МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

САМАРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра информатики и вычислительной математики

Е.В. Рогачева

Основы программирования в среде Delphi

Часть 1

Учебное пособие

Издательство «Самарский университет»

2002

Печатается по решению Редакционно-издательского совета Самарского государственного университета

УДК 681.142.2
ББК 32.973–01
Р 592

Рогачева Е.В. Основы программирования в среде Delphi. Часть I. Самара: Изд-во «Самарский университет», 2002. – 128 с.

В предлагаемом учебном пособии описаны основные понятия языка программирования Object Pascal и среды Delphi, базирующейся на этом языке. Изложение материала сопровождается примерами программных проектов. Пособие также содержит наборы индивидуальных заданий по каждой изучаемой теме. Предполагается, что обучающиеся знакомы с операционной системой Windows и основными приемами работы в ней.

Для студентов механико-математического факультета, а также для студентов естественнонаучных специальностей, изучающих программирование.

УДК 681.142.2
ББК 32.973–01

Рецензент д-р физ.-мат. наук, проф. И.П.Завершинский

Содержание

Введение

Визуальная разработка Windows-приложений. Особенности Delphi. Различия между Windows и DOS. Ориентация Windows-программ на события.

Визуальная разработка Windows-приложений

В настоящее время одной из самых распространенных операционных систем для персональных компьютеров является Windows. Свою популярность она завоевала благодаря понятному и удобному графическому интерфейсу. По большому счету, именно Windows способствовала столь широкому распространению компьютеров: работа на уровне пользователя стала требовать намного меньшей подготовки. Вместе с тем написание Windows-приложения представляет собой более сложную задачу, чем разработка программы для DOS. Необходимость детально разбираться в тонкостях механизмов операционной системы превращала программирование под Windows в искусство, доступное немногим. Нужен был инструмент, который позволил бы упростить, ускорить и, по возможности, автоматизировать процесс разработки приложений. Первый шаг в этом направлении был сделан создателями Windows – фирмой Microsoft, выпустившими программный продукт Visual Basic. В нем была реализована идея визуальной разработки интерфейса с помощью повторно используемых компонентов (кнопок, меню и т.п.). Суть идеи состояла в том, чтобы нужный компонент можно было просто выбрать из готового набора (щелчком на соответствующей пиктограмме), поместить его в разрабатываемое окно и, при необходимости, изменить некоторые его характеристики (надпись на кнопке, например), а среда разработки при этом автоматически генерировала бы часть кода (текста программы), связанного с этим компонентом. Таким образом, проектирование интерфейса, которое занимало значительную часть времени при создании Windows-приложений, становилось делом куда менее трудоемким.

Следом за Visual Basic появилась разработанная фирмой Borland (ныне – Inprise) среда программирования Delphi. Мы не будем подробно обсуждать достоинства и недостатки каждого из этих продуктов (тем более что с момента появления их первых версий они претерпели значительные изменения), однако приведем несколько аргументов в пользу Delphi. Язык Basic является интерпретируемым языком, в то время как Pascal (составляющий основу Delphi) – язык компилируемый. Это обеспечивает приложениям, разработанным с помощью Delphi, большую производительность. Кроме того, Pascal более мощный язык, чем Basic, и в полной мере позво-

ляет разработчику использовать арсенал объектно-ориентированного программирования. Наконец, Delphi предоставляет широкие возможности доступа к базам данных.

Сейчас, когда со времени выхода первых версий Visual Basic и Delphi прошло уже более пяти лет, выпускается ряд программных продуктов, позволяющих разрабатывать Windows-приложения визуально. Отметим, в частности, целую серию Visual-продуктов Microsoft (наряду с Visual Basic в их числе Visual C++, Visual J++, Visual Fox Pro) и такие продукты фирмы Inprise, как C++ Builder (базовый язык C++) и JBuilder (базовый язык Java).

Особенности Delphi

Среда Delphi является визуальным инструментом разработки приложений Windows. Delphi в значительной степени избавляет программиста от рутинной работы, связанной с определением поведения стандартных элементов интерфейса, таких как кнопки или меню, и дает возможность сосредоточиться на алгоритме. При создании небольшого приложения достаточно потратить несколько минут, чтобы спроектировать пользовательский интерфейс. Впрочем, при необходимости можно создать и консольное (выполняющееся в DOS-окне) приложение. Использование принципов объектно-ориентированного программирования, реализованных в языке Object Pascal (иногда в литературе – Delphi Pascal) позволяет писать изящный, четко структурированный (и потому прозрачный для понимания) код. Мощный высокопроизводительный компилятор Delphi генерирует компактные и легко переносимые приложения.

Кроме того, Delphi является удобным средством разработки приложений баз данных, в том числе с архитектурой клиент – сервер, совместимым со многими стандартами (ODBC, IDAPI и т.д.). Причем для создания простейшего приложения, выполняющего стандартные операции над базой данных (навигацию, редактирование, вставку и удаление записей), не требуется писать ни одной строки кода (с Delphi поставляется пример такой программы).

Delphi также предоставляет развитые средства для работы с Internet и мультимедиа. Наконец, к существующим компонентам Delphi можно добавить собственные компоненты (написав их с помощью Delphi).

Возможности Delphi велики, и не исключено, что Вы не будете пользоваться какими-то из них вовсе, а о некоторых не будете даже знать. Тем не менее, изучить основы Delphi под силу даже новичку в программировании.

Разумеется, с момента выпуска первой версии Delphi в 1995 г. (она работала под Windows 3.1) и до настоящего момента (Delphi 7 выпущена летом 2002 г.) среда менялась. Появились новые компоненты, хотя «основной состав» остался прежним. Были добавлены новые и усовершенст-

вованы уже имеющиеся инструменты (особенно это касается интерфейса работы с базами данных). Среда стала более дружественной к пользователю и теперь не только пишет «свою» часть кода, но и стремится подсказать программисту, какое свойство или тип данных ему следует использовать. Вместе с тем основные принципы создания программ остались прежними, и не так уж важно, к какой из версий Delphi Вы сейчас имеете доступ. Научившись работать в какой-либо из них, Вы легко разберетесь в любой другой с помощью подробной и хорошо организованной справочной системы.

Различия между Windows и DOS

Если Вы имеете опыт программирования для DOS, возможно, Вам потребуется некоторое время, чтобы привыкнуть к некоторым особенностям программирования для Windows. Эти особенности, разумеется, связаны с тем, как устроена Windows. Мы схематично опишем принципы работы программ под управлением той или иной операционной системы. Можете пропустить этот параграф и перейти сразу к следующему, если достаточно знакомы с этим вопросом. Цель нескольких следующих абзацев – убедить читателя, что Windows отличается от DOS не только графическим интерфейсом пользователя, делающим ее более привлекательной и понятной для человека, не имеющего большого опыта работы с компьютером. Windows – это куда более мощная операционная система, нежели DOS.

Операционная система DOS является однозадачной. Это значит, что при работе DOS-приложение получает в свое распоряжение практически все ресурсы системы – оперативную память, время процессора, и никакие другие программы не смогут использовать эти ресурсы до тех пор, пока работающая программа не завершится.

В Windows 3.1 за управление ресурсами отвечает операционная система. Распределение памяти выглядит примерно так: все работающие в данный момент программы обращаются к одной и той же куче байтов, помечая тот или иной байт как свой. Пока все они работают корректно, Windows управляется с ними весьма успешно. Однако в Windows 3.1 реализована так называемая невытесняющая (коллективная) многозадачность. Это означает, что операционная система определяет только порядок, в котором работающие программы получают время процессора. А время, которое процессор будет занят некоторой программой, определяется самой программой. Поэтому сбой, случившийся при работе одной из программ, угрожает стабильной работе остальных, а главное – может привести к нарушению работы операционной системы.

И DOS, и Windows 3.1 являются 16-разрядными операционными системами. Когда системе нужно сослаться на конкретную ячейку памяти, к которой обращается программа, она может использовать для указания

адреса этой ячейки не более чем 16-разрядные двоичные числа. Самое большое десятичное число, которое может быть записано с помощью 16 нулей и единиц – это 65535, известное так же, как 64 Кб. В этих 64 Кб и должна была оперировать программа DOS. А если, к примеру, она должна была работать с достаточно большим массивом, добиваться этого приходилось с помощью различных ухищрений. Ограничение в 64 Кб было связано с тем, что процессоры фирмы Intel до версии 286 включительно могли одновременно оперировать не более чем с 16 битами информации. Использование так называемого защищенного режима в Windows 3.1 позволило работать с 24-разрядными числами и адресовать до 16 Мб памяти. Но уже 386-й процессор являлся 32-разрядным и был готов работать с 4 Гб памяти. Что же говорить о Pentium IV?

В Win32 – операционных системах (так мы будем называть Windows'95, Windows NT и более старшие версии 32-разрядных операционных систем Windows) реализована многозадачность с вытеснением. Каждая программа получает время процессора строго по графику – вне зависимости от того, что происходит с другими программами. Обычно говорят, что каждая программа имеет свой поток, а каждому потоку операционная система назначает квант (интервал) времени. Когда программа запускается на выполнение, операционная система создает для нее отдельное – виртуальное – адресное пространство (разумеется, каждому виртуальному адресу должен быть сопоставлен адрес реально существующей ячейки памяти, и эту задачу также решает операционная система).

Нужно отметить, что вышесказанное относится только к 32-разрядным программам. Когда несколько 16-разрядных программ выполняются под управлением Win32, они разделяют между собой одно и то же виртуальное адресное пространство, и некорректное поведение одной из них по-прежнему может нарушить работу остальных – но только 16-разрядных – программ.

Ориентация Windows-программ на события

При работе с Windows-программой Вы не можете не заметить, что программа всегда ожидает от пользователя (в данном случае, от Вас) каких-то действий. Если Вы, к примеру, напечатали столбец цифр в таблице Excel, то не стоит ожидать, что программа самостоятельно вычислит их среднее значение. Для этого придется проделать некоторые действия: вызвать функцию вычисления среднего значения и указать параметры в диалоговом окне. И лишь после нажатия кнопки «ОК» Excel займется подсчетом. Другой пример: чтобы нарисовать картинку во встроенном графическом редакторе, поставляемом вместе с MS Windows, следует перемещать нужным образом мышку, удерживая нажатой левую кнопку. Наконец, для завершения работы любого активного Windows-приложения необходимо

одновременно нажать клавиши <Alt> и <F4>. Действия пользователя или системы во время выполнения приложения называют событиями.

Все Windows-приложения способны реагировать на события. Когда пользователь нажимает на клавишу или перемещает мышь, операционная система распознает это и посылает приложению сообщение о действиях пользователя. Если приложение предусматривает отклик (реакцию) на произошедшее событие, то будут выполнены какие-либо действия. В противном случае сообщение игнорируется приложением, и пользователь ничего не заметит. Можно сколько угодно перемещать мышь, не нажимая ее клавиш, над окном графического редактора – ничего не произойдет. В то же время подобное перемещение мыши над раскрытым меню любого приложения приводит к выделению того или иного пункта.

Приложения DOS действуют совершенно иначе. Когда такая программа запущена на выполнение, она сразу же начинает что-нибудь делать (например, предлагает пользователю ввести какие-то данные). Чтобы узнать, не нажал ли пользователь какую-либо клавишу, не передвинул ли он мышь, программе приходится самой спрашивать об этом операционную систему, вызывая соответствующие встроенные процедуры.

Итак, программа, работающая под управлением DOS, должна самостоятельно запрашивать операционную систему о каких-либо конкретных действиях пользователя (принято говорить о процедурной модели DOS). Программа, работающая под управлением Windows, ожидает сообщений от операционной системы и имеет возможность либо отреагировать на сообщение, либо проигнорировать его (принято говорить об ориентированной на события или же основанной на сообщениях модели Windows).

Глава 1. Delphi. Первое знакомство

Интегрированная среда разработки Delphi. Практикум. Анатомия проекта. Файлы форм. Классификация лексем. Подробно о структуре проектов и модулей.

Интегрированная среда разработки Delphi

Когда Вы запустите Delphi – двойным щелчком на соответствующем значке или с помощью меню «Пуск» – Вы увидите на экране монитора несколько окон (Рис.1.1). Вверху располагается главное окно Delphi, содержащее главное меню, панель инструментов (почти два десятка так называемых «быстрых» кнопок с пиктограммами, соответствующих основным пунктам главного меню) и Палитру Компонентов. Главное меню содержит множество команд: как стандартных для Windows-приложений (открытия и сохранения файлов, редактирования, поиска), так и специфичных для Delphi. В дальнейшем мы хотя бы несколько строчек посвятим описанию каждого из пунктов меню, но будем делать это по мере надобности. Конечно, если Вы программировали на Pascal или C и пользовались при этом инструментами фирмы Borland, назначение многих пунктов меню окажется для Вас очевидным.

Как обычно, можно управлять отображением и расположением «быстрых» кнопок. Более того, с помощью мыши можно перемещать любую инструментальную панель в любое удобное с точки зрения пользователя место экрана. В последних версиях Delphi реализована технология плавающих окон (dockable windows, буквально – способных пристыковываться), которая позволяет пользователю соединять и размещать по своему усмотрению как существующие панели инструментов и окна Delphi, так и созданные им панели инструментов. Кроме того, она может быть легко использована в приложениях.

✓ ***Примечание.*** *Плавающим окном является, например, панель Microsoft Office. Ее можно расположить в любом месте экрана монитора или же «закрепить» у какого-либо края экрана.*

Палитра Компонентов состоит из нескольких страниц и содержит основные «строительные блоки» Delphi – компоненты, сгруппированные в соответствии со своим назначением. В дальнейшем, описывая компоненты, мы будем указывать их стандартное размещение. Отметим, что пользователь может добавлять на любую из страниц новые компоненты и удалять

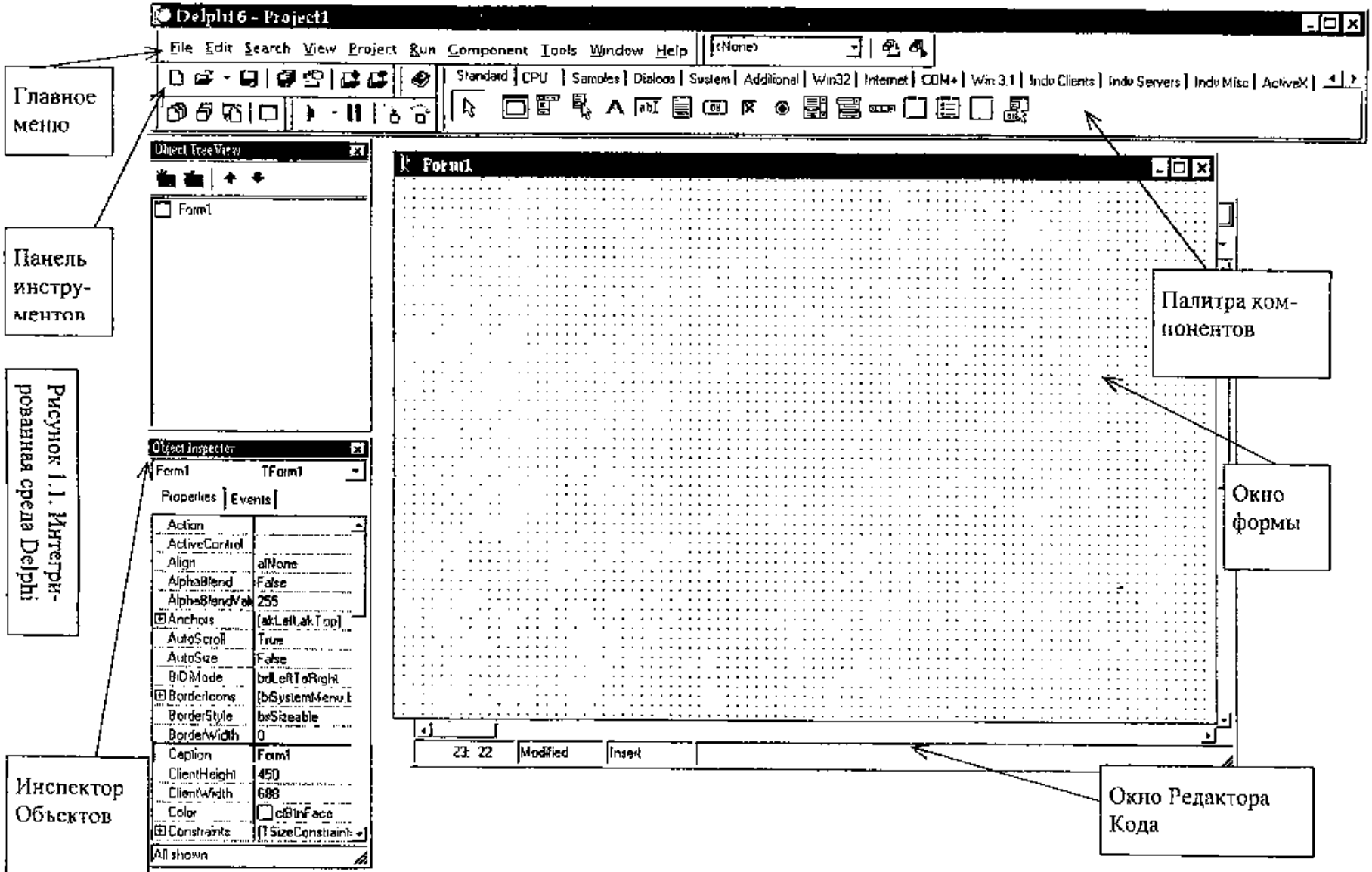

Рисунок 1.1. Интегрированная среда Delphi

уже размещенные и, более того, может создавать и удалять целые страницы Палитры.

Пока можно представлять себе компоненты как заготовки элементов управления. Это, конечно, не слишком корректное утверждение, но оно хорошо иллюстрирует суть дела. Например, на странице Standard располагаются такие компоненты, как Button (кнопка), MainMenu (главное меню), Edit (элемент редактирования). Эти элементы управления знакомы любому пользователю Windows.

✓ ***Примечание.*** *Если подвести к пиктограмме на Палитре Компонентов указатель мыши и задержать его на одну-две секунды, то чуть ниже пиктограммы появится маленькое окошко (hint) с названием компонента. Так, если остановить мышку над компонентом метки (большая буква A), Вы увидите маленький прямоугольник с надписью «Label».*

Каждый компонент обладает определенным (фиксированным для данного компонента) набором присущих ему свойств – разнородных характеристик, которые чаще всего определяют внешний вид компонента и его связи с другими компонентами. Кроме набора свойств у компонента существует также набор событий, на которые он способен реагировать. Так, например, компонент Button – это кнопка стандартного размера (в Delphi 6.0 – 25×75 пикселей), которая готова отреагировать более чем на полтора десятка событий (в том числе нажатие кнопки, перемещение над ней мыши, потерю фокуса), но еще «не знает» как. Задача программиста и заключается в том, чтобы добавить необходимую специфику каждому используемому им компоненту.

Главное окно всегда будет находиться на экране, когда Вы работаете в Delphi: его минимизация или закрытие приведут, соответственно, к минимизации или закрытию всех окон Delphi. Остальные окна независимы друг от друга.

Слева под главным окном располагается окно Инспектора Объектов (Object Inspector). Инспектор Объектов имеет две страницы – свойств (Properties) и событий (Events). Этот инструмент позволяет получить доступ ко всем использованным компонентам. На странице свойств определяются различные характеристики компонентов, а на странице событий – действия, которые этот компонент должен выполнять, когда происходит то или иное событие. Обе страницы разделены на две части: слева расположены названия свойств или событий, а справа – значения свойств или названия обработчиков событий. Правая часть страницы событий изначально всегда пуста.

✓ ***Примечание.*** *Обработчиком события (event handler) называют специальную процедуру (часть программного кода), которая описывает реакцию компонента на произошедшее событие (и будет выполняться при наступлении этого события).*

Справа от Инспектора Объектов находится Проектировщик Форм (Form Designer), который мы ради простоты будем часто называть окном формы. Форма – это особый компонент, реализующий стандартные возможности окна Windows (по сути, заготовка окна). Именно в форме Вы будете размещать компоненты, выбранные из Палитры Компонентов. Следует учитывать, что при работе приложения Вы увидите на экране примерно то же самое, что и во время разработки: Delphi действует по принципу WYSIWYG (What You See Is What You Get – Что Вы Видите, То Вы и Получаете). На рисунке – только одна форма, но в сложном приложении может быть несколько (и даже несколько десятков) форм. Для добавления новой формы следует воспользоваться соответствующим пунктом главного меню: Палитра Компонентов не предоставляет такой возможности.

Сразу после запуска Delphi окно Редактора Кода (Code Editor) обычно скрывается за окном формы (на рисунке эти окна расположены так, чтобы их было видно одновременно). Окно Редактора Кода на рисунке содержит только одну страницу с названием Unit1. Страниц может быть много, причем действует правило: каждой форме соответствует страница, которая автоматически создается Delphi при добавлении этой формы. Такая страница уже содержит определенный код (текст программы), подобный тому, который можно видеть в Unit1. Могут существовать страницы, не связанные с формами, но не существует форм, не связанных со страницами. Если закрыть страницу, соответствующую некоторой форме, эта форма также исчезнет с экрана (обратное неверно).

В окне Редактора Кода Вы будете писать Вашу программу, точнее, ту ее часть, которую не сможет написать Delphi. Разумеется, реализация, например, численного метода решения уравнения останется за программистом, хотя и в этом случае некоторые «приятные мелочи» способны облегчить работу. А вот при разработке интерфейса приложения Delphi автоматически генерирует значительную часть кода. Так, при помещении на форму какого-либо компонента из Палитры Компонентов его описание сразу же появляется в окне Редактора Кода.

В последних версиях Delphi (начиная с 4) слева к окну Редактора Кода обычно пристыковывается окно Проводника Кода (Code Explorer). Это окно содержит информацию о типах, переменных, константах, процедурах и функциях, модулях, используемых приложением. Проводник Кода также позволяет быстро осуществлять навигацию по коду.

✓ ***Примечание.*** *Кроме описанных выше окон, на рабочем столе Windows могут присутствовать и другие, например, Менеджер Проектов (Project Manager) или окна отладки (Debug Windows). Чтобы открыть их, следует воспользоваться соответствующими командами меню. Подробное описание работы с этими и другими окнами будет дано в главе «Интегрированная среда разработки».*

Практикум

Пример 1.1. Ваш первый проект

В Delphi принято использовать термин «проект» (Project) для обозначения разрабатываемого приложения. Проект состоит из файлов модулей (unit source files), каждый из которых должен иметь имя, отличное от имени проекта. Delphi обеспечивает автоматическое именование проектов и файлов, их составляющих, но лучше, если Вы сами будете давать информативные имена Вашим файлам и проектам. С точки зрения автора удобно разработать собственную систему именования (хотя бы в рамках одного проекта), а каждый проект размещать в отдельном каталоге.

Наступило время перейти от слов к делу и создать Ваш первый проект. По традиции это будет приветствие миру: «Привет, мир!» (или «Hello, world!» – как Вам больше нравится). С помощью Delphi можно создать множество вариантов приветствия, и два из них будут реализованы.

В первой версии программы-приветствия не потребуется писать никакого кода. Начните новый проект, выбрав в меню «File» пункт «New Application» (в дальнейшем мы будем использовать для обозначения пунктов главного меню сокращенную запись, например: «File» | «New Application»). Форма, которую Вы увидите на экране, имеет заголовок Form1. Мы заменим стандартный заголовок формы строкой приветствия. Щелкните мышкой по заголовку Инспектора Объектов, чтобы активизировать его. Инспектор Объектов сейчас отображает свойства формы (других объектов у нас пока просто нет). Выберите на странице «Property» свойство Caption (скорее всего, эта строчка сразу окажется подсвеченной) и вместо строки «Form1» наберите:

```
Привет, мир!
```

Текст заголовка формы сразу же изменится.

✓ ***Примечание.*** *Вполне возможно, что Delphi не захочет сразу «заговорить» по-русски. Выберите в диалоговом окне, вызываемом с помощью «Tools» | «Editor Options», страницу «Display» и в выпадающем списке «Editor font» найдите кириллический шрифт (например, Courier New Cyr). (В более ранних версиях Delphi нужно обратиться к пункту меню «Environment Options».)*

Из довольно длинного списка свойств формы обратите внимание на четыре: Height, Width, Left, Top. Первые два обозначают высоту и ширину формы (в пикселях) соответственно, вторая же пара указывает расположение формы на экране относительно его левого верхнего угла. Попробуйте изменить размеры формы с помощью мышки – и значения свойств Height и Width тотчас же изменятся в Инспекторе Объектов. Перемещение формы немедленно отразится на значениях свойств Left и Top. Можно поступить и наоборот: менять значения свойств в Инспекторе Объектов. Форма изменит свой вид или местоположение в соответствии с новыми значениями свойств.

После того, как Вы получите окно желаемых размеров, и расположите его на экране так, как Вам захочется, следует сохранить проект. Выберите «File»|«Save Project As...», чтобы вызвать обычное диалоговое окно Windows сохранения файла. Delphi предложит Вам сохранить файл под именем Unit1.pas в рабочем каталоге по умолчанию (возможно, это будет каталог Projects, создаваемый при типичной инсталляции Delphi). Вы можете согласиться с предложенными именем и каталогом, но лучше выберите (или создайте) свой каталог для хранения проекта (например, подкаталог Hello в каталоге Chapter1) и назовите файл h1_First.pas. Когда Вы нажмете <OK>, появится следующее диалоговое окно, в котором требуется ввести имя проекта. По умолчанию проект будет сохранен в том же каталоге, в котором Вы только что сохранили файл. Назовите проект hello1.dpr и нажмите <OK>.

✓ ***Примечание.*** *Принцип именования файлов практически во всех главах пособия состоит в следующем: перед именем файла (которое, по возможности, отражает его назначение) ставится префикс из сокращенного до двух-трех букв имени проекта и символа подчеркивания (в данном случае h1_ – из hello1).*

Никогда не называйте файл и проект одинаковыми именами! Delphi не позволит Вам этого сделать, выдав сообщение об ошибке. Однако при работе с некоторыми младшими версиями можно таким образом потерять содержимое файла.

Теперь выполните команду «Run» из одноименного меню (или нажмите кнопку с зеленым треугольником на панели инструментов) – и Вы увидите Ваше первое Windows-приложение в работе. Обратите внимание: с экрана исчез Инспектор Объектов, стали недоступными некоторые «быстрые» кнопки и пункты меню. Редактор Кода и Палитра Компонентов по-прежнему остаются видимыми. И, главное, на панели задач Windows появилась кнопка с названием «Hello1».

✓ ***Примечание.*** *Команда «Run» выполняет два действия: компиляцию приложения и последующий запуск его на выполнение.*
Компьютер «понимает» только машинные коды, и в роли «переводчика» с Object Pascal выступает компилятор Delphi. Поэтому прежде чем приложение сможет быть исполненным, оно должно быть откомпилировано. Вы можете сначала выбрать команду «Compile hello1» из меню «Project», и только затем – команду меню «Run» | «Run».
Отметим, что выбор команды «Run» приводит к компиляции приложения в том случае, если оно не было откомпилировано ни разу или если с момента последней компиляции были внесены какие-либо изменения.

Как мы уже говорили, форма автоматически реализует стандартные возможности любого Windows-окна. Это значит, что можно менять размеры окна, перемещать его по экрану, свернуть его или, напротив, развернуть на весь экран. Чтобы завершить работу приложения и вернуться в Delphi, нужно нажать на кнопку с крестиком в правом верхнем углу окна (или воспользоваться клавиатурной комбинацией <Alt> <F4>).

Какие файлы создает Delphi

А теперь посвятим несколько минут изучению файлов, которые создала среда Delphi при компиляции приложения (полный список файлов в каталоге, в котором был сохранен проект, можно получить, например, с помощью Проводника Windows). Вы обнаружите файлы h1_First.dcu, h1_First.dfm, h1_First.pas, Hello1.cfg, Hello1.dof, Hello1.dpr, Hello1.exe и Hello1.res. Суммарно эти файлы занимают 291 Кб, причем размер исполняемого файла Hello1.exe составляет 286 Кб (возможно, у Вас получатся несколько иные цифры). Не так уж мало для одного небольшого приложения. Каким бы большим ни был Ваш жесткий диск, Вам хотя бы иногда приходится «наводить порядок» и удалять не слишком нужные файлы. Поэтому полезно знать, какие из файлов Delphi можно удалять без опаски.

Разработчики Delphi подразделяют создаваемые ею файлы на три большие группы:

— файлы, содержащие исходный код (Pascal source files);
— файлы, генерируемые компилятором (compiler-generated files);
— другие файлы, используемые при построении приложений (other files used to build application);

(названия приводятся в соответствии со справочной системой Delphi).

К группе файлов, содержащих исходный код, относят файлы:
— с расширением .pas – файлы модулей (unit source files), в которых содержится код Вашего проекта. Их не следует удалять, если Вы не собираетесь полностью уничтожить проект. Вы можете удалять резервные копии этих файлов, которые сохраняются с расширением .~pa;

— с расширением .dpr – файлы проектов (project files). Каждому приложению Delphi соответствует единственный файл проекта и один или несколько файлов модулей. Файл проекта содержит команды, позволяющие собрать воедино Ваше приложение (для тех, кто имеет опыт программирования на Pascal: файл проекта является в некотором смысле аналогом главного (main) файла в программе, состоящей из нескольких модулей). Не следует менять что-либо в этом файле без большой необходимости. Не следует также удалять этот файл, если, конечно, Вы не собираетесь уничтожить весь проект. Для этого файла также создается резервная копия с расширением .~dp, которую можно удалить;
— с расширением .dpk – файлы пакетов (package source files). Эти файлы подобны файлам проектов (и обращаться с ними следует также), но используются при создании пакетов – динамически связываемых библиотек специального вида.

Фактически все файлы исходного кода являются текстовыми файлами, написанными по определенным правилам. Компилятор производит их синтаксический анализ и затем преобразовывает в машинный код, понятный компьютеру.

✓ ***Примечание.*** *Вы можете «проследить» за процессом компиляции, включив опцию «Show compiler progress» на странице «Preferences» диалогового окна, вызываемого с помощью «Tools» | «Environment Options».*

Компилятор Delphi генерирует файлы:
— с расширением .dcu – файлы, содержащие скомпилированный код модулей программы. При первой компиляции для каждого файла модуля создается файл с тем же именем, что и у файла модуля, но расширением .dcu. В дальнейшем Delphi будет компилировать только новые файлы модулей (поскольку не находит для них файлов с расширением .dcu) и файлы, в которых были сделаны изменения со времени последней компиляции. Конечно же, .dcu-файлы всегда можно восстановить при компиляции приложения. Однако существуют ситуации, когда удаление этих файлов создает большие проблемы (при разработке и использовании собственных компонентов). А пока Вы работаете с простыми приложениями, не случится ничего страшного, если Вы удалите эти файлы;
— с расширением .exe – исполняемые файлы. Файл с таким расширением создается путем компиляции файла проекта, когда созданы все файлы .dcu для файлов модулей, составляющих проект. Он содержит весь скомпилированный код и формы. Файл .exe имеет то же имя, что и файл проекта. Наличие .exe-файла позволяет запускать приложение (как из интегрированной среды, так и вне ее). Исполняемые файлы имеют достаточно большой размер и не слишком уменьшаются при помещении их в архив. Если готовое приложение не будет часто использоваться, исполняемый файл

можно удалить: восстановить этот файл (если на Вашем компьютере установлена Delphi) – дело нескольких минут;
— с расширением .dll – файлы-библиотеки динамической компоновки (DLL – Dynamic Linked Library). В таких библиотеках содержатся процедуры и функции, к которым могут обращаться несколько программ одновременно. Существенно, что при исполнении программы библиотеки загружаются в память в нужный момент и выгружаются, как только программа перестает к ним обращаться. Если Вы сами написали библиотеку и у Вас есть ее полный исходный код, можете при необходимости удалить .dll-файл;

✓ ***Примечание.*** *Следует очень осторожно обращаться с файлами .dll, которые созданы не Вами. Библиотеки динамической компоновки играют очень важную роль в операционной системе Windows. Загляните, например, в каталог Windows \ System или Delphi \ Bin – и Вы насчитаете не один десяток файлов с расширением .dll, которые обеспечивают работу Windows и Delphi. Иногда файлы, содержащие библиотеки динамической компоновки, имеют расширение, отличное от .dll, например, .drv.*

— с расширением .bpl – файлы, создаваемые при компиляции пакетов. Эти файлы имеют то же имя, что и файлы с расширением .dpk. Фактически это Windows DLL с некоторыми специфическими для Delphi особенностями;
— с расширением .dcp – файлы, создаваемые при компиляции пакетов. Каждому файлу с расширением .dpk соответствует единственный файл с таким же именем и расширением .dcp. Такой файл содержит заголовок пакета и все .dcu-файлы пакета, просто присоединенные один к другому (как если бы над ними выполнили операцию конкатенации). Обращаться с ними нужно так же, как с файлами .dcu.

Кроме описанных выше, Delphi создает при построении приложений файлы:
— с расширением .dfm – файлы форм, разновидность файлов ресурсов Windows (файлы ресурсов Windows – это специальные файлы, в которых содержатся видимые и (или) логические элементы программ, например, растровые изображения, меню, таблицы строк). Они отвечают за визуальное представление Вашего проекта: всякое приложение Delphi (за исключением консольного) имеет, по крайней мере, одну форму. Каждый .dfm-файл содержит двоичное представление одной формы и ему обязательно соответствует .pas-файл с тем же именем. Удалять файлы с расширением .dfm следует только в том случае, когда удаляется весь проект. Отметим, что IDE Delphi дает возможность просматривать и даже редактировать файлы форм в текстовом представлении;
— с расширением .res – стандартные файлы ресурсов Windows, содержащие ярлыки (icons) приложений. Каждому приложению соответствует .res-

файл, имеющий то же имя, что и файл проекта. Если Вы не изменяете ярлык, который Delphi предлагает по умолчанию, можете при большой необходимости удалить этот файл: восстановить его несложно. К файлам ресурсов, имена которых не совпадают с именами приложений, следует относиться осторожно;

— с расширением .dof – файлы, содержащие настройки компилятора и компоновщика, установленные в диалоговом окне «Project»|«Options» («Параметры проекта»). Каждому проекту соответствует один файл с расширением .dof и именем, совпадающим с именем проекта. Не рекомендуется удалять этот файл по крайней мере до тех пор, пока работа над проектом не завершена;

— с расширением .dsk – файлы, содержащие конфигурацию рабочей области и некоторые другие настройки IDE, в основном установленные в диалоговом окне «Tools»|«Environment Options». Для каждого проекта создается свой файл с расширением .dsk с тем же именем, что и файл проекта. Его также не рекомендуется удалять до завершения работы над проектом;

— с расширением .cfg – файлы конфигурации проекта, содержащие директивы компилятора. Каждому файлу проекта соответствует файл с таким же именем и расширением .cfg. Пока работа над проектом не завершена, удалять этот файл не следует.

Это все еще не полный список файлов, которые могут быть созданы Delphi при разработке приложений. Но поскольку роль других типов файлов ограничена «узкой специализацией» использующих их приложений (например, приложений баз данных), мы будем рассматривать их по мере необходимости.

✓ ***Примечание.*** *Если Вы работаете с другой версией Delphi, то, возможно, не обнаружите файлов с какими-либо расширениями. Так, например, пакеты были введены только в Delphi 3, а роль файлов .dof в Delphi 1 выполняли файлы .opt. Уточнить, зачем нужны файлы с теми или иными расширениями, можно с помощью справочной системы.*

Пример 1.2. Создание обработчика события

Созданное Вами приложение hello1 не делало почти ничего, только выводило на экран окно с заголовком «Привет, мир!». Обыкновенное пустое окно Windows, которое можно переместить, увеличить или уменьшить, свернуть, закрыть. Сейчас мы рассмотрим создание приложения, которое будет реагировать на определенные Вами события.

Окно нового приложения уже не будет пустым: в нем будет два компонента. Процесс помещения на форму компонентов из Палитры Компонентов интуитивно ясен. Достаточно щелкнуть левой клавишей мыши по нужному компоненту на Палитре, а затем – в форме. Если Вы выбрали не

тот компонент, выбор можно отменить щелчком на кнопке со стрелкой слева от Палитры Компонентов.

Начните новый проект и поместите на форму метку (Label) и кнопку (Button) со страницы Standard Палитры Компонентов. По умолчанию, при помещении на форму Delphi дает компонентам соответствующие имена, добавляя порядковый номер. Нумерация ведется отдельно для компонентов каждого типа. Так, помещенные Вами на форму компоненты получат имена Button1 и Label1.

Когда компоненты находятся на форме, их можно перемещать или изменять их размеры с помощью мыши или Инспектора Объектов. Более широкие возможности по редактированию предоставляет меню «Edit», ряд пунктов которого дублирован в контекстном меню, вызываемом нажатием правой клавиши мыши в области формы. Некоторые действия можно производить над группой выделенных компонентов.

✓ ***Примечание.*** *Если компонент выбран (выделен), то на форме он обрамлен маленькими черными прямоугольниками – маркерами. Чтобы одновременно выделить на форме несколько компонентов (группу), следует щелкнуть по каждому из них мышкой, удерживая нажатой клавишу <Shift>. Группа компонентов выделяется серыми маркерами.*

Так, например, компоненты легко выравнивать друг относительно друга, выбирая пункт меню «Edit» | «Align...» или же соответствующий пункт контекстного меню и отмечая в появляющемся диалоговом окне нужные способы выравнивания по горизонтали и по вертикали. Выделите компоненты Button1 и Label1 как группу и выровняйте их по левым сторонам (Left sides).

Как и форма, компоненты кнопки и метки имеют свойство Caption (Заголовок). Замените в Инспекторе Объектов заголовок кнопки строкой «Щелкни!» (или «Click me!»), а заголовок метки – пустой строкой. Вы можете действовать при этом двумя способами: активизировать Инспектор Объектов и, поочередно выбрав Button1 и Label1 из выпадающего списка, произвести необходимые изменения, или же выбирать компонент на форме, а затем обращаться к Инспектору Объектов – он переключится на выбранный Вами компонент самостоятельно.

Теперь перейдите на страницу событий Инспектора Объектов для компонента Button1. Вы увидите достаточно длинный список событий, на которые может прореагировать кнопка. Нас сейчас интересует только событие OnClick, означающее одинарный щелчок на кнопке левой клавишей мыши.

✓ ***Примечание.*** *Наименования всех событий в Инспекторе Объектов начинаются с префикса On (с англ. – Включено). Это общепринятое соглашение, а не обязательное требование.*

Выберите событие OnClick. В поле события (т.е. в правой части страницы в выбранной строке) появится кнопка со стрелкой – признаком выпадающего списка. В настоящий момент он не содержит ни одного элемента. Дважды щелкните левой клавишей мыши в верхней строке списка. В результате активизируется Редактор Кода, в котором появятся (автоматически) следующие строки:

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

end;
```

Delphi создала «заготовку» обработчика события – одинарного щелчка левой клавишей мыши (Click) на компоненте Button1. Кроме этого, в поле события OnClick добавится название обработчика этого события Button1Click.

Название обработчика события создается средой Delphi по следующему правилу: к имени компонента (Button1) добавляется наименование события без префикса On. Квалификатор TForm1 в названии процедуры фактически означает, что кнопка Button1 расположена на форме Form1. Пустая строчка между **begin** (начать) и **end** (кончить) предназначена для кода, который должен быть написан программистом. Конечно, он может занимать больше, чем одну строку, но сейчас нам вполне хватит отведенного пространства. Напечатайте между **begin** и **end**:

```
Label1.Caption := 'Привет!';
```

✓ ***Примечание.*** *Знак «:=» представляет собой единое целое (пишется без пробела) и обозначает операцию присваивания. В результате выполнения этой операции переменная, расположенной слева от знака операции, получает значение (выражения), расположенного справа от знака операции.*

Итак, процедура, которую Вы (совместно с Delphi) написали, обеспечивает получение свойством Caption компонента Label1 в качестве значения строки 'Привет!', причем только тогда, когда во время выполнения приложения на кнопке Button1 (имеющей заголовок «Щелкни!») пользователь щелкнет левой клавишей мыши.

✓ ***Примечание.*** *Следует четко понимать разницу между приложением во время проектирования и приложением во время выполнения. Во время проектирования Вы можете помещать на форму компоненты из Палитры компонентов (или удалять их), изменять их размеры и расположение, определять события, связанные с ними. Однако эти события могут наступить только во время выполнения. Так, щелчок левой клавишей мыши на кнопке Button1 во время разработки приведет к выделению ее черными маркерами, а то же действие во время выполнения изменит заголовок метки.*

Сохраните Ваш проект под именем hello2.dpr, а файл с исходным кодом – под именем h2_Second.pas и запустите его на выполнение (Windows 3.1 не поддерживает длинных имен файлов, и если Вы работаете в этой среде, сократите имя файла до h2_Secnd.pas). Вы увидите форму с кнопкой. Щелкните по кнопке мышкой, и в том месте, где во время проектирования была расположена метка, появится надпись «Привет!».

Завершите работу приложения hello2 с помощью системного меню или клавиатурной комбинации <Alt><F4> и вернитесь в Delphi. В следующем разделе мы займемся подробным изучением внутреннего устройства проекта.

Анатомия проекта

Delphi позволяет создавать как приложения с графическим пользовательским интерфейсом (GUI, Graphic User Interface), так и консольные, выполняющиеся в окне DOS. Консольному приложению не нужна форма, и оно может состоять только из одного файла исходного кода (файла .dpr). А приложение GUI содержит хотя бы одну форму, и, следовательно, должно содержать по крайней мере два файла исходного кода – файл проекта и файл модуля, связанного с формой.

Давайте посмотрим, как устроен проект, создаваемый Delphi, на примере проекта Hello2. Файлов, содержащих исходный код, всего два: h2_Second.pas и Hello2.dpr. По умолчанию Delphi не показывает файл проекта в окне Редактора Кода. Этот файл полностью генерируется Delphi, и необходимость редактировать его возникает довольно редко. Чтобы просмотреть содержимое файла проекта, выберите в меню «Project» команду «View Source». В окне Редактора Кода рядом с закладкой «h2_Second» появится вторая закладка «Hello2». Файл проекта должен содержать следующие строки:

Листинг 1.1. Файл hello2.dpr

```
01   program hello2;
02
03   uses
```

```
04      Forms,
05      h2_Second in 'h2_Second.pas' {Form1};
06
07    {$R *.RES}
08
09    begin
10      Application.Initialize;
11      Application.CreateForm(TForm1, Form1);
12      Application.Run;
13    end.
```

✓ ***Примечание.*** *На самом деле Delphi не нумерует строки. Номера поставлены автором, чтобы облегчить ссылки на ту или иную строку.*

✓ ***Примечание.*** *Жирным шрифтом Delphi выделяет зарезервированные (ключевые) слова языка Object Pascal (reserved words, keywords). Эти слова имеют строго определенное значение, которое не может быть изменено программистом. Полный список зарезервированных слов можно получить с помощью справочной системы Delphi.*
Отметим, что среда Delphi безразлична к регистру символов, поэтому, например, Hello2, hello2, HELLO2 или даже hELlO2 означает для нее одно и то же. Это относится и к зарезервированным словам.

В файле проекта (часто называемом программой) выделяют следующие разделы: заголовок (heading), предложение **uses** (uses clause, необязательный элемент) и блок (block of declarations and statements, дословно: блок объявлений и операторов). Все эти разделы можно видеть в Листинге 1.1.

Заголовок (строка 01) состоит из зарезервированного слова **program** (программа), которое позволяет компилятору идентифицировать данный файл как файл проекта приложения, и имени проекта. Завершается заголовок программы точкой с запятой. В нашем случае имя проекта – hello2.

✓ ***Примечание.*** *Проект получает имя при сохранении проекта. Файл проекта и сам проект не могут иметь разные имена: таково требование Delphi.*

Вторая строка пуста, равно как и строки 06 и 08. Вообще говоря, пустых строк могло бы не быть вовсе, но пространственное разделение логически обособленных блоков кода делает текст программы более ясным для понимания и удобочитаемым. Имейте это в виду, когда будете писать свой код.

Строка 03 начинается с еще одного зарезервированного слова – **uses** (использует). Это слово открывает список имен всех использованных модулей, который обычно называют предложением uses. В этот список вхо-

дят имена как стандартных (входящих в состав самой Delphi) модулей, так и пользовательских (в данном случае написанных Вами). Они перечисляются через запятую, а в конце списка ставится точка с запятой. Имена модулей могут следовать в любом порядке, однако принято записывать имена пользовательских модулей после имен стандартных.

Следующие две строки (04 и 05) указывают, что же именно использует программа: стандартный модуль Forms, позволяющий приложению работать с формами, и Ваш модуль h2_Second. Delphi знает, где найти стандартный модуль, а для модуля, написанного Вами, приходится указывать (с помощью зарезервированного слова **in**), что он содержится в файле h2_Second.pas.

✓ ***Примечание.*** *Если файл проекта и файл модуля находятся в разных каталогах, будет указано не только имя файла, но и путь к нему.*

Кроме этого, в фигурных скобках указана форма, которая связана с этим файлом. Как правило, все, что записывается в фигурных скобках, считается комментарием и игнорируется компилятором. Существует лишь одно исключение – комментарий вида {$ ...}. Такой комментарий называется директивой компилятора.

Примером директивы компилятора является строка 07. Включение директивы {$R *.RES} в файл проекта позволяет связать файл ресурсов (как правило, в нем содержится пиктограмма) с приложением. Отметим, что символ * в директиве $R обозначает имя файла с исходным кодом (без расширения), в котором эта директива находится.

Строки 09 и 13 являются началом и окончанием блока кода (тела программы). Строки, расположенные между ними, обеспечивают инициализацию приложения (10), создание формы Form1 (11) и запуск приложения на выполнение (12). Блок кода (а вместе с ним и программа) заканчивается точкой.

Теперь обратимся к модулю h2_Second и проанализируем его построчно. Файл модуля несколько отличается от файла проекта. Переключитесь в Редакторе Кода на закладку h2_Second, и Вы увидите значительно более пространный код. Сейчас мы ограничимся описанием его основных разделов и проведем их сравнение с разделами файла Hello2.dpr.

Листинг 1.2. Файл h2_Second.pas

```
01   unit h2_Second;
02
03   interface
04
05   uses
06     Windows, Messages, SysUtils, Classes, Graphics,
```

```
07    Controls, Forms, Dialogs, StdCtrls;
08
09  type
10    TForm1 = class(TForm)
11      Label1: TLabel;
12      Button1: TButton;
13      procedure Button1Click(Sender: TObject);
14    private
15      { Private declarations }
16    public
17      { Public declarations }
18    end;
19
20  var
21    Form1: TForm1;
22
23  implementation
24
25  {$R *.DFM}
26  procedure TForm1.Button1Click(Sender: TObject);
27  begin
28   Label1.Caption := 'Привет';
29  end;
30
31  end.
```

В этом файле можно выделить следующие части: заголовок модуля (строка 01), интерфейсный раздел (строки 03 – 22), раздел реализации (строки 23 – 29) и раздел инициализации, содержащий в данном случае только зарезервированное слово **end** (строка 31). Эти разделы обязательно присутствуют в любом модуле.

Заголовок модуля состоит из ключевого слова **unit** и имени модуля (h2_Second). Имя модуля должно совпадать с именем файла, в котором он содержится. Компилятор ищет модуль в одноименном файле, и, если не находит, производит поиск имени файла, совпадающего с именем модуля, усеченным до 8 символов. Вы не должны вручную менять имя файлов модулей и проектов. Поэтому самый безопасный путь – сразу после создания нового модуля сохранять его как файл под нужным именем. Заголовок модуля, как и заголовок программы, завершается точкой с запятой.

Раздел интерфейса начинается зарезервированным словом **interface** и продолжается до начала раздела реализации. Это – «видимая» часть модуля. Вся информация, содержащаяся в нем (объявления типов, констант, переменных, процедур), может быть доступна другим модулям и программам.

Раздел реализации начинается зарезервированным словом **implementation** и продолжается до начала раздела инициализации. Раздел реализации – «скрытая» часть модуля. В нем могут содержаться как объявления, так и конкретный код. К содержимому этой части модуля никакой другой модуль или программа не могут обратиться напрямую.

Обратите внимание: в интерфейсном разделе помещен только заголовок обработчика события:

```
13      procedure Button1Click(Sender: TObject);
```

в то время как его код располагается в разделе реализации (строки 26 – 29).

Коротко прокомментируем строки модуля h2_Second, составляющие содержимое разделов интерфейса и реализации.

Строки с 05 по 07 представляют собой список использованных модулей (предложение **uses**) раздела интерфейса. Он заметно длиннее, чем в файле проекта. В данном случае все модули в этом списке являются стандартными.

Строки 09 – 18 представляют собой объявление типа формы TForm1.

✓ ***Примечание.*** *Delphi (как и Pascal) – строго типизированный язык, и тип определяет не только набор характеристик некоторого объекта, но и действия, которые могут быть над ним произведены. По соглашению, принятому в Delphi, для имен большинства типов используется префикс T.*

Для нас сейчас существенны только три строки из этого объявления:

```
11      Label1: TLabel;
12      Button1: TButton;
13      procedure Button1Click(Sender: TObject);
```

Фактически они указывают, что на форме находятся метка Label1 (типа TLabel) и кнопка Button1 (типа TButton), а также определена процедура (procedure) – обработчик события «одинарный щелчок левой клавишей мыши на кнопке Button1».

✓ ***Примечание.*** *Строки, подобные строкам 11 и 12 появляются, когда компоненты помещаются на форму, а строка, подобная строке 13 – когда создается обработчик события.*

Строки 20 и 21 указывают на существование экземпляра формы с названием Form1 типа TForm1, описанного выше.

Строка 25 – директива компилятора, подобная той, что записана в файле проекта. Наличие этой директивы позволяет связать .pas-файл с формой, хранящейся в файле .dfm с тем же именем.

Строки 26 – 29 – тот самый обработчик события, в который Вы вписали свою строку кода.

Заканчивается модуль зарезервированным словом **end** и точкой.

Файлы форм

Файл формы, имеющий расширение .dfm, является в реальности двоичным файлом, в котором хранится информация о форме и размещенных на ней компонентах. Delphi дает возможность просматривать и редактировать этот файл в текстовом представлении. Изучение текстового представления файлов форм может оказаться полезным, например, при переносе проекта в другую версию Delphi. Кроме этого, файл формы в текстовом виде позволяет составить достаточное представление о форме, не видя ее. Вы можете встретить в книгах текстовые файлы форм вместо их изображения. Файлы форм могут приоткрыть некоторые секреты Delphi, связанные с иерархией и взаимоотношениями компонентов. Научиться читать и понимать файлы форм не так уж сложно. Поэтому продолжим рассмотрение проекта Hello2.

В проекте Hello2 есть только одна форма и, соответственно, один файл формы h2_Second.dfm. Чтобы просмотреть файл h2_Second.dfm в текстовом виде, Вы можете либо воспользоваться контекстным меню формы и выбрать в нем команду «View as text», либо открыть этот файл с помощью главного меню «File»|«Open» (Delphi автоматически преобразует его к текстовому виду).

✓ ***Примечание.*** *Вместе с Delphi поставляется программа Convert.exe (она находится в подкаталоге Bin), позволяющая получить текстовое представление файлов форм вне Delphi. Эта программа запускается из командной строки и в качестве параметров может принимать имена одного или нескольких файлов форм (.dfm). В результате каждый файл формы будет конвертирован в файлы с таким же именем, но с расширением .txt (фактически, текст DOS). Если же в качестве параметров Convert.exe указать текстовые файлы, они будут преобразованы в файлы форм.*
Программа Convert.exe особенно полезна, когда нужно перенести проект из одной версии Delphi в другую.

Скорее всего, значения Left, Top, Width и Height в Вашем файле будут отличаться от приведенных здесь.

Листинг 1.3. Файл Second.dfm в текстовом представлении.

```
01   object Form1: TForm1
02     Left = 192
03     Top = 107
```

```
04      Width = 544
05      Height = 375
06      Caption = 'Form1'
07      Color = clBtnFace
08      Font.Charset = DEFAULT_CHARSET
09      Font.Color = clWindowText
10      Font.Height = -11
11      Font.Name = 'MS Sans Serif'
12      Font.Style = []
13      OldCreateOrder = False
14      PixelsPerInch = 96
15      TextHeight = 13
16      object Label1: TLabel
17        Left = 96
18        Top = 24
19        Width = 3
20        Height = 13
21      end
22      object Button1: TButton
23        Left = 208
24        Top = 128
25        Width = 75
26        Height = 25
27        Caption = 'Щелкни!'
28        TabOrder = 0
29        OnClick = Button1Click
30      end
31    end
```

Как можно видеть, в этом файле содержатся значения свойств и заголовки обработчиков событий формы, метки и кнопки. Эти характеристики сгруппированы с помощью слов **object** и **end**, причем блоки, описывающие кнопку (строки 22 – 30) и метку (строки 16 – 21), вложены в блок, описывающий форму.

Файл формы можно редактировать. Так, например, в проекте Hello2 мы оставили без изменений заголовок формы. Впечатайте в строке 06 файла формы вместо 'Form1' слова 'Мой второй проект'. У Вас должно получиться:

```
06    Caption = 'Мой второй проект'
```

Теперь с помощью контекстного меню вернитесь к обычному представлению формы (команда «View as form») или сохраните файл .dfm с помощью главного меню, а затем вновь откройте файл .pas. Заголовок формы изменится.

Наконец, проведем еще одно преобразование файла формы Second.dfm. После слова **end**, завершающего описание кнопки Button1 (т.е. перед словом **end**, завершающим описание формы целиком, а вместе с тем и файл), впечатайте следующие строки:

```
object Button2: TButton
  Left = 258
  Top = 178
  Width = 75
  Height = 25
  Caption = 'Выход'
  TabOrder = 0
end
```

Такая запись является описанием кнопки Button2, имеющей заголовок «Выход». Значения Left и Top отличаются от значений кнопки Button1 на 50 пикселей. Если теперь Вы перейдете от текстового представления формы к обычному, то обнаружите чуть ниже и правее кнопки Button1 (с заголовком «Щелкни!») кнопку Button2 (с заголовком «Выход»). Обратившись к странице свойств Инспектора Объектов, Вы можете убедиться, что значения свойств Left, Top, Width, Height и Caption кнопки Button2 соответствуют введенным при редактировании текстового представления Second.dfm. Но если теперь Вы откроете файл модуля Second.pas, то не увидите там никаких изменений: к строкам (см. Листинг 1.2)

```
10      TForm1 = class(TForm)
11        Label1: TLabel;
12        Button1: TButton;
```

не добавилось описание кнопки Button2:

```
Button2: TButton;
```

Несмотря на то, что кнопка Button2 присутствует в выпадающем списке Инспектора Объектов и располагается на форме, Ваш проект не собирается ее замечать.

Формально можно дописать еще строку в блок описания кнопки Button2:

```
object Button2: TButton
    Left = 258
    Top = 178
    Width = 75
    Height = 25
    Caption = 'Выход'
```

```
    TabOrder = 0
    OnClick = Button2Click
  end
```

Но это не приведет к созданию «заготовки» обработчика события в редакторе кода, хотя на странице событий в Инспекторе Объектов внесенные изменения отразятся. Вы можете убедиться в этом самостоятельно, изменив соответствующим образом файл формы. Более того, попытка создать обычным образом обработчик события OnClick для кнопки Button2 не будет успешной: двойной щелчок в строке события OnClick ни к чему не приведет. Придется вновь перейти к текстовому представлению файла формы и удалить строчку

```
OnClick = Button2Click
```

✓ ***Примечание.*** *Конечно, можно дописать недостающие строки в файле модуля самостоятельно – образец в виде кнопки Button1 имеется. Но, во-первых, есть риск (и большой) наделать ошибок, а во-вторых, зачем Вам Delphi, если Вы не доверяете ей?*

Теперь обработчик события создается, и в файле модуля появляются соответствующие строки. Вы можете написать в этом обработчике события Button2Click строчку, подобную

```
Label1.Caption := 'Привет от Button2!';
```

откомпилировать проект и запустить его на выполнение. Он будет работать, хотя описание кнопки Button2 так и не появится. Разве что Вы добавите его вручную.

Коротко говоря, прибегать к такому способу добавления компонентов в форму (путем корректировки текстового представления файла формы) без крайней необходимости не следует. Delphi ожидает от Вас определенных действий при создании проекта и точно знает, что нужно написать в файле модуля, когда Вы помещаете на форму компонент из Палитры Компонентов. Она внесет необходимые изменения и в файл формы. Словом, Delphi отлично выполняет свою часть работы, и Вам редко придется редактировать строки, написанные ею.

Классификация лексем

Программы и модули фактически являются текстовыми файлами, написанными по определенным синтаксическим правилам. Чтобы компьютер мог выполнить запрограммированные действия, необходимо преобразовать эти файлы к машинному коду.

Это преобразование, называемое трансляцией, и осуществляет компилятор. Первый этап трансляции – выделение лексем.

✓ ***Примечание.*** *Лексема (token) – это минимальная синтаксически значимая единица в тексте программы.*

Второй этап трансляции – синтаксический анализ (последовательность лексем должна быть записана в соответствии с правилами языка). Результатом этого этапа является программа, записанная во внутренних кодах компилятора.

✓ ***Примечание.*** *Если компилятор обнаруживает синтаксические ошибки, в нижней части Редактора Кода появляется дополнительное окно Messages (Сообщения), в котором указывается местоположение ошибки (имя модуля и строка) и ее характер. После исправления ошибок следует заново откомпилировать проект.*

На третьем этапе трансляции генерируется собственно машинный код.

Лексемы подразделяются на следующие категории: специальные символы, идентификаторы, зарезервированные слова, директивы, числа, метки и символьные строки. При этом между находящимися рядом идентификаторами, зарезервированными словами, числами и метками должен присутствовать как минимум один символ-разделитель. В других случаях использование разделителя необязательно, но желательно – в целях улучшения читабельности программы. Разделители не могут входить в состав лексем, исключением являются лишь символьные строки.

Разделителями с точки зрения Object Pascal являются пробел, символ табуляции, составной символ конца строки (переход на следующую строку и возврат каретки), а также комментарий. Символы-разделители служат лишь для разграничения лексем и игнорируются компилятором.

✓ ***Примечание.*** *Компилятор также игнорирует любую последовательность символов, если она находится после зарезервированного слова* **end**, *заканчивающегося точкой.*

Комментарий – это специальным образом оформленная последовательность символов, используемая для документирования программы. Комментарий может быть записан тремя способами:

с помощью фигурных скобок: весь текст, заключенный между левой фигурной скобкой и правой фигурной скобкой, считается комментарием. Текст может занимать как одну, так и несколько строк:

```
{Это    комментарий}
```

с помощью круглых скобок и звездочек, которые действуют так же, как и фигурные:

```
(* И это – комментарий *)
```

с помощью двух прямых слешей (иногда говорят о комментариях C – стиля). Текст, записанный после двух прямых слешей, считается комментарием. Признаком окончания комментария в этом случае является конец текущей строки:

```
// И такие комментарии тоже возможны.
```

✓ ***Примечание.*** *Допускается (хотя и не приветствуется) вложение комментариев разных типов, например:*

```
{комментарий первого уровня (* комментарий второго уровня *)}
```

```
(* { // а это еще более глубокое вложение комментариев } *)
```
Вложение комментариев одного типа недопустимо.

Если после открывающей скобки (фигурной или круглой со звездочкой) следующий символ – знак доллара $, такой комментарий является директивой компилятора. В параграфе «Анатомия проекта» мы уже обсуждали директиву компилятора {$R *.RES}

Опишем каждую из категорий лексем более подробно.

Специальные символы – это символы или пары символов, имеющие в языке Object Pascal предопределенное назначение. Эти символы (отдельные или входящие в пару) не являются буквами или цифрами. В приведенной ниже таблице перечислены все специальные символы, используемые в Delphi 6.

#	$	&	@	^	.	+	–	*	/	'
:	;	,	<	>	=	(	)	[	]	{
}	(*	*)	(.	.)	..	//	:=	<=	>=	<>

✓ ***Примечание.*** *Круглые скобки со звездочкой являются эквивалентом фигурных (это уже было отмечено при обсуждении комментариев), а круглые скобки с точкой действуют так же, как квадратные скобки.*

Идентификаторы (или имена) представляют собой последовательность букв латинского алфавита, цифр от 0 до 9 и символов подчеркивания, начинающуюся с буквы или символа подчеркивания. Идентификатор может иметь любую длину, но значимыми являются первые 255 символов. В качестве идентификаторов не могут быть использованы зарезервированные слова. Идентификаторы применяются для именования констант, типов, полей, свойств, переменных, процедур, функций, модулей, программ, библиотек и пакетов.

✓ ***Примечание.*** *В Delphi используются также квалифицируемые (уточненные) идентификаторы, которые выглядят следующим образом:*

```
<Идентификатор1>.<Идентификатор2>
```

Вы уже использовали такой синтаксис в проекте Hello2, когда меняли заголовок метки:

```
Label1.Caption := 'Привет!'
```

Действительно, не только метка, но и кнопка, и форма имеют свойство Caption (заголовок), и уточнить, какой же именно заголовок имеется в виду, совершенно нелишне. Существуют и другие случаи, когда необходимо использовать квалифицируемые идентификаторы.

Зарезервированные слова фактически являются словами английского языка (или их сокращениями), имеющими в языке Object Pascal предопределенное назначение, которое не может быть изменено. Эти слова в Редакторе Кода выделяются полужирным шрифтом (здесь и далее предполагается, что используются настройки по умолчанию). В приведенной ниже таблице перечислены все зарезервированные слова Delphi 6.

and	exports	mod	shr
array	file	nil	string
as	finalization	not	then
asm	finally	object	threadvar
begin	for	of	to

case	function	or	try
class	goto	out	type
const	if	packed	unit
constructor	implementation	procedure	until
destructor	in	program	uses
dispinterface	inherited	property	var
div	initialization	raise	while
do	inline	record	with
downto	interface	repeat	xor
else	is	resourcestring	
end	label	set	
except	library	shl	

Некоторые слова из этой таблицы Вам уже знакомы, остальные же будут изучаться по мере надобности. Полный список зарезервированных слов Вашей версии Delphi Вы можете получить с помощью справочной системы.

✓ ***Примечание.*** *В Object Pascal есть еще несколько слов, которые зарезервированными не считаются, но при определенных обстоятельствах действуют как зарезервированные. Мы затронем эту тему при обсуждении классов и исключений.*

Директивы похожи на зарезервированные слова. Директивы также имеют специальное назначение в Object Pascal, но оно может быть изменено пользователем (хотя разработчики Delphi и не рекомендуют это делать). Дело в том, что из контекста всегда ясно, когда слово используется в качестве директивы, а когда – играет роль пользовательского идентификатора. Директивы – это, скорее, дополнительные указания (чаще всего относящиеся к процедуре или функции), описывающие особенности реализации программного объекта. В приведенной ниже таблице перечисляются все директивы Object Pascal, действующие в Delphi 6.

absolute	external	override	reintroduce
abstract	far	package	requires
assembler	forward	pascal	resident
automated	implements	private	safecall
cdecl	index	protected	stdcall
contains	message	public	stored
default	name	published	virtual
dispid	near	read	write
dynamic	nodefault	readonly	writeonly
export	overload	register	

Редактор Кода выделяет директивы так же, как и зарезервированные слова, – полужирным шрифтом.

✓ ***Примечание.*** *Читателю, знакомому с языком Pascal, известны стандартные процедуры ввода и вывода read и write. Они по-прежнему применяются при работе с файлами и в консольных приложениях. Директивы read и write используются в объявлениях классов*

Числа, с которыми оперирует Object Pascal, могут быть как целыми, так и вещественными. Целые числа могут быть представлены как в десятичной, так и в шестнадцатеричной системе счисления, а вещественные – только в десятичной. Числа записываются как последовательность цифр, без пробелов и запятых, которая может предваряться знаками «+» или «–». Использование знака «+» не является обязательным: любое число без знака считается положительным. Для отделения дробной части от целой в вещественных числах используется точка. Допускается экспоненциальная (иногда употребляется название «научная») запись чисел, например: 6.3E–2 (= $6.3 \cdot 10^{-2}$), 12.23e2 (=$12.23 \cdot 10^{2}$ = 1223) или 34e+3 (=$34 \cdot 10^{3}$ = 34000). Числа, записанные в экспоненциальном виде, всегда считаются вещественными (даже если фактически они являются целыми).

Целые шестнадцатеричные числа записываются с префиксом $ (знак доллара), например: $1C. Можно использовать числа из диапазона от $00000000 до $FFFFFFFF.

В качестве метки может быть использован любой допустимый идентификатор либо последовательность не более чем из четырех десятичных цифр. Такая последовательность не может иметь знак, а нули перед значащими цифрами игнорируются.

Символьные строки могут состоять из строк в одиночных кавычках (quoted string) и строк управляющих символов (control string), а также их комбинаций. Строки в одиночных кавычках представляют собой последовательность не более чем 255 символов кода ANSI (для латинских букв совпадает с ASCII – кодом), заключенную в апострофы (одиночные кавычки). Это означает, что можно использовать буквы как латинского, так и русского алфавита.

```
'This is a string. Это - строка.'
```

Если в такую строку необходимо включить символ апострофа (как часть строки), нужно напечатать его два раза подряд (без пробелов):

```
'I don''t know.'
```

Чтобы убедиться в этом, откройте проект Hello2 и замените строку приветствия (строчка 28 листинга 1.2) приведенной выше строкой. Заголовок метки Label1 после щелчка на кнопке будет выглядеть следующим образом:

```
I don't know.
```

Если между одиночными кавычками нет ни одного символа, строка считается пустой:

```
'' // Такая строка считается пустой.
' ' // А это строка, состоящая из символа "пробел".
```

Строки в одиночных кавычках должны быть записаны «в одну строку» и не допускают переносов.

Строки управляющих символов являются последовательностью управляющих символов, каждый из которых состоит из префикса # и целого числа от 0 до 255 (записанного в десятичной или шестнадцатеричной системе счисления). Эти числа являются кодами ANSI и с их помощью можно получить полные аналоги строк в одиночных кавычках:

```
#89#101#115 // Так записывается слово "Yes" в кодах ANSI.
```

Отметим, что если в такой строке несколько символов, между ними не должно быть пробелов. Также не следует делать пробелы, если управляющие символы используются наряду с символами (или строками) в одиночных кавычках:

```
'Y'#101's' // Еще один вариант записи слова "Yes".
```

Поскольку среди кодов ANSI есть и коды непечатаемых символов, таких как «перевод строки», «возврат каретки», с помощью строк управляющих символов можно создавать более сложные структуры:

```
'Смотри в корень!'#13#10#9'Козьма Прутков'
```

Если заголовок метки в проекте Hello2 получит в качестве значения эту строчку, то во время выполнения Вы увидите на экране следующее:

```
Смотри в корень!
    Козьма Прутков.
```

✓ ***Примечание.*** *Когда заголовок метки или какое-либо другое свойство, принимающее строковое значение, редактируется в Инспекторе Объектов, использовать управляющие символы по прямому назначению невозможно.*

Лексемы объединяются в смысловые единицы более высокого уровня – объявления (declaration), выражения (expression) и операторы (statement). Можно провести аналогию с тем, как составляются предложения из слов, причем в «обычном» языке так же существует несколько типов предложений.

Объявления (в литературе также используется термин «описания») служат для определения идентификаторов, которые впоследствии могут использоваться в выражениях и операторах (необъявленные идентификаторы в программе использовать нельзя). Существуют свои синтаксические особенности при объявлении меток, констант, типов, переменных, процедур и функций. К объявлениям также принято относить предложения exports (exports clause), однако они редко встречаются в программах и, как правило, применяются при написании библиотек (DLL).

Выражения играют в Object Pascal ту же роль, что формулы в математике – задают правила обработки данных. Выражения определяют, как вычислить то или иное значение, но не определяют, как использовать полученный результат. Выражения являются составными частями операторов и объявлений.

Операторы представляют собой описания действий, которые должны быть исполнены программой.

Подробно о структуре проектов и модулей

В параграфе «Анатомия проекта» мы обсудили внутреннее строение проекта Hello2 и файлов, его составляющих. Здесь мы намерены рассмотреть структуру файлов проекта и модуля более формально.

Изучение языка – в значительной степени итеративный процесс. Можно дать непротиворечивое и последовательное определение каждому встречающемуся в языке объекту, но продемонстрировать, как используется этот объект, «взятый в отдельности», практически невозможно. Поэтому мы приведем здесь лишь те определения (и в том объеме), которые нас сейчас интересуют. Необязательные элементы мы будем заключать в квадратные скобки, повторяющиеся элементы – в фигурные скобки, а понятия (не являющиеся зарезервированными словами или именами) – в угловые скобки. Если квадратные или фигурные скобки используются в качестве специальных символов языка, такие скобки заключаются в одинарные кавычки. Вертикальная черта «|» будет использоваться в качестве союза «или». Группа символов «::=» означает «по определению есть».

✓ ***Примечание.*** *Внимательный читатель заметит, что это несколько упрощенная версия металингвистических формул Бэкуса – Наура.*

Файл проекта (приложения) состоит из следующих элементов:

```
[program <имя программы> [<список параметров>];]
 //заголовок программы

[uses <список использованных модулей>;] // предложение uses

<блок>.
```

✓ ***Примечание.*** *Краткую сводку грамматических правил, согласно которым пишутся программы, можно получить с помощью оперативной справки, выбрав formal grammar. Здесь и далее синтаксические описания приводятся в соответствии с документацией Delphi.*

Заголовок программы считается необязательным элементом. Однако на практике Delphi автоматически даст имя программе при первом сохранении проекта. В качестве имени программы может выступать любой допустимый идентификатор. Необязательный элемент <список параметров> – «наследство» стандартного Pascal (в качестве параметров обычно выступали файлы ввода и вывода). Компилятор Delphi игнорирует любые параметры, содержащиеся в этом списке.

✓ ***Примечание.*** *Если заголовок отсутствует (например, программист специально удалил его), при первом сохранении проекта Delphi сообщит об ошибке в предложении uses (в дальнейшем это сообщение не появляется). Проект будет сохранен, но нарушится связь с модулями и формами, добавляемыми средствами Delphi. Отметим также, что в отсутствие заголовка программист должен явно указать файл ресурсов, связываемый с приложением. Впрочем, если разрабатывается простое консольное приложение, заголовок можно опустить.*

Предложение **uses** также является необязательным. В нем действительно нет необходимости, если разрабатывается консольное приложение, состоящее только из файла проекта. Во всех остальных случаях список использованных модулей будет содержать, по меньшей мере, один элемент. При добавлении в проект нового модуля Delphi автоматически дополняет список использованных модулей именем модуля и именем файла, в котором этот модуль содержится. Если файл модуля находится в том же каталоге, что и файл проекта, Delphi ограничивается именем файла, в противном случае указывается и путь к нему. Поэтому перемещение отдельных файлов "вручную" нежелательно; лучше всего создавать для каждого проекта свой каталог, помещать в этот каталог все относящиеся к проекту файлы и при необходимости перемещать его целиком.

Блок объявлений и операторов определяется следующим образом:

```
<блок> ::=
     [<раздел объявлений>]
     <раздел операторов>

<раздел объявлений> ::=
     [<раздел (объявлений) меток>]
```

```
      [<раздел (объявлений) констант>]
      [<раздел (объявлений) типов>]
      [<раздел (объявлений) переменных>]
      [<раздел (объявлений) процедур и функций>}

<раздел операторов> ::= <составной оператор>
```

Объявления меток, констант, типов, переменных, процедур и функций могут чередоваться в произвольном порядке. Более того, в целях улучшения структурированности кода можно многократно использовать разделы объявлений одних и тех же программных конструкций. Главное – следить за тем, чтобы при описании некоторого элемента не использовались элементы, не описанные ранее.

✓ ***Примечание.*** *Разделы объявлений иногда называют секциями*

```
<составной оператор> ::=
     begin
     <список операторов>
     end

<список операторов> ::= <оператор> {; <оператор>}
```

Файл проекта завершается точкой.

Файл модуля устроен несколько сложнее:

```
unit <имя модуля>;

<интерфейсный раздел>
<раздел реализации>
<раздел инициализации>.

<интерфейсный раздел> ::=
     interface
     [uses <список использованных модулей>]
     <раздел объявлений интерфейса>

<раздел реализации> ::=
     implementation
     [uses <список использованных модулей>]
     <раздел объявлений>

<раздел инициализации> ::=
     initialization <список операторов>
     [finalization <список операторов>] end
     | begin <список операторов> end
     | end

<раздел объявлений интерфейса> ::=
     {<раздел (объявлений) констант>]
     [<раздел (объявлений) типов>}
```

```
[<раздел (объявлений) переменных>]
[<раздел заголовков процедур и функций>]
```

Файл модуля, как и файл проекта, заканчивается точкой.

Обратите внимание, что и интерфейсный раздел, и раздел реализации могут содержать предложение **uses**. Это позволяет легко выйти из ситуации, когда модули должны ссылаться друг на друга. Однако любой используемый модуль может фигурировать в списке только одного из этих предложений uses. Следует помнить, что указывать в предложении uses данного модуля нужно лишь имена тех модулей, к ресурсам которых данный модуль обращается непосредственно. Если два модуля (или больше), имена которых указаны в предложении uses, объявляют ресурсы (константы, переменные, типы, процедуры или функции) с одинаковыми именами, компилятор будет обращаться к ресурсу, объявленному в модуле, имя которого в предложении uses расположено ближе к концу предложения.

Отметим, что в модуле всегда есть раздел инициализации, хотя очень часто этот раздел состоит из зарезервированного слова **end**, завершающего модуль. Тем не менее, иногда (в том числе и в документации Delphi) разделом инициализации называют часть модуля, начинающуюся зарезервированным словом **initialization** и продолжающуюся до зарезервированного слова **finalization**, а в случае его отсутствия – до конца модуля. Часть модуля, начинающуюся словом **finalization**, называют разделом завершения.

Раздел инициализации следует использовать, например, в том случае, если в модуле определяются структуры данных, которые следует инициализировать до начала выполнения программы. Отметим, что разделы инициализации модулей выполняются до старта приложения в том же порядке, в котором имена этих модулей расположены в предложении **uses** файла проекта.

✓ ***Примечание.*** *Несмотря на то, что первым методом, который вызывается в файле проекта Delphi, является Application.Initialize, приложение сначала будет выполнять код, расположенный в разделах инициализации модулей, используемых приложением.*

В разделе завершения размещается код, который должен выполняться при завершении (возможно, аварийном) работы приложения. Выполнение разделов завершения модулей происходит в порядке, обратном порядку следования имен этих модулей в предложении uses.

✓ ***Примечание.*** *Старайтесь избегать ситуаций, когда выполнение раздела инициализации или завершения данного модуля зависит от выполнения соответствующих разделов других модулей.*

Теперь рассмотрим более подробно синтаксис разделов объявлений.

Всякий оператор в тексте программы может быть помечен не более чем одной меткой путем указания этой метки через двоеточие перед этим оператором. Синтаксис меток рассматривался в предыдущем параграфе. Сейчас отметим лишь, что необходимо объявлять метки даже в том случае, если они представляют собой последовательность цифр.

```
<раздел объявления меток> ::=
    label <метка> {,<метка>);
```

Пример:

```
label 89, MyLabel;
```

Константой называют значение, которое определено в тексте программы и не может меняться во время выполнения программы. В Object Pascal существует несколько различных конструкций, для которых используется термин «константа», и не все из них необходимо объявлять. Так, принято говорить о числовых константах (любое записанное в явном виде в программе число), строковых константах (примером является слово «Привет!», которое мы присваивали заголовку метки в проекте Hello2). Впоследствии мы встретимся еще с несколькими видами констант, в том числе и предопределенными. Но есть два вида констант, которые создаются (т.е. под них резервируется память) с помощью объявлений: это истинные константы (true constant) и типизированные константы (typed constant). Обычно в литературе истинные константы называют просто константами, и мы также будем следовать этой традиции. Истинная константа – это объявленный идентификатор, значение которого не может быть изменено. Типизированные константы отличаются от истинных тем, что могут во время выполнения программы изменять свое значение.

✓ ***Примечание.*** *Поведение типизированных констант может регулироваться директивой компилятора {$J}. Начиная с версии 6 по умолчанию действует директива {$J–}, которая «превращает» типизированную константу в истинную. Директива {$J+} позволяет типизированной константе получить новое значение в ходе выполнения программы.*

```
<раздел объявления констант> ::=
     const {<объявление константы>;}

<объявление константы> ::=
     <идентификатор> = <константное выражение>
     | <идентификатор> : <тип> = <типизированная константа>

<типизированная константа> ::=
     <константное выражение> |
     <константа-массив> |
     <константа-запись>
```

Пример:

```
const
     Avogadro_Number = 6.022E23;
     k_B = 1.3807e-23;
     R = Avogadro_Number * k_B;
     g : Single = 9.8;
```

Объявления истинных и типизированных констант могут чередоваться после зарезервированного слова **const** в произвольном порядке.

Константное выражение – это выражение, которое может быть вычислено компилятором до исполнения программы. В константное выражение, в частности, могут входить числа, символьные строки, уже определенные истинные константы.

Типизированная константа может иметь любой тип, кроме файлового и типа Variant (типы рассматриваются в следующей главе). Для некоторых типов (тип – массив, тип – запись) действуют особые правила. Они будут приведены при описании соответствующих типов.

Тип определяет множество значений данных этого типа, множество операций над данными этого типа, а также способ хранения данных в памяти компьютера. В Object Pascal существует ряд предопределенных типов, объявлять которые не требуется. Вместе с тем языком предоставляются широкие возможности по созданию пользовательских типов данных, которые объявляются по следующему образцу:

```
<раздел объявления типов> ::=
     type
          {<объявление типа> ;}

<объявление типа> ::=
     <имя типа> = <тип> |
```

В качестве имени типа может быть использован любой допустимый идентификатор. При объявлении различных пользовательских типов существуют свои синтаксические особенности, которые мы будем рассматривать по мере изучения этих типов.

Переменная – это программный объект, способный принимать и изменять значение в ходе выполнения программы. Формально переменная представляет собой идентификатор, используемый в качестве адреса области памяти, где хранится значение (переменной). Объявление переменной резервирует для нее память в соответствии с типом. Раздел объявления переменных имеет следующий синтаксис:

```
<раздел объявления переменных> ::=
     var  {<объявление переменной>;}

<объявление переменной> ::=
     <список идентификаторов>: <тип>

<список идентификаторов> ::= <идентификатор>,{<идентификатор>}

Пример:
     var
          i: integer;
          x, y: real;
```

✓ ***Примечание.*** *Вообще говоря, это не единственный способ объявления переменной. Так, глобальную (т.е. объявленную непосредственно в соответствующем разделе модуля или файла проекта) переменную можно сразу инициализировать:*

```
<идентификатор> : <тип> = <константное выражение>;
```

если только ее тип не является файловым или вариантным. Инициализировать переменные можно только по отдельности, например:

```
var k: real = sqrt(2); d: real = sqrt(3);
```

При разработке достаточно больших и сложных программ удобно как с точки зрения восприятия, так и с точки зрения отладки разделять решаемую задачу на подзадачи. В Object Pascal существуют различные способы для выделения подзадач программным образом: проекты разделяются на модули, в рамках модуля используются процедуры и функции (напомним, что обработчики событий являются процедурами); кроме того, есть и другие возможности.

Процедуры и функции, которые часто называют подпрограммами, действительно практически в точности воспроизводят структуру «настоящей» программы (файла

проекта). Объявления процедур и функций выглядят весьма похоже, отличие состоит в том, что в заголовке функции указывается тип значения этой функции:

```
<раздел объявлений процедур и функций> ::=
          {<объявление процедуры> | <объявление функции>}

<объявление процедуры> ::=
     <заголовок процедуры>;[<директива> {; <директива>}]
<блок>;

<объявление функции> ::=
     <заголовок функции>;[<директива> {; <директива>}]
<блок>;

<заголовок процедуры> ::=
procedure <идентификатор>[(<список формальных параметров>)]

<заголовок функции> ::=
function <идентификатор>[(<список формальных параметров>)]:
                    <тип возвращаемого функцией значения>
```

Использование формальных параметров делает процедуры и функции более универсальными. Формальные параметры определяют лишь типы данных, над которыми оперирует функция или процедура, но не фиксируют конкретные значения. При вызове процедуры или функции список формальных параметров должен быть заменен соответствующим списком фактических параметров.

✓ ***Примечание.*** *В Object Pascal определено достаточно большое количество стандартных процедур и функций. Чтобы использовать их, достаточно указать модуль, в котором они объявлены, в одном из предложений uses Вашего модуля (обычно в предложении uses раздела реализации). Наиболее часто используемые стандартные функции объявлены в модуле System, который автоматически присоединяется к любому проекту.*

Процедура или функция, описанная в разделе реализации некоторого модуля, может быть доступна в других модулях, если ее заголовок помещен в интерфейсном разделе этого модуля:

```
<раздел заголовков процедур и функций> =
     {<заголовок процедуры>; [<директива> {; <директива>}]} |
     {<заголовок функции>; [<директива> {; <директива>}]}
```

При изучении проекта Hello2 мы уже сталкивались с подобной ситуацией: процедура – обработчик события щелчка на кнопке была описана в разделе реализации модуля h2_Second.pas, а ее заголовок указан в интерфейсном разделе этого модуля.

Вопросы

1. Что такое обработчик события?
2. Как создать обработчик события для объекта?
3. Поясните, что означает следующая запись:
   TForm1.Button3Click (Sender: TObject) ?
4. Из каких разделов состоит файл проекта? Покажите эти разделы в файле Вашего первого проекта Hello1.
5. Какие разделы являются обязательными в файле модуля?
6. Какие расширения у имен файлов проекта и модуля?
7. Должен ли совпадать заголовок модуля с именем файла, в котором этот модуль содержится?
8. Для чего нужны директивы компилятора в файлах hello1.dpr и h1_first.pas?
9. Какие из файлов, созданных при разработке проекта hello1, можно удалить, чтобы впоследствии можно было восстановить исполняемый файл?
10. Как получить текстовое представление файла формы?

Задания

1. Разработайте приложение, которое будет выполнять описанные ниже действия.
   a) При работе приложения в его окне (форме) выводится фраза «Delphi приветствует Вас».
   b) В левом верхнем углу окна приложения расположена кнопка. При одинарном щелчке на кнопке левой клавишей мыши она (кнопка) смещается на 10 пикселей вправо и 10 пикселей вниз.
   c) В окне приложения выводится надпись «Hello!». При одинарном щелчке левой кнопкой мыши на этой надписи осуществляется ее «перевод» на русский: «Привет!».
   d) В окне приложения расположены кнопка и метка. Метка имеет заголовок: «Это мой проект». Одинарный щелчок на кнопке левой клавишей мыши приводит к замене существующего заголовка формы заголовком метки.
   e) В окне приложения расположена метка. Одинарный щелчок левой клавишей мыши на метке приводит к смещению метки на 30 пикселей вниз, а щелчок на форме – смещает метку на 15 пикселей вверх и 5 пикселей вправо.
   f) В окне приложения расположены метка и кнопка. Одинарный щелчок на метке приводит к тому, что кнопка перемещается и закрывает собой метку. Щелчок на кнопке возвращает кнопку в исходное положение.
   g) В окне приложения расположены метка и кнопка. Метка имеет заголовок «Прыгай!», заголовок у кнопки отсутствует. Одинарный щелчок левой клавишей мыши на метке приводит к исчезновению заголовка метки и появлению этого заголовка у кнопки, щелчок на кнопке восстанавливает исходную ситуацию.
   h) В окне приложения расположены метка и кнопка. Одинарный щелчок левой клавишей мыши на метке приводит к смещению метки и кнопки на 20 пикселей вниз, а щелчок на кнопке – к смещению метки и кнопки на 20 пикселей влево.
   i) В окне приложения расположена кнопка. Одинарный щелчок левой клавишей мыши на кнопке приводит к увеличению ее во всех направлениях (вверх, вниз, вправо и влево) на 10 пикселей.
   j) В окне приложения расположена кнопка. Одинарный щелчок левой клавишей мыши на кнопке приводит к уменьшению высоты формы на 20 пикселей, а щелчок на форме – к увеличению высоты формы на 10 пикселей.

k) В окне приложения расположены метка и кнопка. Щелчок на кнопке приводит к ее перемещению в правый нижний угол формы, а метки в левый верхний угол. Щелчок на форме меняет их местами.

l) В окне приложения расположена кнопка. Щелчок по кнопке приводит к тому, что она заполняет собой всю доступную область формы.

m) В окне приложения расположена метка. Щелчок по форме помещает метку точно по центру формы.

n) В окне приложения расположены метка и кнопка. Щелчок по кнопке приводит к тому, что заголовок проекта становится заголовком метки, а заголовок метки становится заголовком кнопки.

o) В окне приложения расположена кнопка. Щелчок по кнопке делает ее квадратной, щелчок по форме уменьшает длину и высоту кнопки в 1,5 раза.

2. Изучите с помощью справочной системы Delphi свойство Visible (для кнопки или метки – это не принципиально). Разработайте приложение, которое будет выполнять описанные ниже действия. Используйте свойство Visible для управления видимостью кнопок и меток.

a) В окне приложения расположены метка с заголовком «Delphi приветствует Вас!» и кнопка. Одинарный щелчок левой клавишей мыши на кнопке приводит к исчезновению метки с экрана, щелчок в любом месте формы – к возникновению надписи.

b) В окне приложения расположены две кнопки, но изначально видна только одна из них. Одинарный щелчок левой клавишей мыши на видимой кнопке приводит к ее исчезновению, и появлению ранее невидимой. Щелчок на форме делает видимыми одновременно обе кнопки.

c) В окне приложения расположена кнопка. Одинарный щелчок левой клавишей мыши на кнопке приводит к исчезновению кнопки с экрана, щелчок на форме – к ее появлению на 5 пикселей правее исходного положения.

d) В окне приложения расположены две метки с заголовками «Мышка 1» и «Мышка 2». Одинарный щелчок левой клавишей мыши на форме приводит к появлению кнопки с заголовком «Кошка». Щелчок на кнопке приводит к исчезновению меток.

e) В окне приложения расположены две кнопки с заголовками «День» и «Ночь». Одинарный щелчок левой клавишей мыши на кнопке «День» приводит к появлению метки с заголовком «Солнце», щелчок на кнопке «Ночь» приводит к исчезновению «Солнца» и к появлению такой же «Луны» и двух «Звезд».

f) В окне приложения расположены по кругу несколько меток. Изначально видима только одна из них. Одинарный щелчок левой клави-

шей мыши на видимой метке приводит к ее исчезновению с экрана, но делает видимой следующую.

g) В окне приложения расположены кнопка и метка, причем кнопка изначально закрывает метку. Одинарный щелчок левой клавишей мыши на кнопке приводит к ее исчезновению с экрана, щелчок на форме вновь показывает ее.

h) В окне приложения расположены три кнопки и метка. Изначально видимы только кнопки. Одинарный щелчок левой клавишей мыши по любой из кнопок приводит к появлению метки с заголовком «Выравнивание». Щелчок по метке располагает кнопки, выравнивая их по левому краю. Щелчок на форме вновь делает метку невидимой.

i) В окне приложения расположены две кнопки. Изначально видима только одна из них. Одинарный щелчок левой клавишей мыши на видимой кнопке приводит к обмену заголовками по следующей схеме. Заголовок видимой кнопки становится заголовком формы, а заголовок формы – заголовком как видимой, так и невидимой кнопок. После этого видимая кнопка исчезает с экрана, а невидимая появляется на форме.

j) В окне приложения расположены две кнопки (с заголовками «Включить» и «Выключить») и метка. Изначально видимы кнопка «Выключить» и метка. Одинарный щелчок левой клавишей мыши на кнопке «Выключить» приводит к исчезновению этой кнопки и метки и появлению кнопки «Включить». Щелчок на кнопке «Включить» приводит к появлению метки и кнопки «Выключить» и исчезновению кнопки «Включить».

k) В окне приложения расположены две кнопки: «Щука» и «Карась» и три метки – «Водоросли». Щелчок по кнопке «Карась» перемещает «Щуку» на 25 пикселей ближе к «Карасю», а «Карась» прячется в «Водорослях»

l) В окне приложения расположены три кнопки: «Дворник», «Ветер» и «Тротуар» (эту кнопку поместите внизу формы и сделайте ее достаточно длинной) и несколько меток – «Листьев». Нажатие на кнопку «Дворник» приводит к исчезновению «Листьев», нажатие на кнопку «Ветер» возвращает «Листья» на прежние места.

m) В окне приложения расположены две кнопки: «Кошка» и «Собака» и три метки: «Рыба», «Молоко», «Кость». Нажатие на кнопку «Кошка» приводит к исчезновению «Рыбы» и «Молока», нажатие на кнопку «Собака» приводит к исчезновению «Кошки», «Молока» и «Кости».

n) В окне приложения расположены две кнопки: «Кошка» и «Хозяйка» и метка «Сосиска». Кнопка «Хозяйка» изначально невидима. Нажатие на кнопку «Кошка» приводит к исчезновению «Сосиски» и появ-

лению «Хозяйки», нажатие на кнопку «Хозяйка» приводит к исчезновению «Кошки».

о) В окне приложения расположены пять кнопок: «Семечко», «Росток», «Бутон», «Цветок», «Плод». Изначально видима только кнопка «Семечко». Нажатие на эту кнопку приводит к ее исчезновению и появлению «Ростка», нажатие на кнопку «Росток» делает невидимой ее и видимой кнопку «Бутон», и так далее. Нажатие на кнопку «Плод» делает видимой кнопку «Семечко».

Глава 2. Простые типы данных

Стандартные типы данных. Перечислимый и интервальный типы. Совместимость и приведение типов. Практикум. Знаки операций и выражения. Классификация операторов.

Стандартные типы данных

Прежде чем компьютер сможет выполнить программу, компилятор должен преобразовать ее текст в машинный код. Машинный код фактически представляет собой последовательность нулей и единиц.

✓ ***Примечание.*** *То, что машинный код является двоичным, связано с техническими, аппаратными особенностями компьютеров. Мы не будем обсуждать это подробно, однако заметим, что каждому символу, который используется при записи машинного кода, требуется сопоставить определенное (причем устойчивое) физическое состояние памяти. Технически два устойчивых состояния реализовать проще. Кроме того, простота гарантирует надежность, страхует от ошибок. Сравните, например, таблицу умножения, составленную для 0 и 1, с той, что Вы учили в школе.*

В виде последовательности нулей и единиц записываются как команды (символы и группы символов, имеющие предопределенное значение), так и данные. В языках высокого уровня реализована концепция типов данных, и это позволяет программисту не задумываться о машинном представлении данных. Тип определяет множество значений, которые могут принимать данные этого типа, множество операций над данными этого типа, а также способ хранения данных этого типа в памяти компьютера.

Object Pascal является строго типизированным языком. Случается, что из-за этого текст программы выглядит более громоздким, чем хотелось бы. Но преимущество строгой типизации состоит в том, что ряд ошибок удается диагностировать еще на стадии компиляции программы. Впрочем, в Object Pascal существуют механизмы, позволяющие обойти требование строго определять тип, но у Вас должны быть очень веские причины, чтобы использовать обходные пути.

В этом параграфе мы подробно рассмотрим классификацию типов Object Pascal. Во-первых, можно говорить о стандартных (предопределенных) типах и типах, определяемых пользователем. Все стандартные типы имеют заранее известные имена и автоматически распознаются компилятором (объявлять их не нужно).

✓ ***Примечание.*** *Имена стандартных типов не являются зарезервированными словами.*

Во-вторых, типы могут быть фундаментальными (fundamental) и общими (generic). Диапазон и формат записи фундаментальных типов (многие стандартные типы являются фундаментальными) не меняется в зависимости от процессора и операционной системы (обычно говорят – платформы). Общие же типы являются зависимыми от платформы. Использование общих типов позволяет оптимизировать код для конкретной системы, но может стать источником ошибок при переходе к другой реализации (при изменении объема памяти, который отводится для значений такого типа).

Наконец, согласно документации, типы Object Pascal подразделяются на простые (simple), строковые (string), структурированные (structured), указательные (pointer), процедурные (procedural) типы и вариантные (variant) типы. Мы возьмем за основу именно эту классификацию, однако будем указывать, является ли тот или иной тип стандартным или пользовательским, фундаментальным или общим. В настоящей главе мы ограничимся рассмотрением простых и строковых типов.

К простым относятся порядковые и вещественные типы. Множество значений порядкового типа является линейно упорядоченным. Это означает, что каждый элемент имеет единственный элемент, который ему предшествует, и единственный элемент, который за ним следует.

Среди порядковых типов, в свою очередь, выделяют целые, символьные, булевы, или логические, типы (все эти типы являются стандартными), а также перечислимые типы и интервальные типы, или типы – диапазоны (эти типы относятся к определяемым пользователем).

Группа целых типов включает в себя два общих целых типа и семь фундаментальных. Для 32-битного компилятора Delphi 6 общие целые типы имеют следующие форматы записи и диапазоны значений:

— Integer, 32-битное целое со знаком. Диапазон значений – от –2147483648 (= -2^{31}) до 2147483647 (= 2^{31}–1);
— Cardinal, 32-битное целое без знака. Диапазон значений – от 0 до 4294967295 (= 2^{32}-1).

Фундаментальные целые типы:

— ShortInt, 8-битное целое со знаком. Диапазон значений – от –128 до 127;
— Byte, 8-битное целое без знака. Диапазон значений – от 0 до 255;
— SmallInt, 16-битное целое со знаком. Диапазон значений – от –32768 до 32767;
— Word, 16-битное целое без знака. Диапазон значений – от 0 до 65535;
— LongInt, 32-битное целое со знаком. Диапазон значений – от –2147483648 до 2147483647;
— LongWord, 32-битное целое без знака. Диапазон значений – от 0 до 424967295;

— Int64, 64-битное целое со знаком. Диапазон значений – от –9223372036854775808 до 9223372036854775807.

Группа символьных типов состоит из двух фундаментальных типов и одного общего. К фундаментальным относятся:

— AnsiChar, 8 бит. Диапазон значений – расширенный набор символов ANSI (набор символов Windows);

— WideChar, 16 бит. Диапазон значений – набор символов Unicode. (Первые 256 символов Unicode совпадают с символами ANSI).

Общий тип:

— Char, который эквивалентен AnsiChar.

В группу булевых (логических) типов входит один общий и три фундаментальных типа:

— Boolean, 8 бит (общий).

Фундаментальные типы:

— ByteBool, 8 бит;

— WordBool, 16 бит;

— LongBool, 32 бит.

Диапазон значений всех булевых типов исчерпывается двумя предопределенными константами, True и False. Считается, что значению False соответствует 0 (причем в случае типов, занимающих более 1 байта = 8 бит, число 0 во всех байтах), а значению True – любое ненулевое число. Однако присваивать числовые значения переменным булевого типа непосредственно нельзя.

Первый из типов, Boolean, является предпочтительным: любое логическое выражение возвращает значение этого типа. Три других типа введены для совместимости с Windows и другими языками программирования.

Для всех порядковых типов определены следующие стандартные функции:

— Ord(X): Longint. Принимает в качестве аргумента выражение порядкового типа, возвращает порядковый номер значения этого аргумента (кроме аргументов типа Int64).

— Pred(X). Принимает в качестве аргумента выражение порядкового типа, возвращает значение, предшествующее значению этого выражения.

— Succ(X). Принимает в качестве аргумента выражение порядкового типа, возвращает значение, следующее за значением этого выражения.

— High(X). Принимает в качестве аргумента переменную порядкового типа или идентификатор порядкового типа. Возвращает максимальное значение типа.

— Low(X). Принимает в качестве аргумента переменную порядкового типа или идентификатор порядкового типа. Возвращает минимальное значение типа.

✓ ***Примечание.*** *Функции High и Low могут также применяться при работе с массивами и строками.*

Кроме этих функций, при работе с порядковыми типами могут использоваться процедуры Inc и Dec (инкремента и декремента). Они принимают в качестве аргументов один обязательный параметр порядкового типа и один необязательный параметр типа LongInt. Эти процедуры увеличивают значение порядкового типа на то количество единиц, которое указано во втором параметре. Значение по умолчанию для необязательного параметра – единица.

Группа вещественных типов состоит из одного общего типа и шести фундаментальных. Общий тип:

— Real, 64 бит. Диапазон значений от $5.0 \cdot 10^{-324}$ до $1.7 \cdot 10^{308}$, значащих цифр 15 – 16.

К фундаментальным типам относятся:

— Real48, 48 бит. Диапазон значений от $2.9 \cdot 10^{-39}$ до $1.7 \cdot 10^{38}$, значащих цифр 11 –12. Этот тип в предыдущих версиях Object Pascal являлся «подлинным» типом Real. Введен для обратной совместимости;

— Single, 32бит. Диапазон значений от $1.5 \cdot 10^{-45}$ до $3.4 \cdot 10^{38}$, значащих цифр 7 – 8;

— Double, 64 бит. Диапазон значений от $5.0 \cdot 10^{-324}$ до $1.7 \cdot 10^{308}$, значащих цифр 15 –16. В настоящей реализации языка совпадает с общим типом Real;

— Extended, 80 бит. Диапазон значений от $3.6 \cdot 10^{-4951}$ до $1.1 \cdot 10^{4932}$, значащих цифр 19 – 20;

— Comp, 64 бит. Этот тип часто называют целым в формате вещественного. Диапазон значений от $-2^{63}+1$ до $2^{63}-1$, или от –9223372036854775808 до 9223372036854775807. В отличие от целых типов он не считается порядковым, хотя переменные и константы этого типа фактически могут иметь только целые значения;

— Currency, 64 бит. Этот тип от всех остальных отличает наличие фиксированной десятичной точки, которая отводит под дробную часть числа 4 позиции. На самом деле это тоже «почти целый» тип, как и Comp. Диапазон значений от –922337203685477.5808 до 922337203685477.5807, значащих цифр 19 – 20. Назначение этого типа понятно из названия (currency с англ. – валюта). Такой тип позволяет наиболее корректно обходиться с денежными величинами.

Строка, с точки зрения Object Pascal, это последовательность символов, заключенная в одиночные кавычки. Delphi работает с тремя стандартными строковыми типами (есть еще указательные типы, которые можно трактовать как строковые, но мы пока их не обсуждаем):

— ShortString, известные также, как строки «старого стиля». Максимальная «емкость» такой строки –255 символов, а в памяти она занимает от 2 до 256 байт (1 байт = 8 бит). Эти строки используются для обеспечения обратной совместимости с Delphi 1.0;

— AnsiString, или длинные строки. Такие строки могут содержать до 2^{31} символов и занимать в памяти от 4 байт до 2 гигабайт. Учитывая современные характеристики компьютеров, ограничение на длину строки скорее будет наложено операционной системой, нежели Delphi. Строки AnsiString содержат обычные ANSI – символы, каждый из которых занимает 8 бит;

— WideString предназначены для хранения символов Unicode (каждый из которых занимает 16 бит). Такие строки могут содержать до 2^{30} символов и занимать в памяти от 4 байт до 2 гигабайт, как и длинные строки.

Зарезервированное слово **string**, которое может быть использовано при объявлении переменной, константы или другого программного объекта строкового типа, действует подобно идентификатору общего типа. По умолчанию (при включенной директиве компилятора {$H+}), **string** считается тождественным AnsiString, но отключение этой директивы приводит к компиляции кода, который трактует **string** как ShortString.

Пользовательские типы на основе строкового создавать очень легко. Обычно это делают, если заранее известно, что максимальная длина строки будет небольшой, например, 8 или 25 символов. Тогда можно объявить пользовательские типы:

```
type
    T_DOSFileName = string[8];
    T_String25 = string[25];
```

просто указав в квадратных скобках после зарезервированного слова **string** максимальную длину строки. Базовым типом в этих и подобных объявлениях всегда будет считаться тип ShortString (и, следовательно, нельзя объявить пользовательский строковый тип с длиной, превышающей 255 символов).

Отметим, что компилятор спокойно относится к использованию в выражениях различных типов строк, и автоматически производит необходимые преобразования.

Перечислимый и интервальный типы

Перечислимый (перечисляемый) тип является порядковым типом, который может определяться программистом. В качестве примера пере-

числимых типов, используемых в Delphi, можно привести тип TFormBorderStyle, значения которого соответствуют различным стилям границы формы (т.е. определяют наличие у формы заголовка и доступ к пунктам системного меню), или тип Boolean. Перечислимые типы вводятся главным образом для удобства программиста. Текст программы становится куда более понятным, если для обозначения дней недели применяются сокращения типа mon, tue, wed, а не цифры 1, 2, 3 и т.д. Объявление перечислимого типа выглядит следующим образом:

```
<имя типа> = (<список идентификаторов>)

<список идентификаторов> ::=
    <идентификатор>{, <идентификатор>}
```

Идентификаторы рассматриваются как константы определяемого типа.

Пример:

```
type
    week = (mon, tue, wed, thi, fri, sat, sun);
    T_Fruits = (frApple, frKiwi, frPear, frOrange);
```

Тип week (неделя) определяется перечислением идентификаторов, соответствующих дням недели, начиная с понедельника.

✓ ***Примечание.*** *Обратите внимание, что идентификаторы не являются строковыми константами.*

Тип T_Fruits задан перечислением четырех фруктов: яблока, киви, груши и апельсина. При этом использованы следующие соглашения: имя типа начинается с T, а идентификаторы имеют префикс fr, указывающий на их принадлежность к типу T_Fruits. Эти соглашения не являются обязательными, но автор находит их удобными, и, как правило, будет им следовать.

✓ ***Примечание.*** *Подобные соглашения используют разработчики Delphi. Например, уже упоминавшийся тип TFormBorderStyle объявляется так:*

```
type TFormBorderStyle = (bsNone, bsSingle, bsSizeable,
bsDialog, bsToolWindow, bsSizeToolWin);
```

Автор отделяет в имени типа префикс T символом подчеркивания, чтобы отличать типы Delphi от своих собственных.

Допустимо использовать определения типов непосредственно в разделе объявления переменных:

```
var
    WeekDay: (mon, tue, wed, thi, fri, sat, sun);
    MyFruit: (Apple, Kiwi, Pear, Orange);
```

однако такой подход может ограничить программиста при манипуляциях с данными. Например, следующее объявление будет отвергнуто компилятором:

```
var
    MyFruit1: (Apple, Kiwi, Pear, Orange);
    MyFruit2: (Apple, Kiwi, Pear, Orange);
```

Дело в том, что компилятор трактует каждое неявное определение типа как новый тип. Поэтому, если переменные находятся в разных списках идентификаторов, их типы (с точки зрения компилятора) различны, и попытка присвоить значение переменной из одного списка переменной из другого списка приведет к ошибке «Incompatible types» (несовместимые типы). В случае перечислимых типов ситуация осложняется тем, что идентификаторы, перечисленные в круглых скобках, рассматриваются как константы определяемого типа. Таким образом, объявление переменной MyFruit2 будет некорректным подобно повторному объявлению константы MyConst в следующем примере:

```
const
    MyConst: Integer = 1; MyConst: Real = 1.0;
```

В обоих случаях компилятор сообщит об ошибке «Identifier redeclared» (повторное объявление идентификатора). Объявление переменных перечислимого типа без явного объявления этого типа корректно записывается так:

```
var
    MyFruit1, MyFruit2: (Apple, Kiwi, Pear, Orange);
```

По той же причине нельзя, например, определить совместно с типом week («неделя») еще один перечислимый тип – workweek («рабочая неделя»), используя те же сокращения для дней недели:

```
type
    week = (mon, tue, wed, thi, fri, sat, sun);
    workweek = (mon, tue, wed, thi, fri);
```

Это объявление типов будет «забраковано» компилятором. Тип workweek следует объявить как интервальный.

Интервальный тип чаще всего используется для дополнительного контроля значений переменных и является каким-либо диапазоном уже определенного (стандартного или пользовательского) порядкового типа. Объявление такого типа выглядит следующим образом:

```
<имя типа> =
    <константное выражение 1>..<константное выражение 2>
```

Пример:

```
type
    workweek = mon .. fri;
    T_MyDigits = 0 .. 9;
    T_MyChars = 'A' .. 'Z';
    T_SomeNumbers = -128 .. 128;
```

Здесь <константное выражение 1> и <константное выражение 2> – любые допустимые константные выражения, которые принимают значения одного и того же типа, называемого базовым. Так, базовым для типа workweek будет тип week (он должен быть объявлен ранее), а для типа T_MyDigits – стандартный тип Byte.

✓ ***Примечание.*** *Вообще говоря, использование выражений в объявлениях интервального типа может вызвать определенные проблемы. Пример приведен в документации:*

```
const
    X = 50;
    Y = 10;
type
Scale = (X - Y) * 2..(X + Y) * 2;
```

Попытка откомпилировать такой код приведет к ошибке. Круглая скобка, следующая сразу за знаком «=», с точки зрения компилятора является началом объявления перечислимого типа. Исправить «ошибку» можно следующим образом:

```
Scale = 2 * (X - Y)..(X + Y) * 2;
```

В качестве примера интервального типа, используемого Delphi, можно упомянуть тип TBorderStyle, являющийся диапазоном базового типа TFormBorderStyle:

```
TBorderStyle = bsNone..bsSingle;
```

Значения типа TBorderStyle определяют стиль границы для некоторых элементов управления (в том числе для компонентов Edit и Memo, которые будут изучаться далее в этой главе).

Переменные интервального типа обладают всеми свойствами переменных базового типа, однако при выполнении программы значение переменной интервального типа должно принадлежать заявленному диапазону.

✓ ***Примечание.*** *Чтобы компилятор реально производил проверку границ, следует отметить опцию «Range checking» на странице «Compiler» диалогового окна «Project Options», которое вызывается при выборе пункта «Options» меню «Project». Альтернативный способ состоит в использовании директивы компилятора {$R+} для части кода. Вместе с тем нужно учитывать, что проверка границ несколько увеличивает размеры программы и делает ее медленнее. Поэтому применять проверку границ стоит только при отладке. При генерации окончательного варианта приложения следует исключить отладочную информацию.*

Совместимость и приведение типов

Как уже подчеркивалось, Object Pascal является строго типизованным языком. Вместе с тем он предоставляет программисту широкие возможности по объявлению собственных типов данных. В связи с этим возникает проблема совместного использования данных различных типов или, говоря коротко, совместимости типов. Два типа могут быть тождественными (полностью совместимыми), совместимыми в выражениях, совместимыми по присваиванию, либо несовместимыми. В настоящем параграфе мы ограничимся примерами совместимости только рассмотренных типов данных.

Типы T1 и T2 считаются тождественными в следующих случаях:

— если при объявлении T1 и T2 использовался один и тот же идентификатор типа;
— если тип T2 объявлен эквивалентом T1.

Пример:

```
type
     T1 = Integer;
     T2 = Integer;
     T3 = T2;
     T4 = T3;
```

Первые две строки примера иллюстрируют применение одного и того же идентификатора типа Integer, тип T3 объявлен как эквивалент типа T2, а тип T4 – как эквивалент типа T3. В результате все объявленные типы будут взаимно тождественны друг другу и стандартному типу Integer.

✓ ***Примечание.*** *Если требуется создать тип, не тождественный оригиналу, следует использовать зарезервированное слово* **type** *в объявлении типа, например:*

```
type TNotIdentInt = type Integer;
```

Подчеркнем, что тождественность достигается за счет использования в объявлениях идентификаторов типа. Так, следующие объявления:

```
type
    T5 = (Apple, Kiwi, Pear, Orange);
    T6 = (Apple, Kiwi, Pear, Orange);
```

полностью идентичные с «нормальной», человеческой точки зрения, являются разными (притом неверно объявленными) типами с точки зрения компилятора (мы затрагивали эту проблему при обсуждении перечислимого типа в предыдущем параграфе).

Язык накладывает требование тождественности типов только для фактических и формальных параметров-переменных процедур и функций (см. главу 6). В остальных случаях подобной строгости не требуется. Интуитивно ясно, что вряд ли стоит запрещать перемножение целого и вещественного чисел, или добавления символа к строке. Операндами различных операций могут быть программные объекты не обязательно тождественных типов.

Типы T1 и T2 считаются совместимыми в выражениях, если выполняется одно из следующих условий:

- эти типы тождественны;
- оба типа являются целыми;
- оба типа являются вещественными;
- один из типов является целым, а другой – вещественным;
- один из типов является диапазоном другого;
- оба типа являются диапазонами одного базового типа;
- один из типов является строковым, а другой – строковым или символьным.

Говорят, что типы T1 и T2 совместимы по присваиванию, если программной конструкции типа T1 может быть присвоено значение программной конструкции типа T2. Совместимость по присваиванию определяется следующими правилами:

- типы T1 и T2 тождественны (исключение составляют файловые и опирающиеся на файловый типы);
- типы T1 и T2 являются совместимыми (в выражениях) порядковыми типами, причем диапазон значений типа T2 входит в диапазон значений типа T1;
- типы T1 и T2 являются вещественными типами, причем диапазон значений T2 входит в диапазон значений T1;

— тип T1 – вещественный, тип T2 – целый;
— типы T1 и T2 являются строковыми типами;
— тип T1 – строковый, а тип T2 – символьный.

Есть еще ряд правил совместимости (как в выражениях, так и по присваиванию), касающихся конкретных пользовательских типов. Такие правила будут рассматриваться по мере изучения этих типов.

Вместе с тем, иногда возникает необходимость рассматривать значение одного типа как если бы оно было другого типа. Такая постановка вопроса вполне корректна: любые данные хранятся в памяти ЭВМ в виде последовательности нулей и единиц, которая может быть интерпретирована, вообще говоря, различными способами. В Object Pascal существует две возможности приведения типов: для переменных и для значений. Синтаксически приведение типов выглядит одинаково как для переменной, так и для значения:

```
<идентификатор типа>(<выражение>);
```

однако существуют определенные различия в применении. В случае приведения типов для значений, как тип приводимого выражения, так и тип, к которому оно приводится, должны быть либо оба порядковыми, либо указательными. Далее, такая конструкция не может быть использована в левой части оператора присваивания и не может быть частью квалифицируемого идентификатора.

Пример:

```
    var I:integer;
    ...
    I := Integer('A');
...
```

В приведенном примере переменная I получит значение 65, если была введена латинская буква A, или 192, если буква A была русской.

Когда приводится тип для переменной, действуют иные правила. Можно приводить переменную любого типа к любому другому типу, если их размеры совпадают:

```
var
    s : Single; st : string[3];
    ...
    st := 'AAA'
    s := Single(st);
...
```

Исключение составляют взаимные преобразования между вещественными и целыми типами. Нельзя написать ни `s:=Single(i);` ни `i:=Integer(s)`. Однако первый из этих операторов присваивания заме-

няется просто `s := i`, поскольку типы Single и Integer совместимы по присваиванию, а вместо второго следует использовать либо `i:=Round(s)`, либо `i:=Trunc(s)`. Функция Round возвращает значение, округленное в соответствии с обычными математическими правилами (т.е. 5,3 будет округляться до 5, а 5,7 – до 6), а Trunc просто отбрасывает дробную часть числа вне зависимости от ее величины (результатом преобразования как числа 5,3, так и числа 5,7 будет 5).

Конструкция приведения типа для переменной, в отличие от конструкции приведения типа для значения, может быть частью квалифицируемого идентификатора и может находиться не только в правой, но и в левой части оператора присваивания, например:

```
var
    ch : Char;
    ...
    Byte(ch) := 90;
...
```

В результате переменная ch получит значение 'Z' (код ANSI этого символа 90).

Практикум

Пример 2.1. Исследование типов

Разработаем приложение, которое позволит нам изучить порядковые типы – как стандартные, так и определяемые пользователем.

Поместите на форму три кнопки и один компонент Memo. Замените стандартные заголовки кнопок: Button1 на «Изменить курсор», Button2 на «Тест стандартного типа», а Button3 – на «Тест пользовательского типа».

☒ ***Заголовки и имена.*** *Когда Вы работали с первыми двумя проектами, то, наверное, обратили внимание, что, несмотря на изменение заголовка, в программе кнопки и метки по-прежнему именовались Button1, Label1 и т.п. Название объекта в программе (в отличие от его заголовка) определяется не свойством Caption, а свойством Name. Если на форме размещены одна или две кнопки, их свойства Name можно оставить без изменений. В противном случае стоит задуматься над тем, чтобы дать кнопкам, меткам и другим объектам в проекте осмысленные названия. Как обычно, можно использовать любые допустимые идентификаторы (в нашем случае, например, ChangeCurBtn, TestStTypeBtn и TestUsTypeBtn).*

Программист волен изменить имена программных объектов в любой момент. Даже если какие-то обработчики событий уже созданы, Delphi автоматически заменит их названия. Но все строки, написанные вручную, исправлять придется также вручную.

Компонент Memo находится на странице Standard (перед компонентом Button). Этот компонент фактически поддерживает функциональные возможности простейшего текстового редактора (как NotePad, например). Сейчас для нас важно знать, что он может хранить строки символов и предоставляет доступ к этим строкам по индексу. Такие возможности обеспечивает его свойство Lines. Первоначально в этом свойстве хранится только одна строка: имя компонента: (Memo1 в нашем случае). Активизация свойства Lines приведет к тому, что в правой части страницы свойств Инспектора Объектов рядом со словом TStrings (тип свойства Lines) появится кнопка с многоточием. Одинарный щелчок по этой кнопке приведет к запуску редактора списка строк. Вы можете удалить имя компонента, а также дописать свои строки. Все сделанные изменения вступят в силу сразу после нажатия «ОК». Кнопка «Code Editor» (переход в Редактор Кода) предназначена для тех, кому покажется тесно в рамках окна редактора списка строк. Нажатие на эту кнопку приведет к тому, что в Редакторе Кода добавится еще одна страница с названием Form1.Memo1.Lines, заменяющая редактор списка строк (он при этом исчезнет с экрана).

Строки можно добавлять и во время выполнения программы, используя процедуру Add:

```
Memo1.Lines.Add(s);
```

где s – строка.

Компонент Memo, как и большинство компонентов, обладает свойствами Height и Width, Left и Top, управляющими размерами и расположением объекта. Кроме этого, у него имеется свойство Align (выровнять), позволяющее согласовать размеры Memo с размерами формы. Установка этого свойства, например, в alTop, приведет к тому, что компонент Memo будет размещен вдоль всего верхнего края формы (высота его не изменится). Свойство Alignment отвечает за выравнивание текста внутри компонента Memo и может принимать три значения, соответствующие выравниванию по левому краю, правому краю и по центру.

Компонент Memo может содержать информации больше, чем показывать в данный момент. Во время выполнения Вы можете «прокручивать» окно Memo с помощью клавиш со стрелками. Можно также установить полосы прокрутки, какие обычно имеют Windows – программы. Для этого следует изменить свойство ScrollBars. По умолчанию его значение ssNone, что означает отсутствие полос прокрутки. Вы можете установить это свойство в ssHorizontal, ssVertical или ssBoth (соответственно, появляются горизонтальная, вертикальная или обе полосы прокрутки).

Как и у большинства текстовых редакторов, принятый по умолчанию вид курсора для компонента Memo – вертикальная черта. Однако программист может легко изменить его, просто изменив свойство Cursor (свойство Cursor есть у многих компонентов, в том числе и у формы, и у

кнопки). Более того, возможно использование других курсоров, не определенных в Delphi.

Наконец, еще одно важное свойство – Font (шрифт). Щелчок на кнопке с многоточием в правой части строки Font Инспектора Объектов приведет к появлению стандартного диалогового окна «Выбор шрифта», знакомого Вам по другим приложениям Windows.

Список свойств компонента Memo, конечно, не исчерпывается приведенными выше. Назначение других свойств Вы можете выяснить с помощью Справочной системы Delphi.

Создайте для каждой из кнопок обработчики события OnClick. Щелчок на кнопке «Курсор» должен приводить к изменению внешнего вида курсора при позиционировании мышки над компонентом Memo. Если Вы обратитесь к справочной системе, то выясните, что тип TCursor определен как интервальный тип, допускающий значения от –32768 до 32767. Реально в Delphi предопределены стандартные курсоры (их имена содержатся в выпадающем списке, который Вы можете получить, выбирая свойство Cursor в Инспекторе Объектов), которым соответствуют значения от 0 (crDefault) до –21. При всех остальных значениях используется изображение курсора по умолчанию, если программистом не определено иначе. Поскольку TCursor является порядковым типом, для изменения курсора можно использовать функцию Pred (или Succ – в зависимости от начального значения):

Листинг 2.1a

```
procedure TForm1.ChangeCurBtn(Sender: TObject);
begin
    Memo1.Cursor := pred(Memo1.Cursor);
end;
```

Было бы хорошо предусмотреть возможность возврата к первоначальному виду курсора. Так, например, можно создать обработчик события – щелчка левой клавишей мыши на форме, в результате которого курсор компонента Memo получает значение, принятое по умолчанию:

Листинг 2.1b

```
procedure TForm1.FormClick(Sender: TObject);
begin
    Memo1.Cursor := crDefault;
end;
```

Конечно, существует множество способов реализации подобного поведения. Быть может, Вам покажется лучшим решением поместить на форму

еще одну кнопку – «Сброс», и устанавливать значение курсора crDefault именно при щелчке на ней.

Сейчас уже можно посмотреть первые результаты Вашей работы. Сохраните проект и запустите его на выполнение (предлагаемые имена: p2_1_test.pas для файла и pr2_1.dpr для проекта).

✓ ***Примечание.*** *Следует всегда сохранять проекты перед запуском на выполнение. Конечно, Вы можете доверить это Delphi. Выберите «Tools»|«Environment Options», и на странице «Preferences» появившегося диалогового окна отметьте флажок «Editor files» в группе «Autosave Options». Второй флажок «Desktop» следует отмечать при работе над большим проектом, чтобы он загружался вместе с Delphi.*

Теперь займемся второй кнопкой «Тест типа». Щелчок на этой кнопке должен приводить к выводу в компоненте Memo строк по образцу:

```
Тест типа ShortInt
High(ShortInt) = ...
Low(ShortInt) = ...
Значение переменной x типа ShortInt ...
После применения Inc получено ...
После применения Succ получено ...
```

Разумеется, вместо многоточий в Memo должны появиться реальные значения. Для этого Вам придется объявить переменную типа ShortInt. Самый простой путь – дописать объявление Вашей переменной в уже существующий раздел объявления переменных, рядом с объявлением переменной Form1 (в интерфейсном разделе). Однако существует правило: не стоит без большой необходимости объявлять глобальные переменные. Переменная должна появляться только там, где она используется. В нашем случае нужно объявить ее в рамках раздела реализации, в методе, который отвечает за появление строк в компоненте Memo:

Листинг 2.1с

```
procedure TForm1.TestStTypeBtnClick(Sender: TObject);
var x: ShortInt;
begin
    x := 126;
    Memo1.Lines.Add('Тест типа: ShortInt');
    Memo1.Lines.Add('High(ShortInt)='
                    +IntToStr(High(ShortInt)));
    // функция IntToStr преобразует целый аргумент
    // в значение строкового типа
    // верхняя граница исследуемого типа
```

```
    Memo1.Lines.Add('Low(ShortInt)='
                    +IntToStr(Low(ShortInt)));
    // нижняя граница исследуемого типа
    Memo1.Lines.Add('Значение переменной x типа ShortInt'
                    + IntToStr(x));
    Inc(x);
    Memo1.Lines.Add('После применения Inc получено: '+
                    IntToStr(x));
    x := Succ(x);
    Memo1.Lines.Add('После применения Succ получено: '+
                    IntToStr(x));
end;
```

В качестве начального значения переменной x выбрано 126. Это число всего на единицу отстоит от верхней границы типа ShortInt (127). К значению x последовательно применяется сначала процедура Inc, увеличивающая свой параметр на единицу, а затем – функция Succ, возвращающая элемент порядкового типа, следующий за данным. В нашем случае после применения процедуры Inc значение переменной x станет равным 127, а функция Succ возвратит значение –128. Таким образом, последний элемент стандартного порядкового типа считается предшествующим первому (при желании можно протестировать любой другой стандартный порядковый тип). В отличие от стандартных, порядковые типы, определенные пользователем, ведут себя иначе: автоматического перехода на первый элемент не происходит. Чтобы убедиться в этом, используем в качестве тестируемого типа тип week, определенный в параграфе «Перечислимый и интервальный типы». Поместить объявление этого типа следует в начало раздела реализации:

Листинг 2.1d

```
implementation

{$R *.DFM}

type
    week = (sun, mon, tue, wed, thi, fri, sat);
...
```

Вообще говоря, объявления типов часто располагаются в интерфейсном разделе. Однако, поскольку мы точно знаем, что никакому другому модулю в этом проекте (просто потому, что в проекте содержится всего один модуль) не понадобится использовать данные типа week, объявить этот тип целесообразно именно в разделе реализации.

В обработчике события OnClick для кнопки TestUsTypeBtn нужно будет объявить переменную типа week, и практически в точности воспроизвести код обработчика события TForm1.TestStTypeBtnClick.

✓ ***Примечание.*** *Конечно, Вы можете просто перекопировать код процедуры TForm1.TestStTypeBtnClick и исправить его надлежащим образом. Но делать это следует только после того, как Delphi выполнит свою часть работы и создаст «заготовку» обработчика события для третьей кнопки (см. также параграф «Файлы форм» Главы 1).*

Единственная проблема состоит в том, что не существует стандартной функции преобразования данных типа week в строковый тип (подобной IntToStr), и ее придется написать самостоятельно. Описание функции должно располагаться до ее первого вызова, поэтому поместим его в начало раздела реализации, следом за объявлением типа:

Листинг 2.1е

```
implementation

{$R *.DFM}

type
     week = (sun, mon, tue, wed, thi, fri, sat);

     function WeekToStr(day: week): string;
     begin
          case a of
               sun: result :='воскресенье';
               mon: result :='понедельник';
               tue: result :='вторник';
               wed: result :='среда';
               thi: result :='четверг';
               fri: result :='пятница';
               sat: result :='суббота';
          else result := 'нет такого дня недели';
          end;
     end;
...
```

По аналогии с предопределенными функциями преобразования наша функция называется WeekToStr, принимает в качестве аргумента один параметр типа week и возвращает строку. Возвращаемое значение присваивается специальной переменной result, которая считается определенной в любой функции и имеет тот же самый тип, что и тип результата функции

(можно считать, что result – автоматически объявляемая локальная переменная). Теперь осталось отредактировать обработчик события TestUsTypeBtnClick:

✓ ***Примечание.*** *В Pascal возвращаемое значение должно было обязательно присваиваться имени функции. В Delphi можно поступить также, а можно использовать переменную result.*

Листинг 2.1f

```
procedure TForm1.TestUsTypeBtnClick(Sender: TObject);
var w : week;
begin
  w := fri;
  Memo1.Lines.Add('Тест типа: week');
  Memo1.Lines.Add('High(week) = ' +
                        WeekToStr(High(week)));
  Memo1.Lines.Add('Low(week) = ' +
                        WeekToStr(Low(week)));
  Memo1.Lines.Add('Значение переменной x типа Week '
                        + WeekToStr(w));
  w := Succ(w);
  Memo1.Lines.Add('После применения Succ получено: '
                        + WeekToStr(w));
  Inc(w);
  Memo1.Lines.Add('После применения Inc получено: ' +
                        WeekToStr(w));
  Memo1.Lines.Add('Контроль значения Ord(w) = ' +
                        IntToStr(Ord(w));
end;
```

Сохраните и запустите проект на выполнение. При нажатии на кнопку «Тест пользовательского типа» компонент Memo будет заполнен следующими строками:

```
Тест типа: week
High(week)=суббота
Low(week)=воскресенье
Значение переменной x типа Week пятница
После применения Succ получено: суббота
После применения Inc получено: нет такого дня недели
Контроль значения Ord(w) = 7
```

Последняя выводимая строка показывает порядковый номер значения в объявлении типа. Поскольку значение с номером 7 в объявлении ти-

па отсутствует, то при преобразовании этого значения в строку выбирается ветвь **else** оператора варианта **case**. Попробуйте заключить эту строку в фигурные скобки и откомпилировать проект заново. Вместо слов «нет такого дня недели» не будет выводиться ничего.

✓ ***Примечание.*** *Если откомпилировать проект с включенной проверкой границ ({$R+}), то нажатие на кнопку «Тест пользовательского типа» приведет к ошибке времени выполнения «Range check error», в то время как «Тест стандартного типа» будет пройден без осложнений.*

Пример 2.2. Перевод целых чисел из десятичной системы счисления в шестнадцатеричную

Существует стандартный алгоритм для перевода чисел из одной системы счисления в другую. Однако перевод числа из десятичной системы счисления в шестнадцатеричную можно осуществить средствами Delphi, не проводя никаких вычислений, а просто используя нужный формат для вывода данных в виде строки.

Для работы со строками существует довольно много стандартных функций. Одна группа функций преобразует строковые значения в значения других (как правило, числовых) типов. Другие, наоборот, позволяют преобразовать строку в число (одна из таких функций была использована в предыдущем примере). Есть функции, которые сравнивают строки, анализируют их на наличие определенных символов, меняют регистр. Кроме этого, существуют функции форматирования строк. Большинство таких функций объявлено в модуле SysUtils, который не нужно указывать в списке используемых модулей (он подключается автоматически к любому проекту).

☒ ***Format, StrToInt и Val.*** *Прежде чем приступить к написанию приложения, изучите с помощью оперативной справки функции Format и StrToInt, а также процедуру Val.*

Поместите на форму элемент редактирования (компонент Edit, страница Standard), одну кнопку и одну метку.

Компонент Edit используется для ввода (реже – вывода) различных значений. Его свойство Text (тип свойства – TCaption = **string**) может содержать какую-либо строку, требования к которой практически аналогичны требованиям к заголовку формы или кнопки. Введенная пользователем строка считывается из этого свойства, а затем при необходимости может быть преобразована, например, в целое число. Пока мы не будем обсуждать, как запретить ввод некорректных данных. Предположим, что Вы (как грамотный пользователь) не ошибаетесь.

Первоначально свойство Text содержит строку Edit1, которую можно сразу уничтожить. Вводить какое-либо новое значение необязательно.

Когда пользователь вводит или стирает символ в компоненте Edit, происходит событие OnChange, являющееся для Edit событием по умолчанию. Это означает, что создать обработчик события OnChange можно двойным щелчком левой клавишей мыши на соответствующем компоненте Edit, расположенном на форме. Точно также событие OnClick является событием по умолчанию для кнопки, и двойной щелчок левой клавишей мыши на кнопке во время разработки приведет к созданию обработчика этого события. Разумеется, каждый компонент, имеет только одно событие по умолчанию (конечно, если он вообще имеет события). Обработчики же остальных событий создаются обычным образом с помощью Инспектора Объектов (как было описано ранее).

Измените свойства Name и Caption Ваших компонентов следующим образом:

Компонент	Name	Caption
Form1	IntToHexForm	Системы счисления 10 --> 16
Button1	ClearBtn	Очистить
Label1	HexLabel	<пустая строка>
Edit1	IntEdit	

и очистите свойство Text компонента IntEdit.

Кнопка ClearBtn должна очищать свойство Text компонента IntEdit во время выполнения. Следовательно, в обработчике события OnClick этой кнопки нужно написать оператор

```
IntEdit.Text := '';
```

который присваивает свойству Text компонента IntEdit пустую строку. Код, который отвечает за считывание введенного десятичного числа и преобразование его в шестнадцатеричное, будет располагаться в обработчике события OnChange элемента редактирования.

Первое, что следует сделать – считать введенное пользователем значение из свойства Text. Используем для этого процедуру Val, преобразующую строку в число. Ее заголовок выглядит следующим образом:

```
procedure Val(S; var V; var Code: Integer);
```

Первый параметр – значение строкового типа, второй – переменная числового (целого или вещественного) типа, в которую будет записываться результат преобразования, третий – переменная целого типа, принимающая значение 0, если строка может быть интерпретирована как число, или номер первого символа, который не может входить в число (т.е., не являющийся цифрой или десятичной точкой).

В качестве первого параметра мы можем использовать непосредственно значение IntEdit.Text, но для второго и третьего параметров придется ввести дополнительные переменные. Поскольку эти переменные нигде больше не понадобятся, достаточно записать их в разделе объявления переменных обработчика события OnChange:

Листинг 2.2a

```
procedure IntToHexForm.IntEditChange(Sender: TObject);
var
     decimal : Integer;
     // число в десятичной системе счисления
     i: Integer; // номер ошибочного символа
begin
     Val(IntEdit.Text,decimal,i);
     // так строка из Edit превращается в целое число,
     // если это возможно
     ...
end;
```

Полученное десятичное число должно быть преобразовано в шестнадцатеричное. Это довольно просто сделать с помощью функции Format. Справочная система Delphi подробно описывает возможности этой функции. Для нас же сейчас важно, что спецификатор формата %x преобразует аргумент, являющийся десятичным числом к шестнадцатеричному виду. Поэтому окончательно в обработчике события onChange будут записаны следующие операторы:

Листинг 2.2b

```
...
begin
     Val(IntEdit.Text,decimal,i);
     HexLabel.Caption := Format('%x', [decimal]);
end;
```

Сохраните Ваш проект (предлагаемые имена модуля: ih_trans.pas, проекта IntToHex.dpr) и запустите его на выполнение. Попробуйте ввести какое-нибудь целое число. Преобразование к шестнадцатеричному виду будет происходить каждый раз при вводе новой цифры.

✓ ***Примечание.*** *Можно дополнить проект простейшей реакцией на ошибку:*

```
...
begin
     Val(IntEdit.Text,decimal,i);
     if (i<>0) then HexLabel.Caption := 'Ошибка'
```

```
    else HexLabel.Caption := Format('%x', [decimal]);
end;
```

При работе в «интерактивном» режиме такой путь вполне приемлем.

Выше мы упоминали функцию StrToInt, которая, как можно заключить из названия, преобразует строку в целое число. Попробуйте использовать эту функцию вместо процедуры Val. При этом нужно обратить внимание на следующие моменты. Во-первых, придется изменить процедуру очистки элемента редактирования, поскольку пустая строка не может быть преобразована в целое число с точки зрения функции StrToInt. Решением проблемы может быть запись в элемент редактирования 0. Во-вторых, если Вы введете не цифру, это приведет к ошибке. Ваша программа при этом не «зависнет», и Вы сможете продолжить с ней работу. В дальнейшем мы обязательно обсудим, как должна реагировать программа на ошибочно вводимые значения или какие-то другие некорректные действия со стороны пользователя.

Знаки операций и выражения

Выражения фактически являются «кирпичиками», из которых построены объявления и операторы. Пример простейших выражений – переменные и константы. Более сложные выражения составляются из более простых с помощью знаков операций, а также с использованием таких конструкций языка, как вызовы функций, конструкторы множеств, индексы и приведение типов. Объекты операции называют операндами. В зависимости от количества операндов операции подразделяются на унарные и бинарные. К унарным операциям в Object Pascal относятся операции @ (взятия адреса), **not** (отрицания), а также ^ (разыменования указателя). Знаки операций + и – могут обозначать как унарную операцию (сохранения или изменения знака числа), так и бинарную. Все остальные операции – бинарные. За исключением операции разыменования знаки унарных операций располагаются перед операндом; знаки бинарных операций располагаются между операндами.

✓ ***Примечание.*** *В некоторых других языках программирования существуют знаки операций, требующие трех операндов.*

При вычислении выражений последовательность выполнения операций определяется их приоритетом. В документации Object Pascal приводится следующая таблица приоритетов (знак операции разыменования указателя в нее не входит):

Знак операции	Приоритет
@, **not**	Первый (высший)
*, /, **div**, **mod**, **and**, **shl**, **shr**, **as**	Второй
+, –, **or**, **xor**	Третий
=, <>, <, >, <=, >=, **in**, **is**	Четвертый (низший)

Как можно видеть (см., например, раздел formal grammar оперативной справки), наивысший приоритет имеют унарные операции, второй – бинарные операции типа умножения, за ними следуют бинарные операции типа сложения, а самый низкий приоритет имеют бинарные операции типа отношения.

✓ ***Примечание.*** *На взгляд автора, было бы вполне правомерно включить в группу операций наивысшего приоритета все унарные операции: и операцию разыменования, и унарный плюс, и унарный минус. Однако будем следовать заявленному приоритету документации Delphi.*

В случае равенства приоритетов операции выполняются слева направо – в порядке записи. Изменить естественный порядок выполнения операций можно с помощью круглых скобок. Подвыражения, заключенные в круглые скобки, вычисляются в первую очередь. При этом, если уровней вложенности несколько, вычисление начинается с «самых внутренних» подвыражений. Следует помнить, что иногда использование круглых скобок там, где компилятору это не требуется, может сделать код более легким для восприятия.

Операции можно классифицировать по характеру осуществляемых действий. Как мы уже говорили, множество операций над данными определяется их типом. Вместе с тем, в выражениях могут использоваться операнды различных типов (о такой возможности обычно говорят как о совместимости типов в выражениях). Наконец, один и тот же знак операции может иметь различный смысл в зависимости от типа операндов. По этим причинам классификация получается не слишком стройной:

— арифметические операции;
— логические (Boolean) операции;
— побитовые (bitwise) операции;
— операции отношения;
— операции над строками;
— операции над множествами;
— операции над указателями;
— операции над классами;
— операция взятия адреса (@).

В этом параграфе мы рассмотрим первые четыре вида операций (в приложении к уже изученным типам данных), а также операции над строками. Остальные виды операций мы будем изучать совместно с соответствующими типами.

К арифметическим операциям относятся сохранение знака (унарный плюс +), изменение знака (унарный минус –), сложение (+), вычитание (–), умножение (*), деление (/), а также целочисленное деление (div) и взятие остатка от целочисленного деления (mod). В качестве операндов первых шести операций могут выступать величины любого целого или вещественного типов. Тип результата, как правило, определяется типами операндов: если операнд (для унарной операции) или оба операнда (для бинарной операции) – целого типа, то и результат будет целым. В противном случае (один или оба операнда – вещественные) результат будет вещественным. Исключение составляет операция деления: вне зависимости от типа операндов результат будет иметь вещественный тип (иногда операцию деления называют операцией вещественного деления).

✓ ***Примечание.*** *Более точно, результат арифметической операции будет иметь тип Extended, если хотя бы один из операндов – вещественный, тип Int64, если все операнды – целого типа, и хотя бы один из них имеет тип Int64, и Integer – в противном случае; для операции деления результат всегда будет иметь тип Extended.*

Названия «целочисленное деление» и «взятие остатка от целочисленного деления» говорят сами за себя. Операндами в таких операциях могут быть только величины целых типов, целочисленным будет и результат (оговорка относительно Int64 имеет силу и в этом случае).

✓ ***Примечание.*** *Результатом операции целочисленного деления x* **div** *y, где x и y – целые числа будет целое число, ближайшее к числу x/y со стороны нуля. Результат операции взятия остатка от деления x* **mod** *y можно представить с помощью операции div следующим образом: x – (x* **div** *y) * y (возможна обратная замена)*

Примеры:

Сохранение знака	+X	Изменение знака	–8
Сложение	X + Y	Вычитание	Z – A
Умножение	I * R	Деление	U / I
Целочисленное деление	X div 2	Взятие остатка от деления	X mod 2

Логических операций четыре: одна унарная – логическое отрицание (**not**), и три бинарных – конъюнкция (логическое «И»; **and**), дизъюнкция (логическое «ИЛИ»; **or**) и разделительная дизъюнкция (логическое «Исключающее ИЛИ»; **xor**). Их операнды могут иметь любой логический тип, результат же всегда будет типа Boolean. Компилятор Delphi позволяет проводить вычисления логических выражений по полной или по сокращенной схеме. По умолчанию принята сокращенная схема. Это означает, что выражения, содержащие **and** и **or**, вычисляются только в той части, которая оказывает определяющее влияние на результат. Например, при вычислении выражения A **or** B компилятор не будет вычислять значение B, если значение A оказалось истинным: согласно законам алгебры логики, выражение A **or** B будет истинным, если хотя бы одно из подвыражений будет истинным. Аналогично, при вычислении выражения C **and** D компилятор ограничится вычислением C, если оно окажется ложным.

Полная схема вычислений предполагает вычисление каждой операции конъюнкции или дизъюнкции в выражении, даже если результат уже ясен. Применение сокращенной схемы вычислений уменьшает время исполнения программы, однако, полная схема вычислений может оказаться необходимой при использовании в качестве операндов функций с так называемым побочным эффектом. Более подробное обсуждение этого вопроса и содержательные примеры будут приведены при рассмотрении таких функций. Пока же отметим лишь, что использование полной схемы вычислений логических выражений можно задать, отметив опцию «Complete Boolean Evaluation» на странице «Compiler» диалогового окна, вызываемого командой меню «Project» | «Options». Альтернативой для части кода служит директива компилятора {$B+}.

Примеры:

Значение операнда A	Значение операнда B	Операция	Запись операции	Результат
True	–	Отрицание	**not** A	False
True	False	Конъюнкция	A **and** B	False
True	False	Дизъюнкция	A **or** B	True
True	False	Разделительная дизъюнкция	A **xor** B	True

Побитовые операции выполняются над двоичным представлением целых величин. Среди этих операций есть одна унарная – операция отрицания (**not**), и пять бинарных: конъюнкция (**and**), дизъюнкция (**or**), разделительная дизъюнкция (**xor**), а также операции побитового сдвига влево (**shl**; shift left) и операция побитового сдвига вправо (**shr**; shift right). Операнды побитовых операций могут иметь любой целый тип, а тип результата определяется следующим образом. Результат операции отрицания имеет тот же тип, что и единственный операнд. Тип результата любой из операций побитового сдвига совпадает с типом первого операнда. Конъюнкция, дизъюнкция и разделительная дизъюнкция приводят к результату целого типа (одного из предопределенных), диапазон значений которого является наименьшим вмещающим в себя значения обоих операндов.

✓ ***Примечание.*** *В действительности, если не включена проверка границ ({$R+}), присваивание переменной любого целого типа значения переменной любого другого целого типа считается допустимым:*

```
var a: ShortInt; b: Byte;
begin
    a:= –8;
    b:= a;
...
```

В этом случае будет осуществлено автоматическое приведение типов, и b получит значение 248 (в то же время явная запись b:= –8 приведет к ошибке компиляции). Впрочем, даже при включенной проверке границ Delphi не станет беспокоить Вас без повода: если переменной a в приведенном выше примере присваивается положительное значение, все будет работать. Более того, нормально компилируется и исполняется следующий код:

```
var a, c: ShortInt; b: Byte;
begin
    a:= –8;
    b:= 254;
    {$R+}
    c:= a or b;
    {$R–}
...
```

Однако если переменная c будет объявлена как Byte, Вы получите сообщение об ошибке «Range Check Error».

Разумеется, все вышесказанное справедливо и для любых других стандартных типов, принадлежащих одной группе (например, вещественных).

Примеры:

Рассмотрим применение побитовых операций к двум переменным A и B типа Byte.

	Десятичное представление	Двоичное представление
Значение операнда A	13	00001101
Значение операнда B	3	00000011
Результат операции		
not A	242	11110010
A and B	1	00000001
A or B	15	00001111
A xor B	14	00001110
A shl B	104	01101000
A shr B	1	00000001

Заметим, что вычислить результат операции побитового сдвига влево можно умножением первого операнда A на 2^B, а результат операции побитового сдвига вправо – путем целочисленного деления первого операнда на 2^B.

Следующая группа операций, которые мы обсудим, – это операции отношения. К ним относятся: операция равенства (=), операция неравенства (<>), а также операции сравнения «меньше» (<), «меньше или равно» (<=), «больше» (>), «больше или равно» (>=). Все шесть операций отношения являются бинарными. Их операндами могут быть величины любого простого или строкового типа, а результат операции отношения всегда имеет тип Boolean.

✓ ***Примечание.*** *Отметим, что список типов операндов неполон. Например (в силу общности понятий эквивалентности и неэквивалентности), операции равенства и неравенства могут применяться к операндам ряда других типов. По мере изучения новых типов мы будем указывать, какие операции определены для значений этих типов.*

Действие операций отношения над операндами простых типов соответствует их обычной математической интерпретации. Единственное требование – операнды должны быть совместимых типов, «смешивать» разрешается только целые и вещественные значения. Если же операнды строкового типа (величины символьных типов трактуются как строки единичной длины), они сравниваются посимвольно, в соответствии с порядком символов ANSI – кода.

Примеры:

Равенство	I = 10
Неравенство	X <> Y
Меньше	c < 'v'
Меньше или равно	a <= 5.87
Больше	st > 'try'
Больше или равно	intNum >= realNum

Группа операций над строками состоит из только что рассмотренных операций отношения (для случая операндов строкового типа), а также операции конкатенации (+). Операндами операции конкатенации могут быть величины любых строковых типов. Существует только одно ограничение: если один из операндов имеет тип WideChar, то другой должен быть длинной строкой. Результатом операции конкатенации (сцепления) является строка, состоящая из символов обоих операндов, причем символы второго операнда помещаются после символов первого. Полученная строка

совместима с любым строковым типом. Однако если оба операнда являются короткими строками (ShortString), то и результат будет короткой строкой (т.е. будет содержать не более 255 символов; все последующие символы будут отсечены).

Пример:

```
'При'+'мер'
```

Теперь приведем формальные правила, по которым строятся выражения.

```
<выражение> ::= <простое выражение>
          {[<операция типа отношения> <простое выражение>]}

<простое выражение> ::=
          [+|-] <терм> {[<операция типа сложения> <терм>]}

<терм> ::=
     <множитель> {[<операция типа умножения> <множитель>]}

<множитель> ::=
     <десигнат> [(<список выражений>)]|(<выражение>)|<число>|
     <строка> | not<множитель>|<выражение приведенного типа>|
     <конструктор множества> | nil | @ <десигнат>

<выражение приведенного типа> ::=
          <идентификатор типа> (<выражение>)

<список выражений> ::=
          <выражение> {, <выражение>}

<десигнат> ::=
          {<квалифицируемый идентификатор> [.<идентификатор> |
          '['<список выражений>']' | ^]}

<квалифицируемый идентификатор> ::=
          [<идентификатор модуля>.] <идентификатор>
```

Здесь приведено полное определение понятия «множитель», но поскольку мы пока рассмотрели не все типы данных (и виды операций), оставим последнюю строку этого определения без комментариев. Отметим лишь, что конструкторы множеств будут обсуждаться в главе 4, посвященной множествам, а операция взятия адреса и зарезервированное слово **nil** – при изучении указателей. Что касается десигната (designator, означаемое), сейчас вполне достаточно ограничиться представлением о нем как о переменной или константе.

В заключение этого параграфа – несколько слов о константных выражениях. Как мы уже говорили, константное выражение – это выражение, которое может быть вычислено компилятором до исполнения программы. В константное выражение могут входить числа, символьные строки, уже определенные истинные константы, значения перечислимых типов, а также специальные константы True, False и **nil** (кроме этого, в константном выражении могут применяться знаки операций, операторы приведения типов и конструкторы множеств). Не допускается использование в константном выражении переменных, указателей, а также функций, за исключением следующих предопределенных:

Abs	Exp	Length	Ord	Sqr
Addr	Frac	Ln	Pred	Sqrt
ArcTan	Hi	Lo	Round	Succ
Chr	High	Low	Sin	Swap
Cos	Int	Odd	SizeOf	Trunc

Константное выражение фактически своим видом определяет тип истинной константы. Так, если константное выражение представляет собой целое число, то его типом считается LongInt, но если его значение выходит за пределы диапазона этого типа, типом константы будет Int64. В случае вещественного значения типом константы будет Extended. Константа – символьная строка будет совместима с любым строковым типом. Если же длина этой строки равна 1 (т.е., это один символ), такая константа будет совместима и с любым символьным типом.

Классификация операторов

Операторы описывают действия, выполняемые программой. Любой оператор может иметь метку, которая отделена от него двоеточием. Друг от друга операторы отделяются точкой с запятой (;).

Все операторы Object Pascal подразделяются на две группы: простые и структурированные:

```
<оператор> ::=
     [<метка>:] <простой оператор> | <структурированный опера-
тор>
```

Простыми называют операторы, которые не содержат в себе других операторов.

```
<простой оператор> ::=
          <пустой оператор> | <оператор присваивания> |
          <оператор процедуры> | <оператор перехода> |
          <оператор наследования>
```

Пустой оператор не имеет специального обозначения. Он используется как вспомогательная синтаксическая конструкция. Например, запись «;;» может быть истолкована как пустой оператор, находящийся между двумя разделителями.

✓ ***Примечание.*** *В Object Pascal (как и в стандартном Pascal) необязательно ставить точку с запятой между оператором и зарезервированным словом end. Если же символ «;» поставлен, то считается, что после него находится пустой оператор.*

Оператор присваивания выглядит следующим образом:

```
<оператор присваивания> ::= <десигнат> := <выражение>
```

Пример:

```
var x: integer;
begin
     ...
     x := 5;
     ...
end.
```

В результате текущее значение программного объекта в левой части оператора присваивания заменяется новым значением, задаваемым выражением в правой части оператора присваивания (в левой части оператора присваивания обычно указывается переменная, типизованная константа или идентификатор функции). Новое значение программного объекта при этом становится текущим, а прежнее текущее значение утрачивается безвозвратно.

Оператор перехода передает управление оператору, имеющему соответствующую метку:

```
<оператор перехода> ::= goto <метка>
```

Пример:

```
label jump;
var x,y: integer;
begin
     ...
     x :=5;
     goto jump;
     y := 5; // этот оператор пропускается при выполнении
jump: y := x-2;
     ...
end.
```

Вообще говоря, использование оператора перехода не приветствуется стандартами структурного программирования. Считается (и не без оснований), что операторы **goto** делают программу более запутанной и сложной для понимания. В языке существуют различные конструкции, позволяющие в большинстве случаев обойтись без операторов перехода.

Приведем также определения операторов вызова процедуры и наследования:

```
<оператор процедуры> ::= <десигнат> [ (<список выражений>) ]

<оператор наследования> ::= inherited
```

Структурированные операторы содержат в себе другие операторы, причем порядок выполнения этих операторов не обязательно является последовательным. Структурированные операторы подразделяются на пять групп:

```
<структурированный оператор> ::=
          <составной оператор> | <условный оператор> |
          <оператор цикла> | <оператор присоединения> |
          <оператор обработки исключительной ситуации>
```

Составной оператор является конструкцией, вспомогательной с точки зрения синтаксиса. Этот оператор обычно применяют, когда синтаксис языка допускает использование только одного оператора, в то время как программная логика требует нескольких. Определение составного оператора было дано в Главе 1, однако мы повторим его ради полноты изложения:

```
<составной оператор> ::=
     begin
     <список операторов>
     end

<список операторов> ::= <оператор> {; <оператор>}
```

✓ ***Примечание.*** *Пару зарезервированных слов* **begin** *и* **end** *часто называют операторными скобками.*

Оператор присоединения подробно рассматривается в Главе 5 «Записи и работа с ними», оператор обработки исключительной ситуации также изучается совместно с понятием исключения.

Условных операторов в Pascal два: это условный оператор **if**, который обычно и называют условным оператором, и условный оператор **case**, для которого чаще используются названия «оператор выбора» или «оператор варианта».

```
<условный оператор> ::=
          <условный оператор if> | <условный оператор case>

<условный оператор if> ::=
          if <условие> then <оператор> [else <оператор>]

<условие> ::= <выражение>
```

При выполнении условного оператора сначала вычисляется условие – выражение, результат которого обязательно должен иметь тип Boolean. Если вычисления дают True в качестве значения выражения, будет выполнен оператор, указанный в ветви **then**, в противном случае (если значение выражения – False) выполняется оператор, указанный в ветви **else**. Затем управление передается оператору, следующему за условным оператором. Ветвь **else** может отсутствовать.

```
Пример:
var x, y: integer;
begin
     ...
     if x > 0 then y := x*x else y := -1;
     ...
end.
```

Иногда требуется использовать вложенные условные операторы. При этом действуют следующие правила. Зарезервированное слово **else** считается относящимся к ближайшему стоящему перед ним зарезервированному слову **if**, с которым еще не было

соотнесено ни одно зарезервированное слово **else**. Во избежание путаницы следует использовать операторные скобки.

Примеры:

```
var x, y: integer;
begin
     ...
     if x < -1 then
               if x > -5 then y:= x + 5 // выполняется при
5<x<-1
               else y:= x*(-2); // выполняется при x < -5
     . .
end.
```

```
var x, y: integer;
begin
     ...
     if x < -1 then begin
               if x > -5 then y:= x + 5 // выполняется при
5<x<-1
               end
     else y := x*(-2); // выполняется при x > -1
     ...
end.
```

Обратите внимание, что зарезервированное слово **else** в приведенных выше примерах записано на том же уровне, что и соответствующее ему **if**. В действительности местоположение **else** не играет никакой роли с точки зрения компилятора, но иерархический способ записи более удобен для восприятия.

Синтаксис условного оператора **case** следующий:

```
<условный оператор case> ::=
          case <селектор> of
          <вариант> [{;<вариант>}]
          [else <оператор> ] [;]
          end

<селектор> ::= <выражение>

<вариант> ::=
          <метка варианта> [{,<метка варианта>}] : <оператор>

<метка варианта> ::=
          <константное выражение> [.. <константное выражение> ]
```

При выполнении оператора операторе **case** сначала вычисляется выражение – селектор, результатом которого должно быть значение любого порядкового типа – как стандартного, так и определяемого пользователем. Если выражение – селектор принимает значение, совпадающее со значением какого-либо константного выражения из меток вариантов, выполняется оператор, указанный в этом варианте. Если среди меток вариантов не нашлось подходящего значения, будет выполняться ветвь **else** (если она

есть). Затем управление получает (т.е. выполняется) оператор, следующий за оператором **case**. Отметим, что константное выражение, присутствующее в одной из меток варианта, не может присутствовать более ни в какой другой метке варианта.

Примеры:

```
type T_MyNums = 0..9;
var  k: T_MyNums;
     s: string;
begin
     ...
     case k of
          1,3,5,7,9: s := 'Нечетное число';
          2,4,6,8:   s := 'Четное число';
     else
          s := 'Ноль';
     end;
     ...
end.
```

```
type T_MyNums = 0..9;
var  k: T_MyNums;
     s: string;
begin
     ...
     case k of
          3,6,9:     s := 'Это число нацело делится на 3';
          2,4,6,8:   s := 'Это число нацело делится на 2';
     {Компилятор сообщит об ошибке «Duplicate case label»,
     поскольку число 6 используется в двух метках вариантов}
     end;
     ..
end.
```

Для организации циклов Pascal предоставляет три конструкции: оператор цикла с предусловием (**while**), оператор цикла с постусловием (**repeat**) и оператор цикла с параметром (счетчиком) (**for**):

```
<оператор цикла> ::=
     <оператор while> | <оператор repeat> | <оператор for>
```

Оператор **while** является наиболее универсальным оператором, с помощью которого можно описать любое циклическое действие.

```
<оператор while> ::=
          while <условие> do <оператор>

<условие> ::= <выражение>
```

Результатом выражения (его часто называют выражением, управляющим циклом) может быть только значение типа Boolean. Если значение этого выражения истинно (т.е. равно True), будет выполнен оператор, указанный после зарезервированного

слова do (его обычно называют телом цикла). Затем управление будет возвращено оператору while и выражение будет вычислено вновь. Как только значение выражения станет ложным (False), управление будет передано оператору, следующему за оператором while. Практическое следствие состоит в том, что в теле цикла обязательно должны происходить какие-то действия, ведущие к изменению выражения. В противном случае цикл никогда не прервется.

```
Пример:
const n = 13;
var i, f: integer;
begin
     ...
     {Вычисление 13!}
     f := 1; {0!}
     i := 1;
     while i <= n do begin
          f := f*i;
          i := i+1;
          (Изменение i влияет на результат выражения,
          управляющего циклом)
     end;
     ...
end.
```

Оператор **repeat** может оказаться весьма полезным, когда некоторые действия необходимо выполнить хотя бы один раз.

```
<оператор repeat> ::=
          repeat <оператор> [{;<оператор>}] until <условие>

<условие> ::= <выражение>
```

Как и в случае оператора **while**, результат выражения должен иметь тип Boolean. Порядок выполнения оператора repeat следующий: сначала выполняются операторы тела цикла (т.е. операторы, заключенные между зарезервированными словами **repeat** и **until**), а затем вычисляется выражение. Если значение выражения ложное (False), управление возвращается оператору **repeat**, в противном случае управление передается оператору, следующему за оператором **repeat**.

✓ ***Примечание.*** *В теле цикла оператора **repeat** (в отличие от других операторов цикла) могут быть записаны несколько операторов. Использование операторных скобок необязательно.*

```
Пример:
const n = 13;
var i, f: integer;
begin
     ...
     (Вычисление 13!)
     f := 1; {0!}
     i := 1;
```

```
    repeat
        f := f*i;
        i := i+1;
        (Изменение i влияет на результат выражения,
        управляющего циклом)
    until i > n;
    end;
    ...
end.
```

Оператор **for** можно использовать, если число итераций заранее известно. Такое требование, конечно, сужает сферу применения оператора **for**. Тем не менее, в силу своей наглядности он применяется практически всегда, когда это возможно. В Pascal допускается две формы оператора: с возрастающим счетчиком (для краткости мы будем говорить о форме **to**) и с убывающим счетчиком (форма **downto**).

```
<оператор for> ::=
        for  := <параметр цикла>
            <выражение> to <выражение> do <оператор> |
        for <параметр цикла> :=
            <выражение> downto <выражение> do <оператор>

<параметр цикла> ::= <квалифицируемый идентификатор>
```

Параметр цикла (его также называют счетчиком) должен быть (локальной) переменной любого порядкового типа. Выражения должны давать результат того же самого или совместимого типа. Выражение, стоящее перед зарезервированным словом **to** (**downto**), называют начальным выражением, а выражение, расположенное после зарезервированного слова **to** (**downto**), называют конечным выражением.

Выполнение оператора **for** происходит следующим образом. Вначале вычисляются начальное и конечное выражения, а счетчику присваивается значение начального выражения. Чтобы тело цикла было исполнено хотя бы раз, для оператора в форме **to** начальное выражение не должно превосходить конечное выражение, а для оператора в форме **downto** начальное выражение должно быть не меньшим, чем конечное выражение. Если это условие соблюдено, то после выполнения тела цикла счетчик автоматически увеличивается или уменьшается на единицу (в смысле рассматриваемого порядкового типа). Последняя итерация происходит при значении счетчика, равном значению конечного выражения. По завершении цикла **for** значение счетчика считается неопределенным.

Примеры:

```
const n = 13;
var i, f: integer;
begin
    ...
    {Вычисление 13!}
    f := 1; {0!}
    for i := 1 to n do f := f*i;
    ...
end.
```

```
const n = 13;
var i, f: integer;
begin
     ...
     {Вычисление 13!}
     f := 1; {в случае цикла for в форме downto нельзя
     провести прямую аналогию с 0!}
     for i := n downto 1 do f := f*i;
     ...
end.
```

✓ ***Примечание.*** *В приведенных примерах в качестве счетчика использовалась переменная целого типа, значение которой автоматически увеличивалось (или уменьшалось – в зависимости от формы цикла) на единицу. Для порядковых типов, не являющихся целыми, происходил бы переход к следующему (или предыдущему) значению. Изменить величину шага в Pascal невозможно; некоторые другие языки это позволяют.*

Важным отличием оператора **for** от операторов цикла с предусловием и с постусловием является то, что начальное и конечное выражение вычисляются только один раз – при входе в цикл. Даже если в рамках цикла значения операндов, входящих в состав этих выражений, меняются, это никаким образом не может повлиять на ход выполнения цикла. Присвоение какого-либо значения параметру запрещено.

Пример:

```
var i, k, n: integer;
begin
     ...
     k := 2;
     n := 0;
     for i:= 1 to k do
          begin
               n := n + 2*i;
               k := k+1;
               {а попытка изменить переменную цикла, например,
               i := i+1;
               приведет к ошибке компиляции «Assignment to
               FOR-Loop variable 'i'»}
          end;
     {По завершении цикла k будет иметь значение 4, а n 6}
     ...
end.
```

Так, приведенный выше пример будет нормально откомпилирован и исполнен. Тем не менее, этот код вряд ли можно назвать легко читаемым и понимаемым. Даже если программная логика диктует подобные действия, стоит пожертвовать компактностью и использовать две переменных вместо одной.

Нужно отметить, что «при прочих равных» цикл, заданный оператором **for**, будет выполняться быстрее, чем аналогичные действия, записанные с помощью **while** или **repeat**, поскольку не нужно каждый раз вычислять условие, управляющее продолжением цикла.

Вопросы

1. Используйте в примере 2.1 в качестве тестового типа любой интервальный тип (например, –100..100). В качестве исходного значения переменной этого типа присвойте значение верхней границы. Что произойдет, если вы напишете в программе
   x := x + 1;
   Будет ли отличие, если Вы напишете вместо этого Inc(x)? Succ(x)? Почему?
2. Какой тип является базовым для типа T_SomeNumber?
3. Дополните пример 2.1 так, чтобы использовать все функции, работающие с порядковыми типами. Изучите, как они действуют на примере перечислимого типа T_fruits.
4. Какие события являются событиями по умолчанию для кнопки и элемента редактирования? Для формы? Для метки? Для компонента Memo?
5. Исследуйте тип Boolean с помощью программы, написанной в примере 2.1. Объясните полученные результаты. Будут ли наблюдаться различия при компиляции с включенной и с отключенной проверкой границ?
6. Исследуйте любой из логических типов (кроме Boolean) с помощью программы, написанной в примере 2.1. Изменится ли что-нибудь, если вместо применения процедуры Inc увеличить значение переменной путем повторного вызова функции Succ? Как влияет на результат включение и отключение проверки границ при компиляции?
7. В примере 2.1 использовался тип week. Дополните программу определением типа workweek (на базе week) и исследуйте этот тип.
8. Чем отличаются типизированные константы от истинных констант? Чем отличаются типизированные константы от переменных?
9. Можете ли Вы назвать переменную Index? Begin?
10. Можете ли Вы объявить переменную Integer типа Integer? Если да, то как?

Задания

1 Изучите с помощью справочной системы Delphi функции FormatDateTime, StrToDateTime, DateTimeToStr, а также Date, Time и Now. Начните новый проект и поместите на форму кнопку и компонент Memo. Разработайте приложение, которое при щелчке на кнопке будет выводить в компоненте Memo две строки следующего содержания (для их создания используйте функции форматирования):

a) Строка 1: «Число 1234.5678 можно записать как 1.2345678E+03 или, приближенно, как 1234.57» и строка 2, которая должна содержать текущую дату: «Сегодня – …(число) (месяц) года».

b) Строка 1: «Число Пи можно записать с разной точностью: с 2 знаками после запятой: 3.14, с 4 знаками после запятой: 3.1415, с 7 знаками после запятой: 3.1415926» и строка 2, которая должна содержать текущее время: «Сейчас – …часов … минут».

c) Строка 1: «Число E (основание натурального логарифма) можно записать с разной точностью: с 4 знаками после запятой: …, с 7 знаками после запятой: …, с 10 знаками после запятой: …» и строка 2, которая должна содержать завтрашнюю дату: «Завтра – …(день недели), … (месяц)».

d) Строка 1: «Сегодня курс доллара к рублю составил … руб.» (в рублях с указанием двух знаков после запятой) и строка 2 «Сегодня …(число, месяц) торги закончились в … (часы : минуты)».

e) Строка 1: «Число 125609342 – целое, но если нужно, может быть записано как вещественное 125609342,00 или как 125 609 342,00» и строка 2: «Сегодня – … (день недели сокращенно), … (число, месяц сокращенно), … (год двумя цифрами)».

f) Строка 1:«Значение ln 10 = … часто используется при переходе от десятичных логарифмов к натуральным. Это число можно записать в научном формате …» и строка 2 «Сегодня … (день недели полностью), … (число, месяц полностью), время …(часы, минуты)».

g) Строка 1: «Когда аргументом функции Format является вещественное число, например, 987654.321, его можно записать как в обобщенном формате: …, в научном формате: … или даже денежном формате: …» и строка 2 «…часов …минут …секунд …(число) … (месяц) …(год)».

h) Строка 1: «Если требуется отформатировать вещественное число, например 1357.2468, то количество знаков до и после запятой можно задать как аргументы. Тогда в формате 3.5 это число выглядит как

..., а в формате 5.3 как ...» и строка 2: «Вчера был ...(день недели полностью), ... (число, месяц сокращенно)».

i) Строка 1 «Функция Format позволяет использовать повторяющиеся слова в качестве аргументов строкового типа и коротко записать строчку: ложка, ложка, вилка, ложка, вилка, вилка, ложка, вилка, ложка» и строка 2 «Текущее время можно записать как ...(часы : минуты) или как ... (часы : минуты : секунды)».

j) Строка 1 «Текущая версия Delphi – ...(номер Вашей версии). Delphi может форматировать числа. Delphi может переводить числа в 16-ричную систему. Delphi может сообщить текущую дату и время» и строка 2 «Текущая дата и время: ...(год) ...(месяц) ...(день), ...(день недели), ...(часов) ...(минут) ... (секунд)».

k) Строка 1 «Delphi умеет форматировать строки. Строки умеет Delphi форматировать...» (все возможные осмысленные комбинации). Строка 2 «До конца занятия осталось ...(минут): (секунд)».

l) Строка 1 «Число ln 2 = ... играет важную роль в теории информации. Вот как оно выглядит с 8 знаками после запятой:...». Строка 2: «Послезавтра, в (день недели) около (час) у меня будет важная встреча».

m) Строка 1 «Очень легко выводить повторяющиеся числа: 10, 10, 20,30, 20, 20, 30». Строка 2 «Вчера занятия закончились в (часы): (минуты), сегодня занятия закончатся в (часы): (минуты)».

n) Строка 1 «Я работаю с Delphi (номер версии). Сейчас уже вышла версия (номер версии)». Строка 2 «До Нового года осталось (дни) (часы) (минуты) (секунды)».

o) Строка 1 «Курс рубля к доллару сегодня изменился на (руб.) (коп.). Изменение за неделю составило (руб.) (коп.)». Строка 2 «Я родился (родилась) ...(день) (месяц) (год). Это был (день недели)».

2 Разработайте приложение, которое будет выполнять описанные ниже действия (используйте типизированные константы). Поместите на форму кнопку и компонент Memo – для вывода результатов. При каждом щелчке левой клавишей мыши на кнопке выводится результат очередного элементарного шага – сложения или умножения.

a) Вычисляется сумма всех нечетных чисел от 1 до 21, выводится результат после каждого сложения.

b) Вычисляется факториал 9! Результат выводится после каждого умножения.

c) Вычисляется полусумма всех чисел от 100 до 150. Результат выводится после каждого сложения.

d) Вычисляется произведение всех чисел, кратных 3, от 3 до 63. Результат выводится после каждого умножения.

e) Вычисляется произведение дробей вида 1/N, где N меняется от 0.5 до 9.5 с шагом 0.5. Результат выводится после каждого умножения.

f) Вычисляется (последовательно) (1/2) × (3/4) × (5/6) × (7/8) × (9/10). Результат выводится после каждого умножения.

g) Вычисляется сумма квадратов всех четных чисел от 12 до 30. Результат выводится после каждого сложения.

h) Вычисляется произведение квадратных корней всех чисел, кратных 5, от 5 до 50. Результат выводится после каждого умножения.

i) Вычисляется 21!! (двойной факториал означает операцию, примененную только к нечетным, как в данном случае, поскольку 21 – нечетное число, или только к четным числам).

j) Вычисляется (((3^3)^3)^3), результат выводится после каждого возведения в степень.

k) Вычисляется сумма квадратных корней всех чисел, одновременно кратных 3 и 7 в диапазоне от 1 до 200. Результат выводится после каждого сложения.

l) Вычисляется 2^n, n=15, результат выводится после каждого возведения в степень.

m) Вычисляется сумма дробей вида k/(k+2), k=1..25, результат выводится после каждого сложения.

n) Вычисляется сумма слагаемых вида $(-1)^n /k!$, k=1..8, результат выводится после каждого сложения.

o) Вычисляется произведение квадратов чисел вида 1/N, где N меняется от 0,5 до 5,5 с шагом 0,5.

Указание.

При выполнении второго задания для хранения промежуточных результатов используйте типизированные константы, объявленные непосредственно в обработчике события.

Глава 3. Массивы в Delphi

Объявление массивов. Особенности использования динамических массивов. Практикум. Управляющие конструкции.

Объявление массивов

Тип массив (иначе – регулярный тип) является первым из группы структурированных типов, к рассмотрению которой мы сейчас приступим. Документация Delphi относит к структурированным типам множества (set), массивы (array), записи (record), а также файлы (file), классы (class), ссылки на класс (class-reference type, metaclass) и интерфейсы. За исключением множеств, элементами которых могут быть только значения порядковых типов, остальные структурированные типы могут содержать данные любых типов (в том числе и структурированных, причем вложенность структур не ограничивается).

✓ ***Примечание.*** *Объявление любого структурированного типа может начинаться с зарезервированного слова* **packed** *(упакованный). В стандартном Pascal – это требование к компилятору представить значения структурированного типа наиболее экономным образом, возможно, ценой замедления доступа к ним. Компилятор Object Pascal производит упаковку автоматически всегда, когда это возможно.*

Тип массив относится к типам, определяемым пользователем, и представляет собой набор тем или иным образом перенумерованных однотипных элементов. Синтаксис объявления такого типа следующий:

```
<тип массив> ::=
[packed] array ['['<тип индекса>{,<тип индекса>}']']
               of <базовый тип>

<тип индекса> ::= <порядковый тип>

<базовый тип> ::= <тип>
```

Индексы используются для нумерации элементов массива. Количество индексов называют размерностью массива. Типом индекса может быть любой порядковый тип, диапазон значений которого не превышает 2 Гигабайт. На практике наиболее часто в качестве типов индексов используются интервальные типы, базирующиеся на целых типах. Базовый тип – это тип элементов массива. Базовым может быть любой допустимый тип.

Когда типы индексов указаны, можно вычислить количество элементов массива. В этом случае память для переменных такого типа – массива может быть распределена заранее, на стадии компиляции программы. Массивы с заранее определенным количеством элементов называют статическими.

Приведем пример объявлений статических массивов:

```
type
    T_Vector = array [0..100] of char;
    {объявление одномерного массива символов}
    T_SomeThing = array [Boolean, T_Fruits, 0..2]
                  of Byte
    {объявление трехмерного массива целых чисел}
    T_Matrix = array [1..10, 10..20] of Integer;
    {объявление двумерного массива целых чисел}
```

Последнее объявление эквивалентно следующему:

```
    T_Matrix = array [1..10] of array [10..20]
               of Integer;
```

При работе с многомерными массивами часто бывает удобно трактовать их как массивы массивов. В этом случае объявление, например, типа TMatrix следует заменить парой объявлений:

```
    T_SomeVector = array [10..20] of Integer;
    T_SomeMatrix = array [1..10] of T_SomeVector;
```

Теперь можно объявить переменные этих типов:

```
var
    Vector : T_Vector;
    SomeThing: T_SomeThing;
    Matrix : T_Matrix;
```

Если в объявлении типа – массива типы индексов опущены, массив является динамическим. Количество элементов динамического массива может изменяться в ходе выполнения программы, и память для переменных такого типа должна также распределяться в ходе выполнения программы.

Фактически индексы такого массива имеют тип Integer и принимают только неотрицательные значения и, таким образом, ограничение в 2Гб, накладываемое на диапазон значений типов индексов статических массивов, действует и для динамических массивов (сравните с длинными строками). Нумерация элементов начинается только с нуля (для каждой размерности).

✓ ***Примечание.*** *Динамические массивы впервые появились в Delphi 4.0. В более ранних версиях допустимо объявлять и использовать только статические массивы.*

✓ ***Примечание.*** *Читатель, хорошо знакомый с каким-либо языком высокого уровня, вероятно, догадывается, что внутри динамических массивов «спрятаны» указатели. Однако на настоящем этапе мы намерены рассматривать динамические массивы, как программные объекты, имеющие определенное поведение, оставляя пока в стороне их внутреннюю структуру.*

Объявления динамических массивов и переменных соответствующих типов выглядят следующим образом:

```
type
    T_DynVector = array of Real;
    T_DynMatrix = array of array of Integer;
    T_DynCube = array of T_DynMatrix;

var
    DynVector1, DynVector2 : T_DynVector;
    DynMatrix1, DynMatrix2 : T_DynMatrix;
    DynCube: T_DynCube;
```

Как обычно, допустимо определять типы непосредственно при объявлении переменных, например:

```
var
    AnotherVector : array [0..100] of char;
    AnotherDynVector : array of Real;
```

но при этом следует помнить о возможных сложностях, связанных с совместимостью типов (см. главу 2).

Доступ к отдельным элементам массива (как статического, так и динамического) осуществляется с помощью индексов, указанных в квадратных скобках (квадратные скобки также называют операцией взятия индекса – subscript operator). Так, например, использование обозначения Vector[4] позволяет Вам работать с пятым элементом массива Vector (поскольку нумерация элементов в значениях типа T_Vector начинается с нуля). В памяти элементы массива располагаются следующим образом:

Элемент № 1 массива (для Vector – Vector[0]) → базовый адрес

Элемент № 2 → (Vector[1]) базовый адрес + размер типа элемента массива (в случае Vector – 1 байт)

Элемент № 3 → базовый адрес + размер типа элемента массива × 2

и т.д.

✓ ***Примечание.*** *Когда нумерация элементов массива (обычно одномерного) начинается с нуля, принято говорить о массиве с нулевой базой. Упакованный одномерный массив символов с нулевой базой принято называть упакованной строкой (packed string).*

Если речь идет о двумерном массиве, то его элементы располагаются в памяти в соответствии со следующим принципом: фиксируется первый индекс, а второй меняется от начала до конца диапазона (т.е. сначала Matrix[1,10], затем Matrix[1,11] и так далее, до Matrix[1,20]). После этого первый индекс увеличивается на единицу, и процесс повторяется (Matrix[2,10], Matrix[2,11] и т.д.). Используя аналогию с таблицей (которая может рассматриваться как пример двумерного массива), первый индекс часто называют номером строки, а второй – номером столбца. Для трехмерных массивов добавляется еще и наименование страницы для третьего индекса. Правило размещения в памяти элементов массивов с размерностью больше 2 такое же: первым меняется последний индекс, затем – предпоследний и т.д.

✓ ***Примечание.*** *К элементам массивов, размерность которых больше единицы, можно обращаться, указывая каждый индекс в собственной паре квадратных скобок, например:* `Matrix[5][12]`.

Понимание того, как массивы размещаются в памяти, оказывается полезным, когда требуется проинициализировать массив в программе, например, при объявлении константы соответствующего типа. При объявлении константы типа массив элементы массива заключаются в круглые скобки и перечисляются через запятую. Если массив имеет размерность больше 1, то дополнительно в круглые скобки заключаются значения для каждой размерности.

Пример:

```
type
    T_SmallVector = array [0..3] of char;
    T_SmallMatrix = array [0..2, 0..3] of Integer;
const
    C_SmallVector : T_SmallVector = ('a','b','c','1');
    C_SmallMatrix: T_SmallMatrix =
                      ((1,2,5,9),(3,4,7,11),(6,8,13,20));
    C_Fibonacci_6: array [0..5] of char =
                                  ('1','1','2','3','5','8');
// допустимо объявить тип константы непосредственно в
// разделе const
```

✓ ***Примечание.*** *Константы типа массив не могут содержать значения файлового типа на любом уровне вложенности.*

Поскольку тип – массив задается пользователем, а не является стандартным, компилятор не может самостоятельно установить тождественность типов, имеющих разные имена, но полностью идентичные описания, например:

```
type
T_AB1 = array [0..9] of Byte;
T_AB2 = array [0..9] of Byte;
var
    v1, v11 : T_AB1;
    v2 : T_AB2;
...
begin
...
v1 := v11; // это работает
...
v1 := v2; // ошибка
...
```

Такой код просто не будет откомпилирован, а в окне сообщений появится строка, извещающая о несовместимости типов T_AB1 и T_AB2. По присваиванию типы – массивы совместимы только если они тождественны. Вместе с тем ничто не мешает привести типы или произвести то же присваивание поэлементно:

```
var i :Byte;
...
begin
...
v1 := T_AB1(v2); // один из «обходных путей»
...
for i := 0 to 9 do
    v1[i] := v2[i]; // другая возможность
...
```

Единственное, что при этом следует контролировать, – возможность выхода за границы массива.

В силу этой же причины никаких специальных операций над переменными типа – массива не определено, поэтому говорить о совместимости типов в выражениях не имеет смысла, за единственным исключением. Одномерные массивы (упакованные и неупакованные) с нулевой базой и базовым типом Char совместимы в выражениях с символьными и строко-

выми типами. Они также совместимы по присваиванию со строковыми типами в том смысле, что переменной (или типизированной константе) строкового типа может быть присвоено значение одного из таких типов – массивов.

```
Пример:
var s : String; sv : T_SmallVector;
    // напомним, что T_SmallVector= array [0..3] of char;
...
begin
...
s := sv;
...
```

Обратное неверно: попытка присвоить переменной sv значение s (sv:=s) приведет к ошибке компиляции с выдачей сообщения о несовместимости типов. Однако переменные и типизированные константы описанных выше типов – массивов могут быть инициализированы с помощью строковой константы. Так, объявленную в одном из предыдущих примеров константу C_Fibonacci_6 можно было инициализировать следующим образом:

```
C_Fibonacci_6 : array [0..5] of char = '112358';
```

✓ ***Примечание.*** *К отдельным элементам массива, разумеется, могут применяться любые операции, определенные для базового типа.*

Особенности использования динамических массивов

Как уже говорилось, динамические массивы не имеют фиксированной длины: она может меняться во время выполнения программы. Память для таких структур распределяется динамически при помощи процедуры SetLength, например:

```
...
SetLength (DynVector1, 30);
SetLength (DynMatrix1, 10, 15);
```

Первая строка кода приведет к созданию массива вещественных чисел, перенумерованного от нуля до 29 (=30 – 1), и матрицы целых чисел, состоящей из 10 строк и 15 столбцов, перенумерованных, соответственно, от 0 до 9 и от 0 до 14. Перераспределить выделенную память можно путем нового вызова процедуры SetLength:

```
...
SetLength (DynVector1, 50);
```

Длина массива DynVector1 увеличится до 50 элементов, при этом существующие элементы не затираются.

✓ ***Примечание.*** *Вообще говоря, изменить размер уже созданного массива можно также с помощью функции Copy:*

```
DynVector1 := Copy (DynVector1, 3, 5);
```

Результатом выполнения этого оператора станет уменьшение размера массива DynVector1 до 5 элементов, причем он будет заполнен пятью элементами исходного массива, начиная с элемента №3. Функция Copy определена следующим образом:

```
function Copy(S; Index, Count: Integer): array;
```

где S – динамический массив, Index – индекс, с которого начинается выделение, Count – счетчик, определяющий, сколько элементов выделяется. Однако в документации рекомендуется использовать эту функцию только при передаче фактического параметра соответствующего типа в другую процедуру или функцию.

Существует одна важная особенность, проявляющаяся при присваивании динамических массивов друг другу. В отличие от статических массивов (и даже длинных строк) оператор присваивания не приводит к переписыванию значений из одного массива в другой. Происходит нечто иное: область памяти, на которую ссылается переменная в правой части оператора присваивания, как бы получает второе имя:

```
...
DynVector1[5] := 10.0;
DynVector2 := DynVector1;
{предварительно распределять память для DynVector2 не нужно}
DynVector1[5] := 20.0;
...
```

После выполнения всех этих операторов элемент DynVector2[5] будет иметь значение 20.0, как и DynVector1[5]. Если бы массивы DynVector1 и DynVector2 были бы статическими, то элемент DynVector2[5] имел бы значение 10.0. Чтобы получить два одинаковых динамических массива, нужно провести поэлементное присваивание:

```
...
    SetLength(DynVector2, 50);
    for k := 0 to High(DynVector2) do
        DynVector2[k] := DynVector1[k];
    {k - переменная целого типа}
...
```

Функция High используется для определения верхней границы массива. Разумеется, в приведенном примере известно, что k должно меняться до 49, но бывают случаи, когда нужно подстраховаться. Можно, конечно, пойти еще дальше и написать:

```
for k := Low(DynVector2) to High(DynVector2) ...
```

Однако нижняя граница динамического массива всегда 0 (впрочем, эти функции, а также функцию Length можно использовать и для статических массивов).

✓ ***Примечание.*** *При использовании массивов (и статических, и, особенно, динамических) очень важно контролировать попытки выхода за их границы. Получение сообщения «Access violation...» – еще не самое плохое, что может произойти: неизвестно, с какой областью памяти и каким образом будет работать программа. Поэтому при разработке и отладке следует включать опцию компилятора «Range checking».*

В завершение этого раздела – несколько слов о многомерных динамических массивах. Delphi позволяет создавать многомерные динамические массивы, имеющие переменную длину по одной или нескольким размерностям. Такой подход позволяет порой существенно сэкономить память. Классический пример приведен в документации – создание треугольной матрицы. Здесь он воспроизводится с незначительными изменениями и комментариями:

```
var
    A: array of array of string; {двумерный массив строк}
    I, J: Integer;
begin
    SetLength(A, 10);
    {распределение памяти для переменной A: указывается,
    что она будет иметь 10 строк}
    for I:=Low(A) to High(A) do
    //внешний цикл (по строкам)
    begin
        SetLength(A[I], I+1);
        {теперь распределяем память для каждой строки в
        отдельности. Если указать в качестве длины строки
        не I+1, а I, то главная диагональ будет пуста}
        //теперь - внутренний цикл (по столбцам)
        for J:=Low(A[I]) to High(A[I]) do
        A[I,J] := '('+IntToStr(I)+','+IntToStr(J)+')';
        {и заполняем полученную структуру строками вида
        '(I,J)'. Так, например, элемент A[4,2] получит
        значение '(4,2)'}
    end;
end;
```

✓ ***Примечание.*** *Для заполнения матрицы использованы два цикла for, вложенные друг в друга. В данном случае внешний цикл – цикл по строкам, а внутренний – по столбцам. Это означает, что для каждого значения I, нумерующего строки, выполняется цикл по J: значение J меняется от 0 (Low(A[I]) до I+1 (High(A[I])). Ограничений на вложенность циклов не существует.*

Отметим, что элементы многомерного динамического массива существуют достаточно автономно друг от друга. Так, допустимо распределять память только для некоторых из них:

```
...
    SetLength(DynMatrix2, 8);
    SetLength(DynMatrix2[5], 12);
    DynMatrix2[5,10] := 44;
    ...
    SetLength(DynCube, 4,4);
    SetLength(DynCube[2,3], 3);
    DynCube(2,3,2):=1;
...
```

Первые три (содержательные) строки кода показывают распределение памяти для целочисленной матрицы DynMatrix2. Сначала устанавливается, что эта матрица будет иметь 8 строк, а затем – что пятая строка содержит 12 элементов (12 столбцов). Одному из этих элементов присваивается конкретное значение. Следующая группа строк демонстрирует подобный подход к трехмерному массиву.

Чтобы освободить память, занятую динамическим массивом, нужно вызвать процедуру Finalize или присвоить массиву специальное пустое значение **nil**:

```
...
    Finalize(DynMatrix2);
    DynVector1 := nil
```

Практикум

☒ *Как в первом, так и во втором примере используются динамические массивы. Если версия Delphi, с которой Вы работаете, не предоставляет такой возможности, Вам придется несколько изменить код программы. Во-первых, вместо динамических массивов следует объявить статические массивы соответствующей размерности. Во-вторых, все вызовы процедур SetLength и Finalize нужно удалить или закомментировать. Наконец, вызовы функции High необходимо заменить обращением к целочисленной переменной, содержащей фактическую длину массива (такая переменная уже есть среди объявленных).*

Пример 3.1. Умножение многочленов

Приложение, которое мы сейчас разработаем, будет умножать задаваемый пользователем многочлен (любой степени) на многочлен вида $ax + b$ (коэффициенты a и b также задаются пользователем).

Итак, нужно обеспечить ввод следующих данных: степени многочлена, его коэффициентов, а также коэффициентов a и b. Для размещения коэффициентов многочлена будем использовать одномерный динамический массив, а для отображения исходного и полученного многочленов – компоненты Memo.

Поместите на форму два компонента Memo, выровняйте их и установите вертикальные полосы прокрутки (т.е. значение ssVertical свойства ScrollBars). Справа разместите два элемента редактирования Edit, две метки и одну кнопку BitBtn со страницы Additional Палитры компонентов. Внизу разместите один элемент редактирования и три кнопки BitBtn.

Кнопка BitBtn отличается от обычной кнопки тем, что на ее поверхности может располагаться изображение. Однако по умолчанию изображение отсутствует. Существует два способа повлиять на внешний вид кнопки. Первый способ состоит в изменении свойства Kind (Вид), изначально установленного в bkCustom. В выпадающем списке можно выбрать любой из 11 вариантов «настройки» – bkOk или bkCancel, например. Результатом будет появление на поверхности кнопки соответствующего заголовка и картинки справа от него: зеленой галочки для OK и красного креста для Cancel. Побочный эффект заключается в автоматическом изменении еще некоторых свойств, в том числе ModalResult, и, возможно, Cancel или Default (давая название свойству, разработчики, вероятно, имели в виду и устаревшее значение слова Kind – способ, манера поведения). Так, если был выбран вид bkCancel, то свойство ModalResult получит значение mrCancel, а свойство Cancel – значение True. Эти установки позволяют закрывать окно при нажатии клавиши <Esc>. Кнопка OK (Kind = bkOK) будет реагировать на нажатие клавиши <Enter>. Другие виды кнопок также предусматривают дополнительные настройки.

✓ ***Примечание.*** *Обратите внимание, что повторная установка свойства Kind в bkCustom (после того, как было выбрано любое другое значение) не приводит к автоматическому «сбросу» дополнительно установленных свойств.*

Поэтому, если какие-либо кнопки в приложении используются «стандартным образом», проще всего установить нужное значение свойства Kind. Если же ни один из вариантов не удовлетворяет запросам про-

граммиста, то следует оставить значение по умолчанию bkCustom и изменять другие свойства по отдельности. За появление картинки на кнопке отвечает свойство Glyph (с англ. – глиф, символическое изображение). В поле этого свойства указан его тип – TBitmap, а при активизации свойства появляется кнопка с многоточием, вызывающая диалоговое окно «Picture Editor» (сравните со свойством Lines компонента Memo). Вы можете найти большую коллекцию изображений в каталоге Program Files \ Common Files \ Borland Shared \ Images \Buttons. Расположением изображения на кнопке позволяет управлять свойство Layout.

✓ ***Примечание.*** *При желании Вы можете попробовать самостоятельно создать картинку для кнопки. Для кнопки обычного размера это должен быть 16-цветный рисунок размером 18 × 18 пикселей. Delphi предоставляет для создания изображений довольно простой в обращении редактор Image Editor. Чтобы запустить его, достаточно выбрать соответствующий пункт в меню «Tools». Разумеется, можно воспользоваться и стандартным для MS Windows редактором изображений Paint.*

Измените свойства формы, кнопок и компонентов Memo следующим образом:

Компонент	Name	Caption
Form1	frmMultiPoly	Умножение многочленов
Memo1	mmInitialPoly	–
Memo2	mmResultPoly	–
BitBtn1	bbtnMultiply	Умножить
Edit1	edtAcoef	–
Edit2	edtBcoef	–
BitBtn2	bbtnEnter	Ввести степень многочлена
BitBtn3	bbtnInput	Ввести коэффициент
BitBtn4	bbtnClose	Закрыть
Edit3	edtEnter	–

✓ ***Примечание.*** *В этом примере применяется схема именования компонентов, отличная от использованной ранее. Название компонента (например, BitBtn) сокращается до двух – четырех букв (bbtn) и используется в качестве префикса в имени (свойстве Name). Применяйте тот подход, который представляется Вам наиболее удобным, или придумайте свой способ называть компоненты. В любом случае, читать текст программы намного проще, если имя переменной свидетельствует о ее назначении.*

Очистите свойства Lines компонентов Memo и свойства Text элемента редактирования. Выберите для кнопок любые подходящие (с Вашей

точки зрения) изображения. Для кнопки «Закрыть» можете воспользоваться стандартными установками.

С точки зрения пользователя приложение будет работать следующим образом. Сначала необходимо ввести степень многочлена, а затем – по одному – его коэффициенты (ввод этих данных осуществляется в элементе редактирования edtEnter). По окончании ввода многочлен отображается в одном из компонентов Memo (mmInitialPoly). Для ввода каждого из коэффициентов *a* и *b* предусмотрено по элементу редактирования (edtAcoef и edtBcoef соответственно). Нажатие кнопки «Умножить» приводит к перемножению многочленов и отображению результата в другом компоненте Memo (mmResultPoly). В заголовке формы будут появляться инструкции для пользователя.

Целесообразно установить значение свойства Visible кнопок «Ввести коэффициент» и «Умножить», а также элементов редактирования коэффициентов в False. Это – «страховка» от обращения к несуществующему массиву коэффициентов многочлена. Память для него будет распределена только после ввода степени многочлена.

Довольно распространенный подход состоит в том, чтобы отделять «вычислительную» часть приложения от «интерфейсной». Согласно этому подходу объявление типа – динамического массива вещественных чисел – для хранения коэффициентов многочлена и функцию перемножения многочленов следует разместить в отдельном модуле.

Такого модуля в нашем проекте пока нет, его нужно создать. Для этого выберите в меню «File» пункт «New» и в появившемся диалоговом окне «New Items» на странице «New» выделите объект Unit (модуль). Щелчок на кнопке «OK» – и в окне редактора кода появится вторая страничка с именем Unit2. Сохраните Вашу работу: назовите модуль, отвечающий за пользовательский интерфейс poly_display.pas, вновь созданный Unit2 – poly_multiply.pas, а проект – Polynomial.dpr.

Для модуля poly_multiply.pas Delphi предлагает следующую заготовку:

```
unit poly_multiply;

interface

implementation

end.
```

Поместите в интерфейсный раздел этого модуля объявление типа для хранения коэффициентов многочлена и заголовок процедуры, перемножающий многочлены:

Листинг 3.1a

```
interface

type T_Polynom = array of real;

procedure Multiply(p: T_Polynom; a, b :real ; var res:
                        T_Polynom);
(p - массив коэффициентов многочлена, a, b - коэффициенты
многочлена вида ax+b, res - многочлен, получающийся в ре-
зультате перемножения двух описанных выше)
```

✓ ***Примечание.*** *Зарезервированное слово* **var** *в заголовке процедуры – новая деталь. Подробно о параметрах процедур и функций можно прочитать в главе 6. Пока достаточно считать, что таким образом указывается на существование некоторой переменной вне процедуры, в которую и будет занесен результат вычислений.*

Теперь добавьте имя модуля poly_multiply в предложение uses интерфейсного раздела модуля poly_display:

Листинг 3.1b

```
unit poly_display;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Buttons,
  poly_multiply;
```

Это позволит использовать в модуле poly_display данные типа T_Polynom и вызывать процедуру Multiply.

✓ ***Примечание.*** *Не следует в модуле poly_display повторно объявлять тип T_Polynom: с точки зрения компилятора это будет другой тип (хотя, в данном случае, и совместимый).*

Отложим пока реализацию процедуры Multiply и займемся организацией взаимодействия с пользователем.

Нам потребуются шесть переменных. Одна переменная (целого типа) будет использоваться в качестве счетчика при вводе массива коэффициентов. Еще две переменные: одна типа T_Polynom и одна целого типа нужны, соответственно, для обозначения одномерного динамического массива ко-

эффициентов и степени многочлена. Две вещественные переменные будут служить для хранения значений коэффициентов a и b. Наконец, строковая переменная Msg вводится, чтобы снабдить программу элементарными инструкциями, которые будут появляться в заголовке формы.

Поместите объявления этих переменных в раздел общедоступных объявлений типа формы:

Листинг 3.1c

```
...
type
  TfrmMultiPoly = class(TForm)
    edtEnter: TEdit;
    bbtnEnter: TBitBtn;
    bbtnInput: TBitBtn;...
...
  private
  { Private declarations - раздел частных объявлений}
  public
  { Public declarations - раздел общедоступных объявлений}
  inputCount : integer; {счетчик вводимых коэффициентов}
  order : integer; {степень многочлена}
  polynom : T_Polynom; {массив коэффициентов многочлена}
  Acoef: real; {коэффициенты многочлена, на который будет}
  Bcoef: real; {умножаться многочлен, заданный
                пользователем}
  msg : string; {строка для сообщений в заголовке формы}
end;
...
```

✓ ***Примечание***. *Конечно, можно было бы дополнить объявлениями переменных секцию var интерфейсного раздела или же создать такую секцию в разделе реализации. Однако был избран другой путь исходя из следующих соображений. Данные, которые содержатся в этих переменных, будут использоваться только формой и компонентами, расположенными на ней. И массив коэффициентов полинома в этом смысле имеет равные права с кнопками или компонентами Memo (объявления которых Delphi автоматически добавляет в объявление типа формы при помещении их на форму). Далее, когда нам понадобятся вспомогательные процедуры, их объявления также будут размещаться в разделе Public declaration после объявления переменных (подобно тому как обработчики событий размещаются после объявления компонентов).*

Вообще говоря, это уже объектно-ориентированное программирование. Мы детально обсудим такой синтаксис в главе «Объекты Delphi». Пока же (если Вас категорически не устраивает приведенное выше обоснование) можете объявлять автономные переменные и процедуры.

Следующий шаг заключается в создании обработчиков событий – действий пользователя. Сначала пользователь должен ввести степень многочлена в окне элемента редактирования edtEnter. Именно такая инструкция должна появиться в заголовке формы при ее создании. Одновременно в переменную msg записывается следующая инструкция для пользователя:

Листинг 3.1d

```
procedure TfrmMultiPoly.FormCreate(Sender: TObject);
begin
 Caption := 'Введите степень многочлена и нажмите кнопку
"Ввести степень многочлена"';
 msg := 'Вводите коэффициенты многочлена, начиная со стар-
шего';
end;
```

Для обновления указаний напишем процедуру UpDateFormCaption, которая будет изменять заголовок формы. Эта процедура будет вызываться всякий раз, когда пользователь совершит содержательное (с точки зрения нашего приложения) действие. Поместите заголовок процедуры в разделе общедоступных объявлений после объявления переменных:

```
    {Public declarations}
    ...
    msg : string; {строка для сообщений в заголовке формы}
    procedure UpdateFormCaption;
...
```

а собственно процедуру – в любом месте раздела реализации:

Листинг 3.1e

```
procedure TfrmMultiPoly.UpdateFormCaption;
begin
 Caption := Msg; {равноценно frmMultiPoly.Caption}
end;
```

✓ ***Примечание.*** *В старших версиях Delphi (начиная с 4) можно получить «заготовку» процедуры (входящей в состав объекта), подобную «заготовке» обработчика события, нажав (одновременно) клавиши <Ctrl> + <Shift> +C.*

После нажатия на кнопку «Ввести степень многочлена» должны выполняться следующие действия. Во-первых, в переменную order заносится значение степени многочлена, во-вторых, распределяется память для массива коэффициентов многочлена, в-третьих, становится видимой кнопка «Ввести коэффициент». Далее, поскольку надобность в кнопке «Ввести

степень многочлена» отпадает, она становится невидимой. Кроме того, обновляется инструкция для пользователя и инициализируется счетчик вводимых коэффициентов.

Листинг 3.1f

```
procedure TfrmMultiPoly.bbtnEnterClick(Sender: TObject);
begin
  order := StrToInt(edtEnter.Text);
                (получаем значение степени многочлена)
  inputCount := order; (присваиваем его счетчику)
  SetLength(polynom, order+1);
                (распределяем память для многочлена)
  bbtnInput.Visible := true;
          (делаем видимой кнопку "Ввести коэффициент")
  bbtnEnter.Visible := false;
     (делаем невидимой кнопку "Ввести степень многочлена")
  UpdateFormCaption(); (и обновляем заголовок формы)
  edtEnter.SetFocus();
          (передаем фокус ввода элементу редактирования)
end;
```

✓ ***Примечание.*** *Процедуры UpdateFormCaption и SetFocus не имеют параметров, и при их вызове вполне достаточно написать:*

```
UpdateFormCaption; (а не UpdateFormCaption();)
```

или же

```
edtEnter.SetFocus;
(так же без скобок - пустого списка параметров)
```

Object Pascal поддерживает оба варианта вызова (стандартный Pascal – только вариант без скобок).

Выбор той или иной формы записи – скорее дело вкуса, однако автор находит удобным использование скобок при вызове процедур или функций, не имеющих параметров. В этом случае при чтении программы легко отличить имена процедур и функций от имен переменных. (Читатель, имеющий опыт работы с C и подобными ему по синтаксису языками, вероятно, также оценит возможность использовать скобки)

Процедура SetFocus, вызываемая в последней строке обработчика события OnClick кнопки bbtnEnter, делает активным элемент редактирования.

Когда программа выполняется, в любой момент времени на форме может быть активен только один элемент (он визуально выделен). Обычно говорят, что этот элемент имеет фокус ввода. Практически это означает, что именно активный элемент будет реагировать на события клавиатуры или мыши, происходящие в данный момент. Таким образом, в нашем слу-

чае пользователь сможет сразу вводить старший коэффициент многочлена (последний элемент массива).

Кнопка «Ввести коэффициент» отвечает за заполнение массива коэффициентов многочлена.

Листинг 3.1g

```
procedure TfrmMultiPoly.bbtnInputClick(Sender: TObject);
begin
  polynom[inputCount] := StrToFloat(edtEnter.Text);
                          {читаем значение коэффициента}
  inputCount := inputCount - 1;
                         {уменьшаем значение счетчика на 1}
  msg := 'Вводите коэффициент при степени '+
                                    IntToStr(inputCount);
                         {формируем следующую инструкцию}
  edtEnter.SetFocus();
        {передаем фокус ввода элементу редактирования}
  IsPolyFull();
  {проверяем, закончено ли заполнение массива
  коэффициентов}
  UpdateFormCaption();
        {и обновляем заголовок формы}
end;
```

Проверка значения счетчика происходит в процедуре IsPolyFull. Если оно стало отрицательным, все коэффициенты (первого) многочлена введены. Теперь нужно получить значения коэффициентов a и b.

Заголовок этой процедуры, как и процедуры UpdateFormCaption, следует поместить в раздел Public declaration типа формы:

```
  procedure UpdateFormCaption;
  procedure IsPolyFull(); {проверяет, закончен ли ввод
                           коэффициентов}
```

а объявление процедуры – в любом месте раздела реализации:

Листинг 3.1h

```
procedure TfrmMultiPoly.IsPolyFull;
var s : string;
begin
 if inputCount < 0 then begin
   bbtnInput.Visible:= false; {элементы, обеспечивающие}
   edtEnter.Visible:=false; {ввод, становятся невидимыми,}
   bbtnMultiple.Visible := true; {а кнопка "Умножить" и}
```

```
    edtAcoef.Visible := true; {элементы редактирования}
    edtBcoef.Visible := true; {коэффициентов - доступными}
    edtAcoef.SetFocus();
    msg:=
     'Ввод коэффициентов полинома завершен. Введите a и b';
    s := FormatPolynom(polynom);
    DisplayPolynom(mmInitialPoly, s);
  end; // then
end;
```

Функция FormatPolynom и процедура DisplayPolynom обеспечивают отображение многочлена. Первая преобразует массив коэффициентов в строку вида $a[n]\cdot x\hat{}n + a[n-1]\cdot x\hat{}(n-1) + ... + a[1]\cdot x\hat{}1 + a[0]$, где n – степень многочлена, а знак «^» используется для обозначения возведения в степень, вторая выводит эту строку в компонент Memo. Заголовки функции и процедуры нужно, как и ранее, поместить в раздел общедоступных объявлений типа формы:

```
function FormatPolynom(p : T_Polynom) : string;
         {подготовка к отображению полинома в Memo}
procedure DisplayPolynom(var md : TMemo; sp : string);
         {отображение полинома}
```

а раздел реализации дополнить следующим кодом:

Листинг 3.1i

```
function     TfrmMultiPoly.FormatPolynom(p:     T_Polynom):
string;
  var i : integer; {текущий счетчик}
      sf : string; {строка для отображения коэффициента}
begin
  result := '';
  for i := length(p)-1 downto 1 do begin
   sf := format('%5.2f', [p[i]]);
{выбираем из массива очередной коэффициент, начиная со
старшего}
   result := result + sf + ' * x^'+IntToStr(i) + ' + ';
{и добавляем его в результирующую строку, умножая на x в
соответствующей степени}
  end; // for
  result := result + format('%5.2f', [p[0]]);
end;
```

```
procedure TfrmMultiPoly.DisplayPolynom(var md : TMemo;
                                       sp : string);
begin
  md.Lines.Add(sp); // просто добавляем строку
end;
```

Функция FormatPolynom получает в качестве единственного параметра массив коэффициентов многочлена (значение типа T_Polynom) и возвращает значение строкового типа. Стандартная функция format используется для отображения вещественных коэффициентов многочлена с двумя знаками после запятой. Ко всем коэффициентам, кроме последнего, слева дописывается *x* в соответствующей степени.

Процедура DisplayPolynom имеет два параметра: компонент Memo и строку, которая должна быть отображена в этом компоненте.

Теперь нужно создать обработчики событий OnClick для оставшихся кнопок. Кнопка «Закрыть» служит для завершения приложения:

Листинг 3.1j

```
procedure TfrmMultiPoly.bbtnCloseClick(Sender: TObject);
begin
 Finalize(polynom);
 // освобождается память, занятая массивом
 Close;
end;
```

✓ ***Примечание.*** *Close – это стандартная процедура, закрывающая форму. Если свойство Kind кнопки bbtnClose было установлено в bkClose, вызов этой процедуры не нужен.*

И, наконец, нажатие на кнопку «Умножить» вызывает процедуру Multiply для перемножения многочленов и отображает результат во втором компоненте Memo (mmResultPoly):

Листинг 3.1k

```
procedure TfrmMultiPoly.bbtnMultiplyClick(Sender:TObject);
var ww: T_Polynom; // переменная для результата
    rf: string; // строка для отображения в Memo
begin
  {считываем значения коэффициентов a и b}
  Acoef := StrToFloat(edtAcoef.Text);
  Bcoef := StrToFloat(edtBcoef.Text);
  {Распределяем память для результирующего многочлена}
  SetLength(ww, order + 2);
  {умножение многочлена на ax+b повысит его степень на 1}
```

```
  {вызываем процедуру Multiply}
  Multiply(polynom, Acoef, Bcoef, ww);
  {форматируем результат}
  rf := FormatPolynom(ww);
  {и отображаем его}
  DisplayPolynom(mmResultPoly, rf);
  {новое сообщение для пользователя}
  msg := 'Заданный полином умножен на '+ FloatToStr(Acoef)
                                  + 'x+' + FloatToStr(Bcoef);
  UpdateFormCaption;
  Finalize(ww); // освобождаем память, занимаемую массивом
end;
```

Теперь остается только написать собственно процедуру Multiply. Перейдите в Редакторе Кода на страницу модуля poly_multiply и наберите в разделе реализации следующий код:

Листинг 3.11

```
implementation

{эта процедура перемножает произвольный полином с полиномом вида ax + b}

procedure Multiply(p: T_Polynom; a, b: real;
                   var res: T_Polynom);
var i : integer;
begin
  {свободный член полинома просто будет умножен на b}
  res[0] := p[0] * b;
  {следующие коэффициенты вычисляются по формуле:
                                    a*p[i-1] + b*p[i]}
  for i := 1 to (length(p)-1) do
    res[i] := a * p[i-1] + b * p[i];
  {последний коэффициент формируется отдельно}
  res[length(p)] := a * p[length(p)-1];
end;

end. // implementation
```

Сохраните проект, откомпилируйте и запустите его на выполнение. Протестируйте приложение, используя простые примеры (такие, что результаты вычислений легко можно получить «вручную»). Отметим, что коэффициенты *a* и *b* могут быть изменены во время работы программы, а первый многочлен вводится пользователем один раз. Чтобы ввести другой многочлен, приходится запускать приложение снова. Исправить эту ситуа-

цию довольно просто. Поместите на форму еще одну кнопку (BitBtn или Button – как Вам нравится), назовите ее «Новый многочлен» и создайте обработчик события – щелчка мышью на этой кнопке. В этом обработчике события следует сделать видимыми кнопку bbtnEnter и элемент редактирования edtEnter, а свойство Visible кнопок «Умножить» и «Ввести» и элементов редактирования коэффициентов многочлена напротив, установить в значение False. Потребуется также написать новое сообщение для пользователя.

Пример 3.2. Вычисление обратной матрицы

В настоящем разделе мы разработаем приложение, вычисляющее матрицу, обратную задаваемой пользователем. Благодаря своей компактности и простоте матричные обозначения получили широкое распространение в математике, физике, механике. При различных расчетах довольно часто оказывается необходимым найти матрицу, обратную данной. Если матрица имеет небольшую размерность (2×2, 3×3), обратную ей несложно получить вручную. Однако с увеличением размерности возрастает и объем вычислений.

Начните новый проект. Поместите на форму элемент редактирования, четыре кнопки и компонент StringGrid (сетка строк), расположенный на странице Additional Палитры Компонентов. Этот компонент будет использоваться для ввода и отображения элементов матрицы.

Сетка строк (иногда ее называют таблицей строк) позволяет представить текстовые данные в виде таблицы. Сетка состоит из колонок (column) и рядов (row), пересечения которых образуют ячейки. Каждая из ячеек может содержать строку. Доступ к конкретной ячейке осуществляется с помощью свойства Cells[ACol, ARow], где целые числа ACol и ARow указывают координаты ячейки: колонку и ряд соответственно. Размер ячейки определяется свойствами DefaultColWidth (ширина колонки) и DefaultRowHeight (высота ряда), а ширина линий, разделяющих ячейки, – свойством GridLineWidth.

Нумерация колонок и рядов начинается с нуля. При необходимости можно работать с какой-либо колонкой или рядом как с единым целым, обратившись к ним с помощью свойств Cols[Index] или Rows[Index] (Index – целое число, номер колонки или ряда).

Количество колонок и рядов задаются свойствами ColCount и RowCount соответственно. По умолчанию значения обоих свойств равны 5. Свойства FixedCols и FixedRows содержат число колонок и рядов в фиксированной области.

Свойство Options предоставляет широкие возможности по настройке сетки. Мы упомянем лишь то, что включение опции goEditing позволяет редактировать содержимое ячеек, а включение опции goTabs –

перемещаться между ячейками с помощью клавиши табуляции <Tab> и клавиатурной комбинации <Shift><Tab>. Другие возможности читателю предлагается изучить самостоятельно с помощью оперативной справки.

✓ ***Примечание.*** *Обратите внимание, что в Инспекторе Объектов свойство Options помечено значком «+». Нажмите на этот значок, чтобы получить полный список опций. Упомянутые выше опции goEditing и goTabs по умолчанию выключены (установлены в False). Включите эти опции, а все остальные оставьте без изменений.*

Расположите элемент редактирования и все кнопки вдоль нижней границы формы, а левый верхний угол сетки строк совместите с левым верхним углом клиентской области формы. Измените значения свойств формы и расположенных на ней компонентов в соответствии со следующей таблицей:

Компонент	Name	Caption
Form1	frmReverse	Введите размерность матрицы
StringGrid1	sgMatrix	
BitBtn1	bbtnInputDim	Ввести
BitBtn2	bbtnReverse	Обратить
BitBtn3	bbtnHelp	Помощь
BitBtn4	bbtnClose	Закрыть
Edit1	edtInputDim	

Очистите свойство Text элемента редактирования и настройте внешний вид кнопок.

Как и в предыдущем примере, «вычислительная» часть проекта будет отделена от «интерфейсной». Добавьте в проект еще один модуль и сохраните проделанную работу. Назовите проект MatrixReverse.dpr, модуль Unit1 – mt_MainFrm.pas, модуль Unit2 – mt_CalcsRev.pas.

Сначала займемся интерфейсным разделом модуля mt_CalcsRev. Во-первых, объявите тип матрицы вещественных чисел:

Листинг 3.2а

```
unit mt_CalcsRev;

interface

type
  I_RealVector = array of real; // вспомогательный тип -
```

```
  // одномерный массив вещественных чисел
  T_RealMatrix = array of T_RealVector; // матрица -
  // двумерный массив вещественных чисел
...
```

Кроме того, в интерфейсный раздел следует поместить заголовок процедуры, вычисляющей матрицу, обратную данной, а в раздел реализации – «заготовку» для этой процедуры:

Листинг 3.2b

```
...
  procedure CalcReverse(var rmt: T_RealMatrix;
                        var msg: string);
  {эта процедура вычисляет обратную матрицу. Переменная
  msg используется для сообщения о результате вычисления.
  Если обратной матрицы не существует, исходная матрица не
  преобразуется, а в строковой переменной передается
  сообщение о том, что исходная матрица является
  вырожденной}

implementation

  procedure CalcReverse(var rmt: T_RealMatrix;
                        var msg: string);
  begin

  end;
...
```

Реализацию процедуры CalcReverse мы пока отложим и перейдем к разработке пользовательского интерфейса. Переключитесь в Редакторе Кода на страницу mt_MainFrm и добавьте имя модуля mt_CalcsRev в предложение uses интерфейсного раздела модуля mt_MainFrm:

Листинг 3.2c

```
unit mt_MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Buttons, Grids,
  mt_CalcsRev;
```

Далее, нам потребуются три переменные: одна – Dimension, целого типа – для обозначения размерности матрицы, вторая переменная – MyMatrix, типа T_RealMatrix – для обозначения собственно матрицы вещественных чисел, а третья, Msg (строкового типа), позволит формировать инструкции и сообщения для пользователя. Поместите объявления этих переменных в раздел общедоступных объявлений типа формы TfrmReverse (подобно тому, как это было сделано в примере 3.1).

✓ ***Примечание.*** *Обратная матрица может быть определена только для квадратной матрицы, поэтому для обозначения размерности матрицы достаточно одной переменной.*

Теперь создадим обработчики событий. Самым простым и коротким будет обработчик щелчка мышкой на кнопке «Закрыть» (bbtnClose):

Листинг 3.2d

```
procedure TfrmReverse.bbtnCloseClick(Sender: TObject);
begin
  Finalize(MyMatrix);
  // освобождаем память, занимаемую матрицей
  Close; // закрываем форму и завершаем приложение
end;
```

При щелчке мышкой на кнопке «Помощь» (bbtnHelp) будет выводиться специальное окно сообщения, информирующее пользователя о некоторых особенностях работы с приложением. В обработчике события формируются две строки: собственно сообщения, которое будет выводиться в этом окне, и заголовка окна. Затем эти строки преобразуются к типу PChar. Это – специальный строковый тип, с которым «умеет» работать Windows. Преобразование типов необходимо, поскольку далее строки передаются в стандартную функцию MessageBox, первые два параметра которой должны иметь именно тип PChar.

Листинг 3.2e

```
procedure TfrmReverse.bbtnHelpClick(Sender: TObject);
var scomm : string; // строка сообщения
    scap : string; // строка заголовка
    pcomm, pcap : PChar; // преобразованные строки
begin
 scomm := 'При вводе элементов матрицы удобно'#13#10;
 scomm := scomm +
    'пользоваться клавишами <Tab> и <Shift> <Tab>'#13#10;
 scomm := scomm +
    'Можно проверить правильность вычислений,'#13#10;
```

```
 scomm := scomm +
    'повторно нажав кнопку "Обратить". '#13#10;
 scomm := scomm +
    'Конечно, могут быть небольшие несовпадения,'#13#10;
 scomm := scomm +'связанные с погрешностью вычислений.';
 scap := 'Очень короткая справка';
 pcomm := PChar(scomm);
 pcap := PChar(scap);
 Application.MessageBox(pcomm, pcap, MB_OK);
// третий параметр отвечает за появление в окне кнопки OK
end;
```

✓ ***Примечание.*** *Функция MessageBox (определяемая как: **function** MessageBox(**const** Text, Caption: PChar; Flags: Longint): Integer;) приложения (Application) в действительности является оболочкой соответствующей функции Windows API, требующей большее число параметров. Поэтому при вызове функции нужно обязательно указывать квалификатор Application.*

✓ ***Примечание.*** *Если Вы работаете в Delphi 1 (или в любой из старших версий, но с отключенной директивой компилятора {$H–}), то приведение типов string (в данном случае – короткой строки) и PChar невыполнимо. Глубинные причины этого мы обсудим, когда будем подробно изучать строки, а пока приведем краткий рецепт того, «как сделать код работающим». Во-первых, вместо типа string в объявлении переменных, используйте тип, определенный как **array** [0..300] **of** char. Вообще говоря, вместо «300» может быть написано и «250», и «500»; принципиально лишь то, что нумерация массива начинается с 0. Затем измените строки, завершающие формирование текста сообщения и заголовка следующим образом:*

```
 scomm := scomm +'связанные с погрешностью вычислений.'#0;
 scap := 'Очень короткая справка'#0;
```

Символ #0, с точки зрения Windows, является признаком конца строки. Теперь осталось переписать следующие две строки:

```
 pcomm := scomm;
 pcap := scap;
```

– и программа будет нормально компилироваться и выполняться.

Щелчок на кнопке «Ввести» должен приводить к обновлению значения размерности матрицы и соответствующим изменениям внешнего вида формы:

Листинг 3.2f

```
procedure TfrmReverse.bbtnInputDimClick(Sender: TObject);
var temp : string;
begin
 temp := edtInputDim.Text; // считываем введенное значение
 Dimension := StrToInt(temp);
```

```
// устанавливаем размерность матрицы
 SetLength(MyMatrix, Dimension, Dimension);
 // распределяем память для матрицы
 Msg :=
     'Вводите элементы матрицы в соответствующие ячейки';
 // формируем новую инструкцию
 UpdateFormCaption; // обновляем заголовок формы
 UpdateGrid; // настраиваем сетку
end;
```

Процедура UpdateFormCaption полностью идентична одноименной процедуре в примере 3.1, а процедура UpdateGrid приводит в соответствие количество рядов и колонок в сетке с количеством строк и столбцов матрицы. Поместите заголовок этой процедуры в разделе общедоступных объявлений типа TfrmReverse, а приведенный ниже код – в любом месте раздела реализации:

Листинг 3.2g

```
procedure TfrmReverse.UpdateGrid;
begin
 sgMatrix.RowCount := Dimension;
 sgMatrix.ColCount := Dimension;
 {устанавливаем значения для числа строк и столбцов}
 sgMatrix.Height :=
  sgMatrix.RowCount * (sgMatrix.DefaultRowHeight +
                       sgMatrix.GridLineWidth);
 sgMatrix.Width :=
  sgMatrix.ColCount * (sgMatrix.DefaultColWidth +
                       sgMatrix.GridLineWidth);
  {вычисляем и устанавливаем новые размеры сетки с учетом
  ширины линий, разделяющих клетки}
end;
```

После того, как новые размеры сетки установлены, пользователь должен заполнить значениями все ее клетки. Затем при щелчке мышкой на кнопке «Обратить» происходит считывание этих значений, вычисление обратной матрицы и отображение результата. Соответствующий обработчик события будет иметь следующий вид:

Листинг 3.2h

```
procedure TfrmReverse.bbtnReverseClick(Sender: TObject);
begin
 ReadGrid; // считываем введенные значения
 CalcReverse(MyMatrix, Msg); // вычисляем обратную матрицу
```

```
 UpDateFormCaption; // обновляем заголовок формы
 DisplayGrid; // отображаем полученную матрицу
end;
```

Заголовки процедур ReadGrid и DisplayGrid также следует добавить в раздел Public declaration типа TfrmReverse, а их объявления поместить в любом месте раздела реализации:

Листинг 3.2i

```
procedure TfrmReverse.ReadGrid;
var i, j : integer;
begin
 for i := 0 to Dimension - 1 do
  for j := 0 to Dimension - 1 do
   begin
(чтобы получить доступ к ячейке из массива Cells сетки,
нужно в качестве первого индекса указать столбец, а в ка-
честве второго - строку. Поэтому меняем местами индексы)
    MyMatrix[i,j] := StrToFloat(sgMatrix.Cells[j,i]);
   end;
end;

procedure TfrmReverse.DisplayGrid;
var i, j : integer; temp : string;
begin
 for i := 0 to Dimension - 1 do
  for j := 0 to Dimension - 1 do
   begin
    temp := Format('%8.2f' ,[MyMatrix[i,j]]);
// см. замечание, сделанное в процедуре ReadGrid
    sgMatrix.Cells[j,i] := temp;
   end;
end;
```

Сохраните проект, откомпилируйте его, запустите на выполнение и проверьте работоспособность всех кнопок. Единственное, чего пока не может сделать программа – это вычислить обратную матрицу. К реализации этой процедуры мы сейчас и приступим.

✓ ***Примечание.*** *Обратная матрица будет вычисляться методом Гаусса – Жордана (автор надеется, что читатель знаком с основами линейной алгебры). Краткое описание этого метода приводится ниже.*

☒ *Рассмотрим следующие элементарные преобразования над матрицей X: умножение строки на число, замену строки линейной комбинацией этой строки с какой-либо другой строкой матрицы, перестановку двух строк. Идея метода Гаусса – Жордана заключается в отыскании такой последовательности преобразований, которая преобразует матрицу X к единичной матрице. Обозначая элементарные преобразования как L_i, можем записать:*

$$L\,X = (L_N \ldots L_2\,L_1)\,X = I,$$

откуда следует, что $L = X^{-1}$. *Это означает, что такая последовательность преобразований, будучи примененной к единичной матрице соответствующего размера, даст в результате матрицу, обратную* X.

Порядок действий обычно выглядит так. Сначала (посредством описанных выше преобразований) обращают в нуль все элементы первого (в нашем случае – нулевого) столбца за исключением диагонального элемента (который должен стать равным 1). Затем то же самое проделывают со вторым столбцом и так далее. Соответствующие преобразования единичной матрицы проделывают параллельно.

При применении метода могут возникнуть следующие проблемы. Во-первых, если элементы матрицы значительно отличаются по абсолютной величине, снижается точность вычислений. Поэтому перед применением метода Гаусса – Жордана желательно преобразовать матрицу так, чтобы ее элементы были (примерно) одного порядка. Во-вторых, на месте некоторого диагонального элемента, допустим, X[k,k] может оказаться нуль. В таком случае нужно найти строку m, такую, что X[k,m] ≠ 0, и поменять ее местами со строкой k. Если такой строки нет (т.е. столбец k – нулевой), матрица необратима. Отметим также, что точность вычислений будет выше, если на месте X[k,k] окажется максимальный по абсолютной величине элемент столбца k.

Запишем сначала «набросок» процедуры CalcReverse: только строки комментариев, описывающих, какие действия нам нужно выполнить:

Листинг 3.2k

```
procedure CalcReverse (var rmt: T_RealMatrix;
                       var msg: string);
begin
{1. Находим максимальный элемент в первом столбце (и запоминаем номер строки)}
{2. Меняем местами первую строку и строку с максимальным (по абсолютной величине) элементом в первом столбце. Если все элементы столбца равны нулю, выдать сообщение о необратимости матрицы.}
{3. Повторяем эти преобразования для единичной матрицы соответствующего размера}
{4. Делим все элементы первой строки на диагональный элемент}
{5. Повторяем эти преобразования для единичной матрицы}
```

```
{6. И обращаем в нуль все элементы первого столбца кроме
диагонального (повторяем эти преобразования для единичной
матрицы)}
{повторяем все действия для второго столбца и т.д. При
этом первая строка должна сохранять свое местоположение, а
значит, исключаться из рассмотрения в процедуре поиска
максимального элемента во втором столбце}
end;
```

Таким образом, нужно организовать цикл по столбцам матрицы, досрочно прерываемый в случае ее необратимости. Ясно также, что потребуется несколько переменных, в том числе переменная типа T_RealMatrix для описания единичной матрицы и переменные – счетчики. Удобно (как с точки зрения восприятия, так и с точки зрения отладки) ввести ряд вспомогательных процедур, описывающих отдельные этапы вычисления обратной матрицы. Фактически эти процедуры будут соответствовать пунктам комментариев. Приведем окончательный листинг процедуры CalcReverse (комментарии были несколько изменены при организации цикла):

Листинг 3.21

```
procedure CalcReverse(var rmt : T_RealMatrix;
                      var msg : string);
const eps = 10E-7;
{Сравнение с нулем вещественного числа может быть некор-
ректным, поэтому обычно вводят достаточно малую констан-
ту.}
var imt : T_RealMatrix; // "единичная" матрица
{в конце концов она станет равной обратной исходной матри-
це, но мы для краткости будем именовать ее "единичной"}
    h : integer; i_max : integer;
    i, j : integer; max_el : real;
begin
  h := High(rmt); // определяем размер обращаемой матрицы
  CreateAndFill_Identity(imt, h);
  // создаем единичную матрицу соответствующей размерности
  for j := 0 to h do begin
{1. Находим максимальный элемент в j-м столбце матрицы rmt
(и запоминаем номер строки i_max)}
    SeekMaxInCol(rmt, j, h, i_max);
    {если этот элемент равен нулю, то матрица необратима}
    if abs(rmt[i_max, j]) < eps then begin
      msg := 'матрица необратима!'; break; end; // then
    if i_max <> j then begin
{2. Меняем местами строку j и строку с максимальным (по
абсолютной величине) элементом в j-м столбце.}
      ChangeRows(rmt, j, i_max, h);
```

```
{3. Повторяем эти преобразования для единичной матрицы}
      ChangeRows(imt, j, i_max, h);
    end; // then
    max_el := rmt[j,j];
    // значение максимального элемента
{4. Делим все элементы j-й строки на максимальный элемент}
    DivideRowByMax(rmt, j, h, max_el);
{5. Повторяем эти преобразования для единичной матрицы}
    DivideRowByMax(imt, j, h, max_el);
{6. И обращаем в нуль все элементы j-го столбца кроме диа-
гонального (повторяем эти преобразования для единичной
матрицы)}
    ZeroCol(rmt, imt, j, h);
  end; // for j
  if msg <> 'матрица необратима!' then begin
    msg := 'Обратная матрица';
    CopyMatrix(imt, rmt);
{копируем результат (imt) в исходную матрицу rmt}
    Finalize(imt);
{освобождаем память, выделенную для imt}
  end; // if
end;
```

Начнем рассмотрение вспомогательных процедур с самой простой – CopyMatrix. Ее код практически не требует пояснений:

Листинг 3.2m

```
procedure CopyMatrix(var source, dest : T_RealMatrix);
{эта процедура физически копирует одну матрицу в другую.
 поскольку она является внутренней, проверки соответствия
 размерности источника и копии не производится}
var i, j : integer; dim : integer;
begin
  dim := high(source);
  for i := 0 to dim do
    for j := 0 to dim do
      dest[i, j] := source[i, j];
end;
```

Процедура CreateAndFill_Identity создает и заполняет должным образом единичную матрицу соответствующего размера:

Листинг 3.2n

```
procedure CreateAndFill_Identity(var im : T_RealMatrix;
                                     hi : integer);
```

```
(параметры: динамическая матрица и номер последнего столб-
ца (строки) обращаемой матрицы)
var i, j : integer; // счетчики
begin
  SetLength(im, hi+1, hi+1); // распределяем память
  for i := 0 to hi do
    for j := 0 to hi do
      if i <> j then im[i,j] := 0 else im[i,j] := 1;
      // и заполняем матрицу
end;
```

Процедура SeekMaxInCol имеет четыре параметра: собственно матрицу, номер столбца, в котором ищем максимальный элемент, номер последнего столбца (строки) обращаемой матрицы (вообще говоря, может быть опущен или заменен размерностью матрицы), а также номер строки, в которой обнаружился максимальный элемент заданного столбца. Эта процедура производит последовательный перебор элементов заданного столбца (начиная с диагонального) и определяет номер строки, в которой расположен максимальный из рассматриваемых элементов.

Листинг 3.2о

```
procedure SeekMaxInCol(const rm : T_RealMatrix;
                       js, hi : integer;
                       var imax : integer);
var i : integer; maxe : real;
begin
  imax := js; (начинаем со строки js (предыдущие уже пре-
образованы к нужному виду)
  maxe := abs(rm[js, js]); //"текущий" максимальный эле-
мент
  for i := js to hi do
    if abs(rm[i, js]) > maxe then begin
      maxe := abs(rm[i,js]); imax := i; end; // then
end;
```

Процедура ChangeRows обменивает местами строки с номерами js и imax в матрице rm (параметр hi имеет то же назначение, что и в предыдущем случае):

Листинг 3.2р

```
procedure ChangeRows(var rm:T_RealMatrix;
          js, imax: integer; hi : integer);
var t : real; k : integer;
begin
  for k := 0 to hi do begin
```

```
    t := rm[js, k]; rm[js, k] := rm[imax, k];
    rm[imax, k] := t;
  end ; // for k
end;
```

После перестановки строк нужно добиться того, чтобы на главной диагонали исходной матрицы оказалась единица. Эта задача решается в процедуре DivideRowByMax. Параметр ist – номер строки, m –число, на которое следует поделить ее элементы (rm и hi имеют прежний смысл).

Листинг 3.2q

```
procedure DivideRowByMax(var rm: T_RealMatrix;
                             ist, hi : integer; m : real);
var k : integer;
begin
  for k := 0 to hi do
  rm[ist, k] := rm[ist, k] / m;
end;
```

Наконец, рассмотрим процедуру ZeroCol. Она имеет два параметра типа T_RealMatrix: rm – исходная матрица, im – единичная матрица (точнее матрица, получившаяся из единичной к этому моменту), а также два целых параметра: js – номер столбца, hi – номер последнего столбца матрицы (как и ранее). Процедура ZeroCol обращает в нуль все элементы столбца js исходной матрицы кроме диагонального, последовательно заменяя строки матрицы их линейными комбинациями со строкой js. Такие же действия производятся и над «единичной» матрицей.

Листинг 3.2r

```
procedure ZeroCol(var rm, im: T_RealMatrix;
                    js, hi : integer);
var coef : real; {коэффициент в линейной комбинации строк
k и js, обращающий в нуль элемент rm[k,js]}
    k, l : integer;
begin
  for k := 0 to hi do begin
    if k <> js then begin
      coef := rm[k, js]; // поскольку rm[js,js] = 1
      for l := 0 to hi do begin
        rm[k, l] := rm[k, l] - rm[js, l] * coef;
        im[k, l] := im[k, l] - im[js, l] * coef;
// преобразуем строку с номером k, проходя по столбцам (l)
      end ; // for l
    end // then
```

```
    else continue; // собственно строчку js пропускаем
  end; // for k
end;
```

После объявления всех вспомогательных процедур следует сохранить проект и запустить его на выполнение. Проверьте, правильно ли работает программа, используя в качестве тестовых единичные матрицы разных размерностей, небольшие квадратные матрицы (2 × 2, 3 × 3), обратные которым легко найти вручную, а также матрицы с нулевыми столбцами и строками.

По окончании цикла (for j) в процедуре CalcReverse исходная матрица должна превратиться в единичную. Это легко проверить: закомментируйте строку, вызывающую процедуру CopyMatrix, и вновь запустите программу на выполнение. Теперь, нажав на кнопку «Обратить», Вы увидите на экране единичную матрицу.

Если матрица оказывается необратимой, приложение отображает «моментальный снимок» матрицы, полученной из исходной, на том шаге преобразований, на котором это выяснилось. Попробуйте изменить приложение так, чтобы в этом случае отображалась собственно исходная матрица. Возможно, Вы также найдете удобным использовать не один, а два компонента StringGrid (один для исходной матрицы, другой – для результата).

Управляющие конструкции

Теперь, когда у Вас есть некоторый опыт применения различных операторов языка Object Pascal (как условных, так и циклических), настало время рассмотреть «общую картину».

Если не стремиться дать строгое определение, то можно сказать, что алгоритм (а всякая программа является алгоритмом, записанным средствами языка программирования) – это конечная последовательность шагов, описывающая решение некоторой логической или математической задачи. Если все шаги алгоритма выполняются последовательно (и однократно), такой алгоритм называется линейным. Однако на практике часто требуется выполнить то или иное действие в зависимости от полученных результатов или же повторить несколько раз какое-либо действие или группу действий (т.е. нарушить «естественный» порядок выполнения программы). В теории программирования для описания этих и некоторых других ситуаций были разработаны специальные конструкции, получившие название управляющих. К настоящему моменту существует практически стандартный набор управляющих конструкций. Этот набор является в некотором смысле «сверхполным»: многие конструкции могут быть заменены другими, хотя и не без потери наглядности (отметим, что еще в 70-х гг. было доказано, что для реализации алгоритма любой сложности достаточно конструкции линейного следования и двух управляющих конструкций: выбора и цикла с предусловием [2], [3]).

✓ ***Примечание.*** *Своим названием управляющие конструкции, по всей вероятности, обязаны командам передачи управления.*

Приложение представляет собой последовательность машинных команд и данных. Когда приложение запускают на выполнение, оно загружается в оперативную память, после чего процессор строго последовательно выбирает из памяти команды. Когда некоторая команда будет исполнена, процессор передает управление следующей команде. Чтобы изменить ход выполнения приложения, используются специальные команды передачи управления, задача которых – сообщить процессору, какая команда будет выполняться следующей. При этом инструкция может носить как безусловный, так и условный характер (т.е. решение о передаче управления принимается в зависимости от полученных во время выполнения приложения результатов).

Операторы языка высокого уровня, как правило, заменяют целую группу машинных команд, и говорить о передаче управления от оператора к оператору можно лишь с некоторой долей условности.

В различных языках программирования управляющие конструкции реализуются по-разному: это могут быть и операторы языка, и стандартные процедуры; реализация некоторых управляющих конструкций может вообще отсутствовать. Так, например, в языке Java всем возможным управляющим конструкциям соответствуют операторы, в то время как в Object Pascal большинство из реализованных управляющих конструкций представлено операторами языка, меньшая часть – стандартными процедурами.

Все управляющие конструкции можно разделить на три группы: переходы, ветвления и циклы. Сейчас мы подробно рассмотрим конструкции, входящие в эти группы, и их конкретную реализацию в Object Pascal (отметим, однако, что «практически стандартный» набор управляющих конструкций все же не является жестко стандартизированным, и разночтения при использовании различных источников возможны).

✓ ***Примечание.*** *Вообще говоря, цикл может быть организован путем комбинирования переходов и ветвлений. Однако цикл – важная (и часто используемая) алгоритмическая структура, чем и обусловлено введение соответствующих управляющих конструкций.*

Начнем рассмотрение с ветвлений (их также называют конструкциями условия). Теоретически возможно существование четырех управляющих конструкций, и все они реализованы в Object Pascal.

1. Полная условная конструкция. Предполагает проверку некоторого условия и последующее выполнения некоторого действия (или группы действий), если условие соблюдается, и выполнение какого-либо другого действия (или группы действий) в противном случае. Полная условная конструкция реализована в Object Pascal посредством оператора **if**, имеющего ветвь **else**.
   Пример:
   ```
   if x >= 0 then y := x else y := - x;
   ```
2. Неполная условная конструкция. Предполагает проверку некоторого условия и последующее выполнение некоторого действия (группы действий), если условие соблюдается. В противном случае никаких специальных действий не предусмотрено, управление получает следующий оператор. Неполная условная конструкция реализована в Object Pascal посредством оператора **if**, в котором отсутствует ветвь **else**
   Пример:
   ```
   if (p div 2) = 0 then q := 1;
   ```
3. Полная конструкция выбора. Предполагает анализ значения выражения-переключателя, при этом некоторым (заранее определенным) значениям пе-

переключателя сопоставлены некоторые действия (группы действий). Если переключатель принимает одно из этих значений, выполняется действие (группа действий), соответствующее этому значению. Кроме того, существует действие (группа действий), выполняющееся, если переключатель принимает значение, не совпадающее ни с одним из заранее определенных. Полная конструкция выбора реализована в Object Pascal посредством оператора **case**, имеющего ветвь **else**.

Пример:

```
case k of
  1..3 : s := 'small';
  7..9 : s := 'big';
  else s := 'normal';
end;
```

4. Неполная конструкция выбора. Предполагает анализ значения выражения-переключателя, при этом некоторым (заранее определенным) значениям переключателя сопоставлены некоторые действия (группы действий). Если переключатель принимает одно из этих значений, выполняется действие (группа действий), соответствующее этому значению. Если переключатель принимает значение, не совпадающее ни с одним из заранее определенных, никаких специальных действий не выполняется, управление передается следующему оператору. Неполная конструкция выбора реализована в Object Pascal посредством оператора **case**, в котором отсутствует ветвь **else**.

   Пример:

```
case c of
  'Y' : v := v + 1;
  'N' : v := v - 1;
end;
```

Следующая группа управляющих конструкций – циклы (конструкции повторения). В теории описываются четыре конструкции, две из них реализованы в Object Pascal полностью, одна – частично. Прежде чем мы рассмотрим каждую из конструкций в отдельности, опишем черты, общие для всех циклов. При организации цикла можно выделить следующие этапы. Во-первых, это инициализация цикла (на практике обычно – присвоение начальных значений переменным, входящим в выражение, управляющее циклом); во-вторых, вычисление выражения, управляющего циклом (обычно условия); в-третьих – выполнение операторов, составляющих тело цикла. Наконец, последний, четвертый, этап – изменение параметров выражения, управляющего циклом (этим обеспечивается конечность цикла). Этапы со второго по четвертый выполняются многократно – до тех пор, пока значение выражения, управляющего циклом, будет допустимым. Порядок выполнения этих этапов (за исключением, быть может, первого) жестко не определен и различается в разных конструкциях.

1. Конструкция цикла с предусловием. Предполагает проверку некоторого условия и, если это условие соблюдается, выполнение действия (группы действий), обычно называемых телом цикла. Затем управление вновь передается в начало цикла, и процесс повторяется. Если условие не соблюдается, управление передается следующему (за оператором цикла) оператору. Чтобы цикл рано или поздно был завершен, необходимо, чтобы по крайней мере один оператор в теле цикла изменял проверяемое условие. Если в тот момент, когда оператор цикла получает управление, условие не соблюдается, тело цикла не будет выполнено ни разу. Конструкция цикла с предусловием реализована в Object Pascal с помощью оператора **while**.

Пример:

```
s := 0;
j := 1;
while j <= 10 do begin
  s := s + j;
  j := j + 1;
end;
```

2. Конструкция цикла с постусловием. Предполагает выполнение некоторого действия (группы действий) и последующую проверку некоторого условия. Если это условие не соблюдается, управление передается на начало цикла, и процесс повторяется. Если же условие соблюдено, управление передается следующему (за оператором цикла) оператору. Чтобы цикл рано или поздно был завершен, необходимо, чтобы по крайней мере один оператор в теле цикла изменял проверяемое условие. Тело цикла будет выполняться по меньшей мере один раз. Конструкция цикла с предусловием реализована в Object Pascal с помощью оператора **repeat**.

Пример:

```
s := 0;
j := 0;
repeat
  j := j + 1;
  s := s + j;
until j >= 10;
```

3. Конструкция цикла с параметром (со счетчиком). Для параметра определяются начальное и конечное значения, которые должны быть известны к тому моменту, когда оператор цикла получит управление. Когда цикл начинает выполняться, значение параметра становится равным начальному. Каждая итерация приводит к (автоматическому) изменению (увеличению или уменьшению) параметра на постоянную величину – шаг (шаг может быть как положительным, так и отрицательным). Цикл прекращается, как только значение параметра выходит за границу заданного диапазона. Конструкция цикла с параметром реализована во всех версиях Pascal, в том числе и в Object Pascal, лишь частично – с помощью оператора for. Напомним, что в операторе for счетчик должен быть порядкового типа, причем увеличиваться (форма **to**) или уменьшаться (форма **downto**) счетчик может только на единицу (в смысле этого порядкового типа). В общем случае такая конструкция предоставляет возможность управлять значением шага, кроме того, шаг и параметр могут быть и дробными.

Пример:

```
s := 0;
for j := 1 to 10 do s := s + j;
```

4. Конструкция бесконечного цикла. Предполагает непрерывное повторение некоторого действия (группы действий). (По такому принципу реализованы, например, ловушки сообщений операционных систем, «бегущие» строки). В Object Pascal такая конструкция непосредственно не реализована, однако бесконечный цикл легко организовать с помощью либо оператора цикла с предусловием, либо оператора цикла с постусловием.

Примеры:

```
while true do {оператор};
repeat {операторы} until false;
```

Последнюю группу управляющих конструкций составляют операторы перехода. Выделяется семь возможных конструкций, пять из которых реализованы в Pascal.

1. Конструкция безусловного перехода. Предполагает безусловный (выполняемый всегда) переход от одного действия к другому, которое не является следующим (в последовательности операторов). Эта конструкция реализована в Object Pascal с помощью оператора **goto**.

   Пример:

```
      x := 5;
      goto fff;
      y := x * x;
fff: y := x * x * x;
```

✓ ***Примечание.*** *Без крайней необходимости использовать конструкцию безусловного перехода не рекомендуется.*

2. Одноуровневая конструкция выхода из цикла. Предполагает завершение цикла вне зависимости от значения выражения, управляющего циклом, и передачу управления оператору, следующему непосредственно за телом цикла. Таким образом, завершается именно (и только) тот цикл, в теле которого присутствует данная конструкция. В Object Pascal одноуровневая конструкция выхода из цикла реализована с помощью стандартной процедуры Break.

✓ ***Примечание.*** *Здесь и далее в этом параграфе для стандартных процедур, не имеющих параметров, указываются только имена. В противном случае приводятся заголовки процедур.*

   Пример:

```
s := 0; j := 1;
while j <= 10 do begin
  s := s + j;
  if s > 40 then break;
  j := j+1;
end;
s := s*2;
```

3. Многоуровневая конструкция выхода из цикла. Эта конструкция имеет смысл только для вложенных циклов. Предполагает завершение (внутреннего) цикла вне зависимости от значения выражения, управляющего циклом, и передачу управления на любой указанный уровень вложенных циклов. Таким образом завершается не только тот цикл, в теле которого присутствует данная конструкция, но и один или несколько внешних (по отношению к рассматриваемому) циклов. В Object Pascal многоуровневая конструкция выхода из цикла не реализована. Осуществить описанные действия можно с помощью оператора **goto**.

   Пример:

```
{A - двумерный массив целых чисел}
            for j := 0 to 100 do
              for k :=0 to 200 do
                if A[j, k] = 0 then goto find_0;
find_0 : s := 'Нулевой элемент в массиве есть';
```

4. Одноуровневая конструкция продолжения цикла. Предполагает пропуск выполнения части операторов тела цикла (записанных после этой конструкции) и передачу управления в начало цикла. В Object Pascal одноуровневая конструкция продолжения цикла реализована с помощью стандартной процедуры Continue.

Примеры:

```
s:= 0;//подсчитаем сумму чисел от 0 до 9, не учитывая 5
for j := 0 to 9 do begin
  if j = 5 then Continue;
  s:= s+j; // когда j=5, этот оператор не выполняется.
end; // по завершении цикла s = 40.
{а теперь то же самое с помощью цикла while}
s := 0; j := 0;
while j <= 9 do begin
  j := j + 1;
  {принципиально, что увеличение j происходит до вызова
  Continue. Если оператор присваивания поместить после
  условного оператора, цикл станет бесконечным.
  В случае оператора for увеличение параметра цикла
  происходит автоматически (см. выше)}
  if j = 5 then Continue else s := s + j;
end;
```

5. Многоуровневая конструкция продолжения цикла. Эта конструкция имеет смысл для вложенных циклов. Предполагает пропуск выполнения части операторов тел (внутренних) циклов (записанных после этой конструкции) и передачу управления в начало одного из внешних (по отношению к рассматриваемому) циклов. В Object Pascal многоуровневая конструкция продолжения цикла не реализована. Осуществить описанные действия можно с помощью оператора goto.

Пример:

```
          k := 0; i := 0;
mylabel: while i < 5 do begin
           j := 1;
           i := i + 1;
           while j < 3 do begin
             k := k + i*j;
             if j = 2 then goto mylabel; {переход}
             j := j + 1;
           end;
           k := k + i; {не выполняется при j=2}
         end;
{Результат: k = 45. Если оператор k := k + i поместить
до заголовка цикла по j, по завершении обоих циклов
k станет равным 60. Если бы внутренний цикл не
прерывался, по окончании вычислений, k стало бы
равным 90.}
```

6. Конструкция выхода из процедуры (функции). Предполагает возвращение управления от текущей вызванной процедуры вызывающей (процедуре, функции или приложению) и передачу его оператору, следующему за вызовом процедуры (функции). В Object Pascal вызываемая процедура (функция) возвращает управление вызывающей после выполнения последнего операто-

ра тела этой процедуры (функции); конструкция «досрочного» выхода из процедуры (функции) реализована с помощью процедуры Exit.

Пример:

```
function CheckY (var Y: Real) : Boolean;
begin
  if Y>=0 then
    begin result:=True; Y:= sqr(Y); Exit; end;
  if sqr(Y) > 1 then begin result:= false; Exit; end;
  result := True;
  Y := Y+5;
end;
```

✓ ***Примечание.*** *Прервать выполнение процедуры (функции) можно также с помощью стандартной процедуры Abort. Мы обсудим эту процедуру при изучении исключительных ситуаций.*

✓ ***Примечание.*** *В ряде языков (C, Java, Basic) конструкция выходов из процедуры (функции) отвечает не только за прерывание выполнения этой процедуры (функции), но и за возвращение значения. В Pascal возвращаемый функцией результат должен быть присвоен специальной переменной result, либо имени функции; при этом такое присваивание не завершает выполнения функции.*

7. Конструкция выхода из программы. Предполагает завершение выполнения программы и передачу управления операционной системе. Приложения, разработанные средствами Object Pascal, могут завершаться различным образом. Консольные приложения завершаются после выполнения последнего оператора, либо – «досрочно» – с помощью стандартной процедуры Halt (**procedure** Halt [(ExitCode: Integer)], код выхода ExitCode – необязательный параметр). GUI – приложения завершаются при вызове Application.Terminate (который происходит в том числе и при закрытии (главной) формы приложения); процедура Halt также позволяет прервать выполнение приложения.

✓ ***Примечание.*** *Приложение может быть прервано и процедурой RunError (procedure RunError [(ErrorCode: Byte)], код ошибки ErrorCode не является обязательным параметром). Эта процедура (как и Abort) будет рассматриваться при изучении исключительных ситуаций. Отметим также, что выход из приложения может быть осуществлен и процедурой Exit, если текущей процедурой является собственно приложение*

Вопросы

1. Запишите константу типа массив, равную транспонированной матрице C_SmallMatrix.
2. Будет ли компилироваться следующий код:
```
var a :array[0..2, 0..3] of Integer;
  b : T_SmallMatrix;
begin
...
a := b;
...
```
3. Какой тип является базовым для типа T_SmallMatrix? Какие типы имеют индексы?
4. Запишите объявления статического массива, содержащего четыре вещественных элемента, и такого же динамического массива. Можно ли присвоить переменную первого типа переменной второго типа?
5. Какое значение будет иметь DynVector1[3] в результате выполнения следующего кода?
```
...
DynVector1[3] := 10.0;
DynVector2 := DynVector1;
DynVector2[3] := DynVector2[3] - 4;
```
6. Имеет ли смысл запись: SetLength(DynMatrix[2,3], 1)?
7. Какое значение получит DynVector2[0] в результате выполнения следующего кода?
```
...
DynVector2 := Copy(DynVector1, 3, 5);
DynVector1[3] := - 4;
...
```
8. Запишите объявление динамического двумерного массива символов Symb и проинициализируйте следующие его элементы: Symb[0, 0]; Symb[3,0]; Symb[5,0]; Symb[5, 3]; Symb[5, 7].
9. Приложение, разработанное в примере 3.1, позволяет пользователю задавать многочлены вида $a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + ... + a_1 \cdot x^1 + a_0$, где $n > 0$. Дополните приложение таким образом, чтобы пользователь мог задавать коэффициенты и при отрицательных степенях x. (Разумеется, ему придется ввести и максимальное по модулю значение отрицательной степени).
10. Вычисление матрицы, обратной заданной (пример 3.2), можно сделать более наглядным. Внесите изменения в процедуру CalcReverse так, чтобы нажатие на кнопку «Обратить» приводило к отображению очередного шага на пути от исходной матрицы к единичной.

Задания

1. Изучите с помощью справочной системы Delphi процедуру Randomize и функцию Random. Разработайте приложение, которое будет выполнять описанные ниже действия над числами и многочленами, которые могут как задаваться пользователем, так и генерироваться случайным образом:
   a) Перемножать два целых числа, имеющих не более 100 цифр в десятичной записи;
   b) Записывать в двоичной системе счисления целое число, имеющее не более 100 цифр в десятичной записи;
   c) Вычислять факториал числа, не превосходящего 100;
   d) Возводить многочлен (степени не выше 100) в квадрат;
   e) Вычислять ((...((*x*^*x*)^*x*)...)^*x*); задается целое *x* и сколько раз его нужно возвести в степень (количество возведений в степень);
   f) Умножать разность двух заданных многочленов на их сумму;
   g) Вычислять двойной факториал числа, не превосходящего 150;
   h) Перемножать два вещественных числа, имеющих не более 50 цифр в десятичной записи;
   i) Записывать в двоичной системе счисления вещественное число, имеющее не более 50 цифр в десятичной записи;
   j) Вычислять значение многочлена вида Q(P(x)); коэффициенты многочленов P и Q и значение x – вещественные числа;
   k) Записывать в шестнадцатеричной системе счисления вещественное число, имеющее не более 150 цифр в десятичной записи;
   l) Вычислять значение многочлена вида Q(P(Q(x))), коэффициенты многочленов и значение x – целые числа;
   m) Вычислять куб целого числа, имеющего не более 70 цифр в десятичной записи;
   n) Записывать в шестнадцатеричной системе счисления целое число, имеющее не более 200 цифр в десятичной записи;
   o) Вычислять значение многочлена P(P(... P(x))...), коэффициенты многочлена – целые числа, x – вещественное число.
2. Разработайте приложение, которое выполняет описанные ниже действия. Предусмотрите, что целочисленные матрицы A и B могут заполняться как пользователем (вручную), так и автоматически, посредством генерации случайных чисел.

a) Решить уравнение вида AX = B, где A, B, X – квадратные матрицы (одинаковой размерности);

b) Привести квадратную матрицу A к верхнему треугольному виду: ниже главной диагонали должны быть расположены только нулевые элементы;

c) Вычислить сумму и произведение двух квадратных матриц A и B;

d) Привести квадратную матрицу A к нижнему треугольному виду: выше главной диагонали должны быть расположены только нулевые элементы;

e) Вычислить A^n, где A – квадратная матрица, n – задается пользователем;

f) Вычислить определитель квадратной матрицы A;

g) Вычислить выражение AB – BA (коммутатор), A, B – квадратные матрицы;

h) Определить, является ли квадратная матрица A ортогональной (матрица называется ортогональной, если $A^T \times A = I$);

i) Определить, является ли квадратная матрица A унитарной (матрица называется унитарной, если $A^+ \times A = I$);

j) Разложить матрицу в сумму симметричной и асимметричной матриц;

k) Решить уравнение Ax = b, где A – квадратная матрица, x, b – вектора соответствующей длины;

l) Вычислить $A' = B^{-1} A B$;

m) Вычислить алгебраическое дополнение к элементу $a_{j,k}$ матрицы A; j и k задаются пользователем;

n) Вычислить exp(A) с точностью до члена порядка n, n задается пользователем;

o) Вычислить выражение AB + BA (антикоммутатор), A, B – квадратные матрицы.

Литература

1. Абрамов С.А., Гнездилова Г.Г., Капустина Е.Н., Селюн М.И. Задачи по программированию. М.: Наука, Гл. ред. физ.-мат. лит., 1988.
2. Аладьев В.З., Хунт Ю.А., Шишаков М.Л. Основы информатики. Учебное пособие. М: ИИД "Филин", 1998.
3. Бен-Ари М. Языки программирования. Практический сравнительный анализ. М.: Мир, 2000.
4. Вирт Н. Алгоритмы + структуры данных = программы. М.: Мир, 1985.
5. Вирт Н. Алгоритмы и структуры данных. СПб.: Невский диалект, 2001.
6. Дантеманн Дж., Мишел Д., Тейлор Д. Программирование в среде Delphi. Киев: НИПФ "ДиаСофт ЛТД", 1995.
7. Дарахвелидзе П.Г., Марков Е.П. Delphi 4. СПб.: БХВ – Санкт-Петербург, 1999.
8. Калверт Ч. Базы данных в Delphi 4. Руководство разработчика. Киев: Издательство "Диасофт", 1999.
9. Калверт Ч. Delphi 2. Энциклопедия пользователя. Киев: НИПФ "Диасофт ЛТД", 1996.
10. Конопка Р. Создание оригинальных компонент в среде Delphi. Киев: НИПФ – «ДиаСофт Лтд.», 1996.
11. Лишнер Р. Секреты Delphi 2. Киев: НИПФ – «ДиаСофт Лтд.», 1996.
12. Марченко А.И. Программирование в среде Borland Pascal 7.0. Киев: ВЕК, К.: ЮНИОР, 1996.
13. Матчо Д., Фолкнер Дж. Delphi. М.: БИНОМ, 1995.
14. Окулов С.М. Основы программирования. М.:ЮНИМЕДИАСТАЙЛ, 2002.
15. Пильщиков В.Н. Сборник упражнений по языку Паскаль. М.: Наука, Гл. ред. физ.-мат. лит., 1989.
16. Сван Т. Основы программирования в Delphi для Windows'95. Киев: "Диалектика", 1996.
17. Тейксейра С., Пачеко К. Delphi 4. Руководство разработчика. Киев; М.; СПб.: Издательский дом «Вильямс», 1999.