`	Событие наступает при нажатии клавиши символа.
9.	`OnKeyUp`	Событие возникает при отпускании любой клавиши.

Исходя из таблицы 9.1, для некоторого окна может быть создано девять процедур для обработки событий. Например, процедура по обработке события `OnCreate` (создание окна) может в это время выполнить некоторые подготовительные операции, такие как открытие файлов, инициализацию некоторых

переменных и т.д.

Разумеется, не обязательно писать обработчики событий для всех возможных событий. В этом случае, если отсутствует обработчик какого-нибудь события, то это событие просто не будет обработано вашим приложением. Например, если в приложении отсутствует обработчик события `OnKeyDown`, то на нажатие любой клавиши на клавиатуре приложение не будет реагировать.

9.2. Визуальное программирование в среде Lazarus

9.2.1. Создание графического приложения

Для создания графического приложения можно в главном меню открыть пункт **Файл -> Создать.** Во вкладке **Проект** выберите пункт **Приложение**, рис. 9.1

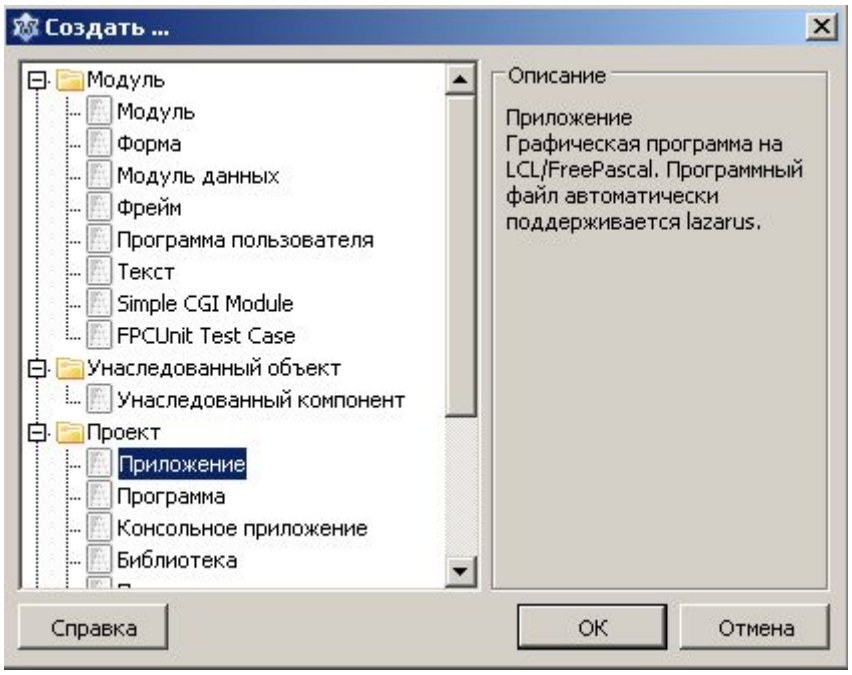


Рис. 9.1 Создание графического приложения

Нажмите **OK**.

Второй способ. Выберите пункт меню **Проект, Создать проект…** (рис. 9.2).

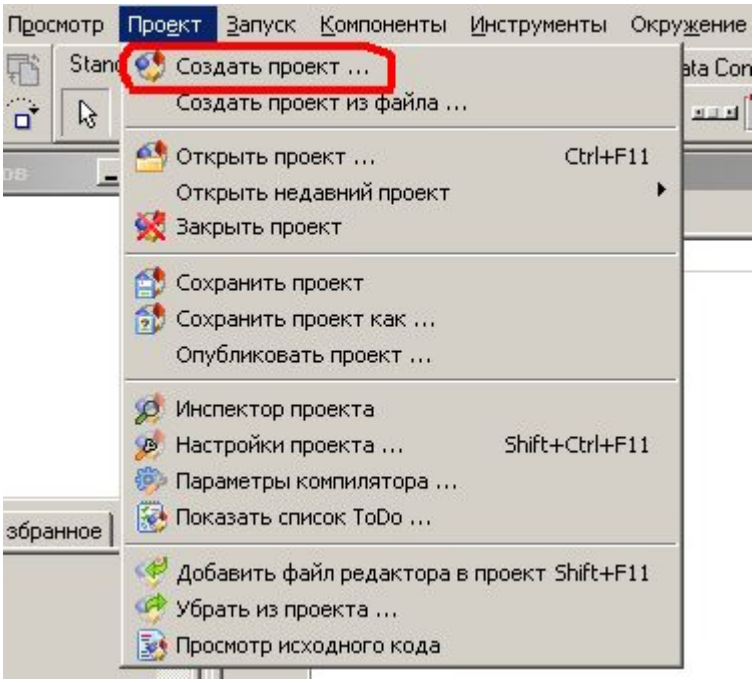


Рис. 9.2. Меню "Проект"

Выберите **Приложение** и нажмите **Создать**, рис. 9.3

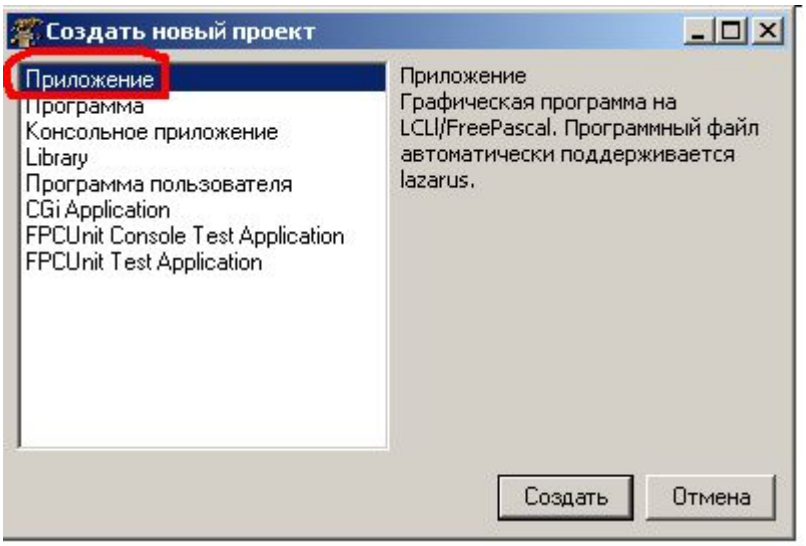


Рис. 9.3. Другой способ создания графического приложения

Кроме того, если в настройках окружения (**Окружение, Настройки окружения, Файлы**) снять галочку "Открывать последний проект при запуске" (рис. 9.4.), то Lazarus автоматически будет создавать новый проект GUI-приложения при каждом запуске.

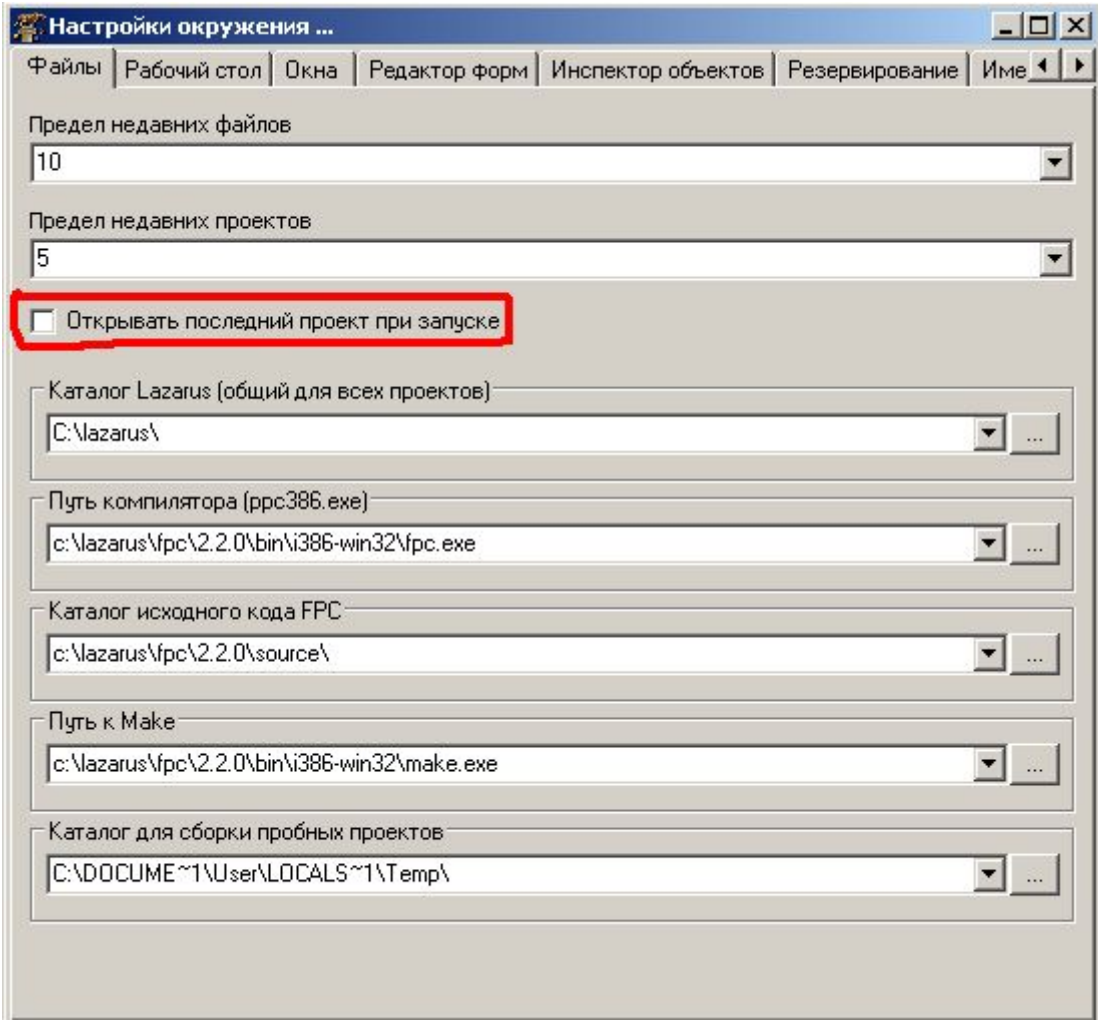


Рис. 9.4. Окно "Параметры IDE"

На экране появится пустая форма, рис. 9.5.

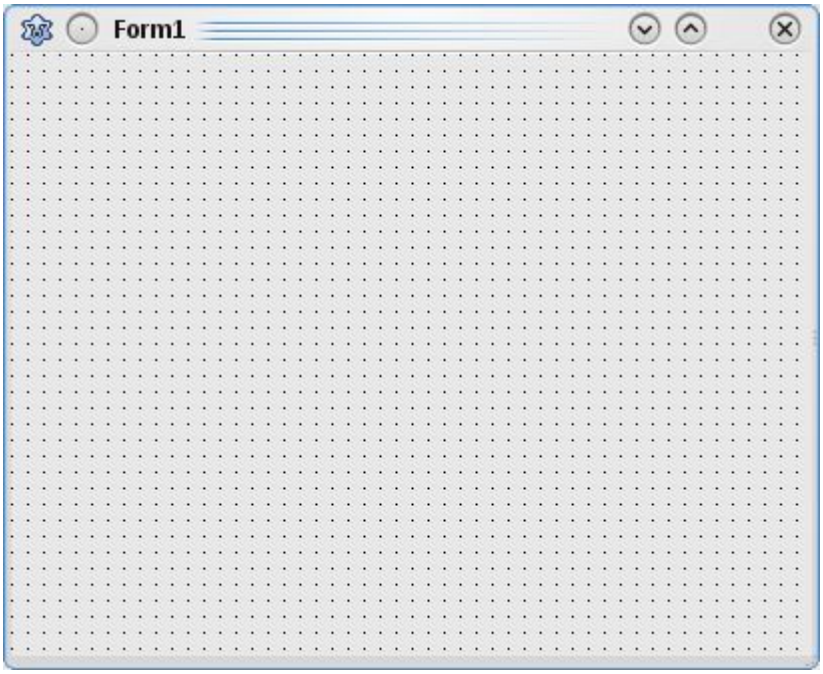


Рис. 9.5. Форма нового проекта

Если формы не видно, нажмите клавишу F12.

В окне редактора исходного кода Lazarus автоматически создаст следующий код:

```
unit Unit1;
{$mode objfpc}{$H+}
interface

uses
  Classes, SysUtils, LResources, Forms, Controls,
  Graphics, Dialogs;

type
  TForm1 = class(TForm)
  private
    { private declarations }
  public
    { public declarations }
end;

var
  Form1: TForm1;

implementation

initialization
  {$I unit1.lrs}

end.
```

Мы видим, что Lazarus создал для нас модуль со стандартным именем Unit1. А в теле модуля создается класс `TForm1` основанный на базовом классе `TForm` и описывает объект – стандартное графическое окно.

Сразу же сохраните свой проект в нужной папке. Если необходимо, то создайте новую папку. При сохранении помните, что имена модуля и проекта не должны совпадать, то есть файл проекта (lpr) и файл модуля (pas) должны иметь разные имена, потому что Lazarus присваивает имя модулю (в исходном коде) такое же, какое и имя файла модуля, а программе по имени файла проекта. Это необходимо сделать, иначе компилятор может впоследствии не найти

модуль по ссылке на него в файле проекта.

Исходный код основной программы (проекта) будет сохранен в файле с именем <Имя проекта>.lpr и имеет вид:

```
program project1;
{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Interfaces, // this includes the LCL widgetset
  Forms, Unit1, LResources
  { you can add units after this };

{$IFDEF WINDOWS}{$R project1.rc}{$ENDIF}

begin
  {$I project1.lrs}
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Здесь в объявлении `uses` перечисляются модули, подключаемые в проект по умолчанию. Кроме того, Lazarus автоматически включил имя только что созданного модуля. По умолчанию это `Unit1`. Далее директивой

```
{$IFDEF WINDOWS}{$R project1.rc}{$ENDIF}
```

для операционной системы Windows включается файл описания ресурсов. Под

ресурсами понимаются *ресурсы* приложения: пиктограммы, курсоры, битовые образы и пр.

В исполняемой части программы содержится еще одна директива

```
{$I project1.lrs}
```

с помощью которой подключается автоматически генерируемый файл ресурсов Lazarus. Заметьте, что это не файл ресурсов Windows.

Последние три оператора

```
Application.Initialize;
Application.CreateForm(TForm1, Form1);
Application.Run;
```

реализуют обращение к методам объекта `Application`. В объекте `Application` собраны данные и подпрограммы, необходимые для нормального функционирования программы в среде операционной системы. `Lazarus` автоматически создает объект-программу `Application` для каждого нового проекта. Метод `Initialize` отвечает за инициализацию (первоначальную настройку) приложения. Метод `CreateForm` создает главную форму приложения Form1 (окно приложения). После вызова метода `Run` осуществляется запуск нашего приложения.

Без особой необходимости не следует редактировать код проекта. Поэтому при создании проекта этот код не виден. `Lazarus` "скрывает" этот код от излишне любопытных. Но, если "очень хочется", то можно посмотреть его в меню **Проект->Просмотреть исходный код проекта** или открыть файл проекта с расширением .lpr любым текстовым редактором.

А исходный код нашего приложения будет сохранен в файле <имя моду-

ля>.pas (по умолчанию Unit1.pas).

Откомпилируйте и выполните свое приложение. Вы увидите пустое окно. Ну и что тут такого, скажете вы. По большому счету вы правы, ничего особенного. Но если вдуматься, то мы с вами только что, без видимых усилий, создали полноценное приложение с графическим интерфейсом! Окно вашей программы обладает всеми свойствами стандартных графических окон. Его можно свернуть, можно развернуть во весь экран, можно менять размеры. Окно можно перемещать в любое место экрана. Так же как и любое другое окно, оно имеет строку заголовка и системное меню. Не так уж и мало! И все это на основе стандартного класса TForm. В Lazarus имеется немало таких стандартных классов, на основе которых можно создавать приложения практически любой сложности!

Как уже отмечалось, класс описывает некоторый объект. Чтобы иметь возможность обращаться в программе к этому объекту, необходимо создать экземпляр класса. Это делается с помощью объявления

```
var
  Form1: TForm1;
```

Теперь мы можем работать с этим объектом, обращаясь к нему по имени `Form1`.

9.2.2. Форма и ее основные свойства

При создании нового проекта появляется пустая форма (рис. 9.5.). Следует заметить, что форма и окно приложения это не одно и то же. Форма – это то, что вы видите во время проектирования, а окно – это то, что видит пользователь во время выполнения вашего приложения. Таким образом, с помощью формы вы проектируете вид окна вашего приложения. Кроме того, форме соот-

ветствует класс, производный от базового класса TForm.

В главе VIII мы с вами рассматривали спецификаторы доступа `private`, `protected` и `public` с помощью которых можно управлять видимостью членов класса извне.

Имеется еще один спецификатор – `published` (опубликованный). В этом разделе помещаются свойства, которые пользователь может устанавливать на этапе проектирования, но доступны и во время выполнения, т.е. содержат так называемую RTTI – информацию (run-time type information). Обычно в разделе `published` размещаются только свойства и события.

Все компоненты Lazarus имеют опубликованный интерфейс, который отображается в инспекторе объектов.

Рассмотрим подробнее некоторые свойства формы. Как вы уже поняли, свойства формы, равно как и события, представляют собой опубликованную часть интерфейса класса формы. Их можно будет увидеть в инспекторе объектов во вкладке `Свойства`, рис 9.6.

Рис. 9.6. Вкладка "Свойства" Инспектора объектов

Свойство `Caption` – заголовок окна, представляет собой некоторый текст. По умолчанию присваивается значение "`Form1`". Желательно здесь указывать краткое содержание программы, например "Расчет заработной платы". Очень часто разработчики здесь выводят название программы и имя документа, связанного с этой программой.

Свойство `Name` – имя формы в программе. По этому имени можно обращаться к форме как к объекту в программе. По умолчанию присваивается имя `Form1`. Желательно давать осмысленные имена, особенно если в программе имеется несколько форм. Именование должно подчиняться требованиям языка Паскаль, т.е. имя не должно содержать недопустимые символы, пробелы и т.д. (см. главу 5).

Свойство `Left` устанавливает координаты левого верхнего угла окна по горизонтали.

Свойство `Top` устанавливает координаты левого верхнего угла окна по вертикали.

Сами размеры окна задаются свойствами `Height` и `Width`. Размеры задаются в пикселах.

Положение окна при запуске определяется свойством `Position`, оно может принимать следующие значения:

`poDesigned` – положение окна и его размеры остаются такими же, что и при проектировании;

`poDefault` – положение окна и его размеры определяется автоматически операционной системой;

`poDefaultPosOnly` – положение окна определяется автоматически операционной системой, а размеры соответствуют установкам при проектировании;

`poDefaultSizeOnly` – размеры окна определяется автоматически операционной системой, а положение соответствуют установкам при проектирова-

нии;

`poScreenCenter` или `poDesktopCenter` – окно выводится в центре экрана, размер определяется при проектировании;

`poMainFormCenter` – форма отображается в центре главной формы, размер определяется при проектировании, если имеется только одна главная форма, то этот параметр соответствует `poScreenCenter`;

`poOwnerFormCenter` – форма отображается в центре той формы, которая является владельцем данной формы.

Для того чтобы выбрать нужное значение, необходимо щелкнуть на название свойства. Появится раскрывающийся список, из которого и можно выбрать требуемое значение, рис. 9.7.

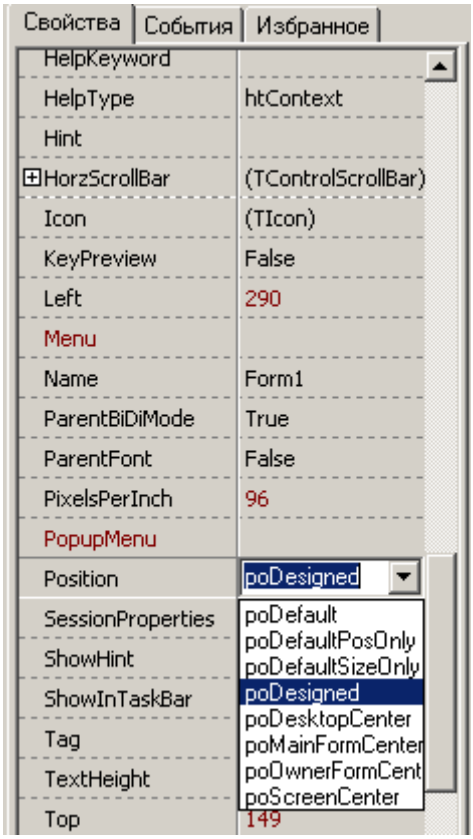


Рис. 9.7. Возможные значения свойства "Position"

Управлять положением формы на экране можно также с помощью свойства `Align`. Оно может принимать значения:

`alNone` – положение формы и его размеры не меняются;

`alBottom` – форма располагается внизу экрана;

`alLeft` – форма располагается в левой части экрана;

`alRight` – форма располагается в правой части экрана;

`alTop` – форма располагается вверху экрана;

`alClient` – форма занимает весь экран;

Перечисленные свойства относятся к так называемым простым свойствам, для задания которых достаточно просто ввести одно необходимое значение или выбрать из раскрывающегося списка также только одно значение.

Есть свойства, которые называются составными или сложными. Они помечаются в инспекторе объектов знаком "+" или кнопкой с троеточием. Например, свойство `Font` является составным. Составные свойства, как следует из названия, состоят из нескольких значений. На рис. 9.8 показан пример задания значений для свойства `Font`.

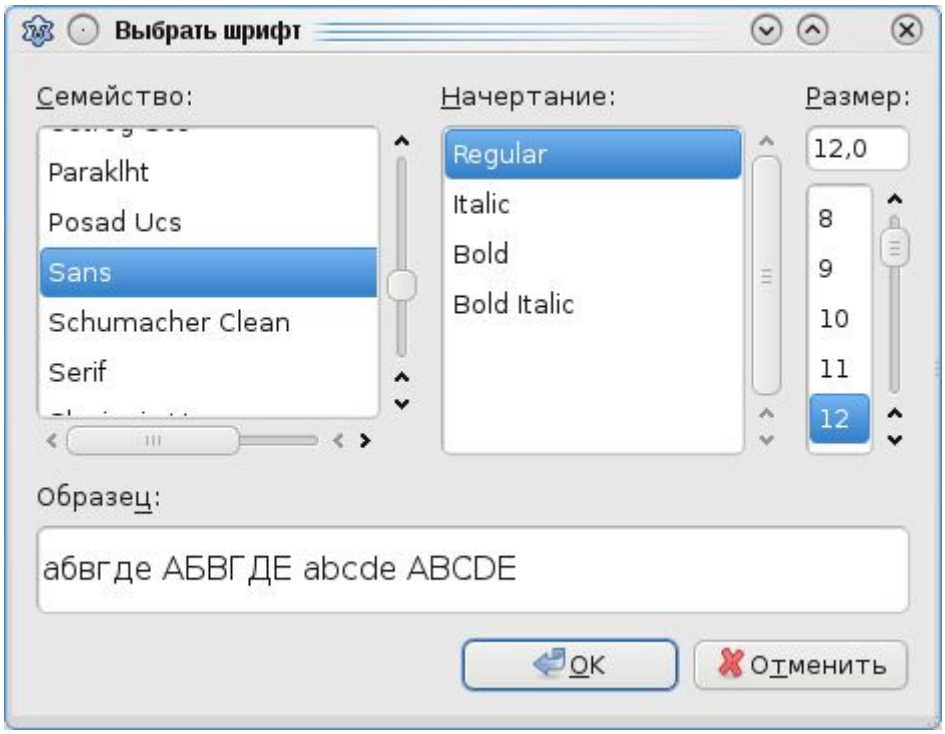


Рис. 9.8. Окно выбора шрифта (Linux)

С другими свойствами формы мы будем знакомиться в дальнейшем по мере необходимости.

Над объектами – экземплярами класса можно производить некоторые действия, причем перечень действий определен в самом классе. Никаких других действий с ними производить нельзя. Эти действия иначе называются методами класса.

Например, для класса формы существует метод Show – показать, метод Hide – скрыть, метод Close – закрыть.

В редакторе исходного кода Lazarus есть прекрасный инструмент CodeTools с помощью которого можно просмотреть все свойства и методы того или иного класса.

Посмотрим, например свойства и методы класса TForm1. Для этого в редакторе исходного кода после строк

```
initialization
{$I unit1.lrs}
```

наберите `Form1.` и немного подождите. Перед вами появится список всех свойств и методов класса TForm1, рис. 9.9.

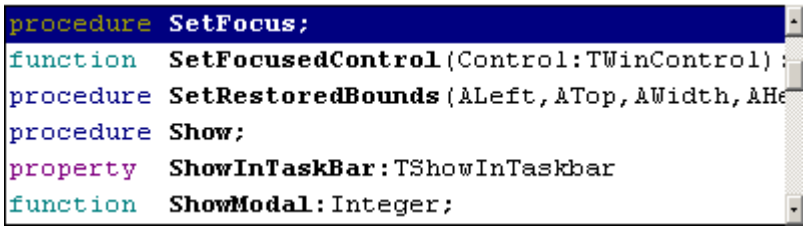


Рис. 9.9. Окно "Code Tools"

Пользуясь подсказками CodeTools можно значительно ускорить процесс набора кода программы.

Форма может быть модальной и немодальной. Модальная форма это такая форма, которая не позволяет открывать другие формы, пока она сама не будет закрыта. К таким формам чаще всего относятся диалоговые окна. Чтобы отобразить форму в модальном режиме необходимо вызвать метод `ShowModal`.

По умолчанию главная форма приложения открывается в немодальном режиме.

9.2.3. Компоненты

Суть визуального программирования заключается в том, что вы из набора компонентов библиотеки LCL переносите на форму нужные вам визуальные компоненты, настраиваете их под собственные потребности и формируете дизайн вашей программы. Компоненты также как и форма являются некоторыми графическими объектами. И каждый компонент реализован в виде класса. Например, компонент TLabel (надпись) реализован в виде класса. Название компонента соответствует имени класса. То есть, когда мы ведем речь о компоненте TLabel, мы подразумеваем класс TLabel.

Компоненты бывают видимыми и невидимыми. При проектировании форма выступает в роли контейнера для компонентов. При этом на форму можно разместить и невидимые компоненты.

Свойства и методы компонентов также отображаются в инспекторе объектов. Чтобы увидеть их, достаточно выделить требуемый компонент на форме.

9.2.4. Обработчики событий

Свойства объекта определяют его внешний вид (размер, шрифт, цвет и т.д.), а совокупность событий определяют поведенческую сторону объекта. Обработчиком события является процедура, которая выполняет те или иные действия в ответ на наступление события. Т.е. с помощью этой процедуры (обработчика события) реализуется реакция объекта на событие, например на щелчок мыши.

Таким образом, задача программиста сводится к тому, чтобы определить необходимые свойства объектов в его приложении и написать обработчики тех событий, на которые должен реагировать тот или иной объект приложения.

Инспектор объектов позволяет определить обработчики событий, на которые должна реагировать форма или ее компоненты. Во вкладке `События` в левой колонке приведен список всех событий для данного объекта. Не обязательно разрабатывать обработчики для всех событий. Как мы уже отмечали, если для некоторого события отсутствует его обработчик, то приложение просто не будет реагировать на это событие. Создайте новый проект в Lazarus. В инспекторе объектов откройте вкладку `События`. Выберите событие `OnCreate`, рис. 9.10. Это событие возникает при создании окна приложения. Щелкните по кнопке с троеточием. В редакторе кода появится заготовка кода процедуры обработчика данного события, рис. 9.11.

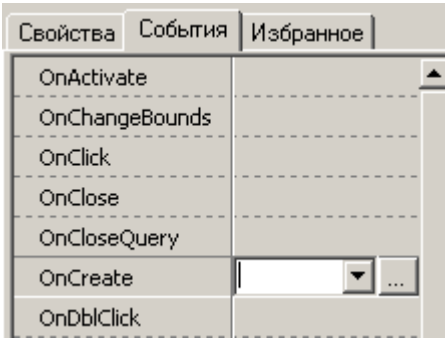


Рис. 9.10 Вкладка "События"

.

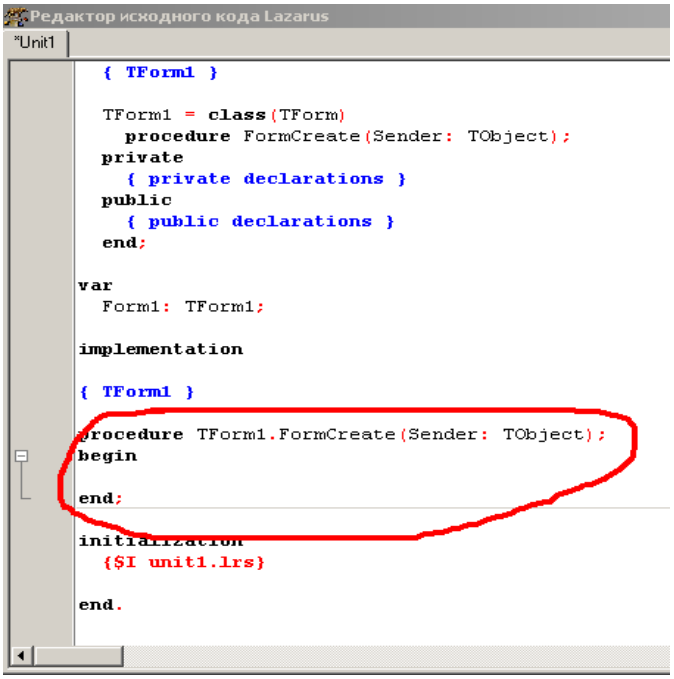


Рис. 9.11 Заготовка кода процедуры обработчика события

Обратите внимание, Lazarus автоматически присвоил процедуре имя `FormCreate`, присоединив к нему имя класса `TForm1`. В инспекторе объектов также появилось имя процедуры `FormCreate` – обработчика события `OnCreate`. Перейдите в редактор исходного кода, в обработчике события `FormCreate` введите следующий код:

```
Form1.Caption:= 'Моё первое графическое приложение';
```

Запустите свое приложение. Вы увидите, что в строке заголовка окна вместо стандартного `Form1`, появился ваш текст.

Из этого примера мы можем сделать один очень важный вывод. Оказывается свойства объекта можно изменять динамически во время выполнения приложения. Для доступа к свойству объекта необходимо указать имя этого объекта (в нашем случае формы `Form1`) и через точку имя свойства (`Caption`).

Далее, в инспекторе объектов выберите событие `OnClick`. В обработчике события введите код:

```
Form1.Caption:= 'Зачем ты на меня нажал?';
```

Запустите приложение. Щелкните мышью по окну вашего приложения. Вы видите, что в строке заголовка окна текст меняется. Т.е. можно воочию убедиться, что ваше приложение на самом деле реагирует на нажатие мыши.

Выберите теперь, например, событие `OnDblClick`. Если вы раскроете раскрывающийся список, то вы увидите список уже имеющихся обработчиков событий, рис. 9.12.

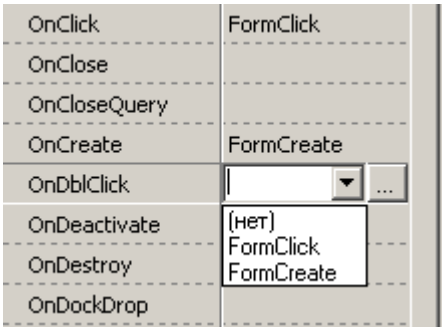


Рис. 9.12. Список уже имеющихся обработчиков событий

Если вам нужно, чтобы ваше приложение реагировало на какое-то событие точно таким же образом, как и на некоторое другое событие, и обработчик этого события уже имеется, то вы можете просто выбрать из списка нужный вам обработчик события. В нашем примере для события `OnDblClick` вы можете выбрать обработчик `FormClick` или `FormCreate`. Этим можно избежать неоправданного дублирования кода.

9.2.5. Простейшие компоненты

Рассмотрим теперь некоторые компоненты библиотеки LCL Lazarus. Доступ к компонентам в Lazarus организован через палитру компонентов, рис. 9.13.

Рис. 9.13. Палитра компонентов Lazarus

Палитра компонентов состоит из страниц или вкладок, в которых и находятся компоненты, сгруппированные по некоторому признаку. На рисунке 9.13 показана страница `Standard`, в которой расположены наиболее часто используемые, так называемые стандартные компоненты. По названию страницы можно понять, какие компоненты находятся в той или иной вкладке. Например, ясно, что на странице `Dialogs` находятся компоненты для организации диалога с пользователем.

В соответствии с предназначением книги, мы не будем рассматривать все компоненты (для этого необходимо писать новую книгу примерно такого же объема, что и эта!).

9.2.5.1. Компонент `TLabel`

Этот компонент, его еще называют меткой или надписью, позволяет ото-

бразить некоторый текст на экране. Основные свойства этого компонента:

- `Name` – имя в приложении, используется для доступа к компоненту.
- `Caption` – текст, который будет отображаться в поле компонента.
- `Color` – цвет фона надписи.
- `Font` – это свойство является составным. С помощью этого свойства можно настроить параметры шрифта, для отображения текста надписи: цвет букв (`Color`), размер шрифта (`Size`), название шрифта (`Name`) и т.д.
- `Alignment` – выравнивание текста надписи. Здесь `taLeftJustify` – по левому краю, `taCenter` – по центру, `taRightJustify` – по правому краю.
- `Height` – высота надписи в пикселах.
- `Width` – ширина надписи в пикселах.
- `AutoSize` – автоматическое изменение размера надписи в зависимости от длины текста. `True` – вертикальный и горизонтальный размер надписи зависит от длины текста, `False` – размеры компонента устанавливаются вручную, выравнивание текста производится свойством `Alignment`.
- `WordWrap` – если имеет значение `True`, то производится перенос текста по словам, т.е. текст надписи отображается в несколько строк, иначе текст выводится в одну строку.
- `Visible` – определяет видимость компонента во время выполнения приложения. По умолчанию имеет значение `True`, т.е. компонент виден в момент выполнения приложения.

Таковы наиболее часто используемые свойства компонента `TLabel`. В инспекторе объектов вы найдете еще множество свойств, разобраться с которыми вы можете самостоятельно. Могу порекомендовать вам книгу Архангельского А.Я. "Программирование в Delphi 7" [4], в которой все свойства и не только этого компонента разобраны очень и очень подробно. Хотя там изложение идет для Delphi, тем не менее, большинство свойств компонентов совпадают.

Наиболее важными свойствами являются `Name` и, естественно `Caption`.

В `Caption` вы задаете текст надписи, а в `Name` имя, которое будет использоваться вами для доступа к компоненту в программе.

Рассмотрим пример. Практически в любой книге посвященной программированию для начинающих рассматривается пример, которому даже присвоено имя – `Hello World`. Эта простейшая программа просто выводит текст приветствия на экран. Не отступим от этой традиции и мы. Давайте напишем эту программу. Запустите Lazarus, создайте новый проект. Если вы в настройках окружения убрали галочку "Открывать последний проект при запуске", то новый проект создастся автоматически. Перенесите на форму компонент `TLabel`. Это делается очень легко и просто. В палитре компонентов на странице `Standard` (по умолчанию открыта именно эта страница) найдите компонент `TLabel`, рис. 9.14.

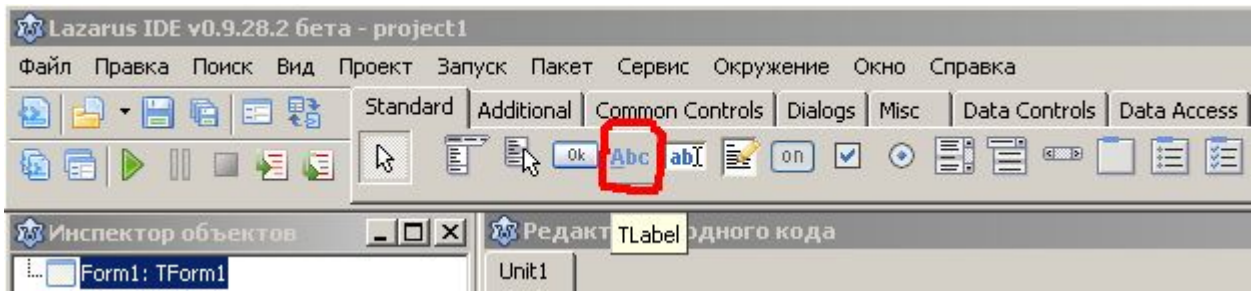


Рис. 9.14. Страница "Standard" палитры компонентов

Кроме того, если немного подержать указатель мыши над компонентом, выведется всплывающая подсказка с именем компонента. Нажмите на значок компонента, затем щелкните мышью на свободном месте формы. Выбранный вами компонент будет перенесен на форму. Он будет также автоматически выделенным, так что все его свойства будут доступны в инспекторе объектов.

Нажмите на свойство `Caption`, его значение (по умолчанию `Label1`) будет выделено синим цветом. Можете сразу набирать текст надписи. Наберите `"Hello, World!"` Этот текст появится не только в окошке ввода свойства `Caption`, но и сразу будет виден в поле текста компонента, рис. 9.15.

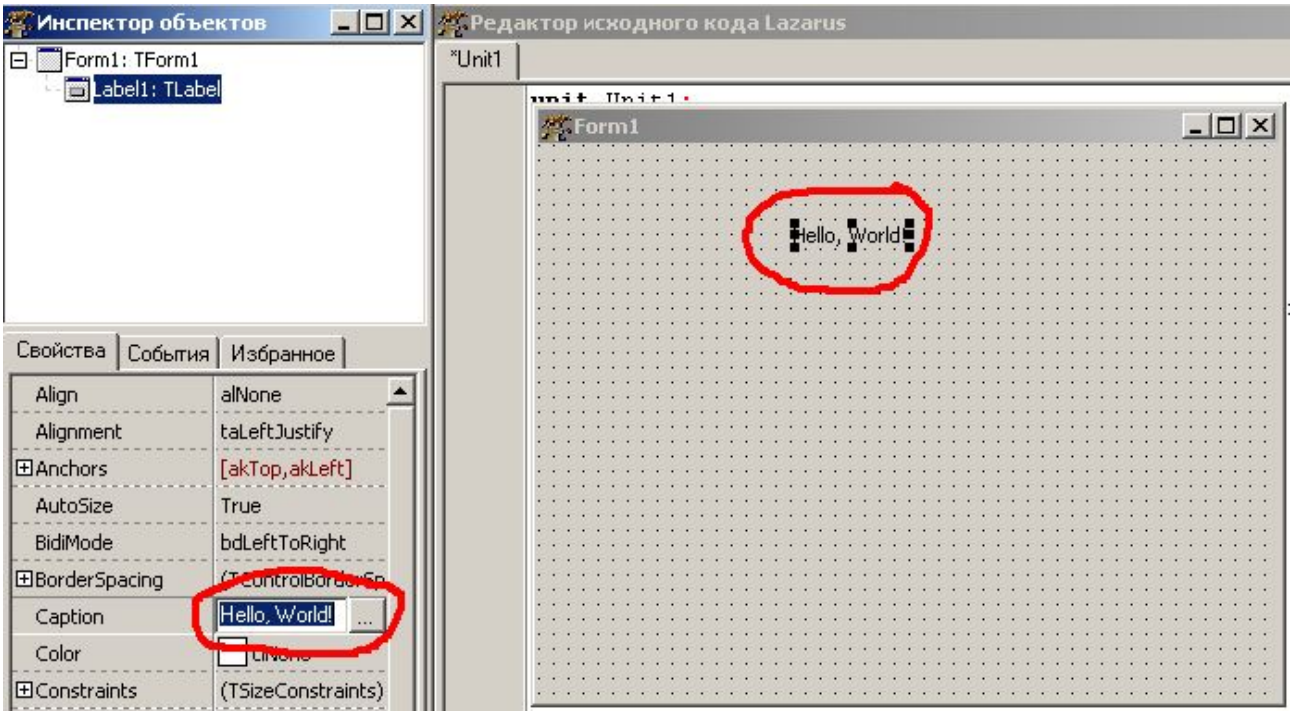


Рис. 9.15. Пример ввода текста надписи

Посмотрим, какие изменения произошли в коде по сравнению с программой с пустым окном (см. раздел 9.2.1). Мы видим, что Lazarus добавил в описание класса формы `TForm1` экземпляр класса `TLabel` – `Label1`, причем автоматически, без нашего участия! Как только мы добавляем на форму тот или иной компонент, Lazarus тут же добавит экземпляры классов соответствующих компонентов, формируя таким образом описание класса формы вашего приложения.

Нажмите клавишу F9. После компиляции и запуска вашего приложения на экран будет выведено окно с надписью "Hello, World!", рис. 9.16.

Рис. 9.16. Окно графического приложения с компонентом TLabel

Рассмотрим еще одно интересное свойство, которое довольно часто применяется. Это свойство `Visible`. С помощью этого свойства можно управлять видимостью компонента во время работы приложения.

Давайте создадим приложение, в котором будем динамически менять свойства компонента, а также посмотрим последовательность действий для разработки обработчика событий.

Создайте новый проект. Перенесите на форму три компонента `TLabel`. Компонент `Label1` разместите в центре формы. Свойство `Caption` оставьте без изменения. Компонент `Label2` разместите в правом верхнем углу формы, примерно так, как показано на рисунке 9.17.

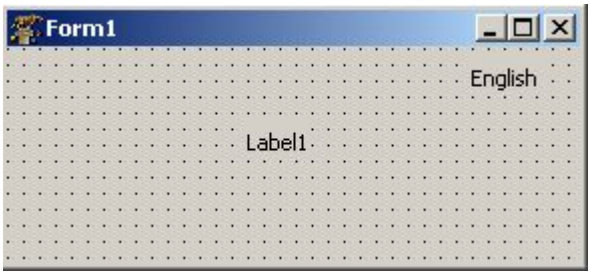


Рис. 9.17. Форма программы

Свойство `Caption` измените на "`English`". Компонент `Label3` разместите прямо на компонент `Label2` так, чтобы он закрывал его (оказывается так можно делать!). Свойство `Caption` измените на "`Русский`".

Выделите форму. Проще всего это сделать в инспекторе объектов, рис. 9.18.

Откройте вкладку `События`, выберите событие `OnCreate`. Нажмите на кнопку с троеточием или дважды щелкните по полю с треугольником. Lazarus тут же добавит в описание класса формы соответствующий метод, а в редакторе исходного кода появится заготовка для реализации этого метода – обработчика события, рис. 9.19.

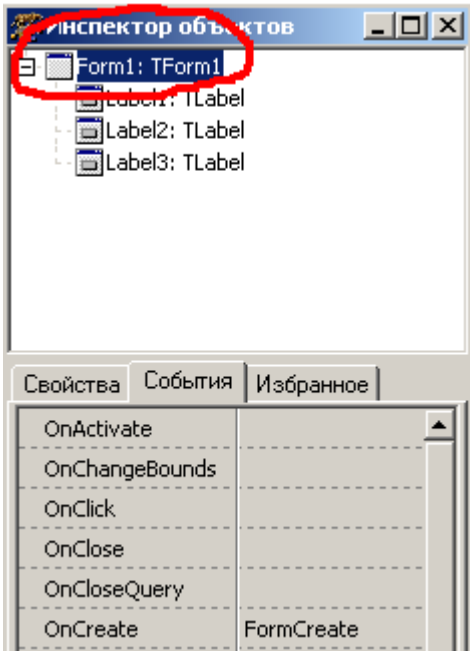


Рис. 9.18. Инспектор объектов. События формы

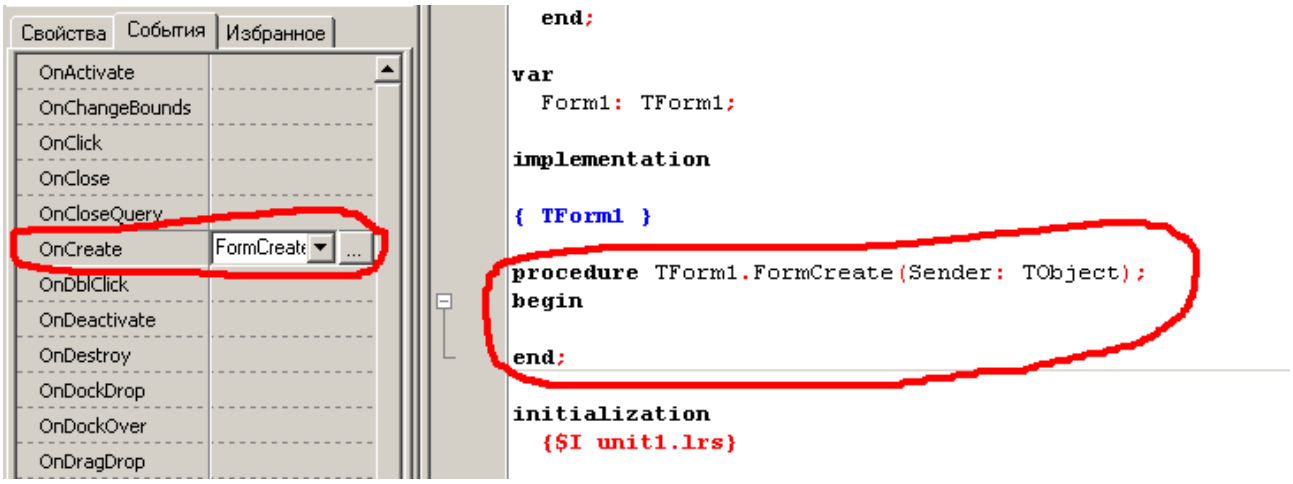


Рис. 9.19. Заготовка обработчика события

Обработчик событий это обычная процедура. В обработчике события `OnCreate` между операторами `begin` и `end` введите следующий код (для удобства привожу текст процедуры полностью:

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

```
    Label1.Caption:= 'Привет, Мир!';
    Label2.Visible:= true;
    Label3.Visible:= false;
  end;
```

Выделите компонент `Label2`, опять же проще будет через инспектор объектов. Во вкладке `События`, выберите событие `OnClick`. В его обработчике введите код:

```
  procedure TForm1.Label2Click(Sender: TObject);
  begin
    Label2.Visible:= false;
    Label3.Visible:= true;
    Label1.Caption:= 'Hello, World!!';
  end;
```

Теперь выделите компонент `Label3`, выберите событие `OnClick`. В его обработчике введите код:

```
  procedure TForm1.Label3Click(Sender: TObject);
  begin
    Label2.Visible:= true;
    Label3.Visible:= false;
    Label1.Caption:= 'Привет, Мир!';
  end;
```

Запустите приложение, рис. 9.20.

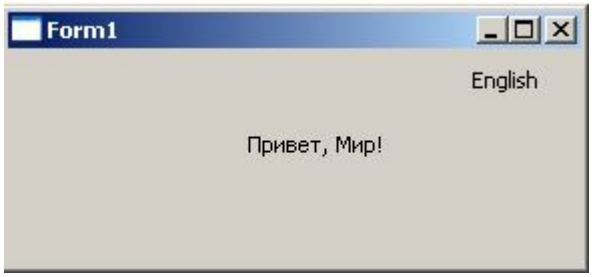


Рис. 9.20. Окно приложения

Если нажать на надпись `English`, приветствие выведется на английском языке, а надпись изменится на `Русский`, рис. 9.21.

Рис. 9.21. Окно приложения после нажатия на надпись "English"

Если мы снова нажмем на надпись `Русский`, текст приветствия вновь выведется на русском языке, а текст надписи изменится на `English`.

Вам, наверное, приходилось видеть программы, в которых реализован многоязычный интерфейс. Переключение с одного языка на другой в них реализован примерно по такому же методу, который мы с вами только что реализовали.

По уже установившейся нашей с вами традиции посмотрим еще раз код программы и попробуем найти недостатки. Это код настолько прост, что, кажется, здесь нечего и улучшать. В общем-то, вы правы. Единственное замечание, которое можно сделать относится, скорее, к стилю программирования. Я имею в виду способы именования объектов в программе. Есть довольно много различных рекомендаций при выборе имен объектов. В частности, для именования переменных многие используют так называемую венгерскую нотацию, то есть правила написания имен переменных. Смысл этих правил в том, чтобы по

имени переменной можно было определить ее тип. Для этого используют префиксы, например во многих наших программах для обозначения указателей мы использовали префикс " p "(pHead, pCurrent и т.д.). Исходя из этого, для переменных целого типа можно применять префикс " i ", для переменных логического типа префикс " b " и т.д.

Для компонентов тоже можно использовать префиксы, например для TLabel уместно использовать префикс "lb".

Существуют и другие способы именования, тем более что есть немало и ярых противников венгерской нотации. Не стану их описывать. При желании вы можете найти все это в Интернете.

В принципе, каких-то жестких требований на этот счет нет. Главное, вы должны найти, выработать свой, собственный стиль, понятный не только вам, но и другим. Будете вы использовать префиксы, не будете использовать это ваше дело. Основной принцип здесь – понятность, то есть все обозначения должны быть понятны и не только вам, но и другим.

Представьте себе, что вы написали какую-то достаточно большую программу. Через несколько месяцев, а, может быть лет, вам или кому-то другому (конечно, с вашего разрешения!) понадобилось ее модернизировать. Если в вашей программе будут обозначения, ну типа "aaa" или "x123", то в вашей программе будет трудно разобраться не только тому другому, но и вам самому!

К хорошему стилю также относится наличие в программе комментариев. И здесь не должно быть крайностей. Сухие односложные комментарии типа "x – это переменная" одна крайность. Многочисленные, многословные комментарии чуть ли не после каждого оператора другая крайность. Важно найти золотую середину!

Не забывайте также об использовании отступов при записи кода программы. Во всех наших программах этой книги мы использовали отступы и комментарии.

Вернемся к нашей программе. Здесь используются три компонента

TLabel. Lazarus присвоил им по умолчанию имена Label1, Label2 и Label3. Понятно, что это надписи. Поэтому применение слова Label вполне логично и оправдано. Но дальше идет простая нумерация! А если в программе 10, 20, 100 надписей? Очень скоро вы запутаетесь, что означает, например Label5 и, например, Label12.

Поэтому при небольшом числе одинаковых компонентов вполне допустимо использовать имена по умолчанию, но все же лучше, особенно если количество одинаковых компонентов больше трех, выбирать информативные имена, которые отражают суть того или иного объекта. При этом не бойтесь придумывать имена в русском контексте, хотя записывать их приходится английскими буквами. Например, имя Stroka будет понятна любому русскоязычному программисту. Хотя, если вы будете работать, ну скажем, в Microsoft, то за этот идентификатор вас слегка пожурят и попросят изменить на другое, более понятное им имя!

Исходя из этих рассуждений, перепишем нашу программу в виде:

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes,  SysUtils,  LResources,  Forms,  Controls,
  Graphics, Dialogs, StdCtrls, Windows;
type
  { TForm1 }
  TForm1 = class(TForm)
    L_Hello: TLabel;  // здесь будет выводиться само приветствие
    L_English: TLabel; // для переключения на английский язык
    L_Russian: TLabel; // для переключения на русский язык
    procedure FormShow(Sender: TObject);
```

```
    procedure L_EnglishClick(Sender: TObject);
    procedure L_RussianClick(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;

implementation

{ TForm1 }

procedure TForm1.FormShow(Sender: TObject);
begin
  L_Hello.Caption:= 'Привет, Мир!';
end;


procedure TForm1.L_EnglishClick(Sender: TObject);
begin
  L_English.Visible:= false;
  L_Russian.Visible:= true;
  L_Hello.Caption:= 'Hello, World!';
end;
procedure TForm1.L_RussianClick(Sender: TObject);
begin
  L_English.Visible:= true;
  L_Russian.Visible:= false;
  L_Hello.Caption:= 'Привет, Мир!';
end;
```

```
initialization
  {$I unit1.lrs}
end.
```

Повторяю, вы вольны выбирать любые имена для своих объектов в программе, в частности, и здесь вы можете придумать другие имена для компонентов, более логичные, информативные и понятные с вашей точки зрения. Этот пример лишь иллюстрация к тому, что было сказано выше.

Имеются ли различия при создании собственных классов в программах с графическим интерфейсом по сравнению с консольными приложениями?

Давайте реализуем последний пример из главы V с конструкторами и виртуальными функциями в виде GUI-приложения. Для этого создайте новый проект и поместите на форму четыре компонента `TLabel`. Код программы:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,   Forms,
  Controls, Graphics, Dialogs, StdCtrls, Unit2;
type
  { TForm1 }
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    procedure FormShow(Sender: TObject);
  private
```

```
    { private declarations }
  public
    { public declarations }
  end;

type
  TStudent = class(THuman)
    private
      group: string;
    public
      constructor Create(gr: string);
      function GetData: string;
      function Status: string; override;
  end;
type
  TProfessor = class(THuman)
    private
      kafedra: string;
    public
      constructor Create(kaf: string);
      function GetData: string;
      function Status: string; override;
  end;
var
  Form1: TForm1;
  Student: TStudent;
  Professor: TProfessor;
  fname: string;
```

```
implementation

function TStudent.Status: string;
begin
  Result:= ' – студент';
end;

constructor TStudent.Create(gr: string);
begin
  inherited Create('Андрей', 'Аршавин');
  group:= gr;
end;

function TStudent.GetData: string;
begin
  Result:= inherited GetData + ', группа ' + group;
end;

constructor TProfessor.Create(kaf: string);
begin
  inherited Create('Алексей', 'Попов');
  kafedra:= kaf;
end;
function TProfessor.Status: string;
begin
  Result:= ' - преподаватель';
end;

function TProfessor.GetData: string;
```

```
begin
  Result:= inherited GetData + ', кафедра ' + kafedra;
end;

{ TForm1 }

procedure TForm1.FormShow(Sender: TObject);
begin
  Student:= TStudent.Create('"Арсенал"');
  fname:= Student.GetData;
  Label1.Caption:= 'Это: ' + fname;
  Professor:= TProfessor.Create('"Телеканал Россия 2"');
  fname:= Professor.GetData;
  Label2.Caption:= 'Это: ' + fname;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
  Student.group:= '"ПОВТАС-1/09"';
  fname:= Student.GetData;
  Label3.Caption:= 'Это: ' + fname;
  Professor.name:= 'Иван';
  Professor.fam:= 'Иванов';
  Professor.kafedra:= '"Программирование"';
  fname:= Professor.GetData;
  Label4.Caption:= 'Это: ' + fname;
  Student.Free;
  Professor.Free;
end;
initialization
```

```
  {$I unit1.lrs}
end.

unit Unit2;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
  type

  { THuman }

  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      constructor Create(n, f: string);
      function GetData: string;
      function Status: string; virtual; abstract;
   end;

implementation

{ THuman }
```

```
constructor THuman.Create(n, f: string);
begin
  Fname:= n;
  Ffam:= f;
end;

procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
  Fname:=AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam=AValue then exit;
  Ffam:=AValue;
end;
function THuman.GetData: string;
begin
  Result:= name + ' ' + fam + Status;
end;
end.
```

Мы видим, что практически никакой разницы нет, за исключением того, что реализации методов классов следует помещать в раздел `implementation` модуля.

9.2.5.2. Кнопки TButton, TBitBtn и TSpeedButton

Кнопка является элементом управления, предназначенным для запуска каких-то действий или команд. Щелчок по кнопке мышью вызывает событие

OnClick в обработчике которого программист и инициирует выполнение каких-либо действий, команд и процедур. В палитре компонентов имеются две разновидности кнопок. На странице Standard имеется компонент TButton, а на странице Additional компонент TBitBtn.

Рассмотрим свойства, присущие только кнопке. Такие его свойства, как Name, Left, Top, Height, Width, Font, Visible и др. имеют тот же смысл, что и для рассмотренного ранее компонента TLabel.

- Caption – текст, отображаемый на кнопке.
- Enabled – признак доступности кнопки. По умолчанию имеет значение True, то есть кнопка доступна. Если False, то кнопка в данный момент недоступна. Надпись на кнопке имеет блеклый вид, нажатие на кнопку не приводит ни к каким действиям, даже если имеется обработчик события OnClick.
- Default – если установлено значение True, то нажатие клавиши Enter будет эквивалентно щелчку по кнопке мышью.
- Cancel – если установлено значение True, то нажатие клавиши Esc будет эквивалентно щелчку по кнопке мышью.

Компонент TBitBtn отличается от TButton тем, что на нем можно отображать пиктограммы. Перечисленные выше свойства для TButton имеют место и для TBitBtn. Кроме этого, этот компонент имеет и свои особые свойства.

- Kind – задает тип кнопки. Имеются несколько предопределенных типов кнопки с готовой пиктограммой и текстом:
  - ✓ bkAbort – с текстом " Прервать ".
  - ✓ bkAll – с текстом "Все".
  - ✓ bkCancel – с текстом "Отмена".
  - ✓ bkClose – с текстом "Закрыть".
  - ✓ bkCustom – произвольный текст, устанавливается программистом.
  - ✓ bkHelp – с текстом "Справка".

- ✓ `bkIgnore` – с текстом "Пропуск".
- ✓ `bkNo` – с текстом "Нет".
- ✓ `bkNoToAll` – с текстом "Нет для всех".
- ✓ `bkOK` – с текстом "ОК".
- ✓ `bkRetry` – с текстом "Повтор".
- ✓ `bkYes` – с текстом "Да".
- ✓ `bkYesToAll` – с текстом "Да для Всех".

- `Glyph` – если вас не устраивают предлагаемые рисунки, вы можете выбрать другие. Будет открыто диалоговое окно, необходимо указать путь к этому рисунку.
- `Margin` – задает расстояние от края кнопки до рисунка (в пикселах). По умолчанию -1. В этом случае рисунок и текст располагаются в центре.
- `Layout` – определяет положение рисунка на кнопке. Можно выбрать:
  - ✓ `blGlyphLeft` – слева.
  - ✓ `blGlyphRight` – справа.
  - ✓ `blGlyphBottom` – снизу.
  - ✓ `blGlyphTop` – сверху.
- `Spacing` – задает расстояние в пикселах между рисунком и текстом кнопки.

Рассмотрим пример. Положите на пустую форму две кнопки `TButton` и одну `TBitBtn`. Для кнопки `Button1` установите свойство `Caption=`"Показать кнопку", для кнопки `Button2` установите свойство `Caption=`"Сделать доступной". Для кнопки `BitBtn1` установите свойство `Kind=` `bkClose`, `Enabled=` `False`, `Visible=` `False`, рис. 9.22.

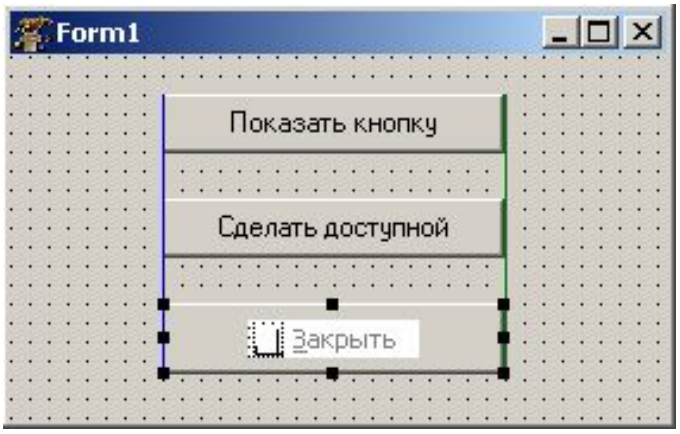


Рис. 9.22. Форма приложения

В обработчик события `OnClick` для `Button1` введите код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  BitBtn1.Visible:= true;
end;
```

В обработчик события `OnClick` для `Button2` введите код:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  BitBtn1.Enabled:= true;
end;
```

Откомпилируйте и запустите приложение. Вы видите, что кнопка `BitBtn1` сразу после запуска не видна. Нажмите на кнопку "`Показать кнопку`". Кнопка `BitBtn1` станет видна, но будет недоступна. Теперь нажмите на кнопку " `Сделать доступной`". Кнопка `BitBtn1` станет доступна. Закройте с ее помощью приложение.

Кнопки `TSpeedButton` (она находится на странице `Additional`)

обычно используются в качестве быстрых кнопок в панелях инструментов. На них также можно поместить некоторое изображение. Разумеется, их можно использовать и как обычные кнопки. Отличительной особенностью этих кнопок является возможность фиксации нажатого состояния. Для этого используются свойства `GroupIndex` и `AllowAllUp`.

По умолчанию свойство `GroupIndex` имеет значение 0. В этом случае кнопка ведет себя как обычная кнопка. Если значение `GroupIndex > 0`, то при нажатии на кнопку она останется в нажатом состоянии. Если свойству `AllowAllUp` присвоить значение `= true`, то при повторном нажатии на кнопку она возвратится в нормальное состояние.

Если присвоить свойствам `GroupIndex` нескольких кнопок одинаковое значение `> 0`, то при нажатии следующей кнопки она останется в нажатом состоянии, а предыдущая кнопка вернется в обычное отжатое состояние (свойство `AllowAllUp` должно иметь значение `= true` для всех этих кнопок).

Узнать в каком состоянии находится кнопка (нажатом или отжатом) можно по свойству `Down`. Если `Down = true`, то кнопка в нажатом состоянии.

9.2.6. Организация ввода данных. Однострочные редакторы TEdit, TLabeledEdit

Любая программа в своей работе использует какие-то исходные данные. Чаще всего эти данные вводятся пользователем с клавиатуры.

Для организации ввода используются компоненты `TEdit`, `TLabeledEdit` и др. Сигналом завершения ввода и начала обработки введенных данных служит обычно нажатие пользователем кнопки. Для завершения ввода также часто используется клавиша `Enter`.

9.2.6.1. Компонент TEdit

С помощью этого компонента можно вводить и редактировать некоторый текст. Этот компонент носит также название однострочный редактор.

- Text – вводимое значение содержится в этом свойстве в виде строки символов. По умолчанию содержит строку "Edit1".
- AutoSelect – если установлено значение True, то при передаче фокуса на компонент, весь текст в поле редактирования будет выделен.
- ReadOnly – если установлено значение True, то текст в поле редактирования доступен только для чтения, при этом сохраняется возможность копирования текста в буфер обмена.

Рассмотрим пример, консольный вариант которого мы рассматривали в разделе 6.3.1.3. Сделаем этот же пример в виде графического приложения. Как вы помните, в примере имитировался случай авторизации пользователя для входа в систему.

Перенесите на форму компонент TEdit, очистите свойство Text. Также нам понадобятся кнопка и надпись. Расположите их на форме так, как показано на рисунке 9.23, соответственно изменив свойства Caption формы, надписи и кнопки.

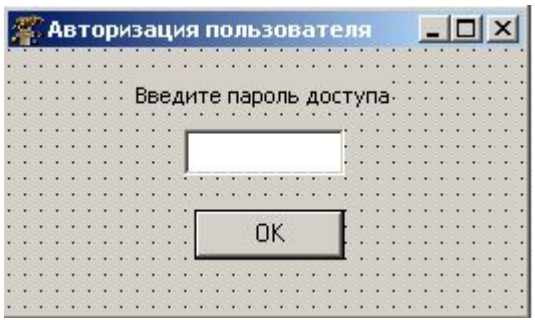


Рис. 9.23. Форма приложения

В редакторе исходного кода в обработчик события OnCreate формы введите следующий код:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  passw:= '1234';
  n:= 1;
end;
```

А в обработчик `OnClick` кнопки следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if passw <> Edit1.Text then
  begin
    Edit1.SetFocus;
    if n = 1 then
    begin
      ShowMessage('Неправильный пароль!');
      inc(n);
    end
    else
    begin
      if n = 3 then
      begin
        ShowMessage('Вам отказано в доступе');
        Close();
      end
      else
      begin
        Stroka:= 'Вы ' + IntToStr(n) + ' раза ввели неправиль-
ный пароль';
```

```
        Stroka:= Stroka + ' После 3-й попытки вам будет отказано в доступе';
        ShowMessage(Stroka);
        inc(n);
      end
    end
  end
  else
  begin
    ShowMessage('Вы успешно авторизовались!');
    Close();
  end;
end;
```

В разделе описания переменных модуля, добавьте следующие описания переменных:

```
var
  Form1: TForm1;
  passw: string[4];
  Stroka: string;
  n: integer;
```

Обратите внимание – чтобы введенные нами переменные были доступны в обработчиках, мы поместили их объявления в разделе `interface`.

Запустите приложение и поэкспериментируйте с вводом различных паролей. Правильный пароль "1234".

В программе мы использовали функцию `IntToStr(n)`, которая переводит целое число в ее строковое представление и стандартную процедуру `ShowMessage`, для вывода сообщения с кнопкой `OK`. Единственным парамет-

ром этой процедуры является строка символов.

Можно ли улучшить программу?

Многие пользователи привыкли завершать ввод в однострочном редакторе нажатием клавиши `Enter`. Для того чтобы реализовать эту возможность просто установите у кнопки свойство `Default` равным `True` в инспекторе объектов. Однако учтите, если у вас в программе несколько кнопок, то сработает та кнопка, которая имеет в этот момент фокус ввода. Отслеживать фокус при большом числе компонентов становится делом довольно обременительным. Выход заключается в написании обработчика в компоненте `Edit1` на событие `OnKeyPress`. В этом обработчике имеется параметр `Key`, которому система передает код нажатой клавиши. Код клавиши `Enter` равен #13 (см. табл. 6.4. в разделе 6.3.1). Отсюда обработчик будет выглядеть следующим образом:

```
procedure TForm1.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
  if Key <> #13 then exit;
    if passw <> Edit1.Text then
  begin
    Edit1.SetFocus;
    if n = 1 then
    begin
      ShowMessage('Неправильный пароль!');
      inc(n);
    end
    else
    begin
      if n = 3 then
      begin
```

```
        ShowMessage('Вам отказано в доступе');
        Close();
      end
      else
      begin
        Stroka:= 'Вы ' + IntToStr(n)+ ' раза ввели неправильный пароль';
        Stroka:= Stroka + ' После 3-й попытки вам будет отказано в доступе';
        ShowMessage(Stroka);
        inc(n);
      end
    end
  end
  else
  begin
    ShowMessage('Вы успешно авторизовались!');
    Close();
  end;
end;
```

От обработчика `Button1Click` код отличается лишь одним первым оператором

```
if Key <> #13 then exit;
```

К сожалению, здесь имеет место почти полное дублирование кода. Чтобы избежать этого создадим процедуру и в обработчиках будем просто ее вызывать. Окончательно программа будет иметь вид:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,   Forms,
  Controls, Graphics, Dialogs,
  StdCtrls;

type
  { TForm1 }
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure  Edit1KeyPress(Sender:  TObject;  var  Key:
char);
    procedure FormCreate(Sender: TObject);
    procedure avtorization;
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;
  passw: string[4];
  Stroka: string;
```

```
  n: integer;

implementation
{ TForm1 }
procedure TForm1.avtorization;
begin
  if passw <> Edit1.Text then
  begin
    Edit1.SetFocus;
    if n = 1 then
    begin
      ShowMessage('Неправильный пароль!');
      inc(n);
    end
    else
    begin
      if n = 3 then
      begin
        ShowMessage('Вам отказано в доступе');
        Close();
      end
      else
      begin
        Stroka:= 'Вы ' + IntToStr(n) + ' раза ввели неправиль-
ный пароль';
        Stroka:= Stroka + ' После 3-й попытки вам будет отказано в
доступе';
        ShowMessage(Stroka);
        inc(n);
```

```
      end
    end
  end
  else
  begin
    ShowMessage('Вы успешно авторизовались!');
    Close();
  end;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  passw:= '1234';
  n:= 1;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  avtorization;
end;
procedure TForm1.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
  if Key <> #13 then exit;
  avtorization;
end;
initialization
  {$I unit1.lrs}

end.
```

9.2.6.2. Компонент TLabeledEdit

В палитре компонентов на странице Additional имеется еще одна разновидность однострочного редактора TLabeledEdit. Это фактически тот же самый однострочный редактор, но с присоединенной к нему надписью. Во многих случаях для удобства пользователя над окнами редактирования необходимо помещать некоторый текст, поясняющий, что за информацию следует вводить пользователю в этом окне. В этих случаях удобнее использовать TLabeledEdit.

Довольно часто необходимо организовывать ввод числовой информации. При этом всегда необходимо помнить, что данные вводятся в символьном виде, то есть в свойстве Text компонентов TEdit и TLabeledEdit содержится строка символов. Если введено число, то для работы с ним как с числом, необходимо эту строку символов преобразовать во внутреннее представление числа. Для этого используются функции StrToInt – для преобразования строки в целое число и StrToFloat – для преобразования в вещественное число. С другой стороны, для того чтобы вывести числовые данные, например результаты вычислений, необходимо выполнить обратное преобразование чисел в их строковое представление. Для этого применяются функции IntToStr – для преобразования целого числа в строку и FloatToStr – для преобразования вещественного числа в строку.

Рассмотрим пример.

Создайте новый проект. Поместите на форму три компонента TLabeledEdit, четыре кнопки TSpeedButton, одну кнопку TBitBtn и компонент TStatusBar, расположенную на странице Common Controls, примерно так, как показано на рис. 9.24.

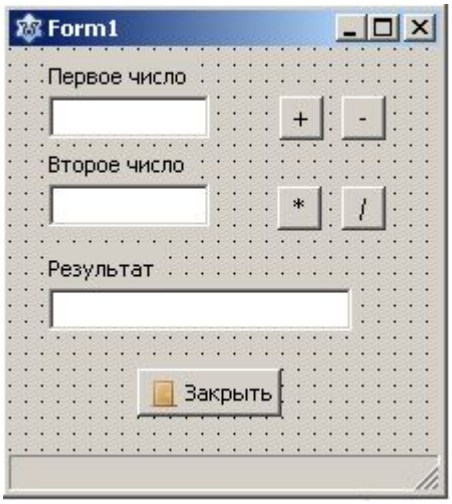


Рис. 9.24 Форма приложения

Для `LabeledEdit1` раскройте свойство `EditLabel` и установите свойство `Caption = "Первое число"`. Точно так же для `LabeledEdit2` установите свойство `Caption = "Второе число"` и для `LabeledEdit3` `Caption = "Результат"`.

Для `SpeedButton1` установите свойство `Caption = " + "`. Для `SpeedButton2` свойство `Caption = " - "`, для `SpeedButton3` `Caption = " * "` и `SpeedButton2` свойство `Caption = "/"`.

Установите для всех кнопок `SpeedButton` свойства `GroupIndex= 1`, свойства `AllowAllUp= true`. В обработчик события `OnShow` для `Form1` введите код:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  LabeledEdit1.SetFocus;
end;
```

В обработчик события `OnKeyPress` для `LabeledEdit1` введите код:

```
procedure TForm1.LabeledEdit1KeyPress(Sender: TObject;
                                      var Key: char);
```

```
begin
  if Key = #13 then
  begin
    LabeledEdit2.SetFocus;
    SpeedButton1.Down:= false;
    SpeedButton2.Down:= false;
    SpeedButton3.Down:= false;
    SpeedButton4.Down:= false;
    exit;
  end;
end;
```

В обработчик события `OnClick` для `SpeedButton1` введите код:

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
              + StrToInt(LabeledEdit2.Text));
  StatusBar1.SimpleText:= 'Сложение';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;
```

В обработчик события `OnClick` для `SpeedButton2` введите код:

```
procedure TForm1.SpeedButton2Click(Sender: TObject);
begin
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
              - StrToInt(LabeledEdit2.Text));
```

```
  StatusBar1.SimpleText:= 'Вычитание';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;
```

В обработчик события `OnClick` для `SpeedButton3` введите код:

```
procedure TForm1.SpeedButton3Click(Sender: TObject);
begin
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
              * StrToInt(LabeledEdit2.Text));
  StatusBar1.SimpleText:= 'Умножение';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;
```

В обработчик события `OnClick` для `SpeedButton4` введите код:

```
procedure TForm1.SpeedButton4Click(Sender: TObject);
begin
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
                    div StrToInt(LabeledEdit2.Text));
  StatusBar1.SimpleText:= 'Деление нацело';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;
```

Здесь новым для нас является компонент `TStatusBar`, который позволяет выводить в окне так называемую строку состояния. В этой строке приложение обычно выводит различную вспомогательную информацию. В нашей программе будет выводиться информация о последней выполненной операции.

В строке состояния можно определить одну или несколько независимых панелей. Если вы хотите иметь всего одну панель, то установите свойство `SimplePanel = true`. В этом случае для вывода текста в строку состояния используйте свойство `SimpleText`. Если вам необходимо иметь несколько панелей установите свойство `SimplePanel = false`. Тогда для вывода текста в нужную панель используйте свойство `Panels[k]`, где номер `k` панели (нумерация с 0) или подсвойство `Items`, например:

```
StatusBar1.Panels[0].Text:= 'Какой либо текст';
StatusBar1.Panels.Items[0].Text:= 'Какой либо текст';
```

Откомпилируйте и запустите программу. Опробуйте работу кнопок.

В процессе ввода информации пользователи довольно часто допускают ошибки. В большинстве случаев эти ошибки носят случайный и непреднамеренный характер, но могут вызвать сбой или даже аварийное завершение программы. Поэтому программист должен предусматривать соответствующие меры защиты своих программ от такого рода ошибок.

В нашем примере мы реализовали действия для целых чисел. При вводе чисел пользователь может, например, ввести недопустимый символ или вместо целого ввести вещественное число. В этой программе никакого контроля при вводе нет. В таких случаях программа, как правило, аварийно завершается. В 5.1.14 и 5.1.25 мы уже касались этого вопроса.

При возникновении какой-либо ошибочной ситуаций (говорят еще исключительной ситуации) генерируется так называемое исключение. К возникновению исключений могут привести не только действия пользователя, но и ряд других обстоятельств. Например, обращение к несуществующему файлу, обращение к устройству, которое в этот момент не готово, выход за пределы выделенной динамической области памяти, деление на ноль, переполнение разряд-

ной сетки и пр. Программист обязан и для таких случаев предусмотреть какие-то действия, чтобы не допустить краха своей программы.

Отвлечемся на некоторое время от изучения компонентов и рассмотрим некоторые способы обнаружения и организации соответствующей реакции программы в случае возникновения каких-либо ошибочных ситуаций.

9.2.7. Обработка исключений. Компонент `TMaskEdit`. Организация контроля ввода данных

Для обработки стандартных исключений в Lazarus имеются специальные классы. Если в программе не предусмотрена обработка какого-нибудь исключения, то оно обрабатывается глобальным обработчиком. Он обеспечивает стандартную реакцию на возникшее исключение – выводит предупреждение на экран с кратким описанием причины, вызвавшее исключение, рис. 9.25.

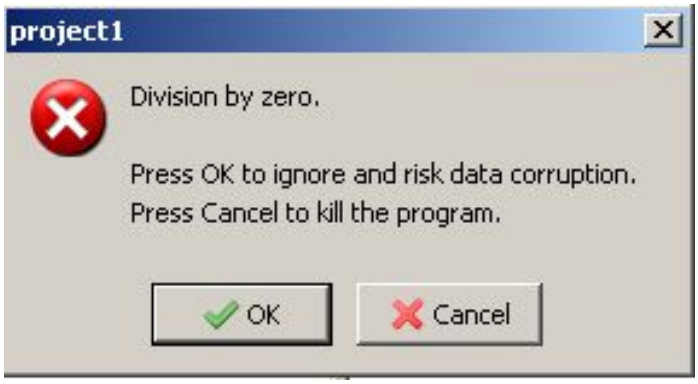


Рис. 9.25. Стандартная реакция на исключение

Здесь указано, что произошла попытка деления на ноль и предлагаются варианты действий – продолжить выполнение программы или завершить ее.

В большинстве случаев стандартной реакции оказывается недостаточной. Поясню примером. Предположим, в программе создается временный файл, который после завершения программы должен быть удален. Пусть во время работы программы с файлом произошла ошибка. Скажем, был прочитан недопусти-

мый символ. Обработка такой ошибки программистом не был предусмотрен. Тогда последует стандартная реакция, рис. 9.26.

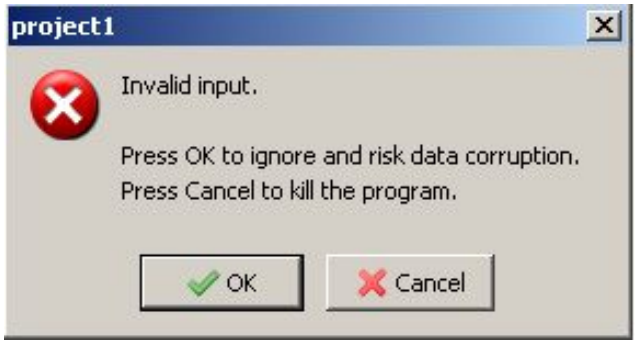


Рис. 9.26. Стандартная реакция на исключение

Пользователь закрывает программу, нажав кнопку `Cancel`. А временный файл, который должен был быть уничтожен, останется на диске! Существуют очень много программ, которые оставляют после своей работы всякий "мусор" на диске, если они завершаются аварийно. И это происходит из-за того, что программист не предусмотрел соответствующих действий по очистке мусора в случае возникновения исключительных ситуаций.

В таких случаях гораздо лучше, если программист организует свою собственную обработку исключения, в которой и выполнит очистку мусора. Кроме того, программист знает логику работы своего приложения, знает – где именно произошло исключение и, в большинстве случаев, почему это произошло, поэтому он может выдать пользователю более развернутое описание причины ошибки, к тому же на русском языке, а также предусмотреть такие действия, которые могли бы пользователю продолжить работу (если это возможно).

Рассмотрим некоторые классы исключений.

- `EConvertError` – ошибка преобразования типов. Чаще всего происходит при преобразовании строки в число.
- `EDivByZero` – попытка деления целого числа на ноль.
- `EZeroDivide` – попытка деления вещественного числа на ноль.

- `ERangeError` – выход за пределы допустимого диапазона индекса или порядкового типа.
- `EIntOverflow` – переполнение в операциях с целыми числами, т.е. попытка присвоения переменной целого типа значения больше допустимого.
- `EOverflow` – переполнение в операциях с вещественными числами.
- `EUnderflow` – потеря значащих разрядов в операциях с вещественными числами.
- `EAccessViolation` – попытка обращения к недействительному адресу в памяти. Чаще всего возникает из-за неправильной работы с указателями.

Для обработки исключений используются две конструкции. Первая конструкция имеет вид:

```
try
< Потенциально "опасные" операторы, при выполнении которых могут возникнуть исключительные ситуации >

except
on класс исключения 1 do < оператор обработки исключения 1 >;
on класс исключения 2 do < оператор обработки исключения 2 >;
.....
on класс исключения n do < оператор обработки исключения n >;
else
< операторы обработки остальных исключений >
end;
```

Данная конструкция означает, что после ключевого слова `try` и до клю-

чевого слова `except` следуют операторы, при выполнении которых возможно возникновение исключений. После `except` следуют операторы, которые образуют секцию обработки исключений. Признаком конца секции служит ключевое слово `end`. Внутри секции программист указывает классы исключений (говорят еще типы исключений) после слова `on`, а затем после ключевого слова `do` оператор обработки исключения, причем оператор может быть составным. После необязательного `else` следуют операторы обработки исключений, не вошедшие в перечень `on`. Если программисту нужно только установить сам факт исключения, независимо от типа, то он может просто записать обработчик исключения после слова `except`.

Вторая конструкция имеет вид:

```
try
< Потенциально "опасные" операторы, при выполнении которых могут возникнуть исключительные ситуации >
finally
<  операторы, которые должны быть выполнены в любом случае, независимо от того, произошло исключение или нет  >
end;
```

В чем их различие? В конструкции `try..except` если при выполнении операторов секции `try` возникло исключение, то управление передается в секцию `except`, где и происходит обработка исключения. Если же исключения не произошло, то операторы блока `except` просто пропускаются. В конструкции `try..finally` операторы будут выполнены независимо от того, произошло исключение или нет.

Рассмотрим пример:

```
try
```

```
  num:=StrToInt(Stroka);
except
on EConvertError do
   ShowMessage('Ошибка преобразования строки в целое число');
end;
```

Здесь сообщение будет выведено только в том случае, когда невозможно преобразование строки символов в целое число, то есть когда возникнет исключение `EConvertError`.

Конструкции `try..except` и `try..finally` могут быть вложены друг в друга на неограниченную глубину. Рассмотрим реализацию примера с файлом.

```
Procedure Test;
var
  F: TextFile;
  number: integer;
  s: string;
begin
  AssignFile(F, 'Data.txt');
  Rewrite(F);
  s:= '12#4';        // В файл намеренно записывается
  Writeln(F, s);   // ошибочная строка символов
  Reset(F);
  try  // начало секции (блока) try..except
    Readln(F, s);
    try  // начало секции try..finally
      number:= StrToInt(s);
    finally
```

```
        CloseFile(F);  // эти два оператора будут выполнены
        DeleteFile('Data.txt'); // в любом случае
      end;   // конец секции try..finally
    except
      on EConvertError do
         ShowMessage('Ошибка преобразования');
    end;  // конец секции try..except
  end;
```

Вернемся к примеру, в котором осуществляется ввод двух целых чисел и выполняются четыре арифметических действия (сложение, вычитание, умножение и деление нацело), рис. 9.24. Модифицируем программу, добавив в него обработку исключений. Перепишите обработчики события `OnKeyPress` для `LabeledEdit1` и событий `OnClick` кнопок `SpeedButton1`, `SpeedButton2`, `SpeedButton3` и `SpeedButton4` в виде:

```
procedure TForm1.LabeledEdit1KeyPress(Sender: TObject;
                                      var Key: char);
begin
  if Key = #13 then
  begin
    try
      StrToInt(LabeledEdit1.Text);
    except
      on EConvertError do
      begin
        ShowMessage('Ошибка преобразования! Вероятно, ' +
                    'Вы ошиблись при вводе числа');
        exit;
```

```
      end;
    end;
    LabeledEdit2.SetFocus;
    SpeedButton1.Down:= false;
    SpeedButton2.Down:= false;
    SpeedButton3.Down:= false;
    SpeedButton4.Down:= false;
    exit;
  end;
end;

procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  try
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
                  + StrToInt(LabeledEdit2.Text));
  except
    on EConvertError do
    begin
      ShowMessage('Ошибка преобразования! Вероятно, ' +
                  'Вы ошиблись при вводе второго числа');
      exit;
    end;
  end;
  StatusBar1.SimpleText:= 'Сложение';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;
procedure TForm1.SpeedButton2Click(Sender: TObject);
```

```
begin
  try
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
                   - StrToInt(LabeledEdit2.Text));
  except
    on EConvertError do
    begin
      ShowMessage('Ошибка преобразования! Вероятно, ' +
                  'Вы ошиблись при вводе второго числа');
      exit;
    end;
  end;
  StatusBar1.SimpleText:= 'Вычитание';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;

procedure TForm1.SpeedButton3Click(Sender: TObject);
begin
  try
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
                   * StrToInt(LabeledEdit2.Text));
  except
    on EConvertError do
    begin
      ShowMessage('Ошибка преобразования! Вероятно, ' +
                  'Вы ошиблись при вводе второго числа');
exit;
    end;
```

```
  end;
  StatusBar1.SimpleText:= 'Умножение';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;
procedure TForm1.SpeedButton4Click(Sender: TObject);
begin
  try
  LabeledEdit3.Text:=IntToStr(StrToInt(LabeledEdit1.Text)
                div StrToInt(LabeledEdit2.Text));
  except
    on EConvertError do
    begin
      ShowMessage('Ошибка преобразования! Вероятно, ' +
                  'Вы ошиблись при вводе второго числа');
      exit;
    end;
    on EDivByZero do
    begin
      ShowMessage('Ошибка! Произошло деление на ноль. Вероятно, ' +
                  'Вы ошиблись при вводе второго числа');
      exit;
    end;
  end;
  StatusBar1.SimpleText:= 'Деление нацело';
  LabeledEdit1.SetFocus;
  LabeledEdit1.SelectAll;
end;
```

Теперь при вводе недопустимого символа для целого числа, а также при

вводе числа 0 в качестве второго операнда программа перехватывает исключения и реагирует соответствующим образом.

Имейте в виду, если вы запускаете программу из среды Lazarus, то исключения будет перехватывать отладчик. Поэтому лучше запускать программу из командной строки или в настройках окружения для отладчика добавьте нужные вам типы исключений в список игнорирования.

Обработку исключений вполне можно применять и в консольных приложениях. Вспомним программу из 5.1.14. Организуем контроль ввода данных, используя механизм исключений.

```
program int_operations_control;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, SysUtils;
var
  A, B, C: integer;
begin
  writeln(UTF8ToConsole('Введите два числа'));
  readln(A, B);
  writeln('A= ', A, ' B= ', B);
  C:= A + B;
  writeln(UTF8ToConsole('Демонстрация сложения, C= '), C);
  C:= A * B;
  writeln(UTF8ToConsole('Демонстрация умножения, C= '), C);
  try
    C:= A div B;
    writeln(UTF8ToConsole('Демонстрация деления нацело, C='),
                                C);
  except
```

```
  on EDivByZero do
    begin
      writeln(UTF8ToConsole('Ошибка!! Деление на ноль.'));
      writeln(UTF8ToConsole('Нажмите любую клавишу'));
      readkey;
      exit;
    end;
  end;
  C:= A mod B;
  writeln(UTF8ToConsole('Остаток от деления, C= '), C);
  C:= A - B;
  writeln(UTF8ToConsole('Демонстрация вычитания, C= '), C);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Сравните эту программу с программой из 5.1.25.

Контроль и обработку ошибок с применением механизма исключений удобнее использовать, когда работа программы уже не зависит от действий пользователя, т.е. все необходимые данные от пользователя уже получены, программа перешла непосредственно к обработке данных – идут вычисления, происходит чтение и запись в файлы, обращения к различным внешним устройствам и т.д. На этапе же ввода данных, когда пользователь в режиме диалога вводит какие-то данные с клавиатуры, удобнее использовать другой способ контроля. Поскольку обработка исключений происходит после завершения ввода пользователем, то есть этим происходит только констатация факта, что ошибка пользователем уже совершена. Приходится организовывать повторный ввод, причем с того места, где пользователь ошибся. А это приведет к усложнению кода. Гораздо разумнее и эффективней при вводе данных просто не давать

пользователю совершить ошибку. Например, если пользователь должен вводить только целые числа, проще контролировать введенные пользователем символы и, если будет попытка ввести недопустимый для целых чисел символ, просто этот символ игнорировать.

Для этого удобнее использовать другую разновидность компонента `TEdit` – `TMaskEdit` из страницы `Additional`.

9.2.7.1. Компонент `TMaskEdit`

Для этого удобнее использовать другую разновидность компонента `TEdit` – `TMaskEdit` из страницы `Additional`. В этом компоненте имеется свойство `EditMask`, с помощью которого можно задать маску ввода. Маска это некий шаблон, задающий какие символы может вводить пользователь в окне ввода. Недопустимые символы игнорируются. Маска состоит из трех частей, между которыми ставится точка с запятой (;). В первой части маски – шаблоне записываются символы (табл. 9.2), которые указывают какие символы можно вводить в каждой позиции.

Таблица 9.2.

Символ	Шаблон ввода
!	Означает, что в `EditText` недостающие символы предваряются пробелами. В случае отсутствия символа пробелы размещаются в конце.
0	Означает, что в данной позиции должна быть цифра.
9	Означает, что в данной позиции может быть цифра или ничего.
#	Означает, что в данной позиции может быть цифра, знак « + », знак «-» или ничего.

Далее через точку с запятой (;) записывается 1 или 0 в зависимости от того, надо или нет, чтобы символы, добавляемые маской, включались в свойство

`Text` компонента. В третьей части маски указывается символ-заполнитель, используемый для обозначения позиций, в которых еще не осуществлен ввод. Установить нужную маску можно прямо в свойстве `EditMask`, введя необходимые символы маски или в редакторе масок, открыть который можно нажав кнопку с троеточием, рис. 9.27.

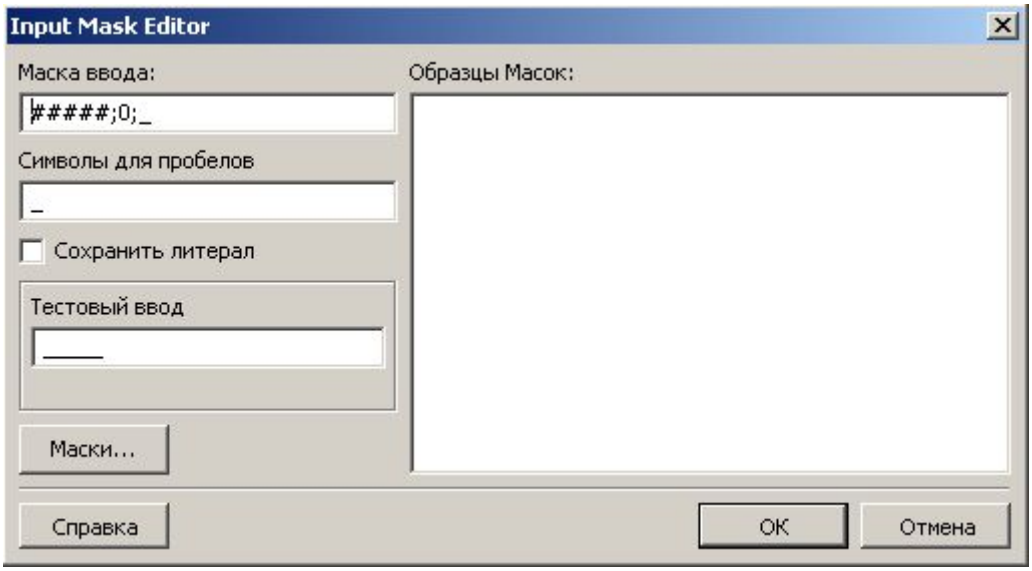


Рис. 9.27. Окно редактора масок

Прочитать результат ввода можно или в свойстве `Text`, которое, в зависимости от вида второй части маски, включает или не включает в себя символы маски, или в свойстве `EditText`, содержащем введенный текст вместе с символами маски.

Итак, давайте применим для нашего примера компонент `TMaskEdit`. Удалите из формы `LabeledEdit1` и `LabeledEdit2`. Перенесите на их место два компонента `TMaskEdit`. Также вставьте в форму два компонента `TLabel`. В общем, восстановите внешний вид формы как на рисунке 9.24.

Установите следующие свойства `MaskEdit1`:

- `AutoSelect = true`
- `EditMask = #9999;0;`

- `TabOrder = 0`
- свойство `Text` оставьте пустым.

В качестве символа заполнителя в окошке "Символы для пробелов" в редакторе масок введите пробел вместо знака подчеркивания.

Установите свойства `MaskEdit2`:

- `AutoSelect = true`
- `EditMask = #9999;0;`
- `TabOrder = 1`
- свойство `Text` оставьте пустым.

Установите символ заполнитель такой же, что и для `MaskEdit1`.

В `LabeledEdit3` и `BitBtn1` свойству `TabStop` присвойте значение `false`.

Обработчики событий запишите в виде:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  MaskEdit1.SetFocus;
end;

procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  LabeledEdit3.Text:= IntToStr(StrToInt(MaskEdit1.Text)
                    + StrToInt(MaskEdit2.Text));
  StatusBar1.SimpleText:= 'Сложение';
  MaskEdit1.SetFocus;
  MaskEdit1.SelectAll;
end;
```

```
procedure TForm1.SpeedButton2Click(Sender: TObject);
begin
  LabeledEdit3.Text:= IntToStr(StrToInt(MaskEdit1.Text)
                   - StrToInt(MaskEdit2.Text));
  StatusBar1.SimpleText:= 'Вычитание';
  MaskEdit1.SetFocus;
  MaskEdit1.SelectAll;
end;

procedure TForm1.SpeedButton3Click(Sender: TObject);
begin
  LabeledEdit3.Text:= IntToStr(StrToInt(MaskEdit1.Text)
                   * StrToInt(MaskEdit2.Text));
  StatusBar1.SimpleText:= 'Умножение';
  MaskEdit1.SetFocus;
  MaskEdit1.SelectAll;
end;
procedure TForm1.SpeedButton4Click(Sender: TObject);
begin
  try
  LabeledEdit3.Text:= IntToStr(StrToInt(MaskEdit1.Text)
                 div StrToInt(MaskEdit2.Text));
  except
  on EDivByZero do
    ShowMessage('Ошибка! Произошло деление на ноль.' +
      ' Вероятно Вы ошиблись при вводе второго числа');
  end;
  StatusBar1.SimpleText:= 'Деление нацело';
  MaskEdit1.SetFocus;
```

```
  MaskEdit1.SelectAll;
end;
```

Как видите, обработку исключений на возможные ошибки преобразования типов мы убрали, поскольку MaskEdit не позволит пользователю ввести недопустимые символы. Но в обработчике операции деления исключение, возникающее при попытке деления на ноль мы, естественно, оставили.

Переход от MaskEdit1 к MaskEdit2 и обратно осуществляйте клавишей Tab. Впрочем, можете добавить обработчик OnKeyPress для MaskEdit1:

```
procedure TForm1.MaskEdit1KeyPress(Sender: TObject;
                                   var Key: char);
begin
  if Key = #13 then
  begin
    MaskEdit2.SetFocus;
    SpeedButton1.Down:= false;
    SpeedButton2.Down:= false;
    SpeedButton3.Down:= false;
    SpeedButton4.Down:= false;
    exit;
  end;
end;
```

Давайте теперь попробуем сами реализовать контроль ввода для компонента TLabeledEdit без использования TMaskEdit. Реализуем полный контроль ввода целых чисел с учетом знака. Для этого создадим простое приложение. Поместите на форму два компонента TLabeledEdit и одну кнопку TButton, рис. 9.28.

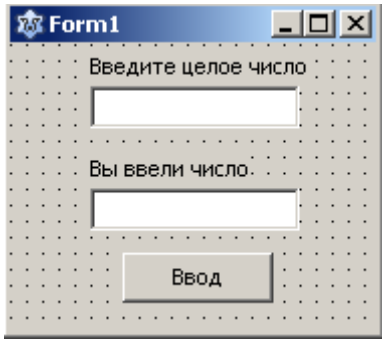


Рис. 9.28. Форма приложения

Установите свойство `TabOrder=0` и `TabStop=true` для `LabeledEdit1`, и `TabOrder=1`, `TabStop=true` для `TButton1`, а для `LabeledEdit2` свойство `TabStop=false`. Это для того, чтобы при нажатии клавиши `Tab` фокус автоматически перемещался с `LabeledEdit1` на кнопку и обратно. Кроме того, установите свойство `AutoSelect` для `LabeledEdit1` равным `true`.

Будем вводить числа в окне `LabeledEdit1`, затем введенное число просто выведем в `LabeledEdit2`, но не "сразу"! Сначала преобразуем его во внутреннее представление целого числа для того, чтобы "выловить" ошибки. Если будут ошибки при вводе, то естественно, преобразования не произойдет и возникнет исключение `EConvertError`. Для отладки и тестирования нашего приложения нам хватит и стандартной реакции системы, поэтому обработку исключения мы делать не будем. Добавьте лишь исключение `EConvertError` в список игнора через меню **`Окружение->Параметры>Отладчик->Исключения языка`**, чтобы удобнее было запускать наше приложение из среды Lazarus.

Итак, с чего начнем? Задача вполне ясная. Необходимо обеспечить ввод пользователем только цифр от 0 до 9 и знака минус (-) (знак плюс (+) перед числом обычно не ставится). Для этого проще всего воспользоваться определенным в Паскаль типом – множество и оператором `in` для определения принадлежности символа к допустимым.

В `LabeledEdit1` для события `OnKeyPress` напишите следующий обработчик:

```
procedure TForm1.LabeledEdit1KeyPress(Sender: TObject;
                                      var Key: char);
begin
{ разрешаем только цифры, знак минус и кл. BackSpace }
  if not (Key in ['0' .. '9', '-' , #8])
  then
  begin
    Key:= #0;
  end;
end;
```

В операторе `if` если пользователь нажимает на клавиши отличные от цифр, клавиши `BackSpace` и знака "-" параметру Key присваивается нулевой символ.

Для события `OnClick` кнопки напишите следующий обработчик:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  LabeledEdit1.SetFocus;
  n:= StrToInt(LabeledEdit1.Text);
  LabeledEdit2.Text:= IntToStr(n);
end;
```

Не забудьте объявить переменную *n* целого типа:

```
var
  Form1: TForm1;
```

```
  n: integer;
```

Запустите приложение. Попробуйте ввести символы, отличные от цифр и знака минус. Вы видите, что недопустимые символы просто не появляются в окне ввода. И что? Задача решена? К сожалению, нет!

Введите несколько цифр, а затем знак минус. Например, "2010-". Если нажать на кнопку, получите сообщение об ошибке. Потому что знак числа ставится перед первой значащей цифрой. Или введите сначала два или более знака (-), затем цифры, например, "--2010". Опять получите ошибку.

Вы скажете, ну не может быть, чтобы пользователь допускал такие ошибки! Не настолько же пользователь глуп и туп! И вы правы! Речь может идти о случайных ошибках. Мы на протяжении этой книги не раз говорили об этом. Программа должна быть защищена от любых случайных и непреднамеренных и случайных ошибок пользователя!

Значит после ввода цифр, знак (-) тоже должен быть запрещен. Кроме того, необходимо разрешить ввод знака числа только один раз, причем вначале числа! Для этого введем логическую переменную, назовем ее sign. Первоначально ее значение будет равно false. Как только пользователь нажал на хотя бы один допустимый символ, включая знак числа, переменной sign присваиваем значение true.

Обработчик для LabeledEdit1 переписываем в следующем виде, заодно разрешим пользователю заканчивать ввод нажатием клавиши Enter:

```
procedure TForm1.LabeledEdit1KeyPress(Sender: TObject;
                                      var Key: char);
begin
  if Key = #13 then
  begin
    sign:= false;
```

```
    n:= StrToInt(LabeledEdit1.Text);
    LabeledEdit2.Text:= IntToStr(n);
  end;
{ разрешаем только цифры, знак минус и кл. BackSpace }
  if not (Key in ['0' .. '9', '-' , #8])
  then
  begin
    Key:= #0;
    exit;
  end;
  if (Key = '-') and (sign)
  then Key:= #0;
  sign:= true;
end;
```

Оператор

```
if (Key = '-') and (sign)
then Key:= #0;
```

не позволит пользователю ввести знак числа после значащих цифр и, кроме того, дважды ввести знак (-).

Теперь, кажется все. Но нет, еще не все! Представьте ситуацию - пользователь хотел ввести отрицательное число, но забыл ввести знак минус, а цифры уже ввел. Попробуйте ввести несколько цифр, затем вернуть курсор на начало строки и попытайтесь ввести (-). У вас не получится! Даже если вы удалите введенные цифры клавишей `Delete` или `BackSpace`. Потому что `sign` имеет значение `true`. Ситуация не столь проста, как кажется. Чтобы ввести отрицательное число нужно сначала нажать на кнопку "Ввод" или на клавишу `Enter`, т.е. пользователь будет вынужден ввести число, которое не хотел вводить!

Придется контролировать положение курсора. У компонента есть свойство `SelStart`, которое указывает на текущее положение курсора. Используя это свойство и функцию `Pos`, перепишем обработчик следующим образом:

```
procedure TForm1.LabeledEdit1KeyPress(Sender: TObject;
                                      var Key: char);
begin
  if Key = #13 then
  begin
    sign:= false;
    n:= StrToInt(LabeledEdit1.Text);
    LabeledEdit2.Text:= IntToStr(n);
    exit;
  end;
{ разрешаем только цифры, знак минус и кл. BackSpace }
  if not (Key in ['0' .. '9', '-' , #8])
  then
  begin
    Key:= #0;
    exit;
  end;
  if (Key = '-') and (sign)
  then
  begin
    if (LabeledEdit1.SelStart <> 0)
    then
    begin
      Key:= #0;
      exit;
```

```
    end
    else
      if (Pos('-', LabeledEdit1.Text) <> 0)
      then  Key:= #0;
  end;
  sign:= true;
end;
```

Обработчик работает следующим образом – если текущий символ знак "-" и цифры введены, то проверяем позицию курсора. Если курсор установлен на начало и минуса нет, то разрешаем ввод знака.

Уф! Наконец-то должно все заработать как надо. Но есть нюанс!

Попробуйте ввести только один знак (-) и больше ничего. Опять получите ошибку. Но это уже слишком, воскликнете вы! Кому придет в голову вводить один только минус. Ну, а вдруг, чисто случайно? И еще. А что если, пользователь ничего не введя, опять же случайно, нажмет кнопку или `Enter`, т.е. будет введена пустая строка?

У вас не возникает ощущения, что мы пишем программу для американцев? (Как говорил М. Задорнов – "Ну, тупые…!).

Сделаем так, если длина строки равна единице и это знак (-), то запрещаем ввод.

```
procedure TForm1.LabeledEdit1KeyPress(Sender: TObject;
                                      var Key: char);
begin
  if Key = #13 then
  begin
    sign:= false;
    if Length(LabeledEdit1.Text) = 0 // если пустая строка
    then exit;
```

```
    if (Length(LabeledEdit1.Text) = 1) and
       (LabeledEdit1.Text = '-')
    then exit;
    n:= StrToInt(LabeledEdit1.Text);
    LabeledEdit2.Text:= IntToStr(n);
    exit;
  end;
{ разрешаем только цифры, знак минус и кл. BackSpace }
  if not (Key in ['0' .. '9', '-' , #8])
  then
  begin
    Key:= #0;
    exit;
  end;
  if (Key = '-') and (sign)
  then
  begin
    if (LabeledEdit1.SelStart <> 0)
    then
    begin
      Key:= #0;
      exit;
    end
    else
      if (Pos('-', LabeledEdit1.Text) <> 0)
      then  Key:= #0;
  end;
  sign:= true;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  LabeledEdit1.SetFocus;
  sign:= false;
  if Length(LabeledEdit1.Text) = 0 // если пустая строка
  then exit;
  if (Length(LabeledEdit1.Text) = 1) and
     (LabeledEdit1.Text = '-')
  then exit
  else
  n:= StrToInt(LabeledEdit1.Text);
  LabeledEdit2.Text:= IntToStr(n);
end;
```

Между прочим, компонент `TMaskEdit` при установленной маске на ввод только чисел тоже не "ловит" одиночный минус и "молча, проглатывает" пустую строку!

Почему такие "глупые" ситуации надо контролировать? Представьте себе, что пользователь должен вводить большое количество числовых данных. Спустя некоторое время после начала ввода пользователь начинает уставать и, чисто машинально, вводит пустую строку или одиночный минус. Ваша программа аварийно завершается из-за ошибки преобразования (`EConvertError`)! Вся проделанная пользователем работа пошла "коту под хвост". Как же он будет вас проклинать! Именно вас – разработчика программы!

Этот пример еще раз демонстрирует, как надо последовательно улучшать и "доводить" алгоритм и программу до достижения требуемой функциональности. Пусть сам алгоритм, быть может, не является оптимальным. Не в этом суть! Это конкретный (хотя и простой) пример того, как надо разрабатывать реальные программы. Уже неоднократно отмечал, что с первого раза невозможно

написать полностью правильно работающую программу. Если можно так выразиться, этот пример показывает метод "последовательных приближений" разработки программ.

А теперь попробуем реализовать контроль ввода вещественных чисел. Ясно, что надо добавить проверку на символы точки (.) и запятой (,). Использование точки или запятой для отделения целой части числа от дробной зависит от настроек операционной системы `Windows`. Хотя `Lazarus` спокойно преобразует строку в число и с точкой, и с запятой. А вот `Delphi` "ругается", если разделитель не совпадает с настройками `Windows`. Так что будем учитывать оба разделителя. Для определения какой разделитель используется системой, служит глобальная переменная `DecimalSeparator`, значением которой и является символ разделителя. При этом поступим следующим образом, если пользователь ввел "не тот" разделитель, то просто заменим его на нужный. Так что, наша программа будет "обладать" уже кое-каким интеллектом! При реализации применим, для разнообразия, несколько другой способ, нежели в предыдущей программе. Будем использовать оператор выбора `case .. of`. Основываясь на опыте разработки предыдущей программы, будем учитывать не только одиночный минус, но и одиночный разделитель, и комбинацию "-" и разделитель. Опять же для разнообразия используем компонент `TEdit`. Код программы:

```
unit Unit1;
{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, LResources, Forms,
  Controls, Graphics, Dialogs, StdCtrls;
```

```
type

  { TForm1 }

  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject;
                            var Key: char);
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;
implementation

{ TForm1 }

procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.SetFocus;
  if (Edit1.Text <> '') and  // если пустая строка
```

```
     (Edit1.Text <> '-') and // если одиночный минус
     (Edit1.Text <> DecimalSeparator) and {если одиночный разделитель}
     (Edit1.Text <> '-' + DecimalSeparator) {если "-" и разделитель}
  then Edit2.Text:= FloatToStr(StrToFloat(Edit1.Text));
  Edit1.Clear;
end;

procedure TForm1.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
  case Key of
    '0'..'9', #8:; // если цифры и кл. BackSpace, ввод разрешен
    #13: // если нажата клавиша <Enter>
      begin
        if (Edit1.Text <> '') and
           (Edit1.Text <> '-') and
           (Edit1.Text <> DecimalSeparator) and
           (Edit1.Text <> '-' + DecimalSeparator)
        then
          Edit2.Text:=FloatToStr(StrToFloat(Edit1.Text));
        Edit1.SetFocus;
        Edit1.Clear;
        end;
    '.',',':  // если разделители
    begin
      if Key <> DecimalSeparator then
      Key := DecimalSeparator; // замена разделителя
```

```
      if Pos(DecimalSeparator, Edit1.Text) <> 0
      then Key:= #0;
    end;

    '-' :  // если знак минус
    begin
    if (Edit1.SelStart <> 0)  // если одиночный минус
    then
    begin
      Key:= #0;
      exit;
    end
    else
      if (Pos('-', Edit1.Text) <> 0) // два минуса
      then  Key:= #0;
    end;
  else
  Key := #0;
  end;
end;


initialization
  {$I unit1.lrs}
end.
```

И, наконец, самый простой способ контроля ввода числовых данных это использование функции `val()`. Функция была рассмотрена нами в 6.3.1.4.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,   Forms,
  Controls, Graphics, Dialogs, StdCtrls;
type
  { TForm1 }
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject;
                            var Key: char);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
  code: integer;
  value: real;
implementation
{ TForm1 }
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.SetFocus;
  val(Edit1.Text, value, code);
  if code <> 0 then exit;
  Edit2.Text:= FloatToStr(value);
end;

procedure TForm1.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
if Key = #13 then
  begin
    Edit1.SetFocus;
    val(Edit1.Text, value, code);
    if code <> 0 then exit;
    Edit2.Text:= FloatToStr(value);
  end;
end;
initialization
  {$I unit1.lrs}
end.
```

9.2.8. Специальные компоненты для ввода чисел TSpinEdit, TFloatSpinEdit и TUpDown

Если необходимо вводить последовательность чисел с заранее определенным шагом изменения значений, то бывает удобнее использовать специальные компоненты TSpinEdit и TFloatSpinEdit, расположенные на вкладке Misc. Основные свойства этих компонентов:

- `Increment` – шаг приращения числа. Разумеется, для `TSpinEdit` шаг может быть только целым, а для `TFloatSpinEdit` может быть и дробным.
- `MaxValue` – максимально возможное значение числа.
- `MinValue` – минимально возможное значение числа.
- `ReadOnly` – изменить значение пользователь не может, правда программно изменить значение можно.
- `Value` – указывает текущее значение. Во время проектирования можно задать начальное значение числа, с которого будет начинаться ввод.
- `DecimalPlaces` – только для `TFloatSpinEdit`, указывает количество разрядов после запятой.

Вводить числа можно не только с помощью кнопок прокрутки, но и с клавиатуры. При этом обеспечивается контроль на ввод недопустимых символов.

В компоненте `TSpinEdit` нельзя ввести число вне установленного диапазона как с помощью кнопок прокрутки, так и с клавиатуры. В компоненте `TFloatSpinEdit` если ввод производится с помощью кнопок, то переход за пределы указанного при проектировании диапазона блокируется, т.е. например, нажимая на кнопку увеличения нельзя ввести число больше `MaxValue` или нажимая на кнопку уменьшения, нельзя ввести число меньше `MinValue`. Но можно ввести числа вне этого диапазона с клавиатуры. Однако если после этого нажать на любую из кнопок прокрутки, то в окне ввода появится число, первое из возможных в установленном диапазоне! Например, вы установили `MaxValue` = 10.0, с клавиатуры ввели число 24.5. После этого нажали кнопку увеличения, в окне компонента появится число 10.0. Если нажать на кнопку уменьшения, то появится число 9.5.

Упомяну еще об одном компоненте `TUpDown` из вкладки `Common Controls`, который позволяет вводить целые числа. Основные свойства у него практически такие же, как и у `TSpinEdit`, только по-другому названные. Например, свойство `MaxValue` в `TUpDown` называется `Max`, `MinValue` в

TUpDown называется Min, свойство Value называется Position. Если его использовать "в одиночку", то текущее значение (Position) будет не видно пользователю. Поэтому его чаще всего используют совместно с TEdit. Для этого имеется свойство Associate. Используя раскрывающийся список, можно указать нужный компонент. В этом случае TUpDown "прижимается" к выбранному компоненту и оба компонента превращаются в один, но только визуально! В программе обращаться к ним надо отдельно по своим именам. Свойством AlignButton можно управлять расположением TUpDown вокруг компонента (слева, справа, сверху, снизу). Также имеется свойство Orientation, позволяющее управлять положением кнопок TUpDown (вертикально или горизонтально). И, наконец, свойство Wrap определяет, что будет происходить, если пользователь попытается ввести значение больше максимального или меньше минимального. При Wrap=false ввод числа с помощью кнопок вне указанного диапазона (Min..Max) блокируется, а при Wrap=true происходит автоматический переход с Min на Max при нажатии кнопки на уменьшение и с Max на Min при нажатии кнопки на увеличение. При вводе с клавиатуры следует помнить, что этот компонент абсолютно не защищен от ввода недопустимых символов.

В целом можно сказать, что рассмотренные компоненты удобны лишь в случае, если вводимые числа имеют постоянный шаг приращения и известен диапазон допустимых значений. При вводе случайных чисел, особенно, если они достаточно далеко "отстоят" друг от друга по значению, их применение может причинить пользователю лишь неудобства.

Итак, мы с вами рассмотрели способы организации ввода и контроля числовых данных. И хотя мы можем быть теперь уверены, что пользователь ввел числа, а не что-нибудь другое, к сожалению, при выполнении программы, все еще могут появиться ошибки. Для поиска и локализации ошибок выполняют специальные действия, называемые тестированием и отладкой программы.

9.2.9. Тестирование и отладка программы

Какие бывают виды ошибок?

Ошибки можно разделить на три группы. К первой относятся так называемые синтаксические ошибки. Это ошибки, допущенные непосредственно в момент кодирования, т.е. записи (набора в редакторе исходного кода) текста программы. Такие ошибки легче всего находить и исправлять, так как компилятор сам укажет на такие ошибки.

Ко второй группе ошибок относятся так называемые логические ошибки или их еще называют ошибками времени исполнения. Иностранные программисты их называют `bugs` – жучки. Логические ошибки проявляются во время запуска приложения. В большинстве случаев, логические ошибки приводят к получению неверных результатов. Очень часто приложение завершается аварийно. Программист получает только сообщение об ошибке. Если логическая ошибка тривиальная, то уже по сообщению можно определить, где именно кроется ошибка. Однако бывают ошибки, которые находятся с большим трудом.

Третья группа ошибок – это алгоритмические ошибки. Это такие ошибки, при которых внешне правильно и без ошибок работающая программа выдает в корне неверные результаты или решает совсем не ту задачу.

На практике довольно трудно сразу определить, к какой группе относится та или иная ошибка – к логической или алгоритмической. Если программа выдает неправильные результаты (но работает!), то следует, прежде всего, проверить сам алгоритм решения задачи. И если обнаруживается ошибка в алгоритме, то эта ошибка алгоритмическая. Если же алгоритм правильный, то ошибку следует признать логической.

Процесс обнаружения алгоритмических ошибок называется тестированием. При тестировании задается некоторый ряд входных данных, называемых модельными данными, для которых заранее известен результат (вспомните задачу вычисления значений синуса, разд. 5.1.26) или для которых можно "вруч-

ную" вычислить требуемый результат. Например, если это программа начисления заработной платы, то очень часто, на этапе внедрения, расчет заработной платы производится параллельно с помощью программы и бухгалтерии. Или, если программа сортирует какие-то данные, можно просто посмотреть на полученные результаты и определить, правильно программа работает или нет.

Процесс поиска, обнаружения и исправления логических ошибок называется отладкой. В основном процесс отладки сводится к определению в коде программы того места, где "сидит" ошибка (жучок!). Для отладки программы в Lazarus имеются несколько полезных инструментов.

✓ **Точки останова (Breakpoints).** Выполнение программы приостанавливается по достижению некоторого места (строки кода). Точку останова задает сам программист. Для этого надо в редакторе исходного кода просто щелкнуть мышью по полю, где показаны номера строк кода напротив нужной строки. Точка останова будет выделена красным цветом и помечена галочкой, рис. 9.29. При запуске программы отладчик выполнит все операторы до точки останова и остановит выполнение программы. Далее вы можете продолжить выполнение программы пошагово.

✓ **Пошаговое выполнение со входом.** При нажатии клавиши F7 будут выполнены все операторы текущей строки (поэтому рекомендуется располагать в строке строго по одному оператору, чтобы иметь возможность контролировать выполнение каждого оператора в отдельности). Каждое последующее нажатие клавиши F7 вызовет выполнение оператора в следующей строке. Выполняемый в данный момент оператор будет выделен серым цветом, а ее номер помечен зеленой стрелкой, рис. 9.29. Если в коде встречается вызов процедуры или функции, то будут выполнены все операторы тела этой процедуры (функции) также в пошаговом режиме.

✓ **Пошаговое выполнение в обход.** При каждом нажатии клавиши F8 также будут выполнены все операторы текущей строки. Если в коде встречается вызов процедуры или функции, то будут выполнены все операторы тела процеду-

ры (функции) без входа, т.е. "сразу" и указатель будет установлен на следующей после вызова подпрограммы строке.

✓ **Окно наблюдений.** Пожалуй, самый главный и полезный инструмент отладки. В этом окне можно наблюдать за текущими значениями переменных в процессе выполнения программы. Это средство поможет вам проследить за жизненным циклом интересующих вас переменных и увидеть, как изменяются их значения после выполнения того или иного оператора. Открыть это окно можно командой меню **Вид-> Окна отладки ->Окно наблюдений.** Еще проще открыть комбинацией клавиш **<Ctrl+Alt+W>**. Вид окна наблюдений показан на рисунке 9.30. Чтобы добавить новую переменную щелкните правой клавишей в окне наблюдений, в открывшемся диалоговом окне введите имя переменной или выражение, рис. 9.31. Можно назначить постоянное местоположение окна наблюдений в настройках окружения, например, так как показано на рис. 9.29. Окно наблюдений имеет такую же ширину, что и инспектор объектов и частично его перекрывает. Это логично, так как в момент отладки инспектор объектов нам не нужен и окно наблюдений не мешает нам работать в редакторе исходного кода. Увидеть значение переменной можно также просто наведя мышь на эту переменную в редакторе исходного кода. На рисунке 9.29. показан момент, когда мышь была наведена на переменную *n* в 56 строке редактора кода.

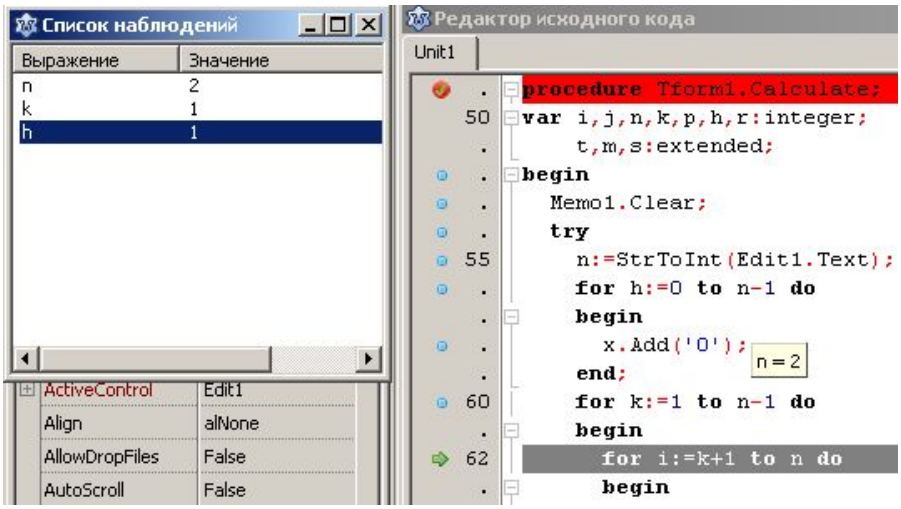


Рис. 9.29. Пошаговое выполнение приложения

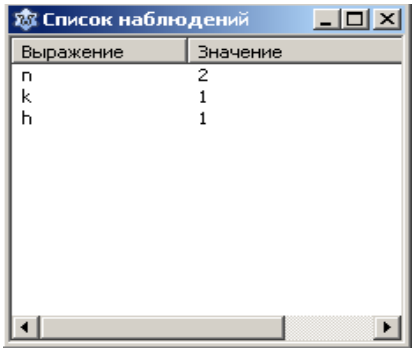


Рис. 9.30. Окно списка наблюдений

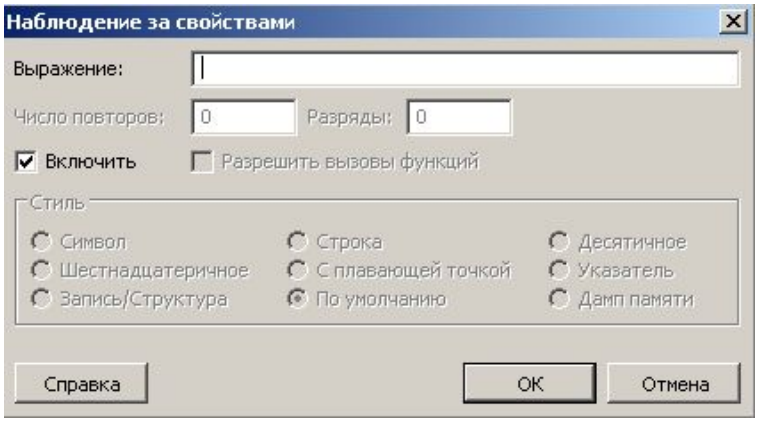


Рис. 9.31. Окно добавления новой переменной в список наблюдения

Принято считать, что отладка предшествует тестированию. То есть программист, разрабатывая программу, сначала отлаживает ее (обнаруживает и "убивает" всех "жучков"), затем приступает к тестированию. Но, вообще говоря, тестирование и отладка взаимосвязанные процессы. Зачастую при тестировании обнаруживаются ошибки, которые исправляются…отладкой, и наоборот, во время отладки обнаруживаются многие алгоритмические ошибки. Поэтому это словосочетание практически всегда употребляют вместе – тестирование и отладка.

В принципе, тестирование начинается тогда, когда ваше приложение стала вполне работоспособной, то есть перестала "вылетать", "зависать", "зацикливаться" и так далее и стала выдавать результаты. Правильные ли это результаты выявляются тестированием. Но, кроме правильности получаемых результатов, во время тестирования очень важно проверять "недопустимые", "невозможные" и даже "глупые" действия пользователя. Если он не должен вводить число 0, вы

обязательно должны проверить, как поведет себя ваша программа при вводе именно этого самого нуля и т.д. В рассмотренных нами примерах мы, по существу, этим и занимались!

9.2.10. Компоненты отображения и выбора данных

Эта группа компонентов позволяет в наглядном виде отображать информацию, а также легко осуществлять выбор нужных данных. Информация в этих компонентах содержится в виде списка (набора) строк. Для работы со списками строк разработан специальный класс `TStrings`. Этот класс является абстрактным и инкапсулирует методы для работы с наборами строк. От этого класса наследуются специализированные классы в компонентах отображения и выбора данных, которые обеспечивают доступ и обработку набора строк в соответствии с функциональностью компонента. Поскольку методы добавления и удаления строк в классе `TStrings` не реализованы и объявлены как абстрактные, все классы наследники в компонентах отображения и выбора перекрывают эти методы. Для ввода, редактирования и вывода многострочных данных удобнее использовать компонент `TMemo`.

9.2.10.1. Компонент `TMemo`

Окно компонента ведет себя как обычный текстовый редактор типа "Блокнот", т.е. доступны все стандартные функции редактирования (выделение, копирование, вставка, удаление и пр.).

Информация в `TMemo` содержится в виде совокупности (массива) строк типа `TStrings`. Каждый элемент массива содержит ровно одну строку. Доступ к отдельной строке осуществляется с помощью свойства `Lines` по ее номеру (индексу). Индекс указывается, как и положено для массивов, в квадрат-

ных скобках. Нумерация строк начинается с нуля. Общее количество строк содержится в свойстве `Lines.Count`.

Если строка не умещается целиком в окне, то можно установить свойство `WordWrap= true` и тогда не уместившаяся часть строки будет автоматически перенесена на следующую строку.

Можно также установить полосы прокрутки свойством `ScrollBars`. Возможные значения свойства:

- `ssNone` – установлено по умолчанию, полосы прокрутки отсутствуют.
- `ssVertical` – установить вертикальную полосу прокрутки.
- `ssHorizontal` – установить горизонтальную полосу прокрутки.
- `ssBoth` – установить и вертикальную и горизонтальную полосы.
- `ssAutoVertical` – вертикальная полоса в окне компонента видна, но не доступна, пока окно не заполнится по вертикали.
- `ssHorizontal` – горизонтальная полоса в окне компонента видна, но не доступна, пока окно не заполнится по горизонтали.
- `ssAutoBoth` – объединяет два предыдущих значения.

Запретить пользователю редактирование можно, установив свойство `ReadOnly=true`.

Добавление новой строки при вводе данных с клавиатуры осуществляется нажатием клавиши `Enter`, при этом свойство `WantReturns` должно быть установлено равным `true`. Если `WantReturns= false`, то для перехода на новую строку необходимо нажать `Ctrl+Enter`.

Свойство `SelStart` указывает начало выделенного текста, а `SelLength` – длину выделенного текста (количество символов).

В свойстве `Text` весь набор строк представляется в виде одной строки с разделителями возврат каретки и перенос строки (#13#10) между строками.

Посмотрим, как программно заполнять поле ввода компонента. Чтобы добавить строку в `TMemo` необходимо воспользоваться методами:

- `function Add(const S: string): integer;` – добавляет строку S в конец набора строк `TMemo` и возвращает ее индекс.
- `procedure Append(const S: string);` – просто добавляет строку S в конец набора строк.

Чтобы добавить целый набор строк, например, из другого компонента `TMemo`, можно применить методы:

- `procedure AddStrings(TheStrings: Tstrings);` – где `TheStrings` набор строк типа `TStrings`, добавляет этот набор строк к существующему.
- `procedure Assign(Source: TPersistent);` – полностью очищает содержимое `TMemo` и загружает новый набор строк из `Source`.

Для вставки строки в произвольное место списка строк существует метод

`procedure Insert(Index: integer; const S: string);` – где `Index` номер (индекс) строки куда вставляется строка `S`. При этом старая строка не исчезает, а сдвигается вниз вместе со всеми нижележащими строками (их индексы автоматически увеличиваются на единицу).

Заменить содержимое какой-либо строки можно простым оператором присваивания, например, чтобы заменить содержимое строки с индексом *k* достаточно записать оператор:

`Memo1.Lines[k]:='`Содержимое заменяемой строки`';`

Добавлять строки оператором присваивания тоже можно, но при одном условии, нельзя применять этот способ во время создания формы, т.е. в обработчике `OnCreate` формы.

И, наконец, чтобы удалить строку применяется метод

`Delete(Index: integer);`

Давайте применим полученные знания на простом примере. Создайте новый проект. Положите на форму два компонента `TMemo`, две надписи и кнопку примерно так, как показано на рисунке 9.32.

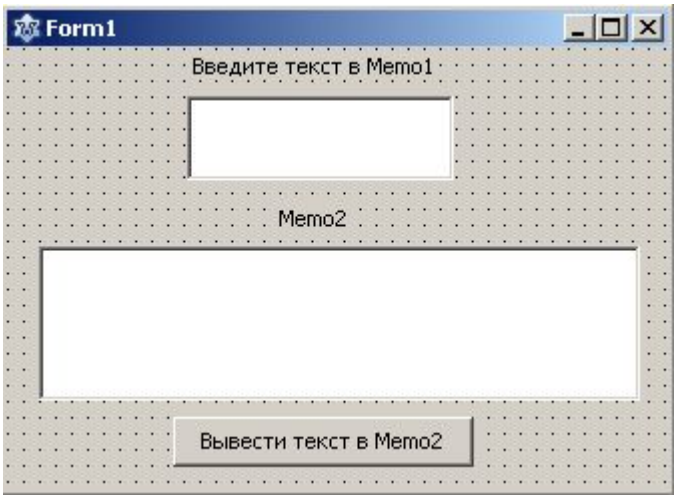


Рис. 9.32. Форма с компонентами TMemo

В обработчик `OnCreate` формы введите код:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Memo1.Lines.Append('Это длинная строка,  '+ //  метод Append
                     'не умещающаяся в окне Memo1');
  Memo1.Lines.Add('А это короткая строка'); //  метод Add
end;
```

В обработчик `OnClick` кнопки введите код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: integer;
begin
  Memo1.Lines[2]:= 'Акинфеев';   //  добавление строк
  Memo1.Lines[3]:= 'Аршавин';    //  оператором присваивания
  Memo1.Lines.Delete(2);         //  удаление строки
  Memo1.Lines.Insert(2, 'Гус Иванович');// вставка строки
  Memo2.Clear;    //  очистка набора строк Memo2
```

```
  for i:= 0 to Memo1.Lines.Count - 1 do// добавление строк
  Memo2.Lines.Append(Memo1.Lines[i]);// в цикле
  //Memo2.Lines.Add(Memo1.Lines[i]);// можно применить Add
  {добавление всего содержимого Memo1}
  Memo2.Lines.AddStrings(Memo1.Lines);// 1-й способ
  Memo2.Lines.Assign(Memo1.Lines);// 2-й, с очисткой
end;
```

Для установки параметров шрифта имеется свойство `Font`. В нем имеются такие подсвойства, как `Name` – название шрифта, `Size` – размер, `Color` – цвет, `Style` – стиль начертания шрифта и другие. Свойство `Aligment` – служит для выравнивания текста в окне `TMemo`. Оно имеет значения:

`taLeftJustify` – выравнивание по левому краю.

`taCenter` – выравнивание по центру.

`taRightJustify` – выравнивание по правому краю.

Свойство `Font` также доступно в программе. Для предыдущего примера добавьте на форму еще одну кнопку, как показано на рис. 9.33.

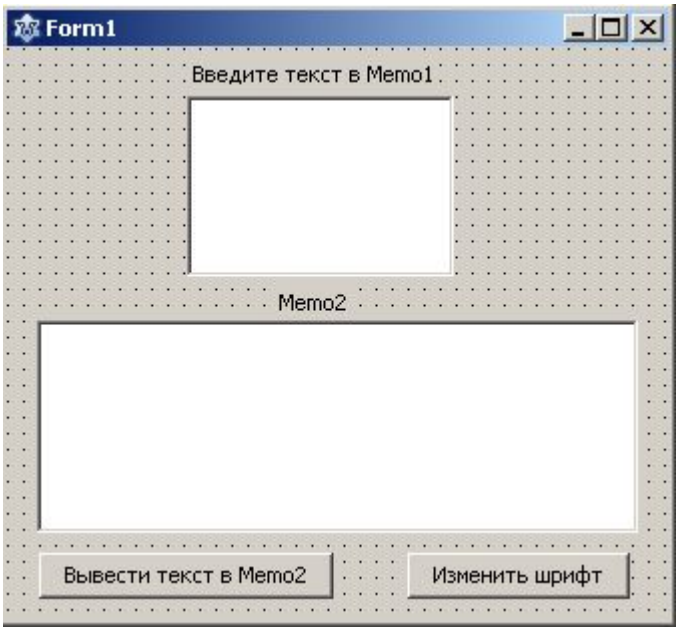


Рис. 9.33. Форма приложения

Напишите для этой кнопки следующий обработчик:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Memo2.Font.Size:= 12;
  Memo2.Font.Color:= clBlue;
end;
```

После запуска приложения и нажатия на кнопку "Изменить шрифт" текст в `Memo2` изменит свой размер и цвет.

Возникает вопрос, а нельзя ли предоставить самому пользователю право изменять атрибуты шрифта? Для этого существует специальный компонент выбора свойств шрифта `TFontDialog`. Он расположен на странице `Dialogs` палитры компонентов. Компонент является не визуальным, так как во время выполнения пользователь увидит не сам компонент, а стандартный диалог, вызываемый при обращении к этому компоненту. Поэтому его можно расположить в любом месте формы. Показ диалога инициируется специальным методом компонента `Execute`. Это функция, которая возвращает `true`, если в результате диалога пользователь в действительности выбрал какие-то параметры шрифта. Выбранные параметры запоминаются в свойствах компонента `TFontDialog`. Если же пользователь ничего не выбрал и нажал на кнопку `Отмена` или клавишу `Esc`, то функция возвращает `false`.

Таким образом, для вызова диалога необходимо записать:

```
if FontDialog1.Execute then
Memo1.Font.Assign(FontDialog1.Font);
```

Создайте новое приложение. Перенесите на форму компонент `TMemo`, `TFontDialog`, две кнопки и две надписи, как показано на рис. 9.34.

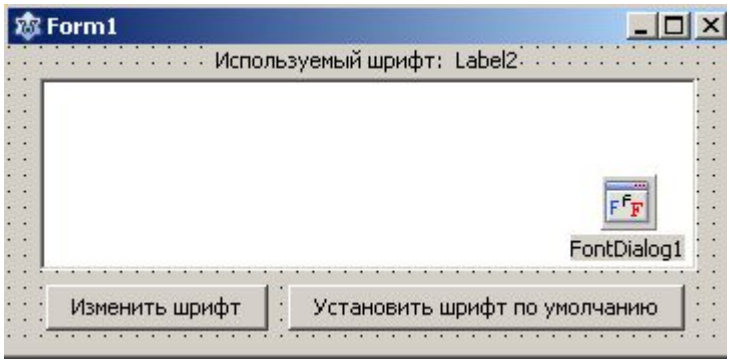


Рис. 9.34. Форма приложения с дополнительным компонентом TFontDialog

В обработчик кнопки "Изменить шрифт" введите код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if FontDialog1.Execute then
  Memo1.Font.Assign(FontDialog1.Font);
  Label2.Caption:= Memo1.Font.Name + ', '
                 + IntToStr(Memo1.Font.Size);
  Memo1.SetFocus;
end;
```

В обработчик кнопки "Установить шрифт по умолчанию" введите код:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Memo1.Font.SetDefault;
  Label2.Caption:= Memo1.Font.Name + ', '
                 + IntToStr(Memo1.Font.Size);
end;
```

И, наконец, в обработчик `OnCreate` формы введите код:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Label2.Caption:= Memo1.Font.Name + ', '
                 + IntToStr(Memo1.Font.Size);
end;
```

Теперь пользователь может сам по своему усмотрению устанавливать параметры шрифта, рис. 9.35, 9.36.

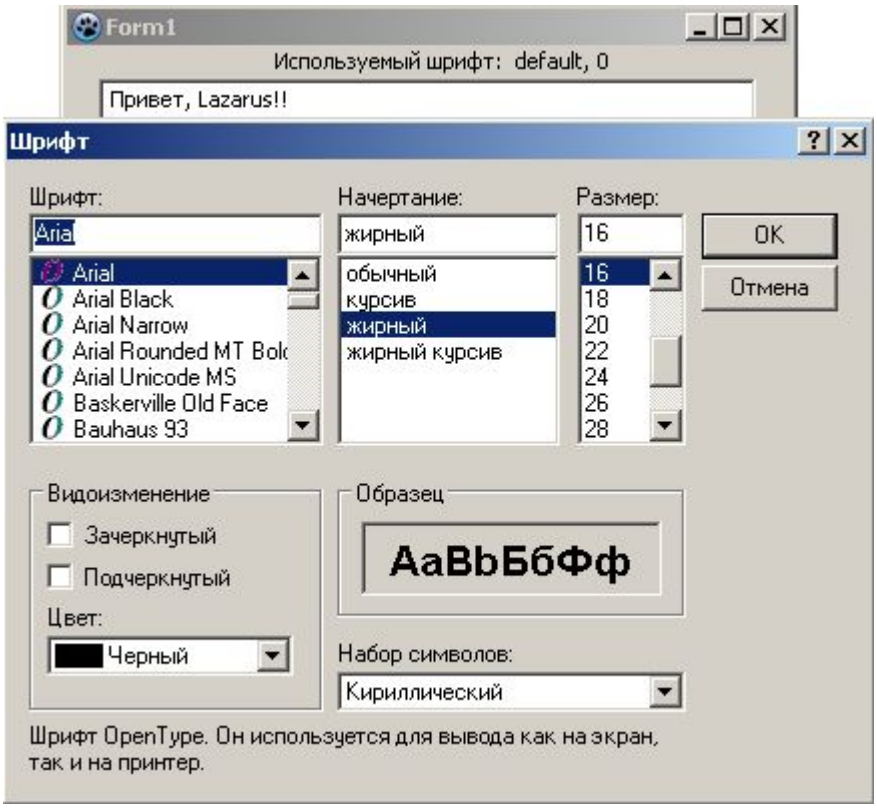


Рис. 9.35. Стандартный диалог выбора шрифта в Windows.

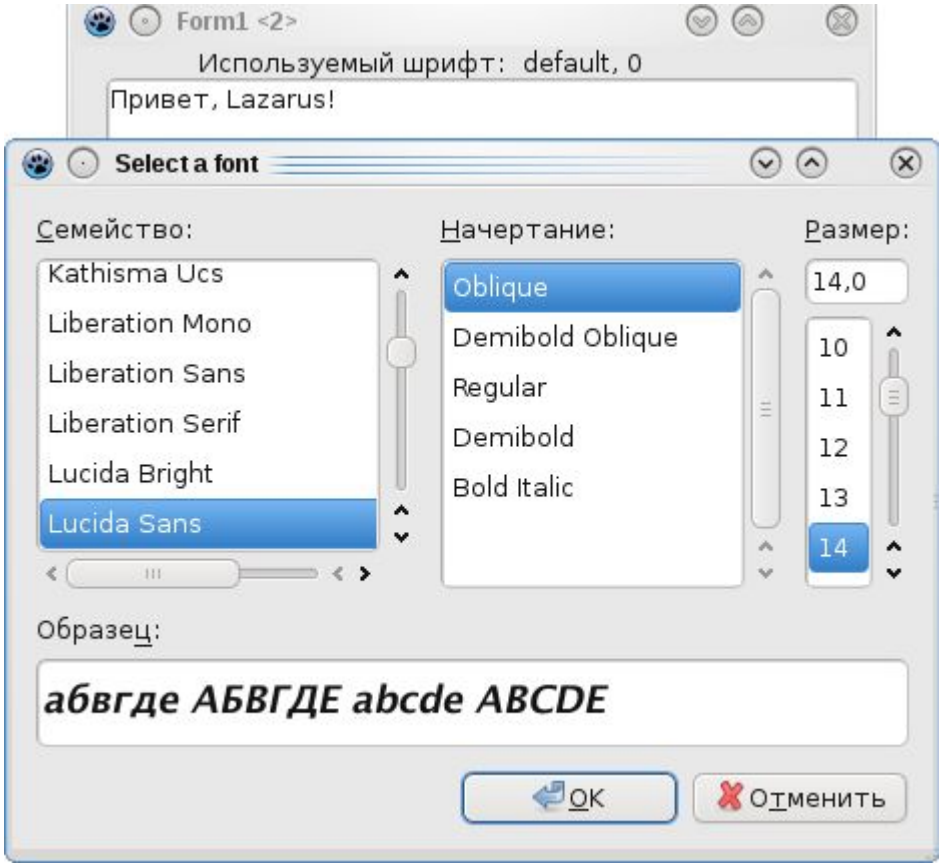


Рис. 9.36. Стандартный диалог выбора шрифта в Linux.

Точно так же пользователь может устанавливать цвет фона окна `TMemo`. Для этого используется компонент выбора цвета `TColorDialog`, расположенный в той же вкладке `Dialogs`. Видоизмените только что рассмотренный пример. Добавьте на форму компонент `TColorDialog` и кнопку, рис. 9.37.

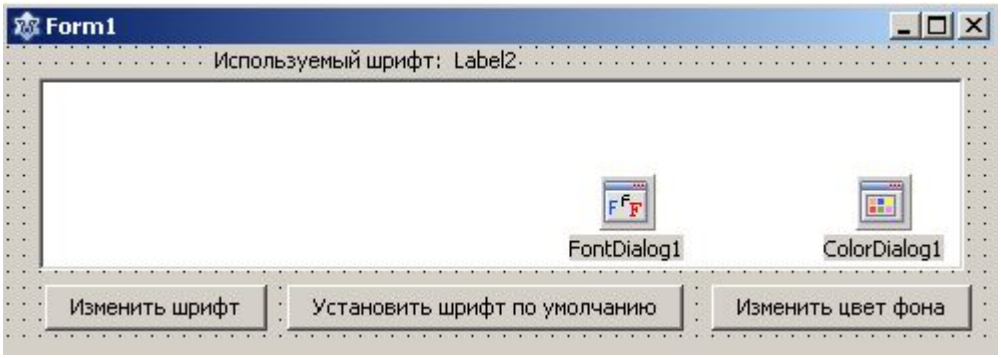


Рис. 9.37. Форма приложения

Напишите следующий обработчик для кнопки "Изменить цвет фона":

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  if ColorDialog1.Execute then
  Memo1.Color:= ColorDialog1.Color;
end;
```

Теперь пользователь может выбирать и устанавливать нужный ему цвет фона, рис. 9.38, 9.39.

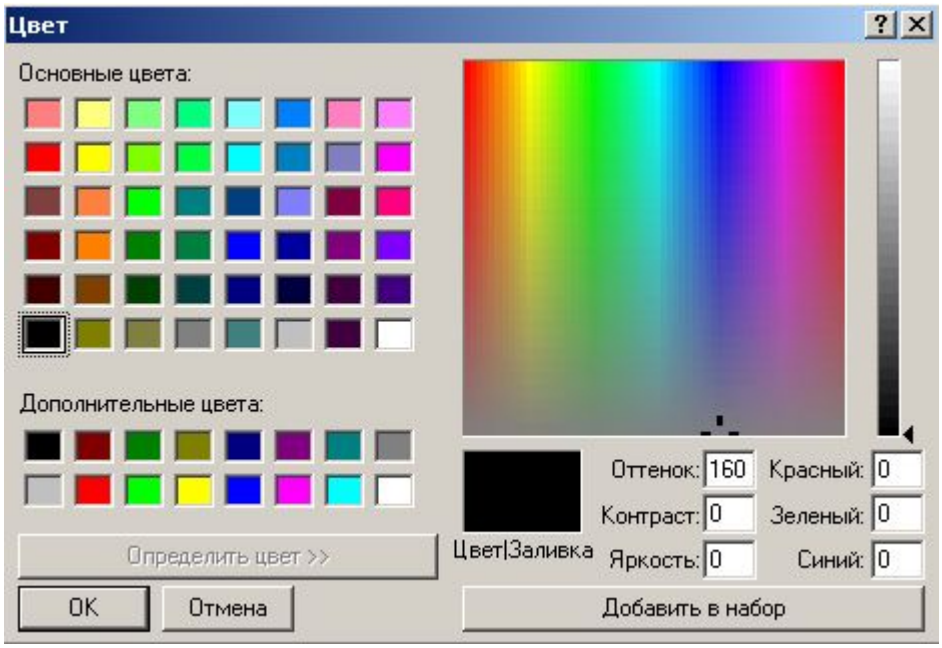


Рис. 9.38. Стандартный диалог выбора цвета фона в Windows.

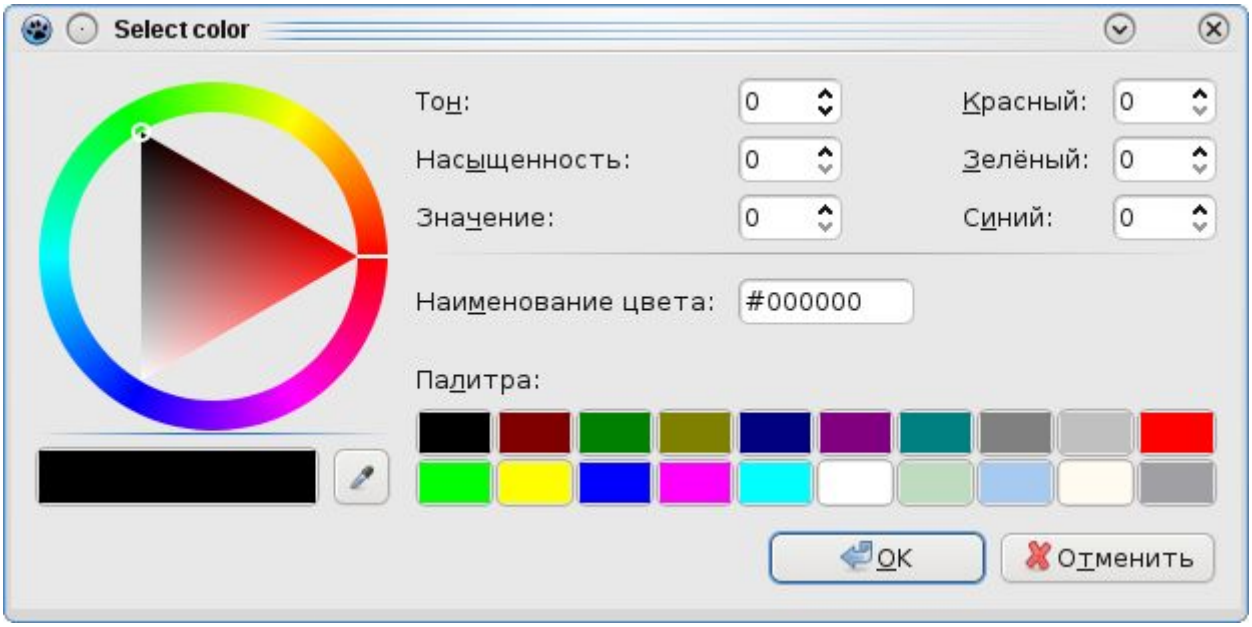


Рис. 9.39. Стандартный диалог выбора цвета фона в Linux.

Для дальнейшего нам понадобится умение работать с классом `TStringList`. Класс `TStringList` имеет широкий набор свойств и методов для работы со списками строк. Рассмотрим основные свойства и методы этого класса.

Свойства класса:

Табл. 9.3.

`Count: integer;`	Количество строк в наборе.
`Strings[Index: integer]: string;`	Строка с индексом Index
`Text: string;`	Весь набор строк представляется в виде одной строки с разделителями возврат каретки и перенос строки (#13#10) между строками.
`Duplicates: TDuplicates;`	Разрешает или не разрешает добавление одинаковых строк. Если `Duplicates= dupIgnore` – одинаковая строка не добавляется, при `Duplicates= dupAccept` – добавление одинаковых строк в набор строк разрешено, при `Duplicates= dupError` – при попытке добавления одинаковых строк генерируется исключение `EListError`
`Sorted: boolean`	Если `Sorted= true` – строки набора автоматически сортируются по алфавиту.

Методы класса:

Табл. 9.4.

`function Add(const S: string): integer;`	Добавляет строку в набор данных и возвращает ее индекс
`procedure AddStrings(strings: TStrings);`	Добавляет к текущему набору новый набор строк
`procedure Append(const S:string) ;`	Добавляет к текущему набору новый набор строк, но не возвращает индекс вставленной строки
`procedure Assign(Source: TPersistent);`	Уничтожает прежний набор строк и загружает из Source новый набор. В случае не удачи возникает исключение EconvertError
`procedure Clear;`	Очищает набор данных и освобождает связанную с ним память
`procedure Delete(Index: integer);`	Уничтожает элемент набора с индексом Index и освобождает связанную с ним память

`function Equals(Strings:TStrings): boolean;`	Сравнивает построчно текущий набор данных с набором Strings и возвращает True, если наборы идентичны
`function IndexOf(const S: string): integer;`	Для строки S возвращает ее индекс или -1, если такой строки в наборе нет
`procedure Insert(Index: integer; const S: Strings;`	Вставляет строку в набор и присваивает ей индекс Index
`procedure LoadFromFile (const FileName: Strings;`	Загружает набор из файла
`procedure LoadFromStream (Stream: TStream) ;`	Загружает набор из потока
`procedure Move(Curindex, Newindex: integer;`	Перемещает строку из положения Curindex в положение Newlndex
`procedure SaveToFile(const FileName: Strings;`	Сохраняет набор в файле
`procedure SaveToStream (Stream: TStream);`	Сохраняет набор в потоке
`function Find(const S: string; var Index: integer): boolean;`	Поиск в отсортированном наборе строк строки `S`. Если такая строка имеется, то ее индекс содержится в параметре `Index`. Если набор строк неотсортирован, то необходимо использовать функцию `IndexOf`
`Sort;`	Сортирует строки списка, свойство Sorted которого установлено в false, в возрастающей алфавитной последовательности. Если Sorted = true, то список сортируется автоматически.

Как видите, свойства и методы класса `TStringList` практически совпадают (кроме некоторых) с `TMemo`. Оно и понятно, ведь и класс `TStringList` и класс `TMemo` являются потомками класса `TStrings`. Только строки в `TStringList` содержатся в свойстве `Strings`, а в `TMemo` в свойстве `Lines`.

Задавать значения строкам с помощью оператора присваивания можно только после того, как строка создана, например методами `Add` или `AddStrings`.

Довольно часто `TMemo` заполняют содержимым какого-нибудь текстового файла или набор строк `TStringList` формируется из записей текстового файла. Для этого существует метод

```
LoadFromFile(const FileName: string);
```

где `FileName` имя файла, включая путь к нему. Если файл расположен в одной папке с программой, путь указывать не обязательно.

Точно так же, набор строк можно сохранить в виде текстового файла методом

```
SaveToFile(const FileName: string);
```

При работе с текстовыми файлами следует иметь в виду, что при выводе в `TMemo` файла с русским текстом, в Windows буквы будут отображаться некорректно. Это вызвано тем, что текстовые файлы в Windows имеют кодировку `CP-1251`. Необходимо сначала преобразовать содержимое файла в кодировку `UTF-8`. Это можно сделать следующим образом:

```
procedure Ansi_UTF8;
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create; // создание списка строк
  tfile.LoadFromFile('File in Russian.txt');
  str:= tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // преобразование в кодировку UTF-8
  {$ENDIF}
  Memo1.Lines.Add(str);
  tfile.Free;
end;
```

Функция `SysToUTF8()` преобразует строку из системной кодировки (в

данном случае CP1251) в кодировку UTF-8.

Наборы строк TStringList "невидимы" для пользователя, в отличие от набора строк TMemo. Обычно с помощью TStringList производят различные операции с набором строк, а в TMemo выводят уже окончательно сформированные (например, отсортированные) строки. Некоторые операции с набором строк можно производить и непосредственно в TMemo, но, обычно, так не делают. Так как, если вы будете делать какие-то операции со строками в TMemo, а количество строк достаточно велико, то во время выполнения вашего приложения в его окне будет что-то быстро "мелькать". Разумеется, это нехорошо. Так что, операции со строками в TMemo "на виду" у пользователя является признаком дурного тона!

Здесь мы создаем список строк типа TStringList, методом LoadFromFile загружаем файл. Для этого создайте текстовый файл с именем 'File in Russian' с русским текстом в той же папке, что и ваш проект. В примерах, содержащихся на приложенном к книге диске, такой файл имеется. Используя свойство Text, копируем содержимое списка в одну строку. Затем, если приложение выполняется в Windows, используя функцию SysToUTF8(), преобразуем полученную строку в UTF-8. И только после этого добавляем в Memo1. Если вы используете Linux, то применять функцию SysToUTF8() не надо, так как строка уже в UTF-8. В принципе ничего страшного не произойдет, если вы примените функцию SysToUTF8() к строке, которая уже в кодировке UTF-8. Функция просто вернет исходную строку, не преобразовывая ее. Если хотите, вы можете убрать директивы компилятора

```
{$IFDEF WINDOWS}
{$ENDIF}
```

Обратите внимание, что после завершения работы с TStringList мето-

дом Free память, выделенная набору строк, освобождается.

Точно так же, при сохранении списка строк TMemo в файл необходимо обратное преобразование, например, вот таким образом:

```
procedure UTF8_Ansi;
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create; // создание списка строк
  str:=Memo1.Text;
  {$IFDEF WINDOWS}
    str:= UTF8ToSys(str);// преобразование в системную кодировку
  {$ENDIF}
  tfile.Add(str);
  tfile.SaveToFile('File in Russian.txt');
  tfile.Free;
end;
```

В этой программе загружается файл с фиксированным именем 'File in Russian.txt' находящийся в той же папке, что и сама программа. Сохранение производится в файл с тем же именем.

Естественно, можно предоставить возможность пользователю самому выбирать файл для открытия и задавать имя файла при сохранении. Для этого используется компоненты TOpenDialog и TSaveDialog из вкладки Dialogs. Рассмотрим некоторые свойства этих компонентов:

- DefaultExt - указывает значение расширения файла по умолчанию. В нашем случае укажите DefaultExt= .txt
- Filter – задает шаблоны фильтра при выборе файлов. Если в окне этого

свойства нажать на кнопку с троеточием, появится окно редактора фильтров, рис. 9.40.

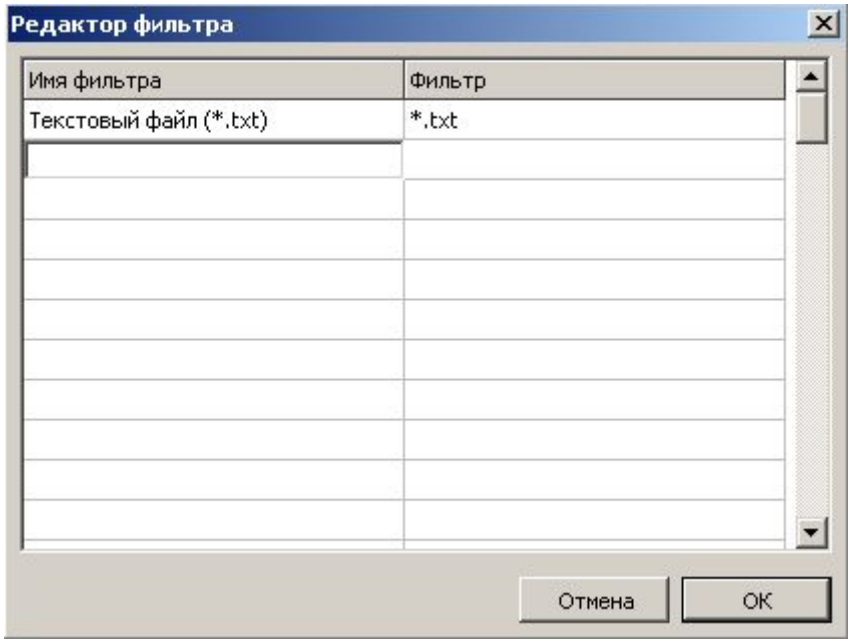


Рис. 9.40. Редактор фильтров

Задайте фильтр как показано на рисунке. Тогда значение свойства будет равно `Filter= 'Текстовый файл (*.txt)|*.txt'`, т.е. имя фильтра и сам фильтр разделены вертикальной линией. Если вы зададите еще один фильтр, то они будут разделены точкой с запятой. Таким образом, значение свойства `Filter` можно задавать программно.

- `FilterIndex` – определяет номер фильтра, который будет по умолчанию показан пользователю в момент открытия диалога.
- `InitialDir` – определяет начальный каталог, который будет открыт в момент начала работы пользователя с диалогом. Если значение этого свойства не задано, то открывается текущий каталог или тот, который был открыт при последнем обращении пользователя к соответствующему диалогу в процессе выполнения данного приложения.
- `Title` – задает заголовок диалогового окна.

Для вызова диалога открытия и загрузки файла в `TMemo` следует написать следующий обработчик:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  fname, str: string;
  tfile: TStringList;

begin
  tfile:= TStringList.Create; // создание списка строк
  if OpenDialogl.Execute
  then fname:= OpenDialogl.FileName;

  // преобразование в системную кодировку
  fname:= UTF8ToSys(fname);
  tfile.LoadFromFile(fname);
  str:= tfile.Text;

  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // преобразование в кодировку UTF-8
  {$ENDIF}

  Memo1.Lines.Add(str);
  tfile.Free;
end;
```

После запуска приложения и нажатии кнопки `Button1` будет открыт диалог открытия файла, рис. 9.41, 9.42.

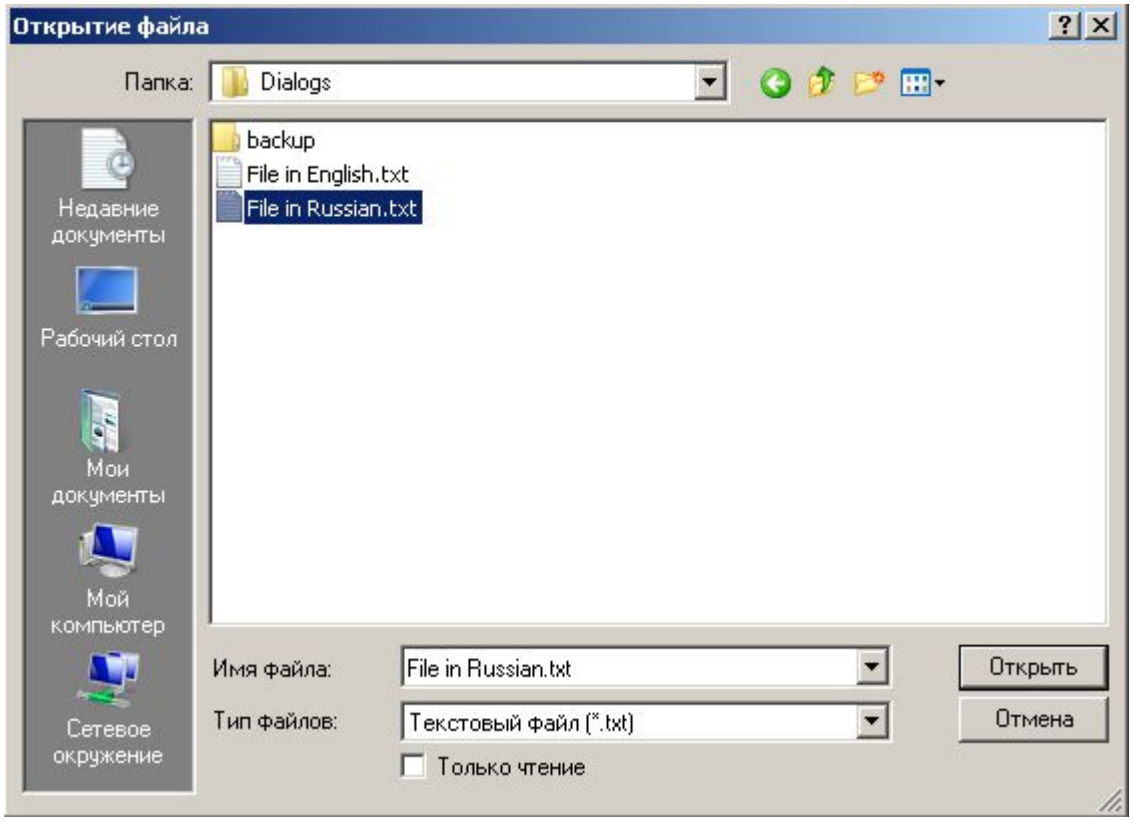


Рис. 9.41. Стандартный диалог открытия файла в Windows.

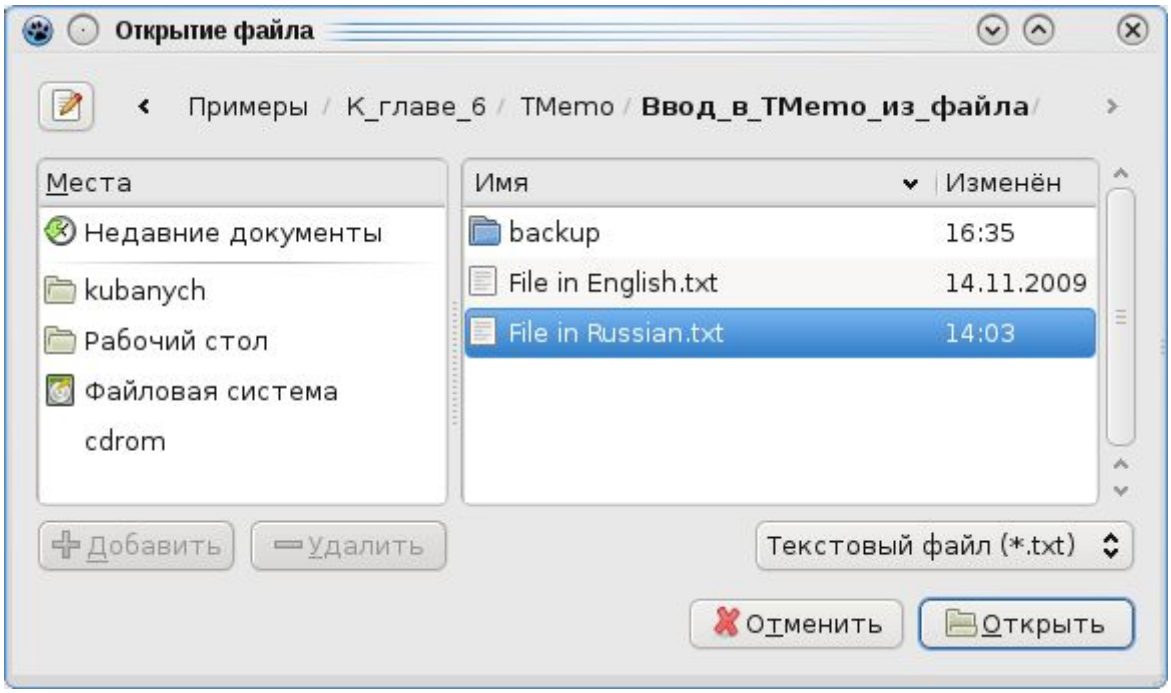


Рис. 9.42. Стандартный диалог открытия файла в Linux.

После выбора пользователем файла, в свойстве

`OpenDialog1.FileName` будет находиться имя файла вместе с путем к нему. Обратите внимание на оператор

```
fname:= UTF8ToSys(fname);
```

Если имя файла, а также путь содержит кириллицу, то необходимо строку с именем файла и путем преобразовать в системную кодировку.

Для вызова диалога сохранения файла необходимо написать следующий код:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  fname, str: string;
  tfile: TStringList;
begin
  tfile:= TStringList.Create; // создание списка строк
  str:=Memo1.Text;
  {$IFDEF WINDOWS}
    str:= UTF8ToSys(str);// преобразование в системную кодировку
{$ENDIF}
  tfile.Add(str);
  SaveDialogl.FileName:= fname;
  if SaveDialogl.Execute
  then fname:= SaveDialogl.FileName;
  // преобразование в системную кодировку
  fname:= UTF8ToSys(fname);
  tfile.SaveToFile(fname);
  tfile.Free;
end;
```

При нажатии кнопки `Button2` будет открыт стандартный диалог сохранения файла, рис. 9.43, 9.44.

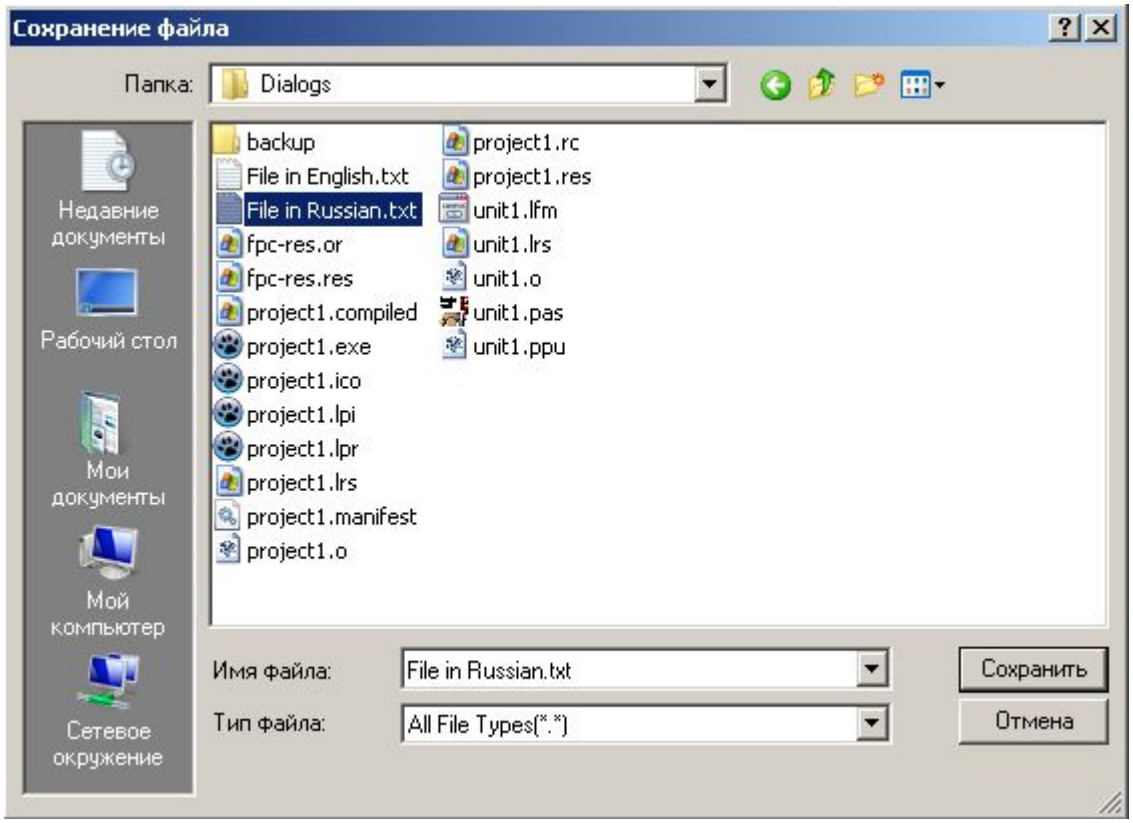


Рис. 9.43. Стандартный диалог сохранения файла в Windows.

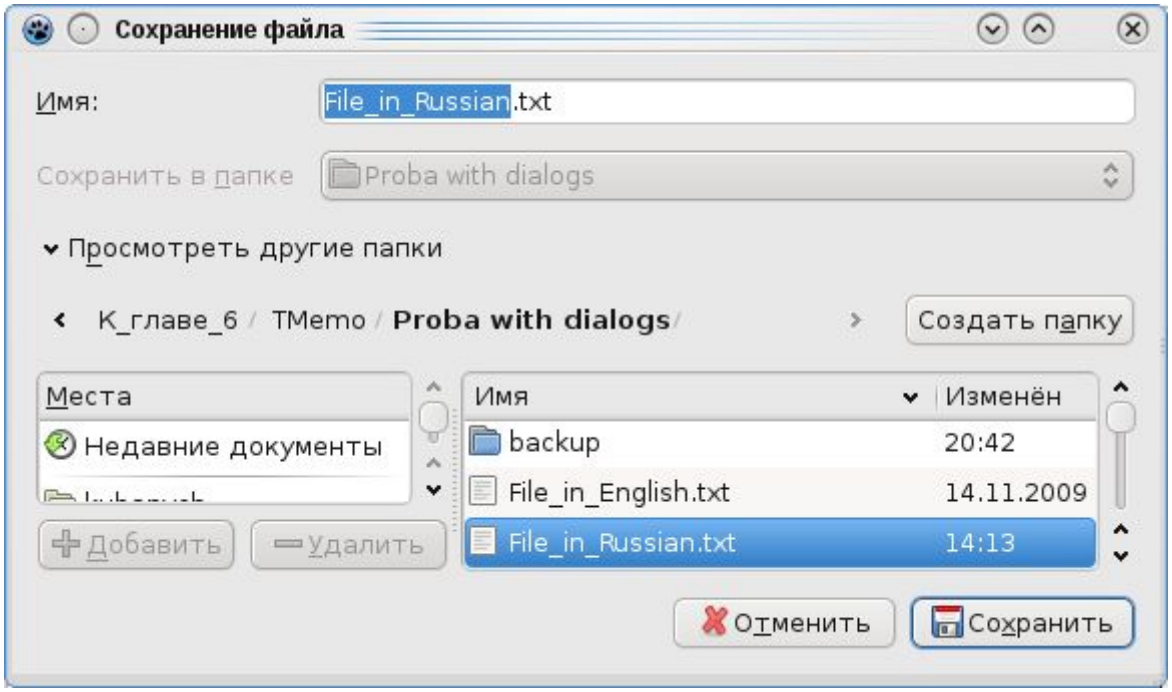


Рис. 9.44. Стандартный диалог сохранения файла в Linux.

Так, загружать файлы в `TMemo` и сохранять мы научились. Теперь давайте посмотрим, какие манипуляции со строками можно будет делать.

Во-первых, можно сортировать. Если набор строк содержит только символы латиницы, организовать сортировку достаточно тривиально. Можно установить просто свойство `Sorted: true` в `TStringList` и строки будут отсортированы по алфавиту автоматически. Можно применить метод `Sort`.

Создайте новый проект. Спроектируйте окно будущего приложения на форме так, как показано на рис. 9.45.

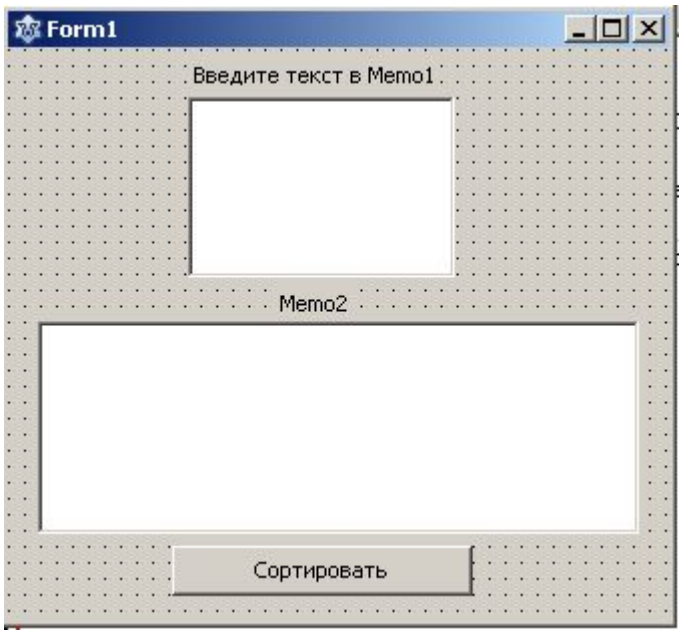


Рис. 9.45. Форма приложения

В обработчике нажатия кнопки введите код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  strList: TStringList;
begin
  strList:= TStringList.Create;
  strList.Duplicates:= dupAccept;
```

```
  strList.AddStrings(Memo1.Lines);
  strList.Sort;
  Memo2.Clear;
  Memo2.Lines.AddStrings(strList);
  strList.Free;
end;
```

В Memo1 введите несколько строк латиницей. При нажатии кнопки "Сортировать" в Memo2 выведутся строки, отсортированные в алфавитном порядке. Обратите внимание, что после завершения работы с TStringList методом Free память, выделенная набору строк, освобождается.

Однако этот код не работает для строк, содержащих кириллицу. Происходит это, опять же, из-за кодировки UTF-8. По умолчанию TStringList использует следующую функцию, которая работает с ANSI строками:

```
function TStringList.DoCompareText(const s1,s2 : string)
: PtrInt;
begin
    if FCaseSensitive then
       result:= AnsiCompareStr(s1,s2)
    else
       result:= AnsiCompareText(s1,s2);
end;
```

К счастью, среди методов класса TStringList имеется метод CustomSort. Его объявление в классе имеет вид:

```
type
TStringListSortCompare = function(List: TStringList;
Indexl, Index2: integer): integer;
```

```
procedure CustomSort(CompareFn: TStringListSortCompare);
```

Можно воспользоваться методом `CustomSort` передав ему в качестве параметра свою функцию сравнения. Чтобы отсортировать список, построим функцию сравнения двух элементов `S1` и `S2` с индексами `indexl` и `index2` соответственно, которая должна возвращать:

< 0 — если, `S1` < `S2`;

0 — если, `S1` = `S2`;

> 0 — если, `S1` > `S2`;

Функция имеет вид:

```
function ListSortRus(List: TStringList; Index1, Index2:
integer): integer;
begin
  Result:= WideCompareText(UTF8Decode(List[Index1]),
           UTF8Decode(List[Index2]));
end;
```

Здесь используется функция:

```
function WideCompareText(const S1: WideString; const S2:
WideString): integer;
```

Она работает со строками `WideString`. Тип `WideString` представляет собой динамически размещаемые в памяти строки, длина которых ограничена только объемом свободной памяти. Каждый символ строки типа `WideString` является `Unicode`-символом. Функция возвращает результат:

< 0 — если, `S1` < `S2`;

0 — если, `S1` = `S2`;

> 0 — если, S1 > S2;

Это как раз то, что нам нужно! В разделе implementation добавьте описание функции

```
function ListSortRus(List: TStringList; Index1, Index2: integer): integer;
begin
  Result:= WideCompareText(UTF8Decode(List[Index1]),
           UTF8Decode(List[Index2]));
end;
```

Функция

UTF8Decode(const S: UTF8String): WideString - преобразует строку формата UTF-8 к строке формата Unicode.

Перепишите обработчик OnClick кнопки:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  strList: TStringList;
begin
  strList:= TStringList.Create;
  strList.Duplicates:= dupAccept;
  strList.AddStrings(Memo1.Lines);
  strList.CustomSort(@ListSortRus);
  Memo2.Clear;
  Memo2.Lines.AddStrings(strList);
  strList.Free;
end;
```

В обработчике изменилась только одна строка. Вместо метода `Sort` вызывается `CustomSort(@ListSortRus)`. Как видите, чтобы передать функцию как параметр, надо перед именем функции поставить знак @.

Теперь сортировка будет правильно работать как для латиницы, так и для кириллицы.

Во-вторых, довольно часто необходимо осуществлять поиск. Причем не только отдельных строк целиком, но и некоторой части строки (говорят еще подстроки). Если говорить о тексте, то необходимо найти слово, фразу или предложение, причем предложение может занимать несколько строк.

Для поиска удобно использовать свойство `Text` компонента `TMemo` в котором весь набор строк представляется в виде одной длинной строки и функцию `Pos`, которую мы изучали в 6.3. Составим программу поиска слов в текстовом файле. Создайте новый проект. Поместите на форму компоненты как показано на рис. 9.46.

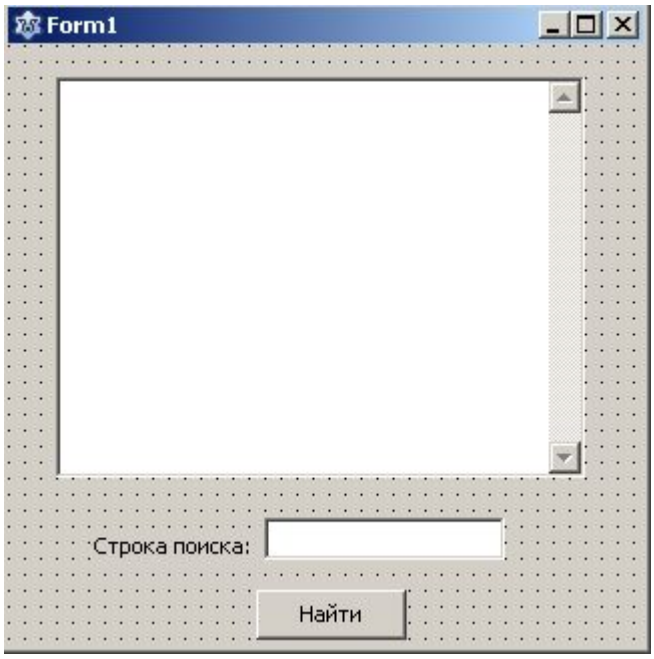


Рис. 9.46. Форма приложения

Введите следующий код:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  Memo1.Lines.LoadFromFile('File in English.txt');
  Edit1.SetFocus;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  k: integer;
begin
  Edit1.SetFocus;
  Edit1.SelectAll;
  if Edit1.Text= '' then
  begin
    ShowMessage('Введите строку для поиска');
    exit;
  end;
  k:= Pos(Edit1.Text, Memo1.Text);
  if k > 0 then
  begin
    Memo1.SetFocus;
    Memo1.SelStart:= k - 1 ;
    Memo1.SelLength:= Length(Edit1.Text);
  end
  else
    ShowMessage('Строка не найдена');
end;
```

Во время создания окна программы в `Memo1` загружается текстовый файл.

В обработчике нажатия кнопки осуществляется поиск фрагмента текста, введенного пользователем в поле ввода `Edit1`. Для выделения найденного фрагмента используется свойство `Memo1.SelStart` и `Memo1.SelLength`. Чтобы найденный текст был выделен необходимо, чтобы фокус был установлен в `Memo1`. В этом примере мы осуществляли поиск в текстовом файле, в котором содержится только английские буквы. Для текста, содержащего кириллицу, наша программа работать корректно не будет.

Поэтому нам придется видоизменить программу для поиска в текстовых файлах, содержащих кириллицу. В обработчике `OnCreate` формы загружаем кириллический текст в `Memo1`, предварительно преобразовав его в `UTF-8` (для Windows). В обработчике `OnClick` мы используем уже знакомую нам функцию `UTF8Decode`, а также функции `UTF8Pos` и `UTF8Length`. Не забудьте включить модуль `LCLProc` в объявление `uses`.

```
procedure TForm1.FormShow(Sender: TObject);
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile('File in Russian.txt');
  str:= tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // преобразование в кодировку UTF-8
  {$ENDIF}
  Memo1.Lines.Add(str);
  tfile.Free;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
  k: integer;
  text_UTF8, str_UTF8: string;
begin
  Edit1.SetFocus;
  if Edit1.Text= '' then
  begin
    ShowMessage('Введите строку для поиска');
    exit;
  end;
  text_UTF8:= UTF8Decode(Memo1.Text);
  str_UTF8:= UTF8Decode(Edit1.Text);
  k:= UTF8Pos(str_UTF8, text_UTF8);
  if k > 0 then
  begin
    Memo1.SetFocus;
    Memo1.SelStart:= k - 1;
    Memo1.SelLength:= UTF8Length(str_UTF8);
  end
  else
    ShowMessage('Строка не найдена');
end;
```

Позволим пользователю заканчивать ввод в `Edit1` нажатием клавиши `Enter`. Чтобы избежать дублирования кода, создадим процедуру `CyrillicSearch` и в обработчиках будем вызывать эту процедуру:

```
unit Unit1;
```

```
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,   Forms,
  Controls, Graphics, Dialogs, StdCtrls, LconvEncoding,
  LCLProc;
type
  { TForm1 }

  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    Memo1: TMemo;
    procedure Button1Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject;
                            var Key: char);
    procedure CyrillicSearch;
    procedure FormShow(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{ TForm1 }
procedure TForm1.CyrillicSearch;
```

```
var
  k: integer;
  text_UTF8, str_UTF8: string;
begin
  Edit1.SetFocus;
  if Edit1.Text= '' then
  begin
    ShowMessage('Введите строку для поиска');
    exit;
  end;
  text_UTF8:= UTF8Decode(Memo1.Text);
  str_UTF8:= UTF8Decode(Edit1.Text);
  k:= UTF8Pos(str_UTF8, text_UTF8);
  if k > 0 then
  begin
    Memo1.SetFocus;
    Memo1.SelStart:= k - 1;
    Memo1.SelLength:= UTF8Length(str_UTF8);
   end
  else
    ShowMessage('Строка не найдена');
end;

procedure TForm1.FormShow(Sender: TObject);
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
```

```
  tfile.LoadFromFile('File in Russian.txt');
  str:= tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // преобразование в кодировку UTF-8
  {$ENDIF}
  Memo1.Lines.Add(str);
  tfile.Free;
  Edit1.SetFocus;
end;

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

procedure TForm1.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
  if Key = #13 then
  begin
    Key:= #0;
    CyrillicSearch;
  end;
end;
initialization
  {$I unit1.lrs}
end.
```

При такой реализации наша программа находит лишь первое вхождение

искомой подстроки. Чтобы найти все вхождения будем "скармливать" функции Pos строку каждый раз без последней найденной подстроки. Для этого используем свойства SelStart и SelLength Memo1 и функцию UTF8Copy таким образом:

```
text_UTF8:=UTF8Copy(UTF8Decode(Memo1.Text),
           Memo1.SelStart + 1 + Memo1.SelLength,
           Length(UTF8Decode(Memo1.Text)) -
           Memo1.SelStart - Memo1.SelLength);
```

Функцию UTF8Copy мы рассматривали в 6.3.1.4. Чтобы выделить в Memo1 следующую найденную подстроку оператор

```
Memo1.SelStart:= k - 1;
```

надо заменить на

```
Memo1.SelStart:=Memo1.SelStart +
                Memo1.SelLength + k - 1;
```

Теперь программа находит все вхождения заданной подстроки в Memo1. Для нахождения следующего вхождения нужно нажимать на кнопку "Найти". Нажимать на клавишу Enter нельзя, так как фокус находится в Memo1. Чтобы продолжить поиск нажатием Enter надо клавишей Tab или мышью установить фокус на Edit1. Появляется некоторое неудобство в работе. Если же мы в программе сразу после нахождения подстроки и его выделения передадим фокус Edit1, то с найденного фрагмента выделение будет снято. Что тоже нехорошо.

Можно добавить обработчик Memo1KeyPress, чтобы только при нажатии Enter передать фокус Edit1. Итак, код улучшенной программы (предва-

рительно установите в `Memo1` свойство `WantReturns= false`):

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, FileUtil, LResources, Forms,
  Controls, Graphics, Dialogs, StdCtrls, LconvEncoding,
  LCLProc;
type
  { TForm1 }

  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    Memo1: TMemo;
    procedure Button1Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject;
                            var Key: char);
    procedure CyrillicSearch;
    procedure FormShow(Sender: TObject);
    procedure Memo1KeyPress(Sender: TObject;
                            var Key: char);
  private
    { private declarations }
  public
    { public declarations }
  end;
```

```
var
  Form1: TForm1;
implementation
{ TForm1 }

procedure TForm1.CyrillicSearch;
var
  k: integer;
  text_UTF8, str_UTF8: string;
begin
  Edit1.SetFocus;
  if Edit1.Text= '' then
  begin
    ShowMessage('Введите строку для поиска');
    exit;
  end;
  text_UTF8:= UTF8Copy(UTF8Decode(Memo1.Text),
              Memo1.SelStart + 1 + Memo1.SelLength,
              UTF8Length(UTF8Decode(Memo1.Text)) -
              Memo1.SelStart - Memo1.SelLength);
  str_UTF8:= UTF8Decode(Edit1.Text);
  k:= UTF8Pos(str_UTF8, text_UTF8);
  if k > 0 then
  begin
    Memo1.SetFocus;
    Memo1.SelStart:= Memo1.SelStart +
                     Memo1.SelLength + k - 1;
    Memo1.SelLength:= UTF8Length(str_UTF8);
   end
```

```
  else
  begin
    ShowMessage('Строка не найдена');
    Memo1.SelStart:= 0;
    Memo1.SelLength:= 0;
  end;
end;

procedure TForm1.FormShow(Sender: TObject);
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile('File in Russian.txt');
  str:= tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // преобразование в кодировку UTF-8
  {$ENDIF}
  Memo1.Lines.Add(str);
  Memo1.SelStart:= 0;
  Memo1.SelLength:= 0;
  tfile.Free;
  Edit1.SetFocus;
end;

procedure TForm1.Memo1KeyPress(Sender: TObject;
                               var Key: char);
begin
```

```
  if Key = #13 then
     Edit1.SetFocus;
     CyrillicSearch;
end;

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

procedure TForm1.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
  if Key = #13 then
  begin
    Key:= #0;
    CyrillicSearch;
  end;
end;
initialization
  {$I unit1.lrs}
end.
```

Практически во всех текстовых редакторах функцию поиска можно запустить, используя сочетание клавиш `Ctrl+F`. А продолжение поиска нажатием клавиши `F3`. Возьмите, например, `Word`, "Блокнот" или "Редактор исходного кода" `Lazarus`.

Давайте и мы реализуем такой же вызов поиска в `TMemo`. Заодно научимся использовать в программе несколько форм. Создайте новый проект. Поместите

на форму компонент `TMemo`, рис. 9.47.

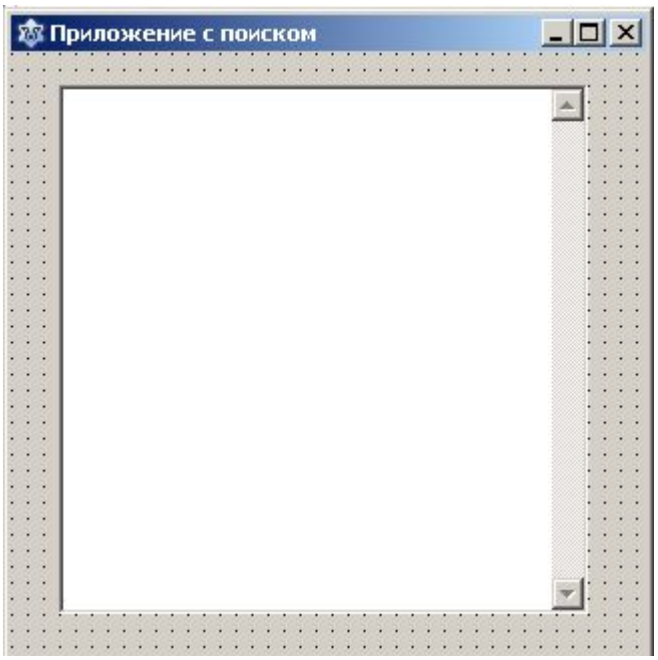


Рис. 9.47. Главная форма приложения

В меню "Файл" выберите пункт "Создать форму". В окне редактора исходного кода появится новая пустая форма и соответствующий ей модуль с именем `Unit2`. Поместите на вторую форму надпись, `TEdit` и кнопку, рис. 9.48.

Рис. 9.48. Вторая форма приложения

Если у вас в приложении несколько форм, то для открытия нужной нажмите `Shift+F12` и в появившемся окне выберите нужную вам форму, рис. 9.49.

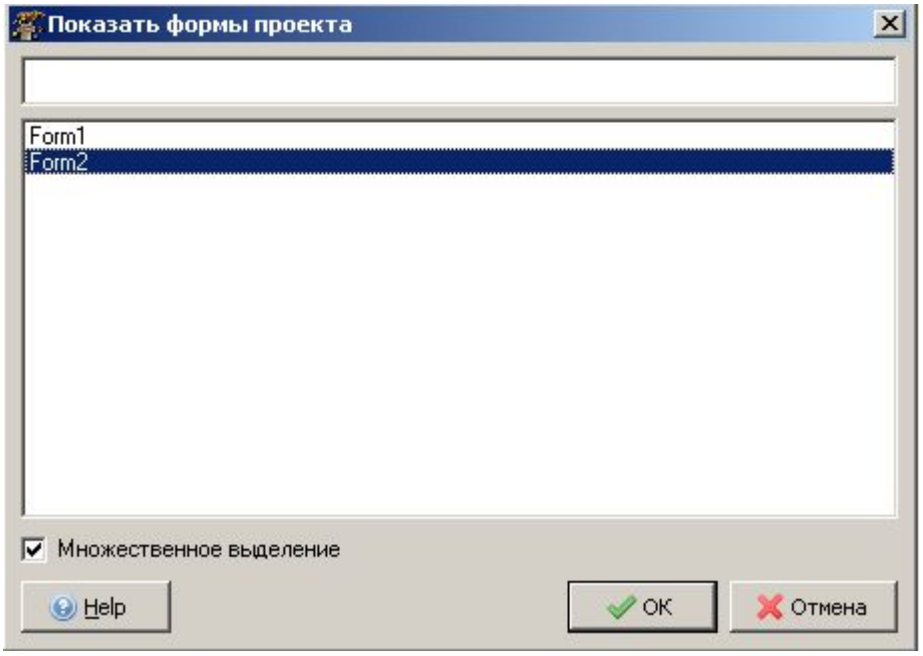


Рис. 9.49. Окно выбора формы

В `Unit1` после ключевого слова `implementation` поместите оператор

```
uses Unit2;
```

А в `Unit2` после ключевого слова `implementation` поместите оператор

```
uses Unit1;
```

Ранее, для распознавания нажатого символа мы использовали обработчик события `OnKeyPress`, однако при этом событии нельзя распознать нажатие функциональных и служебных клавиш. Для их распознавания подходит событие `OnKeyDown`.

Заголовок обработчика события `OnKeyDown` имеет следующий вид:

```
procedure TForm1.Memo1KeyDown(Sender: TObject;
          var Key: Word; Shift: TShiftState);
```

Параметр Key определяет нажатую в момент события клавишу. Так же как и для события OnKeyPress он определен как var, т.е. может изменяться в обработчике события. Но, обратите внимание, что это целое число, а не символ. Коды клавиш можно указывать в десятичном или шестнадцатеричном виде. Для некоторых клавиш введены также именованные константы, которые облегчают написание программы, поскольку не требуют помнить численные коды клавиш. Например, вместо

```
if (Key = 13) then ... ;
```

можно записать

```
if (Key = VK_RETURN) then ... ;
```

Для остальных клавиш удобнее использовать функцию ord, которая возвращает код символа.

Параметр Shift это множество элементов, отражающих нажатые в это время служебные клавиши. Элементы этого множества ssShift – нажата клавиша Shift, ssAlt – нажата клавиша Alt и ssCtrl – нажата клавиша Ctrl. Поскольку Shift является множеством, проверять наличие в нем тех или иных элементов надо операцией in. Таким образом, для определения нажатия клавиш Ctrl+F можно написать следующий код:

```
if((Key = ord('F')) and (ssCtrl in Shift)) then...;
```

При нажатии Ctrl+F будем открывать окно диалога поиска. Окно будем открывать как модальное методом ShowModal. Модальным окном называется такое окно, в котором пользователь не может переключаться на другие окна, пока не закроет текущее окно. Код программы поиска в TMemo будет состоять

из двух модулей.

Модуль `Unit1`:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, FileUtil, LResources, Forms,
  Controls, Graphics, Dialogs, StdCtrls, LconvEncoding,
  LCLProc;
type
  { TForm1 }
  TForm1 = class(TForm)
    Memo1: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure Memo1KeyDown(Sender: TObject;
                           var Key: Word;
                           Shift: TShiftState);
    procedure CyrillicSearch;

  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
```

```
uses Unit2;
{ TForm1 }
procedure TForm1.CyrillicSearch;
var
  k: integer;
  text_Area: string;
begin
  text_Area:= UTF8Copy(UTF8Decode(Memo1.Text),
              Memo1.SelStart + 1 + Memo1.SelLength,
              UTF8Length(UTF8Decode(Memo1.Text)) -
              Memo1.SelStart - Memo1.SelLength);
  k:= UTF8Pos(str_Search, text_Area);
  if k > 0 then
  begin
    Memo1.SetFocus;
    Memo1.SelStart:= Memo1.SelStart +
                     Memo1.SelLength + k - 1;
    Memo1.SelLength:= UTF8Length(str_Search);
  end
  else
  begin
    ShowMessage('Строка не найдена');
    Memo1.SelStart:= 0;
    Memo1.SelLength:= 0;
  end;
end;
procedure TForm1.Memo1KeyDown(Sender: TObject;
                              var Key: Word;
                              Shift: TShiftState);
```

```
begin
  if((Key = ord('F')) and (ssCtrl in Shift)) then
  begin
    Memo1.SelStart:= 0;
    Memo1.SelLength:= 0;
    Form2.ShowModal;
   end
  else
  if Key = 114 then   // десятичный код клавиши F3
     CyrillicSearch;
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile('File in Russian.txt');
  str:= tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // преобразование в кодировку UTF-8
  {$ENDIF}
  Memo1.Lines.Add(str);
  tfile.Free;
end;
initialization
  {$I unit1.lrs}
end.
```

Модуль `Unit2:`

```
unit Unit2;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,   Forms,
  Controls, Graphics, Dialogs, StdCtrls;
type
  { TForm2 }
  TForm2 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject;
                            var Key: char);
    procedure GetSearchString;
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form2: TForm2;
  str_Search: string;
implementation
uses Unit1;
{ TForm2 }
procedure TForm2.GetSearchString;
begin
```

```
  if Edit1.Text='' then
  begin
    ShowMessage('Введите строку для поиска');
    exit;
  end;
  Edit1.AutoSelect:= true;
  if str_Search <> UTF8Decode(Edit1.Text) then
  begin
    str_Search:= UTF8Decode(Edit1.Text);
  end;
  Form2.Close; // закрытие формы
  Form1.CyrillicSearch;
end;
procedure TForm2.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
  if Key = #13 then
  begin
    Key:= #0;
    GetSearchString;
  end;
end;
procedure TForm2.Button1Click(Sender: TObject);
begin
  GetSearchString;
end;
initialization
  {$I unit2.lrs}
end.
```

В нашем приложении предполагается, что пользователь обязательно воспользуется функцией поиска. Но ведь это происходит не всегда! Вспомните, часто ли вы пользуетесь поиском? А по умолчанию все формы создаются в момент запуска приложения. Значит, если пользователь не воспользовался поиском, форма `Form2` будет создана "зря"! Она будет занимать лишнюю память. В хороших программах такого не должно быть, особенно если имеются много форм. Выход заключается в том, чтобы создавать форму тогда, когда это необходимо. Для этого в свойствах проекта (меню Проект-> Параметры проекта-> Формы) уберите `Form2` из списка автосоздаваемых форм в список доступных, рис. 9.50.

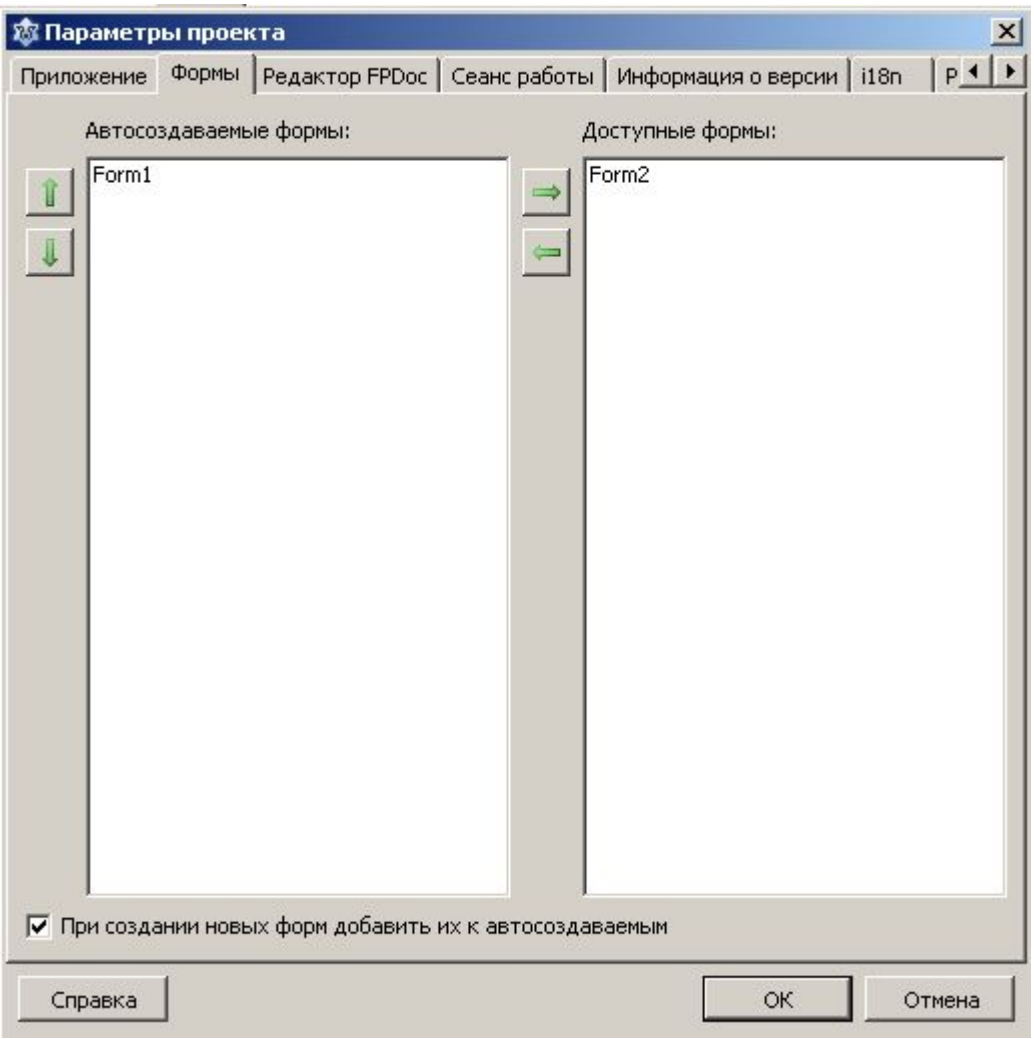


Рис. 9.50. Окно списка автосоздаваемых форм

Будем создавать `Form2` только тогда, когда пользователь нажмет комбинацию клавиш `Ctrl+F`. В обработчике события `Memo1KeyDown` перед оператором

```
Form2.ShowModal;
```

запишите оператор

```
Application.CreateForm(TForm2, Form2);
```

После того, как нам форма уже не нужна, мы ее закрываем оператором

```
Form2.Close;
```

Но закрытие формы еще не означает, что она удалена из памяти. Она еще "жива зараза" и мы можем в любой момент ее снова показать методом `ShowModal`, на случай, если пользователь повторно запустит функцию поиска.

Форма будет "окончательно" уничтожена после закрытия главного окна приложения. Следовательно, перед созданием формы мы должны проверить, может быть, она уже была создана. Теперь уже перед оператором

```
Application.CreateForm(TForm2, Form2);
```

вставьте оператор

```
if not Assigned(Form2) then
```

Приведу окончательную редакцию обработчика `Memo1KeyDown`:

```
procedure TForm1.Memo1KeyDown(Sender: TObject;
                              var Key: Word;
                              Shift: TShiftState);
begin
  if((Key = ord('F')) and (ssCtrl in Shift)) then
  begin
    Memo1.SelStart:= 0;
    Memo1.SelLength:= 0;
    if not Assigned(Form2) then
      Application.CreateForm(TForm2, Form2);
      Form2.ShowModal;
   end
  else
  if Key = 114 then    // десятичный код клавиши F3
     CyrillicSearch;
end;
```

В заключение, напишем программу решения системы линейных алгебраических уравнений методом Гаусса с использованием TMemo. Консольный вариант программы мы рассматривали в 4.3.4. и 6.4.2.

Вначале приведу программу, написанную одним из моих студентов. Затем мы с вами вместе разберем недостатки программы, а потом попробуем улучшить эту программу.

Форма программы имеет вид, рис. 9.51.

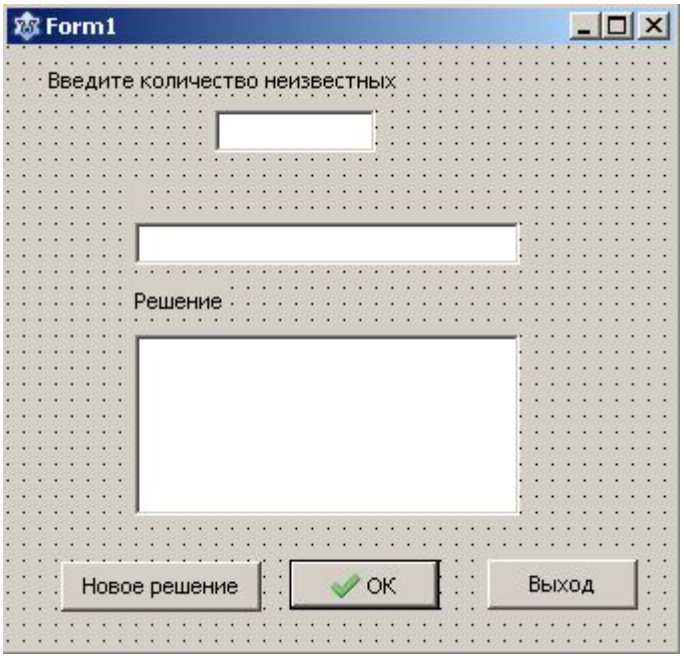


Рис. 9.51. Форма приложения

Код программы следующий:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, LResources, Forms, Controls,
  Graphics, Dialogs, StdCtrls, Buttons;
type

  { TForm1 }

  TForm1 = class(TForm)
    BitBtn1: TBitBtn;
    Button1: TButton;
    Button2: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Memo1: TMemo;
    procedure BitBtn1Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Edit1Change(Sender: TObject);
```

```
    procedure Edit1KeyPress(Sender: TObject;
                            var Key: char);
    procedure Edit2Change(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Calculate;
  private
    { private declarations }
  public
    A, b, x: TStrings;
    v, v2: integer;
    { public declarations }
  end;

var
  Form1: TForm1;

implementation

{ TForm1 }

procedure Tform1.Calculate;
var i,j,n,k,p,h,r:integer;
    t,m,s:extended;
begin
  Memo1.Clear;
  n:=StrToInt(Edit1.Text);
  for h:=0 to n-1 do
  begin
    x.Add('0');
  end;
  for k:=1 to n-1 do
  begin
    for i:=k+1 to n do
    begin
      if a.Strings[(k-1)*n+(k-1)]='0' then
      begin
        {Начало блока перестановки уравнений}

        p:=k;  {В блок схеме используется буква l,
                но она похожа на цифру 1,
                поэтому используем идентификатор p}

        for r:=i to n do
```

```
        if abs(StrToFloat(a.Strings[(r-1)*n+(k-1)]))>
        abs(StrToFloat (a.Strings[(p-1)*n+(k-1)]))
        then p:=r;
        if p<>k then
        begin
          for j:=k to n do;
          begin
            t:=StrToFloat(a.Strings[(k-1)*n+(j-1)]);
            a.Strings[(k-1)*n+(j-1)]:=
            a.Strings[(p-1)*n+(j-1)];
            a.Strings[(p-1)*n+(j-1)]:=FloatToStr(t);
          end;
            t:=(StrToFloat (b.Strings[k-1]));
            b.Strings[k-1]:=b.Strings[p-1];
            b.Strings[p-1]:=FloatToStr (t);
        end;
      end; // Конец блока перестановок уравнений
      m:=StrToFloat (a.Strings[(i-1)*n+
      (k-1)])/StrToFloat (a.Strings[(k-1)*n+(k-1)]);
      a.Strings[(i-1)*n+(k-1)]:='0';
      for j:=k+1 to n do
      begin
        a.Strings[(i-1)*n+(j-1)]:=
        FloatToStr(StrToFloat(a.Strings[(i-1)
        *n+(j-1)])-(m*StrToFloat(a.Strings[(k-1)*
         n+(j-1)])));
      end;
      b.Strings[i-1]:=FloatToStr(StrToFloat
      (b.Strings[i-1])-
      (m*StrToFloat(b.Strings[k-1])));
    end;
  end;
  {Проверка существования решения}
  if StrToFloat (a.Strings[(n-1)*n+(n-1)])<>0 then
  {Решение существует и единственно}
  begin
    x.Strings[n-1]:=(FloatToStr(StrToFloat
    (b.Strings[n-1])/StrToFloat(a.Strings[(n-1)*
     n+(n-1)])));
    for i:=n-1 downto 1 do
    begin
      s:=0;
      for j:=i+1 to n do
```

```
      begin
        s:=s-StrToFloat (a.Strings[(i-1)*n+(j-1)])*
             StrToFloat (x.Strings[j-1]);
      end;
      x.Strings[i-1]:=(FloatToStr((StrToFloat
      (b.Strings[i-1])+s)/
      StrToFloat (a.Strings[(i-1)*n+(i-1)])));
    end;
    Memo1.Lines.Add('Решение:');
    for j:=0 to x.Count-1 do
      Memo1.Lines.Add('X'+IntToStr(j+1)+
                      '='+x.Strings[j]);
  end
  else
  if StrToFloat (b.Strings[n-1])=0 then
     Memo1.Lines.Add('Система уравнений'+
                     '  не имеет решения.')
  else
     Memo1.Lines.Add('Система уравнений'+
           ' имеет бесконечное множество решений.');
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  A:=TStringList.Create;
  b:=TStringList.Create;
  x:=TStringList.Create;
  v:=1;
  v2:=1;
end;

procedure TForm1.Edit2Change(Sender: TObject);
begin
  if (Edit2.Text= '-') or (Edit2.Text= '') then exit;
  StrToFloat (Edit2.Text);
  BitBtn1.Enabled:=true;
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
var n:integer;
begin
  if Edit2.Text<>'' then
  begin
    n:=StrToInt(Edit1.Text);
```

```
      Edit2.SetFocus;
      Edit2.SelectAll;
      if v2=1 then
        Edit1.Enabled:=false;
      if v2<=n+1 then
      begin
        if v<=n+1 then
          v:=v+1
        else
          v:=1;
        if v<=n+1 then
        begin
          if v<=n then
          Label3.Caption:='Введите a' +
                         IntToStr(v2) +
                           IntToStr(v);
          if v>n then
            Label3.Caption:='Введите b'+IntToStr(v2);
          A.Add(Edit2.Text);
        end;
        if v>n+1 then
        begin
          b.Add(Edit2.Text);
          v2:=v2+1;
          v:=1;
          if v2<=n then
            Label3.Caption:='Введите a' +
                           IntToStr(v2)+
                           IntToStr(v)
          else
          begin
            Label3.Caption:='';
            calculate;
            BitBtn1.Enabled:=false;
            Button1.SetFocus;
          end;
        end;
      end;
    end
    else
    Edit2.SetFocus;
  end;
  procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  b.Clear;
  a.Clear;
  x.Clear;
  Memo1.Clear;
  Edit1.Enabled:=true;
  Edit1.Clear;
  Edit2.Clear;
  Edit1.SetFocus;
  v:=1;
  v2:=1;
  Label3.Caption:='';
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
procedure TForm1.Edit1Change(Sender: TObject);
begin
  if strtoint(Edit1.Text)>1 then
  begin
    BitBtn1.Enabled:=true;
    Edit2.Enabled:=true;
    Edit2.Clear;
    Label3.Caption:='Введите a11';
    Edit2.SetFocus;
  end
  else
  begin
    BitBtn1.Enabled:=false;
    Edit2.Enabled:=false;
  end;
end;

procedure TForm1.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
  if Key = #13 then Key := #0;
end;

initialization
  {$I Unit1.lrs}
end.
```

Проанализируем программу с двух точек зрения – с точки зрения пользователя и с точки зрения программиста.

С точки зрения пользователя мы видим, что программа может решать систему уравнений не более чем с девятью неизвестными. Попробуйте ввести двузначное число. У вас не получится!

Во-вторых, хотя ввод коэффициентов и организован более или менее удовлетворительно, к сожалению, у пользователя нет возможности обозреть ранее введенные коэффициенты. При большом числе уравнений здесь немудрено допустить ошибки, перепутать коэффициенты. Хотя подсказка, какой коэффициент надо вводить в данный момент, присутствует.

Обратимся теперь к коду программы. В программе используется класс `TStringList`. Но, вообще говоря, класс `TStringList` не предназначен для вычислений! Он предназначен для обработки строк, т.е. для обработки символьной информации. Посмотрите сколько здесь производится преобразований из символьного представления в числовое и обратно.

В программах такого рода, где в основном производятся вычисления, обычно преобразования выполняются всего два раза! В начале работы программы (при вводе исходных данных) строки символов преобразуются в числа, производятся все необходимые вычисления и в конце работы программы полученные результаты (числовые) преобразуются в их строковое представление для вывода на экран или принтер. При всех прочих равных условиях, эта программа будет работать на порядок (если не больше) медленнее, чем программа, в которой сначала все данные преобразованы в числа.

Теперь посмотрите на код, которым реализован ввод коэффициентов. Там "сам черт ногу сломит"! Используется слишком много условных операторов.

Кроме того, в программе напрочь отсутствует контроль вводимых данных. Хотя стандартная реакция системы поможет избежать аварийного завершения программы, рис. 9.52.

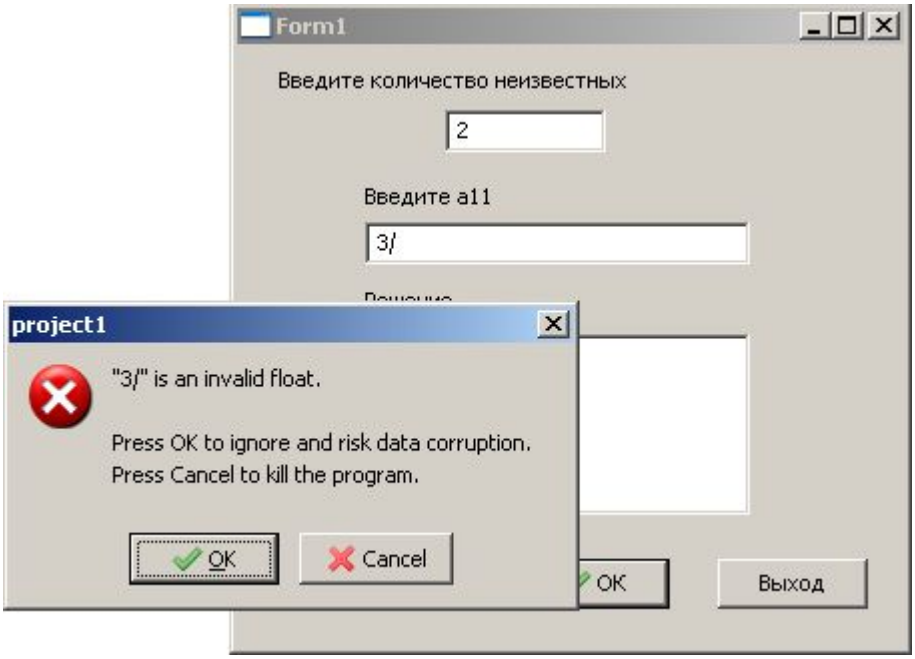


Рис. 9.52. Пример возникновения исключения

Нажав кнопку ОК, пользователь еще может ввести правильное число.

Давайте исправим программу с учетом сделанных выше замечаний. Чтобы пользователь мог видеть все введенные коэффициенты, а также редактировать их удобнее применить компонент `TStringGrid`. Поэтому, сначала познакомимся с этим компонентом.

9.2.10.2. Компонент `TStringGrid`

`TStringGrid` находится во вкладке `Additional` и имеет вид, рис. 9.53.

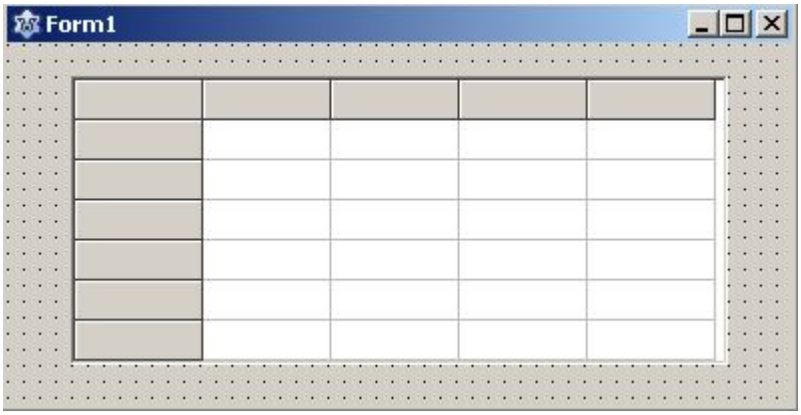


Рис. 9.53. Вид компонента TStringGrid

Компонент представляет собой таблицу, состоящую из строк `Rows` и столбцов `Cols`. В свою очередь таблица это двумерный массив, значениями которого являются строки символов и, следовательно, имеет тип `string`. Доступ к данным осуществляется через свойство `Cells`. Ячейке таблицы, находящейся на пересечении столбца с номером `Col` и строки с номером `Row`, соответствует элемент массива `Cells[Col,Row]`. Обратите внимание, вначале указывается столбец, а затем строка. Нумерация столбцов и строк начинается с нуля. Основные свойства компонента `TStringGrid` следующие:

- `ColCount` – количество столбцов таблицы;
- `RowCount` – количество строк таблицы;
- `FixedCols` – количество фиксированных столбцов таблицы. Обычно фиксируется один, самый левый столбец и используется для задания постоянной информации, например, заголовка столбца. Но можно зафиксировать и больше столбцов. При этом зафиксированные столбцы выделяются цветом и при горизонтальной прокрутке таблицы остаются на месте;
- `FixedRows` – количество фиксированных строк таблицы. Точно так же, фиксируется обычно одна строка для задания заголовка, но можно зафиксировать и больше строк. Строки выделяются цветом и при вертикальной прокрутке таблицы остаются на месте;
- `FixedColor` – цвет фиксированных строк и столбцов.
- `VisibleColCount` – количество видимых (прокручиваемых) столбцов, равно `ColCount` – `FixedCols`;
- `VisibleRowCount` – количество видимых (прокручиваемых) строк, равно `RowCount` – `FixedRows`;
- `ScrollBars` – определяет наличие в таблице полос прокрутки. Если указать значение `ssAutoBoth`, то полосы прокрутки будут появляться и исчезать автоматически в зависимости от того, помещается таблица в окно компонента или нет.

Во вкладке `Options` свойств `TStringGrid` определены ряд свойств, наиболее важными из которых являются:

- `goEditing` – разрешает или запрещает редактирование содержимого ячеек таблицы. `true` – редактирование разрешено, `false` – запрещено;
- `goTab` – разрешает (`true`) или запрещает (`false`) использование клавиши `<Tab>` для перемещения курсора в следующую ячейку таблицы;
- `goAlwaysShowEditor` – признак нахождения компонента в режиме редактирования. Если значение свойства `false`, то для того, чтобы в ячейке появился курсор, надо начать набирать текст, нажать клавишу `<F2>` или сделать щелчок мышью.

Итак, вернемся к реализации метода Гаусса решения системы линейных алгебраических уравнений. Для ввода коэффициентов расширенной матрицы системы воспользуемся компонентом `TStringGrid`. Создайте новый проект и спроектируйте вид приложения так, как показано на рисунке 9.54. Установите следующие свойства компонента `TStringGrid`:

```
ColCount = 1; RowCount = 1; FixedCols = 0; FixedRows = 0;
goEditing = true; goTab = true; goAlwaysShowEditor= true;
```

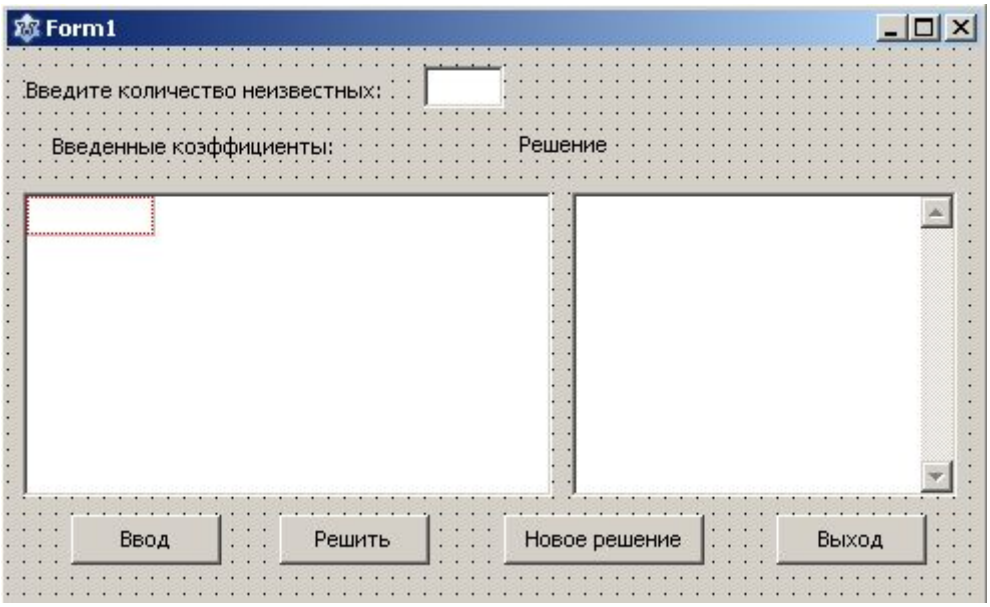


Рис. 9.54. Форма приложения

Код программы:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   LResources,   Forms,   Controls,
  Graphics, Dialogs, StdCtrls, Buttons, Grids, LCLType,
  LCLProc;
type
  { TForm1 }
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label4: TLabel;
    Memo1: TMemo;
    StringGrid1: TStringGrid;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject;
                            var Key: char);
    procedure FormShow(Sender: TObject);
    procedure Gauss(var vector: array of extended;
                    var b: array of extended;
                    var x: array of extended;
                    var n: integer);
    procedure StringGrid1KeyDown(Sender: TObject;
                                 var Key: Word;
      Shift: TShiftState);
  private
    { private declarations }
  public
    { public declarations }
  end;
```

```
var
  Form1: TForm1;
  n: integer;
  a: array of array of extended;{матрица коэффициентов системы,
двумерный динамический массив}
  vector: array of extended; {преобразованный одномерный
 динамический массив }
  b: array of extended;
  x: array of extended;
implementation
{ TForm1 }
procedure TForm1.Gauss(var vector: array of extended;
                       var b: array of extended;
                       var x: array of extended;
                       var n: integer);
var
  a: array of array of extended;{матрица коэффициентов системы,
двумерный динамический массив}
  i, j, k, p, r: integer;
  m, s, t: real;
begin
  try
    SetLength(a, n, n); {установка фактического размера
    массива Преобразование одномерного массива в двумерный}
    k:= 1;
    for i:= 0 to n - 1 do
    for j:= 0 to n - 1 do
    begin
      a[i, j]:= vector[k];
      k:= k + 1;
    end;
    for k:= 0 to n - 2 do
    begin
      for i:= k + 1 to n - 1 do
      begin
        if (a[k, k]= 0) then
        begin
          {перестановка уравнений}
          p:= k; // в алгоритме используется буква l,
                 // но она похожа на 1
                 // Поэтому используем идентификатор p
          for r:= i to n - 1 do
          begin
```

```
          if abs(a[r, k]) > abs(a[p, k]) then p:= r;
        end;
        if p <> k then
        begin
          for j:= k to n - 1 do
          begin
            t:= a[k, j];
            a[k, j]:= a[p, j];
            a[p, j]:= t;
          end;
          t:= b[k];
          b[k]:= b[p];
          b[p]:= t;
        end;
      end; // конец блока перестановки уравнений
      m:= a[i, k] / a[k, k];
      a[i, k]:= 0;
      for j:= k + 1 to n - 1 do
      begin
        a[i, j]:= a[i, j] - m * a[k, j];
      end;
      b[i]:= b[i] - m * b[k];
    end;
  end;
  {Проверка существования решения}
  if a[n - 1, n - 1] <> 0  then
  begin
    x[n - 1]:= b[n - 1] / a[n - 1, n - 1];
    for i:= n - 2 downto 0 do
    begin
      s:= 0;
      for j:= i + 1 to n - 1 do
      begin
        s:= s - a[i, j] * x[j];
      end;
      x[i]:= (b[i] + s) / a[i, i];
    end;
    Memo1.Lines.Add('Решение:');
    for i:= 0 to n - 1 do
      Memo1.Lines.Add('x' + IntToStr(i + 1) +
                      '=' + FloatToStr(x[i]));
    end
    else
```

```
      if b[n - 1] = 0 then
         Memo1.Lines.Add('Система уравнений' +
                          ' не имеет решения.')
      else
         Memo1.Lines.Add('Система уравнений' +
                  ' имеет бесконечное множество решений.');
  except
    on EInvalidOP do
    Memo1.Lines.Add('Неправильные данные. Система уравнений' +
                    ' не имеет решения.');
    on EMathError do
    Memo1.Lines.Add('Неправильные данные. Система уравнений' +
                    '  не имеет решения.');
    on EZeroDivide do
    Memo1.Lines.Add('Неправильные данные. Система уравнений' +
                    '  не имеет решения.');
    on EOverflow do
    Memo1.Lines.Add('Неправильные данные. Система уравнений' +
                    '  не имеет решения.');
    on EUnderflow do
    Memo1.Lines.Add('Неправильные данные. Система уравнений' +
                    '  не имеет решения.');
    on EAccessViolation do
    Memo1.Lines.Add('Неправильные данные. Система уравнений' +
                    '  не имеет решения.');
  end;
    a:= nil;  // освобождение памяти
end;
procedure TForm1.StringGrid1KeyDown(Sender: TObject;
                                    var Key: Word;
                                    Shift: TShiftState);
begin
  if (Key = VK_Return)  then
  begin
    if (StringGrid1.Col >= n)   then
    begin
      StringGrid1.Row:= StringGrid1.Row + 1;
      StringGrid1.Col:= 0;
      Key:= 0;
    end;
   end;
end;
```

```
procedure TForm1.FormShow(Sender: TObject);
begin
  StringGrid1.ColCount:= 1;
  StringGrid1.RowCount:= 1;
  Edit1.SetFocus;
  Button3.Visible:= true;
  Button4.Visible:= false;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.Clear;
  StringGrid1.Clean;
  StringGrid1.ColCount:= 1;
  StringGrid1.RowCount:= 1;
  Edit1.Clear;
  Edit1.SetFocus;
  Button3.Visible:= true;
  Button4.Visible:= false;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  {освобождение памяти,
 распределенной для динамических массивов}
  a:= nil;
  vector:= nil;
  x:= nil;
  b:= nil;
  Close();
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
  Edit1.SetFocus;
  if Length(Edit1.Text) = 0 // если пустая строка
  then exit;
  n:= StrToInt(Edit1.Text);
  StringGrid1.ColCount:= n + 1;
  StringGrid1.RowCount:= n ;
  Stringgrid1.SetFocus;
  Button3.Visible:= false;
  Button4.Visible:= true;
end;
procedure TForm1.Button4Click(Sender: TObject);
begin
```

```
  if (StringGrid1.Col >= n)   then
  begin
    StringGrid1.Row:= StringGrid1.Row + 1;
    StringGrid1.Col:= 0;
  end
  else
    StringGrid1.Col:= StringGrid1.Col+1;
end;
procedure TForm1.Button5Click(Sender: TObject);
var i, j, k, code:integer;
begin
  {Установка реальных размеров динамических массивов}
  SetLength(a, n, n);
  SetLength(vector, n * n);
  SetLength(b, n);
  SetLength(x, n);
  for j:= 0 to n - 1 do
  for i:= 0 to n - 1 do
  begin
    val(StringGrid1.Cells[i, j], a[i, j], code);
    if code <> 0 then
    begin
      ShowMessage('Ошибка при вводе коэффициентов матрицы');
      StringGrid1.SetFocus;
      exit;
    end;
  end;
  for j:= 0 to n - 1 do
  begin
    val(StringGrid1.Cells[n, j], b[j], code);
    if code <> 0 then
    begin
      ShowMessage('Ошибка при вводе свободных членов');
      StringGrid1.SetFocus;
      exit;
    end;
  end;
  code:=0;
  {Преобразование двумерного массива в одномерный}
  k:= 1;
  for j:= 0 to n - 1 do
  for i:= 0 to n - 1 do
  begin
```

```
    vector[k]:= a[i, j];
    k:= k + 1;
  end;
  {Вызов процедуры решения системы линейных
   алгебраических уравнений методом Гаусса}
  Gauss(vector, b, x, n);
end;
procedure TForm1.Edit1KeyPress(Sender: TObject;
                               var Key: char);
begin
if Key = #13 then
  begin
    if Length(Edit1.Text) = 0  // если пустая строка
    then exit;
    n:=StrToInt(Edit1.Text);
    StringGrid1.ColCount:= n + 1;
    StringGrid1.RowCount:= n ;
    Stringgrid1.SetFocus;
    exit;
  end;
  {разрешаем только цифры, знак минус и кл. BackSpace}
  if not (Key in ['0' .. '9', #8])
  then
  begin
    Key:= #0;
    exit;
  end;
end;
initialization
  {$I Unit1.lrs}
end.
```

В программе пользователь может вводить коэффициенты и редактировать их прямо в `TStringGrid`. После нажатия кнопки "Решить" данные из `TStringGrid` преобразуются в числовое представление и записываются в динамические массивы `a` – матрица коэффициентов системы, `b` – вектор свободных членов. Далее осуществляется вызов процедуры `Gauss()`решения системы линейных алгебраических уравнений методом Гаусса, которая практически не отличается от консольного варианта. Если решение системы существует, по-

лученный вектор $x[x_1, x_2, \ldots, x_n]$ преобразуется в строку и выводится на экран.

Для контроля ввода данных в `Edit1` мы использовали метод, примененный нами в 9.2.7.1. Для контроля ввода в `StringGrid1` функцию `Val()`, а при вычислениях применили механизм исключений.

9.2.10.3. Компоненты выбора

В этих компонентах можно организовать выбор каких-то элементов из списка. Весь список содержится в свойстве `Items` и имеет тип `TString`. Элементами списка являются строки. Строки нумеруются, начиная с нуля. Доступ к строке осуществляется с помощью указания индекса элемента в свойстве `Items`, например `Items[k]` или свойства `Items.Strings`.

Компонент `TListBox`

Рассмотрим компонент `TListBox`. Он расположен на странице `Standard`. Основные свойства компонента:

- `MultiSelect` – признак множественного выбора. Если `MultiSelect = true`, то разрешается выбор одновременно нескольких элементов.
- `ExtendedSelect` – если `ExtendedSelect= true` и `MultiSelect= true`, то выбор нескольких элементов можно производить стандартным способом, т.е. при нажатой клавише `Shift` можно выбрать несколько элементов, расположенных подряд, а при нажатой клавише `Ctrl` выбрать элементы в произвольном порядке.
- `Count` – общее количество элементов в компоненте.
- `ItemIndex` – индекс выбранного элемента (при `MultiSelect= false`). Если выбрано несколько элементов (`MultiSelect= true`), то содержит индекс элемента, на котором установлен фокус.
- `Selected[i]` – если выбран элемент с индексом i, то значение этого свой-

ства равно `true`.

- `Sorted` – если равно `true`, то элементы компонента автоматически сортируются.

Рассмотрим несколько примеров для того, чтобы освоить простейшую технику работы с этим компонентом. Поместите на форму два компонента `TListBox` и три кнопки, так как показано на рис. 9.55.

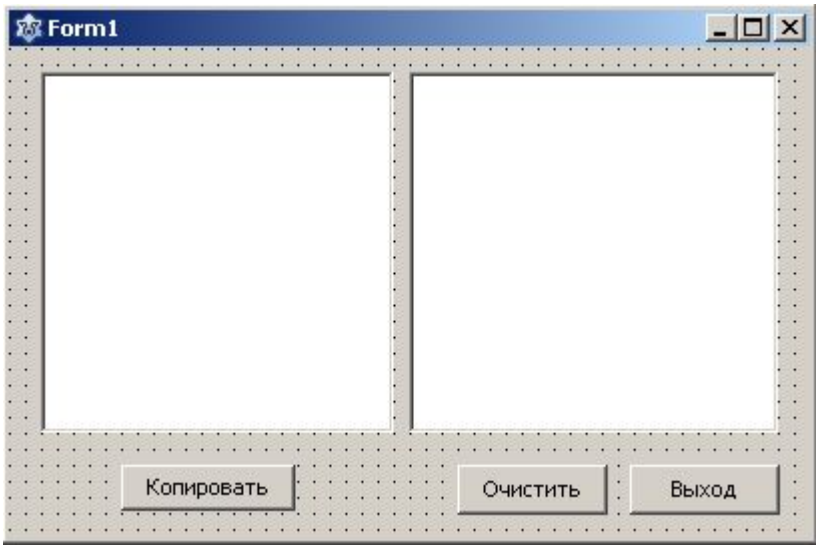


Рис. 9.55. Форма приложения

Заполним программно `ListBox1` содержимым текстового файла, в котором содержатся фамилии, допустим студентов. Затем будем просто помещать выбранные элементы в `ListBox2`. Сначала реализуем выбор одиночного элемента. Вот код этой программы:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,  SysUtils,  FileUtil,  LResources,  Forms,
  Controls, Graphics, Dialogs, StdCtrls;
```

```
type
  { TForm1 }
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    ListBox1: TListBox;
    ListBox2: TListBox;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{ TForm1 }
procedure TForm1.Button2Click(Sender: TObject);
begin
  with ListBox1 do
  begin
    if ItemIndex >= 0 then
      ListBox2.Items.Add(Items[ItemIndex]);
  end;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox2.Clear;
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
  Close;
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile('List.txt');
  str:= tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // преобразование в кодировку UTF-8
  {$ENDIF}
  with ListBox1 do
  begin
    Items.Text:= str;
  end;
  tfile.Free;
 end;
initialization
  {$I unit1.lrs}
end.
```

Если установить в `ListBox2` свойство `Sorted = true`, то элементы

будут выведены в отсортированном виде. При добавлении нового элемента, он будет помещен в нужное место автоматически.

В этой реализации, если при одном и том же выбранном элементе нажать на кнопку "Копировать" несколько раз, то этот элемент попадет в `ListBox2` также несколько раз, т.е. строки в `ListBox2` окажутся не уникальными.

Чтобы элементы в `ListBox2` не повторялись, напишем функцию, которая проверяет не содержится ли уже этот элемент в `ListBox2`. Если такой элемент имеется, функция возвращает `true`, если нет, то `false`.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,   Forms,
  Controls, Graphics, Dialogs, StdCtrls;
type
  { TForm1 }
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    ListBox1: TListBox;
    ListBox2: TListBox;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    function Search(str: string): boolean;
  private
```

```
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{ TForm1 }
function TForm1.Search(str: string): boolean;
var
  i: integer;
begin
  Result:= false;
  for i:= 0 to ListBox2.Count - 1 do
  if ListBox2.Items[i] = str then
  begin
    Result:= true;
    exit;
  end
  else Result:= false;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  with ListBox1 do
  begin
    if (ItemIndex >= 0) and (not Search(GetSelectedText))
    then ListBox2.Items.Add(Items[ItemIndex]);
  end;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox2.Clear;
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
  Close;
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile('List.txt');
  str:= tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // преобразование в кодировку UTF-8
  {$ENDIF}
  with ListBox1 do
  begin
    Items.Text:= str;
  end;
  tfile.Free;
end;
initialization
  {$I unit1.lrs}
end.
```

Реализуем теперь множественный выбор. Обработчик события

`Button2Click` несколько видоизменится, т.к. при множественном выборе для того, чтобы определить выбран ли этот элемент или нет, необходимо использовать свойство `Selected`.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, FileUtil, LResources, Forms,
  Controls, Graphics, Dialogs, StdCtrls;
type
  { TForm1 }
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    ListBox1: TListBox;
    ListBox2: TListBox;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    function Search(str: string): boolean;
  private
    { private declarations }
  public
    { public declarations }
  end;
var
```

```
  Form1: TForm1;
implementation
{ TForm1 }
function TForm1.Search(str: string): boolean;
var
  i: integer;
begin
  Result:= false;
  for i:= 0 to ListBox2.Count - 1 do
  if ListBox2.Items[i] = str then
  begin
    Result:= true;
    exit;
  end
  else Result:= false;
end;
procedure TForm1.Button2Click(Sender: TObject);
var
  i: integer;
begin
  with ListBox1 do
  for i:= 0 to Items.Count - 1 do
  begin
    if (Selected[i]) and (not Search(Items[i]))
    then
      ListBox2.Items.Add(Items[i]);
  end;
end;
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  ListBox2.Clear;
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
  Close;
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile('List.txt');
  str:= tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str);// преобразование в кодировку UTF-8
  {$ENDIF}
  with ListBox1 do
  begin
    Items.Text:= str;
  end;
  tfile.Free;
  ListBox1.MultiSelect:= true;
  ListBox2.Sorted:= true;
end;
initialization
  {$I unit1.lrs}
end.
```

В компоненте можно отображать не только строки, но и изображения, например пиктограммы. Однако мы в этой книге эти вопросы рассматривать не будем.

Компонент `TComboBox`

Этот компонент расположен на странице `Standard`. Внешний вид компонента показан на рис. 9.56.

Рис. 9.56. Компонент TComboBox

При нажатии на кнопку с треугольником появится раскрывающийся (говорят еще выпадающий) список. Методы работы с таким списком практически не отличаются от `TListBox`. По сравнению с `TListBox` у него есть преимущество, заключающееся в том, что можно редактировать элементы списка, а также добавлять в список новые элементы. Кроме того, компонент позволяет сэкономить пространство, когда на форме расположены много компонентов. Собственно для этих целей он и используется. К недостаткам можно отнести то, что в нем нельзя выбрать одновременно несколько элементов. Основные свойства компонента:

- `Items` – содержит элементы списка. Доступ к отдельным элементам возможен по индексу, например `Items[k]` содержит элемент с номером (индексом) `k` (нумерация начинается с нуля).
- `Style` – стиль отображения данных в компоненте, имеет следующие значения:

1. `csDropDown` – раскрывающийся список с окном редактирования, позволяющим пользователю редактировать или добавлять новые строки в список;
2. `csDropDownList` – раскрывающийся список. В этом режиме разрешено выбирать только существующие элементы списка. Редактировать или добавлять строки в этом режиме нельзя;
3. `csSimple` – список всегда раскрыт, по существу совпадает с `ListBox`, но имеется возможность редактирования и добавления новых строк;
4. `csOwnerDrawFixed` – раскрывающийся список со строками одинаковой высоты, в которых могут отображаться изображения и текст;
5. `csOwnerDrawVariable` – раскрывающийся список со строками разной высоты, в которых могут отображаться изображения и текст.

- `Text` – содержит выбранный элемент списка или вновь введенную строку;
- `ItemIndex` – содержит индекс выбранного элемента. По умолчанию в момент проектирования `ItemIndex = -1`. Если вы запустите приложение с таким значением `ItemIndex`, то в окне `ComboBox` ничего не будет отображено. Особенно если вы используете свойство `Style = csSimple`, да к тому же, забудете увеличить высоту `Height` компонента. В этом случае пользователю будет выведено маленькое пустое окошко без кнопки раскрытия! Поэтому желательно либо во время проектирования, либо программно, например, при создании формы (`OnCreate`) устанавливать значение `ItemIndex`. Чаще всего устанавливают `ItemIndex = 0`, в этом случае в окно `ComboBox` будет выведен первый элемент списка. Но можно присвоить `ItemIndex` и другое значение (естественно внутри допустимого диапазона индексов элементов списка). Например, того элемента, который должен быть выбран по умолчанию. Во время выполнения приложения значение `ItemIndex` может принимать значение -1. Это происходит в том случае, если в окне компонента производилось редактирование текущего элемента, т.е. по значению `ItemIndex = -1` можно узнать, что редактирование проводилось и предпринять соответствую-

щие действия, см. пример ниже;

- `DropDownCount` – задаёт количество элементов списка, выводимых без полосы прокрутки.
- `DroppedDown` – указывает состояние компонента.

  При `DroppedDown= true`, список раскрыт.

  Рассмотрим несколько примеров.

  Создайте новый проект, поместите на форму компоненты `TComboBox`, `TLabel` и `TButton`, как показано на рис. 9.57.

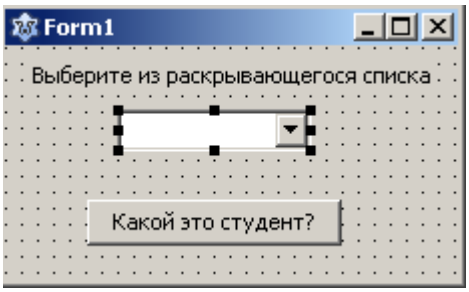


Рис. 9.57. Форма программы

Установите свойство `Style` компонента `ComboBox1` равным `csDropDownList`. В свойстве `Items` введите любые три фамилии, например, как на рисунке 9.58. Пусть это будут фамилии некоторых студентов.

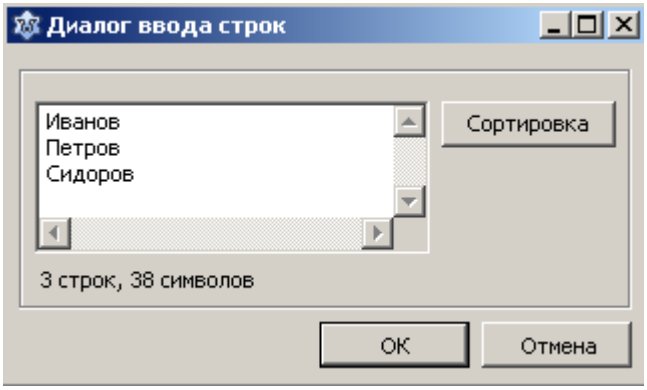


Рис. 9.58. Диалог ввода строк в компонент

В обработчик события `OnClick` введите следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with ComboBox1 do
  case ItemIndex of
  0: ShowMessage('Это отличник');
  1: ShowMessage('Это двоечник');
  2: ShowMessage('А этот середняк');
  else
    ShowMessage('Никто не выбран');
  end;
end;
```

При запуске программы в окне `ComboBox1` не будет отображена фамилия студента. Пользователь должен раскрыть список. Как мы уже говорили, это не совсем удобно. Сделаем так, чтобы при запуске программы отображалась первая фамилия. Кроме того, пусть сообщение выводится сразу после выбора нужного элемента в списке `ComboBox1`, т.е. обойдемся без кнопки. Для этого используем событие `OnSelect`.

И еще, во многих случаях в окно `ComboBox1` выводится значение, которое должно быть принято как значение по умолчанию. Если пользователя это значение устраивает, то, обычно, ему достаточно просто нажать клавишу `Enter`. Реализуем такую функциональность нашего приложения. Итак, код программы (не забудьте удалить из формы кнопку):

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,   Forms,
```

```
  Controls, Graphics, Dialogs, StdCtrls, LCLType;
type
  { TForm1 }
  TForm1 = class(TForm)
    ComboBox1: TComboBox;
    Label1: TLabel;
    procedure ComboBox1KeyPress(Sender: TObject;
                                var Key: char);
    procedure ComboBox1Select(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{ TForm1 }
procedure TForm1.ComboBox1Select(Sender: TObject);
begin
  with ComboBox1 do
  case ItemIndex of
  0: ShowMessage('Это отличник');
  1: ShowMessage('Это двоечник');
  2: ShowMessage('А этот середняк');
  else
    ShowMessage('Никто не выбран');
  end;
```

```
end;
procedure TForm1.ComboBox1KeyPress(Sender: TObject;
                                   var Key: char);
begin
  if Key = #13 then ComboBox1Select(Sender);
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  ComboBox1.ItemIndex:= 0;
end;
initialization
  {$I unit1.lrs}
end.
```

Сделаем более полезную программу. Предположим, в некоторых текстовых файлах содержатся списки нескольких групп студентов. Пусть для определенности групп будет три. Необходимо, в зависимости от выбора пользователя, выводить список студентов выбранной группы. Список группы будем выводить в `TListBox`. В новом проекте сформируйте вид окна вашего приложения, рис. 9.59.

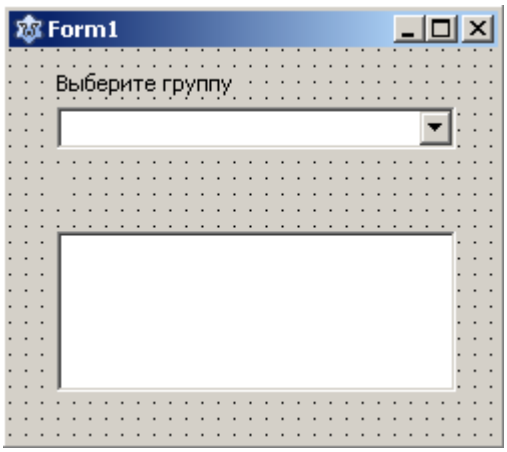


Рис. 9.59. Форма приложения

В свойстве `Items` введите любые три названия групп, например, как на рисунке 9.60.

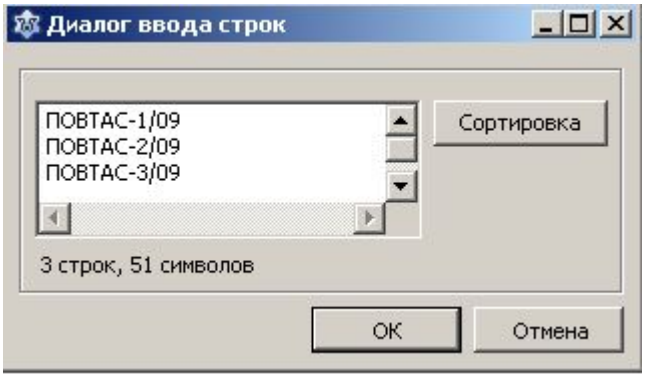


Рис. 9.60. Диалог ввода строк в компонент

Создайте три текстовых файла с именами `List1.txt`, `List2.txt` и `List3.txt` с фамилиями студентов. Заполнять `TListBox` содержимым файла мы уже умеем, для простоты предположим, что файлы находятся в той же папке, что и наше приложение. Код программы:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,  SysUtils,  FileUtil,  LResources,  Forms,
  Controls, Graphics, Dialogs, StdCtrls, LCLType;
type
  { TForm1 }
  TForm1 = class(TForm)
    ComboBox1: TComboBox;
    Label1: TLabel;
    Label2: TLabel;
    ListBox1: TListBox;
    procedure ComboBox1KeyPress(Sender: TObject; var Key:
```

```
char);
    procedure ComboBox1Select(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure LoadListGroup(namefile: string);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{ TForm1 }

procedure TForm1.LoadListGroup(namefile: string);
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile(namefile);
  str:= tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // преобразование в кодировку UTF-8
  {$ENDIF}
  with ListBox1 do
  begin
    Items.Text:= str;
  end;
```

```
  tfile.Free;
  ListBox1.Sorted:= true;
end;
procedure TForm1.ComboBox1Select(Sender: TObject);
begin
  with ComboBox1 do
  begin
    Label2.Caption:='Список группы ' + Text;
    case ItemIndex of
    0: LoadListGroup('List1.txt');
    1: LoadListGroup('List2.txt');
    2: LoadListGroup('List3.txt');
    else
      ShowMessage('Группа не выбрана');
    end;
  end;
end;

procedure TForm1.ComboBox1KeyPress(Sender: TObject;
                                   var Key: char);
begin
  if Key = #13 then
  begin
    ComboBox1Select(Sender);
    Key:=#0;
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
  ComboBox1.ItemIndex:= 0;
  ComboBox1.Style:= csDropDownList;
end;
initialization
  {$I unit1.lrs}
end.
```

Загрузку в TListBox содержимого текстового файла мы оформили в виде процедуры, в которую в качестве параметра передается имя файла, соответствующее выбранной группе.

В заключение рассмотрим пример, как в TComboBox реализуются корректировка и добавление элементов в список. Создайте форму, рис. 9.61.

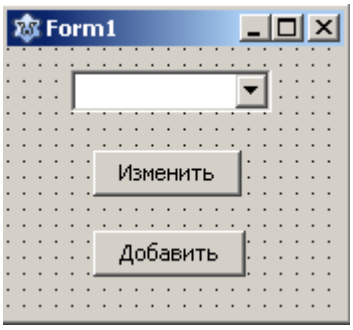


Рис. 9.61. Форма приложения

Теперь уже установите свойство Style компонента ComboBox1 равным csDropDown. В свойстве Items введите любые данные. В обработчике OnSelect в переменной oldItemIndex мы запоминаем индекс выбранного элемента. Если нажата кнопка "Изменить", то оператор

```
ComboBox1.Items[oldItemIndex]:= ComboBox1.Text;
```

изменит выбранный элемент новым содержимым. Если пользователь нажмет

кнопку "Добавить", то оператор

```
ComboBox1.Items.Add(ComboBox1.Text);
```

добавит новую запись в список.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,   Forms,
  Controls, Graphics, Dialogs, StdCtrls;
type
  { TForm1 }
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    ComboBox1: TComboBox;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure ComboBox1Select(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
```

```
  oldItemIndex: integer;

implementation
{ TForm1 }
procedure TForm1.ComboBox1Select(Sender: TObject);
begin
  oldItemIndex:= ComboBox1.ItemIndex;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  ComboBox1.ItemIndex:= 0;
  ComboBox1.Style:= csDropDown;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if ComboBox1.ItemIndex = - 1 then
  ComboBox1.Items[oldItemIndex]:= ComboBox1.Text;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  if ComboBox1.ItemIndex = - 1 then
  ComboBox1.Items.Add(ComboBox1.Text);
end;
initialization
  {$I unit1.lrs}
end.
```

Компоненты выбора – переключатели

Эта группа компонентов реализуют такие элементы графического интерфейса, как флажки и переключатели. С помощью таких элементов можно осуществлять выбор и установку каких-либо опций, режимов, состояний и т.д. Таким образом, в этих компонентах, в отличие от таких компонентов как `TListBox` и `TComboBox`, производится выбор не данных как таковых, а каких-то свойств и характеристик объектов, используемых в приложении. Выбор пользователя в этих компонентах подразумевает ответ пользователя типа "Да", "Нет". Рассмотрим компонент `TCheckBox`.

Этот компонент реализует элемент графического интерфейса флажок. Применяется также термин индикатор. Многие программисты говорят также просто "чек бокс". Обычно в диалоговом окне присутствуют несколько флажков. Состояние каждого флажка не зависит от других флажков, поэтому часто употребляется термин группа независимых переключателей, рис. 9.62.

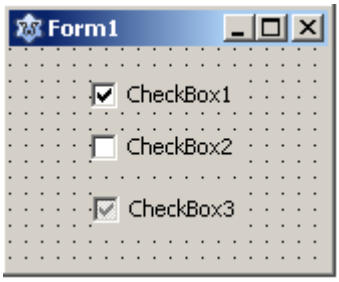


Рис. 9.62. Вид компонента TCheckBox

Основные свойства компонента:

- `State` – компонент `TCheckBox` может находиться в общем случае в трех состояниях:
    1. Флажок выбран, в окошке компонента стоит галочка. Этому состоянию соответствует значение `State=cbChecked`;
    2. Флажок не выбран, в окошке компонента галочка отсутствует. Этому

состоянию соответствует значение `State=cbUnchecked`;

3. Флажок выбран, в окошке компонента стоит галочка. Но само окошко и галочка серого цвета. Этому состоянию соответствует значение `State=cbGrayed`; Это, так называемое, промежуточное состояние. Чаще всего применяется в тех случаях, когда тот или иной режим, та или иная опция должна быть обязательно выбрана (задействована), но так, чтобы пользователь был проинформирован об этом. Очень часто это состояние используется в комбинации со свойством `Enabled=false`, т.е. данная опция автоматически установлена, пользователь об этом знает, но менять эту опцию нельзя.

- `AllowGrayed` – разрешает или не разрешает применение состояния `State=cbGrayed`. Если `AllowGrayed= false` (значение по умолчанию), то возможны только два состояния, флажок выбран и флажок не выбран;
- `Checked` – если `Checked` = `true`, то флажок выбран. Значение `State= cbChecked`. Если же `Checked = false`, то `State` равен либо `cbUnchecked`, либо `cbGrayed`;
- `Caption` – задает текст, связанный с флажком;

Для того чтобы определить выбран флажок или нет достаточно записать оператор

```
if CheckBox1.Checked then <действия, если флажок выбран>
else <действия, если флажок не выбран>;
```

или

```
case CheckBox1.State of
  cbChecked: < соответствующие действия >;
  cbUnchecked: < соответствующие действия >;
```

```
  cbGrayed: < соответствующие действия >;
end;
```

Рассмотрим компонент, реализующий переключатель `TRadioButton`, рис. 9.63. Говорят также радиокнопка или зависимый переключатель.

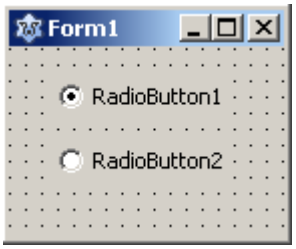


Рис. 9.63. Вид компонента TRadioButton

Следует заметить, что использование одного переключателя не имеет смысла, поскольку механизм переключателей предназначен для организации выбора пользователем только одного из нескольких возможных режимов (опций, свойств и т.д.). Поэтому необходимо использование как минимум двух компонентов `TRadioButton`. Свойства компонента:

- `Checked` – определяет включен или не включен переключатель. Если переключатель включен, то он отмечается точкой внутри кружочка, а `Checked= true`;
- `Caption` – задает текст надписи рядом с кнопкой.

Обычно компоненты `TRadioButton` размещаются в специальном компоненте-контейнере (об этом чуть позже), но можно положить их и на форму. Как только вы разместите несколько компонентов `TRadioButton` на форму, они начинают функционировать как единая группа переключателей. Т.е. даже на этапе проектирования вы не сможете установить `Checked= true` одновременно нескольким радиокнопкам, а можете установить это свойство равным `true` только у одного переключателя!

Для удобной работы с группами компонентов `TCheckBox` и `TRadioButton` существуют специальные контейнеры `TCheckGroup` и `TRadioGroup`. Контейнер это такой компонент, на котором можно размещать другие компоненты. Иначе их еще называют панелями, рис. 9.64.

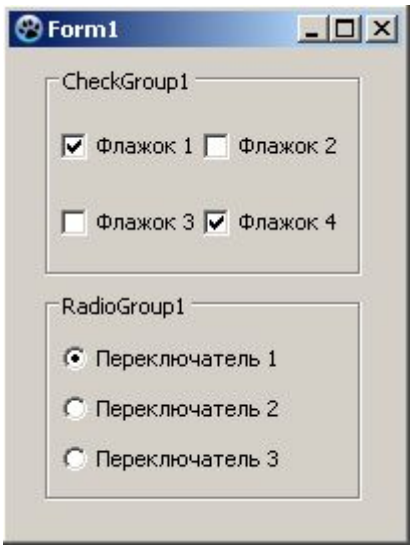


Рис. 9.64. Компоненты-контейнеры TCheckGroup и TRadioGroup

Общие свойства контейнеров `TCheckGroup` и `TRadioGroup`:

- `Caption` – текст заголовка панели;
- `Columns` – задает количество столбцов. На рис. 6.70. панель `CheckGroup1` содержит два столбца, а панель `RadioGroup1` состоит из одного столбца;
- `ColumnLayout` – задает способ размещения дочерних компонентов в столбцах. Имеет значения: `clHorizontalThenVertical` – компоненты размещаются по горизонтали, `clVerticalThenHorizontal` – компоненты размещаются по вертикали. На рисунке флажки размещены по горизонтали;
- `Items` – содержит список строк типа `TStrings` с текстами заголовков соответствующих компонентов. Строки можно формировать как при проектировании, так и программно. При проектировании необходимо раскрыть редактор строк `Items` и ввести необходимые строки. Сколько было введено строк, столько и будет создано переключателей или флажков. Доступ к тексту флажка

или переключателя осуществляется по индексу `Items[k]`, индексы как обычно начинаются с нуля.

Определение того, какой элемент пользователь выбрал в компонентах `TCheckGroup` и `TRadioGroup` различаются. Так в `TCheckGroup` используется свойство `Checked[k]`, где `k` индекс флажка. Например:

```
if CheckGroup1.Checked[k]
then ShowMessage('Выбран флажок ' + IntToStr(k + 1));
```

В `TRadioGroup` для этих же целей можно использовать свойство `ItemIndex`, которое указывает индекс выбранного переключателя, например:

```
with RadioGroup1 do
begin
  k:= ItemIndex + 1;
  case ItemIndex of
  0: ShowMessage('Выбран переключатель ' + IntToStr(k));
  1: ShowMessage('Выбран переключатель ' + IntToStr(k));
  2: ShowMessage('Выбран переключатель ' + IntToStr(k));
  else
    ShowMessage('Не выбран ни один переключатель');
  end;
end;
```

По умолчанию `ItemIndex= -1`. Это означает, что ни один переключатель не выбран. Чтобы программно или при проектировании указать выбранный переключатель достаточно `ItemIndex` присвоить значение из допустимого диапазона.

В рассмотренных нами контейнерах TCheckGroup, TRadioGroup расположение переключателей и флажков формируется автоматически самим контейнером. Иногда бывает удобнее применить другой контейнер, а именно TGroupBox, где можно более рационально использовать площадь контейнера. Кроме того, в TGroupBox можно размещать любые компоненты. Разница от рассмотренных ранее специальных контейнеров TCheckGroup, TRadioGroup заключается в том, что необходимо использовать компоненты TCheckBox и TRadioButton.

Внешний вид рассмотренных компонентов приведен на рис. 9.65.

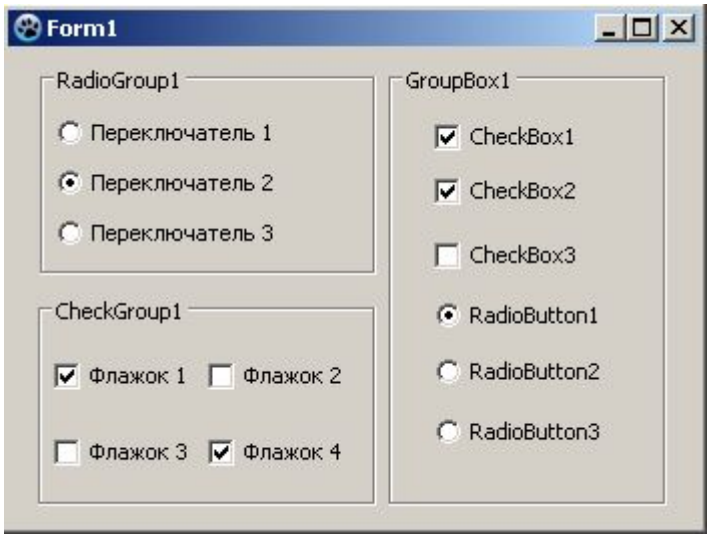


Рис. 9.65. Компонент-контейнер TGroupBox

9.2.10.4. Компоненты отображения структурированных данных

Компонент TTreeView

Этот компонент (вкладка Common Controls) применяется для отображения данных, имеющих иерархическую структуру или, иначе говоря, данных,

которые можно представить в виде нескольких уровней. К таким данным, например, относятся структура какого-нибудь предприятия, генеалогическое дерево человека, файловая структура диска и т.д.

В компоненте `TTreeView` информация отображается в виде связанных узлов. Каждый узел состоит из некоторого текста, собственно составляющих данные. Может содержать также некоторую пиктограмму и, кроме того, с каждым узлом может быть связан некоторый объект. Узел может иметь подузлы, т.е. узлы, лежащие на следующем, низшем уровне. Также любой узел, кроме узла самого верхнего уровня, может входить в другой узел в качестве подузла. Информация об узлах дерева содержится в свойстве `Items`. Информацию в компонент можно заносить как вручную, так и программно.

Для ручного заполнения существует специальный редактор элементов `TReeView`, открыть который можно выбрав пункт `Items` в инспекторе объектов и нажав кнопку с многоточием или просто дважды щелкнув по самому компоненту на форме.

Для ввода нового элемента нажмите на кнопку "Новый элемент". Чтобы ввести для вновь введенного узла его подузлы, нажмите на кнопку "Новый подэлемент". Чтобы ввести узел, расположенный на одном уровне с каким-то узлом, его необходимо предварительно выделить. После этого нажмите на кнопку "Новый элемент", рис. 9.66.

В поле "Текст" введите содержательную информацию для этого узла.

Кнопка "Загрузить" позволяет загрузить данные из текстового файла. В файле должны содержаться тексты узлов. Нижележащие уровни (подузлы) отмечаются отступом. Так, только что введенную информацию, мы могли бы загрузить из файла, записи которого имели бы вид:

Ректор

  Проректор по учебной работе

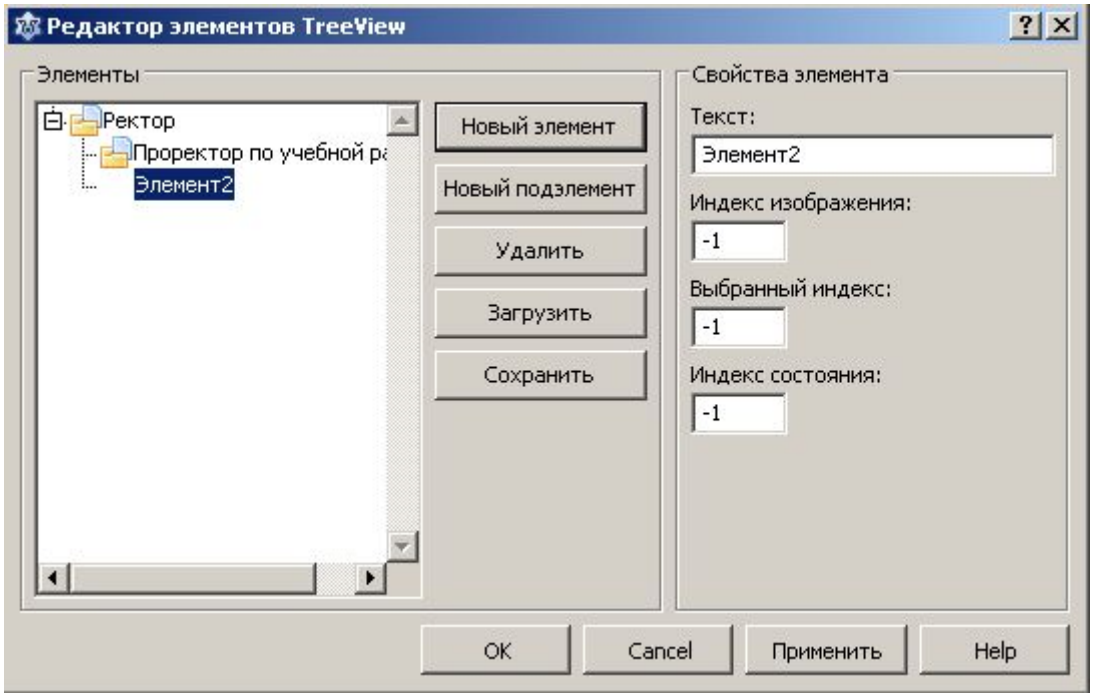


Рис. 9.66. Редактор элементов TTreeView

Теперь об индексах. Как видно из рисунка, рядом с текстом узла находится рисунок (пиктограмма). Рисунки размещаются в специальном компоненте `TImageList`, расположенном в той вкладке `Common Controls`. Поместите компонент `TImageList` в любом месте формы, это невидимый компонент (естественно при выполнении). В нем также имеется специальный редактор для занесения изображений. Чтобы открыть его проще всего дважды щелкнуть по самому компоненту, рис. 9.67.

Нажмите кнопку "Добавить…". Откроется стандартное диалоговое окно открытия файла изображения. Я выбрал два понравившихся мне рисунка из папки, где установлен Lazarus. Каждое изображение имеет свой индекс, по которому осуществляется доступ к нему. Индексы начинаются с нуля. Основные свойства компонента `TImageList` – высота `Height` и ширина `Width`. Эти свойства позволяют задать размер изображения. По умолчанию размер изображения 16x16 пикселей.

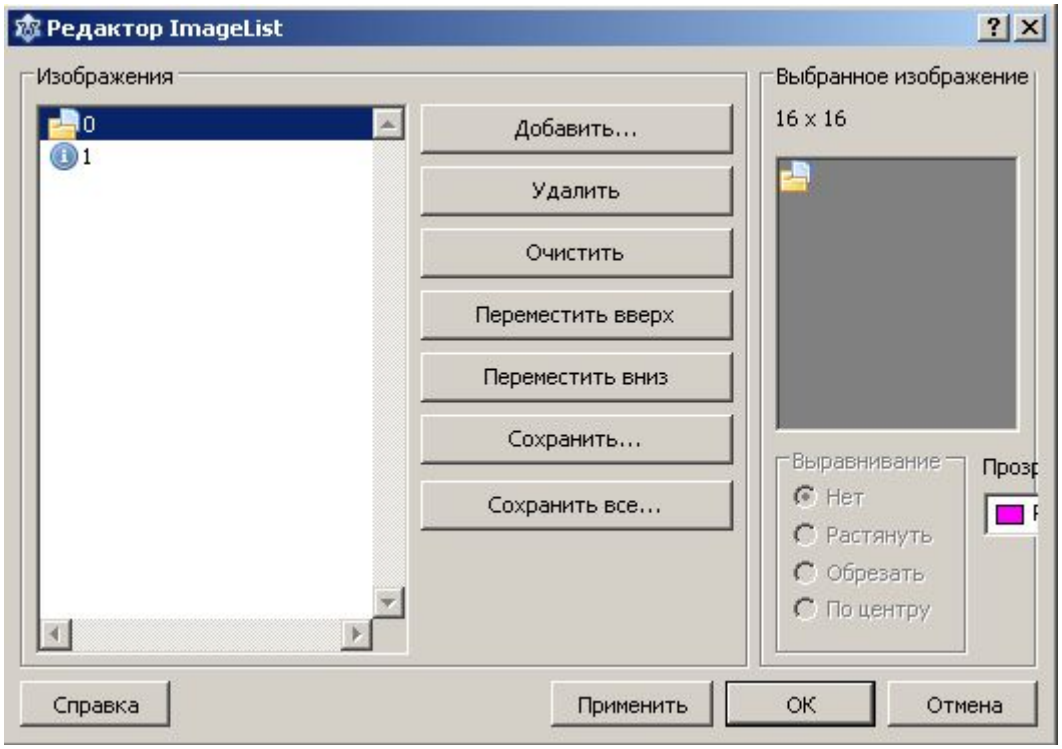


Рис. 9.67. Редактор компонента TImageList

Чтобы связать компонент `TImageList` с `TTreeView` выделите компонент `TTreeView`, выберите в инспекторе объектов свойство `Images` и в выпадающем списке укажите нужный компонент, например `ImageList1`. Или программно запишите оператор

```
TreeView1.Images:= ImageList1;
```

Свойство `ImageIndex` позволяет указать индекс изображения в `TImageList`, которое будет появляться рядом с текущим узлом. Свойство `SelectedIndex` позволяет указать индекс изображения в `TImageList`, которое будет появляться рядом с текущим узлом при его выборе (выделении). И, наконец, свойство `StateIndex` позволяет добавить дополнительную пиктограмму, которое будет появляться рядом с текущим узлом. Изображения, соответствующие `StateIndex` хранятся в другом компоненте `TImageList`, а связь осуществляется через свойство `StateImages`. Так что,

если вы собираетесь использовать дополнительные пиктограммы, вы должны поместить на форму второй компонент `TImageList` и заполнить его соответствующими изображениями. По умолчанию значения всех индексов равны -1, рис. 9.66. Это означает, что пиктограммы отсутствуют.

Рассмотрим некоторые другие свойства компонента `TTreeView`:

- `Images: TImageList` – содержит набор изображений, которые будут использоваться при прорисовке узлов;
- `Indent: integer` – определяет отступ в пикселях от левого угла узла для всех его подузлов;
- `Items: TTreeNodes` – содержит список всех узлов. Доступ к любому узлу осуществляется по индексу. Индексация начинается с нуля;
- `ReadOnly: boolean` – запрещает/разрешает редактирование текста надписей узлов;
- `RightClickSelect: boolean` – разрешает выбор узлов правой кнопкой мыши;
- `Selected: TTreeNode` – содержит список всех выбранных узлов. Если ничего не выбрано, то равно `nil`;
- `SelectionCount: Cardinal` – количество выбранных узлов;
- `Selections[Index: integer]: TTreeNode` – доступ к выбранным узлам по индексу;
- `RowSelect: boolean` – разрешает выделение цветом линий выбранных узлов. Игнорируется, если `ShowLines` равен `false`;
- `ShowButtons: boolean` – разрешает/запрещает показ стандартных кнопок раскрытия подузлов. По умолчанию `true`. Если `false`, узел раскрывается двойным щелчком мыши;
- `ExpandSignType` – определяет вид значка раскрытия узла. Может принимать значения `tvestPlusMinus` - значок в виде плюса/минуса,

`tvestArrow` – значок в виде стрелки, `tvestTheme` – вид значка определяется системными настройками.

- `ShowLines: boolean` – разрешает/запрещает показ линий;
- `ShowRoot: boolean` – разрешает/запрещает показ линий, идущих от самого верхнего уровня иерархии. Игнорируется, если `ShowLines = false`;
- `SortType: TSortType` – указывает способ сортировки узлов: `stNone` – нет сортировки, `stData` – сортировка по связанным с узлами данными объектов, `stText` – сортировка по тексту надписей, `stBoth` – сортировка по тексту и по данным;
- `Topitem: TTreeNode` – определяет корневой узел.

Методы компонента:

- `procedure FullCollapse` – сворачивает все узлы, кроме узлов самого верхнего уровня;
- `procedure FullExpand` – раскрывает все узлы;
- `function IsEditing: boolean` – возвращает `true`, если пользователь редактирует какой-либо узел;
- `procedure LoadFromFile(const FileName: string)` – загружает данные в компонент из файла;
- `procedure SaveToFile(const FileName: string)` – сохраняет в файле содержимое компонента;

Рассмотрим события, связанные с `TTreeView`:

- `OnChange` – возникает при изменении состояния выбора узла, например при выделении одного из узлов или при отмене выделения;
- `OnExpanding` – возникает перед раскрытием узла. В обработчик события `OnExpanding` передается параметр `var AllowExpansion: boolean`, который можно задать равным `false`, если вы хотите запретить раскрытие этого узла;

- OnExpanded – возникает после раскрытия узла;
- OnCollapsing – возникает перед сворачиванием узла. В обработчик OnCollapsing передается параметр var AllowCollapse: boolean, разрешающий или запрещающий сворачивание узла;
- OnCollapsed – возникает после сворачивания узла;
- OnCompare – возникает при сравнении двух узлов Node1 и Node2. В параметре Compare обработчик должен вернуть отрицательное число, если Node1<Node2, ноль, если Node1=Node2 и положительное число, если Node1>Node2;
- OnDeletion – возникает при удалении узла;
- OnEdited –возникает при завершении редактирования текста надписи в узле. В обработчик передаются узел Node и S – новая надпись.

Для работы с узлами Items предусмотрен специальный класс TTreeNodes. Рассмотрим некоторые свойства и методы этого класса.

- Count – количество узлов в Items;
- Item[Index: integer] – узел с индексом Index;

Методы класса TTreeNodes:

- function Add(Node: TTreeNode; const S: String): TTreeNode – добавляет узел в конец того списка, в котором зарегистрирован узел Node. Если Node = nil, добавляется корневой узел для всего компонента;
- function AddChild(Node: TTreeNode; const S: string): TTreeNode – добавляет узел в конец списка item дочерних узлов узла Node;
- function AddChildFirst(Node: TTreeNode; const S: String): TTreeNode – добавляет узел в начало списка Item дочерних узлов узла Node;
- function AddFirst(Node: TTreeNode; const S: String): TTreeNode – добавляет

узел в начало того списка, в котором зарегистрирован узел `Node`;

- `procedure BeginUpdate` – блокирует обновление экрана до тех пор, пока не будет выполнен метод `EndUpdate`. Используется при большом количестве операций по удалению и вставке узлов, поскольку при этом может возникнуть мерцание экрана;
- `procedure Clear` – очищает список всех узлов и подузлов компонента;
- `procedure Delete(Node: TTreeNode)` – удаляет узел `Node`;
- `procedure EndUpdate` – отменяет блокировку обновления экрана методом `BeginUpdate`;
- `function GetFirstNode: TTreeNode` – возвращает самый первый узел в списке `Items[0]`;
- `function GetNode(Itemid: HTreeItem): TTreeNode` – возвращает узел по его идентификатору `Itemid`;
- `function Insert(Node: TTreeNode;`
  `const S: string): TTreeNode` – вставляет узел непосредственно перед узлом `Node`;

Для работы непосредственно с узлом в классе `TTreeNode` имеет свой набор методов, свойств и событий:

- `AbsoluteIndex: integer` – возвращает абсолютный индекс узла (с учетом всех подузлов);
- `Count: integer` – содержит количество подузлов в списке `Item`;
- `Cut: boolean` – помечает узел статусом "вырезанный";
- `Expanded: boolean` – равен `true`, если узел раскрыт;
- `Focused: boolean` – равен `true`, если на узле установлен фокус;
- `HasChildren: boolean` – `true`, если узел имеет дочерние узлы;
- `ImageIndex: TImageIndex` – равен индексу связанной с узлом пиктограммы;

- `Index: Longint` – содержит индекс узла в списке дочерних узлов его родительского узла;
- `IsVisible: boolean` – равен `true`, если узел виден;
- `Level: integer` – содержит иерархический уровень узла;
- `Owner: TTreeNodes` – содержит ссылку на владельца данного узла;
- `Parent: TTreeNode` – содержит ссылку на родительский узел;
- `Selected: boolean` – равен `true`, если узел выделен;
- `SelectedIndex: integer` – номер пиктограммы для выделенного узла;
- `Text: Strings` – содержит текст узла;
- `function AlphaSort: boolean -` сортирует узлы по алфавиту свойств `Text` и возвращает `true` в случае успеха;
- `procedure Collapse(Recurse: boolean)` – закрывает все узлы (`Recurse` = `true`) или только раскрытые (`Recurse` = `false`);
- `procedure Delete -` удаляет текущий узел;
- `procedure DeleteChildren` – удаляет дочерние узлы;
- `function EditText: boolean` – переводит текст узла в режим редактирования;
- `procedure EndEdit(Cancel: boolean)` – заканчивает редактирование текста и сохраняет изменения, если `Cancel= false`;
- `procedure Expand(Recurse: boolean)` – открывает узел (и все подузлы, если `Recurse= true`);
- `function GetFirstChild: TTreeNode` – возвращает ссылку на первый подузел или `nil`, если нет подузлов;
- `function GetLastChild: TTreeNode -` возвращает ссылку на последний подузел или `nil`, если нет подузлов;
- `function GetNext: TTreeNode` – возвращает ссылку на очередной подузел;

- `function HasAsParent(Value: TTreeNode): boolean` – возвращает `true`, если `Value` – родительский узел;
- `function IndexOf(Value: TTreeNode): integer` – возвращает индекс узла `Value`.

Давайте теперь завершим процесс ввода в `TReeView1` структуру типичного университета, рис. 9.68.

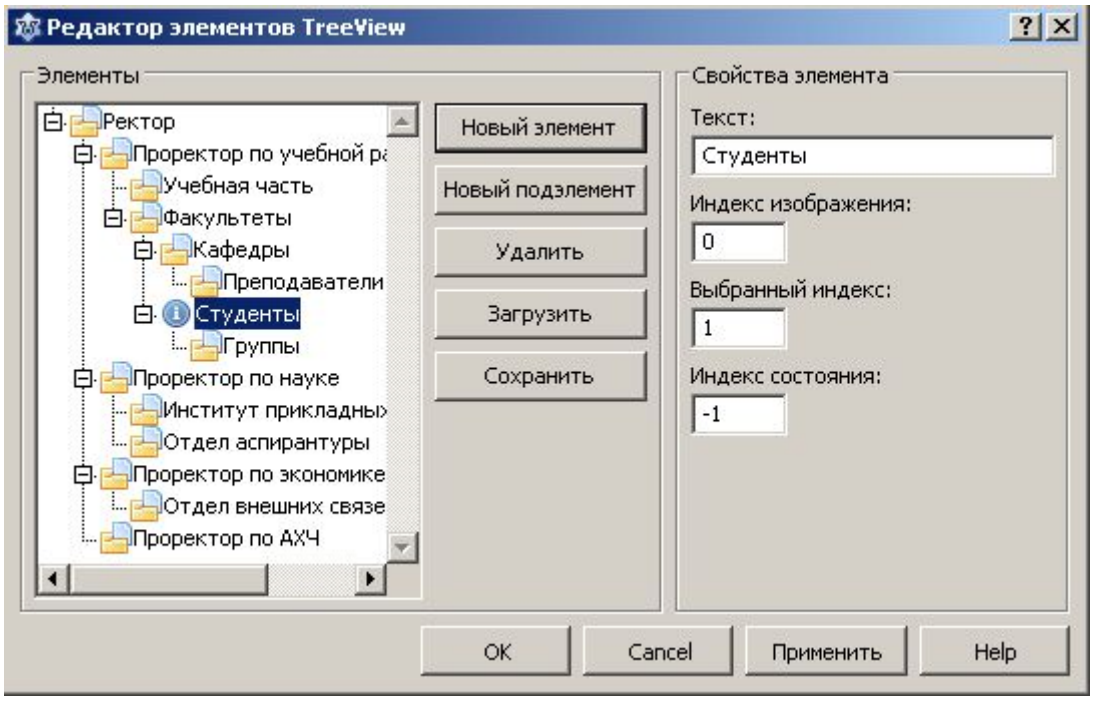


Рис. 9.68. Пример ввода элементов с помощью редактора TTreeView

Поместите на форму компонент `TStatusBar` и напишите следующие обработчики:

```
procedure TForm1.TreeView1Change(Sender: TObject;
                                 Node: TTreeNode);
begin
  if Node.Selected then
    StatusBar1.SimpleText:= 'Выбран узел: ' + Node.Text;
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  TreeView1.Images:= ImageList1;
  TreeView1.ExpandSignType:= tvestPlusMinus;
end;
```

В обработчике OnCreate формы мы связали изображения, содержащиеся в ImageList1 с TreeView1, а также указали вид кнопки раскрытия узла. В обработчик OnChange передается узел Node, состояние которого изменилось (например, при щелчке пользователя на узел, он будет выделен). Узнать, выделен (выбран) узел или нет, можно по свойству Selected. Текст этого узла содержится в свойстве Text.

Теперь создадим это же дерево программным путем. Кроме вывода в TStatusBar информации о выбранном пользователем узле, выполним какие-то еще действия. Например, выведем список групп студентов при нажатии узла "Группы". Создайте новый проект, поместите на форму компоненты TTreeView, TComboBox, TListBox, TStatusBar, TImageList и три компонента TLabel, так как показано на рисунке 9.69.

Поначалу компоненты TComboBox, TListBox и два TLabel будут невидимы. Только при выборе узла " Группы " мы сделаем их видимыми, а при выборе другого узла, снова сделаем их невидимыми.

Подготовьте три текстовых файла со списками студентов или возьмите файлы из примера, где мы рассматривали компонент TComboBox.

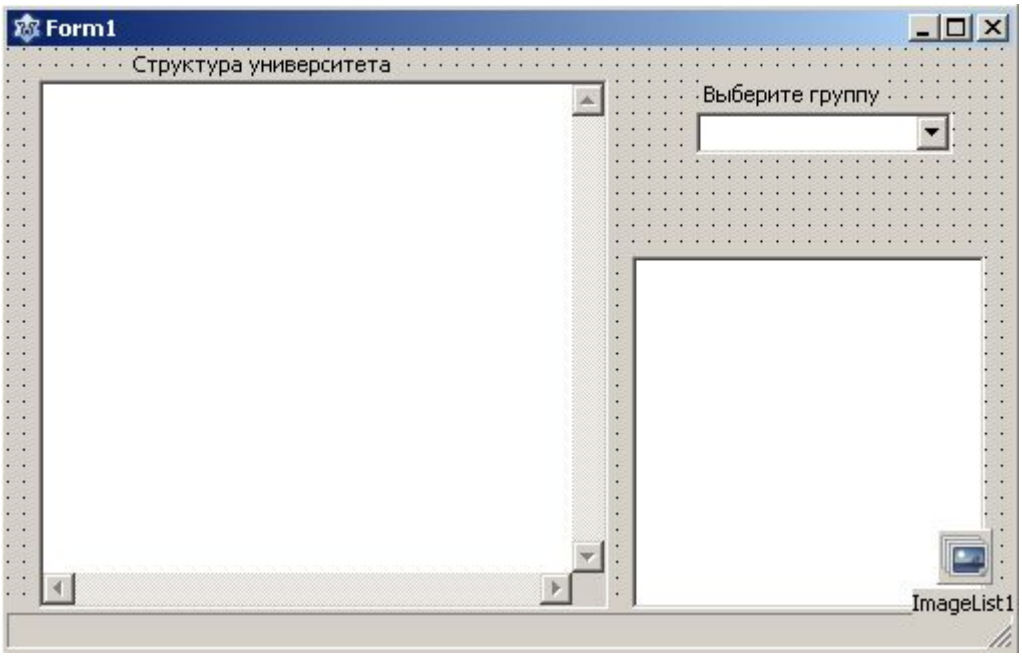


Рис. 9.69. Форма приложения

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   FileUtil,   LResources,   Forms,
  Controls, Graphics, Dialogs, ComCtrls, StdCtrls;
type
  { TForm1 }
  TForm1 = class(TForm)
    ComboBox1: TComboBox;
    ImageList1: TImageList;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    ListBox1: TListBox;
    StatusBar1: TStatusBar;
    TreeView1: TTreeView;
```

```
    procedure      ComboBox1KeyPress(Sender:      TObject;
                                     var Key: char);
    procedure ComboBox1Select(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure TreeView1Change(Sender: TObject;
                              Node: TTreeNode);
    procedure LoadListGroup(namefile: string);
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{ TForm1 }
procedure TForm1.FormCreate(Sender: TObject);
var
  i: integer;
begin
  with TreeView1 do
  begin
    Images:= ImageList1;
    ExpandSignType:= tvestPlusMinus;
    Items.Clear;
    Items.Add(nil, 'Ректор');
    Items.AddChild(Items.Item[0],
                   'Проректор по учебной работе');
    Items.AddChild(Items.Item[1], 'Учебная часть');
```

```
    Items.AddChild(Items.Item[1], 'Факультеты');
    Items.AddChild(Items.Item[3], 'Кафедры');
    Items.AddChild(Items.Item[4], 'Преподаватели');
    Items.AddChild(Items.Item[3], 'Студенты');
    Items.AddChild(Items.Item[6], 'Группы');
    Items.AddChild(Items.Item[0], 'Проректор по науке');
    Items.AddChild(Items.Item[8],
                   'Институт прикладных исследований');
    Items.AddChild(Items.Item[8], 'Отдел аспирантуры');
    Items.AddChild(Items.Item[0], 'Проректор по экономике');
    Items.AddChild(Items.Item[11], 'Отдел внешних связей');
    Items.AddChild(Items.Item[0], 'Проректор по АХЧ');
    for i:= 0 to Items.Count - 1 do
    begin
      Items.Item[i].ImageIndex:= 0;
      Items.Item[i].SelectedIndex:= 1;
    end;
    FullExpand;
  end;
  Label1.Caption:= 'Выберите группу';
  Label2.Caption:= '';
  Label3.Caption:= 'Структура университета';
  Label1.Visible:= false;
  Label2.Visible:= false;
  ComboBox1.Visible:= false;
  ListBox1.Visible:= false;
end;
procedure  TForm1.TreeView1Change(Sender:  TObject;  Node:
```

```
TTreeNode);
begin
  StatusBar1.SimpleText:= 'Выбран: ' + Node.Text;
  if Node.AbsoluteIndex = 7 then
  begin
    Label1.Visible:= true;
    ComboBox1.Visible:= true;
    ComboBox1.SetFocus;
    ComboBox1.ItemIndex:= 0;
  end
  else
  begin
    Label1.Visible:= false;
    Label2.Visible:= false;
    ComboBox1.Visible:= false;
    ListBox1.Visible:= false;
  end;
end;
procedure TForm1.ComboBox1Select(Sender: TObject);
begin
  with ComboBox1 do
  begin
    Label2.Visible:= true;
    Label2.Caption:='Список группы ' + Text;
    ListBox1.Visible:= true;
    case ItemIndex of
    0: LoadListGroup('List1.txt');
    1: LoadListGroup('List2.txt');
    2: LoadListGroup('List3.txt');
```

```
    else
      ShowMessage('Группа не выбрана');
    end;
  end;
end;
procedure TForm1.LoadListGroup(namefile: string);
var
  tfile: TStringList;
  str: string;
begin
  tfile:= TStringList.Create;
  tfile.LoadFromFile(namefile);
  str:= tfile.Text;
  {$IFDEF WINDOWS}
    str:= SysToUTF8(str); // преобразование в кодировку UTF-8
  {$ENDIF}
  with ListBox1 do
  begin
    Items.Text:= str;
  end;
  tfile.Free;
  ListBox1.Sorted:= true;
end;
procedure  TForm1.ComboBox1KeyPress(Sender:  TObject;  var
Key: char);
begin
  if Key = #13 then
  begin
    ComboBox1Select(Sender);
```

```
    Key:=#0;
  end;
end;
initialization
  {$I unit1.lrs}
end.
```

Обратите внимание узлы, находящиеся на первом уровне, имеют одинаковые индексы `Items.Item[]`. Например, у всех проректоров `Items.Item[0]`. Поскольку наше дерево фиксировано, добавление новых узлов и подузлов в программе не предусмотрено и, следовательно, индекс узла "Группы" нам известен, для определения выбора узла "Группы" мы воспользовались свойством `AbsoluteIndex`:

```
if Node.AbsoluteIndex = 7 then
```

Но можно было поступить и по-другому:

```
if Node.Text = 'Группы' then
```

Компонент `TTreeView` можно применить для отображения дерева папок дисков, например как в левом окне Проводника Windows. Попробуем и мы реализовать эту задачу. Скажу сразу, в панели компонентов во вкладке "`Misc`" имеется специализированный компонент `TShellTreeView` предназначенный именно для этого. Но мы попытаемся сделать это сами, так сказать "ручками", ведь "не боги горшки обжигают"!

Сначала нам необходимо познакомиться с некоторыми функциями, которые будут нам нужны для работы.

Функции поиска файлов

Для поиска файлов обычно применяются следующие функции:

- `FindFirst` - определяется следующим образом:

```
function FindFirst (Const Path: String; Attr: Longint;
                    out Rslt : TSearchRec): Longint;
```

Параметр `Path` – задает путь к каталогу, а также маску поиска файлов. Маска может содержать символы – шаблоны:

? – любой одиночный символ;

* - сколько угодно символов или даже отсутствие символов.

Например, маска *.* означает любые файлы с любым расширением, а маска *.pas означает любые файлы, с расширением pas.

Параметр `Attr` – атрибуты файла. Определяет, какие типы файлов следует искать. Может принимать следующие значения:

- `faReadOnly` – файлы только для чтения;
- `faHidden` – скрытые файлы;
- `faSysFile` – системные файлы;
- `faDirectory` – папки и каталоги;
- `faArchive` – архивные файлы;
- `faAnyFile` – любые файлы.

Можно применять любые их комбинации, например, для поиска скрытых и системных файлов можно задать `faHidden + faSysFile`.

Переменная `Rslt` представляет собой тип данных запись `TSearchRec`, определяется следующим образом (привожу только те поля, которые представляют для нас интерес):

```
Type
  TSearchRec = record
    Time: Longint;
    Size: Int64;
    Attr: Longint;
    Name: TFileName;
  end;
```

Здесь `Attr` - атрибуты файла (см. выше), `Time` - время и дата создания или последнего обновления файла в системном формате, `Size` - длина файла в байтах, Name - имя и расширение файла.

Директива `out` идентифицирует параметр только для вывода. Это эквивалентно `var` за исключением того, что значение не может быть изменено подпрограммой.

Функция `FindFirst` ищет файлы, соответствующие параметрам `Path` и `Attr`, возвращая 0, если первое соответствие найдено. Результат поиска при этом сохраняется в `Rslt`. Если соответствие не найдено, то функция возвращает отрицательное число и в `Rslt` ничего не записывается.

- `FindNext` - функция ищет следующий соответствующий файл, как задано в параметрах поиска функции `FindFirst`.

Определяется следующим образом:

```
function FindNext(var Rslt: TSearchRec): Longint;
```

Возвращает 0, если соответствие найдено, результат поиска сохраняется в `Rslt`.

- `FindClose` – освобождает ресурсы, используемые функциями `FindFirst` и `FindNext` при поиске файлов. Определена модуле `SysUtils`

следующим образом:

```
procedure FindClose (var F : TSearchRec);
```

Эту процедуру нужно вызывать всегда, после окончания поиска.

Напишем программу, которая ищет в текущем каталоге файл с расширением *.pas и, если находит, то выводит его в `TMemo`. Кроме того, в `TStatusBar` выводит имя файла, дату его создания и размер в байтах. Пусть все это происходит в момент показа главной формы программы, т.е. в обработчике `OnShow`.

```
procedure TForm1.FormShow(Sender: TObject);
var
 srRec: TSearchRec;
 tfile: TStringList;
 str: string;
begin
  if FindFirst('*.pas', faAnyFile, srRec) <> 0 then exit;
  tfile:= TStringList.Create;
  tfile.LoadFromFile(srRec.Name);
  str:= tfile.Text;
  Memo1.Lines.Add(str);
  tfile.Free;
  StatusBar1.SimpleText:='Файл: ' + srRec.Name + ', создан '
  + DateToStr(FileDateToDateTime(srRec.Time)) +
  ', размер: '+ IntToStr(srRec.Size) + ' байтов';
  FindClose(srRec);
end;
```

Здесь для преобразования системного формата даты и времени в обычный

формат дата-время мы воспользовались функцией `FileDateToDateTime()`, а для преобразования в строку функцией `DateToStr()`.

Обратите внимание, при загрузке файла в `TMemo` мы не стали преобразовывать его в кодировку `UTF-8` как это мы делали при загрузке текстового файла. Дело в том, что файл с исходным кодом модуля с расширением *.pas создается `Lazarus` и он уже в кодировке `UTF-8`!

Также нам понадобится функция

```
function SetCurrentDir(const NewDir: string ): boolean;
```

Устанавливает текущую директорию в `NewDir`, возвращая `true` в случае успеха.

Следующие две функции нужны только тем, кто использует Windows. Заядлым "линуксоидам", которые "на дух" не переносят Windows, описание этих функций могут пропустить.

Функция `GetLogicalDriveStrings` возвращает список всех дисков, зарегистрированных в Windows. Описание:

```
function GetLogicalDriveStrings(nBufferLength: DWORD;
                                lpBuffer: LPSTR): DWORD;
```

Параметр `nBufferLength` определяет максимальный размер массива (буфера) символов, указанного в параметре `lpBuffer`.

Параметр `lpBuffer` – указатель типа `PChar` на массив, который содержит строки с нулевым символом (#0) в конце.

В случае успешного завершения работы функции, возвращаемое значение является общей длиной (в символах) всех строк скопированных в буфер. В массиве `lpBuffer` содержатся имена всех дисков (включая логические дис-

ки, CD-ROM, съемные носители, такие как флешки, дискеты и т.д.), разделенные друг от друга завершающим нулём #0.

Если функция терпит неудачу, возвращаемое значение является нулем.

Функция `GetDriveType` - определяет и возвращает тип носителя. Описание:

```
function GetDriveType(lpRootPathName: LPCSTR):UINT;
```

`GetDriveType` использует всего один параметр – указатель на устройство. Функция возвращает одно из следующих значений:

- 0 – мнемоническое обозначение `Drive_Unknown`: диск не определен или не существует;
- 1 – `Drive_No_Root_Dir`: неверный путь;
- 2 – `Drive_Removable`: съемное устройство (дискета, флешка и т.д.);
- 3 – `Drive_Fixed`: тип устройства - фиксированный (жесткий диск);
- 4 – `Drive_Remote`: удаленный (сетевой) диск;
- 5 – `Drive_CDROM`: это устройство CD-ROM;
- 6 – `Drive_RAMDisk`: виртуальный диск, созданный в оперативной памяти.

Напишем функцию, которая определяет имена всех разделов жесткого диска (или дисков, если на компьютере их имеются несколько). Для простоты будем рассматривать только жесткие диски.

```
function TForm1.Find_Logical_Disks(): boolean;
const
  Hard_Disk = 3; // рассматриваем только жесткие диски
var
  size: LongWord;
  Drives: array[0..128] of char;
```

```
  pDrive: PChar;
begin
  size:= GetLogicalDriveStrings(SizeOf(Drives), Drives);
  if size = 0 then
  begin
    Result:= false;
    exit;
  end;
  if size > SizeOf(Drives) then
  raise Exception.Create(SysErrorMessage(ERROR_OUTOFMEMORY));
  pDrive:= Drives; // устанавливаем указатель на Drives
  while pDrive^ <> #0 do
  begin
     // если тип устройства жесткий диск
    if GetDriveType(pDrive) = Hard_Disk then
    begin
      s:= pDrive;
      s:= Copy(s, 1, 2);  // берем только имя раздела и двоеточие
      Memo1.Lines.Add(s); // добавляем имя раздела в Memo1
    end;
    inc(pDrive, 4);
  end;
end;
```

Здесь вопрос может вызвать оператор

```
inc(pDrive, 4);
```

Дело в том, что функция `GetLogicalDriveStrings()` записывает в массив (буфер) `Drives` имена устройств в виде 'Однобуквенный символ имени

устройства:\#0', например 'C:\#0', т.е. каждая строка массива состоит из четырех символов.

Найденные имена логических дисков мы сохраняем в Memo1, причем мы вырезали два последних символа (обратный слеш и #0). Можно, конечно, использовать массив строк. Но, поскольку количество установленных на том или ином компьютере жестких дисков и созданных в них разделов нам неизвестно, пришлось бы использовать динамический массив. И, к тому же, пришлось бы этот массив передавать в другие функции и процедуры. Поэтому проще использовать Memo1, ведь по сути это и есть динамический массив строк! Просто в программе мы его не будем показывать, установив свойство Visible в false. С помощью следующей процедуры мы записываем в TTreeView содержимое Memo1.

```
procedure TForm1.SetAllDirectories;
var
  i: integer;
  Node: TTreeNode;
begin
  TreeView1.BeginUpdate;
  for i:= 0 to Memo1.Lines.Count - 1 do
  begin
    Node= TreeView1.Items.AddChild(nil, Memo1.Lines[i]);
    Node.ImageIndex:= 0;
    Node.SelectedIndex:= 0;
    Node.HasChildren:= true;
  end;
  TreeView1.EndUpdate;
end;
```

Обратите внимание, если вы заносите в `TTreeView` несколько элементов (узлов), то может возникнуть неприятное мерцание экрана. Поэтому мы использовали `TreeView1.BeginUpdate` перед началом записи узлов и `TreeView1.EndUpdate` после завершения записи.

Далее мы указали, что узел имеет дочерние подузлы оператором

```
Node.HasChildren:= true;
```

В этом случае рядом узлом появится значок раскрытия узла. Ну, что же, теперь мы можем реализовать задачу отображения дерева папок и каталогов в `TTreeView`. Создайте новый проект. Поместите на форму компоненты `TTreeView`, `TMemo` и `TImageList`. Код программы:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, FileUtil, LResources, Forms,
  Controls, Graphics, Dialogs, ComCtrls, StdCtrls
  {$IFDEF WINDOWS}
   ,Windows
  {$ENDIF};
type
  { TForm1 }
  TForm1 = class(TForm)
    ImageList1: TImageList;
    Memo1: TMemo;
    TreeView1: TTreeView;
    procedure FormCreate(Sender: TObject);
```

```
    procedure TreeView1Change(Sender: TObject;
                              Node: TTreeNode);
    procedure TreeView1Expanding(Sender: TObject;
       Node: TTreeNode; var AllowExpansion: Boolean);
    function Real_Directory(sname: string): boolean;
    procedure Show_Only_Dir(ParentNode: TTreeNode);
    {$IFDEF WINDOWS}
    function Find_Logical_Disks(): boolean;
    procedure SetAllDirectories;
    {$ENDIF}
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{ TForm1 }
function TForm1.Real_Directory(sname: string): boolean;
begin
  result:= (sname <> '.') and (sname <> '..');
end;

{$IFDEF WINDOWS}
// Эти функции нужны только для Windows
function TForm1.Find_Logical_Disks(): boolean;
const
  Hard_Disk = 3; // рассматриваем только жесткие диски
```

```
var
  size: LongWord;
  Drives: array[0..128] of char;
  pDrive: PChar;
  s: string;
begin
  size:= GetLogicalDriveStrings(SizeOf(Drives), Drives);
  if size = 0 then
  begin
    Result:= false;
    exit;
  end;
  if size > SizeOf(Drives) then
  raise Exception.Create(SysErrorMessage(ERROR_OUTOFMEMORY));
  pDrive:= Drives; // устанавливаем указатель на Drives
  while pDrive^ <> #0 do
  begin
     // если тип устройства жесткий диск
    if GetDriveType(pDrive) = Hard_Disk then
    begin
      s:= pDrive;
      s:= Copy(s, 1, 2);  // берем только имя раздела и двоеточие
      Memo1.Lines.Add(s); // добавляем имя раздела в Memo1
    end;
    inc(pDrive, 4);
  end;
end;
procedure TForm1.SetAllDirectories;
var
```

```
  i: integer;
  Node: TTreeNode;
begin
  TreeView1.BeginUpdate;
  for i:= 0 to Memo1.Lines.Count - 1 do
  begin
    Node:= TreeView1.Items.AddChild(nil, Memo1.Lines[i]);
    Node.ImageIndex:= 0;
    Node.SelectedIndex:= 0;
    Node.HasChildren:= true;
  end;
  TreeView1.EndUpdate;
end;
{$ENDIF}

procedure TForm1.FormCreate(Sender: TObject);
var
  Node: TTreeNode;
  srNode, srChild: TSearchRec;
  searchMask: string;
  SetDirWin: boolean = false;
begin
  Memo1.Clear;
  Memo1.Visible:= false;
  TreeView1.Images:= ImageList1;
  TreeView1.ExpandSignType:= tvestPlusMinus;
  TreeView1.BeginUpdate;
    {$IFDEF WINDOWS}
        // Определение логических дисков
```

```
    if Find_Logical_Disks() then
    begin
      SetAllDirectories;
      SetDirWin:= true;
    end
    else
    {Если произошла ошибка в функции Find_Logical_Disks(),
     то выбираем корневой каталог текущего диска}
      SetCurrentDir('\');
  {$ELSE}
    SetCurrentDir('/'); // корневой каталог в Linux
  {$ENDIF}
  if not SetDirWin then
  begin
    path:= GetCurrentDir;
    if FindFirst(path + '*', faDirectory, srNode) = 0
    then
    begin
      repeat
        // Показываем только каталоги
        if (srNode.Attr and faDirectory <> 0)
        and (Real_Directory(srNode.Name)) then
        begin
          Node:= TreeView1.Items.AddChild(nil,
                 SysToUTF8(srNode.Name));
          Node.ImageIndex:= 0;
          Node.SelectedIndex:= 0;
          {$IFDEF WINDOWS}
          searchMask:= path + srNode.Name + '\*';
```

```
            {$ELSE}
              searchMask:= path + srNode.Name + '/*';
            {$ENDIF}
            if FindFirst(searchMask, faDirectory,
               srChild) = 0 then
            repeat
              if (srChild.Attr and faDirectory <> 0) and
                  Real_Directory(srChild.Name)
              then Node.HasChildren:= true;
            until (FindNext(srChild) <> 0) or
                  Node.HasChildren;
            // Освобождение занятых ресурсов
            SysUtils.FindClose(srChild);
          end;
        until FindNext(srNode) <> 0;
        // Освобождение занятых ресурсов
        SysUtils.FindClose(srNode);
      end;
    end;
  TreeView1.EndUpdate;
end;

procedure TForm1.Show_Only_Dir(ParentNode: TTreeNode);
var
  srNode, srChild: TSearchRec;
  Node: TTreeNode;
  path: string;
  searchMask: string;
begin
```

```
  Node:= ParentNode;
  path:= '';
  repeat
    {$IFDEF WINDOWS}
    path:= UTF8ToSys(Node.Text) + '\' + path;
    {$ELSE}
    path:= '/' + Node.Text + '/' + path;
    {$ENDIF}
    Node:= Node.Parent;
  until Node = nil;
  if FindFirst(path + '*', faDirectory, srNode) = 0 then
  repeat
  if (srNode.Attr and faDirectory <> 0) and
      Real_Directory(srNode.Name) then
  begin
    Node:= TreeView1.Items.AddChild(ParentNode,
           SysToUTF8(srNode.Name));
    Node.ImageIndex:= 0;
    Node.SelectedIndex:= 0;
    {$IFDEF WINDOWS}
      searchMask:= path + srNode.Name + '\*';
    {$ELSE}
      searchMask:= path + srNode.Name + '/*';
    {$ENDIF}
    if FindFirst(searchMask, faDirectory, srChild) = 0
then
    repeat
      if (srChild.Attr and faDirectory <> 0) and
          Real_Directory(srChild.Name)
```

```
      then Node.HasChildren:= true;
    until (FindNext(srChild) <> 0) or Node.HasChildren;
    SysUtils.FindClose(srChild);
  end;
  until FindNext(srNode) <> 0;
  SysUtils.FindClose(srNode);
end;

procedure TForm1.TreeView1Change(Sender: TObject; Node:
TTreeNode);
begin
  if Node = nil then exit;
  TreeView1.BeginUpdate;
  Node.DeleteChildren;
  Show_Only_Dir(Node);
  Node.Expanded:= true;
  TreeView1.EndUpdate;
end;

procedure TForm1.TreeView1Expanding(Sender: TObject;
Node: TTreeNode;
  var AllowExpansion: Boolean);
begin
  TreeView1.BeginUpdate;
  Node.DeleteChildren;
  Show_Only_Dir(Node);
  TreeView1.EndUpdate;
end;
initialization
```

```
  {$I unit1.lrs}
end.
```

Рассмотрим процедуру `FormCreate`. В ней, если это Windows, с помощью функции `Find_Logical_Disks()` и процедуры `SetAllDirectories` определяются имена разделов жестких дисков и выводятся в `TreeView1`. Окно программы для Windows будет иметь вид, рис. 9.70:

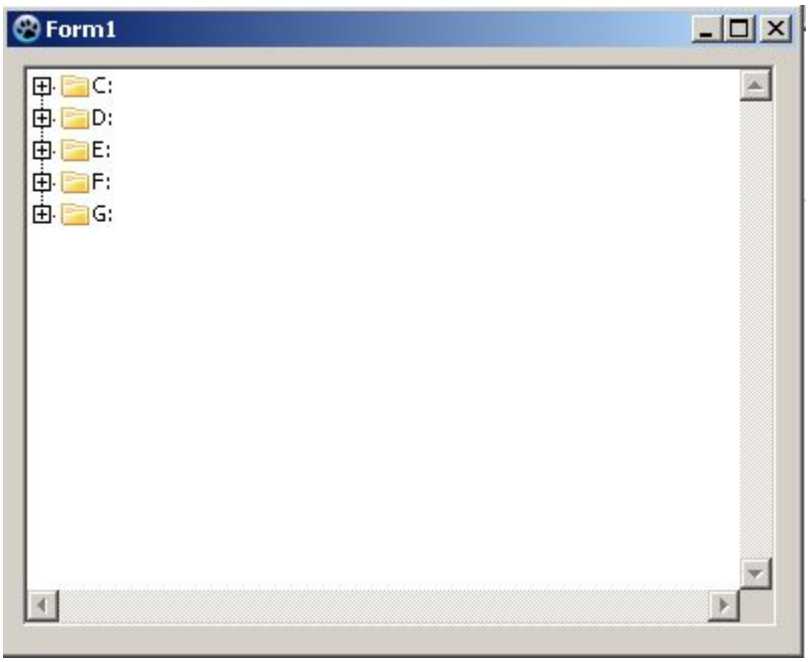


Рис. 9.70. Вид окна программы в Windows

В Linux файловая система организована совсем по-другому. Поэтому мы оператором

```
SetCurrentDir('/');
```

устанавливаем корневой каталог и программа сканирует каталоги только верхнего уровня, рис. 9.71.

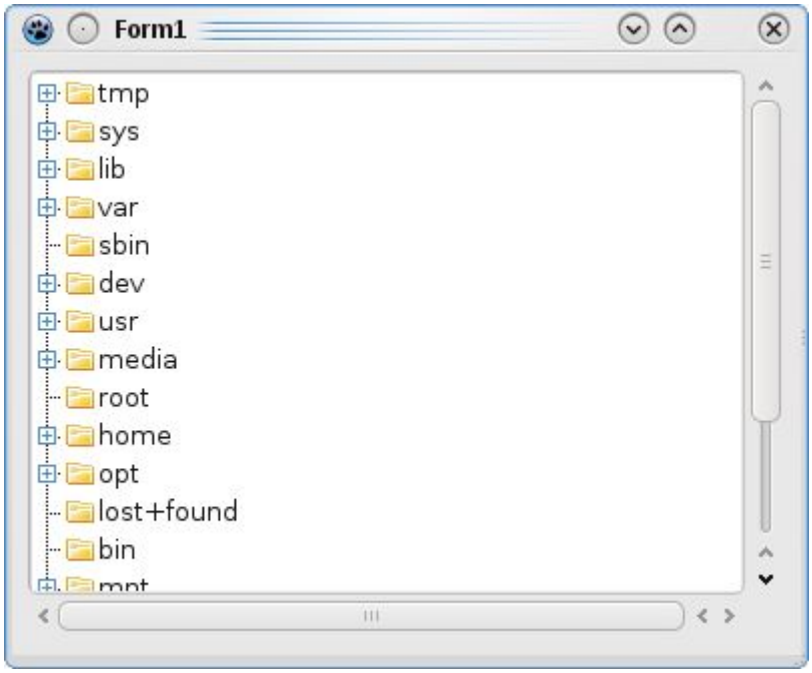


Рис. 9.71. Вид окна программы в Linux

Таким образом, наша программа не сканирует все диски и каталоги. Программа вначале лишь определяет буквы дисков и разделов дисков (если это Windows) или сканирует только каталоги самого верхнего уровня, расположенные в корневом каталоге (если это Linux).

Лишь при нажатии пользователем на какой-либо узел или на кнопку раскрытия этого узла, программа начнет сканировать каталоги, расположенные на один уровень ниже и только для текущего каталога. Поясню примером. Пусть мы находимся в каталоге `C:\lazarus\components\lazreport` или `/usr/lib/lazarus/ components/lazreport`.

При нажатии на узел с именем '`lazreport`' программа просканирует этот каталог и найдет пять подкаталогов `doc`, `images`, `samples`, `source`, `tools`. При этом эти подкаталоги имеют в свою очередь еще вложенные подкаталоги, но всех их программа в данный момент сканировать не будет! Для того чтобы определить, что, например, каталог `samples` имеет вложенные подкаталоги, программе достаточно найти только один подкаталог (первый попавшийся!). В нашем случае это будет каталог `barcode`.

Это позволяет значительно ускорить работу программы.

Рассмотрим процедуру `Show_Only_Dir`, которая и реализует изложенный алгоритм. Процедура вызывается в обработчиках `OnChange` и `OnExpanding`. Вначале, процедура определяет полный путь к текущему каталогу, двигаясь снизу вверх от дочернего узла к родительскому и до корневого. Затем запускается цикл поиска `FindFirst – FindNext` с помощью которого находятся все подкаталоги текущего каталога. Вложенный цикл `FindFirst – FindNext` позволяет определить, имеются ли у найденных подкаталогов вложенные подкаталоги. Этот внутренний цикл завершается, как только вложенный каталог найден, а его родитель помечается признаком `Node.HasChildren:= true`.

Отметим еще, что перед вызовом процедуры `Show_Only_Dir` мы оператором

```
Node.DeleteChildren
```

удаляем все потомки текущего узла, поскольку процедура заново формирует их. Точно по такому же алгоритму сканируется корневой каталог в обработчике `OnCreate`.

Вот, собственно и все! Осталось пояснить, для чего предназначена функция `Real_Directory()`.

Как известно, в каждом каталоге имеются две особые записи. Одна из них обозначается просто точкой и является ссылкой на текущий каталог, а вторая запись, обозначаемая двумя точками (по стандарту так и читается точка-точка) является ссылкой на родительский каталог. Функции `FindFirst–FindNext` их находят и идентифицируют как каталоги. Но, обычно, в дереве каталогов их не принято показывать. Поэтому функция `Real_Directory()`проверяет, не является ли имя найденного каталога точкой или точкой-точкой и возвращает `true`, если это не так.

Компонент TListView

В отличие от `TListBox` компонент `TListView` позволяет отображать данные в разных стилях или режимах. Так, данные могут отображаться в виде списка, крупных или мелких пиктограмм и, наконец, в виде таблицы. Стиль отображения задается свойством `ViewStyle`, табл. 9.5.

Табл. 9.5.

Значение	Стиль отображения данных
`vsIcon`	Элементы списка появляются в виде больших значков с надписью под ними. Картинки для больших значков хранятся в компоненте `TImageList` указанном в свойстве `LargeImages`. Возможно их перетаскивание.
`vsSmallIcon`	Элементы списка появляются в виде маленьких значков с надписью справа. Картинки для маленьких значков хранятся в компоненте `TImageList` указанном в свойстве `SmallImages`. Возможно их перетаскивание
`vsList`	Элементы списка появляются в колонке один под другим с надписью справа. Перетаскивание невозможно
`vsReport`	Элементы списка появляются в нескольких колонках один под другим. В первой содержится маленький значок и надпись, в остальных — определенная программистом информация. Если свойство `ShowColumnHeaders` установлено в значение `true`, колонки снабжаются заголовками

Элементы списка содержатся в свойстве `Items` объекта `TListItem`. Компонент можно заполнять как вручную, во время проектирования, так и программно.

Для ручного заполнения имеется редактор, похожий на редактор `TTreeView`. В нем точно так же задаются "Новый элемент" и "Новый подэлемент". Как и в `TTreeView` изображения необходимо предварительно занести в `TImageList`, затем указывать индекс нужного изображения в свойстве `ImageIndex`, рис. 9.72.

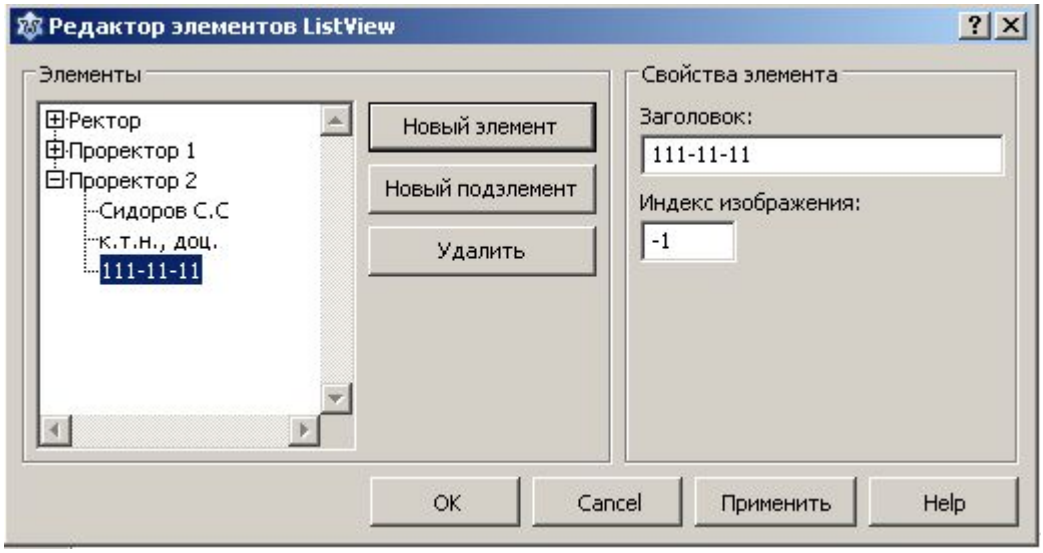
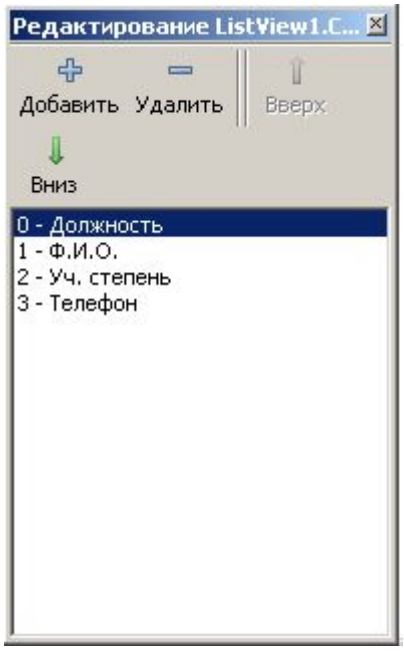


Рис. 9.72. Редактор элементов TListView

Но подэлементы будут видны только в режиме `vsReport`. Для того чтобы информация, введенная на рис. 9.72. была видна, необходимо создать четыре колонки (столбца). Для этого в Инспекторе объектов нажмите кнопку с многоточием напротив свойства `Columns`, появится редактор колонок рис. 9.73. Чтобы добавить колонку нажмите кнопку "Добавить".

В Инспекторе объектов появятся свойства этой колонки, рис. 9.74. Свойство `Caption` задает заголовок колонки, свойство `Width` ширину колонки. Установите необходимые свойства для каждого из четырех колонок и запустите программу, рис. 9.75.



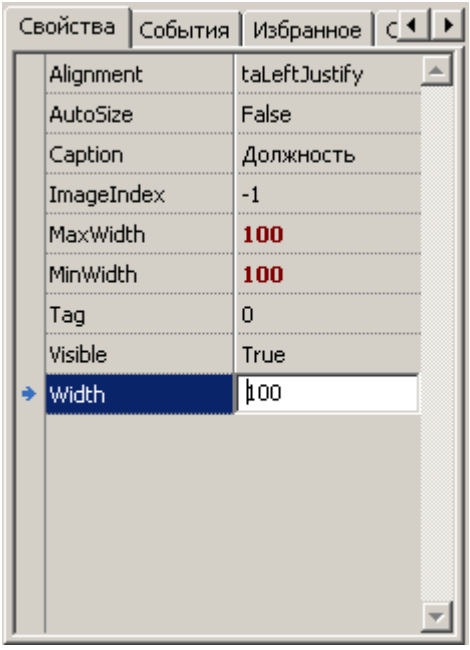


Рис. 9.73. Редактор колонок Рис. 9.74. Свойства колонки

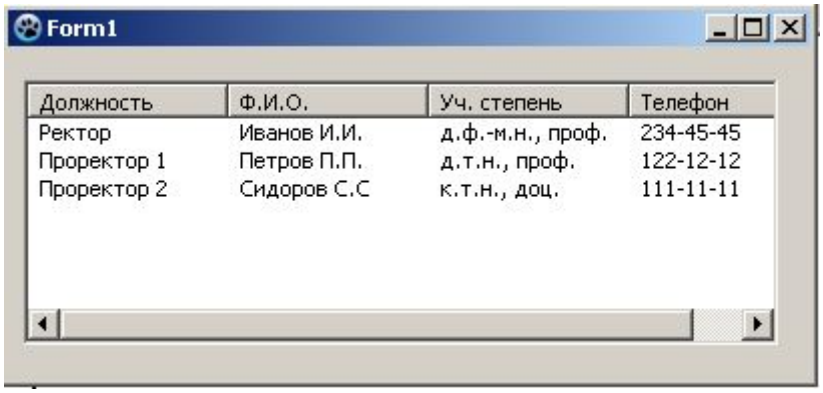


Рис. 9.75. Окно программы

Рассмотрим основные свойства компонента:

- `Columns` – содержит объекты – колонки (столбцы). Колонки видны только для стиля отображения `ViewStyle = vsReport`. При этом должна быть определена хотя бы одна колонка. Иначе элементы не будут видны;
- `Column[Index: integer]` – открывает доступ к колонкам элементов по их индексам;
- `ColumnClick: boolean` – разрешает или запрещает генерацию события

OnColumnClick;

- GridLines: boolean – разрешает или запрещает показ линий в режиме ViewStyle = vsReport;
- HideSelection: boolean – запрещает или разрешает сохранять выбор элементов при потере компонентом фокуса;
- Items: TListItems – содержит список всех элементов;
- LargeImages: TImageList – указывает источник крупных пиктограмм;
- MultiSelect: boolean – разрешает/запрещает множественный выбор;
- ReadOnly: boolean - запрещает/разрешает редактирование надписей;
- ShowColumnHeaders: boolean - разрешает/запрещает показ заголовков колонок в режиме ViewStyle = vsReport;
- SmallImages – указывает источник мелких пиктограмм;
- SortType – указывает способ сортировки элементов, возможные значения stNone, stData, stText, stBoth;
- StateImages – указывает источник пиктограмм для выбранных элементов;

Свойства и методы класса TListItems:

- Item[Index: integer]: TListItem – открывает индексный доступ к элементам списка;
- Count – содержит количество элементов в Item;
- function Add: TListItem – добавляет очередной элемент к списку;
- procedure Clear – очищает список;
- procedure BeginUpdate – блокирует обновление экрана до тех пор, пока не будет выполнен метод EndUpdate. Используется при одновременной вставке нескольких элементов списка для предотвращения мерцания экрана;
- procedure Delete(Index: Integers) – удаляет элемент списка с

индексом Index;

- procedure EndUpdate - отменяет блокировку обновления экрана методом BeginUpdate;
- function IndexOf(Value: TListItem) – возвращает индекс элемента Value;
- function Insert(Index: integer): TListItem – вставляет новый элемент после элемента, заданного Index;

Очень важное свойство SubItems: TStrings.

При помощи этого свойства с каждым элементом списка может быть связан целый набор строк и объектов. Сначала следует создать необходимое количество колонок. При этом следует учесть, что первая из них будет отведена под сам элемент списка. Последующие же колонки будут отображать тексты строк из свойства Items.SubItems.

Заполним программно TListView данными из рис. 9.72. В новом проекте поместите компонент TListView и в обработчике OnShow формы введите следующий код:

```
procedure TForm1.FormShow(Sender: TObject);
var
  ListItem: TListItem;
begin
  with ListView1 do
  begin
    ViewStyle:= vsReport;
    // создаем колонку
    Columns.Add;
    // заголовок колонки
    Columns.Items[0].Caption:='Должность';
```

```
// ширина колонки
Columns.Items[0].Width:= 100;
// создаем колонку
Columns.Add;
// заголовок колонки
Columns.Items[1].Caption:='Ф.И.О.';
// ширина колонки
Columns.Items[1].Width:= 100;
// создаем колонку
Columns.Add;
// заголовок колонки
Columns.Items[2].Caption:='Уч. степень';
// ширина колонки
Columns.Items[2].Width:= 100;
// создаем колонку
Columns.Add;
// заголовок колонки
Columns.Items[3].Caption:='Телефон';
// ширина колонки
Columns.Items[3].Width:= 100;
// добавляем первый элемент
ListItem:= Items.Add;
ListItem.Caption:= 'Ректор';
// добавляем подэлементы первого элемента
ListItem.SubItems.Add('Иванов И.И.');
ListItem.SubItems.Add('д.ф.-м.н., проф.');
ListItem.SubItems.Add('234-45-45');
// добавляем второй элемент
```

```
    ListItem:= Items.Add;
    ListItem.Caption:= 'Проректор 1';
    // добавляем подэлементы второго элемента
    ListItem.SubItems.Add('Петров П.П.');
    ListItem.SubItems.Add('д.т.н., проф.');
    ListItem.SubItems.Add('122-12-12');
    // добавляем третий элемент
    ListItem:= Items.Add;
    ListItem.Caption:= 'Проректор 2';
    // добавляем подэлементы третьего элемента
    ListItem.SubItems.Add('Сидоров С.С.');
    ListItem.SubItems.Add('к.т.н., доц.');
    ListItem.SubItems.Add('111-11-11');
  end;
end;
```

Продолжим наши опыты с Проводником. Давайте, теперь реализуем правое окно Проводника. Для этой цели компонент `TListView` подходит как нельзя лучше. Создайте новый проект. Поместите на форму `TTreeView`, затем компонент `TListView`. Не забудьте поместить на форму также `TMemo` и `TImageList`.

Если мы посмотрим, как работает Проводник, то мы увидим, что при нажатии кнопки раскрытия узла-каталога в левом окне, узел просто раскрывается, а в правом окне ничего не появляется. Если же мы нажмем на сам узел-каталог, то он также раскрывается и, в то же время, содержимое этого каталога (подкаталоги и файлы) появляются в `ListView1`.

Для отображения содержимого узла-каталога можно использовать ту же самую процедуру `Show_Only_Dir()`, только найденные подкаталоги теперь

будут добавляться не только `TTreeView`, но и в `TListView`. Но эта процедура используется для вывода каталогов в левом окне, поэтому создадим процедуру с другим именем. Для вывода файлов, если в данном подкаталоге они имеются, также создадим процедуру. Алгоритм тот же самый, только функции `FindFirst-FindNext` будут искать уже не каталоги, а файлы. Причем скрытые файлы мы показывать не будем. Код программы:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, FileUtil, LResources, Forms,
  Controls, Graphics, Dialogs, ComCtrls, StdCtrls,
  ExtCtrls, LCLType,

  {$IFDEF UNIX}
    unix;
  {$ELSE}
    Windows;
  {$ENDIF}

type
  { TForm1 }
  TForm1 = class(TForm)
    ImageList1: TImageList;
    ListView1: TListView;
    Memo1: TMemo;
    TreeView1: TTreeView;
    procedure FormCreate(Sender: TObject);
```

```
    procedure TreeView1Change(Sender: TObject;
                                  Node: TTreeNode);
    procedure TreeView1Expanding(Sender: TObject;
       Node: TTreeNode; var AllowExpansion: Boolean);
    function Real_Directory(sname: string): boolean;
    procedure Show_Only_Dir(ParentNode: TTreeNode);
    procedure Show_in_ListView_Dir(ParentNode: TTreeNode);
    procedure Show_in_ListView_Files(ParentNode: TTreeNode);

    {$IFDEF WINDOWS}
    function Find_Logical_Disks(): boolean;
    procedure SetAllDirectories;

    {$ENDIF}
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;
  path: string;
implementation

{ TForm1 }

function TForm1.Real_Directory(sname: string): boolean;
begin
```

```
  result:= (sname <> '.') and (sname <> '..');
end;

{$IFDEF WINDOWS}
// Эти функции нужны только для Windows
function TForm1.Find_Logical_Disks(): boolean;
const
  Hard_Disk = 3;  // рассматриваем только жесткие диски
var
  size: LongWord;
  Drives: array[0..128] of char;
  pDrive: PChar;
  s: string;
begin
  size:= GetLogicalDriveStrings(SizeOf(Drives), Drives);
  if size = 0 then
  begin
    Result:= false;
    exit;
  end;
  if size > SizeOf(Drives) then
    raise Exception.Create(SysErrorMessage(ERROR_OUTOFMEMORY));
  pDrive:= Drives; // устанавливаем указатель на Drives
  while pDrive^ <> #0 do
  begin
    // если тип устройства жесткий диск
    if GetDriveType(pDrive) = Hard_Disk then
    begin
      s:= pDrive;
```

```
      s:= Copy(s, 1, 2);  // берем только имя раздела и двоеточие
      Memo1.Lines.Add(s); // добавляем имя раздела в Memo1
    end;
    inc(pDrive, 4);
  end;
end;

procedure TForm1.SetAllDirectories;
var
  i: integer;
  Node: TTreeNode;
begin
  TreeView1.BeginUpdate;
  for i:= 0 to Memo1.Lines.Count - 1 do
  begin
    Node:= TreeView1.Items.AddChild(nil, Memo1.Lines[i]);
    Node.ImageIndex:= 0;
    Node.SelectedIndex:= 0;
    Node.HasChildren:= true;
  end;
  TreeView1.EndUpdate;
end;
{$ENDIF}

procedure TForm1.FormCreate(Sender: TObject);
var
  Node: TTreeNode;
  srNode, srChild: TSearchRec;
  searchMask: string;
```

```
  SetDirWin: boolean = false;
begin
  Memo1.Clear;
  Memo1.Visible:= false;
  TreeView1.Images:= ImageList1;
  TreeView1.ExpandSignType:= tvestPlusMinus;
  TreeView1.BeginUpdate;
    {$IFDEF WINDOWS}
       // Определение логических дисков
      if Find_Logical_Disks() then
      begin
        SetAllDirectories;
        SetDirWin:= true;
      end
      else
      {Если произошла ошибка в функции Find_Logical_Disks(),
       то выбираем корневой каталог текущего диска}
        SetCurrentDir('\');
    {$ELSE}
      SetCurrentDir('/'); // корневой каталог в Linux
    {$ENDIF}
    if not SetDirWin then
    begin
      path:= GetCurrentDir;
      if FindFirst(path + '*', faDirectory, srNode) = 0
      then
      begin
        repeat
          // Показываем только каталоги
```

```
          if (srNode.Attr and faDirectory <> 0)
          and (Real_Directory(srNode.Name)) then
          begin
          Node:= TreeView1.Items.AddChild(nil,
                 SysToUTF8(srNode.Name));
          Node.ImageIndex:= 0;
          Node.SelectedIndex:= 0;
          {$IFDEF WINDOWS}
            searchMask:= path + srNode.Name + '\*';
            {$ELSE}
              searchMask:= path + srNode.Name + '/*';
            {$ENDIF}
            if FindFirst(searchMask, faDirectory,
               srChild) = 0 then
            repeat
              if (srChild.Attr and faDirectory <> 0) and
                  Real_Directory(srChild.Name)
              then Node.HasChildren:= true;
            until (FindNext(srChild) <> 0) or
                   Node.HasChildren;
            // Освобождение занятых ресурсов
            SysUtils.FindClose(srChild);
          end;
        until FindNext(srNode) <> 0;
        // Освобождение занятых ресурсов
        SysUtils.FindClose(srNode);
      end;
    end;
  TreeView1.EndUpdate;
```

```
  // установка свойств ListView1
  with ListView1 do
  begin
    Align:= alClient;
    ViewStyle:= vsReport;
    Columns.Add;
    Columns.Items[0].Width:= 425;
    ScrollBars:= ssAutoBoth;
    SmallImages:= ImageList1;
  end;
end;

procedure TForm1.Show_Only_Dir(ParentNode: TTreeNode);
var
  srNode, srChild: TSearchRec;
  Node: TTreeNode;
  searchMask: string;
begin
  Node:= ParentNode;
  path:= '';
  repeat
    {$IFDEF WINDOWS}
    path:= UTF8ToSys(Node.Text) + '\' + path;
    {$ELSE}
    path:= '/' + Node.Text + '/' + path;
    {$ENDIF}
    Node:= Node.Parent;
  until Node = nil;
  if FindFirst(path + '*', faDirectory, srNode) = 0 then
```

```
  repeat
  if (srNode.Attr and faDirectory <> 0) and
      Real_Directory(srNode.Name) then
  begin
    Node:= TreeView1.Items.AddChild(ParentNode,
           SysToUTF8(srNode.Name));
    Node.ImageIndex:= 0;
    Node.SelectedIndex:= 0;
    {$IFDEF WINDOWS}
      searchMask:= path + srNode.Name + '\*';
    {$ELSE}
      searchMask:= path + srNode.Name + '/*';
    {$ENDIF}
    if FindFirst(searchMask, faDirectory, srChild) = 0
    then
    repeat
      if (srChild.Attr and faDirectory <> 0) and
          Real_Directory(srChild.Name)
      then Node.HasChildren:= true;
    until (FindNext(srChild) <> 0) or Node.HasChildren;
    SysUtils.FindClose(srChild);
  end;
  until FindNext(srNode) <> 0;
  SysUtils.FindClose(srNode);
end;

procedure TForm1.Show_in_ListView_Dir(ParentNode:
                                        TTreeNode);
var
```

```
  srNode, srChild: TSearchRec;
  Node: TTreeNode;
  ListItem: TListItem;
  searchMask: string;
begin
  Node:= ParentNode;
  path:= '';
  repeat
    {$IFDEF WINDOWS}
    path:= UTF8ToSys(Node.Text) + '\' + path;
    {$ELSE}
    path:= '/' + Node.Text + '/' + path;
    {$ENDIF}
    Node:= Node.Parent;
  until Node = nil;
  if FindFirst(path + '*', faDirectory, srNode) = 0 then
  repeat
  if (srNode.Attr and faDirectory <> 0) and
      Real_Directory(srNode.Name) then
  begin
    Node:= TreeView1.Items.AddChild(ParentNode,
           SysToUTF8(srNode.Name));
    if srNode.Attr and faHidden = 0 then
    begin
      ListItem:= ListView1.Items.Add;
      ListItem.Caption:= Node.Text;
      ListItem.ImageIndex:= 0;
    end;
    {$IFDEF WINDOWS}
```

```
      searchMask:= path + srNode.Name + '\*';
    {$ELSE}
      searchMask:= path + srNode.Name + '/*';
    {$ENDIF}
    if FindFirst(searchMask, faDirectory, srChild) = 0
    then
    repeat
      if (srChild.Attr and faDirectory <> 0) and
          Real_Directory(srChild.Name)
      then Node.HasChildren:= true;
    until (FindNext(srChild) <> 0) or Node.HasChildren;
    SysUtils.FindClose(srChild);
  end;
  until FindNext(srNode) <> 0;
  SysUtils.FindClose(srNode);
end;

procedure TForm1.Show_in_ListView_Files(ParentNode:
                                          TTreeNode);
var
  srNode: TSearchRec;
  Node: TTreeNode;
  ListItem: TListItem;
begin
  Node:= ParentNode;
  path:= '';
  repeat
    {$IFDEF WINDOWS}
    path:= UTF8ToSys(Node.Text) + '\' + path;
```

```
    {$ELSE}
    path:= '/' + Node.Text + '/' + path;
    {$ENDIF}
    Node:= Node.Parent;
  until Node = nil;
  if FindFirst(path + '*', faAnyFile, srNode) = 0 then
  repeat
  if srNode.Attr and faDirectory = 0 then
  begin
    if srNode.Attr and faHidden = 0 then
    begin
      ListItem:= ListView1.Items.Add;
      ListItem.Caption:= SysToUTF8(srNode.Name);
      ListItem.ImageIndex:= 1;
    end;
  end;
  until FindNext(srNode) <> 0;
  SysUtils.FindClose(srNode);
end;

procedure TForm1.TreeView1Change(Sender: TObject;
                                 Node: TTreeNode);
begin
  if Node = nil then exit;
  TreeView1.BeginUpdate;
  ListView1.BeginUpdate;
  ListView1.Clear;
  Node.DeleteChildren;
  Show_in_ListView_Dir(Node);
```

```
  Show_in_ListView_Files(Node);
  Node.Expanded:= true;
  ListView1.EndUpdate;
  TreeView1.EndUpdate;
end;
procedure TForm1.TreeView1Expanding(Sender: TObject;
      Node: TTreeNode; var AllowExpansion: Boolean);
begin
  TreeView1.BeginUpdate;
  Node.DeleteChildren;
  Show_Only_Dir(Node);
  TreeView1.EndUpdate;
end;
initialization
  {$I unit1.lrs}
end.
```

Почему мы разделили вывод в `TListView` каталогов и файлов на две процедуры? Если их выводить в одной процедуре, то они будут отображаться вперемешку. А ведь принято, чтобы сначала отображались каталоги, и лишь затем файлы. Здесь сортировка не поможет.

Обратите внимание, для файлов мы выбрали другую пиктограмму.

Теперь нам нужно реализовать раскрытие папки в `TListView` по двойному щелчку по его имени. Для этого напишем две процедуры, одна из которых показывает только каталоги, а другая только файлы. Вызываться они будут в обработчике события `OnDblClick` для `TListView`.

```
procedure TForm1.Show_ListView_Dir(DirName: string);
var
```

```
  srNode: TSearchRec;
  ListItem: TListItem;
begin
  oldpath:= path;
  {$IFDEF WINDOWS}
  path:= path + UTF8ToSys(DirName) + '\' ;
  {$ELSE}
  path:= '/' + path + DirName + '/';
  {$ENDIF}
  ListView1.Clear;
  if FindFirst(path + '*', faDirectory, srNode) = 0 then
  repeat
  if (srNode.Attr and faDirectory <> 0) and
      Real_Directory(srNode.Name) then
  begin
    ListItem:= ListView1.Items.Add;
    ListItem.Caption:= SysToUtf8(srNode.Name);
    ListItem.ImageIndex:= 0;
  end;
  until FindNext(srNode) <> 0;
  SysUtils.FindClose(srNode);
end;
procedure TForm1.Show_ListView_Files(DirName: string);
var
  srNode: TSearchRec;
  ListItem: TListItem;
begin
  path:= '';
  {$IFDEF WINDOWS}
```

```
  path:= oldpath + UTF8ToSys(DirName) + '\' ;
  {$ELSE}
  path:= '/' + oldpath + DirName + '/';
  {$ENDIF}
  if FindFirst(path + '*', faAnyFile, srNode) = 0 then
  repeat
  if srNode.Attr and faDirectory = 0 then
  begin
    if srNode.Attr and faHidden = 0 then
    begin
      ListItem:= ListView1.Items.Add;
      ListItem.Caption:= SysToUTF8(srNode.Name);;
      ListItem.ImageIndex:= 1;
    end;
  end;
  until FindNext(srNode) <> 0;
  SysUtils.FindClose(srNode);
end;

procedure TForm1.ListView1DblClick(Sender: TObject);
var
  s: string;
  DirName: string;
  srNode: TSearchRec;
begin
  if ListView1.Selected = nil then exit;
  {$IFDEF WINDOWS}
  s:= path + UTF8ToSys(ListView1.Selected.Caption);
  {$ELSE}
```

```
  s:= path + ListView1.Selected.Caption;
  {$ENDIF}
  if FindFirst(s, faAnyFile, srNode) = 0 then
  if (srNode.Attr and faDirectory) <> 0 then
  begin
    DirName:= ListView1.Selected.Caption;
    Show_ListView_Dir(DirName);
    Show_ListView_Files(DirName);
    SysUtils.FindClose(srNode);
  end;
end;
```

Добавим в наш Проводник еще одну функциональность, а именно по двойному щелчку на имя исполняемого файла будем его запускать. Запуск исполняемого файла (внешнего приложения) в Windows проще всего организовать с помощью функции `WinExec`. Эта функция определена следующим образом:

```
function WinExec(lpCmdLine:LPCSTR; uCmdShow:UINT):UINT;
```

Параметр `lpCmdLine` является указателем на строку с нулевым символом в конце, содержащую имя выполняемого файла Можно включить полный путь к файлу и, если необходимо, параметры командной строки. Если имя файла указано без пути, то Windows будет искать в каталогах выполняемый файл в следующей последовательности:

- Каталог, из которого загружено приложение;
- Текущий каталог;
- Системный каталог Windows, возвращаемый функцией `GetSystemDirectory`;
- Каталог Windows, возвращаемый функцией `GetWindowsDirectory`;

- Список каталогов из переменной окружения PATH.

Параметр uCmdShow определяет форму представления окна запускаемого приложения Windows. Чаще всего используется значение 1, при котором окно запускаемого приложения активизируется и отображается на экране. Если это окно в данный момент свернуто или развернуто, то оно восстанавливается до своих первоначальных размеров и отображается в первоначальной позиции.

В Linux исполняемый файл можно запустить с помощью функции Shell.

Определена в модуле unix:

```
function Shell(const Command: String): cint;
```

Параметр Command задает имя файла с путем или без.

Обработчик события OnDblClick перепишите следующим образом:

```
procedure TForm1.ListView1DblClick(Sender: TObject);
var
  s: string;
  progr: pchar;
  srNode: TSearchRec;
  DirName: string;
begin
  if ListView1.Selected = nil then exit;
  {$IFDEF WINDOWS}
  s:= path + UTF8ToSys(ListView1.Selected.Caption);
  {$ELSE}
  s:= path + ListView1.Selected.Caption;
  {$ENDIF}
  if FindFirst(s, faAnyFile, srNode) = 0 then
```

```
  if (srNode.Attr and faDirectory) <> 0 then
  begin
    DirName:= ListView1.Selected.Caption;
    Show_ListView_Dir(DirName);
    Show_ListView_Files(DirName);
    exit;
  end;
  if (srNode.Attr and faHidden) = 0 then
  begin
    progr:= pchar(s);
    {$IFDEF UNIX}
      shell(progr);
    {$ELSE}
      WinExec(progr, 1);
    {$ENDIF}
  end;
end;
```

В Linux при таком способе запуска исполняемого файла запустить консольное приложение не удастся, поскольку консольное приложение должно запускаться в консоли или терминале. Кроме того, существует более эффективный способ запуска внешних приложений, подходящий как для Linux, так и для Windows. Это использование процессов и потоков (нитей). Не вдаваясь в подробности, отмечу, что при запуске программы порождается процесс. Если в программе запускается другая программа, то порождается дочерний процесс.

Создать и запустить дочерний процесс можно следующим образом:

```
var
  AProcess: TProcess;
begin
```

```
    AProcess:= TProcess.Create(nil);
    AProcess.CommandLine:= 'Имя исполняемого файла';
    AProcess.Execute;
    AProcess.Free;
 end;
```

Перепишем обработчик `OnDblClick`:

```
procedure TForm1.ListView1DblClick(Sender: TObject);
var
  s: string;
  AProcess: TProcess;
  srNode: TSearchRec;
  DirName: string;
begin
  if ListView1.Selected = nil then exit;
  {$IFDEF WINDOWS}
  s:= path + UTF8ToSys(ListView1.Selected.Caption);
  {$ELSE}
  s:= path + ListView1.Selected.Caption;
  {$ENDIF}
  if FindFirst(s, faAnyFile, srNode) = 0 then
  if (srNode.Attr and faDirectory) <> 0 then
  begin
    DirName:= ListView1.Selected.Caption;
    Show_ListView_Dir(DirName);
    Show_ListView_Files(DirName);
    exit;
  end;
```

```
  if (srNode.Attr and faHidden) = 0 then
  begin
    AProcess:= TProcess.Create(nil);
    {$IFDEF UNIX}
      if MessageDlg('Открывать в терминале?', mtCustom,
                    [mbYes, mbNO], 0) = mrYes then
      s:= 'xterm -T ''Lazarus Run Output''' +
          ' -e $(TargetCmdLine)' + s;
    {$ENDIF}
    AProcess.CommandLine:= s;
    AProcess.Execute;
    AProcess.Free;
  end;
end;
```

Окинем еще раз взглядом код всей нашей программы. Мы видим, что в ней имеются пять похожих процедур. Конечно, они не совсем похожи! Каждая процедура решает свою задачу. Я имею в виду, что в них имеются повторяющиеся участки кода. Хотелось бы написать одну общую процедуру. Ясно, что эта "универсальная" процедура будет довольно-таки запутанной из-за необходимости производить многочисленные проверки. В таких случаях перед программистом всегда встает дилемма, что выбрать – простой и ясный код, но с дублированием или код "с наворотами".

Все же большинство профессиональных программистов предпочтут более простой и прозрачный код, несмотря на то, что размер исполняемого файла может оказаться больше. Особенно, если планируется программу в дальнейшем модифицировать. Ведь спустя некоторое время разбираться даже в своей собственной программе бывает достаточно тяжело.

Однако мы все же попробуем объединить эти пять процедур в одну. Назо-

вем эту процедуру ShowExpandNode и введем два дополнительных параметра ShowMode: integer и IsinTListView: boolean.

Пусть ShowMode = 0, если необходимо вывести каталоги в TTreeView.

ShowMode = 1, если необходимо вывести только каталоги в TListView, но узел берется из TTreeView.

ShowMode = 2, если необходимо вывести только файлы в TListView из каталога выбранного в TTreeView.

IsinTListView = false, если мы находимся в TTreeView.

IsinTListView = true, если мы находимся в TListView.

Окончательно код программы будет выглядеть следующим образом:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, FileUtil, LResources, Forms,
  Controls, Graphics, Dialogs, ComCtrls, StdCtrls,
  ExtCtrls, LCLType, Process,

  {$IFDEF UNIX}
    unix;
  {$ELSE}
    Windows;
  {$ENDIF}
type

  { TForm1 }

  TForm1 = class(TForm)
```

```
    ImageList1: TImageList;
    ListView1: TListView;
    Memo1: TMemo;
    TreeView1: TTreeView;
    procedure FormCreate(Sender: TObject);
    procedure ListView1DblClick(Sender: TObject);
    procedure TreeView1Change(Sender: TObject; Node:
                              TTreeNode);
    procedure TreeView1Expanding(Sender: TObject;
          Node: TTreeNode; var AllowExpansion: Boolean);
    function Real_Directory(sname: string): boolean;
    procedure ShowExpandNode(ParentNode: TTreeNode;
                     DirName: string; ShowMode: integer;
                     IsinTListView: boolean);
    function GetPath(ParentNode: TTreeNode): string;
    function GetPath_1(DirName: string;
                       p: string): string;
    {$IFDEF WINDOWS}
    function Find_Logical_Disks(): boolean;
    procedure SetAllDirectories;
    {$ENDIF}
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
  path, oldpath: string;
```

```
implementation

{ TForm1 }

function TForm1.Real_Directory(sname: string): boolean;
begin
  result:= (sname <> '.') and (sname <> '..');
end;

{$IFDEF WINDOWS}
// Эти функции нужны только для Windows
function TForm1.Find_Logical_Disks(): boolean;
const
  Hard_Disk = 3;  // рассматриваем только жесткие диски
var
  size: LongWord;
  Drives: array[0..128] of char;
  pDrive: PChar;
  s: string;
begin
  size:= GetLogicalDriveStrings(SizeOf(Drives), Drives);
  if size = 0 then
  begin
    Result:= false;
    exit;
  end;
  if size > SizeOf(Drives) then
    raise Exception.Create(SysErrorMessage(ERROR_OUTOFMEMORY));
  pDrive:= Drives; // устанавливаем указатель на Drives
```

```
  while pDrive^ <> #0 do
  begin
    // если тип устройства жесткий диск
    if GetDriveType(pDrive) = Hard_Disk then
    begin
      s:= pDrive;
      s:= Copy(s, 1, 2);  // берем только имя раздела и двоеточие
      Memo1.Lines.Add(s); // добавляем имя раздела в Memo1
    end;
    inc(pDrive, 4);
  end;
end;

procedure TForm1.SetAllDirectories;
var
  i: integer;
  Node: TTreeNode;
begin
  TreeView1.BeginUpdate;
  for i:= 0 to Memo1.Lines.Count - 1 do
  begin
    Node:= TreeView1.Items.AddChild(nil, Memo1.Lines[i]);
    Node.ImageIndex:= 0;
    Node.SelectedIndex:= 0;
    Node.HasChildren:= true;
  end;
  TreeView1.EndUpdate;
end;
{$ENDIF}
```

```
function TForm1.GetPath(ParentNode: TTreeNode): string;
var
  Node: TTreeNode;
begin
  Node:= ParentNode;
  path:= '';
  repeat
    {$IFDEF WINDOWS}
    path:= UTF8ToSys(Node.Text) + '\' + path;
    {$ELSE}
    path:= '/' + Node.Text + '/' + path;
    {$ENDIF}
    Node:= Node.Parent;
  until Node = nil;
  Result:= path;
end;

function TForm1.GetPath_1(DirName: string;
                          p: string): string;
begin
  {$IFDEF WINDOWS}
  p:= p + UTF8ToSys(DirName) + '\' ;
  {$ELSE}
  p:= '/' + p + DirName + '/';
  {$ENDIF}
  Result:= p;
end;

procedure TForm1.FormCreate(Sender: TObject);
```

```
var
  Node: TTreeNode;
  srNode, srChild: TSearchRec;
  searchMask: string;
  SetDirWin: boolean = false;
begin
  Memo1.Clear;
  Memo1.Visible:= false;
  TreeView1.Images:= ImageList1;
  TreeView1.ExpandSignType:= tvestPlusMinus;
  TreeView1.BeginUpdate;
    {$IFDEF WINDOWS}
       // Определение логических дисков
      if Find_Logical_Disks() then
      begin
        SetAllDirectories;
        SetDirWin:= true;
      end
      else
      {Если произошла ошибка в функции Find_Logical_Disks(),
       то выбираем корневой каталог текущего диска}
        SetCurrentDir('\');
    {$ELSE}
      SetCurrentDir('/'); // корневой каталог в Linux
    {$ENDIF}
    if not SetDirWin then
    begin
      path:= GetCurrentDir;
      if FindFirst(path + '*', faDirectory, srNode) = 0
```

```
then
begin
  repeat
    // Показываем только каталоги
    if (srNode.Attr and faDirectory <> 0)
    and (Real_Directory(srNode.Name)) then
    begin
    Node:= TreeView1.Items.AddChild(nil,
           SysToUTF8(srNode.Name));
    Node.ImageIndex:= 0;
    Node.SelectedIndex:= 0;
    {$IFDEF WINDOWS}
      searchMask:= path + srNode.Name + '\*';
      {$ELSE}
        searchMask:= path + srNode.Name + '/*';
      {$ENDIF}
      if FindFirst(searchMask, faDirectory,
         srChild) = 0 then
      repeat
        if (srChild.Attr and faDirectory <> 0) and
            Real_Directory(srChild.Name)
        then Node.HasChildren:= true;
      until (FindNext(srChild) <> 0) or
             Node.HasChildren;
      // Освобождение занятых ресурсов
      SysUtils.FindClose(srChild);
    end;
  until FindNext(srNode) <> 0;
  // Освобождение занятых ресурсов
```

```
        SysUtils.FindClose(srNode);
      end;
    end;
  TreeView1.EndUpdate;
  // установка свойств ListView1
  with ListView1 do
  begin
    Align:= alClient;
    ViewStyle:= vsReport;
    Columns.Add;
    Columns.Items[0].Width:= 425;
    ScrollBars:= ssAutoBoth;
    SmallImages:= ImageList1;
  end;
end;

procedure TForm1.ShowExpandNode(ParentNode: TTreeNode;
                      DirName: string; ShowMode: integer;
                                 IsinTListView: boolean);
var
  srNode, srChild: TSearchRec;
  Node: TTreeNode;
  SearchMode: LongInt;
  ListItem: TListItem;
  searchMask: string;
begin
  if not IsinTListView then
    path:= GetPath(ParentNode)
  else
```

```
begin
  if ShowMode = 1 then
  begin
    oldpath:= path;
    ListView1.Clear;
  end;
  path:= GetPath_1(DirName, oldpath);
end;
if ShowMode = 2 then
  SearchMode:= faAnyFile
else
  SearchMode:= faDirectory;
if FindFirst(path + '*', SearchMode, srNode) = 0 then
repeat
if ((srNode.Attr and faDirectory <> 0) and
          Real_Directory(srNode.Name)) or
    ((srNode.Attr and faDirectory = 0) and
                        (ShowMode = 2)) then
begin
  if ShowMode <> 2 then
  if not IsinTListView then
     Node:= TreeView1.Items.AddChild(ParentNode,
            SysToUTF8(srNode.Name));
  case ShowMode of
  0: begin
       Node.ImageIndex:= 0;
       Node.SelectedIndex:= 0;
     end;
  1: begin
```

```
        ListItem:= ListView1.Items.Add;
        ListItem.Caption:= SysToUTF8(srNode.Name);
        ListItem.ImageIndex:= 0
      end;
    2: if srNode.Attr and faDirectory = 0 then
      begin
        if srNode.Attr and faHidden = 0 then
        begin
          ListItem:= ListView1.Items.Add;
          ListItem.Caption:= SysToUTF8(srNode.Name);;
          ListItem.ImageIndex:= 1;
        end;
      end;
    end;
    if not IsinTListView then
    begin
      {$IFDEF WINDOWS}
        searchMask:= path + srNode.Name + '\*';
      {$ELSE}
        searchMask:= path + srNode.Name + '/*';
      {$ENDIF}
      if FindFirst(searchMask, faDirectory, srChild) = 0
      then
      repeat
        if (srChild.Attr and faDirectory <> 0) and
                  Real_Directory(srChild.Name)
        then Node.HasChildren:= true;
      until (FindNext(srChild) <> 0) or Node.HasChildren;
      SysUtils.FindClose(srChild);
```

```
    end;
  end;
  until FindNext(srNode) <> 0;
  SysUtils.FindClose(srNode);
end;

procedure TForm1.ListView1DblClick(Sender: TObject);
var
  s: string;
  AProcess: TProcess;
  srNode: TSearchRec;
  DirName: string;
  IsinTListView: boolean;
begin
  if ListView1.Selected = nil then exit;
  {$IFDEF WINDOWS}
  s:= path + UTF8ToSys(ListView1.Selected.Caption);
  {$ELSE}
  s:= path + ListView1.Selected.Caption;
  {$ENDIF}
  if FindFirst(s, faAnyFile, srNode) = 0 then
  if (srNode.Attr and faDirectory) <> 0 then
  begin
    DirName:= ListView1.Selected.Caption;
    IsinTListView:= true;
    ShowExpandNode(nil, DirName, 1, IsinTListView);
    ShowExpandNode(nil, DirName, 2, IsinTListView);
    SysUtils.FindClose(srNode);
    exit;
```

```
  end;
  if (srNode.Attr and faHidden) = 0 then
  begin
    AProcess:= TProcess.Create(nil);
    {$IFDEF UNIX}
      if MessageDlg('Открывать в терминале?', mtCustom,
         [mbYes, mbNO], 0) = mrYes then
      s:= 'xterm -T ''Lazarus Run Output''' +
          ' -e $(TargetCmdLine)' + s;
    {$ENDIF}
    AProcess.CommandLine:= s;
    AProcess.Execute;
    AProcess.Free;
  end;
end;

procedure TForm1.TreeView1Change(Sender: TObject;
                                 Node: TTreeNode);
var
  IsinTListView: boolean;
begin
  if Node = nil then exit;
  TreeView1.BeginUpdate;
  ListView1.BeginUpdate;
  ListView1.Clear;
  Node.DeleteChildren;
  IsinTListView:= false;
  ShowExpandNode(Node, '', 1, IsinTListView);
  ShowExpandNode(Node, '', 2, IsinTListView);
```

```
  Node.Expanded:= true;
  ListView1.EndUpdate;
  TreeView1.EndUpdate;
end;

procedure TForm1.TreeView1Expanding(Sender: TObject;
      Node: TTreeNode; var AllowExpansion: Boolean);
var
  IsinTListView: boolean;
begin
  TreeView1.Items.BeginUpdate;
  Node.DeleteChildren;
  IsinTListView:= false;
  ShowExpandNode(Node, '', 0, IsinTListView);
  TreeView1.Items.EndUpdate;
end;
initialization
  {$I unit1.lrs}
end.
```

Дальнейшее наполнение функциональностью нашего Проводника предоставляю вам, уважаемый читатель. В частности, попробуйте сделать следующее – при двойном клике на какой-нибудь файл необходимо запустить связанное с этим файлом приложение. Например, при двойном щелчке на файл *.doc должен запускаться текстовый редактор `Word` и в нем открываться этот файл.

Ну и все те функции, которыми должен обладать любой файловый менеджер, т.е. реализовать операции копирования, перемещения и удаления файлов и каталогов и пр.

Можете "замахнуться" на создание файлового менеджера типа

TotalCommander или Krusader или Double Commander (который, кстати, разрабатывается на Lazarus). Дело не в том, что вы будете "изобретать велосипед". Это будет отличнейшей тренировкой для вас!

А я, с вашего позволения, перейду к рассмотрению других вопросов.

9.2.11. Организация меню. Механизм действий - Actions

9.2.11.1. Компонент TMainMenu

Рассмотрим способы создания и организации меню. В меню пользователь выбирает не данные, а возможные действия. Таким образом, меню это специальный механизм, позволяющий пользователю внутри вашего приложения выбрать какие-то действия, например, открыть файл, сохранить файл, установить какие-то параметры и т.д.

Для создания главного меню используется компонент TMainMenu (вкладка Standard). Компонент не визуальный. Пункты меню содержатся в свойстве Items. Чтобы начать формирование пунктов меню достаточно дважды щелкнуть по компоненту на форме или нажать на кнопку с многоточием в свойстве Items компонента в инспекторе объектов. Откроется специальный редактор меню, рис. 9.76.

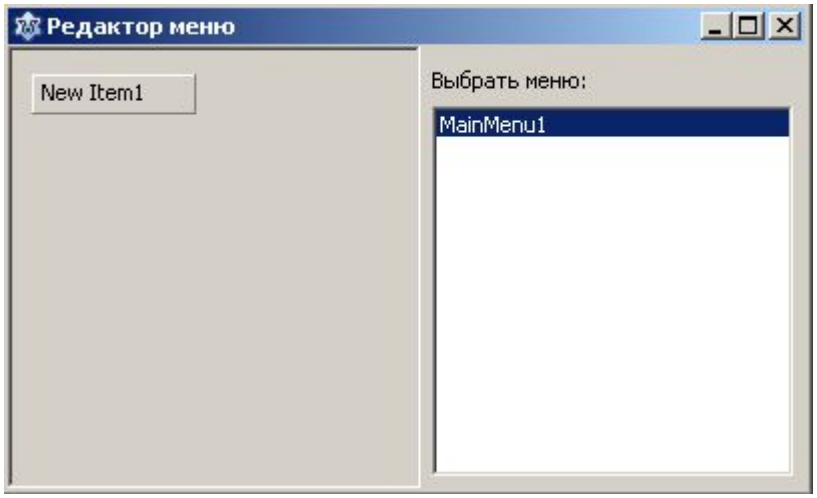


Рис. 9.76. Редактор меню

В инспекторе объектов в свойство `Caption` введите имя пункта меню. Чтобы создать следующий пункт меню или подменю, установите курсор на текущий элемент меню и нажмите на правую клавишу мыши, рис. 9.77. Выберите нужное действие, например, для создания подменю нажмите "Создать подменю", рис. 9.78.

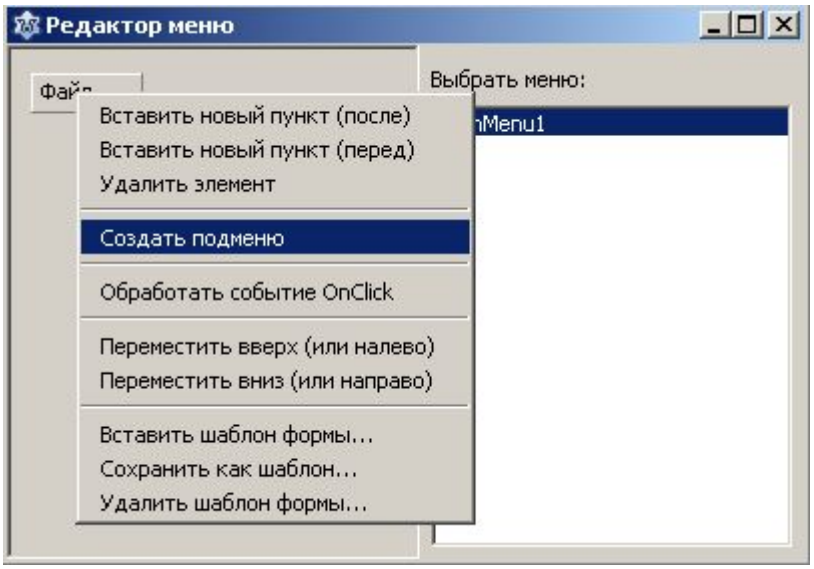


Рис. 9.77. Контекстное меню редактора

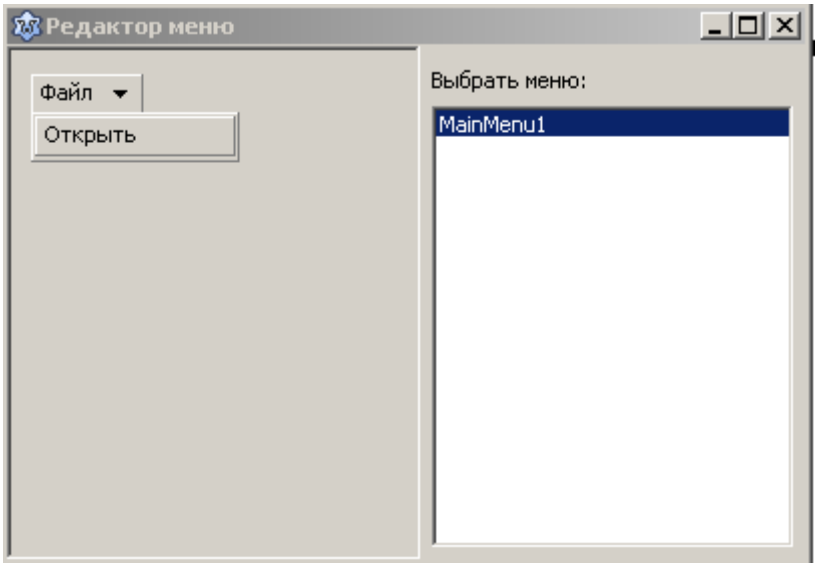


Рис. 9.78. Пример создания подменю

Lazarus предоставляет возможность при создании меню применить шаблоны меню. Имеется три стандартных шаблона для меню "Файл", "Правка" и "Справка", рис. 9.79.

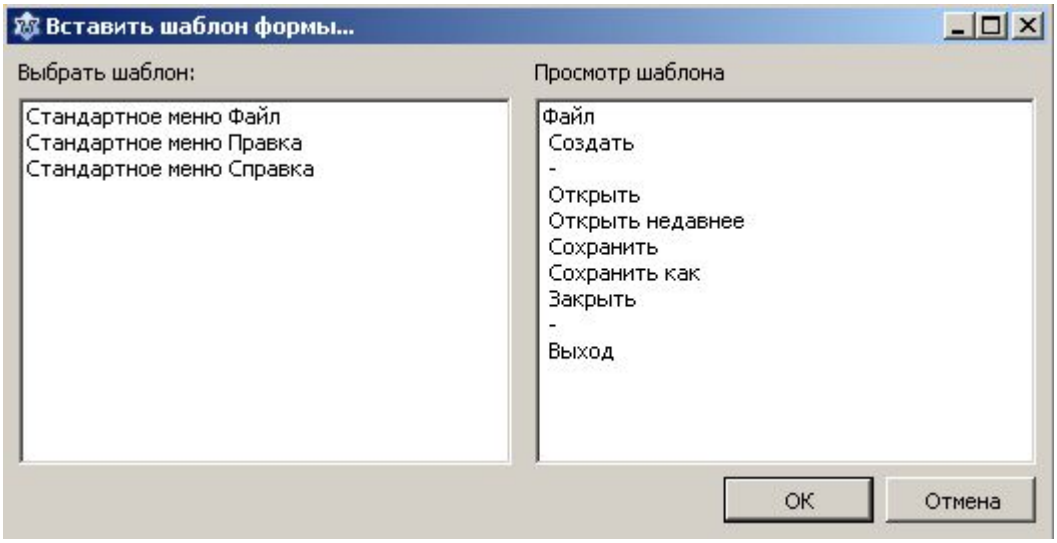


Рис. 9.79. Шаблоны меню

При создании меню имеется возможность вставки рядом с текстом пункта меню пиктограммы. Для этого поместите на форму компонент `TImageList`, заполните его соответствующими рисунками. Затем в `TMainMenu` в свойстве `Images` укажите имя `TImageList` в программе. А при создании подпункта меню в свойстве `ImageIndex` укажите индекс соответствующего изображения.

Также вы можете добавить горячие клавиши для данного пункта меню. Для этого служит свойство `Shortcut`. Вы можете прямо ввести нужное сочетание клавиш в поле ввода этого свойства или нажав на кнопку с многоточием, выбрать нужное сочетание из появившегося редактора выбора сочетаний клавиш, рис. 9.80.

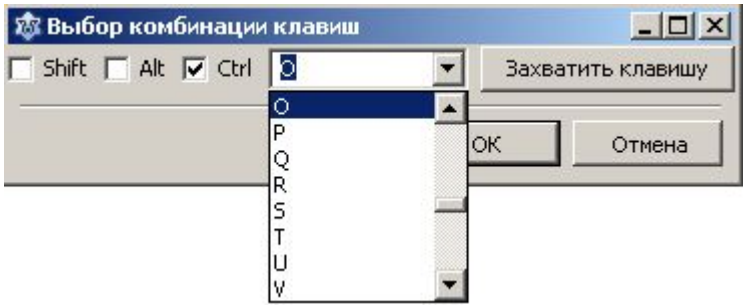


Рис. 8.80. Установка сочетаний клавиш

Для создания разделительной линии между пунктами меню достаточно создать новый элемент и в свойство `Caption` ввести знак "-".

Попробуйте реализовать пункт меню "Файл" Lazarus. Рисунки вы можете найти в папке установки Lazarus (подпапка `Images`), рис. 9.81.

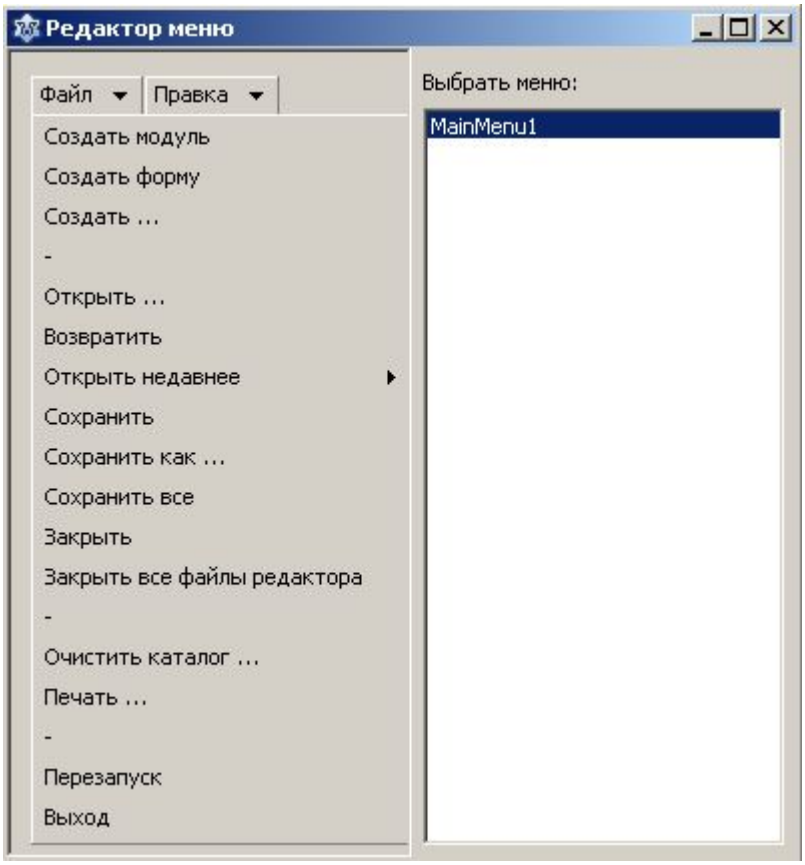


Рис. 9.81. Пример создания меню

При этом вместо безликих имен пунктов меню (в программе) типа

MenuItem1, MenuItem2 и т.д. желательно присваивать им осмысленные имена, например MCreateModule, MCreateForm.

Для того чтобы реализовать действие пункта меню необходимо написать соответствующий обработчик события OnClick. Находясь в редакторе меню, достаточно просто дважды щелкнуть по этому пункту.

Еще с четвертой главы за мной тянется один должок. В 4.3.3. мы с вами рассматривали метод наименьших квадратов, а программу мы так и не написали. Но, как говорится, всему свое время. И вот это время наступило. Приступим к реализации этого метода.

Но сначала вкратце рассмотрим компонент TChart, который мы будем использовать в программе для вывода графиков. Основным свойством компонента является Series – наборы данных (серии), на основе которых и строятся графики или диаграммы. Если дважды щелкнуть по компоненту, размещенному на форме, то мы попадем в редактор серий, рис. 9.82.

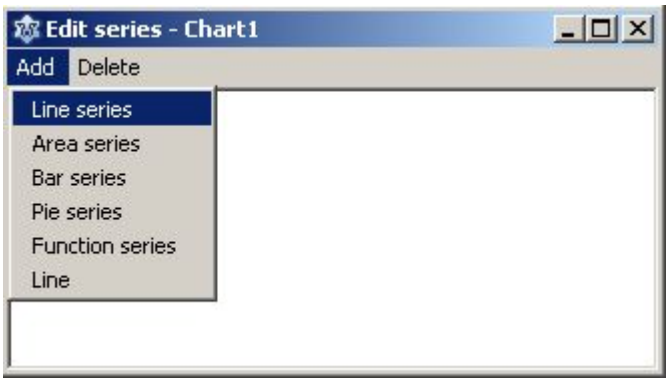


Рис. 9.82. Редактор серий

Существуют разные типы Series для построения разных типов графиков или диаграмм, например Line series – для построения линий (графиков), Pie series – для построения круговых диаграмм и т.д. Выберите тип серии Line series. В инспекторе объектов появится объект ChartLineSeries1 типа TLineSeries.

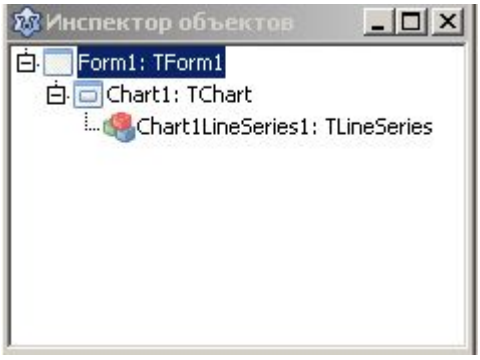


Рис. 9.83. Создание объекта типа TLineSeries

Создать программно объект типа `TLineSeries` можно оператором

```
Chart1LineSeries1:= TLineSeries.Create(Chart1);
```

Для добавления серии в `TChart` существует метод

```
procedure AddSeries(ASeries: TBasicChartSeries);
```

Добавление точек в объект типа `TLineSeries` производится с помощью функции

```
function AddXY(X, Y: Double): Integer;
```

Свойство `ShowPoints: boolean` позволяет показывать или не показывать точки на графике.

Свойство `SeriesColor: TColor` позволяет указать цвет линии.

Чтобы вывести график функции, например, `sin(x)` можно написать следующую процедуру:

```
procedure Plot_sin;
var
  i, n: integer;
  Chart1LineSeries1: TLineSeries;
```

```
begin
  n:= 100;
  Chart1LineSeries1:= TLineSeries.Create(Chart1);
  Chart1LineSeries1.SeriesColor:= clRed;
  Chart1LineSeries1.ShowPoints:= false;
  Chart1.AddSeries(Chart1LineSeries1);
  Chart1.Title.Visible:= true;
  Chart1.Title.Text.Text:= 'График функции sin(x)';
  for i:= 0 to n - 1 do
    Chart1LineSeries1.AddXY(i*Pi*0.02,
                            sin(i*Pi*0.02));
end;
```

Итак, давайте начнем реализацию метода наименьших квадратов. Создайте новый проект, поместите на форму компоненты `TMainMenu`, `TOpenDialog` и `TChart`, как показано на рис. 9.84. Меню программы должно состоять из следующих пунктов:

- Файл
- Определение коэффициентов полинома
- Графики

В свою очередь, меню "Файл" должен состоять из подпунктов "Открыть" и "Выход". Меню "Определение коэффициентов полинома" должна содержать пункт "Вычислить". И меню "Графики" должен содержать пункты "График экспериментальных точек" и "Подобранная кривая", рис.9.85.

Присвойте имена меню в программе как показано на рис. 9.86.

Вставим в пункты меню рядом с их текстами пиктограммы. Для этого поместите на форму компонент `TImageList` и заполните его подходящими значками. В свойстве `Images MainMenu1` укажите имя `TImageList`. Затем для каждого пункта меню укажите соответствующие индексы в свойствах `ImageIndex`.

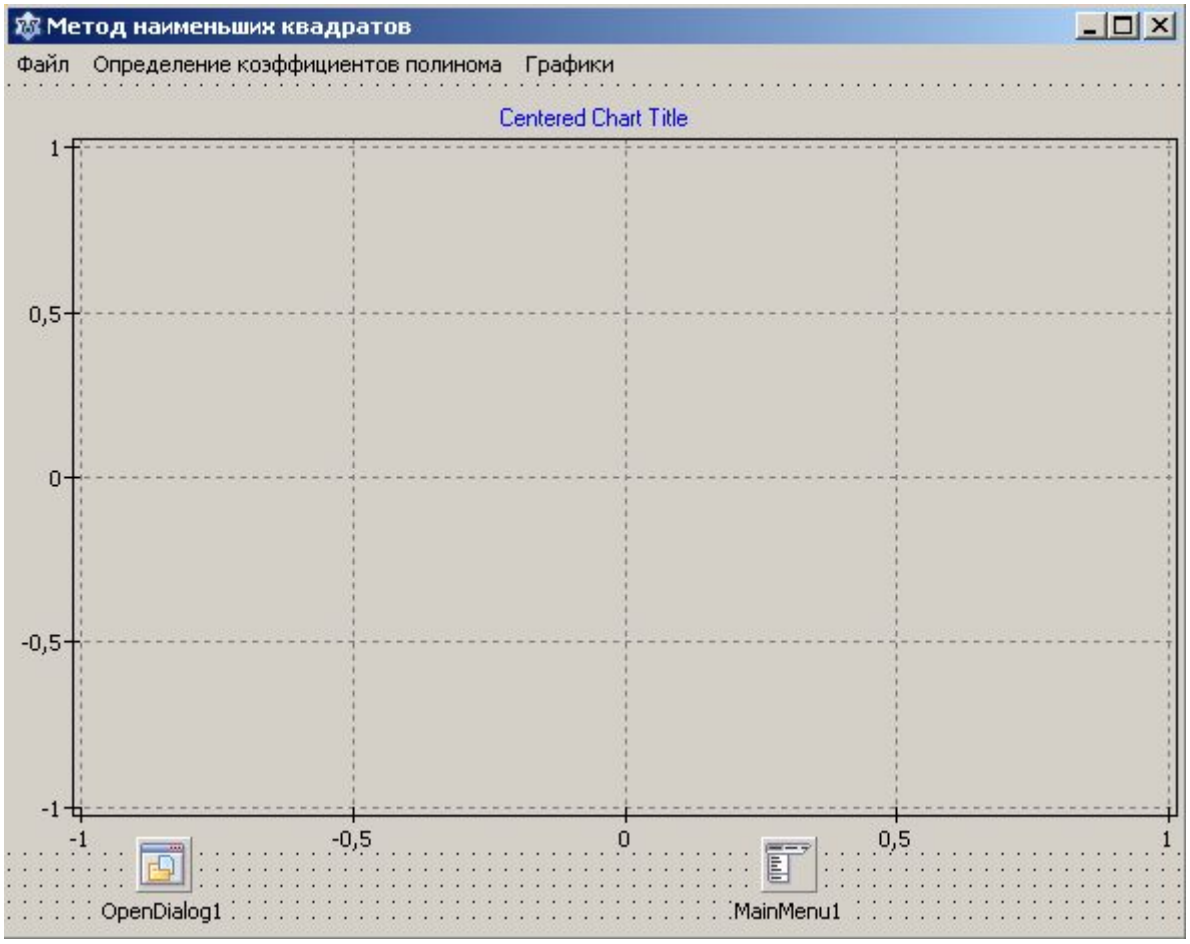


Рис. 9.84. Форма приложения

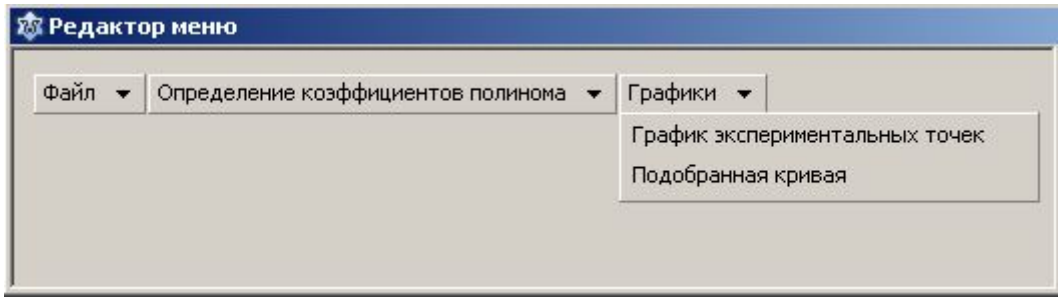


Рис. 9.85. Создание меню программы с помощью редактора меню

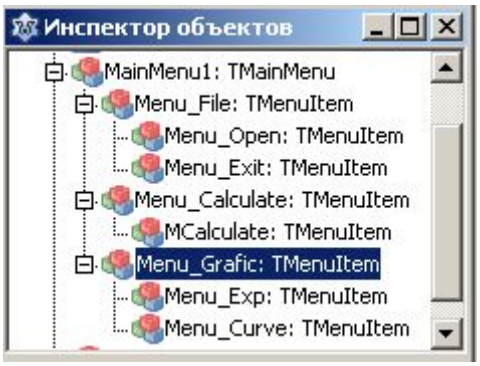


Рис. 9.86. Имена меню в программе

Код программы:

```
unit unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,   SysUtils,   LResources,   Forms,   Controls,
  Graphics,   Dialogs,   ExtCtrls,   TAGraph,   TASeries,
  Buttons, StdCtrls, Menus, FileUtil;
type
  { TForm1 }
  TForm1 = class(TForm)
    Chart1: TChart;
    MainMenu1: TMainMenu;
    MCalculate: TMenuItem;
    Menu_File: TMenuItem;
    Menu_Calculate: TMenuItem;
    Menu_Grafic: TMenuItem;
    Menu_Open: TMenuItem;
    Menu_Exit: TMenuItem;
    Menu_Exp: TMenuItem;
    Menu_Curve: TMenuItem;
    OpenDialog1: TOpenDialog;
    procedure FormCreate(Sender: TObject);
    procedure Menu_ExpClick(Sender: TObject);
    procedure Menu_CurveClick(Sender: TObject);
    procedure Menu_OpenClick(Sender: TObject);
    procedure MCalculateClick(Sender: TObject);

  private
```

```
    { private declarations }
  public
    { public declarations }
  end;

// Процедура решения СЛАУ методом Гаусса
procedure gauss(vector: array of real; b: array of real;
                var x: array of real; n: byte;
                var solve: byte);
// n - размерность системы,
// solve=0, если решение единственное,
// solve=1, если система не имеет решения,
// solve=2, если система имеет бесконечное количество решений,

// Функция, подбираемая методом
// наименьших квадратов
function fx(t: real): real;

// Функция возведения в степень
function stepen( x: real; n: byte): real;

var
  Form1: TForm1;
  n: byte;
  x1, y1: real;
  x, y, z: array of real;
  Fname: string;

implementation
```

```
function fx(t: real): real;
begin
  Result:= z[0] + z[1]*t + z[2]*t*t +
           z[3]*t*t*t + z[4]*sqr(sqr(t));
end;

function stepen( x: real; n: byte): real;
var
  i: integer;
begin
  Result:= 1;
  for i:= 1 to n do
  Result:= Result*x;
end;
// Реализация метода Гаусса
procedure Gauss(vector: array of real; b: array of real;
                var x: array of real; n: byte;
                var solve: byte);
var
 a: array of array of real; { матрица коэффициентов системы,
 двумерный динамический массив}
 i, j, k, p, r: integer;
 m, s, t: real;
begin
 SetLength(a, n, n); // установка фактического размера массива

 { Преобразование одномерного массива в двумерный }
 k:=0;
 for i:=0 to n-1 do
```

```
  for j:=0 to n-1 do
  begin
    a[i, j]:= vector[k];
    k:=k+1;
  end;
for k:=0 to n-2 do
begin
  for i:=k+1 to n-1 do
  begin
    if (a[k, k]=0) then
    begin
      { перестановка уравнений}
      p:=k; // в алгоритме используется буква l, но она похожа на 1
            // Поэтому используем идентификатор p
      for r:=i to n-1 do
      begin
        if abs(a[r, k]) > abs(a[p, k]) then p:=r;
      end;
      if p<>k then
      begin
        for j:= k to n-1 do
        begin
          t:=a[k, j];
          a[k, j]:=a[p, j];
          a[p, j]:=t;
        end;
        t:=b[k];
        b[k]:=b[p];
        b[p]:=t;
```

```
      end;
    end; // конец блока перестановки уравнений
    m:=a[i,k]/a[k, k];
    a[i, k]:=0;
    for j:=k+1 to n-1 do
    begin
      a[i, j]:=a[i, j]-m*a[k, j];
    end;
    b[i]:= b[i]-m*b[k];
  end;
end;
{Проверка существования решения}
if a[n-1,n-1] <> 0  then
begin
  x[n-1]:=b[n-1]/a[n-1,n-1];
  for i:=n-2 downto 0 do
  begin
    s:=0;
    for j:=i+1 to n-1 do
    begin
      s:=s-a[i, j]*x[j];
    end;
    x[i]:=(b[i] + s)/a[i, i];
  end;
  solve:= 0;
end
else
if b[n-1] = 0 then
begin
```

```
    MessageDlg('Система имеет бесконечное ' +
               'количество решений', mtInformation,[mbOK], 0);
    solve:= 2;
  end
  else
  begin
    MessageDlg('Система не имеет решений',
                   mtInformation,[mbOK], 0);
    solve:= 1;
  end;
  { освобождение памяти,
    распределенной для динамического массива }
  a:=nil;
end;

{ TForm1 }

procedure TForm1.FormCreate(Sender: TObject);
begin
  Chart1.Title.Text.Text:='Метод наименьших квадратов';
  MCalculate.Enabled:= false;
  Menu_Exp.Enabled:= false;
  Menu_Curve.Enabled:= false;
end;

procedure TForm1.Menu_ExpClick(Sender: TObject);
{Процедура вывода графика экспериментальных
 данных по точкам}
var
```

```
  i: integer;
  gr1: TLineSeries;
begin
  Chart1.Visible:= true;
  gr1:= TLineSeries.Create(Chart1);
  Chart1.AddSeries(gr1);
  for i:= 0 to n - 1 do
  gr1.AddXY(x[i], y[i]);
end;

procedure TForm1.Menu_CurveClick(Sender: TObject);
{Процедура вывода совмещенных графиков
 экспериментальных данных по точкам и
 подобранной методом наименьших квадратов
 кривой, наилучшим образом приближающейся
 к экспериментальным данным}
var
  i: integer;
  gr1, gr2: TLineSeries;
begin
  Chart1.Visible:= true;
  gr1:= TLineSeries.Create(Chart1);
  gr1.ShowPoints := true; // график с точками
  gr1.ShowLines := false; // не соединять точки линиями
  Chart1.AddSeries(gr1);
  gr2:= TLineSeries.Create(Chart1);
  gr2.ShowLines := true;
  Chart1.AddSeries(gr2);
  for i:= 0 to n - 1 do
```

```
  gr1.AddXY(x[i], y[i]);
  for i:= 0 to n - 1 do
  gr2.AddXY(x[i], fx(x[i]));
end;

procedure TForm1.Menu_OpenClick(Sender: TObject);
// процедура выбора, открытия  и чтения файла данных
var
  f: TextFile;
  i: integer;
begin
  if OpenDialog1.Execute then
    Fname:= OpenDialog1.FileName
  else exit;
  Fname:= UTF8ToSys(Fname);// преобразование в системную кодировку
  AssignFile(f, Fname);
  Reset(f);
  // отключение контроля ошибок ввода/вывода
  {$I-}
  // чтение количества экспериментальных точек
  Readln(f, n);
  if IOResult <> 0 then
  begin
    ShowMessage('Ошибка при чтении из файла!');
    exit;
  end;
  // распределение памяти под динамические массивы
  SetLength(x, n);
  SetLength(y, n);
```

```
  for i:= 0 to n - 1 do
  begin
    read(f, x[i]);
    if IOResult <> 0 then
    begin
      ShowMessage('Ошибка при чтении из файла!');
      exit;
    end;
  end;
  for i:= 0  to n - 1 do
  begin
    read(f, y[i]);
    if IOResult <> 0 then
    begin
      ShowMessage('Ошибка при чтении из файла!');
      exit;
    end;
  end;
  {$I+}
  CloseFile(f);
  MCalculate.Enabled:= true;
end;
procedure TForm1.MCalculateClick(Sender: TObject);
var
  i, j, k, l: integer;
  b, vector: array of real;
  s: real;
  solve: byte;
begin
```

```
  SetLength(z, 5);
  SetLength(b, 5);
  SetLength(vector, 25);
  j:= 0;
  for k:= 0 to 4 do
  for l:= 0 to 4 do
  begin
    s:= 0;
    for i:= 0 to n - 1 do
    s:= s + stepen(x[i], k + l);
    vector[j]:= s;
    j:= j+1;
  end;
  for k:= 0 to 4 do
  begin
    s:= 0;
    for i:= 0 to n - 1 do
    s:= s + y[i]*stepen(x[i], k);
    b[k]:= s;
  end;
  // решение СЛАУ
  gauss(vector, b, z, 5, solve);
  if solve = 0 then
  begin
    Menu_Exp.Enabled:= true;
    Menu_Curve.Enabled:= true;
  end;
end;
initialization
```

```
  {$I unit1.lrs}
end.
```

Для того чтобы понять программу вспомните реализацию метода Гаусса решения системы линейных алгебраических уравнений, причем в варианте с применением динамических массивов. В той программе матрицу коэффициентов системы мы сначала преобразовывали в одномерный динамический массив, поскольку передавать в качестве формального параметра в функцию или процедуру можно только одномерный динамический массив.

Поэтому в программе метода наименьших квадратов в процедуре `MCalculateClick` формируется одномерный динамический массив `vector`, который на самом деле является матрицей коэффициентов системы. Его мы и передаем (вместе с вектором свободных членов) в процедуру `Gauss`.

Перед пуском программы подготовьте в любом текстовом редакторе файл следующего вида:

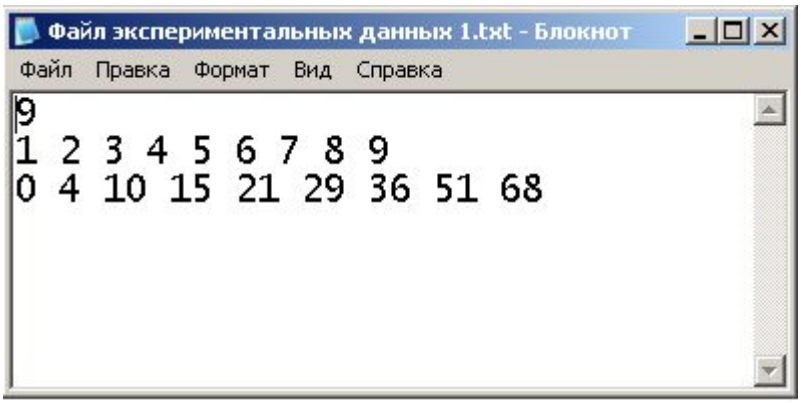


Рис. 9.87. Файл данных

Файл имеет следующую структуру. Первая строка содержит количество точек. Вторая строка содержит значения Z_i, третья строка содержит значения P_i.

Обратите внимание, что до загрузки файла с экспериментальными данны-

ми пункты меню "Вычислить", "График экспериментальных точек" и "Подобранная кривая" необходимо сделать недоступными. После загрузки файла, пункт меню "Вычислить" надо сделать доступным, а пункты "График экспериментальных точек" и "Подобранная кривая" по-прежнему недоступными. Только после того, как будут выполнены необходимые вычисления, пункты "График экспериментальных точек" и "Подобранная кривая" становятся доступными.

9.2.11.2. Компонент `TToolBar`

Если в вашей программе имеется достаточно большое и разветвленное меню, то почти всегда имеет смысл добавить в программу так называемые кнопки быстрого доступа. Эти кнопки будут дублировать наиболее часто используемые пункты меню. Например, если в программе открывается много файлов, то одним из удобств, предоставляемых пользователю, будет наличие кнопки для быстрого доступа к этой операции. В сколько-нибудь больших программах обычно требуется разместить несколько кнопок быстрого доступа. Однако если они разбросаны по разным местам, то это затрудняет работу пользователя. Чаще всего кнопки быстрого доступа объединяют в виде инструментальной панели.

Компонент `TToolBar` представляет собой как раз панель для размещения кнопок быстрого доступа. В принципе в этот компонент можно вставлять любые компоненты, но поскольку он предназначен, прежде всего, для создания инструментальных панелей, в него и помещают кнопки. Причем `TToolBar` содержит в себе специальный компонент `TToolButton` – инструментальная кнопка. Для того чтобы поместить инструментальную кнопку в `TToolBar` нажмите правую клавишу мыши и выберите пункт `NewButton`, рис. 9.88.

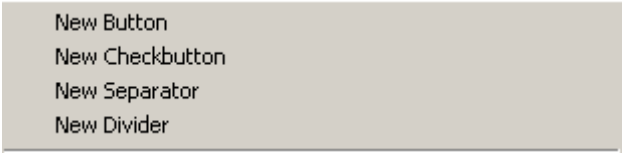


Рис. 9.88. Добавление новой кнопки

Изображение на кнопке задается свойством `ImageIndex` и определяет индекс изображения в компоненте `TImageList`. В момент выполнения приложения некоторые кнопки на инструментальной панели по тем или иным причинам могут быть недоступны, например, до выполнения определенных условий. Программист может для этих кнопок указать другие изображения. Для этого нужно поместить на форму еще один компонент `TImageList` и в свойстве `DisabledImages` указать его имя. Точно так же свойство `HotImages` позволяет указать контейнер изображений для кнопок, когда над ними проходит указатель мыши. На практике, чаще всего, предпочитают все же иметь один контейнер изображений для всех случаев.

`New Checkbutton` задает разновидность кнопки, которая после нажатия на нее остается в нажатом состоянии. Это так называемая западающая кнопка. Чтобы отжать ее необходимо нажать на нее еще раз.

`New Separator` задает разделитель (сепаратор). Позволяет разделить группы кнопок по функциональному назначению. Во время выполнения приложения на этом месте появляются две вертикальные линии.

`New Divider` тоже разделитель, но в момент выполнения появляется одна вертикальная линия.

Этим четырем разновидностям кнопок соответствуют следующие значения свойства `Style`:

- `tbsButton` – обычная инструментальная кнопка;
- `tbsCheck` – западающая кнопка;
- `tbsSeparator` – двойной разделитель;
- `tbsDivider` – одинарный разделитель.

Кроме того, свойство `Style` имеет еще одно значение `tbsDropDown`. При нажатии на кнопку появляется выпадающее меню. Но для этого надо в свойстве `MenuItem` указать имя соответствующего пункта меню. Если при проектировании этого пункта меню были заданы такие свойства, как `ImageIndex`, `Hint`, `Enabled`, `Visible`, то все эти свойства автоматически присвоятся и данной кнопке.

Свойство `Caption` позволяет вывести рядом с кнопкой некоторый текст, при этом текст будет виден, если свойство `ShowCaptions` компонента `TToolBar` установлено в `true`. Обычно в инструментальных панелях тексты кнопок не показывают, но тогда важное значение приобретает свойство `Hint` (подсказка). Ведь даже если рисунок кнопки достаточно информативен, пользователю иногда бывает трудно определить, для чего предназначена эта кнопка. Свойство `Hint: string` позволяет вывести всплывающую подсказку при наведении на кнопку указателя мыши. Подсказка будет показана при условии, что свойство `ShowHint= true`.

Свойство `Wrap` позволяет управлять размещением кнопок в несколько рядов. Если вам необходимо, чтобы после текущей кнопки следующая кнопка располагалась в новом ряду, вам нужно установить свойство `Wrap` текущей кнопки в `true`.

Давайте доработаем предыдущую программу. Поместите на форму компонент и спроектируйте вид инструментальной панели как показано на рис. 9.89.

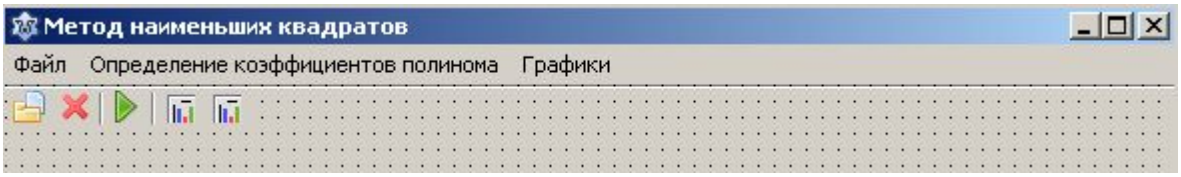


Рис. 9.89. Вид инструментальной панели приложения

Очень важно иметь в виду, что панель инструментов мы создали для того чтобы продублировать некоторые пункты меню (в нашем случае все). Обработ-

чики событий `OnClick` для этих пунктов уже существуют и, поэтому, нет никакой необходимости писать новые обработчики! Для каждой кнопки просто укажите имена уже готовых обработчиков. Например, выделите первую инструментальную кнопку. Она будет дублировать пункт меню **Файл->Открыть**. В инспекторе объектов во вкладке `События` в строке `OnClick` нажмите на кнопку с треугольником и выберите необходимый обработчик, рис. 9.90.

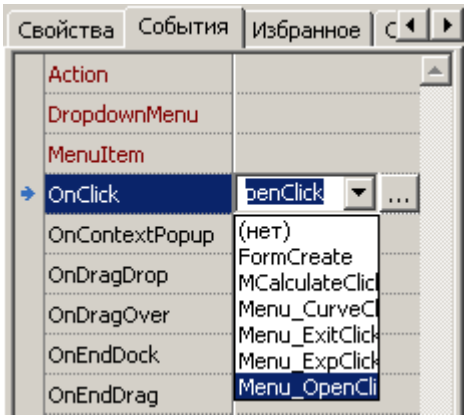


Рис. 9.90. Выбор нужного обработчика

В коде программы произойдут минимальные изменения. В обработчике `OnCreate` формы добавьте операторы:

```
TB_Calculate.Enabled:= false;
TB_Graf_Exp.Enabled:= false;
TB_Graf_Curve.Enabled:= false;
```

Чтобы три последние кнопки были недоступны. В обработчике `Menu_OpenClick` после оператора

```
MCalculate.Enabled:= true;
```

добавьте оператор

```
TB_Calculate.Enabled:= true;
```

А в обработчике `MCalculateClick` после оператора

```
Menu_Curve.Enabled:= true;
```

добавьте операторы

```
TB_Graf_Exp.Enabled:= true;
TB_Graf_Curve.Enabled:= true;
```

Ну и, разумеется, в класс формы будут добавлены объекты классов `TToolBar` и `TToolButton`.

Посмотрите, как мы минимальными усилиями реализовали такой достаточно сложный элемент графического интерфейса как панель инструментов.

9.2.11.3. Компонент `TActionList`

В предыдущей программе мы назначали для кнопки инструментальной панели тот же обработчик, что и для соответствующего пункта меню. Есть другой, более общий, метод унификации поведенческой стороны приложения. Это так называемый механизм действий `Actions`. Этот метод является более предпочтительным и рекомендуемым при разработке сложных программ. Основное достоинство метода – код программы становится более прозрачным и структурированным, а процесс проектирования и разработки приложения более стандартизованным и менее трудоемким. При этом под действием понимается организация реакции программы на какие-то действия пользователя, например нажатие на кнопку на панели инструментов. Почему же мы говорим действия, а не реакции. Потому что мы в программе заранее предусматриваем круг возможных действий пользователя и организуем соответствующие реакции на эти действия. Таким образом, исходным посылом является действие – `Action`. А

на действия пользователя, которые не были предусмотрены, программа просто не будет реагировать. Например, если вы не предусмотрели обработчик для нажатия какой-нибудь кнопки, то, сколько бы вы ни нажимали на эту кнопку, никакого результата не будет.

Механизм действий дает удобное средство для организации централизованной реакции программы на те или иные действия пользователя. Компонент `TActionList` содержит список названий действий и связанных с ними имен обработчиков, которые эти действия реализуют. Чтобы связать какой-то компонент с нужным действием, достаточно раскрыть в Инспекторе объектов список в свойстве `Action` и выбрать нужное название. При этом все свойства этого действия и обработчик автоматически перенесутся в данный компонент, включая изображение, надпись на компоненте, а также связанная с ним всплывающая подсказка.

Поместите на форму компоненты `TActionList` и `TImageList`. Занесите в `TImageList` несколько пиктограмм. В свойстве `Images` компонента `TActionList` укажите имя компонента `TImageList`. Дважды щелкните по компоненту `TActionList` на форме. Появится редактор списка действий. Для того чтобы добавить новое действие вы можете:

- Нажать на кнопку с изображением знака плюс. В правом окне редактора появится новое действие со стандартным именем `Action1`;
- Нажать на маленький треугольник рядом с кнопкой с изображением знака плюс. Появится раскрывающийся список, рис. 9.91 в котором вы можете выбрать "Новое действие" или "Новое стандартное действие";

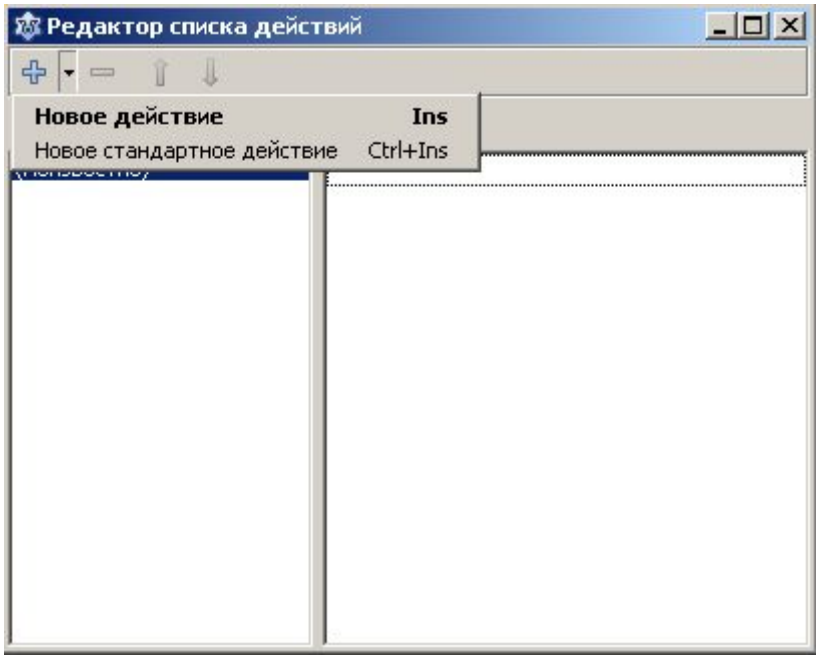


Рис. 9.91. Редактор списка действий

- В редакторе списка действий щелкните правой клавишей мыши. Появится контекстное меню, рис. 9.92;

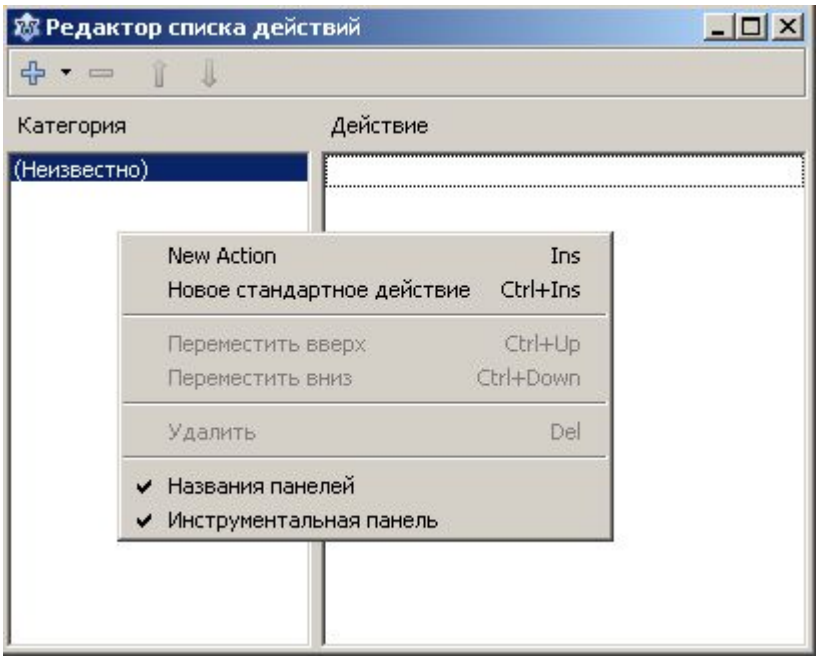


Рис. 9.92. Добавление нового действия

- И наконец, вы можете использовать клавишу `Ins` для ввода нового действия

или нажать `Ctrl+Ins` для ввода нового стандартного действия.

Теперь, что такое стандартное действие. Разработчиками Lazarus созданы специальные классы, реализующие типовые, наиболее часто встречающиеся действия, так что для них даже не требуется писать обработчики! Если вы выберете "Новое стандартное действие", появится окно со списком стандартных действий, рис. 9.93.

На рисунке вы видите, что стандартные действия разбиты на категории. Там же вы видите имена соответствующих классов. Все остальные действия (по терминологии Lazarus `unknown` – неизвестные) относятся к классу `TAction`. Но лучше говорить просто действия или нестандартные действия.

Рис. 9.93. Стандартные действия

Выберем, например стандартное действие `TFileOpen`, рис. 9.94, 9.95.

Рис. 9.94. Свойства

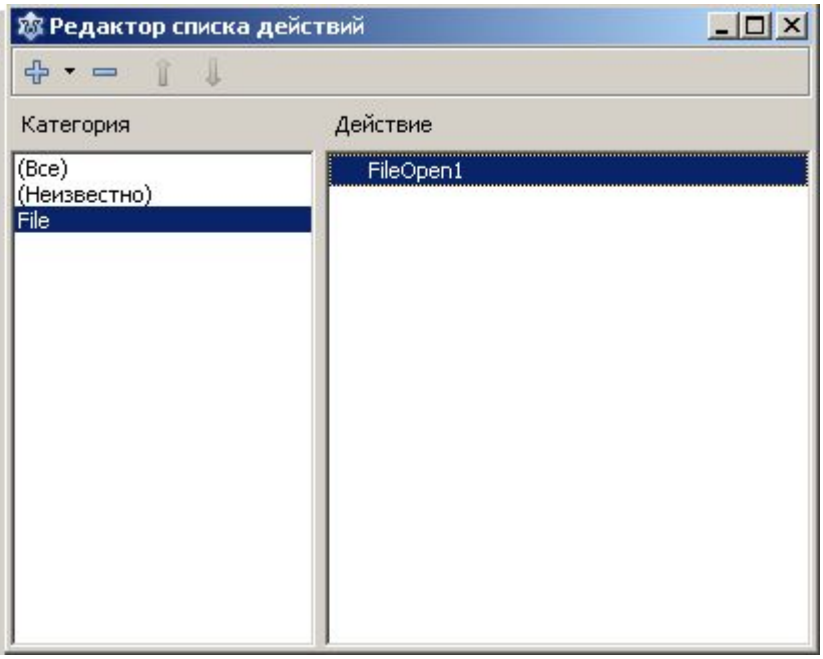


Рис. 9.95 Добавление стандартного действия

Мы видим, что в Инспекторе объектов автоматически заполнились свойства `Caption`, `Hint`, `ShortCut`. Задайте необходимое значение свойству `ImageIndex`. В редакторе списка действий рядом с именем действия тут же появится соответствующее изображение, рис. 9.96.

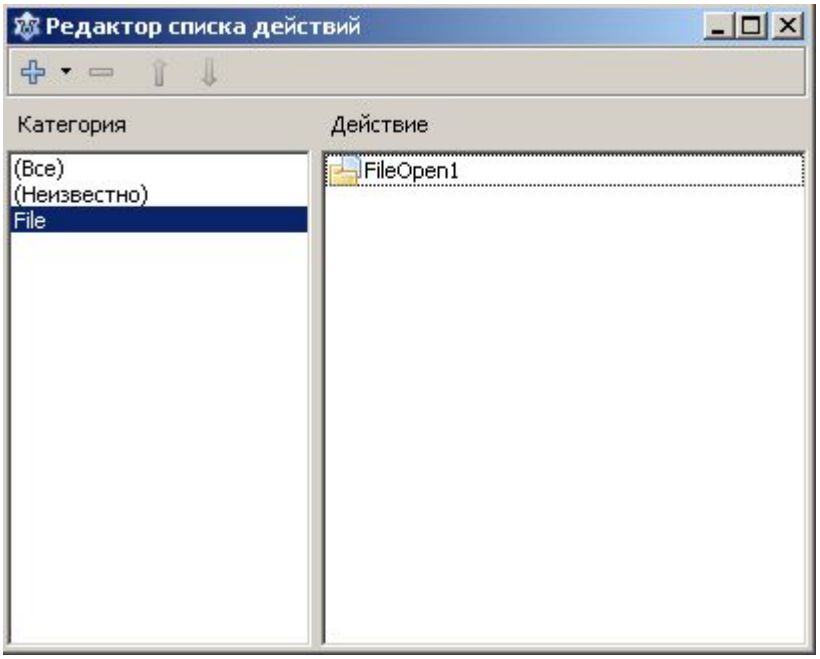


Рис. 9.96. Добавление пиктограммы

Раскрыв свойство Dialog, вы можете настроить свойства стандартного диалога открытия файла. При этом обратите внимание, мы на форму компонент TOpenDialog не переносили! Он уже реализован в классе TFileOpen.

При добавлении нестандартного действия вам необходимо будет заполнить свойства этого действия, в частности, упомянутые выше свойства Caption, Hint, ShortCut. И обязательно написать обработчик OnExecute, реализующий нужное действие. Это событие возникает, когда пользователь инициировал действие, например, нажал на кнопку. Кроме этого события для каждого действия определены еще два события OnHint и OnUpdate. Событие OnHint возникает в момент показа подсказки, когда пользователь задержал указатель мыши над интерфейсным компонентом. Обработчик определен следующим образом:

```
procedure FileOpen1Hint(var HintStr: string;
                        var CanShow: Boolean);
```

Вы можете заменить текст подсказки Hint, указав новый текст в параметре HintStr.

Событие OnUpdate возникает, когда пользователь "ничего не делает". Вы можете использовать это событие для выполнения каких-то подготовительных операций.

В заключение отметим, что в стандартных действиях диалогов, требующих выбора (категории Search, Dialog, File) отсутствуют события OnExecute и OnUpdate. Вместо них введены события BeforeExecute (возникает перед вызовом диалога), OnAccept (возникает, если пользователь произвел выбор) и OnCancel (возникает, если пользователь в диалоге выбор не произвел).

Например, для того чтобы получить имя файла, выбранного пользователем в стандартном действии FileOpen1 надо в обработчике OnAccept записать

```
Fname:= FileOpen1.Dialog.FileName;
```

Напишем теперь программу метода наименьших квадратов с применением механизма действий, заодно определим порядок или методику разработки таких приложений.

Прежде всего, необходимо продумать и определить список тех действий (на бумаге), которые мы будем реализовывать. Записывается это в свободной форме, так, чтобы было понятно самому себе.

Итак, мы знаем, что экспериментальные данные содержатся в текстовом файле. Поэтому первое действие очевидно. Необходимо предоставить пользователю возможность в диалоге открыть файл с экспериментальными данными. Назовем это действие "Открыть файл". Заодно отмечаем для себя, что проще всего для этого использовать стандартное действие. Второе действие - выполнить необходимые вычисления, согласно алгоритму метода наименьших квадратов. Пусть это действие будет называться "Вычисления". Затем, необходимо дать пользователю возможность просмотреть график, построенный по заданным экспериментальным точкам. Назовем это действие "Построение графика по заданным экспериментальным точкам". И, наконец, необходимо построить совмещенный график. График по экспериментальным точкам и график подобранной кривой для того, чтобы пользователь мог визуально оценить точность полученных результатов. Присвоим этому действию название "Построение совмещенного графика". Предусмотрим еще одно стандартное действие – выход из программы. После того, как список действий составлен, уже более или менее ясно, как будет выглядеть главное меню нашей программы. Отмечаем для себя, какие пункты меню следует продублировать в инструментальной панели. Также обязательно отметьте, какие пункты меню и кнопки инструментальной панели должны быть вначале доступны, а какие нет. И когда следует недоступные пункты и кнопки сделать доступными.

Конечно, при разработке больших и сложных программ невозможно все

заранее предусмотреть и учесть. По мере продвижения работы над программой будут добавляться новые действия и, быть может, удаляться некоторые старые. Но составление хотя бы приблизительного списка действий перед началом работы совершенно необходимо. Это позволит вам мысленно обозреть общую структуру вашей будущей программы и значительно поможет вам в дальнейшей работе.

Итак, используя составленный список действий, в редакторе списка действий создайте следующие действия, рис. 9.97.

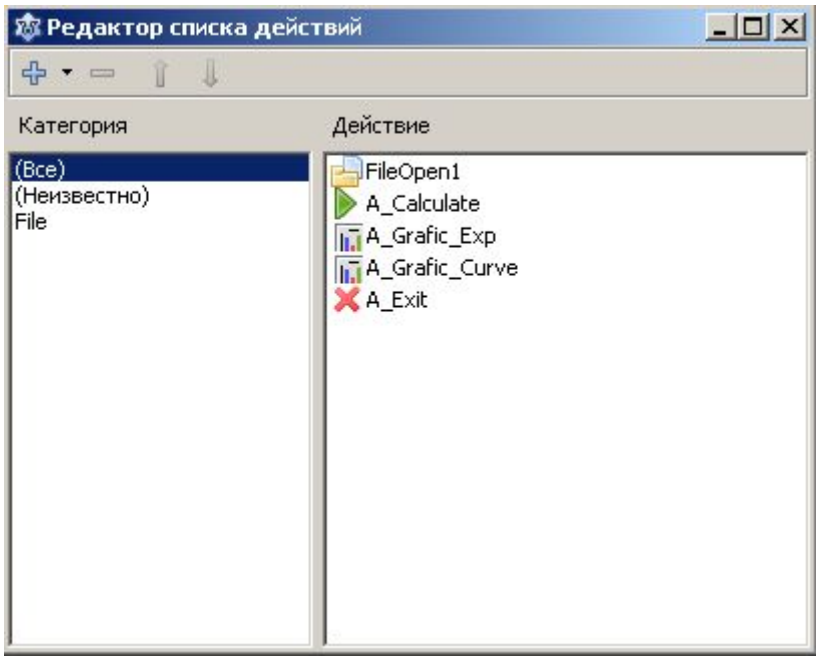


Рис. 9.97. Список действий

Перенесите на форму компонент `TMainMenu`. Создайте пункт меню "Файл". Присвойте ему имя `Menu_File`. Создайте подпункт. Присвойте имя `Menu_Open`. Другие свойства пока не заполняйте! Рис. 9.98.

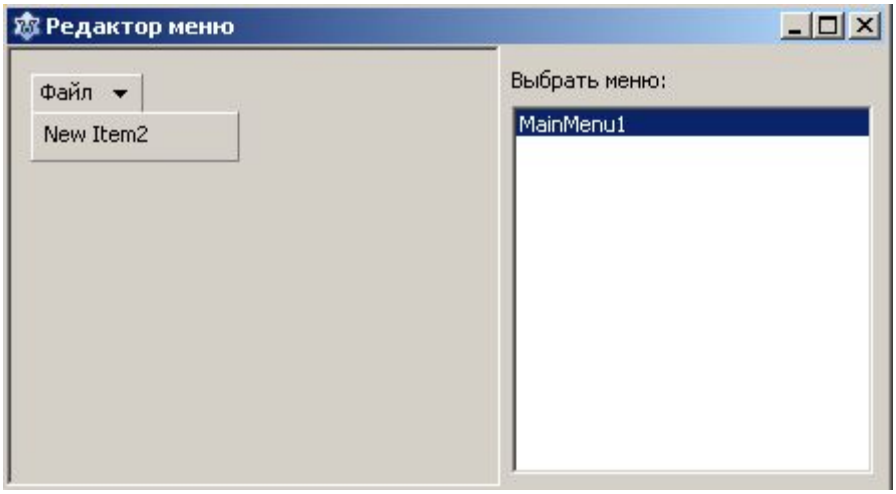
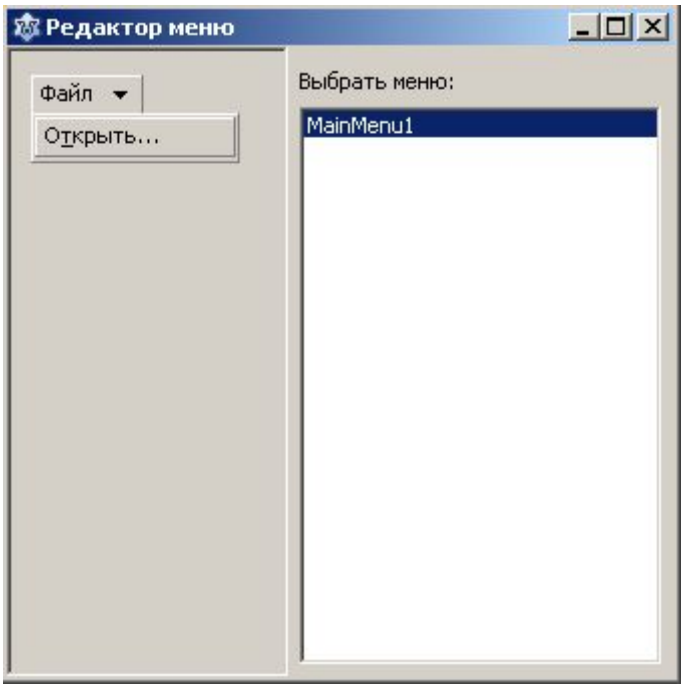


Рис. 9.98. Создание меню приложения

Теперь в свойстве `Action` раскройте список и выберите действие `FileOpen1`. Свойство `Caption` `Menu_Open` вместо `New Item2` автоматически изменилось на "Открыть…", рис. 9.99, 9.100.



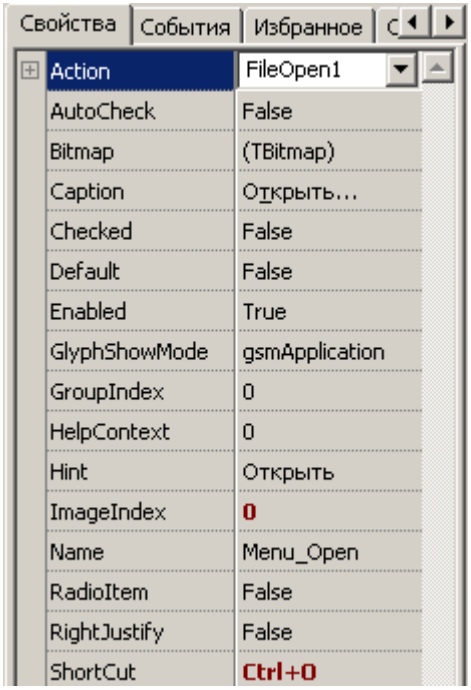


Рис. 9.99. Создание меню приложения

Рис. 9.100. Свойства

Кроме того, автоматически заполнились свойства `Hint`, `ImageIndex` и `ShortCut`! Нам теперь осталось написать обработчик для действия

FileOpen1. Поскольку это стандартное действие – открытие файла, нужно написать обработчик события OnAccept.

```
procedure TForm1.FileOpen1Accept(Sender: TObject);
// процедура выбора, открытия и чтения файла данных
var
  f: TextFile;
  i: integer;
  Fname: string;
begin
  Fname:= FileOpen1.Dialog.FileName;
  Fname:= UTF8ToSys(Fname); //преобразование в системную кодировку
  AssignFile(f, Fname);
  Reset(f);
  // отключение контроля ошибок ввода/вывода
  {$I-}
  // чтение количества экспериментальных точек
  Readln(f, n);
  if IOResult <> 0 then
  begin
    ShowMessage('Ошибка при чтении из файла!');
    exit;
  end;
  // распределение памяти под массивы
  SetLength(x, n);
  SetLength(y, n);
  for i:= 0 to n - 1 do
  begin
    read(f, x[i]);
```

```
    if IOResult <> 0 then
    begin
      ShowMessage('Ошибка при чтении из файла!');
      exit;
    end;
  end;
  for i:= 0  to n - 1 do
  begin
    read(f, y[i]);
    if IOResult <> 0 then
    begin
      ShowMessage('Ошибка при чтении из файла!');
      exit;
    end;
  end;
  {$I+}
  CloseFile(f);
  A_Calculate.Enabled:= true;
end;
```

Обработчик практически совпадает с обработчиком `Menu_OpenClick` предыдущей программы. Но есть существенные различия. Во-первых, как мы уже отмечали, мы компонент `TOpenDialog` в явном виде вообще не используем. А имя файла мы получаем с помощью свойства `FileName` класса `TFileOpen`. Во-вторых, чтобы сделать доступными пункт меню "Вычислить" и соответствующую кнопку на панели инструментов в предыдущей программе в обработчике `Menu_OpenClick` мы вынуждены были записывать два оператора

```
MCalculate.Enabled:= true;
TB_Calculate.Enabled:= true;
```

А здесь достаточно написать один

```
A_Calculate.Enabled:= true;
```

Сформируйте теперь остальные пункты меню и назначьте им соответствующие действия из `TActionList`. Напишите требуемые обработчики.

Теперь поместите на форму `TToolBar`. Добавьте необходимые кнопки и точно так же для каждой кнопки в свойстве `Action` установите соответствующие действия.

Окончательный код программы будет следующим:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes,  SysUtils,  FileUtil,  LResources,  Forms,
  Controls, Graphics, Dialogs, ActnList, Menus, StdActns,
  ComCtrls, TAGraph, TASeries;
type
  { TForm1 }
  TForm1 = class(TForm)
    A_Grafic_Curve: TAction;
    A_Grafic_Exp: TAction;
    A_Calculate: TAction;
    ActionList1: TActionList;
    Chart1: TChart;
```

```
    A_Exit: TFileExit;
    FileOpen1: TFileOpen;
    ImageList1: TImageList;
    MainMenu1: TMainMenu;
    Menu_Curve: TMenuItem;
    Menu_Exp: TMenuItem;
    Menu_Calculate: TMenuItem;
    MCalculate: TMenuItem;
    Menu_Grafic: TMenuItem;
    Menu_Exit: TMenuItem;
    Menu_File: TMenuItem;
    Menu_Open: TMenuItem;
    ToolBar1: TToolBar;
    TB_Open: TToolButton;
    TB_Exit: TToolButton;
    TB_Divide_1: TToolButton;
    TB_Calculate: TToolButton;
    TB_Divide_2: TToolButton;
    TB_Graf_Exp: TToolButton;
    TB_Graf_Curve: TToolButton;
    procedure A_CalculateExecute(Sender: TObject);
    procedure A_ExitExecute(Sender: TObject);
    procedure A_Grafic_CurveExecute(Sender: TObject);
    procedure A_Grafic_ExpExecute(Sender: TObject);
    procedure FileOpen1Accept(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  public
```

```
    { public declarations }
  end;
procedure gauss(vector: array of real; b: array of real;
                var x: array of real; n: byte;
                var solve: byte);
// Процедура решения СЛАУ методом Гаусса
// n - размерность системы,
// solve=0, если решение единственное,
// solve=1, если система не имеет решения,
// solve=2, если система имеет бесконечное количество решений,

function fx(t: real): real;
// Функция, подбираемая методом
// наименьших квадратов
function stepen( x: real; n: byte): real;
// Функция возведения в степень
var
  Form1: TForm1;
  n: byte;
  x1, y1: real;
  x, y, z: array of real;

implementation

function fx(t: real): real;
begin
  Result:= z[0] + z[1]*t + z[2]*t*t +
           z[3]*t*t*t + z[4]*sqr(sqr(t));
end;
```

```
function stepen( x: real; n: byte): real;
// процедура возведения в целую степень
var
  i: integer;
begin
  Result:= 1;
  for i:= 1 to n do
  Result:= Result*x;
end;
// Реализация метода Гаусса
procedure Gauss(vector: array of real; b: array of real;
                var x: array of real; n: byte;
                var solve: byte);
var
 a: array of array of real; { матрица коэффициентов системы,
 двумерный динамический массив}
 i, j, k, p, r: integer;
 m, s, t: real;
begin
 SetLength(a, n, n); // установка фактического размера массива

 { Преобразование одномерного массива в двумерный }
 k:=0;
 for i:=0 to n-1 do
   for j:=0 to n-1 do
   begin
     a[i, j]:= vector[k];
     k:=k+1;
   end;
```

```
for k:=0 to n-2 do
begin
  for i:=k+1 to n-1 do
  begin
    if (a[k, k]=0) then
    begin
      { перестановка уравнений}
      p:=k; // в алгоритме используется буква l, но она похожа на 1
            // Поэтому используем идентификатор p
      for r:=i to n-1 do
      begin
        if abs(a[r, k]) > abs(a[p, k]) then p:=r;
      end;
      if p<>k then
      begin
        for j:= k to n-1 do
        begin
          t:=a[k, j];
          a[k, j]:=a[p, j];
          a[p, j]:=t;
        end;
        t:=b[k];
        b[k]:=b[p];
        b[p]:=t;
      end;
    end; // конец блока перестановки уравнений
    m:=a[i, k]/a[k, k];
    a[i, k]:=0;
    for j:=k+1 to n-1 do
```

```
    begin
      a[i, j]:=a[i, j]-m*a[k, j];
    end;
    b[i]:= b[i]-m*b[k];
  end;
end;
{Проверка существования решения}
if a[n-1,n-1] <> 0  then
begin
  x[n-1]:=b[n-1]/a[n-1,n-1];
  for i:=n-2 downto 0 do
  begin
    s:=0;
    for j:=i+1 to n-1 do
    begin
      s:=s-a[i, j]*x[j];
    end;
    x[i]:=(b[i] + s)/a[i, i];
  end;
  solve:= 0;
end
else
if b[n-1] = 0 then
begin
  MessageDlg('Система имеет бесконечное ' +
             'количество решений', mtInformation,[mbOK], 0);
  solve:= 2;
end
else
```

```
 begin
   MessageDlg('Система не имеет решений',
                  mtInformation,[mbOK], 0);
   solve:= 1;
 end;
 { освобождение памяти,
   распределенной для динамического массива }
 a:=nil;
end;

{ TForm1 }

procedure TForm1.FileOpen1Accept(Sender: TObject);
// процедура выбора, открытия и чтения файла данных
var
  f: TextFile;
  i: integer;
  Fname: string;
begin
  Fname:= FileOpen1.Dialog.FileName;
  Fname:= UTF8ToSys(Fname); //преобразование в системную кодировку
  AssignFile(f, Fname);
  Reset(f);
  // отключение контроля ошибок ввода/вывода
  {$I-}
  // чтение количества экспериментальных точек
  Readln(f, n);
  if IOResult <> 0 then
  begin
```

```
    ShowMessage('Ошибка при чтении из файла!');
    exit;
  end;
  // распределение памяти под массивы
  SetLength(x, n);
  SetLength(y, n);
  for i:= 0 to n - 1 do
  begin
    read(f, x[i]);
    if IOResult <> 0 then
    begin
      ShowMessage('Ошибка при чтении из файла!');
      exit;
    end;
  end;
  for i:= 0  to n - 1 do
  begin
    read(f, y[i]);
    if IOResult <> 0 then
    begin
      ShowMessage('Ошибка при чтении из файла!');
      exit;
    end;
  end;
  {$I+}
  CloseFile(f);
  A_Calculate.Enabled:= true;
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Chart1.Title.Text.Text:='Метод наименьших квадратов';
  A_Calculate.Enabled:= false;
  A_Grafic_Exp.Enabled:= false;
  A_Grafic_Curve.Enabled:= false;
  Chart1.Visible:= false;
end;

procedure TForm1.A_ExitExecute(Sender: TObject);
begin
  Close;
end;

procedure TForm1.A_Grafic_CurveExecute(Sender: TObject);
{Процедура вывода совмещенных графиков
 экспериментальных данных по точкам и
 подобранной методом наименьших квадратов
 кривой, наилучшим образом приближающейся
 к экспериментальным данным}
var
  i: integer;
  gr1, gr2: TLineSeries;
begin
  Chart1.Visible:= true;
  gr1:= TLineSeries.Create(Chart1);
  gr1.ShowPoints := true; // график с точками
  gr1.ShowLines := false; // не соединять точки линиями
  Chart1.AddSeries(gr1);
```

```
  gr2:= TLineSeries.Create(Chart1);
  gr2.ShowLines := true;
  Chart1.AddSeries(gr2);
  for i:= 0 to n - 1 do
  gr1.AddXY(x[i], y[i]);
  for i:= 0 to n - 1 do
  gr2.AddXY(x[i], fx(x[i]));
end;
procedure TForm1.A_Grafic_ExpExecute(Sender: TObject);
{Процедура вывода графика экспериментальных
 данных по точкам}
var
  i: integer;
  gr1: TLineSeries;
begin
  Chart1.Visible:= true;
  gr1:= TLineSeries.Create(Chart1);
  Chart1.AddSeries(gr1);
  for i:= 0 to n - 1 do
  gr1.AddXY(x[i], y[i]);
end;
procedure TForm1.A_CalculateExecute(Sender: TObject);
var
  i, j, k, l: integer;
  b, vector: array of real;
  s: real;
  solve: byte;
begin
  SetLength(z, 5);
```

```
  SetLength(b, 5);
  SetLength(vector, 25);
  j:= 0;
  for k:= 0 to 4 do
  for l:= 0 to 4 do
  begin
    s:= 0;
    for i:= 0 to n - 1 do
    s:= s + stepen(x[i], k+l);
    vector[j]:= s;
    j:= j+1;
  end;
  for k:= 0 to 4 do
  begin
    s:= 0;
    for i:= 0 to n - 1 do
    s:= s + y[i]*stepen(x[i], k);
    b[k]:= s;
  end;
  gauss(vector, b, z, 5, solve); // решение СЛАУ
  if solve = 0 then
  begin
    A_Grafic_Exp.Enabled:= true;
    A_Grafic_Curve.Enabled:= true;
  end;
end;
initialization
  {$I unit1.lrs}
end.
```

9.2.11.4. Создание приложений с изменяемыми размерами окон

Для этих целей используются компоненты `TPanel` и `TSplitter`. Компонент `TPanel` – панель является контейнером, на котором могут размещаться любые другие компоненты. Компонент позволяет создавать на форме отдельные независимые области и группировать в них функционально связанные интерфейсные элементы.

У большинства компонентов, в том числе и у `TPanel` имеется свойство `Align` – выравнивание. Свойство может принимать следующие значения:

- `alNone` – выравнивание отсутствует;
- `alLeft` – компонент занимает левую часть клиентской области компонента-контейнера (на панель можно помещать сколько угодно других компонентов `TPanel`) или формы;
- `alRight` – компонент занимает правую часть клиентской области компонента-контейнера;
- `alTop` – компонент занимает верхнюю часть клиентской области компонента-контейнера;
- `alBottom` – компонент занимает нижнюю часть клиентской области компонента-контейнера;
- `alClient` – компонент занимает всю свободную клиентскую область компонента-контейнера;
- `alCustom` – выравнивание определяется родительским компонентом.

Поместите на пустую форму компонент `TPanel` и поэкспериментируйте с различными значениями свойства `Align` компонента. При этом учитывайте следующие моменты. Согласно стандартам графического интерфейса в окне программы первой должна идти строка меню. Если вы поместите на форму компонент `TMainMenu` и создадите хотя бы один пункт меню, то даже если вы присвоите свойству `Align TPanel` значение `alClient`, панель не займет

всю область формы. Т.е. меню не будет закрыто панелью!

Значения `alTop` и `alBottom` имеют больший приоритет, чем `alLeft` и `alRight`. Исходя из сказанного, при проектировании внешнего вида, например Проводника, можно создать примерно такой каркас приложения, рис. 9.101.

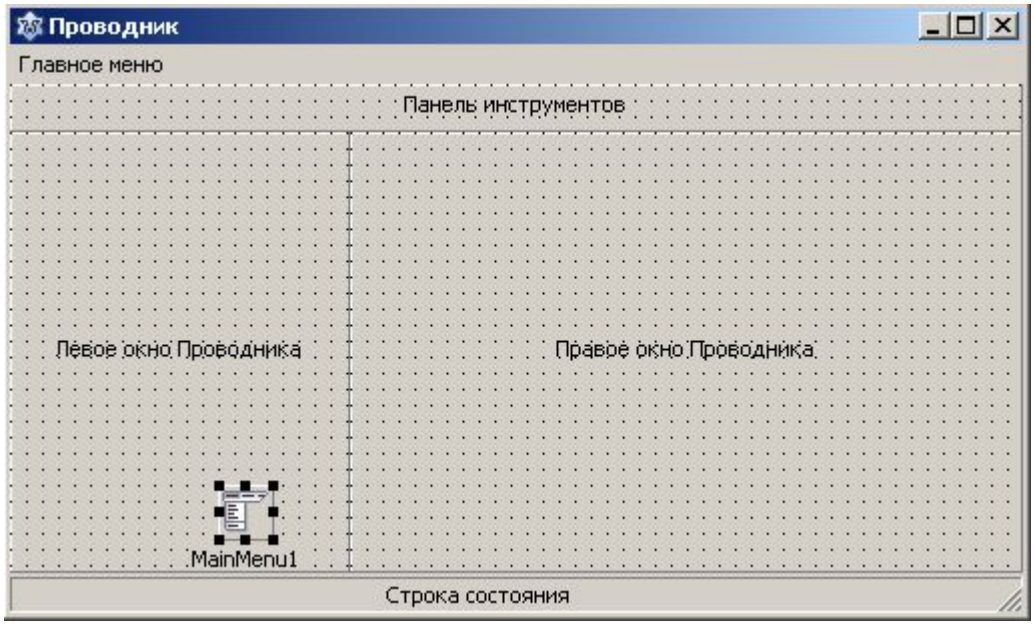


Рис. 9.101. Пример проектирования внешнего вида приложения

Сначала с помощью `TMainMenu` создаем хотя бы один пункт меню. Затем помещаем компонент `TStatusBar` – строку состояния. Компонент имеет по умолчанию свойство `Align= alBottom`. Помещаем на форму три компонента `TPanel`. Первой панели присваиваем значение `Align= alTop`, в нем мы будем размещать панель инструментов, второй панели присваиваем значение `Align= alLeft`, сюда мы поместим компонент `TTreeView`, причем свойству `Align` этого компонента присваиваем значение `alClient`. Третьей панели присвоим значение `Align= alClient`, на него мы поместим компонент `TListView` и его свойству `Align` присвоим значение `alClient`.

Теперь, если пользователь нажмет на кнопку раскрытия окна во весь экран, все компоненты будут пропорционально изменять свои размеры и ваше прило-

жение в этом случае сохранит "приличный" вид.

Практически у всех компонентов имеется также свойство Anchors с помощью которого осуществляется привязка компонента к родительскому компоненту. Обычно Lazarus в зависимости от значения свойства Align компонента автоматически подставляет наиболее подходящие значения и свойству Anchors.

Группа свойств TPanel позволяет управлять вешним видом панели. Это такие свойства как BevelInner, BevelOuter, BevelWidth, BorderStyle, BorderWidth. Просто пробуйте задавать различные значения этим свойствам и вы все сами поймете.

Часто требуется изменять размеры не окна приложения в целом, а размеры компонентов в окне. Например, изменить размеры левого окна Проводника (компонент TTreeView). Причем в данном случае пользователь может менять только ширину окна. Поэтому правильнее говорить об изменении границ панелей, на которых расположены компоненты. Для этого применяется компонент TSplitter – разделитель (вкладка Additional). Чтобы вставить разделитель между двумя панелями, необходимо сначала поместить первую панель, задать стиль выравнивания (свойство Align). Для нашего случая задаем для первой панели Align= alLeft. Затем помещаем на форму TSplitter. По умолчанию он имеет свойство Align= alLeft. Поэтому разделитель автоматически прижмется к панели. Теперь помещаем на форму вторую панель и задаем Align= alClient. Все, теперь во время работы приложения пользователь может перемещать границы панелей, ухватившись мышью за разделитель. При этом форма курсора мыши изменится на двунаправленную стрелку. Задать ширину разделителя можно в свойстве Width. Разберитесь самостоятельно с другими свойствами TSplitter.

Литература

1. Демидович Б.П., Марон И.А. "Основы вычислительной математики", М.: "Наука", 1970.

2. Копченова Н.В., Марон И.А. "Вычислительная математика в примерах и задачах", М.: "Наука", 1972.

3. Гутер Р.С., Овчинский Б.В. " Элементы численного анализа и математической обработки результатов опыта", М.: "Наука", 1970.

4. Архангельский А.Я. "Программирование в Delphi 7", М.: "Бином", 2003.

5. Иванова Г.С., Ничушкина Т.Н., Пугачев Е.К. "Объектно-ориентированное программирование", М.: Изд-во МГТУ им. Н.Э. Баумана, 2003.

6. Кэнту М. "Delphi 7: Для профессионалов", СПб.: Питер, 2004.

7. Ершов А.П. "Введение в теоретическое программирование", М.: "Наука", 1977.

8. Вирт Н. "Алгоритмы и структуры данных", М.: Мир, 1989.

9. Кнут Д.Э. "Искусство программирования", т.1. "Основные алгоритмы", М.: "Мир", 1976.

10. Кнут Д.Э. "Искусство программирования", т.3. "Сортировка и поиск", М.: "Мир", 1978.

11. Ламуатье Ж.П. "Упражнения по программированию на Фортране-IV", М.: Мир, 1978.

12. Дж. Бакнелл "Фундаментальные алгоритмы и структуры данных в Delphi", СПб ДиаСофтЮП, 2003 г.

13. Сайт FreePascal.ru – http://www.freepascal.ru/

14. Форум сообщества ALT Linux – http://forum.altlinux.org/

15. Форум программистов и сисадминов – http://www.cyberforum.ru/

16. http://freepascal.org/

17. http://lazarus.freepascal.org/

Алфавитный указатель

A

Actions. *См*. Механизм действий
Array, 300

B

Boolean, 156

C

Char, 279
CodeTools, 587

D

Dynamic, 560

F

Free Pascal Compiler, 164

G

GUI, 97

I

IDE, 165
Inherited, 555
Integer, 155

L

Lazarus, 165

O

Override, 555

P

Private, 523
Property, 529
Protected, 523
Public, 523
Published, 582

R

Real, 155
Record, 334

S

Self, 550
String, 280

T

TStringList, 671
TStrings, 661

U

Unit, 323

V

Var, 157
Virtual, 555

А

Алгоритм, 115
  Евклида, 117

Б

Блок- схема, 119
Булевая
  алгебра, 45
  переменная, 45

В

Вкладка, 106
Вычислительная система, 66

Г

Граф, 442

Д

Дек, 484
Дерево, 442
  двоичное (бинарное), 445
  обход, 446
Дешифратор, 65

З

Запись, 334

И

Идентификатор, 154
Инспектор объектов, 170
Интерфейс, 79
  графический, 571
Информатика, 16
Информация, 17
  единицы измерения, 19
  ресурсы информационные, 17
  формы представления, 18
Исключения, 624
  классы исключений, 626
  обработка, 627
    except, 627
    finally, 628
    try, 627

К

Клавиатура, 90
Код, 31
  дополнительный, 33
  модифицированный, 37
  обратный, 34
  прямой, 31
Кодовая страница, 277
Команда, 71
Комментарий
  многострочный, 159
  однострочный, 159
Компоненты, 587
  TActionList, 852
  TBitBtn, 608
  TButton, 608
  TChart, 832
  TCheckBox, 749
  TCheckGroup, 752
  TColorDialog, 670
  TComboBox, 737
  TEdit, 611
  TFloatSpinEdit, 654
  TFontDialog, 667
  TImageList, 756
  TLabel, 591
  TLabeledEdit, 620
  TListBox, 727
  TListView, 789
  TMainMenu, 828
  TMaskEdit, 636
  TMemo, 662
  TOpenDialog, 676
  TPanel, 874
  TRadioButton, 751
  TRadioGroup, 752
  TSaveDialog, 676
  TSpeedButton, 611
  TSpinEdit, 654
  TSplitter, 876
  TStatusBar, 624
  TStringGrid, 717
  TToolBar, 848
  TTReeView, 754
  TUpDown, 655
  палитра, 169, 591
Компьютер, 55, 66
  классификация, 73
  поколение, 55
  структура, 67
Консольное приложение, 176

Л

Линейка, 102
  прокрутки, 102

М

Массив, 300
динамический, 307
Меню, 99
Метод
Гаусса, 144
двойного пересчета, 139
наименьших квадратов, 141, 832
пошаговой детализации, 131
Механизм действий, 853
Модуль, 322
системный, 328
Монитор, 88
Мультимедиа, 76
Мышь, 91

О

Обработчик события, 588
Объектно-ориентированное
программирование (ООП), 508
абстракция, 508
инкапсуляция, 509, 518
свойства, 529
спецификаторы доступа, 523
класс, 510
деструктор, 569
дочерний, 538
конструктор, 561
метод, 510
виртуальный, 556
динамический, 560
поля, 510
родительский, 538
члены, 513
экземпляр, 510
классификация, 509
наследование, 509, 538
объект, 508
полиморфизм, 510, 548
позднее связывание, 555
раннее связывание, 550
Окно, 581
Оператор, 153
ввода
read, readln, 162
выбора, 232
вывода
write, writeln, 161
декремента
dec, 163
инкремента
inc, 163
присваивания, 160
составной, 219
условный, 215
цикла
с параметром, 228
с постусловием, 222
с предусловием, 221
Операции
div, 204
mod, 204
Операционная система
Windows, 97
Отладка, 658
Очередь, 484

П

Память, 68
внешняя, 69
оперативная, 69, 82
основная, 68
постоянная, 69
Панель
задач, 100
инструментов, 102
Папка, 108
Переключатель, 106
Переменная
глобальная, 253
имя, 155
локальная, 252
символьная, 279
статическая, 256
строковая, 279
тип, 155
Поиск, 422
двоичный (бинарный), 428
линейный, 422
Порт, 90
Принципы фон Неймана, 67
Программа, 71
Протокол, 80

Процедура, 250
Append, 340
BlockRead, 358
BlockWrite, 358
Close, 339
Delete, 290
Erase, 339
Insert, 291
Read, 340
Readln, 340
Reset, 338
Rewrite, 338
Seek, 348
SetLength, 288
Str, 294
Val, 295
Wtite, 342
Wtiteln, 342
Процессор, 67, 85

Р

Регистр, 63

С

Связанный список, 450
Символ, 279
Системы счисления, 20
базис, 20
двоичная система, 22
непозиционные, 20
позиционные, 20
Событие, 573
Сообщение, 573
Сортировка, 382
быстрая (алгоритм Хоара), 410
вставками, 396
выбором, 389
ключ, 383
обменная (метод "пузырька"), 383
файла
внешняя, 383
внутренняя, 383
эффективность, 383
Стек, 446
Строка, 279
Схема
И, 58
ИЛИ, 60
ИЛИ-НЕ, 61
инвертор, 57
И-НЕ, 60
функциональная, 57
электрическая принципиальная, 57
Счетчик, 64

Т

Тестирование, 657
Триггер, 62

У

Указатель, 272, 474
нетипизированный, 272
разыменование, 475
типизированный, 272

Ф

Файл, 107, 334
нетипизированный, 337
текстовый, 337
тип, 336
типизированный, 337
Флажок, 105
Форма, 581
модальная, 587
Функция, 251
Assign, 338
chr, 279
CompareStr, 297
Concat, 285
Copy, 289
Eof, 339
FileExists, 339
FilePos, 348
FileSize, 348

IntToStr, 297
IoResult, 236
Length, 286
LowerCase, 296
ord, 279
Pos, 292
pred, 279
StrToInt, 297
succ, 279
Trim, 299
upcase, 279
UpperCase, 296
UTF8Copy, 289
UTF8Decode, 299
UTF8Length, 288
UTF8LowerCase, 296
UTF8UpperCase, 296
логическая, 45
дизъюнкция, 47
конъюнкция, 46
отрицание, 46
стандартная
abs, 208
round, 212
sqr, 208
sqrt, 209
trunc, 212

Ч

Число, 20
с плавающей точкой, 27
с фиксированной точкой, 27

Ш

Шина, 85
адресная, 85
данных, 85
команд, 86

Я

Язык
машинный, 71
программирования, 116
Ярлык, 99