Ю. Л. Кетков, А. Ю. Кетков

bhv

СВОБОДНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

FREE PASCAL

для студентов и школьников

Основы программирования на языке Free Pascal

Технология работы в интегрированной среде FP IDE

Системные библиотеки: использование и создание

Графические системы FP IDE: Graph и OpenGL

Рабочая версия профессиональной системы программирования

Задания, решения, примеры программ

ИНФОРМАТИКА И ИНФОРМАЦИОННО-КОММУНИКАЦИОННЫЕ ТЕХНОЛОГИИ

+CD

Юлий Кетков
Александр Кетков

СВОБОДНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

FREE PASKAL

для студентов и школьников

Санкт-Петербург
«БХВ-Петербург»
2011

УДК 681.3.068(07)+800.92FreePascal
ББК 32.973.26-018.2
К37

Кетков, Ю. Л.

К37 Свободное программное обеспечение. FREE PASCAL для студентов и школьников / Ю. Л. Кетков, А. Ю. Кетков. — СПб.: БХВ-Петербург, 2011. — 384 с.: ил. + CD-ROM — (ИиИКТ)

ISBN 978-5-9775-0604-5

Пособие предназначено для изучения компилятора Free Pascal и интегрированной среды FP IDE.

Подробно разобраны основы программирования на языке Free Pascal: история создания и развития языка Pascal, простые типы данных, строковые данные, структурированные типы данных — массивы. Рассматриваются вопросы организации типовых блоков обработки данных — процедур и функций, работа с файлами. Показаны работа с системными библиотеками и создание собственных библиотечных модулей. Книга включает информацию о возможностях двух графических систем, входящих в поставку FP IDE: модуль Graph, использующий традиционный подход, характерный для графических библиотек версий Turbo Pascal, и современный пакет OpenGL. Весь излагаемый материал ориентирован на учебный процесс, представлено большое количество примеров и программ. Прилагаемый компакт-диск содержит готовую к работе систему программирования Free Pascal, дистрибутив Free Pascal и программы, рассматриваемые в книге.

Для образовательных учреждений

УДК 681.3.068(07)+800.92FreePascal
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Людмила Еремеевская*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 02.09.10.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 30,96.
Тираж 1500 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

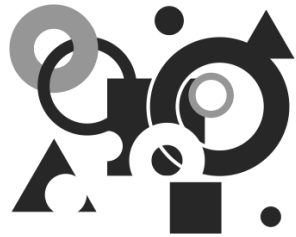
Санитарно-эпидемиологическое заключение на продукцию № 77.99.60.953.Д.005770.05.09
от 26.05.2009 г. выдано Федеральной службой по надзору
в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0604-5

Оглавление

Предисловие



Эпоха создания промышленных средств вычислительной техники и разработки программ, вдохнувших жизнь в компьютерное железо, насчитывает немногим более 60 лет. Первый этап программирования в кодах компьютера выявил самое узкое место в этой новой профессии. Выход готовой продукции за год работы достаточно хорошо подготовленного программиста составлял порядка 2—3 тыс. строк машинного кода, что соответствовало примерно 2—3 законченным программам, которые современные системы программирования позволяют создать за несколько дней.

Для преодоления этого препятствия ведущие производители компьютерной техники и научно-исследовательские организации, вкусившие первые плоды новой технологии решения задач, предприняли коллективные усилия по созданию первых алгоритмических языков — Фортрана (1954 г.) и Алгола (1958 г.). Каждая строка, написанная на алгоритмическом языке, превращалась в десятки машинных инструкций, что позволяло на порядок повысить объем создаваемой программной продукции. Серьезный вклад в развитие первого этапа автоматизации программирования в нашей стране был связан с идеей логических схем программ, предложенных А. А. Ляпуновым (1953). Реализация этих идей нашла свое воплощение в первых программирующих программах для отечественных ЭВМ БЭСМ (А. П. Ершов) и "Стрела" (М. Р. Шура-Бура, С. С. Камынин, Э. З. Любимский).

Но быстро развивающаяся сфера компьютерных приложений требовала большего. И очередным прорывом в технологии программирования стала разработка библиотек типовых алгоритмов, реализующих хорошо известные и вновь разрабатываемые численные методы. Новый подход, известный под названием БСП (библиотеки стандартных программ), потребовал консолидации усилий специалистов, работавших в разных прикладных областях. Коллективный разум был необходим и для поиска оптимальных (по быстродействию и памяти) алгоритмов, и для анализа их точности, и для выявления скрытых ошибок. Одними из первых эту проблему осознали ведущие компьютерные издания Нового и Старого света. Упомянем только некоторые из них — Communications of the ACM (Association for Computing Machinery — ассоциация по вычислительной технике, США), Mathematical Tables and other Aids to Computation (Математические таблицы и другие средства вычислений, США), "Zeitschrift fuer Angewandte Mathematik und Physik" (журнал прикладной математики и физики, Германия), журнал "Вычислительная математика и математическая физика" (СССР). В этих журналах за период 60—70-х годов прошлого столетия было опубликовано более тысячи алгоритмов, преимущественно на язы-

ках Фортран и Алгол. Многие из них были положены в основу ряда популярных и общедоступных библиотек стандартных подпрограмм.

Пользователи всемирно известных моделей серий IBM/360 и их отечественных аналогов ЕС-ЭВМ с ностальгией вспоминают библиотеку научных программ SSP (Scientific Subroutine Package). Два раздела программ из этой библиотеки в 1974 г. были переведены на русский язык и опубликованы издательством "Статистика". Наиболее яркий представитель современных систем технических вычислений, — пакет MATLAB, — начинал свое развитие с библиотек подпрограмм линейной алгебры (LINPACK) и набора алгоритмов анализа собственных значений (EISPACK), разработанных и опубликованных в 70-е годы прошлого столетия.

К числу удачных разработок некоммерческого характера того времени относятся проекты по созданию алгоритмических языков C, C++, Pascal и операционной системы UNIX. На начальной стадии в их развитии принимали участие энтузиасты из лаборатории Bell известной американской телефонной компании AT&T (Д. Ритчи, К. Томпсон, Б. Керниган, Б. Страуструп) и сотрудники технологического института в Цюрихе (Н. Вирт, К. Йенсен). Затем эти проекты ушли в самостоятельное плавание, во время которого к их совершенствованию приложили свои усилия другие энтузиасты, сотрудники разных фирм и учебных заведений.

Мы упомянули некоторые моменты в становлении технологии программирования с целью подчеркнуть вклад в мировой прогресс возможности открытого общения представителей науки и техники, далеко не всегда преследовавших коммерческие цели.

Да, конечно разработка программной продукции требует серьезных денежных вливаний. Сложные системы изначально содержат ошибки, которые разработчик должен устранять, функции этих систем приходится развивать, чтобы они отвечали новым требованиям пользователей (вспомните Service Pack 2 и Service Pack 3 для Windows XP). Разработка, поставка и сопровождение программных продуктов — одна из наиболее прибыльных сфер современной рыночной деятельности. Свидетельством тому служит гигантская империя Microsoft, возглавляемая одним из богатейших людей планеты Билом Гейтсом. Для удержания рыночных позиций ведущие разработчики программной продукции тщательно скрывают исходные коды. Более того, в последнее время крупные фирмы всерьез задумываются уже не о продаже пакетов программ конечным пользователям, а о предоставлении им услуг по решению типовых задач на своих серверах. С одной стороны множественная "сдача в аренду" может принести большую прибыль разработчику, с другой стороны, не в накладе остаются и провайдеры компьютерных сетей. Однако высокая стоимость лицензионных услуг порождает не только массовое недовольство пользователей, но и создает питательную среду для компьютерного пиратства.

Всемирная паутина максимально сблизила пользователей разного уровня и молодых профессионалов, желающих обрести заслуженный статус. Одним из перспективных направлений для приложения их усилий является разработка свободного программного обеспечения, доступ к которому открыт для любого пользователя Интернета. Именно так появились международные проекты Free

BASIC, Free Pascal, Open BASIC, OpenOffice и многие другие. Не перевелись еще энтузиасты и филантропы.

Приобретая эту книгу по цене, едва оправдывающей расходы издательства, вы получаете еще и CD-ROM — "удочку, с помощью которой можно научиться ловить рыбу" и подготовить себя к последующему профессиональному росту. В отличие от многих книг, где подобные вложения сопровождаются лишь демо-версиями той или иной системы, на нашем диске представлена полностью готовая к работе самая свежая профессиональная версия бесплатной системы программирования.

Книга состоит из трех частей. *Часть I* составляют главы, которые знакомят читателей с основами программирования на языке Free Pascal и технологией работы в интегрированной среде FP IDE. Эта среда позволяет создавать консольные 32-разрядные приложения Windows, в которых сняты ограничения широко распространенных в нашей стране версий Turbo Pascal и Borland Pascal. В первую очередь это связано с возможностью использовать всю оперативную память, объем которой на современных ПК превышает 2—3 Гбайт. Вдобавок, компилятор Free Pascal "знаком" с расширенным набором команд современных микропроцессоров и может использовать их мощь при создании приложений. Наконец, прикладные программы получили доступ к широкому спектру услуг операционной системы и богатой коллекции статических и динамических библиотек. Достаточно важным преимуществом компилятора FPC (Free Pascal Compiler) является его универсальность. Наряду с 32-разрядными приложениями он может создавать и 64-разрядные приложения. Различные версии FPC работают под управлением разных операционных систем практически на всех средствах вычислительной техники. Существуют и так называемые кроссплатформенные версии FPC, когда вы на компьютере одного типа создаете приложение, предназначенное для работы на компьютере другого типа.

Глава 1 содержит краткий экскурс в историю создания языка Pascal и его развития. Она не является глубоким исследованием, затрагивающим влияние ряда личностей и фирм на развитие языка. Мы осведомлены о вкладе британских ученых в становление первого стандарта языка, но очень слабо информированы о попытках внесения в Pascal элементов объектно-ориентированного подхода, предпринятых в калифорнийском университете UCSD (University of California at San Diego). В нашей стране история языка ассоциируется с десятком ступенек, по которым прошли все отечественные пользователи, осваивавшие продукцию фирмы Borland.

Глава 2 знакомит читателей с набором нескольких достаточно простых по замыслу, но разнообразных по назначению программ со способами их организации и выполнения в интегрированной среде. Ее цель — продемонстрировать структуру программы и некоторые простейшие приемы программирования.

В *главе 3* достаточно подробно описывается интегрированная среда FP IDE. Перечень функциональных услуг, представляемых командами главного меню и многочисленными диалоговыми окнами, всплывающими во время работы, достаточно велик. Нельзя обойтись без их детального описания и рекомендаций по их практическому использованию. Однако к некоторым услугам мы вынуждены при-

бегать ежедневно, другие могут оказаться менее востребованными, а в отдельные команды и подкоманды главного меню пользователь может ни разу и не заглянуть. Так как довольно сложно прогнозировать уровень запросов различных пользователей, то мы включили в состав третьей главы тот набор функциональных услуг, который рано или поздно понадобится всем. А сведения по деталям настройки многочисленных параметров среды вынесли в приложение.

Глава 4 посвящена описанию простых типов данных, без которых не обходится ни одна программа. Здесь приводятся сведения о внешнем и внутреннем представлении числовых, символьных и логических данных, о способах ввода их значений и вывода результатов работы программы, об операциях, выполняемых над данными разного типа.

В *главе 5* рассматриваются строковые данные, представляющие в компьютере текстовую информацию — один из наиболее распространенных объектов массовой обработки. Наряду с традиционными для языка Pascal "короткими строками" значительное внимание уделено более современным наборам строк неограниченной длины, допускающим как однобайтовую кодировку ASCII, так и двухбайтовую Unicode. Приводятся сведения о типовых операциях, процедурах и функциях, связанных с обработкой текстовой информации, с прямыми и обратными преобразованиями числовых и символьных данных.

Глава 6 знакомит читателей с наиболее распространенным представителем структурированных типов данных — массивами. Здесь описываются преимущества, представляемые массивами при составлении программ с элементами циклической обработки. Приводятся фрагменты программ, демонстрирующие ряд операций линейной алгебры. Не забыт и раздел, связанный с использованием динамических массивов, память для которых выделяется и освобождается во время работы программы.

В *главе 7* рассматриваются объекты типа "множество". Этот тип данных, изначально появившийся в языке Pascal, не очень характерен для других алгоритмических языков. Но он немного расширил сферу приложения математических методов хотя бы в плане использования новых понятий и операций. В качестве примера построена модель "решета Эратосфена" для определения простых чисел.

Понятию *"запись"*, обобщающему наши представления о строках таблицы, посвящена *глава 8*. Этот тип данных очень востребован при обработке табличной информации, хранящейся как в оперативной памяти, так и расположенной на внешних носителях (преимущественно в файлах).

Одним из самых важных разделов *части I* является *глава 9*, в которой рассматриваются вопросы организации типовых блоков обработки данных — процедур и функций. Особое внимание уделено механизму общения по данным между автономными программными единицами. Приводится несколько способов передачи параметров в вызываемые процедуры (функции) с объяснением преимуществ и недостатков каждого из них. В качестве аргументов функций и процедур демонстрируется использование одномерных и двумерных массивов, параметров процедурного типа, параметров по умолчанию. На примере некоторых рекурсивных

функций отмечаются как положительные, так и отрицательные стороны этого приема программирования.

Последняя глава *части I* посвящена работе с файлами. Как показывает опыт проведения олимпиад, в школьной информатике этому разделу уделяется недостаточное внимание. Однако большинство практических приложений извлекают свои исходные данные обычно из текстовых или двоичных файлов и сохраняют результаты своей окончательной или промежуточной работы также в файлах. Файлы являются основным средством обмена информацией во всемирной паутине.

Часть II книги посвящена использованию системных библиотек и созданию собственных библиотечных модулей. В *главе 11* описывается структура модуля и приводится список основных модулей, поставляемых в составе системы FP IDE. На примере данных типа рациональные дроби демонстрируется, как построить модуль для обработки данных нового типа. С применением элементов объектно-ориентированного подхода во вновь построенном модуле вводится понятие *объекта* и строится модель библиотеки, предоставляющей более удобные средства интерфейса.

Глава 12 знакомит читателя с возможностями программного управления дисплеем в текстовом режиме (модуль `CRT`). Здесь описываются как функции, непосредственно связанные с заданием параметров окна вывода (положение, размер, позиция курсора, цветовые атрибуты текста), так ряд вспомогательных функций, обеспечивающих редактирование текста на экране, общение с клавиатурой.

В *главе 13* приводятся сведения о функциях, которые можно условно назвать математическими. Большая их часть сосредоточена в модуле `Math`, некоторое количество содержится в модуле `System`. В главе описываются детали использования относительно мало известных функций (в частности функций округления и статистической обработки данных).

Глава 14 посвящена вопросам обработки календарных дат и интервалов времени с использованием таких компонентов операционной системы, как календарь и часы, и процедур, включенных в модуль `DateUtils`. Приводится краткая историческая справка о разных этапах измерения времени. Подробно описываются возможности полутора сотен процедур модуля, включенного в состав FP IDE.

Часть III книги включает информацию о возможностях двух графических систем, входящих в поставку FP IDE. Первая из них (модуль `Graph`) использует традиционный подход, характерный для графических библиотек версий Turbo Pascal. Здесь в полном объеме сохранен графический интерфейс, присущий стандарту BGI (Borland Graphics Interface). Это означает, что сохранен весь набор графических процедур, но учтены новые функциональные возможности современных дисплеев (разрешение, новые цветовые гаммы, пока еще не поддерживаемые в полном объеме). Вторая графическая система — современный пакет OpenGL (Open Graphics Library — открытая графическая библиотека).

Глава 15 посвящена полному описанию графических возможностей модифицированного стандарта BGI. В *главе 16* изложены основы работы с пакетом OpenGL. В рамках данной книги отсутствует возможность описания более чем 250 процедур

самого пакета OpenGL и большого количества библиотек, так или иначе поддерживающих базовое графическое ядро и его расширения.

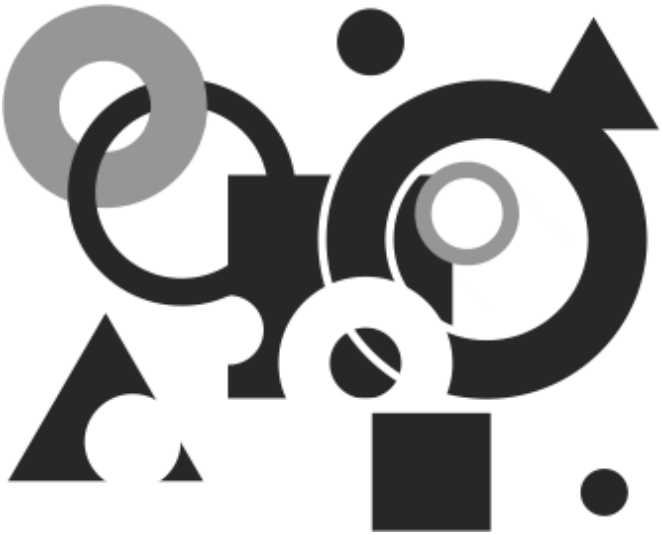
В *приложение 1* включена краткая справка по синтаксису и семантике основных объектов языка Free Pascal — данных и операторов. *Приложение 2* содержит сведения, необходимые для настройки параметров компилятора FPC и интегрированной среды. Таблица с сообщениями об ошибках во время выполнения приложения входит в состав *приложения 3*. *Приложение 4* содержит описание компакт-диска.

Кроме того, в книге представлен список литературы, которой позволит вам расширить свои знания.

Благодарности

Это пятая книга, написанная авторским коллективом Кетковых и выпускаемая издательством "БХВ-Петербург". И все эти годы авторы совместно и плодотворно работали с редакторами издательства, ощущая их благожелательное отношение и профессионализм. В этой книге особую благодарность заслуживает Анна Сергеевна Кузьмина.

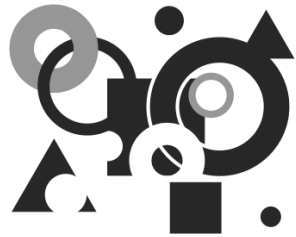
Дополнительно хотелось бы выразить признательность Александру Ивановичу Кузнецову, сотруднику НИИ ПМК, с которым авторы книги на протяжении трех лет обкатывали систему Free Pascal на студенческих и школьных олимпиадах Нижегородской области.



ЧАСТЬ I

Основы программирования в среде FP IDE

Глава 1



Введение в Free Pascal

1.1. Исторический обзор

В 1965 г. был объявлен международный конкурс на создание нового алгоритмического языка — преемника АЛГОЛА-60. В конкурсе принял участие молодой швейцарский ученый Никлаус Вирт (Nicklaus E. Wirth), работавший в то время доцентом Стэндфордского университета. В 1967 г. его проект комиссией был отвергнут — победителем оказался АЛГОЛ-68. Но Вирт продолжил работу над своим замыслом и, вернувшись в Швейцарию, вместе с коллегами из Федерального института технологии (ETH, Цюрих) разработал новую версию языка, названную в честь известного французского инженера Блеза Паскаля — создателя одного из первых механических калькуляторов. В 1970 г. под руководством Н. Вирта был разработан первый транслятор с языка Паскаль, а в 1971 г. появилась первая публикация — техническая документация для пользователей новой системы программирования. Гораздо более известная публикация — совместная книга Никлауса Вирта и его помощницы Кэтлин Йенсен появилась через четыре года. В 1982 г. она была переведена на русский язык. В 1987 г. за создание языка Паскаль Н. Вирт был удостоен медали "Пионер вычислительной техники" (Computer Pioneer) — самой престижной награды Международного компьютерного сообщества (IEEE Computer Society). В первую очередь, язык Паскаль предназначался для использования в учебных заведениях. Цели и задачи, стоявшие перед создателями Паскаля, включали:

- удовлетворение требованиям структурного программирования (программа должна состоять из небольшого количества типовых, легко заменяемых синтаксических конструкций — блоков);
- развитие набора структурированных данных и приемов их обработки (в первую очередь, записей — аналогов хорошо известных таблиц);
- написание надежных программ (за счет введения строгой типизации данных и запрета на использование различных правил умолчания);
- краткость языка (первое полное описание Паскаля занимало 30 страниц против 600 страниц руководства по языку PL/1 в документации по IBM/360).

Важную роль в разработке стандарта языка сыграл Британский институт стандартизации и его рабочая группа во главе с А. Эддиманом. Первый британский

стандарт BS6192 появился в 1982 г., а год спустя Международная организация по стандартам (International Organization for Standardization, ISO) приняла соответствующий документ — ISO 7185. Следует сказать, что Н. Вирт отрицательно отнесся к расширениям языка, предложенным британскими коллегами. Он считал, что нововведения нарушают основные принципы, изложенные выше. Поэтому в дальнейшем Н. Вирт отошел от Паскаля и занялся развитием новых систем программирования на базе языков Modula, Oberon, Zonnon. Наверное, он не во всем был прав: язык программирования — не только средство обучения, но и рабочий инструмент для решения практических задач.

Паскаль довольно долго оставался средством для изучения программирования в учебных заведениях, т. к. ни одна серьезная компьютерная фирма его не поддерживала. Перелом по отношению к Паскалю наметился в 1983—1984 гг., когда за его реализацию взялся молодой французский студент Филипп Канн. Предварительно он прошел длительную стажировку у Н. Вирта, после чего написал сверхскоростной компилятор Turbo Pascal и придумал удачную интегрированную среду (IDE, Integrated Development Environment — интегрированная среда разработки) для только что появившихся персональных компьютеров IBM PC. Среда объединяла редактор исходного кода, компилятор, загрузчик и не очень сложные средства отладки. Вместо многократного формирования командных строк по запуску того или иного компонента системы программирования в интегрированной среде достаточно было нажать пару кнопок. Простота и скорость работы в интегрированной среде послужили одним из главных факторов для привлечения массового пользователя.

В дальнейшем многие фирмы заимствовали идеи Филиппа Канна в своих программных продуктах. Для завоевания рынка Ф. Канн отправился в США, занял деньги у своих дальних родственников и начал продавать Turbo Pascal по смехотворной цене — $49,95 (для сравнения напомним, что незадолго до этого Билл Гейтс ухитрился назначить цену в $500 за интерпретатор Бейсика на домашнем компьютере Altair). Торговый успех (за первый месяц после серьезной рекламной подготовки было продано порядка 3000 копий) заложил фундамент для создания фирмы Borland International. Ее дешевая программная продукция, на ура воспринятая во всем мире и особенно в нашей стране, быстро заполнила вакуум инструментальных средств на IBM-совместимых ПК, где кроме ассемблера и встроенного Бейсика ничего практически не было. Первый коммерческий успех фирмы Borland был достигнут после появления версии Turbo Pascal 2.0 (середина 1984 г.). За ней последовали более развитые версии 3.0 с "черепашьей" графикой (осень 1985 г.), 4.0 (начало 1988 г.) с полноценной графической библиотекой BGI (Borland Graphics Interface), 5.0 (август 1988 г.) с развитыми средствами отладки и возможностью построения оверлейных программ. В мае 1989 г. появилась версия 5.5, в которой были заложены основы объектно-ориентированного подхода. Развитие этого направления продолжилось в версиях TP 6.0 (1990) и TP 7.0 (1992). Последняя версия была выпущена в двух модификациях — для разработки приложений только MS-DOS (TP 7.0) и приложений как MS-DOS, так и Windows (BP 7.0).

Следующий серьезный прорыв в развитии языка Паскаль был связан с преодолением препятствий, выдвигаемых операционной системой при создании 32-разрядных приложений, функционирующих под управлением Windows. Прежние 16-разрядные приложения, создававшиеся в системах программирования Turbo Pascal и Borland Pascal, упирались в серьезные ограничения как по использованию ресурсов (работа с большой оперативной и внешней памятью, новые возможности расширенной системы команд и т. п.), так и по стандартным требованиям к организации приложений, выдвигаемым операционной системой. Первый шаг в преодолении барьера Windows для непрофессиональных пользователей был предпринят в конце 1991 г. компанией Microsoft, которая выпустила на рынок хит последующих 3—4 лет — систему визуального программирования Visual Basic. Ее примеру последовала и фирма Borland, которая уже в 1995 г. разработала на базе языка Object Pascal среду визуального программирования Delphi. Объектно-ориентированный подход, реализованный в языке Object Pascal, дал пользователям Delphi большое преимущество в создании новых компонентов, реализующих не только интерфейсные функции. В системе Visual Basic для разработки VBX- и OCX-компонентов применялись системные средства, не доступные рядовому пользователю. Поэтому появление Delphi было с восторгом воспринято не только мало искушенными пользователями языка Паскаль, но и системными программистами. Благодаря этой среде за относительно короткий срок фирме Borland удалось выпустить на рынок систему визуального программирования на базе языка C++ (Borland C++Builder).

В нашей стране продукция фирмы Borland получила широкое распространение, в первую очередь, в учебных заведениях средней и высшей школы. Ее охотно используют научные и технические предприятия для разработки новых программных продуктов. Существует обширный список литературы на русском языке, посвященной программированию в средах Turbo Pascal, Borland Pascal и Delphi.

Однако не следует забывать, что, во-первых, все упомянутые системы программирования являются коммерческими продуктами, затраты на легальное приобретение которых не всегда по карману многим отечественным пользователям. В этом плане особенно страдают от нехватки средств высшие и средние учебные заведения. Во-вторых, кроме операционных систем типа Windows на IBM-совместимых персональных компьютерах расширяется использование существенно более надежных и более компактных операционных систем типа Linux с сопутствующим программным обеспечением, разрабатываемым и распространяемым на некоммерческой основе в соответствии с правилами FSF (Free Software Foundation). Основное соглашение этого фонда известно под аббревиатурой GPL (General Public License) или, как неологизм, *copyleft* (в противовес известному символу авторского права *copyright*). Это соглашение-лицензия является обязательным для всех разработчиков и пользователей свободно распространяемого программного обеспечения (ПО). Смысл его довольно простой: получив бесплатно дистрибутив ПО вместе со всеми исходными текстами программ, вы можете доработать любой компонент этого ПО и устранить замеченные ошибки. Полученный или модифицированный таким образом дистрибутив вы не имеете права включать в состав любого коммер-

чески создаваемого продукта. Более того, продавая или передавая созданную вами модификацию третьим лицам, вы обязаны передать им дополнительно все исходные тексты своих изменений и пополнить список авторов, включая всех своих предшественников. Такого рода программные продукты, содержащие элементы творчества многих профессионалов, не обязательно являющихся членами первоначального коллектива разработчиков, составляют категорию проектов GNU. К числу наиболее известных GNU-проектов относятся различные версии операционной системы Linux, компиляторы с языков ассемблера (GA.exe, GAS.exe), C и C++ (GCC.exe), Фортран (GFortran.exe), универсальный редактор Emacs, отладчик (GDb.exe), библиотека научных программ (GSL) и др.

В 1993 г. в Интернете стартовал очередной GNU-проект Free Pascal Compiler (сокращенно — FPC), который поначалу ставил целью разработку 32-разрядного компилятора входного языка Turbo Pascal для нескольких операционных систем (DOS, Windows, Linux, ...) и персональных компьютеров разных производителей (*x*86, AMD, Power PC, ...). Инициатором этого проекта был немецкий программист Флориан Клэмпфель (Florian Klämpfl, **florian@freepascal.org**). Следует отметить, что основная часть компилятора FPC писалась на Паскале, и уже к началу 1995 г. появилась версия, которая могла компилировать собственный текст, переводя его код на язык ассемблера IBM PC. В настоящее время на сайте разработчиков (**http://www.freepascal.org**) доступна версия FPC-2.2.4, которая датируется апрелем 2009 г. Говорят, что аппетит приходит во время еды. Поэтому авторов проекта FPC потянуло на большее — они решили создать систему программирования, которая бы не только понимала последнюю версию языка Turbo Pascal, но и могла поддерживать языковые возможности системы Delphi.

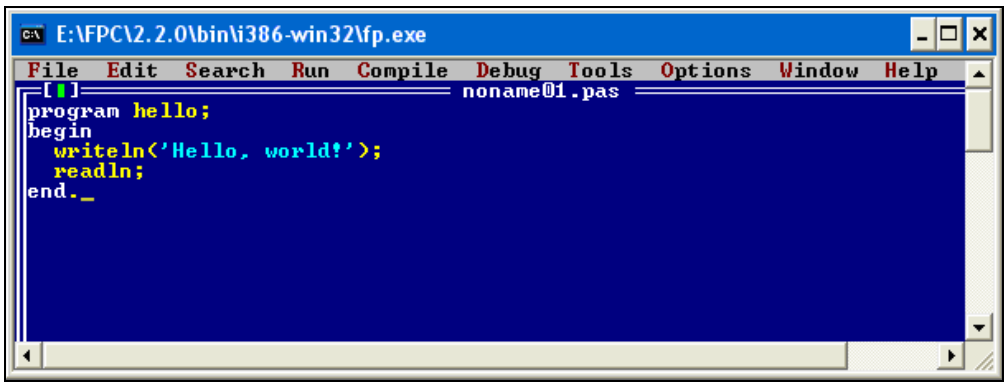


Рис. 1.1. Внешний вид интегрированной среды Free Pascal IDE for Win32 for i386

Для этой цели пришлось усовершенствовать не только компилятор fpc.exe, но и разработать несколько вариантов интегрированных сред, которые бы автоматически формировали запуск таких автономных GNU-утилит, как компилятор с ассемблера ga.exe, библиотекарь (архиватор) ar.exe, загрузчик ld.exe и отладчик gdb.exe.

Первая такая среда (Free Pascal IDE for Win32 for i386) почти один в один напоминала интегрированные среды ранних версий фирмы Borland (рис. 1.1). Ее разработка была начата в 1998 г. и продолжается по сию пору. Последняя версия среды (ver 1.0.10) датируется апрелем 2009 г. Ее авторами являются венгр Бржи Габор (Bbrczi Gabor), француз Пьер Мюллер (Pierre Muller, **muller@janus.u-strasbg.fr**) и голландец Петер Времан (Peter Vreman, **peter@freepascal.org**).

Годом позднее стартовал более амбициозный проект Lazarus. Интерфейс Lazarus IDE и возможности этой среды напоминают систему визуального программирования Delphi. Приведенная на рис. 1.2 бета-версия IDE датируется мартом 2009 года. К ее разработке и наполнению визуальными компонентами причастен довольно большой коллектив программистов из разных стран.

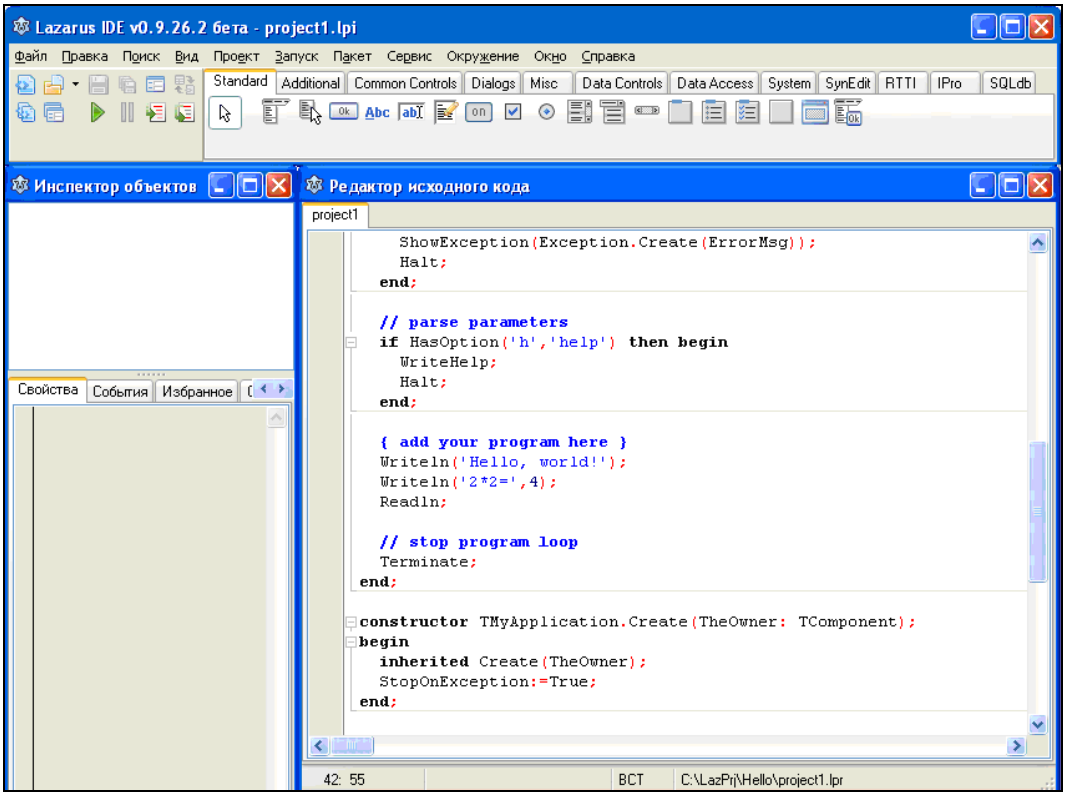


Рис. 1.2. Интегрированная среда Lazarus

В справке о среде содержится следующий текст: "Lazarus — это IDE для создания (графических и консольных) приложений при помощи компилятора Free Pascal. Free Pascal — это компилятор языков Pascal и Object Pascal, распространяемый под лицензией (L)GPL и работающий под Windows, Linux, Mac OS X, FreeBSD, и не только.

Lazarus — это недостающий элемент, который позволит вам разрабатывать программы для всех вышеперечисленных платформ в Delphi-подобном окружении. Эта IDE является инструментом RAD (Rapid Application Development — быстрая разработка приложений), включающим в себя дизайнер форм".

Добавим к этому новую возможность создания 64-разрядных приложений для постоянно возрастающего количества операционных систем (DOS, Win32, OS/2, Linux, FreeBSD, NetBSD, AmigaOS, BeOS, Mac OS и др.) и разнообразных технических платформ (*x*86, *x*86_64, AMD64, ARM, Motorola, PowerPC, Sparc и т. д.).

1.2. Структура программы на языке Free Pascal

Структура программы на языке Free Pascal по форме мало чем отличается от установившихся правил оформления программ в системах Turbo Pascal и в консольных приложениях Delphi (листинг 1.1).

Листинг 1.1. Структура программы

```
program prog_name;
uses Unit1,Unit2,...;        {список используемых модулей}
{-------------------------------------------------------}
const
  {раздел объявления глобальных констант}
type
  {раздел описания глобальных типов}
var
  {раздел объявления глобальных переменных}
label
  {раздел объявления меток в теле основной программы}
procedure
  {раздел описаний пользовательских процедур}
function
  {раздел описаний пользовательских функций}
//-------------------------------------------------------}
begin
  {тело основной программы}
end.
```

В приведенной схеме жирным шрифтом выделены наиболее существенные служебные слова языка программирования, которые предшествуют тому или иному фрагменту программы. Штриховые линии делят программу на три части.

Оператор `program`, с которого начинается первая часть, носит название *заголовка программы*. Он не является обязательным, но его рекомендуется включать в текст программы для указания ее имени (`prog_name`). В первой части программы с помощью служебного слова `uses` (от англ. *uses* — использует) перечисляются имена модулей, которые компилятор должен подключить к исполняемой программе. Модуль (по терминологии Паскаля — `Unit`) играет роль библиотеки подпрограмм (функций и процедур). Он может быть системным или пользовательским. Для обеспечения работы вашей программы используемые модули должны быть предварительно протранслированы и находиться на диске в виде файлов с расширениями tpu (от Turbo Pascal Unit). Если программа использует наиболее востребованный системный модуль `System`, то он подключается автоматически.

Вторую часть программы составляют разделы *объявлений* и *описаний*. Все версии систем программирования на базе языка Паскаль, следуя традициям фирмы Borland, разрешают перечислять приведенные выше разделы в произвольном порядке и даже фрагментировать содержимое любого раздела. При этом должно соблюдаться единственное правило. Если описание/объявление `S1` используется в описании/объявлении `S2`, то `S1` должно быть описано первым. Термином *"глобальный"* мы хотели подчеркнуть, что соответствующие объекты (константы, типы, переменные) можно использовать не только в теле основной программы, но и в любой процедуре/функции, входящей в соответствующий раздел ваших описаний. Любой из разделов второй части может отсутствовать.

В теле основной программы должно быть описано хотя бы одно разумное действие, оправдывающее использование компьютера. Например, практически все руководства по различным алгоритмическим языкам не обходятся без программы, приветствующей мир (листинг 1.2).

Листинг 1.2. Программа `Hello`

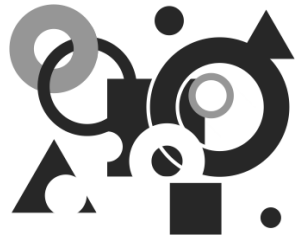
```
Program Hello;
begin
  writeln('Hello, world!');
end.
```

В оформлении описания любой функции или процедуры, включенной в текст вашей программы, присутствуют почти все те же элементы, которые имеют место в основной программе.

Есть только две характерные особенности:

- *заголовок функции* начинается с обязательного оператора `function`.
  Например:
  ```
  function rasst(x1,x2:double):double;
  ```
- *заголовок процедуры* начинается с обязательного оператора `procedure`.
  Например:
  ```
  procedure sum(a,b:double; var c:double);
  ```
- завершающий `end` заканчивается не точкой, а *точкой с запятой*.

Таким образом, программу на языке Free Pascal можно представлять в виде большой матрешки (*основной* или *головной* программы), в которую *вложены* описания используемых данных и подпрограмм. В свою очередь, каждая подпрограмма (процедура или функция) имеет вид *вложенной матрешки*. Этим Паскаль принципиально отличается от языков Basic, C, Fortran, где вложения подобного рода запрещены. Глубина вложения каким-то конкретным числом не ограничена, но на практике довольно редко можно встретить программы с уровнем вложения более 4—5.



Глава 2

Знакомство с простыми программами

Эта глава предназначена для начинающих изучать Паскаль, и ее без ущерба могут пропустить читатели, изучавшие Паскаль в школе или в институте. Хотя повторение — мать учения. Задачи, представленные в этой главе, знакомят читателя с видом программы и некоторыми приемами программирования.

Задача 2.1

Необходимо написать программу, которая вычислит и выведет на экран результаты в виде таблицы (табл. 2.1).

Таблица 2.1

x	sin(x)	cos(x)
0.0	0.000000	1.000000
0.1	0.099833	0.995004
0.2	0.198669	0.980067
0.3	0,295520	0.955336
0.4	0.389418	0.921061
0.5	0.479426	0.877583
0.6	0.564642	0.825336
0.7	0.644218	0.764842
0.8	0.717356	0.696708
0.9	0.783327	0.621610

Обратите внимание на то, что в таблице аргумент `x` задается в радианах. Хотелось бы написать более универсальную программу, которая выводит `k` строк подобной таблицы, начиная с заданного значения `x0`. Приведенный выше фрагмент таблицы должен получиться при `k=10` и `x0=0`.

Самый простой (и одновременно самый тупой) вариант тела программы может состоять из строк, представленных в листинге 2.1.

Листинг 2.1. Программа `table1`

```
program table1;
var
  x:double;
begin
  writeln('x','    sin(x)','    cos(x)');
  x:=0.0;
  writeln(x,sin(x),cos(x));
  x:=x+0.1;
  writeln(x,sin(x),cos(x));
  x:=x+0.1;
  writeln(x,sin(x),cos(x));
  x:=x+0.1;
  writeln(x,sin(x),cos(x));
  x:=x+0.1;
  writeln(x,sin(x),cos(x));
  x:=x+0.1;
  writeln(x,sin(x),cos(x));
  x:=x+0.1;
  writeln(x,sin(x),cos(x));
  x:=x+0.1;
  writeln(x,sin(x),cos(x));
  x:=x+0.1;
  writeln(x,sin(x),cos(x));
  x:=x+0.1;
  writeln(x,sin(x),cos(x));
end.
```

В программе `table1` использована единственная переменная с именем `x`, которая может принимать вещественные значения с удвоенной точностью (тип `double`). Первый оператор `writeln` предназначен для вывода заголовка таблицы. Его три аргумента представлены строковыми константами, которые, в принципе, можно было бы объединить в одну строку. Затем переменной `x` присваивается начальное значение. Операция присваивания, образованная двумя символами `:=`, унаследована Паскалем от своего предшественника — языка АЛГОЛ-60. После этого 10 раз повторяется пара операторов, которая выводит на экран значения `x`, `sin(x)` и `cos(x)` и увеличивает значение переменной `x` на `0.1`.

В поле редактора FPC IDE эта программа выглядела бы следующим образом (рис. 2.1).

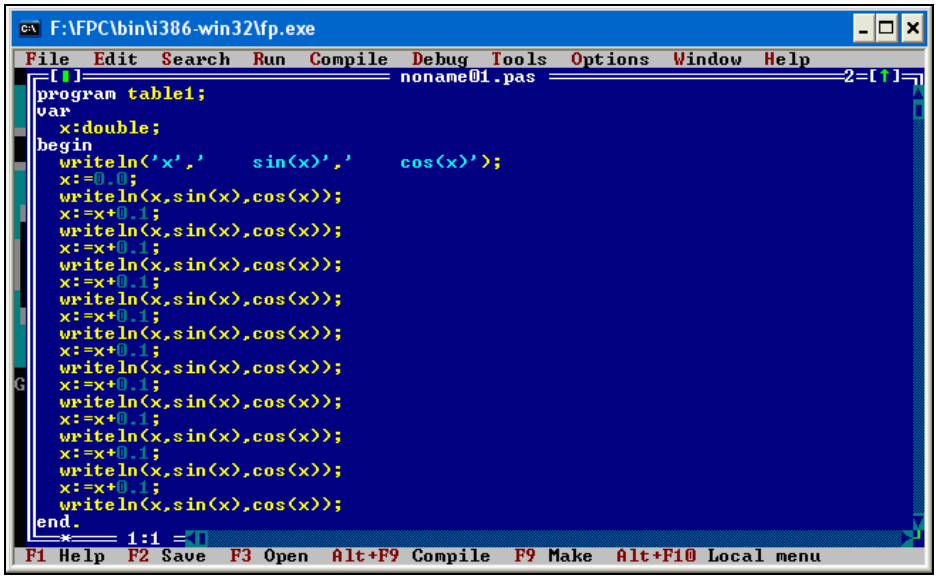


Рис. 2.1. Внешний вид первого варианта программы

Попытка запустить эту программу (команда **Run → Run**) будет приостановлена средой, предлагающей сначала сохранить текст, набранный в поле редактора (рис. 2.2). Такое сохранение вновь набранного или только что модифицированного текста надо обязательно делать перед запуском программы, т. к. во время ее выполнения могут произойти непредвиденные события, и текст программы может пропасть.

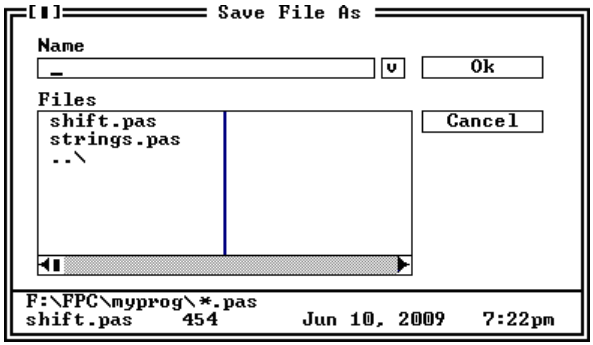


Рис. 2.2. Окно для сохранения текста программы

Сохранив программу под именем table.pas (расширение среда добавляет сама) и снова запустив ее, мы не заметим на экране никаких изменений. Дело в том, что после выполнения программы управление автоматически возвращается среде, а за то ничтожное время, которое программа работала, результаты промелькнули на экране слишком быстро. *Увидеть их можно, нажав комбинацию клавиш*

<Alt>+<F5>. Однако, чаще всего перед заключительным `end` вставляют оператор `readln`, который приостановит работу программы до нажатия клавиши <Enter>.

Результаты, выданные программой и приведенные на рис. 2.3, очень слабо напоминают вид таблицы, указанной в задании. Во-первых, по умолчанию использован так называемый "научный формат" записи вещественных чисел. Символ "`E`" означает, что число, расположенное слева, надо умножить на 10 в степени, расположенной справа. Во-вторых, нам выдано слишком много значащих цифр.

```
Running "f:\fpc\myprog\table1.exe "
x    sin(x)    cos(x)
 0.00000000000000E+000 0.00000000000000E+0000 1.00000000000000E+0000
 1.00000000000000E-001 9.98334166468281 58E-0002 9.95004165278025 77E-0001
 2.00000000000000E-001 1.98669330795061 23E-0001 9.80066577841241 63E-0001
 3.00000000000000E-001 2.95520206661339 56E-0001 9.55336489125606 02E-0001
 4.00000000000000E-001 3.89418342308650 46E-0001 9.21060994002885 10E-0001
 5.00000000000000E-001 4.79425538604202 95E-0001 8.77582561890372 74E-0001
 6.00000000000000E-001 5.64642473395035 34E-0001 8.25335614909678 31E-0001
 7.00000000000000E-001 6.44217687237691 02E-0001 7.64842187284488 45E-0001
 8.00000000000000E-001 7.17356090899522 72E-0001 6.96706709347165 47E-0001
 9.00000000000000E-001 7.83326909627483 33E-0001 6.21609968270664 53E-0001
```

Рис. 2.3. Выдача результатов программы table1.pas

С этим легко побороться — достаточно в операторе `writeln` после каждого выводимого значения указать его общую длину и количество цифр в дробной части:

```
writeln(x:4:1, sin(x):10:6, cos(x):10:6);
```

Однако самая большая нелепость в первом варианте программы заключается в десятикратном повторении однотипных операций. Любой алгоритмический язык, и Free Pascal в том числе, предусматривает возможность циклического повторения таких действий. Этих способов не один, мы воспользуемся оператором цикла `for`. В результате второй вариант нашей программы будет выглядеть так, как представлено в листинге 2.2.

Листинг 2.2. Программа `table2`

```
program table2;
var
  x:double;
  k:integer;          {целое, счетчик циклов}
begin
  writeln('x','    sin(x)','    cos(x)');
  x:=0;
  for k:=1 to 10 do  {тело цикла повторяется 10 раз}
    begin            {начало тела цикла}
      writeln(x:4:1,sin(x):10:6,cos(x):10:6);
      x:=x+0.1;
    end;             {конец тела цикла}
  readln;            {задержка для просмотра результатов}
end.
```

Второй вариант программы выводит результаты в соответствии с заданием. Единственная небрежность — заголовок таблицы немного смещен относительно столбцов соответствующих значений (рис. 2.4).

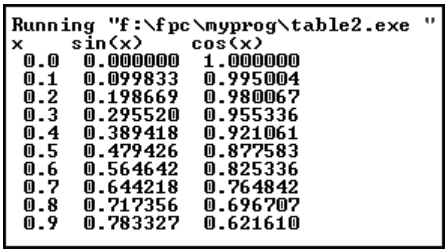


Рис. 2.4. Усовершенствованный вывод

Это легко исправить, добавив к оператору вывода заголовков указание о требуемой длине выводимого текста или нужное количество пробелов в соответствующих текстовых константах:

```
writeln('  x','    sin(x)','    cos(x)');
```

Завершая работу над первой программой, мы можем включить в нее ввод таких данных, как начальное значение `x` и количество `k` выводимых строк таблицы (листинг 2.3).

Листинг 2.3. Программа `table3`

```
program table3;
var
  x:double;
  j,k:integer;        {j - счетчик циклов}
begin
  readln(k,x);        {ввод исходных данных}
  writeln('  x','    sin(x)','    cos(x)');
  for j:=1 to k do    {тело цикла повторяется k раз}
    begin
      writeln(x:4:1,sin(x):10:6,cos(x):10:6);
      x:=x+0.1;
    end;
  readln;
end.
```

Обратите внимание на ввод числовых данных по запросу программы — в нашем примере программа остановит свою работу на первом же операторе тела программы и будет ждать завершения ввода значений `k` и `x` с клавиатуры. Между набираемыми значениями должен быть хотя бы один пробел. Внутри чисел пробелы

недопустимы, ибо они рассматриваются как разделители числовых данных. Признаком завершения набора является нажатие клавиши <Enter>.

Задача 2.2

Натуральное число *N* называется *простым*, если оно имеет только два разных делителя — 1 и само себя. Примеры простых чисел — 2, 3, 5, 7, 11, 13, ... Некоторые математики считают, что 1 не является простым числом, т. к. имеет единственный делитель. Считать 1 составным числом тоже нет никакого резона. Задачей нашей следующей программы является запрос у пользователя числа *N* и анализ его на простоту.

В математике известен достаточно сложный алгоритм такого анализа, требующий минимального числа операций. Но мы воспользуемся следующим алгоритмом, более затратным по времени, но более простым по реализации. Если число N — четное и не равно 2, то оно составное. Если N — нечетное, то будем делить его на последовательные нечетные числа (3, 5, 7, 9, 11, ..., M, где M не превосходит $\sqrt{N}$) и анализировать остатки от деления. Первый же нулевой остаток свидетельствует о том, что N — составное число. Если все остатки будут отличны от 0, то N — простое.

Первый вариант программы, реализующий описанный алгоритм, может выглядеть так, как представлено в листинге 2.4.

Листинг 2.4. Программа `prime1`

```
program prime1;
const
  msg1=' is prime';
  msg2=' is not prime';
label
  m1;
var
  d,N: word;                        {целое, без знака}
  msg: string;                      {строка сообщения}
begin
  write('N = ');                    {вывод приглашения}
  readln(N);                        {ввод N}
  msg := msg1;                      {на случай, если N простое}
  if N<4 then goto m1;              {при N<4 - простое}
  msg := msg2;                      {на случай, если N составное}
  if not odd(N) then goto m1;       {при N четном - составное}
  d:=3;                             {первый делитель - 3}
  while d*d<=N do                   {пока делитель меньше корня из N}
    begin
      if (N mod d)=0 then goto m1; {если остаток = 0}
```

```
      d:=d+2;                    {увеличение делителя на 2}
    end;
  msg:=msg1;
m1:
  writeln(N,msg);                {вывод сообщения }
  readln;
end.
```

В этой программе появились новые объекты и операторы. Во-первых, в разделе констант мы заготовили два сообщения — `msg1` и `msg2`, одно из которых по результатам анализа будет присвоено переменной строкового типа `msg`. Во-вторых, у нас появился раздел меток, в котором описана метка `m1`. Метка располагается перед оператором, на который планируется переход с помощью оператора `goto`, и отделяется от него двоеточием. В-третьих, мы воспользовались операцией `mod`, которая находит *остаток* от деления двух целочисленных операндов. Еще одна новая функция `odd` принимает значение "истина" (`true`), если ее аргумент *нечетный*. Наконец, в нашей программе активно используется условный оператор `if`, который в случае истинности заданного условия выполняет оператор, расположенный вслед за служебным словом `then`. Если заданное условие не выполнено, то срабатывает оператор, следующий за `if`.

Предположим, что для решения каких-то других задач нам потребуется не один раз заниматься подобным анализом. В этом случае было бы полезно выделить фрагмент проверки на простоту в отдельную функцию, например, с именем `prime`, которая бы возвращала логическое значение `true` (истина), если ее аргумент — простое число. Тогда второй вариант программы мог бы выглядеть так, как представлено в листинге 2.5.

Листинг 2.5. Программа `prime2`

```
prog prime2;
var
  N: Word;
function prime(N:Word):boolean;
var
  d: Word;
begin
  Result:=true;
  if N<=3 then exit;
  Result:=false;
  if not odd(N) then exit;
  d:=3;
  while d*d<=N do
```

```
  begin
    if (N mod d)=0 then exit;
    d:=d+2;
  end;
  Result:=true;
end;
begin
  write('N = ');
  readln(N);
  if N=0 then exit;
  if prime(N) then writeln(N,' is prime')
     else writeln(N,' is not prime');
  readln;
end.
```

Количество строк во втором варианте увеличилось незначительно, зато структура основной программы стала предельно простой. Кроме того, здесь использован относительно новый для Паскаля способ формирования значения функции с помощью системной переменной `Result`. Он имеет некоторые преимущества перед традиционным для многих алгоритмических языков способом:

```
prime:=true;
```

или

```
prime:=false;
```

Впервые появившийся оператор `exit` завершает работу функции или основной программы. Но суть главного выигрыша вы поймете, оценив идею формирования следующей программы.

Пример работы программы `prime2` приведен на рис. 2.5.

```
Running "f:\fpc\myprog\prime2.exe "
N = 1
1 is prime

Running "f:\fpc\myprog\prime2.exe "
N = 2
2 is prime

Running "f:\fpc\myprog\prime2.exe "
N = 3
3 is prime

Running "f:\fpc\myprog\prime2.exe "
N = 11
11 is prime

Running "f:\fpc\myprog\prime2.exe "
N = 21
21 is not prime

Running "f:\fpc\myprog\prime2.exe "
N = 101
101 is prime
```

Рис. 2.5. Анализ простых чисел

Задача 2.3

В 1742 г. Христиан Гольдбах высказал предположение, что любое четное число можно представить в виде суммы двух простых чисел, а любое нечетное — в виде суммы трех простых чисел. Если принять, что 1 является простым числом, то вторая часть гипотезы Гольдбаха автоматически вытекает из первой. В 2000 г. известный английский книгоиздатель Фейбер даже объявил о награде в $1 000 000 тому, кто докажет или опровергнет эту гипотезу. Приз пока не востребован. Задача нашей следующей программы — запросить четное число *N* и найти хотя бы одно его разложение на сумму двух простых чисел (то, что задача имеет не единственное решение, вытекает из конкретных примеров: 4 = 2 + 2 и 4 = 1 + 3).

Идея реализуемого алгоритма предельно проста. Пусть, например, `N=a+b` и `a<=b`. Будем перебирать в цикле всевозможные слагаемые: `a` от `1` до `N/2` и `b=N-a`. Если оба слагаемых окажутся простыми, то искомое разложение найдено. Естественно включить в нашу новую программу функцию анализа на простоту (листинг 2.6), построенную в предыдущем примере.

Листинг 2.6. Программа `Goldbach`

```
program Goldbach;
var
  N,a: word;
function prime(N:Word):boolean;
  ...          {текст функции можно скопировать сюда}
end;
begin
  write('N = ');
  readln(N);
  if odd(N) then exit;      {выход, если N - нечетное}
  for a:=1 to N div 2 do
    if prime(a) and prime(N-a) then
    begin
      writeln(N,'=',a,'+',N-a);
    end;
  readln;
end.
```

В новой программе впервые использована операция целочисленного деления `div`. Она позволяет найти целочисленное частное от деления двух целочисленных операндов. Использование обычной операции деления (`/`) в данном случае привело бы к ошибке — конечное значение управляющей переменной `a` в цикле `for` может быть только целочисленным. В Паскале обычная операция деления (`/`), даже целочисленных операндов, всегда приводит к вещественному результату. Это еще одно

принципиальное отличие языка Паскаль, например, от языка C. В последнем операторе `if` записано составное логическое условие, которое принимает значение "истина" только при истинности обоих операндов, объединенных операцией "И" (`and`).

Программа `Goldbach` находит все разложения четного числа на сумму двух простых. Очень жаль, что за нее нельзя получить хотя бы часть вознаграждения, обещанного Фейбером. Примеры работы программы приведены на рис. 2.6.

Из этого примера вы должны усвоить один из важнейших приемов программирования: если вам удалось выделить некоторый типовой алгоритм в процедуру или функцию, то в последующем это поможет сократить время на разработку и отладку новых программ.

```
Running "f:\fpc\myprog\goldbach.exe "
N = 4
4=1+3
4=2+2

Running "f:\fpc\myprog\goldbach.exe "
N = 400
400=3+397
400=11+389
400=17+383
400=41+359
400=47+353
400=53+347
400=83+317
400=89+311
400=107+293
400=131+269
400=137+263
400=149+251
400=167+233
400=173+227
```

Рис. 2.6. Разложение четных чисел

Задача 2.4

В четвертом задании мы продемонстрируем самый примитивный алгоритм сортировки элементов числового массива. Программа должна будет запросить количество *N* сортируемых целочисленных данных, ввести их с клавиатуры, отсортировать в порядке возрастания и вывести отсортированный массив.

Алгоритм сортировки "в лоб" организован следующим образом. Сначала первый элемент массива `a[1]` сравнивается со всеми последующими элементами. Как только обнаруживается элемент `a[j]`, который меньше, чем `a[1]`, их значения меняются местами. Добравшись до последнего элемента `a[N]`, мы добились того, что на первом месте оказалось наименьшее число. Затем мы начинаем сравнивать элемент `a[2]` со всеми остальными. И так продолжается до тех пор, пока неотсортированными остаются `a[N-1]` и `a[N]`. Последнее сравнение либо оставляет все на своих местах, либо приводит к перестановке последней пары.

Описанный алгоритм реализуется с помощью программы из листинга 2.7.

Листинг 2.7. Программа `sort_num`

```
program sort_num;
var
  a: array [1..100] of integer;
  N: byte;
  i,j: byte;
  tmp: integer;
begin
  write('N=');              {приглашение ко вводу N}
  readln(N);                {ввод N}
  for i:=1 to N do          {цикл ввода элементов массива}
    begin
    write('a[',i,'] = ');
    readln(a[i]);
    end;
  for i:=1 to N-1 do        {внешний цикл сортировки}
    for j:=i+1 to N do      {внутренний цикл сортировки}
      if a[i]>a[j] then     {сравнение элементов}
        begin               {блок перестановки пары a[i] и a[j]}
          tmp:=a[i];
          a[i]:=a[j];
          a[j]:=tmp;
        end;
  for i:=1 to N do          {цикл вывода отсортированного массива}
    writeln(a[i]);
  readln;
end.
```

В этой программе появились новые типы данных. Во-первых, массив `a`, объявляемый с помощью служебного слова `array` (массив), вслед за которым указывается диапазон изменения индексов элементов массива (от 1 до 100) и их тип (`integer`). Во-вторых, для переменных `i`, `j`, `N` выбран тип `byte`, обеспечивающий хранение данных в диапазоне от 0 до 255. Элемент `a[i]` сравнивается со всеми остальными — `a[j]`. Поэтому в цикле по `i` появился вложенный цикл по `j`. Наконец, для перестановки пары `a[i]` и `a[j]` понадобилось использовать временную переменную `tmp`. Соответствующий блок перестановки данных — достаточно широко используемый программистский прием.

Результат работы программы сортировки приведен на рис. 2.7.

Задача 2.5

Применим описанный выше алгоритм для сортировки фамилий. Два строковых значения в Паскале сравниваются побуквенно — сначала первая буква первого операнда с первой буквой второго операнда, затем сравниваются вторые буквы и т. д.

Так как коды букв во внутренней кодировке компьютера упорядочены по алфавиту, то сравнение строк ничем не отличается от сравнения чисел. Поэтому новая программа (листинг 2.8) получится из предыдущей путем мизерных переделок.

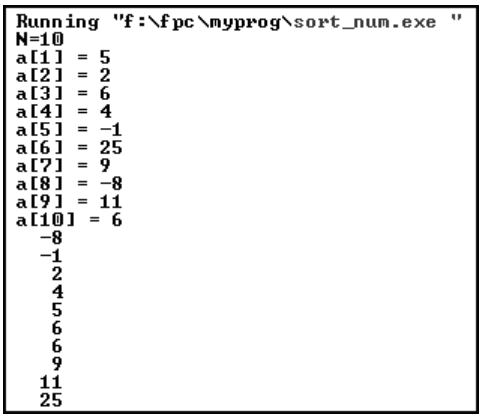


Рис. 2.7. Сортировка числового массива

Листинг 2.8. Программа `sort_nam`

```
program sort_nam;
var
  a: array [1..100] of string [20];
  N: byte;
  i,j: byte;
  tmp: string [20];
begin
  write('N=');              {приглашение ко вводу N}
  readln(N);                {ввод N}
  for i:=1 to N do          {цикл ввода элементов массива}
    begin
    write('a[',i,'] = ');
    readln(a[i]);
    end;
  for i:=1 to N-1 do        {внешний цикл сортировки}
    for j:=i+1 to N do      {внутренний цикл сортировки}
      if a[i]>a[j] then     {сравнение элементов}
```

```
        begin              {блок перестановки пары a[i] и a[j]}
          tmp:=a[i];
          a[i]:=a[j];
          a[j]:=tmp;
        end;
  for i:=1 to N do         {цикл вывода отсортированного массива}
    writeln(a[i]);
  readln;
end.
```

Теперь элементы массива `a` могут хранить строковые значения длиной до 20 символов. Переменную `tmp` тоже пришлось объявить типа `string [20]`.

Пример работы программы сортировки фамилий приведен на рис. 2.8.

```
Running "f:\fpc\myprog\sort_nam.exe "
N=10
a[1] = Иванов
a[2] = Петров
a[3] = Сидоров
a[4] = Козлов
a[5] = Портнов
a[6] = Аникин
a[7] = Федосеев
a[8] = Степанов
a[9] = Бендер
a[10] = Семериков
Аникин
Бендер
Иванов
Козлов
Петров
Портнов
Семериков
Сидоров
Степанов
Федосеев
```

Рис. 2.8. Сортировка фамилий

Построенная программа сортировки фамилий спасует, если вводимые фамилии содержат буквы "ё" или "Ё", т. к. их числовые коды не соответствуют порядку букв в русском алфавите. Если в обычных словарях между большими и малыми буквами разница не делается, то в компьютерных программах приходится учитывать, что коды малых букв больше, чем коды любой большой буквы. Поэтому при составлении программы, которая должна игнорировать это различие, приходится поступать следующим образом: во всех сортируемых словах коды букв приводятся к одному регистру, т. е. либо все заменяется только на большие буквы, либо только на малые. В алгоритмических языках даже предусмотрены специальные функции такого рода, но в большинстве своем они правильно обрабатывают буквы латинского алфавита и обычно не изменяют все остальные символы. Не так уж и сложно написать собственную программу, правильно обрабатывающую русские тексты. В последующих главах мы покажем, как это делается.

Задача 2.6

Еще одна программа, связанная с обработкой символьной информации. Она должна определить, является ли введенная фраза палиндромом. Один из наиболее цитируемых в литературе палиндромов — фраза *"А роза упала на лапу Азора"*. Произношения ее при чтении слева направо и справа налево звучат одинаково. Естественно, что при этом не обращают внимания на разницу в больших и малых буквах и не учитывают расположение пробелов в строке и ее перевертыше.

Чтобы не усложнять алгоритм очередной программы, мы будем считать, что введенная фраза `S1` набирается только малыми буквами. Первым нашим шагом должно стать удаление пробелов, мешающих решению задачи. Устранив пробелы и получив строку `S2`, мы сформируем строку `S3`, в которой символы следуют в обратном порядке. Если строки `S2` и `S3` окажутся равными, то введенная фраза является палиндромом.

Описанный выше алгоритм реализуется программой из листинга 2.9.

Листинг 2.9. Программа `palindrome`

```
program palindrome;
var
  S1,S2,S3: string;
  j,n: byte;
begin
  write('S1 = ');                    {приглашение ко вводу}
  readln(S1);                        {ввод S1}
  S2 := '';                          {чистка S2}
  for j:=1 to length(S1) do          {цикл удаления пробелов}
    begin
      if S1[j] = ' ' then continue;
      S2 := S2 + S1[j];             {присоединение не пробела}
    end;
  S3 := S2;                          {для выравнивания длин S2 и S3}
  n := length(S2);
  for j:=1 to n do                   {цикл переворота S2}
    S3[j] := S2[n+1-j];
  if S2=S3 then writeln('S1 - palindrome')
           else writeln('S1 - not palindrome');
  readln;
end.
```

В этой программе мы впервые использовали функцию `length`, которая позволяет определить длину строки, т. е. количество содержащихся в ней символов. Еще одна специфика этой программы — возможность выделить тот или иной символ

строки по его номеру, например, `S1[j]` или `S3[j]`. Символы в строке нумеруются с 1, их можно читать или присваивать им новые значения. В цикле удаления пробелов важную роль играет оператор `continue` (продолжить), после которого продолжение тела цикла обходится, но сам цикл продолжается. В этом же цикле использована операция *конкатенации* (от англ. *concatenation* — сцепка), обозначаемая знаком `+`. Ее действие сводится к присоединению второго операнда вслед за последним символом первого операнда. Еще одна деталь, на которую следует обратить внимание, связана с использованием полной формы условного оператора `if... then... else` (если... то... иначе). Оператор, расположенный справа от `then`, не должен завершаться точкой с запятой. Точка с запятой в Паскале считается признаком конца оператора, тогда как фрагмент `else...` является не самостоятельным оператором, а всего лишь продолжением предыдущего фрагмента.

Работа программы `palindrome` продемонстрирована на рис. 2.9.

```
Running "f:\fpc\myprog\palindrome.exe "
S1 = а роза упала  на лапу   азора
S1 - palindrome

Running "f:\fpc\myprog\palindrome.exe "
S1 = мама, я хочу домой
S1 - not palindrome

Running "f:\fpc\myprog\palindrome.exe "
S1 = абракадабра
S1 - not palindrome

Running "f:\fpc\myprog\palindrome.exe "
S1 = 12345  5 4 3 2 1
S1 - palindrome
```

Рис. 2.9. Анализ палиндромов

Последнюю программу данного раздела можно отнести к разряду игровых. Сфера коммерческого создания игровых программ — одна из наиболее динамично развивающихся. Как правило, она оказывает большое влияние на совершенствование аппаратных и программных средств. Повышается эффективность систем отображения, появляются новые графические ускорители, стоимость которых иногда превышает стоимость всего остального оборудования, создаются специализированные библиотеки процедур для разработки игровых программ. Современные графические платы оснащаются десятками и сотнями графических сопроцессоров, обеспечивающих фантастическую производительность за счет параллельного выполнения типовых процедур в задачах визуализации игровых сцен. Эти средства начинают проникать не только в индустрию игрового "софта", но и в системы виртуальной реальности, используемые в кино и на телевидении, в системы визуализации результатов научных экспериментов и т. д.

Задача 2.7

Задача нашей "игровой" программы существенно проще — она должна перетасовать колоду карт, которую предполагается использовать при реализации какой-нибудь карточной игры. В программах карточных игр предлагаются различные способы кодировки

карт. Мы остановимся на самом примитивном способе, когда каждой карте колоды присвоен порядковый номер, например, от 1 до 36 или от 1 до 52.

Идея алгоритма базируется на использовании датчика случайных чисел. Представим себе, что в нашем распоряжении имеется физический (типа лототрона) или программно реализованный датчик случайных чисел из диапазона `[1, N]`, где `N` равно 36 или 52. Первоначально наша колода напоминает фабричную упаковку карт, в которой все разложено по мастям и внутри каждой масти по возрастающим номиналам — 2, 3, 4, ..., 10, валет, дама, король, туз. Это соответствует тому, что коды карт образуют последовательность натуральных чисел — 1, 2, 3, ..., `N`. Обращаемся к датчику случайных чисел и получаем число `k`, принадлежащее диапазону `[1..N]`. Меняем местами первый элемент массива `A[1]` с элементом `A[k]`. Затем повторяем аналогичную операцию достаточно много раз, например 10 000. После этого первоначальный массив `A` окажется достаточно хорошо перемешанным. И если последовательность случайных чисел, выдаваемых генератором, окажется непредсказуемой, то такой алгоритм можно взять на вооружение.

В арсенале стандартных функций языка Паскаль имеется функция `random(M)`, которая при каждом новом обращении выдает очередной элемент последовательности равномерно распределенных случайных целых чисел из диапазона `[0, M-1]`. Ею мы и воспользуемся в новой программе (листинг 2.10).

Листинг 2.10. Программа `mixer`

```
program mixer;
const
  Nmax=36;
var
  A: array [1..Nmax] of byte;
  j,k,tmp: byte;
begin
  for j:=1 to Nmax do
    A[j]:=j;                {формируем фабричную "колоду"}
  for j:=1 to 10000 do
    begin
      k:=random(Nmax)+1;
      tmp:=A[1]; A[1]:=A[k]; A[k]:=tmp;
    end;
  for j:=1 to Nmax do       {цикл вывода перетасованной колоды}
    write(A[j]:4);
  readln;
end.
```

Обратите внимание на использование в программе константы `Nmax`. Если мы перейдем на колоду из 52 карт, то нам понадобится изменить единственную строчку в программе. Без константы `Nmax` такой переход потребовал бы изменения в четырех операторах.

Первый же запуск программы `mixer` приводит к вполне приличным результатам (рис. 2.10). Каждое число из диапазона `[1..36]` представлено по одному разу (т. е. дублирующихся карт нет), да и об их порядке тоже ничего плохого сказать нельзя.

```
Running "f:\fpc\myprog\mixer.exe "
  14   8  13   2  22   7  23  17  10  33  35  27  28  34  16  25  31   6   4  21
  20   9  11  36   3  29  30  18   5  12  24  19   1  26  15  32
```

Рис. 2.10. Структура перетасованной колоды карт

Но второй запуск программы `mixer` приводит к тем же результатам. Такого повтора в карточной игре терпеть нельзя. Игра с заранее известным порядком карт в колоде — это не игра. Причина кроется в работе системной функции `random()`. Когда она вызывается в первый раз, то в ее теле присутствуют две начальные константы, назовем их `x` и `y`. По вполне определенному алгоритму из них генерируется новое случайное число `z`. Перед тем как возвратить результат `z`, функция осуществляет пересылки `y→x` и `z→y`. Теперь становится понятным, что при повторном запуске программы функция `random()` начинает свою работу с тех же самых значений `x` и `y`, и последовательность "случайных" чисел снова повторяется.

Для того чтобы "возмутить" начальное состояние программы `random`, используется процедура `randomize`, которая, применяя различные ухищрения, непредсказуемым образом изменяет значения начальных констант `x` и `y`. Например, это можно сделать по текущему показанию системных часов, меняющих свое состояние каждую миллисекунду (листинг 2.11).

Листинг 2.11. Модификация программы `mixer`

```
program mixer;
const
  Nmax=36;
var
  A: array [1..Nmax] of byte;
  k,tmp: byte;
  j: integer;
begin
  randomize;                {возмущаем датчик случайных чисел}
  for j:=1 to Nmax do
    A[j]:=j;                {формируем фабричную "колоду"}
  for j:=1 to 10000 do
```

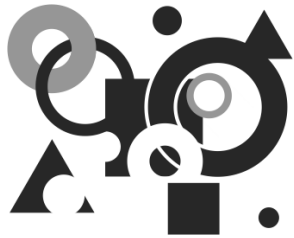
```
    begin
      k:=random(Nmax)+1;
      tmp:=A[1]; A[1]:=A[k]; A[k]:=tmp;
    end;
  for j:=1 to Nmax do       {цикл вывода перетасованной колоды}
    write(A[j]:4);
  readln;
end.
```

На рис. 2.11 приведены результаты пяти последовательных запусков модифицированной программы `mixer`. Теперь ее можно использовать при разработке карточных игр.

```
Running "f:\fpc\myprog\mixer.exe "
  31   6  35  15   3  34  33  25  26  23  21  18   9  14   1  27   2  17  11  12
  13  22  24  20  16  32  30  10  36   8   4   7  28  29  19   5
Running "f:\fpc\myprog\mixer.exe "
  22  12  11   4   3  10  13  24  33  27  35  29  21  15  14   8  34   9  19   7
  31  23  17  28  20  26  16  36   5  18   6   1  30  25  32   2
Running "f:\fpc\myprog\mixer.exe "
  20  16  31  22  11  17  36  32  18  12  19  21  27  24  13  14  23   2  30  29
  10  34  33   9  28   1   8  15  26   3  25  35   4   6   5   7
Running "f:\fpc\myprog\mixer.exe "
  36  16  27   6  15  25  24  13  23  20   8  30  14  18   9  21   7  19   3  11
  31  29  32   2  28  17  35  12  34  10  26   5   1  33  22   4
Running "f:\fpc\myprog\mixer.exe "
  26  13  36  10   8  30  24  19  32  27  15   3  29  35  28   6  21  17   1  14
  33  25  20  16   9   7   4  31  18  12  11  23   5   2  22  34
```

Рис. 2.11. Результаты работы модифицированной программы mixer

Глава 3



Интегрированная среда FP IDE

Интегрированная среда FP IDE, разработка которой продолжается и в настоящее время, обеспечивает режим работы, напоминающий условия программирования в средах Borland Pascal и Borland C++. После запуска программы fp.exe и набора первых строк новой программы на экране появляется главное окно (рис. 3.1).

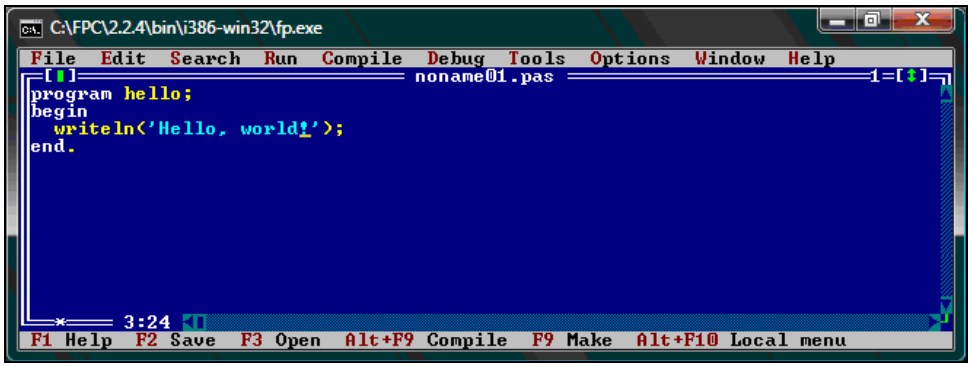


Рис. 3.1. Вид главного окна среды FP IDE

В верхней части расположена строка команд главного меню, выбор которых осуществляется либо с помощью указателя мыши, либо нажатием комбинации клавиш <Alt>+первая выделенная буква команды: <Alt>+<F> обеспечивает вход в меню **File** (Файл), <Alt>+<E> — в меню **Edit** (Редактирование) и т. д. В нижней части главного окна представлены наиболее часто исполняемые команды главного меню, для вызова которых достаточно нажать одну или две клавиши.

3.1. Главное меню интегрированной среды

Меню **File** (рис. 3.2) включает команды, с которых либо начинается, либо заканчивается сеанс работы с очередной программой. Большинство из этих команд знакомо пользователям по другим системам программирования.

Рис. 3.2. Команды меню **File**

- Команда **New** (Новый) подготавливает редактор к набору новой программы — чистит рабочее поле редактора, присваивает первой новой программе имя по умолчанию (noname01.pas) и переводит курсор в первую позицию первой строки. Координаты курсора отслеживаются редактором и отображаются в строке его состояния (левая нижняя часть поля редактора). Если во время сеанса пользователь несколько раз набирает новые программы, то им присваиваются последовательные имена — noname01.pas, noname02.pas и т. д. Для каждой из них создается новое окно редактора.
- Команда **New from template** (Новый по шаблону) предлагает использовать шаблон — некоторую заготовку для нового файла. Имеется возможность заполнить или откорректировать поля шаблона, после чего работа продолжается на поле редактора.
- Команда **Open** (Открыть) вызывает диалоговое окно для выбора и загрузки в поле редактора существующего файла.
- Команда **Reload** (Перезагрузить) производит перезагрузку текущего файла (т. е. файла, загруженного последним).
- Команда **Print** (Печать) выводит на принтер содержимое текущего окна редактора.
- Команда **Print setup** (Установки принтера) вызывает диалоговое окно для установки параметров принтера, формата бумаги и полей документа.
- Команда **Save** (Сохранить) сохраняет содержимое текущего окна редактора под текущим именем файла. Если файлу в поле редактора еще не присваивалось имя, отличное от системного, то при первом сохранении файлу необходимо присвоить уникальное имя.
- Команда **Save as** (Сохранить как) открывает диалоговое окно, в котором необходимо ввести имя, под которым будет запоминаться содержимое текущего окна редактора. Новое имя не обязательно должно совпадать с именем, под которым файл был открыт. Такая возможность позволяет хранить на диске несколько версий одной программы.
- Команда **Save all** (Сохранить все) сохраняет содержимое всех открытых окон редактора.

- Команда **Change dir** (Изменить каталог) открывает диалоговое окно, в котором можно выделить каталог, заменяющий текущий каталог, с текстом исходной программы.
- По команде **Command shell** (Командный процессор) происходит переход в режим командного процессора. После этого можно выполнить одну или несколько команд операционной системы. Выход из командного процессора по команде **Exit** возвращает управление FP IDE.
- Команда **Exit** (Выход) обеспечивает выход из IDE. Если в окнах редактора находятся несохраненные файлы, то перед выходом пользователю предлагают их запомнить.

Под командой **Exit** расположен список файлов, с которыми пользователь работал в последнее время. Выбор нужного файла из этого списка позволяет осуществить быструю загрузку.

Меню **Edit** (рис. 3.3) обеспечивает доступ к командам редактирования, большинство из которых знакомо всем пользователям. Однако реализации некоторых из них присущи особенности, редко встречающиеся в других программных продуктах.

В первую очередь это касается операций вырезания (команда **Cut**) и копирования (команда **Copy**) выделенного фрагмента текста. Соответствующий фрагмент текста из поля редактирования помещается в *локальный буфер обмена*, не имеющий ничего общего с *глобальным буфером* clipboard, который обслуживается операционной системой. Естественно, что команда вставки (**Paste**) выбирает фрагмент, расположенный в локальном буфере обмена. Содержимое последнего отображается по команде **Show clipboard** (Показать содержимое буфера обмена).

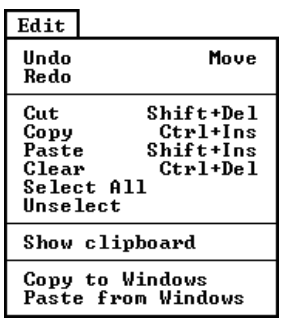


Рис. 3.3. Меню **Edit**

- Для обмена между полем редактора и системным буфером Windows в меню **Edit** предусмотрены две специальные команды — **Copy to Windows** (Копировать в Windows) и **Paste from Windows** (Вставить из Windows).
- Следующая новинка, которая носит, скорее, косметический характер, связана с командой отмены действия последней операции по редактированию **Undo** (Откат). Последовательность команд редактирования запоминается в специальном буфере, что позволяет осуществить возврат на один или более шагов. Особен-

ность, на которую мы обращаем ваше внимание, заключается в том, что в строке меню с командой **Undo** фиксируется характер отменяемого действия. На рис. 3.3 в этой строке содержится слово **Move** (Переместить), символизирующее, что последним действием на поле редактирования было перемещение выделенного фрагмента.

- Команда **Redo** (Отказ от отката) повторяет последнее действие по редактированию, которое было отменено командой **Undo**. Таким образом, возможен отказ от нескольких последних отмененных операций.
- Команда **Cut** (Вырезать) удаляет выделенный текст в поле редактора и копирует его в локальный буфер обмена. При этом прежнее содержимое буфера обмена пропадает. Вновь занесенная в буфер порция на следующем шаге редактирования еще может быть восстановлена. Локальный буфер обмена является собственностью FP, и для других приложений он не доступен.
- Команда **Copy** (Копировать) копирует выделенный фрагмент текста в локальный буфер обмена. Прежнее содержимое буфера при этом пропадает.
- Команда **Paste** (Вставка) вставляет содержимое локального буфера обмена в текущее окно редактора, начиная с текущей позиции курсора. Содержимое буфера обмена при этом сохраняется.
- Команда **Clear** (Удалить) удаляет выделенный фрагмент.
- Команда **Select All** (Выделить все) выделяет весь текст в текущем окне редактора. Выделенный текст может быть скопирован в локальный буфер обмена или удален.
- Команда **Unselect** (Отменить выделение) отменяет выделение.

Меню **Search** (Поиск) (рис. 3.4) обеспечивает доступ к диалоговым окнам поиска и замены, а также включает команды перехода к различным объектам программы и просмотра соответствующей информации.

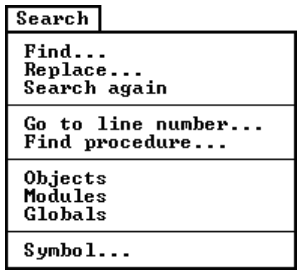


Рис. 3.4. Команды меню **Search**

- Команда **Find** (Найти) открывает стандартное диалоговое окно поиска (рис. 3.5). Искомый текст набирается в поле с пометкой **Text to find**. В полях, расположенных ниже, можно задать режим поиска:
  - поиск с игнорированием разницы между прописными и строчными буквами (**Case sensitive**);

- поиск полного слова (**Whole words only**);
- направление поиска — вниз (**Forward**) или вверх (**Backward**) от текущей позиции курсора;
- во всем тексте на поле редактора (**Global**) или только в выделенном фрагменте (**Selected text**);
- начиная с позиции курсора (**From cursor**) или с начала области видимости (**Entire scope**)

После нажатия кнопки **OK** начинается поиск в активном окне IDE. Если текст найден, то он подсвечивается.

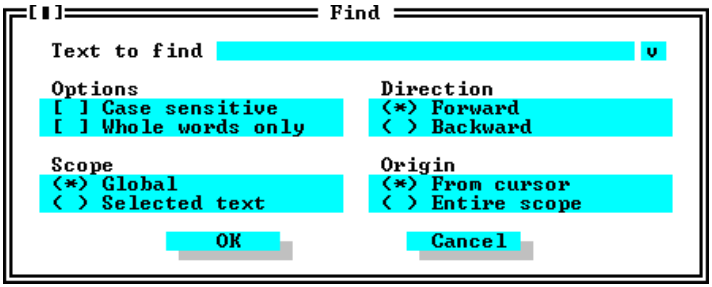


Рис. 3.5. Диалоговое окно поиска

◈ Команда **Replace** (Заменить) открывает стандартное диалоговое окно замены (рис. 3.6), в котором следует набрать заменяемый (**Text to find**) и замещающий (**New text**) тексты.

В дополнение к описанным ранее режимам поиска здесь добавляется возможность запроса подтверждения на очередную замену (**Prompt on replace**). Замена происходит в активном окне.

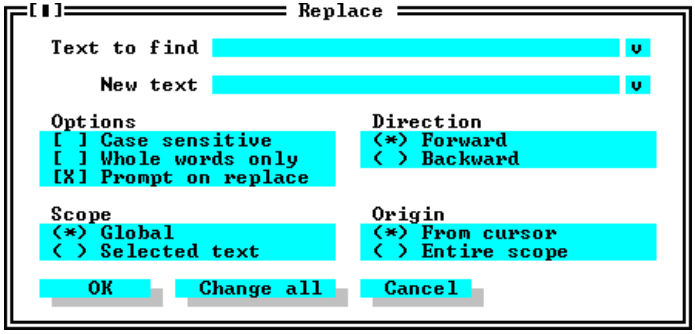


Рис. 3.6. Диалоговое окно замены

◈ Команда **Search again** (Найти еще раз) повторяет последний поиск или замену с прежними параметрами.

- Команда **Go to line number** (Перейти к строке номер) открывает диалоговое окно (рис. 3.7), в котором набирается номер нужной строки программы.

  После нажатия кнопки **OK** курсор в окне редактора переводится в начало указанной строки.

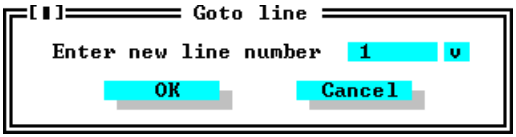


Рис. 3.7. Окно перехода на заданную строку

Если программа или модуль откомпилированы с информацией для просмотра, то в меню **Search** становятся доступными следующие команды:

- команда **Find procedure** (Найти процедуру) используется для перехода на начало текста процедуры или функции с указанным именем. В текущей версии IDE эта операция пока не реализована;
- команда **Objects** (Объекты) обеспечивает просмотр информации об указанном объекте;
- команда **Modules** (Модули) предназначена для просмотра информации об указанном модуле;
- команда **Globals** (Глобальные) позволяет просмотреть информацию о глобальной переменной с указанным именем;
- команда **Symbol** (Имя) предназначена для просмотра информации обо всех идентификаторах программы с выделением и отображением сведений о требуемом объекте.

В меню **Run** (Пуск) включены команды, необходимые для исполнения создаваемой программы (рис. 3.8).

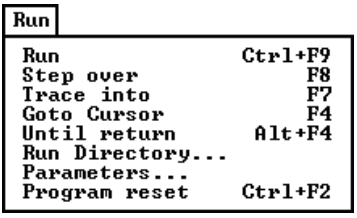


Рис. 3.8. Команды меню **Run**

Большинство команд являются традиционными для многих систем программирования. Среди них:

- команда **Run** (Пуск) реализует выполнение исходной программы, начиная с головного файла. Если текст в поле редактора был модифицирован, то его предварительно откомпилируют;

- команда **Step over** (Шаг без захода) выполняет код, соответствующий очередной строке исходной программы. Если текущей строкой является вызов процедуры, то процедура выполняется в автоматическом режиме, после чего восстанавливается пошаговый режим;
- команда **Trace into** (Шаг с заходом) выполняет код текущей строки. Если в нем находится вызов процедуры или обращение к функции, то выполнение строк вызванной подпрограммы осуществляется в пошаговом режиме;
- команда **Goto Cursor** (Пуск до курсора) осуществляет автоматическое выполнение программы до строки исходного текста, в которой находится курсор;
- команда **Until return** (До возврата) используется во время исполнения функции или подпрограммы в пошаговом режиме. Она позволяет выполнить текущую подпрограмму автоматически до момента выхода на вызывающую программу;
- команда **Run Directory** (Установка текущего каталога) позволяет на время выполнения программы установить нужный текущий каталог;
- команда **Parameters** (Параметры) вызывает диалоговое окно (рис. 3.9), в котором набираются параметры командной строки, используемые при запуске программы;

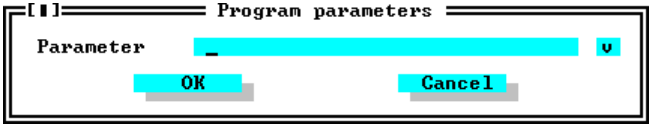


Рис. 3.9. Окно для набора параметров командной строки

- команда **Program reset** (Сброс программы) восстанавливает состояние программы, предшествовавшее ее запуску. Этой командой приходится пользоваться довольно часто в процессе отладки программы (выход из зависаний, сброс отладочного режима и т. п.).

Меню **Compile** (Компилировать) включает все команды, необходимые для компиляции программ и модулей (рис. 3.10).

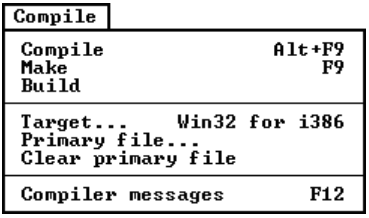


Рис. 3.10. Команды меню **Compile**

- Команда **Compile** (Компилировать) компилирует содержимое активного окна редактора независимо от того, является ли оно головной программой или нет.

- Команда **Make** (Собрать) компилирует содержимое активного окна, а также любые файлы и модули, состояние которых могло измениться с момента последней компиляции. Если головной файл был задан, то и он перекомпилируется заново.
- Команда **Build** (Пересобрать) компилирует содержимое активного окна и всех связанных с ним файлов и модулей независимо от того, изменялись они или нет с момента последней компиляции. Если головной файл был задан, то и он перекомпилируется заново.
- Команда **Target** (Исполнитель) используется для назначения операционной системы, под управлением которой должна работать компилируемая программа.
- Команда **Primary file** (Головной файл) используется для назначения головного файла. Если он задан, то команды компиляции или запуска работают с ним, а не с содержимым активного окна редактора. Головной файл может оказаться и незагруженным в среду IDE.
- Команда **Clear primary file** (Отменить головной файл) отменяет ранее назначенный головной файл. После этой команды головным считается файл, расположенный в активном окне редактора.
- Команда **Compiler messages** (Сообщения компилятора) отображает окно с сообщениями компилятора. В этом окне находятся сообщения, связанные с процессом компиляции самой последней программы.

Меню **Debug** (Отладка) (рис. 3.11) содержит команды, необходимые для отладки программы, такие как фиксация точек останова и задание контролируемых выражений.

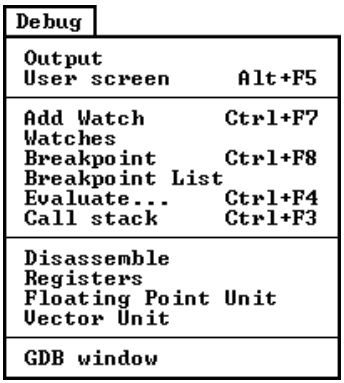


Рис. 3.11. Команды меню **Debug**

- По команде **Output** (Вывод) отображается окно, в котором представлены результаты, выводимые программой пользователя.
- По команде **User screen** (Экран пользователя) дисплей переключается на экран вывода работающей программы.

- По команде **Add Watch** (Добавить контролируемое выражение) в список контролируемых выражений можно добавить новое выражение, которое должно быть вычислено в IDE и отображено в специальном окне. Обычно в список контролируемых выражений включают имена переменных.
- По команде **Watches** (Контролируемые выражения) в отдельном окне отображается список всех контролируемых выражений, заданных пользователем.
- По команде **Breakpoint** (Точка останова) можно установить точку останова в текущей строке исходной программы и задать дополнительные условия, при соблюдении которых останов действительно произойдет.
- Команда **Breakpoint List** (Список точек останова) отображает в отдельном окне список назначенных точек останова и дополнительных условий, связанных с каждой из этих точек.
- По команде **Evaluate** (Вычислить) в специальном окне можно набрать выражение, которое будет вычислено в точке останова. Все операнды выражения в этой точке должны быть доступны.
- Команда **Call stack** (Стек вызовов) отображает список адресов незавершенных в текущей точке останова обращений к процедурам. При включении в компилируемую программу дополнительной информации показываются имена файлов и номера соответствующих строк.
- Команда **Disassemble** (Дизассемблирование) осуществляет переход в программу на языке ассемблера, которая создается в процессе компиляции исходной программы. Пошаговое выполнение этой программы с одновременной возможностью просмотра содержимого машинных регистров доступно программистам, владеющим языком ассемблера.
- Команда **Registers** (Регистры) отображает текущее содержимое регистров процессора (CPU).
- Команда **Floating Point Unit** (Сопроцессор) отображает текущее содержимое регистров сопроцессора (FPU).
- Команда **Vector Unit** (Векторный модуль) показывает текущее содержимое регистров MMX (или их эквивалента, если команды MMX эмулируются).
- По команде **GDB window** (Окно GDB) на экране появляется консоль отладчика GDB. Она может быть использована для интерактивной отладки. В этом режиме на отладочной консоли можно набирать команды GDB и видеть результаты их исполнения.

  Более подробно средства отладки анализируются в *разд. 3.4*.

Меню **Tools** (Инструменты) (рис. 3.12) включает команды управления некоторыми утилитами. Пользователь имеет возможность заменить любую из этих утилит или добавить новую.

Первая треть меню **Tools** управляет окном, в которое утилиты выводят свои сообщения. По команде **Messages** (Сообщения) это окно активизируется. С помощью команд **Goto next** (Перейти к следующему) и **Goto previous** (Перейти к предыдущему) можно перемещаться по сообщениям утилит.

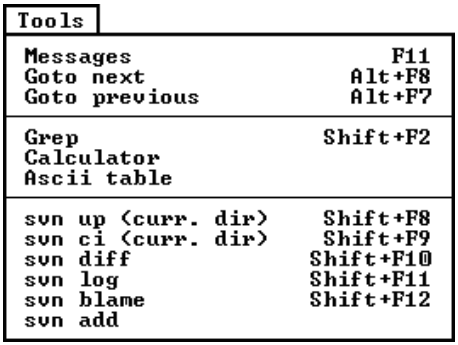


Рис. 3.12. Команды меню **Tools**

По команде **Grep** (Быстрый просмотр) выводится справка по регулярным выражениям — универсальному языку поиска информации в текстовых файлах. Из этого же окна может быть запущена программа расширенного поиска grep.exe, которую фирма Borland всегда включала в комплект своих систем программирования. С некоторыми элементами языка регулярных выражений многие пользователи встречались. Это специальные символы типа `*.pas` или `ab?.cpp`, используемые при поиске файлов (символ `*` представляет комбинацию отображаемых символов любой длины, в том числе и пустую строку, а символ `?` соответствует любому отображаемому символу). Регулярные выражения позволяют задать и более общие условия поиска, например, конструкция `[A-Z][0-9]` описывает все строки, содержащие прописные (заглавные) буквы и цифры, перемежающиеся в любом порядке. Сама программа grep.exe в поставку системы Free Pascal не входит, ее нужно найти и записать в один из каталогов, доступных в IDE.

Утилита **Calculator** (Калькулятор) запускает калькулятор (рис. 3.13), обладающий гораздо меньшими возможностями по сравнению с калькулятором Windows. Калькулятор выполняет действия над двумя операндами: содержимым окна локального дисплея и регистром, расположенным в памяти калькулятора. Список выполняемых операций приведен в табл. 3.1.

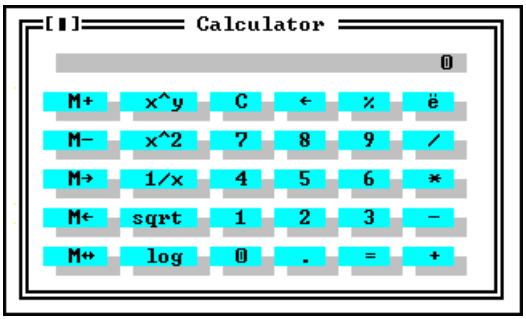


Рис. 3.13. Калькулятор системы FP

Таблица 3.1

Операция	Кнопка
Добавить отображенное число к регистру в памяти	**M+**
Вычесть отображенное число из регистра в памяти	**M-**
Переместить содержимое регистра памяти на дисплей	**M→**
Записать содержимое дисплея в регистр памяти	**M←**
Поменять местами содержимое регистра и дисплея	**M↔**

По команде **Ascii table** (Таблица ASCII) на экран выводится таблица всех символов и их ASCII-кодов (рис. 3.14).

Рис. 3.14. Таблица ASCII

Выделив в этом окне интересующий нас символ, мы получим в нижней части окна его десятичный и шестнадцатеричный коды. Любой символ из этого окна можно перенести в текущую позицию поля редактора одним из следующих способов:

- дважды щелкнув мышью по символу;
- нажав клавишу <Enter>, когда курсор мыши находится на нужном символе.

Это полезно в тех случаях, когда в состав текстовой константы нужно включить символы псевдографики. Окно с таблицей ASCII продолжает оставаться видимым до тех пор, пока не будет открыто новое служебное окно.

Раздел команд **svn** связан с использованием *системы управления версиями* (subversion), ориентированной на разработчиков больших программных комплексов. При этом неизбежно возникает большое количество версий различных файлов, каталогов, наборов данных. Для наведения порядка, связанного с хранением такого хозяйства, историей его создания, определением различий между версиями, возможностью отката на одну из точек разработки большого проекта, были приняты некоторые стандартные требования на идентификацию имен и версий файлов, а также на операции по регистрации всех наборов данных и программ в специальном хранилище — репозитарии. Одна из программ (TortoiseSVN), поддерживающих указанные наборы данных и операции по обмену с репозитарием, была использо-

вана авторами системы FP. Следы этой деятельности остались в главном меню, но на рядовых пользователей эти операции не распространяются. Тем более что программа управления версиями в стандартную поставку FP не входит.

Меню **Options** (Параметры) (рис. 3.15) предоставляет коллекцию диалоговых окон для настройки параметров всех компонентов системы Free Pascal и среды IDE. Параметров этих довольно много, и выбор тех или иных значений требует основательных знаний о влиянии каждой характеристики на изменение режима работы системы в целом.

Достаточно подробные сведения по настройке параметров компилятора и интегрированной среды приведены в *приложении 2*.

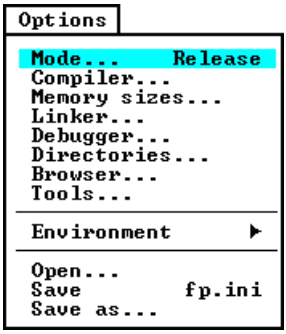


Рис. 3.15. Команды меню **Options**

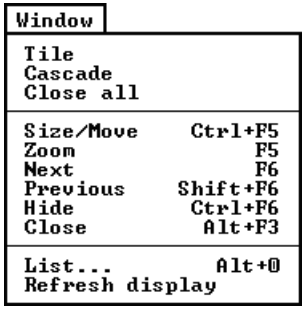


Рис. 3.16. Команды меню **Window**

Меню **Window** (Окно) (рис. 3.16) включает команды по управлению окнами интегрированной среды.

- Две первые команды — **Tile** (Плитка) и **Cascade** (Каскад) — устанавливают один из вариантов отображения открытых окон. В первом случае площадь экрана делится на несколько равных частей, в каждой из которых отображается по одному окну (в некоторых программных системах такое расположение описывают термином *"черепица"*). Во втором случае окна выстраиваются друг за другом подобно колоде карт, выставляя напоказ только свои заголовки.
- По команде **Close all** (Закрыть все) закрываются все открытые окна.
- Команда **Size/Move** (Размер/Перемещение) включает режим, при котором можно изменить размеры активного окна или переместить его, используя клавиши-стрелки управления курсором.
- Команда **Zoom** (Масштаб) включает режим, при котором последовательные нажатия клавиши <F5> то увеличивают размеры окна до максимально возможного, то возвращают его в обычное состояние.
- Команды **Next** (Следующее) и **Previous** (Предыдущее) обеспечивают переход по очереди открытых окон в прямом или обратном направлениях.
- Команда **Hide** (Скрыть) скрывает активное окно.
- По команде **Close** (Закрыть) активное окно закрывается.

- Команда **List** (Список) отображает перечень открытых окон.
- По команде **Refresh display** (Освежить дисплей) осуществляется перерисовка экрана.

Меню **Help** (Помощь) включает команды входа в оглавления файла помощи и перемещения по кадрам его разделов (рис. 3.17).

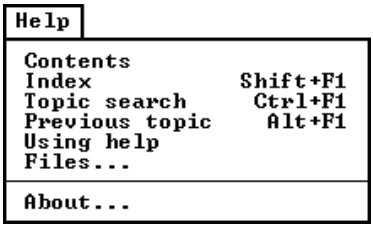


Рис. 3.17. Команды меню **Help**

Следует отметить, что файл помощи в составе систем FP и IDE не поставляется. Он состоит из огромного количества файлов, преимущественно в формате HTML, поставляемых автономно в виде zip-архива. После его распаковки создается каталог, содержащий более 10 000 файлов общим объемом порядка 49 Мбайт. В таком виде его еще нельзя использовать. Необходимо выполнить команду **Help → Files** и указать в открывшемся окне путь к каталогу HTML. Затем надо создать оглавление архива — файл с именем fpctoc.htx — и подключить его к среде IDE. Время выполнения этой процедуры — порядка 10 минут. Только после этого содержание файла помощи (**Contents**) и указатель терминов (**Index**) у вас будут не пусты. Кроме обычных кадров помощи небольшого размера, посвященных тому или иному термину, в вашем распоряжении окажутся несколько руководств:

- руководство пользователя (**User's guide**);
- руководство программиста (**Programmer's guide**);
- справочник по языку Паскаль (**Pascal Language Reference guide**);
- справочник по параметрам командной строки и ключам компилятора (**Command-line options and switches Reference chart**);
- справочник по программам и данным библиотеки периода выполнения (**Run-Time Library reference manual**).

3.2. Редактирование текста программы

Текстовый редактор, включенный в состав IDE, предназначен, главным образом, для набора и корректирования текстов исходных программ. В этом плане он работает так же, как и большинство текстовых редакторов, поэтому в пояснении нуждаются лишь некоторые моменты.

3.2.1. Режим вставки

Обычно IDE работает в режиме вставки. Это означает, что набираемый текст вставляется перед текстом, который находится правее позиции курсора. В альтернативном режиме набираемый текст заменяет существующий текст. В режиме вставки курсор выглядит как горизонтальная мерцающая черточка, в режиме замены — как мерцающий прямоугольник. Переключение режимов происходит при нажатии либо клавиши <Insert>, либо комбинации клавиш <Ctrl>+<V>.

3.2.2. Блоки

IDE поддерживает работу с выделенными фрагментами точно так же, как это принято в среде Turbo Pascal. Это слегка отличается от того, как с такими данными работают приложения Windows.

Текст можно выделить тремя способами:

- используя мышь, вы перемещаете ее с зажатой левой кнопкой над выделяемым текстом;
- используя клавиатуру, вы нажимаете комбинацию клавиш <Ctrl>+<K><B> для маркировки начального символа выделяемого фрагмента и <Ctrl>+<K><K> для маркировки последнего символа;
- используя клавиатуру, вы зажимаете клавишу <Shift> и перемещаете курсор клавишами-стрелками к концу выделяемого фрагмента.

Кроме этого можно использовать клавишные команды для выделения текущего слова (<Ctrl>+<K><T>) и текущей строки (<Ctrl>+<K><L>).

В FP IDE выделенный текст сохраняет свое состояние и после вывода курсора за пределы отмеченного фрагмента, который с этого момента именуется *блоком*.

Над блоком могут быть выполнены следующие команды:

- перемещение блока в позицию, отмеченную курсором (<Ctrl>+<K><V>);
- копирование блока в позицию, отмеченную курсором (<Ctrl>+<K><C>);
- удаление блока (<Ctrl>+<K><Y>);
- запись блока в файл (<Ctrl>+<K><W>);
- чтение содержимого файла в блок (<Ctrl>+<K><R>). Если в поле редактора в этот момент существует выделенный блок, то он считанным набором не заменяется. Файл считывается в позицию курсора, после чего прочитанная порция выделяется;
- отступ блока вправо (<Ctrl>+<K><I>);
- выступ блока влево (<Ctrl>+<K><U>);
- печать содержимого блока.

Если выполняется поиск или замена, то область операции ограничивается содержимым блока.

3.2.3. Установка закладок

IDE позволяет сделать закладку в текущей позиции курсора. Максимальное количество одновременно существующих закладок — 9. Каждой из них присваивается номер с помощью нажатия комбинации клавиш <Ctrl>+<K>+<номер>. Для возврата на ранее сделанную закладку используется комбинация клавиш <Ctrl>+ +<Q>+<номер>.

ЗАМЕЧАНИЕ

После выхода из IDE закладки не сохраняются.

3.2.4. Подсветка синтаксиса

В IDE предусмотрена подсветка синтаксических конструкций Паскаля с раскраской элементов различных групп в соответствующие цвета. Включение или отключение раскраски синтаксических конструкций производится в окне, появляющемся в результате выполнения цепочки команд **Options → Environment → Editor** (см. рис. 3.18, поле **Syntax highlight**).

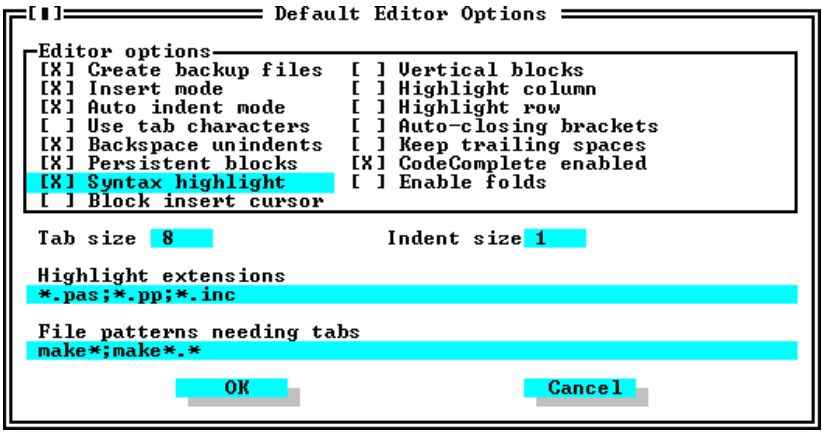


Рис. 3.18. Настройки редактора

Синтаксические конструкции языка, раскрашиваемые в разные цвета, разбиты на следующие группы:

- **Whitespace** — так называемые белые пробелы. Они соответствуют пробелам между словами, и для них используется цвет фона;
- **Comments** — все виды комментариев в языке Free Pascal;
- **Reserved words** — все зарезервированные слова языка;
- **Strings** — константные строковые выражения;
- **Numbers** — десятичные числа;

- **Hex numbers** — шестнадцатеричные числа;
- **Assembler** — вставки на ассемблере;
- **Symbols** — имена переменных, типов;
- **Directives** — директивы компилятора;
- **Tabs** — табуляторные пропуски (Tab) могут быть окрашены цветом, отличным от цвета фона.

3.2.5. Автоматическое завершение слов

Работая в режиме автоматического формирования служебных слов, редактор пробует догадаться о возможных окончаниях после распознавания нескольких первых символов, набранных пользователем (рис. 3.19), и предлагает наиболее подходящее полное слово из специально подготовленной таблицы. Если этот выбор совпадает с вашими намерениями, достаточно нажать клавишу <Enter>.

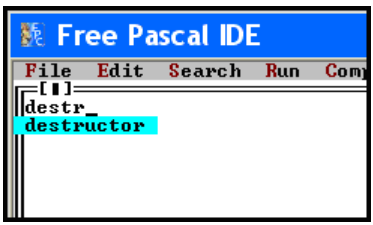


Рис. 3.19. Автоматическое завершение слова

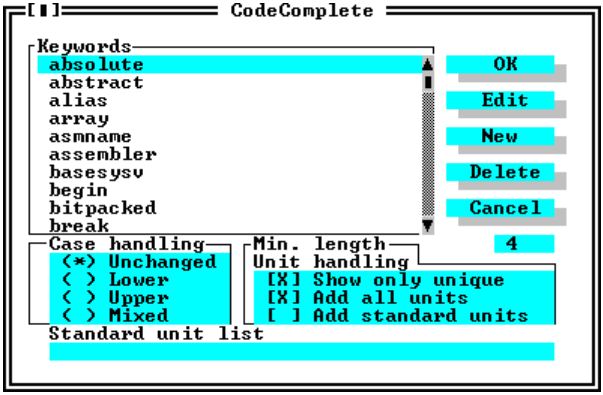


Рис. 3.20. Таблица автоматически завершаемых слов

Пользователь имеет возможность пополнять или модифицировать таблицу служебных слов, за набором которых редактор автоматически следит. Для этого нужно выполнить команду **Options → Environment → CodeComplete**. В появляющемся окне в алфавитном порядке представлен список слов. При нажатии кнопки **OK** список сохраняется и диалоговое окно закрывается. При нажатии

кнопки **Edit** вы можете отредактировать подсвеченное ключевое слово. По кнопке **New** можно добавить новое слово к списку. Нажимая кнопку **Delete**, вы удаляете из списка подсвеченное слово. По кнопке **Cancel** все сделанные изменения отменяются и диалоговое окно закрывается.

Изменения, сделанные в списке слов, станут доступными при следующем запуске IDE.

3.2.6. Шаблоны кода

Шаблоны кода используются для вставки в набираемый текст заранее сформированных заготовок. Каждой заготовке должно быть присвоено уникальное имя. Например, с именем `ifthen` может быть связан следующий фрагмент кода:

```
If | Then
begin
end
```

Такой фрагмент может быть вставлен в набираемый текст либо после набора соответствующего имени, либо после клавишной команды <Ctrl>+<J>, если курсор установлен справа от имени шаблона. Если перед курсором отсутствует имя шаблона, то появляется всплывающее окно для выбора нужного шаблона. Если в шаблоне обнаружен символ `|`, то курсор останавливается на нем, символ удаляется, освобождая место для набора нестандартной вставки. В приведенном выше примере между словами `If` и `Then` должно быть вставлено условное выражение.

Шаблоны кодов могут добавляться и редактироваться в окне, появляющемся по команде **Options → Environment → CodeTemplates** (рис. 3.21).

Назначение кнопок в этом окне достаточно очевидно. Все новые и измененные шаблоны кода станут доступными при следующем старте IDE.

Замечание

Дублирование имен шаблонов кода не допустимо.

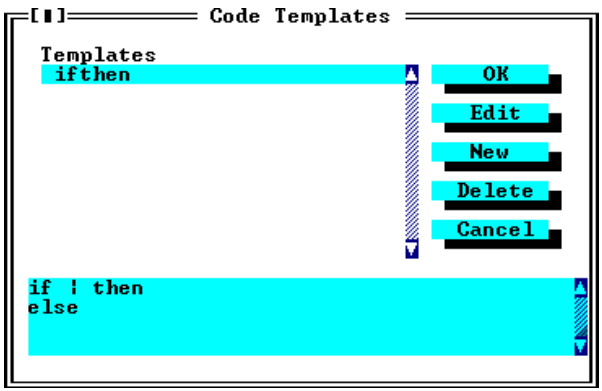


Рис. 3.21. Окно для набора шаблонов

3.3. Выполнение программы

Откомпилированная программа может быть запущена из IDE. Это можно сделать одним из следующих способов:

- выполнить команду меню **Run → Run**;
- нажать комбинацию клавиш <Ctrl>+<F9>.

Если программе должны быть переданы параметры командной строки, то для этого надо выполнить команду меню **Run → Parameters**. Диалоговое окно по заданию параметров командной строки представлено на рис. 3.9.

После старта программа выполняется автоматически до тех пор, пока не происходит одно из следующих событий:

- нормальное завершение программы;
- возникновение ошибки;
- обнаружение точки останова;
- прерывание пользователем выполнения программы.

Последняя альтернатива возможна в том случае, если программа откомпилирована с отладочной информацией.

Еще одна возможность связана с остановом программы по достижении строки исходной программы, в которой перед стартом находился курсор. Это можно сделать одним из следующих способов:

- выполнить команду меню **Run → Goto Cursor**;
- нажать клавишу <F4>.

Опять же, это возможно в том случае, если программа откомпилирована с отладочной информацией.

Имеется возможность выполнения программы строка за строкой. При этом нажатие клавиши <F8> приводит к выполнению очередной строки исходной программы. Если программа еще не стартовала, то нажатие клавиши <F8> запускает программу, в которой выполняется ее первая строка и происходит останов, после чего в окне редактора IDE отображается выполненная строка. В том случае, если очередной строкой оказывается вызов процедуры, то останов происходит после автоматического завершения процедуры, а затем и возврат из нее на следующую после вызова строку. Для пошагового выполнения вызываемой процедуры надо нажать клавишу <F7>.

Если мы находимся в режиме пошагового выполнения процедуры, то по команде меню **Run → Until return** происходит автоматическое выполнение процедуры до оператора выхода из нее.

Если выполнение программы надо прекратить до ее останова, то можно:

- либо выполнить команду меню **Run → Program reset**;
- либо нажать комбинацию клавиш <Ctrl>+<F2>.

3.4. Отладка программ

Основные средства отладки достаточно консервативны. Еще на ЭВМ первого поколения программисты набирали на пульте адрес команды, на которой автоматическое выполнение программы прекращалось, и появлялась возможность просмотреть содержимое машинных регистров и ячеек оперативной памяти. Вторым магическим средством был перевод компьютера в пошаговый режим работы, в котором очередное нажатие кнопки <ПУСК> приводило к выполнению следующей команды программы. На некоторых ЭВМ была предусмотрена возможность останова работы программы в момент записи данных в ячейку с указанным адресом. Сегодня кодами машинных команд пользуются очень редкие профессионалы, да и те предпочитают более продвинутые средства вроде услуг ассемблера. Большинство пользователей работает с алгоритмическими языками высокого уровня. Однако старинные средства отладки сохранились в несколько модернизированном виде.

Так называемые *точки останова* (breakpoints) теперь задаются не по адресам машинных команд, а на входе во фрагмент исполняемой программы, соответствующий началу указанной строки исходного текста. Вместо просмотра содержимого ячеек оперативной памяти появилась возможность вывода на дисплей значений переменных и даже формул, заданных в обычном для программиста виде. Контролируемые таким образом выражения в англоязычной литературе обозначают терминами *watches*. Вместо однотактного режима выполнения программы по одной машинной команде появилась возможность "простукивать" исходную программу по одной строке. Конечно, каждое из этих новшеств обрастает дополнительными деталями, но, в целом, процесс развития средств отладки сохранился на первобытном уровне. Основные средства отладки в среде FP IDE сосредоточены в меню **Run** и **Debug** (рис. 3.22).

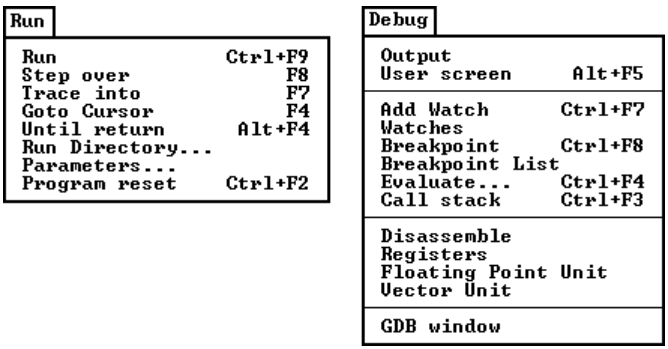


Рис. 3.22. Отладочные команды системы в среде IDE Free Pascal

Начнем с возможности прерывания автоматического режима выполнения программы. По команде **Run → Run** (клавишный аналог — <Ctrl>+<F9>) программа начнет выполняться в автоматическом режиме, и при достаточно объемной выдаче

результатов на экран мы ничего путного увидеть не успеем. Конечно, имеется возможность подменить стандартный вывод записью в файл, запуская программу с параметром командной строки:

```
>prog.exe >1.txt
```

В этом случае мы сможем прочитать из файла с именем 1.txt все результаты, выданные программой, но обнаружить причины неправильной работы программы таким способом мы не сможем. Желательно остановить работу программы в подозрительной точке и проанализировать значения наиболее важных переменных. А потом продолжить автоматическую работу до следующей точки останова. Для назначения *первой* точки останова можно воспользоваться командой **Goto Cursor** (клавишный аналог — <F4>) из меню **Run**. Предварительно нужно установить курсор в поле редактора в ту строку, перед выполнением которой мы хотим прервать работу программы.

После выхода на первый останов и просмотра нужных данных у нас появляются следующие возможности — продолжить работу в автоматическом режиме или перейти на пошаговое выполнение программы. В первом случае можно воспользоваться командой **Continue** из меню **Run** (клавишный аналог — <Ctrl>+<F9>), но тогда дальнейшее выполнение программы будет до конца только автоматическим. А можно опять перевести курсор на строку следующего останова и снова воспользоваться командой **Goto Cursor**. Конечно, такой выбор следующей точки останова нельзя признать удовлетворительным. Существует и более разумный подход, но о нем — далее.

После выхода в точку останова можно продолжить работу программы в пошаговом режиме. Для этой цели предназначены клавишные команды <F8> (аналог команды **Step over**) и <F7> (аналог команды **Trace into**). Каждое нажатие одной из этих клавиш приводит к выполнению очередной строки в исходной программе. Если выполняемая строка содержала обращение к процедуре, то по нажатию клавиши <F8> процедура будет выполнена в автоматическом режиме, а после возврата из процедуры сохранится пошаговый режим выполнения программы. Такой процесс целесообразен, когда вы уверены в правильности работы процедуры. Если у вас появились сомнения, то и процедуру разумно "простучать" в пошаговом режиме. В этом случае надо воспользоваться клавишей <F7>. Наконец, существует ситуация, когда, выполнив несколько шагов в процедуре, вам захотелось заставить ее доработать в автоматическом режиме. На этот случай вам предлагается команда **Until return** (клавишный аналог — <Alt>+<F4>).

Более рациональный способ задания точек останова заключается в том, что вы устанавливаете их заранее или добавляете при выходе в ту или иную точку программы. Для того чтобы сделать текущую строку точкой останова, достаточно выполнить клавишную команду <Ctrl>+<F8> (аналог команды **Breakpoint** из меню **Debug**). При этом текущая строка окрасится в красный цвет. Повторное нажатие той же комбинации клавиш отменит точку останова в текущей строке, и ее цвет станет нормальным. За один присест можно набрать несколько точек останова.

Список всех точек останова, установленных к текущему моменту, можно увидеть в окне, которое появляется по команде **Breakpoint List** из меню **Debug** (рис. 3.23).

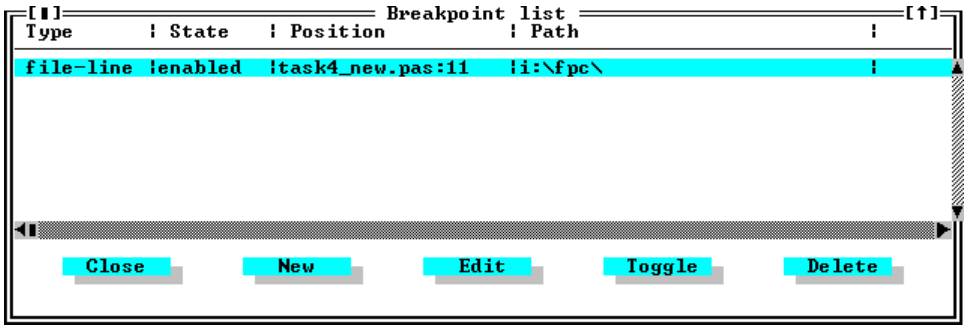


Рис. 3.23. Список точек останова

Новые точки останова можно добавлять, редактируя строки в этом окне. Вторая колонка позволяет, не удаляя точку останова из списка, сделать ее пассивной (**Disabled**) — теперь останов на ней не произойдет. Для этой цели используется кнопка **Toggle** (Переключить). Повторное нажатие этой кнопки меняет статус точки останова на противоположный. Полное удаление точки останова осуществляется с помощью кнопки **Delete**.

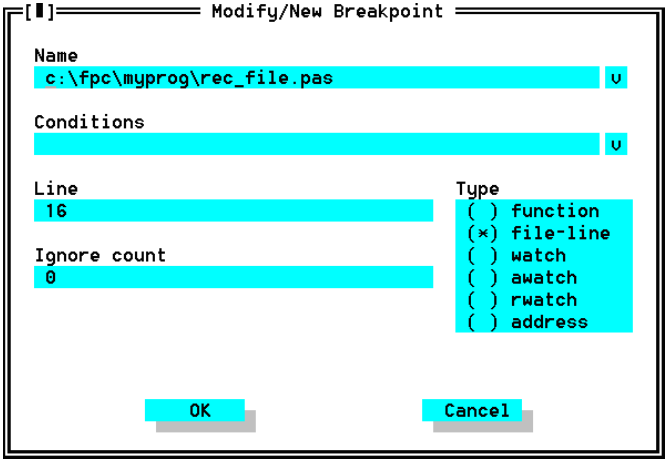


Рис. 3.24. Дополнительные условия останова

Нажатие кнопки **Edit** открывает диалоговое окно (рис. 3.24), в котором можно оговорить дополнительные условия останова в той или иной точке. Точка останова, установленная обычным способом (<Ctrl>+<F8>), заставляет компьютер прервать выполнение программы при каждом попадании в начало соответствующей строки исходной программы. Представьте себе ситуацию, когда вы установили точку ос-

танова на внутренней строке цикла, который должен выполняться сотни раз. Не устать бы, заставляя программу продолжаться после каждого останова. Ошибки в циклах чаще всего возникают либо при первом, либо при последнем прохождении цикла. Дополнительно можно проконтролировать условие досрочного завершения цикла.

Поэтому условия фактического останова в данной точке программы можно дополнить некоторыми проверками. К таким возможностям относятся:

- задание номера прохождения через данную строку (номер набирается в нижней строке диалогового окна);
- задание логического условия, при выполнении которого должен произойти останов.

Логическое условие останова записывается в строке, над которой расположена надпись **Conditions**. Формат записи логического выражения такой же, каким вы пользуетесь в операторе `if`. С помощью этого средства вы можете установить ситуацию, когда, например, в данном операторе в переменную `x` записывается недопустимое (с точки зрения алгоритма) значение:

```
x <= 0
```

Нулевое значение счетчика проходов через заданную точку останова означает, что останов должен произойти при каждом попадании на заданную строку программы (именно этой ситуации соответствует надпись на рис. 3.24 — **Ignore count**).

Если в точке останова одновременно заданы дополнительные условия останова по счетчику и по истинности логического выражения, то для фактического останова должны выполниться оба условия.

Наличие в программе нескольких точек останова позволяет перемещаться между ними в автоматическом режиме, что повышает производительность отладки.

Теперь о возможностях просмотра значений переменных и выражений в точках останова. Первая из них заключается в том, что предварительно с помощью команды **Add Watch** из меню **Debug** мы набираем нужные имена и формулы в открывающемся окне (рис. 3.25).

В момент останова с помощью команды **Watches** из меню **Debug** в нижней части экрана открывается окно, в котором представлены все набранные выражения и значения тех из них, которые можно было сосчитать в этой точке.

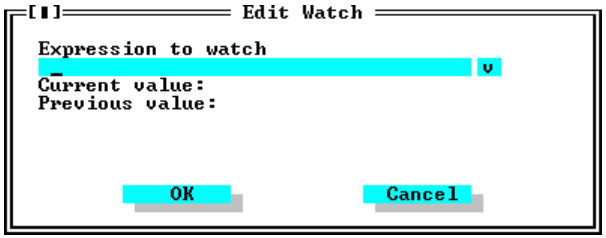


Рис. 3.25. Окно для набора контролируемого выражения

Вторая возможность оценить значение выражения, не включенного в список **Watches**, предоставляется командой **Evaluate** (Вычислить). Соответствующее диалоговое окно представлено на рис. 3.26.

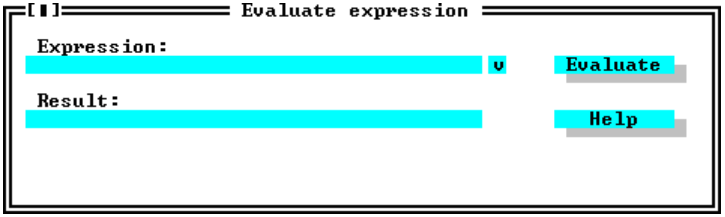


Рис. 3.26. Окно для вычисления значения выражения

Операнды выражения, набираемого в поле **Expression**, должны быть доступны в текущей точке останова.

Дополнительным средством отладки является использование системной процедуры `Assert` (от англ. *assert* — утверждать), допускающей один из двух следующих форматов вызова:

```
Assert(условное_выражение);
Assert(условное_выражение, сообщение);
```

Если в момент выполнения процедуры `Assert` условное выражение принимает значение `True`, то работа процедуры завершается без каких-либо дополнительных действий и выполнение программы продолжается. В случае нарушения заданного условия процедура `Assert` прекращает выполнение программы и выдает либо системное сообщение, либо сообщение, предусмотренное вторым аргументом в обращении. Процедура `Assert` функционально дублирует действия, возникающие в точке останова, нагруженной дополнительным логическим условием. Разница только в том, что точка останова переводит программу в режим ручного изучения возникшей ситуации. Использование процедуры `Assert` не приводит к таким задержкам и позволяет включить в окно вывода программы дополнительное смысловое сообщение по поводу происшедшего события. Кроме того, несколько одинаковых обращений к процедуре `Assert` можно включить в конце каждого участка программы, подозреваемого в нарушении условий правильного функционирования алгоритма.

Обращения к процедурам `Assert` будут проигнорированы компилятором, если вы не включите в текст своей программы соответствующую разрешающую директиву (листинг 3.1).

Листинг 3.1. Использование процедуры `Assert`

```
program assert1;
{$ASSERTIONS ON}
```

```
var
  x : integer=1;
  y : integer=1;
begin
  x := x-1;
  assert(x<>0,'Значение x равно 0');
  y := y div x;
  writeln(y);
  readln;
end.
```

Окно вывода этой программы содержит строку с предусмотренным сообщением и координатами точки, в которой было обнаружено нарушение условия:

```
Running "c:\fpc\myprog\assert1.exe "
Значение x равно 0 (Assert1.pas, line 8).
```

Дополнительно выдается сообщение в диалоговом окне, представленном на рис. 3.27.

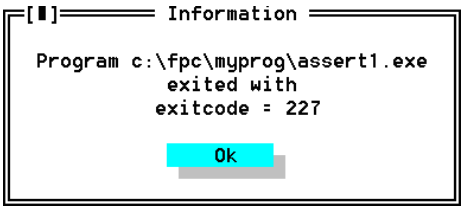


Рис. 3.27. Сообщение процедуры `Assert`

Если бы в обращении к процедуре `Assert` отсутствовал второй аргумент, то в окне вывода программы появилось бы системное сообщение:

```
Assertion failed (Assert1.pas, line 8).
```

Для отладки программы ее следует компилировать с отладочной информацией. Это происходит в следующих случаях:

- программа выполняется шаг за шагом;
- программа запускается до попадания в одну из точек останова;
- во время выполнения программы осуществляется контроль за содержимым переменных или ячеек памяти.

3.4.1. Использование точек останова

Точки останова прерывают выполнение программы, когда достигается одна из установленных точек останова. В этот момент управление передается IDE, после чего выполнение программы может быть продолжено.

Для набора точки останова в текущей строке исходной программы можно выполнить команду **Debug → Breakpoint** или нажать комбинацию клавиш <Ctrl>+ +<F8>. Окно, в котором хранится информация о точке останова, приведено на рис. 3.28.

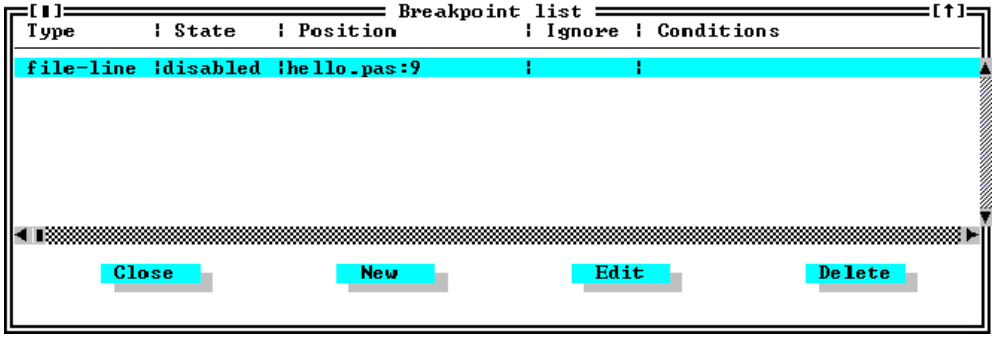


Рис. 3.28. Окно со списком точек останова

Список точек останова можно увидеть, выполнив команду **Debug → Breakpoint List**. В этом окне можно инициировать следующие операции:

- **New** — отобразить свойства точки останова при наборе новой точки;
- **Edit** — показать свойства точки останова для изменения ее свойств;
- **Delete** — удалить точку останова.

При наборе новой точки или изменении свойств ранее установленной точки используются поля окна, приведенного на рис. 3.29.

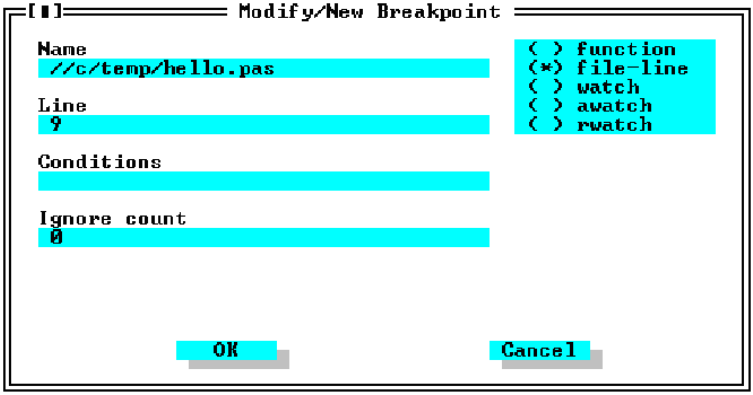


Рис. 3.29. Окно для набора новых точек останова и модификации старых

Вы можете установить тип точки, выбрав из списка типов нужную строку:

- **function** — точка останова в функции. Выполнение программы останавливается при вызове функции с указанным именем;

- **file-line** — точка останова в строке исходной программы. Программа останавливается при попадании на указанную строку в файле с заданным именем;
- **watch** — точка останова выражения. Вы можете ввести выражение, и программа остановится при изменении значения этого выражения;
- **awatch** (access watch) — точка останова выражения. Можно ввести выражение, которое является ссылкой на ячейку памяти, и программа остановится при любом обращении к этой ячейке (при записи или чтении);
- **rwatch** (read watch) — точка останова выражения. Останов происходит при чтении из указанной ячейки.

В окне **Modify/New Breakpoint** есть также поля:

- **Name** — имя функции или файла, в котором должен произойти останов;
- **Line** — номер строки, в которой должен произойти останов (только для точек останова типа **file-line**);
- **Conditions** — здесь указывается условие, при вычислении которого для останова получится значение `True`. Формула условия должна быть набрана прописными буквами;
- **Ignore count** — количество прохождений через заданную точку без останова. После достижения указанного значения происходит останов.

3.4.2. Контролируемые выражения

Контролируемые выражения могут использоваться, если программа компилируется с отладочной информацией. Их значения вычисляются IDE и отображаются в отдельном окне. Когда происходит останов программы (например, в точке останова), текущие значения контролируемых выражений можно увидеть в этом окне.

Набор новых контролируемых выражений производится по команде **Debug → Add Watch** или с помощью нажатия комбинации клавиш <Ctrl>+<F7>. При этом появляется диалоговое окно, в котором видны и все ранее набранные выражения. Так как IDE использует GDB, то формулы должны набираться прописными буквами.

Список всех контролируемых выражений и их текущие значения отображаются в окне, открывающемся по команде **Debug → Watches**. По нажатию клавиши <Enter> или <Пробел> отображается текущее значение подсвеченного выражения.

3.4.3. Стек обращений

Стек обращений позволяет проследить динамику выполнения программы. В нем в обратном порядке отображены названия вызванных и еще не завершенных к моменту останова процедур. Для просмотра стека надо выполнить команду **Debug → Call stack**. Здесь будут показаны имена и адреса всех активных процедур. Если им переданы параметры, то их значения тоже отображаются. При нажатии клавиши

<Пробел> на выделенной строке этого окна в поле редактора подсвечивается соответствующая строка исходного файла.

3.4.4. Окно GDB

Окно GDB обеспечивает непосредственную связь с отладчиком. В этом окне набираются команды, адресованные GDB, здесь же отображаются ответные сообщения. Более подробную информацию можно найти в руководстве по GDB.

3.5. Настройка среды и системы (предварительные сведения)

В начальной стадии знакомства со средой Free Pascal мы ограничимся минимальным набором сведений, необходимых для эксплуатации системы. Более подробная информация приведена в *приложении 2*.

Первый шаг настройки среды заключатся в формировании и подключении файлов справочной информации. Если вы решили установить FP IDE в каталоге c:\FPC, то убедитесь, что в файле fp.ini, расположенном в каталоге c:\FPC\2.4.0\bin\i386-win32, правильно указан путь для поиска разделов помощи. Секция `Help` в этом файле должна иметь вид:

```
[Help]
Files="C:\FPC\2.4.0\html\fpctoc.htx|HTML Index"
```

Номер версии вашей среды FP IDE (в приведенной выше строке указана версия `2.4.0`) может быть и более свежим. Но содержимое файлов помощи обновляется гораздо реже, чем происходит смена версий. Поэтому справочная информация, приведенная на диске, прилагаемом к книге, еще долго не устареет.

Если строка `Files=...` соответствует месту расположения файлов помощи и файл с оглавлением всех справочных кадров был построен (файл fpctoc.htx), то при входе в команду **Help → Index** должно появиться окно, верхняя часть которого представлена на рис. 3.30.

Рис. 3.30. Начало индексного файла, открываемого командой **Help → Index**

Если это окно появилось пустым, то либо неправильно указана строка в секции `Help` файла fp.ini, либо индексный файл организован по неверным ссылкам. Для перестройки индексного файла вам придется воспользоваться цепочкой команд **Help → Files... → New**.

Начиная очередной сеанс работы в интегрированной среде, вы должны убедиться, что установлен нужный вам режим работы компилятора. Для этого необходимо взглянуть на состояние команды **Options → Mode** (рис. 3.15). К команде **Mode** приходится прибегать довольно часто, т. к. в процессе разработки программы надо пользоваться отладочным режимом работы компилятора (**Debug**). Именно в этом режиме компилятор формирует вспомогательные таблицы и делает различные вставки в программу, позволяющие использовать отладочные средства. После завершения отладки следует перейти в режим **Release**, обеспечивающий изготовление программы без лишних вставок. По команде **Mode** открывается диалоговое окно, представленное на рис. 3.31.

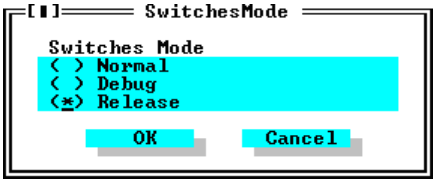


Рис. 3.31. Окно установки режима компиляции

Вообще говоря, каждый пользователь может настроить среду на удобный для него режим работы, заказав те или иные опции в панелях настройки компилятора. Для этого целесообразно сделать нужные установки в режиме **Normal** и переключаться в него единственным указанием ключа в окне **SwitchesMode**.

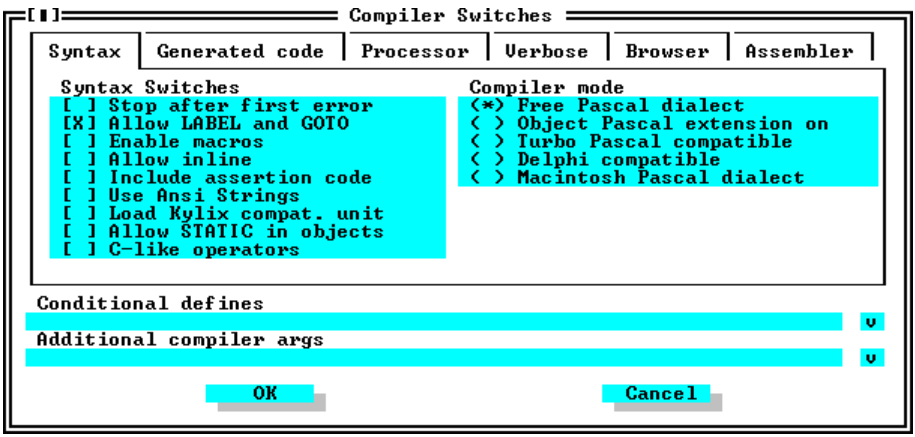


Рис. 3.32. Окно ключей компилятора

Еще одна важная характеристика режима работы компилятора связана с выбором того диалекта входного языка, на котором написана ваша программа. Эта установка производится в диалоговом окне, которое появляется в результате выполнения команды **Options → Compiler** (рис. 3.32). В списке переключателей **Compiler mode** вы должны поставить звездочку на нужной строке.

Убедитесь, что на вкладке **Processor** (рис. 3.33) выделен тип компьютера, для которого предназначена создаваемая вами программа.

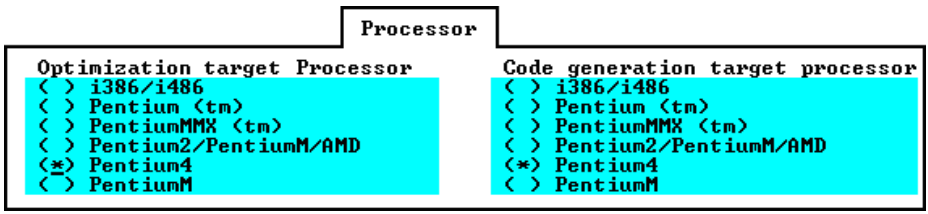


Рис. 3.33. Окно выбора характеристик процессора

Еще одна настройка, которой вам иногда придется пользоваться, связана с заданием каталогов, в которых могут быть расположены нестандартные модули, подключаемые файлы и библиотеки. Соответствующее диалоговое окно (рис. 3.34) вызывается по команде **Options → Directories**. В тех трех строках, которые вы видите на рисунке, один раз нужно изменить имя диска, если вы расположили каталог FPC в другом месте. В случае необходимости вы можете добавить новые строки, используя которые система найдет дополнительные модули, подключаемые файлы и библиотеки.

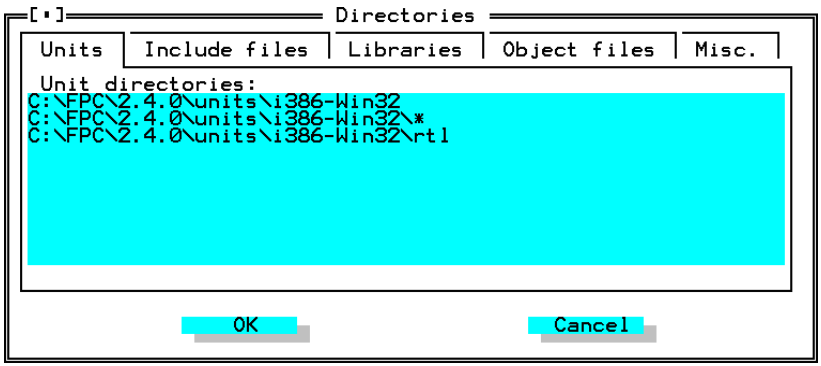


Рис. 3.34. Окно набора каталогов

Последняя полезная настройка, к которой вы можете прибегнуть на стадии освоения системы Free Pascal, связана с внешним видом среды, точнее, с количеством строк и размером символов в поле редактора. При выполнении команд **Options → Environment → Preferences** появляется диалоговое окно, представленное на

рис. 3.35. В панели **Video mode** вы можете выбрать один из следующих форматов: **40×25**, **80×25**, **80×30**, **80×43**, **80×50** и **80×60**.

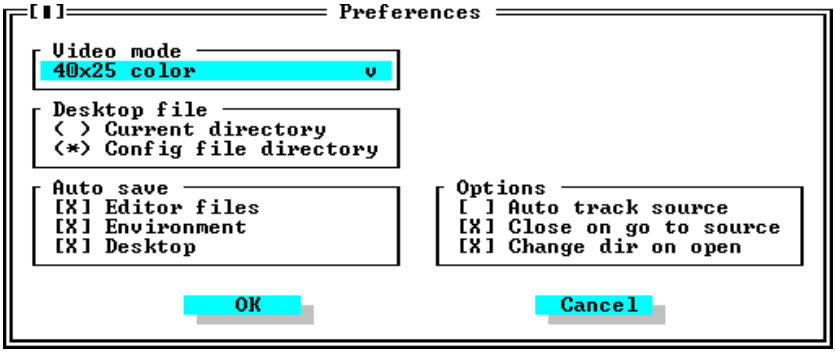
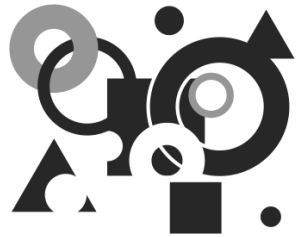


Рис. 3.35. Управление параметрами среды

Глава 4



Простые типы данных в языке Free Pascal

Просматривая примеры задач, приведенные в *главе 2*, вы, наверное, заметили, что их решения с помощью ЭВМ содержат две части — объявления (описания) *данных* и обрабатывающие их *программы* (головной программы, процедур и функций). С этим наблюдением удачно согласуется название одной из книг, принадлежащих перу автора Паскаля — Н. Вирту, — "Алгоритмы + структуры данных = = программы". Поэтому изучение любого алгоритмического языка целесообразно начинать со знакомства с типами данных, которые можно обрабатывать с помощью средств этого языка. Знание типов данных и их основных характеристик (диапазон возможных значений, занимаемые ресурсы оперативной памяти, точность и эффективность соответствующих компьютерных операций) позволят вам грамотно выбрать способы кодирования и обработки тех или иных параметров решаемой задачи.

В первом приближении данные, используемые в программах на языке Free Pascal, можно разделить на две категории — *простые* (одиночные, скалярные) данные и *сложные* (составные, структурированные) данные.

С типами простых данных компилятор хорошо "знаком", и для объектов этого типа достаточно указания на принадлежность тому или иному классу. Иногда такие типы данных называют *стандартными* или *базовыми*. Условная классификация данных простого типа приведена на рис. 4.1.

Для данных сложного типа от пользователя требуются дополнительные указания, такие как, например, размеры массива и тип его элементов, описание полей записи и т. п. Иногда данные сложного типа называют *пользовательскими*. Условная классификация данных сложного типа приведена на рис. 4.2. Один из компонентов сложных данных, — *классы*, — представляет собой нечто большее, чем только набор данных. По сути дела, это неразрывное объединение некоторого набора данных и обрабатывающих их подпрограмм. С помощью классов пользователи могут вводить в свои программы новые типы данных, описания которых в алгоритмическом языке отсутствуют. В таких случаях пользователи обязаны дополнительно описывать процедуры, обеспечивающие выполнение нужных операций над новыми данными. Такой прием носит название *перегрузки* (или *переопределения*) операций, знакомых компилятору по данным базовых типов.

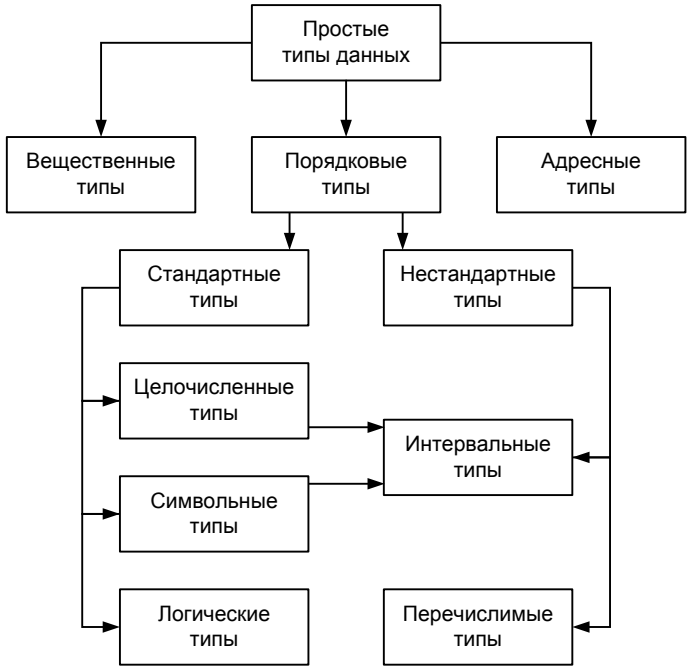


Рис. 4.1. Классификация данных простого типа

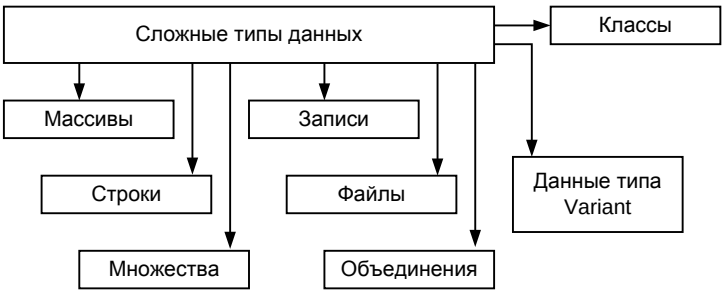


Рис. 4.2. Классификация данных сложного типа

Математики привыкли оперировать с множествами объектов различной природы. Какие-то из объектов могут выделяться индивидуальными свойствами, другие, как близнецы, могут совпадать по всем параметрам. Чтобы однозначно выделить тот или иной объект, приходится придумывать какие-то дополнительные характеристики. Например, в множестве, содержащем пять единиц, можно каждую из них покрасить в разный цвет.

Предположим, что мы имеем дело с конечным множеством из N различных объектов. Тогда объекты можно выстроить в очередь, присвоив каждому из них порядковый номер, например, 1, 2, 3, ..., N. После этого можно говорить об упорядоченности объектов и использовать понятные всем в житейском смысле терми-

ны — *первый, последний, предшествующий, следующий*. Точно так же можно поступить и с бесконечным множеством объектов, например, с натуральными числами — 0, 1, 2, 3, ... В математике для подобных множеств, соответственно, используют термины *"конечное"* и *"счетное"*. Однако не с любым числовым множеством можно так поступить. Если идет речь обо всех неотрицательных вещественных числах на числовой оси, то первое (наименьшее) число выделить можно — это 0. Говорить об упорядоченности вещественных чисел тоже можно — фиксированное значение на числовой оси меньше чисел, расположенных правее. А вот перенумеровать вещественные числа нельзя: между двумя любыми претендентами ($a < b$) всегда найдется бесконечное множество промежуточных чисел, например вида $\frac{a+b}{k}$, где $k > 1$. То есть из множества вещественных чисел, принадлежащих любому интервалу числовой оси, нельзя выстроить дискретную очередь, в которой было бы четко известно, кто за кем стоит. О таких множествах математики говорят, что они имеют *мощность континуума* (мощность — это в некотором смысле мера количества объектов в множестве).

Компьютерное представление числовых данных несколько отличается от идеальных математических формулировок. Здесь тоже в ходу термины *"целочисленные данные"* и *"вещественные данные"*. Для ограниченного множества натуральных чисел, дополненного нулем, принят термин *"целое число без знака"*. Если целочисленное значение может быть не только положительным, но и отрицательным, то о нем говорят как о *целом числе со знаком*. Числа, у которых может быть дробная часть, относят к разряду вещественных и говорят о них как о *числах с плавающей запятой*. В англоязычной литературе принят термин — с плавающей точкой *(floating point)*. Компьютерная специфика начинается с того, что для представления числовых данных в ЭВМ, во-первых, используется не общепринятая десятичная система, а обычно двоичная. Во-вторых, для хранения числовых данных в оперативной памяти выделяется фиксированное число двоичных разрядов, обычно кратных байту — 1, 2, 4, 8 или 10 байт. При этом возникает ряд проблем, о которых "чистые" математики не задумывались.

С целыми числами дело обстоит приблизительно так же, как и с конечным множеством. В зависимости от количества байтов, выделяемых для хранения того или иного целочисленного типа, и наличия/отсутствия знака можно определить минимально и максимально представимые числа. Например, для однобайтовых целых чисел без знака допустимым интервалом является [0, 255]. Если старший двоичный разряд байта используется как признак знака числа, допустимый интервал смещается влево — [–128, +127]. Понятия *"следующий"* и *"предыдущий"* для внутренних значений из допустимого интервала здесь совпадают с общепринятыми. Но аппаратная арифметика в компьютере такова, что прибавление единицы к максимально допустимому целому числу дает в качестве результата минимально допустимое число, а вычитание единицы из минимального числа приводит к максимально допустимому значению. Таким образом, допустимый диапазон данных из устоявшегося представления как об отрезке числовой оси превращается в *кольцо*, где вслед за минимальным числом расположено максимальное.

С данными вещественного типа дело обстоит тоже не совсем гладко. Во-первых, из-за использования двоичной системы некоторые числа, представимые конечным числом десятичных разрядов, после перевода в двоичную систему превращаются в бесконечные периодические дроби, хвосты которых отбрасываются. В результате десятичное число 0.2 в памяти компьютера оказывается чуть-чуть меньше своего истинного значения, и его умножение, например, на 5 не приводит к точной единице. Такого рода нюансы приводят к появлению накапливающихся погрешностей, что должно аккуратно учитываться при анализе точности получаемых результатов. Во-вторых, в силу ограниченности количества двоичных разрядов, выделяемых для хранения вещественных чисел, возникает интервал (–min, +min), в котором все данные эквивалентны нулю (здесь min — минимально допустимое по модулю значение). В этом смысле значение min и есть тот шаг, который мог бы превратить вещественные данные в очень длинную дискретную очередь. Однако создатели алгоритмических языков на это не пошли, и стандартные функции, определяющие предшественника (`pred()`), или следующего за текущим числом (`succ()`) к вещественным данным не применимы.

4.1. Числовые данные

Полный перечень целочисленных типов данных, используемых в программах на языке Free Pascal, приведен в табл. 4.1. По сравнению с языком Object Pascal здесь появился новый тип 8-байтовых чисел без знака (`QWord`).

Таблица 4.1

Тип	Длина, байт	Наименьшее значение	Наибольшее значение
`Byte`	1	0	255
`Word`	2	0	65 535
`LongWord`	4	0	4 294 967 295
`Cardinal`	4	0	4 294 967 295
`QWord`	8	0	18 446 744 073 709 551 615
`ShortInt`	1	–128	127
`SmallInt`	2	–32 768	32 767
`Integer`	4	–2 147 483 648	2 147 483 647
`LongInt`	4	–2 147 483 648	2 147 483 647
`Int64`	8	-2^{63}	$2^{63} - 1$

Список вещественных типов данных приведен в табл. 4.2.

Таблица 4.2

Тип	Длина, байт	Наименьшее значение	Наибольшее значение	Количество десятичных цифр
`Single`	4	-1.5×10^{45}	3.4×10^{38}	7—8
`Real48`	6	-2.9×10^{39}	1.7×10^{38}	11—12
`Double`	8	-5×10^{324}	1.7×10^{308}	15—16
`Extended`	10	-3.6×10^{4951}	1.1×10^{4932}	19—20
`Comp`	8	-2^{63}	$2^{63} - 1$	19—20
`Currency`	8	От –922337203685477.5808 до +922337203685477.5807		19—20

Тот факт, что целочисленный формат `Int64` и вещественный формат `Comp` имеют идентичное машинное представление, объясняется историей развития языка Паскаль. Изначально в языке был предусмотрен только формат `Comp`, отнесенный к вещественным данным из-за слишком больших границ допустимого интервала. Арифметические операции над данными такого типа выполнялись с помощью специальных подпрограмм. По мере развития технических возможностей IBM PC и расширения команд сопроцессора, ориентированного, главным образом, на выполнение операций над данными с плавающей запятой, 8-байтовый тип целочисленных данных стал одним из аппаратных форматов сопроцессора. Для сохранения преемственности со старыми программами, в которых тип `Comp` считался вещественным, список форматов вещественных типов данных в Паскале сохранили. (Ну, не обижать же язык из-за отсутствия длинных целочисленных данных!) Вдобавок в реализациях языков C++ и Pascal в среде Windows форматы `Integer` и `LongInt` друг от друга не отличаются (в стандарте языка не оговорено, сколько байтов должно отводиться для хранения этих данных).

4.2. Внешнее представление числовых констант

Почти в каждой программе, написанной на том или ином алгоритмическом языке, встречаются числовые константы, которые используются как операнды арифметических или логических выражений, как начальные значения переменных числового типа, как значения именованных числовых констант, как границы индексов массивов и т. п. В любом из перечисленных случаев используется термин *"литеральная константа"*, т. е. константа, записанная с помощью некоторого на-

бора символов (литер). По внешней форме записи литеральной числовой константы компилятор принимает решение о машинном формате ее хранения в оперативной памяти компьютера.

Для записи целочисленных десятичных констант используются десятичные цифры (0, 1, 2, ..., 9) и знаки числа (+, –). Решение о выборе подходящего числового формата компилятор принимает, руководствуясь самым простым правилом — для хранения целочисленной литеральной константы выбирается самый маленький диапазон, в котором константа может быть представлена. Приведенная в листинге 4.1 программа демонстрирует это правило, используя стандартную функцию `SizeOf`.

Листинг 4.1. Программа `types`

```
program types;
begin
  writeln(1:8 ,'  ',SizeOf(1));
  writeln(-1:8 ,'  ',SizeOf(-1));
  writeln(255:8,'  ',SizeOf(255));
  writeln(-255:8,'  ',SizeOf(-255));
  writeln(32767:8,'  ',SizeOf(32767));
  writeln(-32767:8,'  ',SizeOf(-32767));
  writeln(32769:8,'  ',SizeOf(32769));
  writeln(-32769:8,'  ',SizeOf(-32769));
  writeln(65536:8,'  ',SizeOf(65536));
  writeln(-65536:8,'  ',SizeOf(-65536));
  readln;
end.
```

Результаты ее работы с комментариями, дописанными вручную, таковы:

```
       1  1     // Тип Byte (нет знака)
      -1  1     // Тип ShortInt (есть знак)
     255  1     // Тип Byte
    -255  2     // Тип SmallInt
   32767  2     // Тип SmallInt
  -32767  2     // Тип SmallInt
   32769  2     // Тип Word
  -32769  4     // Тип Integer
   65536  4     // Тип LongWord
  -65536  4     // Тип Integer
```

Free Pascal допускает использование *шестнадцатеричных*, *восьмеричных* и *двоичных* целочисленных констант без знака. Если первый тип констант был присущ языку Паскаль изначально, то два следующих типа перекочевали во Free Pascal из других алгоритмических языков.

Записи шестнадцатеричной константы предшествует символ `$`, вслед за которым располагается комбинация десятичных цифр (0, 1, 2, ..., 9), прописных (A, B, C, D, E, F) или строчных (a, b, c, d, e, f) букв, используемых для обозначения чисел от 10 до 15. Каждая пара шестнадцатеричных цифр, начиная с младших, занимает в оперативной памяти один байт. Собственно этим и определяется машинный формат шестнадцатеричных констант — 1, 2, 4 или 8 байт. Недостающие старшие разряды дополняются нулями. Например:

```
$1A     // Тип Byte, соответствующее десятичное значение - 26
$a      // Дополняется до $0a, тип - Byte
$123    // Дополняется до $0123, тип - Word
```

Записи восьмеричной константы предшествует символ `&`, вслед за которым располагается комбинация цифр от 0 до 7. Каждая восьмеричная цифра в оперативной памяти заменяется тройкой двоичных разрядов, а старшие биты, дополняющие число до границы байта, заполняются нулями:

```
&17     // Тип Byte, четыре старших бита дополнены нулями
&723    // Тип Word, семь старших битов дополнены нулями
```

Записи двоичной константы предшествует символ `%`, вслед за которым располагается комбинация двоичных цифр (0, 1):

```
%111    // Тип Byte, пять старших битов дополнены нулями
```

Внешнее представление вещественной десятичной константы отличается от целого числа тем, что в записи могут присутствовать:

- точка, отделяющая целую часть числа от дробной;
- порядок числа, начинающийся с большой (`E`) или малой (`e`) латинской буквы, вслед за которой располагается значение порядка со знаком.

В записи вещественного числа может присутствовать либо одна из этих составляющих, либо обе одновременно. Например:

```
3.  (или 3.0)
3E0 (или 3e0)
3.14
0.314E1 (или 0.314E+1)
31.4e-1 (или 31.4E-1)
```

4.3. Внутренний формат числовых данных

Под внутренним форматом имеется в виду машинное представление числовой информации. Мы уже упоминали, что в оперативной памяти компьютера числовые данные представлены в двоичной системе. Для целых положительных чисел это представление может быть записано следующим образом:

$$N = b_k b_{k-1} b_{k-2} \ldots b_2 b_1 b_0.$$

Здесь b_j — двоичная цифра, принимающая значение 1 или 0 в зависимости от того, присутствует или не присутствует слагаемое 2^j в разложении числа *N*. Для нахождения двоичных цифр в разложении числа *N* можно записать:

$$N = b_k \cdot 2^k + b_{k-1} \cdot 2^{k-1} + b_{k-2} \cdot 2^{k-2} + ... + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 .$$

Очевидно, что после деления числа *N* на 2 степень первых *k* слагаемых понизится на 1, а в остатке окажется цифра b_0. Если частное от первого деления вновь разделить на 2, то следующий остаток будет равен цифре b_1. Так продолжают до тех пор, пока очередное частное не окажется равным 0. Последующие деления на 2 ничего кроме нулевого частного и нулевого остатка давать не будут. Так как лидирующие нули нас не интересуют, то процедура перевода целого числа в двоичную систему завершается. Продемонстрируем этот алгоритм на примере $N - 57$:

- первое деление на 2: частное = 28 остаток = 1 (цифра b_0);
- второе деление на 2: частное = 14 остаток = 0 (цифра b_1);
- третье деление на 2: частное = 7 остаток = 0 (цифра b_2);
- четвертое деление на 2: частное = 3 остаток = 1 (цифра b_3);
- пятое деление на 2: частное = 1 остаток = 1 (цифра b_4);
- шестое деление на 2: частное = 0 остаток = 1 (цифра b_5).

Таким образом, $57_{10} = 111001_2$.

Если число *N* — дробное, то его двоичное представление выглядит так:

$$N = 0.b_{-1}b_{-2}b_{-3}...$$

Это равенство можно переписать в следующем виде:

$$N = b_{-1} \cdot 2^{-1} + b_{-2} \cdot 2^{-2} + b_{-3} \cdot 2^{-3} + ...$$

Если эту сумму умножить на 2, то в разряде целых окажется цифра b_{-1}, а степени всех остальных слагаемых в дробной части повысятся на 1. Повторное умножение новой дробной части переведет в разряд целых цифру b_{-2}. Этот процесс может закончиться естественным образом, когда очередная дробная часть окажется равной 0. Но может и не закончиться, т. к. дробные части могут периодически повторяться. В этом случае процесс перевода завершают по достижении требуемого количества дробных разрядов. Продемонстрируем этот алгоритм на примерах.

Для `N1=0.75`:

- первое умножение `N1` на 2: дробная часть = 0.5 целая часть = 1 (цифра b_{-1});
- второе умножение на 2: дробная часть = 0 целая часть = 1 (цифра b_{-2}).

Таким образом, $0.75_{10} = 0.11_2$.

Теперь для `N2=0.2`:

- первое умножение `N2` на 2: дробная часть = 0.4 целая часть = 0;
- второе умножение на 2: дробная часть = 0.8 целая часть = 0;
- третье умножение на 2: дробная часть = 0.6 целая часть = 1;
- четвертое умножение на 2: дробная часть = 0.2 целая часть = 1;
- пятое умножение на 2: дробная часть = 0.4 целая часть = 0.

(начинается периодическое повторение).

Таким образом, $0.2_{10} = 0.0011001100110011..._2$.

Второй пример демонстрирует тот факт, что некоторые конечные десятичные дроби в двоичной системе представлены периодическими дробями. В силу конечного числа разрядов, которое выделяется в оперативной памяти для хранения числовой информации, хвост периодической дроби приходится отбрасывать. Поэтому для некоторых данных изначально происходит небольшая потеря точности.

Если вещественное число `N` содержит и целую часть, и дробную, то каждую из них переводят в двоичную систему по описанным выше алгоритмам, а потом их объединяют. Например, $57.75_{10} = 111001.11_2$.

Опытные программисты используют более эффективный способ перевода. Сначала десятичные числа переводят в шестнадцатеричную систему (целые — путем деления на 16, дробные — путем умножения на 16), а затем каждую шестнадцатеричную цифру заменяют ее четырехразрядным двоичным эквивалентом. Например, число 57 переводят следующим образом:

- первое деление на 16: частное = 3 остаток = 9 (двоичный код — 1001);
- второе деление на 16: частное = 0 остаток = 3 (двоичный код — 0011).

Результат: 00111001.

При переводе дроби 0.2 период наблюдается после первого же умножения:

первое умножение на 16: дробная часть = 0.2 целая часть = 3.

Результат: 0.(0011).

Наконец, самые ленивые пользователи для перевода целых чисел могут прибегнуть к услугам калькулятора Windows (рис. 4.3).

Вещественные числа хранятся в оперативной памяти компьютера в виде двух компонентов — значащих цифр числа (так называемой *мантиссы*) и множителя (двоичного *порядка*). В IBM-совместимых компьютерах мантисса обычно находится в диапазоне (2, 1] или (1, 0.5]. При таком выборе происходит минимальная потеря точности. Такую мантиссу принято называть *нормализованной*. Обозначим знак числа через *s*, мантиссу через *m*, а порядок числа через *p*. Тогда вещественное число *N* можно представить в следующем виде:

$$N = (-1)^s \cdot m \cdot 2^p .$$

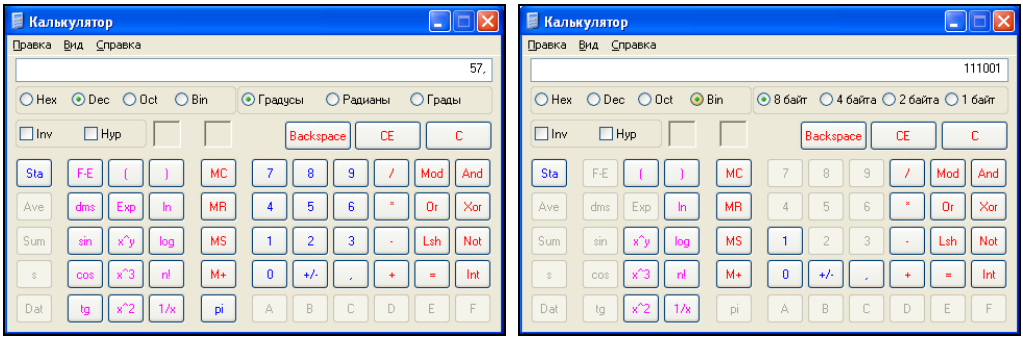


Рис. 4.3. Перевод целых чисел в двоичную систему с помощью калькулятора

Для хранения знака числа отводится один двоичный разряд, который равен 1 для отрицательных чисел и 0 — для положительных. Порядок числа (вместе со своим знаком) в зависимости от выбранного машинного формата занимает от 8 до 15 двоичных разрядов. Остальные разряды заняты мантиссой. Но и тут не обошлось без приятного пустячка. Так как у чисел, отличных от 0, старший разряд нормализованной мантиссы равен 1, то в оперативной памяти его не хранят. Когда вещественные операнды извлекаются из оперативной памяти в регистры микропроцессора, то старшую единицу мантиссы восстанавливают. Хранение старшего бита нормализованной мантиссы "в уме" позволяет хранить в оперативной памяти лишний двоичный разряд мантиссы, увеличивая тем самым точность ее представления.

Название *"формат с плавающей точкой"* (floating point) обязано своим происхождением тому обстоятельству, что умножение мантиссы на порядок (степень двойки) вызывает перемещение точки (ее положение в мантиссе фиксировано) вправо (при положительном порядке) или влево (при отрицательном порядке).

4.3.1. Дополнительный код для целых отрицательных чисел

Отрицательные целые числа в компьютере представляются в *дополнительном* коде. Это означает следующее. Если для хранения чисел отводится n двоичных разрядов, то числа N и $-N$ дополняют друг друга до 2^n. Предположим, что для хранения числа 57 в памяти отведен один байт. Тогда двоичное представление этого числа имеет вид: 00111001. Его дополнением до 2^8 является следующий код: 11000111, который и представляет значение –57.

Выбор дополнительного кода для хранения отрицательных чисел обусловлен спецификой выполнения наиболее распространенных арифметических операций сложения и вычитания. В дополнительном коде эти действия выполняются за минимальное время.

На практике дополнительный код числа *–N* получают следующим образом. Сначала число *N* переводят в двоичную систему, затем все разряды полученного двоичного числа заменяют обратными (инвертирование кода) и к результату добавляют 1. Опытные программисты используют еще более скоростной алгоритм: при инвертировании разрядов положительного числа младшую значащую единицу не переворачивают и сохраняют все последующие нули.

Программистам полезно помнить, что число -1_{10} в дополнительном коде представлено сплошными единицами (для однобайтовых данных — 11111111_2 или $FF).

4.3.2. Операции над целочисленными данными

Арифметические операции

Кроме четырех обычных операций сложения (`a+b`), вычитания (`a-b`), умножения (`a*b`) и деления (`a/b`) в языке Free Pascal предусмотрены целочисленное деление (`a div b`) и нахождение остатка от деления (`a mod b`). Результат всех приведенных выше операций над целочисленными операндами `a` и `b`, за исключением операции деления (`a/b`) является целочисленным. Операция обычного деления (`a/b`) всегда дает вещественный результат. Этим Паскаль отличается от многих алгоритмических языков. При условии, что оба операнда принадлежат одному и тому же целочисленному типу, существует вероятность, что результат операции может оказаться за пределами границ, допустимых для данного типа. Если при работе компилятора включен контроль за выходом из допустимого интервала (`{$R+}`), то такая ситуация будет зафиксирована как ошибочная. Если такой контроль отключен, то скажется эффект "кольца", упомянутый ранее. Следующий пример демонстрирует обе ситуации (листинг 4.2).

Листинг 4.2. Программа `range`

```
program range;
{$R-}
var
  a:byte=255;
  b:byte=3;
  c:byte;
begin
  c:=a+b;
  writeln(c);
  readln;
{$R+}
  c:=a+b;
  readln;
end.
```

Первое сложение дает `c=2`. На втором сложении генерируется сообщение об ошибке (рис. 4.4).

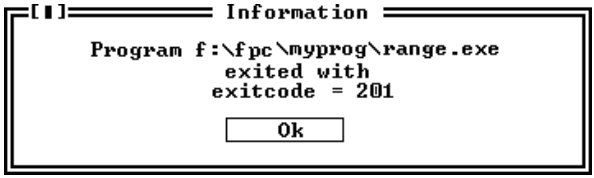


Рис. 4.4. Сообщение о выходе за пределы допустимого диапазона

Поразрядные логические операции

Над целочисленными данными (обычно без знака) можно выполнять поразрядные (битовые) логические операции — логическое сложение (`a or b`), логическое умножение (`a and b`), исключающее "ИЛИ" (`a xor b`) и отрицание (`not a`). Приведенная в листинге 4.3 программа дает представление о каждой из этих операций.

Листинг 4.3. Программа `Bool_op`

```
program Bool_op;
var
  a:byte=$C;
  b:byte=$A;
begin
  writeln('a = ',a:9);
  writeln('b = ',b:9);
  writeln('a  or b = ',(a or b):3);
  writeln('a and b = ',(a and b):3);
  writeln('a xor b = ',(a xor b):3);
  writeln('not a   = ',(not a):3);
  readln;
end.
```

Результаты ее работы с комментариями, добавленными вручную, выглядят так:

```
a =         12    // Двоичный код = 00001100
b =         10    // Двоичный код = 00001010
a  or b =   14    // Двоичный код = 00001110
a and b =    8    // Двоичный код = 00001000
a xor b =    6    // Двоичный код = 00000110
not a   =  243    // Двоичный код = 11110011
```

Операции сдвига

Целочисленный двоичный код `x` может быть сдвинут вправо (операция `shr`) или влево (операция `shl`) на `n` разрядов:

```
y := x shr n;
z := x shl n;
```

Использовать эти операции надо осторожно. Дело в том, что операции сдвига разделяются на две категории — логический сдвиг (операнд `x` — число без знака) и арифметический сдвиг (операнд `x` — число со знаком). Сдвиг на `n` разрядов влево эквивалентен умножению `x` на 2^n, а сдвиг на `n` разрядов вправо — целочисленному делению на 2^n. Когда число `x` отрицательно, оно представлено в дополнительном коде и имеет в старших разрядах группу единиц, в том числе и в старшем (знаковом) разряде. После сдвига вправо число должно остаться отрицательным. Поэтому при арифметическом сдвиге вправо знаковый разряд как бы раздваивается. Во-первых, он должен сдвинуться и, в то же время, остаться на своем месте. При логическом сдвиге старший разряд числа не является знаковым, поэтому он просто сдвигается. Для более подробного знакомства со спецификой операций сдвига предлагается следующая программа (листинг 4.4).

Листинг 4.4. Программа `shift`

```
program shift;
{$R-}
var
  x:byte;
  y:shortint;
  j:integer;
begin
  x:=8;
  for j:=1 to 5 do
    begin  x:=x shl 1; write(x:6);  end;
  writeln;
  x:=8;
  for j:=1 to 5 do
    begin  x:=x shr 1; write(x:6);  end;
  writeln;
  y:=-8;
  for j:=1 to 5 do
    begin  y:=y shl 1; write(y:6);  end;
  writeln;
  y:=-8;
  for j:=1 to 5 do
```

```
    begin  y:=y shr 1; write(y:6);  end;
  readln;
end.
```

Результат ее работы:

```
 16    32    64   128     0
  4     2     1     0     0
-16   -32   -64  -128     0
 -4    -2    -1    -1    -1
```

Обратите внимание на тот факт, что при пятом сдвиге чисел влево происходит выход за пределы допустимого диапазона, поэтому в начале программы пришлось отключить контроль границ интервала. В противном случае программа была бы снята из-за ошибки. И еще одно — при сдвиге числа –1 вправо видимых изменений не происходит: единица, вылетающая за пределы младшего разряда, тотчас же компенсируется единицей, подбрасываемой из-за размножения знака.

4.3.3. Арифметические операции над вещественными числами

Над вещественными данными разрешены только четыре арифметические операции — сложение (`a+b`), вычитание (`a-b`), умножение (`a*b`) и деление (`a/b`).

4.4. Числовые данные интервального типа

Подмножество целочисленных данных, принадлежащее заданному диапазону [min, max], относят к числовым данным *интервального* типа. При объявлении таких данных в Паскале принята следующая форма записи:

```
Type
   qq = 1..10;   // min=1, max=10
...
Var
   x : qq;
   y : -5..12;   // min=-5, max=12
```

Стандартные целочисленные типы в языке Free Pascal можно рассматривать как данные интервального типа:

```
Type
   SmallInt = -32768..32767;
   Byte = 0..255;
```

Для целочисленных данных интервального типа, диапазон представления которых отличается от базовых данных, определены все операции, допустимые для

стандартных числовых типов. Однако ситуации, когда результат арифметической операции над данными интервального типа не принадлежит установленному диапазону [min, max], рассматриваются как ошибочные.

4.5. Нечисловые данные порядкового типа

Как уже отмечалось ранее, к данным *порядкового* типа относятся такие конечные множества данных, которые можно тем или иным способом упорядочить, т. е. "выстроить" в очередь и приписать каждому элементу порядковый номер, выбрав для этого, например, натуральные числа. Кроме описанных ранее стандартных целочисленных типов и интервальных подмножеств целых чисел к порядковым данным относятся:

- данные логического типа;
- перечисления;
- символьные данные.

4.5.1. Данные логического типа

Free Pascal поддерживает данные логического типа, которые могут принимать одно из двух значений — `True` (Истина) или `False` (Ложь). Алгебра логики, заложенная английским математиком Джоном Булем, оперирует логическими (булевыми) переменными, которым обычно приписывается одно двух состояний — 1 (т. е. истина) и 0 (т. е. ложь). В языках программирования, допускающих работу с данными типа `Boolean`, имеет место сходная ситуация. Язык Free Pascal так же, как и Object Pascal, разрешает использовать четыре типа логических данных, специфика которых отражена в табл. 4.3.

Таблица 4.3

Имя типа	Длина в байтах	Числовое значение `True`	Числовое значение `False`
`Boolean`	1	1	0
`ByteBool`	1	Не 0	0
`WordBool`	2	Не 0	0
`LongBool`	4	Не 0	0

Казалось бы, что для хранения значения булевой переменной достаточно одного бита, в котором может храниться либо 0 (`false`), либо 1 (`true`). Но дело в том, что неделимой единицей оперативной памяти является байт, а при обращении к оперативной памяти в зависимости от "ширины" шины данных (это определяется

аппаратурой ПК) можно одновременно выбрать 2, 4 или 8 байт. Дополнительные типы логических данных, предусмотренные языком Free Pascal, обеспечивают связь с программами, написанными на других алгоритмических языках, где могут использоваться приведенные в табл. 4.3 форматы логических данных.

Над логическими данными могут выполняться следующие логические операции:

- логическое сложение (операция "ИЛИ") — `or`;
- логическое умножение (операция "И") — `and`;
- логическое отрицание (операция "НЕ") — `not`;
- исключающее "ИЛИ" — `xor`.

Результаты выполнения этих операций приведены в табл. 4.4.

Таблица 4.4

x	y	x or y	x and y	x xor y	not x
false	false	false	false	false	true
false	true	true	false	true	
true	false	true	false	true	false
true	true	true	true	false	

Используя логические переменные и значения логических функций и объединяя их знаками логических операций, можно конструировать логические выражения, которые принимают значения `true` или `false`. Операндами логических формул могут быть и результаты операций отношения (табл. 4.5).

Таблица 4.5

Операция	Пояснение	Операция	Пояснение
>	Больше	<=	Меньше или равно
>=	Больше или равно	=	Равно
<	Меньше	<>	Не равно

Например, принадлежность значения `x` интервалу [`a`, `b`] определяется истинностью выражения:

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

Иногда результат вычисления логической формулы становится известным до того момента, когда проанализированы действия всех операций. Например, если в приведенном выше условии результат сравнения `a <= x` оказался ложным, то проверять следующее неравенство смысла не имеет. Это означает, что последние действия в формуле могут не повлиять на результат вычисления логического выражения. Поэтому большинство разумных компиляторов по умолчанию не производят

обработку логической формулы до конца, если ее значение уже определилось. Однако в этом, в общем-то разумном, правиле таится потенциальная опасность. Например:

```
x:=false;
y:=x and f(x);
```

С точки зрения "чистого" математика операция логического умножения коммутативна, т. е. (`a and b`) ≡ (`b and a`). С точки зрения программиста при вычислении второго выражения могут возникнуть проблемы. Предположим, что логическая функция `f(x)` во время своей работы меняет значения аргумента `x`, а он передавался по ссылке. Поэтому порядок вычисления "сомножителей" может повлиять на результат вычисления формулы. А если функция `f(x)` выполняет еще какие-то специальные действия, задуманные программистом, то компилятор ее в данной ситуации не вызовет. Поэтому программисту предоставляется возможность предупредить компилятор о том, что в логической формуле нужно проделать все вычисления (см. директиву `{$B}`).

4.5.2. Данные перечислимого типа

Перечисления представляют собой упорядоченный список символьных "значений":

```
Type
   rainbow = (red, orange, yellow, green, aqua, blue, purple);
Var
   col: rainbow;
```

В программе переменной `col` может быть присвоено одно из перечисленных значений:

```
col:=green;
```

В последующем значение переменной перечислимого типа можно сравнить с одним из возможных ее значений:

```
if col=red then ...
```

По описанию перечисления компилятор присваивает каждому мнемоническому "значению" числовой порядковый номер, начиная с 0 (`red=0`, `orange=1`, `yellow=2` и т. д.). В принципе, над мнемоническими "значениями" можно производить такие же операции, как и над целыми числами (если это имеет смысл). Например:

```
col := orange + green;   // аналог 1+3
```

После этого значению переменной `col` будет соответствовать мнемоническое значение `aqua`, имеющее порядковый номер `4`.

В чем смысл замены целочисленных констант такими символьными обозначениями? Дело в том, что в конкретных прикладных задачах удобно иметь дело с мнемоническими обозначениями характеристик некоторых объектов. Например, имея дело с цветами радуги, удобно ввести обозначение для палитры радуги (`rainbow`) и перечислить в ней цвета в том порядке, как они упорядочены в природе

(*"**к**аждый **о**хотник **ж**елает **з**нать, где **с**идит **ф**азан"* — **к**расный, **о**ранжевый, **ж**елтый, **з**еленый, **с**иний, **г**олубой, **ф**иолетовый).

В программах обработки экономической информации очень часто приходится иметь дело с названиями месяцев, дней недели. В таких случаях полезно прибегать к перечислениям типа:

```
Type
   week=(sunday=1, monday, tuesday, wednesday, thursday,
         friday, saturday);
   month=(jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct,
          nov, dec );
```

В приведенных примерах присутствует некоторая особенность. Нумерация констант по умолчанию с 0 противоречит общепринятым нормам работы с календарем. Поэтому для первой константы указано нестандартное значение 1, а все последующие будут пронумерованы в порядке возрастания номеров (`monday=2`, `tuesday=3`, ...).

Перечисления очень широко используются многими системными программами, особенно графическими:

```
Type
  line_style=(SOLID_LINE,      // сплошная линия
              DOTTED_LINE,     // пунктирная линия
              CENTER_LINE,     // штрихпунктирная линия
              DASHED_LINE,     // штриховая линия
              USERBIT_LINE);   // линия, определяемая пользователем
  COLORS=(BLACK,BLUE,GREEN,CYAN,RED,MAGENTA,BROWN,...);
```

Системный набор операций над переменными типа перечислений довольно ограниченный: им можно присваивать значения из объявленного списка, сравнивать значения однотипных переменных, передавать в качестве параметров другим функциям. Попытка вывести их значения приводит к появлению на экране приписанных им числовых номеров.

Функция `ord`, примененная к элементу перечислимого типа, позволяет узнать числовой код, приписанный данному мнемоническому значению. Например:

```
ord(yellow)=2
ord(mar)=3
ord(USERBIT_LINE)=4
```

К данным перечислимого типа можно применять функции `pred` и `succ` для определения предшествующего или следующего элемента. Например:

```
pred(yul)=yun
succ(yul)=aug
```

Попытка узнать, кто находится перед первым элементом перечисления или после последнего, фиксируется как ошибка (при условии, что включен режим контроля интервалов).

Так как для перечислений установлена очередность, то из них могут быть образованы данные *интервального* типа. Например:

```
Type
  WorkDay = sunday..thursday; // рабочие дни недели
```

4.5.3. Символьные данные

К данным типа `char` относятся объекты, представленные в оперативной памяти восьмибитовыми двоичными кодами от 0 до 255.

Первую группу таких объектов с кодами от 0 до 31 относят к группе *управляющих* символов. Среди управляющих символов чаще других используются следующие:

- "пусто" (символ NIL с кодом 0);
- "звуковой сигнал" (символ BEL с кодом 8);
- "табуляторный пропуск" (символ Tab с кодом 9);
- "конец строки" (символ LF с кодом 10);
- "перевод каретки (символ CR с кодом 13);
- "отмена" (символ Esc с кодом 27).

Вторую группу составляют так называемые *отображаемые* символы. При их выводе на экран каждый объект занимает одно знакоместо, в котором отображаются буква, цифра, скобка, знаки операций и препинания. Символ "пробел" с кодом 32 не имеет на экране видимого графического отображения, но он занимает знакоместо, используемое для разделения "слов" — цепочек видимых символов. Среди отображаемых символов присутствует единственный управляющий символ "забой" (Backspace с кодом 127), назначение которого — стереть на экране предшествующий символ.

Для кодировки символьных данных используются специальные таблицы — кодовые страницы (code page) ASCII (American Standard Code for Information Interchange — американский стандартный код для обмена информацией). Основа ASCII была заложена фирмой IBM, когда для кодировки символов латинского алфавита использовались семиразрядные двоичные коды. По мере проникновения компьютерной техники в разные страны выяснилось, что одним латинским алфавитом не обойтись, национальные алфавиты включали довольно много специальных знаков. И тогда пришлось расширить кодировку символов до восьми двоичных разрядов (т. е. до одного байта), но для каждой страны была создана своя кодовая страница ASCII. Первую половину каждой кодовой страницы занимали символы, предложенные фирмой IBM, а вторая половина была предназначена для кодировки символов национального алфавита. В нашей стране используются две кодовые страницы с номерами 866 (для работы под управлением MS-DOS и в 32-разрядных консольных приложениях, изготовленных в среде Windows) и 1251 (для работы в настоящих приложениях Windows). Таблицы соответствующих кодировок можно встретить почти в каждой книге по программированию, но мы предложим доволь-

но простую программу, воспроизводящую символы и их числовые коды для кодовой страницы 866.

Чтобы таблица хорошо разместилась на экране дисплея, сформируем ее в виде 10 колонок, каждая из которых содержит по 23 строки. В колонке предлагается разместить трехзначный числовой код (старший незначащий ноль мы воспроизводить не будем) и соответствующий ему символ таблицы ASCII. В первой строке первой колонки мы расположим первый отображаемый символ — "пробел" с числовым кодом 32. Во второй колонке этой же строки будет находиться символ с числовым кодом 55 и т. д. Таким образом, формула вычисления числового кода при переходе к следующей колонке может быть записана в следующем виде:

$$k = 32 + 23 \times i + j,$$

где i — номер колонки, пробегающий значения от 0 до 9, j — номер строки, пробегающий значения от 0 до 22.

Программа представлена в листинге 4.5.

Листинг 4.5. Программа `ASCII_866`

```
program ASCII_866;
var
  i,j,k: byte;
begin
  writeln('Code Page 866 : ASCII code':50);
  for j:=0 to 22 do
    for i:=0 to 9 do
      begin
        k:=32+23*i+j;
        if k<256 then write(k:3,' ',chr(k),'   ')
                 else write('        ');
       end;
  readln;
end.
```

Результат ее работы приведен на рис. 4.5.

Обратите внимание на тот факт, что коды малых русских букв расположены не сплошным массивом: между буквами "п" и "р" вклинились символы псевдографики, используемые для формирования таблиц, клетки которых ограничены одинарными и двойными линиями. Второй момент, осложняющий обработку русских текстов, связан с тем, что расположение кодов букв "ё" и "Ё" не соответствует порядку букв в русском алфавите.

В кодовой странице 1251 диапазон кодов малых русских букв сплошной, но ситуация с ненормальным расположением кодов букв "ё" и "Ё" сохранилась.

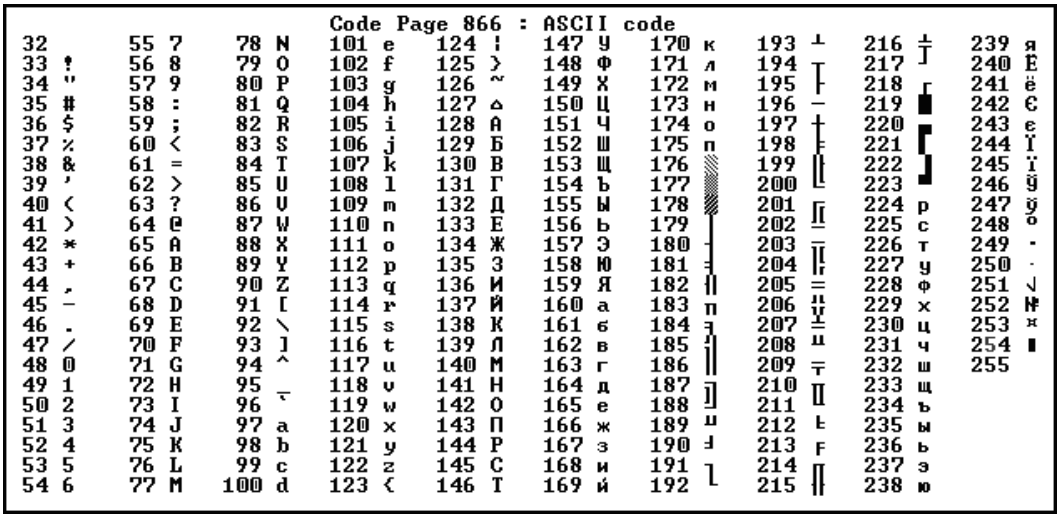


Рис. 4.5. Кодовая страница 866

Для работы с символьными данными используются следующие стандартные функции:

- `ord(x)` — возвращает код ASCII, соответствующий символу-аргументу;
- `chr(n)` — возвращает символ, код которого равен `n` (`0 ≤ n ≤ 255`).

Например:

```
ord('S') = 62    chr(100) = 'd'
```

Так как последовательным кодам символов соответствуют числа натурального ряда, то из них могут быть организованы *интервальные* данные. Например:

```
Type
  LowSym = 'a'..'z';
```

Одной из интересных особенностей Паскаля является возможность использовать символы в качестве индексов элементов массивов:

```
Var
  qq : array ['a'..'z'] of word;
  j : char;
  ...
  for j:='a' to 'z' do
    qq[j]:=0;
  ...
```

Для записи в тексте программы литеральных символьных констант Free Pascal предлагает два формата:

```
Var
  s1: char = 'Q';       // символ в одинарных кавычках (буква Q)
  s2: char = #13;       // символ с кодом 13 в таблице ASCII (CR)
```

4.6. Адресные объекты

Как правило, любые объекты программы снабжаются индивидуальным именем (идентификатором), которое используется в тех случаях, когда мы собираемся извлечь значение объекта или заменить его новым значением. Однако в оперативной памяти компьютера активные в данный момент объекты расположены в ячейках (байтах, словах, двойных или четверных словах), каждая из которых имеет фиксированный адрес. Многие алгоритмические языки предоставляют программисту возможность узнать во время исполнения программы адрес того или иного активного объекта. Такая информация может оказаться более удобной при выполнении некоторых операций, например, при передаче данных в подпрограммы. Так как значение адреса занимает в памяти всего четыре байта, то передача адреса обходится дешевле, чем пересылка значения, занимающего в памяти гораздо больше места.

К объектам адресного типа относятся *указатели на данные* различного типа и *указатели на точки входа* в подпрограммы и функции (по терминологии Паскаля — данные процедурного типа).

Указатели на данные, в свою очередь, могут принимать в качестве своего значения адрес объекта фиксированного типа (по терминологии Паскаля — типизированные указатели) или адрес, с которого в оперативной памяти начинает размещаться объект любого типа (по терминологии Паскаля — нетипизированные указатели). Для объявления данных типа "указатель" используется одна из следующих конструкций:

```
имя_указателя: ^тип;        {типизированный указатель}
имя_указателя: pointer;     {нетипизированный указатель}
```

Типизированному указателю можно присваивать адреса переменных того же типа, а нетипизированному указателю — адреса объектов любого типа. Однако перед извлечением значения, на начальный адрес которого "смотрит" нетипизированный указатель, его надо обязательно привести к типу соответствующего данного. Грубо говоря, типизированный указатель "знает" адрес и длину объекта, на который он "смотрит". Поэтому имеется возможность по текущему адресу определить, где находится следующий или предыдущий объект того же типа. Для этого достаточно к текущему значению указателя прибавить единицу или вычесть из него единицу (значение указателя при этом фактически увеличится или уменьшится на длину соответствующего типа данных). Это напоминает ситуацию с индексами элементов массива: `a[j]` соответствует значению `j`-ого элемента массива, `a[j+1]` — значению следующего элемента, хотя адреса соседних элементов массива могут отличаться больше, чем на 1.

Более детальный разбор операций с типизированными и нетипизированными указателями мы отложим до изучения функций и процедур.

4.7. Ввод/вывод данных простого типа

Из ряда программ, приводившихся в предыдущих разделах, вы заметили, что ввод осуществляется с помощью процедур `read` (от англ. *read* — читать) и `readln` (от англ. *read line* — читать строку), вывод — с помощью процедур `write` (от англ. *write* — писать) или `writeln`. Добавка символов `ln` означает, что после выполнения соответствующей операции курсор на экране дисплея переводится в начало следующей строки.

Каких-то особенностей при вводе числовой информации в процедурах `read`/`readln` нет. Их операндами являются имена переменных всех числовых типов, а к набору на клавиатуре соответствующих значений предъявляются естественные требования:

- тип значения должен соответствовать типу переменной (единственное исключение делается для данных вещественных типов, вводимые значения для которых могут быть и целочисленными);
- пробелы в набираемых значениях не допустимы, любой пробел воспринимается как разделитель между данными (подряд идущие пробелы эквивалентны одному пробелу);
- набор значений завершается по нажатию клавиши <Enter>.

Количество переменных в операторе ввода не обязательно должно соответствовать количеству значений, набираемых пользователем на клавиатуре. Если переменных больше, то после получения сигнала <Enter> программа будет ждать доввода недостающих значений. Если количество набранных значений превышает число аргументов процедуры ввода, то лишние данные могут остаться в буфере ввода и будут востребованы при выполнении следующей операции `read`/`readln`. Но могут и пропасть, если предшествующий ввод был реализован по процедуре `readln`. Приведенный в листинге 4.6 пример демонстрирует разные сочетания двух смежных процедур ввода с превышением количества набираемых значений.

Листинг 4.6. Программа `input1`

```
program input1;
var
  x: integer;
  y: real;
begin
  write('x=');  {приглашение ко вводу}
  read(x);
  writeln('x=',x);
  read(y);
  writeln('y=',y);
  write('x=');  {приглашение ко вводу}
```

```
  read(x);
  readln(y);
  writeln('x=',x);
  writeln('y=',y);
  write('x=');  {приглашение ко вводу}
  readln(x);
  read(y);
  writeln('x=',x);
  writeln('y=',y);
  readln;
end.
```

Результат работы программы `input1` отображен на рис. 4.6. После каждого приглашения ко вводу пользователь набирал два числа вместо одного. В двух первых случаях второе число сохранилось в буфере ввода и было востребовано программой без дополнительного набора значения переменной `y`. В третьем случае лишнее число "пропало" и значение `y` пришлось набирать заново.

```
Running "f:\fpc\myprog\input1.exe "
x=1 1.5
x=1
y= 1.50000000000000E+000
x=2 2.5
x=2
y= 2.50000000000000E+000
x=3 3.5
4.5
x=3
y= 4.50000000000000E+000
```

Рис. 4.6. Результат перебора на вводе

Без дополнительных указаний о форме вывода вещественных данных типа `double` система использует формат по умолчанию — у *мантиссы* выводится 15 *значащих цифр* (первая в целой части) и *десятичный порядок* (после буквы `E`), содержащий знак порядка и три его цифры. Мантисса должна быть умножена на 10 в степени порядка. Эта форма вывода не всегда бывает удобной, и чаще всего данные вещественного типа выводят в формате с *фиксированной точкой*, указывая при этом общую длину числа `n` (число позиций, выделяемых под выводимое значение на экране) и количество `m` цифр в его дробной части:

```
writeln('y=',y:n:m);
```

При этом выводимое число *прижимается к правой границе* отведенного поля и *округляется* с учетом (`m+1`)-й цифры в дробной части: если эта цифра не менее 5, то к `m`-й дробной цифре добавляется единица. В примере, приведенном выше, можно было бы задать `n=4` (в общем числе позиций надо не забывать о знаке числа и точке, отделяющей целую часть от дробной) и `m=1`. Если количество позиций, остающихся под целую часть, оказывается недостаточным, то система расширяет общую длину поля вывода и число выводится правильным. Однако в этом случае при вы-

воде таблиц некоторые значения могут смещаться, нарушая общую стройность колонок.

Следующий пример вывода вещественных данных разного типа демонстрирует не только системные форматы по умолчанию, но и точность представления данных. В программе `pi_out` выводится значение π (листинг 4.7).

Листинг 4.7. Программа `pi_out`

```
program pi_out;
var
  pi_s: single;
  pi_r: real;
  pi_d: double;
  pi_e: extended;
begin
  writeln('pi_Calc=3.14159265358979323846264338327795');
  writeln('pi=    ',pi);
  pi_s:=3.14159265358979323846264338327795;
  pi_r:=3.14159265358979323846264338327795;
  pi_d:=3.14159265358979323846264338327795;
  pi_e:=3.14159265358979323846264338327795;
  writeln('pi_s=  ',pi_s);
  writeln('pi_r=  ',pi_r);
  writeln('pi_d=  ',pi_d);
  writeln('pi_e=  ',pi_e);
  readln;
end.
```

Результат ее работы приведен на рис. 4.7.

```
Running "f:\fpc\myprog\pi_out.exe "
pi_Calc=3.14159265358979323846264338327795
pi=     3.1415926535897932E+0000
pi_s=   3.1415926E+00
pi_r=   3.14159265358979E+000
pi_d=   3.14159265358979E+000
pi_e=   3.1415926535897932E+0000
```

Рис. 4.7. Вывод вещественных данных по умолчанию

Величина π, принятая за эталон точности и содержащая 31 цифру в дробной части, скопирована из калькулятора Windows. Ее значение можно вывести только как строку символов. В переменной `pi`, которую компилятор рассматривает как *системную функцию* (а правильнее было бы называть ее *системной константой*), выведено 16 верных знаков после запятой. На самом деле, эта константа представлена в формате `extended`, сохраняющем не менее 19 десятичных цифр. Те же са-

мые 16 значащих цифр в дробной части мы видим и у переменной `pi_e`. Однако для переменных `pi_r` и `pi_d` формат по умолчанию предусматривает вывод 14 знаков после запятой (тип `double` гарантирует хранение 15—16 десятичных цифр). В формате `single` после запятой выводится 8 значащих цифр, но две последние не соответствуют точному значению. Вообще говоря, их выводить и не следовало, т. к. тип `single` гарантирует хранение не более 7—8 десятичных цифр.

Программа `out_round` демонстрирует не только режимы округления при выводе вещественных данных, но и некоторые программистские трюки: задание нулевых значений `n` и `m` в формате вывода, прижим выводимых данных к левой границе поля при `n<=0` (листинг 4.8).

```
Running "f:\fpc\myprog\out_round.exe "
1
1.2
1.23
1.235
1.2346
1.23457
1.234568
1.2345679
1.23456789
1.234567890
          1
        1.2
       1.23
      1.235
     1.2346
    1.23457
   1.234568
  1.2345679
1.23456789
1.234567890
1
1.2
1.23
1.235
1.2346
1.23457
1.234568
1.2345679
1.23456789
1.234567890
```

Рис. 4.8. Управление форматом вывода

Листинг 4.8. Программа `out_round`

```
program out_round;
var
  x:extended=1.23456789;
  j:integer;
begin
  for j:=0 to 9 do
    writeln(x:0:j);
  for j:=0 to 9 do
    writeln(x:10:j);
```

```
  for j:=0 to 9 do
    writeln(x:-10:j);
  readln;
end.
```

Результат ее работы показан на рис. 4.8.

Приведенные варианты управления шириной поля применимы к выводу целочисленных данных и любой нечисловой информации.

С вводом значений символьных переменных дело обстоит несколько сложнее по двум причинам. Во-первых, символ "пробел" рассматривается как обычный отображаемый символ, а не как разделитель вводимых данных. Во-вторых, от нажатия клавиши <Enter> в буфер ввода поступает управляющий символ с кодом 13. И все они наряду с обычными символами могут быть считаны из буфера. Об этом свидетельствует пример из листинга 4.9.

Листинг 4.9. Программа `input2`

```
program input2;
var
  ch1:char='1';
  ch2:char='2';
  ch3:char='3';
begin
  writeln('ord(old ch1)=',ord(ch1));
  writeln('ord(old ch2)=',ord(ch2));
  writeln('ord(old ch3)=',ord(ch3));
  write('ch1=');
  readln(ch1,ch2,ch3);
  writeln('ord(new ch1)=',ord(ch1));
  writeln('ord(new ch2)=',ord(ch2));
  writeln('ord(new ch3)=',ord(ch3));
  readln;
  readln;
end.
```

Результат работы представлен на рис. 4.9.

```
Running "f:\fpc\myprog\input2.exe "
ord(old ch1)=49
ord(old ch2)=50
ord(old ch3)=51
ch1= 4
ord(new ch1)=32
ord(new ch2)=52
ord(new ch3)=13
```

Рис. 4.9. Ввод данных типа `char`

Обратите внимание на то, что после приглашения к вводу были нажаты три клавиши — <Пробел>, <4> и <Enter>. Три соответствующих символа с кодами 32, 52 и 13 были извлечены из буфера клавиатуры при последующем выполнении оператора `readln(ch1,ch2,ch3)`.

Вообще говоря, ввод значений символьных данных правильнее организовывать с помощью функции `ReadKey` и вводить их по одному, не нажимая клавишу <Enter>. Нужно только не забыть подключить к своей программе модуль `Crt`, в котором находится подпрограмма `ReadKey` (листинг 4.10).

Листинг 4.10. Программа `input3`

```
program input3;
uses Crt;
var
  ch1:char='1';
  ch2:char='2';
  ch3:char='3';
begin
  writeln('new ch1=');
  ch1:=ReadKey;
  writeln('ord(new ch1)=',ord(ch1));
  writeln('new ch2=');
  ch2:=ReadKey;
  writeln('ord(new ch2)=',ord(ch2));
  writeln('new ch3=');
  ch3:=ReadKey;
  writeln('ord(new ch3)=',ord(ch3));
  readln;
  readln;
end.
```

Результат работы программы `input3` приведен на рис. 4.10. В ответ на каждое приглашение к вводу были нажаты те же клавиши, что и в предыдущем примере — <Пробел>, <4> и <Enter>. Но никакого следа на экране не осталось, т. к. в этом режиме ввод происходит без эхо-сигнала. Именно так можно вводить секретные сообщения типа паролей.

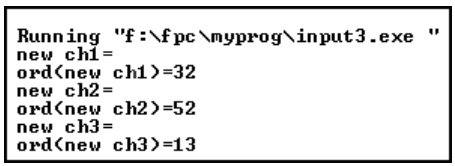


Рис. 4.10. Ввод данных с помощью функции `ReadKey`

С помощью функции `ReadKey` вводят коды специальных клавиш, для которых не нашлось места в таблице ASCII. К таковым, в частности, относятся функциональные клавиши <F1>, <F2>, ... , "стрелки" управления курсором, клавиши редактирования (<Insert>, <Delete>, <Page Up>, <Page Down>, <Home>, <End>) и др. От их нажатия в буфер клавиатуры поступает так называемый *скан-код*, состоящий из двух байт. Первый байт, извлекаемый из буфера клавиатуры по функции `ReadKey`, при этом равен нулю (для обычных символов первый байт совпадает с кодом ASCII). И только по значению следующего байта, который извлекается повторным обращением к функции `ReadKey`, можно определить, какая из специальных клавиш была нажата. Вы можете составить нехитрую программу, которая будет выводить скан-коды интересующих вас клавиш.

Данные логического типа выводятся не так уж и часто: их главное назначение — участие в формировании логических условий, управляющих ходом выполнения программы. Но, тем не менее, вывести их значения можно (листинг 4.11).

Листинг 4.11. Программа `input4`

```
program input4;
var
  b1:boolean=true;
  b2:boolean=false;
begin
  writeln('b1=',b1,' b2=',b2);
  readln;
end.
```

Результат такого вывода представлен на рис. 4.11.

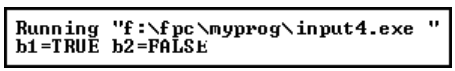


Рис. 4.11. Вывод логических данных

Однако вводить значения логических данных нельзя (такая попытка пресекается компилятором). Их проще формировать в программе, действуя в соответствии с алгоритмом решения задачи.

Данные перечислимого типа в операторах ввода/вывода использовать нельзя (такие попытки пресекаются компилятором). Поэтому приходится искать обходные пути. Например, для вывода можно воспользоваться оператором выбора (переключателем) — листинг 4.12.

Листинг 4.12. Программа `input5`

```
program input5;
type
  RGB=(red,green,blue);
```

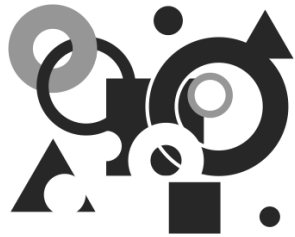
```
var
  a:RGB=green;
begin
  case a of
    red  : writeln('a=red');
    green: writeln('a=green');
    blue : writeln('a=blue');
  end;
  readln;
end.
```

Образец вывода показан на рис. 4.12.

```
Running "f:\fpc\myprog\input5.exe "
a=green
```

Рис. 4.12. Вывод значений перечислимого типа

Для ввода значений такого рода обычно используют порядковые номера перечислимых данных с последующей программной заменой на соответствующую мнемоническую константу.



Глава 5

Обработка строковой информации

Free Pascal поддерживает работу со строковыми константами и переменными четырех типов: `String`, `PChar`, `AnsiString` и `WideString`.

Тип `String` (строка) появился в самой первой версии Паскаля. Строки такого типа сейчас принято называть *короткими строками*, т. к. максимальное количество символов, присутствующих в значениях этого типа, не превосходит 255. Таким ограничением короткие строки обязаны способу их представления в оперативной памяти компьютера. Короткое строковое значение, содержащее *k* символов, занимает в памяти *k* + 1 байт, в первом из которых хранится текущая длина строки (т. е. число *k*), а в следующих — однобайтовые значения кодов символов в кодировке ASCII. Короткие строки с *фиксированной максимальной длиной*, задаваемой пользователем, объявляются следующим образом:

```
type
  S20 = String [20];
var
  s1 : S20;
  s2 : String [35];
```

Строка `s1`, объявленная *неявно*, может принимать любые строковые значения, содержащие не более 20 символов. Превышение указанной длины влечет за собой отсечение лишних хвостовых символов без какого-либо предупреждения. Для хранения значения строковой переменной `s2`, объявленной *явно*, во время работы программы будет выделено 36 байт, первый из которых резервируется под указание фактической длины.

К любому символу короткой строки можно обратиться *по индексу* — порядковому номеру символа в строке:

- `s1[1]` — первый символ в строке `s1`;
- `s1[2]` — второй символ в строке `s1`

  и т. д.

В байте с именем `s1[0]` находится текущая длина строки `s1`. Если старый добрый Паскаль позволял некоторые вольности с извлечением длины по явному обра-

щению к нулевому байту (например, `m:=ord(s1[0]);`) и даже изменение этой длины по прихоти программиста (например, `s1[0]:=char(5);`), то в более поздних системах программирования такие попытки блокируются. Для определения длины текущего значения следует использовать системную функцию `Length` (например, `m:=Length(s1);`).

До тех пор, пока строковой переменной не присвоено значение, в байте длины находится 0, и такая строка называется *пустой*. Очистить строку, т. е. сделать ее пустой, можно и с помощью следующего присваивания:

```
s1:='';
```

Присваивание строке нового значения "уничтожает" предыдущее значение, хотя чистка лишних хвостовых байтов при этом не делается (для уменьшения времени выполнения операции):

```
s1:='Hello, world!';
...
s1:='Привет!';
```

И хотя после второго присвоения значения меньшей длины в байтах `s1[8]`, `s1[9]`, ... сохранились буквы `'world!'`, к новому значению они уже отношения не имеют — просто занимают место на выделенном участке памяти. Если переменной `s1` в последующем будет присвоено более длинное значение, то байты `s1[8]`, `s1[9]`, ... будут затерты новыми данными.

Если указание о длине в явном или неявном объявлении строковой переменной отсутствует, то для хранения ее значения в памяти выделяется 256 байт, что равносильно объявлению `s3:String[255]`. В языке Free Pascal появился эквивалент такого объявления:

```
var
  s3:ShortString;
```

Строки типа `PChar` были заимствованы Паскалем из языка C, где используется другой способ хранения строковых данных: вместо указания длины, предшествующей символьному значению, цепочка символов `s1`, `s2`, ..., `sk` завершается признаком конца строки — нулевым байтом. Для строк такого типа в англоязычной литературе существуют термины *Zstring* (аббревиатура от Zero string) и *null-terminated string*. В языке Free Pascal длина строк типа `PChar` не ограничена и основное их назначение — передача строковых данных в программы, написанные на языке C или C++. С точки зрения внутреннего представления имя строковой переменной типа `PChar` является указателем на строковое значение, располагаемое компилятором в "куче", где каждый символ представлен однобайтовым кодом ASCII. Обращение к такому указателю по имени строки типа `PChar` соответствует выборке или изменению значения всей строки. К символам строки типа `PChar` также можно обращаться по индексу, который отсчитывается от 0. Большинство операций, выполняемых над строками типа `PChar`, сосредоточено в модуле `Strings`. Читатели, знакомые с функциями обработки строк по языкам C, C++, найдут много похожих названий функций и процедур, встречавшихся в заголовочном файле string.h.

Строки типа `AnsiString` представляют собой данные символьного типа неограниченной длины. Они уже фигурировали в языке Object Pascal и активно использовались в разных версиях визуальных сред Delphi. Их внутреннее представление в системе Free Pascal довольно хитроумное. Имя переменной типа `AnsiString` также является указателем на значение, находящееся в куче, но в отличие от `PChar` этот указатель *типизирован*, т. е. "знает" не только адрес, но и длину значения. В куче наряду со значением строки хранится еще и *счетчик ссылок* на данное значение. Если со значением связана единственная активная переменная `s1` типа `AnsiString`, то в счетчике ссылок находится 1. Если значение переменной `s1` присваивается другой переменной `s2` того же типа, то содержимое `s1` *не копируется* на новое место. Вместо этого выполняются следующие операции:

- из содержимого счетчика ссылок `s2` вычитается 1. Если счетчик ссылок стал равен 0, то область памяти, занимавшаяся предыдущим значением `s2`, освобождается;
- к содержимому счетчика ссылок на значение `s1` прибавляется 1, и указатель `s1` переписывается в `s2`. Таким образом, при повторном присвоении длинного значения в куче сохраняется единственный экземпляр данного, и на этом экономится время выполнения операции.

Строковые данные типа `WideString` (дословно — "широкая" строка) напоминают по способу хранения значения типа `PChar`, т. е. они закачиваются признаком конца строки. Однако для кодировки символов широких строк используются двухбайтовые коды таблицы Unicode. Признак конца строки, содержащий нулевой код, здесь также двухбайтовый.

5.1. Короткие строки

Для объявления переменных типа "короткая строка" используются служебные слова `String` с необязательным указанием максимальной длины или `ShortString`. Объявление глобальных переменных может сопровождаться их инициализацией, т. е. присвоением начального значения:

```
type
  s20 = String [20];
var
  s1 : String [10] = 'Hello, world!';
  s2 : s20 = 'Free Pascal';
  s3 : ShortString = 'Мама, я хочу домой';
```

В строковых значениях разрешено употреблять русские буквы, и с их выводом в системе Free Pascal никаких проблем не возникает (в отличие от консольных приложений Borland C++Builder и Delphi). Дело в том, что режим набора текста программы в среде FP IDE выполняется в кодовой странице 866. И с этой же кодировкой символов работает консольное приложение (листинг 5.1).

Листинг 5.1. Программа `hello_r`

```
program hello_r;
begin
  writeln('Hello');
  writeln('Привет');
  readln;
end.
```

Результат работы программы `hello_r` приведен на рис. 5.1.

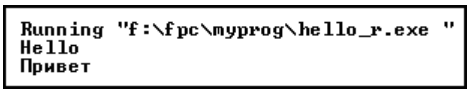


Рис. 5.1. Вывод текстов в программах Free Pascal

Строковые данные вводятся и выводятся обычным образом. Значение, вводимое в короткую строку, может содержать пробелы, которые воспринимаются как обычные отображаемые символы (листинг 5.2).

Листинг 5.2. Программа `input_str`

```
program input_str;
var
  s:ShortString;
begin
  write('s=');
  read(s);
  writeln('s = ',s);
  readln;
  readln;
end.
```

Результат работы программы `input_str` приведен на рис. 5.2.

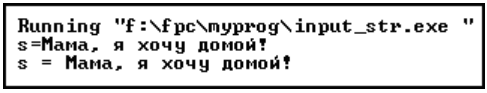


Рис. 5.2. Ввод значения короткой строки

За один прием нельзя ввести два или более значений строковых переменных, даже заключая их в одинарные кавычки и разделяя пробелами или запятыми (листинг 5.3).

Листинг 5.3. Программа `input_str2`

```
program input_str2;
var
  s1,s2:ShortString;
begin
  write('s1,s2 = ');
  read(s1,s2);         {Так делать нельзя}
  writeln('s1=',s1);
  writeln('s2=',s2);
  readln;
  readln;
end.
```

Все, что набрано до нажатия клавиши <Enter>, попало в переменную `s1` (рис. 5.3).

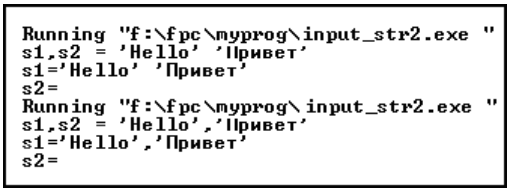


Рис. 5.3. Неудачная попытка ввода двух строковых значений

С помощью функций `Length` и `SizeOf` можно определить длину текущего значения короткой строки и объем памяти, выделенный компилятором для хранения соответствующей переменной (листинг 5.4).

Листинг 5.4. Программа `str4`

```
program str4;
var
  s:String[10] = 'Hello';
begin
  writeln('длина(s) : ',Length(s));
  writeln('выделено памяти под s : ',SizeOf(s));
  readln;
end.
```

Значения этих функций для переменной `s` приведены на рис. 5.4.

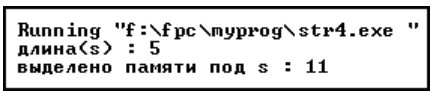


Рис. 5.4. Применение функций `Length` и `SizeOf`

5.2. Операции над символами и фрагментами коротких строк

С операцией сравнения строк мы познакомились в программе `sort_nam` (см. листинг 2.7). Но в словарях, где слова располагаются в лексикографическом порядке (т. е. по алфавиту), не делается разницы между большими и малыми буквами. Хотелось бы реализовать такой же подход и в программах. Однако в таблице ASCII коды всех малых букв русского и латинского алфавитов располагаются вслед за кодами больших букв, следовательно, в числовом эквиваленте код любой малой буквы больше кода соответствующей большой буквы. Для того чтобы нейтрализовать эту разницу, можно воспользоваться двумя способами. Во-первых, можно прибегнуть к одной из функций — `UpCase` или `LowerCase`, с помощью которых в тексте производится замена всех букв на большие или малые. К сожалению, эта возможность распространяется только на буквы латинского алфавита. Во-вторых, вместо обычных операций отношения (больше, меньше, равно) можно воспользоваться функцией сравнения строковых данных `ShortCompareText(s1,s2)`, которая выполняет свою роль, игнорируя разницу между большими и малыми буквами. Эта функция возвращает целочисленный результат, который положителен, если `s1>s2`, равен нулю при `s1=s2` и отрицателен, если `s1<s2`. Однако и эта функция приспособлена только для текстов, содержащих латинские буквы. В этом нетрудно убедиться, анализируя программу `com_str` (листинг 5.5) и результаты ее работы (рис. 5.5).

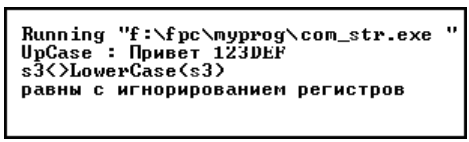


Рис. 5.5. Результаты сравнения текстов

Листинг 5.5. Программа `com_str`

```
program com_str;
var
  s1:String[10]='Привет';
  s2:String[15]='привет';
  s3:String[10]='123ABC';
  s4:String[10]='123def';
begin
  writeln('UpCase : ',UpCase(s1),' ',UpCase(s4));
  if s3=LowerCase(s3) then writeln('s3=LowerCase(s3)')
     else writeln('s3<>LowerCase(s3)');
```

```
  if ShortCompareText(s3,LowerCase(s3))=0
     then writeln('равны с игнорированием регистров');
  readln;
end.
```

Для обработки строковых данных с русскими буквами придется написать свои подпрограммы, аналогичные функциям `UpCase` и `LowerCase`. Мы продемонстрируем такое преобразование на примере замены кодов всех малых букв латинского и русского алфавитов на соответствующие коды больших букв (листинг 5.6).

Листинг 5.6. Программа `upconvert`

```
program upconvert;
const
  s1='abcdefghijklmnopqrstuvwxyz';
  s2='абвгдеёжзийклмнопрстуфхцчшщъыьэюя';
function UpCase_r(s:String):String;
var
  j,ch:byte;
begin
  for j:=1 to length(s) do
    begin
      ch:=ord(s[j])-32;
      if ('a'<=s[j]) and (s[j]<='z') or
         ('a'<=s[j]) and (s[j]<='п')then
           s[j]:=char(ch);
      if ('р'<=s[j]) and (s[j]<='я') then
           s[j]:=char(ch-48);
      if s[j]='ё' then s[j]:='Ё';
    end;
  UpCase_r:=s;
end;
begin
  writeln(s1);
  writeln(UpCase_r(s1));
  writeln(s2);
  writeln(UpCase_r(s2));
  readln;
end.
```

Результат ее работы приведен на рис. 5.6. При наборе программы `upconvert` не забывайте об одной технической детали: в первом сравнении на принадлежность символа `s[j]` диапазону от `'a'` до `'z'` малая буква `'a'` набирается на *латинском*

регистре, а во втором сравнении при проверке на принадлежность диапазону от `'а'` до `'п'` — на *русском* регистре. По начертанию отличить латинскую букву `'а'` от русской нельзя, но для правильной работы программы эта деталь существенна.

Рис. 5.6. Преобразование символов к верхнему регистру

Перечень наиболее часто используемых процедур обработки коротких строк и их фрагментов приведен в табл. 5.1.

Таблица 5.1

Формат вызова	Выполняемая операция
`Delete(s,ind,count);`	Удаление `count` символов из строки `s`, начиная с символа `s[ind]`
`s:=Concat(s1,s2,...,sn);`	Конкатенация (объединение) строк `s1`, `s2`, ..., `sn`
`s1:=Copy(s,ind,count);`	Копирование `count` символов из строки `s`, начиная с символа `s[ind]`
`Insert(s1,s2,ind);`	Вставка в `s2` строки `s1`, начиная с символа `s2[ind]`
`k:=Length(s1);`	Определение длины строки `s1`
`k:=Pos(s1,s2);`	Определение позиции вхождения строки `s1` в строку `s2`

Программа `str_proc` демонстрирует приемы их использования (листинг 5.7).

Листинг 5.7. Программа `str_proc`

```
program str_proc;
var
  s1:string='Hello, world!';
  s2:string;
  k:byte;
begin
  delete(s1,1,5);
  writeln('s1=',s1);
  s2:=concat('Привет',s1,' Привет');
  writeln('s2=',s2);
  writeln('length(s2)=',length(s2));
```

```
  writeln('pos(''world'',s2)=',pos('world',s2));
  k:=pos('Привет',s2);
  writeln('Первое вхождение ''Привет'' в s2 с позиции ',k);
  insert('! Hello!',s2,22);
  writeln('s2=',s2);
  readln;
end.
```

Результаты ее работы приведены на рис. 5.7. Обратите внимание на то, как в строковых константах задается символ *одиночной кавычки*, как правило, используемый в качестве признака начала и конца строкового значения.

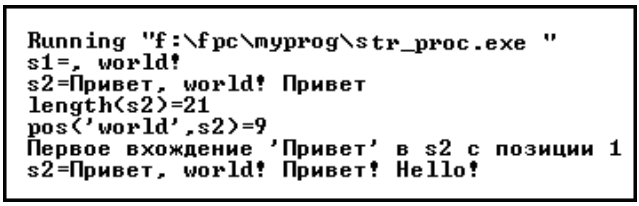


Рис. 5.7. Результаты работы программы `str_proc`

Функция конкатенации (`concat`) в языке Паскаль является некоторым рудиментом, т. к. эта же операция выполняется как "сложение" строковых значений:

```
s2:='Привет'+s1+'Привет';
```

Более того, такая форма записи операции конкатенации *короче*, да и выполняется она *быстрее*, т. к. здесь не требуется передавать параметры функции в стек и извлекать их оттуда.

Для формирования коротких строк, заполненных пробелом или заданным символом, можно воспользоваться функциями `Space` и `StringOfChar`:

```
s3:=Space(10);           {формирование строки из 10 пробелов}
s4:=StringOfChar('=',5); {эквивалент: s4:='=====';}
```

Если во время работы программы потребовалось изменить максимальную длину короткой строки (либо в сторону увеличения, либо в сторону уменьшения), то необходимо обратиться к процедуре `SetLength`:

```
var
  s1:string[10];      // первоначальное объявление
  ...
  SetLength(s1,20);   // изменение максимальной длины
```

В принципе, программист может сам запрашивать оперативную память под короткие строки во время выполнения программы и освобождать такие ресурсы по мере их использования. Делается это с помощью операторов `New` и `Dispose`. Приведенный в листинге 5.8 пример демонстрирует применение этих процедур.

Листинг 5.8. Программа dyn_string

```
program dyn_string;
type
  dyn_str=string[80];
var
  p_str:^dyn_str;        {объявление типизированного указателя}
begin
  New(p_str);            {запрос памяти}
  p_str^:='Hello, world!';
  writeln(p_str^);
  Dispose(p_str);        {освобождение памяти}
  readln;
end.
```

5.3. Прямые и обратные преобразования числовых данных

При выводе числовой информации на экран дисплея системные программы осуществляют перевод данных из машинного представления в строковый вид. Ввод числовых данных с клавиатуры тоже сопровождается преобразованием внешнего символьного представления чисел в их внутренний машинный формат. Средства языка позволяют делать аналогичные преобразования в оперативной памяти по соответствующим обращениям из программы.

5.3.1. Традиционные функции и процедуры

К этой категории отнесены функции и процедуры, использовавшиеся в ранних версиях систем Turbo Pascal. Полный их перечень приведен в табл. 5.2. Система Free Pascal расширила их сферу действия на все типы числовых данных

Таблица 5.2

Формат вызова	Выполняемое преобразование
`s:=BinStr(num,k);`	Преобразование целочисленного значения `num` из машинного формата в символьное представление, содержащее `k` двоичных разрядов
`s:=OctStr(num,k);`	Преобразование целочисленного значения `num` из машинного формата в символьное представление, содержащее `k` восьмеричных разрядов

***Таблица 5.2** (окончание)*

Формат вызова	Выполняемое преобразование
`s:=HexStr(num,k)`	Преобразование целочисленного значения `num` из машинного формата в символьное представление, содержащее `k` шестнадцатеричных разрядов
`Str(v_num,s);`	Преобразование числового значения `v_num` любого типа в строковую переменную `s` типа `String`
`Val(s,v_num,ind)`	Преобразование числа любого типа из символьного представления `s` в числовую переменную `v_num`. Целочисленная переменная `ind` — признак завершения операции

Три первые функции преобразуют целочисленное значение любого типа — со знаком или без знака. Это очень нехарактерные для Паскаля подпрограммы, т. к. допускают наличие аргумента `num`, имеющего тип, совместимый с данными многих типов (`Byte`, `SmallInt`, `ShortInt`, `Word`, `Integer`, `LongInt`, `Int64`, `Qword`). Результат преобразования напоминает целочисленное значение, получающееся при выводе данных на дисплей. Единственное неудобство состоит в задании количества `k` символов в возвращаемом значении. Этот параметр допускает значения из диапазона `Byte`. Если вы зададите величину `k` слишком маленькой, то старшие разряды в переводимом значении будут потеряны и никакое сообщение о произошедшем не выдается. А ведь было бы удобно, например, при `k=0` получать точно переведенное значение без лидирующих нулей, т. е. такой же результат переменной длины, как и при выводе на дисплей. С точностью до указанного ограничения результат, возвращаемый функциями `BinStr`, `OctStr` и `HexStr`, имеет тип `ShortString`. Программа `conv_1` демонстрирует некоторые возможности указанных функций (листинг 5.9).

Листинг 5.9. Программа `conv_1`

```
program conv_1;
var
  n1: integer=45;
  n2: integer=-1;
begin
  writeln(n1,' в двоичной системе = ,,BinStr(n1,32));
  writeln(n1,' в восьмеричной системе = ',OctStr(n1,10));
  writeln(n1,' в шестнадцатеричной системе = ',HexStr(n1,8));
  writeln(n2,' в двоичной системе = ',BinStr(n2,32));
  writeln(n2,' в восьмеричной системе = ',OctStr(n2,10));
  writeln(n2,' в шестнадцатеричной системе = ',HexStr(n2,8));
  readln;
end.
```

Результаты ее работы приведены на рис. 5.8.

```
Running "f:\fpc\myprog\conv_1.exe "
45 в двоичной системе = 00000000000000000000000000101101
45 в восьмеричной системе = 0000000055
45 в шестнадцатеричной системе = 0000002D
-1 в двоичной системе = 11111111111111111111111111111111
-1 в восьмеричной системе = 7777777777
-1 в шестнадцатеричной системе = FFFFFFFF
```

Рис. 5.8. Преобразование числовых данных в строку

Процедуры `Str` и `Val` предназначены для прямого и обратного преобразования числовых значений любого типа (целочисленных и вещественных). Результат работы процедуры `Str` абсолютно такой же, как и при выводе соответствующего числового значения `v_num` на экран с помощью оператора `write`. Более того, при обращении к процедуре `Str` можно также указывать длину результирующей строки `n` и количество цифр `m` после запятой:

```
Str(x:n:m, s);
```

Процедура `Val` в качестве своего исходного аргумента может получить строку `s`, в которой может содержаться числовое значение с недопустимым символом, прерывающим нормальную работу алгоритма перевода. Тогда в целочисленную переменную `ind` заносится индекс ошибочного символа. Если процесс преобразования завершился удачно, то содержимое `ind` равно `0`. Строковый аргумент процедуры `Val` может быть представлен символьной константой в двоичном, восьмеричном или шестнадцатеричном формате. Более того, конвертируемому значению могут предшествовать один или несколько пробелов в начале:

```
Val('  $1E4',x,ind);    {перевод шестнадцатеричного числа}
```

Процедуры `Str` и `Val` также относятся к разряду нехарактерных для Паскаля подпрограмм, т. к. один из их аргументов может иметь произвольный числовой тип, определяемый в момент соответствующего обращения.

5.3.2. Новые функции преобразования числовых данных

В системе Free Pascal довольно много функций по прямому и обратному преобразованию числовых данных, представленных в машинном и символьном форматах. В некоторых случаях они функционально повторяют действия традиционных функций и процедур, в других случаях включают дополнительные указания по форматированию числовых данных. Наконец, появились и новые функции. Перечень дополнительных функций приведен в табл. 5.3.

Таблица 5.3

Формат обращения	**Выполняемые действия**
`s:=IntToBin(num,k);`	Преобразование целочисленного значения `num` из машинного формата в символьное представление, содержащее `k` двоичных разрядов
`s:=IntToHex(num,k);`	Преобразование целочисленного значения `num` из машинного формата в символьное представление, содержащее `k` шестнадцатеричных разрядов
`s:=IntToRoman(num);`	Преобразование целочисленного значения `num` из машинного формата в символьное представление в римской системе представления
`v_num:=RomanToInt(s);`	Преобразование символьного представления целого числа из римской системы счисления в машинный формат
`s:=IntToStr(num);`	Преобразование целочисленного значения `num` из машинного формата в символьное представление десятичного числа
`v:=StrToInt(s);`	Преобразование целочисленного значения из символьного представления в машинный формат типа `Integer`
`v:=StrToInt64(s);`	Преобразование целочисленного значения из символьного представления в машинный формат типа `Int64`
`v:=StrToQWord(s);`	Преобразование целочисленного значения из символьного представления в машинный формат типа `QWord`
`s:=FloatToStr(num);`	Преобразование вещественного значения `num` в символьное представление
`v:=StrToFloat(s);`	Преобразование символьного представления вещественного числа в машинный формат типа `Extended`
`v:=FloatToCurr(num);`	Преобразование вещественного значения `num` в машинный формат представления денежных единиц
`v:=StrToCurr(s);`	Преобразование символьного представления денежных единиц в машинный формат типа `Currency`

В функции `IntToBin` возможно задание необязательного третьего параметра, который определяет количество двоичных цифр, после которых вставляется дополнительный пробел (довольно часто при записи двоичных чисел прибегают к четверкам или триадам битов, разделенных пробелами).

Функция преобразования вещественного значения в символьный формат допускает задание дополнительных аргументов, управляющих форматом результата:

```
s:=FloatToStr(num, Format[, Precision, m]);
```

Параметр `Format` может принимать одно из следующих значений:

- `ffCurrency` — перевод в символьный формат денежных единиц;
- `ffExponent` — перевод в символьный формат с плавающей запятой;
- `ffFixed` — перевод в символьный формат с фиксированной запятой;
- `ffGeneral` — перевод в формат с плавающей или фиксированной запятой;
- `ffNumber` — перевод в формат десятичного числа со вставкой символа разделителя "тысяч".

Параметр `Precision` с последующим числовым аргументом `m` управляет количеством значащих цифр.

Выбор того или иного представления в формате `ffGeneral` определяется системой по величине преобразуемого значения.

5.3.3. *Format* — универсальная функция преобразования данных

Наиболее широкими возможностями по преобразованию данных разного типа в их символьное представление обладает функция `Format`. Ее идеология заимствована из языков C, C++. В упрощенном варианте обращение к функции `Format` выглядит следующим образом:

```
s:=Format('форматные указатели', [список значений]);
```

Квадратные скобки, выделяющие второй аргумент, здесь являются обязательным элементом синтаксической конструкции. Функция `Format` возвращает результат преобразования в виде значения типа `String`. Чтобы не запутаться во всех возможностях этой универсальной функции, продемонстрируем некоторые из них на следующем примере (листинг 5.10).

Листинг 5.10. Программа `format1`

```
program format1;
uses SysUtils;
var
  i:integer=123;
  f:single=pi;
  ch:char='A';
  s:string='Hello, world!';
begin
  writeln(Format('i=%5d f=%8.2f ch=%s s=%s',[i,f,ch,s]));
  readln;
end.
```

Результат преобразования представлен на рис. 5.9.

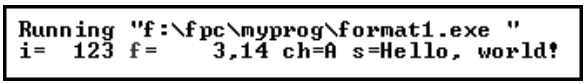


Рис. 5.9. Форматный вывод

В строке форматных указателей нашего примера находятся четыре указателя, каждый из которых начинается с символа `%`. Указатель `'%5d'` предписывает преобразовать целочисленное машинное значение (символ `d` — от англ. *decimal*) в строку из 5 позиций. То, что предшествует первому форматному указателю (`i=`), переносится в результат без каких-либо преобразований. Второй форматный указатель `'%8.2f'` управляет преобразованием вещественного значения (символ `f` — от англ. *float*), для которого в результирующей строке отводится 8 позиций, в том числе 2 позиции под дробную часть. Указатель `'%s'` говорит о том, что "преобразованию" подвергается строковое значение. Все остальные символы из форматной строки, не относящиеся к форматным указателям, переносятся в результирующую строку без изменений (`'f='`, `'ch='`, `'s='`). Количество преобразуемых значений в списке равно четырем, так что на каждое значение приходится свой форматный указатель. В общем случае количество форматных указателей и количество преобразуемых значений могут отличаться.

Форматный указатель всегда начинается с символа `%`. В тех случаях, когда мы хотим поместить символ процента в результирующую строку, форматная строка должна содержать два таких символа подряд — `%%`. В самом общем виде форматный указатель может содержать следующие компоненты:

`%[index:][-][w][.n]`α

Обязательными компонентами любого форматного указателя являются начальный символ `%` и последняя буква α. Именно эта буква оказывает главное влияние на характер преобразования.

Целое число `index` определяет порядковый номер элемента из списка преобразуемых значений, к которому относится данный форматный указатель. Индексация данных позволят повторять некоторые элементы из списка значений в разных местах результирующей строки. Отсчет индексов в списке значений ведется от 0. В табл. 5.4 приведены два примера использования индексов.

Таблица 5.4

Обращение	**Результат**
`Format('%d %d %d %0:d %d',[1,2,3,4])`	`1 2 3 1 2`
`Format('%d %d %d %0:d %3:d',[1,2,3,4])`	`1 2 3 1 4`

Символ "минус", предшествующий числовому значению ширины поля (w), означает, что на отведенных w позициях преобразованное значение должно быть прижато к левой границе поля. По умолчанию действует прижим к правой границе поля.

Число n, которое в руководствах иногда называют "точностью", по-разному влияет на вывод числовых и строковых данных. Для вещественных чисел в формате с фиксированной точкой оно задает количество цифр в дробной части. Для целых чисел и строк значение n определяет количество обязательно отображаемых цифр или символов.

Последняя буква форматного указателя, определяющая тип преобразования, может быть как строчной, так и прописной. Список возможных значений символа α приведен в табл. 5.5.

Таблица 5.5

Буква	Характер преобразования
d	Целое десятичное число со знаком (decimal)
e	Вещественное число с плавающей запятой (по умолчанию 15 значащих цифр)
f	Вещественное число с фиксированной запятой (по умолчанию 2 цифры в дробной части)
g	Обобщенный формат вещественных данных (e или f)
m	Число в денежном формате
n	Вещественное число с фиксированной запятой и символом разделения "тысяч"
p	Значение указателя (до 8 шестнадцатеричных цифр)
s	Значение строки
u	Целое десятичное число без знака (unsigned)
x	Целое число в шестнадцатеричном формате

Для дополнительного управления форматом денежных единиц необходимо использовать системные переменные CurrencyString, CurrencyFormat, NegCurFormat, ThousendSeparator, DecimalSeparator.

5.4. Строки типа *AnsiString*

Стандарт строк, утвержденных американским национальным институтом стандартов (American National Institute Standards, ANSI), довольно широко распространен в современных системах программирования. Его активно использует одна из лучших сред визуального программирования — Delphi. Этот тип данных включен и в состав языка Free Pascal. Главное преимущество строк типа AnsiString по сравнению

с короткими строками — отсутствие каких-либо ограничений на длину обрабатываемых данных. Но это преимущество имеет и обратную сторону медали — работа с динамически выделяемой памятью сопряжена со значительными накладными расходами. Как только переменной такого типа присваивается очередное значение, имеющее длину, отличную от длины предыдущего значения, приходится возвращать ранее занимаемый ресурс и выделять место под новое значение переменной. Это неизбежно приводит к появлению разрозненных освобожденных участков памяти, выполнению дополнительных работ по "уборке мусора" и, как следствие, к замедлению работы программы.

Как правило, строки неограниченной длины объявляются с помощью служебного слова `AnsiString`:

```
var
  as: AnsiString;
```

В момент объявления такой строки компилятор выделяет 4 байта для хранения указателя на соответствующее значение и заносит туда ссылку на константу `Nil`. Это означает, что строка пока что пуста. Это правило распространяется как на *глобальные*, так и на *локальные* `Ansi`-переменные, а также на *поля записей* того же типа. Оперативная память для хранения длинных строк выделяется в "куче" в момент присвоения переменной `as` первоначального или очередного значения. С этим значением связывается еще и *счетчик ссылок* (reference count), в котором фиксируется количество `Ansi`-переменных, использующих в текущий момент это значение. Когда `Ansi`-переменная перестает быть активной (выходит из "области видимости") из счетчика ссылок вычитается 1. Если с этим значением была связана единственная `Ansi`-переменная, то в момент ее активного существования в счетчике ссылок находилась единица. С потерей активности и появлением нуля в счетчике ссылок соответствующий участок кучи освобождается, а указатель пассивной переменной сбрасывается в `Nil`.

Разницу в тактике выделения и освобождения памяти под короткие и длинные строки демонстрирует программа `Ansi_1` (листинг 5.11).

Листинг 5.11. Программа `Ansi_1`

```
program Ansi_1;
var
  as1: AnsiString='Hello, world!';
  ss1: String[20]='Hello, world!';
begin
  writeln('SizeOf(as1) = ',SizeOf(as1));
  writeln('Length(as1) = ',Length(as1));
  as1:='';
  writeln('SizeOf(as1) = ',SizeOf(as1));
  writeln('Length(as1) = ',Length(as1));
  writeln('SizeOf(ss1) = ',SizeOf(ss1));
```

```
  writeln('Length(ss1) = ',Length(ss1));
  ss1:='';
  writeln('SizeOf(ss1) = ',SizeOf(ss1));
  writeln('Length(ss1) = ',Length(ss1));
  readln;
end.
```

Результаты ее работы приведены на рис. 5.10. Обратите внимание, что независимо от длины текущего значения под переменную `ss1` выделен 21 байт, тогда как после опустошения длинной строки `as1` от нее остался только четырехбайтовый указатель.

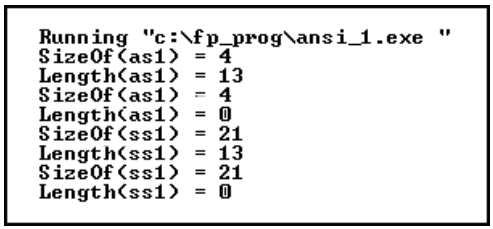


Рис. 5.10. Объемы занимаемой памяти и длины значений

Особым способом обрабатываются константы типа `AnsiString`:

```
const
  cas: AnsiString='Hello, world!';
```

В счетчик ссылок каждой такой константы заносится –1, что в машинном представлении для чисел без знака (счетчик ссылок — всегда число не отрицательное) соответствует максимально допустимому значению. Это позволяет хранить значения `Ansi`-констант до конца выполнения программы, т. к. сбросить такой счетчик до нуля практически не реально.

Второй способ причисления строк к типу `AnsiSting` связан с включением директивы `{$H+}`. В этом режиме строки, объявленные с помощью служебного слова `String` *без указания максимальной длины*, тоже рассматриваются компилятором как данные типа `AnsiString`:

```
{$H+}
var
  s1: String[20];        {строка типа ShortString}
  s2: String;            {строка типа AnsiString}
...
```

Над значениями строк неограниченной длины можно выполнять все те же операции, что и над короткими строками. Значения строк типа `AnsiString` можно присваивать переменным типа `ShortString` с естественным обрезанием результата по максимальной длине короткой переменной. Без каких-либо ограничений разрешены обратные присвоения.

Кроме того, в модуле `StrUtils` предусмотрено порядка 90 дополнительных функций и процедур для обработки строк типа `AnsiString`.

5.5. Строки типа *PChar*

Строки типа `PChar` тоже относятся к строкам неограниченной длины, но способ их представления в памяти отличается от `Ansi`-строк. Имя переменной типа `PChar` тоже является указателем на значение переменной, хранящееся в куче. Но этот указатель не типизирован — он "знает" адрес значения, но не "знает" его длину. Признаком конца строки типа `PChar` является байт с нулевым значением, который автоматически добавляется вслед за последним значащим символом. В момент объявления переменной типа `PChar` компилятор выделяет 4 байта под указатель и заносит туда константу `Nil`, что эквивалентно созданию пустой строки. К любому символу значения строки типа `PChar` можно обратиться по его индексу, отсчет которых производится от 0.

Константа или переменная типа `PChar` может быть объявлена или создана разными способами, которые демонстрируются в программе `pchar_1` (листинг 5.12).

Листинг 5.12. Программа `pchar_1`

```
program pchar_1;
const
  pcs1 : PChar = '1 : Hello, world!';
var
  pcs2 : PChar = '2 : Hello, world!';
  ss3 : String = '3 : Hello, world!'+#0+'ABC';
  as4 : AnsiString = '4 : Hello, world!';
  ss5 : String[20];
begin
  writeln('pcs1 = ',pcs1);
  writeln('pcs2 = ',pcs2);
  writeln(' ss3 = ',ss3);
  pcs2 := @ss3[1];
  writeln('pcs2 = ',pcs2);
  pcs2 := PChar(as4);
  writeln('pcs2 = ',pcs2);
  ss5 := '5 : Hello, world!'#0;
  pcs2 := @ss5[1];
  writeln('pcs2 = ',pcs2);
  readln;
end.
```

Результаты ее работы приведены на рис. 5.11. Обратите внимание на некоторые технические детали программы `pchar_1`. Во-первых, когда мы собираемся присвоить переменной типа `PChar` значение короткой строки, то должны сами позаботиться о включении в ее состав байта с нулевым кодом (`#0`). Во-вторых, если такой байт не является последним (см., например, значение переменной `ss3`), то все следующие за ним байты к `PChar`-строке не присоединяются. В-третьих, указателю `PChar`-строки присваивается *адрес* первого значащего символа короткой строки. Наконец, если указателю типа `PChar` надо присвоить указатель типа `AnsiString`, то нельзя обойтись без явного преобразования типов (`PChar(as4)`).

```
Running "f:\fpc\myprog\pchar_1.exe "
pcs1 = 1 : Hello, world!
pcs2 = 2 : Hello, world!
 ss3 = 3 : Hello, world! ABC
pcs2 = 3 : Hello, world!
pcs2 = 4 : Hello, world!
pcs2 = 5 : Hello, world!
```

Рис. 5.11. Формирование данных типа `PChar`

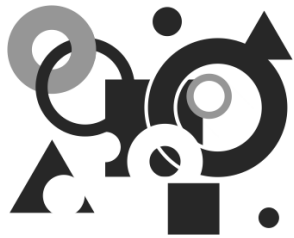
Кроме операций ввода/вывода и присвоения над данными типа `PChar` можно выполнять довольно много других операций. Но все они подобно языкам C, C++ реализуются с помощью 26 процедур и функций, включенных в состав модуля `Strings`.

5.6. Строки типа *WideString*

"Широкие" строки типа `WideString` очень похожи на `PChar`-строки. Их главное отличие состоит в том, что на кодировку каждого символа в значении широкой строки отводятся два байта, в которые записывается код из таблицы Unicode. Признаком конца значения широкого кода является двухбайтовый ноль. Естественно, что создавать значения широких строк нужно в специальном текстовом редакторе.

Для обработки широких строк можно использовать почти все подпрограммы и функции, описанные в *разд. 5.2* (`Length`, `Copy`, `Delete`, `Insert`, `Pos`, `SizeOf`).

Глава 6



Массивы в языке Free Pascal

Массивы — один из наиболее распространенных объектов ряда разделов вычислительной математики. Существует довольно много прикладных задач, постановка и алгоритмы решения которых используют матричную символику. Среди них — исследование систем линейных алгебраических и обыкновенных дифференциальных уравнений, определение областей устойчивого поведения динамических систем, расчеты электрических цепей, задачи линейного и нелинейного программирования, статистическая обработка результатов наблюдений и многие другие.

С точки зрения систем программирования, массив представляет собой набор *однотипных данных*, расположенных в оперативной памяти таким образом, чтобы по индексам элементов можно было легко вычислить адрес соответствующего значения. С примерами простейших одномерных массивов мы уже сталкивались при изучении строковых данных, каждый символ которых был представлен однобайтовым (ASCII) или двухбайтовым (Unicode) кодом. Если одномерный массив `A` состоит из элементов, расположенных в памяти подряд по возрастанию индексов, и каждый элемент занимает по `k` байт, то адрес `i`-го элемента вычисляется по формуле:

```
адрес(A[i])=адрес(A[0])+i*k
```

Если мы имеем дело с двумерным массивом `B` размерности `M×N`, расположенным в памяти по строкам, то адрес элемента `B[i,j]` вычисляется по формуле:

```
адрес(B[i,j])=адрес(B[0,0])+(i*N+j)*k
```

Приведенные выше формулы незначительно усложняются в тех случаях, когда начальные индексы отсчитываются не только от нуля, что характерно для Паскаля.

Использование массивов позволяет заменить большое количество индивидуальных имен каждого объекта одним групповым именем набора данных, вслед за которым в квадратных скобках задаются один или несколько индексов, определяющих местоположение требуемого значения. Естественно, что такая возможность упрощает и массовую обработку данных в соответствующих циклах программы. Несколько примеров, приведенных далее, демонстрируют фрагменты программ, реализующих некоторые алгоритмы линейной алгебры (листинги 6.1—6.4).

Листинг 6.1. Сложение векторов (c = a + b)

Для сложения двух векторов одинаковой размерности используются следующие формулы:

$$\mathbf{a} = (a_1, a_2, ..., a_n),$$
$$\mathbf{b} = (b_1, b_2, ..., b_n),$$
$$\mathbf{c} = \mathbf{a} + \mathbf{b} = (c_1, c_2, ..., c_n),$$
$$c_j = a_j + b_j.$$

Соответствующий фрагмент программы выглядит следующим образом:

```
...
for j := 1 to n do
  c[j] := a[j]+b[j];
```

Листинг 6.2. Скалярное произведение векторов (a, b)

Скалярное произведение двух векторов вычисляется по формуле: $\sum_{k=1}^{n} a_k \times b_k$.

Соответствующий фрагмент программы выглядит следующим образом:

```
...
s := 0;
for k := 1 to n do
  s := s + a[k]*b[k];
```

Листинг 6.3. Умножение двух квадратных матриц (C = A×B)

Формула для вычисления элемента имеет вид: $c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$.

Соответствующий фрагмент программы выглядит следующим образом:

```
...
for i:=1 to n do
  for j:=1 to n do
    begin
      s := 0;
      for k := 1 to n do
        s := s + a[i,k]*b[k,j];
      c[i,j]:=s;
    end;
```

Листинг 6.4. Определение минимального элемента вектора

Задача сортировки числовых массивов сводится к определению минимального элемента вектора, который может быть найден следующим образом:

```
min := a[1];
for j := 2 to n do
  if min > a[j] then min := a[j];
```

В большинстве задач приходится иметь дело с массивами, элементами которых являются числа того или иного типа, символы и строки фиксированной длины. Однако приходится обрабатывать и битовые массивы, каждый элемент которых представлен одним или несколькими двоичными разрядами, соответствующими, например, кодам цветности пикселов графических изображений.

6.1. Статические и динамические массивы языка Free Pascal

Free Pascal так же, как и Object Pascal, поддерживает массивы двух категорий. Первую из них составляют *традиционные* массивы Паскаля, при объявлении которых в явном или косвенном виде указываются *конкретные границы изменения каждого индекса*:

```
const
  k=5;
  q=6;
type
  mat_q_k = array [1..q,1..k] of integer;
var
  sa1: array [3..10] of byte;   // явное задание границ
  sa2: mat_q_k;                 // косвенное задание границ
```

В ряде алгоритмических языков приняты соглашения о минимальном значении каждого индекса — в C и C++ индексы отсчитываются от 0, в Фортране — от 1, в Бейсике минимальной границей индекса можно управлять (оператор `OPTION BASE`). Паскаль допускает в качестве минимальных индексов любые значения порядковых данных. И ими могут быть не только числа, но и, например, символы:

```
var
  ch: array ['A'..'Z'] of integer;
  str: string;
...
begin
  ...
  inc(ch[str[j]]);
  ...
```

Приведенный фрагмент наиболее простым способом позволяет подсчитать частоту появления тех или иных букв в обрабатываемом тексте.

Существует несколько вариантов для объявления многомерных массивов. Например, целочисленная матрица `sa2`, содержащая `q` строк и `k` столбцов, может быть включена в текст программы еще одним из следующих описаний:

```
var
  sa2: array [1..q,1..k] of integer;
```

или

```
var
  sa2: array [1..q][1..k] of integer;
```

или

```
var
  sa2: array [1..q] of array [1..k] of integer;
```

В языке Free Pascal для обозначения традиционных массивов используется термин *"статические массивы"*, хотя он не совсем точно описывает логику выделения памяти для их хранения. Дело в том, что традиционные массивы в зависимости от места их определения делятся на *глобальные* и *локальные*. Глобальные массивы описываются в одном из разделов объявлений *головной программы*. В отличие от них описания локальных массивов встречаются *внутри функций* или *подпрограмм*. Для хранения глобальных массивов компилятор выделяет память *перед началом работы программы и чистит ее* (числовые массивы заполняются нулями, а строковые — пустыми строками). Глобальные массивы хранятся в памяти до окончания работы программы и доступны в любой программной единице (функции или подпрограмме). В отличие от этого память для хранения локальных массивов выделяется *во время работы программы* в тот момент, когда вызывается та или иная программная единица. Эта память *не чистится и доступна только в рамках той программной единицы, где она объявлена*. При возврате из программной единицы память из-под локальных массивов *освобождается*. Поэтому локальные массивы появляются в оперативной памяти динамически, и термин "статический" в такой ситуации не очень удачен.

Вторую категорию составляют действительно *динамические* массивы, объявление которых *не содержат указания о границах изменения индексов*:

```
var
  da1: array of integer;           {одномерный массив}
  da2: array of array of integer;  {двумерный массив}
```

Так как объявление динамического массива не сопровождается указанием о длине, то компилятор выделяет для каждого динамического массива (глобального или локального) по 4 байта. В них хранится указатель на начало значений элементов динамического массива. В начальный момент значения всех таких указателей равны 0, что соответствует "пустым" динамическим массивам (ситуация напоминает стратегию распределения памяти под строки типа `AnsiString`).

Фактическое выделение памяти под динамические массивы производится *только во время работы программы* путем вызова процедуры `SetLength` (дословно — *установить длину*):

```
SetLength(da1,100);
```

Такое обращение эквивалентно "статическому" описанию вида:

```
da1: array [0..99] of integer;
```

Индексы динамических массивов *всегда отсчитываются от 0*, поэтому в обращении к процедуре `SetLength` кроме имени динамического массива задается количество элементов. Память, *впервые выделяемая* динамическому массиву, *всегда чистится*.

Во время работы программы к процедуре `SetLength` можно обращаться много раз. Если при очередном обращении новая длина больше предыдущей, то значения ранее вычисленных элементов сохраняются, а всем добавляемым элементам присваиваются нулевые значения. Если новая длина динамического массива меньше текущей, то сохраняются значения начальных элементов, "лишние" элементы будут безвозвратно потеряны.

Для выделения памяти под двумерный динамический массив к процедуре `SetLength` обращаются с тремя параметрами, задавая количество строк и количество столбцов:

```
SetLength(da2,4,6);
```

Такое обращение эквивалентно "статическому" описанию вида:

```
da2: array [0..3, 0..5] of integer;
```

Если динамический массив был объявлен в процедуре или функции, то он является локальным и после выхода из программной единицы память, занимаемая значениями элементов массива, освобождается.

К дополнительным средствам досрочного возврата памяти, занятой элементами динамического массива, относятся следующие способы:

```
da1 := Nil;          {прямая засылка нуля в указатель}
da2 := Nil;
```

или

```
SetLength(da1,0);    {косвенная засылка нуля в указатель}
SetLength(da2,0);
```

или

```
Finalize(da1);       {финальное освобождение ресурсов}
Finalize(da2);
```

6.2. Определение длины и размеров массивов

Под термином *"длина одномерного статического массива"* обычно понимают объем оперативной памяти в байтах, занятых элементами массива.

Для определения этой характеристики обычно прибегают к функции `SizeOf`:

```
var
  sa1: array [3..15] of double;
begin
  ...
  writeln(SizeOf(sa1));
```

Каждый элемент массива `sa1` занимает 8 байт. Поскольку в массиве объявлено 13 элементов, то для хранения их значений компилятор отведет 8×13 = 104 байта. Именно эта величина и будет выведена оператором `writeln`.

Размер одномерного статического массива совпадает с количеством его элементов. И для определения этой характеристики обычно прибегают к функции `Length`:

```
Length(sa1) = 13
```

В языке Free Pascal предусмотрены еще две функции — `Low` и `High` для определения *минимального и максимального* значения индекса *одномерного статического массива*:

```
Low(sa1) = 3
High(sa1)= 15
```

С одномерными динамическими массивами дело обстоит несколько иначе:

```
var
  da1: array of double;
```

Независимо от того, выделена ли оперативная память под значения элементов динамического массива или еще не выделена, результат функции `SizeOf(da1)` всегда равен 4. Фактически это объем памяти, занимаемый указателем `da1` на соответствующие данные. Функция `Length` выдает фактическое значение, равное текущему количеству элементов динамического массива, и пока обращения к процедуре `SetLength` еще не было, эта характеристика равна 0. Применять функцию `Low` к динамическим массивам смысла не имеет, т. к. минимальное значение индекса у таких массивов всегда равно 0. До обращения к процедуре `SetLength` значение функции `High(da1)` равно –1. В любом случае для одномерного динамического массива имеет место следующее равенство:

```
High(da1) = Length(da1) - 1
```

Программа `matrix_1` демонстрирует вышесказанное (листинг 6.5).

Листинг 6.5. Программа `matrix_1`

```
program matrix_1;
var
  sa1: array [3..15] of double;
  da1: array of double;
```

```
begin
  writeln('Length(sa1) = ',Length(sa1));
  writeln('SizeOf(sa1) = ',SizeOf(sa1));
  writeln('Low(sa1) = ',Low(sa1));
  writeln('High(sa1)= ',High(sa1));
  writeln;
  writeln('Length(da1) = ',Length(da1));
  writeln('SizeOf(da1) = ',SizeOf(da1));
  writeln('Low(da1) = ',Low(da1));
  writeln('High(da1)= ',High(da1));
  writeln;
  SetLength(da1,13);
  writeln('Length(da1) = ',Length(da1));
  writeln('SizeOf(da1) = ',SizeOf(da1));
  writeln('Low(da1) = ',Low(da1));
  writeln('High(da1)= ',High(da1));
  readln;
end.
```

Результаты ее работы приведены на рис. 6.1.

```
Running "f:\fpc\myprog\matrix_1.exe "
Length(sa1) = 13
SizeOf(sa1) = 104
Low(sa1) = 3
High(sa1)= 15

Length(da1) = 0
SizeOf(da1) = 4
Low(da1) = 0
High(da1)= -1

Length(da1) = 13
SizeOf(da1) = 4
Low(da1) = 0
High(da1)= 12
```

Рис. 6.1. Значения стандартных функций для одномерных массивов

С двумерными массивами, как статическими, так и динамическими, ситуация кажется еще запутаннее. Однако и в ней не так уж и сложно разобраться, если усвоить следующие соглашения языка Паскаль. Во-первых, для двумерного массива `q` размером `n×m` компилятор заводит `n` указателей. Первый из них имеет имя `q`, совпадающее с именем массива. Он является указателем *на начало массива* и одновременно *на первую строку массива*. Если минимальное значение первого индекса равно `k`, то указатель `q[k]` тоже "смотрит" на первую строку массива (т. е. `q` и `q[k]` — это два имени одного и того же указателя). Следующий указатель с именем `q[k+1]` "смотрит" на вторую строку массива и т. д.

Эти детали позволят вам разобраться с результатами работы программ `matrix_2` (листинг 6.6 и рис. 6.2) и `matrix_3` (листинг 6.7 и рис. 6.3).

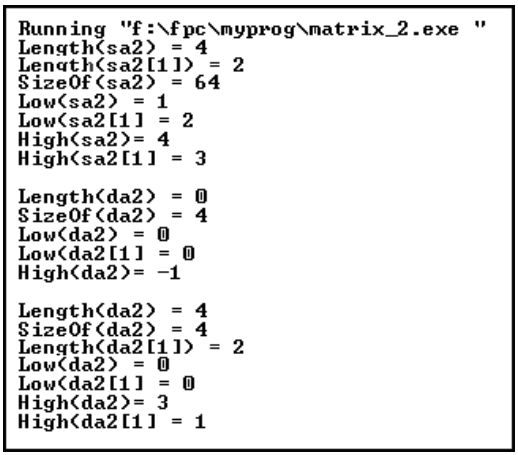


Рис. 6.2. Результаты работы программы `matrix_2`

Листинг 6.6. Программа `matrix_2`

```
program matrix_2;
var
  sa2 : array [1..4,2..3] of double;
  da2 : array of array of double;
begin
  writeln('Length(sa2) = ',Length(sa2));
  writeln('SizeOf(sa2) = ',SizeOf(sa2));
  writeln('SizeOf(sa2[1]) = ',SizeOf(sa2[1]));
  writeln('Low(sa2) = ',Low(sa2));
  writeln('Low(sa2[1] = ',Low(sa2[1]));
  writeln('High(sa2)= ',High(sa2));
  writeln('High(sa2[1] = ',High(sa2[1]));
  writeln;
  writeln('Length(da2) = ',Length(da2));
  writeln('SizeOf(da2) = ',SizeOf(da2));
  writeln('Low(da2) = ',Low(da2));
  writeln('Low(da2[1] = ',Low(da2[1]));
  writeln('High(da2)= ',High(da2));
  writeln;
  SetLength(da2,4,2);
  writeln('Length(da2) = ',Length(da2));
```

```
  writeln('SizeOf(da2) = ',SizeOf(da2));
  writeln('SizeOf(da2[1]) = ',SizeOf(da2[1]));
  writeln('Low(da2) = ',Low(da2));
  writeln('Low(da2[1] = ',Low(da2[1]));
  writeln('High(da2)= ',High(da2));
  writeln('High(da2[1] = ',High(da2[1]));
  readln;
end.
```

Листинг 6.7. Программа `matrix_3`

```
program matrix_3;
var
  a: array [1..3,1..2] of integer = ((1,2),
                                     (3,4),
                                     (5,6));
  p: ^integer;
begin
  p:=@a;
  writeln('a    = ',p^,' a+1    = ',p^+1);
  p:=@(a[1]);
  writeln('a[1] = ',p^,' a[1]+1 = ',p^+1);
  p:=@(a[2]);
  writeln('a[2] = ',p^,' a[2]+1 = ',p^+1);
  p:=@(a[3]);
  writeln('a[3] = ',p^,' a[3]+1 = ',p^+1);
  readln;
end.
```

```
Running "f:\fpc\myprog\matrix_3.exe "
a    = 1 a+1    = 2
a[1] = 1 a[1]+1 = 2
a[2] = 3 a[2]+1 = 4
a[3] = 5 a[3]+1 = 6
```

Рис. 6.3. Результаты работы программы `matrix_3`

6.3. Инициализация глобальных статических массивов

Объявление глобальных статических массивов может быть совмещено с присвоением их элементам начальных значений.

Для описаний следующего вида используется термин *"инициализация"*:

```
var
  sa1: array [1..10] of integer = (1,2,3,4);
  sa2: array [1..2, 1..3] of integer = ((1,2,3),
                                        (4,5,6));
```

6.4. Выделение памяти локальным и глобальным массивам

Программа `matrix_4` демонстрирует тактику выделения памяти для глобальных массивов — статического `sa1` и динамического `ds1` (листинг 6.8). Массиву `sa1` память выделяется *сразу* и всем его элементам присваиваются *нулевые значения*. Для массива `ds1` сначала выделяется 4-байтовый указатель, в который заносится `Nil`. После первого обращения к процедуре `SetLength` массиву `ds1` выделяется *чистая память*. При повторном обращении к процедуре `SetLength` массиву `ds1` выделяется *новая память*, в которую копируются накопленные ранее данные, а хвост — чистится. Функция `SizeOf` для статического массива выдает объем занятой памяти в байтах, для динамического — только объем памяти, занятой указателем. Функция `Length` выдает длину любого массива в количестве элементов.

Листинг 6.8. Программа `matrix_4`

```
program matrix_4;
var
  sa1: array [1..10] of integer;
  da1: array of integer;
  j: integer;
begin
  for j:=1 to 10 do
    write(sa1[j]:4);
  writeln;
  writeln('Length(sa1) = ',Length(sa1));
  writeln('SizeOf(sa1) = ',SizeOf(sa1));
  writeln('Length(da1) = ',Length(da1));
  writeln('SizeOf(da1) = ',SizeOf(da1));
  SetLength(da1,5);
  for j:=0 to 4 do
  begin
    write(da1[j]:4);
    da1[j]:=j;
  end;
```

```
  writeln;
  SetLength(da1,10);
  for j:=0 to 9 do
    write(da1[j]:4);
  writeln;
  writeln('Length(da1) = ',Length(da1));
  writeln('SizeOf(da1) = ',SizeOf(da1));
  readln;
end.
```

```
Running "f:\fpc\myprog\matrix_4.exe "
   0   0   0   0   0   0   0   0   0   0
Length(sa1) = 10
SizeOf(sa1) = 40
Length(da1) = 0
SizeOf(da1) = 4
   0   0   0   0   0
   0   1   2   3   4   0   0   0   0   0
Length(da1) = 10
SizeOf(da1) = 4
```

Рис. 6.4. Результаты работы программы `matrix_4`

Программа `matrix_5` демонстрирует тактику выделения памяти для локальных массивов — статического массива `lsa` и динамического массива `lda` (листинг 6.9). Локальному статическому массиву память выделяется в момент входа в подпрограмму, но эта память *не чистится*. Локальному динамическому массиву поначалу выделяется 4 байта под указатель, но после обращения к процедуре `SetLength` выделяется *чистая память*.

Листинг 6.9. Программа `matrix_5`

```
program matrix_5;
procedure qq;
var
  lsa: array[1..5] of integer;
  lda: array of integer;
  j: integer;
begin
  writeln('SizeOf(lsa)=',SizeOf(lsa));
  writeln('Length(lsa)=',Length(lsa));
  writeln('SizeOf(lda)=',SizeOf(lda));
  writeln('Length(lda)=',Length(lda));
  for j:=1 to 5 do
    write(lsa[j],' ');
  writeln;
  SetLength(lda,5);
```

```
  writeln('SizeOf(lda)=',SizeOf(lda));
  writeln('Length(lda)=',Length(lda));
  for j:=0 to 4 do
    write(lda[j]:4,' ');
  writeln;
end;
begin
  qq;
  readln;
end.
```

```
Running "f:\fpc\myprog\matrix_5.exe "
SizeOf(lsa)=20
Length(lsa)=5
SizeOf(lda)=4
Length(lda)=0
21036948 4210404 2147336192 4238980 4234500
SizeOf(lda)=4
Length(lda)=5
   0    0    0    0    0
```

Рис. 6.5. Результаты работы программы `matrix_5`

6.5. Операции над однотипными массивами

В Паскале выделяют массивы, *совместимые по операции присваивания*. К ним относятся массивы, объявленные с использованием одного и того же типа:

```
type
  m10_b = array [1..10] of byte;
var
  a1 : m10_b;
  a2 : m10_b;
  a3, a4 : m10_b;
  a5 : array [1..10] of byte
  a6, a7 : array [1..10] of byte;
```

В приведенном выше фрагменте массивы `a1`, `a2`, `a3` и `a4` совместимы между собой по присваиванию. Массивы `a6` и `a7` совместимы между собой, но не совместимы ни с первой группой массивов, ни с массивом `a5`. Массив `a5` не совместим ни с одним из остальных массивов. Столь странное определение *совместимости* объясняется довольно просто: при изменении в программе единственной строчки совместимые массивы продолжают оставаться совместимыми. Например, мы меняем описание типа `m10_b`. Одновременно с этим автоматически меняются типы массивов `a1`, `a2`, `a3` и `a4`. Но это изменение не влияет на типы массивов `a5`, `a6` и `a7`. Изменение типа в последней строке оказывает воздействие только на массивы `a6` и `a7`.

Для массивов, совместимых по присваиванию, допускается операция вида:

```
a1 := a2;
```

Эта операция по-разному выполняется для статических и динамических массивов. Для статических массивов такое присвоение эквивалентно копированию содержимого одного массива в элементы другого массива, т. е. при этом содержимое одного участка оперативной памяти переписывается в область определения другого массива. Для динамических массивов такого копирования не происходит. Вместо этого копируется только значение указателя на массив, а область памяти, ранее занимаемая обновляемым массивом, освобождается (похожая ситуация имела место при присвоении значения строки типа `AnsiString` другой строке того же типа).

6.6. Модуль *Matrix*

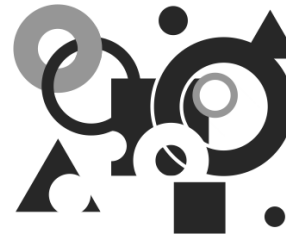
В состав системы Free Pascal включен модуль (unit) `Matrix`, построенный по классической схеме с использованием классов. Он обеспечивает:

- формирование и инициализацию объектов типа *вектор* размером 2, 3 и 4;
- формирование и инициализацию объектов типа *квадратные матрицы* размером 2×2, 3×3 и 4×4;
- выполнение всех арифметических операций над объектами указанных типов;
- выполнение основных операций линейной алгебры (извлечение или изменение строк или столбцов, формирование нулевых или единичных матриц, транспонирование матриц, вычисление определителей, обращение матриц).

Элементами векторов и матриц могут быть данные любых вещественных типов — `single`, `double`, `extended`.

Основное назначение функций и процедур модуля `Matrix` — помощь в создании программ обработки двумерных и трехмерных векторных графических объектов. С помощью этих подпрограмм довольно легко реализуются различные аффинные преобразования в 2D- и 3D-графике (перемещение точек, параллельный перенос отрезков, сжатие и растяжение геометрических фигур, повороты и др.).

Глава 7



Множества

Если массив представляет собой упорядоченный набор однотипных данных, то множество — это *не упорядоченный* набор *не повторяющихся* объектов *любой природы*. Максимальное количество объектов, из которых может состоять множество, не должно превышать 255. Специфика любого множества заключается в том, что при его описании должен быть перечислен весь список значений, который может входить в состав множества. Способ такого перечисления может быть разным:

```
type
  s16 = set of (0..9,'a'..'f');
var
  a1: s16;
  bukwa: set of ('a', 'c', 'f', 'd', 'b', 'e');
  cifra: set of 0..9;
```

В этом примере объявлен тип с именем `s16`, описывающий множество, в которое могут входить числа (цифры от 0 до 9) и символы (малые буквы латинского алфавита от `'a'` до `'f'`). В разделе переменных объявлена переменная с именем `a1` типа `s16`. Пока что она представляет пустое множество, не содержащее ни одного элемента. Но в процессе работы программы в состав `a1` может быть включена любая комбинация из неповторяющихся данных, допустимых типом `s16`:

```
a1:=[5,1,'b'];
```

Комбинацию значений элементов множества, заключенную в квадратные скобки, принято называть *конструктором множества*. Конструктор множества может быть и пустым, например, `[]`.

Переменные `bukwa` и `cifra`, допустимые значения которых заданы явно, после объявления тоже пока пусты.

Над элементами конкретного множества определены операции *сложения* (добавления одного или нескольких элементов из допустимого набора), *вычитания* (удаления из текущего значения множества одного или нескольких элементов) и проверки присутствия в составе множества указанного значения (операция `in`, выдающая результат типа `boolean`):

```
a1 := a1 - [5,'b'];      {теперь a1=[1]}
a1 := a1 + [3];          {теперь a1=[1,3]}
if 5 in a1 then ...      {это условие не выполнено}
```

Над двумя множествами `A` и `B` определены операции, принятые в математике, — *объединение* (сумма множеств `A+B`), *пересечение* (общая часть множеств `A*B`) и *вычитание* (разность `A-B`, т. е. элементы `A`, не принадлежащие `B`). Содержимое двух множеств можно сравнивать, однако из шести возможных операций отношения допустимы лишь четыре — проверка на равенство (`if A = B then...`), на неравенство (`if A <> B then...`), на больше или равно (`if A >= B then...`), на меньше или равно (`if A <= B then...`).

Наиболее интересным для математиков примером использования множеств является программа построения таблицы простых чисел методом Эратосфена. Эратосфен Киренский (282—202 гг. до н. э.) — известный древнегреческий ученый. Для формирования таблицы простых чисел он взял длинный папирус и написал на нем все числа от 2 до 1000. Затем он с помощью шила проткнул каждое второе число после 2, т. е. исключил все остальные четные числа. Сохранив первое оставшееся после 2 число 3, он проткнул каждое третье число, т. е. исключил числа, кратные 3. Мы надеемся, что как умный человек, Эратосфен дважды не прокалывал ранее проколотые числа (например, 6 кратно и 2, и 3). Затем после 5 было проколото каждое пятое число и т. д. В конце концов, на папирусе остались простые числа, имеющие только два разных делителя — единицу и само себя. Описанный алгоритм известен в литературе под названием "решето Эратосфена" (от англ. *sieve* — решето). В листингах 7.1 и 7.2 приводятся две программы — `sieve_1` и `sieve_2`. Одна из них формирует множество простых чисел путем их накапливания, а вторая — путем вычеркивания составных чисел.

Листинг 7.1. Программа `sieve_1`

```
program sieve_1;
const
  maxN = 255;
var
  primes: set of 2..maxN;
  i,j: integer;
begin
  primes:=[2..maxN];
  for i:=2 to maxN do
    if i in primes then
    begin
      write(i:4);
      for j:=1 to (maxN div i) do
        primes:=primes-[i*j];
    end;
    readln;
end.
```

Результат работы программы `sieve_1` приведен на рис. 7.1.

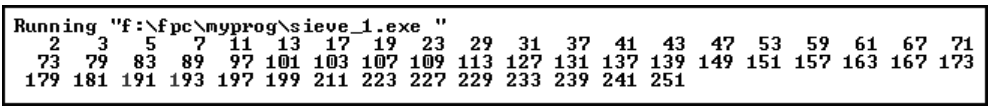


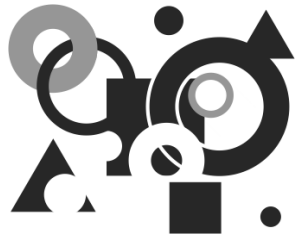
Рис. 7.1. Результаты работы программы `sieve_1`

Листинг 7.2. Программа `sieve_2`

```
program sieve_2;
const
  N=255;
var
  Sieve : set of 2..N = [2..N];
  Primes: set of 2..N;
  Next: byte=2;
  j: word;
begin
  repeat
    while not(Next in Sieve) do
      Next:=Next+1;
    Primes:=Primes+[Next];
    j:=Next;
    while j<=N do
      begin
        Sieve:=Sieve-[j];
        j:=j+Next;
      end;
  until Sieve=[];
  for j:=2 to N do
    if j in Primes then write(j:4);
  readln;
end.
```

Результаты работы программ `sieve_1` и `sieve_2` идентичны.

Глава 8



Записи

Своим происхождением *записи* обязаны таблицам — одному из наиболее распространенных документов представления данных, который возник задолго до появления ЭВМ. Одна из первых программных систем, взявших на вооружение записи — "Коммерческий Транслятор" (COMTRAN), разработанный в 1959 г. группой сотрудников IBM. Позднее данные типа запись составили основу алгоритмического языка COBOL (COmmon Business Oriented Language), ориентированного на обработку коммерческих документов.

Для объявления данных типа запись в Паскале используется служебное слово `record` (запись). Продемонстрируем этот тип данных на примере описания полей в каждой строке табл. 8.1.

Таблица 8.1

Тип	Знак	Длина (байт)	Количество значащих цифр	Диапазон	
				Min	Max
`Byte`	Нет	1	3	0	255
`ShortInt`	Есть	1	3	–127	128
`Word`	Нет	2	5	0	65 535

```
type
  MinMax = record
    Min : extended;
    Max : extended;
  end;
  NumData = record
    Tip : string [12];    {поле для обозначения типа}
    Sgn : boolean;        {поле для указания знака}
    Len : byte;           {поле для указания длины}
    Num_Cif : byte;       {поле для числа значащих цифр}
    Range : MinMax;       {поле для задания диапазона}
  end;
```

Структура полей каждой строки таблицы, эквивалентная формату записи, обычно описывается в разделе типов данных. Описание начинается с *имени типа*, вслед за которым после знака равенства записывается служебное слово `record`. Каждое поле записи характеризуется *именем поля* и *типом значения*, которое может на этом поле располагаться. Концом описания записи считается закрывающая операторная "скобка" `end`, завершаемая символом *точки с запятой*.

В нашем примере поле *Диапазон* (имя `Range`), в свою очередь, является подтаблицей, содержащей два поля с именами `Min` и `Max`. Поэтому в описании типов сначала описана запись с именем `MinMax`, а уже потом этот тип данного использован в описании формата поля `Range`.

После описания структуры данных в разделе `type` мы можем использовать указанный тип для описания отдельных переменных или массивов:

```
var
  b1: NumData;
  a1: array [1..10] of NumData;
```

Для присвоения конкретных значений полям записи используются *составные* имена:

```
b1.Tip := 'byte';        {это поле имеет тип String}
b1.Sgn := false;         {это поле имеет тип boolean}
b1.Len := 1;             {это поле имеет тип byte}
b1.Num_Cif := 3;         {это поле имеет тип byte}
b1.Range.Min := 0;       {это поле имеет тип extended}
b1.Range.Max := 255;
a1[1].Tip := 'ShortInt';
a1[1].Sgn := true;
a1[1].Len := 1;
a1[1].Num_Cif := 3;
a1[1].Range.Min := -127;
a1[1].Range.Max := 128;
```

Описание переменных типа запись можно совместить с формированием начального значения полей:

```
type
  TPoint = record
    x, y : single;
  end;
  TVector = array [0..1] of TPoint;
var
  P1: TPoint = (x:0.5; y:-0.5);
  Line: TVector = ((x:0; y:0),(x:1.5; y:2.0));
```

В качестве типов полей могут выступать и перечисления:

```
type
  Person = record
    Name: string [20];
    Sex: (Male, Female);
    Age: byte;
    Married: Boolean;
  end;
var
  a1: Person;
const
  b1: Person = (Name:'Иванов'; Sex:Male; Age:25; Married:true);
```

Если переменные описаны с помощью одного и того же типа, то они совместимы по присваиванию, и в программе может быть использован оператор следующего вида:

```
a1 := b1;
```

Данные типа запись появились в одной из самых ранних версий Паскаля и долгое время их основным назначением были обработка таблиц, хранящихся в оперативной памяти, и обмен данными с записеориентированными файлами. Однако именно с записями были связаны первые шаги по развитию языка Паскаль в направлении объектно-ориентированного программирования.

8.1. Упрощение доступа к полям записи

Использование составных имен доставляет определенные неудобства для программистов. Поэтому в Паскале был придуман способ сокращенного набора имен полей. Он заключается в том, что перед фрагментом программы, активно использующим имена полей, помещается специальный заголовок — оператор `with` с опускаемой добавкой.

```
type
  BirthDay = record
    Day, Month: byte;
    Year: word;
  end;
var
  MySon: BirthDay;
```

Полное обращение к полям записи выглядит следующим образом:

```
MySon.Day := 14;
MySon.Month := 3;
Myson.Year := 1972;
```

Укороченный вариант обращения к полям записи имеет вид:

```
with MySon do
  Day:= 14;
  Month := 3;
  Year := 1972
end;
```

Особенно ощутимый выигрыш достигается при наличии вложенных полей (т. е. таких полей, которые, в свою очередь, являются записями):

```
var
  MySon: record
    Name: String;
    BD: BirthDay;
```

К полям записи `MySon` возможен один из следующих вариантов обращения:

```
MySon.BD.Month := 3;
...
with MySon.BD do
  Month := 3;
...
with MySon, BD do
  Month := 3;
...
with MySon do
  with BD do
    Month :=3;
```

8.2. Записи с вариантами

Примеры записей, приводившиеся до сих пор, характеризовались фиксированным набором полей. Однако иногда встречаются документы, имеющие в начале фиксированный набор полей, а в конце — набор полей, количество и состав которых зависят от некоторого параметра. Типичным примером документа такого рода является анкета служащего, используемая в отделах кадров. Начальные поля такой анкеты одинаковы и для мужчин, и для женщин (фамилия, имя, отчество, дата и место рождения, адрес и т. п.). Однако в дополнительных полях для мужчин могут присутствовать такие поля как отношение к воинской службе, учетно-воинская специальность и другие данные, характерные только для представителей мужского пола. Такого рода записи, у которых последние поля могут иметь несколько разных форматов, называют *записями с вариантами*.

Мы рассмотрим специфику их обработки на конкретном примере. Представим себе запись с именем `TFigure` (Фигура), которая предназначена для хранения ин-

формации об одном из следующих графических объектов — прямоугольнике (`Rectangle`), квадрате (`Quadrate`) и окружности (`Circle`). Очевидно, что каждая из этих фигур характеризуется разным набором параметров. Общими данными для них могут быть:

- координаты точки привязки *x*, *y* (т. е. точки, определяющей положение фигуры в области отображения);
- тип фигуры.

Структура такой записи может быть описана следующим образом:

```
type
  TShape = (Rectangle, Circle, Quadrate);
  TFigure = record
    x,y : double;
    z : TShape;
    case TShape of
      Rectangle : (Height,Width : double);
      Circle : (Radius : double);
      Quadrate : (Length : double);
  end;
```

Описанная таким образом структура может быть использована в программе вычисления площади фигуры (листинг 8.1).

Листинг 8.1. Программа вычисления площади фигуры `rec_var`

```
program rec_var;
type
  TShape = (Rectangle, Circle, Quadrate);
  TFigure = record
    x,y : double;
    z : TShape;
    case TShape of
      Rectangle : (Height,Width : double);
      Circle : (Radius : double);
      Quadrate : (Length : double);
  end;
var
  q : TFigure;
function S(p:TFigure):double;
begin
  with p do
    case z of
      Rectangle: S:=Height*Width;
```

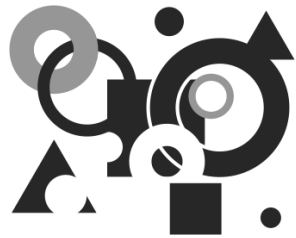
```
      Circle   : S:=pi*sqr(Radius);
      Quadrate : S:=sqr(Length);
    end;
  end;
begin
  q.z:=Rectangle;
  write('Height = '); readln(q.Height);
  write('Width = ');  readln(q.Width);
  writeln('Area = ',S(q):6:2);
  q.z:=Circle;
  write('Radius = '); readln(q.Radius);
  writeln('Area = ',S(q):6:2);
  q.z:=Quadrate;
  write('Length = '); readln(q.Length);
  writeln('Area = ',S(q):6:2);
  readln;
end.
```

Результаты работы программы `rec_var` приведены на рис. 8.1.

```
Running "f:\fpc\myprog\rec_var.exe "
Height = 5
Width = 4
Area =  20.00
Radius = 10
Area = 314.16
Length = 5
Area =  25.00
```

Рис. 8.1. Вычисление площади фигур

Обратите внимание на следующую деталь в программе `rec_var`: в качестве названия поля *Длина* квадрата использовано имя `Length`, совпадающее с названием системной функции. На самом деле имена функций не относятся к зарезервированным словам языка, и в программе их можно использовать для обозначения каких-либо объектов.



Глава 9

Подпрограммы — процедуры и функции

Подпрограммы — один из наиболее ранних приемов автоматизации программирования. Если алгоритм решения задачи содержит фрагменты, которые могут быть использованы не один раз в нескольких местах программы, то такие фрагменты можно выделить в программные единицы (процедуры или функции). Обращение к однажды написанному фрагменту программы с заданием новых входных данных (параметров программной единицы) позволяет существенно сократить общий объем программы. Хорошо отработанный фрагмент типового алгоритма может оказаться полезным и при решении других задач. Его можно автономно протранслировать и включить в состав библиотеки подпрограмм, которые по мере надобности могут вызываться вашей программой и использоваться наравне с системными функциями и процедурами. По идеологии Паскаля библиотеки подпрограмм оформляются в виде модулей — файлов с расширением tpu (от Turbo Pascal Unit).

9.1. Оформление процедур

Объявление процедуры начинается с заголовка, имеющего в общем виде следующий формат:

```
procedure name_proc[(list_arg)];[directives;]
```

Здесь:

- `procedure` — служебное слово;
- `name_proc` — имя процедуры;
- `list_arg` — необязательный список параметров;
- `directives` — одна или несколько уточняющих директив.

Директивы, уточняющие типы и способы передачи параметров, могут располагаться в разных местах заголовка — сразу после служебного слова `procedure`, перед именем параметра в списке `list_arg`, в конце заголовка.

Квадратные скобки в заголовке процедуры свидетельствуют о необязательном присутствии того или иного компонента. Список параметров содержит информа-

цию о данных, которыми вызывающая программа и процедура обмениваются между собой. Некоторые процедуры выполняют свои функции без обмена данными с вызывающей программой, и тогда список параметров может отсутствовать. Типичным примером процедуры такого сорта является системная процедура очистки экрана в консольном приложении:

```
clrscr;    {имя - аббревиатура от Clear Screen}
```

Некоторые параметры в списке аргументов представляют исходные данные, необходимые для работы процедуры. Их обычно называют *входными* данными. Другие параметры могут быть представлены переменными, в которые процедура заносит результаты своей работы. Это — так называемые *выходные* данные. Некоторые параметры могут совмещать обе функции — сначала их значения используются как входные данные, и в них же формируются результаты работы процедуры.

Список параметров принято называть списком *формальных* аргументов. Термин "формальный" подчеркивает, что за обозначением параметра не скрывается какое-либо конкретное (фактическое) значение. Значения входных параметров превращаются в *фактические* аргументы только в момент обращения к процедуре. Выходные параметры приобретают конкретные значения в процессе выполнения процедуры.

Форма представления параметров в списке определяет их *имена*, *типы* соответствующих данных и *способы обмена* между подпрограммой и вызывающей программой. В качестве разделителя между параметрами разного типа в заголовке процедуры используется *точка с запятой*. Имена двух или более параметров *одного типа* обычно объединяют в общее описание, разделяя их *запятыми* в заголовке процедуры. Например, заголовок процедуры, осуществляющей обмен данными между двумя областями оперативной памяти, может быть оформлен следующим образом:

```
procedure swap(var a,b; size:word);
```

В списке аргументов представлены три параметра с именами `a`, `b` и `size`. Два первых параметра передаются *по адресам*, определяющим начальное местоположение соответствующих областей оперативной памяти. Об этом свидетельствует директива `var` (от англ. *variable* — переменная). Тип параметров `a` и `b` не задан — в таких случаях принято говорить о *нетипизированных* параметрах. Третий параметр `size`, имеющий тип `word`, передается *по значению* (об этом свидетельствует отсутствие какой-либо директивы перед именем формального параметра). Он задает размер в байтах областей оперативной памяти, участвующих в обмене. В зависимости от типа формальных параметров и способа их передачи вызывающая программа должна использовать фактические аргументы, совместимые с соответствующим формальным параметром. В приведенном примере в качестве фактических значений, соответствующих параметрам `a` и `b`, могут выступать *имена данных любого типа*. Фактическим аргументом, соответствующим параметру `size`, может быть *любое выражение* типа `word` или совместимое с этим типом по присваиванию (например, типа `byte`).

Механизм передачи параметров унифицирован в большинстве систем программирования. Он определяет следующие моменты:

- порядок передаваемых параметров;
- признак передаваемых данных;
- физический носитель данных, в который помещаются передаваемые параметры;
- способ восстановления физического носителя.

В системах программирования на базе языка Паскаль принят *прямой* порядок передачи параметров — слева направо по списку аргументов. В системах программирования на базе языков C и C++ порядок передачи параметров *обратный* — справа налево. При составлении программ, использующих подпрограммы, на разных алгоритмических языках, с этим обстоятельством приходится считаться и принудительно устанавливать тот порядок, который удовлетворяет вызываемую процедуру. Директива `pascal` заставляет компилятор использовать прямой порядок передачи параметров, директива `cdecl` (аббревиатура от C Declaration) является указанием об обратном порядке передачи параметров. В рамках программы использующей процедуры, написанные только на языке Паскаль, прибегать к указанным директивам смысла не имеет — по умолчанию действует режим, характерный для Паскаля.

В качестве признака передаваемых данных действует одно из следующих указаний:

- параметр является *типизированным значением* (имени параметра не предшествует ни одна директива);
- параметр передается *по адресу* (директива `var`), и его значение в процедуре может быть изменено;
- параметр передается как *константа* (директива `const`), и его имя в процедуре не может находиться в левой части оператора присваивания (т. е. параметр можно использовать только как входной). Параметр с указанием `const` передается по адресу;
- параметр объявлен *выходным* (директива `out`), и его имя в процедуре может находиться только в левой части оператора присваивания. Выходной параметр передается по адресу.

В списке формальных параметров может присутствовать комбинация директив `const var`, означающая, что процедура получает адрес физического аргумента, но не имеет права менять его значение.

В системах программирования на IBM-совместимых компьютерах используются два основных физических носителя для передачи параметров — через *стек* и через *машинные регистры*.

Первый способ более распространен, т. к. он не ограничивает объем передаваемой информации. В качестве стека используется определенный участок оперативной памяти с фиксированным, но управляемым диапазоном адресов (размер стека можно регулировать при настройке компилятора). Специальный машинный регистр "следит" за очередным доступным участком стека. По адресу, хранящемуся в этом регистре, можно положить нужную порцию данных в стек и одновремен-

но продвинуть содержимое регистра стека. Для этой цели система машинных команд предусматривает специальную операцию `PUSH` (от англ. *push* — затолкнуть). Вторая машинная операция `POP` (от англ. *pop up* — выскочить наверх) позволяет извлечь из стека очередную порцию данных с одновременной коррекцией регистра стека. Системная программа, обслуживающая стек, следит за тем, чтобы стек не переполнился при записи и не оказался пустым при извлечении данных. Иногда механизм работы стека сравнивают с *магазином* огнестрельного оружия — в нем пуля, попавшая последней в рожок автомата, стреляет первой. Этим же объясняется технология обслуживания стека LIFO (Last Input — First Output, т. е. последним вошел — первым вышел). Зарядка магазина имитирует запись в стек, а процедура стрельбы напоминает извлечение данных из стека.

Запись фактических параметров в стек компилятор осуществляет перед вызовом процедуры. Извлечение входных аргументов из стека и запись результатов своей работы по адресам выходных параметров производит процедура. Однако после работы процедуры в стеке могут остаться какие-то данные, которые необходимо удалить, подготовив стек к последующим вызовам. Функцию очистки стека может выполнить сама процедура или вызывающая ее программа. Выбор исполнителя этой операции регулируется директивой по передаче параметров (табл. 9.1).

Таблица 9.1

Директива	Порядок передачи параметров	Кто чистит стек?
`pascal`	Слева направо	Вызываемая процедура
`cdecl`	Справа налево	Вызывающая программа
`stdcall`	Справа налево	Вызываемая процедура
`safecall`	Справа налево	Вызываемая процедура

В чем заключается специфика передачи параметров по адресам и по значению. Во-первых, адрес параметра занимает 4 байта, тогда как значения таких параметров как массивы, строки или записи могут занимать довольно большую память. Во-вторых, передавая процедуре адрес какого-либо объекта, вызывающая программа может получить по этому адресу новое значение. Фактическим параметром, передаваемым по адресу, может быть только имя объекта или указатель на него.

Если фактический аргумент передается как значение, то в его качестве может выступать имя объекта или любое выражение того же типа. Вычисленное значение фактического аргумента заносится в стек. В процедуре для его хранения выделяется временная память на тех же основаниях, как и для локальных переменных процедуры. Поэтому величина формального параметра-значения может быть использована и как входной параметр, и как изменяемый промежуточный результат. Но изменение такого параметра никак не скажется на соответствующем фактическом аргументе — после выхода из процедуры память, занимаемая всеми локальными переменными, освобождается.

Режим передачи параметров через регистры диктуется директивой `register`. Компилятор воспринимает это указание, как необязательное, и может установить соответствующий режим при соблюдении двух условий. Во-первых, количество передаваемых параметров должно быть не более трех, во-вторых, у компилятора имеется возможность освободить нужное количество регистров. Формирование адресов и/или значений фактических аргументов в регистрах сокращает время передачи и извлечения параметров, тем самым повышая скорость работы процедуры.

Структура описания процедуры, расположенного после ее заголовка, ничем не отличается от структуры описания головной программы. Здесь могут присутствовать все те же разделы объявлений типов, локальных переменных, меток, внутренних функций и процедур. Единственная особенность завершения тела процедуры заключается в использовании оператора `end` с последующей точкой с запятой. В листинге 9.1 приводится пример процедуры перемножения двух вещественных квадратных матриц 10-го порядка, выполненной в традициях ранних версий Паскаля.

Листинг 9.1. Перемножение квадратных матриц

```
const
  N = 10;
type
  matN = array [1..N,1..N] of double;
var
  a,b,c : matN;
procedure mat_mult(A,B:matN; var C:mat_N);
var
  i,j,k: byte;
  s: double;
begin
  for i:=1 to N do
    for j:=1 to N do
    begin
      s:=0;
      for k:=1 to N do
        s:=s+A[i,k]*B[k,j];
      C[i,j]:=s;
    end;
end;
```

Нормальным завершением работы процедуры считается выход на завершающий `end`. Досрочный возврат из процедуры реализуется с помощью оператора `exit` (выход).

9.2. Оформление функций

Функция представляет собой частный вид процедуры, результатом работы которой является *единственное значение*. Его принято называть *значением, которое возвращает функция*. Такой результат позволяет использовать функцию в качестве операнда любой формулы соответствующего типа:

```
y:=f1(x,z)*sin(f2(u))+f3(v);
```

Объявление функции начинается с заголовка, который в общем случае имеет вид:

```
function name_fun[(list_arg)]:tip; [directives];
```

По сравнению с заголовком директивы здесь присутствует новый обязательный компонент: `tip` — тип возвращаемого значения.

В качестве примера оформления функции в листинге 9.2 приводится фрагмент, реализующий вычисление скалярного произведения векторов в традициях ранних версий Паскаля.

Листинг 9.2. Вычисление скалярного произведения

```
const
  N = 10;
type
  vecN = array [1..N] of double;
var
  a,b : vecN;
  c : double
function vec_mult(A,B:vecN):double;
var
  i: byte;
  s: double;
begin
  s := 0;
  for i:=1 to N do
    s := s+A[i]*B[i];
  vec_mul := s;
end;
```

Дополнительной спецификой оформления функций является способ возврата вычисленного результата: его надо присвоить "переменной", имя которой совпадает с именем функции.

В языке Object Pascal и во всех версиях Delphi появился более удобный способ возврата значения функции: введена специальная системная переменная с именем

`Result`. Ее не надо объявлять, и тип этой "переменной" всегда совпадает с типом функции. Поэтому в предыдущем примере оператор `vec_mul:=s;` можно заменить на `Result:=s;`. На самом деле возможности переменной `Result` немного шире: она может использоваться как в левой, так и в правой части оператора присваивания. Поэтому оформление функции `vec_mult` могло выглядеть следующим образом:

```
function vec_mult(A,B:vecN):double;
var
  i: byte;
begin
  Result := 0;
  for i:=1 to N do
    Result := Result + A[i]*B[i];
end;
```

В языке Free Pascal появилась еще одна возможность возврата значения вычисляемой функции:

```
exit(s);
```

Этот формат заимствован из языка C, C++, только служебное слово `return` (возврат) здесь заменено на `exit`.

В рамках одной функции можно одновременно пользоваться и переменной с именем `Result`, и переменной с именем функции. Например

```
function qq : integer;
begin
  qq:=1;                {значение функции равно 1}
  Result:=Result*2;     {значение функции равно 2}
  Result:=Result+3;     {окончательное значение функции равно 5}
end;
```

Чтобы понять разницу между передачей параметра функции по значению и по адресу, рассмотрим две функции — `f1` и `f2`, которые возвращают удвоенное значение своего аргумента:

```
function f1(x:integer):integer;
begin
  x:=x*2;
  Result:=x;
end;
```

Аргумент функции `f1` передается по значению, поэтому изменение параметра `x` в теле функции не оказывает никакого влияния на фактический аргумент, заданный в конкретном обращении:

```
var
  y: integer=2;
  z: integer;
```

```
begin
  z:=f1(y);       {z=4, y=2}
  y:=f1(z+1);     {z=4, y=10}
```

Совсем по-другому работает функция `f2`, которая получает свой аргумент по адресу:

```
function f2(var v : integer):integer;
begin
  x:=x*2;
  Result:=x;
end;
```

Обращение `z:=f2(y);` приведет не только к изменению значения переменной `z` (`z=4`), но одновременно изменится и значение переменной `y` (`y=4`). К функции `f2` нельзя обращаться с аргументом, представленным формулой.

Очень интересную возможность представляет использование параметра без описания его типа в заголовке функции или подпрограммы. Рассмотрим пример с функцией `Equal`, которая умеет сравнивать данные любого типа (листинг 9.3). Идея ее алгоритма основана на том, что функция кроме адресов сравниваемых данных (`a1` и `a2`) получает и их длину (`size`) в байтах. Адреса полученных объектов преобразуются в адреса однобайтовых массивов достаточно большой длины, и операция сравнения производится побайтно.

Листинг 9.3. Программа сравнения данных любого типа `comp_all`

```
program comp_all;
function Equal(var a1,a2; size:word):boolean;
type
  Bytes=array [0..MaxInt div 2] of byte;
var
  N: integer;
begin
  N:=0;
  while (N<size)and(Bytes(a1)[N]=Bytes(a2)[N]) do inc(N);
  Result:=(N=size);
end;
type
  Vector = array [1..10] of integer;
  Point = record
    x,y:integer;
  end;
var
  vec1:Vector=(0,1,2,3,4,5,6,7,8,9);
  vec2:Vector=(0,1,2,3,4,5,4,3,2,1);
```

```
  N : integer=4;
  P : Point = (x:2; y:3);
begin
  writeln(Equal(vec1,vec2,SizeOf(Vector)));
  writeln(Equal(vec1,vec2,SizeOf(Integer)*N));
  writeln(Equal(vec1[3],P,8));
  readln;
end.
```

Результат работы программы `comp_all` приведен на рис. 9.1[1].

```
Running "f:\fpc\myprog\comp_all.exe "
FALSE
TRUE
TRUE
```

Рис. 9.1. Сравнение данных разного типа

В функции `Equal` довольно изобретательно реализовано вычисление возвращаемого значения. Всякий раз, когда байты, сравниваемые в цикле, совпадают, к переменной `N` добавляется единица. Если все сравниваемые байты окажутся равны, то после завершения цикла в переменной `N` должна накопиться величина `size` и результатом логической формулы (`size=N`) будет истина. Конечно, работу функции можно было бы прервать досрочно — при обнаружении первой пары несовпадающих байтов:

```
for N:=1 to size do
  if Bytes(a1)[N]<>Bytes(a2)[N]then exit(False);
exit(true);
```

9.3. Параметры подпрограмм по умолчанию

В языке Free Pascal имеется возможность объявить процедуру или функцию, у которой значения нескольких *последних аргументов* списка `list_arg` заданы со значениями по умолчанию:

```
function mid3(x1:word; x2:word=1; x3:word=2);
begin
  Result:=(x1+x2+x3) div 3;
end;
```

[1] Если при компиляции у вас возникают ошибки, полните команду **Options → Compiler** и в группе **Compiler mode** установите опцию **Object Pascal extension** on или **Delphi compatible**. (Первая опция предпочтительнее.) — *Прим. ред.*

К функции mid3 можно обращаться с одним (обязательным) параметром, с двумя и тремя:

```
mid3(6) = 3          // эквивалент обращения mid3(6,1,2)
mid3(6,3) = 3        // эквивалент обращения mid3(6,3,2)
mid3(6,3,5) = 4
```

В любом случае параметры, не указанные в обращении, заменяются соответствующими значениями по умолчанию.

9.4. Параметры подпрограмм — одномерные массивы

В данном разделе мы прокомментируем различные приемы передачи и обработки параметров, являющихся одномерными массивами. В пределах одной программы arg_array1 реализованы несколько функций, определяющих максимальный элемент в одномерном целочисленном массиве (листинг 9.4).

Листинг 9.4. Программа arg_array1

```
program arg_array1;
type
  ar5=array [1..5] of integer;
var
  a:ar5 = (2,1,3,0,5);
//-----------------------------------
function max1(b:ar5):integer;
// Массив передается по значению и загромождает стек
var  i,j:integer;
begin
  j:=Low(b);       // минимальная граница индекса
  Result:=b[j];    // пробный максимум - первый элемент
  for i:=j+1 to High(b) do
    if Result < b[i] then Result:=b[i];
end;
//-------------------------------------------
function max2(var b:ar5):integer;
// Массив передается по адресу, и его элементы можно изменить
var  i,j:integer;
begin
  j:=Low(b);       // минимальная граница индекса
  Result:=b[j];
```

```
  for i:=j+1 to High(b) do
    if Result < b[i] then Result:=b[i];
end;
//-----------------------------------------------
function max3(const b:ar5):integer;
// Массив передается по адресу, но его элементы изменить нельзя
var  i,j:integer;
begin
  j:=Low(b);       // минимальная граница индекса
  Result:=b[j];
  for i:=j+1 to High(b) do
    if Result < b[i] then Result:=b[i];
end;
//-----------------------------------------------
function max4(var b; n:integer):integer;
// Массив передается как указатель без типа.
// Использовано приведение типа.
// n - число элементов в массиве.
type
  int=array [0..(MaxInt div 4)-1] of integer;
var  i:integer;
begin
  Result:=int(b)[1];
  for i:=1 to n-1 do
    if Result<int(b)[i] then Result:=int(b)[i];
end;
//-------------------------------------------------
function max5(var b;n:integer):integer;
// Массив передается как указатель без типа.
// Использовано приведение к абсолютному адресу.
var
  a:array [0..0] of integer absolute b;
  i:integer;
begin
  {$R-}            // отключение контроля за индексами
  Result:=a[0];
  for i:=1 to n-1 do
    if Result<a[i] then Result:=a[i];
  {$R+}            // включение контроля за индексами
end;
```

```
//-------------------------------------------------
function max6(b : array of integer):integer;
// Использование открытого массива
var  i:integer;
begin
  Result:=b[0];
  for i:=1 to High(b) do
    if Result < b[i] then Result:=b[i];
end;
//-------------------------------------------------
begin
  writeln('Max1 = ',max1(a));
  writeln('Max2 = ',max2(a));
  writeln('Max3 = ',max3(a));
  writeln('Max4 = ',max4(a,5));
  writeln('Max5 = ',max5(a,5));
  writeln('Max6 = ',max6(a));
end.
```

Результаты работы программы `arg_array1` приведены на рис. 9.2.

```
Running "f:\fpc\myprog\arg_array1.exe "
Max1 = 5
Max2 = 5
Max3 = 5
Max4 = 5
Max5 = 5
Max6 = 5
```

Рис. 9.2. Максимальный элемент в одномерном массиве

Функции `max1`, `max2`, ..., `max5` реализованы в традициях ранних версий языка Паскаль.

Формальный аргумент функции `max1` описан с применением глобального типа `ar5`. Это означает, что функция `max1` может обрабатывать массивы только указанного типа. И если в рамках вашей программы приходится обрабатывать массивы других размерностей, то нужно подключать другие функции. За такой подход Паскаль справедливо подвергался нападкам. Дополнительный недостаток функции `max1` заключается в том, что в стек записываются все элементы обрабатываемого массива. А это расточительно как с точки зрения использования памяти, так и дополнительных потерь времени на передачу исходных данных.

Функция `max2` лишена двух последних недостатков. При ее использовании в стек записывается только адрес обрабатываемого массива, так что использование ресурсов по памяти и быстродействию здесь на высоте. Однако главный недостаток заключается в том, что функция `max2` может обрабатывать массивы только

фиксированной размерности. И хотя в ее теле задействованы универсальные функции `Low` и `High`, компилятор забракует все обращения, в которых тип фактического аргумента отличается от `ar5`.

Функция `max3` отличается от функции `max2` только тем, что ее аргумент может использоваться только как входной, но в рамках поставленной задачи это не ограничивает работу алгоритма.

В функции `max4` реализован более прогрессивный подход, позволяющий обрабатывать массивы любой длины. Первый ее аргумент определяет начальный адрес обрабатываемого массива, а второй — задает количество элементов в массиве. Чтобы не нарушить правила использования нетипизированного параметра `b`, его *приводят к типу с именем* `int`, представляющему целочисленный массив с огромным диапазоном по индексу. Память для массива такого размера никто выделять не собирается, т. к. фактический массив, передаваемый в качестве значения аргумента `b`, уже находится в памяти среди набора данных вызывающей программы. А максимально допустимый индекс в типе `int` не позволит нарушить границы индексов в цикле обработки элементов приведенного массива. Дополнительным преимуществом функции `max4` является возможность определения максимального элемента в любом фрагменте массива. Для этого в качестве первого фактического аргумента достаточно задать имя первого элемента фрагмента, а в качестве второго фактического аргумента — длину фрагмента.

В функции `max5`, по существу, реализован аналогичный поход. Здесь объявлен локальный массив `a` минимального размера — он содержит единственный элемент `a[0]` и его адрес совпадает с начальным адресом параметра `b` (указание `...absolute b`). Чтобы отключить контроль за индексами элементов массива `a`, который неизбежно будет нарушен при `n>0`, на время работы цикла контроль блокируется.

Самый современный подход, связанный с использованием *открытых массивов*, реализован в функции `max6`. Формальный параметр, называемый открытым массивом, описывается в заголовке подпрограммы *без указания границ индексов*. Принято, что минимальный индекс элементов открытого массива равен 0, а максимальный определяется функцией `High`. Поэтому в функции `max6` не приходится заниматься приведением типов, изменением режима контроля за индексами и прочими ухищрениями, которые были использованы в функциях `max4` и `max5`. На практике придерживаются следующего правила — для создания универсальных подпрограмм, которые могут обрабатывать массивы любой длины, всегда старайтесь использовать открытые массивы.

Если параметром подпрограммы является открытый массив, то в качестве фактического аргумента может быть задан так называемый *конструктор массива*:

```
y:=max6([5,7,x,x+z]);
```

Такое обращение эквивалентно следующим традиционным действиям:

```
var
  x,y,z:integer;
  a:array[0..3];
.................
```

```
x:=...;
z:=...;
a[0]:=5;
a[1]:=7;
a[2]:=x;
a[3]:=x+z;
y:=max6(a);
```

Однако есть и принципиальное внутреннее отличие: если фактическим аргументом является настоящий массив, то функция `max6` получает его адрес. Если же использован конструктор массива, то в стеке создается его копия, и подпрограмма получает адрес копии.

Применение открытого массива в функции `max6` предоставляет еще одну возможность, которой не может похвастаться большинство других алгоритмических языков. Пусть у нас имеется целочисленный массив `b`, объявленный с интервалом индексов от 1 до 100. Тогда допустимы следующие обращения к функции `max6`:

```
y1:=max6(b[1..50]);   // поиск максимума в первой половине массива
y2:=max6(b[51..100]); // поиск максимума во второй половине массива
```

9.5. Параметры подпрограмм — двумерные массивы

В этом разделе мы прокомментируем некоторые приемы передачи и обработки параметров, являющихся двумерными массивами.

Процедура `mat_add1` выполняет сложение двух квадратных матриц в традиционном стиле ранних версий Паскаля (листинг 9.5).

Листинг 9.5. Процедура `mat_add1`

```
const
  N=10;
type
  matN = array [1..N,1..N] of integer;
var
  a, b, c: matN
procedure mat_add1(A,B:matN; var C:matN);
// Сложение квадратных матриц C=A+B
var
  i,j:integer;
begin
  for i:=1 to N do
```

```
    for j:=1 to N do
      C[i,j]:=A[i,j]+B[i,j];
end;
...
```

Основным недостатком этой процедуры является то, что она "умеет" складывать только матрицы размером N×N при фиксированном значении N. Кроме того, два первых параметра являются значениями, и это обязывает компилятор перед вызовом процедуры mat_add1 копировать оба слагаемых в стек.

Процедура mat_add2 избавлена от этих недостатков за счет использования нетипизированных формальных параметров (листинг 9.6).

Листинг 9.6. Процедура mat_add2

```
procedure mat_add2(var A,B,C; m:integer);
// Сложение квадратных матриц C=A+B
var
  x:array [0..0] of integer absolute A;
  y:array [0..0] of integer absolute B;
  z:array [0..0] of integer absolute C;
  i,j:integer;
begin
  {$R-}
  for i:=0 to m-1 do
    for j:=0 to m-1 do
      z[i*m+j]:=x[i*m+j]+y[i*m+j];
  {$R+}
end;
```

Еще одной изюминкой в процедуре mat_add2 является использование *приведенных индексов*. Если бы внутренние массивы x, y и z были объявлены как двумерные, то смещение элемента с индексами [i, j] относительно начального элемента с индексами [0, 0] вычислялось бы по формуле i*m+j.

Следующая процедура mat_mul1 выполнена в соответствии с математическими правилами умножения квадратных матриц, однако ей присущи недостатки, отмеченные в процедуре mat_add1 — загромождение стека, возможность перемножить матрицы единственного размера (листинг 9.7). Кроме того, бездумное использование процедуры mat_mul1 может привести к ошибочному результату, и о такой ошибке никто программиста не предупредит.

Листинг 9.7. Процедура mat_mul1

```
procedure mat_mul1(A,B:matN; var C:matN);
// Умножение квадратных матриц C=A*B
```

```
var
  i,j,k:integer;
  s:integer;
begin
  for i:=1 to N do
    for j:=1 to N do
      begin
        s:=0;
        for k:=1 to N do
          s:=s+A[i,k]*B[k,j];
        C[i,j]:=s;
      end;
end;
```

Если третий аргумент не совпадает ни с первым, ни со вторым, то процедура `mat_mul1` работает идеально правильно. Но если хотя бы одно из этих двух условий будет нарушено, то элементы одного из сомножителей в процессе работы подпрограммы будут портиться, и общий результат окажется неверным.

Для того чтобы избавиться от недостатков процедуры `mat_mul1`, целесообразно воспользоваться *двумерными динамическими массивами* (листинг 9.8).

Листинг 9.8. Программа `mul_mat`

```
program mul_mat;
type
  dyn_ar2 = array of array of integer;
var
  a,b,c : dyn_ar2;
  i,j : integer;
procedure mat_mul2(A,B:dyn_ar2; var C:dyn_ar2);
var
  i,j,k:integer;
  s:integer;
begin
  for i:=0 to High(A) do
    for j:=0 to High(A) do
      begin
        s:=0;
        for k:=0 to High(A) do
          s:=s+A[i,k]*B[k,j];
        C[i,j]:=s;
      end;
end;
```

```
begin
  SetLength(a,2,2);     {запрос памяти под первый сомножитель}
  SetLength(b,2,2);     {запрос памяти под второй сомножитель}
  SetLength(c,2,2);     {запрос памяти под результат}
  a[0,0]:=1;    a[0,1]:=2;
  a[1,0]:=3;    a[1,1]:=4;
  b[0,0]:=5;    b[0,1]:=6;
  b[1,0]:=7;    b[1,1]:=8;
  mat_mul2(a,b,c);
  for i:=0 to 1 do
    begin
      for j:=0 to 1 do write(c[i,j]:4);
      writeln;
    end;
  readln;
end.
```

Результаты работы программы `mul_mat` приведены на рис. 9.3.

```
Running "f:\fpc\myprog\mul_mat.exe "
  19  22
  43  50
```

Рис. 9.3. Результат умножения матриц

Единственное, что было бы полезно добавить к процедуре `mat_mul2`, так это сравнение указателя `C` с указателями `A` и `B`. В случае хотя бы одного совпадения можно было бы выдать сообщение о недопустимом формате обращения к процедуре.

9.6. Подпрограммы с параметрами процедурного типа

Кроме формальных параметров, соответствующих тем или иным типам данных, иногда приходится передавать подпрограммам имена других процедур или функций. Параметры такого типа принято называть *процедурными*. Их фактические значения определяют адрес точки входа в соответствующую процедуру или функцию, количество, порядок и типы аргументов этой процедуры или функции.

Для объявления переменных процедурного типа используются конструкции типа заголовков процедур или функций *без указания конкретного имени* функции или подпрограммы:

```
var
  F : function (x,y:integer): integer;
```

```
  PS: procedure(const S:String);
  PV: procedure;
```

Переменной F может быть присвоено имя функции (адрес точки входа в любую функцию) от двух целочисленных аргументов, которая возвращает целочисленное значение. Пусть, например, у нас определены три функции, удовлетворяющие указанным условиям:

```
function Sum(a,b:integer):integer;
begin
  Result:=a+b;
end;

function Diff(a,b:integer):integer;
begin
  Result:=a-b;
end;

function Mult(a,b:integer):integer;
begin
  Result:=a*b;
end;
```

Если мы выполним F:=Sum;, то результатом обращения к функции F(i1,i2) будет сумма i1+i2. Если переменной F будет присвоено значение Diff, то результатом обращения к функции F(i1,i2) будет разность i1-i2. После присвоения F:=Mult; функция F(i1,i2) возвратит произведение i1*i2. Таким образом, "переменная" F позволяет заменить обращение к нескольким разным функциям, и это расширяет возможности языка программирования подобно тому, как использование обычных переменных предоставляет более широкие возможности, чем употребление констант. В частности, во время выполнения программы можно узнать, какой из функций соответствует текущее значение переменной процедурного типа:

```
if @F=@Sum then ...
```

Переменной PS может быть присвоено имя любой процедуры, единственным аргументом которой является короткая строка. А переменной PV может быть назначена любая процедура без параметров.

Особенно удобно использование параметров процедурного типа, когда приходится создавать подпрограммы, вызывающие другие подпрограммы и функции, имена которых заранее не фиксированы. Примером задачи такого рода является разработка подпрограммы-функции, вычисляющей приближенное значение определенного интеграла:

$$y = \int_a^b f(x)dx .$$

Существует довольно много методов приближенного вычисления площади криволинейной трапеции на интервале $[a, b]$, ограниченной кривой $y = f(x)$. Простейший их них, известный под названием *метода прямоугольников*, сводится к разбиению интервала интегрирования на n отрезков (обычно равной длины) и замене площади элементарной криволинейной трапеции площадью прямоугольника (рис. 9.4).

Более качественным является замена площади криволинейной трапеции площадью обычной трапеции (рис. 9.5).

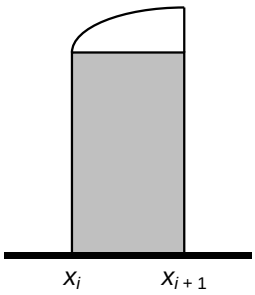


Рис. 9.4. Метод прямоугольников ($S_i \approx f(x_i) \cdot h$)

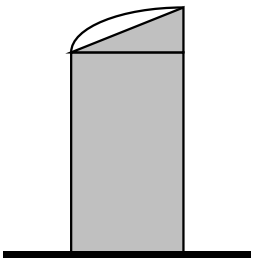


Рис. 9.5. Метод трапеций ($S_i \approx 0{,}5 \cdot (f(x_i) + f(x_{i+1})) \cdot h$)

В листинге 9.9 приводится программа вычисления определенного интеграла, в которую включены универсальные функции интегрирования методом прямоугольника (функция `prjam`) и методом трапеций (функция `trap`). В качестве подынтегральной функции выбрана функция `f(x)=x`. Границы интервала интегрирования [`a`, `b`] и количество `n` элементарных отрезков разбиения вводятся по запросу программы.

Листинг 9.9. Программа `integral`

```
program integral;
type
  fun=function(x : double):double;
var
  a,b:double;
  n:integer;
function prjam(a,b:double;n:integer;f:fun):double;
var
  h:double;
  i:integer;
begin
  Result:=f(a)+f(b);
```

```
  h:=(b-a)/n;
  for i:=1 to n-1 do
    Result:=Result+f(a+i*h);
  Result:=Result*h;
end;

function trap(a,b:double;n:integer;f:fun):double;
var
  h:double;
  i:integer;
begin
  Result:=(f(a)+f(b))/2;
  h:=(b-a)/n;
  for i:=1 to n-1 do
    Result:=Result+f(a+i*h);
  Result:=Result*h;
end;

function y(x:double):double;
begin
  Result:=x;
end;
begin
  write('a = '); readln(a);
  write('b = '); readln(b);
  write('n = '); readln(n);
  writeln(prjam(a,b,n,y));
  writeln(trap(a,b,n,y));
  readln;
end.
```

Результаты ее работы приведены на рис. 9.6 (точное значение интеграла равно 0.5).

```
Running "f:\fpc\myprog\integral.exe "
a = 0
b = 1
n = 100
 5.05000000000000E-001
 5.00000000000000E-001
```

Рис. 9.6. Вычисление интеграла

9.7. Рекурсивные подпрограммы

К рекурсивным программам относятся процедуры или функции, которые *прямо* или *косвенно* (через цепочку других подпрограмм) обращаются сами к себе. При организации такой последовательности вызовов возникают две проблемы. Во-первых, цепочка обращений не должна выполняться бесконечно долго (т. е. не должно произойти так называемое *зацикливание*). Для этого в рекурсивной подпрограмме обязательно должна присутствовать проверка условия, при выполнении которого цепочка рекурсивных обращений разрывается. В случае косвенной рекурсии (рис. 9.7) возникает проблема последовательности описания рекурсивной подпрограммы и посредника в цепочке вызовов.

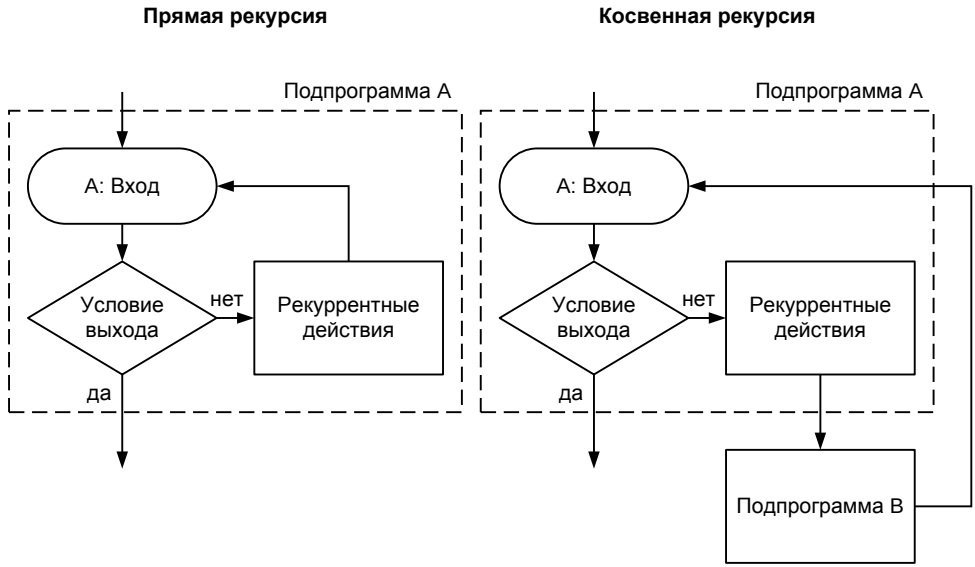


Рис. 9.7. Блок-схемы рекурсивных подпрограмм

В Паскале действует правило: описание объекта должно появиться в программе раньше момента его использования. Если подпрограмма `A` обращается к подпрограмме `B`, то описание подпрограммы `B` должно быть включено в текст программы раньше, чем описание подпрограммы `A`. Но так как в рекурсивной цепочке подпрограмма `B` обращается к подпрограмме `A`, то компилятор должен был встретить сначала описание подпрограммы `A`. Чтобы разорвать этот заколдованный круг, в языке Паскаль были разрешены так называемые *опережающие объявления*. Такие объявления представляют собой заголовки процедур или функций, снабженные директивой `forward` (идущий впереди других):

```
procedure B(list_arg); forward;
procedure A(list_arg);
```

```
  {описание процедуры A, которая вызывает процедуру B}
end;

procedure B(list_arg);
  {описание процедуры B, которая вызывает процедуру A}
end;
```

Рекурсии в программировании появились как следствие рекурсивных алгоритмов, достаточно широко применяемых в математике. Вспомните классическое определение факториала:

$$n! = n \cdot (n-1)!\,.$$

Для приближенного вычисления значения $y = \sqrt{x}$ метод Ньютона приводит к рекуррентному соотношению:

$$y_{n+1} = 0.5 \cdot \left(y_n + \frac{x}{y_n} \right).$$

Метод Зейделя для приближенного решения систем линейных алгебраических уравнений сводится к итерациям вида:

$$\begin{pmatrix} x_1 = a_{11} \cdot x_1 + a_{12} \cdot x_2 + & \dots & a_{1n} \cdot x_n - b_1 \\ x_2 = a_{21} \cdot x_1 + a_{22} \cdot x_2 + & \cdots & a_{2n} \cdot x_n - b_2 \\ \vdots & \ddots & \vdots \\ x_n = a_{n1} \cdot x_1 + a_{n2} \cdot x_2 + & \cdots & a_{nn} \cdot x_n - b_n \end{pmatrix}.$$

Использование рекурсивных процедур и функций позволяет писать достаточно компактные программы. Как правило, любую рекурсивную программу можно преобразовать в обычный цикл, который быстрее работает и менее прожорлив по использованию оперативной памяти. Однако преобразование такого рода не всегда является очевидным.

9.7.1. Вычисление наибольшего общего делителя

Исторически одним из первых рекурсивных алгоритмов является способ вычисления наибольшего общего делителя (НОД) двух целых чисел, приписываемый Евклиду. Алгоритм Евклида базируется на трех следующих фактах:

- НОД(n_1, n_2) = НОД(n_2, n_1). Этот факт сомнения не вызывает;
- НОД(0, n_2) = n_2. Так как ноль делится на любое число, то и этот факт является истинным;
- НОД(n_1, n_2) = НОД(n_2, n_3), где $n_3 = n_1 \bmod n_2$ ($n_1 > n_2$).

  Справедливость третьего факта вытекает из равенства $n_1 = k \times n_2 + n_3$ ($k \geq 1$).

  Алгоритм Евклида включает следующие шаги:

1. Раздели большее на меньшее (n_1 / n_2) и найди остаток от деления (n_3).

2. Замени НОД(n_1, n_2) на НОД(n_3, n_2).
3. Если $n_3 = 0$, то НОД = n_2. В противном случае повтори шаги 1 и 2.

Первый вариант функции `NOD`, в точности повторяющий шаги алгоритма Евклида, выглядит так, как представлено в листинге 9.10.

Листинг 9.10. Функция `NOD`

```
function NOD(n1,n2:integer):integer;
var
  tmp: integer;
begin
  if n1 > n2 then
  begin
    tmp:=n1;  n1:=n2;  n2:=tmp;
  end;
  if n1 = 0 then NOD:=n2
  else NOD:=NOD(n2 mod n1, n1);
end;
```

Так как `NOD(n1,n2)=NOD(n2,n1)`, то перестановку `n1` и `n2` можно не делать — функция `NOD` иногда лишний раз будет вхолостую обращаться сама к себе. Поэтому приведенный вариант функции `NOD` можно сократить:

```
function NOD(n1,n2:integer):integer;
begin
  if n1 = 0 then NOD:=n2
  else NOD:=NOD(n2 mod n1, n1);
end;
```

Выглядит изящнее, но работает медленнее, чем предыдущий вариант.

Не забывайте, что при каждом входе в рекурсивную функцию под ее локальные переменные и рабочие ячейки для хранения промежуточных результатов выделяются новые ячейки памяти.

9.7.2. Числа Фибоначчи

Более поздний рекурсивный алгоритм связывают с именем итальянского математика Фибоначчи (XII—XIII вв.). Он занимался оценкой потомства кроликов при следующих предположениях: все начинается с разнополой пары, ежегодно приносящей приплод в виде новой пары — самца и самки. Дети начинают пополнять популяцию по такой же схеме через два года после своего рождения. Считая, что смертность отсутствует, получаем:

- *в начале эксперимента* — **1 пара** (родители);

- *через 1 год* — **2 пары** (родители и дети 1-го поколения);
- *через 2 года* — **3 пары** (родители; дети 1-го поколения; дети 2-го поколения);
- *через 3 года* — **5 пар** (родители; дети 1-го, 2-го и 3-го поколений; потомки детей 1-го поколения).

Фибоначчи вывел формулу для количества пар: 1, 1, 1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, ... В общем виде очередное число Фибоначчи образуется как сумма двух предыдущих:

$$F_n = F_{n-2} + F_{n-1}.$$

Кроме курьезной задачи о потомстве виртуальных кроликов, числа Фибоначчи используются в задачах поиска экстремума функций.

Функция, которая вычисляет значение *n*-го числа Фибоначчи, может быть представлена следующей рекурсивной программой (листинг 9.11).

Листинг 9.11. Функция вычисления числа Фибоначчи

```
function Fib(n:integer):longint;
begin
  if n<3 then Fib:=1
  else Fib:=Fib(n-1)+Fib(n-2);
end;
```

Эта функция каждый раз при входе в нее дважды обращается сама к себе. И на ее примере можно оценить степень неэффективности алгоритма. В табл. 9.2 первая строка указывает число сложений, затрачиваемых для данного *n* по обычной схеме вычислений, а вторая строка — число сложений по рекуррентной схеме.

Таблица 9.2

n	0	1	2	3	4	5	6	7	8	9	10
Обычные сложения	0	0	1	2	3	4	5	6	7	8	9
Сложения с рекурсией	0	0	1	2	4	7	12	20	33	54	88

9.7.3. Вычисление факториала

В большинстве книг по программированию в качестве примера рекурсивной функции демонстрируется программа вычисления факториала, текст которой выглядит достаточно компактно:

```
function fact(n:integer):extended;
begin
  if n=0 then Result:=1
  else Result:=n*fact(n-1);
end;
```

Включим эту функцию в следующую головную программу (листинг 9.12).

Листинг 9.12. Программа вычисления факториала

```
program factorial;
{$R-}
var
  n: integer;
  Res: tip;
function fact(n:integer):extended;
...
begin
  repeat
    readln(n);
    Res:=round(fact(n));
    writeln(n,'! = ',Res);
  until EOF;
end.
```

И попробуем менять тип возвращаемого значения, чтобы определить границу применимости нашей функции для типов `integer`, `comp`, `double`, `extended`. Оказывается, что при `tip=integer` наша функция правильно считает до `n=12` и выдает без сообщения об ошибке следующее:

```
12! = 470001600
```

Но при отключенном контроле выдает неверное значение `13!=1932053504` (вместо `6227020800`). А для `n=17` результат вообще отрицательный (`-288522240`).

При `tip=comp` удается добраться до `20!`, на `21!` фиксируется переполнение. При `tip=double` последний правильный результат выдается для `170!`. Наконец, при `tip=extended` удается вычислить `1754!`. При больших значениях `n` для каждого из приведенных типов возвращаемого значения фиксируется аварийный останов.

Функция вычисления факториала довольно просто заменяется обычным циклом, который работает существенно быстрее и не требует дополнительных затрат по памяти (листинг 9.13).

Листинг 9.13. Функция вычисления факториала без рекурсии

```
function fact1(n:integer):extended;
var
  i: integer;
begin
  Result:=1;
  for i:=2 to n do
    Result:=Result*i;
end;
```

9.7.4. Быстрая сортировка

В 1962 г. известный математик Хоар (C. A. R. Hoare) опубликовал алгоритм сортировки, за которым закрепилось название `quicksort`. Идея этого алгоритма удивительно проста. Сначала выбирается "средний" элемент в сортируемом массиве. Все, что больше этого элемента, переносится в правую часть массива, а все, что меньше, — в левую. После первого шага "средний" элемент оказывается на своем месте. Затем аналогичная процедура повторяется для каждой половины массива. На каждом последующем шаге размер обрабатываемого фрагмента массива уменьшается вдвое. Количество операций, которое требуется, в среднем, для реализации этой процедуры, оценивается константой $n \times \log_2 n$. Программа быстрой сортировки оформлена из процедуры `quick`, к которой обращается вызывающая программа, и рекурсивной процедуры `qs` (листинг 9.14).

Листинг 9.14. Процедура быстрой сортировки массива `qs`

```
procedure qs(var x : array of integer; left, right : integer);
var
  i, j, xx, yy : integer;
begin
  i := left;
  j := right;
  xx:= x[(left + right) div 2];
  repeat
    while (x[i] < xx) and (i < right) do inc(i);
    while (xx < x[j]) and (j > left)  do dec(j);
    if i <= j then
      begin
        yy:=x[i]; x[i]:=x[j]; x[j]:=yy;
        inc(i); dec(j);
      end;
  until i > j;
  if left < j  then qs(x, left, j);
  if i < right then qs(x, i, right);
end;
//-------------------------------------------------------
procedure quick(var x: array of integer; n : integer);
begin
  qs(x,0,n-1);
end;
```

Слабым местом в приведенной реализации является выбор "среднего" элемента. Здесь в качестве границы разбиения выбирается элемент, расположенный в се-

редине массива. В самом худшем случае, когда исходный массив уже отсортирован, такой вариант алгоритма `quicksort` затратит порядка $O(n^2)$ операций. Существует довольно много модификаций алгоритма быстрой сортировки, когда граница разбиения выбирается по-другому. В частности выбор индекса "среднего" элемента организуют случайным образом (`random(n)`) или как среднее значение между первым, средним и последним элементами массива.

9.7.5. Ханойские башни

Одной из самых интересных рекурсивных программ является компьютерная модель игры "Ханойские башни". Придумал эту игру французский математик Э. Люка. На деревянной подставке были установлены три иглы, на первую из которых было насажено несколько дисков разного диаметра (рис. 9.8).

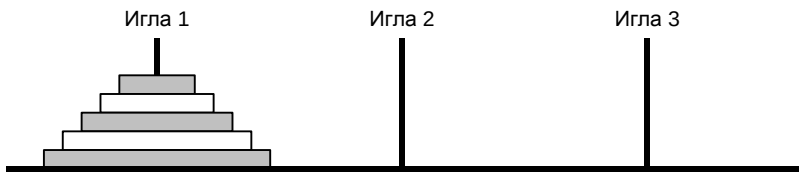


Рис. 9.8. Игра "Ханойские башни"

Цель игры заключалась в переносе пирамиды дисков с первой иглы на третью. По правилам игры за один ход можно перенести один диск, перекладывая его на ту или иную иглу. Переносимый диск может ложиться только на диск большего диаметра.

Рекламная кампания, имевшая целью повышение уровня продаж, сообщала: "Игра является моделью пирамиды браминов, установленной в храме индийского города Бенарес. Настоящая пирамида состоит из 64 золотых дисков, насаженных на один из трех алмазных стержней. Брамины круглосуточно перекладывают диски, чтобы переместить пирамиду с первого стержня на третий в соответствии с указанными выше правилами. За 1 секунду перемещается один диск. Когда пирамида будет полностью перемещена, наступит конец света". История умалчивает о том, как индийский храм перекочевал во Вьетнам.

Можно показать, что "конец света" должен был наступить через $2^{64}-1$ секунд после начала миссии браминов. Доказательство этого факта выполняется по индукции. На малом количестве дисков (n = 1, 2, 3) мы убеждаемся, что операция по переносу выполнима за 1, 3 и 7 шагов. Предполагая, что оценка верна для $(n-1)$-го диска, т. е. требует $(2^{n-1}-1)$ шагов, покажем, что она справедлива и для n дисков. Проведем эту операцию в три этапа:

- перенесем верхние $n-1$ дисков со стержня 1 на стержень 2, затратив $2^{n-1}-1$ шагов;

- перенесем самый большой диск со стержня 1 на стержень 3 за 1 шаг;
- перенесем $n-1$ дисков со стержня 2 на стержень 3, затратив $2^{n-1}-1$ шагов;

Общее число шагов, которое нам потребовалось, равно

$$\left(2^{n-1}-1\right)+1+\left(2^{n-1}-1\right)=2^n-1.$$

Обозначим через `MT` (от англ. *move tower*) исходную задачу — `MT(n,1,3)`. Очевидно, что она может быть сведена к решению трех подзадач меньшей размерности:

- `MT(n-1,1,2)` — перенос верхних $n-1$ дисков со стержня 1 на стержень 2;
- `MD(1,3)` — перенос оставшегося диска со стержня 1 на стержень 3;
- `MT(n-1,2,3)` — перенос $n-1$ диска со стержня 2 на стержень 3.

Эти соображения наводят на мысль, что для моделирования игры было бы полезно написать рекурсивную процедуру переноса башни `MT(n, from_b, to_b, work_b)`, где:

- `n` — количество перемещаемых дисков;
- `from_b` — номер стержня, с которого диски перемещаются;
- `to_b` — номер стержня, на который перемещаются диски;
- `work_b` — номер стержня, используемого в качестве рабочего.

Очевидно, что процедура `MT(n,i,j,k)` сводится к выполнению трех следующих процедур:

```
MT(n-1,i,k,j);
MD(i,j);
MT(n-1,k,j,i);
```

При `n ≤ 1` выполнение этой цепочки процедур должно быть завершено. В результате получается очень изящная программа, моделирующая работу браминов (листинг 9.15).

Листинг 9.15. Программа `Hanoj`

```
program Hanoj;
var  n:integer;
procedure MD(from_b,to_b:integer);
begin
  write(from_b,'->',to_b,'    ');
end;
procedure MT(n, from_b, to_b, work_b : integer);
begin
  if n>0 then
  begin
    MT(n-1,from_b, work_b, to_b);
    MD(from_b, to_b);
    MT(n-1, work_b, to_b, from_b);
```

```
  end;
end;
begin
  write('n = '); readln(n);
  MT(n,1,2,3);
  readln;
end.
```

Результат ее работы по переносу 5 колец отображен на рис. 9.9.

```
Running "f:\fpc\myprog\hanoj.exe "
n = 5
1->2    1->3    2->3    1->2    3->1    3->2    1->2    1->3    2->3    2->1
3->1    2->3    1->2    1->3    2->3    1->2    3->1    3->2    1->2    3->1
2->3    2->1    3->1    3->2    1->2    1->3    2->3    1->2    3->1    3->2
1->2
```

Рис. 9.9. Перенос "Ханойской башни"

9.8. Расширенный вызов функций

Подобно языку C++ в программах на языке Free Pascal допускается вызов функций с игнорированием возвращаемого значения. Такой прием имеет смысл, когда функция помимо вычисления возвращаемого значения производит какие-то полезные дополнительные действия. В листинге 9.16 таким дополнительным действием в функции `f_ex(y)` является изменение аргумента `y`, если его значение отрицательно.

Листинг 9.16. Программа `fun_ext`

```
program fun_ext;
{$X+}          {включение режима расширенного синтаксиса}
function f_ex(var x : integer):integer;
begin
  Result:=-1;
  if x<0 then x:=0
     else Result:=x+10;
end;
var
  y : integer=1;
begin
  writeln('y = ',y:3,' f_ex(y) = ',f_ex(y));
  y:=2*f_ex(y)-100;
```

```
  writeln('y = ',y:3);
  f_ex(y);      {расширенный вызов}
  writeln('y = ',y:3,' f_ex(y) = ',f_ex(y));
  readln;
end.
```

Результаты работы программы `fun_ext` приведены на рис. 9.10.

```
Running "f:\fpc\myprog\fun_ext.exe "
y =   1 f_ex(y) = 11
y = -78
y =   0 f_ex(y) = 10
```

Рис. 9.10. Расширенный вызов функции

9.9. Переопределение функций

Одной из принципиально новых особенностей, характерных для объектно-ориентированного подхода, является возможность объявления в программе нескольких функций или процедур с одинаковыми именами. К ним предъявляется единственное требование — списки их формальных параметров должны отличаться друг от друга. Это отличие может касаться как количества параметров, так и их типов. По этим критериям компилятор сможет, проанализировав каждое обращение, выбрать требуемую подпрограмму.

Представим себе, что в библиотеке `RTL` отсутствует процедура `inc`, и нам предложили написать свой вариант. Так как эта процедура должна обслуживать довольно большое количество целочисленных типов, то очевидно, что единственной процедурой обойтись нельзя. Общая канва для заголовка процедуры просматривается сразу:

```
procedure MyInc(var v:тип; dec:LongWord=1);
```

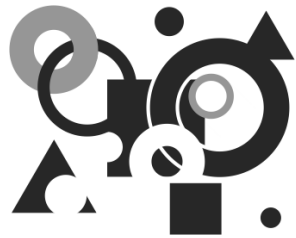
Но в теле процедуры в зависимости от типа переменной `v` мы должны организовать разные проверки на выход результата за допустимые пределы. Кроме того, придется учитывать установленный режим контроля (`{$R+}` или `{$R-}`) и корректировать результат в случае нарушения границ. В результате у нас появится несколько процедур, две из которых мы приводим в листинге 9.17.

Листинг 9.17. Процедуры `MyInc`

```
procedure MyInc(var v:byte; dec:LongInt=1);
var
  tmp:LongInt;
```

```
begin
  tmp:=v+dec;
  if (tmp >= 0) and (tmp < 256) then v:=tmp;
  if tmp < 0 then v:=(tmp mod 256)+256;
  if tmp > 255 then v:=tmp mod 256;
end;
procedure MyInc(var v:ShortInt; dec:LongInt=1);
var
  tmp:LongInt;
begin
  tmp:=v+dec;
  if (tmp >= -128) and (tmp < 128) then v:=tmp;
  if tmp < -128 then v:=((tmp mod 128)) mod 128;
  if tmp > 127 then v:=(tmp mod 128);
end;
```

Глава 10



Работа с файлами

В традиционных службах, связанных с документооборотом, термином *"файл"* (от англ. *file*) обычно обозначают папку, в которой хранятся документы определенного формата. Например, входящие или исходящие письма, анкеты сотрудников организации, ведомости успеваемости студентов и т. п. Именно из этих сфер термин файл и связанные с ним способы хранения и обработки документов перекочевали в информатику.

В языках программирования термином файл пользуются как для обозначения данных, организованных специальным образом, так и для определения способа обмена между именованными участками оперативной памяти и некоторым внешним абонентом. В качестве абонентов могут выступать диски (гибкие, жесткие, оптические, флэш-память), устройства отображения (принтеры, плоттеры, дисплеи), клавиатура, каналы связи. Некоторые системы программирования поддерживают обмен с файлами, расположенными в оперативной памяти. В зависимости от физических свойств абонента и способа организации данных обмен может быть направленными только в одну из сторон (например, только *чтение* с клавиатуры или с диска типа CD-ROM, только *запись* на принтер) или вестись по обоим направлениям (например, обмен с внешними запоминающими устройствами, допускающими как чтение, так и запись).

Неделимыми порциями информации, из которых формируется содержимое файлов, являются *записи* — аналоги документов, организованных по тому или иному шаблону.

Наиболее простую структуру имеют файлы с записями *фиксированной* длины. В таком наборе данных по номеру записи легко вычислить адрес ее расположения на внешнем носителе и организовать так называемый *прямой* обмен. При этом программа имеет возможность обмениваться записями в произвольном порядке, что увеличивает скорость обмена и повышает гибкость доступа к внешним данным. Файлы с записями фиксированной длины обычно называют *двоичными файлами*.

Более сложной является организация файлов с записями *переменной длины*. Для таких файлов длину записи чаще всего помещают в самое начало записи или используют специальный код — *признак конца записи*. В связи с большими затратами на поиск записи с заданным номером здесь используют так называемый *последовательный доступ* — обработка записей ведется друг за другом, в порядке их

расположения в файле. Типичным представителем файлов с записями переменной длины, оканчивающихся признаком конца записи, являются *текстовые файлы*. В текстовых файлах каждая запись представлена строкой (цепочкой последовательных символов), которая обычно завершается парой управляющих символов <Возврат каретки> (англ. <CR> — Carriage Return, символ с кодом 0D) и <Перевод строки> (англ. <LF> — Line Feed, символ с кодом 0A).

10.1. Файлы в стиле Turbo Pascal

Для канонической версии языка Паскаль характерно использование файлов трех типов — *текстовых*, *типизированных* и *не типизированных*. Два последних типа большинство систем программирования рассматривают как двоичные.

Для описания файла, используемого в программе, необходимо:

- объявить переменную файлового типа;
- связать эту переменную с именованной областью данных на внешнем носителе;
- подготовить файл к работе (по терминологии программистов — открыть файл).

Форма объявления переменной файлового типа зависит от типа используемого файла:

```
name_ft : TEXT;                    {для текстовых файлов}
name_ft : FILE;                    {для не типизированных файлов}
name_ft : FILE of tip;             {для типизированных файлов}
```

Далее приводится несколько примеров объявления файловых типов и переменных файлового типа:

```
type
  book = record                    {запись типа книга}
    title : string[20];            {поле заголовка}
    author: string[40];            {поле автора}
    publishinghouse: string[15];   {поле издательства}
    year : integer;                {год издания}
    price : double;                {цена}
  end;
text60 = file of string [60];
var
  f1 : file of char;  {для типизированного файла}
  f2 : text;          {для текстового файла}
  f3 : file;          {для не типизированного файла}
  f4 : text60;        {для типизированного файла}
  f5 : file of book;  {для типизированного файла}
```

Файл, с которым предполагается связать переменную файлового типа `f1`, состоит из записей, каждая из которых имеет длину 1 байт и содержит символ в ко-

дировке ASCII. Переменная `f4` предназначена для связи с файлом, каждая запись которого содержит 60 символов в кодировке ASCII. Типизированный файл, с которым предстоит связать переменную `f5`, должен содержать записи длиной по 87 байт, представляющих структуру данных о книге. Текстовый файл, для которого предназначена файловая переменная `f2`, состоит из строк переменной длины, завершаемых стандартным признаком <EOL> (аббревиатура от End-Of-Line, т. е. пара байт с кодами 0D и 0A). Нетипизированный файл, связываемый с переменной `f3`, по умолчанию состоит из блоков по 128 байт, содержимое которых ничем не регламентируется (среди них могут находиться как отображаемые символы, так и управляющие коды, которые рассматриваются как обычные байты с данными).

Привязка переменной `vf` файлового типа к именованной области внешнего запоминающего устройства (т. е. к физическому имени файла) или к логическому устройству операционной системы осуществляется с помощью системной процедуры `Assign`:

```
Assign(vf, 'name');
```

Здесь `name` — имя дискового файла или логического устройства (`PRN`, `COM1`, `LPT1`, ...).

В состав системных логических устройств входит "пустое" устройство с именем `NUL`. Во время отладки программы в это устройство можно "писать", не заботясь ни о каких ограничениях по длине. При выборке данных это устройство сразу формирует логический признак <EOF> (аббревиатура от End-Of-File), который является сигналом об исчерпании данных.

Следует упомянуть еще о двух системных устройствах с именами `INPUT` и `OUTPUT`, символизирующих стандартные средства ввода и вывода. Эти устройства не надо приводить в состояние готовности (это делает система), но зато входной и выходной потоки можно переназначать, осуществляя ввод не с клавиатуры, а из указанного текстового файла, или заменяя вывод на дисплей записью в заданный дисковый файл. Если вы хотите запустить программу prog.exe с переназначениями такого рода, то в командной строке надо набрать:

```
>prog <1.txt >2.txt
```

Операторы `read` и `readln` вместо клавиатуры будут считывать данные из заранее подготовленного текстового файла с именем 1.txt, а операторы `write` и `writeln` будут выводить результаты в текстовый файл с именем 2.txt. Такое переназначение бывает особенно удобно в процессе отладки программ с большими объемами исходных данных и результатов обработки.

Для окончательного приведения файла в состояние готовности к обмену файл надо "открыть":

- процедура `Reset (vf)` открывает для чтения файл, связанный с переменной `vf`;
- процедура `Rewrite (vf)` открывает для записи файл, связанный с переменной `vf`;
- процедуры `Reset(vf,len)` и `Rewrite(vf,len)` открывают для обмена нетипизированный файл, связанный с переменной `vf`, и заменяют стандартную длину записи (128 байт) на длину `len`;

- процедура `Append(vf)` открывает для пополнения текстовый файл, связанный с переменной `vf`.

Файл, открываемый для чтения, должен существовать. В противном случае процедура `RESET` завершается аварийно и, если контроль ошибок ввода/вывода не отключен (директива `{$I+}`), то программа принудительно завершает свою работу. Если файл, открываемый для записи, ранее существовал, то его прежнее содержимое будет потеряно.

Во всех последующих процедурах обмена данными физическое имя файла заменяет связанная с ним переменная файлового типа. После завершения работы с файлом правила хорошего тона рекомендуют его "закрывать". Для этой цели предназначена процедура `close`:

```
close(vf);
```

После закрытия файла имя переменной `vf` может быть связано с другим файлом, но до этого продолжает сохраняться связь `vf` с ранее назначенным файлом.

10.1.1. Процедуры и функции общего характера

Перечень системных функций и процедур общего характера приведен в табл. 10.1. Процедуры, аргументы которых включают имя переменной `vf` файлового типа, могут применяться к файлам любого типа.

Таблица 10.1

Формат обращения	Выполняемое действие
`Chdir('path')`	Смена текущего каталога
`Eof(vf)`	Опрос ситуации "исчерпание данных"
`Erase(vf)`	Удаление файла
`Flush(vf)`	Выталкивание содержимого буфера обмена в файл
`Getdir(n_d, str)`	Опрос имени текущего каталога на указанном диске
`Ioresult`	Опрос признака завершения файловой операции
`Mkdir('path')`	Создание нового каталога
`Rename(vf,'new_name')`	Переименование файла
`Rmdir('path')`	Удаление пустого каталога

Полный идентификатор дискового файла (полная *спецификация файла*), необходимый для определения его места на диске, состоит из двух компонентов — *пути*, содержащего иерархию вложенных каталогов, и собственно *имени файла*. Например:

```
f:\FPC\MyProg\comp_all.exe
```

Путь начинается с имени логического диска (в примере — `f:`). За логическим диском следует каталог 1-го уровня, вложенный в корневое оглавление (в примере — `FPC`). Далее следует каталог 2-го уровня, вложенный в предыдущий каталог (в примере — `MyProg`). Последней компонентой полной спецификации является имя файла (в примере — `comp_all`) и его расширение (в примере — `exe`). Термин *"текущий каталог"* по отношению к выполняющейся программе относится к каталогу, из которого эта программа была запущена. Если файл данных, к которому обращается программа, находится в текущем каталоге, то при его назначении в процедуре `Assign` можно ограничиться укороченным именем — собственно именем файла и его расширением. В противном случае мы должны указывать полную спецификацию файла. На каждом логическом диске в текущий момент времени один из его каталогов является текущим. Для опроса этой информации предназначена процедура `GetDir`. Ее первый аргумент — число из диапазона [0, 26] определяет номер опрашиваемого диска (0 — текущий диск, 1 — диск A:, 2 — диск B:, 3 — диск C:, ...). Наименование текущего каталога на опрашиваемом диске заносится в строковую переменную `str`.

По соображениям удобства бывает полезно изменить имя текущего каталога на имя другого каталога, в котором находятся обрабатываемые файлы данных. Для этой цели используется процедура `Chdir` (от англ. *change directory* — изменить каталог). После этой операции мы можем пользоваться укороченными именами файлов, а не их полными спецификациями.

Программа имеет возможность создать новый каталог для хранения своих данных (процедура `Mkdir` — от англ. *make directory*) или уничтожить каталог, в котором она создавала временные файлы. Перед удалением такого каталога все временные файлы должны быть уничтожены с помощью процедуры `Erase`. Удаляемый файл предварительно должен быть закрыт. Процедура `Rmdir` (от англ. *remove directory* — удалить каталог) может удалить только пустой каталог.

Довольно часто используемой функцией является логическая функция `EOF`, возвращающая значение `True`, если данные в файле уже исчерпаны. Возвращаемое значение `False` означает, что данные из файла еще можно читать.

Еще одна полезная функция, которую программисты зачастую забывают использовать, связана с информацией, которая заносится в системную переменную `IOResult` после выполнения каждой файловой операции. Если такая операция прошла успешно, то в переменную `IOResult` заносится 0. Ненулевое значение этой переменной свидетельствует об ошибке. Если системный контроль за операциями ввода/вывода был включен (директива `{$I+}`), то ошибочная ситуация приводит к аварийному завершению программы. Но в некоторых случаях программа может предпринять определенные шаги по продолжению своей работы. Например, пользователь в ответ на запрос программы указал неверное имя файла, и при его открытии возникла ошибочная ситуация. Вместо аварийного завершения программа может еще раз запросить у пользователя нужные данные.

Если контроль за операциями ввода/вывода отключен (директива `{$I-}`), то при возникновении ошибочной ситуации работа программы не прекращается. Однако

все последующие файловые операции будут заблокированы до тех пор, пока программа не прочитает содержимое IOResult. После такого чтения система вновь занесет нулевое значение в IOResult и дальнейшие обмены станут возможными.

10.1.2. Работа с текстовыми файлами

Обмен с текстовыми файлами очень похож на обмен с консолью (клавиатура + + дисплей) и выполняется с помощью тех же процедур read, readln, write, writeln. Единственная особенность заключается в том, что список ввода/вывода начинается с имени файловой переменной, ассоциированной с именем текстового файла:

```
var
  fin: text;
  fout: text;
begin
  ...
  assign(fin,'1.txt');
  assign(fout,'2.txt');
  reset(fin);
  rewrite(fout);
  ...
  read(fin,v1,v2,...);
  ...
  writeln(fout,e1,e2,...);
```

Список ввода может включать имена символьных, строковых и числовых переменных. Однако некоторые различия между вводом с консоли и чтением из файла все-таки существуют. Во-первых, они проявляют себя при чтении строк текстового файла в переменные типа char. Предположим, что мы включили в текстовый файл с именем 1.txt две следующие строки:

```
123456789
abcdefghi
```

Такими мы их видим в поле текстового редактора. Но невидимые управляющие символы в текстовом файле тоже присутствуют. И мы постараемся их "проявить" с помощью программы из листинга 10.1.

Листинг 10.1. Программа txt_in1

```
program txt_in1;
var
  fin: text;
  ch: array [1..22] of char;
  i: integer;
```

```
begin
  assign(fin,'1.txt');
  reset(fin);
  for i:=1 to 22 do
    begin
      read(fin,ch[i]);
      write(ord(ch[i]):4);
    end;
  readln;
end.
```

Результат ее работы, приведенный далее, придется расшифровать.

```
Running "c:\fpc\myprog\txt_in1.exe "
  49  50  51  52  53  54  55  56  57  13  10  97  98  99 100 101 102
 103 104 105  26  26
```

Почему мы попытались в цикле прочитать не 18 символов, которые были видны на поле редактора? Мы же помним, что каждая строка в текстовом файле завершается признаком конца — парой кодов 0D и 0A. Поэтому девять видимых и два невидимых, по нашему предположению, должны присутствовать в обеих строках. Вот мы и читаем в цикле 22 символа. Так как некоторые из считанных символов относятся к группе управляющих кодов, для которых видимое изображение может отсутствовать, мы решили выдать не сами символы, а их ASCII-коды. Первые 9 кодов в строке результатов соответствуют цифрам от 1 до 9. Следующий код 13 — управляющий символ CR (его шестнадцатеричный эквивалент 0D). Управляющий код 10 соответствует символу LF (его шестнадцатеричный эквивалент 0A). Далее располагаются ASCII-коды букв от `a` до `i`. А вот завершающей пары CR и LF, которую мы ожидали, нет. Вместо этого в результате присутствуют два управляющих кода, которые символизируют признак конца файла (EOF). На самом деле путем просмотра содержимого файла в шестнадцатеричном формате мы убеждаемся в том, что в первой строке признак EOL присутствует, а после данных второй строки в файле ничего нет. Откуда появились байты с кодом 26? Оказывается, что операционная система (точнее, ее подсистема управления файлами) знает фактическую длину файла и после исчерпания данных генерирует байт с кодом 26. Именно на это значение среагировала бы функция `EOF(fin)`. Этот пример лишний раз напоминает, что при чтении данных из файла мы должны быть уверены, что в файле еще что-то есть.

В переменные строкового типа данные из текстового файла рекомендуется считывать с помощью оператора `readln` и использовать в списке ввода единственную переменную. Байт длины в считываемой строке формируется автоматически (в файле его нет), а управляющие коды CR и LF в состав считываемых данных не включаются. Если длина считываемых данных превышает максимальную длину строковой переменной, то лишние хвостовые байты отбрасываются.

Если в списке оператора `read` встречается имя числовой переменной любого типа, то во входном потоке (т. е. в строке, считанной из файла и уже частично обработанной) пропускаются все пробелы, символы Tab и EOL до первой значащей числовой позиции.

Если оператор `readln` сформировал значения всех переменных своего списка, а в строке, считанной из файла, данные еще есть, то следующий оператор `readln` будет читать свои данные из новой строки.

В операторах вывода в текстовый файл `write` и `writeln` допускается использование форматных указателей, определяющих длину отводимого поля и количество цифр в дробной части числа:

```
write(vf,e1[:w1[:d1]],e2...)
```

Если указание о ширине `w` опущено, то по умолчанию устанавливается `w=23`. Если задаваемая ширина слишком мала (`w<10`), то принудительно устанавливается `w=10`. Если заданное количество значащих цифр равно нулю, то ни дробная часть, ни точка (разделитель целой и дробной части) не выводятся. При `d>18` принудительно устанавливается `d=18`.

Оператор `writeln` отличается от оператора `write` тем, что завершает запись строки в файл и формирует признак конца строки.

При работе с текстовыми файлами могут оказаться полезными логические функции, приведенные в табл. 10.2.

Таблица 10.2

Формат обращения	Выполняемое действие
`b:=EOln(vf);`	Выдает `True`, если при чтении обнаружен признак конца строки
`b:=seekeoln(vf);`	Пропускает все пробелы и символы Tab до первого значащего символа или до EOL. Возвращает `True` при обнаружении EOL
`b:=seekeof(vf);`	Пропускает все пробелы, символы Tab и EOL до тех пор, пока не встретит первый значащий символ или EOF. Возвращает `True` при достижении EOF

Довольно часто можно услышать совет: "Чтобы избежать ошибок при работе с файлами, данные из файла надо считывать таким же образом, как они и записывались". К сожалению, при работе с текстовыми файлами этот совет не всегда приводит к правильным результатам. Об этом свидетельствует программа из листинга 10.2.

Листинг 10.2. Программа `txt_inout`

```
program txt_inout;
var
  j,k1,k2: integer;
  f: text;
```

```
  a: string='Строка';
  b: string;
begin
  assign(f,'3.txt');
  rewrite(f);
  for j:=1 to 10 do
    begin
      writeln(a,j:4,j*2:4);
      writeln(f,a,j:4,j*2:4);
    end;
  close(f);
  writeln;
  reset(f);
  for j:=1 to 10 do
    begin
      readln(f,b,k1,k2);
      writeln(b,k1:4,k2:4);
    end;
  close(f);
  readln;
end.
```

После запуска программы `txt_inout` в файл 3.txt записываются 10 строк:

```
Строка   1   2
Строка   2   4
...
Строка  10  20
```

В этом можно убедиться, просматривая содержимое файла после работы программы. Но до конца программа не дорабатывает, т. к. на 21 строке при первом же считывании из файла фиксируется ошибка с выдачей сообщения:

```
Error 106 : Invalid numeric format
```

Все дело в том, что первой переменной в списке ввода является переменная `b` типа `String`, и ее максимальная длина равна 255 байтам. Поэтому из файла в переменную `b` считывается первая строка полностью, а при чтении числового значения в переменную `k1` обнаруживается недопустимый символ, расположенный в начале второй строки. Ситуацию можно исправить, ограничив максимальную длину переменной `b`:

```
b:String[6];
```

После такого исправления результат работы программы выглядит так, как и положено:

```
Running "c:\fpc\myprog\txt_inout.exe "
Строка   1   2
```

```
Строка   2   4
Строка   3   6
Строка   4   8
Строка   5  10
Строка   6  12
Строка   7  14
Строка   8  16
Строка   9  18
Строка  10  20

Строка   1   2
Строка   2   4
Строка   3   6
Строка   4   8
Строка   5  10
Строка   6  12
Строка   7  14
Строка   8  16
Строка   9  18
Строка  10  20
```

В программе indent.pas вы можете познакомиться с новыми приемами программирования (листинг 10.3). Во-первых, здесь представлена процедура, которая в качестве своих аргументов получает адреса переменных файлового типа (другим способом файловые переменные передавать нельзя). Во-вторых, запуск программы сопровождается заданием ее аргументов в командной строке. Наконец, здесь продемонстрирована возможность не совсем обычного включения цепочки пробелов в состав выводимой строки. Цель программы заключается в сдвиге содержимого всех строк текстового файла вправо на заданное число пробелов. Операции такого рода по сдвигу выделенных блоков вправо (indent) или влево (unindent) характерны для многих текстовых редакторов.

Листинг 10.3. Программа `indent`

```
program indent;
var
  f1,f2: text;
  name1,name2: string;
  k,n: integer;
procedure ind_copy(var f1,f2:text;n:integer);
var
  str:string;
begin
```

```
  while not eof(f1) do
    begin
      readln(f1,str);
      writeln(f2,' ':n,str);
    end;
end;
begin
  if ParamCount < 3 then begin
     writeln('Ошибка при запуске. Должно быть:');
     writeln('indent файл1 файл2 сдвиг');
     exit;
  end;
  name1:=ParamStr(1);
  name2:=ParamStr(2);
  Val(ParamStr(3),n,k);
  assign(f1,name1);    reset(f1);
  assign(f2,name2);    rewrite(f2);
  ind_copy(f1,f2,n);
  close(f1);    close(f2);
end.
```

После запуска программы из командной строки:

```
>txt_inout 3.txt 4.txt 5
```

и просмотра содержимого файлов 3.txt и 4.txt мы наблюдаем картину, представленную на рис. 10.1.

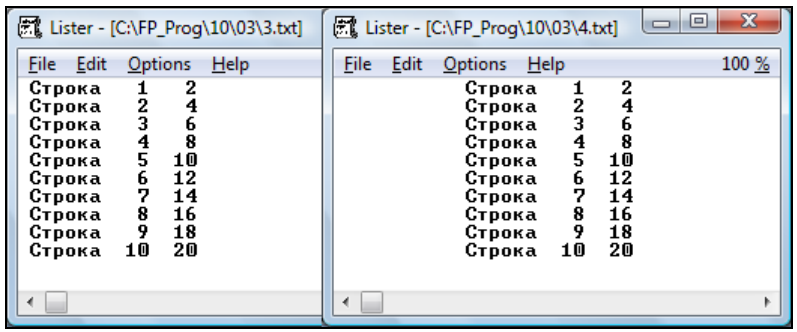


Рис. 10.1. Сдвиг содержимого в текстовом файле

Несмотря на то, что головная программа не является процедурой в общепринятом смысле слова, ее запуск осуществляет операционная система и при этом может передать ей параметры командной строки. Параметры командной строки, набираемые вслед за именем запускаемой программы, разделяются, по крайней мере, хотя

бы одним пробелом. Количество параметров можно извлечь из системной переменной `ParamCount`. Значения параметров представлены в символьном виде массивом строк, откуда их можно извлекать с помощью функции `ParamStr`. Так как в нашем случае последний параметр представляет число (количество пробелов, вставляемое в начале каждой строки), то его приходится преобразовывать в числовой машинный формат с помощью процедуры `Val`. Обратите внимание на конструкцию вида `' ':n`, использованную при записи в файл. Она задает записываемое значение, состоящее из единственного пробела, но помимо этого содержит указание, что под выводимое значение отводится поле из `n` позиций. Выводимый пробел прижимается к правой границе этого поля, а все лидирующие позиции также заполняются пробелами.

10.1.3. Работа с типизированными файлами

Напомним, что типизированный файл состоит из порций данных, тип которых был объявлен при описании соответствующей файловой переменной. В качестве такой порции могут выступать значения скалярных переменных любого типа, строки, содержимое полей записи. Это означает, что все переменные списка ввода в операторе `read` и все выражения в списке оператора `write` должны быть такого же типа.

В чем принципиальное отличие между обменом данными с текстовым и типизированным файлом?

При записи числовой информации в текстовый файл значения данных сначала переводятся из машинного формата в символьное представление, длина которого является переменной. Как правило, размер символьного представления числа существенно превышает его длину в машинном формате. Так, например, однобайтовая величина в символьном виде может "растянуться" на 4 байта. Строковые данные при записи в текстовый файл уменьшают свою длину всего на один байт (байт длины). Поэтому вывод в текстовый файл обычно связан с дополнительными затратами времени на перевод числовых данных и с увеличением объема занимаемой внешней памяти. При чтении данных из текстового файла вновь возникает дополнительная работа по преобразованию числовых значений в соответствующий машинный формат и по формированию байта длины для переменных строкового типа.

Обмен с типизированными файлами лишен обоих недостатков — данные записываются в файл в том же формате, в каком они хранятся в оперативной памяти. Поэтому с точки зрения повышения производительности программы и экономии места на внешних носителях выгоднее работать с типизированными файлами. Может сложиться впечатление, что применение текстовых файлов оправдано в тех задачах, где приходится обрабатывать только строковые данные. На самом деле это не совсем так. Дело в том, что содержимое текстового файла, включающее как числовую, так и текстовую информацию, является универсальным способом представления данных. Их может прочитать человек, они могут быть переданы на другие вычислительные системы, в которых используются другие внутренние

форматы хранения числовой и символьной информации. Но если файл используется только для временного хранения данных, которые в ближайшее время будут использованы на том же компьютере и в рамках той же системы программирования, то предпочтение надо отдать типизированным файлам.

Следует обратить внимание еще на одно преимущество типизированных файлов по сравнению с текстовыми файлами. Так как все записи в типизированном файле равной длины, то имеется возможность вести не только последовательный обмен, но и выбирать нужную запись. Записи в таком файле нумеруются с 0. После открытия файла становится доступной начальная запись. Для перемещения в начало записи с номером `n_REC` используется процедура `seek`:

```
seek(vf, n_REC);
```

Функция `FilePos` позволяет определить номер текущей доступной записи:

```
pos:=filepos(vf);
```

С помощью функции `FileSize` можно узнать количество записей в файле:

```
k:=filesize(vf);
```

Программа `rec_file` сначала демонстрирует возможность последовательного вывода в типизированный файл, а затем считывает записи в обратном порядке (листинг 10.4).

Листинг 10.4. Программа `rec_file`

```
program rec_file;
type
  qq=record
    s: string[5];
    b: byte;
    d: double;
  end;
var
  f: file of qq;
  j: byte;
  rec: qq;
begin
  assign(f,'5.dat');
  rewrite(f);
  rec.s:='Line ';
  for j:=1 to 10 do
    begin
      rec.b:=j;
      rec.d:=sqrt(j);
      write(f,rec);
      writeln(rec.s,rec.b:4,rec.d:10:4);
    end;
```

```
  close(f);
  writeln;
  reset(f);
  for j:=9 downto 0 do
    begin
      seek(f,j);
      read(f,rec);
      writeln(rec.s,rec.b:4,rec.d:10:4);
    end;
  readln;
end.
```

На экране результаты обмена с файлом 5.dat выглядят следующим образом:

```
Running "c:\fpc\myprog\rec_file.exe "
Line    1    1.0000
Line    2    1.4142
Line    3    1.7321
Line    4    2.0000
Line    5    2.2361
Line    6    2.4495
Line    7    2.6458
Line    8    2.8284
Line    9    3.0000
Line   10    3.1623

Line   10    3.1623
Line    9    3.0000
Line    8    2.8284
Line    7    2.6458
Line    6    2.4495
Line    5    2.2361
Line    4    2.0000
Line    3    1.7321
Line    2    1.4142
Line    1    1.0000
```

Файл прямого доступа, *открытый для чтения*, позволяет в процессе обработки производить не только операции *чтения*, но и *записи*. Приведенный в листинге 10.5 пример любопытен еще и тем, что содержимое одного и того же файла можно рассматривать и как набор строк, и как типизированный файл с однобайтовыми записями. В программе `txt_rec` сначала формируется содержимое текстового файла, содержащего случайное количество строк (`k` — количество строк). Каждая строка содержит случайный набор отображаемых символов из первой половины таблицы

ASCII (с кодами от 32 до 127). Число символов в строке от 1 до 40 (j — длина строки). Каждая строка, записываемая в файл, дублируется на экране. Затем текстовый файл закрывается и открывается повторно, но уже как файл прямого доступа. Очередной символ, считываемый из файла, подвергается преобразованию с помощью функции LowerCase. При этом все символы, не принадлежащие интервалу [A..Z], в том числе и управляющие байты, завершающие каждую строку, сохраняются без изменения (для экономии их можно было бы не подвергать преобразованию). Модифицированный таким образом символ *записывается* в файл, открытый для *чтения*. После завершения перекодировки модифицированный файл вновь открывается как текстовый и его содержимое выводится на экран для сопоставления с исходным набором данных.

Листинг 10.5. Программа txt_rec

```
program txt_rec;
var
  ft: text;
  fc: file of char;
  s: string;
  ch: char;
  i,j,k: integer;
begin
  randomize;
  assign(ft,'1.txt');
  rewrite(ft);
  k:=random(10)+1;
  // Создаем текстовый файл со случайным набором символов
  for i:=1 to k do
    begin
      s:='';
      for j:=1 to random(40)+1 do
        s:=s+chr(random(95)+32);  // формирование случайной строки
      writeln(ft,s);    // запись исходной строки в файл
      writeln(s);       // вывод исходной строки на экран
    end;
  close(ft);
  writeln;   // пробел между исходным
             // и модифицированным набором данных
  assign(fc,'1.txt');
  reset(fc);  // открываем файл для чтения как типизированный
  while not EOF(fc) do
```

```
    begin
      read(fc,ch);              // читаем из файла очередной символ
      ch:=LowerCase(ch);        // заменяем большую букву малой
      seek(fc,FilePos(fc)-1);   // возвращаемся на одну запись
      write(fc,ch);             // записываем ее в файл
    end;
  close(fc);
  // Выводим на экран содержимое перекодированного файла
  assign(ft,'1.txt');
  reset(ft);
  for i:=1 to k do
    begin
      readln(ft,s);
      writeln(s);
    end;
  close(ft);
  readln;
end.
```

Результат одного из тестовых запусков выглядит следующим образом:

```
Running "c:\fp_prog\10\05\txt_rec.exe "
Wq!M5>oeT{s?4\z%{=
8iEDB=0r&gwA4~ZY>sC%jB/Jk1:dvGa_gn:m]KR
(R^]MJB%lKu[3
Xd6l1Qa2O6L+?sh~%Ry.6nlR2Mr-Z`Y#\7K_m\
QtqU`!>SM1:h6xm'?;ELu}>Q8|n]U9Iq#puwj@
HBjm~Dxm
1IB[%{mc7*dZ[( XKp{ kZ
wq!m5>oet{s?4\z%{=
8iedb=0r&gwa4~zy>sc%jb/jk1:dvga_gn:m]kr
(r^]mjb%lku[3
xd6l1qa2o6l+?sh~%ry.6nlr2mr-z`y#\7k_m\
qtqu`!>sm1:h6xm'?;elu}>q8|n]u9iq#puwj@
hbjm~dxm
1ib[%{mc7*dz[( xkp{ kz
```

10.1.4. Работа с нетипизированными файлами

Нетипизированный или, по общепринятой терминологии, *двоичный* файл представляет собой длинную цепочку байт с содержимым любой природы. Его основное отличие от типизированных файлов заключается в том, что единицей обменной информации здесь является группа байтов фиксированной длины *(блок)*, в которой

отсутствует распределение по полям заданных типов. Для обмена с двоичным файлом используются специальные процедуры `BlockRead` и `BlockWrite`:

```
BlockRead(vf, buf, n_BLK[, N]);
BlockWrite(vf, buf, n_BLK[, N]);
```

Здесь:

- `buf` — имя переменной (обычно массив типа `byte`), играющий роль буфера в оперативной памяти;
- `n_BLK` — количество блоков, участвующих в обмене;
- `N` — имя переменной, в которую заносится количество блоков, фактически принявших участие в обмене.

По умолчанию при открытии двоичных файлов устанавливается длина блока, равная 128 байтам. Однако пользователь может заказать любую другую длину, которую он считает более удобной для своей программы:

```
Reset(vf, 80);
Rewrite(vf, 512);
```

Четвертый параметр в операторах обмена не является обязательным. Но он может оказаться полезным по ряду причин. При записи на диск может оказаться, что свободного места на диске не хватает для размещения указанного числа блоков. При чтении из файла может возникнуть ситуация, когда общее количество блоков в файле не кратно количеству блоков, указанных в операторе `BlockRead`. И тогда последняя считываемая порция данных будет содержать меньшее количество блоков.

Двоичные файлы наряду с последовательным обменом поддерживают и прямой доступ к блокам, которые подобно записям в типизированных файлах нумеруются с 0. Для перехода к блоку с указанным номером используется процедура `Seek`:

```
Seek(vf, num_BLK);
```

Опрос номера текущего доступного блока производится с помощью функции `FilePos`. Для определения общего количества блоков в файле можно воспользоваться функцией `FileSize`.

В процессе работы с двоичным файлом с помощью процедуры `Truncate (vf)` можно выполнить так называемое *усечение* данных. При этом головная часть файла до текущего блока сохраняется, а хвостовые данные удаляются.

В листинге 10.6 приводится пример копирования файла. Независимо от структуры данных, хранящихся в копируемом файле, он рассматривается как двоичный. На самом деле, способ организации данных в файле и специфика их обработки определяются программистом. В нашем случае необходимо скопировать содержимое файла байт в байт, независимо от того, что в каждом байте находится. В первую очередь, этим обстоятельством диктуется выбор размера блока, равный одному байту. Кроме того, такая длина блока позволяет не задумываться над количеством блоков, хранящихся в файле. На каждом шаге чтения из исходного файла мы запрашиваем некоторое количество блоков и переписываем в копию ровно столько блоков, сколько удалось прочитать. Равенство нулю этой величины фиксирует конец операции копирования.

Листинг 10.6. Программа копирования файла `copy_file`

```
program copy_file;
const
  n_buf=30000; {размер буфера обмена}
var
  buf: array [1..n_buf] of byte;
  vf_in, vf_out: file;
  num_in, num_out: integer;
begin
  assign(vf_in,'5.txt');
  reset(vf_in,1);
  assign(vf_out,'6.bin');
  rewrite(vf_out,1);
  repeat
    blockread(vf_in,buf,n_buf,num_in);
    dlockwrite(vf_out,buf,num_in,num_out);
  until (num_in = 0)or(num_in <> num_out);
  close(vf_in);
  close(vf_out);
end.
```

В ранних версиях систем Turbo Pascal, функционировавших под управлением MS-DOS, придавалось большое значение объему порции данных, участвовавших в обмене. Системное ограничение сверху (32 767) позволяло за один присест прочитать максимальное количество подряд идущих секторов на диске — кластер. Это обеспечивало оптимальную скорость обмена. С увеличением объема винчестеров и ростом размера кластеров за оптимизацией обмена следит электроника, управляющая работой винчестера. Поэтому задание в программе размера порции данных, считываемых за один присест, особой роли не играет.

Еще одна логическая проверка (`num_in <> num_out`) в завершающей строке цикла `repeat` связана с возможным переполнением диска.

Программа `reserve` демонстрирует еще одну операцию, характерную для многих редакторов — создание резервной копии файла с расширением bak (листинг 10.7).

Листинг 10.7. Программа создания резервной копии файла `reserve`

```
program reserve;
uses SysUtils;
const
  N = 32767;
var
  f1, f2: file;
```

```
  rd, wr: integer;
  buf: array [1..N] of byte;
  name1, name2: string;
begin
  if ParamCount=0 then
    begin
      writeln('Must be: reserve file');
      readln; exit;
    end;
  name1:=ParamStr(1);
  assign(f1,name1);
  {$I-} reset(f1,1); {$I+}
  if IOresult <> 0 then
    begin
      writeln('File ',name1,' not found');
      readln;  exit;
    end;
  name2:=ChangeFileExt(name1,'.BAK');
  assign(f2,name2);
  rewrite(f2,1);
  repeat
    blockread(f1,buf,N,rd);
    blockwrite(f2,buf,rd,wr);
  until(rd=0)or(rd <> wr);
  close(f1);
  close(f2);
end.
```

Для запуска этой программы в командной строке должно быть задано имя файла, для которого создается резервная копия. Имя файла с резервной копией формируется с помощью системной функции `ChangeFileExt`. Заполнение файла с резервной копией выполняется по схеме, описанной в предыдущем примере.

Тесная связь между типизированными и нетипизированными файлами подчеркивается программой `bin_rec` (листинг 10.8). Она выполняет точно такие же действия, как и программа `rec_file`. Работа с полями записи здесь моделируется с помощью указателей.

Листинг 10.8. Программа `bin_rec`

```
program bin_rec;
var
  buf: array [1..17] of byte;
  f1: file;
  p: pointer;
```

```
  ps: ^string;
  pi: ^integer;
  pd: ^double;
  j: integer;
begin
  assign(f1,'7.bin');
  p:=@buf[1]; ps:=p;
  p:=@buf[6]; pi:=p;
  p:=@buf[10];pd:=p;
  rewrite(f1,17);
  for j:=1 to 10 do
    begin
      pi^:=j;
      pd^:=sqrt(j);
      blockwrite(f1,buf,1);
      writeln(ps^,pi^:4,pd^:10:4);
    end;
  close(f1);
  writeln;
  reset(f1,17);
  for j:=9 downto 0 do
    begin
      seek(f1,j);
      blockread(f1,buf,1);
      writeln(ps^,pi^:4,pd^:10:4);
    end;
  close(f1);
  readln;
end.
```

10.2. Управление файлами в стиле Windows

В режиме совместимости с Delphi система Free Pascal поддерживает довольно много процедур и функций по управлению каталогами и двоичными файлами. В большинстве своем новые процедуры используют числовые атрибуты — хэндлы (от англ. *handle*), которые операционная система присваивает файлам при их создании или открытии:

```
hF:=FileCreate(f_name);   {создание нового файла с указанным именем}
hF:=FileOpen(f_name, mode);  {открытие существующего файла}
```

Имя файла `f_name` — выражение или переменная типа `string`. Режим работы открываемого файла определяется двоичным кодом `mode`, который формируется

как логическая сумма из нужных наборов признаков (каждый из признаков соответствует двоичному разряду в шкале `mode`):

- `fmOpenRead` — для ввода;
- `fmOpenWrite` — для вывода;
- `fmOpenReadWrite` — для ввода и вывода;
- `fmShareExclusive` — с запретом доступа из других программ;
- `fmShareDenyWrite` — с запретом записи для других программ;
- `fmShareDenyRead` — с запретом чтения из других программ;
- `fmDenyNone` — с разрешением любого доступа из других программ.

Например:

```
hF:=FileOpen('1.txt',fmOpenRead or fmShareDenyRead);
```

Хэндл `hF` играет такую же роль, как и переменная файлового типа. Например, для закрытия файла надо обратиться к процедуре `FileClose`:

```
FileClose(hF);
```

Функции чтения и записи здесь выглядят более естественно:

```
kr:=FileRead(hF, buf, count);
kw:=FileWrite(hF, buf, count);
```

Здесь:

- `buf` — адрес буфера в оперативной памяти, участвующего в обмене;
- `count` — количество байтов, заказанных программой.

В переменные `kr` и `kw` возвращается фактическое количество байтов, которые удалось прочитать или записать.

Для перехода к байту с указанным номером (т. е. для организации прямого доступа) используется функция `FileSeek`:

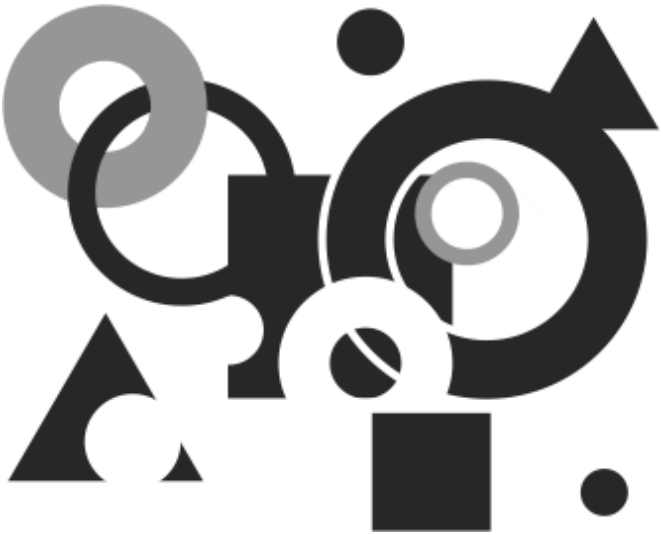
```
pos:=FileSeek(hF,Offset,Origin);
```

Указатель файла, скрытый от пользователя, определяет номер текущего доступного байта, или, как говорят, текущую позицию в файле. Аргумент `Offset` задает смещение в байтах, определяющее новую позицию указателя относительно точки, на которую указывает параметр `Origin`. Если этой точкой является начало файла, то следует задавать `Origin=0`. Если смещение отсчитывается от текущей позиции указателя, третий аргумент должен быть равен 1. Значение `Origin=2` определяет, что точкой отсчета является конец файла. Функция `FileSeek` возвращает число типа `integer` — новую позицию указателя файла.

С помощью логической функции `FileExist(f_name)` можно проверить, существует ли файл с указанным именем. Логическая функция `DeleteFile(f_name)` возвращает значение `True`, если она удалила файл с указанным именем.

Операционная система поддерживает довольно много процедур и функций по манипуляции с атрибутами файлов, по расчленению и объединению компонентов имени (путь, имя, расширение), по управлению каталогами (создание, переименование, удаление), по поиску файлов, чьи имена удовлетворяют заданному шаблону поиска (`FindFirst`, `FindNext`, `FindClose`) и др. Дополнительную информацию по форматам вызова этих подпрограмм вы можете найти в файле помощи Win32.hlp[1].

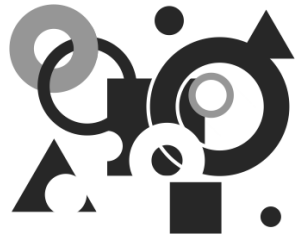
[1] Поиск этого файла попробуйте выполнить в Интернете. — *Прим. ред.*



ЧАСТЬ II

Модули

Глава 11



Модули и объекты

Модули в языке Pascal являются базовым средством для создания библиотек подпрограмм и объявления нестандартных типов данных. В отличие от других программных единиц (головная программа, функция или процедура) первая строка модуля начинается с оператора `Unit`, вслед за которым расположено имя модуля:

```
Unit file_name;
```

Если имя головной программы может не совпадать с именем файла, под которым она хранится на диске, то имя модуля и имя соответствующего файла должны быть одинаковы. Причина этого требования заключается в следующем: при определенных обстоятельствах полная пересборка программы включает перекомпиляцию всех нестандартных модулей. В этом случае компилятор ищет исходный текст модуля по его имени с расширением pas или pp.

Оформление модуля, как и головной программы, заканчивается оператором `end` с завершающей точкой. Однако последовательность всех остальных частей модуля жестко фиксирована.

Первая структурная часть модуля начинается со служебного слова `INTERFACE`. Раздел *интерфейса* должен содержать заголовки всех процедур и функций, доступных внешнему пользователю. Сюда же могут быть включены *общедоступные* типы данных, констант и переменных. В некоторых модулях, имеющих специфическое назначение, содержательная часть интерфейса может отсутствовать. Но служебное слово `Interface` опускать нельзя. Модули подобного типа обычно выполняют некоторое нестандартное действие перед запуском программы.

Второй обязательной частью каждого модуля является раздел *реализации*, началом которого служит служебное слово `IMPLEMENTATION`. В этом разделе объявляются внутренние типы данных, константы и переменные модуля, не доступные внешнему пользователю, приводятся исходные тексты внешних и внутренних процедур и функций. Оформление всех процедур и функций в модуле ничем не отличается от аналогичных конструкций в головных программах. Содержимое раздела реализации тоже может оказаться пустым, но слово `IMPLEMENTATION` опускать нельзя. Такие модули нередко конструируются ради включения в раздел интерфейса полезных типов данных, структур и констант. Например:

```
Unit Calendar;
Interface
```

```
type
ShortMonthName = ('Jan'=1, 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',
                  'Aug', 'Sep', 'Oct', 'Nov', 'Dec');
LongMonthName = ('January'=1, 'February', 'March', 'April', 'May',
                 'June', 'July', 'August', 'September', 'October',
                 'November', 'December');
ShortDayWeek = ('Mon'=1, 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun',);
LongDayWeek = ('Monday'=1, 'Tuesday', 'Wednesday', 'Thursday',
               'Friday', 'Saturday', 'Sunday');
Implementation
end.
```

Третьей необязательной частью модуля является раздел *инициализации*. В ранних версиях Паскаля началом такой секции считалось служебное слово `begin`, которое можно было опускать в случае отсутствия раздела инициализации. Основное назначение раздела инициализации — выполнение некоторых действий, предшествующих старту головной программы. В разделе инициализации, например, могут открываться какие-то файлы, общедоступным переменным модуля могут присваиваться начальные значения, на экран может выдаваться некоторая заставка и т. п. В современных версиях языков Object Pascal и Free Pascal разделу инициализации обычно предшествует новое служебное слово `INITIALIZATION`.

Заключительная необязательная часть модуля, появившаяся только в последних версиях языка, начинается со служебного слова `FINALIZATION`. Обычно ее используют для возврата ресурсов, захваченных программой, при возникновении непредвиденных обстоятельств.

11.1. Стандартные модули Free Pascal

В составе системы FP поставляется набор модулей, готовых к употреблению программами пользователей. Их общее количество превышает 40 единиц, однако лишь половина из них ориентирована на эксплуатацию под управлением Windows. И только порядка десятка могут стать повседневным инструментом большинства программистов. Список таких модулей приведен в табл. 11.1.

Таблица 11.1

Имя модуля	Назначение
`Crt`	Управление дисплеем в "текстовом режиме"
`DateUtils`	Обработка календарных дат и интервалов времени
`DOS`	Опрос и установка системной даты и таймера, работа с прерываниями
`Graph`	Работа с библиотекой графических программ Borland Graphics Interface (BGI)

Таблица 11.1 *(окончание)*

Имя модуля	Назначение
Math	Вычисление элементарных и специальных функций
Strings	Работа со строковыми данными типа PChar
StrUtils	Работа со строковыми данными типа AnsiString
System	Набор наиболее используемых подпрограмм разного назначения
SysUtils	Расширенный аналог такого модуля Delphi
Windows	Поддержка вызова системных функций Windows (Win32 API)

Модуль System автоматически подключается к консольным приложениям, о подключении других модулей программист должен позаботиться сам.

Более или менее подробное описание состава функций и процедур некоторых модулей приведено в различных главах и разделах этой книги — Crt *(см. главу 12)*, DateUtils *(см. разд. 14.2)*, Graph *(см. главу 15)*, Strings и StrUtils *(см. главу 5)*. Для более детального знакомства с содержимым других модулей мы советуем обратиться к документу "Run-Time Library (RTL): Reference guide" (см. файл \Fpc\2.4.0\doc\rtl.pdf на компакт-диске). Это самый объемный том в технической документации, он содержит порядка 1500 страниц.

11.1.1. Создание нестандартного модуля

К нестандартным относятся модули, создаваемые программистами или заимствуемые ими из различных источников. Продемонстрируем технику создания модуля на примере разработки библиотеки подпрограмм выполнения операций над обычными дробями, с которыми российские школьники знакомятся в младших классах, а французские, как выяснилось, часто не имеют представления о дробях даже при поступлении в технические вузы.

Начнем со структуры данных. На наш взгляд, для хранения значения дроби лучше всего подходит запись, содержащая два целочисленных поля — числитель и знаменатель:

```
type
  Ratio=record
    n,d: integer;
  end;
```

Если мы теперь объявим переменную x типа Ratio, то к ее числителю можно обратиться по имени x.n, а к знаменателю — по имени x.d. Конечно, для присвоения дроби значения 1/3 можно поступить следующим образом:

```
x.n:=1;
x.d:=3;
```

Но в этом случае мы, во-первых, не застрахованы от ситуации `x.d=0`, которая может привнести к аварийному останову. Во-вторых, нормальные люди сокращают некоторые дроби, если такая возможность имеется (3/6 = 1/2). Наконец, было бы удобнее выполнять такую операцию в одну строку:

```
x:=Rset(1,3);
```

Согласитесь, что не очень удобно записывать в программе процедуру сложения дробей `x` и `y` по правилам, которые мы учили в школе:

```
z.n:=x.n*y.d+x.d*y.n;
z.d:=x.d*y.d;
```

В две строки — это длинновато, а потом после сложения может возникнуть необходимость сокращения. Было бы лучше написать функцию сложения двух дробей:

```
z:=Radd(x,y);
```

Во-первых, это короче. Во-вторых, в функции сложения можно было бы предусмотреть возможность сокращения результата.

Учитывая соображения подобного рода, мы предлагаем написать группу функций первой необходимости, список которых приведен в табл. 11.2.

Таблица 11.2

Формат вызова	Выполняемое действие
`x:=Rset(n1,d1);`	Присвоение `x=n1/d1` с проверкой `d1≠0` и сокращением
`z:=Radd(x,y);`	Сложение дробей с сокращением
`z:=Rsub(x,y);`	Вычитание дробей с сокращением
`z:=Rmult(x,y);`	Умножение дробей с сокращением
`z:=Rdiv(x,y);`	Деление дробей с проверкой `x.n≠0` и сокращением

Очевидно, что нам понадобятся процедуры для ввода и вывода дробей в привычной для пользователя форме:

```
Rin(x);     // входная информация может иметь вид: 1/3
Rout(x);    // выводить разумно в таком же формате: 1/3
```

Естественно, что ввод должен был сопровождаться проверкой знаменателя на 0 и предусматривать возможность сокращения. А в процедуре вывода оказалось полезным включение дополнительного аргумента — строки, которая могла идентифицировать выводимое значение.

Среди неупомянутых операций над дробями мы решили включить в библиотеку операцию сравнения дробей и функции прямых и обратных преобразований между вещественными данными и традиционными дробями. Функция сравнения возвращает положительный результат, если первая дробь больше второй, нулевой

результат в случае равенства дробей и отрицательный результат, если первая дробь меньше второй.

После всего сказанного представляем на ваш суд модуль `Rational`, который реализует заявленные выше операции (листинг 11.1). Две функции — определения наибольшего общего делителя (`gcd`) и сокращения дроби (`reduce`) — мы решили скрыть от пользователя и не включили их в интерфейсную часть.

Некоторые трудности пришлось преодолеть в процедуре ввода дроби и преобразования значений числителя и знаменателя, заданных во введенной строке. Дело в том, что процедура `val` не умеет переводить отрицательные числа. Пришлось искусственным путем удалять из введенной строки символы "`-`" и "`/`". Еще одна потенциальная опасность подстерегала нас в реализации процедуры `Rout`. Сначала было принято решение анализировать знак дроби как знак произведения `x.n*x.d`. Но в функции преобразования `DtoR` устанавливаются такие большие значения числителя и знаменателя, что они приводили к переполнению в указанном перемножении. Для сохранения знака числителя пришлось поделить его на `abs(x.n)`.

Листинг 11.1. Модуль `Rational`

```
Unit Rational;
INTERFACE
// Общедоступные типы данных
type
  Ratio=record
    n,d: integer;
  end;
// Общедоступные процедуры и функции
  procedure Rout(s:string; x:Ratio);
  procedure Rin(s:string; var x:Ratio);
  function Rset(n,d:integer):Ratio;
  function Radd(x,y:Ratio):Ratio;
  function Rsub(x,y:Ratio):Ratio;
  function Rmult(x,y:Ratio):Ratio;
  function Rdiv(x,y:Ratio):Ratio;
  function Rcomp(x,y:Ratio):integer;
  function RtoD(x:Ratio):double;
  function DtoR(x:double):Ratio;
//-------------------------------------
IMPLEMENTATION
// Определение НОД
function gcd(x,y:integer):integer;
begin
  if y=0 then begin Result:=x; exit; end;
```

```
  Result:=gcd(y,x mod y);
end;

// Сокращение дроби
procedure reduce(var x:Ratio);
var
  n1,d1: integer;
begin
  n1:=abs(x.n);
  if n1=0 then begin x.d:=1; exit; end;
  d1:=gcd(n1,x.d);
  if d1>1 then begin
    x.n:=x.n div d1;  x.d:=x.d div d1;
  end;
end;

// Вывод дроби
procedure Rout(s:string;x:Ratio);
var
  k:integer;
  begin
  write(s);                       // вывод комментария
  if x.n=0 then
  begin writeln('0'); exit; end;
  k:=(x.n div abs(x.n))* x.d;     // анализ знака дроби
  if k<0 then write('-');
  writeln(abs(x.n),'/',abs(x.d));
end;

// Ввод дроби
procedure Rin(s:string; var x:Ratio);
var
  s1,s2: string[20];
  i,j: integer;
begin
  x.d:=1;
  write(s);                       // приглашение ко вводу
  readln(s1);                     // ввод дроби-строки вида 'n/d'
  i:=pos('/',s1);                 // поиск символа '/'
  if i<>0 then begin              // есть ли знаменатель?
     s2:=copy(s1,i+1,10);         // часть строки со знаменателем
     val(s2,x.d,j);               // перевод знаменателя
```

```
    delete(s1,i,10);            // удаление знаменателя из s1
  end;
  if s1[1]='-' then begin       // анализ на '-'
    delete(s1,1,1);             // удаление символа '-'
    val(s1,x.n,j);              // перевод модуля числителя
    x.n:=-x.n;                  // смена знака числителя
  end
  else val(s1,x.n,j);           // перевод числителя
  reduce(x);
end;

// Присвоение дроби значения
function Rset(n,d:integer):Ratio;
begin
  if n*d=0 then begin           // если задан нулевой знаменатель
    Result.n:=0;  Result.d:=1;  exit;
  end;
  Result.n:=abs(n);  Result.d:=abs(d);
  if n*d < 0 then  Result.n:=-Result.n;
end;

// Сложение дробей
function Radd(x,y:Ratio):Ratio;
begin
  Result.n:=x.n*y.d + y.n*x.d;
  Result.d:=x.d*y.d;
  reduce(Result);        // сокращение результата
end;

// Вычитание дробей
function Rsub(x,y:Ratio):Ratio;
begin
  Result.n:=x.n*y.d-y.n*x.d;
  Result.d:=x.d*y.d;
  reduce(Result);
end;

// Умножение дробей
function Rmult(x,y:Ratio):Ratio;
begin
  Result.n:=x.n*y.n;
  Result.d:=x.d*y.d;
```

```
  reduce(Result);
end;

// Деление дробей
function Rdiv(x,y:Ratio):Ratio;
begin
  if y.n=0 then begin    // предотвращение деления на 0
    writeln('Divide by zero');
    readln;
    halt;                // завершение работы при делении на 0
  end;
  Result.n:=x.n*y.d;
  Result.d:=x.d*y.n;
  reduce(Result);
end;

// Сравнение дробей
function Rcomp(x,y:Ratio):integer;
var
  a,b: integer;
begin
  a:= x.n*y.d;  b:= x.d*y.n;
  if a>b then Result:=1
  else  if a<b then Result:=-1
  else Result:=0;
end;

// Преобразование Ratio в Double
function RtoD(x:Ratio):double;
begin
  Result:=x.n/x.d;
end;

// Преобразование Double в Ratio
function DtoR(x:double):Ratio;
begin
  Result.n:=Round(x*9e7);
  Result.d:=90000000;
  reduce(Result);
end;
end.
```

В качестве одного из тестовых примеров была использована программа rational_2.pas (листинг 11.2).

Листинг 11.2. Программа rational_2.pas

```
program rational_2;
uses rational;
var
  x,y,z: ratio;
  d: double;
begin
  x:=Rset(2,3);
  Rout('x=',x);
  y:=Rset(-1,2);
  Rout('y=',y);
  z:=Radd(x,y);
  Rout('x+y=',z);
  z:=Rsub(x,y);
  Rout('x-y=',z);
  z:=Rmult(x,y);
  Rout('x*y=',z);
  z:=Rdiv(x,y);
  Rout('x/y=',z);
  d:=0.75;
  z:=DtoR(d);
  Rout('0.75=',z);
  d:=RtoD(x);
  writeln('2/3=',d:8:6);
  write('Input x :');
  Rin(x);
  Rout('x=',x);
  readln;
end.
```

Результаты ее работы таковы:

```
Running "e:\fpc\myprog\rational_2.exe "
x=2/3
y=-1/2
x+y=1/6
x-y=7/6
x*y=-1/3
x/y=-4/3
```

```
0.75=3/4
2/3=0.666667
Input x :-3/9
x=-1/3
```

11.2. Программирование с объектами

В этом разделе демонстрируются некоторые идеи объектно-ориентированного программирования, реализованные в режиме **Object Pascal extension on**, который устанавливается с помощью команды **Options → Compiler**.

Объект представляет собой сложный тип данных, состоящий из некоторого набора полей и совокупности подпрограмм, так или иначе связанных с обработкой содержимого этих полей. В соответствии с описанием объекта в программе могут быть созданы *переменные данного типа*. Для каждой такой *"переменной"*, как правило, выделяется персональная память для хранения содержимого всех полей, объявленных в объекте. С *переменными-объектами* можно выполнять все операции, предусмотренные набором подпрограмм, включенных в описание объекта. Чтобы отличать поля одного объекта от одноименных полей другого объекта, используют составные имена, такие же, как и в записях. В отличие от полей объекта, которые тиражируются для каждой объявленной переменной, обслуживающие подпрограммы автоматически включаются в программу пользователя и не дублируются.

Для того чтобы пользователь не смог совершить какие-либо несанкционированные операции над содержимым полей в переменной-объекте, прямой доступ к этим полям обычно блокируют. Для этой цели в описании объекта выделяют раздел общедоступных полей и подпрограмм, предваряемый служебным словом `public` (публичный, общедоступный), и раздел внутренних данных и служебных подпрограмм, не доступных для внешнего пользователя. Последний раздел идентифицируется служебным словом `private` (частный, секретный). Доступ к закрытым таким образом полям внешний пользователь может получить только через служебные подпрограммы, в которых неправомерное изменение соответствующих значений пресекается. Такое разделение данных и программ на доступные и недоступные появилось еще в модулях. Поэтому модули особенно удобны для размещения описания объектов.

Мы продемонстрируем процесс описания объекта на примере внедрения данных типа рациональных дробей и создания библиотеки операций над ними. Аналогичная работа была проделана ранее в этой главе по созданию модуля `Rational`. Однако использование технологии объектов позволит более естественно программировать операции над дробями. Если в предыдущем модуле, например, для сложения двух дробей мы вызывали функцию `Radd`:

```
z:=Radd(x,y);
```

то привлечение объектов позволит нам ту же операцию записывать в виде:

```
z:=x+y;
```

Сохраняем за модулем прежнее название и помещаем в раздел интерфейса заголовочную часть описания объекта (листинг 11.3).

Листинг 11.3. Модуль `Rational` с использованием объектов. Раздел `INTERFACE`

```
unit Rational;
// Компилировать в режиме Object Pascal extension on
INTERFACE
type
  Ratio = object
  private
    n,d: integer;
  public
    procedure   Rout(s:string);
    procedure   Rin(s:string);
    constructor Init;
    constructor Init(x:Ratio);
    constructor Init(n1,d1:integer);
end;
  operator + (x,y:Ratio):Ratio;
  operator - (x,y:Ratio):Ratio;
  operator * (x,y:Ratio):Ratio;
  operator / (x,y:Ratio):Ratio;
  function DtoR(v:double):Ratio;
  function RtoD(x:Ratio):Double;
```

Собственно заголовок описания объекта расположен между служебными словами `type` и `end`. Начинается он с имени типа объекта `Ratio`, сопровождаемого служебным словом `object`. Далее описаны два целочисленных поля с именами `n` (от англ. *numerator* — числитель) и `d` (от англ. *denominator* — знаменатель), прямой доступ к которым запрещен. Вслед за служебным словом `public` расположены пять строк с заголовками специальных подпрограмм, именуемых *методами*. Если к предыдущим подпрограммам ввода/вывода мы обращались следующим образом:

```
Rin('x=',x);
Rout('z=',z);
```

то к заменившим их методам нужно обращаться немного по-другому:

```
x.Rin('x=');
z.Rout('z=');
```

Переменная-объект, к которой применяется соответствующая операция, перенесена из фактических параметров подпрограммы в начало обращения к методу, от которого она отделяется точкой.

В следующих строках записаны заголовки еще одного типа специальных *подпрограмм-конструкторов*. Их назначение — сформировать начальные значения полей в переменных-объектах, объявляемых в программе пользователя:

```
var
  x: Ratio;     // объявление переменной-объекта
  ...
  x.Init;       // инициализация полей в переменной-объекте
```

Техника объектно-ориентированного программирования предусматривает использование трех стандартных конструкторов. Конструктор без параметров, который обычно называют *конструктором по умолчанию*, как правило, заполняет поля данных нулями (для числовых данных — нулевыми значениями, для строковых данных — пустыми строками). В нашем случае засылка нуля в знаменатель чревата последующим переполнением, поэтому конструктор `Init` в данной реализации будет немного нестандартным: в числитель он зашлет ноль, а в знаменатель — единицу. Второй тип конструктора носит название *конструктора копирования*. Он пересылает (копирует) значение ранее инициализированной переменной-объекта в новую переменную:

```
y.Init(x);     // значения полей x копируются в поля y
```

В третьем типе конструктора его аргументы в явном виде представляют значения числителя и знаменателя. Дополнительная задача этого конструктора заключается в блокировке попытки задать нулевой знаменатель.

Вслед за заголовком объекта расположены заголовки некоторых подпрограмм, включенных в состав средств, обслуживающих объект. Первые четыре из них, начинающиеся со служебного слова `operator`, — специальные функции, используемые для переопределения операций над переменными-объектами. Данные типа рациональных дробей не входят в перечень объектов, "знакомых" компилятору. Его следует научить, как надо выполнять ту или иную арифметическую операцию над данными, придуманными пользователем. Именно с этой задачей справляются `operator`-функции, реализация которых будет представлена в разделе `IMPLEMENTATION`. Кроме компилятора никто другой к этим функциям обращаться не будет, поэтому для их заголовков придуман особый формат.

Зато две следующие подпрограммы представлены обычными функциями преобразования данных типа `Ratio` в вещественные значения (`RtoD`) и преобразованиями вещественных данных в дробно-рациональное представление (`DtoR`).

Раздел реализации начинается со служебных функций `gcd` и `reduce`, уже знакомых нам по предыдущей версии модуля `Rational`. Вслед за ними приведена прежняя реализация функций конвертирования данных (листинг 11.4).

Листинг 11.4. Модуль `Rational` с использованием объектов. Раздел `IMPLEMENTATION`

```
IMPLEMENTATION
// Вычисление НОД
function gcd(x,y:integer):integer;
```

```
begin
  if y=0 then begin Result:=x; exit; end;
  Result:=gcd(y,x mod y);
end;

// Сокращение дроби
procedure reduce(var x:Ratio);
var
  n1,d1:integer;
begin
  n1:=abs(x.n);
  if n1=0 then begin x.d:=1; exit; end;
  d1:=gcd(n1,x.d);
  if d1>1 then begin
    x.n:=x.n div d1;
    x.d:=x.d div d1;
  end;
end;

// Преобразования между Double и Ratio
function DtoR(v:double):Ratio;
begin
  Result.n:=round(v*9e7);
  Result.d:=90000000;
  reduce(Result);
end;
function RtoD(x:Ratio):Double;
begin
  Result := x.n / x.d;
end;
```

Пока что ничего нового в содержание предыдущих подпрограмм не внесено. Некоторые нюансы появляются в оформлении конструкторов. Для указания их принадлежности объекту с именем `Ratio` к имени конструктора добавлен соответствующий префикс (листинг 11.5).

Листинг 11.5. Модуль `Rational` с использованием объектов. Конструкторы

```
// Конструкторы-инициализаторы объектов типа Ratio.
// Конструктор по умолчанию.
constructor Ratio.Init();
```

```
begin
  n:=0;   d:=1;
end;

// Конструктор копирования
constructor Ratio.Init(x:Ratio);
begin
  n:=x.n;  d:=x.d;
end;

// Конструктор по заданным числителю и знаменателю
constructor Ratio.Init(n1,d1:integer);
begin
  n:=n1;
  d:=d1;
  if d=0 then begin
     d:=1;  exit;
  end;
  reduce(Self);
end;
```

Так как задача любого конструктора — заполнить поля переменной-объекта нужными значениями, то ничего особенного в теле двух первых конструкторов нет. В третьем конструкторе предотвращается попытка присвоить знаменателю нулевое значение и появляется некоторое специальное имя `Self` (дословно — та самая переменная-объект, над которой выполняется инициализация третьего типа). Дело в том, что при произвольно заданных значениях числителя и знаменателя не мешает произвести сокращение дроби, если такая возможность представится. Но для обращения к процедуре сокращения необходим аргумент типа `Ratio`. А среди аргументов конструктора имя инициализируемой переменной отсутствует — при обращении к методам это имя выносится из списка параметров. Но компилятор-то знает, к какому объекту должен быть применен вызываемый метод. Именно этому объекту в момент вызова временно приписывается имя `Self`.

Поскольку подпрограммы ввода/вывода превратились в методы, то при их описании тоже пришлось приклеить префикс `Ratio`. Кроме того, после ввода значения дроби может понадобиться ее сокращение, поэтому в процедуре `Rin` пришлось прибегнуть к услугам переменной `Self` (листинг 11.6).

Листинг 11.6. Модуль `Rational` с использованием объектов. Методы

```
// Метод объекта - вывод дроби
procedure Ratio.Rout(s:string);
```

```
var
  k:integer;
  begin
    write(s);
    if n=0 then
      begin writeln('0'); exit; end;
    k:=(n div abs(n))* d;
    if k<0 then write('-');
    writeln(abs(n),'/',abs(d));
  end;

// Метод объекта - ввод дроби
procedure Ratio.Rin(s:string);
var
  s1,s2:string[20];
  i,j:integer;
begin
  d:=1;
  write(s);
  readln(s1);
  i:=pos('/',s1);
  if i<>0 then begin
     s2:=copy(s1,i+1,10);
     val(s2,d,j);
     delete(s1,i,10);
  end;
  if s1[1]='-' then begin
     delete(s1,1,1);
     val(s1,n,j);
     n:=-n;
  end
  else val(s1,n,j);
  reduce(Self);
end;
```

Переопределение арифметических операций производится по несложному шаблону. Для каждой операции используются два операнда — аргументы переопределяемой операции. Как и любая другая функция, переопределяемая операция должна вернуть результат своей работы в виде значения типа `Ratio`. Для этой цели как нельзя кстати системная переменная `Result`, тип которой определяется типом возвращаемого значения. Выполнение любой операции завершается попыткой

произвести сокращение результата. Единственная особая ситуация может возникнуть в операции деления, когда числитель делителя равен нулю. В этом случае выводится сообщение об ошибке, и после нажатия клавиши <Enter> работа программы завершается (листинг 11.7).

Листинг 11.7. Модуль `Rational` с использованием объектов. Операции

```
// Переопределение арифметических операций
operator + (x,y:Ratio):Ratio;
begin
  Result.n := x.n*y.d + x.d*y.n;
  Result.d := x.d*y.d;
  reduce(Result);
end;
operator - (x,y:Ratio):Ratio;
begin
  Result.n := x.n*y.d - x.d*y.n;
  Result.d := x.d*y.d;
  reduce(Result);
end;
operator * (x,y:Ratio):Ratio;
begin
  Result.n := x.n * y.n;
  Result.d := x.d * y.d;
  reduce(Result);
end;
operator / (x,y:Ratio):Ratio;
begin
  if y.n=0 then begin
     writeln('Деление на 0');
     readln;
     halt;
   end;
   Result.n := x.n * y.d;
   Result.d := x.d * y.n;
   reduce(Result);
end;
```

Для тестирования описанного выше модуля использовалась программа из листинга 11.8.

Листинг 11.8. Тест

```
program test_R;
uses Rational;
var
  x,y,z:Ratio;
  v:double=0.75;
begin
  x.Rin('x=');
  y.Init(1,3);
  x.Rout('x=');
  y.Rout('y=');
  z:=x+y;
  z.Rout('x+y=');
  z:=x-y;
  z.Rout('x-y=');
  z:=x*y;
  z.Rout('x*y=');
  z:=x/y;
  z.Rout('x/y=');
  z:=DtoR(v);
  z.Rout('0.75=');
  v:=RtoD(y);
  writeln('1/3=',v:10:8);
  readln;
end.
```

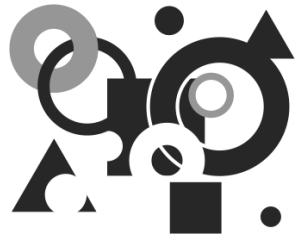
Протокол ее работы таков:

```
Running "c:\fpc\2.2.4\bin\i386-win32\test_r.exe "
x=4/6          // значение введено с клавиатуры
x=2/3
y=1/3
x+y=1/1
x-y=1/3
x*y=2/9
x/y=2/1
0.75=3/4
1/3=0.33333333
```

Конечно же, приведенный пример не исчерпывает все достижения объектно-ориентированного программирования. Класс объектов позволяет не только вводить новые типы данных и определять операции над ними. По классу-родителю могут

быть построены производные классы-дети, которым по наследству передаются данные и методы. В порожденных классах объектов могут добавляться новые данные и создаваться новые методы, переопределяться унаследованные процедуры и функции. Но язык Free Pascal в этом плане не может служить объектом подражания. В нем из-за совместимости с языком Object Pascal появилась веточка, порожденная идеологией *объектов*. А попытка сохранить совместимость с Delphi потребовала нестандартного подхода на базе *классов* фирмы Borland. Оба эти подхода в какой-то мере похожи, но такой стройной теории, которая была, например, развита в языке C++, здесь не получилось. Вот и приходится, работая в среде FP IDE в одном режиме, пользоваться терминологией объектов, а в другом режиме — терминологией классов.

Глава 12



Модуль *Crt*

Своим названием модуль обязан аббревиатуре, образованной от *Cathode-Ray Tube* (катодно-лучевая трубка). Большинство функций и процедур, входящих в состав модуля `Crt`, используются для управления дисплеем в "текстовом" режиме. Если на первых моделях IBM-совместимых ПК текстовый режим поддерживался аппаратными средствами, то под управлением Windows он моделируется. Текстовый экран может занимать всю рабочую поверхность монитора и походить на традиционный экран MS-DOS, а может быть уменьшен и выглядеть как стандартное окно Windows, но при этом сохранять прежнюю функциональность большого экрана.

Пользуясь терминологией MS-DOS, мы будем считать, что в "текстовом" режиме поддерживаются следующие форматы отображения символьной информации:

- черно-белый режим BW40 с отображением до 40 символов в строке;
- цветной режим C40 с отображением до 40 символов в строке;
- черно-белый режим BW80 с отображением до 80 символов в строке;
- цветной режим C80 с отображением до 80 символов в строке.

По умолчанию устанавливается режим C80. Окно приложения в этом режиме воспроизводит содержимое стандартного буфера консоли, который по терминологии MS-DOS назывался текстовой страницей. Основную часть информации этого буфера занимает собственно содержимое окна, в котором каждому знакоместу соответствуют два байта видеопамяти (рис. 12.1). В младшем байте находится код ASCII отображаемого символа, в старшем — его цветовые атрибуты. Цвет фона формируется из трех двоичных разрядов, каждый из которых символизирует наличие или отсутствие соответствующей базовой составляющей цвета (R — Red, G — Green, B — Blue). Цвет контура символа представлен четверкой двоичных разрядов, старший из которых соответствует повышенной яркости цветовой комбинации RGB (I=1). Старший бит цветового атрибута M=1 устанавливает мерцающий режим отображения символа — с частотой примерно в 1 секунду символ то появляется на экране, то пропадает. По умолчанию для всех отображаемых символов действует байт атрибута с кодом $07, что соответствует выводу светло-серых символов на черном фоне.

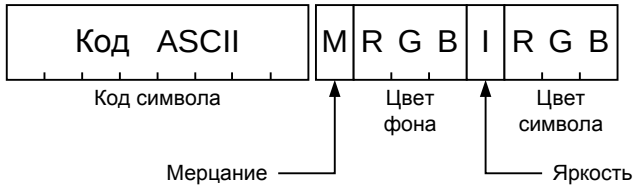


Рис. 12.1. Параметры знакоместа

Дополнительно с текстовой страницей связан набор данных, определяющих текущее положение курсора и его конфигурацию.

12.1. Окно вывода

По умолчанию программа может использовать для вывода результатов всю рабочую площадь окна консольного приложения — 25 строк по 80 символов в строке. Чтобы убедиться в этом, можно воспользоваться программой из листинга 12.1.

Листинг 12.1. Программа `MaxWindow`

```
program MaxWindow;
uses Crt;
var x,y: byte;
begin
  clrscr;
  for y:=0 to 23 do
    for x:=0 to 79 do
      begin
        gotoxy(x+1,y+1);
        write((x+y+1) mod 10);
      end;
  readln;
end.
```

После ее предварительного запуска необходимо привести в соответствие размер буфера, выделяемого операционной системой консольному приложению (ширина — 80, высота — 300), и размер окна приложения (ширина — 80, высота — 25). Чтобы попасть в диалоговое окно Windows, приведенное на рис. 12.2, вы должны зайти в системное меню окна приложения (рис. 12.3) и выполнить команду **Свойства**.

После приведения указанных размеров в соответствие результат работы программы `MaxWindow` будет выглядеть так, как показано на рис. 12.4.

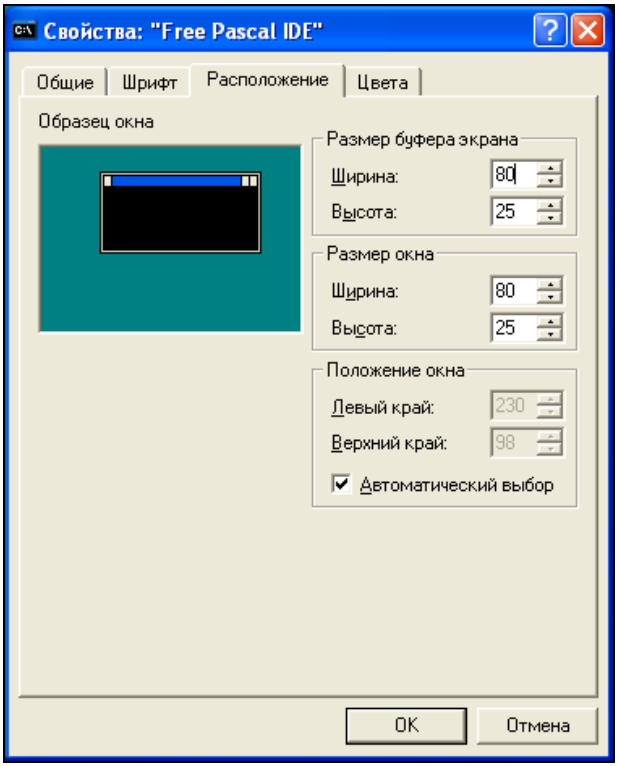


Рис. 12.2. Согласование буфера и окна по высоте

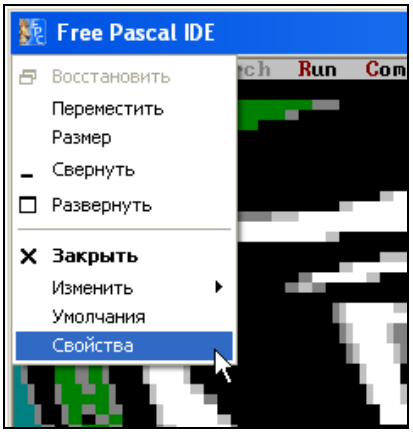


Рис. 12.3. Системное меню окна

Система координат с дискретными значениями *x* и *y*, используемая для идентификации знакомест на "текстовом" экране, устроена следующим образом. *Ось x* направлена вправо, и номера колонок отсчитываются от 1 до 80. *Ось y* направлена вниз, и номера строк отсчитываются от 1 до 25.

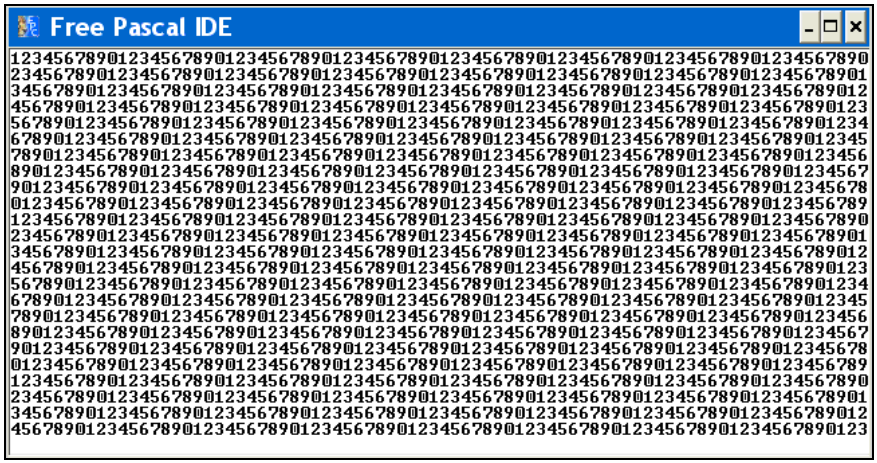


Рис. 12.4. Максимальное заполнение площади вывода

Продолжение вывода после заполнения 25-й строки приводит к подъему содержимого окна вывода и выталкиванию верхних строк за пределы зоны видимости. Именно так все происходило под управлением MS-DOS. Однако Windows предлагает консольному приложению буфер вывода, содержащий 300 строк, поэтому по мере вывода в пределах указанного объема выталкиваемые вверх результаты работы программы не пропадают. Используя полосу вертикальной прокрутки, мы можем вернуться к ранее выданным результатам. При запуске программы на исполнение Windows открывает окно приложения, пользуясь своими системными параметрами (ширина, высота, положение на экране дисплея). Довольно часто ширина окна может оказаться меньше 80 символов, и тогда при выводе длинных строк будут возникать неожиданные переносы в следующую строку. Для предотвращения нежелательных последствий программист может включить в свою программу строку вида:

```
WindMaxX:=80;
```

Габариты окна вывода программы можно уменьшить двумя способами. Во-первых, можно изменить значения системных переменных `WindMaxX` и `WindMaxY`. Во-вторых, в области окна вывода с помощью процедуры `Window` можно установить неподвижное окно меньшего размера (листинг 12.2).

Листинг 12.2. Программа `LowWindow`

```
Program LowWindow;
uses Crt;
var x,y: byte;
begin
  clrscr;
  for y:=0 to 23 do
```

```
    for x:=0 to 79 do
      begin
        gotoxy(x+1,y+1);
        write((x+y+1) mod 10);
      end;
  window(20,5,55,10);
  clrscr;
  writeln;
  writeln(' Строка 2 в новом окне ');
  writeln(' Строка 3 в новом окне ');
  readln;
end.
```

Параметрами этой процедуры являются "координаты" знакомест в левом верхнем и правом нижнем углах нового окна вывода. И тогда результаты работы слегка модифицированной программы выглядят так, как это представлено на рис. 12.5.

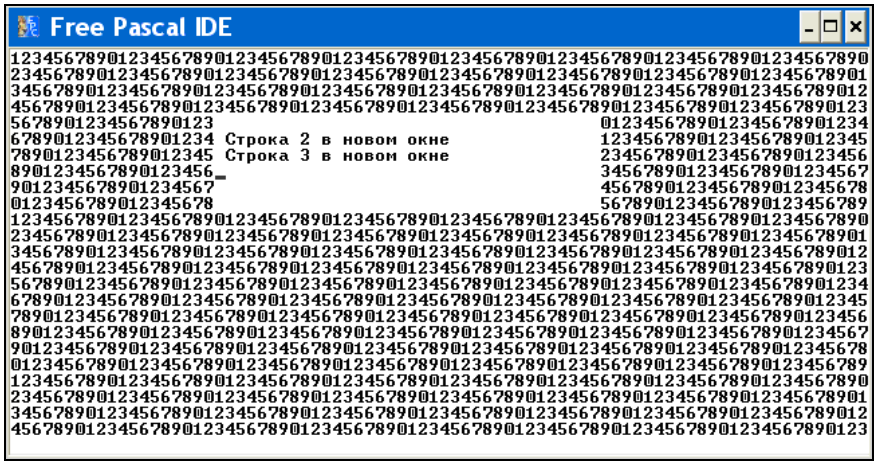


Рис. 12.5. Новое окно вывода с "координатами" углов (20, 5) и (55, 10)

Логика обслуживания маленького окна та же самая. В нем действует своя, локальная система координат, операция очистки экрана (`ClrScr`) распространяет свое действие только на поле нового экрана, не затрагивая ранее выданную окружающую информацию. По мере заполнения маленького окна его верхние строки выталкиваются за пределы зоны видимости (и пропадают). На большом экране можно завести не одно такое окно, но в каждый момент времени активным (т. е. готовым к отображению выводимой информации) будет окно, объявленное последним.

Кроме очистки экрана или действующего окна вывода в двух приведенных программах продемонстрировано перемещение курсора текстового окна с помощью процедуры `GoToXY`. Текущая позиция курсора определяет позицию, с которой

начинается вывод очередных результатов. По мере поступления очередного символа выводимых данных курсор смещается вправо, указывая следующую свободную позицию. Если курсор вынужден сместиться за пределы, установленные значением переменной `WndMaxX`, то он автоматически переходит в начало следующей строки.

Программа имеет возможность опросить текущее местоположение курсора с помощью двух следующих функций без параметров (их бы правильнее называть системными переменными, доступными только для чтения):

- `WhereX` — опрос координаты `x`;
- `WhereY` — опрос координаты `y`.

С их помощью, например, можно осуществить смещение курсора в текущей строке на заданное количество позиций влево или вправо:

```
GoToXY(WhereX+5,WhereY);
```

На конфигурацию курсора и его видимость оказывают влияние три следующие процедуры без параметров:

- `CursorBig` — превращает курсор из горизонтальной черточки в прямоугольник;
- `CursorOff` — отключает видимость курсора;
- `CursorOn` — включает видимость курсора и восстанавливает его стандартный вид.

12.2. Управление атрибутами отображаемого текста

При записи очередного символа в окно вывода (а точнее, в соответствующую позицию видеопамяти) к коду ASCII присоединяется байт атрибута, который хранится в системной переменной `TextAttr`. Мы уже упоминали, что по умолчанию в ней находится код `$07`, что соответствует следующей комбинации битов атрибута:

- биты цвета контура (IRGB) = 0111 (серый цвет, не яркий);
- биты цвета фона (RGB) = 000 (черный цвет);
- бит мерцания = 0 (режим мерцания отключен).

Содержимое бита повышенной яркости (I) можно изменить с помощью одной из следующих процедур без параметров:

- `HighVideo` — в бит I заносится 1 (признак повышенной яркости);
- `LowVideo` — в бит I заносится 0 (признак обычной яркости);
- `NormVideo` — восстанавливается бит яркости, установленный перед запуском программы.

Для изменения цвета контуров символа используется процедура `TextColor`. Ее единственным аргументом является код цветности, замещающий биты IRGB в системной переменной `TextAttr`. Значение этого кода должно принадлежать диапазону [0, 15]. На практике вместо числового кода обычно задается одна из мнемонических констант, список которых приведен в табл. 12.1.

Таблица 12.1

Код	Константа	Цвет	Код	Константа	Цвет
0	`Black`	Черный	8	`DarkGray`	Темно-серый
1	`Blue`	Синий	9	`LightBlue`	Светло-синий
2	`Green`	Зеленый	10	`LightGreen`	Светло-зеленый
3	`Cyan`	Бирюзовый	11	`LightCyan`	Светло-бирюзовый
4	`Red`	Красный	12	`LightRed`	Светло-красный
5	`Magenta`	Фиолетовый	13	`LightMagenta`	Светло-фиолетовый
6	`Brown`	Коричневый	14	`Yellow`	Желтый
7	`LightGray`	Светло-серый	15	`White`	Белый

Для изменения цвета фона предназначена процедура `TextBackground`, использующая в качестве своего единственного аргумента код из диапазона [0, 15]. Три младших двоичных разряда кода определяют цвет фона, а старший разряд замещает бит мерцания (M). Фактический аргумент процедуры `TextBackground` может быть задан одной из мнемонических констант из табл. 12.1.

12.3. Разное

Список остальных процедур и функций модуля `Crt` приведен в табл. 12.2

Таблица 12.2

Формат обращения	Выполняемое действие
Модификация содержимого окна вывода	
`ClrScr;`	Очистка текущего окна вывода с переводом курсора в левый верхний угол (аббревиатура от *Clear Screen*)
`ClrEol;`	Очистка конца текущей строки, начиная с позиции курсора (аббревиатура от *Clear to End Of Line*)
`DelLine;`	Удаление текущей строки с подъемом строк, расположенных ниже (аббревиатура от *Delete Line*)
`InsLine;`	Вставка пустой строки с подъемом текущей строки и всех расположенных выше (аббревиатура от *Insert Line*)
Работа с клавиатурой	
`bv:=KeyPressed;` `if KeyPressed then...`	Анализ — была ли нажата какая-либо клавиша к моменту вызова функции `KeyPressed`. Если такое событие имело место, то функция возвращает значение `True`
`cv:=ReadKey;`	Чтение кода нажатой клавиши (тип `cv` — `char`)

Таблица 12.2 *(окончание)*

Формат обращения	Выполняемое действие
Аудиопроцедуры	
`Sound(f);`	Выдача звукового сигнала
`Delay(n);`	Задержка на `n` миллисекунд
`Nosound;`	Отмена звукового сигнала
Переназначение вывода в файл	
`AssignCRT(vf);`	`vf` — файловая переменная типа `Text`

Процедуры `ClrEol`, `DelLine` и `InsLine` могут использоваться для программирования некоторых операций текстового редактора.

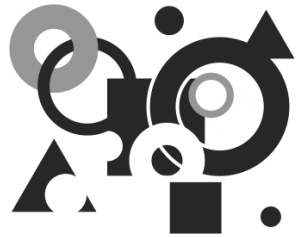
После старта приложения пользователь может нажимать клавиши, не дожидаясь приглашения со стороны программы. Коды нажимаемых клавиш накапливаются в буфере клавиатуры до тех пор, пока приложение не обратится к одной из подпрограмм ввода — `read`, `readln` или `ReadKey`. Если две первые процедуры завершают свою работу только после нажатия клавиши <Enter>, то функция `ReadKey` в таком нажатии не нуждается. Если буфер клавиатуры к моменту выполнения функции `ReadKey` не пуст, то из него извлекается код первого символа. Если этот код соответствует обычному отображаемому символу (т. е. его коду ASCII, отличному от нуля), то он и возвращается в качестве значения функции. Если на выходе буфера клавиатуры находилась пара байтов, соответствующая специальной клавише, то первым будет извлечен нулевой байт, который также будет возвращен в качестве значения функции `ReadKey`. В этом случае надо повторно обратиться к функции `ReadKey` для считывания скан-кода специальной клавиши. Считываемый код из буфера клавиатуры удаляется. Если в момент вызова функции `ReadKey` буфер клавиатуры пуст, то действие программы приостанавливается до тех пор, пока не будет нажата какая-либо клавиша.

В отличие от описанной схемы ввода с возможным ожиданием нажатия клавиши функция `KeyPressed` завершает свою работу без такого ожидания. Она возвращает значение `False`, если в момент ее выполнения буфер клавиатуры пуст, и значение `True` в противном случае. Содержимое буфера клавиатуры при этом не изменяется.

Под управлением Windows функции запуска генератора звука (`Sound`) и его остановки (`NoSound`) выполняются не так, как это происходило под управлением MS-DOS. Раньше генератор начинал издавать непрерывный звук частоты `f` (в герцах) и остановить его можно было спустя некоторое время, регулируемое с помощью задержки (`Delay`). Под управлением Windows аргумент процедуры `Sound` игнорируется, и вместо звука определенной тональности раздается системный звук, по умолчанию напоминающий щелчок по барабану. Поэтому обращение к процедуре `NoSound`, по большому счету, лишено смысла.

Возможностью программного перенаправления выводимых данных в файл (процедура `AssignCRT`) пользуются достаточно редко, т. к. это же действие обеспечивается заданием соответствующего параметра в командной строке.

Глава 13



Библиотечные функции и процедуры

13.1. Модуль *System*

Базовый набор для вычисления элементарных функций (табл. 13.1) и выполнения некоторых вспомогательных операций (табл. 13.2) входит в состав модуля `System`, на автомате подключаемого к каждой программе.

Таблица 13.1. *Элементарные функции модуля* `System`

Функция	Значение	Функция	Значение
`abs(x)`	$\|x\|$	`pi`	π
`arctan(x)`	$\mathrm{arctg}(x)$	`sin(x)`	$\sin(x)$
`cos(x)`	$\cos(x)$	`sqr(x)`	x^2
`exp(x)`	e^x	`sqrt(x)`	$\sqrt{x}$
`ln(x)`	$\log_e(x)$		

Таблица 13.2. *Вспомогательные функции и процедуры модуля* `System`

Формат вызова	Назначение
`dec(i[,di]);`	Аналог оператора `i:=i-1;` или `i:=i-di;`
`v:=frac(x);`	Выделение дробной части `x` (тип результата = типу `x`)
`j:=hi(i);`	Выделение старшего байта, слова или двойного слова в зависимости от типа аргумента
`inc(i[,di]);`	Аналог оператора `i:=i+1;` или `i:=i+di;`

Таблица 13.2 *(окончание)*

Формат вызова	Назначение
`v:=int(x);`	Выделение целой части (тип результата равен типу `x`)
`j:=lo(i);`	Выделение младшего байта, слова или двойного слова в зависимости от типа аргумента
`b:=odd(x);`	`Result = True`, если целочисленное значение `x` — нечетно
`v:=power(x,y);`	Вычисление x^y (`x`, `y` — вещественные)
`i:=random(N);`	Генерация целого случайного числа из диапазона [0, N – 1]
`v:=random;`	Генерация вещественного случайного числа из интервала [0, 1)
`i:=round(x);`	Округление `x` до целого (тип результата `int64`)
`j:=swap(i);`	Перестановка старших и младших разрядов `i` (байтов, слов или двойных слов в зависимости от типа аргумента)
`j:=trunc(x);`	Выделение целой части `x` (тип результата `int64`)

Назначение элементарных функций, приведенных в табл. 13.21, в особых комментариях не нуждается. Нужно только не забывать, что аргументы тригонометрических функций задаются в радианах. Главное значение арктангенса также получается в радианах и принадлежит диапазону $[-\pi/2, \pi/2]$.

В некотором пояснении нуждается функция округления вещественных чисел до ближайшего целого — `round(x)`. В ее реализации использовано так называемое *банковское округление*. Его специфика заключается в том, что округление аргументов, дробная часть которых равна 0.5, производится *к ближайшему четному* числу. Например:

```
round(1.5)=2
round(-1.5)=-2
round(2.5)=2
round(-2.5)=-2
```

Такой способ имеет смысл при массовом суммировании слагаемых с точностью до целого. В этом случае систематическая суммарная ошибка округления не накапливается и, в среднем, не превышает по модулю единицы.

Но в вычислительной математике более предпочтительным считается следующий вариант округления. Если дробная часть вещественного числа строго меньше, чем 0.5, то она отбрасывается. В противном случае к модулю целой части добавляется 1. Именно такой алгоритм реализован в функции `round`, включенной в состав известного пакета технических вычислений MATLAB:

```
round(1.0)=round(1.2)=1
round(1.5)=round(1.8)=2
round(-1.0)=round(-1.2)=-1
```

```
round(-1.5)=round(-1.8)=-2
round(2.5)=3
round(-2.5)=-3
```

Для начинающих программистов довольно сложно привыкнуть к просмотру оперативной памяти в шестнадцатеричном формате. Дело в том, что порядок байтов на экране определяется возрастанием их адресов — слева находятся байты с младшими адресами, а справа — байты со старшими адресами. Но те двоичные числа, которые мы пишем на бумаге, расположены по старшинству разрядов в обратном порядке — слева расположены старшие биты (и байты тоже), а справа — младшие. В технической литературе даже появились специальные термины — *big-endian byte ordering* (старший байт в конце) и *little-endian byte ordering* (младший байт в конце). На разных платформах числовые данные могут использовать прямой или обратный порядок следования байтов. Поэтому при обмене информацией иногда приходится менять порядок следования байтов в числовых данных. Для этой цели предназначены функции `lo(x)`, `hi(x)` и `swap(x)`. Чтобы лучше представить себе их специфику, приведем следующую программу (листинг 13.1).

Листинг 13.1. Программа `big_lit`

```
program big_lit;
var
  a2: array [1..2] of byte=(1,2);
  w: word absolute a2;
  a4: array [1..4] of byte=(1,2,3,4);
  i: integer absolute a4;
  a8: array [1..8] of byte=(1,2,3,4,5,6,7,8);
  qw: QWord absolute a8;
  j: integer;
begin
  writeln;
  writeln('lo(w)=',lo(w));
  writeln('hi(w)=',hi(w));
  for j:=1 to 2 do write(a2[j]:4);
  writeln;
  w:=swap(w);    // перестановка байтов
  writeln('lo(w)=',lo(w));
  writeln('hi(w)=',hi(w));
  for j:=1 to 2 do write(a2[j]:4);
  writeln;
  writeln('lo(i)=',lo(i));
  writeln('hi(i)=',hi(i));
  for j:=1 to 4 do write(a4[j]:4);
```

```
  writeln;
  i:=swap(i);    // перестановка слов
  writeln('lo(i)=',lo(i));
  writeln('hi(i)=',hi(i));
  for j:=1 to 4 do write(a4[j]:4);
  writeln;
  writeln('lo(qw)=',lo(qw));
  writeln('hi(qw)=',hi(qw));
  for j:=1 to 8 do write(a8[j]:4);
  writeln;
  qw:=swap(qw);  // перестановка двойных слов
  writeln('lo(qw)=',lo(qw));
  writeln('hi(qw)=',hi(qw));
  for j:=1 to 8 do write(a8[j]:4);
  writeln;
  readln;
end.
```

Указание `absolute` заставляет компилятор размещать с одного и того же байта оперативной памяти массивы `a2`, `a4`, `a8` и соответствующие переменные такой же длины — `w`, `i`, `qw`. Это дает нам возможность вывести содержимое каждого байта той или иной переменной до и после перестановки младшей и старшей половин чисел:

```
Running "c:\fpc\2.2.4\bin\i386-win32\big_lit.exe "
lo(w)=1
hi(w)=2
   1   2           // до перестановки
lo(w)=2
hi(w)=1
   2   1           // после перестановки
lo(i)=513
hi(i)=1027
   1   2   3   4   // до перестановки
lo(i)=1027
hi(i)=513
   3   4   1   2   // после перестановки
lo(qw)=67305985
hi(qw)=134678021
   1   2   3   4   5   6   7   8   // до перестановки
lo(qw)=134678021
hi(qw)=67305985
   5   6   7   8   1   2   3   4   // после перестановки
```

13.2. Модуль *Math*

Дополнительный набор подпрограмм вычисления элементарных (табл. 13.3) и специальных (табл. 13.4) функций сосредоточен в модуле `Math`.

Таблица 13.3. *Элементарные функции и процедуры модуля* `Math`

Формат вызова	Значение	Формат вызова	Значение
`v:=arccos(x);`	$\arccos(x)$	`v:=lnexp1(x);`	$\ln(x+1)$
`v:=arccosh(x);` `v:=arcosh(x);`	$\operatorname{arch}(x)$	`v:=log10(x);`	$\log_{10}(x)$
`v:=arcsin(x);`	$\arcsin(x)$	`v:=log2(x);`	$\log_2(x)$
`v:=arcsinh(x);` `v:=arsinh(x);`	$\operatorname{arsh}(x)$	`v:=logn(x,y);`	$\log_y(x)$
`v:=arctan2(x,y);`	$\operatorname{tg}(v)=\frac{y}{x}$	`v:=max(x,y);` `v:=min(x,y);`	Тип `x`, `y` — `integer`, `int64` или `extended`
`v:=arctanh(x);` `v:=artanh(x);`	$\operatorname{arth}(x)$	`v:=power(x,y);`	x^y
`v:=cosecant(x);` `v:=csc(x);`	$\frac{1}{\sin(x)}$	`v:=sec(x);` `v:=secant(x);`	$\frac{1}{\cos(x)}$

`v:=cosh(x)`	$\frac{e^x+e^{-x}}{2}$	`v:=sign(x);`	`v=-1`, если `x<0`
			`v=0`, если `x=0`
			`v=+1`, если `x>0`

`v:=cot(x);` `v:=cotan(x);`	$\frac{\cos(x)}{\sin(x)}$	`sincos(x,s,c);`	$s=\sin(x)$ $c=\cos(x)$
`v:=hypot(a,b);`	$\sqrt{a^2+b^2}$	`v:=sinh(x);`	$\frac{e^x-e^{-x}}{2}$
`v:=intpower(x,i);`	x^i (x — `integer`)	`v:=tan(x);`	$\frac{\sin(x)}{\cos(x)}$
`v:=ldexp(x,i);`	$x\times 2^i$	`v:=tanh(x);`	$\frac{e^x-e^{-x}}{e^x+e^{-x}}$

Обратите внимание на процедуру `sincos`. За счет использования команд сопроцессора фирмы Intel эта процедура выполняется быстрее, чем два последовательных обращения к функциям `sin` и `cos`.

Таблица 13.4. *Вспомогательные функции и процедуры модуля* `Math`

Формат вызова	Выполняемое действие
`i:=ceil(x);`	Округление к большему ближайшему целому

`i:=CompareValue(x,y);`	`i =-1`, если `x<y`; (`x,y:integer,int64,extended`)
	`i = 0`, если `x=y`
	`i = 1`, если `x>y`

`DivMod(k1, k2, d, r);`	`d:=k1 div k2;` `r:=k1 mod k2;`

`j:=EnsureRange(i,min,max);`	`j:=i` если `min ≤ i ≤max`
	`j:=min` если `i<min`
	`j:=max` если `i>max`

`i:=floor(x);`	Округление к ближайшему меньшему целому
`FrExp(x,fr,exp);`	Выделение дробной части (`fr`) и порядка (`exp`) вещественного значения `x`
`v:=IfThen(be, et, ef);`	`if be then v:=et else v:=ef;` Оба типа `et` и `ef` — `integer`, `int64`, `double` или `string`
`bv:=InRange(i,min,max);`	`bv:=True`, если `min ≤ i ≤max`, иначе `bv:=False`. Тип `i` — `integer` или `int64`
`bv:=IsInfinity(x);`	`bv:=True`, если `x=1/0` (`Infinity = +∞`)
`bv:=IsNaN(x);`	`bv:=True`, если `x=0/0` (`NaN` — Not a Number)
`bv:=IsZero(x);`	`bv:=True`, если `x=0`
`y:=RoundTo(x,k);`	Округление `x` до `k`-й десятичной цифры (`k≥0` — в целой части, `k<0` — в дробной части)
`bv:=SameValue(x,y);`	`bv:=True`, если `x≡y` (оба — `extended`)
`bv:=SameValue(x,y,eps)`	`bv:=True`, если `|x - y| ≤ eps`
`y:=SimpleRoundTo(x,k);`	То же, что и `RoundTo`

К операции банковского округления (round) в модуле Math добавились еще две операции округления, используемые в коммерции: округление к ближайшему большему целому *(округление в пользу продавца)* и округление к ближайшему меньшему целому *(округление в пользу покупателя)*:

```
ceil ( 1.1) = 2
ceil (-1.1) =-1
ceil ( 1.8) = 2
ceil (-1.8) =-1
floor( 1.1) = 1
floor(-1.1) =-2
floor( 1.8) = 1
floor(-1.8) =-2
```

Две дополнительные функции округления, разницу между возвращаемыми результатами у которых мы не обнаружили, позволяют произвести округление с точностью до заданной десятичной цифры:

```
RoundTo(15849.15849, 0)=15849.0000000000
RoundTo(15849.15849, 1)=15850.0000000000
RoundTo(15849.15849, 2)=15800.0000000000
RoundTo(15849.15849, 3)=16000.0000000000
RoundTo(15849.15849,-1)=15849.2000000000
RoundTo(15849.15849,-2)=15849.1600000000
RoundTo(15849.15849,-4)=15849.1585000000
```

Кроме упомянутых выше элементарных функций в состав модуля Math включены и другие подпрограммы, не так часто упоминаемые в учебниках по программированию.

Одну из таких групп составляют подпрограммы обработки целочисленных и вещественных векторов (табл. 13.5). Их аргументом является либо открытый массив соответствующего типа (например, A), либо указатель на массив (например, pA) в сочетании с количеством N обрабатываемых элементов.

Таблица 13.5

Формат вызова	Выполняемое действие
`k:=MaxIntValue(A);`	Поиск максимального значения в целочисленном массиве
`v:=MaxValue(A);` `v:=MaxValue(pA,N);`	Поиск максимального значения в целочисленном или вещественном массиве
`k:=MinIntValue(A);`	Поиск минимального значения в целочисленном массиве
`v:=MinValue(A);` `v:=MinValue(pA,N);`	Поиск минимального значения в целочисленном или вещественном массиве
`v:=norm(A);` `v:=norm(pA,N);`	Вычисление нормы $\sqrt{a_1^2 + a_2^2 + ... + a_n^2}$

13.2.1. Преобразования угловых величин

Кроме широко используемых *градусов* (degrees) и *радиан* (radians) в различных разделах науки и техники применяют и другие единицы измерения углов. Среди них чаще других встречаются *грады* (grads) и *обороты* (cycles). Град или *метрический градус* был введен в обиход в конце XVIII века во Франции при переходе на метрическую систему мер. Было принято, что в прямом угле содержится 100 градов. В один град входило 100 метрических минут, каждая метрическая минута содержала 100 метрических секунд. Метрические единицы измерения углов упрощали вычисления, связанные с артиллерийскими расчетами, и до сих пор в технических характеристиках наземных и морских орудий можно встретить значения минимальных и максимальных углов наклона ствола в градах. В инженерном формате Калькулятора Windows один из переключателей задания углов — **Грады**. Обороты или *циклы* используются в механике и физике для измерения углов и фаз. Один оборот соответствует повороту на 360°.

В состав модуля `Math` включены 8 функций (табл. 13.6), обеспечивающих преобразования этих угловых величин, хотя их содержательная часть обычно сводится к одной-двум арифметическим операциям. Аргументом всех этих функций может быть любое целое или вещественное значение. Возвращаемый результат имеет тип `extended`.

Таблица 13.6

Имя функции	Выполняемое преобразование
`CycleToRad(x)`	Обороты в радианы
`DegToGrad(x)`	Градусы в грады
`DegToRad(x)`	Градусы в радианы
`GradToDeg(x)`	Грады в градусы
`GradToRad(x)`	Грады в радианы
`RadToCycle(x)`	Радианы в обороты
`RadToDeg(x)`	Радианы в градусы
`RadToGrad(x)`	Радианы в грады

В листинге 13.2 приводится пример программы, в которой вычисляются значения всех единичных углов для каждой системы измерения.

Листинг 13.2. Программа `angles`

```
program angles;
uses Math;
```

```
var
  c1: extended=1;
  c2: extended=90;
  c3: extended=360;
  c4: extended=400;
begin
  writeln;
  writeln('2*пи =':10,2*pi:22:18);
  writeln;
  writeln('1 град =':10,GradToDeg(1):22:18,' градуса');
  writeln('1 град =':10,c1/c4:22:18,' оборота');
  writeln('1 град =':10,GradToRad(1):22:18,' радиана');
  writeln;
  writeln('1 градус =':10,DegToGrad(1):22:18,' града');
  writeln('1 градус =':10,c1/c3:22:18,' оборота');
  writeln('1 градус =':10,DegToRad(1):22:18,' радиана');
  writeln;
  writeln('1 оборот =':10,c4:22:18,' градов');
  writeln('1 оборот =':10,CycleToRad(1):22:18,' радиан');
  writeln('1 оборот =':10,c3:22:18,' градусов');
  writeln;
  writeln('1 радиан =':10,RadToGrad(1):22:18,' града');
  writeln('1 радиан =':10,RadToDeg(1):22:18,' градуса');
  writeln('1 радиан =':10,RadToCycle(1):22:18,' оборота');
  readln;
end.
```

Результаты ее работы таковы:

```
Running "c:\fpc\2.2.4\bin\i386-win32\angles.exe "
    2*пи =   6.28318530717958650
  1 град =   0.90000000000000000 градуса
  1 град =   0.00250000000000000 оборота
  1 град =   0.01570796326794897 радиана
1 градус =   1.11111111111111110 града
1 градус =   0.00277777777777778 оборота
1 градус =   0.01745329251994330 радиана
1 оборот = 400.00000000000000000 градов
1 оборот =   6.28318530717958650 радиан
1 оборот = 360.00000000000000000 градусов
1 радиан =  63.66197723675813400 града
1 радиан =  57.29577951308232100 градуса
1 радиан =   0.15915494309189534 оборота
```

13.2.2. Процедуры и функции для статистики

Для поддержки статистической обработки результатов наблюдений в состав модуля `Math` включены 12 специальных функций и процедур (табл. 13.6). Для того чтобы не запутаться в близких по назначению подпрограммах, поясним некоторые термины из теории вероятностей.

- *Генеральная совокупность* — полная совокупность всех событий или явлений, характеристики которых являются предметом исследования.
- *Выборка* — сокращенный набор событий или явлений, извлеченный из генеральной совокупности. На практике считается, что минимальный объем выборки должен насчитывать не менее 30—35 наблюдений. Существует довольно много правил отбора, позволяющих сформировать репрезентативную выборку, достаточно точно отражающую тенденции общей совокупности.
- *Среднее* — среднее арифметическое значение наблюдаемой величины (по выборке или генеральной совокупности):

$$x_{\text{ср}} = \frac{1}{N}\sum_{i=1}^{N} x_i.$$

- *Стандартное отклонение* — среднеквадратическое отклонение наблюдаемых значений от их среднего. Для его вычисления используется одна из двух формул:

$$\sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - x_{\text{ср}})^2}; \qquad \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - x_{\text{ср}})^2}.$$

В теории вероятностей, изучающей свойства и закономерности случайных величин, различают *дискретные* и *непрерывные* случайные величины.

К первым, например, принадлежат события, связанные с количеством очков, выпадающих при бросании игральной кости. Различных событий здесь всего 6, но частота появления той или иной грани может быть обусловлена неоднородностью материала, из которого изготовлена кость, спецификой поверхности, на которую падает кость, и т. п. Проведя достаточно большое количество испытаний, мы можем определить частоты выпадения той или иной комбинации очков. Пусть m_i — частота появления грани с i очками, а N — общее количество испытаний. Для оценки приблизительного значения вероятности p_i каждого из шести событий служит отношение частоты их появления к общему числу испытаний $\frac{m_i}{N}$.

Для случайной величины x, принимающей дискретные значения x_1, x_2, ..., x_n с вероятностями p_1, p_2, ..., p_n, вводится понятие *среднего* или *математического ожидания*, которое вычисляется по формуле:

$$\mu = \sum_{i=1}^{n} x_i \cdot p_i.$$

Если вероятности появления любого значения одинаковы и равны $\frac{1}{n}$, то значение μ совпадает со среднеарифметическим значением, вычисляемым на практике.

Для непрерывной случайной величины *x*, которая может принимать любые значения на интервале $t \subset -\infty, +\infty$, вводится понятие *функции распределения* $F(t)$, значение которой равно вероятности появления значения *x*, не превышающего *t*. Для большого количества явлений природы, на которые оказывают влияние многие независимые факторы, характерно так называемое *нормальное распределение*. При этом график функции распределения имеет вид перевернутого колокола, максимум которого соответствует математическому ожиданию *x*. На рис. 13.1 представлен один из таких графиков для $\mu = 0$. Второй характеристикой "колокола" (наряду с его максимальной высотой) является ширина основания, пропорциональная крутизне его ветвей. С этой характеристикой связана так называемая *дисперсия* σ, значение которой достаточно хорошо аппроксимируется стандартным отклонением.

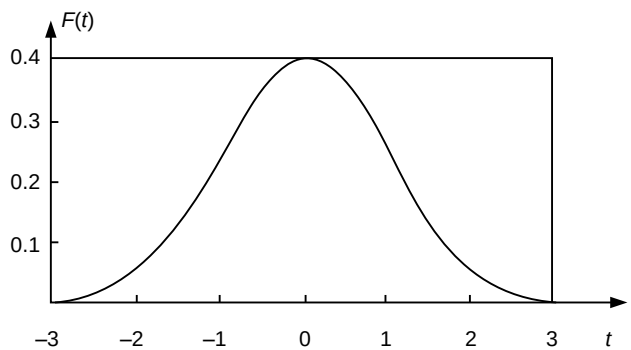


Рис. 13.1. Нормальное распределение ($\mu = 0$, $\sigma^2 = 1$)

Конечно, не все исследуемые явления подчиняются закону нормального распределения. И даже если их функция распределения напоминает колокол, то он может оказаться не очень симметричным, и тогда появляются некоторые дополнительные вероятностные характеристики — коэффициент асимметрии, коэффициенты крутизны и др.

После такого мини-экскурса в область терминологии статистических исследований вам станут понятнее действия, выполняемые с помощью довольно скудного набора подпрограмм, представленных в табл. 13.7.

Таблица 13.7

Имя подпрограммы	Назначение
`Mean`	Вычисление среднего по выборке
`MeanAndStdDev`	Вычисление среднего и стандартного отклонения по выборке

Таблица 13.7 *(окончание)*

Имя подпрограммы	Назначение
`MomentSkewKurtosis`	Вычисление первых моментов, коэффициентов асимметрии и эксцесса
`PopnStdDev`	Вычисление стандартного отклонения по генеральной совокупности
`PopnVariance`	Вычисление дисперсии по генеральной совокупности
`RandG`	Генерация случайных чисел с нормальным законом распределения вероятности по заданным среднему и стандартному отклонению
`StdDev`	Вычисление стандартного отклонения по выборке
`Sum`	Суммирование элементов массива
`SumOfSquares`	Суммирование квадратов элементов массива
`SumsAndSquares`	Вычисление квадратного корня из суммы элементов массива
`TotalVariance`	Вычисление общей (полной) дисперсии
`Variance`	Вычисление дисперсии по выборке

Главным аргументом всех подпрограмм (кроме `RandG`) является массив наблюдаемых значений типа `extended`, который может быть задан либо именем массива, например — `A`, либо указателем на массив типа `extended`, т. е. переменной типа `PExtended`. Во втором случае используется дополнительный целочисленный параметр `N`, определяющий количество обрабатываемых элементов массива.

У функции `Mean` других аргументов, кроме главного, нет. Она возвращает результат типа `extended`:

$$\text{Mean}(A) = \frac{1}{N}\sum_{i=1}^{N} A_i .$$

У процедуры `MeanAndStdDev` кроме главного аргумента присутствуют два дополнительных выходных параметра, например, с именами `M` и `SD`. В переменной `M` возвращается среднее, а в переменной `SD` — стандартное отклонение от среднего:

$$SD = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(A_i - M)^2}.$$

У процедуры `MomentSkewKurtosis` кроме главного аргумента присутствуют шесть дополнительных параметров — `m1`, `m2`, `m3`, `m4`, `skew`, `kurtosis`. Первые четыре задаются именами переменных типа `extended`, в которые записываются значения первых четырех центральных выборочных моментов:

$$m_k = \frac{1}{N}\sum_{i=1}^{N}(A_i - \text{Mean})^k, \quad k = 1, 2, 3, 4.$$

По определению значение первого момента `m1` равно 0. Три оставшиеся используются для вычисления таких вероятностных характеристик, как *коэффициент асимметрии* (`skew`) и мера концентрации распределения случайной величины вокруг среднего (`kurtosis`). В теории вероятностей последнюю величину называют *коэффициентом эксцесса*, и она характеризует *крутость* кривой распределения вокруг среднего:

$$\texttt{skew} = \frac{m_3}{\sqrt{m_2^3}};$$

$$\texttt{kurtosis} = \frac{m_4}{m_2^2}.$$

Функция `RandG(mean, stddev)` используется для генерации случайных чисел, распределенных по нормальному закону (закону Гаусса) вокруг заданного среднего (`mean`) с заданным стандартным отклонением от среднего (`stddev`). В руководстве по подпрограммам RTL приведен следующий пример (листинг 13.3).

Листинг 13.3. Программа `Rand_G`

```
program Rand_G;
uses Math ;
var
  i : integer;
  A : array[1..10000] of extended;
  Mean, StdDev : extended;
begin
  randomize;
  for i:= 1 to 10000 do
    A[i] := Randg(1,0.2);         // заполнение массива по закону Гаусса
  MeanAndStdDev(A,Mean,StdDev);
  writeln('Mean : ',Mean:8:4);    // вывод фактического значения среднего
  writeln('StdDev : ',StdDev:8:4); // вывод стандартного отклонения
  readln;
end.
```

Четырехкратный запуск этого примера дал следующие результаты:

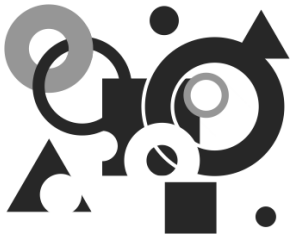
```
.................
Mean :     0.9992
StdDev :   0.1974
.................
Mean :     0.9986
StdDev :   0.1979
.................
```

```
Mean :     1.0027
StdDev :   0.1989
..................
Mean :     0.9977
StdDev :   0.2001
```

По результатам видно, что формируемый массив действительно содержит элементы, характеризующиеся достаточно близкими значениями среднего (`Mean ≈ 1`) и стандартного отклонения (`StdDev ≈ 0.2`). Но абсолютного повторения результатов при этом не наблюдается. Причина кроется в том, что для генерации случайных чисел с нормальным законом распределения используется стандартная функция `random`. А ее начальные параметры обновляются при каждом запуске за счет применения функции `randomize`.

Процедура `SumsAndSquares(A,sum,sum_sq)` вычисляет сумму значений элементов массива `A` и заносит ее в переменную `sum`. В переменную `sum_sq` заносится сумма квадратов значений элементов массива `A`.

Глава 14



Календари, даты, время

14.1. Немного истории

Термин *"календарь"* обязан своим происхождением латинским словам calendae (первый день месяца) и calendarium (долговая книга). В Древнем Риме на первое число каждого месяца приходилось время уплаты процентов по долгам.

Наиболее стабильным длительным промежутком времени с точки зрения ученых, ведущих астрономические наблюдения, является тропический год — интервал между моментами прохождения центра Солнца через так называемую точку весеннего или осеннего равноденствия. Наблюдается это явление 21 марта и 23 сентября в обычный год и со смещением на один день — в високосный год. В этот момент лучи Солнца падают на экватор отвесно и продолжительности ночи и дня совпадают. Длительность тропического года составляет 365 суток 5 час 48 мин и примерно 46 секунд (365.2422 солнечных суток, длительность последних — 24 часа).

Древние календари, в большинстве своем, базировались на фазах Луны. Так, например, в Вавилоне год состоял из 12 лунных месяцев с чередующейся длительностью месяцев 29→30→29→30→... Однако суммарная длительность такого года составляла 354 дня.

Один из первых солнечных календарей был принят в Египте примерно в четвертом тысячелетии до новой эры. По египетскому календарю год состоял из 365 дней и включал 12 месяцев по 30 дней. В конце года 5 дней объявлялись праздничными и не входили ни в один из месяцев.

Наиболее близким к тропическому году стал римский календарь, принятый по указу императора Юлия Цезаря в 47 г. до н. э. Для ликвидации разницы с астрономическим годом длительность 46 года была установлена в 432 дня. После этого вводились три года по 365 дней, за которыми следовал 1 високосный год. Таким образом, средняя продолжительность римского года достигла 365.25 суток, что немного превышало длительность тропического года. Тем не менее за 1500 лет набежало порядка 10 лишних суток.

За очередное улучшение календаря принялся ватиканский ученый Луиджи Лилио. В 1582 г. по указу (булле) папы Григория XIII был принят новый календарь, переход на который в разных странах затянулся на 200—300 лет. В России, например, новый календарь был введен декретом СНК РСФСР в январе 1918 г. (после

31 января наступило 14 февраля, день недели при этом сохранялся автоматически). Даты Римского календаря стали называть "старым стилем", а даты григорианского календаря — "новым стилем". Суть Ватиканской реформы заключалась в следующем. После четверга 4 октября 1582 г. следующим днем объявлялась пятница 15 октября. Так устранялись набежавшие излишки. Второе новшество заключалось в том, что из круглых дат столетий исключались високосные годы, которые не делились нацело на 400. При этом средняя длительность года по григорианскому календарю оказалась длиннее тропического года всего на 26 секунд. Так что разница в 1 сутки сможет набежать примерно к 4860 году.

Обратите внимание на специфику перехода от летоисчисления до новой эры. Первый год новой эры (эры Дионисия), последовавший после первого года до новой эры, отсчитывается от даты рождества Христова. С точки зрения математики, ему должен предшествовать 0-й год, которого не было. Поэтому в алгоритмах исчисления непрерывных календарных дат номера лет до новой эры следует считать отрицательными и увеличивать их на 1.

В конце того же XVI века астроном Джозеф Скалигер предложил новый способ отсчета времени. Начинался он от некоторого условного нуля (12 часов дня всемирного времени в понедельник 1 января 4713 г. до н. э., а с учетом приведенного выше замечания начальный год равен −4712) и представлялся в виде постоянно работающего таймера, отсчитывающего сутки и текущее время с шагом 10^{-5}. Целая часть юлианской даты JD (она была так названа в честь отца Скалигера) определяла количество суток, прошедших от начала отсчета, а дробная часть от 0.0 до 0.99999 соответствовала показанию суточных часов. Переход к следующей единице отсчета происходил в 12 часов очередного дня.

Пример приведен в табл. 14.1.

Таблица 14.1

Григорианская дата	**Юлианская дата**
1 января 4712 г. до н. э. в 12 часов дня	JD = 0.00000
2 января 4712 г. до н. э. в 12 часов дня	JD = 1.00000
3 января 4712 г. до н. э. в 18 часов вечера	JD = 2.25000
1 января 1900 г. в 12 часов дня	JD = 2415021.0
22 июня 1941 г. в 4 часа в утра	JD = 2430167.66667
9 мая 1945 г. в 12 часов дня	JD = 2431585.0
1 января 1999 г. в 12 часов дня	JD = 2451180.0
1 января 2000 г. в 12 часов дня	JD = 2451545.0

Следует заметить, что предложенная идея отсчета времени используется не только астрономами. В системах визуального программирования Borland C++Builder

и Delphi появился класс `TDateTime`, в объектах которого хранится обобщенное значение даты и времени в формате вещественного числа с двойной точностью (`double`). Его целая часть равна количеству дней, прошедших с полуночи 30 декабря 1899 г., а дробная часть соответствует времени дня. По сравнению с юлианской датой изменилась только точка начала отсчета.

Хранение даты и времени в формате юлианского дня представляется достаточно экономичным. Потребуется всего 8 байт для величины типа `double`, тогда как для запоминания символьной строки вида "YYYY/MM/DD HH:MM:SS" необходимо не менее 20 байт. Непрерывный таймер Дж. Скалигера удобен еще и тем, что позволяет очень просто вычислять различные временные интервалы как с точностью до суток, прошедших между двумя календарными датами, так и с точностью до секунд и даже долей секунды.

Итак, цивилизованный мир сегодня живет по григорианскому календарю, для которого типичны следующие количественные характеристики:

- обычный год насчитывает 365 суток (дней) и содержит 12 месяцев, каждому из которых присваивается индивидуальное имя и уникальный порядковый номер от 1 до 12;
- длительность месяцев обычного года (в сутках) описывается последовательностью 31 → 28 → 31 → 30 → 31 → 30 → 31 → 31 →30 → 31 → 30 → 31;
- сутки измеряются часами, минутами и секундами от 00:00:00 до 23:59:59;
- каждые 7 суток, прошедшие от начала года, составляют неделю, имеют индивидуальные имена и уникальные порядковые номера от 1 до 7 (стандарт ISO);
- високосным считается год, номер которого без остатка делится на 4 или делится на 100, но не делится без остатка на 400. В этом году на один день больше, и этот дополнительный день увеличивает длительность второго месяца года.

В быстротечных компьютерных событиях приходится прибегать и к более коротким единицам измерения времени — тысячным и миллионным долям секунды — миллисекундам и микросекундам.

14.2. Модуль *DateUtils*

В модуле `DateUtils` представлена очень внушительная коллекция, насчитывающая более 150 процедур и функций, обеспечивающих обработку календарных дат и интервалов времени. Порядка 25 подпрограмм из числа наиболее используемых включены и в состав модуля `SysUtils`. Несколько устаревших функций сохранены в модуле `Dos`.

Мы уже упоминали, что большинство процедур обработки интервалов времени связано с представлением текущего момента времени в формате `TDateTime`. По существу такое значение совпадает с модифицированной величиной юлианской даты. При этом целая часть вещественного значения типа `TDateTime` равна количеству дней, прошедших с полуночи 30 декабря 1899 г., а дробная часть соответствует времени дня. Так как количество дней за две-три сотни лет, отсчитанных после

1899 г., не превышает миллиона, то в целой части может быть задействовано не более шести цифр. Оставшиеся 10 значащих цифр значения типа `double` могут быть использованы для дробной части. Поэтому ошибка в представлении времени суток в формате `TDateTime` не превышает 10^{-5} секунд, т. е. 10 микросекунд. Именно этим объясняется наличие трех дробных цифр в значении секунд при выводе наиболее полного значения времени:

```
21.11.2009 21:42:20.796
```

Количество процедур и функций в модуле `DateUtils` довольно велико. В значительной мере на это обстоятельство повлияли два следующих фактора. Во-первых, сказался выбор формата упакованных данных. Довольно много процедур обеспечивают формирование, модификацию и извлечение соответствующих компонентов даты и времени. Во-вторых, на количестве процедур сказалось и разнообразие единиц измерения интервалов времени — годы, месяцы, недели, дни, часы, минуты, секунды и миллисекунды.

14.2.1. Ввод и вывод данных формата *TDateTime*

В системе Free Pascal каждая составляющая значения календарной даты и времени на нижнем уровне (год, месяц, день, часы, минуты и т. д.) представлена целочисленным значением типа `Word`. Поэтому с автономным вводом или выводом каждой из них никаких проблем не возникает. Однако это не самый оптимальный способ, т. к. в обыденной жизни мы пользуемся рядом более удобных способов символьной записи дат и времени. Вот лишь некоторые из них:

- 17 июля 2000 г. — подробная запись даты;
- 17.07.2000 — компактная запись даты;
- 17/07/2000 или 17-07-2000 — варианты компактной записи с другими разделителями;
- 17.07.00 — сокращенный вариант компактной записи;
- 12 часов 35 минут — подробная запись времени;
- 12:35 — компактная запись времени.

Возникает естественное желание ввести символьную запись даты и времени в одном из установленных форматов и затем преобразовать введенную строку во внутренний формат `TDateTime`. Такую возможность демонстрирует программа из листинга 14.1.

Листинг 14.1. Программа преобразования форматов дат `date1`

```
program date1;
uses DateUtils, SysUtils;
var
  dt : TdateTime;
  s : string;
  fmt : string = 'dd.mm.yyyy hh:nn:ss';
```

```
begin
  write('Input Date : ');   // запрос даты
  readln(s);                // ввод строки
  dt := StrToDateTime(s);   // преобразование к TDateTime
  writeln('Date = ',FormatDateTime(fmt,dt));
  readln;
end.
```

Протокол ее работы таков:

```
Running "e:\fpc\2.2.4\bin\i386-win32\date1.exe "
Input Date : 17.07.2000 11:03:55
Date = 17.07.2000 11:03:55
```

В этой же программе продемонстрирован способ обратного преобразования даты из формата `TDateTime` в символьную строку с помощью функции `FormatDateTime`.

Обратите внимание на то, что любая ошибка в наборе вводимой даты может привести к аварийной ситуации. Например, если вместо разделителя "точка" использовать запятую. Или вместо допустимого номера дня, например, набрать значение 32:

```
Running "e:\fpc\2.2.4\bin\i386-win32\date1.exe "
Input Date : 32.07.2000 11:03:55
An unhandled exception occurred at $0040CCC0 :
EConvertError : 2000-7-32 is not a valid date specification
```

Существуют две системные переменные, определяющие правила ввода значений дат. Первая из них — `ShortDateFormat` — регулирует порядок следования компонентов даты. Вторая — `DateSeparator` — определяет символ-разделитель между компонентами даты. Их значения по умолчанию можно вывести для того, чтобы познакомиться с установленными правилами набора даты (листинг 14.2).

Листинг 14.2. Программа `date2`

```
program date2;
uses DateUtils, SysUtils;
begin
  writeln('ShortDateFormat = ',ShortDateFormat);
  writeln('DateSeparator = ',DateSeparator);
  readln;
end.
```

Результат свидетельствует о том, что порядок компонентов даты — день (одна или две цифры), месяц (одна или две цифры) и год (четыре цифры).

А в качестве разделителя составляющих даты используется точка:

```
Running "e:\fpc\myprog\date2.exe "
ShortDateFormat = dd.MM.yyyy
DateSeparator = .
```

Вы имеете право изменить значение любой из этих переменных, и после такого изменения вступят в строй новые правила набора даты (листинг 14.3).

Листинг 14.3. Программа `date3`

```
program date3;
uses DateUtils, SysUtils;
var
  dt : TdateTime;
  s : string;
  fmt : string = 'dd.mm.yyyy hh:nn:ss';
begin
  DateSeparator := '-';           // новый разделитель
  ShortDateFormat:='yyyy-mm-dd';  // новая последовательность данных
  write('Input Date : ');   // запрос даты
  readln(s);                // ввод строки
  dt := StrToDateTime(s);   // преобразование к TDateTime
  writeln('Date = ',FormatDateTime(fmt,dt));
  readln;
end.
```

В результате мы изменили порядок следования компонентов и заменили символ-разделитель:

```
Running "e:\fpc\myprog\date3.exe "
Input Date : 2000-07-17 11:03:55
Date = 17.07.2000 11:03:55
```

При вводе даты и времени вместо полной группы их компонентов мы можем ограничиться набором только одного или двух первых значений. В этом случае функция `StrToDateTime` придерживается следующей стратегии. Недостающие поля даты заполняются системными (текущими) значениями месяца и/или года. Недостающие поля времени заполняются нулевыми значениями.

Функция `ScanDateTime` по своему назначению является аналогом функции `StrToDateTime`, но обладает несколько более широкими возможностями. В самом усеченном виде обращение к функции `ScanDateTime` содержит два аргумента (листинг 14.4).

Листинг 14.4. Программа `date4`

```
program date4;
uses DateUtils, SysUtils;
var
  fmt1: string = 'hh:nn:ss dd.mm.yyyy';
  s   : string = '11:03:55 17.07.2000';
  fmt2: string = 'dd.mm.yyyy hh:nn:ss';
  dt  : TdateTime;
begin
  dt := ScanDateTime(fmt1,s);
  writeln('Date = ',FormatDateTime(fmt2,dt));
  readln;
end.
```

Протокол работы программы date4.pas таков:

```
Running "e:\fpc\2.2.4\bin\i386-win32\date4.exe "
Date = 17.07.2000 11:03:55
```

Первый аргумент функции `ScanDateTime` представляет собой шаблон, описывающий формат составляющих даты и времени в значении второго аргумента (строки `s`). Более сложная модификация функции `ScanDateTime` предусматривает задание третьего необязательного целочисленного параметра `ind`. В этом случае сканирование строки `s` начинается с символа `s[ind]`.

Для преобразования даты из формата `TDateTime` в строку с тем или иным вариантом представления значений календарной даты и показаний компьютерных часов используется функция `FormatDateTime`:

```
s := FormatDateTime(Fmt, dt);
```

Эта функция возвращает значение типа `string`, в котором значения даты и времени сформированы в соответствии с форматными указателями в строке `Fmt`. Список возможных форматных указателей приведен в табл. 14.2.

Таблица 14.2

Указатель	Пояснение
`c`	Эквивалент `'ddddd t'` (короткие форматы даты и времени)
`d`	День месяца (одна или две цифры — 1, 2, ..., 9, 10, 11, ...)
`dd`	День месяца (две цифры — 01, 02, ...)
`ddd`	День недели (короткое символьное обозначение)
`dddd`	День недели (полное символьное обозначение)
`ddddd`	Короткий формат даты (например, 21.11.2009)

Таблица 14.2 *(окончание)*

Указатель	Пояснение
m	Месяц (одна или две цифры — 1, 2, ..., 9, 10, 11, 12)
mm	Месяц (две цифры — 01, 02, ..., 12)
mmm	Месяц (короткое символьное обозначение)
mmmm	Месяц (полное символьное обозначение)
y	Год (1 или 2 цифры)
yy	Год (2 цифры)
yyyy	Год (4 цифры)
h	Часы (1 или 2 цифры)
hh	Часы (2 цифры)
n	Минуты (1 или 2 цифры)
nn	Минуты (2 цифры)
s	Секунды (1 или 2 цифры)
ss	Секунды (2 цифры)
t	Короткий формат времени (например, 19:53)
tt	Длинный формат времени (например, 19:53:27.312)
am/pm	12-часовой формат с добавкой am или pm
a/p	12-часовой формат с добавкой a или p
/	Вставка разделителя в датах (обычно — точки)
:	Вставка разделителя во времени (обычно — двоеточия)
'xxx'	Вставка литерального текста xxx
"xxx"	Вставка литерального текста xxx
z	Миллисекунды (z, zz или zzz)

При работе с русифицированной операционной системой использование символьных обозначений для дней недели и месяцев может привести к нераспознаваемым символьным сочетаниям. Это связано с разницей в кодировке обозначений компонентов календаря, используемой в операционной системе, и кодировкой символьных данных в консольном приложении системы FP. В оригинальной операционной системе с установкой соответствующего англоязычного региона никаких проблем с выводом символьных обозначений дней недели и месяцев не возникает.

14.2.2. Опрос значений системных переменных

Для обработки календарных дат и компонентов времени программа может использовать значения специальных переменных, отслеживающих текущие показания системных часов компьютера, а также дат вчерашнего, сегодняшнего и завтрашнего дней. Список этих переменных, из которых разрешено только чтение, приведен в табл. 14.3.

Таблица 14.3

Формат опроса	Получаемое значение
`dt:= Date;`	Текущая дата с нулевыми компонентами времени
`dt:= Time;`	Текущее время с нулевой календарной датой
`dt:= Now;`	Текущая дата и текущее время
`dt:= Today;`	Дата текущего дня (эквивалент `Date`)
`dt:= Tomorrow;`	Дата завтрашнего дня с нулевым временем
`dt:= Yesterday;`	Дата вчерашнего дня с нулевым временем

Нулевая календарная дата соответствует точке начала отсчета в модифицированном юлианском календаре, т. е. 30 декабря 1899 г.

Для работы с указанными системными переменными необходимо подключать модуль `SysUtils` (листинг 14.5).

Листинг 14.5. Программа `date5`

```
program date5;
uses DateUtils, SysUtils;
var
  Fmt:string='dd/mm/yyyy hh:nn:ss.zzz';
begin
  writeln('Now  : ',FormatDateTime(Fmt, Now));
  writeln('Time : ',FormatDateTime(Fmt, Time));
  writeln('Date : ',FormatDateTime(Fmt, Date));
  writeln('Today: ',FormatDateTime(Fmt, Today));
  writeln('Yesterday : ',FormatDateTime(Fmt, Yesterday));
  writeln('Tomorrow : ',FormatDateTime(Fmt, Tomorrow));
  readln;
end.
```

Результаты работы программы date5.pas таковы:

```
Running "e:\fpc\myprog\date5.exe "
Now  : 22.11.2009 18:25:48.421
Time : 30.12.1899 18:25:48.421
Date : 22.11.2009 00:00:00.000
Today: 22.11.2009 00:00:00.000
Yesterday : 21.11.2009 00:00:00.000
Tomorrow : 23.11.2009 00:00:00.0000
```

14.2.3. Упаковка, замена и распаковка составляющих даты и времени

Группу подпрограмм, обеспечивающих упаковку отдельных компонентов и групп компонентов в данные типа `TDateTime`, составляют функции, чьи имена начинаются со слова `Encode` (табл. 14.4). По исходной информации они вычисляют все недостающие компоненты даты и времени, а затем преобразуют их в значение типа `TDateTime`. Все аргументы функций упаковки должны иметь тип `Word`.

Таблица 14.4

Формат обращения	Пояснение
`dt:=EncodeDateTime(Year,Month,Day, Hour,Minute,Second,MilliSecond);`	Упаковка заданных компонентов даты и времени
`dt:=EncodeDateMonthWeek(Year,Month, WeekOfMonth,DayOfWeek);`	Исходные данные — год, месяц, неделя месяца, день недели
`dt:=EncodeDateDay(Year,DayOfYear);`	Исходные данные — год, порядковый день года
`dt:=Encode(Year,WeekOfYear,DayOfWeek);`	Исходные данные — год, неделя года, день недели
`dt:=EncodeDayOfWeekMonth(Year,Month, NthDayOfWeek,DayOfWeek);`	Исходные данные — год, месяц, порядковый номер дня недели в месяце, день недели

Указанный в этой таблице параметр `NthDayOfWeek` означает последовательный номер в месяце дня недели `DayOfWeek`. Например, вторая (`NthDayOfWeek=2`) среда (`DayOfWeek=3`) месяца.

Еще пять функций с такими же названиями, которым предшествует префикс `Try` (попробуй), возвращают логические значения `True` или `False`. Первый результат свидетельствует о том, что значения всех аргументов принадлежат допустимым диапазонам. Второй результат говорит о наличии ошибок в исходных данных. Анализ результата может позволить программе предпринять какие-то меры по ис-

правлению ситуации. Ошибки в данных при упаковке генерируют исключение, приводящее к аварийному завершению программы.

С помощью функций, имена которых начинаются со слова `Recode` (табл. 14.5), вы можете в существующем значении типа `TDateTime` осуществить замены каждой составляющей даты-аргумента или группы ее компонентов на значения, заданные вторым и последующими аргументами. Все замещающие компоненты должны быть типа `Word`.

Таблица 14.5

Формат обращения	**Пояснение**
`dt2:=RecodeDateTime(dt1,Year,Month,Day, Hour,Minute,Second,MilliSecond);`	Замена всех компонентов в значении `dt1`
`dt2:=RecodeDate(dt1,Year,Month,Day);`	Замена года, месяца и дня
`dt2:=RecodeTime(dt1,Hour,Minute, Second,MilliSecond);`	Замена компонентов времени
`dt2:=RecodeYear(dt1,new_Year);`	Замена года
`dt2:=RecodeMonth(dt1,new_Month);`	Замена месяца
`dt2:=RecodeDay(dt1, new_Day);`	Замена дня
`dt2:=RecodeHour(dt1,new_Hour);`	Замена часов
`dt2:=RecodeMinute(dt1,new_Min);`	Замена минут
`dt2:=RecodeSecond(dt1,new_Sec);`	Замена секунд
`dt2:=RecodeMilliSecond(dt1,new_Mlsec);`	Замена миллисекунд

По аналогии с группой функций упаковки в состав модуля `DateUtils` входят пять процедур распаковки. Их имена начинаются со слова `Decode`, входным аргументом является значение даты и времени `dt` в формате `TDateTime`, а выходными параметрами — имена соответствующих переменных типа `Word`. В табл. 14.6 дополнительно включены семь функций по извлечению каждой составляющей даты и времени.

Таблица 14.6

Формат вызова	**Пояснение**
`DecodeDateDay(dt, Year,DayOfYear);`	Извлечение года и порядкового дня в году
`DecodeDateMonthWeek(dt, Year,Month, WeekOfMonth,DayOfWeek);`	Извлечение года, месяца, недели месяца и дня недели
`DecodeDateTime(dt, Year,Month,Day, Hour,Minute,Second,MilliSecond);`	Извлечение всех компонентов

Таблица 14.6 *(окончание)*

Формат вызова	Пояснение
`Decode(dt, Year,WeekOfYear,DayOfWeek);`	Извлечение года, номера недели в году и дня недели
`DecodeDayOfWeekMonth(dt, Year,Month, NthDayOfWeek,DayOfWeek);`	Извлечение года, месяца, количества повторений дня недели в месяце и дня недели
`Year:= YearOf(dt)`	Извлечение года
`Month:= MonthOf(dt)`	Извлечение месяца
`Day:= DayOf(dt)`	Извлечение дня месяца
`Hour:= HourOf(dt)`	Извлечение часов
`Minute:= MinuteOf(dt)`	Извлечение минут
`Second:= SecondOf(dt)`	Извлечение секунд
`MilliSecond:= MillisecondOf(dt)`	Извлечение миллисекунд

14.2.4. Вычисление различных дат в формате *TDateTime*

В состав этой группы включены процедуры и функции, обеспечивающие вычисление дат, соответствующих списку аргументов:

- даты, соответствующей началу того или иного компонента календарной даты;
- даты, соответствующей концу того или иного компонента календарной даты;
- даты, полученной из заданной, путем сложения с приращением указанного компонента календарной даты или времени.

В табл. 14.7 приведены функции, вычисляющие граничные значения (начало, конец) дат различных компонентов.

Таблица 14.7

Формат обращения	Пояснение
`dt:=StartOfAYear(Year);` `dt:=EndOfAYear(Year);`	Начало года, конец года
`dt:=StartOfAMonth(Year, Month);` `dt:=EndOfAMonth(Year, Month)`	Начало месяца, конец месяца
`dt:=StartOfAWeek(Year, WeekOfYear);` `dt:=EndOfAWeek(Year, WeekOfYear)`	Начало недели года, конец недели года

Таблица 14.7 *(окончание)*

Формат обращения	Пояснение
`dt:=StartOfAday(Year, DayOfYear);` `dt:=EndOfAday(Year, DayOfYear);`	Начало дня года, конец дня года
`dt:=StartOfADay(Year, Month, DayOfMonth);` `dt:=EndOfADay(Year, Month, DayOfMonth);`	Начало дня месяца конец дня месяца
`dt:=StartOfAWeek(Year,WeekOfYear,DayOfWeek);` `dt:=EndOfAWeek(Year,WeekOfYear,DayOfWeek);`	Начало дня недели, конец дня недели
`dt2:=StartOfTheDay(dt1);`	Начало дня для `dt1`
`dt2:=EndOfTheDay(dt1);`	Конец дня для `dt1`
`dt2:=StartOfTheWeek(dt1);`	Начало недели для `dt1`
`dt2:=EndOfTheWeek(dt1);`	Конец недели для `dt1`
`dt2:=StartOfTheMonth(dt1);`	Начало месяца для `dt1`
`dt2:=EndOfTheMonth(dt1);`	Конец месяца для `dt1`
`dt2:=StartOfTheYear(dt1);`	Начало года для `dt1`
`dt2:=EndOfTheYear(dt1);`	Конец года для `dt1`

Всем начальным датам соответствует время 00:00:00.000, а конечным — 23:59:59.999.

Список функций, реализующих вычисление дат, получающихся из заданной путем прибавления приращения по той или иной составляющей, приведен в табл. 14.8. Если значение приращения в списке аргументов опущено, то по умолчанию к соответствующему компоненту прибавляется 1. Это соответствует общепринятому употреблению процедуры `inc(x)`. Отличие функций `Inc...` состоит в том, что величина приращения может быть отрицательной, и это позволяет вычислять не только будущие, но и прошедшие даты.

Таблица 14.8

Формат вызова	Пояснение
`dt2:=IncYear(dt1,[dY]);`	Приращение года
`dt2:=IncWeek(dt1,[dW]);`	Приращение недели года
`dt2:=IncDay(dt1,[dD]);`	Приращение дня года
`dt2:=IncHour(dt1,[dH]);`	Приращение часов
`dt2:=IncMinute(dt1,[dM]);`	Приращение минут

***Таблица 14.8** (окончание)*

Формат вызова	Пояснение
`dt2:=IncSecond(dt1,[dS]);`	Приращение секунд
`dt2:=IncMilliSecond(dt1,[dMls]);`	Приращение миллисекунд

Несколько особняком стоят две следующие функции, возвращающие результат типа `Word`:

```
rpt:=NthDayOfWeek(dt);
prew_w:=PreviosDayOfWeek(DayOfWeek);
```

Первая функция определяет количество повторений в месяце дня недели, представленного в дате `dt` формата `TDateTime`. Например, в январе 2009 г. *среда* встречается *четыре* раза, а *четверг — пять* раз.

Вторая функция определяет номер дня недели, предшествующий аргументу `DayOfWeek`. Например, вторнику (`DayOfWeek=2`) предшествует понедельник, т. е. день недели с номером 1. А понедельнику предшествует воскресенье, т. е. день недели с номером 7.

14.2.5. Измерение интервалов времени

В задачах оперативного и долгосрочного планирования довольно часто приходится определять длительность интервалов между двумя календарными датами `dt1` и `dt2`, вычислять даты, отстоящие от заданной точки отсчета на указанный интервал времени вперед или назад.

В составе модуля `DateUtils` присутствуют две группы функций, вычисляющих разницу между двумя датами типа `TDateTime` в различных единицах измерения интервалов времени и выдающих результат либо в виде целого числа, либо в виде приближенного значения типа `double`. В первом случае значение функции соответствует полному числу единиц измерения, полученному путем выделения соответствующих компонентов и отбрасывания дробной части их разности. Названия таких функций начинаются с единицы измерения и заканчиваются словом `Between` (между). Во втором случае имя функции также начинается с единицы измерения временного интервала и заканчивается словом `Span` (промежуток времени). Перечень этих функций приведен в табл. 14.9.

Таблица 14.9

Формат обращения	Пояснение
`IdY:=YearsBetween(dt1,dt2);` `DdY:=YearSpan(dt1,dt2);`	Разница в годах (`Integer`), разница в годах (`Double`)
`IdM:=MonthsBetween(dt1,dt2);` `DdM:=MonthSpan(dt1,dt2);`	Разница в месяцах (`Integer`), разница в месяцах (`Double`)

Таблица 14.9 *(окончание)*

Формат обращения	Пояснение
`IdW:=WeeksBetween(dt1,dt2);` `DdW:=WeekSpan(dt1,dt2);`	Разница в неделях (`Integer`), разница в неделях (`Double`)
`IdD:=DaysBetween(dt1,dt2);` `DdD:=DaySpan(dt1,dt2);`	Разница в днях (`Integer`), разница в днях (`Double`)
`IdH:=HoursBetween(dt1,dt2);` `DdH:=HourSpan(dt1,dt2);`	Разница в часах (`Int64`), разница в часах (`Double`)
`IdM:=MinutesBetween(dt1,dt2);` `DdM:=MinuteSpan(dt1,dt2);`	Разница в минутах (`Int64`), разница в минутах (`Double`)
`IdS:=SecondsBetween(dt1,dt2);` `DdS:=SecondSpan(dt1,dt2);`	Разница в секундах (`Int64`), разница в секундах (`Double`)
`IdMls:=MilliSecondsBetween(dt1,dt2);` `DdMls:=MilliSecondSpan((dt1,dt2);`	Разница в миллисекундах (`Int64`), разница в миллисекундах (`Double`)

При вычислении разницы в месяцах и годах используются следующие приближенные значения:

```
ApproxDaysPerMonth : Double = 30.4375 // среднее число дней в месяце
ApproxDaysPerYear : Double = 365.25   // среднее число дней в году
```

Следующую группу из 30 подпрограмм составляют функции, вычисляющие интервалы времени, отсчитываемые в разных единицах от начала соответствующего фрагмента даты — от начала года, месяца, недели, дня, часа, минуты, секунды. Их названия составлены из полных английских фраз, четко раскрывающих смысл возвращаемого значения (табл. 14.10). В дополнение к своему основному назначению некоторые функции извлекают из даты те или иные компоненты.

Таблица 14.10

Формат обращения	Пояснение
`M:=MonthOfTheYear(dt);`	Число месяцев с начала года

`W:=WeekOfTheYear(dt);`
`W:=WeekOfTheYear(dt,Year);`
`W:=WeekOfTheMonth(dt);`
`W:=WeekOfTheMonth(dt,Year,Month);`
	Число недель с начала года,
	то же + извлечение года,
	число недель с начала месяца,
	то же + извлечение года и месяца

`D:=DayOfTheYear(dt);`
`D:=DayOfTheMonth(dt);`
`D:=DayOfTheWeek(dt);`
	Число дней с начала года,
	число дней с начала месяца,
	число дней с начала недели

`H:=HourOfTheYear(dt);`
`H:=HourOfTheMonth(dt);`
`H:=HourOfTheWeek(dt);`
`H:=HourOfTheDay(dt);`
	Число часов с начала года,
	число часов с начала месяца,
	число часов с начала недели,
	число часов с начала дня

Таблица 14.10 *(окончание)*

Формат обращения	Пояснение

`M:=MinuteOfTheYear(dt);`
`M:=MinuteOfTheMonth(dt);`
`M:=MinuteOfTheWeek(dt);`
`M:=MinuteOfTheDay(dt);`
`M:=MinuteOfTheHour(dt);`
	Число минут с начала года,
	число минут с начала месяца,
	число минут с начала недели,
	число минут с начала дня,
	число минут с начала часа

`S:=SecondOfTheYear(dt);`
`S:=SecondOfTheMonth(dt);`
`S:=SecondOfTheWeek(dt);`
`S:=SecondOfTheDay(dt);`
`S:=SecondOfTheHour(dt);`
`S:=SecondOfTheMinute(dt);`
	Число секунд с начала года,
	число секунд с начала месяца,
	число секунд с начала недели,
	число секунд с начала дня,
	число секунд с начала часа,
	число секунд с начала минуты

`M:=MilliSecondOfTheYear(dt);`
`M:=MilliSecondOfTheMonth(dt);`
`M:=MilliSecondOfTheWeek(dt);`
`M:=MilliSecondOfTheDay(dt);`
`M:=MilliSecondOfTheHour(dt);`
`M:=MilliSecondOfTheMinute(dt);`
`M:=MilliSecondOfTheSecond(dt);`
	Число миллисекунд с начала года,
	число миллисекунд с начала месяца,
	число миллисекунд с начала недели,
	число миллисекунд с начала дня,
	число миллисекунд с начала часа,
	число миллисекунд с начала минуты,
	число миллисекунд с начала секунды

Аргументом всех этих функций является дата в формате `TDateTime`. В зависимости от диапазона соответствующих значений функции возвращают результат от типа `Word` до типа `Int64`.

14.2.6. Сравнение календарных дат и показаний часов

Сравнение календарных и/или временных компонентов может быть выполнено двумя способами. Функции, чьи имена начинаются со слова `Compare` (сравнить), возвращают целочисленный результат. Он может быть положительным, если первый аргумент (или его часть) больше второго аргумента (или соответствующей его части), равен нулю в случае равенства сравниваемых значений, и отрицательным, если первое значение меньше второго. Функции, имена которых начинаются со слова `Same` (такой же), возвращают логическое значение, равное `True` в случае совпадения сравниваемых значений и `False` — в случае их несовпадения. Такой способ сравнения на равенство выполняется быстрее, чем вычитание арифметических значений и последующая проверка на нулевой результат. В табл. 14.11 приводится перечень описанных функций.

Таблица 14.11

Формат обращения	Пояснение
`iv:=CompareDate(dt1,dt2);`	Арифметическое сравнение календарных дат
`bv:=SameDate(dt1,dt2);`	Логическое сравнение календарных дат

Таблица 14.11 *(окончание)*

Формат обращения	Пояснение
`iv:=CompareTime(dt1,dt2);`	Арифметическое сравнение времени дня
`bv:=SameTime(dt1,dt2);`	Логическое сравнение времени дня
`iv:=CompareDateTime(dt1,dt2);`	Арифметическое сравнение обеих составляющих
`bv:=SameDateTime(dt1,dt2);`	Логическое сравнение обеих составляющих

Функция `IsSameDay(dt1,dt2)` дублирует результат работы функции `SameDate`, т. к. возвращает значение `True` только в случае совпадения года, месяца и дня, представленных в обоих операндах.

Не так уж и целесообразна функция `IsToday(dt)`, которая проверяет, совпадает ли календарная дата, указанная в `dt`, с данными сегодняшнего дня. Ее использование эквивалентно вызову `SameDate(dt,Now)`.

К группе процедур сравнения разумно отнести серию логических функций, имена которых начинаются с сочетания `WithinPast`. Они выделяют из двух своих аргументов `dt1` и `dt2` формата `TDateTime` одноименные компоненты календарной даты или показаний часов и сравнивают абсолютную величину их разности с заданным рассогласованием. Если разница дат по указанному компоненту не превосходит допустимое значение, то функция возвращает `True`. При этом следует помнить, что выделяемый компонент никак не связывается со значениями соседних полей и рассматривается как целое число. Если, например, сравнение ведется по годам, то абсолютная разница в *1 год и 364 дня* считается равной *одному* году. В этом можно убедиться на следующем примере (листинг 14.6).

Листинг 14.6. Программа `date6`

```
program date6;
uses DateUtils, SysUtils;
var
  dt1,dt2: TDateTime;
  fmt: string='ddddd tt';
  s1: string='WithinPastYears(dt1,dt2,1) = ';
  s2: string='WithinPastYears(dt2,dt1,1) = ';
begin
  dt1:=EncodeDate(2009,1,1);
  dt2:=EncodeDate(2010,12,31);
  writeln('dt1 = ',FormatDateTime(fmt,dt1));
  writeln('dt2 = ',FormatDateTime(fmt,dt2));
  writeln(s1,WithinPastYears(dt1,dt2,1));
  writeln(s2,WithinPastYears(dt2,dt1,1);
  readln;
end.
```

Далее приводится протокол работы программы date6.pas:

```
Running "e:\fpc\myprog\date6.exe "
dt1 = 01.01.2009 0:00:00
dt2 = 31.12.2010 0:00:00
WithinPastYears(dt1,dt2,1) = TRUE
WithinPastYears(dt2,dt1,1) = TRUE
```

Список функций типа `WithinPast...` приведен в табл. 14.12.

Таблица 14.12

Формат обращения	Проверка
`WithinPastYears(dt2,dt1,dY);`	`|Y(dt1)-Y(dt2)| ≤ dY`
`WithinPastMonths(dt1,dt2,dMon);`	`|Mon(dt1)-Mon(dt2)| ≤ dMon`
`WithinPastWeeks(dt1,dt2,dW);`	`|W(dt1)-W(dt2)| ≤ dW`
`WithinPastDays(dt1,dt2,dD);`	`|D(dt1)-D(dt2)| ≤ dD`
`WithinPastHours(dt1,dt2,dH);`	`|H(dt1)-H(dt2)| ≤ dH`
`WithinPastMinutes(dt1,dt2,dMin);`	`|Min(dt1)-Min(dt2)| ≤ dMin`
`WithinPastSeconds(dt1,dt2,dS);`	`|S(dt1)-S(dt2)| ≤ dS`
`WithinPastMilliSeconds(dt1,dt2,dMls);`	`|Mls(dt1)-Mls(dt2)| ≤ dMls`

Из других функций, которые можно условно отнести к группе процедур сравнения, отметим две — `IsInLeapYear(dt)` и `IsPM(dt)`. Первая определяет принадлежность даты `dt` високосному году. Вторая возвращает значение `False`, если показания часов не достигли полудня, и `True`, если временная составляющая в `dt` перешагнула полуденный рубеж.

14.2.7. Юлианский календарь

Работу с юлианскими датами обеспечивают две функции, реализующие прямое и обратное преобразования между значениями типа `TDateTime` и типа `Double`:

```
var
  jd : Double;
  dt : TDateTime;
  ...
  jd := DateTimeToJulianDate(dt);
  ...
  dt := JulianDateToDateTime(jd);
```

14.2.8. Контроль правильности дат и времени

Для контроля правильности значений календарных дат, времени и их отдельных компонентов в модуле `DateUtils` предусмотрены две группы подпрограмм. Имена логических функций первой группы начинаются с сочетания `IsValid` (в переводе — являются правильными). Они возвращают значение `True`, если все их аргументы принадлежат допустимым интервалам. Список таких функций приведен в табл. 14.13.

Таблица 14.13

Формат обращения	Аргументы
`IsValidDate(Y,M,D);`	Год, месяц, день
`IsValidTime(H,Min,S,Mlsec);`	Часы, минуты, секунды, миллисекунды
`IsValidDateTime(Y,M,D,H,Min,S,Mlsec);`	Дата и время
`IsValidDateDay(Y,DayOfYear);`	Год, день года
`IsValidDateWeek(Y,WeekOfYear,DayOfWeek);`	Год, неделя года, день недели
`IsValidDateMonthWeek(Y,M,WeekOfMonth, DayOfWeek);`	Год, месяц, неделя месяца, день, недели

Имена второй группы функций начинаются со слова `Invalid` (неправильный) и заканчиваются словом `Error`. Они генерируют исключительную ситуацию, если хотя бы один из аргументов функции не принадлежит допустимому интервалу.

На наш взгляд, наличие трех групп функций, так или иначе связанных с проверкой своих аргументов (`TryEncode...`, `IsValid...`, `Invalid...`), — это чересчур.

14.3. Альтернативные средства работы с датами и временем

Одна из дополнительных возможностей определения компонентов текущей календарной даты и показаний системных часов заключается в прямом обращении к соответствующей функции операционной системы (листинг 14.7).

Листинг 14.7. Программа `Win_Date`

```
program Win_Date;
uses Windows,DateUtils,SysUtils;
var
  st: SystemTime;
```

```
  dt: TDateTime;
  Fmt: string='ddddd hh:nn:ss.zzz';
begin
  dt:=Now;
  GetSystemTime(st);
  writeln(FormatDateTime(Fmt,dt));
  writeln('Day Of The Week = ',DayOfTheWeek(dt));
  writeln;
  writeln('Year = ',st.wYear);
  writeln('Month = ',st.wMonth);
  writeln('Day Of Week = ',st.wDayOfWeek);
  writeln('Day Of Month = ',st.Day);
  writeln('Hour = ',st.wHour);
  writeln('Minute = ',st.Minute);
  writeln('Second = ',st.wSecond);
  writeln('Milliseconds = ',st.wMilliseconds);
  readln;
end.
```

В этом примере системная функция `GetSystemTime` возвращает отдельные компоненты системной даты в целочисленных полях структуры типа `SystemTime`.

```
Running "e:\fpc\myprog\win_date.exe "
22.11.2009 12:54:50.281
Day Of The Week = 7

Year = 2009
Month = 11
Day Of Week = 0
Day Of Month = 22
Hour = 9
Minute = 54
Second = 50
Milliseconds = 281
```

Несовпадение отдельных компонентов с форматом `TDateTime` вызвано двумя обстоятельствами. Во-первых, часовой пояс Москвы отличается на три часа от нулевого меридиана, проходящего через обсерваторию Гринвича. Во-вторых, системная нумерация дней недели ведется по модулю 7, поэтому воскресенье имеет нулевой числовой код. Эти детали несложно учесть в программе.

Вторая возможность заключается в использовании устаревших процедур, доставшихся нам как наследие MS-DOS.

Эти процедуры и используемая ими структура данных входят в состав модуля `Dos`:

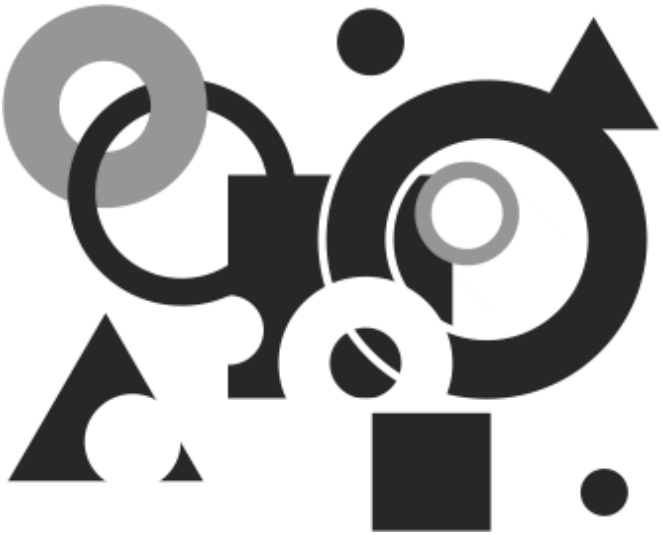
```
type
  DateTime = packed record   // упакованная запись
    Year : word;             // год из диапазона [1980..2099]
    Month: word;             // месяц из диапазона [1..12]
    Day  : word;             // день месяца из диапазона [1..31]
    Hour : word;             // часы из диапазона [0..23]
    Min  : word;             // минуты из диапазона [0..59]
    Sec  : word;             // секунды из диапазона [0..59]
  end;
var
  dt : DateTime;             // запись для хранения даты и времени
  pack_dt : longint;         // для упакованных полей даты и времени
  S100 : word;               // для хранения сотых долей секунды
  WeekDay : word;            // для хранения номера дня недели
GetTime(Hour, Min, Sec, S100);       // опрос системного времени
GetDate(Year, Month, Day, WeekDay);  // опрос системной даты
PackTime(dt,pack_dt);        // упаковка даты и времени
UnpackTime(pack_dt,dt);      // распаковка даты и времени
```

Номер дня недели (`WeekDay`) отсчитывался от нуля, соответствовавшего воскресенью.

Четыре байта, отведенных для хранения упакованных значений компонентов календарной даты и времени, использовались следующим образом:

Year	**Month**	**Day**	**Hour**	**Min**	**Sec/2**
7 бит	4 бита	5 бит	5 бит	6 бит	5 бит

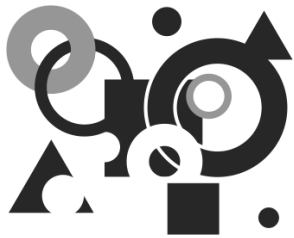
При этом в поле `Year` записывалась разность между текущим годом и началом допустимого интервала (1980), а в поле `Sec/2` — число секунд, деленное пополам (у нечетных значений младшая единица отбрасывалась). Естественно, что распаковка упакованной величины не могла правильно восстановить исходное значение нечетных секунд.



ЧАСТЬ III

Графика

Глава 15



Графические средства языка Free Pascal

В этой главе описывается набор процедур и функций, унаследованный языком Free Pascal от ранней графической библиотеки BGI (Borland Graphics Interface). Он практически повторяет набор графических подпрограмм, реализованных в системах Turbo Pascal и Borland Pascal, с единственным расширением, позволяющим более полно использовать разрешение современных дисплеев. Дополнительной особенностью графики системы Free Pascal является выделение консольному приложению двух окон (рис. 15.1).

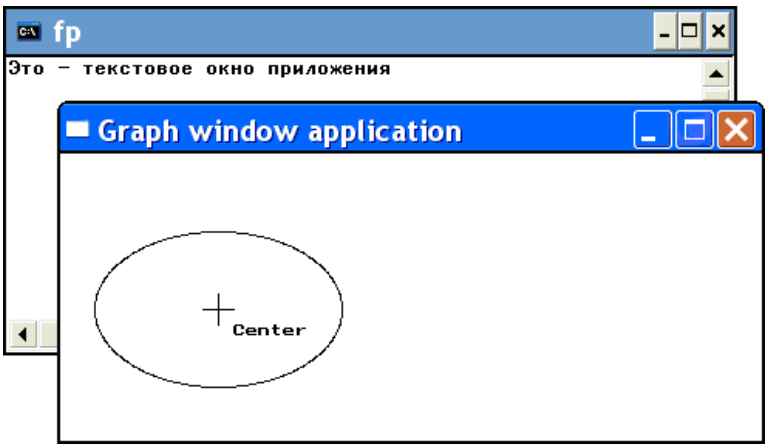


Рис. 15.1. Главное и графическое окна приложения

В главном окне реализуются обычные интерфейсные взаимодействия между пользователем и приложением (ввод/вывод по операторам `read`/`readln`, `write`/`writeln`, `readkey`, `keypressed`), в дополнительном окне выполняются построения графических фигур и отображение пояснительных подписей с помощью процедур BGI.

15.1. Основные характеристики графического окна

15.1.1. Система координат

Рабочее поле графического окна, расположенное под заголовком окна, снабжено *системой координат*, начало которой находится в левом верхнем углу. При этом *ось y* направлена вниз, а *ось x* — вправо. В качестве единиц измерения приняты *пикселы* экрана. Если традиционные дисплеи поддерживали довольно много режимов разрешения, при которых размер пикселов мог меняться в достаточно широком диапазоне, то современные плоские мониторы (плазменные и жидкокристаллические) обычно поддерживают один-два варианта разрешения. Как правило, основной рабочий режим современного монитора устанавливается под управлением операционной системы и выбирается из соображений максимальной четкости изображений. Поэтому то, что раньше подразумевалось под разрешением экрана и было связано с изменением геометрических размеров пиксела, сегодня свелось к изменению размеров графического окна при сохранении физического размера пиксела. Таким образом, среди основных характеристик графического окна присутствуют два параметра, задающие ширину (`MaxX`) и высоту (`MaxY`) рабочего поля в пикселах. Используя стандартные средства управления размерами окна, вы можете во время работы приложения изменить местоположение и габариты графического экрана, но параметры `MaxX` и `MaxY`, установленные при создании графического окна, при этом сохраняют свои первоначальные значения.

15.1.2. Графический курсор

Ряд графических процедур наряду с абсолютными координатами использует и относительные координаты, которые задаются в виде смещений (`dx`, `dy`) относительно позиции *текущей точки* CP (от англ. *Current Point*). При создании графического окна текущая точка помещается в начало координат. Ее последующие перемещения зависят от выполняемых графических операций. Например, при построении отрезка прямой текущая точка переводится в конец отображаемого отрезка. При построении окружности положение текущей точки не изменяется. Текущую точку обычно называют *графическим курсором*, который в отличие от постоянно мигающего текстового курсора не изображается в графическом окне, чтобы не исказить выводимую картинку. Координаты текущей точки программа может опросить с помощью функций `GetX` и `GetY`.

15.1.3. Буфер графического окна

Все, что представлено на рабочем поле графического окна, является отображением некоторого участка видеопамяти — *буфера графического окна*. Основным содержанием буфера являются коды цветности, в которые окрашен каждый пиксел рабочего поля. Для хранения кода цветности каждого пиксела в видеопамяти выделяется до 24 двоичных разрядов. В соответствии с этим изображение в графическом окне может быть монохромным или цветным с более или менее насыщенной цветовой палитрой (16, 64, 256, 256 тыс. или 16 млн оттенков). Код цветности обычно представляет собой комбинацию интенсивностей трех базовых цветов — красного (R — Red), зеленого (G — Green) и синего (B — Blue).

В каждый конкретный момент графическая система имеет дело с двумя ранее установленными цветами — цветом рисования или цветом *переднего плана* (foreground color) и цветом *фона* или цветом *заднего плана* (background color). Цвету фона всегда соответствует нулевой программный код. На самом деле, это не комбинация цветов RGB, а номер специального регистра видеокарты, в котором находится настоящая комбинация базовых цветов. Так что цвет фона не всегда является только черным. Изменение содержимого указанного регистра производится с помощью процедуры `SetBackgroundColor` и сказывается только в тот момент времени, когда программа обращается к процедуре очистки графического экрана (`ClearDevice`).

В отличие от этого, изменение цвета переднего плана, осуществляемое с помощью процедуры `SetColor`, в процедурах рисования сказывается мгновенно в момент записи нового цвета пиксела в соответствующий участок видеопамяти. Видеокарта позволяет сформировать новый цвет пиксела с учетом его предыдущей окраски. Новый код цветности может заменить старый (простое вытеснение) или над обоими кодами может быть выполнена одна из логических операций — `AND`, `OR` или `XOR`. Обратите особое внимание на операцию исключающего "ИЛИ" (`XOR`), которая используется для стирания или "проявления" изображений. Если на код цвета пиксела накладывается такой же двоичный код по операции `XOR`, то результат будет нулевым, что соответствует цвету фона или стиранию ранее окрашенной точки. Если на нулевой код цветности наложить прежний цвет пиксела, то точка на экране "проявится". Этой возможностью часто пользуются при анимации изображений.

Еще одна возможность быстрой смены изображений на экране обеспечивается наличием нескольких графических буферов. При достаточно большом объеме видеопамяти под хранение изображений может быть выделено две или более графических страниц (буферов). Одна из таких страниц объявляется *видимой* (Visual Page), и ее содержимое отображается в графическом окне, а другая — *активной* (Active Page). По умолчанию графическая страница с нулевым номером является одновременно и активной, и видимой. Поэтому результат действия процедур, воспроизводящих тот или иной графический примитив, немедленно отображается на графическом экране. Мы можем объявить другую страницу активной:

```
SetActivePage(1);
```

После этой операции все графические процедуры будут заносить свои изменения в невидимое окно, а на экране по-прежнему будет отображаться содержимое нулевой страницы. Спустя некоторое время мы можем объявить первую страницу видимой:

```
SetVisualPage(1);
```

И тогда ее содержимое мгновенно покажется на графическом экране. Так как время записи в активную страницу совмещено со временем отображения видимой страницы, то их переключение приводит к практически мгновенной смене кадров.

15.2. Создание графического окна

Создание графического окна и переход в режим выполнения графических операций напоминает подготовку к работе с файлами. В жаргоне программистов бытует термин "открыть графику", и это надо сделать прежде, чем вы обратитесь к любой графической процедуре. Выполняется эта операция с помощью процедуры `InitGraph`:

```
InitGraph(gd,gm,'');
```

Два первых ее аргумента — имена переменных типа `SmallInt`. Переменная `gd` символизирует номер, под которым система FP хранит одну из служебных программ, реализующих операции обмена с графическим экраном. Программы такого рода принято называть *графическими драйверами* (graphics driver). Переменная `gm` предназначена для хранения числового кода, определяющего режим, который должен поддерживать графический драйвер (graphics mode). Приведенные имена переменных являются аббревиатурами соответствующих английских терминов (но вы можете использовать и любые другие имена). Третий аргумент процедуры `InitGraph`, представленный пустой строкой, должен указывать путь к каталогу, в котором находится драйвер. Отсутствие значения этого параметра говорит о том, что компилятор должен сам найти нужную программу.

Что касается значения переменной `gd`, то ее обычно задают равной нулю. Это означает, что система FP должна сама опросить параметры графического оборудования компьютера и выбрать подходящие значения переменных `gd` и `gm`. В дальнейшем программист этими переменными не пользуется, но и не должен затирать их значения.

Опытный программист, часто имеющий дело с графическими режимами, перед открытием графики может сам задать нужные ему значения переменных `gd`, `gm`. Для их выбора можно прибегнуть к услугам табл. 15.1 и 15.2 или файлам помощи.

Таблица 15.1

Условное обозначение драйвера	Числовой код	Примечание
`D1bit`	11	Монохромный режим
`D2bit`	12	Минимальный цветной режим, в котором каждая из допустимых палитр обеспечивает отображение четырех цветов
`D4bit`	13	Вспомогательный режим для IBM PC
`D6bit`	14	Для ПК Amiga (полутоновый режим, 64 цвета)
`D8bit`	15	Основной режим для IBM PC
`D12bit`	16	Для ПК Amiga (цветной режим, 4096 цветов)
`D15bit`	17	
`D16bit`	18	
`D24bit`	19	В версии 2.2.4 пока не поддерживается
`D32bit`	20	В версии 2.2.4 пока не поддерживается
`D64bit`	21	В версии 2.2.4 пока не поддерживается

Таблица 15.2

Условное обозначение режима	Числовой код	Примечание
`detectMode`	30000	Автоматическое определение режима
`m320x200`	30001	
`m320x256`	30002	Для ПК Amiga (режим PAL)
`m320x400`	30003	Для ПК Amiga или Atari
`m512x384`	30004	Для ПК Macintosh
`m640x200`	30005	Для IBM PC (режимы EGA, VGA)
`m640x256`	30006	Для ПК Amiga (режим PAL)
`m640x350`	30007	Для IBM PC (режимы EGA, VGA)
`m640x400`	30008	Для IBM PC (режимы VGA, SVGA)
`m640x480`	30009	Для IBM PC (режимы VGA, SVGA)
`m800x600`	30010	Для IBM PC (режимы VGA, SVGA)
`m832x624`	30011	Для ПК Macintosh

Таблица 15.2 *(окончание)*

Условное обозначение режима	Числовой код	Примечание
m1024x768	30012	Для IBM PC (режимы SVGA)
m1280x1024	30013	Для IBM PC (режимы SVGA)
m1600x1200	30014	Для IBM PC (режимы SVGA)
m2048x1536	30015	Для IBM PC (режимы SVGA)

Выбор драйвера фактически означает выбор цветовой палитры, используемой при выполнении графических процедур. Для пользователей IBM PC авторы версии Free Pascal 2.2.4 пока ограничились 8-битной глубиной цвета. В этом режиме можно определить цветовую палитру, содержащую 256 оттенков, выбранных из 256 000 всевозможных цветовых комбинаций. Поддержка режима TrueColor (глубина цвета — 24 или 32 бита) еще не реализована.

Выбор графического режима фактически регламентирует размер рабочего поля графического окна приложения. В тех случаях, когда параметры процедуры `InitGraph` принимают значения `gd=0`, `gm=0` или `gm=30000`, размер графического окна совпадает с полным экраном. Габариты графического окна не могут превышать физическое разрешение монитора.

После обращения к процедуре `InitGraph` целесообразно удостовериться, что операция открытия графики прошла успешно. Оценить это можно по значению системной переменной `GraphResult`, которое должно быть нулевым (равным мнемонической константе `GrOK`):

```
var
  gd,gm: SmallInt;
  err: Integer;
  ...
begin
  InitGraph(gd,gm,'');
  err:=GraphResult;             // опрос GraphResult
  if err <> GrOK then begin     // анализ завершения операции
     writeln('Графика не открылась');
     halt(1);    // останов, если графика не открылась
  end;
end.
```

Несовместимая комбинация значений `gd` и `gm`, как правило, приводит к `Graphresult=-10`, после чего выполнение графических операций невозможно.

Завершая работу с графической системой, программа должна "закрыть графику", обратившись к процедуре `CloseGraph`:

```
var
  gd,gm: SmallInt;
```

```
  err:Integer;
  ...
begin
  InitGraph(gd,gm,''); // инициализация графики (выделение ресурсов)
  circle(100,100,20);  // построение окружности
  readln;
  closegraph;          // освобождение ресурсов
end.
```

В состав группы подпрограмм, связанных с инициализацией графики (табл. 15.3), входит 17 процедур и функций, к большинству из которых пользователю приходится обращаться крайне редко. В некоторых случаях речь идет об устаревших характеристиках вроде *aspect ratio* (соотношение сторон). На мониторах типа EGA (Enhanced Graphics Adapter), которые сейчас можно увидеть только в музеях, пикселы экрана представляли прямоугольник, вытянутый по оси *y*. Поэтому на них окружности рисовались как эллипсы, а квадраты как прямоугольники. Для восстановления правильных пропорций использовался поправочный коэффициент aspect ratio. На современных мониторах пиксел имеет форму квадрата, и необходимость во введении поправочных коэффициентов отпала. Абсолютно никакой информации для пользователя не несут ни имя драйвера, ни имя текущего графического процесса. За исключением очень продвинутых программистов никому не приходится добавлять новые драйверы и регистрировать их в системе FP.

Таблица 15.3

Имя подпрограммы	Назначение
`ClearDevice`	Очистка графического экрана
`CloseGraph`	Завершение работы в графическом режиме, передача управления главному "текстовому" окну
`DetectGraph`	Определение (опрос) графического режима
`GetAspectRatio`	Опрос коэффициентов aspect ratio
`GetModeRange`	Опрос допустимого диапазона графических режимов для текущего драйвера
`GraphDefaults`	Восстановление параметров графической системы по умолчанию
`GetDriverName`	Опрос имени графического драйвера
`GetGraphMode`	Опрос текущего или последнего использованного графического режима
`GetMaxMode`	Опрос максимального номера графического режима для текущего драйвера
`GetModeName`	Опрос имени текущего графического режима

Таблица 15.3 *(окончание)*

Имя подпрограммы	Назначение
GraphErrorMsg	Символьное сообщение, соответствующее ошибочному значению GraphResult
GraphResult	Признак завершения последней графической операции
InitGraph	Инициализация графических драйверов
InstallUserDriver	Установка нового графического драйвера
RegisterBGIDriver	Регистрация нового графического драйвера
RestoreCRTMode	Возврат в текстовый режим
SetGraphMode	Установка графического режима

К упомянутым ранее обязательным процедурам раздела инициализации добавим следующие — очистку графического экрана (ClearDevice), переключения между основным и графическим окнами (RestoreCRTMode и SetGraphMode). Для возврата в ранее установленный графический режим обращение к последней процедуре должно выглядеть следующим образом:

```
SetGraphMode(GetGraphMode);
```

Для получения более подробной информации о причинах возникновения ошибочной ситуации при выполнении графической операции полезно вывести символьное сообщение, соответствующее ненулевому значению GraphResult:

```
writeln(GraphErrorMsg(GraphResult));
```

Помните, что первое же обращение к системной переменной GraphResult сбрасывает ее в ноль. Если вы хотите использовать ее несколько раз, то предварительно нужно скопировать ее значение в промежуточную переменную:

```
err:=GraphResult;
if err<>GrOK then begin
   writeln(GraphErrormsg(err));
   exit(1);
end;
```

15.3. Управление цветом

В цветовом режиме D8bit выбор цвета осуществляется следующим образом. Видеокарта, управляющая окраской пикселов графического окна, использует цифроаналоговый преобразователь (ЦАП), в котором код цветности представлен 18-разрядным двоичным кодом — по 6 разрядов на интенсивность RGB-компонентов. В блоке ЦАП находится 256 регистров, содержимое которых образует текущую

цветовую палитру. Программа имеет возможность опросить или изменить содержимое любого регистра палитры или содержимое всех регистров одновременно.

На выполнение графических операций оказывают влияние три основные характеристики видеокарты. В дополнение к упоминавшимся ранее цветам переднего (foreground color) и заднего плана (background color) добавляется *цвет заливки* (filling color), участвующий в закраске различных замкнутых контуров. Физические компоненты цвета фона (т. е. заднего плана) хранятся в нулевом регистре ЦАП. Поэтому изменение цвета фона, осуществляемое с помощью процедуры `SetBkColor`, сводится к переписи содержимого регистра ЦАП с указанным номером `n_reg` в нулевой регистр ЦАП:

```
SetBkColor(n_reg);              // n_reg из диапазона [0, 255]
```

Цвет рисования, которым воспроизводятся отрезки прямых и контуры различных графических фигур (примитивов — дуг, окружностей, эллипсов, многоугольников и др.), устанавливается с помощью процедуры `SetColor`. Ее единственным аргументом является номер регистра ЦАП, в котором зафиксирован физический цвет переднего плана:

```
SetColor(n_reg);                // n_reg из диапазона [0, 255]
```

Цвет и способ заливки устанавливается с помощью процедуры `SetFillStyle`:

```
SetFillStyle(Pattern, n_reg);  // n_reg из диапазона [0, 255]
```

Параметр `Pattern` (шаблон) определяет способ заполнения замкнутой фигуры (заливка сплошным цветом, всевозможные штриховки, растровая закраска и т. п.). Со всеми способами, поддерживаемыми библиотекой BGI, мы познакомимся позднее.

Для знакомства с палитрой по умолчанию, которая устанавливается при инициализации графической системы, можно воспользоваться программой из листинга 15.1.

Листинг 15.1. Программа `color256`

```
program color256;
uses graph;
var
  gd,gm: SmallInt;
  x,y,i,j: SmallInt;
begin
  gd:=D8bit;
  gm:=m800x600;
  initgraph(gd,gm,'');
  for i:=0 to 15 do
    for j:=0 to 15 do
    begin
      x:=i*40;
```

```
      y:=j*20;
      SetfillStyle(SolidFill,i*16+j);
      Bar(x,y,x+40,y+20);
    end;
  readln;
  closegraph;
end.
```

Эта программа последовательно воспроизводит прямоугольники размером 40×20 пикселов, раскрашивая их в цвет, соответствующий содержимому очередного регистра ЦАП. Результат ее работы представлен на рис. 15.2. Нумерация цветовых оттенков продвигается от 0 до 255. В каждом столбце представлены цвета 16 последовательных регистров ЦАП.

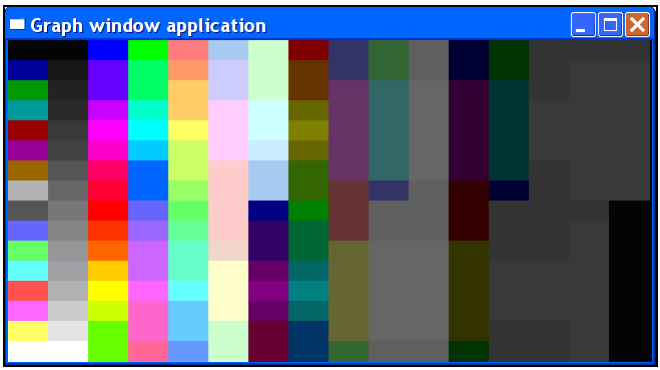


Рис. 15.2. Цветовая палитра по умолчанию

В регистре ЦАП с нулевым номером находится нулевой код, соответствующий тройке минимальных интенсивностей RGB-компонентов — (0, 0, 0). На экране этому набору соответствует черный цвет. В регистре с номером 1 находится тройка (0, 0, 63), соответствующая максимальной интенсивности синего цвета. Регистр с номером 2 содержит тройку (0, 63, 0), и его содержимое воспроизводится зеленым цветом. В следующем регистре "смешаны" максимальные значения синего и зеленого цветов — (0, 63, 63), соответствующие бирюзовому цвету. Содержимое четвертого регистра равно (63, 0, 0) и отображается красным цветом, и т. д. Для обозначения первых 16 цветовых оттенков можно использовать имена мнемонических констант, приведенные в табл. 15.4.

Таблица 15.4

Номер цвета	Цвет	Константа	Номер цвета	Цвет	Константа
0	Черный	`black`	1	Синий	`blue`

Таблица 15.4 *(окончание)*

Номер цвета	Цвет	Константа	Номер цвета	Цвет	Константа
2	Зеленый	green	9	Светло-синий (голубой)	lightblue
3	Бирюзовый (цианový)	cyan	10	Светло-зеленый	lightgreen
4	Красный	red	11	Светло-цианový	lightcyan
5	Малиновый	magenta	12	Светло-красный	lightred
6	Коричневый	brown	13	Светло-малиновый	lightmagenta
7	Светло-серый	lightgray	14	Желтый	yellow
8	Темно-серый	darkgray	15	Белый	white

Подбор других цветов из палитры по умолчанию обычно выполняется экспериментальным путем или с помощью цветовой гаммы, представленной на рис. 15.2.

Функции `GetColor` и `GetBkColor` в аргументах не нуждаются. Они возвращают значения типа `Word`, соответствующие номерам регистров ЦАП, приписанных в данный момент времени к цветам рисования и фона:

```
var
  fc, bc: Word;
  ...
  fc:=GetColor;
  bc:=getBkColor;
```

Для установки или опроса содержимого регистров палитры приходится использовать специальные структуры — записи типа `RGBRec` и `PaletteType`:

```
RGBRec = packed record
   Red : SmallInt;
   Green : SmallInt;
   Blue : SmallInt;
end;
PaletteType = record
   Size : LongInt;
   Colors : array [0..MaxColors] of RGBRec;
end;
```

Запись типа RGBRec представляет собой три двухбайтовых поля, предназначенных для хранения содержимого одного регистра ЦАП. Запись типа PaletteType позволяет хранить содержимое всех регистров палитры. Ее первое поле хранит размер палитры в байтах. Вслед за ним располагается массив, в котором размещается содержимое всех регистров палитры. Для режима D8bit значение MaxColors (максимальный номер доступного цвета) равно 255.

```
var
  DACreg:  RGBRec;
  DACall:  PaletteType;
  max_reg: SmallInt;
  ...
max_reg:=GetPaletteSize;     // опрос макс. номера регистра палитры
GetPalette(DACall);          // опрос всех регистров текущей палитры
GetDefdaultPalette(DACall);    // опрос палитры по умолчанию
SetAllPalette(DACAll);         // установка всех регистров палитры
SetPalette(n_reg,DACreg);      // изменение регистра с номером n_reg
SetRGBPalette(n_reg,vR,vG,vB); // изменение регистра с номером n_reg
```

В процедуре SetRGBPalette значения RGB-компонентов должны принадлежать интервалу [0, 63].

Вообще говоря, особо мудрить с изменением палитры не стоит — 256 цветов палитры по умолчанию вполне достаточно для не очень мудреной BGI-графики.

15.4. Управление точками и фрагментами графического экрана

Каждая точка на графическом экране (пиксел) характеризуется координатами (*x*, *y*) и кодом цветности. Программа имеет возможность узнать или изменить код цветности любого пиксела:

```
var
  col:  Word;
  x, y: SmallInt;
  ...
col:=GetPixel(x,y);  // опрос цвета пиксела
...
PutPixel(x,y,col);   // изменение цвета пиксела
```

Помните, что под кодом цветности (значение переменной col) здесь по-прежнему выступает номер регистра ЦАП, по содержимому которого был или будет окрашен указанный пиксел.

Вывод на экран по одному пикселу может иметь смысл, когда общее количество точек сравнительно невелико. Даже на достаточно скоростном компьютере

(два ядра, тактовая частота более 2 ГГц) время заполнения экрана размером 800×600 составляет порядка 3—4 секунд. Гораздо больший интерес представляют процедуры группового копирования или восстановления пикселов, образующих на экране прямоугольный растровый фрагмент. Так как операция прямого обмена между байтами видеопамяти и оперативной памяти выполняется достаточно быстро, то таким способом можно мгновенно обновить содержимое графического экрана.

Перед копированием прямоугольного фрагмента, координаты которого задаются углами противоположных точек — (`x1`, `y1`) и (`x2`, `y2`), необходимо определить объем в байтах оперативной памяти, необходимый для запоминания информации об указанных пикселах экрана. Для этой цели предназначена функция `ImageSize`:

```
var
  x1, y1, x2, y2: SmallInt;
  size: LongInt;
...
size:=ImageSize(x1,y1,x2,y2);
```

Порядок задания координат важен — точка (`x1`,`y1`) должна соответствовать левому верхнему углу растрового фрагмента, точка (`x2`, `y2`) — правому нижнему. После определения размера буфера можно запросить память под соответствующий динамический массив и скопировать в него нужный фрагмент экрана:

```
var
  buf: array of Byte;          // динамический массив
  s_b: Longint;
...
s_b:=ImageSize(0,0,799,599);  // объем памяти под графический экран
buf:=SetLength(s_b);          // запрос памяти
GetImage(0,0,799,599,buf);    // копирование экрана
```

Обратная операция по копированию буфера оперативной памяти на графический экран выполняется с помощью процедуры `PutImage`:

```
PutImage(x1, y1, buf, mode);
```

В этом случае указываются только координаты левого верхнего угла фрагмента, которые могут и не совпадать с его местоположением в момент копирования. Размеры растрового изображения запоминаются в буфере при копировании фрагмента экрана, поэтому идентичность границ копировавшегося и восстанавливаемого участка экрана будет обеспечена. Последний аргумент процедуры `PutImage` определяет способ взаимодействия прежнего и нового кодов цветности. Обычно значение параметра `mode` задают с помощью одной из следующих мнемонических констант:

- `CopyPut` — новый код цветности заменяет прежний;
- `XORPut` — новый код цветности поразрядно складывается с прежним по модулю 2;

- `ORPut` — новый код цветности логически складывается с прежним;
- `ANDPut` — новый код цветности логически умножается на прежний;
- `NotPut` — новый код цветности инвертируется и заменяет прежний.

Приведенная в листинге 15.2 программа копирует прямоугольный фрагмент, расположенный в левом верхнем углу, и повторяет его справа от исходного изображения.

Листинг 15.2. Программа `image1`

```
program image1;
uses Graph;
var
  gd, gm:SmallInt;
  a: array of Byte; // динамический массив
                    // для хранения фрагмента (буфер)
  size: Byte;       // для размера буфера
begin
  gd:=D8bit;
  gm:=m800x600;
  InitGraph(gd,gm,'');
  Rectangle(10,10,80,60);
  Circle(45,35,20);
  size:=ImageSize(10,10,80,60);  // запрос размера буфера
  SetLength(a,size);             // запрос памяти под буфер
  GetImage(10,10,80,60,a[0]);    // копирование изображения в буфер
  PutImage(100,10,a[0],CopyPut); // копирование буфера на экран
  readln;
  CloseGraph;
end.
```

Результат ее работы приведен на рис. 15.3.

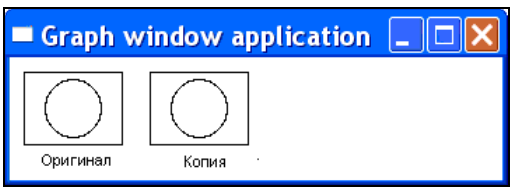


Рис. 15.3. Размножение фрагмента

15.5. Построение прямых и прямоугольников

Перечень процедур, с помощью которых осуществляется построение отрезков прямых и прямоугольников, приведен в табл. 15.5.

Таблица 15.5

Формат обращения к процедуре	Выполняемое действие
`Line(x1,y1,x2,y2);`	Построение отрезка прямой между двумя точками
`LineTo(x2,y2);`	Построение отрезка прямой. Начало отрезка — текущая точка (CP), конец — точка (`x2`, `y2`)
`LineRel(dx,dy);`	Построение отрезка прямой. Начало отрезка — текущая точка, конец — точка (`xCP+dx`, `yCP+dy`)
`MoveTo(x2,y2);`	Перемещение текущей точки в позицию (`x2`, `y2`)
`MoveRel(dx,dy);`	Перемещение текущей точки в позицию (`xCP+dx`, `yCP+dy`)
`Rectangle(x1,y1,x2,y2);`	Построение прямоугольника со сторонами, параллельными координатным осям
`SetLineStyle(Style,Pattern,width);`	Установка стиля воспроизведения линии
`DrawPoly(n,XY);`	Построение ломаной линии с вершинами в точках массива `XY` (`n` — число точек)

В графических построениях участвуют либо абсолютные координаты точек (`x1`, `y1`), (`x2`, `y2`), задаваемые значениями типа `SmallInt` в экранной системе координат, либо приращения (`dx`, `dy`) относительно позиции текущей точки. Абсолютные координаты могут быть и отрицательными. Это означает, что одна из точек графического примитива находится за пределами координатной сетки экрана. Но видимый фрагмент соответствующего графического примитива будет построен. Порядок задания угловых точек прямоугольника жестко фиксирован. Точка (`x1`, `y1`) соответствует левому верхнему углу прямоугольника.

После открытия графики или очистки графического экрана текущая точка (графический курсор) находится в начале координат — в левом верхнем углу экрана. При построении линий текущая точка перемещается в конец отрезка. Построение прямоугольника не изменяет позицию текущей точки. В этом можно убедиться на следующем примере (листинг 15.3).

Листинг 15.3. Программа CP

```
program CP;
uses graph;
var
  gd,gm: SmallInt;

begin
  gd:=D8bit;
  gm:=m800x600;
  initgraph(gd,gm,'');
  Rectangle(20,20,100,100);
  LineTo(100,150);
  LineRel(50,0);
  readln;
  closegraph;
end.
```

Соответствующее графическое окно приведено на рис. 15.4.

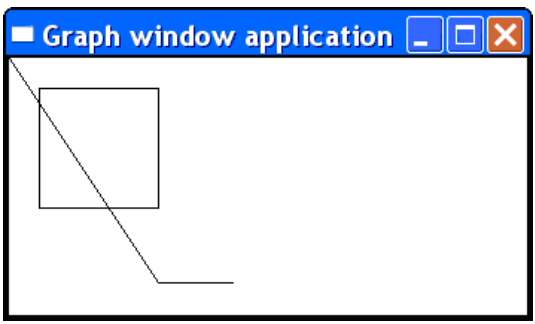


Рис. 15.4. Перемещения текущей точки

Отрезки прямых и контуры прямоугольника могут рисоваться не только сплошными линиями. По умолчанию в графической системе установлен именно этот режим. Однако пользователь может его изменить, обратившись к процедуре `SetLineStyle`.

```
SetLineStyle(Style, Pattern, width);
```

Первый ее параметр `Style` обычно задается одной из следующих мнемонических констант:

- `SolidLn` — режим построения сплошных линий (по умолчанию);
- `DottedLn` — режим построения пунктирной линии;
- `CenterLn` — режим построения штрихпунктирной линии;

- `Dashedln` — режим построения штриховой линии;
- `UserBitln` — режим построения линий по шаблону пользователя.

В первых четырех режимах параметр `Pattern` игнорируется и может быть задан, например, нулем. Его назначение — задать шаблон пользователя, который формируется в виде двухбайтовой шестнадцатеричной константы. Шаблон сканируется слева направо. Его очередной нулевой разряд делает очередной пиксел линии невидимым, а единичный разряд — видимым. После исчерпания разрядов шаблона сканирование его битов повторяется заново. Образцы воспроизведения стандартных типов линий приведены на рис. 15.5. В качестве нестандартного шаблона была использована константа `Pat=$FFFFFF00`.

Третий аргумент процедуры `SetLineStyle` задает толщину линии, которая может равняться одному пикселу (`width=NormWidth`) или двум пикселам (`width=ThickWidth`).

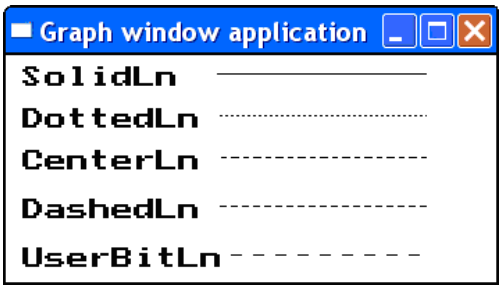


Рис. 15.5. Стили линий

Процедура `DrawPoly` предназначена для построения ломаной линии, вершины которой заданы массивом точек типа `PointType`. Если координаты последней точки массива совпадают с координатами начальной точки, то таким способом можно построить любой многоугольник (листинг 15.4).

Листинг 15.4. Программа `polygon`

```
program polygon;
uses graph;
var
  gd,gm:SmallInt;
  xy:array [1..4] of PointType;
begin
  xy[1].x:=10;   xy[1].y:=10;
  xy[2].x:=100;  xy[2].y:=10;
  xy[3].x:=80;   xy[3].y:=60;
  xy[4].x:=10;   xy[4].y:=10;    // замыкание многоугольника
  gd:=D8bit;
```

```
  gm:=m800x600;
  initgraph(gd,gm,'');
  DrawPoly(4,xy);                  // построение треугольника
  readln;
  CloseGraph;
end.
```

Графический экран этой программы приведен на рис. 15.6.

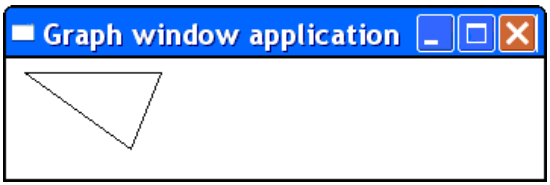


Рис. 15.6. Построение треугольника

Приведенные рис. 15.1, 15.3—15.6 построены искусственно, т. к. стандартно установленный режим рисования воспроизводит все графические элементы белым цветом на черном фоне. Поэтому настоящие графические окна соответствующих программ были обработаны в программе Paint при помощи обращения цвета на выделенных участках рабочего поля.

Дело в том, что попытка рисовать на графическом экране черным по белому не решается простым изменением цвета рисования (`fc:=0`) и цвета фона (`bc:=15`). При включении в приведенную выше программу перед обращением к процедуре `DrawPoly` двух следующих операторов:

```
SetColor(0);     // код черного цвета
SetBkColor(15);  // код белого цвета
```

содержимое графического экрана не меняется. Попытка проявить цвет фона путем стирания содержимого экрана (процедура `ClearDevice`) тоже ни к чему хорошему не приводит — экран становится белым, а рисование цветом фона по белому следов не оставляет. Поэтому приходится идти на "обман" графической системы: код нулевого цвета переносим в регистр ЦАП с номером, отличным от 0. Затем назначаем этот цвет в качестве цвета переднего плана и меняем цвет фона (листинг 15.5).

Листинг 15.5. Рисование черным по белому

```
program poly_B_W;
uses graph;
var
  gd,gm: SmallInt;
  xy: array [1..4] of PointType;
```

```
begin
  xy[1].x:=10;   xy[1].y:=10;
  xy[2].x:=100;  xy[2].y:=10;
  xy[3].x:=80;   xy[3].y:=60;
  xy[4].x:=10;   xy[4].y:=10;
  gd:=D8bit;
  gm:=m800x600;
  initgraph(gd,gm,'');
  SetPalette(1,0);  // черный цвет -> 1-й регистр ЦАП
  SetColor(1);      // назначение цвета рисования
  SetBkColor(15);   // смена цвета фона
  ClearDevice;      // проявление цвета фона
  DrawPoly(4,xy);
  readln;
  CloseGraph;
end.
```

После этих манипуляций перед нами графический экран, вывод которого на принтер не приводит к повышенному расходу тонера (см. рис. 15.6).

15.6. Построение окружностей, эллипсов и дуг

Список процедур, предназначенных для работы с окружностями и эллипсами, приведен в табл. 15.6.

Таблица 15.6

Формат обращения к процедуре	Выполняемое действие
`Arc(x,y,a1,a2,R);`	Построение дуги окружности
`Circle(x,y,R);`	Построение окружности
`Ellipse(x,y,a1,a2,Rx,Ry);`	Построение эллипса или его дуги
`GetArcCoords(vACT);`	Опрос координат последней дуги окружности или эллипса

Точка с координатами (`x`, `y`) задает центр окружности или эллипса. Так как значения координат имеют тип `SmallInt`, то они могут быть отрицательными. Это означает, что центр может находиться за пределами графического экрана, но видимые части соответствующего контура будут нарисованы только в области экрана. Угол `a1` определяет начало дуги и задается в градусах, которые отсчитываются от

направления оси *x* против часовой стрелки. Угол `a2` определяет конечную точку дуги (построение всегда ведется от `a1` к `a2`). Оба угла должны быть неотрицательными (значения типа `Word`). Допускаются комбинации `a1>a2` или `a1<a2`, обе они приводят к построению одной и той же дуги. Параметр `R` задает радиус окружности в пикселах.

Для эллипса или его дуги задаются два радиуса, соответствующие полуосям эллипса, направленным вдоль координатных осей.

С помощью процедуры `GetArcCoords` можно определить координаты граничных точек последней построенной дуги или окружности. Аргументом этой процедуры является имя записи типа `ArcCoordsType`:

```
type ArcCoordsType = record
  x: SmallInt;       // Координата X центра
  y: SmallInt;       // Координата Y центра
  xstart: SmallInt;  // Координата X начала дуги
  ystart: SmallInt;  // Координата Y начала дуги
  xend: SmallInt;    // Координата X конца дуги
  yend: SmallInt;    // Координата Y конца дуги
```

Пример построения окружности, эллипса и их дуг, приведенный на рис. 15.7, выполнен по программе из листинга 15.6.

Листинг 15.6. Построение окружностей, эллипсов и дуг

```
program arc_crc_ell;
uses graph;
var
  gd,gm: SmallInt;
begin
  gd:=D8bit;
  gm:=m800x600;
  initgraph(gd,gm,'');
  Circle(100,100,40);
  Line(100,100,200,100);
  Line(100,100,180,20);
  Arc(100,100,0,45,75);
  Arc(100,100,45,360,80);
  Ellipse(300,100,0,360,90,60);
  Ellipse(300,100,0,45,95,65);
  Line(300,100,420,100);
  Line(300,100,380,20);
  readln;
  CloseGraph;
end.
```

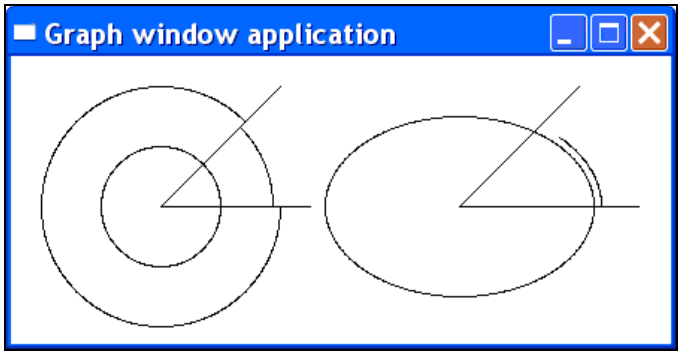


Рис. 15.7. Построение дуг, окружностей и эллипсов

15.7. Закраска и заполнение замкнутых областей

В этом разделе речь пойдет о закрашивании замкнутых областей любым цветом или их заполнение фиксированным цветным узором — шаблоном заливки. Шаблон заливки задается массивом 8×8 бит, где каждый единичный разряд окрашивается указанным цветом, а каждый нулевой разряд — цветом фона. Представьте себе, что весь графический экран покрыт такими узорными квадратиками, примыкающими друг к другу по горизонтали и вертикали. Наложим на этот узор контур замкнутой области, и та часть узора, которая ограничена контуром, используется для окраски внутренней или внешней части нашей фигуры. Если шаблон заполнен только единичными разрядами, то закраска ведется сплошным цветом. Наличие единичных разрядов только в отдельных строках шаблона (например, хотя бы в одной) приводит к горизонтальной штриховке области. Варьирование комбинаций нулей и единиц в шаблоне позволяет создавать довольно много различных узоров — до 32×63.

В графической библиотеке BGI предусмотрено 13 стандартных шаблонов заливки, приведенных в табл. 15.7. Для использования любого из них достаточно указать нужный номер или соответствующую мнемоническую константу. Установка того или иного стиля заливки выполняется с помощью процедуры `SetFillStyle`:

```
SetFillStyle(v_FST);
```

Аргументом этой процедуры является запись типа `FillSettingsType`, содержащая всего два поля:

```
type FillSettingsType = record
  pattern: Word;     // номер шаблона
  color: Word;       // код цветности
end;
```

Таблица 15.7

Номер шаблона	Константа	Способ заполнения
0	`EmptyFill`	Заливается цветом фона (все биты шаблона нулевые)
1	`SolidFill`	Заливается цветом `color`
2	`LineFill`	Штриховка горизонтальными линиями
3	`LtSlashFill`	Тонкая штриховка под 45°
4	`SlashFill`	Толстая штриховка под 45°
5	`BkSlashFill`	Толстая штриховка под 135°
6	`LtBkSlashFill`	Тонкая штриховка под 135°
7	`HatchFill`	Двойная штриховка (0° и 90°)
8	`XHatchFill`	Двойная штриховка (45° и 135°)
9	`InterleaveFill`	Короткие чередующиеся штрихи
10	`WideDotFill`	Редкий точечный растр
11	`CloseDotFill`	Густой точечный растр
12	`UserFill`	Заполнение шаблоном пользователя

Шаблон пользователя устанавливается с помощью процедуры `SetFillPattern`:

```
SetFillPattern(fill_pat,color);
```

Первый параметр представляет собой 8-байтовый массив, биты которого сформированы по желанию пользователя. Например, шаблон, напоминающий кирпичную стену, образованную окрашенными кирпичиками, может быть задан следующим образом:

```
var
  pat94:FillPatternType=($00,$FB,$FB,$FB,$00,$DF,$DF,$DF);
```

Около сотни нестандартных узоров вы можете найти в книге [20].

Процедуры и функции, используемые для работы с залитыми областями, приведены в табл. 15.8.

Таблица 15.8

Формат обращения к процедуре	Выполняемое действие
`Bar(x1,y1,x2,y2);`	Построение залитого прямоугольника
`Bar3D(x1,y1,x2,y2,depth,top);`	Построение параллелепипеда
`FloodFill(x,y,bord_col);`	Заливка внутренней или внешней части замкнутой области

Таблица 15.8 *(окончание)*

Формат обращения к процедуре	Выполняемое действие
`FillEllipse(x,y,Rx,Ry);`	Построение залитого эллипса
`FillPoly(n,xy);`	Построение залитого многоугольника
`GetFillPattern(fill_pat);`	Опрос текущего шаблона заливки
`GetFillSettings(v_FST);`	Опрос параметров заливки
`PieSlice(x,y,a1,a2,R);`	Построение залитого сектора окружности
`Sector(x,y,a1,a2,Rx,Ry);`	Построение залитого сектора эллипса
`SetFillPattern(fill_pat,color);`	Задание пользовательского шаблона
`SetFillStyle(v_FST);`	Установка режима заливки

Пример использования процедуры `Bar` в режиме заливки сплошным цветом приводился в *разд. 15.3*. Результат работы соответствующей программы показывает, что цветной прямоугольник строится без обводки границ. Чтобы построить цветной прямоугольник с четко очерченной границей другого цвета, надо обратиться к двум последовательным процедурам (листинг 15.7).

Листинг 15.7. Программа `fill1`

```
program fill1;
uses graph;
var
  gd,gm: SmallInt;
  pat94: FillPatternType=($00,$FB,$FB,$FB,$00,$DF,$DF,$DF);
begin
  gd:=D8bit;
  gm:=m800x600;
  initgraph(gd,gm,'');
  SetFillPattern(pat94,4);   // назначение шаблона пользователя
  SetFillStyle(UserFill,4);  // установка режима заливки
  Bar(10,10,100,100);        // отображение прямоугольника без границы
  Rectangle(9,9,101,101);    // обводка границы
  readln;
  CloseGraph;
end.
```

Результат ее работы в истинных цветах представлен на рис. 15.8.

Рис. 15.8. Обводка границ прямоугольника

Процедура `Bar3D` ориентирована, главным образом, на построение трехмерных столбчатых диаграмм. В графиках такого рода основным элементом является либо однородный цветной столбик с "крышей", либо столбик без "крыши", над которым можно надстроить еще один или несколько столбиков с другой раскраской. Обращение к процедуре `Bar3D` требует задания следующих параметров:

```
Bar3D(x1, y1, x2, y2, depth, top);
```

Первые четыре аргумента задают угловые точки передней панели столбика. Параметр `depth` определяет глубину столбика и иногда его рекомендуют выбирать равным четверти ширины (`deph=0.25*(x2-x1)`). Естественно, что все эти параметры целочисленные и неотрицательные, задаются они в пикселах. Последний аргумент `top` — логического типа. Если он равен `True`, то крыша столбика рисуется, в противном случае столбик воспроизводится без крыши, и это дает возможность надстроить над ним следующий столбик и не заботиться об удалении невидимых линий. Приведенный в листинге 15.8 пример демонстрирует возможность воспроизведения одиночного (с крышей) и составного столбика (нижний — без крыши, верхний — с крышей).

Листинг 15.8. Программа `bar3d_1`

```
program bar3d_1;
uses graph;
var
  gd,gm: SmallInt;
begin
  gd:=D8bit;
  gm:=m800x600;
  initgraph(gd,gm,'');
  SetColor(Green);
  SetFillStyle(HatchFill,Blue);
  Bar3d(20,20,40,100,10,true);
  SetFillStyle(SolidFill,Yellow);
  Bar3d(80,60,120,100,15,false);
  SetFillStyle(CloseDotFill,Red);
  Bar3d(80,20,120,60,15,true);
```

```
  readln;
  CloseGraph;
end.
```

Результат работы этой программы приведен на рис. 15.9.

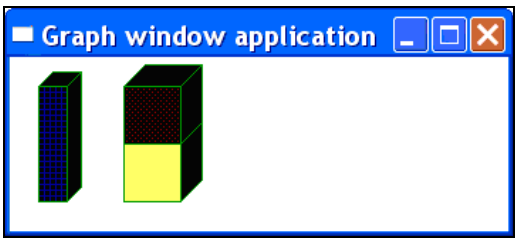


Рис. 15.9. Элементы столбчатых диаграмм

Иногда к процедуре `Bar3D` обращаются с нулевым значением параметра `depth`. В этом случае можно построить залитый прямоугольник с четко выделенной границей.

Процедура `FloodFill` предназначена для заливки или штриховки внутренней или внешней части замкнутого контура:

```
FloodFill(x, y, bord_col);
```

Параметр `bord_col` задает цвет пикселов, составляющих непрерывную границу области. Если точка с координатами (`x`, `y`) находится внутри области, то обработке подвергается внутренняя часть замкнутого контура. В противном случае окраска (штриховка) выполняется для внешней части контура. Разрыв границы контура даже на один пиксел приведет к тому, что краска (штриховка) "просочится" на противоположную часть контура и залитым окажется весь экран или большая его часть. Последнее может произойти, если на пути краски встретится еще один замкнутый контур с таким же цветом границы. Если цвет заливки, установленный в одной из процедур `SetFillPattern` или `SetFillStyle`, не совпадает с цветом границы, то цвет контура после работы процедуры `FloodFill` остается прежним. Однако оба указанных цвета могут совпасть, и тогда залитая область лишится ярко выраженного цвета границы. Повторная перекраска такой области с помощью процедуры `FloodFill` уже невозможна.

Процедура `FillEllipse` предназначена для построения залитого эллипса. При необходимости его граница может быть очерчена с помощью процедуры `Ellipse`.

Параметры процедуры `FillPoly` совпадают с аргументами процедуры `DrawPoly` (листинг 15.9).

Листинг 15.9. Программа `f_polygon`

```
program f_polygon;
uses graph;
```

```
var
  gd,gm: SmallInt;
  xy:    array [1..4] of PointType;
begin
  xy[1].x:=10;   xy[1].y:=10;
  xy[2].x:=100;  xy[2].y:=10;
  xy[3].x:=80;   xy[3].y:=60;
  xy[4].x:=10;   xy[4].y:=10;
  gd:=D8bit;
  gm:=m800x600;
  initgraph(gd,gm,'');
  SetColor(Green);
  SetFillStyle(SolidFill,Green);
  FillPoly(4,xy);
  readln;
  CloseGraph;
end.
```

Залитый по этой программе треугольник представлен на рис. 15.10.

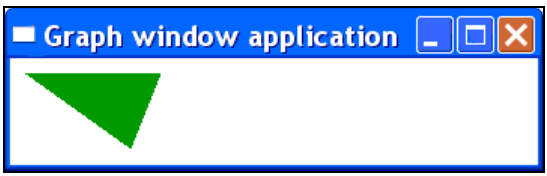


Рис. 15.10. Заливка многоугольника

Секторы окружностей или эллипсов, окрашиваемые с помощью процедур `PieSlice` и `Sector`, часто используются для построения круговых диаграмм (рис. 15.11).

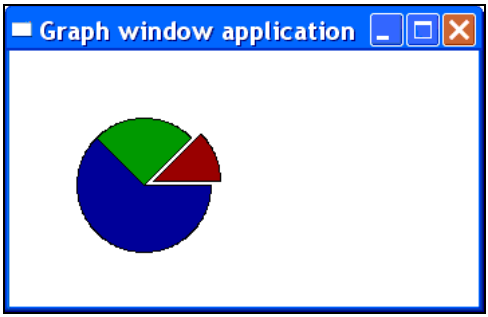


Рис. 15.11. Круговая диаграмма

Опрос текущих параметров заливки осуществляется с помощью процедур `GetFillSettings` и `GetFillPattern`.

15.8. Тексты на графическом экране

Сопровождение изображения пояснительными подписями на графическом экране расширяет возможности программы по представлению результатов работы. В BGI-графике для этой цели используются две процедуры:

```
OutText(msg);
OutTextXY(x, y, msg);
```

В первом случае положение строкового сообщения `msg` зависит от позиции текущей точки (графического курсора). Во втором случае точка привязки текста зависит от координат указанной точки (`x`, `y`). Любой шрифт, который можно использовать при формировании подписей, обладает такими характеристиками как высота и ширина заглавных символов. Если они одинаковы для любых символов алфавита, то шрифт принято называть моноширинным. Например, тексты программ в поле редактора FP набираются моноширинным шрифтом. Если ширины разных букв отличаются друг от друга, то такой шрифт называют пропорциональным. Примером такого шрифта является Times New Roman, которым набрана эта книга. Высота строчных букв составляет примерно 2/3 от высоты прописных букв.

Выбор шрифта и высоты его прописных символов осуществляется с помощью процедуры `SetTextStyle`:

```
SetTextStyle(n_Font, n_Dir, kH);
```

Первый аргумент указывает номер шрифта из диапазона [0, 10]. Он может быть задан либо конкретным числом, либо мнемонической константой (табл. 15.9).

Таблица 15.9

Номер шрифта	Константа	Файл	Пояснение
0	`DefaultFont`		Шрифт по умолчанию, растровый (8×8)
1	`TriplexFont`	trip.chr	Трехобводный, прямой, с засечками
2	`SmallFont`	small.chr	Однообводный, прямой, моноширинный
3	`SansSerifFont`	sans.chr	Двухобводный, прямой, моноширинный
4	`GothicFont`	goth.chr	Готический
5	`Script.chr`	scri.chr	Рукописный, курсив
6	`SimpleFont`	simp.chr	Двухобводный, прямой, пропорциональный
7	`TSGRFont`	tscr.chr	Трехобводный, курсив, с засечками

Таблица 15.9 *(окончание)*

Номер шрифта	Константа	Файл	Пояснение
8	LCOMFont	lcom.chr	
9	EuroFont	euro.chr	
10	BoldFont	bold.chr	Полужирный, прямой

Второй аргумент регулирует направление воспроизведения подписи — по горизонтали (hDir=0 или константе HorizDir) или по вертикали (hDir=1 или константе VertrDir). Третий параметр соответствует высоте символов. Его приходится подбирать, т. к. для разных шрифтов физические размеры букв при одном и том же значении kH могут отличаться. В любом случае, при увеличении kH от 1 до 10 размеры букв увеличиваются.

Рис. 15.12. Шрифт по умолчанию

Рис. 15.13. Шрифт TiplexFont

Шрифт по умолчанию (рис. 15.12) встроен в системный модуль и не содержит русских букв. Однако для всех остальных шрифтов в каталоге \FPC\...\bin должны находиться файлы, указанные в табл. 15.9. В состав стандартной поставки они не входят, но их можно скопировать из любой ранней версии Turbo Pascal или Borland Pascal. Правда, при этом у вас не окажется полноценных шрифтов, дополненных русским алфавитом. Для поиска последних можно воспользоваться Интернетом, хотя качество многих "расширенных" шрифтов иногда оставляет желать лучшего — в некоторых отсутствуют буквы "Ё" и "ё", не очень аккуратно подобраны пробелы после отдельных символов. Больше других повезло шрифту Triplex Font

(рис. 15.13). В рукописном шрифте (рис. 15.14) не оказалось букв "Ё" и "Ъ". Шрифт Simple Font (рис. 15.15) всем хорош, но подпадает под недавно принятый Государственной думой закон "О государственном языке Российской Федерации" в части использования буквы "Ё".

Рис. 15.14. Рукописный шрифт

Рис. 15.15. Шрифт Simple Font

По умолчанию точкой привязки текста является левый верхний угол прямоугольника, окаймляющего текст (рис. 15.16 — левый прямоугольник). Он чуть-чуть шире текста за счет пробела после последней буквы. Низ прямоугольника тоже находится чуть ниже базовой линии расположения букв за счет того, что у некоторых символов имеются подстрочные фрагменты (например, у букв "g", "q", "p", "y"). Существует 9 вариантов расположения точки привязки текста относительно окаймляющего его прямоугольника. Эти позиции обозначены парой индексов, задающих условные смещения по горизонтали и вертикали относительно углов габаритного прямоугольника (см. правый прямоугольник на рис. 15.16). За счет задания смещения по вертикали текст может быть смещен вверх относительно габаритного прямоугольника на $0.5 \times h$ (второй индекс равен 1) или на h (второй индекс равен 2), где h — высота прописной буквы. Соответствующий сдвиг текста влево по горизонтали равен $0.5 \times w$ (первый индекс равен 1) или w (первый индекс равен 2), где w — ширина текста. Таким образом, точку привязки текста можно связывать с левой границей, с его серединой или с правой границей текста.

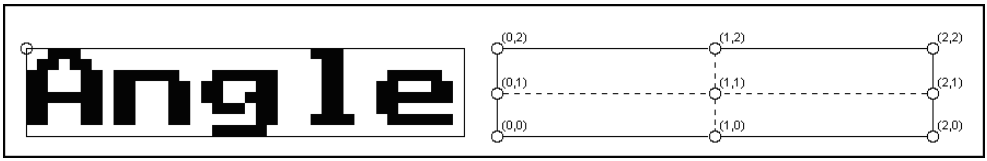


Рис. 15.16. Выбор точки привязки текста

Установка горизонтальных и вертикальных смещений текста относительно габаритного прямоугольника осуществляется с помощью процедуры `SetTextJustify`:

```
SetTextJustify(horiz, vert);
```

Горизонтальное смещение (`horiz`) может быть равно 0 (или константе `LeftText`), 1 (или константе `CenterText`) или 2 (или константе `RightText`). Для вертикального смещения (`vert`) заготовлены три аналогичные константы — `BottomText`, `CenterText`, `TopText`.

Если целочисленный масштабный коэффициент `kH`, задаваемый в процедуре `SetTextStyle`, одновременно изменяет высоту и ширину символов, то с помощью процедуры `SetUserCharSize` можно задать два разных коэффициента, независимо друг от друга влияющих на каждую из этих характеристик. Каждый из таких коэффициентов задается парой целых чисел — множителя и делителя:

```
SetUserCharSize(Multx, Divx, Multy, Divy);
```

Такой способ позволяет устанавливать любой вещественный коэффициент изменения размеров символов по той или иной координате, например равный 3.14 (`Mult=314`, `Div=100`).

Среди полезных функций по управлению текстами отметим две — `TextWidth` и `TextHeight`, позволяющие определить ширину и высоту заданного сообщения. Эти значения могут помочь правильно разместить соответствующее сообщение на графическом экране. В частности, с их помощью было получено изображение текста в габаритном прямоугольнике, приведенное на рис. 15.16 (листинг 15.10).

Листинг 15.10. Программа `text_1`

```
program text_1;
uses Graph;
var
  gd,gm: SmallInt;
  h,w:   Word;
begin
  gd:=D8bit;
  gm:=m800x600;
  InitGraph(gd,gm,'');
  SetTextStyle(0,0,10);
  h:=TextHeight('Angle');
```

```
  w:=TextWidth('Angle');
  MoveTo(300,100);
  Circle(300,100,5);
  Rectangle(300,100,300+w,100+h);
  OutText('Angle');
  readln;
  CloseGraph;
end.
```

15.9. Выделение локальной области на графическом экране

На графическом экране можно создать временное поле рисования с помощью процедуры `SetViewPort`:

```
SetViewPort(x1, y1, x2, y2, Clip);
```

Координаты точек (`x1`, `y1`) и (`x2`, `y2`) задают левый верхний и правый нижний углы прямоугольной области, которая с этого момента становится полем рисования. В момент создания такой локальной области она чистится, а курсор переводится в начало локальной системы координат, т. е. в точку (`x1`, `y1`) полного графического экрана. Параметр `Clip` может принимать одно из двух значений — `True` или `False`. В первом случае включается *режим отсечения*, при котором результаты построений, выходящие за пределы установленной локальной области, на экране не отображаются. Во втором случае объекты, выходящие за пределы локального окна, на графическом экране рисуются (рис. 15.17) — листинг 15.11.

Листинг 15.11. Программа `viewport`

```
program viewport;
uses Graph;
var
  gd,gm: SmallInt;
begin
  gd:=D8bit;
  gm:=m800x600;
  InitGraph(gd,gm,'');
  SetViewPort(50,50,150,100,true);
  Rectangle(0,0,100,50);
  Circle(50,25,30);
  SetViewPort(200,50,300,100,false);
  Rectangle(0,0,100,50);
```

```
  Circle(50,25,30);
  readln;
  CloseGraph;
end.
```

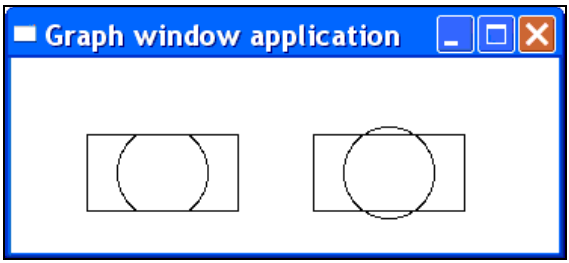
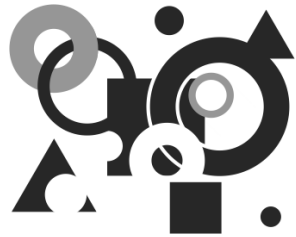


Рис. 15.17. Слева — область с отсечением, справа — без отсечения

Для очистки локальной области рисования используется процедура `ClearViewPort`.

Глава 16



OpenGL

16.1. Немного истории

Графический стандарт OpenGL (Open Graphics Library — открытая графическая библиотека) базируется на разработке фирмы Silicon Graphics. В 1992 г. он был принят ведущими производителями программного и аппаратного обеспечения. В настоящее время пакет OpenGL включается в состав операционных систем наиболее распространенных средств вычислительной техники.

Самой важной особенностью пакета OpenGL является его универсальность — изображение, описанное с помощью процедур и функций пакета, на экране любого компьютера выглядит практически одинаково, независимо от используемых программных или аппаратных средств. Однако эта универсальность привела к тому, что основная функциональность пакета OpenGL направлена на создание описаний графических 2D- и 3D-объектов, манипуляций ими и поддержку процессов визуализации их статических или динамических изображений. Сам пакет OpenGL не поддерживает связь с клавиатурой, мышью или джойстиками, что может оказаться полезным для управления графическими программами не только игрового характера.

Кроме того, в рамках каждой операционной системы пакет OpenGL нуждается в индивидуальной настройке, связанной с заданием множества параметров, режимов работы и т. п. Поэтому вокруг этой библиотеки создаются различные сервисные средства, расширяющие функциональные возможности пакета OpenGL и упрощающие его эксплуатацию на той или иной платформе.

Наиболее известной разработкой такого рода является утилита GLUT (OpenGL Utility Toolkit), реализованная в среде Windows в виде динамической библиотеки glut.dll или glut32.dll[1]. При работе с пакетом OpenGL довольно часто используют одно из его расширений — библиотеку GLU. В ее состав входит довольно много процедур построения трехмерных объектов — цилиндров, конусов, сфер, икосаэд-

[1] Найти последние версии этих библиотек можно на сайте OpenGL.org (см. **http://www.opengl.org/resources/libraries/glut/glut_downloads.php**). Последняя доступная версия на момент написания книги была 3.7beta. Данные библиотеки для удобства читателей помещены на компакт-диск *(см. приложение 4)*.

ров, октаэдров, додекаэдров. И даже чайника, иллюстрации которого с различными световыми эффектами кочуют по всем публикациям, так или иначе связанным с OpenGL. Для построения всех этих тел в программе достаточно написать единственную строку (вызов соответствующей процедуры). На экране можно построить не только изображение твердотельной модели, но и ее проволочный каркас с той или иной степенью детализации поверхности.

О пакете OpenGL написано довольно много книг, но они ориентированы на его использование в программах на языках C, C++. Применению OpenGL в приложениях Delphi посвящена единственная книга М. В. Краснова, опубликованная издательством "БХВ-Петербург" в 2001 г. [31]. В рамках этой главы мы сделали попытку познакомить читателей с основными понятиями современного графического пакета и продемонстрировать несколько небольших программ, которые можно использовать как трамплин в достаточно актуальный раздел информатики.

16.2. Чуть-чуть о математике и физике в машинной графике

Когда мы пишем в программе оператор `y:=sin(x);`, нам и в голову не приходит, что для реализации этого действия в компьютере используются фундаментальные математические понятия, изучаемые в разделах математического анализа и методов вычислений. На самом нижнем уровне вычислений компьютер выполняет очень примитивные арифметические и логические микрооперации, из которых создаются микропрограммы, реализующие команды более высокого уровня. Команды второго уровня образуют аппаратно-зависимую компоненту языка ассемблера. Среди низкоуровневых арифметических операций присутствуют только простейшие действия — сложение, вычитание, умножение и деление. Поэтому алгоритм вычисления значения функции `sin(x)` должен быть сведен именно к ним. Достигается это за счет использования разложения функции в ряд Тейлора, который для функции `sin(x)` имеет следующий вид:

$$\sin(x) = x - \frac{x^2}{2!} + \frac{x^3}{3!} - \frac{x^5}{5!} + \ldots$$

Ряд этот сходится при любом аргументе `x`, но чем больше аргумент, тем большее количество членов ряда приходится вычислять для достижения необходимой точности. Если аргумент превышает 2π, то в силу периодичности функции из аргумента `x` можно вычесть число, кратное 2π. Затем можно воспользоваться формулами приведения аргумента, которые изучаются в школе, и заменить вычисление `sin(x)` на вычисление `-sin(π-x)`. Если понадобится дальнейшее уменьшение аргумента, то можно заменить вычисление `sin(x)` на `cos(0.5*π-x)`. Мы привели эти пространные рассуждения, чтобы показать, почему более полувека назад одна из дипломных работ, выполненных на мехмате МГУ, была посвящена созданию эф-

фективной подпрограммы ЭВМ "Стрела" для вычисления синуса с приведением аргумента к углу, меньшему, чем $\pi/4$.

Сегодня очень небольшое число программистов, использующих алгоритмические языки высокого уровня, догадывается о проблемах вычисления элементарных функций. В лучшем случае они знают, что среди машинных команд сопроцессора (а точнее, среди соответствующих микропрограмм самого нижнего уровня) есть операции, используемые компиляторами.

В настоящем разделе мы не собираемся подробно излагать математические и физические основы, заложенные в современных графических системах (и в OpenGL в том числе) и мощных графических ускорителях. Но с некоторой терминологией и идеями повышения ускорения и наглядности графических операций познакомиться полезно.

Технология визуализации двумерных и трехмерных объектов использует довольно много специальных терминов, характерных для большинства систем машинной графики. С наиболее важными из них, влияющими на свойства отображаемых фигур, мы познакомимся в начале этой главы. Появлению изображения на экране предшествует довольно много процедур, связанных с вычислением цветности каждого пиксела изображения — учет положения и специфики источников света, определение видимых участков объекта и учет влияния их физических характеристик (отражение, преломление, диффузия). Все это вместе взятое составляет суть *рендеринга* — процесса визуализации на экране компьютерной модели.

16.2.1. Аффинные преобразования и однородные координаты

Основным назначением библиотеки OpenGL является создание графических моделей трехмерных объектов, выполнение различных геометрических преобразований над такими моделями, визуализация моделей, различных их сечений и проекций. Один из центральных компонентов рендеринга составляют различные линейные преобразования координат характерных точек отображаемого объекта, обычно вершин граней — сдвиги, повороты, растяжения/сжатия, параллельное или центральное проектирование на ту или иную плоскость. Перечисленные преобразования образуют группу *аффинных преобразований*.

Для отображения трехмерных объектов формулы, связывающие значения преобразованных координат ($x_{\text{нов}}$, $y_{\text{нов}}$, $z_{\text{нов}}$) с их прежними значениями ($x_{\text{ст}}$, $y_{\text{ст}}$, $z_{\text{ст}}$), имеют вид:

$$\begin{aligned} x_{\text{нов}} &= a_{11} \times x_{\text{ст}} + a_{12} \times y_{\text{ст}} + a_{13} \times z_{\text{ст}} + b_1; \\ y_{\text{нов}} &= a_{21} \times x_{\text{ст}} + a_{22} \times y_{\text{ст}} + a_{23} \times z_{\text{ст}} + b_2; \\ z_{\text{нов}} &= a_{31} \times x_{\text{ст}} + a_{32} \times y_{\text{ст}} + a_{33} \times z_{\text{ст}} + b_3. \end{aligned} \qquad (1)$$

Для того чтобы записать указанные равенства в матричной форме $\mathbf{P}_{\text{нов}} = \mathbf{A} \times \mathbf{P}_{\text{ст}}$, используют следующий прием.

В 3D-графике к трем естественным координатам точки добавляют четвертую, всегда равную 1:

```
[x; y; z] → [x; y; z; 1]
```

Тогда описанные в формате (1) аффинные преобразования приобретают следующий вид:

$$\begin{pmatrix} x_{\text{нов}} \\ y_{\text{нов}} \\ z_{\text{нов}} \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x_{\text{ст}} \\ y_{\text{ст}} \\ z_{\text{ст}} \\ 1 \end{pmatrix}. \quad (2)$$

Расширенные таким образом координаты известны под названием *однородных координат*.

Может показаться, что введение однородных координат увеличивает объем исходных данных. Но это не так — на самом деле, искусственно добавляемую единицу можно хранить не в оперативной памяти, а "в уме", используя ее только в момент выполнения преобразований. Очень важно понять, в чем заключается преимущество однородных координат.

Во-первых, для каждого элементарного аффинного преобразования можно довольно просто сформировать соответствующую матрицу. Например, при параллельном переносе трехмерного объекта на заданные смещения по каждой из координат ($x_{\text{нов}} = x_{\text{ст}} + dx$, $y_{\text{нов}} = y_{\text{ст}} + dy$, $z_{\text{нов}} = z_{\text{ст}} + dz$) эта матрица имеет вид:

$$\begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

Для сжатия/растяжения по каждой координате ($x_{\text{нов}} = x_{\text{ст}} \times k_x$, $y_{\text{нов}} = y_{\text{ст}} + k_y$, $z_{\text{нов}} = z_{\text{ст}} + k_z$) используется матрица:

$$\begin{pmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Особое место среди всех матриц аффинных преобразований занимает единичная матрица (англоязычное обозначение — Identity):

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

Она сохраняет исходные координаты ($x_{ст}$, $y_{ст}$, $z_{ст}$) без изменения и выступает в качестве начальной текущей матрицы $\mathbf{A}_1$ в цепочке любых аффинных преобразований.

Познакомиться с матрицами, описывающими различные аффинные преобразования на плоскости и в пространстве, можно, например, в книге В. Н. Порева "Компьютерная графика", опубликованной издательством "БХВ-Петербург" в 2002 г. [32]. От пользователя пакета OpenGL вовсе не требуется запоминания этих матриц, т. к. мнемонические названия соответствующих функций и процедур пакета говорят сами за себя:

- `glRotatef(5,0,0,1)` — поворот (rotate) на 5° вокруг вектора с компонентами (0, 0, 1), т. е. вокруг оси z;
- `glTranslatef(dx,dy,dz)` — параллельный перенос (translate) на заданные смещения координат;
- `glScalef(kx,ky,kz)` — сжатие/растяжение (scale) по каждой координате.

Второй аргумент в пользу однородных координат появляется, как только над графическим объектом нужно выполнить цепочку из двух или более преобразований с матрицами $\mathbf{A}_1$, $\mathbf{A}_2$, ... , $\mathbf{A}_n$. Вместо того чтобы производить пересчет координат на каждом шаге, можно вычислить матрицу итогового аффинного преобразования и только один раз пересчитать координаты всех вершин:

$$\mathbf{A} = \mathbf{A}_n \times \mathbf{A}_{n-1} \times ... \times \mathbf{A}_2 \times \mathbf{A}_1,$$

$$\mathbf{P}_{нов} = \mathbf{A}_n \times (\mathbf{A}_{n-1} \times (... \mathbf{A}_2 \times (\mathbf{A}_1 \times \mathbf{P}_{ст}) ...)) = \mathbf{A} \times \mathbf{P}_{ст}.$$

Умножение матриц 4-го порядка требует гораздо меньше операций, чем многократный пересчет всех вершин отображаемого многогранника. Это преимущество в объеме вычислений оправдывает использование однородных координат в векторных системах машинной графики.

Матричный подход весьма удобен еще и тем, что позволяет очень просто осуществить возврат к положению объекта, предшествующему цепочке выполняемых аффинных преобразований. Достаточно перед выполнением группы преобразований запомнить состояние текущей матрицы (т. е. запомнить 16 коэффициентов), а после завершения цепочки преобразований восстановить прежнее значение текущей матрицы.

Немаловажным аргументом в пользу этой технологии является и стереотипность выполнения аффинных преобразований — использование типовых шаблонов для формирования матриц каждого элементарного преобразования, умножение матриц 4-го порядка. Более того, набор "машинных команд" процессоров нового типа (команды типа MMX, SSE, SSE2, SSE3) предусматривает параллельное выполнение до четырех арифметических операций за один такт.

16.2.2. Растеризация векторных изображений

Такие устройства отображения, как дисплеи и принтеры, обычно используют растровую технологию воспроизведения графических объектов. При этом изо-

бражение состоит из большого количества отдельных цветных точек — пикселов. Однако информацию об отображаемом объекте гораздо выгоднее хранить в векторном формате — коэффициентов соответствующих аналитических уравнений, координат концов отрезков, координат вершин многоугольников и т. п. Во-первых, так экономится объем оперативной памяти. Во-вторых, такие операции как масштабирование и повороты фигур в векторном формате осуществляются без потери качества.

Поэтому одна из первых задач в системах отображения — *растеризация* — связана с преобразованием векторных данных в растровый формат. Она использует довольно простой математический аппарат, но ее лобовое решение приводит к большому объему вычислений, в которых значительная доля операций приходится на умножение (деление) вещественных чисел. Одним из первых исследователей, предложивших эффективный алгоритм вычисления координат точек растрового отрезка, проходящего через две заданные точки, с использованием только целочисленных операций сравнения и сложения (вычитания), был Брезенхейм (Bresenham J. E., 1965 г.). Позднее (1977) он же распространил подобную схему на вычисление растровых координат точек на дуге окружности. Существенно позже аналогичная схема была предложена и для растеризации дуг эллипсов. Перечисленные алгоритмы оказались настолько удачными, что их встраивают в аппаратуру современных графических ускорителей.

16.2.3. Воспроизведение утолщенных линий

Задача воспроизведения утолщенных линий на растровой сетке не такая уж и тривиальная, как может показаться. Лобовое ее решение путем построения нескольких симметричных эквидистант[1] относительно осевой линии приводит к появлению семейства мелких точек, сквозь которые просвечивают фоновые пикселы.

На вид толстых линий оказывает влияние и форма пишущего узла *(апертура пера)*, из-за которой на стыках прямоугольных отрезков могут наблюдаться различного рода зазубрины — выступы, впадины. Чтобы построить качественную прямоугольную рамку с толщиной линии `w`, приходится строить отрезки, концы которых выступают на `w/2` относительно координат осевой линии. Многие графические системы, включая и Windows GDI, предоставляют пользователю возможность задать такие характеристики пера, как стиль сочленения смежных отрезков *(joint style)* и тип окончания отрезка *(end cap)* — закругленный, перпендикулярный осевой, параллельный одной из осей координат в зависимости от наклона отрезка.

[1] Эквидистанта — равноудаленная линия, например, отстоящая на один или несколько пикселов от заданной линии.

16.2.4. Сглаживание зазубрин

При увеличенном изображении наклонных линий на экране можно наблюдать многочисленные изломы между смежными пикселами. Визуальному устранению таких дефектов содействует размещение в уголках изломов дополнительных пикселов, окрашиваемых полутоновыми цветами. Этот прием сглаживания контуров в англоязычной литературе получил название *antialiasing* (дословно — противодействие ступенчатости). В открывающейся операторной скобке языка OpenGL можно встретить аргументы, так или иначе связанные с этим приемом (англ. *smooth* — сглаживание):

- `glBegin(GL_POINT_SMOOTH)` — сглаживание углов квадратных точек;
- `glBegin(GL_LINE_SMOOTH)` — сглаживание контуров отрезков прямых;
- `glBegin(GL_POLYGON_SMOOTH)` — сглаживание контуров многоугольников.

16.2.5. Устранение невидимых частей изображения

Существует довольно много разных по сложности и быстродействию алгоритмов удаления невидимых точек, линий и поверхностей. Среди них метод прямой (от источника освещения) и обратной (от позиции наблюдателя) *трассировки лучей* (ray tracing), *излучательности* (radiosity), *сортировка по глубине*, *Z-буфер* и др.

Наиболее простая идея, которая нашла свое воплощение и в аппаратуре, заложена в алгоритмах *Z*-буфера. Название этих методов связано с выбором направления оси *z*, которая из картинной плоскости (т. е. плоскости экрана) направлена на наблюдателя. Считается, что его позиция достаточно удалена от экрана, и лучи, направленные от картинной плоскости в строну наблюдателя, можно рассматривать как пучок прямых, параллельных оси *z*. Для каждой точки объекта, расположенного перед картинной плоскостью, ее *Z*-координата совпадает с расстоянием от этой точки до плоскости экрана. Поэтому, если на одном и том же луче находятся две или более точки объекта, то видимой окажется та, которая расположена ближе к наблюдателю, т. е. та, которая имеет большую *Z*-координату. Поэтому сравнение *Z*-координат пикселов, проектируемых в одну и ту же точку на картинной плоскости, решает проблему видимости изображения.

На практике простейший алгоритм *Z*-буфера реализуется с помощью двух буферов одинакового размера. Один из них предназначен для хранения двумерной матрицы цветностей пикселов картинной плоскости. При использовании 24-битной кодировки RGB-компонентов и разрешении экрана 1024×768 для такой матрицы потребуется порядка 4 Мбайт. Второй буфер используется для хранения 32-разрядных целочисленных значений *Z*-координат точек отображаемого объекта. В начальный момент буфер цветности заполняется кодом фонового цвета, а буфер *Z*-координат — нулями. При анализе очередного пиксела изображения его *Z*-координата сравнивается с соответствующим расстоянием ранее обработанного

пиксела, уже записанного в Z-буфер. Если новое расстояние больше, то код цветности пиксела заносится в соответствующую позицию буфера цветности, а новая координата Z подменяет предыдущее Z-значение. В противном случае обрабатываемая точка считается невидимой.

16.2.6. Окрашивание граней полигональных моделей

Волновая теория света, ведущая свое начало от работ Христиана Гюйгенса (конец XVII века), считает, что свет представляет собой поток фотонов, испускаемых источником и несущих электромагнитные колебания разной частоты. Для человеческого глаза видимый поток света принадлежит диапазону от 380 нм (нижняя граница фиолетового излучения) до 780 нм (верхняя граница красного излучения).

Под интенсивностью I падающего луча понимают отношение энергии светового потока к площади освещаемой им поверхности. Часть падающих лучей отражается от поверхности, часть поглощается ею, часть проходит через поверхность и преломляется. Благодаря отраженным лучам мы видим освещаемые предметы.

Отражение световых лучей представляет собой довольно сложный процесс, на одной из границ которого находится *абсолютное зеркальное отражение* (specular reflection), а на другой — *полное диффузное рассеяние* (diffuse reflection). Зеркальное отражение имеет место при падении светового луча на идеально гладкую поверхность (помните из школьной физики — угол падения равен углу отражения). К очень гладким поверхностям мы относим такие, у которых шероховатость, образуемая впадинками и выпуклостями, имеет размеры меньшие, чем длина световой волны. Для таких поверхностей угол отражения можно считать равным углу падения, а интенсивность отраженного луча $I_{\text{отр}}$ прямо пропорциональной интенсивности падающего луча I:

$$I_{\text{отр}} = k_s \times I.$$

При встрече луча со слабо шероховатой поверхностью имеет место отклонение отраженного луча от идеального направления на угол α. В этом случае зависимость интенсивности $I_{\text{отр}}$ от интенсивности падающего луча I существенно нелинейна и должна учитывать степень шероховатости освещаемой поверхности. В системах машинной графики с целью уменьшения объема вычислений используют более простые эмпирические формулы. Одной из них является формула Фонга (Phong):

$$I_{\text{отр}} = k_s \times I \times \cos^p \alpha.$$

Показатель степени p (Фонг называл его *коэффициентом резкости бликов* — shininess coefficient) подбирается эмпирическим путем. Для идеального зеркала $p = \infty$, для металлических поверхностей коэффициент резкости бликов находится в интервале от 100 до 500, для обычных материалов его значение не превышает 100.

При очень большой шероховатости (такие поверхности называют матовыми) отражение лучей осуществляется равномерно во все стороны — происходит так

называемое диффузное рассеяние. Если обозначить через φ угол падения луча (угол между направлением светового потока и вектором нормали к освещаемой поверхности), то интенсивность отраженного луча, достигающего позиции наблюдателя, в случае диффузного рассеяния подчиняется закону Ламберта:

$$I_{\text{диф}} = I \times k_d \times \cos\varphi.$$

В интенсивность наблюдаемого излучения, кроме перечисленных выше двух компонентов $I_{\text{отр}}$ и $I_{\text{диф}}$, вносит свой вклад и *фоновая подсветка* (ambient light), создаваемая рассеянным светом, отраженным от других предметов — $I_{\text{фон}}$.

В описанных выше физических явлениях и приведенных формулах явно или неявно присутствует вектор нормали к освещаемой поверхности, построенный в точке падения луча. В машинной графике поверхности трехмерных тел заменяются полигональными моделями, сформированными, как правило, из треугольных или четырехугольных граней (patches). Так как каждая из таких граней принадлежит плоскости, описываемой уравнением $\mathbf{Ax} + \mathbf{By} + \mathbf{Cz} = 0$ с известными коэффициентами, то построение вектора нормали для внутренних точек грани и определение косинуса угла между парами векторов (нормаль, вектор падающего луча) или (нормаль, вектор в позицию наблюдателя) выполняется по довольно простым вычислительным схемам.

Самый примитивный алгоритм однотонного окрашивания граней, известный под названием *flat*, базируется на предположении, что все точки грани освещены одинаково. Это более или менее допустимо при бесконечно удаленном источнике света, когда лучи в световом потоке параллельны, а размеры граней достаточно малы. Обычно выбирается некоторая характерная точка на грани, например, аналог центра масс вершин, в ней вычисляется освещенность, приписываемая всем остальным точкам. Так как при переходе на соседнюю грань освещенность терпит разрыв, то на отображаемом объекте четко прорисовываются ребра и отчетливо видна фасеточная структура поверхности.

Для создания более реалистичного изображения полигональной поверхности Гуро (H. Gouraud — сотрудник Autodesk, один из разработчиков 3D Max) предложил алгоритм неравномерной закраски граней, основанный на применении билинейной интерполяции освещенности. В его схеме по сложным или упрощенным эмпирическим формулам вычисляют освещенность каждой вершины. Затем на отрезках, соединяющих две смежные вершины, для подсчета освещенности каждой точки ребра используют линейную интерполяцию. Обозначим через I_k освещенность вершины с координатами (x_k, y_k, z_k), а через I_{k+1} освещенность соседней вершины с координатами (x_{k+1}, y_{k+1}, z_{k+1}). Тогда параметрические уравнения ребра, соединяющего эти вершины, имеют вид:

$$\begin{aligned} x &= x_k + (x_{k+1} - x_k) \times t, \\ y &= y_k + (y_{k+1} - y_k) \times t, \\ z &= z_k + (z_{k+1} - z_k) \times t, \\ 0 &\le t \le 1. \end{aligned}$$

Для внутренней точки M ребра, определяемой параметром t_m, интенсивность освещенности определяется по формуле:

$$I_M = I_k + (I_{k+1} - I_k) \times t_m.$$

Аналогичный расчет производится и для всех точек смежного ребра этой же грани. Затем пара точек смежных ребер с одинаковым значением параметра t соединяется внутренней хордой и на каждой точке хорды повторно производится линейная интерполяция освещенности.

Если по алгоритму Гуро произвести расчет интенсивности отраженных лучей на смежной грани, то на стыке граней в точках общего ребра освещенность сохраняет свою непрерывность, однако ее первая производная терпит разрыв. Для устранения этого разрыва и создания более плавных цветовых переходов Фонг предложил интерполировать нормали к поверхности. Теперь они уже не образуют строго параллельные пучки на каждой грани, а рассыпаются в стороны наподобие иголок ежа, образуя плавный переход через ребра и вершины. Этот алгоритм связан с большими вычислительными затратами, но изображение поверхности при этом становится намного более реалистичным.

16.3. Графические примитивы языка OpenGL

Графические примитивы — это простейшие геометрические фигуры, из которых конструируются более сложные плоские и пространственные объекты. В состав языка OpenGL включены 10 примитивов, полный перечень которых приведен в табл. 16.1.

Структура любого примитива представляет собой набор точек (вершин), заданных своими координатами на плоскости или в пространстве и соединенных между собой тем или иным способом. Для объявления любой вершины используется оператор `Vertex`, к имени которого добавляется два или три символа — `Vertex2f`, `Vertex3i`, `Vertex4ub`, ... Первая цифра, добавляемая к имени оператора, указывает, сколько координат содержится в описании вершины: две (x, y), три (x, y, z) или четыре (x, y, z, 1). Следующие за ней одна или две буквы определяют машинный формат, в который должны быть преобразованы числовые значения координат:

- `b` — однобайтовое целое со знаком (b — от *byte*);
- `ub` — однобайтовое целое без знака (ub — от *unsigned byte*);
- `s` — двухбайтовое целое со знаком (s — от *short*);
- `us` — двухбайтовое целое без знака (us — от *unsigned short*);
- `i` — четырехбайтовое целое со знаком (i — от *int*);
- `ui` — четырехбайтовое целое без знака (ui — от *unsigned int*);
- `f` — четырехбайтовое вещественное (f — от *float*);
- `d` — восьмибайтовое вещественное (d — от *double*).

Аббревиатура суффиксов, приведенная выше, заимствована из обозначений числовых типов данных, характерных для языка программирования C.

Описание вершин любого примитива заключается в операторные скобки языка OpenGL — `glBegin` и `glEnd`. Открывающая операторная скобка в качестве аргумента использует тип примитива (табл. 16.1).

Таблица 16.1

Тип	Пояснение
`GL_POINTS`	**Точки.** Каждая вершина представляет отдельную точку
`GL_LINES`	**Отрезки прямых.** Последовательная пара точек определяет начало и конец очередного отрезка. Если задано нечетное количество точек, то последняя вершина игнорируется
`GL_LINE_STRIP`	**Ломаная линия.** Смежные точки последовательно соединяются друг с другом
`GL_LINE_LOOP`	**Замкнутая ломаная.** В дополнение к предыдущему последнее звено образуется соединением последней и первой точек
`GL_TRIANGLES`	**Треугольники.** Каждая тройка точек представляет собой вершины отдельного треугольника. Лишние точки в конце (одна или две) игнорируются
`GL_TRIANGLE_STRIP`	**Треугольники, связанные общей стороной.** Точки 1, 2 и 3 образуют первый треугольник, точки 2, 3 и 4 — второй треугольник и т. д.
`GL_TRIANGLE_FAN`	**Треугольники, связанные общей вершиной.** Точки 1, 2 и 3 образуют первый треугольник, точки 1, 3 и 4 — второй треугольник, точки 1, 4 и 5 — третий треугольник и т. д.
`GL_QUADS`	**Четырехугольники.** Каждые четыре точки представляют вершины очередного автономного четырехугольника. Лишние точки в конце (одна, две или три) игнорируются
`GL_QUAD_STRIP`	**Четырехугольники, связанные общей стороной.** Точки 1, 2, 3 и 4 образуют первый четырехугольник, точки 3, 4, 5 и 6 — второй, 5, 6, 7 и 8 — третий и т. д.
`GL_POLYGON`	**Выпуклый многоугольник.** Вершины соединяются друг с другом последовательно: первая со второй, вторая с третьей, ..., последняя с первой. Для неправильно заданных вершин (вогнутый многоугольник, имеет место пересечение ребер и т. п.) результат построения не предсказуем

Пример простейшего описания треугольника:

```
glBegin(GL_TRIANGLES);
  glVertex3f(0,0,1);
  glVertex3f(0,1,0);
  glVertex3f(1,0,0);
glEnd;
```

Вершины, участвующие в описании очередного примитива, могут иметь индивидуальные атрибуты — цвет, размер точки, признак сглаживания острых углов и др. Некоторые из этих характеристик должны предшествовать скобке `glBegin`, и это означает, что их влияние распространяется на все вершины примитива. Другие атрибуты можно употреблять внутри операторных скобок:

```
glPointSize(15);                // задание размера точки
glEnable(GL_POINT_SMOOTH);      // режим сглаживания углов точек
glBegin(GL_TRIANGLES);
  glColor3f(1,0,0);             // цвет первой вершины (красный)
  glVertex3f(0,0,1);
  glColor3f(0,1,0);             // цвет второй вершины (зеленый)
  glVertex3f(0,1,0);
  glColor3f(0,0,1);             // цвет третьей вершины (синий)
  glVertex3f(1,0,0);
glEnd;
glDisable(GL_POINT_SMOOTH);     // отмена режима сглаживания
```

Вы, наверное, уже обратили внимание на то, что операторы языка OpenGL начинаются с префикса `gl`. Аналогичное правило имеет место и для обозначения всех функций библиотек GLU (префикс `glu`) и GLUT (префикс `glut`).

16.4. Управление цветом

Библиотека OpenGL использует два разных подхода к окраске графических компонент. Первый, наиболее употребительный, основан на четырехкомпонентной цветовой палитре RGBA, в которой представлены интенсивности трех базовых цветов (R — красный, G — зеленый, B — синий) и уровень прозрачности (так называемый *альфа-канал*). В большинстве графических систем, эксплуатируемых под управлением Windows, для хранения целочисленных значений этих компонентов выделяется по 8 двоичных разрядов (и это соответствует формату TrueColor — богатейшей палитре с более чем 16 млн цветовых оттенков). В библиотеке OpenGL наряду с целочисленным форматом для задания интенсивностей базовых цветов широко используются вещественные значения из диапазона [0, 1]. Нулевое значение соответствует минимальной интенсивности соответствующего компонента цвета, единичное — максимальной. Уровень прозрачности дает возможность рассмотреть контуры изображений, поверх которых нарисованы новые объекты. Достигается это путем смешения в определенных пропорциях старых красок со вновь накладываемыми. Нулевое значение параметра альфа соответствует абсолютной прозрачности свежего мазка, а единичное значение — абсолютной непрозрачности. Когда речь идет об отображении объекта на экране дисплея, для хранения кода цветности каждого элемента изображения (пиксела) в оперативной памяти или в видеопамяти выделяется 32 бита.

Для установки режима RGB-цветности необходимо указать соответствующий признак в аргументе процедуры `glutInitDisplayMode`:

```
glutInitDisplayMode(GLUT_RGB or ...);
```

Задание цвета вершины графического примитива обычно устанавливается с помощью процедуры `glColor3f` или `glColor4f`:

```
glColor3f(vR,vG,vB);
glColor4f(vR,vG,vB,vA);
```

Значения всех аргументов должны принадлежать диапазону [0, 1]. В первом случае уровень прозрачности устанавливается равным 1.

Второй подход, используемый все реже и реже, предполагает работу с существенно более бедной палитрой, содержащей, например всего 256 цветовых оттенков. Но каждый из них для своего полноценного физического представления требует 18 двоичных разрядов. Вместо того чтобы приписывать каждому пикселу 18-битный код настоящей цветности, с ним связывают 8-битный индекс, по которому из отдельно хранящегося массива палитры можно извлечь полный 18-битный код цвета. Именно так была устроена аппаратура устаревших мониторов типа VGA, позволявшая более компактно хранить информацию о цвете изображения.

16.5. Системы координат

Процедуры OpenGL используют, как правило, пространственную систему координат, представленную на рис. 16.1. Ось *z* в ней направлена в сторону наблюдателя. Вспомните об идее *Z*-буфера, когда точка, имеющая большую координату *Z*, заслоняет точку с меньшей координатой *Z*. Именно так решается проблема удаления невидимых участков при отображении пространственной сцены на плоскость экрана. Диапазон изменения любой пространственной координаты — от –1 до +1. Если реальные размеры конструируемых фигур не попадают в указанный интервал, можно воспользоваться различными способами проецирования объекта, его перемещением в пространстве или масштабированием.

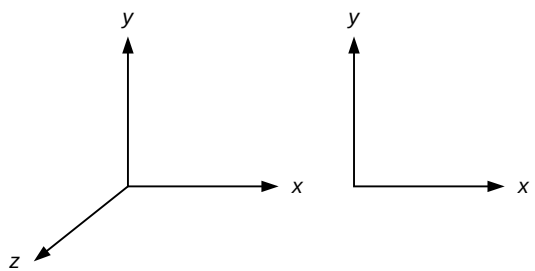


Рис. 16.1. Слева основная система пространственных координат, справа — система координат окна отображения

16.6. Основные аффинные преобразования

Отображаемые геометрические объекты или их фрагменты могут подвергаться различным преобразованиям типа перемещения в пространстве, вращения относительно заданного центра или заданной оси, растяжения или сжатия с индивидуальными или одинаковыми масштабными коэффициентами по каждой координатной оси. Библиотека OpenGL обеспечивает построение различных проекций — ортографической, перспективной, изометрической, диметрической и любой другой, для которой пользователь обнаружил в литературе соответствующую матрицу аффинного преобразования.

В OpenGL предусмотрена работа с тремя матрицами, имеющими разное назначение. Модельно-видовая матрица (условное обозначение `GL_MODELVIEW`) определяет положение объекта в системе пространственных координат и, соответственно, его видом с позиции наблюдателя. Матрица проекций (условное обозначение `GL_PROJECTION`) управляет процессом выбранного режима проецирования трехмерного объекта в двумерную систему координат окна экрана. Матрица текстуры (условное обозначение `GL_TEXTURE`) управляет процессами наложения текстур на поверхность нашего объекта.

В каждый конкретный момент мы можем выполнять то или иное преобразование только с той из указанных матриц, которая объявлена текущей. Этот выбор происходит в результате обращения к процедуре `glMatrixMode`. Ее единственным аргументом является одно из упомянутых ранее условных обозначений. Перед началом цепочки аффинных преобразований в текущую матрицу обычно засылают единичную матрицу с помощью процедуры `glLoadIdentity`. Эта матрица не изменяет начальное состояние управляемого объекта и, как правило, принимается в качестве стартовой текущей матрицы для манипуляций с объектом в системе пространственных координат. Для других преобразований может потребоваться формирование начальной текущей матрицы с нестандартными значениями коэффициентов. В таких случаях прибегают к другой процедуре — `glLoadMatrixf`. В качестве аргумента этой процедуры, как правило, задают адрес вектора, содержащего 16 вещественных чисел, записанных *по столбцам* загружаемой матрицы.

Наиболее часто применяемые преобразования связаны со сдвигами, поворотами и масштабированием.

16.7. Начальные установки системы GLUT

Большинство руководств, методических пособий и опубликованных книг, так или иначе связанных с OpenGL, рекомендуют начинать пролог с обращения к процедуре `glutInit`, передавая ей адреса двух аргументов — счетчика параметров ко-

мандной строки (`@argc`) и указателя на строку, содержащую все параметры, включая и имя запускаемого приложения (`@argv`). В одном из таких руководств, размещенных на сайте Wikipedia, даже приводится пример процедуры на Паскале, извлекающей нужные данные из командной строки и передающей их процедуре инициализации системы GLUT (листинг 16.1).

Листинг 16.1. Пример инициализации

```
procedure glutInitPascal(ParseCmdLine: Boolean);
 var
   Cmd: array of PChar;
   CmdCount, I: Integer;
 begin
   if ParseCmdLine then
     CmdCount := ParamCount + 1
   else
     CmdCount := 1;
   SetLength(Cmd, CmdCount);
   for I := 0 to CmdCount - 1 do
     Cmd[I] := PChar(ParamStr(I));
   glutInit(@CmdCount, @Cmd);
 end;
```

В этой процедуре все правильно, кроме одного — пользоваться этой процедурой не стоит. Вместо того чтобы включать указанную процедуру в текст своего приложения и обращаться к ней с помощью следующего вызова:

```
glutInitPascal(true);
```

в программе на языке Free Pascal достаточно выполнить другой, менее затратный вызов:

```
glutInit(@argc,argv);
```

С точки зрения человека, хоть чуть-чуть знакомого с языком Паскаль, такая строка представляется явно ошибочной, т. к. переменные `argc` и `argv` в программе не объявлены, и компилятор должен это заметить. Другое дело — головная программа на языке C, которая включает аналогичные параметры в заголовок функции `main`:

```
int main(int argc, char *argv[]);
```

Однако компилятор FP не считает приведенное обращение к `glutInit` ошибочным, т. к. в модуле `System`, который автоматически подключается ко всем приложениям FP, содержится описание указанных переменных. Заполнение их нужными значениями происходит также автоматически в момент запуска приложения, и дублировать уже выполненную работу смысла не имеет.

Нам стало интересно, о каких параметрах командной строки, предназначенных для утилит GLUT, идет речь. Дело в том, что в некоторых литературных источниках туманно намекали, что командная строка может быть использована для передачи в GLUT имен файлов, необходимых для выполнения операций текстурирования. Для устранения недомолвок и эффекта испорченного телефона пришлось заглянуть в исходный текст процедуры `glutInit`, благо он открыт для всех на сайте **http://www.opengl.org**. Оказалось, что при проектировании этой процедуры ее автор М. Килгард предусмотрел возможность задания нескольких ключей, размещаемых в командной строке. Мы упомянем только те из них, которые имеют смысл при работе в среде Win32:

- `-display k` — количество мониторов, подключенных к ПК (`k=1` или `k=2`);
- `-geometry W x H + X + Y` — ширина и высота графического окна, его позиция на экране;
- `-iconic` — старт с графическим окном, свернутым в значок.

Хочется верить, что автор завел эти ключи для себя, чтобы экспериментировать с рядом параметров без перетрансляции программы. На самом деле, нагружать пользователя заданием нелепого набора данных не стоило. Количество дисплеев по умолчанию GLUT извлекает из списка оборудования, графическое окно на экране можно перетаскивать и изменять его размеры, сворачивать и разворачивать. Кроме того, влияние ряда ключей автоматически будет изменено при вызове таких процедур как `glutInitWindowSize`, `glutInitWindowPosition`. Одним словом, возня с использованием параметров командной строки сродни пустым хлопотам.

Но, так или иначе, какие-то другие операции, связанные с настройкой утилит GLUT, должны быть выполнены, и начинать их надо с упомянутого ранее обращения к процедуре `glutInit`.

Для задания ширины (`w`) и высоты (`h`) графического окна может быть использована процедура `glutInitWindowSize`:

```
glutInitWindowSize(w, h);       // размеры задаются в пикселах
```

По умолчанию создается окно размером 300×300.

Если вы хотите расположить графическое окно в заданном месте экрана, то можно воспользоваться процедурой `glutInitWindowPosition`:

```
glutInitWindowPosition(X, Y);   // значения X и Y в пикселах
```

По умолчанию Windows располагает графическое окно приложения там, где считает нужным. Однако с помощью стандартных приемов управления с помощью мыши вы можете переместить окно в другое место и, если понадобится, изменить его размеры. Естественно, что в программе должны быть предусмотрены действия по перерисовке изображения в связи с новыми размерами окна.

В группе команд инициализации утилиты GLUT довольно часто используется обращение к процедуре установки режимов отображения. Один из возможных вариантов выбора режима демонстрирует следующий вызов:

```
glutInitDisplayMode(GLUT_DOUBLE or GLUT_RGB or GLUT_DEPTH);
```

Аргумент этой процедуры — целочисленное значение без знака. Отдельные его биты, включаемые с помощью соответствующих мнемонических констант, определяют устанавливаемый режим. Таких признаков довольно много, и полный их список представлен в табл. 16.2. В приведенном примере использованы три следующие бита:

- `GLUT_DOUBLE` — режим отображения с двойной буферизацией;
- `GLUT_RGB` — управление цветом пикселов в формате `truecolor`;
- `GLUT_DEPTH` — окно с буфером глубины (*Z*-буфер).

Таблица 16.2

Признак режима	Пояснение
`GLUT_RGBA`	Управление цветом в формате TrueColor с признаком прозрачности
`GLUT_RGB`	То же самое (этот режим установлен по умолчанию)
`GLUT_INDEX`	Управление цветом по индексу в палитре
`GLUT_SINGLE`	Формирование изображения в единственном буфере (действует по умолчанию)
`GLUT_DOUBLE`	Формирование изображения с двойной буферизацией
`GLUT_ACCUM`	Окно с буфером накопления
`GLUT_ALPHA`	Режим использования свойства прозрачности в цветовых буферах
`GLUT_DEPTH`	Режим использования *Z*-буфера
`GLUT_STENCIL`	Режим использования буфера шаблонов. С помощью шаблонов (трафаретов) отдельные части фигуры могут быть вырезаны
`GLUT_MULTISAMPLE`	Режим, поддерживающий отображение нескольких графических окон
`GLUT_STEREO`	Режим создания стереоизображений. Для их просмотра используется специальная аппаратура (стереоочки, 3D-дисплеи)
`GLUT_LUMINANCE`	Режим специальной обработки красной составляющей цвета в модели RGBA, не содержащей зеленой и синей составляющих

Двойная буферизация, которая имеет смысл при создании динамических сцен, предполагает наличие двух битовых плоскостей — переднего и заднего плана. Содержимое буфера переднего плана отображается на экране дисплея. В это же время на буфере заднего плана выполняются текущие построения и преобразования. По сигналу программы, который инициируется обращением к процедуре `glutSwapBuffers`, происходит смена статуса буферов. Буфер заднего плана становится отображаемым, а буфер переднего плана начинает прием результатов текущих построений и преобразований. Двойная буферизация позволяет формировать изображение следующего кадра, пока пользователю демонстрируется текущий

кадр. Для отображения статических сцен можно использовать схему с одним буфером (признак GLUT_SINGLE), в котором осуществляются текущие построения. В этом случае вывод изображения на экран осуществляет процедура glFlush. Режим удаления невидимых частей с помощью алгоритма Z-буфера имеет смысл при обработке трехмерных объектов.

Для работы с графическим окном, в котором отображаются статические или динамические сцены, очень важно иметь набор подпрограмм, которые должны реагировать на различные события, возникающие во время сеанса. К событиям такого рода относятся перемещения графического окна и изменение его размеров (в том числе сворачивание в значок и распахивание до предшествующего размера), сигналы, поступающие от различных органов управления (мышь, клавиатура, игровые джойстики), истечение промежутка времени, установленного на таймере, и многое другое. Всего таких событий порядка 20. Перечень некоторых из них приведен в табл. 16.3. Там же описаны форматы данных, передаваемые обработчику соответствующего события. Разработчик приложения должен решить, какие из этих событий следует обрабатывать в его программе, и написать текст соответствующих процедур. Вызываться эти процедуры будут из главного цикла утилиты GLUT. Для такого рода подпрограмм обработки событий, инициированных за пределами нашего приложения, существует англоязычный термин — callback-процедуры. Переводят его как *подпрограммы обратного вызова*, но это название зачастую ставит пользователя в тупик. В системах визуального программирования (например, в Delphi) используют более понятные термины: OnDoubleClickButton1(...) — обработчик события "двойной щелчок кнопкой мыши" по объекту с именем Button1.

По существу, речь идет о процедурах, к которым наша программа непосредственно не обращается. Их вызывает "кто-то другой", и по завершению обработки события управление передается "кому-то другому". Этим кем-то другим является главный цикл утилиты GLUT, который осуществляет сверху управление процедурами нашей программы.

Таблица 16.3

Регистратор в GLUT	Событие	Параметры для обработчика
glutDisplayFunc	Перерисовка содержимого окна	Нет
glutReshapeFunc	Восстановление формы	(w, h) — новые ширина и высота
glutKeyboardFunc	Нажатие клавиши	(key, x, y) — код нажатой клавиши и позиция курсора мыши в этот момент
glutMouseFunc	Нажатие кнопки мыши	(button, state, x, y) — кнопка, состояние (нажатие, отпускание) и координаты курсора
glutMotionFunc	Перемещение мыши по графическому окну с нажатой кнопкой	(x, y) — координаты мыши

Таблица 16.3 *(окончание)*

Регистратор в GLUT	Событие	Параметры для обработчика
`glutTimerFunc`	Завершение очередного временного интервала после предыдущего вызова таймера	(`dt`, `@func(v)`, `v`) — интервал времени в миллисекундах, адрес функции, к которой надо обратиться по "звонку" будильника, аргумент, который передается функции обработки события

Написав процедуру обработки того или иного события, мы должны сообщить ее адрес соответствующему компоненту GLUT, отслеживающему данное событие. Этот процесс напоминает регистрацию лиц, ответственных за выполнение той или иной работы. В нужный момент этот исполнитель будет вызван, и ему передадут всю информацию, необходимую для обработки возникшего события.

Названия своих процедур обработки событий программист придумывает сам, но он должен руководствоваться списком аргументов, который его обработчик получит в нужный момент. "Регистрация" обработчиков некоторых событий может выглядеть следующим образом:

```
glutDisplayFunc(@OnRedraw);   // процедура перерисовки окна
glutReshapeFunc(@OnResize);   // процедура учета изменения формы
glutKeyboardFunc(@OnKey);     // обработка событий клавиатуры
```

После выполнения необходимых операций по настройке утилит GLUT выполняется операция по созданию графического окна и запускается главный цикл:

```
glutCreateWindow('Prog 1');          // создание окна с именем Prog 1
glClearColor(0.75, 0.75, 0.75, 1);   // очистка области клиента
glutMainLoop;                        // старт главного цикла Glut
```

16.8. Отображение простейшего двумерного изображения

В 1915 г. известный русский художник-авангардист Казимир Малевич представил на выставке небольшое полотно (размером примерно 80×80 см), бóльшую часть которого занимал черный квадрат. Тогда картина вызвала очень противоречивые отклики критиков. Да и в наше время можно многое услышать в адрес автора от лиц как превозносящих Малевича до небес, так и считающих его не вполне здоровым. Так или иначе, но в 2002 г. один из оригиналов был куплен известным российским предпринимателем В. Потаниным за миллион долларов и передан им на хранение в Эрмитаж. На рис. 16.2 приведено изображение этой картины, которое мы обнаружили в Интернете.

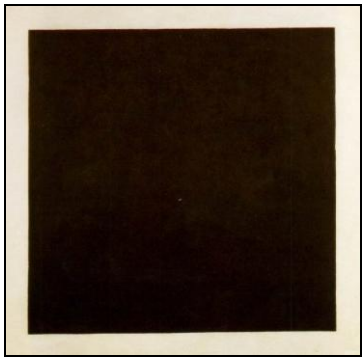


Рис. 16.2. Черный квадрат К. Малевича

Мы попробовали написать программу, которая, используя средства библиотеки OpenGL и утилит GLUT, воспроизводит на экране компьютера нечто подобное. Основная цель этой программы — продемонстрировать технику создания графического окна и простейшие средства обработки событий. При этом мы придерживались основного принципа — ничего лишнего в программе быть не должно (листинг 16.2).

Листинг 16.2. Отображение черного квадрата

```
program Malevich;
uses
  GL,GLUT;
//-----------------------------------------------------------
procedure OnRedraw; cdecl;        // процедура перерисовки сцены
begin
  glClear(GL_COLOR_BUFFER_BIT);   // очистка графического экрана
  glBegin(GL_QUADS);              // заголовок примитива "квадрат"
    glColor3f(0.8, 0.75, 0.75);   // установка цвета холста
    glVertex2f(-1, -1);       // 1-я вершина квадрата холста
    glVertex2f(-1,  1);       // 2-я вершина квадрата холста
    glVertex2f( 1,  1);       // 3-я вершина квадрата холста
    glVertex2f( 1, -1);       // 4-я вершина квадрата холста
    glColor3f(0.1,0.1,0.1);  // установка цвета квадрата
    glVertex2f(-0.9, -0.9);  // 1-я вершина квадрата
    glVertex2f(-0.9,  0.9);  // 2-я вершина квадрата
    glVertex2f( 0.9,  0.9);  // 3-я вершина квадрата
    glVertex2f( 0.9, -0.9);  // 4-я вершина квадрата
  glEnd;                      // конец описания примитива
  glFlush;                    // отображение содержимого буфера
end;
```

```
//-----------------------------------------------------------------------
procedure OnResize(w,h:Longint); cdecl; // если меняются размеры
begin
  glViewport(0, 0, w, h);         // изменение точки зрения
  glLoadIdentity;                 // сброс текущей матрицы
  if w>h then glScalef(h/w,1,1)   // масштабирование по x
  else glScalef(1,w/h,1);         // масштабирование по y
end;
//-----------------------------------------------------------------------
begin
  glutInit(@argc,argv);          // инициализация Glut
  glutCreateWindow('OpenGL: Malevich'); // создание окна
  glutDisplayFunc(@OnRedraw);   // регистрация функции перерисовки
  glutReshapeFunc(@OnResize);   // регистрация функции изменения w, h
  glMatrixMode(GL_MODELVIEW);   // выбор текущей матрицы
  glClearColor(1,1,1,1);        // цвет для очистки экрана
  glutMainLoop;                 // старт главного цикла GLUT
end.
```

Начнем комментировать программу с ее головной части. Во-первых, инициализация GLUT сделана самым экономным способом. Во-вторых, мы не собираемся устанавливать размеры и позицию графического окна, полагаясь на выбор пакета GLUT и операционной системы. К сожалению, при создании графического окна мы не можем использовать в его заголовке русские буквы. Затем мы сообщаем адреса обработчиков двух событий — программы перерисовки изображения на поле клиента (`OnRedraw`) и программы реакции на изменение размеров графического окна (`OnResize`). Следующая строка устанавливает в качестве текущей матрицу модели изображения. Далее устанавливается белый цвет, который будет использоваться для очистки области клиента в графическом окне. Последняя строка запускает главный цикл утилит GLUT.

В начале этого цикла первой вызывается процедура `OnResize`. Это определяется порядком обработки событий, предусмотренным в Windows. По полученным габаритам создаваемого окна (параметры `w` и `h`) устанавливаются правильные размеры поля зрения. Затем происходит загрузка начального состояния в текущую матрицу аффинных преобразований и коррекция текущей матрицы, если ширина и высота оказываются разными.

После фактического создания окна запускается процедура перерисовки, в которой область рисования чистится установленным цветом. Затем на ней формируется изображение двух квадратов разного цвета. Если бы не было обращения к процедуре `glFlush`, то в области клиента изображение квадратов не появилось бы.

Далее продолжается холостой цикл GLUT, до тех пор, пока пользователь не меняет позицию и размеры окна. Для завершения работы приложения можно щелкнуть по стандартной кнопке окна Windows с крестиком.

Еще одно обстоятельство, на которое надо обратить внимание. Пакет GLUT был ориентирован на язык программирования C, поэтому в заголовках процедур обработки событий, которые должны вызываться функциями C, стоит указание `cdecl`. Это означает, что наши процедуры будут настроены на нужный порядок следования параметров, отличный от стандарта языка Паскаль, и по окончанию своей работы выполнят процедуры очистки стека в соответствии с правилами языка C.

Таким образом, наша программа, насчитывающая примерно три десятка строк, мгновенно выполняет работу, над которой Казимир Малевич корпел несколько месяцев (результат работы программы приведен на рис. 16.3).

Рис. 16.3. Электронная подделка под Малевича

Объем приведенной выше версии программы malevich.pas можно сократить еще на 8 строк, если вместо вызова процедур построения графических примитивов воспользоваться более удобной подпрограммой `glRectf`, отображающей прямоугольник по координатам его противоположных вершин (листинг 16.3).

Листинг 16.3. Оптимизированное отображение черного квадрата

```
program malevich_1;
uses
  GL,GLUT;
//-------------------------------------------------
procedure OnRedraw; cdecl;
begin
  glClear(GL_COLOR_BUFFER_BIT);
```

```
  glColor3f(0.8,0.75,0.75);
  glRectf(-1,-1,1,1);
  glColor3f(0.1,0.1,0.1);
  glRectf(-0.9,-0.9,0.9,0.9);
  glFlush;
end;
//------------------------------------------------------
procedure OnReSize(W,H:integer); cdecl;
begin
  glViewport(0, 0, W, H);
  glLoadIdentity;
  if W>H then glScalef(H/W,1,1) else glScalef(1,W/H,1);
end;
//------------------------------------------------------
begin
  glutInit(@argc,argv);
  glutCreateWindow('OpenGL: Malevich');
  glutDisplayFunc(@OnRedraw);
  glutReshapeFunc(@OnReSize);
  glMatrixMode(GL_MODELVIEW);
  glClearColor(1,1,1,1);
  glutMainLoop;
end.
```

Таким образом, на отображение классического полотна нам понадобилось всего порядка 20 строк программы (не считая операторных скобок `begin-end`). При этом соответствующее графическое окно выполнено по всем правилам Windows. Оно сохраняет форму и пропорции изображения при изменении размеров окна, его свертывании в значок и распахивании до прежних размеров.

16.9. Списки изображений

Представьте себе, что нам потребовалось бы изобразить несколько "черных квадратов", отличающихся по размерам, расположенных в разных местах и даже повернутых относительно линии горизонта. Конечно, можно было бы растиражировать несколько экземпляров тела процедуры `Redraw`, помещая перед каждой копией несколько строк с формированием очередной текущей матрицы преобразований. А если бы исходное изображение содержало описание не 8 вершин (как в нашем примере), а нескольких сотен точек? Объем исходной программы сразу бы распух до необозримых размеров.

Можно было бы каждый раз писать процедуру для отображения объектов, но, к счастью, в языке OpenGL предусмотрена возможность создания списка изображений, и вместо повторения стереотипных описаний мы можем обратиться к содержимому ранее созданного списка. Графические примитивы, составляющие элементы списка, заключаются в пару специальных операторных скобок: `glNewList` ... `glEndList`:

```
const
  Num_Lst=1;
...
glNewList(Num_Lst, mode);
  glBegin(GL_QUADS);
    glColor3f(0.8, 0.75, 0.75);
    glVertex2f(-1, -1);
    glVertex2f(-1,  1);
    glVertex2f( 1,  1);
    glVertex2f( 1, -1);
    glColor3f(0.1,0.1,0.1);
    glVertex2f(-0.9, -0.9);
    glVertex2f(-0.9,  0.9);
    glVertex2f( 0.9,  0.9);
    glVertex2f( 0.9, -0.9);
  glEnd;
glEndList;
```

Первый параметр `Num_Lst` в открывающей операторной скобке принимает целочисленное значение и выполняет функцию номера создаваемого списка. Второй параметр `mode` может принимать одно из двух значений — `GL_COMPILE` или `GL_COMPILE_AND_EXECUTE`. В первом случае элементы списка просто заносятся в некоторый служебный массив (регистрируются), во втором случае регистрируются и исполняются.

Если нам понадобится повторить действия, указанные в ранее созданном списке, то достаточно обратиться к процедуре `glCallList`, указав в качестве единственного параметра номер нужного списка `Num_Lst`.

Так как каждый список входит в состав ресурсов приложения, то, завершая работу программы, мы должны позаботиться об удалении созданных списков. Например, эту операцию можно выполнить в процедуре обработчика события клавиатуры:

```
procedure OnKey(Key:byte; X,Y:longint); cdecl;
begin
  if Key = 27 then              // если нажата клавиша <Esc>
    begin
      glDeleteLists(Num_Lst, 1);// удаление списка с номером Num_Lst
      Halt(0);
    end;
end;
```

Как правило, для создания группы списков используют последовательные номера — `Num_Lst`, `Num_Lst+1`, `Num_Lst+2`, ... Поэтому в процедуре удаления второй параметр задает количество удаляемых списков. Для создания группы списков с последовательными номерами можно воспользоваться процедурой `glGenList(Num_lst, k)`. Поначалу все эти списки пусты, но использование операторных скобок `glBegList(Num_lst+2)` означает, что к ранее сформированному списку с указанным номером (или изначально пустому) мы добавляем новые строки.

16.10. Формирование надписей в области рисования

В пакете GLUT предусмотрены довольно скромные возможности для нанесения подписей в области рисования. Во-первых, мы можем установить цвет символов и начальную позицию подписи (координаты левого нижнего угла первого символа):

```
glColor3f(vr, vg, vb);
glRasterPos2f(x, y);
```

Затем в цикле можно отобразить каждую букву подписи `s`, представленную обычной строкой типа `string`, с помощью следующей процедуры:

```
glutBitmapCharacter(font, ord(s[i]));
```

Первым параметром этой процедуры является нетипизированный указатель на выбранный пользователем шрифт. Таких шрифтов в пакете GLUT всего семь, и список указателей, которыми можно пользоваться, приведен в табл. 16.4. К сожалению, ни в одном из них не представлены русские буквы.

Таблица 16.4

Указатель на шрифт	**Пояснение**
`GLUT_BITMAP_8_BY_13`	Растровый моноширинный шрифт размером 8×13 пикселов
`GLUT_BITMAP_9_BY_15`	Растровый моноширинный шрифт размером 9×15 пикселов
`GLUT_BITMAP_TIMES_ROMAN_10`	Шрифт Times Roman высотой 10 пунктов
`GLUT_BITMAP_TIMES_ROMAN_24`	Шрифт Times Roman высотой 24 пункта
`GLUT_BITMAP_HELVETICA_10`	Шрифт Helvetica высотой 10 пунктов
`GLUT_BITMAP_HELVETICA_12`	Шрифт Helvetica высотой 12 пунктов
`GLUT_BITMAP_HELVETICA_18`	Шрифт Helvetica высотой 18 пунктов

Мы внимательно знакомились со справочными материалами по пакету GLUT, представленными в MSDN, и некоторыми вспомогательными материалами на сайте разработчиков Free Pascal, но нигде не обнаружили точных рекомендаций по использованию описанных выше средств. В результате проб и ошибок были установлены две важные детали.

Во-первых, если задание начальной координаты текста (процедура `glRasterPos2f`) предшествует заданию цвета надписи, то при любых параметрах процедуры `glColor3f` надписи рисуются белым цветом. Поэтому надо сначала установить цвет и только после этого задавать позицию подписи.

Во-вторых, воспроизведение каждого символа с помощью процедуры `glutBitmapCharacter` осуществляется путем копирования растровой маски каждого символа выбранного шрифта в соответствующую часть области рисования с автоматическим продвижением указателя позиции на ширину текущего символа. Поэтому аффинные преобразования (сдвиг, повороты, сжатия/расширения) влияют только на начальную позицию подписи, а растровые образы символов при этом остаются параллельными координатным полям области отображения. Бедность набора шрифтов в GLUT объясняется тем, что в соответствующих сервисных программах приходится хранить растровые изображения каждого символа.

Наиболее экономный способ воспроизведения подписей, к которому мы пришли после многочисленных экспериментов, приводится в листинге 16.4 в модификации программы malevich.pas.

Листинг 16.4. Подпись к черному квадрату

```
program Malevich_t;
uses
  GL, GLUT;
//----------------------------------------
procedure writeGL(x,y:single;font:pointer;s:string);
var
  i:integer;
begin
  glRasterPos2f(x, y);
  for i:=1 to length(s) do
    glutBitmapCharacter(font,ord(s[i]));
end;
//----------------------------------------
procedure OnRedraw; cdecl;
var
  x,y:single;
  f:array [0..3] of single;
begin
  glClear(GL_COLOR_BUFFER_BIT);
```

```
  glColor3f(0.8,0.75,0.75);
  glRectf(-1,-1,1,1);
  glColor3f(0.1,0.1,0.1);
  glRectf(-0.9,-0.9,0.9,0.9);
  glColor3f(0,1,1);
  writeGL(0.28,-0.89,GLUT_BITMAP_TIMES_ROMAN_24,'Malevich');
  glFlush;
end;
//----------------------------------------
procedure OnResize(W,H:Longint); cdecl;
begin
  glViewport(0, 0, W, H);
  glLoadIdentity;
  if W>H then glScalef(H/W,1,1) else glScalef(1,W/H,1);
end;
//----------------------------------------
begin
  glutInit(@argc,argv);
  glutCreateWindow('OpenGL: Malevich');
  glutDisplayFunc(@OnRedraw);
  glutReshapeFunc(@OnResize);
  glMatrixMode(GL_MODELVIEW);
  glClearColor(1,1,1,1);
  glutMainLoop;
end.
```

Рис. 16.4. Черный квадрат с подписью

Результат ее работы представлен на рис. 16.4. Единственное неудобство заключается в экспериментальном подборе координат точки привязки текста. Надо учитывать, что границы в области рисования по каждой координате принадлежат диапазону [–1, +1].

16.11. Построение простейшего трехмерного изображения

Приведенная в листинге 16.5 программа Cube_1.pas воспроизводит на экране изображение разноцветного куба средствами библиотеки OpenGL. Массив `Points` задает координаты вершин куба. Каждой вершине присвоен индивидуальный цвет (массив `Colors`). Вдоль каждого ребра по умолчанию производится линейная интерполяция цветовых характеристик смежных вершин, и полученные цвета таким же образом распространяются на внутренние точки граней.

Для правильного построения многогранной фигуры следует различать внешнюю и внутреннюю грани. Грань считается *внешней*, если при взгляде на нее с позиции наблюдателя, находящегося перед гранью за пределами куба, вершины обходятся *против часовой стрелки*. Для этого именно в таком порядке следует задавать координаты точек вершин внешней грани в программе. На рис. 16.5 приведена нумерация вершин куба, использованная в программах `Cube_1` и `Cube_2`.

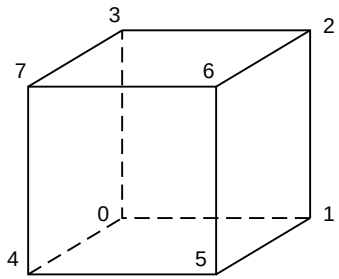


Рис. 16.5. Один из вариантов нумерации вершин

Довольно многие фрагменты программы `Cube_1` напоминают уже прокомментированные операции по созданию графического окна.

В разделе инициализации графики новыми являются процедуры задания режима отображения, в которых выделены три признака — использование двойной буферизации (`GLUT_DOUBLE`) и буфера глубины (`GLUT_DEPTH`). В связи с использованием *Z*-буфера пришлось включить режим отсечения невидимых граней (`glEnable(GL_DEPTH_TEST);`).

В обработчике события, возникающего при изменении размеров графического окна, появились четыре дополнительные процедуры.

Первая из них — `glFrustum` устанавливает область отсечения, ограниченную шестью плоскостями. То, что находится внутри области, мы увидим на графическом экране. Первые два параметра этой процедуры задают координаты отсечения слева и справа. Два следующих параметра определяют координаты отсечения снизу и сверху. Последние два значения определяют *Z*-координаты ближней и дальней плоскостей отсечения.

Процедура `glTranslatef` смещает объект по оси *z*, чтобы его можно было рассмотреть полностью. Когда объект находится очень близко к позиции наблюдателя, то мы видим только его часть (попробуйте прочитать название книги, поднеся ее к глазу на сантиметр).

Два следующих поворота (`glRotatef`) объекта на 30° вокруг оси *x* (вектор с компонентами 1, 0, 0) и на 60° вокруг оси *y* (вектор с компонентами 0, 1, 0) сделаны для того, чтобы увидеть куб под углом, позволяющим рассмотреть не только лицевую грань.

В обработчике события, связанного с необходимостью перерисовать изображение, потребовалось чистить не только плоскость отображения (`GL_COLOR_BUFFER_BIT`), но и *Z*-буфер (`GL_DEPTH_BUFFER_BIT`). Процедура `quad`, использованная в процедуре `OnRedraw`, позволила уменьшить число строк вызова функций OpenGL. Шесть обращений в цикле к процедуре `quad` со специально заготовленным массивом `CubeInd` позволили достаточно компактно выполнить обращения, необходимые для построения всех граней куба. В массиве `CubeInd` находятся четырехкомпонентные векторы с номерами вершин, соответствующих каждой грани и упомянутому выше правилу обхода вершин. Так как процедура `quad` в качестве входного параметра получает адрес вектора с номерами вершин, то она использует вызов процедур `glColor` и `glVertex` в соответствующем формате. Аналогичные приемы программирования вы можете использовать и при построении других многогранников.

Листинг 16.5. Построение разноцветного куба

```
program Cube_1;
uses
  GL,GLUT;
type
  tv=array [0..3] of byte;
  pc=array [0..7,0..2] of single;
var
  Points:pc=((-1,-1,-1),(1,-1,-1),(1,1,-1),(-1,1,-1),
             (-1,-1,1),(1,-1,1),(1,1,1),(-1,1,1));
  Colors:pc=((0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),
             (1,0,1),(1,1,0),(1,1,1));
  CubeInd:array [0..5] of tv = ((0,3,2,1),(2,3,7,6),
            (0,4,7,3),(1,2,6,5),(4,5,6,7),(0,1,5,4));
```

```
//-----------------------------------------------------
procedure quad(const v:tv);
// Процедура построения грани
var
  i,j: byte;
begin
     glBegin(GL_QUADS);
       for i:=0 to 3 do begin
         j:=v[i];
         glColor3fv(Colors[j]);
         glVertex3fv(Points[j]);
       end;
    glEnd;
end;
//-----------------------------------------------------
procedure OnRedraw; cdecl;
var
  i: byte;
begin
  glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
  for i:=0 to 5 do
    quad(CubeInd[i]);
  glutSwapBuffers;
end;
//-----------------------------------------------------
procedure OnResize(W,H:Longint); cdecl;
begin
  glViewport(0, 0, W, H);
  glLoadIdentity;
  glFrustum(-1,1,-1,1,3,10);
  glTranslatef(0,0,-8);
  glRotatef(30,1,0,0);
  glRotatef(60,0,1,0);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity;
  if W>H then glScalef(H/W,1,1) else glScalef(1,W/H,1);
  glMatrixMode(GL_MODELVIEW);
end;
//-----------------------------------------------------
begin
  glutInit(@argc,argv);
```

```
  glutInitDisplayMode(GLUT_DOUBLE or GLUT_RGB or GLUT_DEPTH);
  glutCreateWindow('OpenGL: Cube 1');
  glutDisplayFunc(@OnRedraw);
  glutReshapeFunc(@OnResize);
  glMatrixMode(GL_MODELVIEW);
  glClearColor(0.7,0.7,0.7,1);
  glEnable(GL_DEPTH_TEST);
  glutMainLoop;
end.
```

Результат работы программы `Cube_1` приведен на рис. 16.6.

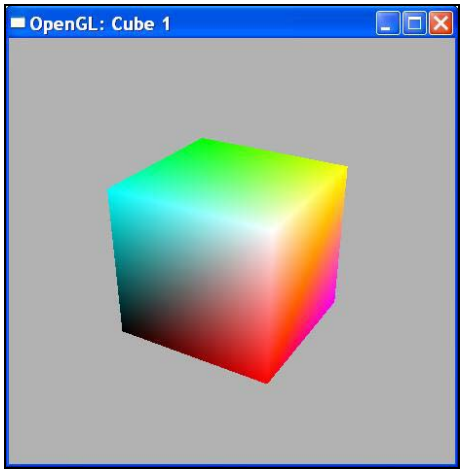


Рис. 16.6. Разноцветный куб

Объем программы `Cube_1` можно немного сократить за счет использования процедуры `glDrawElements`, которая внутри себя организует цикл, подобный шестикратному обращению к процедуре `quad`. Перед обращением к процедуре `glDrawElements` нужно сообщить пакету OpenGL, что он может воспользоваться массивами вершин и цветов, и передать ему адреса этих массивов. Нововведения выделены жирным шрифтом в приведенной ниже программе `Cube_2`.

Листинг 16.6. Оптимизированная программа построения куба

```
program Cube_2;
uses
  GL, GLUT;
type
  tv=array [0..3] of byte;
  pc=array [0..7,0..2] of single;
```

```
var
  Points:pc=((-1,-1,-1),(1,-1,-1),(1,1,-1),(-1,1,-1),
             (-1,-1,1),(1,-1,1),(1,1,1),(-1,1,1));
  Colors:pc=((0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),
             (1,0,1),(1,1,0),(1,1,1));
  CubeInd:array [0..5] of tv = ((0,3,2,1),(2,3,7,6),
            (0,4,7,3),(1,2,6,5),(4,5,6,7),(0,1,5,4));
//-----------------------------------------------------
procedure OnRedraw; cdecl;
var
  i:byte;
begin
  glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
  glDrawElements(GL_QUADS,24,GL_UNSIGNED_BYTE,@CubeInd);
  glutSwapBuffers;
end;
//-----------------------------------------------------
procedure OnResize(W,H:Longint); cdecl;
begin
  glViewport(0, 0, W, H);
  glLoadIdentity;
  glFrustum(-1,1,-1,1,3,10);
  glTranslatef(0,0,-8);
  glRotatef(30,1,0,0);
  glRotatef(60,0,1,0);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity;
  if W>H then glScalef(H/W,1,1) else glScalef(1,W/H,1);
  glMatrixMode(GL_MODELVIEW);
end;
//-----------------------------------------------------
begin
  glutInit(@argc,argv);
  glutInitDisplayMode(GLUT_DOUBLE or GLUT_RGB or GLUT_DEPTH);
  glutCreateWindow('OpenGL: Cube 1');
  glutDisplayFunc(@OnRedraw);
  glutReshapeFunc(@OnResize);
  glMatrixMode(GL_MODELVIEW);
  glClearColor(0.7,0.7,0.7,1);
  glEnable(GL_DEPTH_TEST);
  glEnableClientState(GL_COLOR_ARRAY);
```

```
  glEnableClientState(GL_VERTEX_ARRAY);
  glColorPointer(3,GL_FLOAT,0,@Colors);
  glVertexPointer(3,GL_FLOAT,0,@Points);
  glutMainLoop;
end.
```

Объем исходной программы уменьшился примерно на 10 строк и работает она чуть-чуть быстрее за счет меньших потерь на передачу данных в подпрограмму `glDrawElements` и организацию соответствующих циклов.

16.12. Анимация на плоскости

Вращающийся квадрат — один из самых простых примеров анимации. При описании пакета OpenGL к подобной иллюстрации прибегают столь же часто, как и к программе "Общего привета" ("Hello, world!"). Больше других нам понравилась реализация вращения квадрата, написанная на языке C и приведенная в книге Э. Эйнджела "Интерактивная компьютерная графика" [36]. В этом примере, с одной стороны, демонстрируется управление вращением с помощью кнопок мыши (левая кнопка запускает вращение, правая — останавливает). С другой стороны, здесь появилась возможность разумного использования времени ожидания сообщения, адресованного какому-нибудь из обработчиков событий. Вместо пустого повторения главного цикла приложение может получить управление и использовать время "простоя" для выполнения каких-то вычислений или подготовки информации для отображения следующей сцены. Адрес такой процедуры, предусмотренной в приложении, регистрируется в GLUT с помощью подпрограммы `glutIdleFunc`. Объем нашей модификации примерно на 10% меньше по сравнению с программой Эйнджела.

Программа `Rotating_Quad` выводит на экран изображение красного квадрата и ждет сообщение от мыши (листинг 16.7). Если поступает сигнал о нажатии левой кнопки, в качестве адреса процедуры обработки простоя назначается вход в подпрограмму `OnIdle`, заставляет GLUT обратиться к процедуре перерисовки изображения (опосредованно через процедуру `glutPostRedisplay`). Если поступает сигнал от нажатия правой кнопки, то в качестве адреса процедуры обработки простоя передается нулевой указатель, и GLUT перестает периодически вызывать процедуру `OnIdle`. Ближайший вызов подпрограммы `OnRedraw` может произойти либо при изменении размеров окна, либо при очередном нажатии левой кнопки мыши. На быстром компьютере для замедления вращения квадрата следует уменьшить значение угла поворота `angle`.

Процедура обработки сообщения от мыши получает четыре целочисленных параметра. Первый задает номер нажатой кнопки — левая, правая и центральная (в случае, если мышь трехкнопочная). Второй параметр фиксирует состояние

кнопки — утоплена или отпущена. Два оставшихся соответствуют координатам *x* и *y* курсора мыши.

Листинг 16.7. Вращающийся квадрат

```
program Rotating_Quad;
uses
  GL, GLUT;
var
  angle : single = 0.1;
//------------------------------
procedure OnRedraw; cdecl;
begin
  glClear(GL_COLOR_BUFFER_BIT);
  glRotatef(angle,0,0,1);
  glRectf(-25,-25,25,25);
  glutSwapBuffers;
end;
//--------------------------------
procedure OnResize(w,h:integer); cdecl;
begin
  glViewPort(0,0,w,h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity;
  glOrtho(-50,50,-50,50,-1,1);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity;
  if W>H then glScalef(h/w,1,1) else glScalef(1,w/h,1);
end;
//--------------------------------
procedure OnIdle; cdecl;
begin
  glutPostRedisplay;
end;
//--------------------------------
procedure OnMouse(button,state,x,y:integer); cdecl;
begin
  if state <> GLUT_DOWN then exit;
  if button = GLUT_LEFT_BUTTON  then glutIdleFunc(@OnIdle);
  if button = GLUT_RIGHT_BUTTON then glutIdleFunc(Nil);
end;
```

```
//---------------------------------
begin
  glutInit(@argc,argv);
  glutInitDisplayMode(GLUT_DOUBLE);
  glutCreateWindow('Rotating Quad');
  glClearColor(0.75,0.75,0.75,0);
  glutDisplayFunc(@OnRedraw);
  glutReshapeFunc(@OnResize);
  glutMouseFunc(@OnMouse);
  glColor3f(1,0,0);
  glutMainLoop;
end.
```

На рис. 16.7 представлены две наложенные друг на друга позиции вращающегося квадрата. В каждый текущий момент мы наблюдаем на экране только одну из промежуточных позиций квадрата.

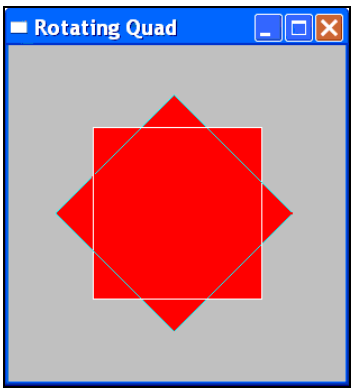


Рис. 16.7. Вращающийся квадрат

16.13. Анимация в пространстве

Следующий пример по логике анимации ничем не отличается от вращающегося квадрата. Однако в нем использованы более сложные объекты — сплошной куб, каркас куба (проволочная модель), сплошная сфера, расположенная внутри куба и слегка выпирающая за его границы, и проволочный каркас сферы (листинг 16.8).

Для воспроизведения сплошного куба и его проволочного каркаса используются процедуры пакета GLUT:

```
glutSolidCube(h);    // построение сплошного куба
glutWireCube(h);     // построение проволочной модели куба
```

Единственный параметр этой процедуры задает размер ребра. Проволочный каркас накладывается на сплошную модель куба с единственной целью — выделить ребра куба, которые на монохромной модели почти не различимы.

Для воспроизведения сплошной сферы и ее проволочного каркаса также используются процедуры пакета GLUT:

```
glutSolidSphere(r, nzp, nzm);
glutWireSphere(r, nzp, nzm);
```

Здесь:

- `r` — радиус сферы;
- `nzp` — число параллелей вокруг оси *z*;
- `nzm` — число меридианов вдоль оси *z*.

Значения `nzp` и `nzm` определяют качество аппроксимации поверхности сферы, чем эти величины больше, тем большее количество криволинейных трапеций укладывается на поверхности и тем лучше выглядит сфера. Однако увеличение числа таких лоскутков ("патчей") требует большего времени для вычисления координат точек на сфере. Обратите внимание на то, что радиус проволочного каркаса задан чуть-чуть больше радиуса сплошной сферы. При совпадении радиусов во время вращения параллели и меридианы превращаются в случайные штриховые линии. Такая же коррекция по длине ребра была сделана и для куба.

Использование *Z*-буфера придает нашей конструкции большую объемность.

Листинг 16.8. Трехмерная анимация

```
program Sphere_in_Cube;
uses
  GL,GLUT;
//-----------------------------
procedure OnRedraw; cdecl;
begin
  glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
  glRotatef(2.0,0,0,1);
  glColor3f(0,0.6,1);
  glutSolidCube(1.98);
  glColor3f(0,0,0);
  glutWireCube(2);
  glColor3f(0,1,1);
  glutSolidSphere(1.35,20,20);
  glColor3f(0,0,0);
  glutWireSphere(1.36,20,20);
  glutSwapBuffers;
end;
```

```
//---------------------------------
procedure OnResize(w,h:integer); cdecl;
begin
  glViewport(0, 0, W, H);
  glLoadIdentity;
  glFrustum(-1,1,-1,1,3,10);
  glTranslatef(0,0,-8);
  glRotatef(30,0,0,1);
  glRotatef(60,1,0,0);
  if W>H then glScalef(H/W,1,1) else glScalef(1,W/H,1);
end;
//---------------------------------
procedure OnIdle; cdecl;
begin
  glutPostRedisplay;
end;
//---------------------------------
procedure OnMouse(button,state,x,y:integer); cdecl;
begin
  if state <> GLUT_DOWN then exit;
  if button=GLUT_LEFT_BUTTON then glutIdleFunc(@OnIdle);
  if button=GLUT_RIGHT_BUTTON then glutIdleFunc(nil);
end;
//---------------------------------
begin
  glutInit(@argc,argv);
  glutInitDisplayMode(GLUT_DOUBLE or GLUT_DEPTH);
  glutCreateWindow('Sphere in Cube');
  glClearColor(0.75,0.75,0.75,0);
  glutDisplayFunc(@OnRedraw);
  glutReshapeFunc(@OnResize);
  glutMouseFunc(@OnMouse);
  glEnable(GL_DEPTH_TEST);
  glutMainLoop;
end.
```

Один из кадров, зафиксированных в момент вращения нашей конструкции, запечатлен на рис. 16.8.

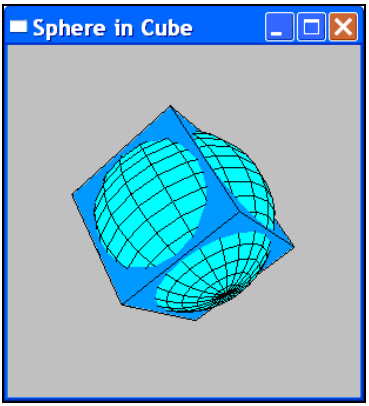


Рис. 16.8. Вращающиеся трехмерные объекты

16.14. Параметры источника света

Для повышения реалистичности сцены можно воспользоваться источниками света. Стандартная версия OpenGL предоставляет пользователю до 8 источников света, для обозначения которых используются константы перечисления `GL_LIGHT0`, `GL_LIGHT1`, ..., `GL_LIGHT7`. Подключение одного или нескольких источников света осуществляется в два этапа. Сначала нужно разрешить режим использования освещения:

```
glEnable(GL_LIGHTING);     // разрешение режима освещения
```

Каждый источник света включается индивидуально:

```
glEnable(GL_LIGHT0);       // включение "лампы" с номером 0
```

Точечный источник света равномерно распространяет все компоненты своего излучения по любому направлению в пространстве. Прожектор имеет более узкий конус излучения с ослаблением интенсивности луча по мере отклонения от его главной оси.

Любому источнику света можно приписать довольно много параметров, характеризующих его положение, интенсивности излучения отдельных компонент света, угол направленности излучения, коэффициенты ослабления интенсивности по мере удаления освещаемого объекта. Перечень основных характеристик источника света приведен в табл. 16.5.

Таблица 16.5

Имя параметра	Смысл и значение параметра
`GL_POSITION`	Положение источника света, обычно задаваемое одним из векторов (`x`, `y`, `z`, `0`) или (`x`, `y`, `z`, `1`)

Таблица 16.5 *(окончание)*

Имя параметра	Смысл и значение параметра
GL_AMBIENT	Интенсивность излучения фоновой компоненты источника, обычно задаваемая вектором (RIA, GIA, BIA, 1)
GL_DIFFUSE	Интенсивность излучения диффузной компоненты источника, обычно задаваемая вектором (RID, GID, BID, 1)
GL_SPECULAR	Интенсивность излучения зеркальной компоненты источника, обычно задаваемая вектором (RIS, GIS, BIS, 1)
GL_SPOT_DIRECTION	Направление излучения прожектора, задаваемое компонентами вектора главной оси излучения
GL_SPOT_EXPONENT	Показатель функции распределения интенсивности излучения
GL_SPOT_CUTOFF	Угол рассеяния, задаваемый в градусах по каждой координатной оси

Четвертая составляющая в позиции источника света определяет удаленность источника от освещаемой сцены. Нулевое значение соответствует бесконечно удаленному источнику света, для которого все лучи, достигающие сцены, имеют одинаковое направление. Угол попадания лучей на освещаемый объект, естественно, может оказать влияние на тень, для построения которой используется соответствующий режим.

Интенсивности того или иного излучения задаются для каждой RGB-составляющей из диапазона от 0 (минимальное значение интенсивности) до 1 (максимальное значение интенсивности).

Показатели функции распределения интенсивности в прожекторе задаются от –1 (минимум) до 1 (максимум).

Для задания значений характеристик параметра с именем name_par разумно воспользоваться векторной модификацией процедуры glLightfv:

```
var
  v:array [0..3] of single = (val1,val2,val3,val4);
  ...
begin
  ...
  glLightfv(GL_LIGHTk,name_par,v);     // k - номер источника
```

Опрос значений характеристик, установленных по умолчанию, можно осуществить с помощью процедуры glGetLightfv:

```
glGetLightfv(GL_LIGHTk, name_par, v);
```

В наиболее простых программах, использующих режим освещенности, обычно задействуется единственный источник — GL_LIGHT0. В листинге 16.9 приводится текст программы LIGHT_0.pas, которая выводит значения параметров этого источника света.

Листинг 16.9. Опрос параметров источника света

```
program LIGHT_0;
{$mode objfpc}
uses
  gl, glu, glut;
var
  v:array [0..3] of single;
procedure print(s:string);
begin
  write(s,'(',v[0]:3:1,', ',v[1]:3:1,', ');
  writeln(v[2]:3:1,', ',v[3]:3:1,')');
end;

begin
  glutInit(@argc,argv);
  glutCreateWindow('LIGHT0 parameters');
  writeln('Параметры источника света GL_LIGHT0 по умолчанию:');
  glGetLightfv(GL_LIGHT0,GL_POSITION,v);
  print('Position = ');
  glGetLightfv(GL_LIGHT0,GL_AMBIENT,v);
  print('Ambient  = ');
  glGetLightfv(GL_LIGHT0,GL_DIFFUSE,v);
  print('Diffuse  = ');
  glGetLightfv(GL_LIGHT0,GL_SPECULAR,v);
  print('Specular = ');
  glGetLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,v);
  print('Spot Direction = ');
  glGetLightfv(GL_LIGHT0,GL_SPOT_EXPONENT,v);
  print('Spot Exponent  = ');
  glGetLightfv(GL_LIGHT0,GL_SPOT_CUTOFF,v);
  print('Spot Cutooff   = ');
  readln;
end.
```

Результаты ее работы выглядят следующим образом:

```
Running "e:\fpc\myprog\LIGHT_0.exe "
Параметры источника света GL_LIGHT0 по умолчанию:
Position = (0.0, 0.0, 1.0, 0.0)
Ambient  = (0.0, 0.0, 0.0, 1.0)
```

```
Diffuse  = (1.0, 1.0, 1.0, 1.0)
Specular = (1.0, 1.0, 1.0, 1.0)
Spot Direction = (0.0, 0.0, -1.0, 1.0)
Spot Exponent  = (0.0, 0.0, -1.0, 1.0)
Spot Cutooff   = (180.0, 0.0, -1.0, 1.0)
```

16.15. Световые характеристики материала

Если предполагается использование источника света с определенными характеристиками, то мы должны спланировать и взаимодействие всех компонентов излучения с освещаемым объектом. Это означает, что следует должным образом подобрать светоотражательные характеристики материалов, представляющих грани объекта. Задача эта довольно сложная, и хотя в литературе можно обнаружить довольно много практических советов, в большинстве случаев хорошие результаты достигаются после многочисленных проб и ошибок.

Если у источника света основные характеристики определяют значения интенсивностей трех типов излучения, то световая модель материала характеризуется значениями коэффициентов отражения для тех же трех типов лучей. Векторные форматы процедур установки этих коэффициентов и опроса текущих значений похожи на процедуры, описанные в предыдущем разделе:

```
glMaterialfv(face, name_par, v);      // установка характеристик
glGetMaterialfv(face, name_par, v);   // опрос характеристик материала
```

Первый аргумент в этих процедурах задает грань объекта, в его качестве может выступать одна из следующих символьных констант:

- `GL_FRONT` — передняя (по отношению к источнику света) грань;
- `GL_BACK` — задняя (по отношению к источнику света) грань;
- `GL_FRONT_AND_BACK` — передняя и задняя грань.

Задание материала только передней грани позволяет сократить время рендеринга, т. к. в этом случае не производится никаких вычислений, связанных с освещенностью задней грани. Перечень имен параметров, значения компонентов которых задаются четырехкомпонентными векторами, приведен в табл. 16.6. В отличие от модели источника света, значения коэффициентов отражения задаются в диапазоне [–1, +1].

Таблица 16.6

Имя параметра	Смысл и значение параметра
`GL_AMBIENT`	Коэффициенты отражения RGB-лучей фонового излучения, обычно задаваемые вектором (`RkA`, `GkA`, `BkA`, `1`). По умолчанию — (`0.2`, `0.2`, `0.2`, `1.0`)

Таблица 16.6 *(окончание)*

Имя параметра	Смысл и значение параметра
GL_DIFFUSE	Коэффициенты отражения RGB-лучей диффузного излучения, обычно задаваемые вектором (RkD, GkD, BkD, 1). По умолчанию — (0.8, 0.8, 0.8, 1.0)
GL_SPECULAR	Коэффициенты отражения RGB-лучей зеркального излучения, обычно задаваемые вектором (RkS, GkS, BkS, 1). По умолчанию — (0, 0, 0, 1)
GL_EMISSION	Интенсивность собственного свечения грани (например, покрытой светящимся слоем). По умолчанию — (0, 0, 0, 1)
GL_AMBIENT_AND_DIFFUSE	Экономное задание при совпадении значений двух параметров (для фонового и диффузного излучений)

В листинге 16.10 приводится пример взаимодействия единственного источника света с гранями вращающегося куба. Идею примера нам подсказал один из уроков по OpenGL, обнаруженный в Интернете, но предлагаемый нами вариант программы вдвое короче.

Листинг 16.10. Использование источника света

```
program light;
{$mode objfpc}
uses
  gl, glu, glut;
var
  angle: single = 0;
  step : single = 0.1;
  diffuseL: array[0..3] of single = (0.8, 0.8, 0.8, 1);
//-------------------------------------------------------
procedure OnRedraw; cdecl;
begin
  glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
  glLoadIdentity;
  glTranslatef(0, 0, -5);
  glRotatef(angle, 1, 0, 0);
  glRotatef(angle, 0, 1, 0);
  glRotatef(angle, 0, 0, 1);
  glColor3f(1, 1, 1);
  glutSolidCube(2);
  angle := angle + step;
```

```
  glutSwapBuffers;
end;
//-----------------------------------------------------------
procedure OnResize(W, H: integer); cdecl;
begin
  glViewport(0, 0, W, H);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity;
  gluPerspective(45, W / H, 0.1, 1000);
  glMatrixMode(GL_MODELVIEW);
end;
//-----------------------------------------------------------
begin
  glutInit(@argc,argv);
  glutInitDisplayMode(GLUT_DOUBLE or GLUT_RGB or GLUT_DEPTH);
  glutCreateWindow('Rotating Cube + Light');
  glClearColor(0, 0.8, 0.2, 0);
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_LIGHTING);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseL);
  glEnable(GL_LIGHT0);
  glEnable(GL_COLOR_MATERIAL);
  glutDisplayFunc(@OnRedraw);
  glutReshapeFunc(@OnResize);
  glutIdleFunc(@OnRedraw);
  glutMainLoop;
end.
```

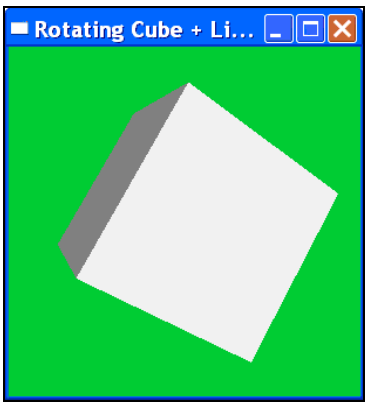


Рис. 16.9. Освещенный куб

На рис. 16.9 приведен один из кадров вращающегося куба, на котором можно рассмотреть разницу между освещенностью видимых граней.

16.16. Вместо эпилога

Пакет OpenGL насчитывает порядка 250 функций. Около 150 функций представлены в библиотеках GLU и GLUT. Тот небольшой пласт, который нам удалось продемонстрировать в разделах этой главы, дал вам в руки инструмент, по мощности заметно превосходящий функциональные возможности графической библиотеки BGI. Что осталось за границей изложенного материала?

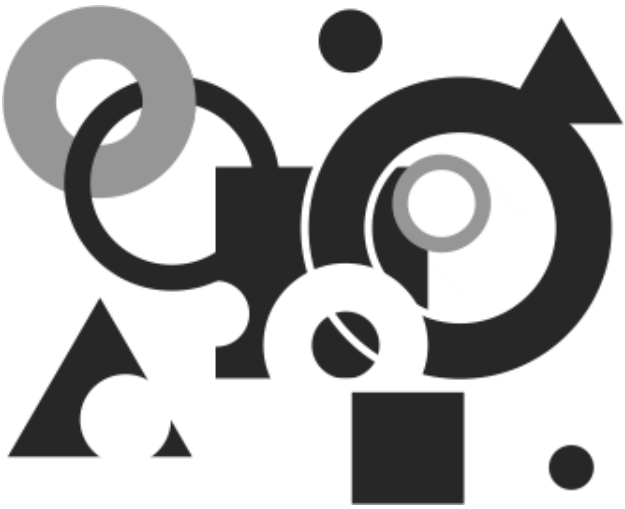
В нашей книге отсутствует информация, связанная с построением плоских и пространственных кривых (сплайны Безье, B-сплайны, NURBS-кривые), с моделированием поверхностей, с выполнением операций над кривыми и поверхностями. Дело в том, что эта тематика требует довольно серьезного знакомства с математическими основами таких дисциплин как аналитическая и дифференциальная геометрия, теория приближений, методы вычислений.

Мы не затронули проблемы повышения реальности изображений за счет наложения плоских и выпуклых текстур на грани объекта. Сама по себе процедура текстурирования напоминает наклеивание обоев на стены в наших квартирах. Правда, для объектов с криволинейными контурами наклеиваемая текстура должна принимать форму объекта и "растягиваться" по размерам соответствующих сторон. Есть некоторые технические нюансы, которые следует учитывать при использовании внешних графических файлов в качестве текстур. Во-первых, ширина и высота прямоугольного растра должны быть заданы степенями двойки, например 128×256. Во-вторых, функции OpenGL, обеспечивающие наложение текстур, используют формат кода цветности, отличный от стандарта, принятого в операционной системе Windows. Функции Windows GDI работают с последовательностью компонентов A-B-G-R, тогда как OpenGL предпочитает кодировку A-R-G-B. Эти два обстоятельства должна учитывать функция, считывающая образ текстуры в оперативную память, иначе перестановка компонентов цветности должна выполняться программой пользователя. Наконец, в составе библиотек OpenGL, GLU и GLUT такой функции просто нет. В системе Delphi роль такой функции выполняет процедура `LoadFromFile`. Следовательно, для работы с внешними текстурами в среде FP IDE необходимо привлекать дополнительную графическую библиотеку. Попытка использовать библиотеку Vampyre Imaging Library, рекомендованную на сайте пользователей Free Pascal, была забракована из-за дополнительных сложностей, связанных с обилием поддерживаемых графических форматов. Одно их перечисление занимает порядка двух страниц.

Довольно важным приемом создания более реальных изображений является построение теней, отбрасываемых объектом при наличии одного или нескольких источников света. Дополнительную возможность, повышающую реальность сцены,

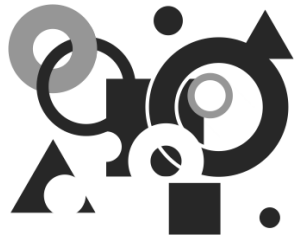
предоставляет "задымление" объектов, которые предполагается разместить на заднем плане. Имитация тумана размывает контуры и отдаляет такие объекты от позиции наблюдателя.

Три технических приема, перечисленные выше, конечно, могут расширить набор ваших дизайнерских средств. Однако современная машинная графика, используемая при создании коммерчески успешных проектов типа спецэффектов для 3D-фильма "Аватар", требует гораздо более глубоких знаний в области алгоритмов и программных средств анимации, реализуемых с помощью специализированной многопроцессорной аппаратуры. То, с чем вы познакомились в двух последних главах нашей книги, — это только первые ступеньки на пути постижения основ и секретов машинной графики в среде FP IDE.



ПРИЛОЖЕНИЯ

Приложение 1



Синтаксис и семантика языка Free Pascal

П1.1. Краткая справка по типам данных

- Формат объявления типа одиночных (скалярных) переменных:

```
type
  имя_типа = тип;

var
  ...
  имя_переменной : тип;
  имя_переменной : имя_типа;
```

- Формат объявления типа элементов массива:

```
type
  имя_типа = array [...] of тип;

var
  ...
  имя_массива : array [...] of тип;
  имя_массива : имя_типа;
```

- Характеристики типов данных представлены в табл. П1.1—П1.4.

Таблица П1.1. *Целочисленные данные*

Тип	Длина (байты)	Минимальное значение	Максимальное значение
`Byte`	1	0	$2^8 - 1$
`ShortInt`	1	-2^7	$2^7 - 1$
`Word`	2	0	$2^{16} -$

Таблица П1.1 *(окончание)*

Тип	Длина (байты)	Минимальное значение	Максимальное значение
`SmallInt`	2	-2^{15}	$2^{15}-1$
`LongWord, Cardinal`	4	0	$2^{32}-1$
`Integer, LongInt`	4	-2^{31}	$2^{31}-1$
`QWord`	8	0	$2^{64}-1$
`Int64`	8	-2^{63}	$2^{63}-1$

Таблица П1.2. *Вещественные данные*

Тип	Длина (байты)	Минимальное значение (по модулю)	Максимальное значение (по модулю)	Число десятичных цифр
`Single`	4	10^{-38}	10^{38}	7—8
`Real48`	6	10^{-38}	10^{38}	11—12
`Double, Real`	8	10^{-308}	10^{308}	15—16
`Comp`	8	0	$2^{63}-1$	19—20
`Currency`	8	10^{-4}	10^{15}	19—20
`Extended`	10	10^{-4932}	10^{4932}	19—20

Таблица П1.3. *Данные логического типа*

Тип	Длина (байты)	Значения	`Ord(False)`[1]	`Ord(True)`
`Boolean`	1	`True` | `False`	0	1
`ByteBool`	1	`True` | `False`	0	Любое значение $\neq 0$
`WordBool`	2	`True` | `False`	0	Любое значение $\neq 0$
`LongBool`	4	`True` | `False`	0	Любое значение $\neq 0$

[1] Данные логического типа относятся к порядковым данным языка Паскаль. Значение функции `Ord` позволяет определить последовательность значений `False` и `True` (обычно `False` < `True`, соответственно значения функции `Ord` равны 0 и 1). Но состав логических данных в Object Pascal расширен для совместимости с другими алгоритмическими языками. Так как в этом случае сказываются ограничения других систем программирования, то имеет место и отклонение в значениях функции `Ord`.

Таблица П1.4. *Данные символьного типа*

Тип	Длина (байты)	Диапазон допустимых значений
`Char`	1	Любой отображаемый символ ASCII, заключенный в одинарные кавычки или заданный своим числовым кодом от 0 до 255. Управляющие символы с кодами от 1 до 26 иногда задают парой символов, первый из которых — `^`, а второй — прописная буква латинского алфавита. Числовой код такого "символа" равен порядковому номеру буквы в алфавите. Например, сочетание `^M` соответствует управляющему символу CR (возврат каретки — символ с кодом 13)

- Формат объявления типа данных перечислимого типа:

```
type
   имя_перечисления = (список обозначений);
```

Элементам списка обозначений присваиваются последовательные номера, начиная с нуля. Подобно языку C, в системе Free Pascal имеется возможность нарушить монотонность этой нумерации при соблюдении единственного правила: номера элементов списка могут только возрастать (не обязательно на единицу).

- Формат объявления данных интервального типа:

```
var
...имя_переменной : min .. max;
```

Минимальное (*min*) и максимальное (*max*) значения, задающие границы интервала, должны принадлежать одному и тому типу порядковых данных. Например, оба представлены целыми числами, символами ASCII, элементами перечисления.

- Формат объявления данных строкового типа (табл. П1.5):

```
type
  имя_типа = тип;


var
......
  имя_переменной : тип;
  имя_переменной : тип[n];
  имя_переменной : имя_типа;
```

Здесь *n* — максимальное количество символов, которое задается только для коротких строк типа `String` (`n` < 256).

Таблица П1.5. *Данные строкового типа*

Тип	Длина строки (байты)	Пояснение
`String`	256 или до 2 Гбайт	По директиве `{$H-}` рассматривается как короткая строка максимальной длины. По умолчанию или по директиве `{$H+}` рассматривается как `AnsiString`. Каждый символ в такой строке представлен однобайтовым кодом (кодировка ASCII). Индексы символов отсчитываются от 1
`String[n]`	`N` + 1	Короткая строка заданной длины. Индексы символов отсчитываются от 1 до `n`
`ShortString`	256	Короткая строка максимальной длины
`AnsiString`	До 2 Гбайт	Указатель на длинную строку переменной длины, расположенную в "куче". Каждый символ в такой строке представлен однобайтовым кодом (кодировка ASCII). Индексы символов отсчитываются от 1
`PChar`	До 2 Гбайт	Указатель на длинную строку переменной длины, расположенную в "куче" и завершающуюся признаком конца строки (т. е. байтом с нулевым кодом). Каждый символ в такой строке представлен однобайтовым кодом (кодировка ASCII). Индексы символов отсчитываются от 0
`WideString`	До 2 Гбайт	Указатель на длинную строку переменной длины, расположенную в "куче" и завершающуюся "широким" признаком конца строки (т. е. двумя байтами с нулевым кодом). Каждый символ в такой строке представлен "широким" двухбайтовым кодом (кодировка Unicode). Индексы символов отсчитываются от 0

- Формат объявления данных типа "запись":

```
type
  имя_типа = record
     имя_поля_1 : тип_1;
     имя_поля_1 : тип_2;
     ...
     имя_поля_k : тип_k;
  end;

var
  ...
  имя_переменной : имя_типа;
```

- Формат объявления данных файлового типа (табл. П1.6):

```
type
  имя_типа = тип;
```

```
var
......
  имя_переменной : тип;
  имя_переменной : имя_типа;
```

Таблица П1.6. *Данные файлового типа*

Тип	Пояснение
`Text`	Указатель на блок управления файлом *текстового* типа. В таком файле каждая запись представлена строкой символов, завершающейся признаком конца строки (End-Of-Line — EOL)
`File of` *имя_типа*	Указатель на блок управления *типизированным* файлом. В таком файле каждая запись имеет фиксированную длину и содержит поля данных, формат которых определяется *именем_типа*
`File`	Указатель на блок управления *нетипизированным* (*двоичным*) файлом. Обмен данными с таким файлом осуществляется порциями *(блоками)* фиксированной длины (по умолчанию — по 128 байт). Байты блока могут содержать двоичную информацию любой природы. Расшифровка содержимого блоков полностью определяется алгоритмом работы приложения

П1.2. Краткая справка по операторам языка Free Pascal

Основные операторы представлены в табл. П1.7.

Таблица П1.7. *Операторы языка Free Pascal*

Формат оператора	Пояснение
Элементарные операторы (ES — Elementary Statements)	
`v:=e;`	**Оператор присваивания.** Как правило, тип значения выражения `e` и тип переменной `v` должны совпадать. Единственное исключение — присвоение целочисленных значений переменным вещественного типа
`v1:=v2:=...:=e;`	**Оператор множественного присваивания.** Значение выражения `e` присваивается нескольким переменным
`v ⊗= e;`	Краткая запись оператора `v:=v⊗e;`, заимствованная из языка C. В качестве знака операции ⊗ могут выступать операции сложения (`+`), вычитания (`-`), умножения (`*`) и деления (`/`). Этот формат может использоваться при включении в текст программы директивы `{$COPERATORS ON}`

Таблица П1.7 *(продолжение)*

Формат оператора	Пояснение
Элементарные операторы (ES — Elementary Statements)	
`n_proc(p1,p2,...);`	**Оператор вызова процедуры** с именем `n_proc` и передачи ей списка фактических параметров
`n_proc;`	Оператор вызова процедуры без параметров
`read(v1,v2,...);`	**Оператор ввода данных** (точнее, вызов системной процедуры) с клавиатуры. Числовые значения, набираемые на клавиатуре, должны отделяться друг от друга одним или несколькими пробелами. Первое числовое значение присваивается переменной `v1`, второе — переменной `v2` и т. д. При вводе строкового значения имя соответствующей строковой переменной может быть *только последним* в списке ввода. Признаком завершения ввода является нажатие клавиши <Enter>. После этого курсор дисплея располагается вслед за последним введенным символом
`readln(v1,v2,...);`	Аналогичный оператор (процедура) с переводом курсора дисплея в начало следующей строки
`write(e1,e2,...);`	**Оператор вывода данных** (точнее, вызов системной процедуры) на экран дисплея. Значение очередного выражения списка переводится из машинного формата в символьное представление в соответствии с типом значения и выдается на экран дисплея. Если числовой элемент списка вывода сопровождается дополнительным указанием типа `e:n` или `e:n:m`, то для его отображения в текущей строке вывода резервируется `n` позиций. В первом случае на выделенном поле с прижимом к правой границе располагается целая часть числа. Во втором случае числовое значение выводится с `m` цифрами в дробной части. Такое форматирование при выводе числовой информации позволяет добиться общепринятого представления табличных данных. После вывода курсор дисплея остается в первой свободной позиции текущей строки вывода
`writeln(e1,e2,...);`	Аналогичный оператор (процедура) с переводом курсора дисплея в начало следующей строки

`label`
`  m;`
`  ...`
`  goto m;`
 	
`m:` *`оператор`*`;`
	Безусловный переход на оператор с меткой `m`. В качестве метки может выступать либо целое число, либо алфавитно-цифровой идентификатор. В любом случае метка `m` должна быть объявлена в разделе `label` и обязана находиться в той же программной единице (процедуре, функции или головной программе), где использован оператор `goto`. От оператора, на который передается управление, метка отделяется двоеточием

***Таблица П1.7** (продолжение)*

Формат оператора	Пояснение
Составной оператор (CS — Compound Statement)	

`begin`
`  AS1;`
`  AS2;`
`......`
`end`
	Составные операторы используются для объединения нескольких произвольных операторов (AS — Arbitrary Statement) в один более сложный оператор. Именно такие конструкции должны использоваться в структурных единицах типа "циклы" и "условные операторы"

Структурные операторы (SS — Structured Statements)	
`if` *условие* `then` `  CS;`	**Условный оператор** (укороченный формат). Если условие соблюдается, то выполняется оператор `CS`. В противном случае выполняется оператор, следующий за `if`

`if` *условие* `then`
`  CS1`
`else`
`  CS2;`
	Условный оператор (полный формат). При соблюдении условия выполняется оператор `CS1`, в противном случае выполняется оператор `CS2`

`case e of`
`  c1:AS1;`
`  c2:AS2;`
`  ...`
`[ else`
`  AS;]`
`end;`
	Оператор выбора (переключатель). Значением переключающего выражения `e` может быть величина порядкового типа (целое число, символ, элемент перечисления). Если значение `e` равно константе `c1`, то выполняется оператор `AS1` (им, в частности, может быть составной оператор) и вслед за ним происходит выход из переключателя. При `e=c2` выполняется оператор `AS2` с последующим выходом из переключателя и т. д. Если значение `e` не совпадает ни с одной из перечисленных констант, то выполняются операторы, расположенные вслед за `else`. Фрагмент с `else` может отсутствовать (сокращенная версия переключателя).
	
	Условие переключения может быть задано не только одной константой:
	`c1,c2,c3:AS1;  //список констант`
	`c4..c5  :AS2;  //диапазон значений`
	
	Синонимом служебного слова `else` является `otherwise`

`for cv:=v1 to v2 do` `  CS;`	**Оператор цикла со счетчиком повторений.** В качестве управляющей переменной `cv` (счетчика цикла) можно использовать только переменную порядкового типа. Цикл начинается с вычисления конечного значения счетчика `v2`. Если это значение меньше начального значения счетчика `v1`, то тело цикла (оператор `CS`) не выполняется. Если `v1<v2`, то цикл начинается с засылки в переменную `cv` ее начального значения `v1`. После очередного выполнения тела цикла значение счетчика увеличивается на 1 и сравнивается с его конечным значением `v2`. Цикл повторяется до тех пор, пока значение счетчика меньше или равно `v2`. Попытка изменить значение счетчика внутри тела цикла считается ошибкой

Таблица П1.7 *(окончание)*

Формат оператора	Пояснение
Структурные операторы (SS — Structured Statements)	
`for cv:=v1 downto v2 do` `  CS;`	**Оператор цикла с обратным пересчетом.** Начальное значение счетчика `v1` должно быть больше конечного `v2`. Перед очередным повторением цикла из счетчика вычитается 1

`repeat`
`  AS1;`
`  AS2;`
`.....`
`until` *условие*`;`
	Цикл с постусловием. Сначала выполняется цепочка операторов, расположенных между `repeat` и `until` (тело цикла). Затем проверяется выполнение условия, и в случае его *нарушения* тело цикла повторяется. Цикл завершает свою работу, когда условие становится истинным

`while` *условие* `do` `  CS;`	**Цикл с предусловием.** Сначала проверяется условие, и если оно не нарушено, то выполняется оператор `CS` (тело цикла). Затем все повторяется, начиная с проверки условия. В случае нарушения условия цикл завершает свою работу

`with name_rec do`
`  name_fld1:=...;`
`  read(name_fld2);`
`  ...`
`end;`
	Конструкция `with` позволяет сократить запись наименований полей записи. В приведенном примере вместо составных имен `name_rec.name_fld1`, `name_rec.name_fld2` использованы более короткие имена

П1.2.1. Специфика описания подпрограмм (процедур и функций)

- Описание процедуры:

```
procedure имя_проц [(список_формальных_параметров)]; [директивы];
  Блок подпрограммы
end;
```

- Описание функции:

```
function имя_функ[(список_формальных_параметров)] : тип; [директивы];
  Блок вычисления значения функции
  Возврат значения функции
end;
```

Список формальных параметров состоит из элементов, между которыми в качестве разделителя используется точка с запятой. Каждый элемент списка может содержать до трех полей:

```
[ характеристика ] имя [ : тип ]
```

Перечень возможных характеристик приведен в табл. П1.8.

Таблица П1.8

Характеристика	Способ передачи и использования параметра
Отсутствует	Параметр передается *по значению*. Это значение, как правило, используется в правой части оператора присваивания (rvalue — от right value). В теле подпрограммы имя параметра эквивалентно локальной переменной. Ее значение можно менять, но вызывающая программа никакой информации об этом не получит
`var`	Параметр передается *по адресу*. Значение такого параметра может использоваться как в левой части оператора присваивания (lvalue — от left value), так и в правой (rvalue). Последнее изменение параметра, выполненное в теле подпрограммы, возвращается в вызывающую программу
`out`	Параметр передается *по адресу*. В текущей версии FP характеристики `out` и `var` эквивалентны. Обычно приставка `out` обозначает *выходной* параметр подпрограммы, который используется только в левой части оператора присваивания
`const`	Используется для обозначения входного параметра, имя которого может находиться только в правой части оператора присваивания (rvalue). В зависимости от длины значения передается либо *по адресу* (длина значения превышает 4 байта), либо *по значению.* Значение такого параметра или элементов такого массива в подпрограмме менять нельзя

Если в элементе списка отсутствует указание о типе, то подпрограмма получает *адрес нетипизированного параметра*. Обращению по такому адресу обязательно должно предшествовать приведение к нужному типу. Для этого можно воспользоваться либо явным приведением к типу (например, `y:=Integer(x);`), либо заданием соответствия между таким адресом и типизированной локальной переменной подпрограммы:

```
var
  y: Double absolute x;
```

Важную роль в списке формальных параметров играют *открытые массивы*:

```
procedure proc1(...; a:array of integer; ...);
```

В качестве параметра `a` вызывающая программа может передать любой одномерный целочисленный массив, например массив `b`, объявленный следующим образом:

```
var
  b: array [3..10] of integer;
```

При этом между элементами массивов `a` и `b` устанавливается следующее соответствие:

- элементу `b[3]` соответствует элемент `a[0]`;

- элементу `b[4]` соответствует элемент `a[1]`;
- ...
- элементу `b[10]` соответствует элемент `a[High(a)]`.

Несколько элементов списка формальных параметров, расположенных подряд и имеющих одинаковые атрибуты, могут быть описаны более компактно. Например, вместо:

```
function f1(a:double; b:double; c:double):double;
```

можно записать:

```
function f1(a,b,c:double):double;
```

Нескольким последним элементам списка формальных параметров могут быть приписаны *значения по умолчанию*. Например:

```
function sum(a1,a2:integer;a3:integer=3;a4:integer=4):integer;
begin sum:=a1+a2+a3+a4; end;
```

К такой функции можно обращаться с двумя, тремя и четырьмя параметрами:

```
sum(1,2)=10
sum(1,2,-2)=5
sum(1,2,-2,-1)=0
```

Вместо опущенных параметров подставляются их значения по умолчанию.

Система Free Pascal допускает использование более полутора десятков директив, размещаемых поодиночке или группами в заголовках функций и процедур. Большинство из них связано с вызовами подпрограмм, написанных в других системах программирования. Одну из них — директиву `cdecl` — мы использовали при общении с графическими пакетами OpenGL, GLU и GLUT. Другая директива — `forward` — была необходима при построении цепочки рекурсивных процедур. На начальной стадии освоения программирования вам вряд ли потребуются другие директивы.

- Способы возврата значения функции.

  Система Free Pascal допускает четыре варианта возврата вычисленного значения функции. Традиционный способ заключается в присвоении возвращаемого значения имени функции:

  ```
  имя_функции := e;
  ```

  Второй вариант, появившийся в языке Object Pascal, заключается в использовании системной переменной `Result`:

  ```
  Result := e;
  ```

  Третий способ заимствован из языка C:

  ```
  return e; //с одновременным выходом из функции
  ```

  Четвертый способ использует модифицированную системную функцию `exit`:

  ```
  exit(e);  // с одновременным выходом из функции
  ```

  Второй способ нам представляется наиболее предпочтительным по двум обстоятельствам. Во-первых, он обладает повышенной наглядностью. Во-вторых,

имя системной переменной `Result` может использоваться и как lvalue, и как rvalue. Имя функции может выступать только как lvalue.

- Формат объявления переменных процедурного типа.

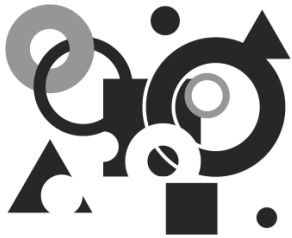
  Среди элементов списка формальных параметров подпрограммы `A` могут присутствовать имена других функций и процедур. Это и есть параметры *процедурного типа*. Специфика их описания состоит в том, чтобы сообщить компилятору порядок и типы аргументов таких функций или процедур. Это понадобится для организации их правильного вызова из тела подпрограммы `A`. Для вызываемых функций дополнительно необходимо указание типа возвращаемого значения.

  Собственно описание данных процедурного типа производится в вызывающей программе и выглядит следующим образом:

```
type
  имя_типа_1 = procedure(список формальных параметров);
  имя_типа_2 = function(список формальных параметров): тип_значения;
```

  В заголовке подпрограммы `A` соответствующему формальному параметру приписывается тип *имя_типа_1* или *имя_типа_2*. При обращении к подпрограмме `A`, использующей указанный параметр, в качестве фактического аргумента задается настоящее имя передаваемой процедуры или функции. На самом деле, вызываемая подпрограмма получает адрес точки входа в программу, реализующую указанную процедуру или функцию.

Приложение 2



Настройка среды и системы

П2.1. Файлы управления работой системы FP IDE

Информация о конфигурации системы и параметрах, определяющих ее работу, сосредоточена в трех файлах:

- fp.dsk — файл, в котором запоминается конфигурация рабочего стола (desktop);
- fp.cfg — файл, в котором запоминается конфигурация интегрированной среды;
- fp.ini — файл, в котором хранится информация, определяющая основной набор управляющих параметров в момент старта интегрированной среды.

Два первых файла формируются в процессе работы системы по параметрам, задаваемым или изменяемым пользователем с помощью различных команд настройки. Их формат соответствует требованиям операционной системы. Третий файл — обычный текстовый файл, который пользователь может изменять, если четко знает смысл и перечень возможных значений соответствующих параметров.

Все три файла могут храниться либо в системном каталоге ...bin\i386-win32, либо в каталоге пользователя, который он может назначить с помощью команды **Open** из меню **Options**.

Содержимое файла fp.ini приводится в листинге П2.1, и с ним полезно познакомиться не только начинающему пользователю.

Листинг П2.1. Содержимое файла fp.ini

```
[Compile]
CompileMode=DEBUG

[Editor]
DefaultTabSize=8
DefaultFlags=20599
DefaultSaveExt=.pas
DefaultIndentSize=1
```

```
[Highlight]
Exts="*.pas;*.pp;*.inc"
NeedsTabs="make*;make*.*"

[SourcePath]
SourceList=""

[Mouse]
DoubleDelay=8
ReverseButtons=0
AltClickAction=6
CtrlClickAction=1

[Search]
FindFlags=4

[Breakpoints]
Count=0

[Watches]
Count=0

[Preferences]
DesktopFileFlags=209
CenterCurrentLineWhileDebugging=1
AutoSaveFlags=7
MiscOptions=6
DesktopLocation=1

[Misc]
ShowReadme=0

[Help]
Files="C:\FPC\2.2.4\html\fpctoc.htx|HTML Index"

[Keyboard]
EditKeys=Borland

[Files]
OpenExts="*.pas;*.pp;*.inc"
RecentFile1=C:\FPC\MyProg\indent.pas,49,18
```

```
RecentFile2=C:\FPC\MyProg\txt_in1.pas,16,10
RecentFile3=C:\FPC\MyProg\txt_inout.pas,13,6
PrinterDevice=prn

[Tools]
Title1="svn ~u~p (curr. dir)"
Program1="svn"
Params1="up $CAP_MSG()"
HotKey1=23296
Title2="svn c~i~ (curr. dir)"
Program2="svn"
Params2="ci $CAP_MSG()"
HotKey2=23552
Title3="svn ~d~iff"
Program3="svn"
Params3="diff $CAP_MSG() $EDNAME"
HotKey3=23808
Title4="svn ~l~og"
Program4="svn"
Params4="log $CAP_MSG() $EDNAME"
HotKey4=34560
Title5="svn ~b~lame"
Program5="svn"
Params5="blame $CAP_MSG() $EDNAME"
HotKey5=34816
Title6="svn ~a~dd"
Program6="svn"
Params6="add $CAP_MSG() $EDNAME"
HotKey6=0
```

По секциям этой распечатки можно расшифровать значения большинства параметров, управляющих работой системы. Однако точной информации о значении разрядов различных битовых шкал в документации и файлах помощи нет.

П2.1.1. Секция *Compile* (Компиляция)

По умолчанию среда FP IDE стартует с отладочным режимом компиляции (**Debug**), в котором создается дополнительная информация, позволяющая использовать встроенные средства отладки.

П2.1.2. Секция *Editor* (Редактор)

В этом разделе зафиксированы стартовые значения трех параметров по умолчанию:

- `DefaultTabSize` — шаг перемещения курсора в поле редактора при нажатии клавиши <Tab>;
- `DefaultSaveExt` — расширение, автоматически добавляемое к имени запоминаемого файла;
- `DefaultIndentSize` — шаг автоматического смещения при наборе вложенных конструкций.

Значение переменной `DefaultFlags` представляет логическую шкалу признаков, смысл отдельных разрядов которой в документации не описан.

П2.1.3. Секция *Highlight* (Подсветка)

Стартовый режим системы обеспечивает подсветку синтаксических конструкций для содержимого файлов с расширениями pas, pp, inc. Именно такую цепочку расширений можно увидеть в окне параметров редактора (см. команду **Options → Environment → Editor**).

По поводу параметра `NeedsTabs` подробная информация отсутствует. Очевидно, что он связан с указанием об обязательном использовании символа Tab.

П2.1.4. Секция *SourcePath* (Путь к исходным программам)

Пустой путь (значение `SourceList`) соответствует каталогу, из которого стартовала система FP IDE. Например:

```
C:\FPC\2.2.4\bin\i386-win32
```

Значение переменной `SourceList` может быть изменено с помощью команд главного меню **File → Change dir** и **Run → Run Directory**.

П2.1.5. Секция *Mouse* (Мышь)

В этом разделе зафиксированы стартовые значения четырех параметров:

- `AltClickAction` — команда, соответствующая нажатию правой кнопки мыши в сочетании с клавишей <Alt> (по умолчанию — вызов браузера для просмотра информации об объекте, на который нацелен указатель мыши);
- `DoubleDelay` — задержка между двумя последовательными нажатиями кнопки, отделяющая событие `Click` от `DoubleClick`;
- `CtrlClickAction` — команда, соответствующая нажатию правой кнопки мыши в сочетании с клавишей <Ctrl> (по умолчанию — вызов кадра помощи по синтаксической конструкции, на которую нацелен указатель мыши);

- `ReverseButtons` — указание о том, что нажатие левой или правой кнопки мыши соответствует их стандартному назначению (т. е. ориентировано на пользователя-правшу).

П2.1.6. Секция *Search* (Поиск)

Параметр `FindFlags=4` задает наиболее распространенный режим поиска — в глобальной зоне видимости, вперед, от позиции курсора, игнорируя разницу между большими и малыми буквами.

П2.1.7. Секция *Breakpoints* (Точки останова)

Нулевое значение параметра `Count`, очевидно, означает, что окно со списком точек останова является пустым.

П2.1.8. Секция *Watches* (Контролируемые выражения)

Нулевое значение параметра `Count`, очевидно, означает, что окно со списком контролируемых выражений является пустым.

П2.1.9. Секция *Preferences* (Предпочтения)

Значение параметра `DesktopFileFlags` соответствует некоторому стандартному набору признаков, определяющих состояние рабочего стола в момент старта интегрированной среды. Описание значений каждого разряда шкалы в документации отсутствует.

По поводу ненулевого значения параметра `CenterCurrentLineWhileDebugging` можно предположить, что при пошаговом выполнении программы текущая подсвеченная строка смещается к центру рабочего поля редактора, чтобы не исчезнуть из видимой зоны редактора. Экспериментально установлено, что текущая строка находится не строго в центре поля редактора, но за пределы видимости она не уходит.

Более определенной представляется расшифровка значения параметра `AutoSaveFlags`, свидетельствующего о включении всех трех режимов автосохранения — файлов на поле редактора (Editor files), переменных среды (Environment) и состояния рабочего стола (Desktop).

По значению параметра `MiscOptions` никаких конкретных выводов сделать нельзя.

Ненулевое значение параметра `DesktopLocation`, возможно, определяет порядок размещения на экране и габариты окон системы (среда FP IDE, окно приложения, графический экран приложения).

П2.1.10. Секция *Misc* (Разное)

Предположительно параметр `ShowReadme` предназначен для автоматического отображения справки об интегрированной среде перед ее запуском (окно **About**). Однако эксперимент с заменой нулевого значения на ненулевое это предположение не подтвердил.

П2.1.11. Секция *Help* (Помощь)

Единственная строка этого раздела содержит информацию о расположении файлов помощи (каталог html) и соответствующем его оглавлении (файл fpctoc.htx):

```
Files="C:\FPC\2.4.0\html\fpctoc.htx|HTML Index"
```

Содержимое этой строки приходится записывать в файл fp.ini вручную после формирования каталога помощи и его оглавления (см. команду **Help → Files**).

П2.1.12. Секция *Keyboard* (Клавиатура)

Параметр `EditKeys=Borland` задает режим, при котором редактор поддерживает клавишные команды, использовавшиеся в ранних системах фирмы Borland.

П2.1.13. Секция *Files* (Файлы)

В этом разделе заданы стартовые значения двух параметров:

- `OpenExts` — расширения файлов, которые высвечиваются в диалоговом окне при выполнении команды **File → Open**;
- `PrinterDevice` — имя устройства, на котором выводится содержимое поля редактора по команде **File → Print**.

Содержание остальных строк представлено именами исходных программ, вызывавшихся пользователем в предыдущих сеансах.

П2.1.14. Секция *Tools* (Инструменты)

Строки этого раздела связаны с формированием части меню **Tools**, относящейся к командам системы управления версиями (SVN).

Основное правило, которое мы рекомендуем пользователям — не вносить непосредственно в файл fp.ini никаких изменений, кроме набора пути к файлам помощи. Все остальные правки, которые пользователь посчитает целесообразными, следует выполнять только через команды и подкоманды меню **Options**.

П2.2. Настройка системы в среде FP IDE

Основные характеристики, влияющие на процесс создания программы, устанавливаются на различных вкладках диалоговых окон меню **Options** (см. рис. 3.15).

К команде **Mode** (Режим) приходится прибегать довольно часто, т. к. в процессе разработки программы надо пользоваться отладочным режимом работы компилятора (**Mode=Debug**). Именно в этом режиме компилятор формирует вспомогательные таблицы и делает различные вставки в программу, позволяющие использовать отладочные средства. После завершения отладки следует перейти в режим **Mode=Release**, обеспечивающий изготовление программы без лишних вставок.

По команде **Compiler** (Компилятор) открывается диалоговое окно **Compiler Switches** (Ключи компилятора), содержащее шесть вкладок (см. рис. 3.32):

- **Syntax** — синтаксис языка;
- **Generated code** — генерируемый код;
- **Processor** — процессор;
- **Verbose** — уровень детализации сообщений;
- **Browser** — браузер;
- **Assembler** — ассемблер.

Список возможностей, перечисленных на вкладке **Syntax**, приведен в табл. П2.1.

Таблица П2.1

Характеристики по выбору	Пояснение
Stop after first error	Останов после первой ошибки
Allow LABEL and GOTO	Разрешить метки и оператор `GOTO`
Enable macros	Разрешить использование макросов
Allow inline	Разрешить встроенные функции
Include assertion code	Разрешить использование процедуры `ASSERT`
Use Ansi Strings	Разрешить использование ANSI-строк
Load Kylix compat. unit	Загрузить модуль для совместимости с Kylix
Allow STATIC in objects	Разрешить свойство `STATIC` в объектах
C-like operators	Разрешить С-подобные операторы

Колонка **Compiler mode** (Режим компилятора) позволяет выбрать тот или иной диалект языка Паскаль.

В строках выпадающего списка **Conditional defines** можно набирать условные определения, действующие на период компиляции.

В выпадающем списке **Additional compiler args** можно набрать дополнительные аргументы командной строки, которые будут переданы компилятору.

Вкладка **Generated code** (рис. П2.1) содержит детали, связанные с включением в программу различных проверок и с оптимизацией кода. Информация о заложенных возможностях приведена в табл. П2.2.

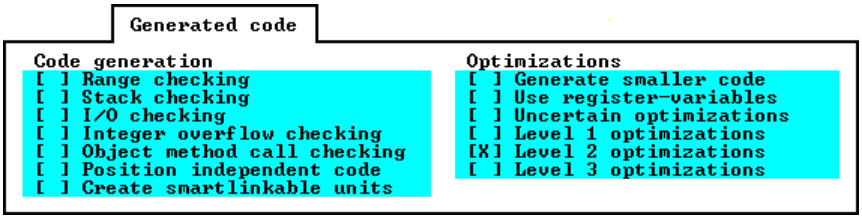


Рис. П2.1. Параметры генерируемого кода

Таблица П2.2

Параметр	Пояснение
Code generation (Генерация кода)	
Range checking	Включить контроль диапазонов
Stack checking	Включить контроль стека
I/O checking	Включить контроль операций обмена
Integer overflow checking	Следить за целочисленным переполнением
Object method call checking	Контролировать вызов методов
Position independent code	Генерировать перемещаемую программу
Create smartlinkable units	Создавать модули с умными связями
Optimizations (Оптимизация)	
Generate smaller code	Включить оптимизацию по длине кода
Use register-variables	Использовать регистровые переменные
Uncertain optimizations	Использовать сомнительную оптимизацию
Level 1 optimizations	Включить оптимизацию уровня 1
Level 2 optimizations	Включить оптимизацию уровня 2
Level 3 optimizations	Включить оптимизацию уровня 3

Контроль диапазонов предусматривает два типа проверок. Во-первых, контролируется выход значений данных за пределы допустимого диапазона. Во-вторых, контролируется принадлежность индексов объявленным границам в массивах. Модули с "умными" связями отличаются от стандартно изготавливаемых модулей тем, что использование "умных" подпрограмм не влечет за собой присоединение к про-

грамме модуля целиком. К программе подключается только вызываемая процедура и те функции, к которым она обращается.

Вкладка **Processor** (см. рис. 3.33) предусматривает выбор процессора, на котором должна исполняться создаваемая программа, и возможность включения оптимизации создаваемого кода.

Вкладка **Verbose** (рис. П2.2) предусматривает возможность вывода из программы дополнительных сообщений — *предупреждений* (**Warnings**), *замечаний* (**Notes**), *подсказок* (**Hints**) и др.

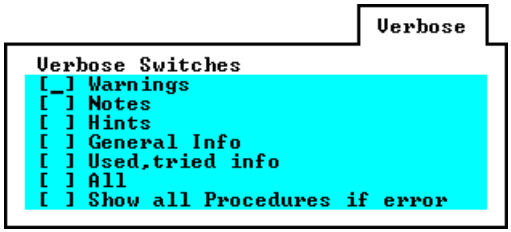


Рис. П2.2. Выбор сообщений компилятора

Вкладка **Browser** (рис. П2.3) предусматривает один из трех вариантов работы с браузером — просмотр только глобальных данных, просмотр глобальных и локальных данных, отключение браузера.

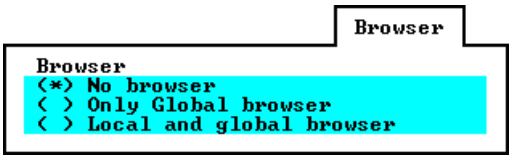


Рис. П2.3. Установка режима работы браузера

Компилятор FPC осуществляет трансляцию в два этапа. На первом этапе исходная программа на одном из диалектов языка Free Pascal переводится в код на языке ассемблера. На втором этапе работает компилятор с языка ассемблера. Поэтому на вкладке **Assembler** (рис. П2.4) присутствует несколько панелей, определяющих как формат промежуточного кода (стиль Intel или AT&T), так и выбор компилятора с языка ассемблера. По умолчанию FP IDE использует компилятор ga.exe. На панели **Assembler info** можно выбрать состав информации, генерируемой вместе с кодом программы на языке ассемблера. Имеется возможность сохранить код программы и текст сопутствующих таблиц. В стандартном режиме все эти файлы создаются как временные, которые уничтожаются после завершения компиляции.

Команда **Memory sizes** открывает окно (рис. П2.5), в котором можно переназначить размеры кучи и стека. Нулевые размеры означают установку обоих параметров по умолчанию.

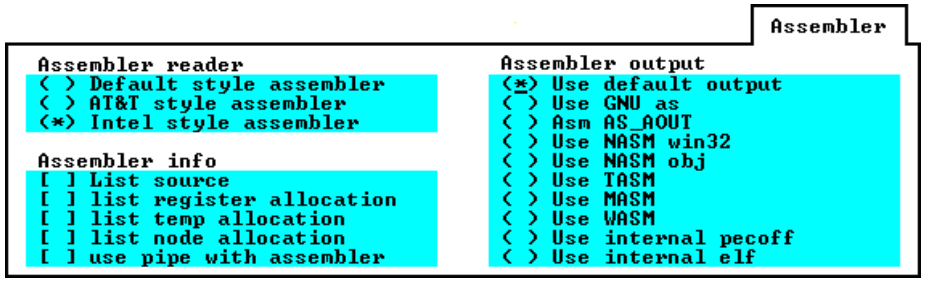


Рис. П2.4. Выбор компилятора с языка ассемблера

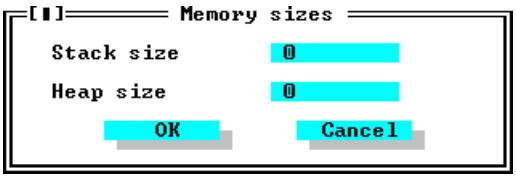


Рис. П2.5. Окно для задания размеров памяти

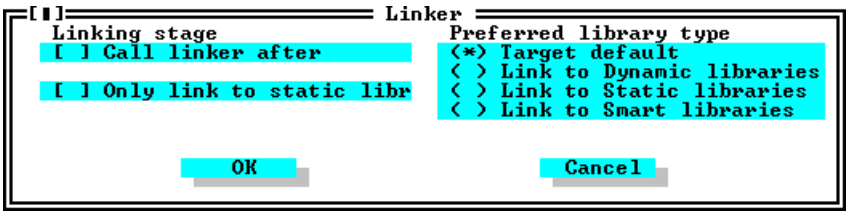


Рис. П2.6. Окно редактора связей (линковщика)

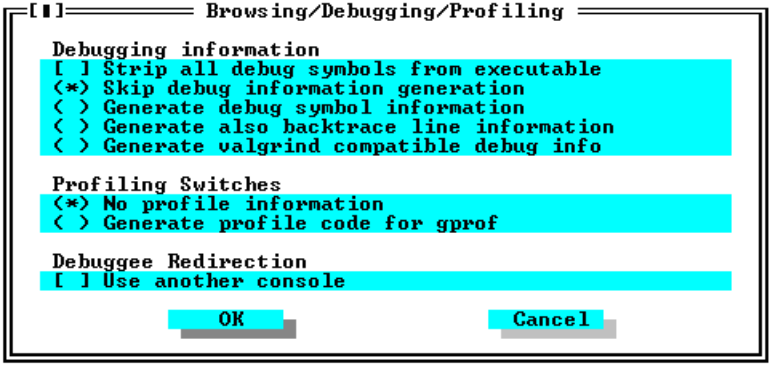


Рис. П2.7. Окно параметров управления отладкой и профилированием

По команде **Linker** вызывается окно, управляющее работой редактора связей (рис. П2.6). Вызов редактора связей может быть отложен на более поздний срок (**Call linker after**), если задачей сеанса является изготовление только объектного

модуля. Можно задать режим, при котором **Linker** будет учитывать связи только с подпрограммами статических библиотек (**Only link to static libr**).

Команда **Debuger** из меню **Options** открывает окно, представленное на рис. П2.7.

Пояснения к параметрам отладки приведены в табл. П2.3.

Таблица П2.3

Параметр	Пояснение
Debugging information (Отладочная информация)	
Strip all debug symbols from executable	Удалить все отладочные объекты из exe-файла
Skip debug information generation	Пропустить генерацию отладочной информации
Generate debug symbol information	Генерировать информацию об отладочных объектах
Generate also backtrace line information	Дополнительно генерировать информацию о последовательности выполнения строк
Generate valgrind compatible debug info	Генерировать отладочную информацию, совместимую с отладчиком Valgrind
Profiling Switches (Ключи профилирования)	
No profile information	Не создавать информацию о профилировании
Generate profile code for gprof	Генерировать вставки для профайлера gprof.exe
Debuggee Redirection (Перенаправление отладки)	
Use another console	Использовать другую отладочную консоль

Опция **Directories** открывает окно с пятью вкладками (рис. 3.34), в которых можно набрать пути к каталогам, где система должна искать модули (**Units**), подключаемые файлы (**Include files**), библиотеки (**Libraries**) и объектные файлы (**Object files**), используемые в программе.

Вкладка **Misc** (от англ. *miscellaneous* — разное) содержит параметры, представленные на рис. П2.8.

В этом окне могут быть набраны пути для размещения exe-файлов (**EXE output directory**), модулей пользователя (**PPU output directory**), кроссплатформенных средств и динамических библиотек.

Опция **Browser** открывает окно (рис. П2.9), в котором можно установить перечень объектов, доступных для просмотра, и задать некоторые режимы отображения (в частности сортировку по именам).

Опция **Tools** (Инструменты) открывает следующее окно, представленное на рис. П2.10. В этом окне задаются параметры системы SVN, ориентированной на учет

и хранение версий достаточно больших программных проектов. Собственно система SVN в состав интегрированной среды не входит, ее надо приобретать отдельно.

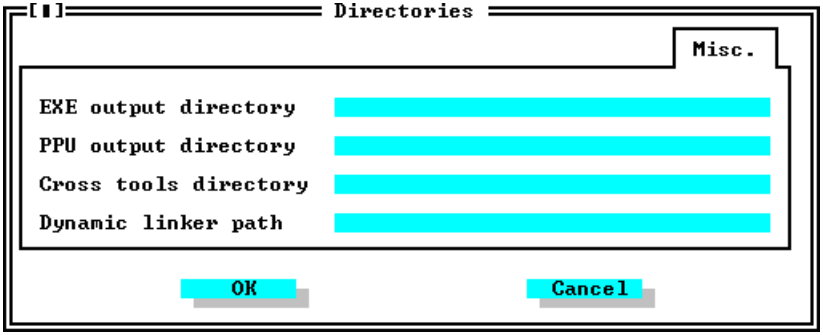


Рис. П2.8. Окно набора дополнительных каталогов

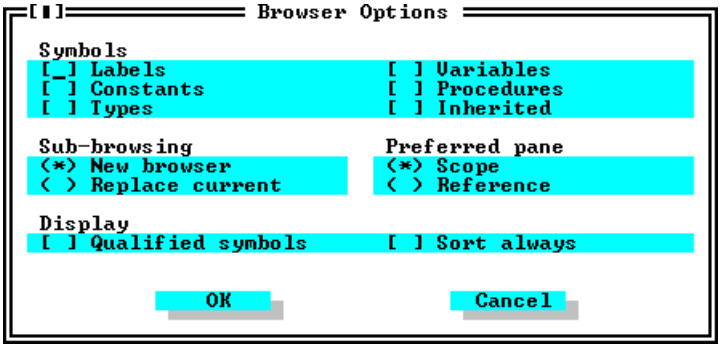


Рис. П2.9. Окно для задания параметров управления браузером

Опция **Environment** (Окружение) вызывает подменю установки параметров *системных переменных среды* (рис. П2.11).

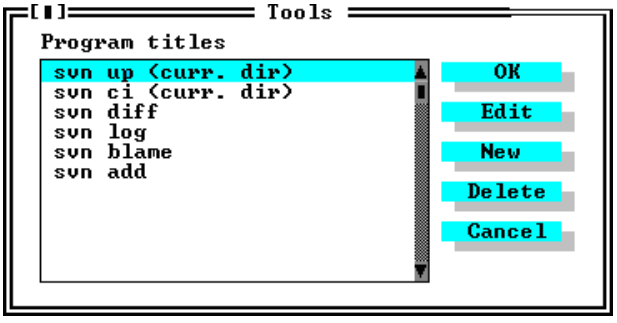


Рис. П2.10. Окно управления параметрами SVN

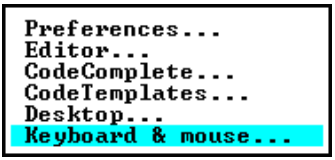


Рис. П2.11. Подменю опции **Environment**

Подкоманда **Preferences** (Предпочтения) открывает диалоговое окно (см. рис. 3.35), в котором можно изменить некоторые параметры среды.

- В панели **Video mode** (Видеорежим) можно выбрать один из вариантов оформления экрана, которые отличаются друг от друга количеством строк (30, 43, 50 или 80) и соответственно высотой окна FP IDE.
- Панель **Desktop file** (Файл рабочего стола) позволяет задать место для хранения параметров рабочего стола (т. е. файла fp.dsk) — в текущем каталоге или в каталоге, где находится файл конфигурации системы fp.cfg.
- В панели **Auto save** (Автосохранение) можно отметить, какие из системных файлов должны запоминаться при выходе из среды — **Editor files** (файлы, вызванные на поле редактора), **Environments** (параметры среды окружения), **Desktop** (параметры рабочего стола).
- Панель **Options** предназначена для фиксации одной из следующих возможностей:
  - **Auto track source** — автоматическая нумерация строк исходной программы;
  - **Close on go to source** — закрывать окно отладки при переходе на исходный код;
  - **Change dir on open** — изменение текущего каталога при открытии файла.

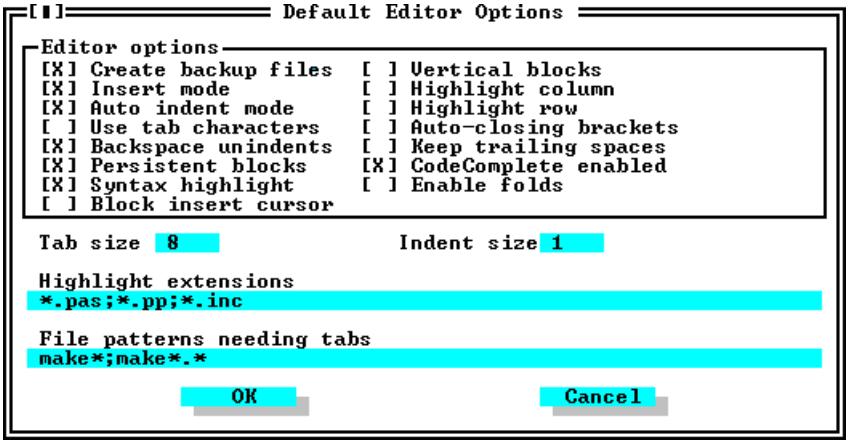


Рис. П2.12. Диалоговое окно
для настройки редактора текстов программ

Замечание

Подкоманда **Preferences** открывает диалоговое окно только в том случае, если в поле редактора IDE открыта какая-либо программа. В противном случае на экране возникнет предупреждающее сообщение об ошибке, предлагающее закрыть среду IDE.

Подкоманда **Editor** открывает диалоговое окно (рис. П2.12), используемое для настройки редактора.

Пояснения, связанные с опциями редактора, приведены в табл. П2.4

Таблица П2.4

Параметр	Пояснение
Create backup files	Создавать резервную копию текущей программы
Insert mode	Режим вставки при наборе текста
Auto indent mode	Режим автоматического отступа
Use tab characters	Использовать символы Tab
Backspace unindents	Отмена отступа клавишей <Backspace>
Persistent blocks	Сохранение выделенности блока при выходе из него
Syntax highlight	Включение подсветки синтаксических конструкций
Block insert cursor	Курсор в виде блока
Vertical blocks	Возможность выделения вертикальных блоков (по столбцам)
Highlight column	Подсветка столбцов
Highlight row	Подсветка строк
Auto-closing brackets	Автоматическое закрытие недостающей скобки
Keep trailing spaces	Сохранение лидирующих пробелов
CodeComplete enabled	Разрешение режима автоматического завершения набора служебных слов
Enable folds	Разрешить режим свертки для ускоренного просмотра программы

Подкоманда **Desktop** открывает диалоговое окно (рис. П2.13), в котором выделяются элементы текущего сеанса, которые могут быть сохранены для последующей работы.

Подкоманда **Keyboard&Mouse** позволяет задать режимы работы кнопок мыши в сочетании с управляющими клавишами (рис. П2.14).

В этом окне вы можете выбрать один из двух стандартов управления операциями **Cut** (Вырезать), **Copy** (Копировать) и **Paste** (Вставить):

- стандарт IBM (CUA — Common User Access);
- стандарт Microsoft.

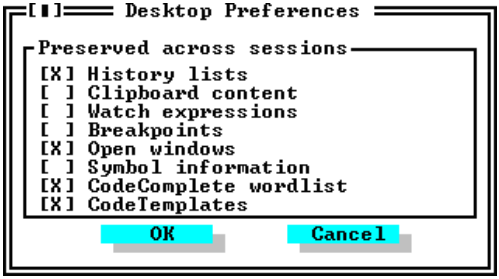


Рис. П2.13. Окно запоминаемых параметров состояния сессии

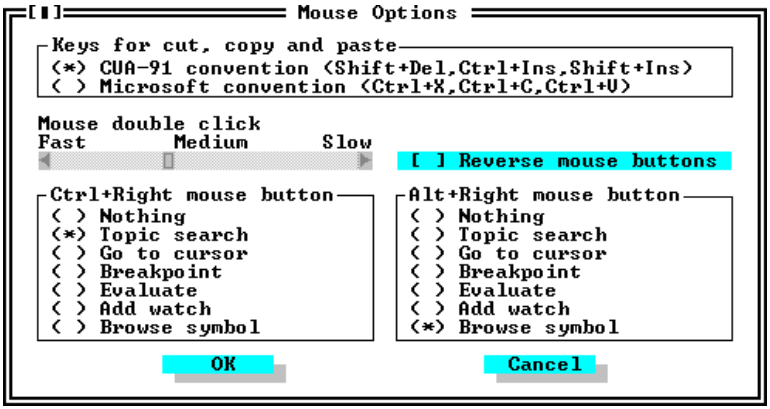


Рис. П2.14. Управление режимом работы мыши и клавишными командами

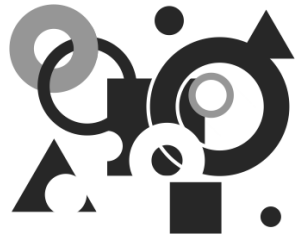
Полоса прокрутки **Mouse double click** используется для регулировки временной границы между двумя последовательными щелчкам, ниже которой регистрируется событие **DoubleClick**.

В панели **Ctrl+Right mouse button** можно зафиксировать одну из операций, выполняемую по одновременному нажатию клавиши <Ctrl> и правой кнопки мыши:

- **Nothing** — ничего не делать;
- **Topic search** — вызвать кадр помощи по служебному слову, на которое нацелен указатель мыши;
- **Go to cursor** — запустить программу с точкой останова на строке, в которой находится указатель мыши;
- **Breakpoint** — выделить в качестве точки останова строку, в которой находится указатель мыши;
- **Evaluate** — вызвать диалоговое окно для вычисления выражения;
- **Add watch** — вызвать диалоговое окно для набора контролируемого выражения;
- **Browse symbol** — вызвать браузер для просмотра информации об объекте, на который нацелен указатель мыши.

Панель **Alt+Right** предназначена для выбора одной из указанных операций по одновременному нажатию клавиши <Alt> и правой кнопки мыши.

Приложение 3



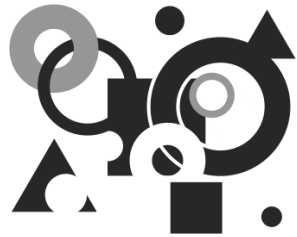
Сообщения об ошибках периода выполнения

Код	Текст сообщения	Пояснение
1	Invalid function number	Неправильный номер функции
2	File not found	Файл не найден
3	Path not found	Ошибка в задании пути
4	Too many open files	Слишком много открытых файлов
5	File access denied	Доступ к файлу запрещен
6	Invalid file handle	Файловая переменная повреждена
12	Invalid file access code	Ошибка в задании режима открытия файла
15	Invalid drive number	Ошибка в задании номера диска
16	Cannot remove current directory	Попытка удалить текущий каталог
17	Cannot rename across drives	Недопустимая попытка переименовать диски
100	Disk read error	Ошибка при чтении с диска (обычно при попытке чтения после `EOF`)
101	Disk write error	Ошибка при записи на диск (обычно при переполнении диска)
102	File not assigned	Вы забыли выполнить оператор `Assign`
103	File not open	Обращение к файлу, который не был открыт
104	File not open for input	Обращение к файлу, который не был открыт для чтения
105	File not open for output	Обращение к файлу, который не был открыт для вывода
106	Invalid numeric format	Из текстового файла считано нечисловое значение
150	Disk is write-protected	Запись на диск запрещена
151	Bad drive request struct length	Ошибка в запросе к диску
152	Drive not ready	Устройство не готово
154	CRC error in data	Ошибка чтения (несовпадение контрольной суммы)

(окончание)

Код	Текст сообщения	Пояснение
156	Disk seek error	Ошибка при поиске записи
157	Unknown media type	Неопознанный носитель информации
158	Sector Not Found	Сектор не найден
159	Printer out of paper	В принтере кончилась бумага
160	Device write fault	Техническая ошибка при записи
161	Device read fault	Техническая ошибка при чтении
162	Hardware failure	Обращение к неподключенному устройству
200	Division by zero	Деление на 0
201	Range check error	Индекс массива или значение переменной вне диапазона
202	Stack overflow error	Переполнение стека
203	Heap overflow error	Переполнение "кучи"
204	Invalid pointer operation	Ошибочная операция с указателем (обычно при освобождении памяти с нулевым значением указателя)
205	Floating point overflow	Переполнение в операции с плавающей запятой
206	Floating point underflow	Исчезновение порядка в операции с плавающей запятой
207	Invalid floating point operation	Ошибка при выполнении операции с плавающей запятой (обычно при задании недопустимого аргумента стандартной функции)
210	Object not initialized	Попытка выполнить операцию над объектом, который не был инициализирован
211	Call to abstract method	Недопустимая попытка обращения к абстрактному методу
212	Stream registration error	При регистрации объекта в модуле встречен недопустимый тип данных
213	Collection index out of range	Индекс коллекции вне диапазона
214	Collection overflow error	Попытка добавить новый элемент к коллекции, заполненной до предельного размера
215	Arithmetic overflow error	Арифметическое переполнение (результат операции выходит за пределы допустимого диапазона)
216	General Protection fault	Попытка обращения к памяти, находящейся за пределами ресурсов, выделенных приложению (в том числе и попытка обращения к данным по нулевому указателю)
217	Unhandled exception occurred	Возникновение непредусмотренного исключительного события
219	Invalid typecast	Недопустимое приведение типа

Приложение 4



Описание компакт-диска

П4.1. Что находится на компакт-диске

Компакт-диск, прилагаемый к книге, содержит следующие папки и файлы:

- каталог Fpc — с рабочей системой программирования Free Pascal;
- каталог FP_Prog — с программами, рассматриваемыми в книге;
- каталог distr — дистрибутив Free Pascal, файлы справочной системы в каталоге html, библиотека GLUT в папке Glut 3.7 beta;
- файл readme.doc — описание компакт-диска.

П4.2. Система программирования FP IDE

Система программирования FP IDE находится на компакт-диске в каталоге Fpc. Он содержит главный подкаталог среды с именем 2.4.0 (так авторы системы идентифицируют свои версии). В главном подкаталоге системы размещены 7 системных подкаталогов:

- bin\i386-win32 включает все файлы FPC и IDE (дополнительный подкаталог i386-win32 символизирует версию, ориентированную на работу с процессорами фирмы Intel под управлением 32-разрядных версий Windows);
- demo, содержащий 16 подкаталогов с наборами проектов демонстрационных программ;
- doc, включающий основные документы по системе программирования Free Pascal в формате PDF;
- examples, содержащий 23 подкаталога с примерами программ;
- html, включающий все страницы файла помощи и индексный файл fpctoc.htx, обеспечивающий доступ к любой странице;
- msg, содержащий набор файлов с сообщениями системы на разных языках. По умолчанию система FP IDE настроена на работу с англоязычным файлом errore.msg, который отражает все нюансы текущей версии. Русскоязычный

файл errorr.msg, по нашим наблюдениям, не вполне синхронизован с версией 2.4.0 и не все его сообщения обеспечивают качественный перевод;

- units\i386-win32 включает набор модулей для работы с внешними библиотеками и формируется за счет многочисленных пользователей.

В отличие от других систем программирования и наиболее востребованных приложений среда FP IDE не пользуется услугами реестра Windows. Для того чтобы установить эту среду на своем компьютере, достаточно скопировать каталог Fpc на диск C: винчестера с компакт-диска, прилагаемый к книге. Если вы скопируете каталог Fpc в другое место, то вам придется исправить некоторые настройки в файлах среды fp.ini, fp.cfg, fpc.cfg. Иначе возможны системные ошибки при компиляции и запуске программ в среде FP IDE.

Если вы желаете установить FP IDE в другое место, вам лучше выполнить установку системы из дистрибутива, скачать который можно с сайта разработчика по адресу: **http://www.freepascal.org/download.var**.

П4.3. Тексты FP-программ

В каталоге FP_Prog на компакт-диске находятся все программы, описанные в книге и упорядоченные по главам. Каталоги первого уровня вложенности обозначены номерами глав. В каталоге FP_Prog\02 находятся все программы *главы 2*, в каталоге FP_Prog\03 — все программы *главы 3* и т. д. Внутри каждого подкаталога главы находятся каталоги второго уровня вложенности с такими же цифровыми обозначениями. Например, в каталоге FP_Prog\02\01 находятся файлы, относящиеся к программе листинга 2.1 из *главы 2*.

Структура большинства подкаталогов программ выдержана по единому принципу:

- файл *xxxx*.pas — исходный текст программы (имя программы совпадает с именем, приведенным в соответствующем листинге книги);
- файл *xxxx*.o — объектный модуль, полученный после компиляции исходного файла. В принципе этот модуль можно было не сохранять, т. к. он является промежуточным результатом трансляции;
- файл *xxxx*.exe — исполняемый модуль, полученный после линковки объектного модуля. Его можно исполнить, чтобы увидеть результат работы той или иной программы в естественных цветах на экране монитора;
- файл с именем *ii_jj*.bmp представляет изображение окна приложения, в котором отражены вводимые данные и результаты работы программы *xxxx*.exe (*ii* — двузначный номер главы, *jj* — двузначный номер программы в главе). Иногда результаты запуска могут не совпадать с приведенным изображением. Как правило, это объясняется использованием случайных данных, показаний компьютерных часов и календаря.

П4.4. Установка и начало работы

П4.4.1. Копирование системы

1. Копирование системы FP IDE на свой компьютер сводится к переписи всех файлов каталога Fpc на диск C:. На каталог C:\Fpc рассчитаны текущие настройки, установленные в некоторых файлах системы. Время копирования может занять несколько минут, т. к. файлы помощи содержат порядка 10 000 очень мелких файлов с расширениями html.
2. После копирования системы на рабочем столе полезно создать ярлык для запуска стартовой программы C:\Fpc\2.4.0\bin\i386-win32\fp.exe (рис. П4.1). Не забудьте изменить подпись к ярлыку и поменять значок на C:\FPC\2.4.0\bin\i386-win32\fp32.ico.

Рис. П4.1. Ярлык FPC IDE

3. Каталог FP_Prog тоже лучше переписать на диск C: (т. к. в файле C:\FPC\2.4.0\bin\i386-win32\fp32.ini прописан рабочий каталог в секции `[Run]`). С текстами этих программ полезно повозиться по двум соображениям. Во-первых, их запуск и совпадение результатов с содержимым bmp-файлов убедит вас в работоспособности результатов трансляции. Во-вторых, внося изменения и дополнения в приведенные тексты, вы научитесь правильно оформлять свои приложения, компилировать их в среде, отлаживать и выполнять. Приучите себя к размещению каждого нового приложения в отдельном каталоге.
4. Для начала сеанса достаточно дважды щелкнуть по созданному на рабочем столе значку, после чего на экране появится окно среды Free Pascal IDE (рис. П4.2).
5. Настройка на каталог с текстом исходной программы может быть выполнена одним из следующих способов.
   - В меню **File** выполняем команду **Change dir**. Если каталог приложения в среде FP IDE набирается впервые, то его проще набрать в поле ввода **Directory name** и нажать кнопку **Ok**. При этом каталог должен быть заранее создан. В противном случае будет автоматически восстановлен каталог по умолчанию, который вы видите в окне **Change Directory** (рис. П4.3).
   - Система запоминает однажды набранное имя каталога, и его можно достать из выпадающего списка. Для этого необходимо щелкнуть по значку "v", имитирующему стрелку вниз, справа от поля **Directory name**. В появившемся списке достаточно совершить двойной щелчок на нужной строке (рис. П4.4).

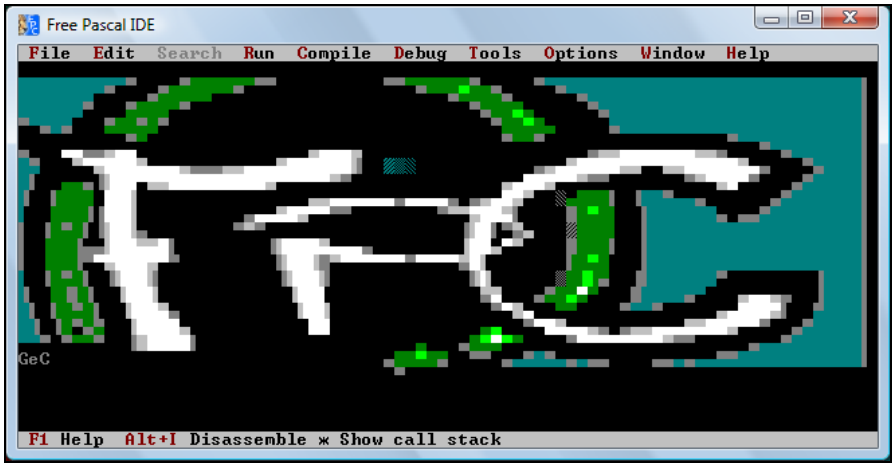


Рис. П4.2. Окно среды FP IDE после старта

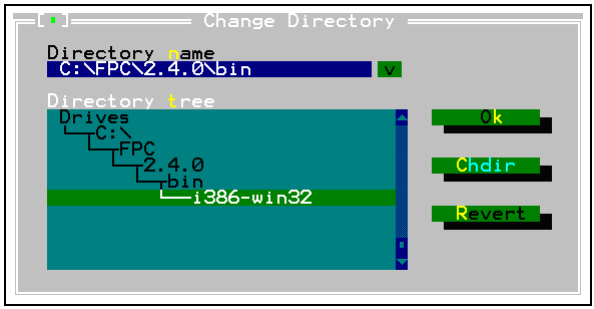


Рис. П4.3. Дерево каталога исходных программ по умолчанию

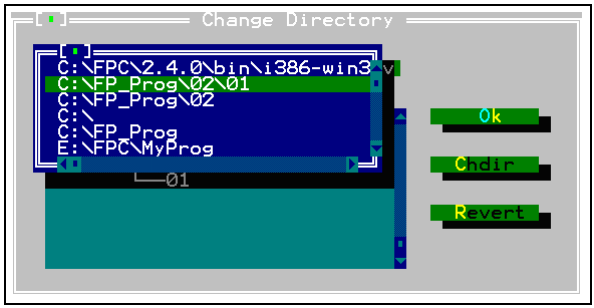


Рис. П4.4. Выпадающий список с именами каталогов

- Наконец, с помощью мыши мы можем совершить путешествие по дереву каталогов. Двойной щелчок по вершине дерева (**Drives**) откроет имена всех логических дисков, существующих на вашем компьютере (рис. П4.5).

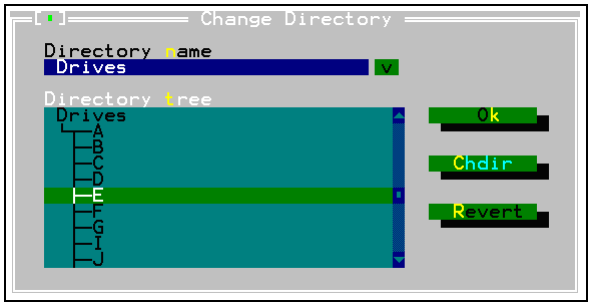


Рис. П4.5. Дерево логических дисков

- Двойной щелчок по имени логического диска откроет дерево каталогов, созданных на этом диске. Продолжим это путешествие до нужного каталога и зафиксируем его в качестве каталога с текстом исходной программы. Предположим, что наше приложение находится в каталоге C:\FP_Prog\02\01 (рис. П4.6).

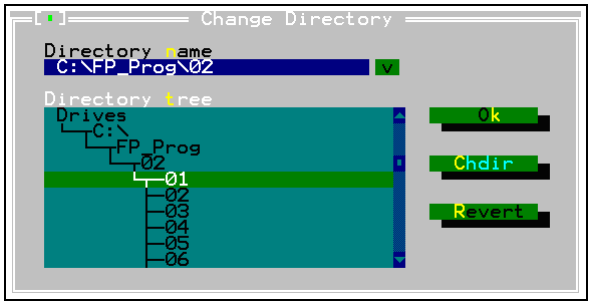


Рис. П4.6. Путь к каталогам программ *главы 2*

- Двойной щелчок по нужному наименованию подкаталога завершает наше путешествие (рис. П4.7). Остается выполнить пару щелчков по кнопкам **Chdir** и **Ok**.

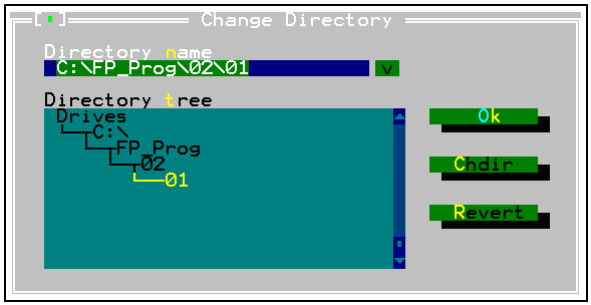


Рис. П4.7. Путь к каталогу первой программы *главы 2*

6. Для того чтобы в каталог с текстом исходной программы попали результаты трансляции и сборки, необходимо проделать аналогичную манипуляцию с командой **Run Directory** в меню **Run**.
7. Когда оба каталога настроены, команда **Open** из меню **File** открывает окно с текстами исходных программ в каталоге пользователя (рис. П4.8).

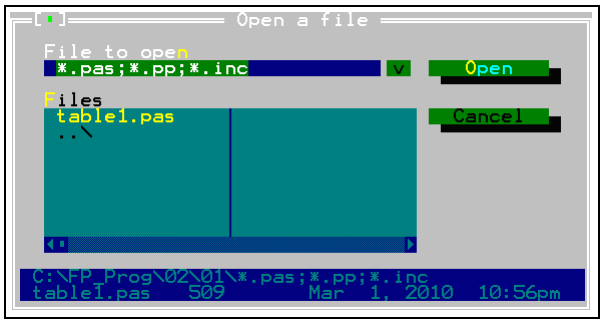


Рис. П4.8. Исходный файл первой программы

П4.4.2. Установка системы из дистрибутива

1. Если вы хотите установить систему Free Pascal самостоятельно (для среды Windows), то запустите файл fpc-2.4.0.i386-win32.exe из каталога distr на компакт-диске.
2. Установка системы стандартная и никаких сложностей у вас не вызовет:
   - Первое окно программы установки — информационное. Нажмите в нем кнопку **Next**.
   - В следующем окне задайте каталог, в который вы хотите установить систему (например, C:\FreePascal\2.4.0). При необходимости воспользуйтесь кнопкой **Browse**. Нажмите кнопку **Next**.
   - Далее выберите тип установки: **Full installation** (Полная), **Minimum installation** (Минимальная), **Custom installation** (Выборочная). Нажмите кнопку **Next**.
   - Затем задайте имя папки, которое появится в главном меню Windows. (Рекомендуется оставить имя, предлагаемое программой установки.) Нажмите кнопку **Next**.
   - В следующем окне отметьте (или оставьте так, как есть) флажки, которые ассоциируют расширения файлов с системой Free Pascal. Нажмите кнопку **Next**.
   - В последнем окне проверьте заданные вами установки и нажмите кнопку **Install**.
3. Установка займет немного времени. Откроется информационное окно с описанием версий Free Pascal. Нажмите кнопку **Next**, а затем — кнопку **Finish**.

4. В главном меню Windows появится группа программ для системы Free Pascal. Не забудьте создать ярлык на рабочем столе *(см. разд. П4.4.1)*.
5. Теперь следует подключить справочную систему (если она вам необходима). Для этого в каталог установки Free Pascal (в нашем случае в C:\FreePascal\2.4.0) скопируйте с компакт-диска папку distr\html.
6. Запустите Free Pascal IDE. Выберите команду **Help → Files**. В диалоговом окне **Install Help Files** нажмите кнопку **New**. В появившемся окне перейдите в каталог C:\FreePascal\2.4.0\html и убедитесь, что в поле **Help file name** указан файл fpctoc.html (рис. П4.9). Нажмите кнопку **Open**.

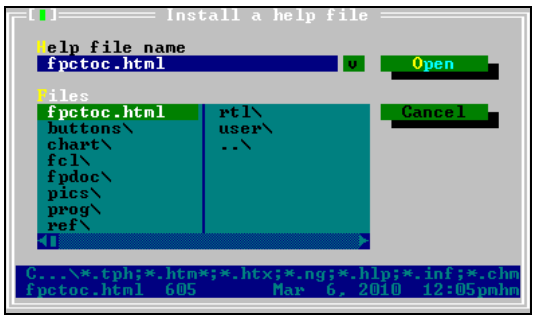


Рис. П4.9. Окно выбора справочного файла

7. На появившийся запрос "Create keyword index from help file?" ("Создать индекс ключевых слов из help-файла?") ответьте утвердительно, нажав кнопку **Yes**. Начнется построение индекса, которое может выполняться очень долго. Наблюдать за этим процессом можно в командной строке среды IDE (прокрутите вниз окно, если строка вам не видна).
8. Когда процесс создания индексного файла будет завершен, в окне **Install Help Files** появится строка **HTML Index** (рис. П4.10). Нажмите кнопку **OK**.

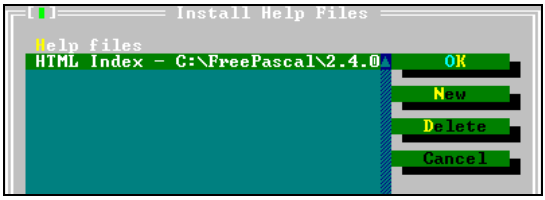


Рис. П4.10. Окно **Install Help Files**

9. Теперь, выбрав команды **Help → Contents** и **Help → Index**, вы можете воспользоваться справочной системой. А в каталоге C:\FreePascal\2.4.0\html будет создан файл fpctoc.htx.

Замечание

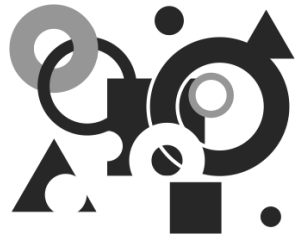
Удалять установленную таким образом систему Free Pascal следует через Панель управления.

П4.4.3. Библиотеки GLU и GLUT

Для работы с программами из *главы 16* вам потребуются библиотеки GLU и GLUT. Скачать последнюю версию можно с сайта OpenGL.com по адресу **http://www.opengl.org/resources/libraries/glut/glut_downloads.php**. На момент написания книги последней доступной версией была 3.7beta. Эта версия также находится на компакт-диске в каталоге \distr\Glut 3.7 beta[1].

Скопируйте файлы glut.dll и glut32.dll в системный каталог Windows — Windows\System32.

[1] GLUT is distributed in source code form; compiled libraries for Win32 are also available. The current version, 3.7, is in late beta. The programs and associated files contained in the distrbution were developed by Mark J. Kilgard (unless otherwise noted). The programs are not in the public domain, but they are freely distributable without licensing fees. These programs are provided without gurantee or warrantee expressed or implied. (GLUT распространяется в виде исходного кода; имеются также скомпилированные библиотеки для Win32. Текущая версия — 3.7 beta. Программы и связанные файлы, содержащиеся в комплекте, были разработаны Марком Дж. Килгардом (если не указано иное). Программы не находятся в общественном достоянии, но они могут свободно распространяться без лицензионных платежей. Эти программы предоставляются без гарантии или гарантий явно выраженных или подразумеваемых.)



Литература

Паскаль, Turbo Pascal

1. Абрамов В. Г., Трифонов Н. П., Трифонова Г. Н. Введение в язык Паскаль. — М.: Наука, 1988. — 320 с.
2. Абрамов С. А., Зима В. С. Начала программирования на языке Паскаль. — М.: Наука, 1987. — 112 с.
3. Вирт Н. Алгоритмы + структуры данных = программы: Пер. с англ. — М.: Мир, 1985. — 406 с.
4. Грогоно П. Программирование на языке Паскаль. — М.: Мир, 1982. — 382 с.
5. Джонс Ж., Харроу К. Решение задач в системе Турбо Паскаль: Пер. с англ. — М.: Финансы и статистика, 1991. — 720 с.
6. Епанешников А. М., Епанешников В. А. Turbo Pascal 7.0. — М.: ДИАЛОГ-МИФИ, 1998. — 288 с.
7. Зуев Е. А. Turbo Pascal: Практическое руководство. — М.: Стрикс, Приор, 1997. — 336 с.
8. Кетков Ю. Л., Кетков А. Ю. Практика программирования: Бейсик, Си, Паскаль. Самоучитель. — СПб.: БХВ-Петербург, 2001. — 480 с.
9. Культин Н. Б. Turbo Pascal в задачах и примерах. — СПб.: БХВ-Петербург, 2000. — 256 с
10. Немнюгин С. А. Turbo Pascal. — СПб.: Питер, 2000. — 496 с.
11. Новичков В. С. и др. Паскаль: Учебное пособие для средних специальных заведений. — М.: Высшая школа, 1990. — 223 с.
12. Йенсен К., Вирт Н. Паскаль: Руководство для пользователя: Пер. с англ. — М.: Финансы и статистика, 1989. — 255 с.
13. Перминов О. Н. Язык программирования Паскаль. — М.: Радио и связь, 1983. — 119 с.
14. Сергиевский М. В., Шалашов А. В. Турбо Паскаль 7.0: Язык, среда и программирование. — М.: Машиностроение, 1994. — 254 с.
15. Семашко Г. Л., Салтыков А. И. Программирование на языке Паскаль. — М.: Наука, 1988. — 128 с.
16. Уилсон И. Р., Эддиман А. М. Практическое введение в Паскаль: Пер. с англ. — М.: Радио и связь, 1983. — 144 с.
17. Фаронов В. В. Turbo Pascal 7.0. Начальный курс. Учебное пособие. — М.: Нолидж, 1997. — 616 с.
18. Фаронов В. В. Turbo Pascal 7.0. Практика программирования: Учебное пособие. — М.: Нолидж, 1997. — 432 с.

Free Pascal, Object Pascal

19. Грудзинский А. О., Мееров И. Б., Сысоев А. В. Методы программирования. Курс на основе языка Object Pascal. — Н. Новгород: изд. ННГУ, 2006. — 392 с.
20. Кетков Ю. Л., Кетков А. Ю. Практика программирования: Visual Basic, C++Builder, Delphi. Самоучитель. — СПб.: БХВ-Петербург, 2002. — 464 с.
21. Культин Н. Б. Delphi 6. Программирование на Object Pascal. — СПб.: БХВ-Петербург, 2001. — 528 с.
22. Сухарев М. В. Основы Delphi. Профессиональный подход. — СПб.: Наука и Техника, 2004. — 600 с.
23. Canneyt M. Reference guide for FCL units. 2009. — 367 p. [Электронный ресурс]. **http://www.freepascal.org**.
24. Canneyt M. Programmer's Guide for Free Pascal. 2009. — 172 p. [Электронный ресурс]. **http://www.freepascal.org**.
25. Canneyt M. Reference guide for Free Pascal. 2009. — 162 p. [Электронный ресурс]. **http://www.freepascal.org**.
26. Canneyt M. Free Pascal. Reference guide for RTL units. 2009. — 1453 p. [Электронный ресурс]. **http://www.freepascal.org**.
27. Canneyt M. User's Guide for Free Pascal, 2009. — 184 p. [Электронный ресурс]. **http://www.freepascal.org**.

Графика

28. Баяковский Ю. М., Игнатенко А. В., Фролов А. И. Графическая библиотека OpenGL. Учебно-методическое пособие. — М.: Изд. отдел факультета ВМК МГУ, 2003. — 132 с.
29. Белецкий Я. Турбо Паскаль с графикой для персональных компьютеров. — М.: Машиностроение, 1991. — 320с.
30. Грызлов В. И., Грызлова Т. П. Турбо Паскаль 7.0. — М.: ДМК, 1998. — 400 с.
31. Краснов М. В. OpenGL. Графика в проектах Delphi. — СПб.: БХВ-Петербург, 2001. — 352 с.
32. Порев В. Н. Компьютерная графика. — СПб.: БХВ-Петербург, 2002. — 432 с.
33. Прокофьев Б. П., Сухарев Н. Н., Храмов Ю. Е. Графические средства Turbo C и Turbo C++. — М.: Финансы и статистика, 1992. — 160 с.
34. Тихомиров Ю. Программирование трехмерной графики. — СПб.: БХВ-Петербург, 1998. — 256 с.
35. Хилл Ф. OpenGL. Программирование компьютерной графики. — СПб.: БХВ-Петербург, 2002. — 1082 с.
36. Эйнджел Э. Интерактивная компьютерная графика. Вводный курс на базе OpenGL, 2-е изд. — М.: Изд. дом "Вильямс", 2001. — 592 с.
37. Bresenham J. E. Algorithm for computer control of a digital plotter// IBM System Journal, vol 4, № 1, 1965. — pp. 25—30.
38. Bresenham J. E. A linear algorithm for incremental digital display of circular arc// Communications of the ACM, vol 20, № 2, 1977. — pp. 100—106.
39. OpenGL. Red Book (русская версия). [Электронный ресурс]. **http://www.progz.ru**.

Предметный указатель

И

К

М

Н

О

П

P

C

T

У

Ф

Х

Ц

Ч

Ш

Э