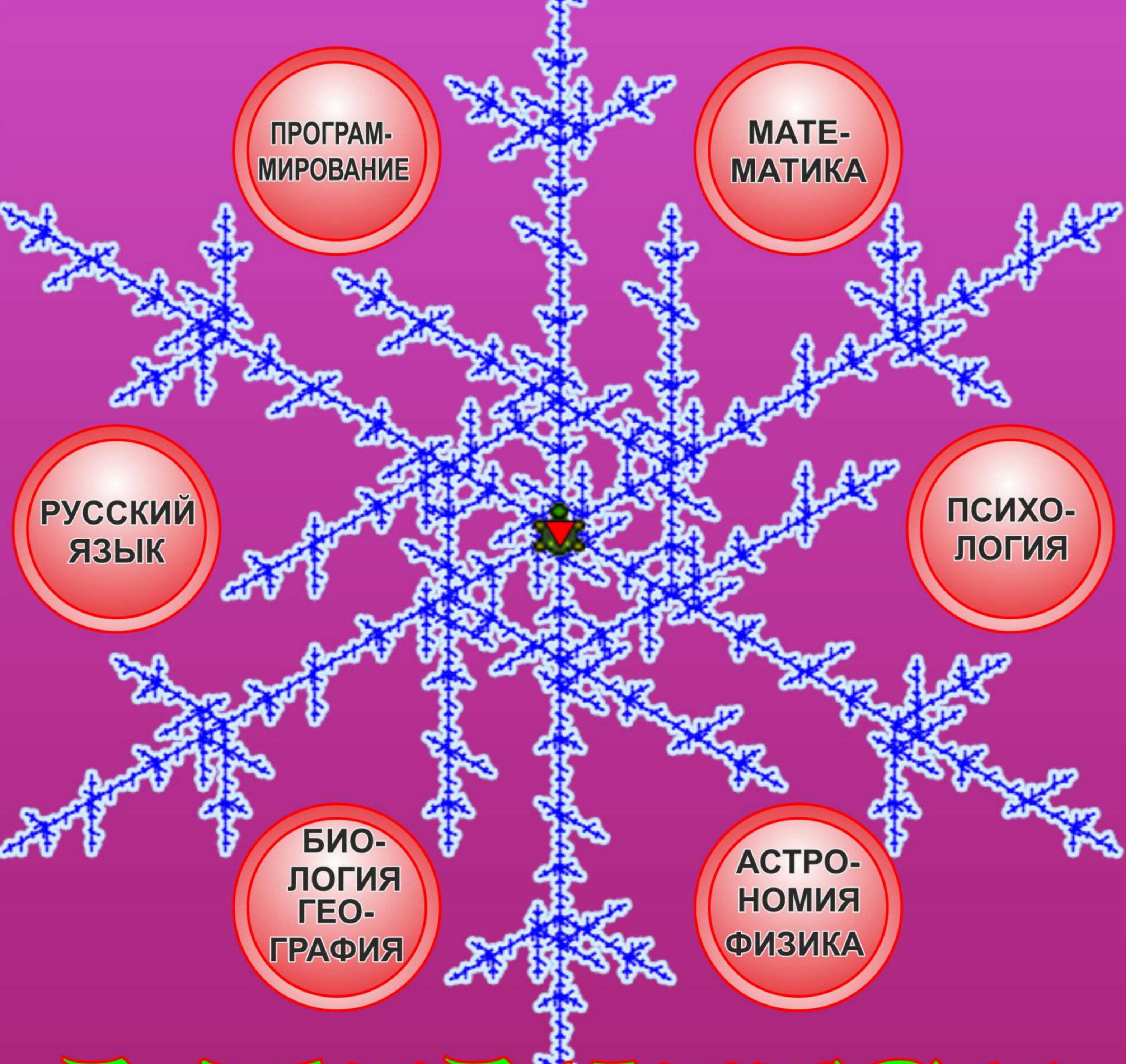
РУБАНЦЕВ ВАЛЕРИЙ

ЗАНИМАТЕЛЬНЫЕ УРОКИ



С ПАСКАЛЕМ

Валерий Рубанцев

ЗАНИМАТЕЛЬНЫЕ УРОКИ

С ПАСКАЛЕМ, или

PascalABC.NET для начинающих

Бесплатное издание детскиекниги.рф

Все права защищены. Никакая часть этой книги не может быть воспроизведена в любой форме без письменного разрешения правообладателей.

Автор книги не несёт ответственности за возможный вред от использования информации, составляющей содержание книги и приложений.

Copyright *Валерий Рубанцев*

Лилия Рубанцева

От автора

Учиться - всегда пригодится!

Русская пословица

Прошло немало лет с тех пор как компьютеры появились в школе и дома. В школе вы изучаете с их помощью информатику (что полезно), а дома он служит вам партнёром для игр (что приятно). А *цель* этой книги - показать, как можно сочетать приятное с полезным, то есть использовать компьютер при изучении школьной программы. В этой книге вы найдёте несколько десятков уроков по разным школьным предметам. Но уроки эти не простые, а *занимательные*! Поэтому на каждом уроке мы будем писать интересные компьютерные программы.

На этих уроках вы **узнаете**:

- как гениальный немецкий художник Альбрехт Дюрер составил знаменитый *магический квадрат* и почему он поместил его на своей гравюре *Меланхолия*;
- о черепахе, на панцире которой был нарисован первый в мире магический квадрат *Ло-шу*;
- и о другой *Черепашке* - она умеет бегать по экрану, оставляя за собой причудливый след в виде замысловатых фигур;
- о *литорее* обычной и мудрой;
- как средневековый математик *Фибоначчи* разводил кроликов, и что из этого вышло;
- что такое *тыблоко*;
- как просеивать числа через *решето Эратосфена*;
- чем занимается *высшая арифметика* и *комбинаторика*;
- о секретах *транслитерации;*
- как рекурсия помогает решать головоломку *Ханойские башни* и зачем нам нужен *стек*.

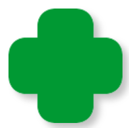
И **научитесь**:

- угадывать результаты футбольных матчей не хуже осьминога Пауля;
- *рисовать* пикселями, линиями, прямоугольниками и эллипсами красивые узоры;
- дрессировать курьёзных компьютерных зверушек - неугомонного *тьюрмита* и фрактальную *Киберчерепашку*;
- расшифровывать *тарабарскую грамоту*;
- устраивать *тараканьи бега*;
- сбрасывать тяжёлые ядра с Пизанской башни;
- писать *консольные* и *графические* приложения в среде *PascalABC.NET;*
- решать шахматную *задачу Гаусса* за пару минут;
- составлять огромные *магические квадраты*;
- говорить на *логопефическом* языке;
- играть в *Супернаборщика* и в *Города*;
- отыскивать *палиндромы*, *факториалы*, *простые* и *суперпростые* числа;
- подсчитывать предметы с первого взгляда…

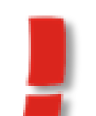
В общем, всего и не перечесть! А теперь – звенит звонок, пора на первый занимательный урок!

Валерий Рубанцев

Условные обозначения, принятые в книге:



Дополнение или замечание



Ненавязчивое требование или указание

Домашнее задание для самостоятельного решения

Папка с исходным кодом программы

Исходные коды программ и выполняемые файлы находятся в папке *Projects*.

Оглавление

ЗАНИМАТЕЛЬНЫЕ УРОКИ С ПАСКАЛЕМ 2
От автора 3
Оглавление 7
Урок 1. Устанавливаем *PascalABC.NET* 9
Урок 2. Запускаем *PascalABC.NET* 21
Урок 3. Как сберечь программу 27
Урок 4. Наша первая программа! 37
Урок 5. Какие бывают числа 56
Урок 6. Консольные приложения 74
Урок 7. Операторы цикла *While* и *Repeat* 90
Урок 8. Признаки делимости 107
Урок 9. Простые числа 123
Урок 10. Файлы 142
Урок 11. Палиндромы 157
Урок 12. Занимательная комбинаторика 183
Урок 13. Занимательная математика 200
Урок 14. Графические приложения 210
Урок 15. Текст в *Графическом окне* 225
Урок 16. Класс *Math* 239
Урок 17. Компьютерная графика 250
Урок 18. Полярная система координат 270
Урок 19. Занимательные игры с пикселями 281
Урок 20. Занимательная прямолинейность 295
Урок 21. Геометрические фантазии 307
Урок 22. Черепашья графика 322
Урок 23. Фрактальная Киберчерепашка 344
Урок 24. Тьюрмиты 375
Урок 25. Элементы управления 409
Урок 26. Занимательное моделирование 430
Урок 27. Занимательная физика 440

Урок 28. Тыблоки ... 460
Урок 29. Занимательная логопедия ... 466
Урок 30. Занимательная транслитерация ... 471
Урок 31. Занимательная латиница ... 475
Урок 32. Занимательная криптография ... 482
Урок 33. Занимательная биология ... 499
Урок 34. Занимательная психология ... 515
Урок 35. Звёздное небо ... 530
Урок 36. С первого взгляда! ... 538
Урок 37. Тараканьи бега по методу Монте-Карло ... 545
Урок 38. Перебор с возвратами ... 573
Урок 39. Занимательная Гауссиана ... 583
Урок 40. Полный перебор ... 590
Урок 41. Рекурсия, или Сказочка про белого бычка ... 603
Урок 42. Занимательная география ... 626
Урок 43. Магические квадраты ... 648
Урок 44. (PascalABC.NET or C#) or (PascalABC.NET and C#) ... 673

Литература ... 691

Урок 1. Устанавливаем *PascalABC.NET*

Le Roi est mort, vive le Roi!

Король умер. Да здравствует король!

«Коронная» фраза герцога д'Юзеса

Слухи о моей смерти сильно преувеличены.

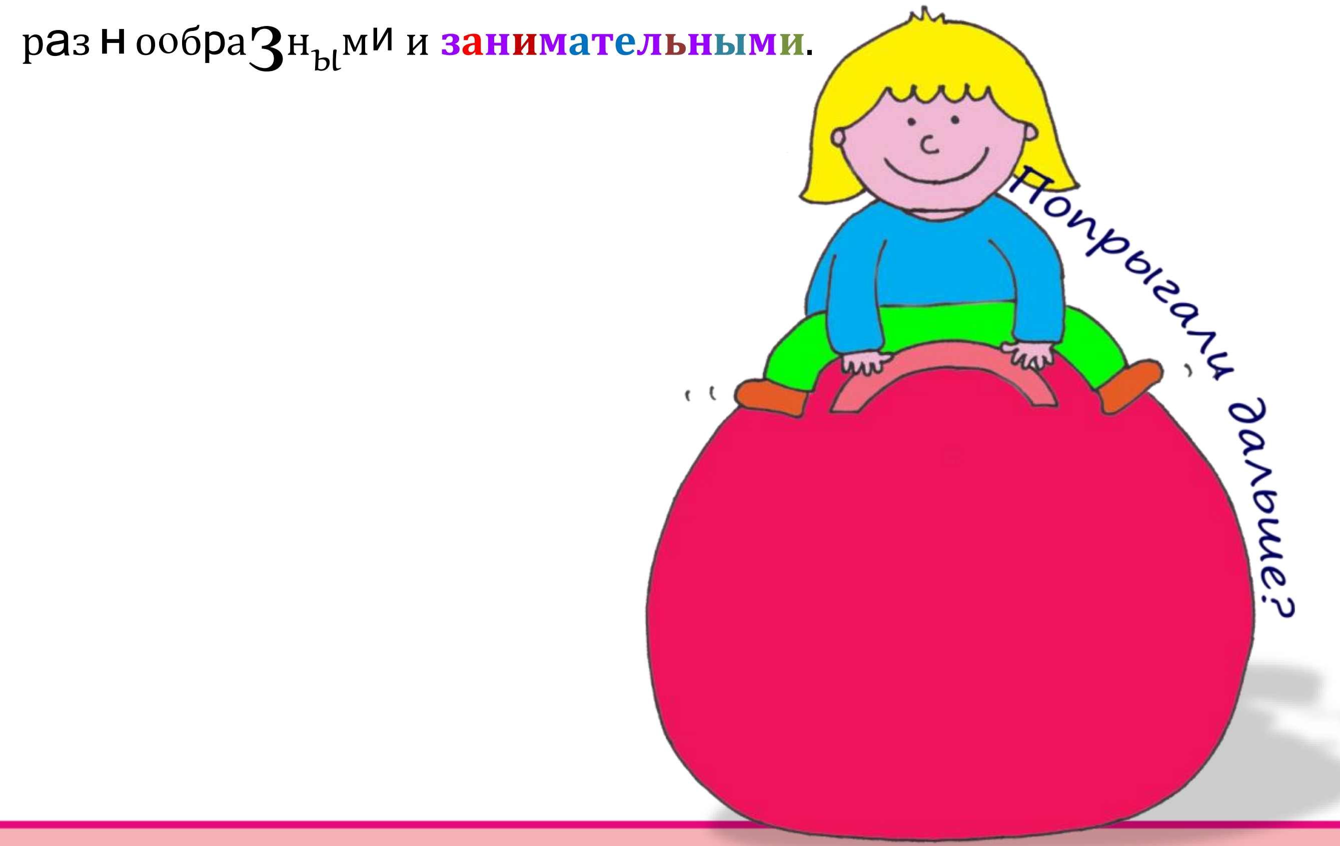
Марк Твен

В современной школе невозможно обойтись без компьютера. Великий комбинатор Остап Бендер сказал бы сейчас: *Компьютер – не роскошь, а средство передвижения по дорогам жизни!* А вот чтобы управлять компьютером, нужно учиться программированию. В самом деле, современный компьютер не должен быть только интерактивным телевизором, средством для общения в чатах и на форумах или игровой приставкой. Это скорее инструмент для воплощения творческих замыслов. С его помощью можно сочинять музыку, писать книги, рисовать картины, обрабатывать фотографии и редактировать фильмы. Эх, да что там говорить: компьютер может почти всё! А пользователю остаётся только освоить нужные программы и иметь творческую натуру, ищущую самовыражения. И это очень занятно, но… Это всё *чужие* программы, а ведь куда интереснее написать *свою* собственную - которая умеет делать то, чего никакая другая сделать не сможет! Правда, для этого придётся овладеть языком программирования, без которого объяснить компьютеру свои замыслы не удастся.

Как известно, лучшим языком для изучения программирования был и остаётся *паскаль*. В школе изучают его морально устаревшую версию для давно умершей операционной системы *ДОС* (*Дисковая Операционная Система*) с невероятно неудобной средой разработки. Если к этим очевидным недостаткам добавить ещё и скучнейшие уроки по скучнейшим же школьным учебникам, то это вполне может создать неверное впечатление, что паскаль уже не соответствует современному уровню образования. Конечно, это не так! На смену *Турбо Паскалю* пришли новые версии языка и среды разработки программ. Для этой книги я

выбрал систему программирования *PascalABC.NET 1.8*. Это отечественная разработка, специально созданная для изучения основных конструкций и приёмов программирования. Она бесплатна, имеет неплохой редактор кода и встроенный отладчик. *PascalABC.NET 1.8*. позволяет писать консольные и графические программы для операционной системы *Windows* и, что особенно важно, эта версия паскаля удачно сочетает простоту и логичность языка паскаль с мощью платформы *.NET*. Это значит, что вы можете не просто *изучать* язык программирования паскаль, но и *писать* на нём разнообразные и полноценные приложения для этой платформы! Более того, вы даже можете параллельно с осваиванием паскаля изучать и основной язык платформы *.NET* – *Си-шарп*. В последней главе вы найдёте несколько примеров программ на этом языке, которые написаны в той же самой среде разработки программ, что и все остальные, «паскальные» примеры в этой книге.

Эта книга – не учебник по паскалю и не справочник по среде разработки, но в ней вы найдёте всю необходимую информацию для того, чтобы хорошо разобраться и в том, и в другом. Это особенно важно потому, что эта книга – *первая*, в которой для изучения программирования выбрана система *PascalABC.NET*. Что касается уроков, то я старался сделать их разнообразными и занимательными.



Начинаем!

Лёд тронулся,

господа присяжные заседатели!

Остап Бендер

В мире существует не одна сотня языков программирования - на все случае жизни. Большинство из них очень сложны для начинающих и могут напрочь отбить желание учиться программированию. Но есть языки, специально предназначенные для обучения. В первую очередь, это *паскаль*, созданный в 1969 году Никлаусом Виртом, и *бейсик,* разработанный несколькими годами раньше профессорами Дартмутского колледжа Томасом Куртом и Джоном Кемени.

Язык *паскаль* назван в честь выдающегося французского математика и физика *Блеза Паскаля* (1623 – 1662), а название языка *бейсик* представляет собой сокращение его полного английского наименования - *Beginner's All-purpose Symbolic Instruction Code* (*универсальный код символических инструкций для начинающих*) – *BASIC*. Несколько неуклюжее название языка объясняется желанием авторов придать ему глубокий смысл: по-английски слово *basic* значит *основной, главный*. Поскольку эти языки программирования действительно являются основными и главными для многих любителей программирования, то их названия стали нарицательными и пишутся с маленькой буквы.

Оба языка до сих пор широко используются в программировании и имеют множество реализаций для всех операционных систем и процессоров. В качестве языка программирования для этой книги выбран *PascalABC.NET*, специально созданный *российскими* разработчиками для обучения школьников и студентов. Скоро вы и сами убедитесь, что выучить его *основы* можно за несколько уроков. Но несмотря на свою относительную простоту, *PascalABC.NET* позволяет писать и *полезные* приложения по школьной программе, и *занимательные* игры с красивой графикой.

Дальше мы будем называть *PascalABC.NET* просто *паскалем*, но вы должны помнить, что речь всегда идёт именно об этой версии, а не о, например, *Турбо Паскале*.

Поскольку эта книга рассказывает об использовании компьютера на *разных* уроках, а не только на уроках программирования, то мы не успеем изучить *все* возможности языка и написать БОЛЬШИЕ программы, но скучно на уроках вам точно не будет!

Чтобы научиться программировать, нужно программировать! А для этого нам потребуются:

1. Сам **компьютер** - без него нельзя!

2. На компьютере должна быть установлена операционная система **Windows XP**, **Windows Vista** или **Windows 7**.

3. Подключение к **Интернету**.

4. Программа **PascalABC.NET**.

5. Знание **английского языка** не обязательно, но несколько английских слов выучить необходимо!

Будем считать, что первые три пункта этого списка уже выполнены. А если нет? - А если нет, вам придётся самоотверженным трудом и примерным поведением - с посильной помощью родителей, конечно, - эти три пункта непременно выполнить. Потому что без них программировать нельзя, а ведь программирование - это самый занимательный школьный предмет!

Начнём же мы наш первый урок с того, что установим *PascalABC.NET* на компьютер. Но прежде его нужно *скачать*. Вот для этого нам просто необходим *Интернет*!

Установочный файл *PascalABCNETWithDotNetSetup18.exe* можно скачать с официального сайта программы, то есть *pascalabc.net*. На главной странице сайта найдите кнопку *Скачать* и нажмите её (Рис. 1.1).

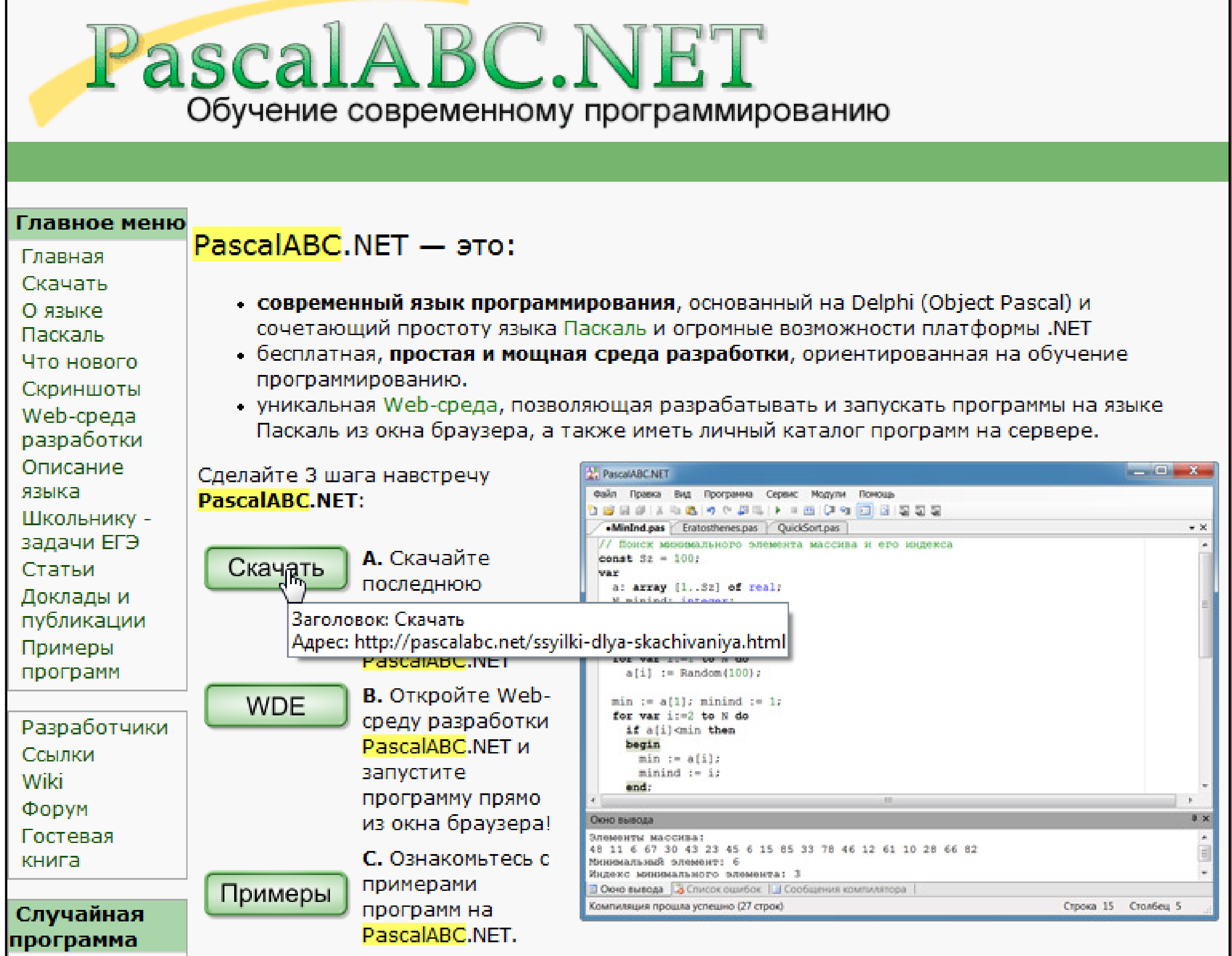


Рис. 1.1. Главная страница сайта *pascalabc.net*

Вы перейдёте на новую страницу со ссылками на скачивание (Рис. 1.2).

Здесь нажмите *первую* кнопку *Скачать*, чтобы при необходимости была установлена и платформа *.NET Framework 4.0*.

В открывшемся диалоговом окне нажмите кнопку *Сохранить* (Рис. 1.3, сверху).

Ссылки для скачивания

PascalABC.NET (версия 1.8, сборка 496 от 01.06.2012)

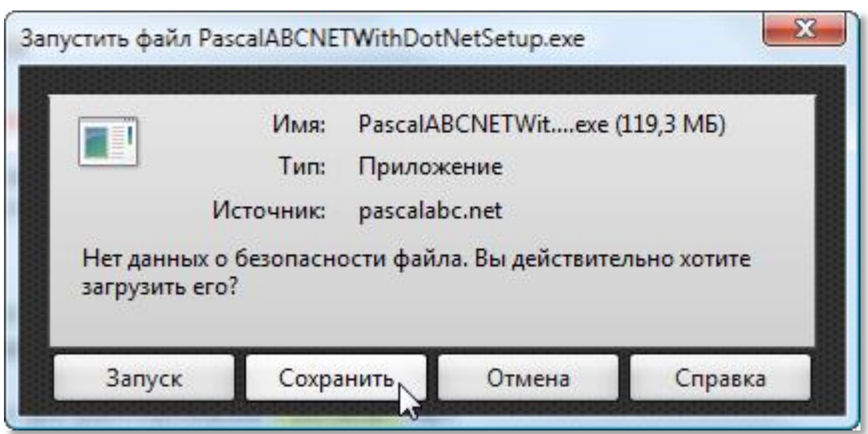
Внимание! Если у Вас установлена предыдущая версия PascalABC.NET, то при **первой** установке PascalABC.NET 1.8 обязательно установите полную версию **PascalABC.NET + Microsoft .NET Framework v4.0.**

Скачать

PascalABC.NET + Microsoft .NET Framework v4.0 (Setup, 118 Mb)
Для первой установки
Содержит:

- Система программирования PascalABC.NET
- Задачник PT4
- Платформа Microsoft .NET Framework **v4.0**
- Russian Language Pack for .NET v4.0 (русификация сообщений о ошибках времени выполнения)
- Framework Class Library Help (документация для всплывающей подсказки)
- **Справка по .NET библиотекам с кодом на PascalABC.NET** (chm-файл, 50 Мб)

Рис. 1.2. Выбираем нужный файл



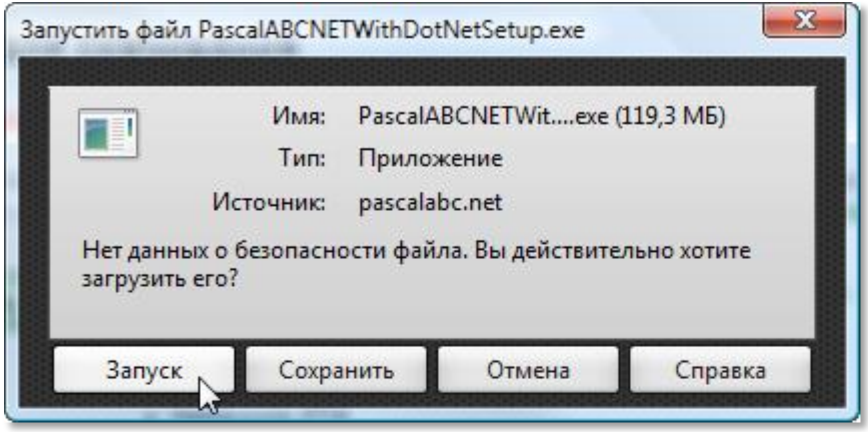


Рис. 1.3. Выбираем способ установки

Затем вам нужно выбрать *папку* для сохранения установочного файла (Рис. 1.4). Когда файл окажется на вашем компьютере, смело дважды кликните по его ярлыку мышкой (Рис. 1.5)!

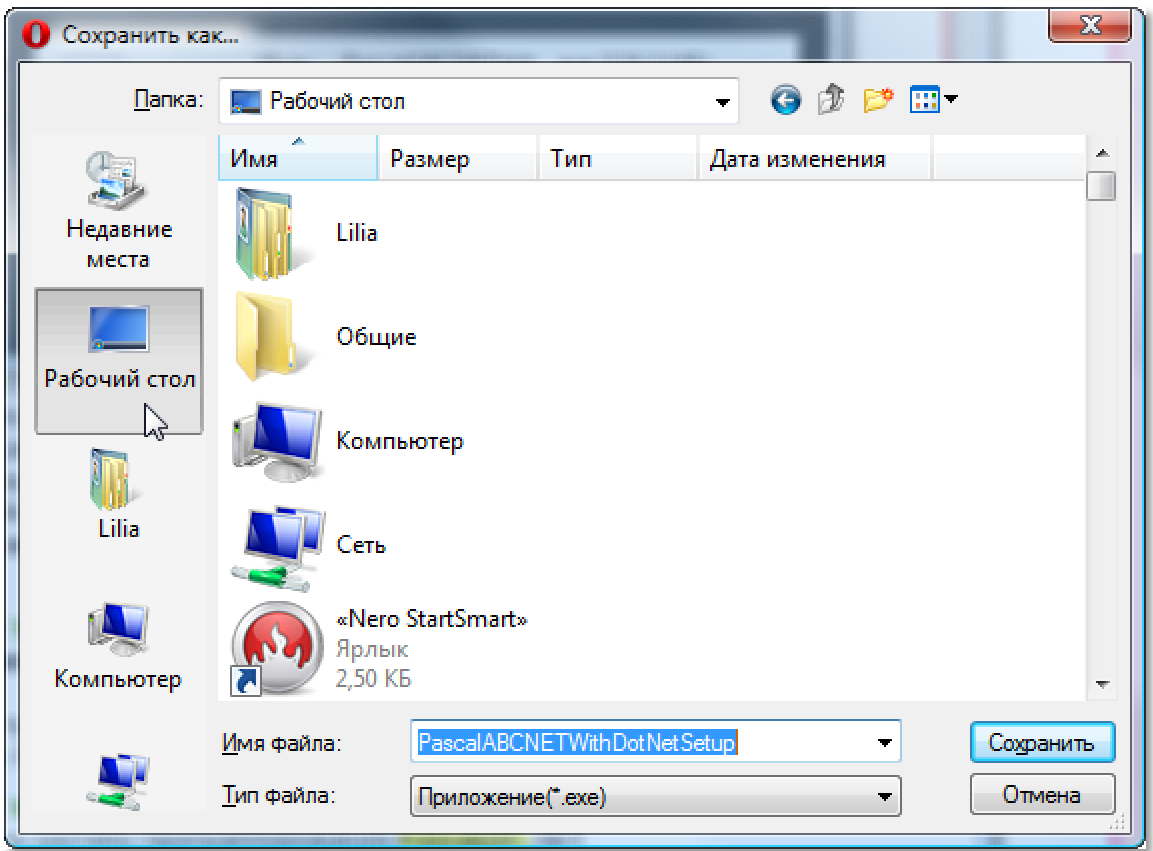


Рис. 1.4. Выбираем место для сохранения установочного файла

Поскольку установочный файл вам, скорее всего, больше не понадобится, то его можно сохранить прямо на *Рабочем столе,* где и появится ярлык.

Рис. 1.5. Ярлык установочного файла

Но вы можете *сразу* установить *PascalABC.NET* на свой компьютер, если предпочтёте кнопку *Запуск* (Рис. 1.3, снизу).

Независимо от выбранного способа установки, вы увидите диалоговое окно (Рис. 1.6).

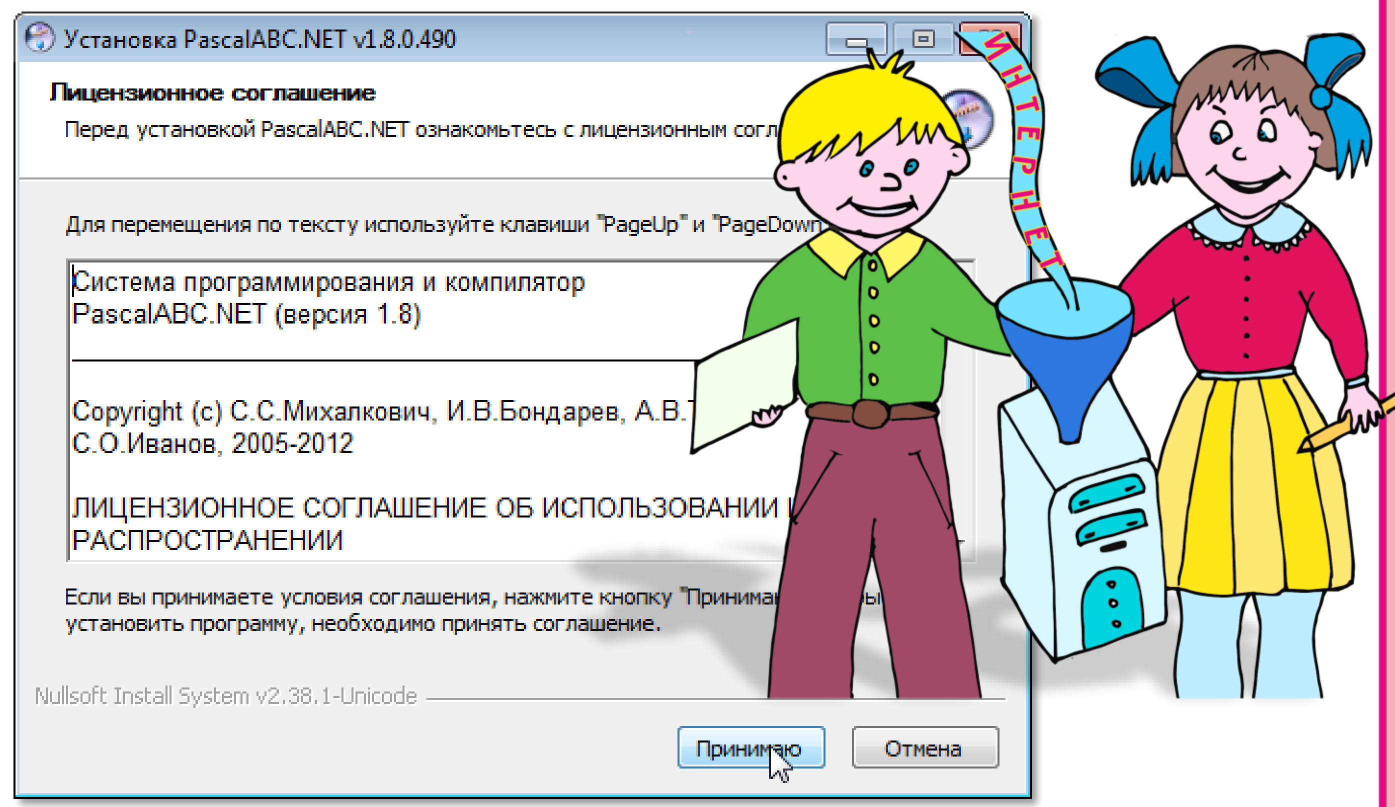


Рис. 1.6. Диалоговое окно лицензирования программы

Вас не должно удивлять такое обилие разных окон - ведь и название самой операционной системы *Windows* переводится как *Окна*.

Снова кликаем по кнопке *Принимаю* и получаем новое диалоговое окно, в котором можно выбрать компоненты программы, которые будут установлены на ваш компьютер (Рис. 1.7). Просто нажмите кнопку *Далее*.

В следующем окне нажмите кнопку *Установить* (Рис. 1.8).

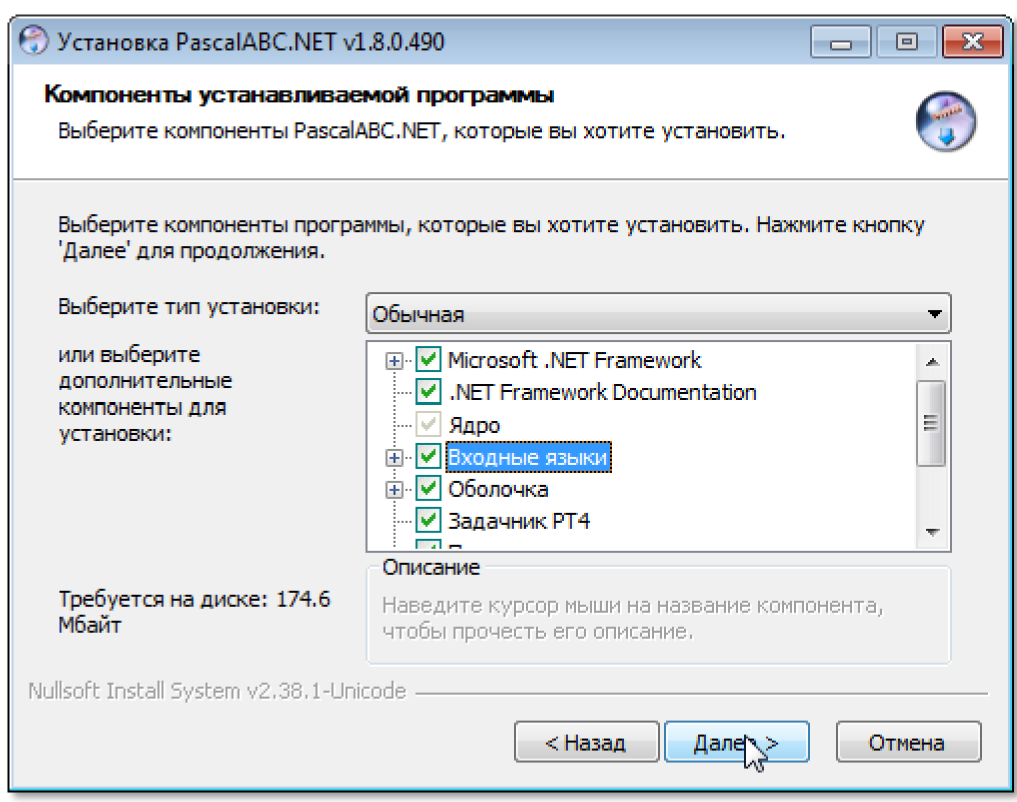


Рис. 1.7. Выбираем компоненты программы

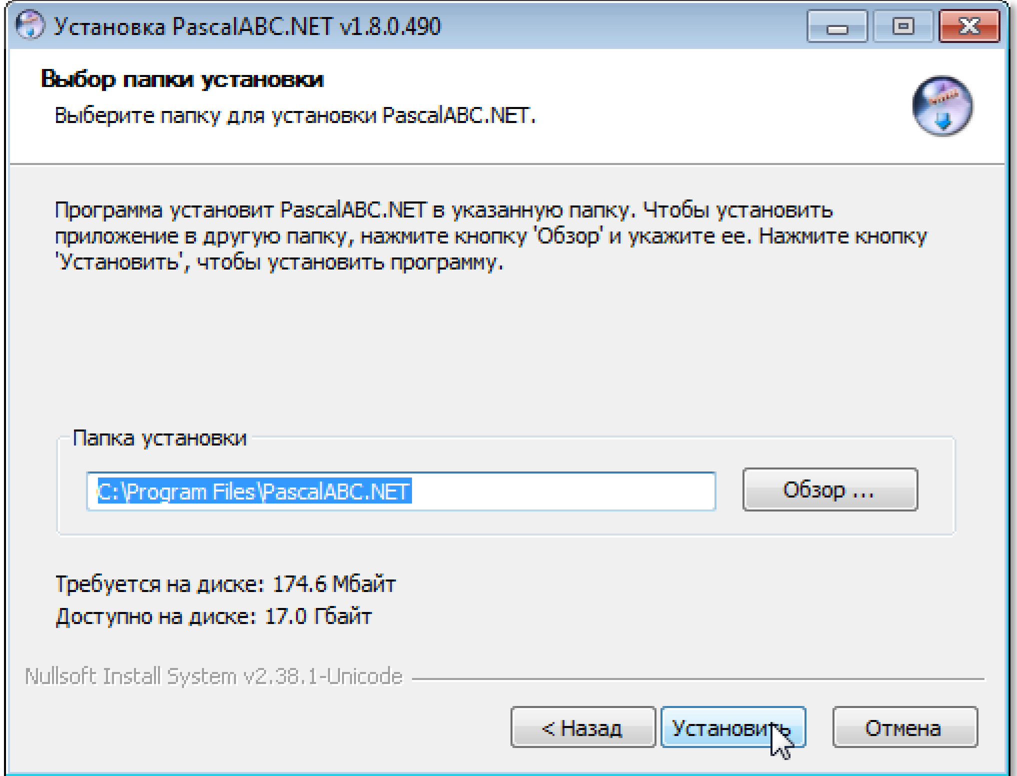


Рис. 1.8. Выбираем папку для установки программы

Вы можете выбрать *любую* папку для программы, но лучше оставить папку по умолчанию, чтобы потом не искать её на диске.

Запомните папку с программой, она вам ещё может пригодиться!

Процесс установки показывает **зелёная** полоска (Рис. 1.9).

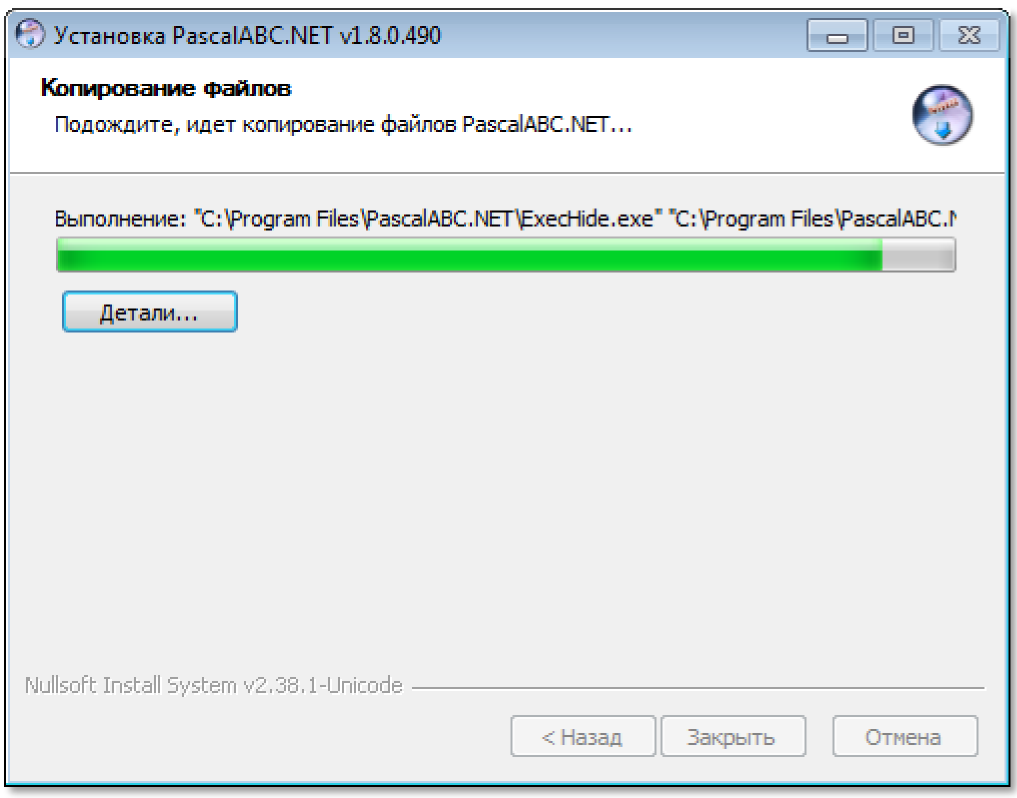


Рис. 1.9. *PascalABC.NET* устанавливается на компьютер

Готово! Прошло всего несколько минут - и вы уже счастливые обладатели замечательной программы!

Осталось завершить установку программы, кликнув по кнопке *Закрыть* (Рис. 1.10).

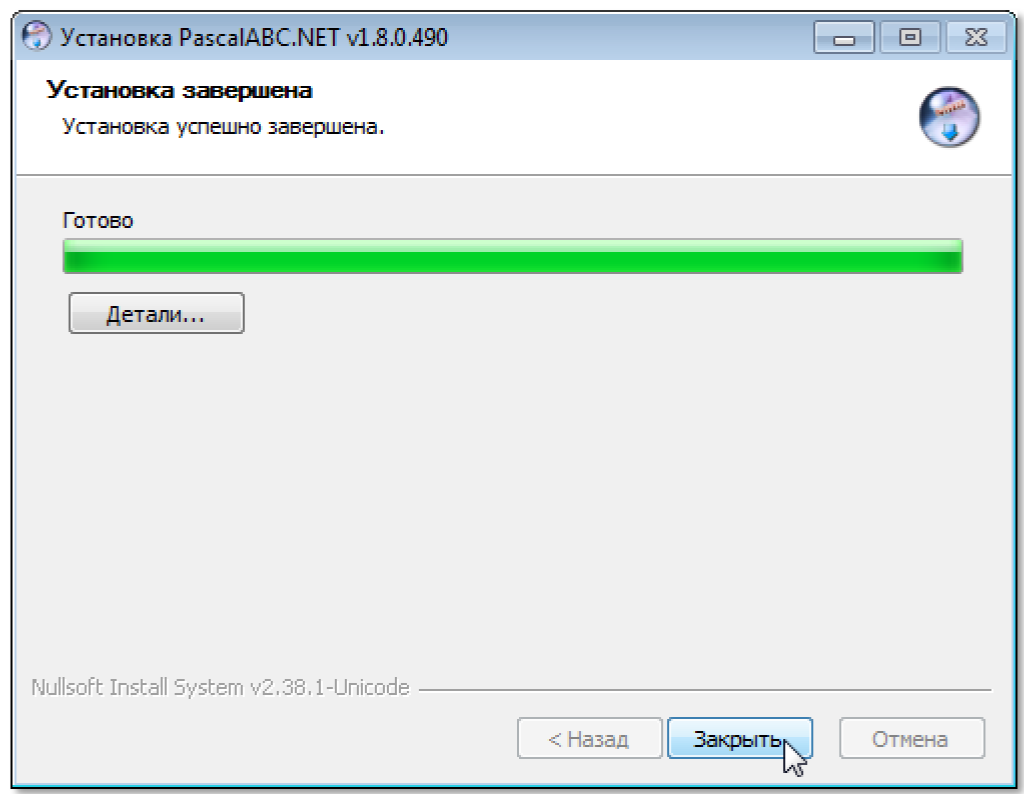


Рис. 1.10. Установка *паскаля* закончена

Вы можете открыть установочную папку и запустить файл *PascalABCNET.exe*, дважды щёлкнув по его значку, но лучше для этого использовать *ярлык* программы на *Рабочем столе* (Рис. 1.11).

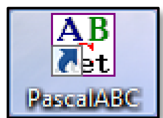


Рис. 1.11. Ярлык программы

И вот настал торжественный момент: мы дважды кликаем по ярлыку и в первый раз запускаем паскаль...

Время этого урока закончилось, с паскалем будем знакомиться на следующем уроке.

1. Обязательно найдите папку, в которую был установлен *PascalABC.NET*, и ознакомьтесь с её содержимым.

2. Запустите паскаль из папки, дважды щёлкнув по названию программы.

3. Закройте паскаль, нажав кнопку с крестиком в правом верхнем углу его окна.

4. Запустите паскаль с *Рабочего стола*.

Урок 2. Запускаем *PascalABC.NET*

Лиха беда - begin!

Программистская пословица

Обычная фирменная заставка у нашего паскаля отсутствует, но программа стартует так быстро, что она и не нужна, - всего через несколько секунд *PascalABC.NET* предстанет перед нами в полной боевой готовности (Рис. 2.1).

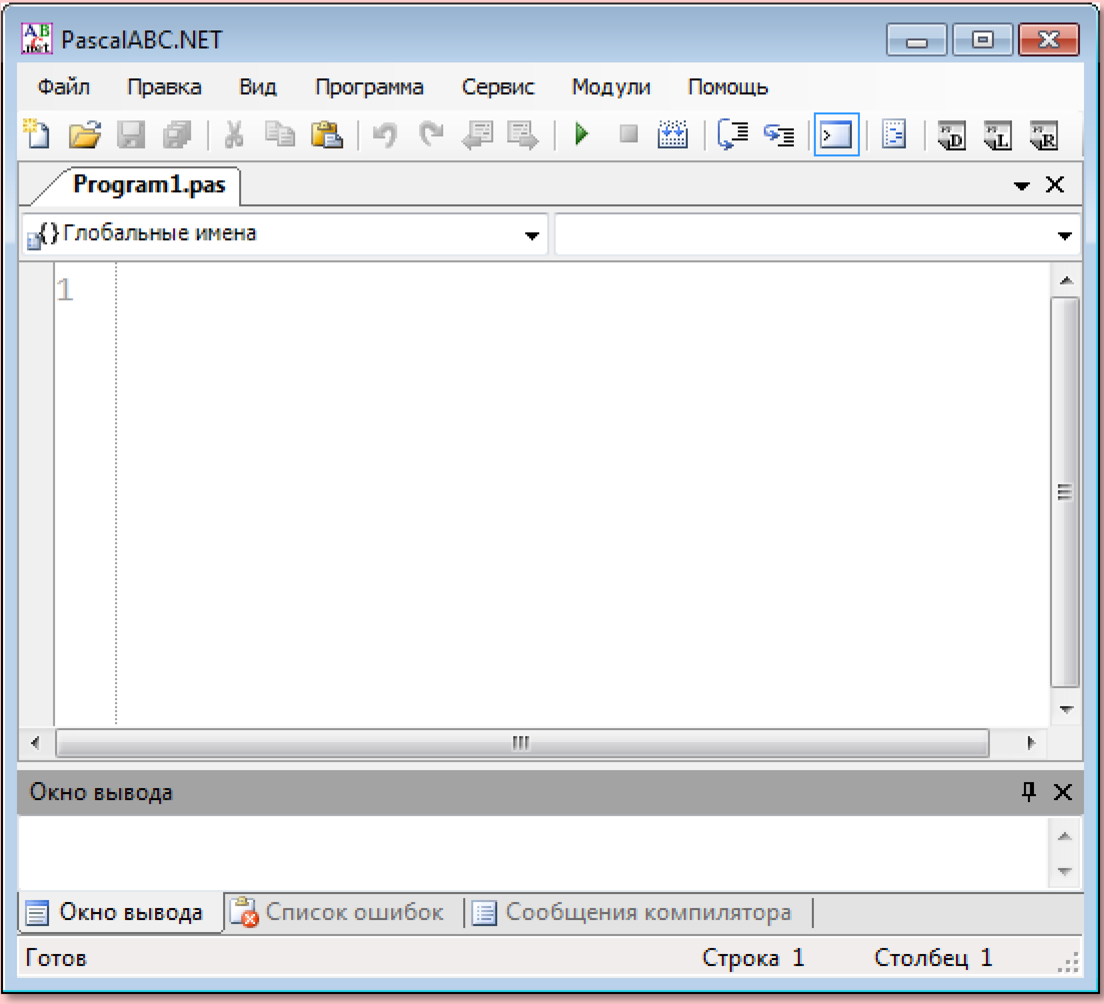


Рис. 2.1. Главное окно программы *PascalABC.NET*

И что особенно приятно: все надписи на *русском языке*!

Освоившись среди кнопок и окон, вы увидите пустой *документ* (или *проект,* или *программу*) под названием *Program1.pas*. Свою

первую программу мы напишем на следующем уроке, а пока давайте посмотрим, как работают чужие программы.
На диске *C*, в папке *PABCWork.NET* вы найдёте папку *Samples* с небольшими программами. Вы можете сразу загрузить их, ведь так хочется тут же, немедленно посмотреть, что умеет делать наш паскаль.

И тут нам пригодится кнопка *Открыть* (Рис. 2.2).

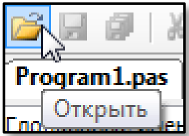


Рис. 2.2. Кнопка *Открыть*

Нажав её, мы получим в ответ диалоговое окно выбора файлов (Рис. 2.3).

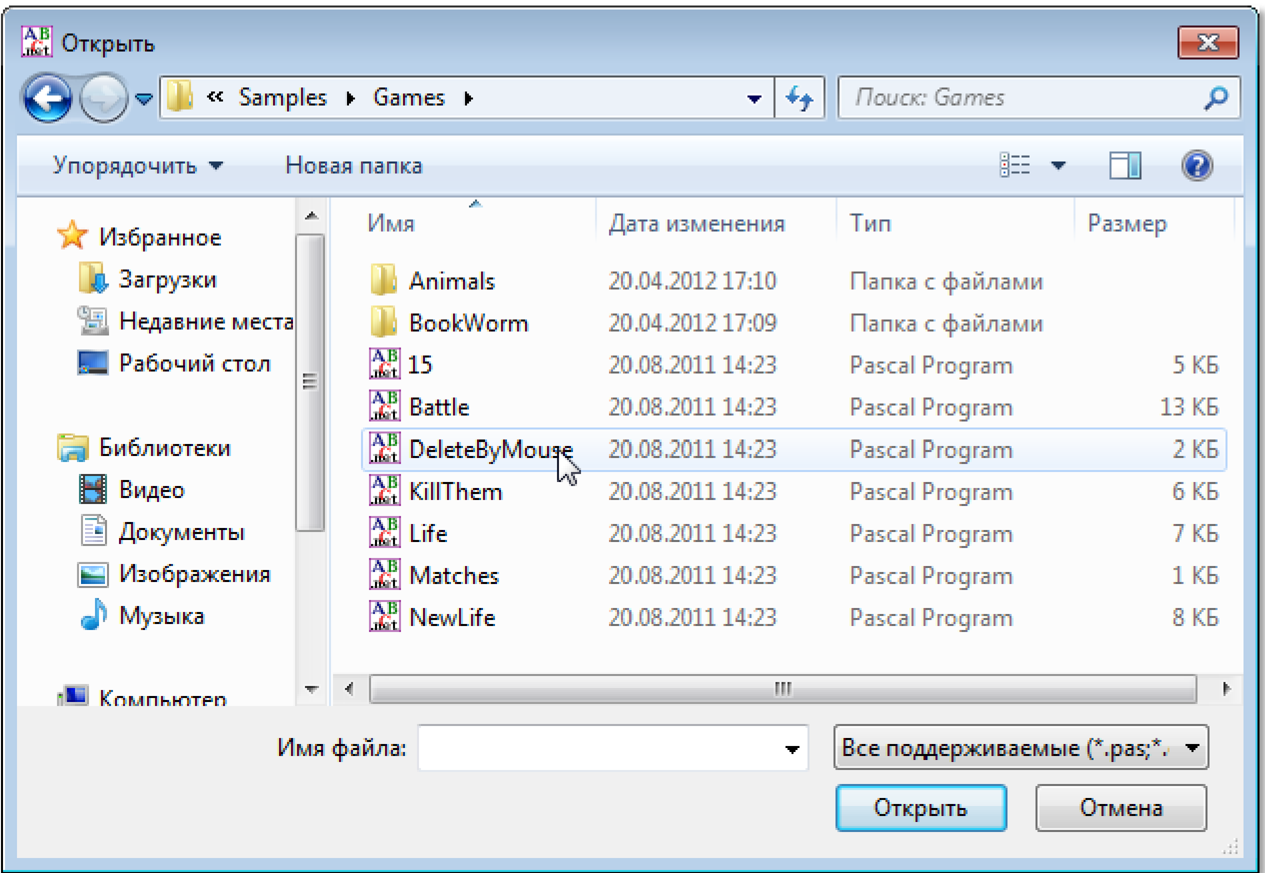


Рис. 2.3. Папка с демонстрационными программами

Перейдите в папку *Games* с играми. Для первого знакомства с паскалем нам вполне сгодится замечательная игрушка *DeleteByMouse*. Наставляем на её название курсор, нажимаем кнопку *Открыть* – и в *Редакторе кода* появляется выбранный нами файл с исходным кодом игры (Рис. 2.4). На закладке *Редактора кода* мы видим название загруженного файла - *DeleteByMouse.pas*.

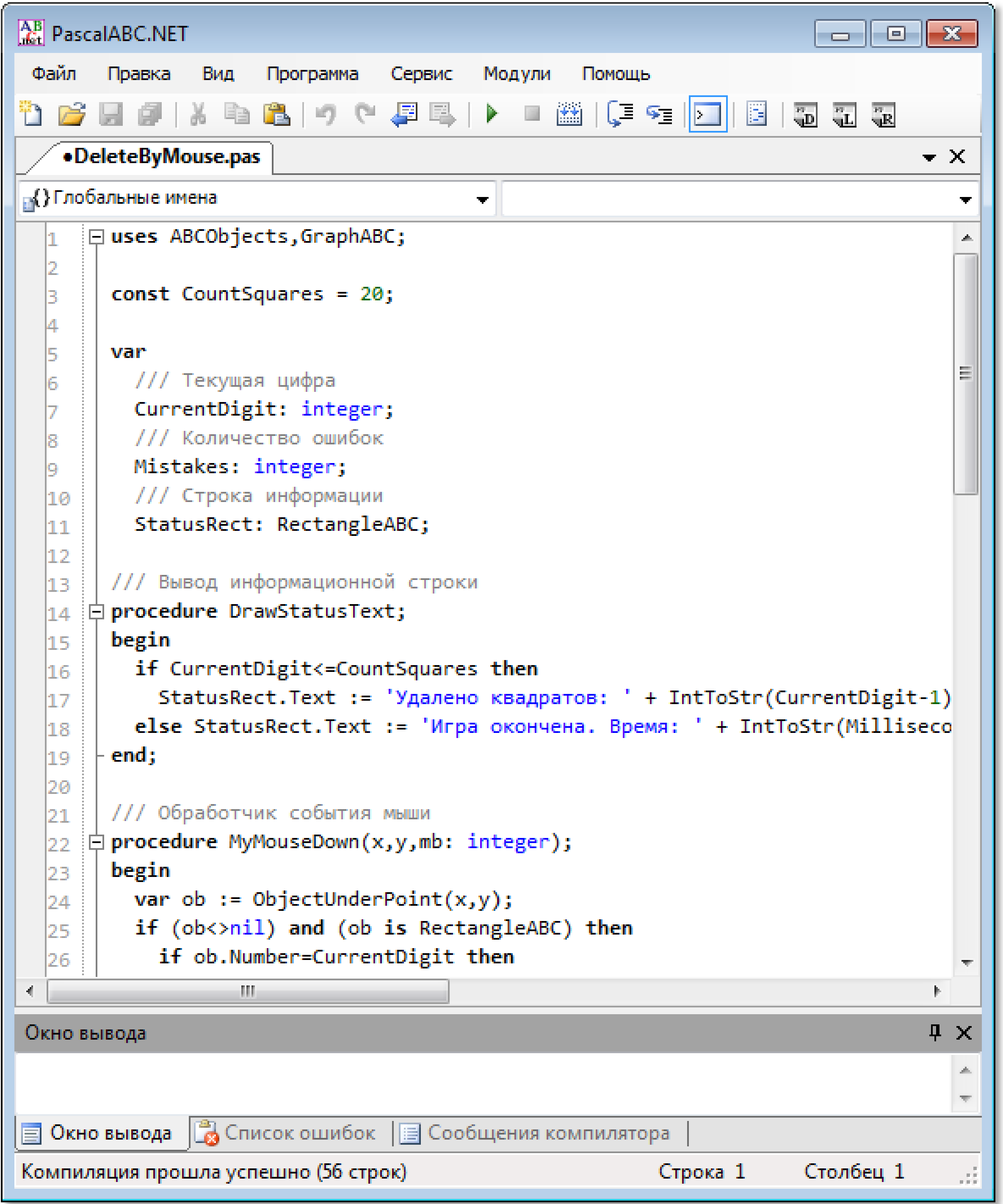


Рис. 2.4. Текст программы загружен!

Чтобы запустить программу, нужно нажать кнопку *Выполнить* с **зелёным** треугольником или клавишу *F9* (Рис. 2.5).

Рис. 2.5. Кнопка *запуска* программы

Через некоторое время на экране появится окно приложения, и вы сможете приступить к игре (Рис. 2.6). Она совершенно незатейливая: нужно щёлкать мышкой по квадратикам с числами, чтобы уничтожить их. При этом нужно начать с *единицы*, затем перейти к *двойке* - и так до самого *последнего* числа.

Рис. 2.6. Щёлкаем числа как орешки!

Обратите внимание, что после названия файла программы в квадратных скобках появилась надпись *Запущен* (Рис. 2.7), сообщая нам, что программа *выполняется*.

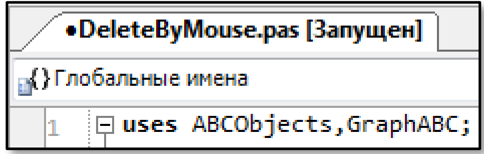


Рис. 2.7. Программа запущена

Чтобы *закончить* игру (*закрыть программу*), достаточно нажать на кнопку с крестиком в правом верхнем углу окна программы, то есть точно так же, как и при работе с любым другим приложением *Windows*.

При желании вы можете запустить и другие программы - вот такие игры вы сами сможете делать с помощью паскаля, но, конечно, не сразу: посмотрите на текст программы (*исходный код*, *листинг*) – он довольно длинный и наверняка совершенно вам непонятный.

Очень интересно посмотреть и на *графические* работы паскаля, которые собраны в папке *Fractals* (Рис. 2.8).

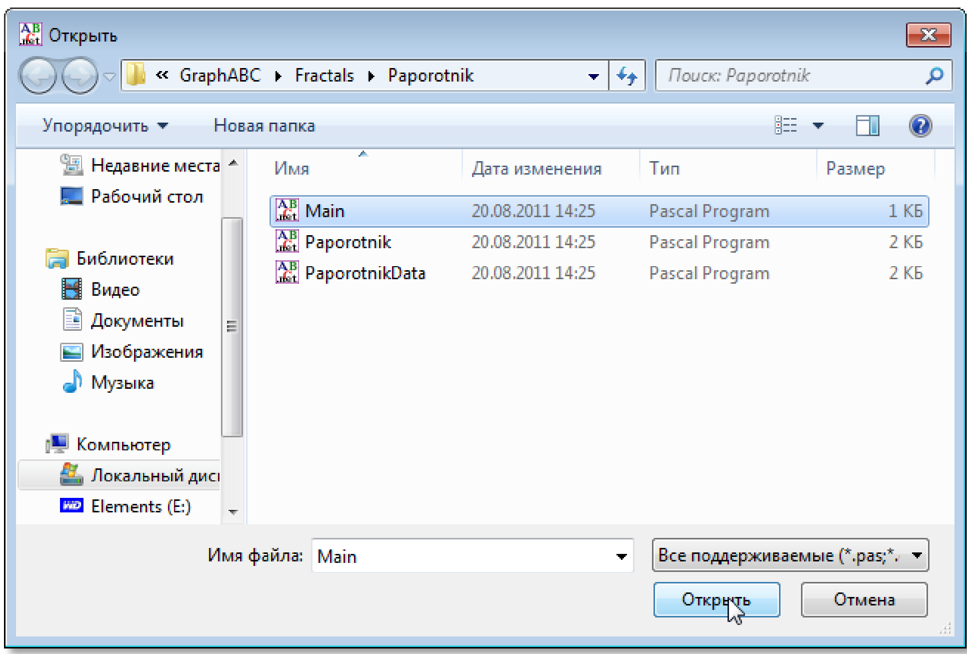


Рис. 2.8. Демонстрационные фракталы

Особенно красивы папоротники (Рис. 2.9).

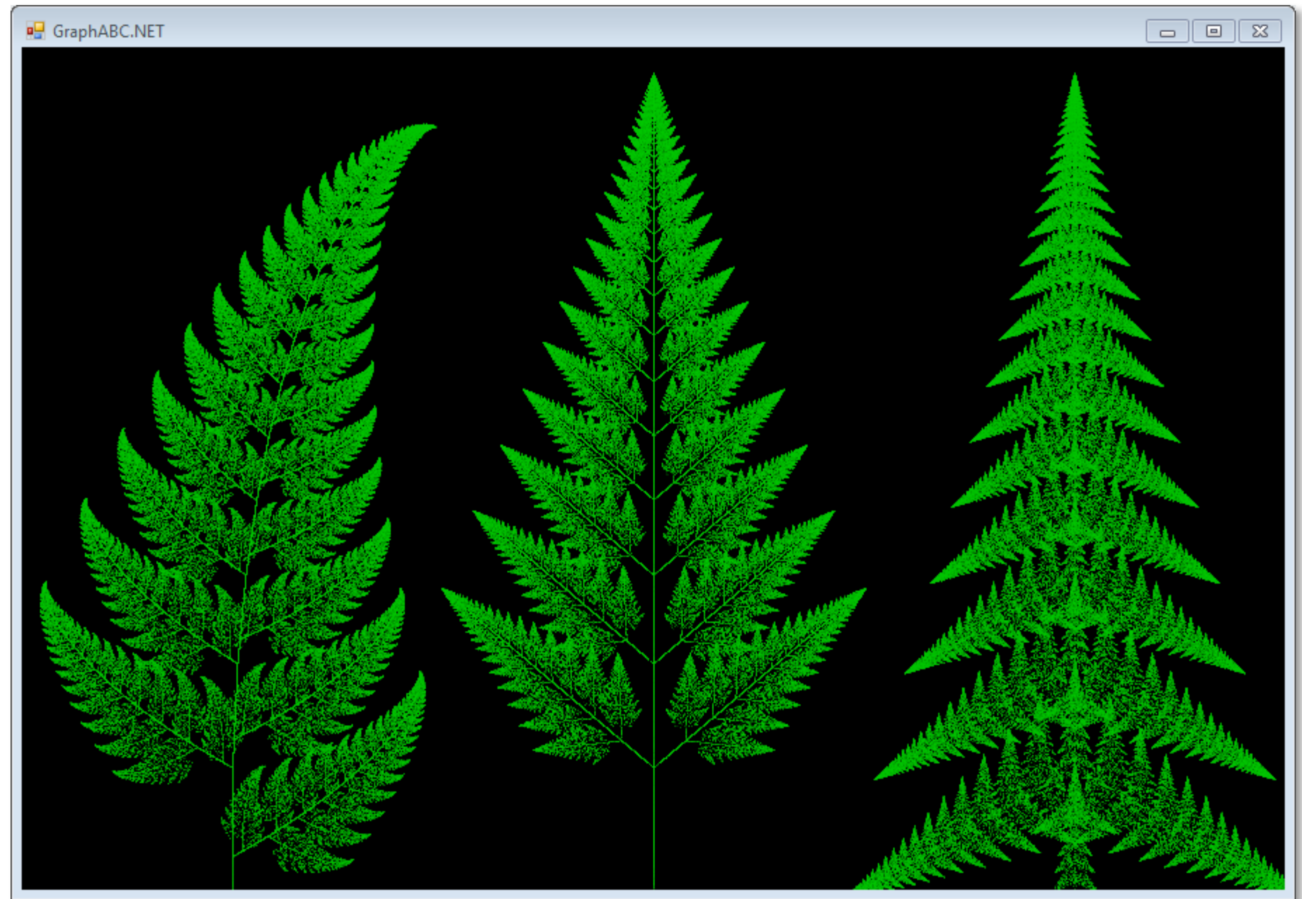


Рис. 2.9. Фрактальные папоротники

1. Загрузите другие программы и оцените возможности нашего паскаля.

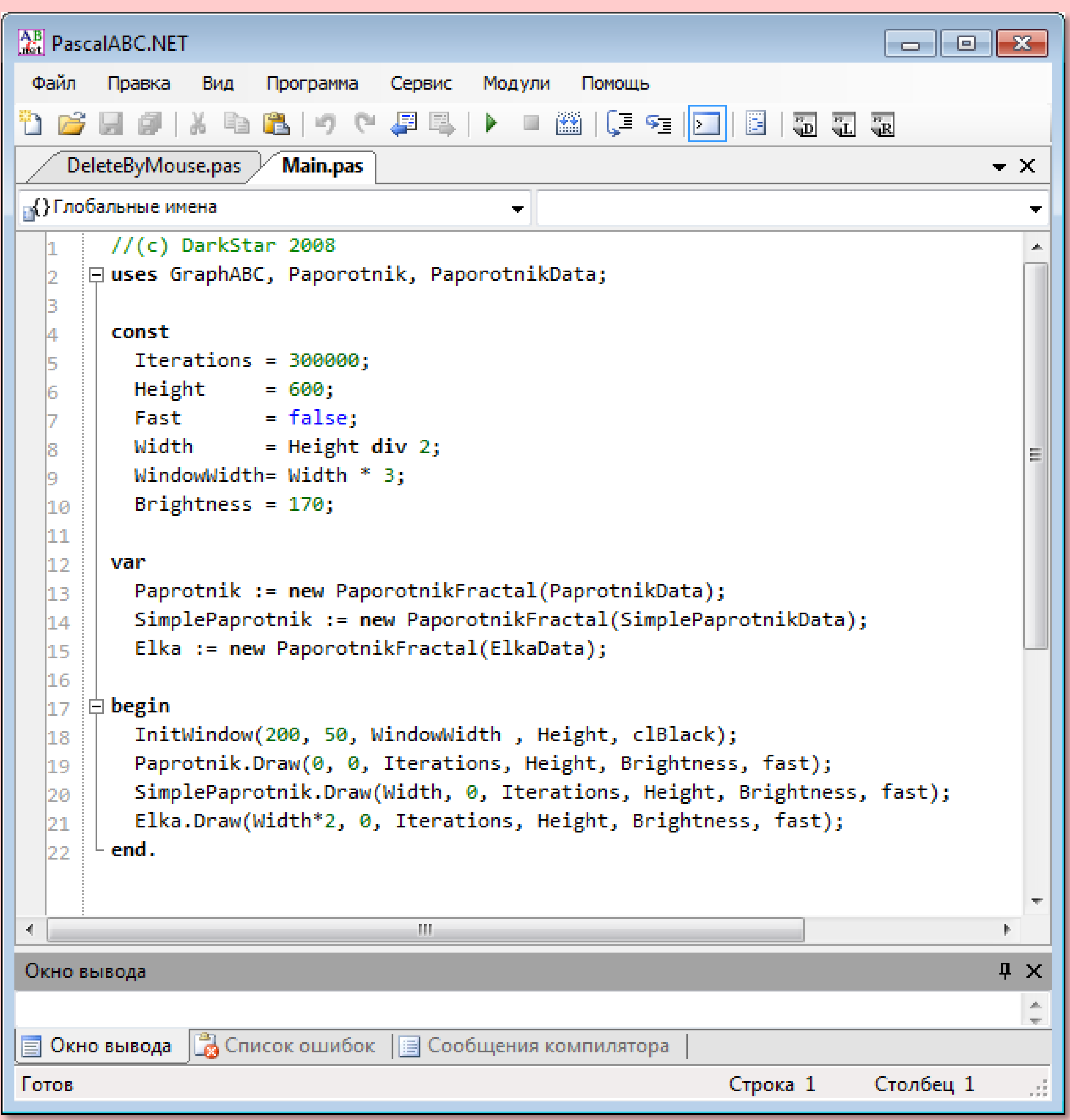
2. Посетите сайт *pascalabc.net*, где вы сможете найти ещё немало программ для изучения, а также много другой полезной информации.

Урок 3. Как сберечь программу

Сохраняйся!

Программистская пословица

Игра - дело хорошее, но у нас есть дела и поважнее! Если вы поиграли в «щелчки» и полюбовались папоротниками, то в *Редакторе кода* будут открыты *два* документа – *DeleteByMouse.pas* и *Main.pas* (Рис. 3.1).




```
//(c) DarkStar 2008
uses GraphABC, Paporotnik, PaporotnikData;

const
  Iterations = 300000;
  Height     = 600;
  Fast       = false;
  Width      = Height div 2;
  WindowWidth= Width * 3;
  Brightness = 170;

var
  Paprotnik := new PaporotnikFractal(PaprotnikData);
  SimplePaprotnik := new PaporotnikFractal(SimplePaprotnikData);
  Elka := new PaporotnikFractal(ElkaData);

begin
  InitWindow(200, 50, WindowWidth , Height, clBlack);
  Paprotnik.Draw(0, 0, Iterations, Height, Brightness, fast);
  SimplePaprotnik.Draw(Width, 0, Iterations, Height, Brightness, fast);
  Elka.Draw(Width*2, 0, Iterations, Height, Brightness, fast);
end.
```

Рис. 3.1. *Текстовый редактор (Редактор кода)* паскаля

В обоих - непонятные пока **разноцветные** слова, которые программисты называют *исходным кодом* (или *исходным текстом*) программы. Он написан латинскими буквами, но не на английском языке (хотя некоторые слова английские), не на немецком и даже не на французском. К сожалению, компьютеры пока не знают ни одного человеческого языка, а понимают только *машинный*, который состоит из нулей и единиц. Но на таком «тарабарском» языке невозможно написать даже самую простую программу, поэтому люди придумали *языки программирования*, чтобы с их помощью объяснять компьютеру, чт***о*** же он должен делать. Выучить язык программирования, например, паскаль тоже непросто, но компьютерный процессор не понимает даже его. Для перевода с языка программирования на язык машинный необходима программа-переводчик, которая называется *транслятором*. Трансляторы бывают двух видов: *интерпретаторы* и *компиляторы*.

Интерпретатор последовательно, строчку за строчкой просматривает исходный текст программы и передаёт соответствующие команды компьютеру. Если оператор языка программирования в какой-нибудь строке текста выполняется сто раз (а, может быть, и миллион!), столько же раз интерпретатор будет переводить текст в команды процессора. Нетрудно догадаться, что программа будет работать медленно. Чтобы ускорить процесс трансляции, иногда сначала весь исходный текст переводят в *промежуточный код*, который затем интерпретируется значительно быстрее.

Другой недостаток интерпретатора состоит в том, что для запуска любой программы необходим весь исходный код, а также сама программа-интерпретатор. То есть сначала нужно запустить программу, затем загрузить в неё исходный текст и только потом выполнить его. Конечно, это создаёт неудобства программистам при разработке программы. Да и поделиться с кем-то своей программой тоже непросто, ведь пользователь также должен установить на своём компьютере программу-интерпретатор и уметь ею пользоваться! Правда, у интерпретатора есть и небольшое преимущество - программа сразу же, без

предварительной обработки начинает выполняться, что очень важно при отладке.

Компиляторы работают по-другому: они *сразу* просматривают весь исходный текст и преобразуют его в *машинный код*, который процессор выполняет очень быстро. Но вот на компиляцию программы уже нужно некоторое время, поэтому при отладке придётся ждать, пока будет скомпилирована *вся* программа, хотя была изменена, может быть, всего одна буква (современные компиляторы, конечно, «умнее» и не перекомпилируют всю программу целиком). В результате компиляции программист получает *выполняемый файл* программы, который в операционной системе *Windows* называется также *приложением*. Его легко отличить от других файлов на диске по расширению *EXE* (сокращение от английского слова *execute* - *выполнять, исполнять*). Чтобы *запустить* приложение, достаточно дважды щёлкнуть по названию файла мышкой.

А теперь вопрос: наш паскаль - интерпретатор или компилятор? - Мы это скоро узнаем, но, прежде всего, вам следует приучить себя всегда *сохранять* исходный текст программы на диске!

О том, что мы не ещё не сохранили его, нам подсказывает *звёздочка* после названия документа.

Чтобы не портить демонстрационные программы, давайте создадим *новый* файл. Нажмите кнопку *Новый* или клавиши *Ctrl+N* (Рис. 3.2).

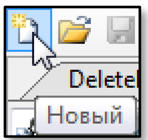


Рис. 3.2. Создаём новую программу

В *Редакторе кода* появится новая вкладка с названием файла по умолчанию – *Program1.pas*. Наберите любой текст с клавиатуры – и после названия файла появится звёздочка (Рис. 3.3.).

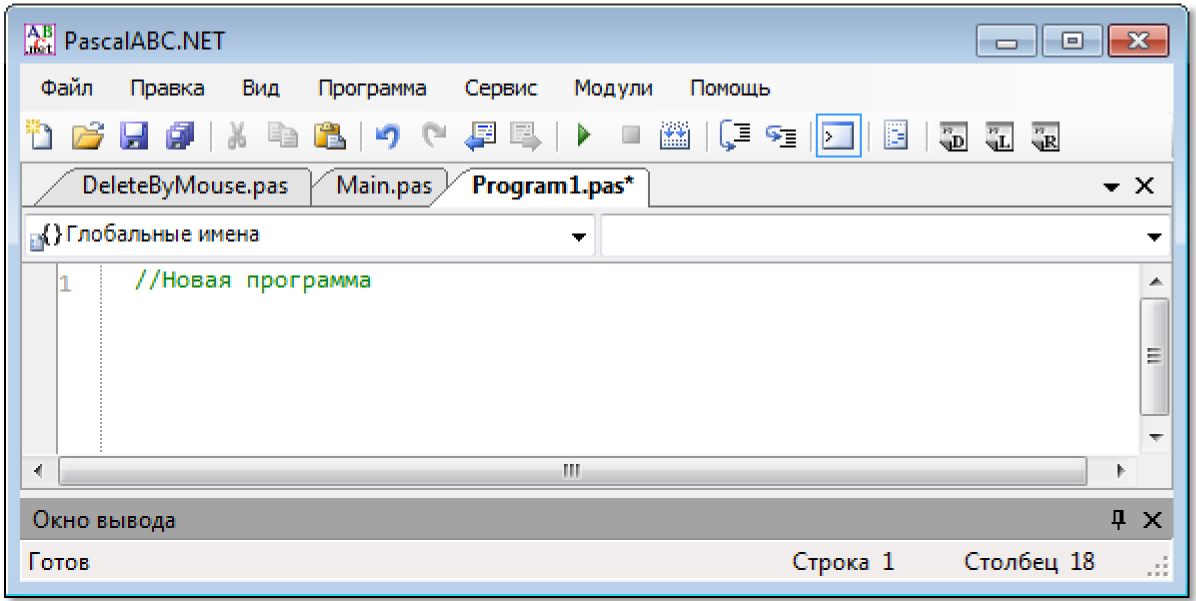


Рис. 3.3. Текст новой программы

Если вы сейчас же попробуете *закрыть* паскаль, то получите табличку с вопросом (Рис. 3.4).

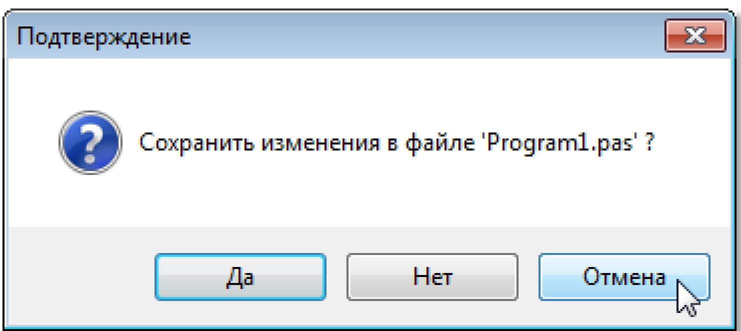


Рис. 3.4. Паскаль напоминает!

Это хорошо, что паскаль, как и фирма *Тефаль*, думает о нас, иначе вся программа (правда, мы ещё ничего не написали, но ведь напишем!) была бы безвозвратно потеряна. В данном случае лучше нажать кнопку *Отмена* и сохранить программу без напоминания. Для этого нажмите кнопку *Сохранить* с изображением дискеты или клавиши *Ctrl+S* (Рис. 3.5).

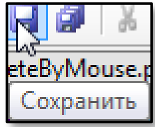


Рис. 3.5. Кнопка для сохранения исходного кода на диске

Откроется диалоговое окно (Рис. 3.6).

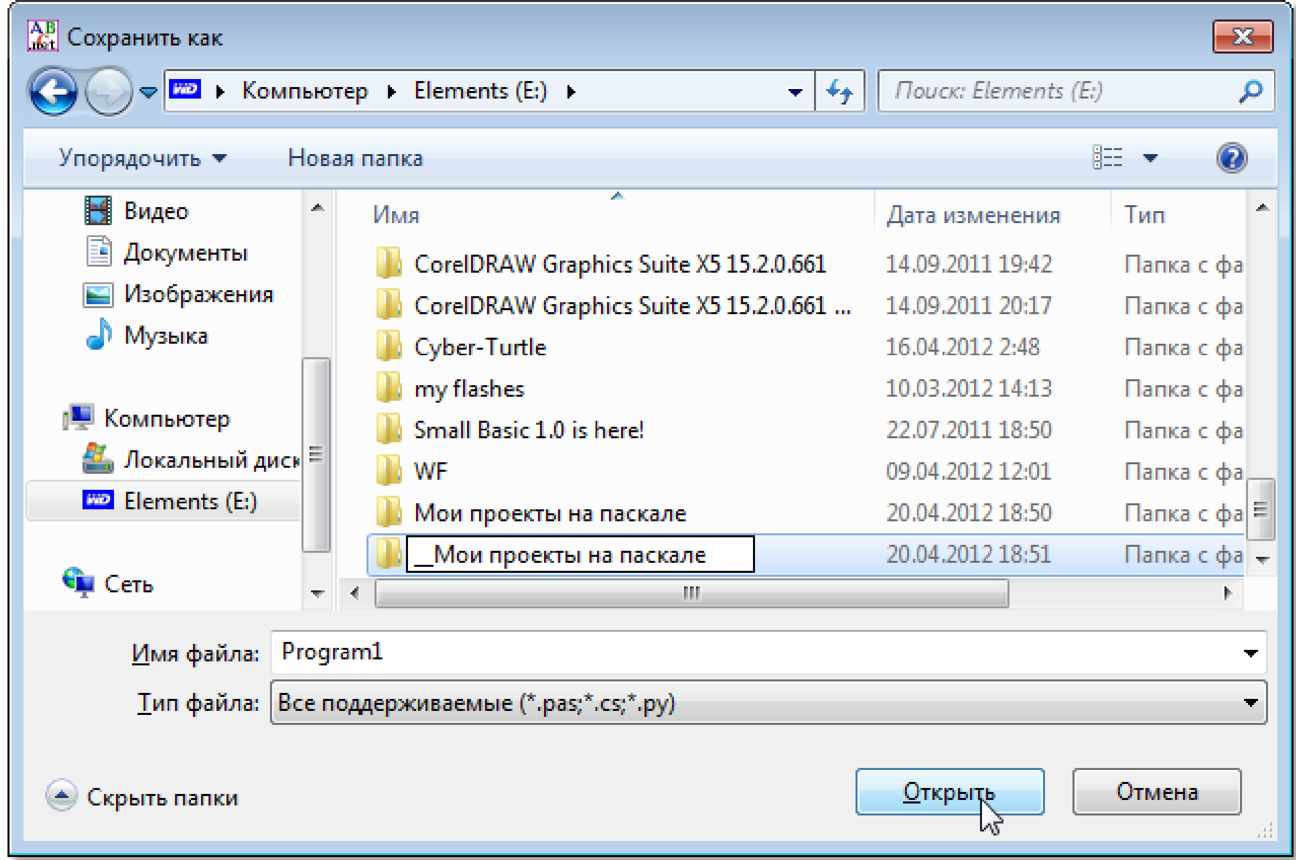


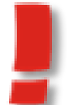
Рис. 3.6. Диалоговое окно для выбора папки и имени файла

В нём нужно выбрать папку для файла и его название.

Конечно, файл с исходным текстом можно сохранить в любом месте на диске. Но найдёте ли вы его потом среди множества других файлов? - Сомнительно! Поэтому для всех своих проектов вообще и на паскале в частности следует завести *отдельную папку* в корневом каталоге диска, тогда вам не составит труда эту папку найти. Например, папку со всеми проектами можно так и назвать *Мои проекты* (или *My Projects*). В ней (или отдельно) заведите папку *Мои проекты на паскале* (или *My PascalABC.NET Projects*). Еще лучше предварить название папки знаками *подчёркивания* (*_Мои проекты*), тогда эти папки всегда будут выше других папок в *Проводнике Windows* (или в файловом менеджере) и найти их будет ещё проще.

Итак, будем считать, что папку для проектов вы завели. Для каждого нового проекта в ней нужно создать *собственную папку*

с названием проекта. Например, наш проект естественно назвать *Новый*.



Всегда давайте проектам и файлам *вразумительные* имена, а не «ёклмнопрст»!

Ну вот, все папки готовы, осталось назвать сам файл и нажать кнопку *Сохранить*.

Готово! В папке проекта появился первый файл – *новый.pas* (Рис. 3.7).

e:__Мои проекты на паскале\Новый*.*

↑Имя	Тип	Размер	Дата
[..]		<папка>	20.04.2012 18:54
новый	pas	17	20.04.2012 18:54

Рис. 3.7. С почином!

Первая часть названия файла (до точки) это как раз то *имя*, которое мы выбрали, а вторая (после точки) - *расширение* файла, которое *паскаль* автоматически добавляет, чтобы отличать его от других файлов. Легко убедиться, что *паскаль* узнаёт свои файлы. Закройте среду разработки *PascalABC.NET*, найдите новую папку и дважды кликните по названию файла - *паскаль* снова запустится и автоматически загрузит исходный текст программы, но теперь документ будет называться *новый.pas*. Если вы подведёте к нему курсор мышки, то появится подсказка, в которой будет указан *полный путь* к файлу, включая название папки с проектом. Обратите внимание: звёздочки в конце названия документа нет. Это говорит о том, что файл был сохранён на диске.

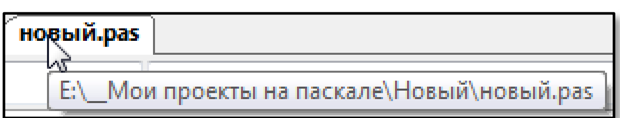


Рис. 3.8. Путь к файлу проекта

Но поставьте курсор в конец первой строчки программы и нажмите *ПРОБЕЛ* - звёздочка снова появится, сигнализируя о том, что документ *изменён* и, возможно, его следует сохранить

на диске. Если вы хотите записать проект в *новую* папку, то выполните команду меню *Файл > Сохранить как...* (Рис. 3.9). Но обычно файл сохраняют в той же самой папке, поэтому достаточно нажать кнопку *Сохранить (или клавиши Ctrl+S).*

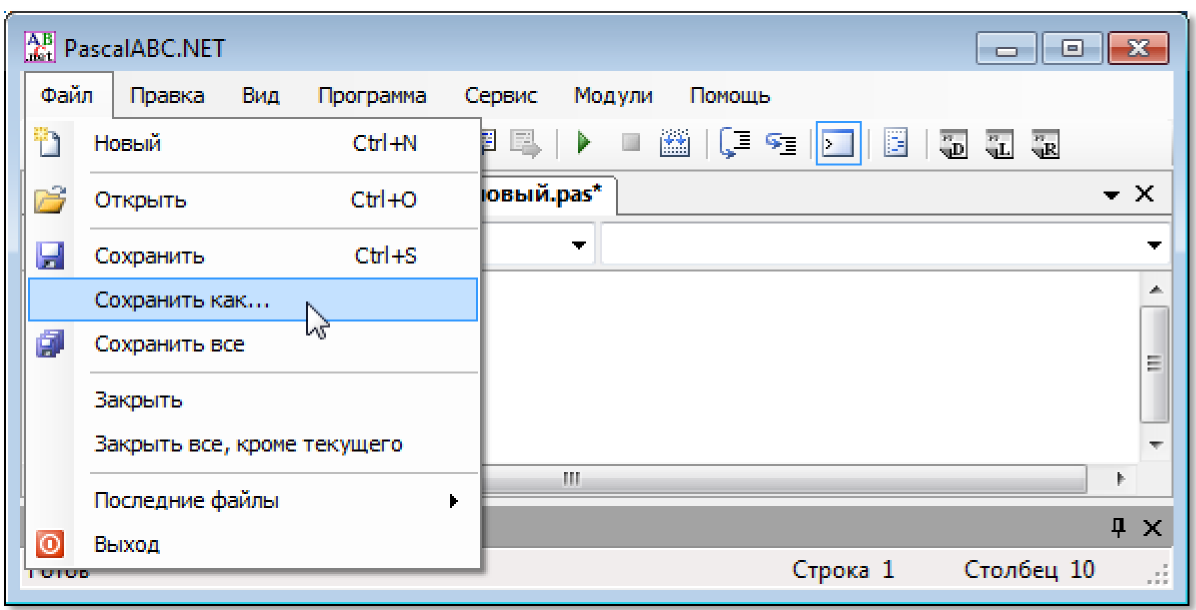
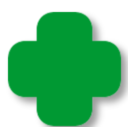


Рис. 3.9. Сохраняем файл в новой папке

Звёздочка, конечно, исчезнет, но первоначальный файл на диске будет *заменён* текущим. Имейте это в виду!



Кнопка *Сохранить* действует по-разному: при *первом* сохранении файла выполняется команда *Сохранить как...*, а при *последующих – Сохранить*.

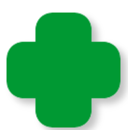
При разработке новой программы регулярно нажимайте кнопку *Сохранить*! Как говорится, жизнь полна неожиданностей, всякое может случиться, и ваш тяжкий труд может пойти прахом. Не беда, если вам снова придётся набрать несколько строчек текста, но вот если программа отлажена, вы нашли и исправили все ошибки, а сохранить её на диске забыли, то вам придётся ещё раз отлавливать всех *жучков* (слова *bug, баг, жучок* на программистском жаргоне означают *ошибку*, которую трудно найти). А отладка программы – это самая ответственная и трудная часть работы любого программиста.

В папке с проектом обязательно заведите папку для *архивных* файлов. Назовите ее *_ARCHIV*, или *_АРХИВ*, или как угодно

иначе, но она должна быть *обязательно*, если вы планируете достаточно долго работать над проектом. В этой папке периодически сохраняйте файлы, которые изменяются при работе, например, файлы *паскаля* с расширением *.pas*. Чтобы файлы занимали меньше места на диске, пользуйтесь *архиватором*. Сжатые файлы последовательно нумеруйте, чтобы всегда можно было вернуться к отлаженной версии программы, если вы наделаете ошибок при дальнейшей работе. Если проект ответственный, то сохраняйте его дополнительно на *другом* диске: вдруг вы случайно сотрёте папку или диск выйдет из строя… И результат месячной (а то и более!) работы над проектом придётся восстанавливать. Принято считать, что нет ничего хуже, чем ждать и догонять, поверьте: писать с самого начала уже готовую программу – куда хуже! Трепетно относитесь к своей работе и берегите её!

Паскаль запоминает последнюю папку, в которой сохранялся файл, поэтому вы сразу попадете в неё, если захотите загрузить файл с диска кнопкой *Открыть.*

Но давайте снова вернёмся к игре *DeleteByMouse* и запустим её, нажав клавишу *F9*. На этот раз мы не будем щёлкать по числам, а сразу закроем программу и полюбопытствуем, чт***о*** же произошло в папке с проектом. А там появился *новый* файл *DeleteByMouse.exe*, который даёт нам прямой ответ на наш прямой вопрос: *PascalABC.NET* умеет *компилировать* программы и создавать выполняемый файл.



Этот файл обязательно появится в папке, когда программа работает, а вот потом он может быть и стёрт с диска. Тут всё зависит от настроек *ИСР*. Об этом мы подробно поговорим на следующем уроке.

PascalABC.NET на самом деле создаёт не двоичный файл, который непосредственно выполняется процессором, а код на *промежуточном языке CIL* (по-английски - *Common Intermediate Language*), как и все другие компиляторы для

платформы *.NET*. Например, *C#, Managed C++, Visual Basic.NET, Visual J#.NET*.

Так как *PascalABC.NET* это не только язык программирования, но и редактор кода, и отладчик, и компилятор, то его называют *интегрированной средой разработки* программ, или сокращенно *ИСР* (по-английски - *Integrated development environment , IDE*).

Если вы захотите поделиться программой с другом, то скопируйте на его компьютер выполняемый файл программы. Также на его компьютере должна быть установлена платформа *Microsoft .NET Framework 4.0*.

Запись файла на диск занимает некоторое время, поэтому прежде чем запускать приложение, дождитесь окончания этой операции, иначе паскаль огорчит вас сообщением об ошибке!

Если файл нуждается в сохранении, то соответствующие кнопки на панели – цветные. Если же файл полностью записан на диск, они становятся серыми (Рис. 3.10).

Рис. 3.10. Активные и неактивные кнопки сохранения файла

1. Загрузите какие-нибудь программы в *PascalABC.NET*, воспользовавшись кнопкой *Открыть (или клавишами Ctrl+O)* с изображением папки (Рис. 3.11).

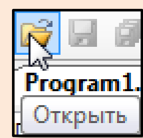


Рис. 3.11. Кнопка для загрузки исходного текста в *Редактор кода*

В появившемся одноимённом диалоговом окне (Рис. 3.12) перейдите в нужную папку и щёлкните по названию файла программы, чтобы выделить его, после чего нажмите кнопку *Открыть* (или сразу *дважды* щёлкните по названию).

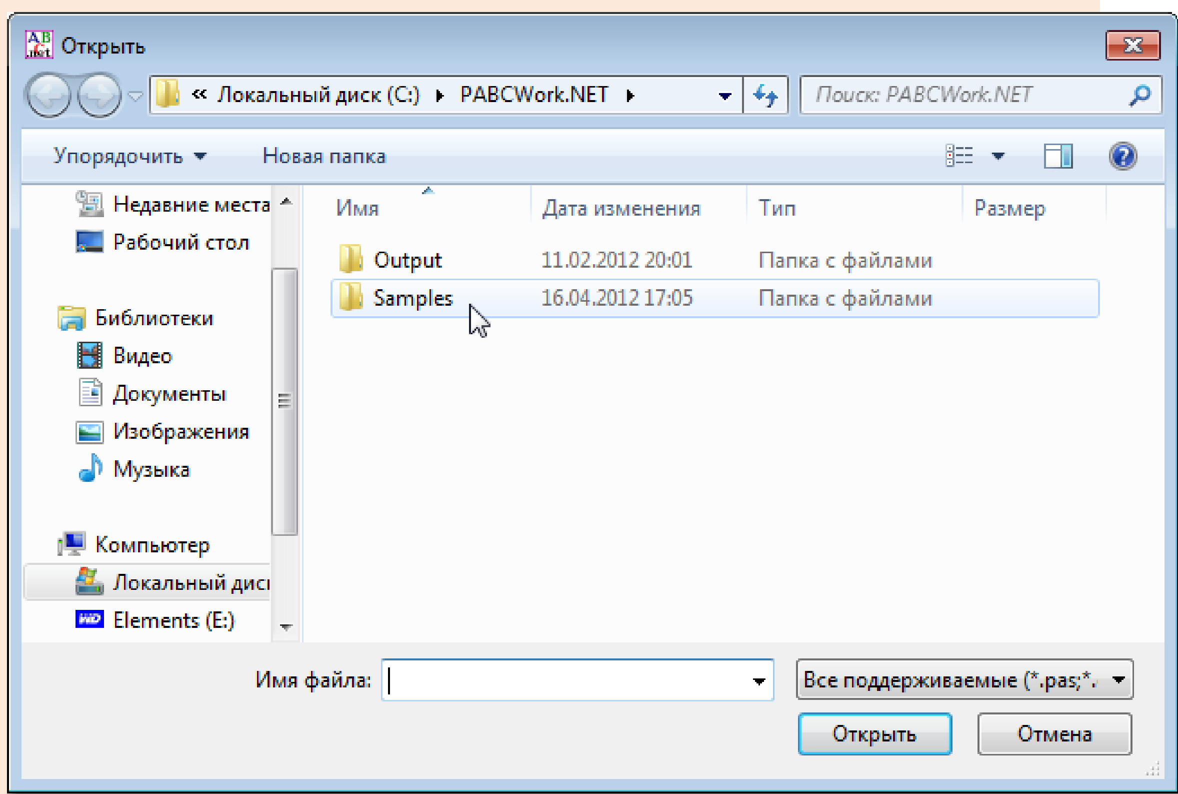


Рис. 3.12. Диалоговое окно для выбора программы

2. Когда в окне *Редактора кода* будет открыто *несколько* документов, вы можете любой из них сделать *активным*, просто щёлкнув по его закладке. Поупражнявшись с окнами, закройте их.

Урок 4. Наша первая программа!

Я поэт, зовусь я Цветик!
От меня вам всем приветик!

Мультфильм про Незнайку

В компьютерной литературе принято начинать обучение программированию с создания приложения, выводящего на экран надпись *Hello World!.* Мы не станем нарушать эту славную традицию, поэтому запустите *PascalABC.NET* и сразу же сохраните документ *Program1.pas* в новой папке *Hello* под тем же названием. Как вы помните, для этого следует нажать кнопку *Сохранить* или выполнить команду меню *Файл > Сохранить как...*, а затем произвести традиционные процедуры для сохранения нового файла. Название документа изменится на *hello.pas*, и мы, наконец, сможем заняться самым интересным в программировании - написать хоть и крохотную программу, но зато своими руками!

Наберите в *Редакторе кода* такой текст:

```
program Hello;

begin
  writeln('Hello World!');
end.
```

Цвет отдельных слов определяется самой *ИСР*, поэтому не ищите кнопок выбора цвета! А выделяются слова не столько для красоты, сколько для удобства ориентирования в исходном коде. Например, *комментарии* выводятся на экран **зелёным** шрифтом. Названия *объектов, методов* (*процедур, функций*), *ключевые слова* – **чёрным**, *переменных* – тоже **чёрным**, а их *значения* – **зелёным**. О смысле этих элементов любой программы мы ещё поговорим, но уже сейчас вы должны обратить внимание на то, что сходные по назначению слова выделяются одним и тем же цветом.

Вот и вся программа! Нажимаем кнопку *Запуск (F9)* и видим на экране скромные плоды нашего скромного труда (Рис. 4.1).

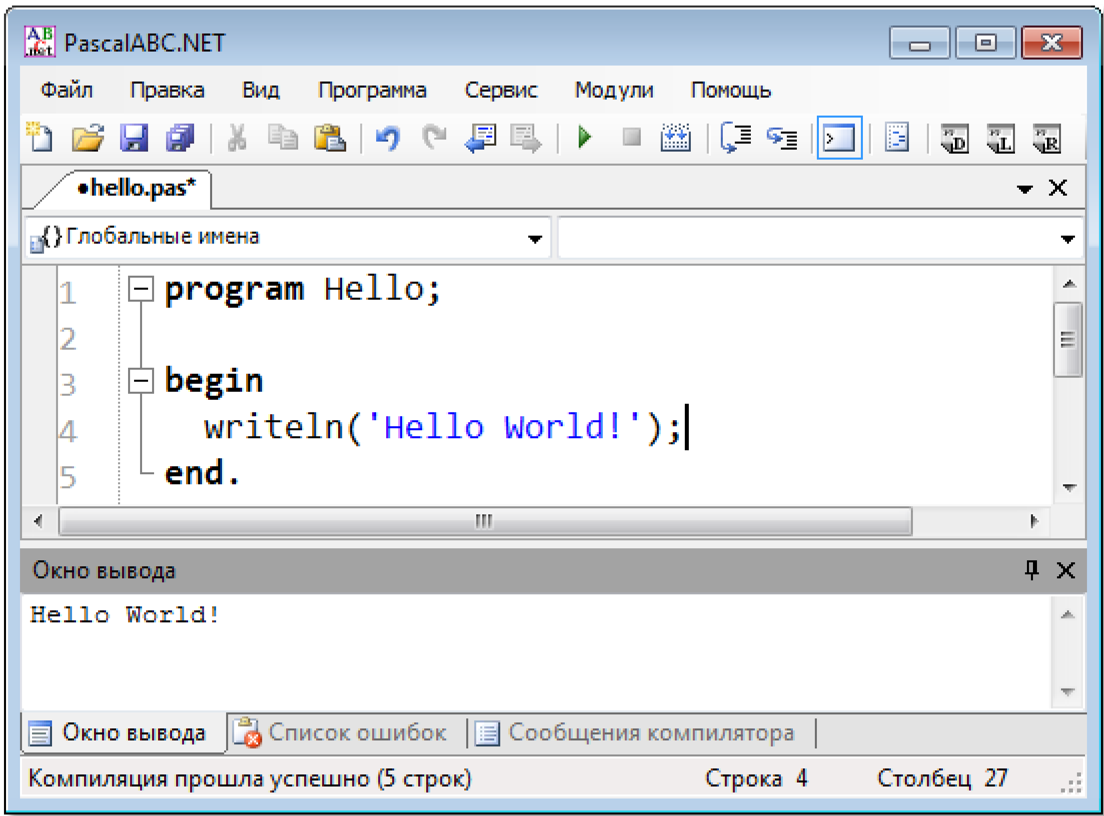


Рис. 4.1. Наша первая программа приветствует мир!

Мы написали сейчас *консольное* приложение. В среде *PascalABC.NET* в режиме *отладки* (а кнопка *Выполнить* запускает программу в *отладочном* режиме) информация выводится в *Окно вывода*, которое находится в нижней части *Главного окна ИСР*.

Чтобы увидеть работу программы в *настоящем* консольном окне, следует запустить программу в *автономном* режиме. Для этого нажмите клавиши *Shift+F9* или выполните команду меню *Программа > Выполнить без связи с оболочкой* (Рис. 4.2).

В этом случае будет создано консольное окно приложения (Рис. 4.3), но отладочный режим теперь уже не действует.

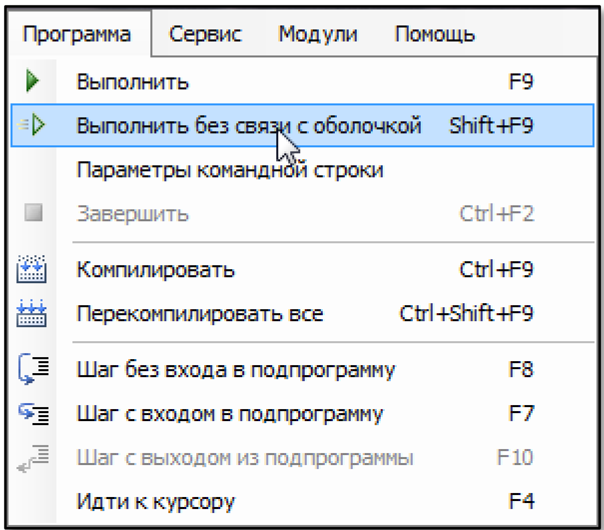


Рис. 4.2. Вывод на консоль

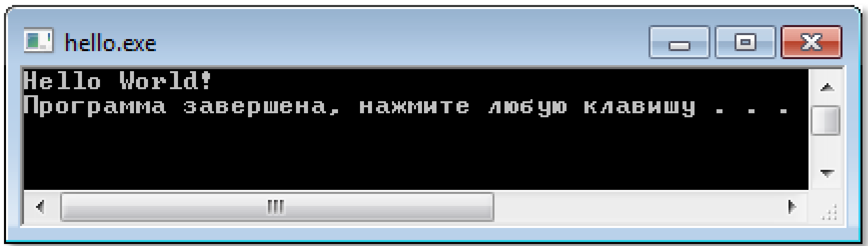


Рис. 4.3. Наша программа в настоящем консольном окне

Настройка ИСР

Прежде чем идти дальше, давайте настроим среду разработки так, чтобы в ней было удобно работать. Выполните команду меню *Сервис > Настройки* (Рис. 4.4).

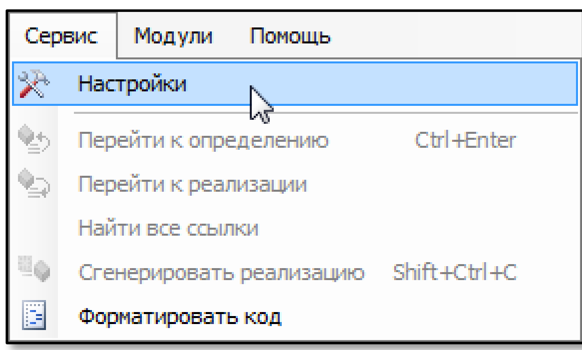


Рис. 4.4. Настраиваем ИСР

Откройте вкладку *Редактор*, установите флажки и задайте более крупный шрифт, чтобы не портить глаза (Рис. 4.5).

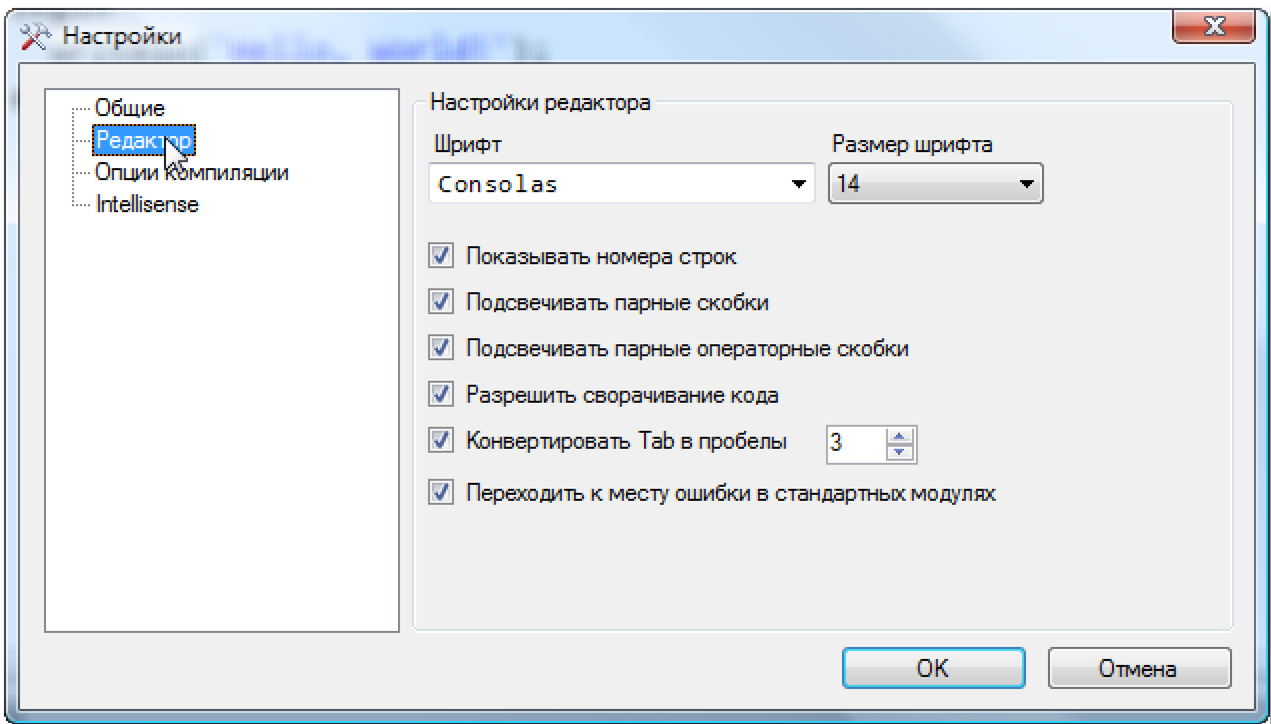
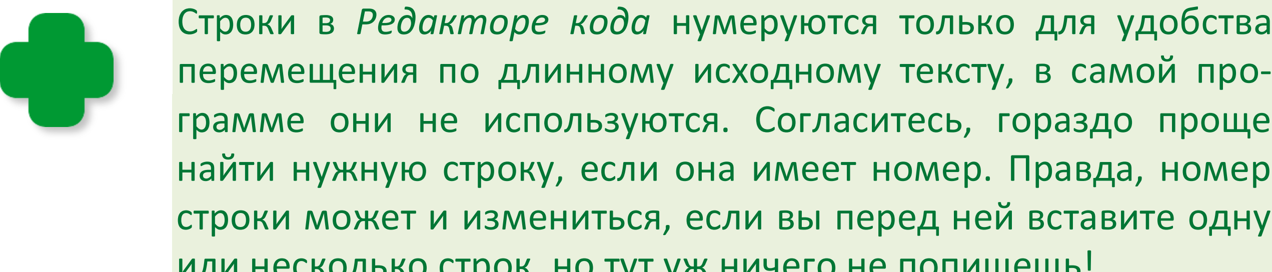


Рис. 4.5. Редактируем *Редактор кода*!



Строки в *Редакторе кода* нумеруются только для удобства перемещения по длинному исходному тексту, в самой программе они не используются. Согласитесь, гораздо проще найти нужную строку, если она имеет номер. Правда, номер строки может и измениться, если вы перед ней вставите одну или несколько строк, но тут уж ничего не попишешь!

Некоторые строки принято оставлять *пустыми*, чтобы отделять друг от друга смысловые части программы.

Как только вы нажмёте клавишу *ВВОД* (или *ENTER*), к тексту добавится ещё одна пустая строка, причем строки нумеруются последовательно, начиная с единицы (Рис. 4.6).

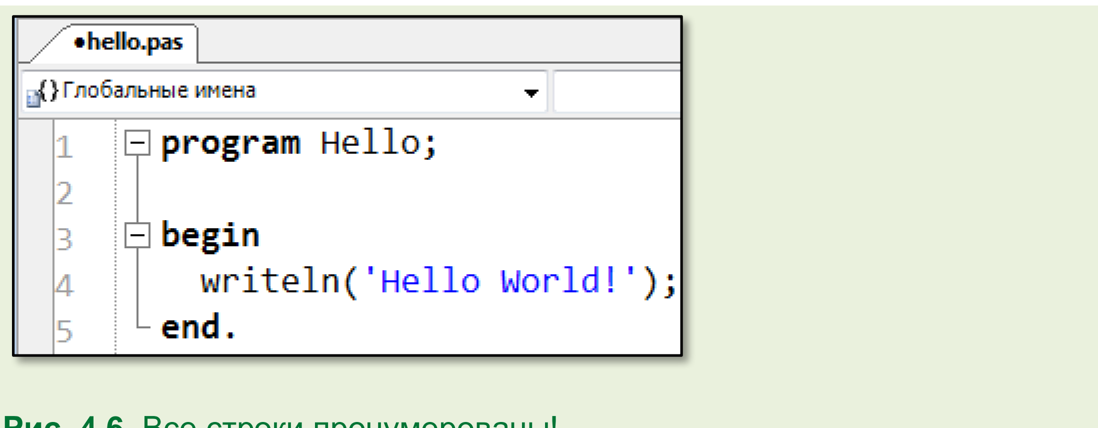


Рис. 4.6. Все строки пронумерованы!

На следующей вкладке – *Опции компилятора* – установите флажки так, как оказано на Рис. 4.7.

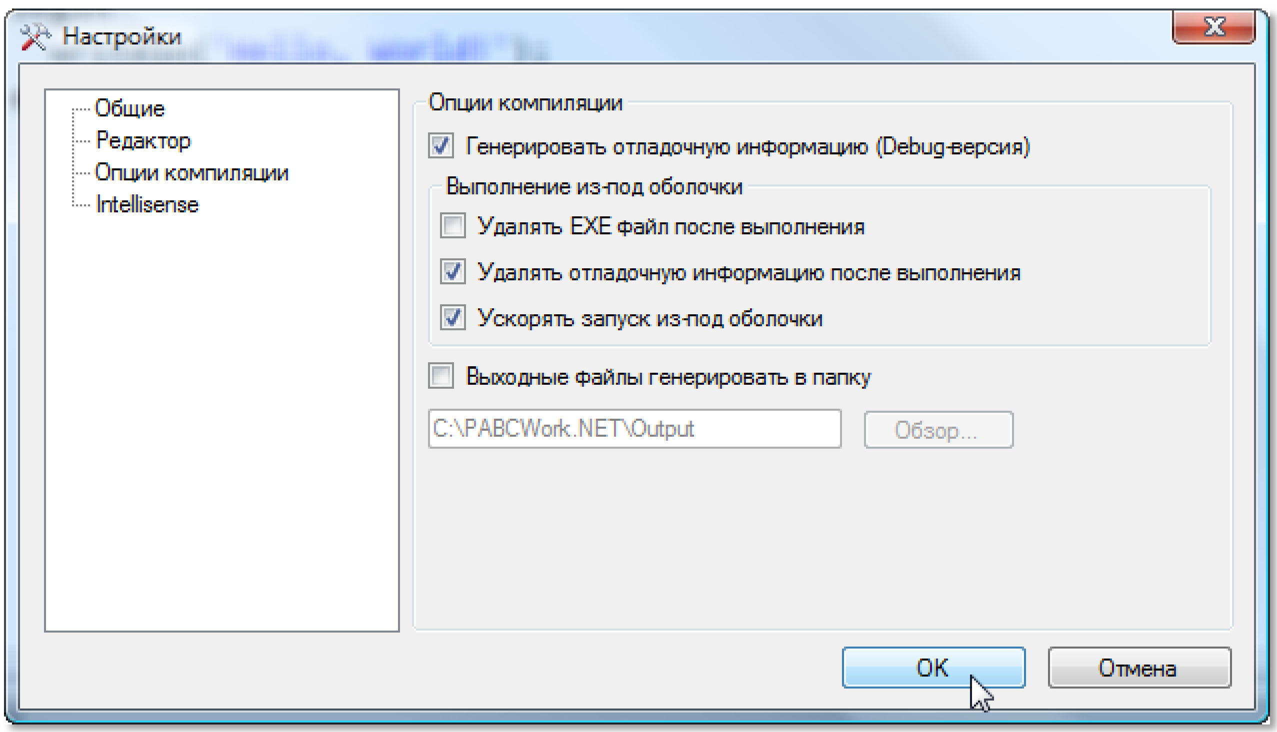
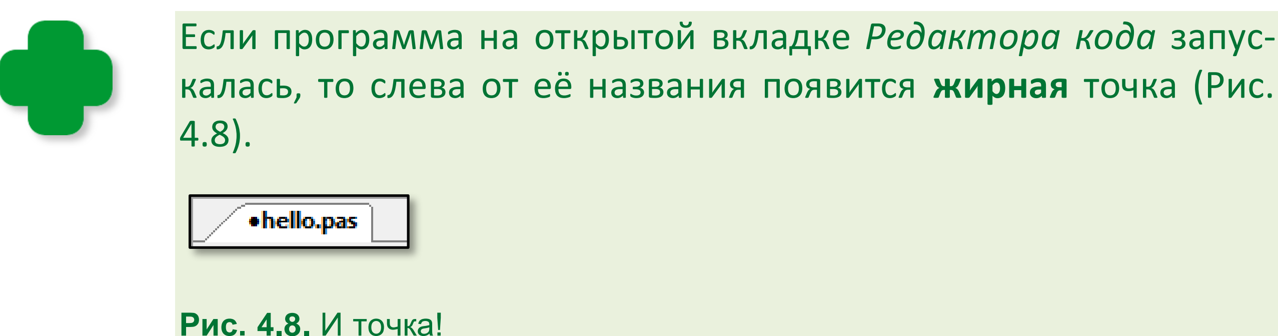


Рис. 4.7. Продолжаем благоустраивать *ИСР*

Теперь выполняемый файл программы с расширением *.EXE* после запуска программы останется в папке в целости и сохранности, и вы сможете запустить его на любом компьютере, на котором установлена платформа *.NET 4.0.* Однако работа программы на диске отличается от работы этой же программы в среде разработки – после вывода сообщения консольное окно тут же *закроется*, так что вряд ли вы успеете его прочитать. Допишите ещё одну строку к исходному коду:

```
program Hello;

begin
  writeln('Hello World!');
  readln();
end.
```

Теперь окно закроется только после нажатия на какую-нибудь клавишу.



Если программа на открытой вкладке *Редактора кода* запустилась, то слева от её названия появится **жирная** точка (Рис. 4.8).

Рис. 4.8. И точка!

И на последней закладке – *Intellisense* – установите *все* флажки (Рис. 4.9).

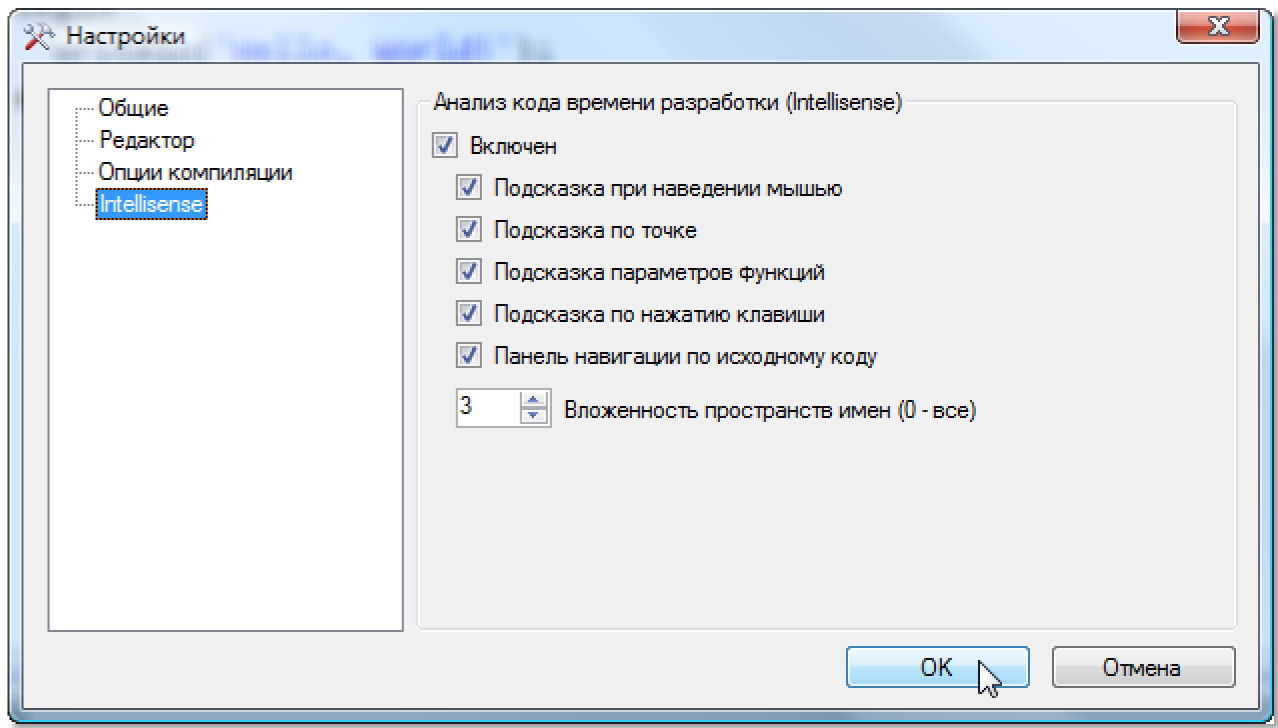


Рис. 4.9. Подсказываем подсказке!

Intellisense – это умная подсказка, которая облегчает набор исходного текста. О ней мы в своё время ещё поговорим.

Структура программы

По правилам паскаля, всякая программа должна начинаться с *заголовка*. Он состоит из двух слов – ключевого слова *program* (это слово по-английски и означает *программа*) и *названия* самой программы. В конце заголовка нужно поставить точку с запятой:

```
program Hello;
```

Но в большинстве современных версий паскаля допускается заголовок не указывать. *Закомментируйте* его и снова запустите программу – она будет работать точно так же:

```
//program Hello;

begin
  writeln('Hello World!');
  readln();
end.
```

После заголовка в более или менее сложных программах идут *объявления* модулей, констант, переменных и других элементов программы. Но они могут и отсутствовать, как, например, в нашей программе.

А вот потом обязательно следует *блок операторов* между *операторными скобками begin – end*.

В паскале роль операторных скобок исполняют *ключевые слова begin* и *end*, которые совсем не похожи на скобки. Но вот в языках *C++, C#* и других для этих целей используют *фигурные скобки*:

{

}

Назначение этих слов легче запомнить, если знать, что с английского слово *begin* переводится как *начало,* а *end* как *конец.* Назначение точки в конце программы понятно без объяснений.

Обратите внимание на *точку* после ключевого слова *end*! Она сигнализирует об окончании текста программы. После этой точки ничего писать нельзя!

Операторы (команды паскаля, завершающиеся точкой с запятой) между операторными скобками образуют *тело* программы. Ни одна программа на паскале, точно так же, как и мы с вами, не может существовать без тела, хотя в нём может не быть вообще ни одного оператора.

Иногда конструкцию

```
begin
   . . .
end.
```

называют *главной,* или *основной* программой, в отличие от подпрограмм и объявлений.

Сохраните изменения в программе, нажав кнопку *Сохранить* или *Сохранить все* (Рис. 4.10).

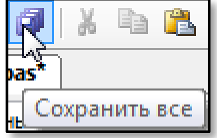


Рис. 4.10. Всесохраняющая кнопка

Кнопка *Сохранить все* записывает на диск не только активный файл, но и все остальные, открытые в *Редакторе кода*, так что будьте с ней осторожны, иначе вы можете случайно изменить файлы на диске!

Эти действия можно выполнить и в меню *Файл* (Рис. 4.11).

Однако вернёмся к нашей первой программе. Вы, должно быть, ожидали большего? - Тогда начните *новый* проект и запишите его в папку **Hello2**. Текст его не сильно отличается от прежнего:

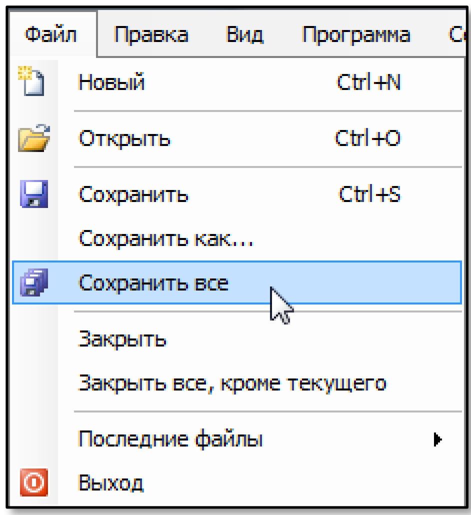


Рис. 4.11. Сохранение из меню

```
uses GraphABC;

begin
   TextOut(10,10,'Hello World!');
end.
```

Если вы забыли, как это делается, то нажмите кнопку *Новый* (Рис. 4.12) или клавиши *Ctrl+N*, и в окне *Редактора кода* появится новый пустой документ с названием по умолчанию.

Рис. 4.12. Кнопка *Новый*

Запишите файл на диск, предварительно создав для него отдельную папку.

Обратите внимание – в начале программы появилась строка

```
uses GraphABC;
```

Она сообщает *ИСР*, что мы хотим использовать в программе модуль *GraphABC*. Он необходим для того, чтобы создавать приложения с *графическим* интерфейсом. Запускаем программу на вы-

полнение - теперь появляется *графическое окно* с приветствием (Рис. 4.13).

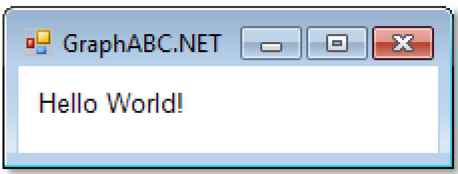


Рис. 4.13. Графическое окно

Этот вариант вас, наверное, удовлетворит полностью, особенно если учесть, что вы набрали всего четыре строки текста, а ваша программа уже успешно работает. Ни в одной другой среде разработки вы не сможете вот так запросто написать работоспособную программу! Это уже здо̀рово, но ведь вы легко можете добавить и другие строки к документу, которые затем появятся на экране. Например, вы можете поприветствовать мир и на родном языке, присовокупив к исходному тексту программы ещё одну строку:

```
uses GraphABC;

begin
   TextOut(10,10,'Hello, World!');
   TextOut(10,30,'Здравствуй, Мир!');
end.
```

Запускаем программу и получаем результат (Рис. 4.14).

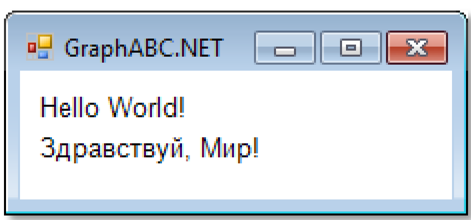


Рис. 4.14. Именно то, что мы хотели!

А заглянем-ка в папку с нашей программой - в ней появился новый файл приложения - *hello2.exe*. Если вы дважды кликнете по нему, то программа запустится, и на экране появится окно с приветствиями. А всё-таки ловкие мы ребята: ничего не зная о программировании, сумели написать полноценное приложение для *Windows!*

Одновременно будет создано и *консольное* окно. Чтобы от него избавиться, вместо кнопки *Выполнить* (*F9*), нажмите клавиши *Shift+F9* или *Ctrl+F9.* В последнем случае программа не запускается, а только компилируется в выполняемый файл на диске. Эта команда доступна также из меню (Рис. .4.15).

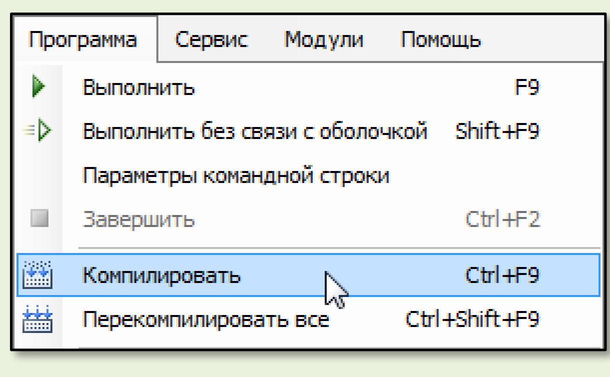


Рис. 4.15. Компилируем программу

Конечно, вы должны помнить, что среда разработки многое сделала за вас, потому что даже создание пустого окна *Windows* требует немало усилий со стороны программиста, а вы можете вывести пустое окно на экран с помощью всего трёх строк:

```
uses GraphABC;

begin
end.
```

Запускаем программу - и перед нами настоящее окно *Windows*, со всеми кнопками (Рис. 4.16).

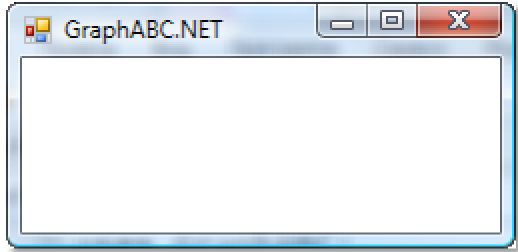


Рис. 4.16. Стандартное окно *Windows*

Вы можете перемещать его по экрану, изменять размеры, сворачивать и разворачивать и, в конце концов, закрыть, нажав на кнопку с крестиком.

Если вы не планируете использовать программу ещё раз, то сохранять её на диске не обязательно.

Как видите, создавать приложения для *Windows*, имея *PascalABC.NET*, очень просто.

Обратите также внимание на то, что сейчас в *Редакторе кода одновременно* открыты *два* документа, и вы можете легко переключаться между ними, просто кликая на нужном вам документе. Вы можете открыть *сколько угодно* документов (Рис. 4.17).

•hello.pas | hello2.pas

{} Глобальные имена

```
1   //program Hello;
2
3   begin
4     writeln('Hello World!');
5     readln();
6   end.
```

Рис. 4.17. Документы в *Редакторе кода*

Такой *многодокументный интерфейс ИСР* очень удобен при работе над несколькими проектами одновременно, потому что вам не придётся постоянно закрывать и открывать нужные вам проекты.

Чтобы изменить последовательность вкладок с документами, возьмитесь за вкладку мышкой и перетащите её в нужное место.

Одно или несколько окон вы можете использовать для *временного хранения* части кода разрабатываемой программы. Вырежьте или скопируйте несколько строк из основной программы и вставьте их во временное окно. Измените эти строки при отладке программы. Если новый вариант программы работает неверно, то вы легко сможете вернуть изменённые строки на место из временного хранилища.

В проектах *Hello* и *Hello2* при запуске программ мы получили два окна - одно **невзрачное** - *консольное*, второе **красочное** - *графическое*. Вам может показаться, что консольное окно вовсе не нужно, если есть графическое, но это не совсем так.

Раньше все программы были *консольными* и выводили информацию исключительно в *текстовом* виде. Не очень красиво, но тогда и компьютеры использовались только для серьёзных вычислений, так что результаты вполне можно было представить в виде строк, состоящих из слов и чисел. «Ну, это было давно!» - скажете вы, и опять будете неправы: и сейчас нередко результат работы программы достаточно вывести в текстовом виде. Например, если вы хотите узнать у компьютера, сколько будет дважды два, то вам совсем не нужно графическое окно. Пишете «программу»:

```
begin
   writeln('2 * 2 = ' + (2*2).ToString());
end.
```

Запускаете её и в консольном окне получаете результат (Рис. 4.18).

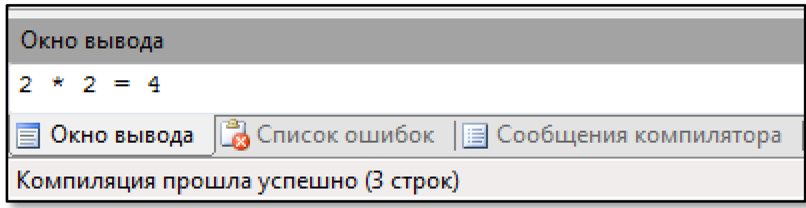


Рис. 4.18. Результат вычислений в *консольном окне*

Всё, вы удовлетворили свое любопытство!

Конечно, когда вы программируете на паскале, программа,

```
uses GraphABC;

begin
   TextOut(10,10,'2 * 2 = ' + (2*2).ToString());
end.
```

которая выводит тот же самый результат в *графическом окне* (Рис. 4.19), ничуть не сложнее, но если бы вам пришлось программировать на *C++*, то разница была бы ощутимой.

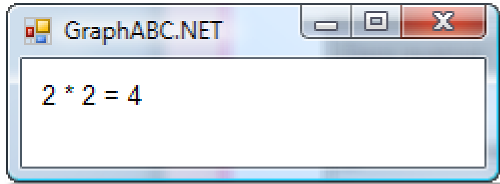


Рис. 4.19. Результат вычислений в *графическом окне*

Поэтому мы будем действовать, как настоящие программисты: если нам будет достаточно только увидеть результат вычислений, то мы будем создавать *консольное приложение* с текстовым окном, а если потребуется графика - приложение *Windows* с *графическим* интерфейсом.

Исходный код программ находится в папках **Hello и Hello2**.

Подсказка

Наверное, вы заметили, что при наборе строки появляется *подсказка* со списком допустимых объектов программы. Например, мы начинаем набирать первую строку программы и, как только мы нажмём клавишу *p*, сразу же увидим *всплывающее окно* подсказки (Рис. 4.20).

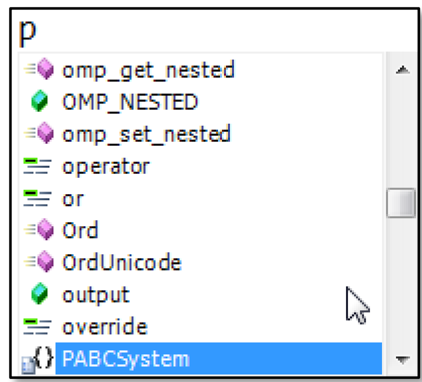


Рис. 4.20. «Интеллектуальная» подсказка

Так как *ИСР* не может знать заранее, какое слово мы хотим набрать, то она просто переходит на первую строку, начинающуюся с буквы *p*. Но нам нужно ключевое слово *program*, которое находится ниже в алфавитном списке, поэтому продолжаем набирать строку дальше (Рис. 4.21).

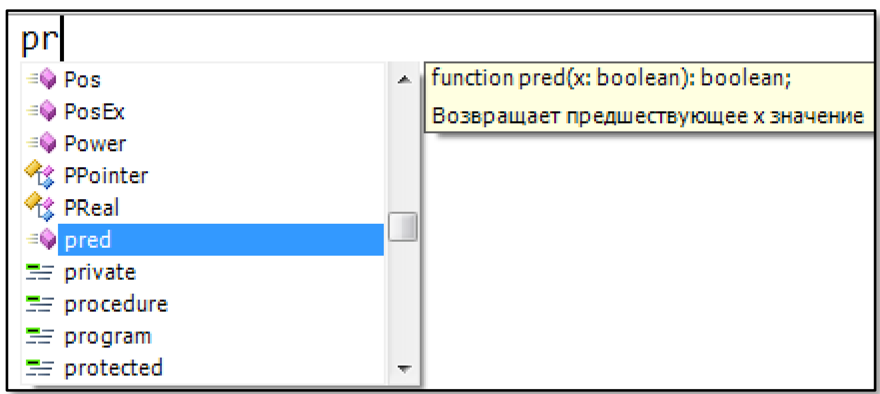


Рис. 4.21. Подсказка автоматически прокручивает список

И ещё дальше (Рис. 4.22).

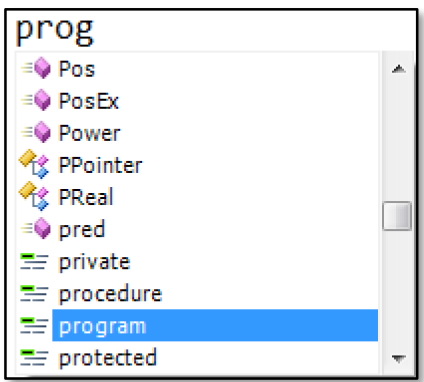
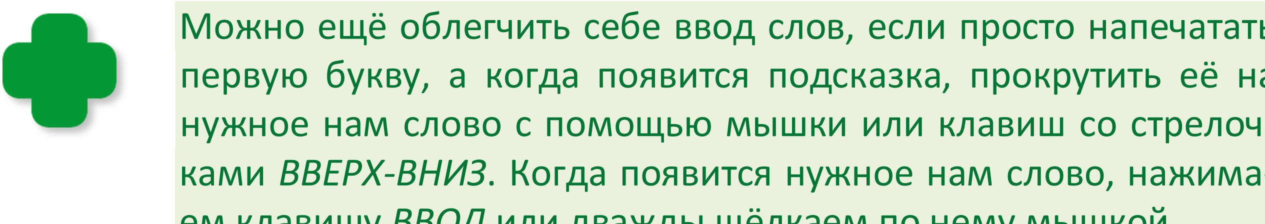


Рис. 4.22. Подсказка нашла нужное слово

Вот теперь в окне подсказки оказалось нужное нам слово, и мы можем не продолжать набор, а просто нажать клавишу *ВВОД*. Слово целиком появится в окне редактирования (Рис. 4.23), а подсказка исчезнет с экрана.

Рис. 4.23. Слово вставлено в строку программы

Заметьте: нам не пришлось до конца вводить довольно длинное слово, и напечатано оно без ошибок, которые мы могли бы сделать!



Можно ещё облегчить себе ввод слов, если просто напечатать первую букву, а когда появится подсказка, прокрутить её на нужное нам слово с помощью мышки или клавиш со стрелочками *ВВЕРХ-ВНИЗ*. Когда появится нужное нам слово, нажимаем клавишу *ВВОД* или дважды щёлкаем по нему мышкой.

Подсказку можно вызвать в любое время, нажав клавиши *CTRL+ПРОБЕЛ*.

Если в исходном коде программы имеются синтаксические ошибки, то подсказка может и не появиться. В этом случае нужно проверить текст.

Обратите также внимание на то, что справа от окна подсказки (Рис. 4.24) появляется информация о текущем объекте (он выделен **синим** фоном). Когда вы хорошо выучите все операторы паскаля, эта информация вам не потребуется, но на первых порах очень даже пригодится!

Название процедур и функций отделяется от названия модуля *точкой*. Как только вы поставите точку после слова *GraphABC*, снова всплывёт окно подсказки, в котором вы сможете выбрать свойство или метод (Рис. 4.24).

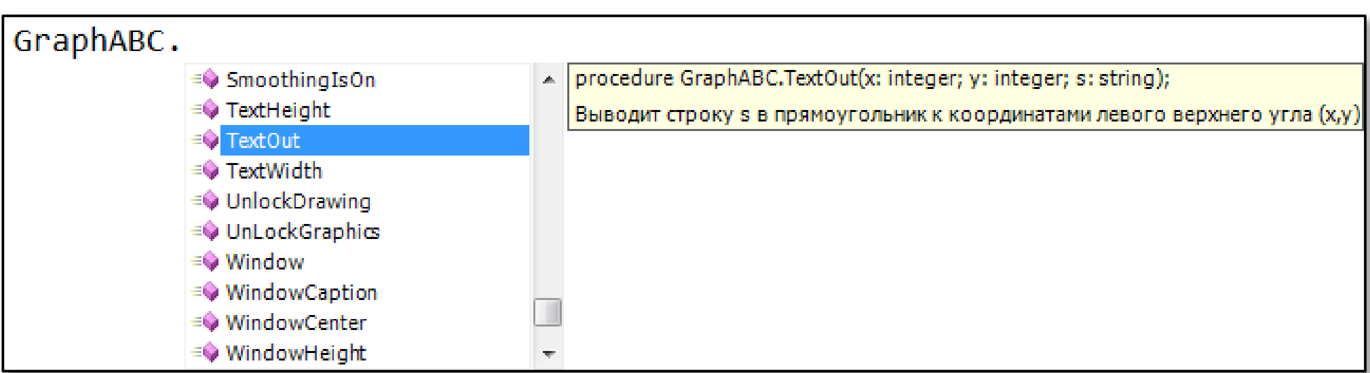


Рис. 4.24. Подсказка выводит список процедур и функций модуля *GraphABC* и информацию о процедуре *TextOut*

Здесь вы найдёте сведения, например, о методе *TextOut*. Вот так, прокручивая мышкой подсказку, можно выучить весь паскаль. Это, конечно, шутка: по словарю немецкого языка говорить не научишься. С паскалем ничуть не проще.

Название модуля в большинстве случаев можно и не писать, но если вы хотите получить список всех его элементов, то напишите название модуля и поставьте точку.

Если вы захотите получить помощь по любому объекту программы, который имеется в исходном коде, подведите к нему курсор мышки и прочитайте короткую справку (Рис. 4.25). Так что весь справочник по паскалю у вас всегда под рукой!

```
GraphABC.TextOut(10,10,'2 * 2 = ' + (2*2).ToString());
    unit GraphABC
    Описание:
      Модуль предоставляет константы, типы, процедуры, функции и классы для рисования в графическом окне
```

```
GraphABC.TextOut(10,10,'2 * 2 = ' + (2*2).ToString());
    procedure GraphABC.TextOut(x: integer; y: integer; s: string);
    Описание:
      Выводит строку s в прямоугольник к координатами левого верхнего угла (x,y)
```

```
' + (2*2).ToString());
 function integer.ToString(): string; virtual;
 Описание:
   Преобразует числовое значение данного экземпляра в эквивалентное ему строковое представление.
```

Рис. 4.25. Справка по модулю *GraphABC*, процедуре *TextOut* и функции *ToString*

Шаблоны кода

Для облегчения и ускорения набора текста в *ИСР* предусмотрена возможность вставки целых *фрагментов* текста (их называют *шаблонами* текста, по-английски - *snippets*). Для этого нужно набрать несколько начальных букв и нажать клавиши *Shift+ПРОБЕЛ*.

Чтобы начать программу в новом документе, мы набираем только **prog** и нажимаем указанные клавиши. Тут же появляется заготовка программы:

```
program Program1;

begin

end.
```

Имя программы совпадает с именем файла. Если вы предварительно запишете пустой документ на диск, например, под именем *test.pas*, то программа будет иметь имя *test*.

Как мы знаем, заголовок программы указывать не обязательно. На этот случай имеется другой шаблон. Набираем буквы **be**, нажимаем клавиши – и получаем «обезглавленную» заготовку программы:

```
begin

end.
```

Для приложений с *графическим* интерфейсом требуется модуль *GraphABC*. Набираем буквы **gr** и после нажатия на клавиши *Shift+ПРОБЕЛ* шаблон для графической программы готов:

```
uses GraphABC;

begin

end.
```

Конечно, такую «операцию» нужно провести только в самом начале разработки программы, но многие конструкции языка необходимо набирать много и часто. Особенно это относится к операторным скобкам. Но – достаточно напечатать только букву **b** и нажать клавиши *Shift+ПРОБЕЛ*, чтобы они заняли своё место в исходном тексте:

```
begin

end;
```

Очень часто приходится набирать слово *integer*, означающее в паскале целый тип. Переменные этого типа встречаются в программах чаще всего, поэтому для него также имеется шаблон кода, который срабатывает после ввода буквы *i*:

```
integer
```

Другие шаблоны кода вам пока будут непонятны, поэтому мы изучим их на следующих уроках.

1. Напишите ещё несколько коротких программ, выводящих в графическое и текстовое окно строчки или результаты арифметических вычислений.

2. Изучите работу с подсказками и шаблонами кода.

Урок 5. Какие бывают числа

Числа правят миром.

Пифагор

Как вы знаете из уроков математики, чисел бесконечно много, но все их можно разбить на отдельные *подмножества* по тем или иным признакам.

Самые первые числа, которые придумали ещё первобытные люди, называются **натуральными**. Они использовались для подсчёта различных предметов, например, яблок или палочек, на которых вы и сами учились считать в первом классе.

Папа спрашивает у сына: «Скажи, сколько будет, если к трём грушам прибавить ещё две груши?

Сын отвечает: «Не знаю, папа, мы в школе решаем задачи только про яблоки!

Множество натуральных чисел обозначается большой латинской буквой **N**, поэтому само множество можно записать так: N = {1, 2, 3, ...}. Иногда к множеству натуральных чисел относят и *нуль* (отсутствие предметов вообще): N = {0, 1, 2, 3, ...}. Множество натуральных чисел является подмножеством всех чисел и также бесконечно.

Если к натуральным числам добавить *отрицательные числа* (и нуль), то получится множество **целых чисел**. Оно обозначается большой латинской буквой **Z** = {... 0, -2, -1, 0, 1, 2, ...}. Нетрудно догадаться, что и целых чисел бесконечно много.

В арифметике обычно используют именно целые числа, но встречаются алгебраические и геометрические задачи, которые нельзя решить без *дробных чисел*.

Рациональные числа можно представить в виде *простой (обыкновенной) дроби* m/n, где

m - целое число;

n - натуральное число, не равное нулю (вы, конечно, помните, что на нуль делить нельзя!).

Множество рациональных чисел обозначается буквой **Q**. Если знаменатель дроби равен 1, то дробь равна числителю, то есть целому числу *n*. Таким образом, все целые числа являются в то же время и рациональными (множество целых чисел это подмножество рациональных). Но не наоборот!

Рациональные числа можно представить также в виде *конечной десятичной дроби* (1/2 = 0,5) или *бесконечной периодической десятичной дроби* (1/7 = 0,1428571...).

Продвигаемся дальше вглубь математики! **Иррациональные числа** не могут быть представлены в виде простой дроби (а также конечной или бесконечной десятичной периодической дроби), таким образом, иррациональным числом называют любое число, представимое в виде *бесконечной не периодичной десятичной дроби*. Примером такой дроби может служить корень квадратный из двойки. Иррациональность этого числа была известна уже древним математикам, которые доказали несоизмеримость стороны и диагонали квадрата.

Иррациональные числа обозначают буквой **I**.

Множество **действительных**, или **вещественных чисел** объединяет множества рациональных и иррациональных чисел. Их принято наглядно представлять в виде точки на *числовой прямой* (Рис. 5.1).

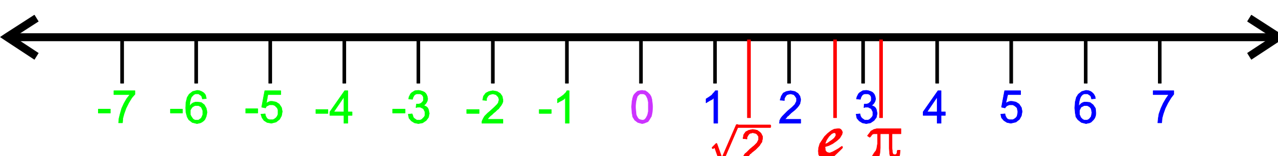


Рис. 5.1. Числовая прямая

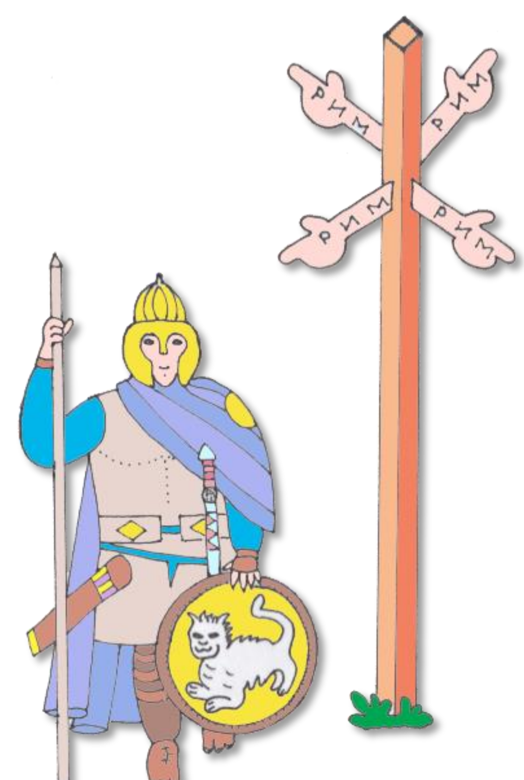
Множество действительных чисел принято обозначать буквой **R** (от их латинского названия *numerus realis*).

К иррациональным относятся знаменитые числа – π (отношение длины окружности к диаметру) и *e* (основание натуральных логарифмов).

Иногда в занимательных задачах присутствуют *комплексные* числа, но нам они не понадобятся.

С множествами чисел мы разобрались, но нас, конечно, интересуют не все числа вообще, а «особенные». И начнём мы с натуральных чисел, но записанных *по-римски*.

Римские числа



Все дороги ведут в Рим.

Из ответа на уроке географии

Древние римляне создали огромную, могущественную империю, которую назвали в свою честь, но что касается чисел, то тут они изрядно начудили и придумали столь вычурную систему обозначения чисел буквами, что даже простейшие арифметические вычисления давались им с большим трудом. Существенно облегчили школярам тяготы учёбы арабские - но придумали их индийцы! - цифры, пришедшие на смену римским.

Римская империя давно пала под натиском варваров, а их цифры сохранились до сих пор в первозданном виде. К счастью, нам не нужно пользоваться ими на уроках математики, но как украшательство мы можем найти их в книгах для обозначения глав, для подсчёта столетий, царей, съездов и других исторических событий.

Поскольку компьютер не понимает не только римских цифр, но даже арабских, то мы напишем простую программу **Rome**, которая поможет нам разобраться в премудростях римской нумерации.

Как это будет по-римски?

Для ввода и вывода информации нам вполне хватит *консольного* окна, но мы его немного *раскрасим*. Но это потом, а пока давайте позаботимся о **переменных**, которые нам пригодятся в программе.

Переменные – это элементы программы, которые *могут изменять своё значение* в ходе её выполнения.

Физически переменная представляет собой область памяти компьютера, в которой хранится значение переменной. Все переменные должны иметь *имя* (*идентификатор*).

Запустите *PascalABC.NET* и сохраните новый проект в папке **Rome** под тем же названием. Начало положено, но документ совершенно пуст, поэтому начнём наполнять его «контентом».

Нам понадобятся *именованные константы* - для хранения некоторых римских и арабских чисел.

Константы – это элементы программы, которые *не могут изменять своего значение* в ходе её выполнения.

Все числа являются *числовыми константами*. Из чисел, знаков операций, идентификаторов методов и свойств объектов, а также круглых скобок можно составлять *константные выражения*:

```
123
 "Z"
Math.Cos(1)
“string1”
Text.GetCharacter(111-32)
```

Именованные константы должны иметь идентификатор (имя), как и переменные. Неименованные константы называют также *литералами*.

Константы объявляют в *начале* программы, причём одновременно с их *инициализацией*, то есть им нужно сразу же *присвоить значение*. *Раздел описания констант* начинается с ключевого слова *const* (от англ. слова *constant* - *константа*). Таких разделов может быть несколько, но обычно все константы программы объявляют в *одном* разделе.

Мы соберём все константы в два *массива* и предварим их комментарием:

```
//ПРОГРАММА ДЛЯ ПЕРЕВОДА АРАБСКИХ
//ЧИСЕЛ В РИМСКИЕ

const
  ROME: array[1..13] of string = ('M', 'CM', 'D', 'CD', 'C', 'XC',
                                  'L', 'XL', 'X', 'IX', 'V', 'IV', 'I');
  ARABIC: array[1..13] of integer = (1000, 900, 500, 400, 100, 90, 50,
                                     40, 10, 9, 5, 4, 1);
```

Все **комментарии** начинаются с двух косых (дробных) черт (слешей) и могут располагаться в начале строки или после оператора, но в любом случае его действие распространяется до *конца* строки, то есть после комментария нет смысла записывать операторы — они всё равно выполняться не будут. Комментарии выделяются **зелёным шрифтом**, поэтому их легко распознать в исходном тексте.

Более подробно о комментариях читайте здесь.

Теперь объявим все *переменные* нашей программы:

```
var
  number: integer;
  sNumber: string;
  n: integer;
```

Раздел описания переменных помещают *после* раздела констант, он начинается с ключевого слова *var* (от англ. слова *variable* - *переменная*).

Проясним их *назначение*. Первая переменная - *number* - это как раз то *арабское* число, которое мы будем переводить в *римское*. Переменной при объявлении также можно присвоить какое-

нибудь *начальное значение*. Если вам безразлично - какое, то всегда присваивайте числовой переменной значение *нуль*:

```
var
  number: integer:=0;
  sNumber: string;
  n: integer:=0;
```

Переменная *number* хранит *числовое* значение, а следующая - *sNumber* – *строковое*. Ей также можно присвоить начальное значение. Естественно, оно не может быть нулём – его роль для строковых переменных играет *пустая* строка, в которой нет ни одного символа. Все строковые значения должны быть заключены в *одиночные кавычки*, иначе при запуске программы возникнет ошибка.

```
var
  number: integer:=0;
  sNumber: string:='';
  n: integer:=0;
```

Давайте из любопытства добавим ещё одну строковую переменную *s*, присвоим ей значение *0* и запустим программу. При компиляции исходного кода в новой строке обнаружится ошибка, вся строка будет выделена красным цветом, а в окне *Список ошибок* мы получим сообщение (Рис. 5.2).

```
10  var
11    number: integer:=0;
12    sNumber: string:='';
13    n: integer:=0;
14
15    s: string:=0;
```

Список ошибок

		Строка	Описание	Файл
✖	1	15	Нельзя преобразовать тип integer к string	rome.pas

Рис. 5.2. Ошибка вышла!

Итак, каждая переменная в паскале имеет *тип*, который мы назначаем ей при объявлении. В дальнейшем тип переменной изменить нельзя! Присваивать значения переменным можно только в соответствии с их типом, то есть числовым переменным – числа, строковым – строки!

Тип переменной указывается после её идентификатора и *двоеточия*. Переменные типа *integer* (от англ. *целый*) могут хранить только *целые числа*, переменные типа *string* (по-английски - *строка*) – только *строковые* и *символьные* (одну букву или другой знак).

Значение переменным задаётся с помощью знака операции *присваивания* – двоеточия и знака равенства **:=**, которые должны быть записаны *слитно*, без пробелов. В математике обходятся только знаком равенства, без двоеточия, поэтому будьте внимательны!

Операторы объявления переменных и констант, а также операторы присваивания должны заканчиваться *точкой с запятой*. Не забывайте об этом!

В стандартном паскале в разделе описания переменных им нельзя присваивать значения, а вот в нашем *паскале* это вполне допустимо. Более того, наш *паскаль* умный и может самостоятельно определять тип переменной по присваиваемому ей значению. Например, все целые числа имеют тип *integer*, а все символы в одинарных кавычках – тип *string*. Перепишем раздел переменных так:

```
var
  number:=0;
  sNumber:='';
  n:=0;
```

Если мы теперь подведём курсор мышки к идентификаторам переменных, то убедимся, что их тип определён верно (Рис. 5.3).

```
var
  number:=0;
  sNu var number: integer;
  n:=0;
```

```
var
  number:=0;
  sNumber:='';
  n:=0; var sNumber: string;
```

Рис. 5.3. Всё тип-топ!

Для контроля над типом переменной её название можно начинать с *префикса*. Например, ***i*** (*integer* - целое) или ***n*** (*number* - число) - для целых чисел, а ***s, str*** (*string* - строка) - для строковых.

Поскольку большинство переменных в программе *целочисленные*, то для них префиксы допустимо не указывать.

Более подробно об этом рассказано в разделе *Идентификаторы...*

Мы вспомнили, что числа бывают натуральные, отрицательные целые и действительные. Перед отрицательным числом, как и в математике, нужно записать знак *минус*, а дробная часть действительного числа отделяется от целой части *десятичной точкой*, а не запятой:

var fNumber:= -1.12345678912345;

Если вы хотите обозначить тип *действительной* переменной, то можете ставить префикс **r** или **d** (*real, double* - числа с плавающей точкой).

Точность действительной переменной для однозначной целой части - 14 знаков после запятой. Нам должно хватить! Но с увеличением целой части точность числа уменьшается таким образом, что всего в числе - 15 значащих цифр.

Важно отметить, что переменные и константы, объявленные в указанных разделах, являются *глобальными*, то есть ими можно пользоваться в любом месте исходного текста программы и они существуют всегда - пока работает программа.

Однако вернёмся к строковой переменной *sNumber*. Она проинициализирована *пустой строкой* - для этого просто два раза нажмите на клавишу с одинарной кавычкой. Пустая строка не содержит ни одного символа (буквы, цифры или знака препинания). Если начальное значение строковой переменной вас не интересует, всегда присваивайте ей пустую строку!

Паскаль при объявлении переменных автоматически присваивает им значения по умолчанию, то есть целые переменные получают значение 0, а строковые – пустую строку `''`. В других языках программирования это может быть не так, поэтому на всякий случай инициализируйте переменные самостоятельно.

Важно, чтобы переменная была объявлена (и проинициализирована) до первого использования в программе, иначе компилятор её просто не найдёт.

Переменная *n* - вспомогательная и служит для расчётов внутри программы. *Вспомогательные переменные* принято обозначать одной или несколькими буквами, а вот для «настоящих» переменных программы следует выбирать осмысленные имена, чтобы потом не гадать об их назначении.

Никогда не используйте одну и ту же переменную (за исключением, естественно, вспомогательных) для *разных* целей. Такая путаница часто приводит к ошибкам, которые очень трудно найти!

Вспомогательные переменные чаще объявляют внутри основной программы, в процедурах или в функциях. Такие переменные называются *локальными*. Они доступны только в том блоке кода, в котором объявлены, и уничтожаются, когда процедура или функция заканчивают свою работу. Естественно, локальные переменные основной части программы столь же «вечны», как и глобальные переменные.

Не пытайтесь давать переменным имена, совпадающие с *ключевыми* (их называют также *зарезервированными*) *словами* паскаля:

and array as begin case class const constructor destructor div do downto else end event except file final finalization finally for foreach function goto if implementation in inherited initialization interface is label lock mod nil not of operator or procedure program property raise record repeat set shl shr sizeof template then to try type typeof until uses using var where while with xor

Это ограничение относится и к именам модулей и классов.

Дальше - ещё интереснее! *Обычные переменные*, о которых мы говорили, могут одновременно хранить *единственное* значение. Но нередко нужна переменная для одной и той же цели, но при этом она должна хранить *несколько* значений. Конечно, можно использовать нужное количество обычных переменных, подставляя после названия соответствующий номер:

```
num1:=1000;
num2:= 900;
num3:= 500;
```

Но для паскаля это три *разные* переменные, поэтому вам придётся к каждой из них обращаться по имени-отчеству, а это бывает очень неудобно. В подобных случаях используют переменные и константы типа **массива**. Их записывают, как обычные переменные и константы, но затем в *квадратных скобках* указывают номер значения переменной (*индекс в массиве*). Причём паскаль разрешает использовать для индексов не только целые числовые значения, но и символьные.

Массив – это набор объектов (*элементов*) одного и того же типа, каждый из которых имеет свой *номер* (*индекс*) в этом наборе. Массивы относятся к *структурным типам* данных.

Для массивов (по-английски, массив - *array*) в *паскале* имеется **шаблон кода**. Напечатайте букву *a* и нажмите клавиши *Shift+ПРОБЕЛ* – в документ будет вставлен текст:

```
array[1..] of ;
```

Поскольку *Редактор кода* не знает числа элементов в массиве и их базового типа, то недостающую информацию необходимо ввести самостоятельно.

Вы легко найдёте в программе *Rome* два массива констант: один - целочисленный - *ARABIC* - для арабских чисел, второй – строковый - *ROME* - для римских. В обоих массивах по 13 элементов, начальные значения которым необходимо сразу же присвоить:

```
const
  ROME: array[1..13] of string = ('M', 'CM', 'D', 'CD', 'C', 'XC',
                                  'L', 'XL', 'X', 'IX', 'V', 'IV', 'I');
  ARABIC: array[1..13] of integer = (1000, 900, 500, 400, 100, 90, 50,
                                     40, 10, 9, 5, 4, 1);
```

Конечно, мы могли бы воспользоваться и массивом *переменных*, но их значения можно случайно изменить, а значения элементов *этих* массивов изменять в программе нельзя!

Если бы не краткость нашей жизни, можно было бы просто записать в массивы все числа от 1 до 3999 «по-арабски» и «по-римски», но поскольку это не так, то мы схитрим и в массивах сохраним только *13* самых необходимых сочетаний арабских и римских чисел, а остальные нам придётся вычислять.

Мы разумно ограничим себя числом 3999 сверху и 1 снизу (а меньше у римлян и не было), потому что другие римские числа записывались с помощью непечатных символов.

Легко понять, почему нам понадобились именно эти числа, – остальные довольно просто собрать из них, как ожерелье - из бусин. И вот как это делается.

Сначала программа считывает с помощью функции *ReadInteger* число, которое пользователь ввёл с клавиатуры, и присваивает переменной *number* значение, равное этому числу:

```
//========================================
//          ОСНОВНАЯ ПРОГРАММА
//========================================
begin
  while(true) do
  begin
    repeat
      writeln('Введите арабское число 1..3999');
      number := ReadInteger();
      //Если задан нуль, то работу с программой заканчиваем:
      if (number = 0) then exit;
    until ((number >= 1) and (number <= 3999));
```

Затем она проверяет введённое число, и если оно равно *нулю*, закрывает приложение, вызвав оператор *exit*.

Оператор **exit** досрочно завершает любую процедуру или функцию, но, поскольку у нас события разворачиваются в *основной* программе, то при выходе из неё всё приложение закрывается.

Для формирования строки с римским числом нам потребуются переменные *n* и *sNumber*. Вся премудрость конвертирования чисел таится в двух циклах *While*. Мы последовательно сравниваем заданное число *number* с теми числами, которые хранятся в массиве *ARABIC*, начиная с тысяч. *Внутренний цикл While* как раз и нужен для того, чтобы определять, сколько в *number* имеется тысяч, сотен, десятков и единиц (все остальные римские числа - 900, 500, 400, 90, 50, 9, 5, 4 - могут быть только в *единственном* числе). Если текущее значение *number* не меньше этих чисел, то из него число вычитаем, а в строку добавляем буквы, соответствующие этому числу в римской записи:

```
    //Формируем строку с римским числом,
    //равным заданному числу number:
    n := 0;
    sNumber := '';
    while (number > 0) do
    begin
      inc(n);
      while (ARABIC[n] <= number) do
```

```
      begin
        sNumber := sNumber + ROME[n];
        number := number - ARABIC[n];
        //writeln(sNumber);
      end
    end;
```

Как только от числа ничего не останется (это контролирует *внешний цикл While*), преобразование числа заканчивается, и мы смело можем печатать на экране перевод римского числа на современный европейский математический язык:

```
    writeln(sNumber);
     writeln();
  end;
end.
```

Вижу, что вам не терпится проверить программу в деле! Тогда смело жмите на кнопку *Запуск*, вводите числа, затем нажимайте клавишу *ВВОД* - и всё у вас получится (Рис. 5.4).

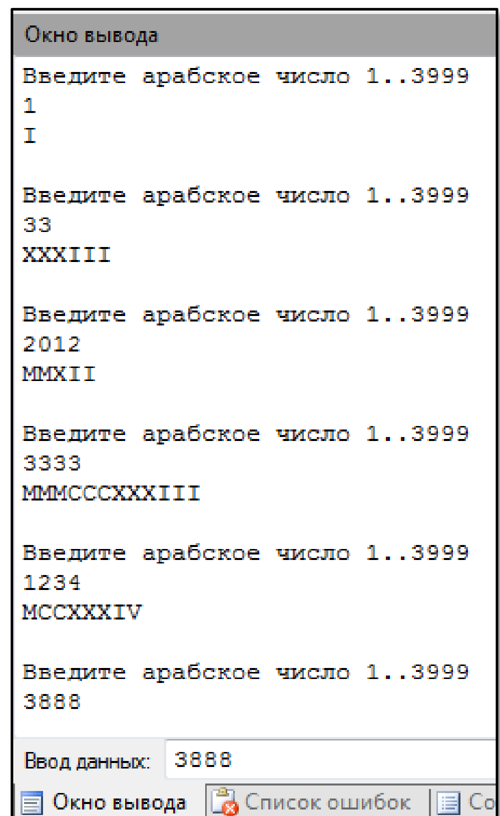


Рис. 5.4. Получилось не очень красиво, но программа работает правильно!

Сделаем дополнительную проверку и зададим числа 9999 и -1. Программа просто игнорирует неверный ввод (Рис. 5.5).

```
Введите арабское число 1..3999
9999
Введите арабское число 1..3999
-1
Введите арабское число 1..3999
```

Рис. 5.5. А программа-то совсем не глупа!

Наконец вводим *нуль* – и работа с программой заканчивается.

При запуске программы кнопкой *Выполнить* или клавишей *F9* вся информация печатается в *Окне вывода*, а набирать арабские числа необходимо в строке *Ввод данных* (Рис. 5.4).

Функция *ReadInteger* не контролирует ввод данных, поэтому при неправильном вводе (например, вместо цифры встретит-ся буква) программа завершится с сообщением об ошибке.

Настоящее консольное окно появится, если вы запустите с диска выполняемый файл приложения *rome.exe* или выполните команду меню *Программа > Выполнить без связи с оболочкой* (клавиши *Shift+F9*). На Рис.5.6 видно, что окна отличаются только надписью в заголовке.

```
E:\_PASCALABC\_Projects...
Введите арабское число 1..3999
1
I

Введите арабское число 1..3999
33
XXXIII

Введите арабское число 1..3999
2012
MMXII

Введите арабское число 1..3999
3888
MMMDCCCLXXXVIII

Введите арабское число 1..3999
1234
MCCXXXIV
```

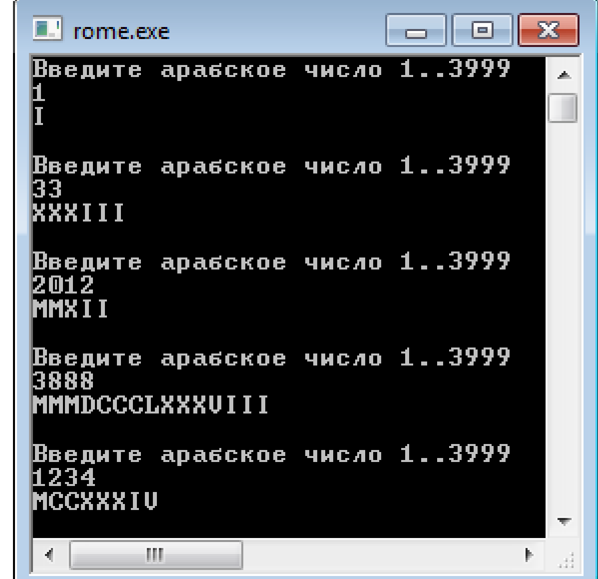


Рис. 5.6. Консольные окна

Исходный код программы находится в папке **Rome**.

Идентификаторы по-венгерски, по-паскальски, по-верблюжьи и по-русски

Все элементы программы – переменные, константы, процедуры, функции, классы, – как и люди, должны иметь собственное и неповторимое *имя*, чтобы программист мог к ним обращаться. Имя элемента программы называют также его *идентификатором.*

Плохие идентификаторы, как и плохие имена, затрудняют жизнь программ и людей. Чтобы облегчить выбор, сотрудник фирмы *Майкрософт* Чарльз Силош предложил заковыристую систему составления имён объектов программы. Как известно, венгерский язык очень сложный, но «венгерской» эту систему прозвали не поэтому - просто Чарльз Силош по происхождению венгр.

Существует немало разновидностей этой системы, но мы остановимся на самой простой и внятной из них. Итак, «по-венгерски» любой идентификатор составляют из *двух* частей:

- *префикса*, записанного строчными буквами, который указывает на тип объекта, и
- *спецификатора*, начинающегося с ПРОПИСНОЙ буквы.

Например, префикс *i* обозначает целый тип *integer*. Тогда координаты точки на экране можно хранить в двух переменных:

```
iX, iY: integer; или
iXCoord, iYCoord:integer; или
iCoordX, iCoordY:integer;
```

Если имеется в виду не точка, а мяч, то идентификатор легко приспособить и к этому случаю:

```
iXBall, iYBall:integer;
```

Префикс может также отражать *назначение* объекта:

c (counter) – счётчик;
f ,**flg**(flag) – переменная логического типа;
p (pointer) – указатель;
n, num (number) – число, количество чего-либо;
cur (current) – текущее значение;
tmp (temporal) – временное значение.

Вы можете выработать свою систему наименования, но некоторые распространённые префиксы желательно при этом учитывать.

Префикс	**Объект**
b, flg	переменная логического типа *boolean*
b	переменная целого типа *byte*
i	переменная целого типа *integer*
r, d	переменная вещественного типа *real, double*
a, arr	массив (*array*)
h	описатель (*handle*)
x,y,z	координаты точки
dx, dy	приращение координат
F	названия полей
C, T	названия классов
id, idx	индекс, идентификатор
frm	форма
lst	список
txt	текстовое поле
btn	командная кнопка
ico, pic, img, bmp	рисунок
tmr	таймер
s, str	строка

Этот список вы можете продолжить сами – место в таблице предусмотрительно оставлено!

Идентификаторы *констант* обычно записывают ПРОПИСНЫМИ буквами, разделяя отдельные слова знаком подчёркивания:

```
MAX_WIDTH= 20;
MAX_HEIGHT= 24;
```

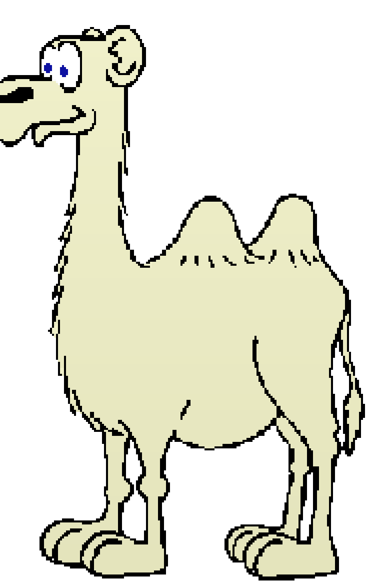
Фирма *Майкрософт* рекомендует все идентификаторы начинать с Заглавной буквы:

```
Hello
Program
Main
```

Если же идентификатор состоит из *нескольких* слов, которые пишутся слитно, то каждое из них должно начинаться с ПРОПИСНОЙ буквы, а все остальные буквы слова должны быть строчными:

```
HelloWorld
MyProgram
MainMethod
```

Такой способ конструирования имен называется *стилем паскаля*.



Верблюжий стиль отличается от паскалевского только тем, что первое слово пишется со *строчной* буквы:

```
program
main
myFirstProgram
mainProgramMethod
```

Достаточно посмотреть на двугорбого верблюда, чтобы понять смысл названия этого стиля!

В *паскале* **не разрешается** использовать *русские слова* для именования переменных и прочих объектов программы. Речь, конечно, идёт не о собственно русских словах, а только о русских буквах, поэтому вы вполне можете записывать идентификаторы *латинскими* буквами, как это принято делать в Интернете.

О правилах записи русских слов (транслитерации) можно прочитать на сайте *http://ru.wikipedia.org/wiki/Translit*. Вы можете использовать и свою систему перевода, главное, чтобы она была единой во всех ваших проектах.

Имена переменных в паскале могут быть очень длинными, например, *qwertzuiopasdfghjklyxcvbnm1234567890*, поэтому не жалейте букв! Если идентификатор состоит из нескольких слов, то их можно отделять друг от друга ПРОПИСНЫМИ буквами, как это принято в паскале, или знаком подчёркивания: *sobakaGryzlaKost, sobaka_gryzla_kost*. Если вы напишете все слова слитно и только строчными буквами, то получится абракадабра, которую будет трудно прочитать: *sobakagryzlakost*. Никогда так не делайте! Важно отметить, что паскалю безразлично, строчными или ПРОПИСНЫМИ буквами записан идентификатор, поэтому переменная *sobakaGryzlaKost* это та же самая переменная, что и *sobakagryzlakost*, и даже *SoBaKaGrYzLaKoSt*. Имейте это в виду.

1. Переведите в римские числа ещё несколько арабских. Убедитесь, что программа работает верно.

2. Составьте небольшую программу, подобную той, что мы написали на четвёртом уроке для вывода в текстовое окно выражения «2 * 2». Она должна проводить сложные арифметические вычисления – с числами, знаками и скобками.

Урок 6. Консольные приложения

Мы уже познакомились с консольными приложениями и договорились использовать *консольное окно* в тех программах, которые не требуют графического интерфейса пользователя.

И хотя консольные приложения могут показаться устаревшими, но, например, в современном языке *C#* имеется класс *System.Console* для создания таких приложений. С их помощью обычно изучают непростой язык *C#*, поскольку в таких программах нет постороннего кода для обслуживания графического интерфейса, который, по сути, не имеет отношения к самому языку программирования. Вот так выглядит простейшее консольное приложение, написанное в *Microsoft Visual C# 2010* (Рис. 6.1).

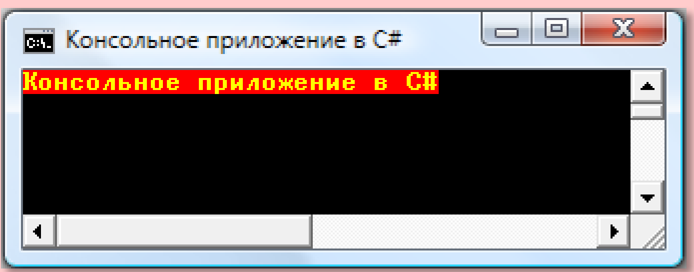


Рис. 6.1. Окно консольного приложения Си-шарпа

Вы можете сравнить окно этого приложения с *консольным окном паскаля* – оно совсем простое (Рис. 6.2).

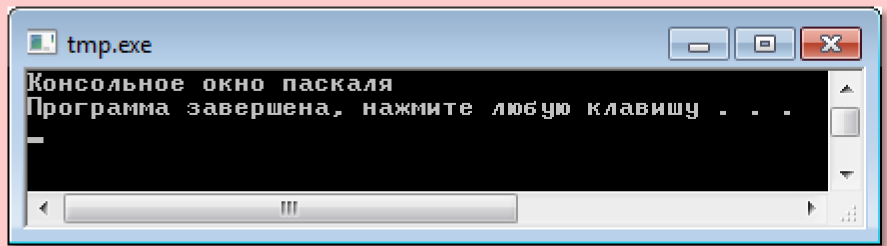


Рис. 6.2. Окно консольного приложения *паскаля*

Впрочем, и код на Си-шарпе выглядит более «устрашающе»:

```
Console.ForegroundColor = ConsoleColor.Yellow;
Console.BackgroundColor = ConsoleColor.Red;
```

```
Console.Title= "Консольное приложение в C#";
Console.WriteLine("Консольное приложение в C#");
Console.Read();
```

по сравнению с кодом на *паскале*:

```
begin
  writeln('Консольное окно паскаля');
end.
```

Справедливости ради следует отметить, что возможностей у класса *Console* в C# гораздо больше, чем у стандартного консольного окна *паскаля*.

Для создания более «продвинутого» консольного окна в *паскале* имеется специальный модуль *CRT*, с которым мы сейчас и познакомимся.

Модуль *CRT* (*Консоль*)

Главное назначение *Консоли* – ввод и вывод текстовой информации в консольных приложениях.

Чтобы консольное окно *появилось* на экране, нужно, прежде всего, добавить к программе модуль *CRT*:

```
uses CRT;
```

Модули объявляют в самом начале программы, до констант. Для этого необходимо после ключевого слова **uses** записать имена модулей через запятую. Объявление должно заканчиваться точкой с запятой.

По умолчанию *размеры* консольного окна равны примерно 680 х 340 пикселей (Рис. 6.3). Можно *изменить размеры* окна в программе, но гораздо удобнее это сделать в работающем приложении. Достаточно нажать среднюю кнопку в заголовке окна, чтобы его высота стала равной высоте *Рабочего стола*.

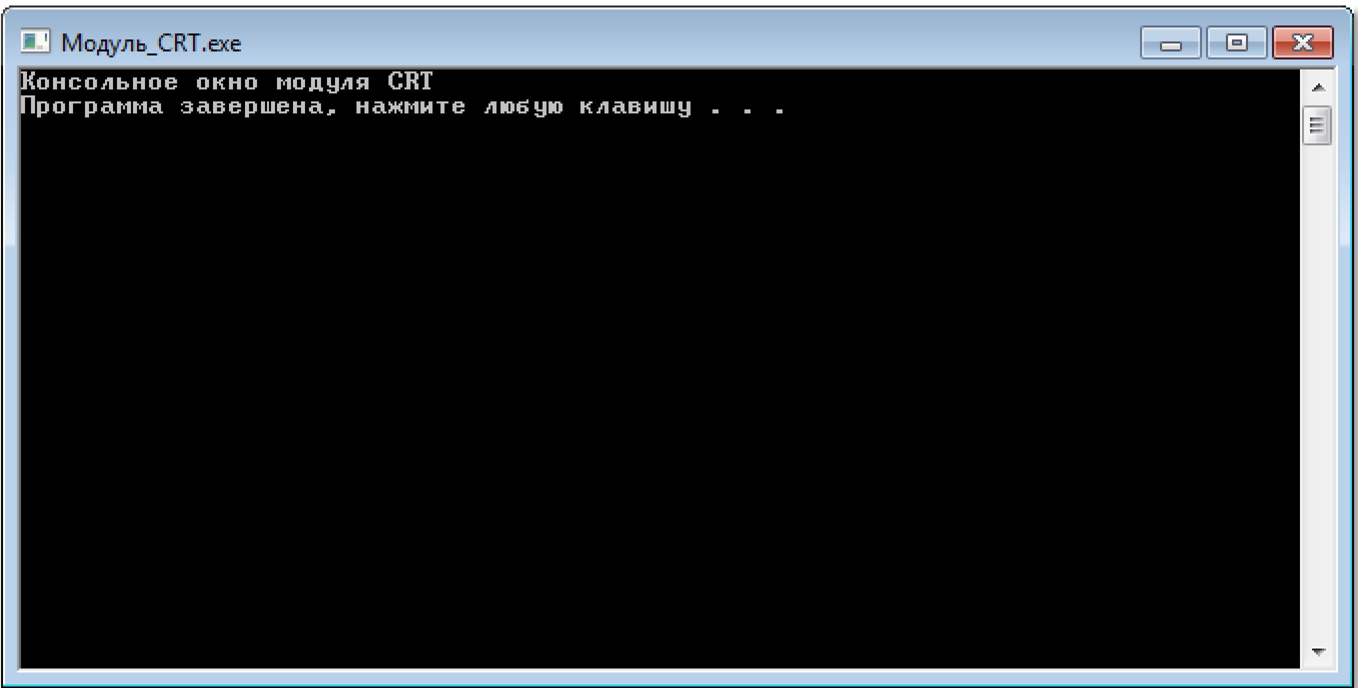


Рис. 6.3. Консольное окно модуля *CRT*

Тот же самый эффект вы получите, если дважды щёлкнете по заголовку окна, - оно вытянется во весь экран; повторный двойной щелчок вернёт ему прежние размеры.

Изменить размеры консольного окна также можно, потянув его мышкой за угол или за границу. При этом максимальная ширина окна остаётся постоянной, даже если кошка потянет за мышку, а Жучка - за кошку...

Такое «упрямство» *консольного окна* объясняется тем, что традиционно консоль имеет ширину в 80 символов. При ширине символов в *консольном окне* 8 пикселей мы как раз и получаем 640 пикселей.

Консольное окно всегда появляется в левом верхнем углу *Рабочего стола*, но его легко передвинуть в любое место, ухватившись мышкой за заголовок.

Если вы не позаботились об этом сами, то в *заголовке окна* будет напечатано название выполняемого файла программы (см. Рис. 6.3), что неплохо при отладке, но совсем не обязательно знать пользователю вашей программы. Впрочем, вы легко можете из-

менить *заголовок окна* с помощью процедур *SetWindowTitle* или *SetWindowCaption*.

Начните новый проект **Модуль_CRT** и сохраните его на диске. Наберите исходный текст:

```
uses CRT;

begin
  SetWindowTitle('Консольное окно модуля CRT');
  writeln('Консольное окно модуля CRT');
end.
```

Чтобы запустить консольное приложение с модулем *CRT*, необходимо выполнить команду меню *Программа > Выполнить без связи с оболочкой* или нажать *клавиши Shift+F9* (Рис. 6.4).

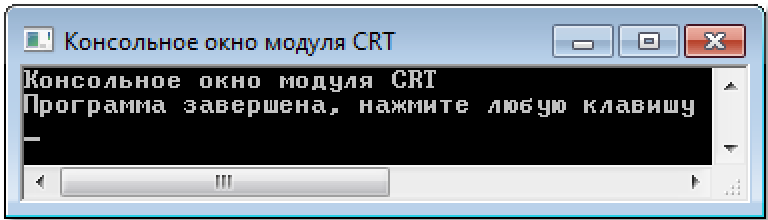


Рис. 6.4. Вот теперь заголовок «правильный»!

Украсив окно заголовком, давайте заставим его работать, то есть сообщать пользователю важную информацию. Для этого предназначены процедуры

Writeln(Данные);
Write(Данные);

Они выводят *Данные* (любые строковые выражения) в *текущую* строку консольного окна. Причём первый метод затем переводит вывод информации на *следующую* строку (Рис. 6.5).

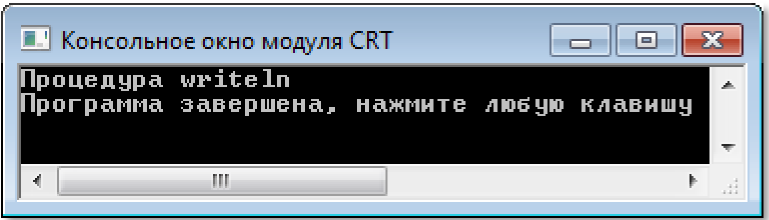


Рис. 6.5. Вывод информации методом *Writeln*

А второй метод продолжает вывод в *ту же самую* строку (Рис. 6.6).

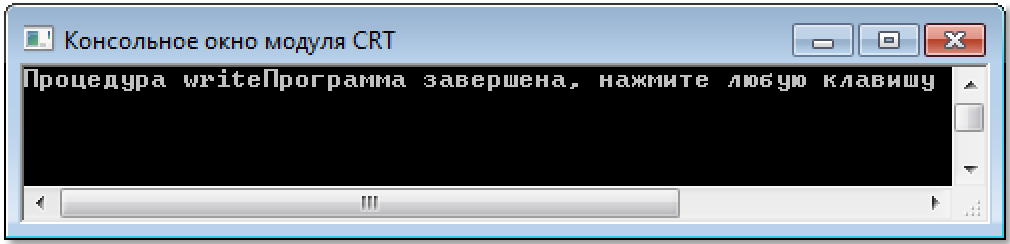


Рис. 6.6. Вывод информации методом *Write*

Чтобы сделать приложение по-настоящему *интерактивным*, нужно научить его *реагировать* на действия пользователя.

Процедура

Read(Список переменных);

ждёт, пока пользователь наберёт какие-нибудь *данные* (числа и строки, разделённые пробелами) с клавиатуры и нажмёт клавишу *ВВОД*, после чего присваивает значения переменным, указанным в круглых скобках через запятую.

Наберите такие строчки:

```
Writeln('Введите число и строку и нажмите клавишу ВВОД! ');
var n:integer;
var s: string;
read(n,s);
writeln(n + ' ' + s);
```

Здесь нам пригодилась процедура *Writeln*, чтобы объяснить пользователю, каких действий от него ожидает программа. Когда пользователь откликнется на просьбу и введёт данные, а потом нажмёт клавишу *ВВОД*, числовое значение 1234 будет присвоено переменной *n*, а строковое значение - переменной *s.* Затем мы формируем строку из этих значений и печатаем её методом *Writeln* (Рис. 6.7).

```
Консольное окно модуля CRT
Введите число и строку и нажмите клавишу ВВОД!
1234 строка
1234  строка
Программа завершена, нажмите любую клавишу . . .
```

Рис. 6.7. Ввод информации процедурой *read*

Процедура

Readln(Список переменных);

действует аналогично.

Функция

ReadInteger: integer;

возвращает целое число в программу (Рис. 6.8):

```
Writeln('Введите число и нажмите клавишу ВВОД! ');
var n: integer;
n:= ReadInteger();
writeln('n = ' + n.ToString());
```

С этой функцией мы уже встречались в программе *Rome* и убедились, что ей нужно пользоваться осторожно, иначе программа закроется с ошибкой.

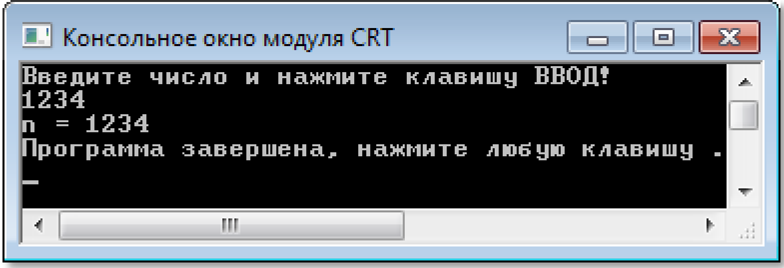


Рис. 6.8. Ввод целого числа функцией *ReadInteger*

Подобные функции имеются и для других типов данных паскаля:

ReadReal: real;

ReadString: string;
ReadChar: char;
ReadBoolean: boolean;

Первая функция возвращает *вещественное* число, вторая - *строку*, третья – *символ*, четвёртая – *логическую* переменную.

Функция

ReadKey: char;

возвращает *один символ*, соответствующий нажатой клавише (Рис. 6.9):

```
repeat
   Writeln('Нажмите символьную клавишу!');
   var ch:= ReadKey();
   writeln('ch = ' + ch);
until (false);
```

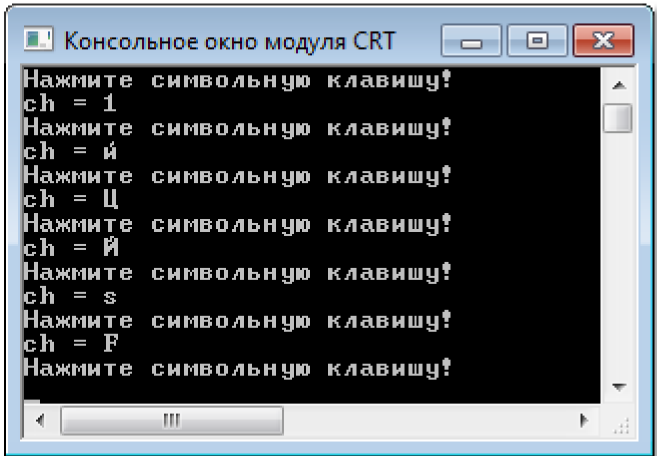


Рис. 6.9. Ввод информации функцией *ReadKey*

Чтобы ввести ЗАГЛАВНУЮ букву, удерживайте нажатой клавишу *Shift*.

Процедура

GotoXY(x, y: integer);

устанавливает текстовый *курсор* (текущий вывод) в заданную позицию *консольного окна*. При этом координаты задаются не в

пикселях, а в *символах*. Для самого первого символа в окне координаты равны *0, 0*. После нулевого символа идёт первый и так далее до 79-ого. Строки нумеруются сверху вниз. С помощью новой программы мы легко разбросаем цифры по всему *консольному окну* (Рис. 6.10):

```
For var i:= 1 To 20 do
begin
   var x:= Random(20);
   var y:= Random(20);
   GotoXY(x, y);
   Write(Random(10));
end;
```

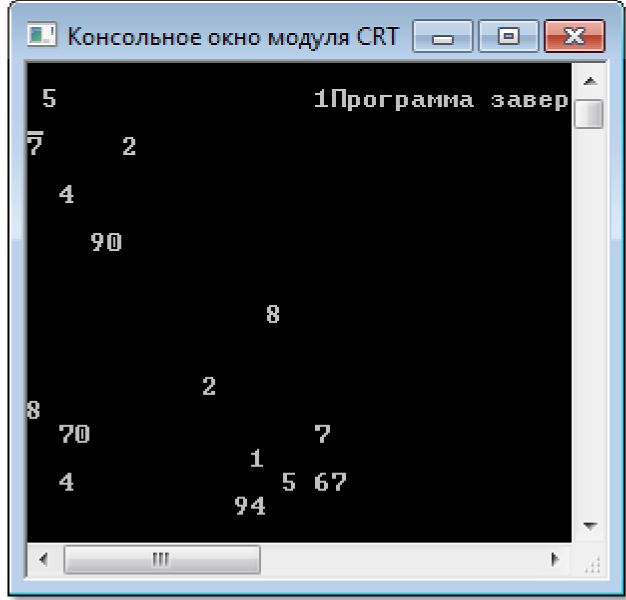


Рис. 6.10. Получилось!

Функции *WhereX* и *WhereY* выполняют обратные действия, то есть возвращают координаты *X* и *Y*, соответственно, текущей позиции курсора в консольном окне:

WhereX: integer;

WhereY: integer;

Процедура

ClearLine;

стирает всю текущую строку.
И нам осталось поговорить о самом прекрасном, что есть в *консольном окне*, – о ***цветах***.

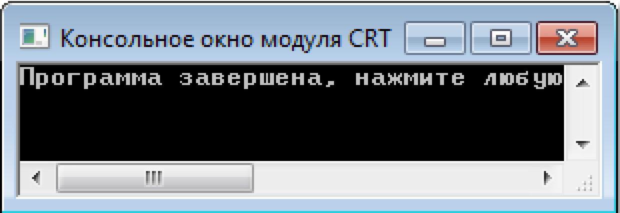
Процедура

ClrScr;

просто *стирает* всю информацию с экрана, окрашивая его текущим цветом фона (по умолчанию **чёрным**) (Рис. 6.11, слева). Позиция вывода устанавливается в начало первой строки.

Если вы хотите окрасить фон текстового окна в цвет, отличный от чёрного, поступайте так (Рис. 6.11, справа):

```
TextBackground(9);
ClrScr;
```



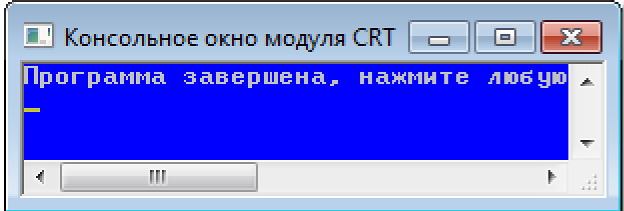


Рис. 6.11. Чистое *консольное окно* – **чёрного** и **синего** цвета

Последние две процедуры *консоли* расцвечивают выводимые *символы* во все цвета радуги.

Процедура

TextColor(c: integer);

устанавливает текущий *цвет символов*, а процедура

TextBackground(c: integer);

цвет фона, на котором эти символы печатаются.

Поскольку параметр *c* в этих процедурах имеет целый тип, то задавать его нужно числами – от 0 до 15. Это очень неудобно, так

как приходится запоминать, какой цвет соответствует тому или иному числу. Для облегчения запоминания в модуле *CRT* имеются именованные *константы*:

```
const
  Black        =  0;
  Blue         =  1;
  Green        =  2;
  Cyan         =  3;
  Red          =  4;
  Magenta      =  5;
  Brown        =  6;
  LightGray    =  7;
  DarkGray     =  8;
  LightBlue    =  9;
  LightGreen   = 10;
  LightCyan    = 11;
  LightRed     = 12;
  LightMagenta = 13;
  Yellow       = 14;
  White        = 15;
```

Ими пользоваться значительно удобнее, чем абстрактными числами. Чтобы не запоминать весь список, можно прибегнуть к такому приёму. Наберите имя модуля – *CRT*, поставьте точку, а затем выбирайте название цвета из списка (Рис. 6.12).

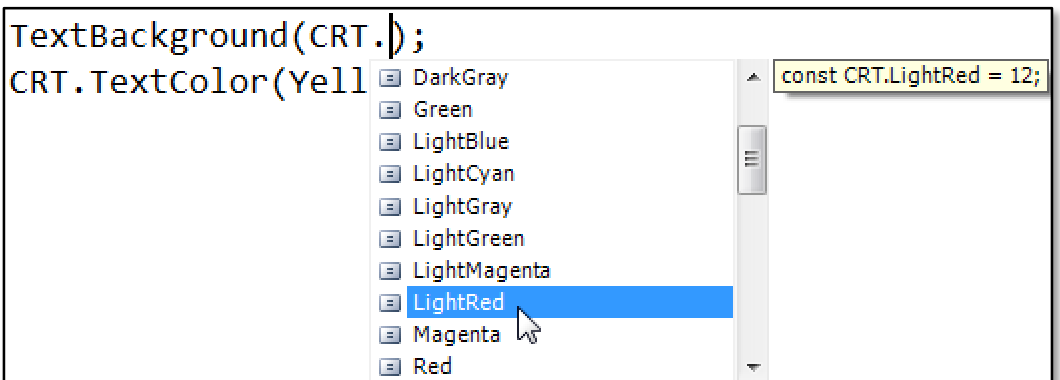
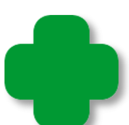


Рис. 6.12. Список цветов консоли



В этом же списке вы найдёте и все процедуры и свойства модуля *CRT*.

Чтобы посмотреть, как выглядят цвета консоли в «натуре», давайте добавим несколько строк к предыдущей программе:

```
TextBackground(0);
ClrScr;

for var i:=0 to 15 do
begin
   TextBackground(i);
   writeln('                                            ');
end;
TextBackground(CRT.LightRed);
TextColor(Yellow);
```

Теперь картина получилась более живописная (Рис. 6.13)!

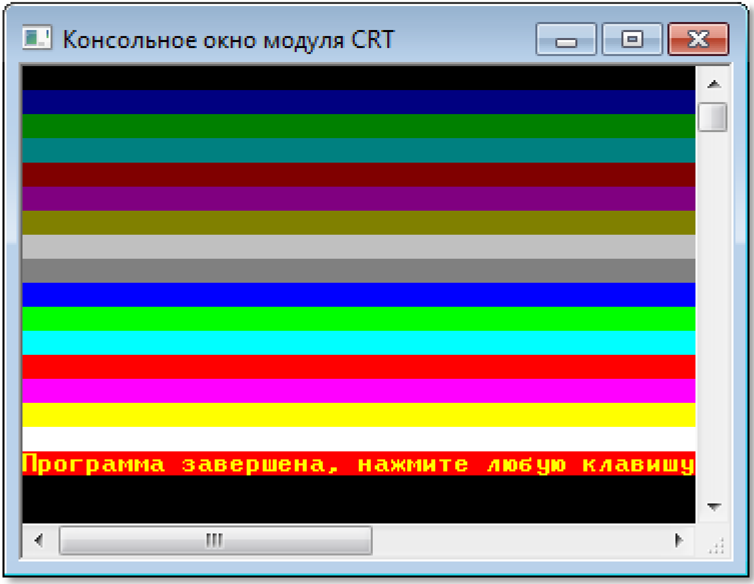


Рис. 6.13. Лепота!

Чтобы вам было легче расцвечивать числа и слова, вот вам **таблица** всех цветов, которыми вы можете пользоваться в *консольном окне* (Рис. 6.14).

Есть хорошие *способы* (*мнемонические приёмы*), помогающие запомнить последовательность цветов в спектре (или в радуге).

1. **Как Однажды Жак-Звонарь Городской Сломал Фонарь**.

2. **Каждый Охотник Желает Знать, Где Сидит Фазан**.

Мне больше нравится первый – он «складный».

	Black 0		DarkGray 8
	Blue 1		LightBlue 9
	Green 2		LightGreen 10
	Cyan 3		LightCyan 11
	Red 4		LightRed 12
	Magenta 5		LightMagenta 13
	Brown 6		Yellow 14
	LightGray 7		White 15

Рис. 6.14. Цветовая палитра *Консоли*

Процедуры и функции ввода-вывода – *read*, *write* и другие – не имеют прямого отношения к модулю *CRT*, поэтому могут использоваться в любых программах.

Исходный код программы находится в папке **Модуль_CRT**.

Раскрашиваем окна!

Но крашу, крашу я заборы,

чтоб тунеядцем не прослыть!

Песенка Царя из мультфильма

Вовка в Тридевятом царстве

Единственное порицание, которого заслуживает наша программа *Rome* – какая-то она невыразительная, **серая**, поэтому давайте сделаем себе красиво!

Для «цветного» варианта программы запишем исходный текст в папку **Rome2** под аналогичным названием – теперь можно смело фантазировать!

Пусть цвет зазывной надписи будет **зелёным**, а арабских чисел - ***жёлтым***! Для этого нам достаточно передать в процедуру *TextColor* значения ***LightGreen*** и ***Yellow***, соответственно:

```
. . .
repeat
   TextColor(CRT.LightGreen);
   write('Введите арабское число 1..3999) ');
   TextColor(Yellow);
. . .
```

Римские числа мы сделаем ***красными***:

```
. . .
TextColor(CRT.LightRed);
 writeln(sNumber);
. . .
```

Вся остальная часть программы осталась без изменений, поэтому, добавив несколько строчек, запускаем программу (Рис. 6.15).

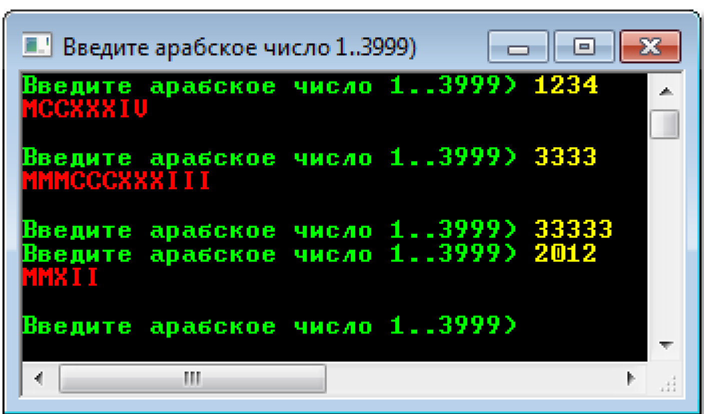


Рис.6.15 . Теперь вся информация выводится распрекрасно!

Исходный код программы находится в папке **Rome2**.

Комментарии

Как птицу видно по полёту, так и хорошего программиста – по комментариям в программе. Их не должно быть много, но они должны хорошо описывать алгоритм (а не элементарные действия, которые и без того всем понятны). Например, комментарий

```
i:= i+3;  // увеличиваем значение переменной i на три
```

ничего не объясняет, а просто пересказывает действие оператора сложения.

Комментарий

```
c:= c+1;  // увеличиваем значение счётчика слов
```

уже значительно лучше, поскольку мы понимаем, что в этом месте кода подсчитывается количество *слов*, а не каких-то абстрактных объектов.

Когда вы только пишете программу, вам и без комментариев всё в ней понятно. Но если вам понадобится модифицировать (изменить) программу, написанную несколько месяцев назад, вот тут-то и начнутся проблемы: без комментариев придётся восстанавливать весь алгоритм заново, а это иногда бывает труднее, чем с начала придумать новый. Не меньше трудностей вы создадите и своим друзьям-товарищам, если поделитесь с ними своей программой. Вряд ли им доставит удовольствие многочасовой разбор «немого» кода. Ещё больше вы сможете насолить им и себе,

если будете называть объекты малопонятными именами. Вывод такой: **не жалейте времени на комментирование программы**. Вам и самому будет легко разобраться в коде, и другие программисты будут вспоминать вас с благодарностью всю оставшуюся жизнь.

А вот транслятору ваши комментарии ни к чему, он просто игнорирует их при компиляции программы, так что от ваших пояснений выполняемый код не увеличится ни на один байт.

Обычно комментарии занимают целиком всю строку либо остаток строки:

```
//увеличиваем значение счётчика слов:
c:= c+1;

c:= c+1;  // увеличиваем значение счётчика слов
```

В этом случае они начинаются двумя слешами, за которыми и следует собственно комментарий. В самом комментарии разрешается использовать любые символы. Такой вид комментариев называется *однострочным*.

Реже комментарии располагают внутри кода (в любом его месте), тогда они обрамляется фигурными скобками:

```
c:= {увеличиваем значение счётчика слов} c+1;
```

Аналогично вы можете закомментировать целый блок кода, поэтому такой комментарий называется *многострочным*:

```
  for var i:=0 to 15 do
  begin
     TextBackground(i);
     writeln('                                            ');
  end;
  TextBackground(CRT.LightRed);
  TextColor(Yellow);

  {
```

```
  for var i:=0 to 15 do
  begin
     TextBackground(i);
     writeln('                                         ');
  end;
  TextBackground(CRT.LightRed);
  TextColor(Yellow);
  }
```

Если вам при отладке программы понадобится временно исключить из программы блок кода, то закомментируйте его символами { }, как многострочный комментарий. Вернуть его к жизни очень просто – уберите эти символы.

В *паскале* имеются и комментарии, начинающиеся с трёх слэшей. Пример такого комментария мы найдём в модуле *CRT* :

```
/// <summary>
/// Модуль для работы с консолью
/// </summary>
unit CRT;
```

Они служат для создания документации и подсказок по вашему проекту. Мы их использовать не будем.

Тип комментария	**Назначение**	**Пример**
Однострочный комментарий	Используется в отдельной строке или после оператора	//комментарий

Многострочный комментарий	Используется внутри строки или для нескольких соседних строк	{
		Комментарий1
		Комментарий2
		}

Однострочный комментарий *XML*-документирования	Как обычный однострочный комментарий, но используется для создания документации проекта	///комментарий

Урок 7. Операторы цикла *While* и *Repeat*

В программе *Rome* мы пользовались для формирования строки с римским числом *циклом While*, поэтому немного поговорим о **циклах**. Очень часто в программах встречаются ситуации, когда одни и те же действия нужно *повторить многократно*. Для таких случаев в языке паскаль припасены три оператора циклов – ***For, While*** и ***Repeat-Until***.

Примеры циклов мы можем найти и в повседневной жизни:
- времена года (циклически сменяют друг друга весна – лето – осень - зима),
- время суток (утро – день – вечер - ночь),
- фазы Луны,
- вращение планет вокруг Солнца и Солнечной системы вокруг центра Галактики,
- спортивные состязания,
- учебный год,
- режим дня,
- дыхание и кровообращение,
- часы,
- биоритмы активности человека – можно даже утверждать, что уникальны именно неповторяющиеся события, а не циклические.

Цикл *while*

While (*пока*) называется оператором цикла с *предусловием*, а *Repeat (повторять)* – оператором цикла с *постусловием*. Он отличается от цикла *While* только тем, что проверка завершения цикла проводится в конце цикла, а не в его начале. Это значит, что цикл *Repeat* всегда выполняется хотя бы один раз, а цикл *While* может вообще не выполняться ни разу. Поскольку различаются эти циклы очень мало, то мы вполне можем рассмотреть

их на одном уроке. Ещё один цикл – *For* - мы разберём на следующем уроке.

Цикл **While** с *единственным* оператором в теле цикла *записывается* так:

```
While условие_выполнения_цикла    do  ← Заголовок цикла
  оператор1;                          ← Тело цикла
```

А с *блоком* операторов - так:

```
While условие_выполнения_цикла    do  ← Заголовок цикла
begin
  оператор1;                          ← Тело цикла
  оператор2;
 . . .
 операторN;
end;                                   ← Конец оператора цикла
```

А работает он так:

1. Проверяется *условие выполнения цикла*.

2. Если оно *ложно* (не выполняется), то цикл заканчивается, и программа переходит на строчку, следующую за ключевым словом *end*.

3. Если оно *истинно*, то последовательно выполняются операторы *оператор1*, *оператор2*, … , *операторN*, которые называются *телом цикла*, и управление передаётся в *п.1*.

Поскольку условие проверяется *до* тела цикла, то операторы могут вообще не выполняться – если выражение с самого начала ложно.

При наборе цикла *while* удобно пользоваться шаблоном кода. Для единственного оператора в теле нажмите клавишу *w*, а затем *Shift+ПРОБЕЛ*:

```
while do
```

Для блока операторов – наберите буквы *wb* и нажмите клавиши *Shift+ПРОБЕЛ*:

```
while do
begin

end;
```

Как мы видим, цикл заканчивается тогда, когда условие станет ложным. А почему *изменяется* значение *условного выражения* в заголовке цикла? – Да потому, что изменились значения переменных, входящих в это условие. И это может произойти только в теле цикла. Отсюда грустный вывод: цикл *While* может вообще никогда не закончиться, и программа зациклится навечно, что очень часто и бывает!

Загрузите программу *Rome* и немного исправьте код:

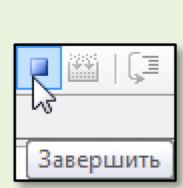
```
  n := 1;
  sNumber := '';
  while (number > 0) do
  begin
    while (ARABIC[n] <= number) do
    begin
      sNumber := sNumber + ROME[n];
      number := number - ARABIC[n];
    end;
    inc(n);
  end;
```

Запустите программу и убедитесь, что она работает верно! Теперь закомментируйте строку *inc(n)*:

```
    end;
      //inc(n);
    end;
```

Запустите программу, введите любое число и нажмите клавишу *ВВОД* – программа зациклится, и римское число вы не увидите никогда. Не томите компьютер и закройте программу.

При работе со стандартной консолью в режиме отладки для этого следует нажать кнопку *Завершить* (Рис. 7.1, слева), или выполнить команду меню *Программа > Завершить* (Рис. 7.1, справа), или нажать клавиши *Ctrl+F2*.



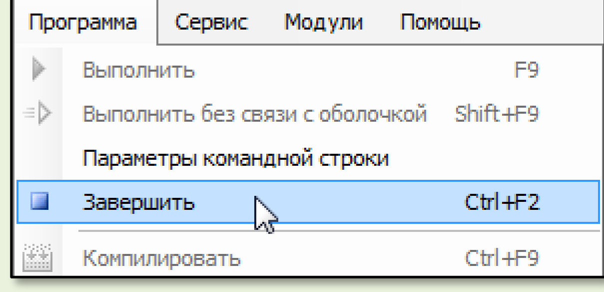
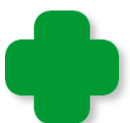


Рис. 7.1. Закрываем зациклившуюся программу

Всегда следите за изменением *переменной* (или переменных), входящей в условие выполнения цикла! Если она не изменяется, то цикл либо не выполнится ни одного раза, либо будет выполняться бесконечно! Обратите внимание на отличие цикла *While* от цикла *For*, в котором, наоборот, переменную цикла изменять нельзя.

Применяйте цикл *For*, если число повторений известно заранее, а в противном случае лучше подойдёт цикл *While*.



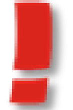
Паскаль позволяет записывать в строке больше одного оператора, поэтому вы можете вместо двух строчек:

```
n := 0;
sNumber := '';
```

«вбить» код в одну:

```
n := 0; sNumber := '';
```

Обычно этого делать не следует: не экономьте «бумагу», оставляйте в строке только *один* оператор, тогда исходный код будет гораздо легче понять.



Вернитесь к первоначальному варианту программы *Rome* и закомментируйте другую строку:

```
n := 0;
sNumber := '';
while (number > 0) do
begin
  inc(n);
  while (ARABIC[n] <= number) do
  begin
    sNumber := sNumber + ROME[n];
    //number := number - ARABIC[n];
  end
end;
```

Что произойдёт с программой при запуске?

Цикл repeat

Как мы уже знаем, *Repeat* – называют оператором цикла с *постусловием*. В нём проверка завершения цикла проводится в *конце* цикла, а не в его начале. Это значит, что цикл *Repeat* всегда выполняется хотя бы один раз.

Цикл **Repeat** записывается так:

Repeat	**← Начало цикла**
оператор1;	**← Тело цикла**

```
   оператор2;
   . . .
   операторN;
Until условие_выполнения_цикла;        ← Конец оператора цикла
```

А работает так:

1. Последовательно выполняются операторы *оператор1, оператор2*, ... , *операторN* в *теле* цикла.
2. Проверяется *условие выполнения цикла после ключевого слова until*.
3. Если оно *истинно* (выполняется), то цикл заканчивается, и программа переходит на строчку, следующую за концом оператора цикла.
4. Если *условие* ложно (не выполняется), то управление передаётся в *п.1*.

Итак, давайте сравним операторы *while* и *repeat*.

1. В цикле *while* условие выполнения цикла проверяется до тела цикла, а в цикле *repeat* – после. Поэтому первый цикл может вообще не выполняться, а второй выполняется, по крайней мере, один раз.

2. Ключевые слова *repeat – until* играют роль операторных скобок, поэтому в теле цикла может быть любое число операторов, которые не нужно заключать в операторные скобки *begin – end*.

3. Цикл *while* выполняется, пока условие *истинно*, а цикл *repeat* – пока оно *ложно*.

4. Любой цикл *repeat* можно заменить циклом *while*, но не наоборот!

5. Цикл *repeat* также может стать *бесконечным*, если условие всегда остаётся ложным.

При наборе цикла *repeat* удобно пользоваться шаблоном кода. Нажми клавишу *r*, а затем *Shift+ПРОБЕЛ*:

```
repeat

until ;
```

Оператор присваивания

Самый «востребованный» оператор в любом языке программирования – это, безусловно, **оператор присваивания**.

Он записывается так:

```
var переменная := выражение;
```

Переменная - это простая переменная, элемент массива или свойство объекта. В результате выполнения этого оператора значение переменной станет равным значению выражения. Выражением может быть другая переменная, константа, константное выражение или функция. Сложные выражения могут быть составлены из переменных и констант, соединённых знаками арифметических операций.

Например:

Оператор присваивания	Результат присваивания, значение переменной
`var Width := 100;`	100
`var Height := Width + 20;`	120
`var Mas: array[1..20] of string;`	Двенадцать

`Mas[12] := 'Двенадцать';`	
`Mas[13] := 'Двенадцать' + 1.ToString();`	Двенадцать1
`var Str := Mas[13] + Height.ToString();`	Двенадцать1120
`var maxNum := Max(1,11););`	11

Различие между оператором присваивания := и знаком равенства = в математике можно показать на примере. Очень часто в программах можно встретить вот такие выражения:

```
var x := 10;
x := x + 1;
```

Первая строка не вызывает никаких возражений: согласно правилам алгебры, неизвестная величина *x* равняется 10. Во второй строке *сначала* вычисляется значение выражения *справа* от знака присваивания, а *затем* оно *присваивается* переменной *слева* от знака присваивания. В нашем случае: в первой строке переменная *x* получила значение 10. Вычисляем значение выражения во второй строке: 10 + 1 = 11. После выполнения оператора присваивания переменная *x* будет равна 11. Конечно, выражение

```
x = x + 1;                                              (1)
```

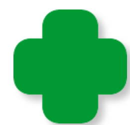
в математике было бы неверным. Поэтому в паскале знак равенства = применяется в условных выражениях при сравнении двух значений, а для присваивания специально введён знак :=, чтобы не путать его со знаком равенства. Во многих других языках программирования знак присваивания совпадает со знаком равенства, что и приводит к «абсурдным» выражениям типа (1).

Отладка программ

Лови жучков!

Программистская пословица

Отладка программ – самый трудный и ответственный момент в программировании. В самом деле, если хотя бы одна ошибка останется в готовом приложении, то оно иногда будет работать неправильно!



Вы, наверное, помните, сколько неприятностей доставил пользователям 2000-й год. А всё потому, что во многих старых программах для обозначения текущего года использовались только две последние цифры, то есть 1999-й год считался 99-м. После 99-го, естественно, наступил год 00. В некоторых программах он трактовался правильно – как 2000-й, в других неправильно – как 1900-й. Эта ошибка объясняется тем, что в начале компьютерной эры программисты даже и не задумывались о том, что их программы будут использовать и в 2000-м году.

Другой курьёзный случай. Клиент одного банка ежемесячно получал требование: погасить долг в течение месяца. Однако сумма долга составляла 0 долларов, 0 центов. Конечно, оплачивать такой счет не имело никакого смысла. Этот пресловутый клиент именно так и думал. И каждый месяц получал новое предупреждение. Так продолжалось до тех пор, пока он не оплатил «долг», выслав банку чек на означенную сумму. А дело тут в том, что компьтерная программа в банке не учитывала, что нулевой долг таковым не является.

Ошибки, которые проявляются только при работе уже готовой программы, принято называть словом *баг* – от английского *bug – жук*.

Этот термин возник в 1945 году, когда небольшая бабочка, незаконно проникшая в вычислительную машину, замкнула реле, чем вызвала ошибки в её работе. С тех пор слово *debugging*, то есть *извлечение скрытых багов*, означает отладку программы.

Все **ошибки** в программе можно условно разделить на синтаксические и логические.

Синтаксические (грамматические) ошибки появляются обычно из-за невнимательности или описки. Такие ошибки нетрудно «обезвредить», поскольку *ИСР* в процессе компиляции легко находит их и сообщает об этом в окне *Список ошибок*, которое находится под окном *Редактора кода* (Рис. 7.2).

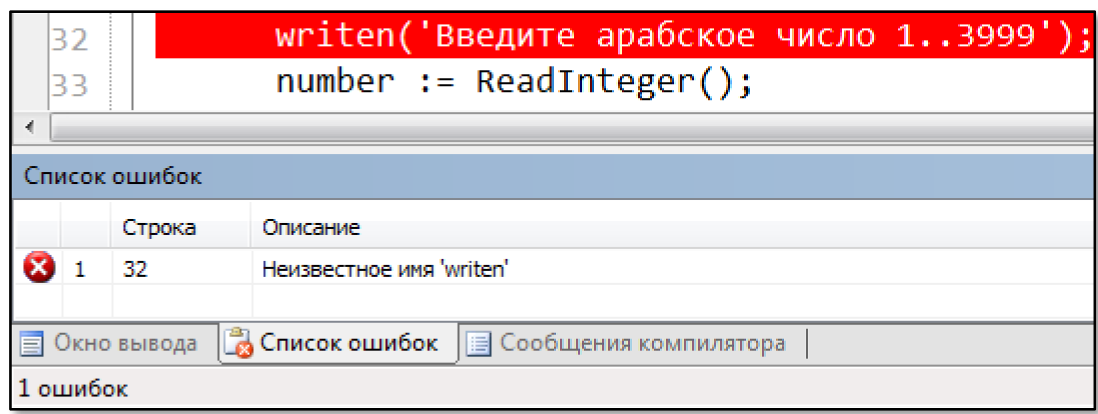


Рис. 7.2. Отладчик нашёл ошибку

Кроме ошибок, отладчик выдаёт и *предупреждения* (Рис. 7.3). Их можно и проигнорировать, поскольку такие действия программы не нарушают её работоспособности, но желательно принимать их во внимание.

Список ошибок

		Строка	Описание	Файл	Путь
⚠	1	7	Переменной 'Str' присвоено значение, но оно нигде далее не используется	Program1.pas	C:\Windows\syst
⚠	2	8	Переменной 'maxNum' присвоено значение, но оно нигде далее не используется	Program1.pas	C:\Windows\syst
⚠	3	23	Переменной 't' присвоено значение, но оно нигде далее не используется	Program1.pas	C:\Windows\syst

Окно вывода | Список ошибок | Сообщения компилятора | Локальные переменные | Просмотр выражений

Рис. 7.3. Отладчик нашёл неиспользуемые переменные

Достаточно щёлкнуть по сообщению, и курсор установится в то место исходного кода, где найдена ошибка. Строка с ошибкой также выделяется **красным** цветом.

Номер строки с ошибкой указываются слева от описания ошибки.

Некоторые ошибки легко распознать благодаря *окрашиванию* элементов программы в разные цвета и по атрибутам шрифта. Например, ключевые слова выделяются **жирным** шрифтом (Рис. 7.4, слева). А если их написать неверно, то шрифт останется обычным (Рис. 7.4, справа).

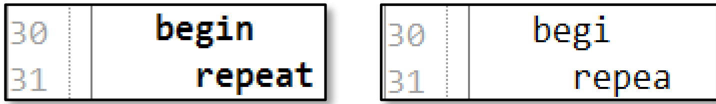
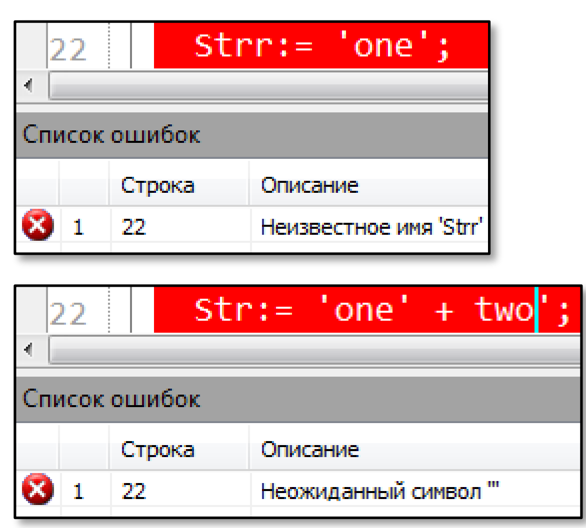


Рис. 7.4. Выделение ключевых слов

Таким образом, при должной внимательности можно легко найти синтаксические ошибки.

Обычные синтаксические ошибки – это неверно написанные идентификаторы, непарные скобки и кавычки, пропуск точки с запятой в конце оператора и двоеточия в знаке присваивания (Рис. 7.5).

Логические ошибки возникают вследствие неправильного алгоритма или неверной записи операторов программы, которые компилятор *паскаля* найти не может, потому что они не противоречат правилам языка. Логические ошибки отыскать очень трудно, а сам поиск таких ошибок называется *отладкой программы*.



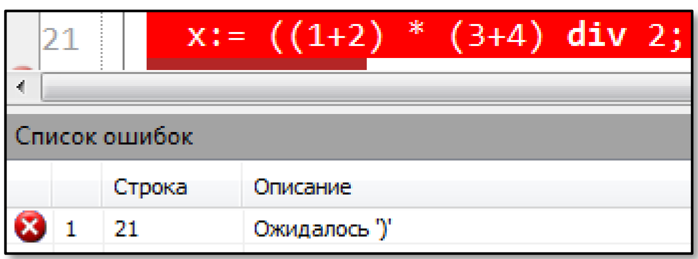


Рис. 7.5. Типичные синтаксические ошибки

В первую очередь, при написании программы следует помнить, что после добавления логически законченного фрагмента кода, сразу же следует проверить, правильно ли он работает. **Никогда не пишите большие куски кода**, иначе найти ошибки будет очень трудно.

Отлаженный промежуточный вариант программы всегда сохраняйте на диске, добавляя к названию программ номер версии: *MyProgram1, MyProgram2* и так далее. Конечно, последние изменения в тексте легко удалить с помощью кнопки *Отмена*, но сможете ли вы всегда вернуться к тому коду, который предшествовал ошибке? – Поэтому **не жалейте времени на сохранение отлаженного кода!**

Отладчик *ИСР* действует только при запуске програмы в *отладочном* режиме с помощью кнопки *F9*. Из этого следует, что консольные приложения с модулем *CRT* необходимо отлаживать самостоятельно. Для этого в критических местах программы нужно вставлять операторы, которые выводят отладочную информацию в окно программы.

Загрузите программу *Rome2* и добавьте во внутренний цикл *while* отладочный оператор *writeln(sNumber):*

```
    while (number > 0) do
    begin
       inc(n);
       while (ARABIC[n] <= number) do
       begin
          sNumber := sNumber + ROME[n];
          number := number - ARABIC[n];
```

```
            writeln(sNumber);
        end
    end;
```

Он напечатает весь процесс формирования строки *sNumber* с римским числом (Рис. 7.6).

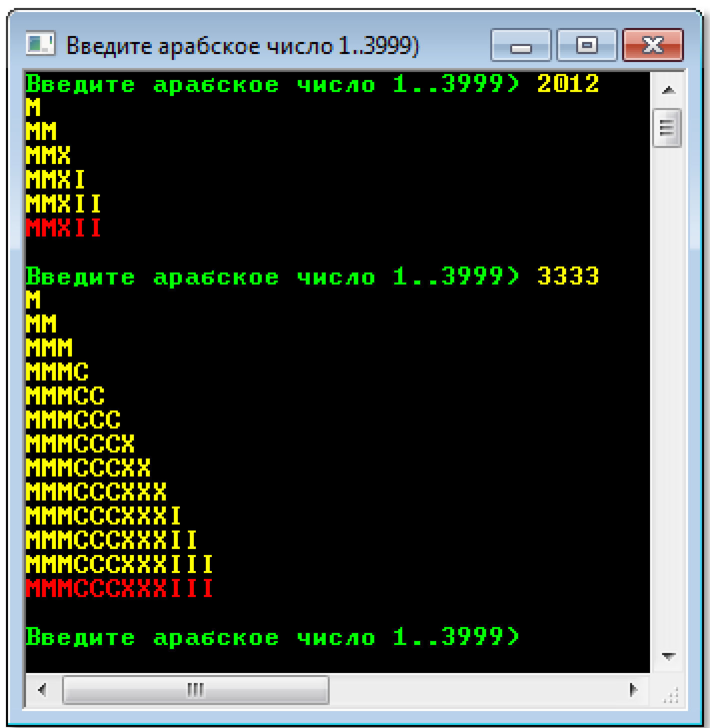


Рис. 7.6. Отладка алгоритма

Так гораздо проще проследить за работой алгоритма!

После проверки программы отладочные операторы можно стереть или закомментировать:

```
    while (number > 0) do
    begin
        inc(n);
        while (ARABIC[n] <= number) do
        begin
            sNumber := sNumber + ROME[n];
            number := number - ARABIC[n];
            //writeln(sNumber);
        end
    end;
```

Вернёмся к бесконечным циклам. Например, вы забыли написать закомментированную строку, и вместо римского числа получили бесконечное ожидание:

```
      while (number > 0) do
      begin
         inc(n);
         while (ARABIC[n] <= number) do
         begin
            sNumber := sNumber + ROME[n];
            //number := number - ARABIC[n];
            writeln(number);
         end
      end;
```

Добавим отладочный оператор *writeln(number)* и запустим программу. Рис. 7.7 показывает, что исходное число *number* не изменяется в цикле, а ведь оно входит в условие выполнения внешнего цикла *while (number > 0),* то есть условие всегда будет истинным (2012 > 0), поэтому цикл не закончится никогда!

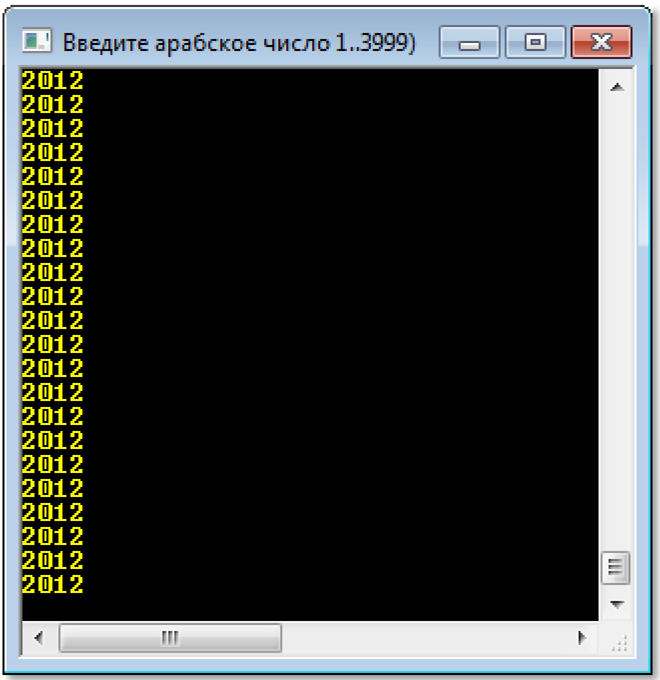


Рис. 7.7. Отладка накладки!

Вот так, с помощью единственного отладочного оператора мы легко нашли причину ошибки!

Аналогично вы можете контролировать значения любых переменных программы, вставляя отладочные операторы в те места кода, где переменные должны изменять свои значения.

После отлова всех жучков отладочные операторы можно удалить или закомментировать. Второй вариант предпочтительнее, так как исходный код может ещё вам пригодиться при написания другой программы, а в ней также могут понадобиться отладочные операторы. Их можно и снова написать, но раскомментировать строку гораздо быстрее и удобнее.

Теперь посмотрим, как действует штатный **отладчик** *ИСР* на примере программы *Rome*.

Пусть мы хотим понаблюдать за изменением значения переменной *n*. Находим строку, в которой она увеличивает своё значение на 1 и ставим, кликнув мышкой, красную точку перед номером строки (Рис. 7.8). Так мы задаём *точку останова* программы при её выполнении.

```
42      while (number > 0) do
43      begin
44        inc(n);
45        while (ARABIC[n] <= number) do
46        begin
47          sNumber := sNumber + ROME[n];
48          number := number - ARABIC[n];
49          //writeln(sNumber);
50        end
51      end;
```

Рис. 7.8. Точка останова программы

Как только программа перейдёт на эту строку, её выполнение приостановится. Строка будет выделена жёлтым цветом и стрелкой (Рис. 7.9). Подведя курсор мышки к переменной, мы сможем прочитать в подсказке её текущее значение (в данном примере – *до* выполнения оператора в этой строке)

```
42      while (number > 0) do
43      begin
44        inc(n);
45        whi  n  3  <= number) do
46        begin
47          sNumber := sNumber + ROME[n];
48          number := number - ARABIC[n];
49          //writeln(sNumber);
50        end
51      end;
```

Рис. 7.9. Значение переменной в подсказке

Точно так же вы можете исследовать и другие переменные программы (Рис. 7.10).

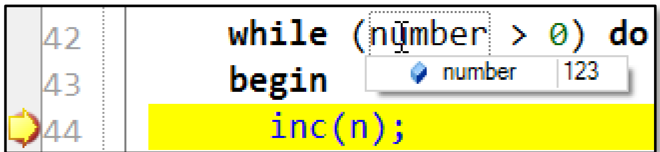


Рис. 7.10. Узнаём значения переменных

Ещё больше информации предоставляет нам окно *Локальные переменные*, которое появляется при остановке программы (Рис. 7.11)

Локальные переменные

Переменная	Значение	Тип
Глобальные переменные		
number	123	integer
sNumber	''	string
n	0	integer
ROME	('M','CM','D','CD','C','XC','L','XL','X','IX','V',...)	rome.$pascal_array1
ARABIC	(1000,900,500,400,100,90,50,40,10,9,5,...)	rome.$pascal_array2

Окно вывода | Список ошибок | Сообщения компилятора | Локальные переменные | Просмотр выражений

Рис. 7.11. Все переменные программы в одном окне

В нём вы легко найдёте текущие значения всех переменных программы – как локальных, так и глобальных.

Вы можете назначить сколько угодно точек останова, в том числе и в запущенной программе! Если мы хотите, например, узнать значение переменной *n* после выполнения оператора *inc(n);*, то добавьте новую точку останова в следующей строке (Рис. 7.12).

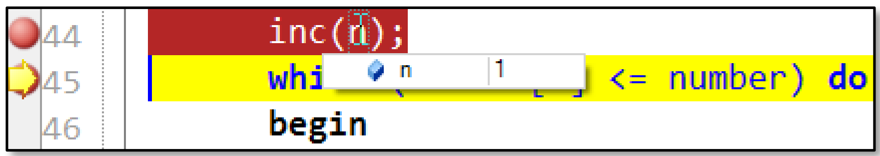


Рис. 7.12. Значение переменной *n* после выполнения оператора инкремента

Чтобы *убрать* точку останова, нужно просто кликнуть по ней мышкой.

После приостановки программы нажмите кнопку *Выполнить* (*F9*), чтобы продолжить её работу.

Урок 8. Признаки делимости

Амёбы умножаются делением.

Математический парадокс

Целые числа и их свойства изучает *теория чисел*, или *высшая арифметика* (есть и такая наука!).

В занимательных задачах (да и в жизни тоже) нередко нужно быстро определить, делится ли одно число на другое или нет. При этом сам результат деления неважен. У каждого натурального числа имеется, по крайней мере, два делителя - это 1 и само число. Если других делителей нет, то число называется *простым*. К ним мы вернёмся на следующем уроке, а сейчас вспомним *признаки* (то есть правила) *делимости*.

Признак делимости на 2

Самый простой признак: *число делится на 2 только тогда, когда его последняя цифра равна 0, 2, 4, 6 или 8*. Если число делится на 2, то оно называется *чётным*, если не делится - *нечётным*. Примеры чётных чисел: 2012, 92, 4, 76, 58.

Иногда этот признак формулируют проще: *число делится на 2 тогда и только тогда, когда его последняя цифра чётная.*

Признак делимости на 3

Число делится на 3 тогда и только тогда, когда сумма его цифр делится на 3. Например, число 2010 кратно 3, поскольку сумма его цифр равна 3: 2 + 1 = 3 (при подсчете нули не учитываем).

Если сумма цифр выражается не однозначным числом, то следует найти сумму *его* цифр. Если в результате сложения цифр получится одно из чисел 3, 6 или 9, то число делится на 3. В противном случае – не делится. Например, сумма цифр числа 123456789 равняется 45. Число двузначное – опять находим сумму цифр: 4 + 5 = 9. Получили *девятку* – значит, исходное число 123456789 делится на три.

Признак делимости на 4

Очевидно, что числа кратные четырём, должны быть чётными. Но этого мало, поэтому мы оставляем от числа только *последние две цифры* и рассматриваем получившееся двузначное число.

Если число двузначное, то ничего отбрасывать не нужно. А если однозначное, то достаточно вспомнить таблицу умножения.

Если это двузначное число делится на 4, то и всё число также делится на четыре.

Почему достаточно рассмотреть только последние две цифры числа? – На этот вопрос легко ответить, если вспомнить, что сотня делится на 4 без остатка. Естественно, любое число сотен также разделится на 4, поэтому разряды сотен, тысяч и так далее в проверяемом числе можно не учитывать.

Можно упростить проверку. Сложите число десятков с половиной единиц. Если сумма четная, то исходное число делится на 4, в противном случае не делится. Опять проверим год 2010. Число из последних двух цифр равно 10. Оно на 4 не делится, значит 2010 не кратно четырем. Возьмем другое число - 4567896. Оставляем две цифры - 96. Складываем 9 с половиной от 6, то есть с тройкой и получаем 12. Это число кратно четырём, значит, число 4567896 делится на 4.

Признак делимости на 5

Это правило очень похоже на признак делимости на двойку. *Число делится на 5, если оно оканчивается на 0 или 5.*

Признак делимости на 6

Число делится на 6, если *одновременно* выполняются *признаки делимости на 2 и 3*.

Признак делимости на 7

Хорошего признака делимости чисел на 7 не существует, зато

имеется немало достаточно сложных и запутанных. Из них мы выберем один – самый простой и «универсальный».

Разбиваем заданное число, начиная с конца, на группы, состоящие из трёх цифр. Например, если мы проверяем число 4567896, то получим три группы цифр:

4 567 896

Кстати говоря, так в книгах зачастую и печатают длинные числа, чтобы легче было распознать разряды сотен, тысяч и так далее.

Теперь первое (считаем сзади!) число мы берём со знаком плюс, второе со знаком минус, третье – снова о знаком плюс. То есть знаки плюс и минус *чередуются*. Составляем из чисел с их знаками арифметическое выражение и вычисляем его значение:

896-567+4 = 333

Если результат делится на 7, то и всё число также делится на 7. В противном случае не делится.

В нашем примере число 333 на 7 не делится, значит, это вывод относится и к исходному числу 4567896.

Признак делимости на 8

Число делится на 8, если оно чётное. Кроме того, *необходимо, чтобы число, составленное из трёх последних цифр, делилось на 8.* Так как делить трёхзначное число на 8 тоже нелегко, то можно воспользоваться тем же приемом, что и в *признаке делимости на 4.*

Тысяча делится на 8 без остатка. Любое число тысяч также разделится на 8, поэтому разряды тысяч и далее в проверяемом числе можно не учитывать.

Рассмотрим три последние цифры. К числу, образованному первыми двумя цифрами, добавьте половину единиц, а затем к числу десятков добавьте половину единиц получившегося числа. Если результат чётное число, то исходное число делится на 8.

Проясним этот алгоритм на *примере*. Начнём с того же числа 2010. Последние три цифры дают двузначное число 10, которое на 8 не делится. Следовательно, не делится и число года. Возьмём другое число - 123457928. Оставляем для проверки трёхзначное число 928. Число из первых двух цифр равно 92. Складываем его с половиной единиц - 4 - и получаем 96. Дальше действуем, как в *признаке делимости на 4*: 9 + 3 = 12. Это число кратно двум, поэтому всё число 123457928 делится на 8.

Признак делимости на 9

Этот признак напоминает правило для тройки. *Число делится на 9 тогда и только тогда, когда сумма его цифр делится на 9.* Раньше мы установили, что число 2010 делится на 3 и сумма его цифр также равна трём. Поэтому, согласно признаку делимости, на 9 оно не делится.

Если сумма цифр выражается не однозначным числом, то следует найти сумму его цифр. То есть действовать так же, как и в признаке делимости на 3.

Признак делимости на 10

Еще проще, чем признак делимости на 5. *Число делится на 10 тогда и только тогда, когда оно заканчивается на 0.* Например, число 2010 делится на 10.

Признак делимости на 11

Самое любопытное правило; не все его знают, но оно помогает очень просто определить, делится ли, например, номер автобусного билета на 11.

Чтобы узнать, делится ли число на 11, нужно подсчитать отдельно сумму цифр, стоящих на *нечётных* и *чётных* местах в исходном числе. Если они равны, то число кратно 11. В противном случае нужно из первой суммы вычесть вторую. Если разность делится на 11, то и всё число делится на 11.

Например, число **123453** делится на 11, так как 1 + 3 + 5 = 2 + 4 + 3 = 9. А число **123456** не делится (проверьте сами!).

Другой признак делимости на 11 полностью совпадает с признаком делимости на 7, но делить сумму чисел нужно на 11.

Признак делимости на 12

Число делится на 12 тогда и только тогда, когда *одновременно выполняются признаки делимости на 3 и 4*.

Признак делимости на 13

Признак делимости на 13 тот же самый, что и для чисел 7 и 11 (второй способ), но делить сумму чисел нужно на 13. Поэтому я недаром назвал этот признак универсальным.

Интересно, что наименьшее число, которое одновременно делится на 7, 11 и 13, равняется 7 х 11 х 13 = 1001, то есть сказочному числу арабских ночей.

Интересные математические фокусы, связанные с признаками делимости, вы найдёте в книге Мартина Гарднера *Математические досуги*, глава 19 **[9]**.

Делится - не делится?

Компьютер должен считать,
а человек - думать.

Программистская пословица

Мы вспомнили признаки делимости чисел, без которых человеку обойтись трудно, а вот компьютеру они совсем не нужны, потому что он для того и сделан, чтобы считать. И надо сказать, делает он это охотно и быстро.

Давайте затеем новый проект. И начнём мы его не с пустого места, а загрузим исходный текст программы *Rome2* в *ИСР* и сохра-

ним его в новой папке под именем **Delimost** (получилось коряво, но вы можете придумать и другое).

Почему мы действуем столь дерзко? - А мы всегда так будем поступать - чтобы не писать каждую программу снова да ладом. Если у нас в закромах и в сусеках уже есть похожая программа, то мы запросто можем использовать её как заготовку для следующей программы.

Итак, программа-шаблон загружена, начинаем её приспосабливать под свои нужды. Поскольку делить одно число на другое значительно проще, чем заниматься переводом чисел на чуждый нам язык, то из всех *переменных* мы оставим только две:

```
//ПРОГРАММА ДЛЯ ПРОВЕРКИ ДЕЛИМОСТИ ЧИСЕЛ

//variables
var
   number: integer;
   sNumber: string;
```

На что сразу следует обратить внимание: в первой же строке буквами ВО ВЕСЬ РОСТ объясняется *назначение программы*. Вы скажете, что всё и так понятно, зачем ещё бухгалтерию разводить? - Отвечу: сегодня понятно, а через месяц, когда у вас на диске скопится ворох таких программ, вспомните ли вы, для чего написали каждую из них? - Не вспомните! А если вы с кем-то поделитесь своими программами - как эти несчастные должны ими пользоваться? Вот так, по лени вы можете потерять уважение своих товарищей.

Далее нам встретится ещё переменная цикла *i*. Её тоже можно внести в список глобальных переменных программы, но делать это необязательно, и вот почему: без циклов не обходится практически ни одна программа, поэтому лучше объявлять переменные цикла только тогда, когда они понадобятся. Другие *вспомогательные* переменные также допустимо не указывать в списке глобальных переменных. Такие переменные

мы будем обозначать *одной* буквой (можно добавлять ещё цифру или букву), чтобы отличать их от более важных переменных, которым необходимо давать более *длинные* и обязательно *осмысленные* имена, чтобы не запутаться в их назначении.

Переходим к *основной части* нашей программы:

```
//=========================================
//          ОСНОВНАЯ ПРОГРАММА
//=========================================
begin
   CRT.SetWindowTitle('Делимость чисел');
```

Если вы хотите разделить длинную программу на *отдельные смысловые составляющие*, то делайте это так, как показано выше, то есть начинайте строку, как комментарий, а затем первую и третью строку заполните знаками равенства, звёздочками или тире. Во второй строке ЗАГЛАВНЫМИ буквами напишите название раздела программы.

Конечно, можно оформить надпись и по-другому, например, добавить *пустые строки*:

```
//=========================================
//
//          ОСНОВНАЯ ПРОГРАММА
//
//=========================================
```

Так заголовок будет выделен ещё лучше. Здесь же можно добавить пояснения о назначении и выполняемых действиях соответствующей части программы.

Так как все строки исходного текста *пронумерованы,* то с помощью команды меню *Правка > Перейти к строке* (клавиатурное сокращение *Ctrl+G*) вы можете совершить быстрый переход к нужной строке длинного кода (Рис. 8.1, слева). Обратите внимание, что в появившемся диалоговом окне (Рис. 8.1, справа) выводится диапазон допустимых номеров строк!

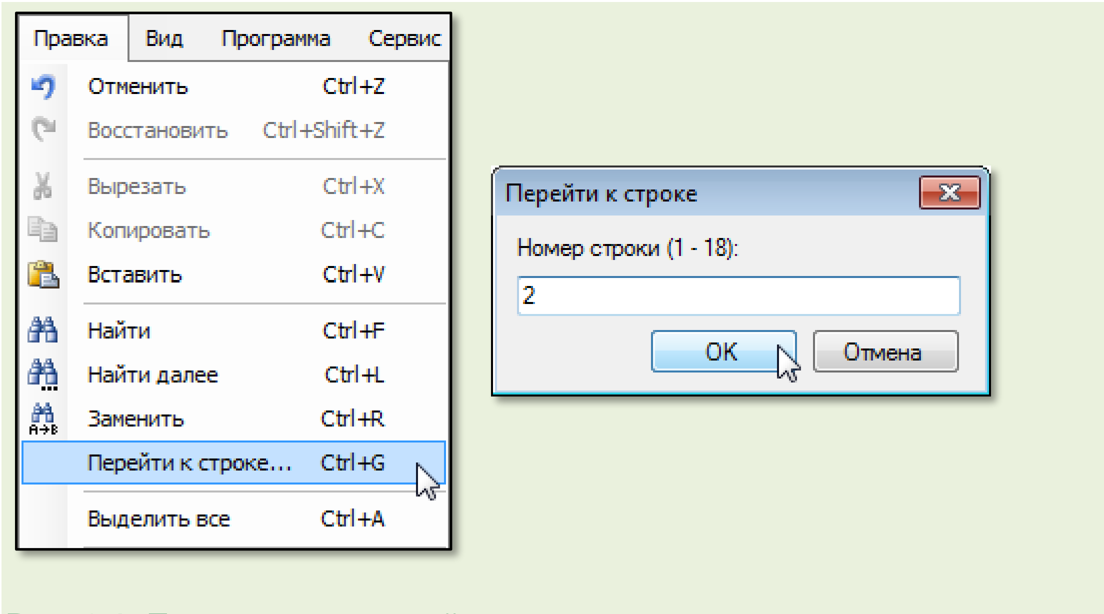


Рис. 8.1. Переходим к нужной строке

Для ввода и вывода информации нам потребуется *консольное окно*.

Пользователь должен ввести с клавиатуры *число*, делимость которого он хочет проверить, и нажать клавишу *ВВОД*. Значение переменной *number* станет равным этому числу:

```
  while(true) do
   begin
     repeat
       //считываем введённое число:
       TextColor(LightGreen);
       write('Введите число: ');
       TextColor(Yellow);
       number := ReadInteger();
```

Для того чтобы пользователь мог проверить *несколько* чисел, а не одно-единственное, мы организуем ввод в *бесконечном* цикле *while*.

Почему он **бесконечный**? – Условие продолжения цикла всегда равно *true*, то есть выполняется *всегда*, поэтому цикл никогда не закончится.

Это пример показывает, что иногда **и бесконечные циклы могут быть полезны!**

Закрыть программу можно, как обычно, то есть нажать на кнопку с крестиком в верхнем правом углу консольного окна, но неплохо и в самой программе предусмотреть такую возможность:

```
        //Если задан нуль, то работу с программой заканчиваем:
        if (number = 0) then exit;
    until (number >= 1);
```

Вряд ли кому-то придёт в голову проверять на делимость *нуль*, поэтому его вполне можно использовать как *условный знак* для прекращения работы с программой. Обратите внимание: чтобы проститься с программой, нам достаточно вызвать оператор *exit*.

Условие выполнения цикла *repeat* мы записали так, что пользователь должен ввести либо нуль, либо число, не меньшее двух:

```
until (number >= 1);
```

Это разумно: делимость единицы проверять как-то несолидно...

```
      writeln;
       //Проверяем, делится ли введённое число на числа  2..31:
       for i: integer := 2 To 31 do
       begin
          if (number mod i = 0) Then
          begin
             TextColor(Yellow);
             sNumber := 'Число делится на    ';
          end
          Else
          begin
             TextColor(CRT.LightRed);
             sNumber := 'Число не делится на ';
          end;

          //выравниваем числа в строке:
          if (i < 10) then
             sNumber += ' ';
          writeln(sNumber + i.ToString());
       end; //for
```

Если пользователь задал «правильное» число, то мы проверяем, делится ли оно на 2, 3, 4 и так далее, до 31. Как мы знаем из ма-

тематики, если одно целое число делится на другое целое число, то остаток от деления должен быть равен *нулю*. А это легко проверить с помощью операции *mod*. Результат выполнения этой операции - *остаток* от деления. Вот и вся премудрость: если остаток нулевой, то заданное число делится на *i*, в противном случае - не делится.

Для наглядности давайте выведем информацию на экран *разными цветами*: заданное число и делители числа пусть будут **жёлтыми**, а остальные числа - **красными**. В строковую переменную *sNumber* мы запишем результат проверки, а затем добавим один пробел для делителей меньше 10, чтобы вывод на экран был аккуратным.

Никогда не пренебрегайте *форматированием* вывода, иначе числа или слова могут быть прочитаны неверно!

Сформированную строку мы выводим на экран вместе с очередным делителем (как вы помните, он равен текущему значению переменной цикла *i*).

Ну, вот, мы и управились с проверкой заданного числа (Рис. 8.2).

Нам осталось после всех «информативных» строк добавить *пустую строку*, чтобы отделить ввод следующего числа:

```
    writeln;
   end;
end.
```

Поскольку мы устроили бесконечный цикл, пользователь может ввести новое число или, удовлетворив своё арифметическое любопытство, покончить с программой, введя нуль.

Не вводите очень длинные числа, иначе программа завершится с ошибкой!

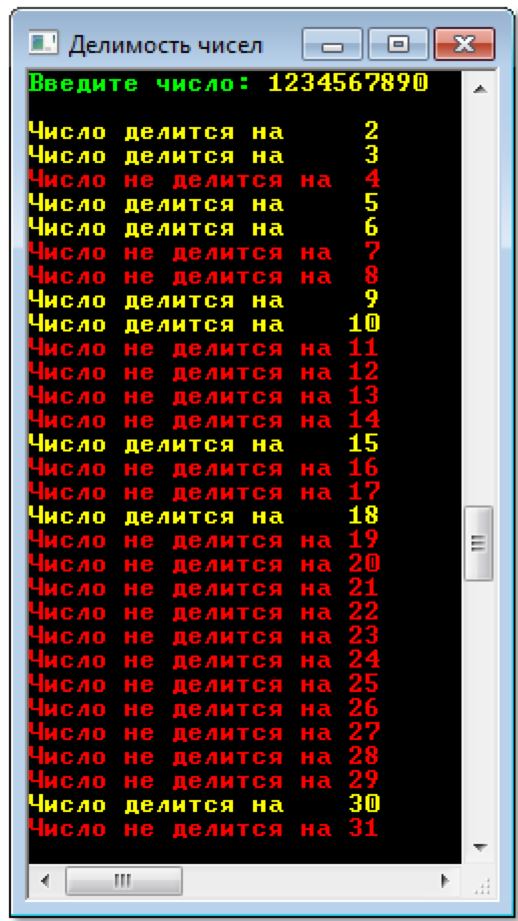


Рис. 8.2. Получилось очень красиво!

Исходный код программы находится в папке **Delimost**.

Цикл *For*

***Цикл For* используется в программах тогда, когда диапазон изменения какой-либо переменной точно известен.** В рассмотренном примере нам нужно было проверить, делится ли заданное число на другие последовательные числа от 2 до 31, поэтому мы поступили совершенно правильно, использовав именно этот вид циклов.

Цикл **For** для *единственного* оператора в теле записывается так:

```
For переменная_цикла:=начальное_значение  to конечное _ значение do
                                              Заголовок цикла

  оператор1;                          ← Тело цикла
```

Цикл **For** для *блока операторов* записывается так:

```
For переменная_цикла:=начальное_значение  to конечное _ значение do
                                              Заголовок цикла
begin
  оператор1;                          ← Тело цикла

  оператор2;

 . . .

 операторN;

end;                               ← Конец оператора цикла
```

А *работает* так:

1. Переменной цикла (она также называется *параметром* цикла) присваивается *начальное значение*.

2. Текущее значение переменной цикла сравнивается с её *конечным значением*. Если оно **меньше** конечного значения или **равно** ему, то последовательно выполняются операторы *оператор1, оператор2, ... , операторN*, составляющие *тело цикла.* Когда программа дойдёт до конца оператора цикла, она снова вернётся в заголовок, где переменная цикла получит следующее значение, на единицу больше текущего (то есть будет автоматически выполнен оператор присваивания *i := i + 1*). Значение переменной цикла увеличится на единицу, и цикл вернётся в начало *п.2*. Таким образом, переменная *i* последовательно принимает значения от 2 до 31, что нам и нужно. Если бы нам потребовалось перебрать первую *сотню* чисел, то мы бы записали заголовок цикла так: *For i := 1 to 100 do*.

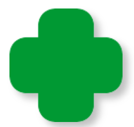
3. Если **больше**, то выполнение цикла заканчивается и управление переходит к следующей за ключевым словом *end* строке – для блока операторов или к строке, следующей за оператором в теле цикла – если он единственный.

При наборе цикла *for* удобно пользоваться шаблоном кода. Для единственного оператора в теле нажмите клавишу *f*, а затем *Shift+ПРОБЕЛ*:

```
for := to do
```

Для блока операторов – наберите буквы *fb* и нажмите клавиши *Shift+ПРОБЕЛ*:

```
for := to do
begin

end;
```



В качестве идентификатора *переменной цикла* часто выбирают короткие имена, часто из одной буквы – *i, j, k, l, m, n*.

Если начальное значение переменной цикла *больше* конечного, то цикл вообще не выполнится ни одного раза, а программа сразу перейдёт на строку, следующую за концом цикла.

Если какой-либо оператор в теле цикла использует переменную *i*, то её значение равно текущему. **Изменить значение переменной цикла в самом цикле нельзя!**

Иногда бывает необходимо, чтобы переменная цикла *уменьшала* своё значение. В этом случае вместо ключевого слова *to* в заголовке цикла следует записать *downto*:

For переменная_цикла:=начальное_значение **downto** конечное _ значение **do**

Естественно в этом случае *начальное значение* должно быть не меньше *конечного*, иначе цикл не выполнится ни разу.

Если параметр цикла должен изменять своё значение на число, отличное от +/-1, то удобнее использовать циклы *while* и *repeat*.

В языке программирования *PascalABC.NET* допускается объявлять переменную цикла непосредственно в его заголовке, как, например, в языке Си-шарп. Это можно сделать двумя способами:

переменная_цикла переменная: тип := начальное значение

var переменная_цикла переменная := начальное значение

Такая переменная доступна только в теле цикла (локальная переменная цикла), а после его окончания она уничтожается.

Программа у нас получилась дельная, но не на все случаи жизни, поэтому попробуйте усовершенствовать её!

1. Измените цикл *For* так, чтобы программа могла проверять делимость введённого числа, например, до 57.

2. Напишите программу *delimost2* (Рис. 8.3), в которой вместо переменной *number* заведите две переменные *dividend* (делимое) и *divisor* (наибольший делитель), чтобы пользователь мог самостоятельно выбирать конечное значение переменной цикла. Для ввода значений используйте такой код:

```
      repeat
         //считываем делимое:
         TextColor(LightGreen);
         write('Введите делимое:             ');
         TextColor(Yellow);
         dividend:= ReadInteger();
         //Если задан нуль, то работу с программой
заканчиваем:
         if (dividend = 0) then exit;
      until (dividend >= 1);

      repeat
         //считываем дедитель:
```

```
      TextColor(LightGreen);
      write('Введите наибольший делитель: ');
      TextColor(Yellow);
      divisor:= ReadInteger();
    until (divisor >= 1);
```

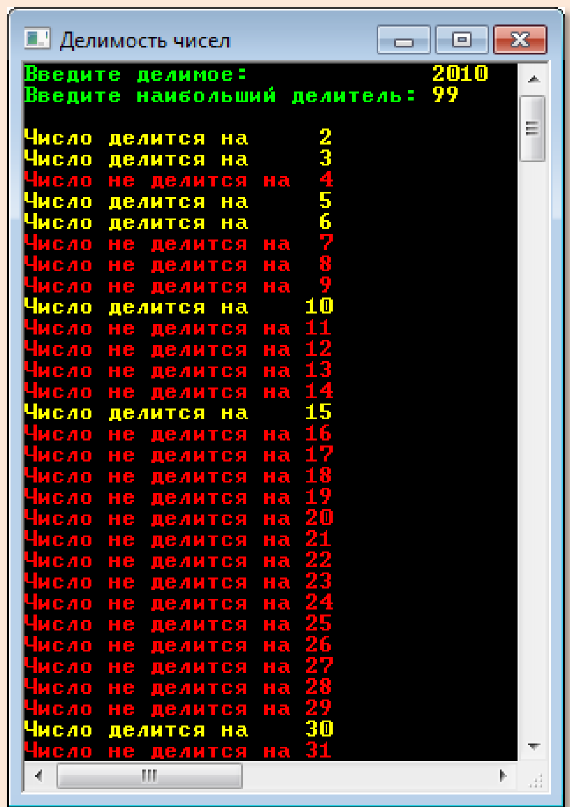


Рис. 8.3. Результат работы программы *delimost2*

В консольном окне хранятся только 300 последних строк, поэтому есть смысл не печатать строки с *отрицательным* результатом.

Либо проверяйте, кратно ли делимое только *наибольшему* делителю. Для этого начальное значение переменной цикла задайте равным конечному или просто уберите цикл вообще, оставив только тело цикла.

Исходный код программы находится в папке **Delimost**.

3. Из *Правил делимости чисел*, которые мы рассмотрели в начале урока, легко вывести признаки делимости чисел на 15, 16, 18, 20, 22, 24, 25, 100, 1000. Попробуйте!

4. Докажите, что число, у которого число тысяч равно числу единиц, а число сотен равно числу десятков, делится на 11.

5. Докажите, что если *двузначное* число в 4 раза больше суммы цифр, то оно делится на 12.

Урок 9. Простые числа

Простыми называются натуральные числа, имеющие в точности *два разных* делителя. Из этого определения следует, что ни нуль, ни единица к простым числам не относятся. Также очень легко установить, что первое простое число - это *двойка*, потому что она делится на единицу и на саму себя (двойка – единственное *чётное* простое число). Дальше мы легко найдем тройку, пятёрку, семёрку. Нам не составит труда продолжить этот ряд: 11, 13, 17, 23, 29, 31. Чтобы отбросить многие *составные* числа, достаточно воспользоваться *признаками делимости*, которые мы рассмотрели на уроке математики. Но что вы скажете о числе 2011? Ни один признак делимости не действует, но это всё равно не позволяет нам однозначно сказать, простое это число или составное. Конечно, мы можем поделить 2011 на все подходящие числа, но если нам потребуется проверить число 2017? Или 514229? Или того пуще - 39916801! Вот в таких случаях правильнее один раз написать программу, чем каждый раз считать вручную. Этим мы сейчас и займёмся.

За основу нового проекта мы возьмем программу *delimost*, которую вы немедля должны загрузить в *ИСР* и тут же записать в новую папку под именем **Prime**.

Переделать нам придётся совсем немного, поскольку действие обеих программ схоже. Нам потребуется только одна *переменная*, назначение которой понятно без слов:

```
//ПРОГРАММА ДЛЯ ПРОВЕРКИ ЧИСЕЛ НА ПРОСТОТУ

var
   number: integer;
```

Дальше идут обычные *проверки*, на которых мы также не будем останавливаться:

```
//===========================================
//          ОСНОВНАЯ ПРОГРАММА
//===========================================
begin
```

```
  CRT.SetWindowTitle('Простые числа');
  while(true) do
  begin
     repeat
        //считываем введённое число:
        TextColor(LightGreen);
        write('Введите число больше 1: ');
        TextColor(Yellow);
        number := PABCSystem.ReadInteger();
        //Если задан нуль, то работу с программой заканчиваем:
        if (number = 0) then exit;
     until (number > 1);
```

А теперь - самое интересное: *главная часть* программы, которая всё будет делать за нас (вы, наверное, помните, что Вовка в Тридевятом царстве хотел того же, но только для себя):

```
      var flg:= true;
      writeln;
      //Проверяем, делится ли введённое число на числа
      //2..корень квадратный из числа number:
      for var i:= 2 To Floor(Sqrt(number)) do
      begin
         if (number mod i = 0) Then
         begin
            TextColor(LightRed);
            writeln('Число составное');
            flg := false;
            break;
         end
      end; //for
      if (flg) then begin
         TextColor(LightGreen);
         writeln('Число простое');
      end;
      writeln;
   end;
end.
```

Обратите пристальное внимание на *заголовок цикла*: мы будем проверять делимость заданного числа на все числа из диапазона *2..корень квадратный* из заданного числа. Для двойки корень

квадратный равен 1.414, то есть меньше начального значения переменной цикла, поэтому цикл выполняться не будет, а программа сразу перейдёт на строку, следующую за строкой

```
end; //for
```

И она сообщит нам, что двойка - простое число. С тройкой будет та же самая история. Для четвёрки начальное и конечное значения цикла совпадут, поэтому цикл выполнится 1 раз. Поскольку остаток от деления четвёрки на двойку равен нулю, то мы получим сообщение, что четвёрка - число составное.

Для следующих чисел *проверка* проходит так: заданное число последовательно делится на все числа, начиная с двойки и кончая корнем квадратным из заданного числа. Так как любое число делится на единицу и само на себя, то два разных делителя у него имеются в любом случае (кроме, единицы, разумеется). Если мы обнаружим ещё *хотя бы один* делитель, то это будет перебор: число заведомо составное - проводить испытания дальше смысла нет. Мы устанавливаем флаг в *false* и выполняем оператор *break* для досрочного завершения цикла *for*:

```
flg := false;
break;
```

Если же в теле цикла значение логической переменной *flg*, первоначально установленной в *true*, не изменится, значит, число *number* – простое и достойно быть напечатанным в консольном окне зелёным цветом:

```
if (flg) then begin
   TextColor(LightGreen);
   writeln('Число простое');
end;
```

Обратите внимание на заголовок цикла:

```
for var i:= 2 To Floor(Sqrt(number)) do
```

Переменная цикла *i* может принимать только *целые* значения, а корень квадратный из числа *number* (его возвращает функция *sqrt*) – число *вещественное*. Нам приходится округлять его до целого с помощью функции *floor*.

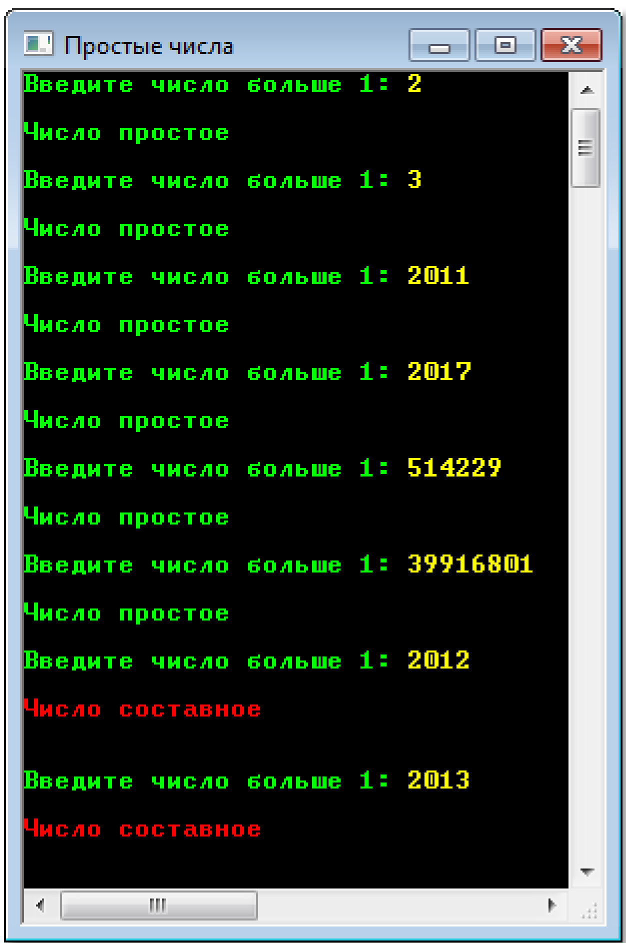


Рис. 9.1. Наша программа справляется с любыми числами!

Как видите, программа у нас получилась короткая и простая, но даже большие числа она проверяет очень быстро (Рис. 9.1).

Помните, что паскаль не умеет работать с ОЧЕНЬ БОЛЬШИМИ ЧИСЛАМИ, поэтому не мучайте программу глупыми вопросами.

Исходный код программы находится в папке **Prime**.

Условный оператор *if*

Мы уже несколько раз встречались в своих программах с **условным оператором *if***, и это не случайно: трудно себе представить программу, которая обошлась бы без него.

С условными операторами будущий программист знакомится уже в раннем детстве благодаря педагогическому усердию родителей. Мама скажет так: ***Если** ты будешь хорошо себя вести, **то** получишь большую сладкую конфету, а **иначе** не пойдёшь гулять во двор*. В переводе на язык паскаля мамины посулы выглядели бы так:

if условие **then**
  результат1;
else
  результат2;

Здесь:
условие – хорошее поведение;
результат1 – конфета;
результат2 – временное ограничение свободы.

Действует эта воспитательная конструкция – как в жизни: **если** (*if*) условие соблюдено, **то** (*then*) подопечный получает конфету, **не** соблюдено (*иначе - else*) – принудительная домашняя отсидка.

Более лапидарный родитель, коим является отец семейства, изложил бы свои требования в более категоричной форме: ***Если** получишь двойку, **то** выпорю*. Тут уж никакой надежды на конфеты и другие сладостные изделия, то есть альтернатива жёсткая: принёс из школы двойку – получил ремня, не принёс – избежал ремня:

if условие **then**
  результат;

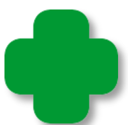
В школе порка запрещена, поэтому там так мало настоящих педагогов-мужчин.

Условный оператор if служит для того, чтобы изменять порядок выполнения операторов в программе в зависимости от некоторого логического условия. Он имеет две формы – *сокращённую* («папину»):

```
if условие  then
begin
   оператор1;
   оператор2;
  . . .
  операторN;
end;
```

и *полную* («мамину»):

```
if условие then
begin
   оператор1;
   оператор2;
   . . .
   операторN;
end
else
begin
   операторN+1;
   операторN+2;
   . . .
   операторN+M;
end;
```



Если в одной из ветвей *if/else* (или в обеих) находится *единственный* оператор, то операторные скобки *begin-end*

ставить не нужно. С другой стороны, при отладке программы вы можете добавить ещё один оператор, что вызовет ошибку, поэтому **не ленитесь ставить операторные скобки, это поможет вам избежать многих ошибок**.

Перед ключевым словом *else* точку с запятой ставить нельзя!

Условие в этих записях – обычное логическое выражение, в котором используются знаки операций сравнения =, <, > и другие. Результат логического выражения может быть либо *истинным* (условие выполняется), либо *ложным* (не выполняется).

Действует условный оператор так.

Если *условие* удовлетворено, **то** выполняются операторы после ключевого слова **then**. Если **не** удовлетворено, то для сокращённой формы все операторы пропускаются, а управление передаётся следующему за ключевым словом *end* оператору. Для полной формы выполняются операторы между ключевыми словами *else* и *end*.

Например, если мы захотим найти большее (*max*) из двух чисел *n1* и *n2*, то легко сделаем это с помощью условного оператора:

```
 var n1:=1;
var n2:=2;
var max: integer;

if n1 > n2 then
   max:= n1
else
  max:= n2;
```

Довольно часто для наглядности выражение, определяющее условие, записывается в *скобках*:

```
if (n1 > n2) then
```

Особенно важно ставить скобки, если условие состоит из *нескольких* выражений, объединённых знаками *логических операций*:

```
if (n1 = 1) and (n2 = 2) then …
if (n1 = 1) or  (n2 = 2) then …
```

Более того, в таких случаях скобки ставить *необходимо*!

При наборе условного оператора удобно пользоваться шаблоном кода. Для единственного оператора в теле наберите буквы *if*, а затем *Shift+ПРОБЕЛ*:

```
if then
```

Для блока операторов – напечатайте буквы *ifb* и нажмите клавиши *Shift+ПРОБЕЛ*:

```
if then
begin

end;
```

Для *полной* формы оператора имеется такой шаблон кода:

```
if then
begin

end
else
begin

end;

end;
```

Для его вызова нужно набрать буквы *ifeb* и нажать клавиши *Shift+ПРОБЕЛ*.

Логические операции

and - логическая операция **И.**
Результат операции тогда и только тогда будет *истинным*, если истинны одновременно *оба* операнда:

```
(7 > 13)  and (7 < 13)  → ложно
(7 <> 13) and (7 < 13)  → истинно
```

or - логическая операция **ИЛИ**.
Результат операции будет *истинным*, если *хотя бы один* операнд истинен:

```
(7 > 13)  or (7 < 13)  → истинно
(7 <> 13) or (7 < 13)  → истинно
```

Редактор кода

Исходный текст программы – это обычный текст, который можно редактировать хотя бы в программе *Блокнот*, которая входит в состав операционной системы *Windows.*

Он состоит из отдельных строк, в каждой из которых записан оператор или комментарий. *Пустые* строки не содержат ни одного символа и служат для отделения логических частей программы друг от друга.

Конечно, в *Блокноте* слова не будут выделяться цветом, как в *Редакторе кода* самого *паскаля*, но в остальном текст в *Блокноте* ничем не отличается от исходного текста в *Редакторе кода*. Это значит, что мы свободно можем копировать текст из *ИСР* в любой текстовый редактор и наоборот.

Основные возможности *Редактора кода* ничем не отличаются от таковых в других подобных программах.

Чтобы **выделить *весь* текст**, одновременно нажмите клавиши *CTRL + A* или выполните команду меню *Правка > Выделить все* (Рис. 9.2).

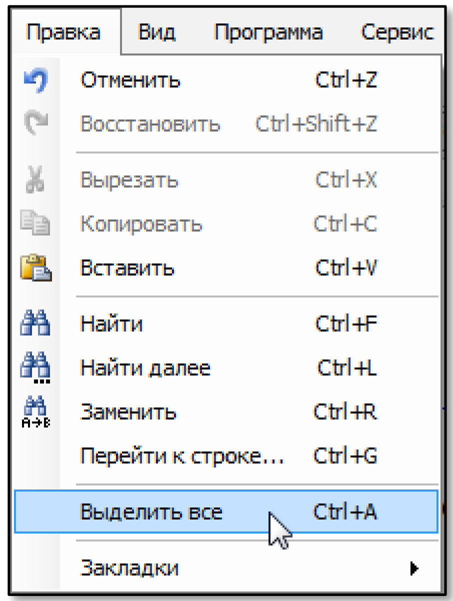


Рис. 9.2. Выделяем *весь* текст

Чтобы **выделить *фрагмент* текста**, установите курсор в начало первой строки фрагмента, нажмите левую кнопку мышки и ведите курсор в его последнюю строку. Выделенный кусок текста окрасится в **синий** цвет (Рис. 9.3).

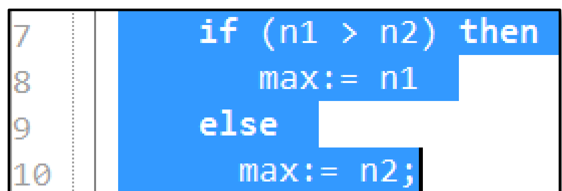


Рис.9.3. Выделяем *часть* текста

Удалить выделенный текст можно клавишами *Del* и *Backspace*.

Удалить выделенный текст **в буфер обмена** можно командой главного меню *Правка > Вырезать* (Рис. 9.4)

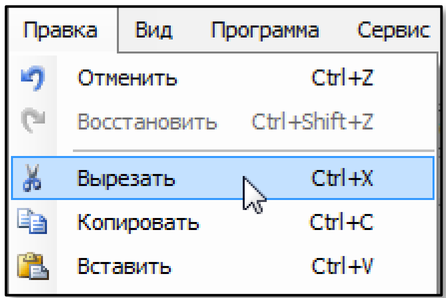


Рис. 9.4. Убираем выделенный текст в буфер

или клавишами *Ctrl + X*.

Скопировать текст в буфер обмена можно командой меню *Правка > Копировать* или клавишами *Ctrl + C*.

Чтобы **вставить** текст из буфера обмена в позицию курсора, выполните команду меню *Правка > Вставить* или нажмите клавиши *Ctrl + V*.

Чтобы **отменить** удаление (или другую операцию редактирования), выполните пункт меню *Правка > Отменить* (Рис. 9.5).

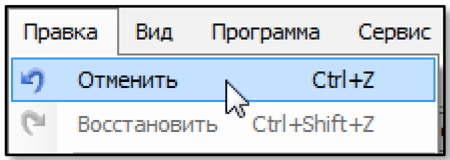


Рис. 9.5. Идём на попятную

или нажмите клавиши *Ctrl + Z*.

Вернуть отмененную операцию можно командой *Правка > Восстановить* или кнопками *Ctrl + Shift +Z*.

Выделенный фрагмент текста можно *перетащить* на другое место, ухватившись за него мышкой. Если при этом удерживать нажатой клавишу *Ctrl*, то выделенный фрагмент будет *скопирован* в новое место.

Если вы нажмёте *правую кнопку мыши*, то появится **всплывающее (контекстное) меню** (Рис. 9.6), в котором также имеются основные команды *редактирования* текста.

Здесь же вы найдёте ещё несколько очень полезных команд.

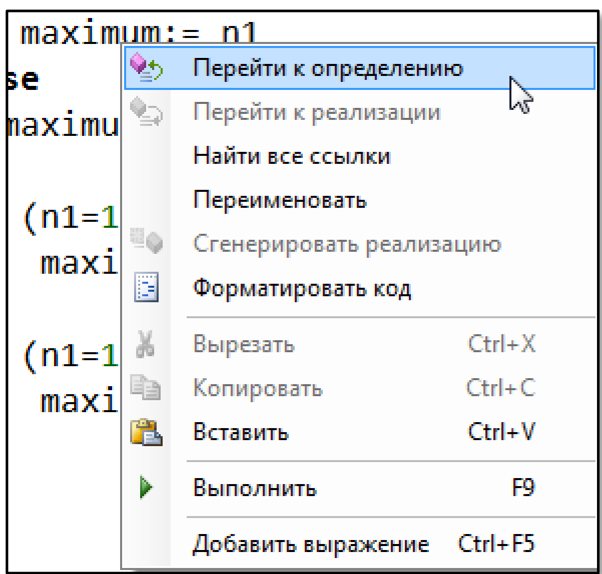


Рис. 9.6. Всплывающее меню

Чтобы найти все строки, в которых записан нужный вам идентификатор, выделите его и в контекстном меню выполните команду *Найти все ссылки* (Рис. 9.7).

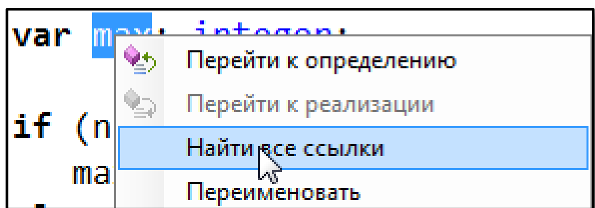


Рис. 9.7. Ищем все строки с идентификатором

В окне *Результат поиска ссылок* будет выведен список всех строк, в которых встречается идентификатор, с указанием номера строки и начальной позиции слова и (Рис. 9.8).

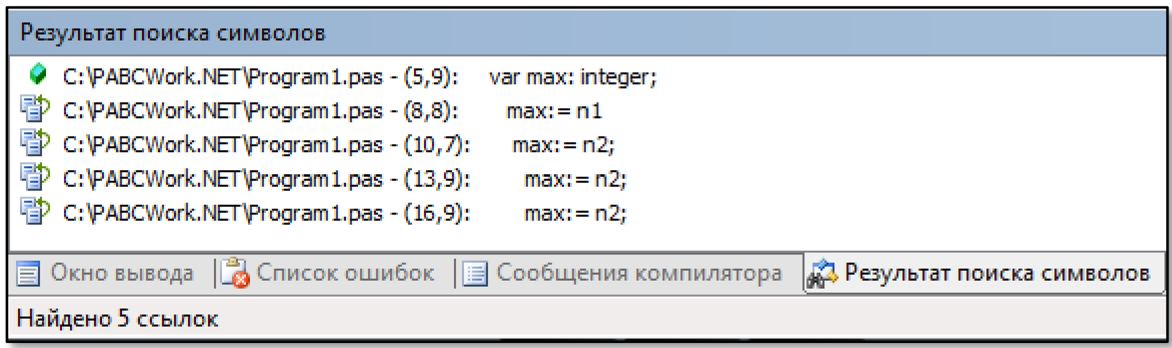


Рис. 9.8. Ссылки найдены!

Следующая команда контекстного меню поможет вам *переименовать* идентификатор во всём исходном коде (Рис. 9.9).

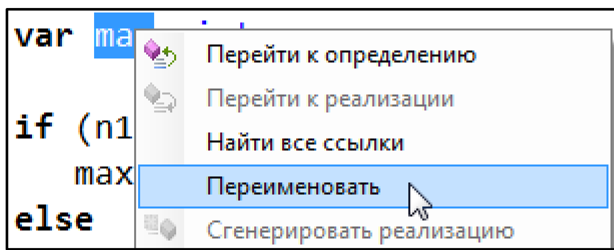
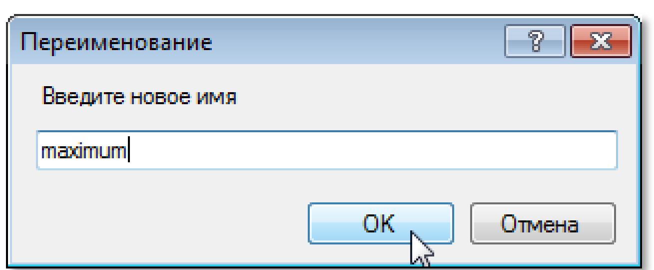


Рис. 9.9. Переименовываем порося в карася

В окне *Переименование* напечатайте новое имя и нажмите кнопку *ОК* (Рис. 9.10).




```
var maximum: integer;

if (n1 > n2) then
   maximum:= n1
else
  maximum:= n2;
```

Рис. 9.10. Переименовали *max* в *maximum*

Самая верхняя команда контекстного меню *Перейти к определению* перенесёт вас в строку, в которой была объявлена переменная, процедура или функция (Рис. 9.11).

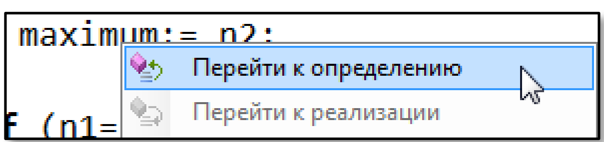


Рис. 9.11. Где начало?

Чтобы **отыскать** нужное слово в тексте, можно воспользоваться и командой главного меню *Правка > Найти* (клавиатурное сокращение *Ctrl+F*) (Рис. 9.12).

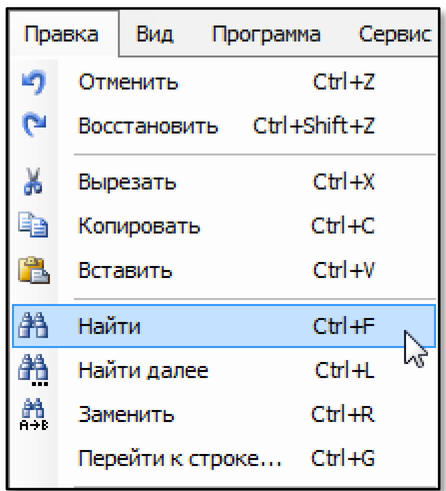


Рис. 9.12. Поиски начинаются

В появившемся диалоговом окне введите слово для поиска и нажмите кнопку *Искать далее* (Рис. 9.13).

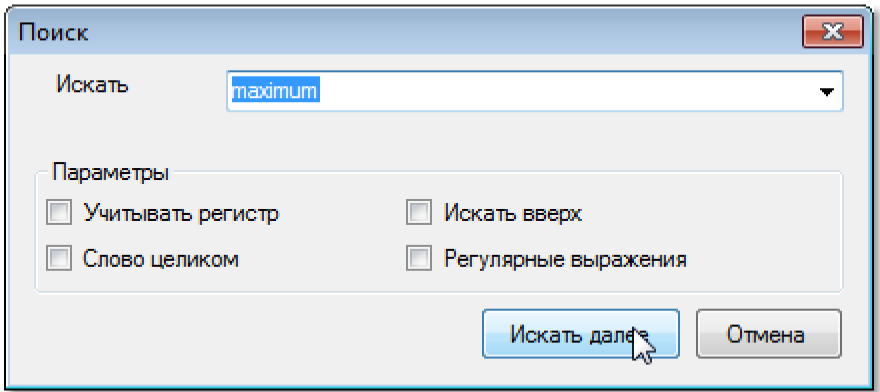


Рис. 9.13. Диалоговое окно поиска слова

Если вы предварительно выделите какое-либо слово, то оно автоматически появится в окне *Искать*.

Первое найденное слово от позиции курсора будет выделено ***синим*** цветом (Рис. 9.14).

Нажимая кнопку *Искать далее*, вы последовательно будете переходить к следующим найденным словам.

```
12        if (n1=1) and (n2=2) then
13           maximum:= n2;
```

Рис. 9.14. Слово найдено!

Ещё проще выделить слово и нажимать клавиши **Ctrl+L**.

Команда меню *Правка > Заменить* (Рис. 9.15) позволяет переименовать идентификатор во всём тексте или только в его части.

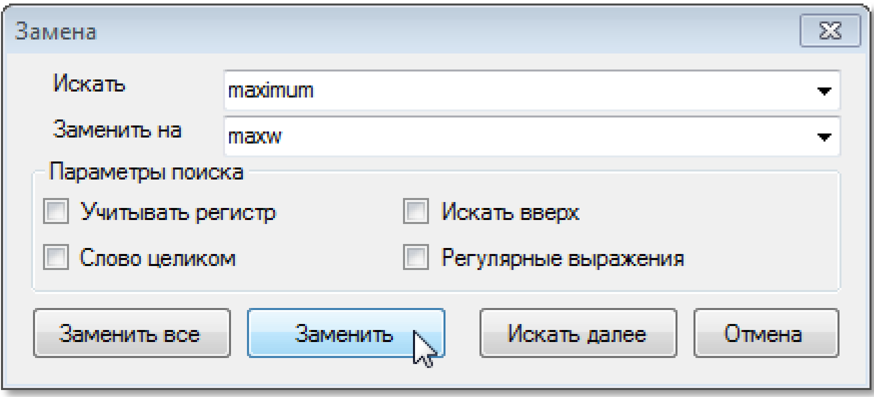


Рис. 9.15. Меняем шило на мыло

И последняя по счету на сегодня, но далеко не последняя по значимости команда меню – *Сервис > Форматировать код* (Рис. 9.16).

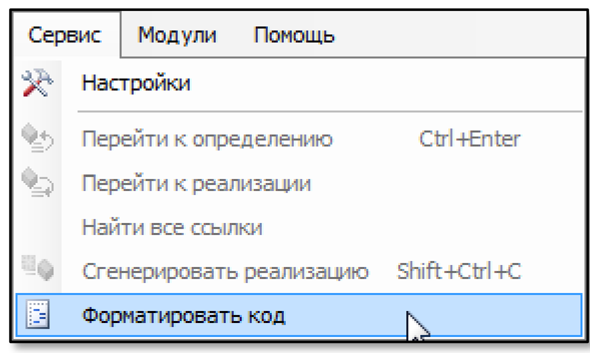


Рис. 9.16. Не формат!

Наберём текст «всмятку»: так, чтобы все операторы были записаны от начала строки (Рис. 9.17).

```
43        while (number > 0) do
44        begin
45        inc(n);
46        while (ARABIC[n] <= number) do
47        begin
48        sNumber := sNumber + ROME[n];
49        number := number - ARABIC[n];
50        end
51        end;
```

Рис. 9.17. Не по-паскалевски!

Теперь совсем непросто найти начало и конец вложенных циклов (если добавить ещё несколько строк в тело циклов, то сделать это будет гораздо труднее, чем в этом коротком примере!). Но выполним команду *Форматировать код* - и строчки будут напечатаны с *отступом* от левого края (Рис. 9.18).

```
43          while (number > 0) do
44          begin
45             inc(n);
46             while (ARABIC[n] <= number) do
47             begin
48                sNumber := sNumber + ROME[n];
49                number := number - ARABIC[n];
50             end
51          end;
```

Рис. 9.18. А вот это по-нашему!

Именно так принято записывать исходный текст на языке паскаль, а теперь *паскалевский* способ применяется во всех языках программирования.

После «принудительного» форматирования комментарии могут «уплыть», так что присматривайте за ними!

Вы можете и сами делать отступы, набирая сходный текст. Как обычно, для этого нужно нажать необходимое число раз клавиши *ПРОБЕЛ* или *TAB* (табуляции). Но иногда можно и запутаться, де-

лая отступы вручную, а автоматическое форматирование работает быстро и без ошибок! Более того, оно поможет вам найти пропущенные ключевые слова. Например, удалим в строке 51 слово *end*, которое обозначает конец внешнего цикла *While* (Рис. 9.19).

```
43        while (number > 0) do
44        begin
45        inc(n);
46        while (ARABIC[n] <= number) do
47        begin
48        sNumber := sNumber + ROME[n];
49        number := number - ARABIC[n];
50        end
51
```

Рис. 9.19. Преднамеренная ошибка

Пропущенное слово можно и не заметить, но давайте выполним команду *Форматировать код* (Рис. 9.20).

```
42        while (number > 0) do
43        begin
44        inc(n);
45        while (ARABIC[n] <= number) do
46        begin
47        sNumber := sNumber + ROME[n];
48        number := number - ARABIC[n];
49            //writeln(sNumber);
50        end
51
52        writeln(sNumber);
```

Рис. 9.20. Ошибка найдена!

Редактор кода окрашивает в красный цвет следующую строку, сигнализируя о допущенной ошибке.

А ещё *Редактор кода* умеет выделять серым цветом парные операторные скобки *begin-end*. Щёлкните по одному из этих слов – и сразу увидите всю пару (Рис. 9.21).

```
44        begin
45           inc(n);
46           while (ARABIC[n] <= number) do
47           begin
48              sNumber := sNumber + ROME[n];
49              number := number - ARABIC[n];
50           end
51        end;
```

Рис. 9.21. Операторные скобки

Редактор кода находит и другие парные скобки – круглые и квадратные:

```
while (number > 0) do
sNumber := sNumber + ROME[n];
```

Некоторые возможности *Редактора кода* мы уже рассмотрели раньше. Поэтому только вкратце вспомним их.

1. Все строки исходного текста *пронумерованы* слева, что облегчает навигацию по длинному документу. Числа в правом нижнем углу *Редактора кода* показывают номер *текущей строки* и позицию текстового *курсора* в ней (Рис. 9.22).

```
52        writeln(sNumber);
```

```
Строка 52    Столбец 18
```

Рис. 9.22. Позиция текстового курсора

2. При наборе операторов и идентификаторов на экране появляется *подсказка*, в которой можно выбрать нужное слово, а также получить информацию о нём.

3. Ещё больше сведений предоставляет *Справочная система*. Для того чтобы больше узнать о каком-либо элементе языка паскаль, установите текстовый курсор на нужном слове,

щёлкнув на нём мышкой и нажмите клавишу *F1*. Появится окно справки (Рис. 9.23) с полной информацией.

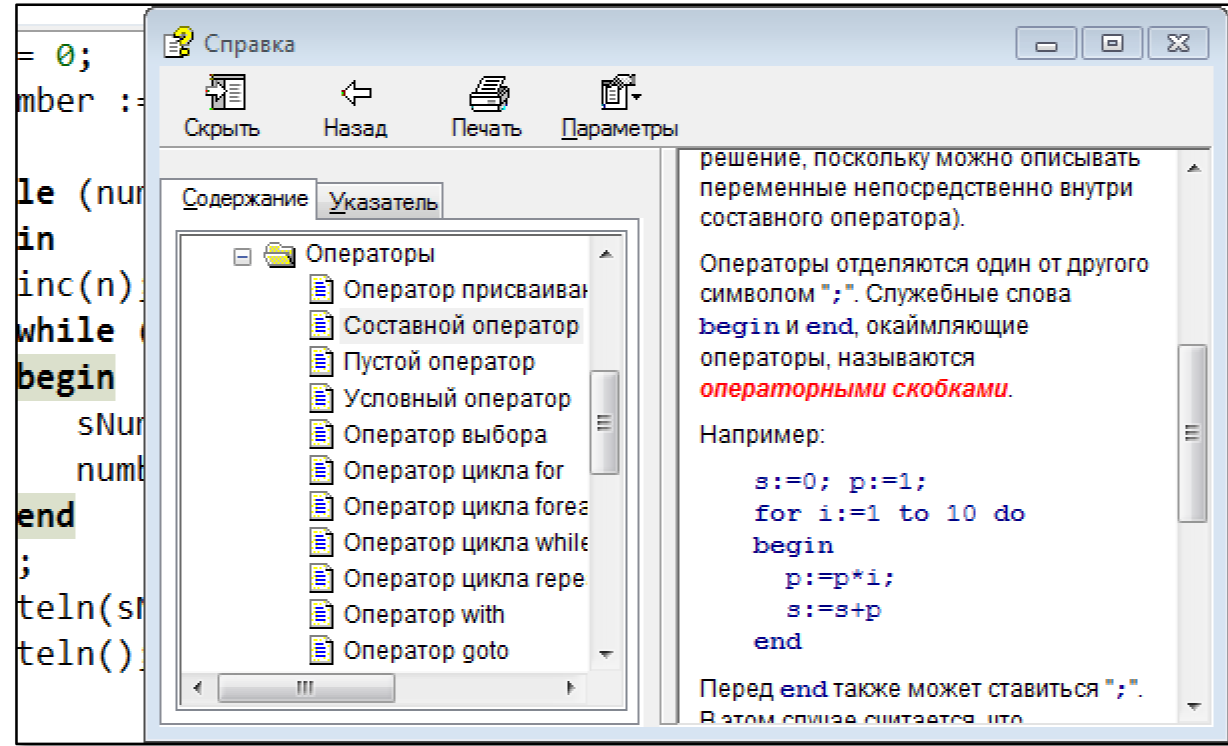


Рис. 9.23. Нас интересует ключевое слово *begin*

А задание вам такое: хорошенько изучите все возможности *Редактора кода* и поупражняйтесь в редактировании текста!

Урок 10. Файлы

Граждане, храните деньги в сберегательной кассе!

Жорж Милославский

Граждане, храните данные в файлах!

Программистская пословица

На предыдущем уроке мы научились ловко определять, простое ли заданное пользователем число или составное. Но представим себе ситуацию, что нам потребовался список всех простых чисел в каком-либо диапазоне, например от 2 до 10000. Не будем же мы тысячи раз вводить числа с клавиатуры и последовательно проверять их, а затем найденные простые числа заносить в список! Действительно, такими нудными делами должен заниматься компьютер, а наша задача – объяснить ему на простом, паскалевском языке суть проблемы.

Так как поиск будет вестись в *диапазоне* чисел, то нам потребуются *две* переменные для хранения *начала* и *конца* диапазона, а также вспомогательная переменная *number* и файловая переменная *f* для записи данных в файл:

```
//ПРОГРАММА ДЛЯ ПОИСКА ПРОСТЫХ ЧИСЕЛ
//В ЗАДАННОМ ДИАПАЗОНЕ

uses CRT;

var
   start: integer;
   finish: integer;
   number: integer;
   f: textfile;
```

Основную часть программы мы начинаем с ввода и проверки чисел, задающих *начало* и *конец* диапазона:

```
//========================================
//          ОСНОВНАЯ ПРОГРАММА
//========================================
begin
   CRT.SetWindowTitle('Простые числа');

   while(true) do
   begin
      TextColor(LightGreen);
      write('Введите начало диапазона: ');
      TextColor(Yellow);
      start := ReadInteger();

      //Если задан нуль, то работу с программой заканчиваем:
      if (start = 0) then exit;
      if (start < 2) then start:= 2;

      repeat
         TextColor(LightGreen);
         write('Введите конец  диапазона: ');
         TextColor(Yellow);
         finish := ReadInteger();
      until (finish >= start);
```

Здесь ничего нового для нас нет.

Закончив проверки, мы последовательно перебираем в цикле *For* числа из заданного диапазона, то есть переменная цикла *j* принимает значения от *start* до *finish*:

```
      Assign(f, 'primes.txt');
      //rewrite(f); //переписать файл
      append(f);    //добавить к файлу
      writeln;
      var flg:= true;
      for var j:= start To finish do
      begin
         number:=j;
         flg := true;
         //Проверяем, делится ли введенное число на числа
         //2..корень квадратный из числа;
         for var i:= 2 To Floor(Sqrt(number)) do
         begin
```

```
                if (number mod i = 0) Then
                begin
                   TextColor(LightRed);
                  //writeln(number.ToString() + ' Число составное');
                   flg := false;
                   break;
                end
             end; //for i
             if (flg) then
             begin
                TextColor(LightGreen);
                writeln(number.ToString() + ' Число простое');
                writeln(f, number.ToString());
             end;
          end;    //for j

          writeln;
          writeln(f, '');
          close(f);
       end;
    end.
```

Во вложенном цикле *for* мы определяем, как и раньше, простоту числа *number* (оно равно *j* и оставлено только для «обратной совместимости» с предыдущей программой). Если оно составное, то мы переходим к проверке следующего числа. Если же простое, то выводим его на экран (Рис. 10.1).

Так в консольное окно будут выведены все простые числа из заданного диапазона, например, 1..1000. Однако в консольном окне сохраняются только последние 300 строк, поэтому начало списка исчезнет навсегда. Конечно, можно взять маленький диапазон, чтобы все простые числа сохранялись в консольном окне. Но и этот способ не очень хорош: всё равно придётся переписывать на бумагу простые числа вручную, а это очень долго и чревато ошибками. Поэтому мы каждое новое простое число запишем в файл на диске.

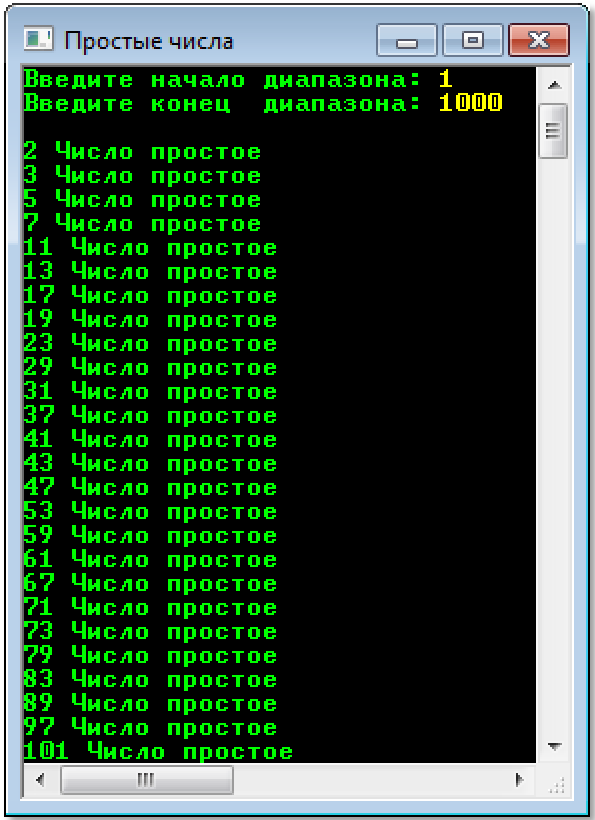


Рис. 10.1. Список простых чисел

Для этого необходимо

- *объявить файловую переменную* типа *textfile* для записи данных в *текстовый файл*:

f: textfile;

- *связать файловую переменную с файлом на диске* с помощью процедуры *assign*:

```
Assign(f, 'primes.txt');
```

В скобках нужно указать имя файловой переменной и имя файла вместе с расширением.

- *открыть файл для записи* с помощью процедуры *append*:

append(f);

Если файла с заданным именем на диске нет, то он будет создан. В противном случае вся информация в этом файле сохранится без изменений, а новая будет добавлена в конец файла. Это значит, что если вы будете пользоваться программой несколько раз, все данные будут записаны в один и тот же файл. Если вас это не устраивает, то переименуйте файл на диске или каждый раз изменяйте в программе имя файла. Другой способ – вместо *append* использовать процедуру *rewrite* , которая полностью переписывает содержимое файла (будьте осторожны, чтобы не уничтожить важную информацию!):

rewrite(f);

Когда программа найдёт простое число, она не только напечатает его в консольном окне, но и запишет в файл:

```
if (flg) then
begin
   TextColor(LightGreen);
   writeln(number.ToString() + ' Число простое');
   writeln(f, number.ToString());
end;
```

Как видите, для этого мы пользуемся той же процедурой *writeln*, что и для печати данных в консольном окне, но дополнительно указываем и файловую переменную *f* (то есть, по сути, имя файла, с которым она связана).

- После вывода всех данных необходимо *закрыть файл* процедурой *close*:

close(f);

В результате наших усилий на диске, в папке с выполняемым файлом программы появится текстовый файл *primes.txt*, который можно открыть в любом текстовом редакторе (Рис. 10.2), посмотреть, изменить, скопировать в любой документ или, наконец, распечатать на принтере!

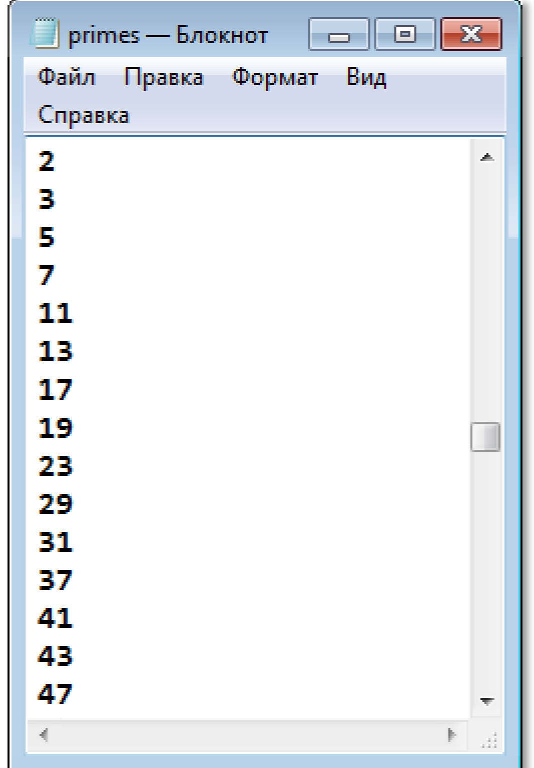


Рис. 10.2. Весь список – пожалуйста!

Итак, для записи данных в текстовый файл необходимо выполнить следующие операции:

1. Объявить файловую переменную (обычно глобальную):

```
var f: textfile;
```

2. Связать её с файлом:

```
assign(f, 'primes.txt');
```

3. Открыть файл для записи. Процедура

```
append(f);
```

добавляет новые данные в конец файла, при этом прежняя информация сохраняется.

А процедура

```
rewrite(f);
```

полностью переписывает файл.

4. Процедура *writeln* записывает число или строку в файл:

```
writeln(f, number);
```

5. После записи данных в файл его нужно закрыть:

```
close(f);
```

Исходный код программы находится в папке **Primes**.

Решето Эратосфена

Греческий математик Эратосфен придумал способ поиска простых чисел, который в его честь назвали *решетом Эратосфена*. Мы могли бы повторить его научный подвиг на бумаге, но лучше воспользоваться компьютером, тем более что алгоритм поиска незамысловатый, и мы без труда переведем его на язык паскаля.

В этом случае нам достаточно знать только *конец* диапазона чисел (переменная *endNumber*), поскольку поиск всегда начинается с *двойки*, а сами натуральные числа мы будем записывать в массив *number*:

```
//РЕШЕТО ЭРАТОСФЕНА

uses CRT;

//variables
var
   endNumber := 0;
   number: array of integer;
   prime := 0;
   f: textfile;
```

Переменную *prime* мы отведём под текущее простое число.

```
//=========================================
//           ОСНОВНАЯ ПРОГРАММА
//=========================================
Begin
   CRT.SetWindowTitle('Решето Эратосфена');
   Assign(f, 'resheto.txt');

   while(true) do
   begin
      TextColor(LightGreen);
      write('Введите конец  диапазона: ');
      TextColor(Yellow);
      endNumber := ReadInteger();
      //Если конец диапазона равен 0, то программу закрываем:
      if (endNumber = 0) then exit;
      if (endNumber < 2) then endNumber:= 2;
      writeln;
```

Дальше мы действуем по *алгоритму Эратосфена*:

1. Записываем все числа, начиная с двойки, в массив:

```
    //Формируем массив натуральных чисел 2..end:
      number:= new integer[endNumber+1];
      For var i:= 2 To endNumber do
          number[i]:=  i;
```

2. Первое простое число равно двум:

```
prime := 2;
```

3. «Вычёркиваем» из массива число *prime*prime*, а затем все числа, начиная с этого числа, через *prime*:

4 – 6 – 8 - … для *prime= 2,*
9 – 12 – 15 - … для *prime= 3*.

И так далее:

```
    repeat
         var i := prime * prime;
         while (i <= endNumber) do
```

```
      begin
         number[i] := 0;
         i += prime;
      end;
```

Конечно, мы не можем зачеркнуть число в массиве, поэтому присваиваем соответствующему элементу массива значение *нуль*, которое будет означать, что это число - *составное*.

Поскольку переменная цикла *i* должна изменять своё значение на число *prime*, отличное от единицы, в каждой итерации, то мы не можем употребить здесь цикл *for*, с которым мы познакомились на восьмом уроке. Он встречается в программах чаще всего. Переменная цикла *for* автоматически увеличивается (или уменьшается) на *единицу* после каждого выполнения тела цикла (итерации). Но в этой программе нам потребовалось увеличивать переменную цикла на число *prime*, которое не равно единице. Здесь нужно использовать цикл *while*.

4. Ищем первое «невычеркнутое» число – его значение в массиве должно отличаться от нулевого:

```
         //ищем следующее простое число:
      inc(prime);
      while ((prime <= Sqrt(endNumber)) and (number[prime] = 0)) do
         inc(prime);
```

Теперь переменная *prime* содержит следующее простое число.

5. Если *prime* не превосходит корня квадратного из максимального числа *end*, то переходим к *п.3*.

```
until (prime > Sqrt(endNumber));
```

В противном случае все простые числа найдены, их значения в массиве - *ненулевые*. Перебираем весь массив и по этому признаку отыскиваем простые числа. Каждое из них мы печатаем в текстовом окне и, как и в первом примере, параллельно записываем в файл:

```
    //печатаем простые числа:
```

```
      append(f);     //добавить к файлу
      TextColor(LightGreen);
      For var j:= 2 To endNumber do
      begin
         If number[j]<>0 Then
         begin
            writeln(number[j]);
            //выводим в файл:
            writeln(f, number[j]);
         end;
      end;
      writeln;
      writeln(f, '');
      close(f);
   end;
end.
```

Для этой программы это вполне уместно сделать, потому что с помощью решета Эратосфена не следует искать большие простые числа.

Этот пример наглядно показывает нам, что для написания эффективной программы нужен хороший алгоритм. А если алгоритм имеется, то перевести его на любой язык программирования совсем несложно (Рис. 10.3).

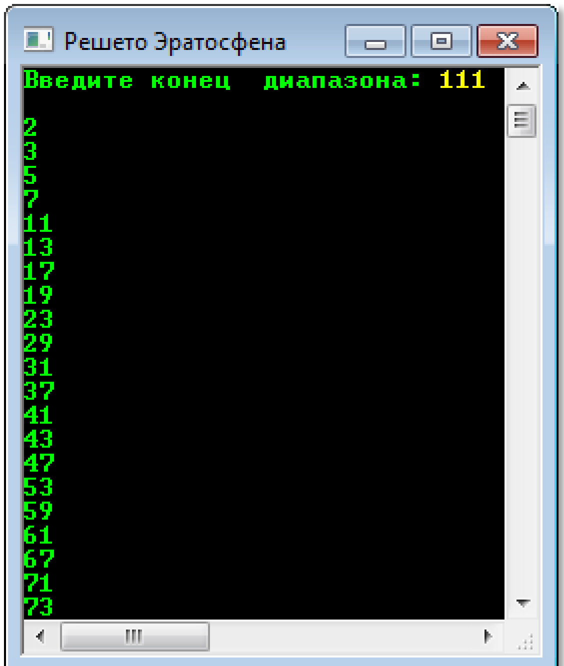


Рис. 10.3. Решето Эратосфена в действии!

Отладка CRT-приложений

Чтобы выводить информацию на экран **разноцветными** шрифтами, необходимо подключать к программе модуль *CRT*, а это значит, что мы не сможем воспользоваться преимуществами встроенного отладчика *ИСР*. В этом случае следует поступать одним из следующих способов.

1. Сначала отладить программу в стандартной консоли, а затем добавить процедуры и функции модуля *CRT*.

2. Сразу написать программу с модулем *CRT*, но закомментировать «опасные» строчки. После отладки – раскомментировать их.

3. Наиболее радикальный способ состоит в том, чтобы расставить в строчках со специфическими операторами *директивы препроцессора*.

Пишем программу, как в предыдущем пункте, но вместо комментариев используем команды компилятора. Они заключаются в фигурные скобки и начинаются со знака *$*. В самом начале программы нужно объявить переменную *DEBUG* (*отладка* – но имя может быть произвольным)

```
//РЕШЕТО ЭРАТОСФЕНА

{$define DEBUG}
```

Теперь все строчки, в которых упоминается модуль *CRT*, обрамляем парами команд:

```
{$ifndef DEBUG} uses CRT; {$endif}
```

Команда *{$ifndef DEBUG}* означает: если переменная *DEBUG* не объявлена, то остальная часть строки, до команды *{$endif}* должна компилироваться. В противном случае её следует пропустить.

Расставляем команды во всей программе:

```
//==========================================
//             ОСНОВНАЯ ПРОГРАММА
//==========================================
```

```
begin
  {$ifndef DEBUG} CRT.SetWindowTitle('Решето Эратосфена'); {$endif}
  . . .
     {$ifndef DEBUG} TextColor(LightGreen); {$endif}
     write('Введите конец  диапазона: ');
     {$ifndef DEBUG} TextColor(Yellow); {$endif}
     . . .
     {$ifndef DEBUG} TextColor(LightGreen); {$endif}
     For var j:= 2 To endNumber do

     . . .
```

Смело запускаем программу кнопкой *Выполнить* (*F9*) и отлаживаем её до полного успеха. Если вы подумали, что сейчас придётся удалять все наши дополнительные команды, то напрасно! Закомментировать нужно всего одну строку:

```
//{$define DEBUG}
```

Все «закомментированные» строчки опять работают!

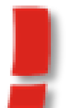
Конечно, расставлять команды препроцессора довольно утомительно, но зато они универсальны: их можно использовать с любыми отладочными операторами, а по-настоящему закомментировать нужно всего одну строку.

Исходный код программы находится в папке **Resheto**.

Вложенные циклы

Мы уже не раз встречались с ситуацией, когда внутри одного цикла располагался второй. Внутри второго цикла можно поместить третий – и так до бесконечности. Главное при организации таких вложенных циклов помнить *правило матрёшки*: **всякий внутренний цикл должен целиком находиться внутри внешнего**:

```
for ...          ← Заголовок первого цикла
begin

    for ...          ← Заголовок второго цикла
    begin

        for ...          ← Заголовок третьего цикла
        begin

        end          ← Конец третьего цикла

    end          ← Конец второго цикла

end;          ← Конец первого цикла
```



Как всегда, **выделяйте отдельные циклы отступами!**

Вложенных циклов может быть любое количество и это могут быть не только циклы *for*, но и *while, и repeat* - в любых сочетаниях.

Перед ключевым словом *end* точку с запятой ставить не обязательно, хотя и можно – тогда этот знак будет обозначать *пустой оператор*, который ничего не делает.

Олимпиадная задача

Давайте разберём *пример* решения задачи с помощью вложенных циклов.

В своё время на городской олимпиаде по математике мне пришлось решать такую задачу: *Подсчитайте, сколько раз пятёрка входит в представление чисел от 1 до 1000 в виде произведения простых чисел: 2, 3, 5, 7, 11,...* (например, 24 = 2 * 2 * 2 * 3; 25 = 5 * 5). Мы не на олимпиаде, поэтому пусть задачу решает компьютер, а мы составим для него простенькую программу:

```
//ОЛИМПИАДНАЯ ЗАДАЧА

uses CRT;

//счётчик пятёрок:
var n5:=0;
var n:=0;
var uslovie:=false;

//==========================================
//            ОСНОВНАЯ ПРОГРАММА
//==========================================
begin
  //проверяем все заданные числа:
  for var i:= 1 to 1000 do
  begin
     n := i;
     uslovie := true;
     while (uslovie) do
     begin
       //если число делится на 5,
       //увеличиваем счётчик пятёрок:
       if (n mod 5 = 0) then
       begin
          n:= n div 5;
          n5 += 1;
       end
       else
       begin
          uslovie:=false;
       end
     end
  end;

  TextColor(CRT.LightGreen);
  writeln('n5= ' + n5.ToString());
  writeln();

end.
```

Здесь мы легко найдём цикл *for*, в котором перебираются все заданные числа, и вложенный в него цикл *while*, который и подсчитывает их делители. Если число делится на 5 нацело, то мы уве-

личиваем счётчик пятёрок на 1, иначе переходим к проверке следующего числа. Олимпиадный «подвох» этой задачи заключается в том, что полученное после деления на 5 число опять может быть кратно 5, то есть его заново нужно проверить. Если это обстоятельство не учесть, то число пятёрок можно было бы подсчитать мгновенно: 1000 : 5 = 200. Таким образом, в первой тысяче чисел ровно 200 делятся на 5. Ещё 40 делятся на 25 (5*5), 8 – на 125 (5*5*5) и одно – на 625 (5*5*5*5). Зная ответ, задачу легко решить, а я на олимпиаде намаялся (Рис. 10.4)!

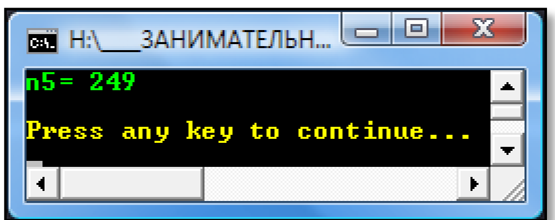


Рис. 10.4. Компьютер справился с олимпиадной задачей!

Исходный код программы находится в папке **1000**.

Добавьте в программу *Primes* переменную *fileName*, а при запуске программы пользователь пусть сам введёт имя файла для сохранения результатов работы программы. Для ввода строки используйте метод *ReadString*:

```
var s:= ReadString;
```

Исходный код программы находится в папке **Primes**.

Урок 11. Палиндромы

«Хоббит,
или Туда и обратно»

Толкиен

Стремление читать слова задом наперёд присуще даже самым отвратительным личностям, к коим мы, без тени сомнения, можем причислить Полиграфа Полиграфыча Шарикова из романа Михаила Булгакова *Собачье сердце*. Вспомните, как он начал своё восхождение к высотам низменных мыслей с причудливого слова *Абырвалг*, которое поначалу поставило в тупик доктора Борменталя и профессора Преображенского. Впрочем, они быстро догадались, что в оригинальном написании это была лишь вывеска магазина *Главрыба*, прочитанное Шариковым с конца.

Чему нас учит классика на данном примере? – Не всякое слово следует читать «по-арабски», дабы не смущать учёные умы!

Впрочем, истории известен и другой, весьма поучительный пример ретроградного чтения, опровергающий предыдущее утверждение. В порядком уже давние времена на Амуре село на мель судно под названием *Сунь Ят-сен* (китайский политик), его пытались стянуть за *корму*, но безуспешно. Это мероприятие продолжалось очень долго – до тех пор, пока один из матросов не повторил лингвистический подвиг Шарикова и не прочитал название судна с конца. Получилось *Не стянусь!* (только не придирайтесь к мелочам). Тогда попробовали зацепить трос за *нос* судна – и оно легко снялось с мели.

Кто мешает тебе выдумать
порох непромокаемый.

Козьма Прутков

Второй пример лучше первого потому, что в результате прочтения названия судна наоборот получилась вполне осмысленная фраза, обернувшаяся практической пользой. Конечно, это всего

лишь случайность, но кто запретит нам придумывать такие «двусмысленные» фразы?

Однако больше известны другие фразы – их можно без ущерба для смысла читать и слева направо, и справа налево. Их называют **палиндромами**. Пожалуй, самый известный палиндром *А роза упала на лапу Азора* придумал Афанасий Фет, но мы знаем этот палиндром только потому, что именно эту «волшебную» фразу диктовала Мальвина своему дубовому ученику Буратино. Легко проверить, что она читается точно так же и в обратную сторону. Потому и волшебная!

В литературе вы найдёте множество примеров фраз-палиндромов (или - более патриотично - *перевёртышей*), порой очень забавных и даже ненормативных. Но – придумывание таких афоризмов – настоящее искусство, которое совершенно не поддаётся алгоритмизации, поэтому мы поставим перед собой чисто техническую задачу – *отыскать слова, которые не изменяются при чтении наоборот*. Их тоже называют палиндромами, и вы наверняка знаете немало таких слов. Например, *РОТОР, ШАЛАШ, КАБАК*. Чтобы найти *все* такие слова, достаточно внимательно просмотреть словарь русского языка, но мы делегируем это пагубное занятие компьютеру.

Палиндромная программа

На этом уроке мы напишем программу **Palindrome** для поиска слов-палиндромов, но сначала давайте придумаем *алгоритм* программы.

Объявим переменные и присвоим им значения:

```
  var string1:= 'РОТОР';
   var string2:= '';
```

Велико искушение присвоить переменной *string2* значение *перевёрнутой* строки *string1*. Например, так:

```
 var len := string1.Length;
```

```
   for var i:= 0 to len-1 do
   begin
       var chr:= string1.Substring(len-i-1,1);
       string2 += chr;
   end;
```

Действительно, если слово не изменяется при чтении задом наперёд, то *string1 = string2*. Сравниваем строки и при их равенстве делаем вывод, что слово-палиндром найдено:

```
  if string1 <> string2 then
      writeln('Не палиндром')
   else
      writeln('Палиндром');
```

Но можно поступить проще и ограничиться *одной* строковой переменной. Для этого достаточно заметить, что в слове-палиндроме одинаковые буквы расположены *симметрично* относительно середины слова, то есть нужно сравнивать первую половину букв со второй (если в слове нечётное количество букв, то букву в середине слова ни с какой другой сравнивать не надо).

```
      var len:= string1.Length;
      var flg:= true;
      for var j:= 1 to len div 2 do
      begin
        var chr1:= string1[len-j+1];
        var chr2:= string1[j];
        if (chr1 <> chr2)  then
        begin
           flg:=false;
           break;
        end;
      end;
      //нашли палиндром:
      if (flg) then
```

С одним словом всё понятно, но нам нужно просмотреть *все* слова русского языка. Обычно палиндромы ищут среди *существительных*, поэтому мы также можем ими ограничиться. Мы уже пользовались файлом, в котором были бережно сохранены существительные из словаря Ожегова и Шведовой. Вы можете загрузить

файл *OSH-W97.txt* и убедиться, что в нём ровно 27407 слов, но не пересчитывать же каждый раз слова в списке! С другой стороны, нам нужно знать их число, чтобы задать размер массива. В этом случае нужно ввести константу *MAX_WORDS* с *наибольшим* предполагаемым числом слов. Объявляем *константы* и *переменные*:

```
//ПРОГРАММА ДЛЯ ПОИСКА ПАЛИНДРОМОВ

uses CRT;

const
   MAX_WORDS = 30000;
   fileNameIn='OSH-W97.txt';
   fileNameOut='palindrome.txt';

var
   //массив-список слов:
   spisok: array [1..MAX_WORDS] of string;
   //число слов в списке:
   nWords: integer;
   f: textfile;
```

И считываем данные в массив *spisok*:

```
//===========================================
//          ОСНОВНАЯ ПРОГРАММА
//===========================================
begin
   SetWindowTitle('ПРОГРАММА ДЛЯ ПОИСКА ПАЛИНДРОМОВ');
   TextColor(LightRed);
   writeln('ИЩЕМ ПАЛИНДРОМЫ');
   writeln;
   TextColor(Yellow);
   readFile;
```

Весь процесс загрузки файла вынесен в отдельную процедуру:

```
//Считываем словарь в массив
procedure readFile();
begin
   nWords:=0;
   var s: string;
   assign(f, fileNameIn);
   reset(f);
```

```
   while not eof(f) do
   begin
      readln(f, s);
      //writeln(s);
      inc(nWords);
      spisok[nWords]:= s;
   end;
   close(f);
end;
```

Раньше мы только *записывали* данные в файл, а вот чтобы *считать* их из файла, необходимы процедуры

reset(f) и
readln(f, s)

Первая *открывает* файл для чтения. Вторая считывает строку в переменную *s*.

В данном случае мы точно знаем, что в словаре 27407 слов, поэтому нам было бы достаточно цикла *for*, чтобы загрузить все слова из файла. Однако не всегда известно, сколько строк в файле, поэтому в подпрограмму введена дополнительная *проверка*:

```
while not eof(f) do
```

Если условие **not** eof(f) **не** выполняется, то конец файла ещё не достигнут, в противном случае - файл закончился. Дальше действие подпрограммы должно быть вам понятно и без дополнительных объяснений.

Процесс загрузки файла мы будем контролировать в консольном окне, печатая каждое новое слово на экране. Конечно, словарь загрузится и без нашего наблюдения, но при отладке программы совсем неплохо «присмотреть» за её работой.

В тексте программы эта строка закомментирована, поскольку каждый раз просматривать тысячи слов не обязательно:

```
//writeln(s);
```

Но зрелище любопытное, так что посмотрите обязательно!

Итак, в процедуре *readFile* мы загрузили все слова в строковый массив *spisok* и теперь можем целиком отдаться поиску палиндромов. Для этого достаточно проверить каждое слово в списке: если оно *симметрично*, то это палиндром, иначе – обычное слово.

Из этого следует, что можно было бы и не загружать слова в массив, а проверять слова на лету. Но обычно слова используются в программе по нескольку раз, поэтому наш способ более универсальный!

Сам поиск палиндромов основан на уже рассмотренном нами алгоритме. Мы последовательно загружаем слова из массива *spisok* в переменную *s* (для наглядности программы и некоторого ускорения работы программы; можно везде пользоваться переменной *spisok[i]*) и проверяем его на «палиндромность». Найденные слова-палиндромы выводим в консольное окно и записываем в файл:

```
//ИЩЕМ ПАЛИНДРОМЫ
procedure findPalindrome();
begin
   assign(f, fileNameOut);
   rewrite(f);
   var s: string;
   //ищем палиндромы в списке слов:
   for var i:=1 to nWords do
   begin
      s:= spisok[i];
      //writeln(s);
      var len:= s.Length;
      var flg:= true;
      for var j:= 1 to len div 2 do
      begin
        var chr1:= s[len-j+1];
        var chr2:= s[j];
        if (chr1 <> chr2)  then
        begin
           flg:=false;
           break;
        end;
      end;
      //нашли палиндром:
```

```
      if (flg) then
      begin
         writeln(s);
         writeln(f,s);
      end;
   end;//For i

   close(f);
end;
```

Обратите внимание на переменные *chr1* и *chr2*, в которые записываются символы очередного слова, симметричные относительно его середины. Без них также можно обойтись, но, согласитесь, так программа читается куда легче!

Нам осталось вызвать процедуру *findPalindrome* из основной программы:

```
   findPalindrome;
   writeln();
   writeln('OK');

   writeln();
end.
```

И убедиться, что мы не зря старались на этом уроке (Рис. 11.1)!

Исходный код программы находится в папке **Palindrome**.

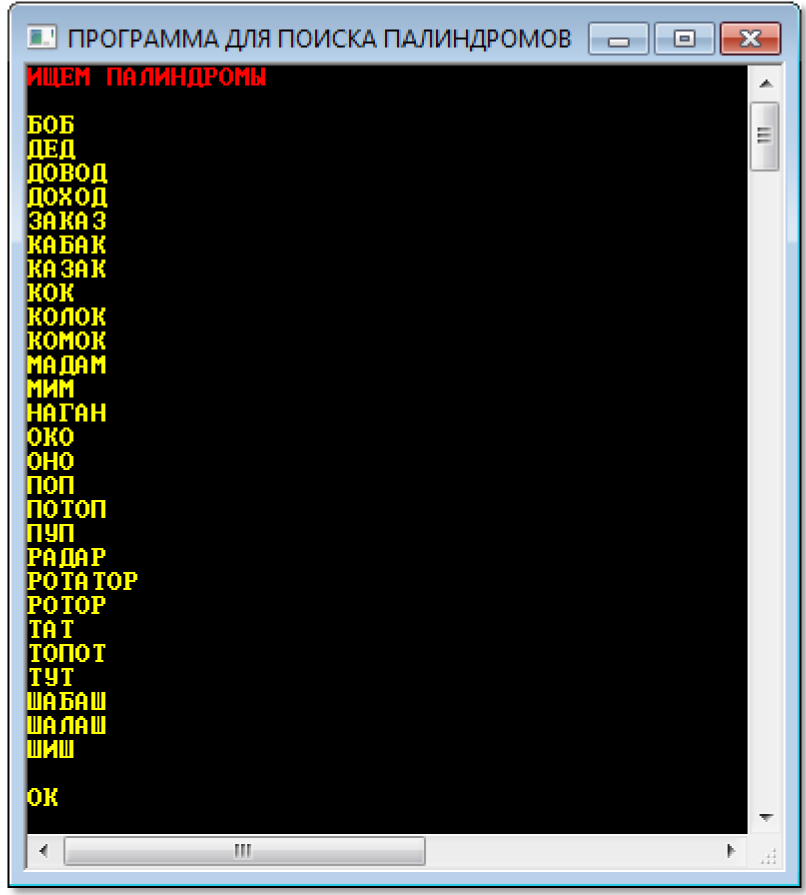


Рис. 11.1. Вот они - перевёртыши!

Исходный код программы находится в папке **Palindrome**.

Подпрограммы

Разделяй и властвуй!

Пословица римских императоров

Никогда не пишите один и тот же код дважды!

Программистская пословица

Обратите внимание: в программе *Palindrome* мы перенесли часть кода в **подпрограммы** *readFile* и *findPalindrome*. Этот приём деления программы на части применяется, когда исходный текст очень длинный и его трудно увидеть целиком на экране - прихо-

дится беспрерывно его прокручивать. Вторая причина - *повторяющиеся* куски кода. Конечно, их легко скопировать в нужные места, но при этом программа станет длиннее и в ней будет сложно ориентироваться. Ещё хуже то, что в случае внесения изменений, нам придётся исправлять программу в нескольких местах, что ведёт к потере времени и увеличивает вероятность ошибки.

Такие законченные части программы называются *подпрограммами*. В паскале подпрограммы принято делить на *процедуры* и *функции*. Их основное отличие заключается в том, что функции возвращают какое-либо значение, вычисленное в ней, а процедуры только выполняют свои операторы.

Процедуры

Процедура *записывается* так:

```
procedure  имя(список параметров);   ← Заголовок процедуры
 разделы описаний констант и переменных
 другие подпрограммы
begin
  оператор1;                           ← Тело процедуры
  оператор2;
  . . .
  операторN;
end;
```

Для процедур (по-английски, *procedure*) в *паскале* имеется шаблон кода. Напечатайте букву *p* и нажмите клавиши *Shift+ПРОБЕЛ* – в документ будет вставлен текст:

```
procedure();
begin
```

```
end;
```

Начинается она с ключевого слова *procedure*, после которого нужно указать *имя* подпрограммы, круглые скобки и двоеточие. Эта строка называется *заголовком* процедуры.

В круглых скобках через точку с запятой перечисляются *формальные параметры* процедуры с указанием их типа:

```
procedure print(n: integer; s: string);
```

Список параметров может быть *пустым*, тогда круглые скобки записывать не обязательно.

При *вызове* процедуры в скобках необходимо указать *фактические* (*действительные*) *параметры*, которые называют *аргументами*. Их число, последовательность и тип должны совпадать с формальными параметрами. Например, процедуру *print* можно вызвать так:

```
print(20, 'Привет');
```

То есть следует указать имя процедуры, а в круглых скобках – фактические параметры. Как и всякий другой оператор, вызов процедуры заканчивается точкой с запятой. Первый аргумент должен иметь *целый* тип, второй – *строковый*. Обратите внимание, что при вызове процедуры аргументы записываются через запятую, а не через точку с запятой, как в её заголовке. Это связано с тем, что в заголовке процедуры допускается параметры *одного* типа указывать через запятую:

```
procedure print(a, b, n: integer; s: string);
```

При вызове процедуры можно подставлять не только константные выражения, но и *переменные*:

```
print(one, two, tree, Stroka);
```

Их идентификаторы не обязаны совпадать с именами формальных параметров. При вызове процедуры вместо параметра *a* будет подставлено значение аргумента *one*, вместо параметра *b* – аргумент *two*, и так далее.

Поскольку число и тип параметров каждой процедуры запомнить трудно, то при наборе кода появляется подсказка с заголовком процедуры, в которой **жирным** шрифтом выделен параметр, значение которого нужно указать в круглых скобках (Рис. 11.2).

```
print(20, );
procedure print(n: integer; s: string);
```

Рис. 11.2. Подсказка и тут работает!

После заголовка процедуры могут следовать разделы объявления констант, переменных и другие подпрограммы.

Между ключевыми словами *begin* и *end* находится *тело* процедуры, которое может состоять из любых операторов.

Внутри одной процедуры можно записать другую процедуру (то есть допускаются *вложенные* процедуры). Также одна процедура может вызывать другую процедуру или даже саму себя.

Это же замечание относится и к функциям.

Любые константы и переменные, объявленные в теле процедуры, являются *локальными*. Они доступны только в самой процедуре и существуют, только пока она работает. После её завершения все локальные переменные и константы уничтожаются. Если имя локальной переменной совпадает с именем глобальной переменной, то использоваться будет *локальная* переменная. Все *параметры* процедуры – это также её локальные переменные.

В целом структура процедуры совпадает со структурой всей программы, но есть и отличия:

1. Процедура обязана иметь заголовок.

2. Процедура заканчивается ключевым словом *end* и точкой с запятой (но не точкой!).

Процедуры обычно описывают после всех разделов программы, непосредственно перед основной программой, начинающейся словом *begin*.

Функции

Функция *записывается* так:

```
function   имя( список параметров): тип;    ← Заголовок функции
 разделы описаний констант, переменных
 другие подпрограммы
begin
  оператор1;                                 ← Тело  функции
  оператор2;
  . . .
  операторN;
end;
```

Для функций (по-английски, *function*) в *паскале* имеется шаблон кода. Напечатайте буквы *fu* и нажмите клавиши *Shift+ПРОБЕЛ* – в документ будет вставлен текст:

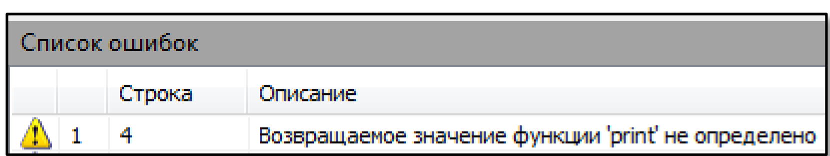
```
function(): ;
begin

end;
```

Три основных *отличия* функций от процедур:

1. Объявление функции начинается с ключевого слова *function*.

2. Каждая функция имеет *тип*, который совпадает с типом возвращаемого значения. Его указывают в конце заголовка функции после двоеточия.

3. Функция *обязана* возвращать значение, которое будет подставлено вместо вызова функции в коде программы. Если вы забудете это сделать, то результат функции может быть каким угодно, что отразится на работоспособности программы. Компилятор не считает это ошибкой, но выдаёт предупреждение (Рис. 11.3).



Список ошибок

		Строка	Описание
⚠	1	4	Возвращаемое значение функции 'print' не определено

Рис. 11.3. Функция-невозвращенец

Скорее всего, функция вернёт значение по умолчанию для своего типа. Например, для числовых типов это будет 0. Естественно, в других реализациях паскаля это может быть иначе.

Опишем простейшую функцию:

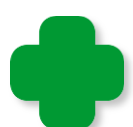
```
function print(s: string): integer;
begin
   print:=1;
end;
```

Она должна возвращать *целое* значение. Для этого в теле функции нужно присвоить *ей* значение указанного типа. Другой способ состоит в том, чтобы присвоить значение специальной переменной *Result*:

```
function print(s: string): integer;
begin
   Result:=2;
end;
```

Модульное программирование

Подпрограммы используют для того, чтобы разделить большую программу на отдельные части (*модули*), которые не зависят друг от друга, поэтому их легче отлаживать, чем программу необъятных размеров. Другое преимущество *модульного программирования* состоит в реализации важнейшего принципа: *Никогда не пишите дважды один и тот же код!* Отлаженные модули могут быть легко присоединены к любой программе и будут работать в ней без ошибок.



Примером модульного подхода может служить книга: она состоит из отдельных частей (томов), разбитых на главы. Сами главы разбиты на параграфы, абзацы, строки.

Любой механизм состоит из отдельных узлов, те, в свою очередь, - из деталей. И даже венец природы являет собой лучший образец модульного мышления Создателя нашего: человек – это совокупность отдельных органов (пищеварения, дыхания, кровообращения, слуха и других), что и позволяет лечить болезни узким специалистам (увы, не всегда успешно именно из-за своей узкой специализации).

В паскале модули могут быть выполнены в виде *процедур* и *функций*. Основная часть программы при необходимости обращается к модулям, поэтому ядро программы можно сделать очень *небольшим*, что облегчит понимание логики работы программы, поскольку основной код получится очень коротким.

Так как программы на таких языках программирования, как бейсик или паскаль, строятся из отдельных частей (процедур и функций), как дома - из кирпичей или блоков, то они называются *процедурными*.

В паскале можно создавать целые библиотеки подпрограмм, которые также называются *модулями* (*unit*), что вызывает некоторую неразбериху в терминологии. В *отдельные* модули обычно помещают процедуры общего назначения, которые могут потре-

боваться во многих программах. Например, процедура *Sound* может воспроизводить звуковые файлы, а процедура *Input* – вводить данные. У вас по мере написания программ также накопится множество таких полезных подпрограмм. Мы уже много раз использовали модуль *CRT*, который предназначен для работы с консолью. Вы легко можете посмотреть, как устроен этот модуль (Рис. 11.4).

```
uses CRT;
```

unit CRT
Описание:
Модуль для работы с консолью

Рис. 11.4. Подключение модуля к программе

Для этого вызовите контекстное меню, нажав правую кнопку мышки, и выполните команду *Перейти к определению* (Рис. 11.5).

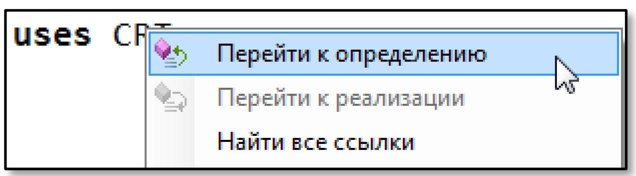


Рис. 11.5. Переходим в модуль

В редакторе кода появится новая закладка с исходным кодом модуля *CRT*:

```
/// <summary>
/// Модуль для работы с консолью
/// </summary>
unit CRT;

// Версия 0.3
// Copyright (c) 2006-2007 DarkStar, SSM (parts)

{$apptype console}
{$gendoc true}

interface

uses
  System;
```

```
const
  Black        = 0;
  Blue         = 1;
. . .
```

Здесь вы найдёте константы стандартных цветов консоли, процедуры и функции. Утолив любопытство, осторожно закройте вкладку – ничего изменять в этом модуле нельзя!

Фракционируем словарь

Довольно часто в программах нужен список, в котором слова располагаются по *длине* (по числу букв): сначала двухбуквенные, затем трёхбуквенные и так далее. У нас пока имеется алфавитный список слов, который хранится в файле *OSH-W97.txt*. Мы можем непосредственно из него загрузить слова нужной нам длины, но тогда придётся просматривать весь файл, что не очень хорошо. Будет правильнее, если мы на его основе создадим такой список, в котором слова отсортированы не только по алфавиту, но и по длине. Для временного хранения слов нам потребуется *строковый массив spisok*:

```
//ПРОГРАММА ДЛЯ СОЗДАНИЯ ФРАКЦИОННОГО СЛОВАРЯ

uses CRT;

const
   MAX_WORDS = 30000;
   fileNameIn='OSH-W97.txt';
   fileNameOut='OSH-W97frc.txt';

var
   // массив-список слов:
   spisok: array [1..MAX_WORDS] of string;
   //число слов в списке:
   nWords: integer;
   //минимальная длина слов:
   minLen: integer;
   //максимальная длина слов:
   maxLen: integer;
```

```
   f: textfile;
```

Считывание алфавитного словаря и сохранение на диске фракционированного словаря мы перенесём в *процедуры*, тогда код основной программы будет коротким и ясным:

```
//===========================================
//          ОСНОВНАЯ ПРОГРАММА
//===========================================
begin
   SetWindowTitle('ФРАКЦИОНИРУЕМ СЛОВАРЬ');
   TextColor(Yellow);

   readFile;
   writeFrac;

   writeln();
   writeln('OK');
   writeln();
end.
```

Прежде всего, нам нужно загрузить список слов в массив *spisok*:

```
//Считываем словарь в массив
procedure readFile();
begin
   nWords:=0;
   minLen:=1;
   maxLen:=1000;
   var s: string;
   assign(f, fileNameIn);
   reset(f);
   while not eof(f) do
   begin
      readln(f, s);
      writeln(s);
      inc(nWords);
      spisok[nWords]:= s;
      var len:= s.Length;
      if (minLen > len) then
         minLen:= len;
      if (maxLen < len) then
         maxLen:= len;
```

```
   end;
   close(f);
end;
```

Сначала слов в массиве нет:

```
nWords:=0;
```

Затем мы загружаем слова из файла в массив и одновременно определяем длину самого *длинного* и самого *короткого* слова. Их начальные значения мы задаём так, чтобы они заведомо были не короче и не длиннее слов в списке:

```
   minLen:=1;
   maxLen:=1000;
```

После завершения процедуры считывания файла мы имеем:

- в массиве *spisok* – все слова в алфавитном порядке;
- в переменных *minLen* и *maxLen* – длину самого короткого и самого длинного слова в массиве.

Нам осталось только записать *фракционированный словарь* в новый файл *fileNameOut*:

```
//Записываем фракционный словарь в файл
procedure writeFrac();
begin
   assign(f, fileNameOut);
   rewrite(f);

   For var i:= minLen To maxLen do
     For var j:= 1 To  nWords do
       if (spisok[j].Length = i) then
          writeln(f, spisok[j]);

   close(f);
end;
```

Во внешнем цикле *for* мы просматриваем весь массив несколько раз, выбирая из него слова равной длины. Сначала самые короткие, потом на одну букву длиннее – и так до самых длинных слов. Найденные слова мы записываем в файл. Метод, конечно, расточительный, но для такого небольшого списка, как наш, вполне допустимый, поскольку операция записи слов на диск заведомо медленнее, чем перебор слов в памяти компьютера.

Всё! Запускаем программу – и через несколько секунд получаем на диске долгожданный фракционированный словарь *OSH-W97frc.txt*. Смотрим в его начало – слова, действительно, расположились строго по числу букв:

```
АД
АЗ
АР
АС
ГО
ЁЖ
ИЛ
ОМ
ОР
ПА
РЭ
СУ
УЖ
УМ
УС
ЩИ
ЮГ
ЮЗ
ЮР
ЮС
ЯД
ЯК
ЯЛ
ЯМ
ЯР
АЗУ
АКР
. . .
```

А в самом конце списка (что символично!) затаилось самое длинное слово – *ЧЕЛОВЕКОНЕНАВИСТНИЧЕСТВО*, в котором 24 буквы.

PascalABC.NET может считывать текстовые файлы в форматах *Юникод и ANSI*, а сохраняет в последнем формате, поэтому фракционный словарь оказался у нас вдвое меньше по размеру, хотя состоит из тех же слов, что и исходный.

Исходный код программы находится в папке **Palindrome**.

Суперпростые числа

До сих пор нам не довелось «сочинить» ни одной настоящей *функции*. Давайте исправим это упущение, а заодно напишем программу **Superprimes** для поиска интереснейших *суперпростых* чисел!

Вообще говоря, под *суперпростыми* понимают разные числа, но мы исследуем такие числа, которые и целиком – простые, и все их «половинные» части – также простые. Поясню на примере.

Возьмём простое число *1373*. Разобьём его всеми способами на две части:

1-373
13-73
137-3

Оказывается, что все «частичные» числа – *1, 3, 13, 73, 137, 373* - также простые. А раз это так, то исходное число *1373* мы по полному праву можем именовать *суперпростым*!

Наша задача - отыскать все суперпростые числа заданной длины *len*:

```
//ПРОГРАММА ДЛЯ НАХОЖДЕНИЯ
//СУПЕРПРОСТЫХ ЧИСЕЛ

uses CRT;

//variables
var
   len := 0;
```

Так как однозначные числа нельзя поделить на части, то они формально суперпростые, но очень неинтересные, поэтому мы разрешим пользователю задавать длину чисел в разумном диапазоне *2..7*:

```
//=========================================
//          ОСНОВНАЯ ПРОГРАММА
//=========================================
begin
   SetWindowTitle('Суперпростые числа');

   while(true ) do
   begin
      repeat
         TextColor(Yellow);
         write('Введите число 2..7 >  ');
         TextColor(CRT.LightGreen);
         len := readInteger;

         //Если задан нуль,
         //то работу с программой заканчиваем:
         if (len = 0) then exit;
      until ((len >= 2) and (len <= 7));
```

По заданной длине чисел мы находим максимальное число *max*, которое на 1 больше максимального из заданных:

```
      writeln;
      //Ищем суперпростые числа
      //заданной длины:
      var max: integer := 1;
      var num: integer;
      var i:   integer;
```

```
      for var l := 1 to len do
         max *= 10;
```

Можно вычислить значение *max* иначе:

```
max:= round(Power(10,len));
```

Например, для *len=2* мы получим 1 с *двумя* нулями – *100*. *Наибольшее двузначное* число на 1 меньше наименьшего трёхзначного, то есть *99*.

Теперь в цикле *for* мы перебираем все числа заданной длины. Например, для двузначных чисел переменная цикла *num* изменяется от 100 : 10 = *10* до *99*:

```
      for num := max div 10 to max - 1 do
      begin
         if (not prime(num)) then
            continue;
```

Поскольку исходное, длинное число должно быть простым, то и его необходимо проверить, и если оно не соответствует предъявленным требованием, цикл переходит к проверке *следующего* числа.

Вот тут нам необходима *функция* для проверки заданного числа на простоту:

```
function prime(number: integer): boolean;
begin

   prime := true;
    //Проверяем, делится ли введенное число на числа
    //2..корень квадратный из числа;
   for var i := 2 To Floor(Sqrt(number)) do
   begin
      if (number mod i = 0) Then
      begin
         prime := false;
         exit;
      end;
```

```
   end; //for i
end;
```

Она возвращает значение *true*, если число *number* простое, и *false* – если составное.

Поскольку алгоритм я переписал из наших предыдущих творений, то обсуждать его было бы лишним. В функции же вам следует обратить внимание только на то, как задаётся возвращаемое значение.

Если длинное число *num* оказалось простым, то нам необходимо проверить и все его составные части. Для этого мы делим его на вспомогательную переменную *i*. При начальном значении, равном 10, мы разобьём число *num* на такие части:

```
1373 div 10 = 137
1373 mod 10 = 3
```

В следующей итерации значение переменной *i* станет равным 10 x 10 = 100, и мы получим такие части исходного числа:

```
1373 div 100 = 13
1373 mod 100 = 73
```

Вы легко продолжите этот процесс расщепления до самого конца. Мы же должны убедиться, что каждая из частей также является простым числом. Для этого мы, естественно, опять вызываем функцию *prime*. Если проверяемое число окажется составным (функция вернёт *false*), то следующие составные части проверять бесполезно, поэтому с помощью оператора *break* мы досрочно выходим из цикла *while*:

```
        i := 10;
        while (i < max) do
        begin
           if (not prime(num div i)) then
              break;
           if (not prime(num mod i)) then
              break;
```

```
            i *= 10;
         end;
         if i = max then
            //печатаем суперпростое число:
            writeln(num.ToString());
      end;
```

На выходе (досрочном или «нормальном») мы сравниваем конечное значение переменной *i* с максимальным - *max*. Легко понять, что если они совпадут, то все составные части исходного числа оказались простыми, и нам ничего не остаётся, как только вынести его на нашу доску почёта в виде консольного окна (Рис. 11.6):

```
//печатаем суперпростое число:

writeln(num.ToString());
      end;

      writeln;
      TextColor(Yellow);
   end;
end.
```

```
Суперпростые числа
Введите число 2..7 >  2

11
13
17
23
31
37
53
71
73

Введите число 2..7 >  3

113
131
137
173
197
311
313
317
373
797

Введите число 2..7 >  4

1373
1997
3137
3797
7331

Введите число 2..7 >  5

73331

Введите число 2..7 >  6

739397

Введите число 2..7 >
```

Рис. 11.6. А что же дальше?

Исходный код программы находится в папке **Superprimes**.

Операторы *continue, break, exit*

Довольно часто в программах возникают ситуации, когда нужно прервать очередную итерацию или вообще *досрочно* выйти из цикла. Например, при поиске суперпростых чисел длинное число *num* может оказаться *не простым* (а в большинстве случаев так и будет!), и тогда проверка его составных частей станет ненужной. Мы должны сразу же перейти к проверке следующего числа, то есть *прервать* выполнение очередной итерации. Для этого мы записали оператор **continue**, который как раз и передаёт управление в заголовок цикла:

```
    for num := max div 10 to max - 1 do
      begin
         if (not prime(num)) then
            continue;
```

По условию задачи, не только целое число, но и *все* его составные части должны быть простыми числами. Это значит, что первое же не простое число должно послужить сигналом к окончанию проверки очередного числа *num*. Поскольку эти проверки мы проводим в другом цикле – *while*, то оператор *continue* закончит выполнение не всего цикла, а только текущей итерации. Нам же необходимо передать управление оператору, следующему за концом цикла *while* (выделен красным цветом):

```
        i := 10;
        while (i < max) do
        begin
           if (not prime(num div i)) then
              break;
           if (not prime(num mod i)) then
              break;
           i *= 10;
        end;
        if i = max then
```

В этом случае необходимо в нужном месте тела цикла записать оператор *break*.

Операторы *continue* и *break* можно использовать только в теле операторов цикла *for, while, repeat*.

Важно отметить, что оператор *break* прерывает только тот цикл, в котором он находится. Это значит, что для полного выхода из вложенных циклов потребуется соответствующее число операторов *break*.

Мы уже неоднократно пользовались в своих проектах и более сильнодействующим средством – оператором *exit*. Он передаёт управление в то место кода, откуда была вызвана процедура или функция. При этом, само собой, досрочно завершается любой цикл. Например, при вводе нуля мы заканчивали не только выполнение цикла *repeat*, но и выходили из «процедуры», которой в этом случае была основная программа, то есть мы *закрывали* приложение:

```
repeat
   TextColor(Yellow);
   write('Введите число 2..7 >  ');
   TextColor(CRT.LightGreen);
   len := readInteger;

   //Если задан нуль,
   //то работу с программой заканчиваем:
   if (len = 0) then exit;
until ((len >= 2) and (len <= 7));
```

1. Совершенно очевидно, что суперпростые числа могут состоять только из цифр 1, 3, 5 и 7. Попробуйте сократить проверки, используя это наблюдение.

2. Легко проверить, что *семизначных* суперпростых чисел нет. Напишите программу, которая могла бы проверять на сверхпростоту *одно* заданное число, но с б**о**льшим число цифр.

Урок 12. Занимательная комбинаторика

Комбинаторика - это раздел математики, который изучает *множества* (совокупности, наборы) каких-либо элементов. Первая книга по комбинаторике вышла в 1666 году под названием *Рассуждения о комбинаторном искусстве*. Её написал известный немецкий математик Готфрид Вильгельм фон Лейбниц, который и придумал название *комбинаторика* этому разделу математики.

Как в жизни, так и в программировании очень часто встречаются *комбинаторные задачи*. Например, сколько различных слов можно составить из букв русского алфавита, сколько существует различных комбинаций при игре в кости двумя или тремя кубиками, сколько разных нарядов можно составить из трёх юбок и четырёх блузок и так далее.

Все комбинаторные задачи решаются с помощью комбинаторных конфигураций: *размещений, перестановок, сочетаний, композиций и разбиений*.

На этом уроке мы познакомимся с *перестановками* элементов. Возьмём множество, состоящее из трёх разных элементов, например, русских букв - *{К, О, Т}.* Всякое слово, составленное из этих букв, и называется перестановкой элементов множества.

Поскольку в множестве элементы *не упорядочены*, то мы выпишем их сначала в произвольном порядке, например так:

1. КОТ

Вот мы и получили первую перестановку, а значит, и первое слово – *КОТ*. Оно «случайно» совпало с настоящим русским словом. Чтобы найти вторую перестановку, поменяем местами вторую и третью буквы:

2. КТО

Тоже получилось неплохо, ведь *кто* - это русское местоимение. Давайте поменяем теперь первую и вторую буквы:

3. ТКО

Такого слова нет (в старой речи была частица *–тко*, имеющая тот же смысл, что и современная -ка: *бери-тко, читай-тко*), а вот четвёртая перестановка снова удачная. Чтобы её получить, поменяем местами вторую и третью буквы:

4. ТОК

Снова меняем первую и вторую буквы и получаем пятую перестановку:

5. ОТК

Не ахти какое слово, но *Отдел технического контроля* тоже сгодится.

И последнюю перестановку мы получаем, переставив две последние буквы:

6. ОКТ

ОКТ – тоже сокращение от *Оптическая когерентная томография*, так что все наши перестановки из трёх букв *К, О, Т* оказались не совсем бессмысленными. Конечно, с другими буквами результат был бы другим.

Но почему мы можем утверждать, что нашли *все* перестановки множества из трёх элементов? – Давайте рассуждать логически. На первом месте в слове может стоять любая из трёх букв. На второе место можно поставить любую из двух оставшихся, а для последнего места останется только одна буква. Таким образом, всего можно составить 3 x 2 x 1 = 6 разных слов. Но мы ровно столько и составили, значит, других слов из этих букв нет (естественно, мы используем каждую букву только *один* раз!).

Если взять множество из четырёх разных элементов, то, рассуждая аналогично, мы придём к выводу, что из них можно составить 4 x 3 x 2 x 1 = 24 разных слова. Этот ряд легко продолжить сколь угодно далеко, а для произведения чисел от единицы до заданного (обозначим его *n*), придумано слово *факториал*. Обозначается факториал числа так:

n! (читаем: *эн-факториал*).

А чтобы его вычислить, нужно перемножить все числа от единицы до этого числа:

n! = 1 x 2 x . . . x (n-1) x n

Формула очень простая, но нетрудно догадаться, что для больших значений *n* факториал будет выражаться огромным числом.

Перемножение последовательных чисел можно доверить циклу *for*, поэтому мы с легкостью вычислим факториал любого числа, не превышающего 20 (на этом числе арифметические возможности *паскаля* заканчиваются).

Факториал

Вычислять факториалы с помощью *паскаля* – сплошное удовольствие!

Начинаем с *переменных*:

```
//ПРОГРАММА ДЛЯ НАХОЖДЕНИЯ
//ФАКТОРИАЛА ЗАДАННОГО ЧИСЛА

uses CRT;

//variables
var number:=0;
    fact: uint64;
```

Для факториала мы выбрали тип *uint64*, чтобы вычислять самые большие (из возможных) факториалы.

Проверяем действия пользователя:

```
//=============================================
//          ОСНОВНАЯ ПРОГРАММА
//=============================================
begin
   SetWindowTitle('Факториал');

   while(true) do
   begin
      repeat
         TextColor(Yellow);
         write('Введите число 0..20 >  ');
         TextColor(CRT.LightGreen);
         number:= readInteger;

         //Если задано отрицательное число,
         //то работу с программой заканчиваем:
         if (number < 0) then exit;
      until ((number >= 0) and (number <= 20));
```

Если заставить программу вычислять очень большой факториал, то это вызовет ошибку, что охарактеризует нас как программистов не с лучшей стороны. Поэтому придётся несколько ограничить свободу пользователя в выборе чисел.

Получив от пользователя число *number*, мы можем приступить к вычислению факториала:

```
    writeln;
      //Вычисляем и выводим в консольное окно
      //факториал заданного числа number:

      fact :=  number;
      for var i:= number -1 downto 2 do begin
          fact *= i;
      end;
      If number=0 Then
         fact := 1;

      //печатаем факториал:
      writeln(number.ToString() + '! = ' + fact.ToString());
      writeln;
```

```
      TextColor(Yellow);
   end;
end.
```

Первое, на что следует обратить внимание: факториал нуля равен единице, это особый случай и его нужно учесть отдельно!

Второе: факториал вычисляется «задом наперед», то есть не как «прямое» произведение 2 * 3 * ... *number*, а как «обратное»: *number* * (*number*-1) * ... 2. Как известно, от перемены мест сомножителей произведение не изменяется, поэтому результат получится верным.

Умножать на единицу, конечно, смысла нет.

Если для нас вычисление факториала, связанное с перемножением огромных чисел, лишено всякого удовольствия, то компьютер делает это играючи (Рис. 12.1)!

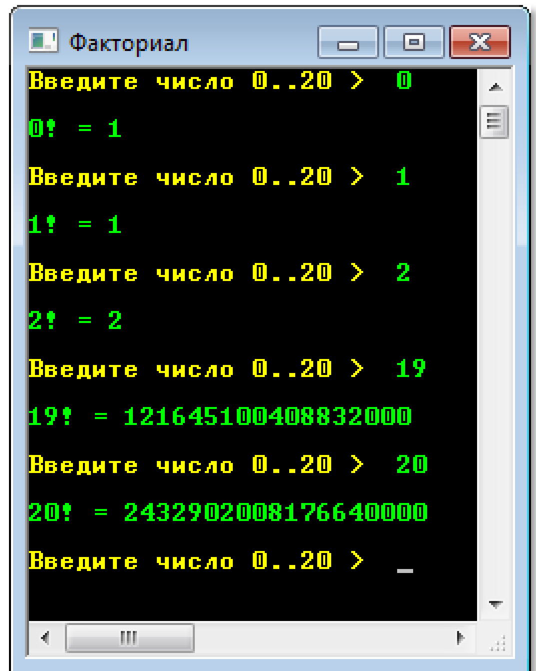


Рис. 12.1. Маловато будет!

А что же делать, если и вправду маловато будет? – Необходимо обратиться за помощью к структуре *BigInteger*, которая находит-

ся в пространстве имён *System.Numerics*. Но, прежде всего, мы должны сообщить компилятору имя сборки *System.Numerics.dll* с этим пространством имён и добавить его в раздел *uses*:

```
//ПРОГРАММА ДЛЯ НАХОЖДЕНИЯ
//ФАКТОРИАЛА ЗАДАННОГО ЧИСЛА
{$reference 'System.Numerics.dll'}

uses CRT, System.Numerics;
```

Тип переменной *fact* нужно изменить:

```
//variables
var number:=0;
    fact: BigInteger;
```

В основной части программы также нужно исправить несколько строк:

```
begin
   SetWindowTitle('Большие факториалы');

   while(true) do
   begin
      repeat
         TextColor(Yellow);
         write('Введите число 0..2000 >  ');
         . . .
      until ((number >= 0) and (number <= 2000));
```

Как вы видите, мы хотим вычислять факториалы всех чисел, до 2000 включительно (вы можете ещё увеличить это число).

Перед вычислением факториала мы инициализируем новый экземпляр структуры *BigInteger* заданным числом *number*, а затем проводим обычное перемножение чисел:

```
      fact := new BigInteger(number);
      for var i:= number -1 downto 2 do begin
          fact := fact*i;
      end;
```

Зато результат мы получили ошеломляющий (Рис. 12.2)!

```
Большие факториалы
Введите число 0..2000 >  1000

1000! = 402387260077093773543702433923003985719374864210714632543799910429938512
39862902059204420848696940480047998861019719605863166687299480855890132382966994
45909974245040870737599188236277271887325197795059509952761208749754624970436014
18278094646496291056393887437886487337119181045825783647849977012476632889835955
73543251318532395846307555740911426241747434934755342864657661166779739666882029
12073791438537195882498081268678383745597317461360853795345242215865932019280908
78297308431392844403281231558611036976801357304216168747609675871348312025478589
32076716913244842623613141250878020800026168315102734182797770478463586817016436
50241536913982812648102130927612448963599287051149649754199093422215668325720808
21333186116811553615836546984046708975602900950537616475847728421889679646244945
16076535340819890138544248798495995331910172335555660213945039973628075013783761
53071277619268490343526252000158885351473316117021039681759215109077880193931781
14194545257223865541461062892187960223838971476088506276862967146674697562911234
08243920816015378088989396451826324367161676217916890977991190375403127462228998
80051954444142820121873617459926429565817466283029555702990243241531816172104658
32036786906117260158783520751516284225540265170483304226143974286933061690897968
48259012545832716822645806652676995865268227280707578139185817888965220816434834
48259932660433676601769996128318607883861502794659551311565520360939881806121385
58600301435694527224206344631797460594682573103790084024432438465657245014402821
88525247093519062092902313649327349756551395872055965422874977401141334696271542
28458623773875382304838656889764619273838149001407673104466402598994902222217659
04339901886018566526485061799702356193897017860040811889729918311021171229845901
64192106888438712185564612496079872290851929681937238864261483965738229112312502
41866493531439701374285319266498753372189406942814341185201580141233448280150513
99694290153483077644569099073152433278288269864602789864321139083506217095002597
38986355427719674282224875758676575234422020757363056949882508796892816275384886
33969099598262809561214509948717012445164612603790293091208890869420285106401821
54399457156805941872748998094254742173582401063677404595741785160829230135358081
84009699637252423056085590370062427124341690900415369010593398383577793941097002
77534720000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000
```

Рис. 12.2. Вот это факториальчик!

И на всё про всё программа затратила доли секунды!

Исходный код программы находится в папке **Factorial**.

Генерируем перестановки

Мы научились ловко узнавать общее число перестановок. Ну пусть мы знаем, что из трёх разных букв можно составить 3! разных слов, но ведь программа *Factorial* не даёт ответа на очевидный вопрос: а *как* составить эти перестановки? С множеством из

трёх элементов мы легко справились, а если взять больше – четыре, пять, а то и восемь?

Оказывается, мы не первые, кто задался этим вопросом. Ещё в семнадцатом веке английские звонари научились выбивать на нескольких разных колоколах «мелодии», состоящие из всех перестановок этих колоколов. Например, для трёх колоколов нужно было сыграть такую мелодию:

Первый колокол – второй – третий.
Второй колокол – первый – третий.
Дальше вы и сами продолжите эту музыку.

Колоколов, конечно, было больше, а последовательность колокольных ударов нужно было держать в голове. Например, в *Книге рекордов Гиннесса* рассказывается о том, что в 1963 году за 17 с лишним часов удалось выбить на восьми колоколах все 8! = 40320 перестановок. В семнадцатом веке, конечно, эти музыкально-комбинаторные экзерсисы были короче, но, тем не менее, запомнить многие сотни перестановок было совсем непросто, поэтому звонари придумывали свои способы для «генерирования» всех колокольных перестановок. Один из таких способов мы и положим в основу компьютерной программы, которая быстро и правильно составит все перестановки элементов заданного множества.

Сначала программа выписывает первую перестановку. Например, для множества из четырёх элементов это будет

1 2 3 4

Затем она циклически меняет местами первый и второй, второй и третий, третий и четвёртый элементы. После этого снова - первый-второй, второй-третий, третий-четвёртый элементы, и так далее, пока не будут сгенерированы все перестановки (Рис. 12.3).

Начнём новый проект с загрузки исходного кода программы *Factorial*, сохраним его в папке **Permutation** и начнём добавлять новый код.

Нам вполне хватит одной *константы* и трёх глобальных *переменных*:

```
//ПРОГРАММА ДЛЯ ГЕНЕРИРОВАНИЯ
//ВСЕХ ПЕРЕСТАНОВОК ЭЛЕМЕНТОВ МНОЖЕСТВА ЧИСЕЛ 1..nElem

uses
   CRT;

const
   MAX_ELEM = 8;

var
   // число элементов в множестве:
   nElem := 0;
   // номер перестановки:
   nPerm := 0;
   // массив, содержащий очередную перестановку:
   p: array [1..MAX_ELEM] of integer;
```

Начало программы мало изменилось по сравнению с вычислением факториала. Главное – ограничить число элементов, иначе список перестановок будет очень длинным.

```
//===========================================
//          ОСНОВНАЯ ПРОГРАММА
//===========================================
begin
   SetWindowTitle('Перестановки');

   while(true ) do
   begin
      repeat
         TextColor(Yellow);
         write('Введите число 1..8 >  ');
         TextColor(CRT.LightGreen);
         nElem := readInteger;
```

```
        //Если задан нуль,
        //то работу с программой заканчиваем:
        if (nElem = 0) then exit;
    until ((nElem >= 1) and (nElem <= 8));
```

Первая перестановка состоит из последовательного ряда чисел *1 .. nElem*:

```
    //генерируем и выводим в консольное окно
    //перестановки:
    var c, pr: array [1..MAX_ELEM] of integer;
    var i: integer;
    for i := 1 to nElem do
    begin
       p[i] := i;
       c[i] := 1;
       pr[i] := 1;
    end;
    nPerm := 0;
    c[nElem] := 0;
    //печатаем первую перестановку:
    writePerm();
```

Процедура *печати* перестановок очень простая:

```
 // ПРОЦЕДУРА ПЕЧАТИ ОЧЕРЕДНОЙ ПЕРЕСТАНОВКИ
 // ЭЛЕМЕНТОВ МНОЖЕСТВА
 procedure writePerm;
 begin
    nPerm := nPerm + 1;
    TextColor(LightGreen);
    Write(nPerm.ToString() + '> ');
    for var i := 1 to nElem do
       Write(p[i].ToString() + ' ');

    writeln;
 end;
```

Если вас интересуют только сами перестановки, без номера, закомментируйте строку

```
Write(nPerm.ToString() + '> ');
```

Все остальные перестановки мы получаем из первой, меняя местами *соседние* элементы. Тут важно определить номера элементов для обмена. Как это работает в программе, проследите сами:

```
      i := 1;
      var k: integer;
      while (i < nElem) do
      begin
         i := 1;
         var x := 0;
         while (c[i] = nElem - i + 1) do
         begin
            pr[i] := -pr[i];
            c[i] := 1;
            if (pr[i] = 1) then
               x := x + 1;
            i := i + 1;
         end; //While
         if (i < nElem) Then
         begin
            if (pr[i] = 1) then
               k := c[i] + x
            Else
               k := nElem - i + 1 - c[i] + x;
            var tmp := p[k];
            p[k] := p[k + 1];
            p[k + 1] := tmp;
            // печатаем очередную перестановку:
            writePerm();
            c[i] := c[i] + 1;
         end;//If
      end;//While

      writeln;
      TextColor(Yellow);
   end;
end.
```

Вы можете сгенерировать перестановки множеств, имеющих до 8 элементов (Рис. 12.3 и 12.4).

```
Перестановки
Введите число 1..8 > 1
1> 1

Введите число 1..8 > 2
1> 1 2
2> 2 1

Введите число 1..8 > 3
1> 1 2 3
2> 2 1 3
3> 2 3 1
4> 3 2 1
5> 3 1 2
6> 1 3 2

Введите число 1..8 > 4
1> 1 2 3 4
2> 2 1 3 4
3> 2 3 1 4
4> 2 3 4 1
5> 3 2 4 1
6> 3 2 1 4
```

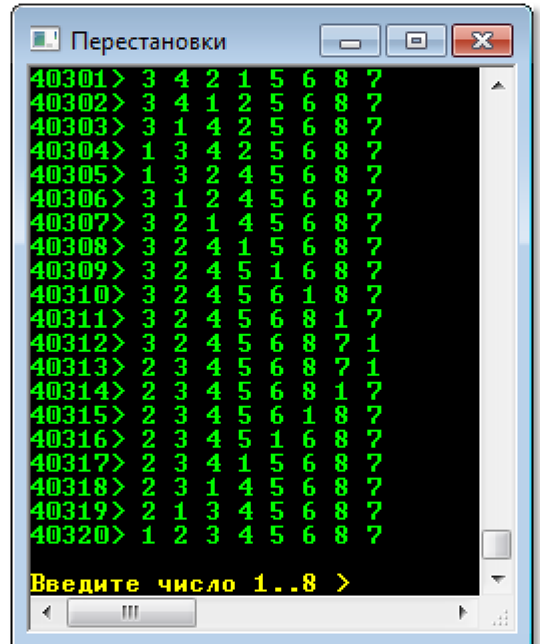


Рис. 12.3. Программа в работе!

Рис. 12.4. Все перестановки множества 1..8 – звонить - не перезвонить!

Исходный код программы находится в папке **Permutation**.

Как нам собрать Дрим тим?

Решим ещё одну жизненно важную задачу. Представим себе, что на Чемпионат мира по футболу нужно послать сборную вашего класса, в котором 16 мальчиков. Как известно, футбольная команда состоит из 11 человек, поэтому нам нужно из 16 человек выбрать только 11.

Остальные мальчики класса, конечно, тоже поедут на чемпионат, но как запасные игроки, а девочки составят команду поддержки. Так что никто не будет обделён вниманием!

Для комбинаторики общее число мальчиков в классе - это число элементов в *множестве*, а число игроков в команде – число элементов *подмножестве*. Таким образом, нам *нужно найти все подмножества заданного множества*. Каждое подмножество ка-

кого-либо множества иначе называют набором из заданного числа элементов, или **сочетанием**. В сочетании порядок элементов не учитывается, поэтому мы отберём только 11 игроков в команду, а как между ними поделить номера на футболках (а, значит, и их амплуа на поле), это уже задача тренера.

За основу новой программы мы возьмём исходный код проекта для генерирования перестановок элементов множества и сохраним его в папке **Subset**.

Объявим новые *переменные*:

```
//ПРОГРАММА ДЛЯ ГЕНЕРИРОВАНИЯ
//ВСЕХ ПОДМНОЖЕСТВ k МНОЖЕСТВА ЧИСЕЛ n=1..nElem

uses CRT;

const
   MAX_ELEM = 30;

var
   // число элементов в множестве:
   nElem := 0;
   // число элементов в подмножестве:
   k := 0;
   // номер подмножества:
   nSubset := 0;
   // массив, содержащий очередное подмножество:
   a: array [1..MAX_ELEM] of integer;
```

В этой программе пользователь должен ввести *два* числа, которые следует *проверить*, иначе расчеты окажутся неверными. Важно учесть, что в подмножестве не может быть элементов больше, чем в множестве.

```
//===========================================
//          ОСНОВНАЯ ПРОГРАММА
//===========================================
begin
   SetWindowTitle('Подмножества');

   while(true ) do
```

```
begin
   repeat
      TextColor(Yellow);
      write('Введите число элементов в множестве 1..30 >    ');
      TextColor(CRT.LightGreen);
      nElem := readInteger;

      //Если задан нуль,
      //то работу с программой заканчиваем:
      if (nElem = 0) then exit;
   until ((nElem >= 1) and (nElem <= 30));

   repeat
      TextColor(Yellow);
      write('Введите число элементов в подмножестве 1..' +
            nElem.ToString() + ' >  ');
      TextColor(CRT.LightGreen);
      k := readInteger;
   until ((k >= 1) and (k <= nElem));
```

Если данные пользователя успешно прошли проверку, то *генерируем* все подмножества:

```
    For var i:= 1 to k do
          a[i]:= i;

      nSubset:=0;
      var p:= k;

      While (p >= 1) do
      begin
         //печатаем очередное подмножество:
         writeSubset();
         if (k = nElem) then
         break;

         If (a[k]= nElem) then
            p:= p - 1
         else
            p:= k;

         If (p >= 1) Then
            For var i:= k downto p do
```

```
                a[i]:= a[p] + i - p + 1;
      end;//While

      writeln;
      TextColor(Yellow);
   end;
end.
```

Затем пользователь может ввести другие данные и продолжить работу с программой.

Процедура *печати* очередного подмножества почти не отличается от процедуры печати перестановок:

```
  //ПРОЦЕДУРА ПЕЧАТИ ОЧЕРЕДНОГО ПОДМНОЖЕСТВА
   //ЭЛЕМЕНТОВ МНОЖЕСТВА
   procedure writeSubset;
   begin
      nSubset := nSubset + 1;
      TextColor(LightGreen);
      Write(nSubset.ToString() + '> ');
      for var i := 1 to k do
         Write(a[i].ToString() + ' ');

      writeln;
   end;
```

Запустив программу и введя наши данные: 16 элементов в множестве и 11 – в подмножестве, мы получим огромный список из 4368 разных сборных класса (Рис. 12.5)! Комбинаторную задачу мы решили, а вот какая из этих сборных действительно станет командой мечты, тут комбинаторика бессильна!

Иногда полезно заранее узнать, сколько же получится сочетаний при тех или иных исходных данных - не печатать же весь список подмножеств, если нам интересно узнать только их число! А на этот случай в комбинаторике припасена несложная формула:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

В нашей программе *n = nElem*.

Как видите, в комбинаторике без факториала не обойтись!

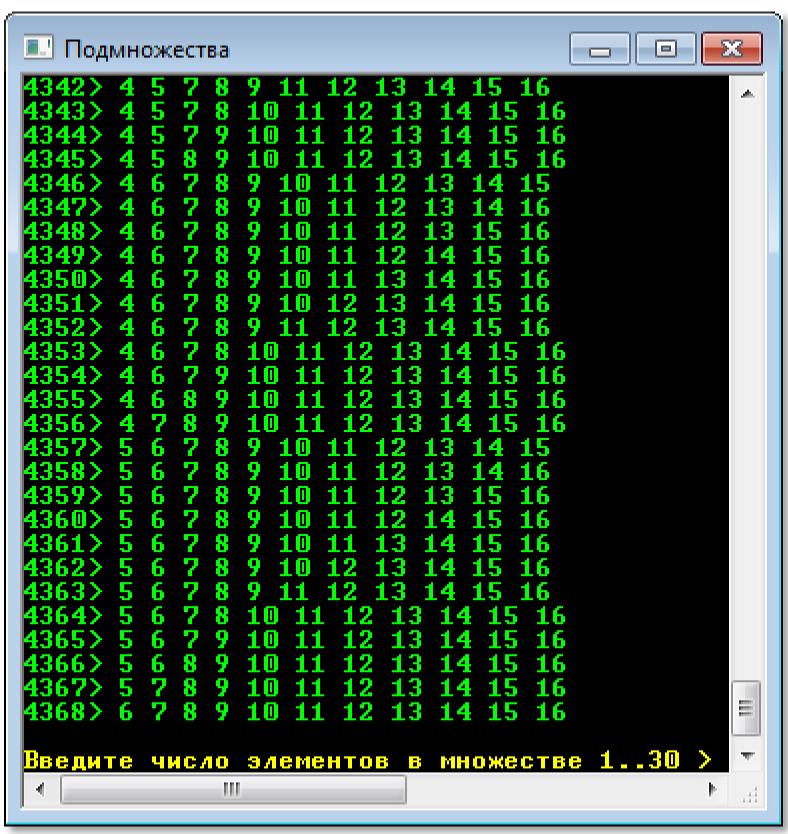


Рис. 12.5. Вот сколько сборных можно послать на чемпионат!

Исходный код программы находится в папке **Subset**.

1. Все перестановки большого числа элементов невозможно увидеть в консольном окне, поэтому организуйте запись результатов работы программы в файл.

2. Подумайте, как вместо чисел использовать другие элементы, например, буквы.

3. Напишите программу, которая вычисляла бы и печатала в консольном окне число сочетаний.

4. Переделайте программу *Factorial* так, чтобы она вычисляла факториал перемножением чисел от 2 до заданного.

Урок 13. Занимательная математика

Предмет математики настолько серьёзен,

что полезно не упускать случаев

делать его немного занимательным.

Блез Паскаль

Давайте уделим немного нашего времени и внимания *числам Фибоначчи*, которые, как нетрудно догадаться, открыл Фибоначчи (он же Леонардо Пизанский), средневековый математик, автор *Книги абака*, которую он написал в 1202 году.

Впрочем, дотошные историки утверждают, что этот ряд чисел был известен в Индии, где он использовался при стихосложении, задолго до Фибоначчи.

В этой книге, среди прочих, есть и задача о размножении кроликов. Подробно мы её рассмотрим на уроке *Занимательное моделирование*, а на этом мы ограничимся только тем, что *найдём все числа Фибоначчи, которые не превышают некоторого заданного числа n.*

Как обычно, начинаем программу с раздела *переменных*:

```
//ПРОГРАММА ДЛЯ НАХОЖДЕНИЯ ВСЕХ ЧИСЕЛ ФИБОНАЧЧИ,
//КОТОРЫЕ НЕ БОЛЬШЕ НЕКОТОРОГО ЗАДАННОГО ЧИСЛА

uses CRT;

var
   number: integer;
   f, f1, f2: integer;
```

После запуска программы пользователь вводит с клавиатуры число, ограничивающее поиски чисел Фибоначчи

```
//============================================
//          ОСНОВНАЯ ПРОГРАММА
//============================================
```

```
begin
   SetWindowTitle('Числа Фибоначчи');

   while(true) do
   begin
      repeat
         TextColor(Yellow);
         write('Введите число 1..40000000 >  ');
         TextColor(CRT.LightGreen);
         number := readInteger;
```

Мы, естественно, должны *проверить* правильность ввода:

```
        //Если задан нуль,
         //то работу с программой заканчиваем:
         if (number = 0) then exit;
      until ((number >= 1) and (number <= 40000000));
      writeln;
```

Если всё нормально, мы удовлетворяем запросы пользователя:

```
      f:=  0;
      f1:= 1;
      f2:= 1;
      TextColor(CRT.LightGreen);
      //Вычисляем и выводим в консольное окно
      //числа Фибоначчи, не превышающие number:
      writeln(0);
      While f2 <= number do
      begin
         writeln(f2);
         f2 := f1+f;
         f  := f1;
         f1 := f2;
      end;//While

      writeln;
      TextColor(Yellow);
   end;
end.
```

Первые три числа мы задаём сразу:

```
f:=  0;
f1:= 1;
```

```
f2:= 1;
```

Первое число равно нулю, поэтому его нужно напечатать сразу, потому что оно в дальнейших расчётах не участвует:

```
writeln(0);
```

Второе число равно 1, а все последующие равны сумме двух предыдущих, то есть третье число - 0 + 1 = 1, четвёртое - 1 + 1= 2, пятое - 1 + 2 = 3 и так далее, до бесконечности:

```
f2 := f1+f;
f  := f1;
f1 := f2;
```

Первое число обозначено одной буквой *f*, а второе буквой *f* и единицей. Логичнее было бы обозначить их как *f1* и *f2*, но нередко последовательность чисел Фибоначчи начинается с *единицы*, а не с нуля. Вот поэтому и возникла у нас такая несуразица!

В цикле *While* мы проверяем, не достигло ли очередное число Фибоначчи заданного нами предела. Как только программа закончит поиск (Рис. 13.1), она вернётся к началу, и пользователь сможет ввести новое число.

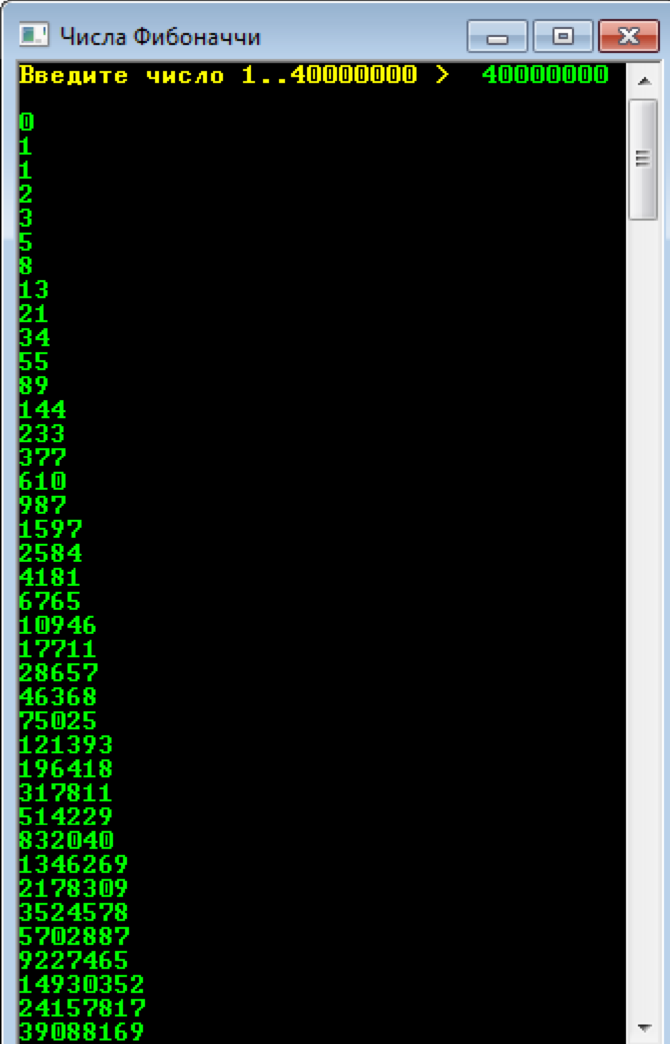


Рис. 13.1. Числа Фибоначчи

Исходный код программы находится в папке **Fibonacci**.

А теперь мы напишем две родственные программы из школьной жизни – для вычисления *НОД* (*наибольшего общего делителя* двух чисел) и *НОК* (*наименьшего общего кратного*). Они вам должны быть хорошо известны (если это не так, то сходите, наконец, в школу!).

Наибольший общий делитель (*НОД*)

Для вычисления *НОД* мы воспользуемся *ускоренным алгоритмом Евклида* для двух чисел - *number1* и *number2:*

```
//ПРОГРАММА ДЛЯ ВЫЧИСЛЕНИЯ НАИБОЛЬШЕГО
//ОБЩЕГО ДЕЛИТЕЛЯ ДВУХ ЧИСЕЛ (НОД)

uses CRT;

var
   number1:=0;
   number2:=0;
   NOD:=0;

//=========================================
//          ОСНОВНАЯ ПРОГРАММА
//=========================================
begin
   SetWindowTitle('НОД');

   while(true) do
   begin
      TextColor(Yellow);
      write('Введите первое число  >     ');
      TextColor(CRT.LightGreen);
      number1 := readInteger;
      //Если задано отрицательное число,
      //то работу с программой заканчиваем:
      if (number1 < 0) then exit;
```

```
      TextColor(Yellow);
      write('Введите второе число  >     ');
      TextColor(CRT.LightGreen);
      number2 := readInteger;
```

Чтобы не обременять пользователя лишними заботами, мы позволим ему вводить числа произвольно, хотя для алгоритма важно, чтобы первое число было *больше* второго, поэтому выправляем ситуацию, если это необходимо:

```
      //если первое число меньше второго,
      //то меняем их значения:
      if number1 < number2 then
      begin
         var n:= number1;
         number1:=number2;
         number2:=n;
      End;//If
```

При равенстве чисел алгоритм сработает правильно.

Если меньшее из чисел (или оба) равны нулю, то за *НОД*, понятное дело, следует принять *первое* число:

```
      if number2=0 then
         NOD := number1
      else
        NOD := speed_euklid();
```

Если же оба числа положительные, то мы начинаем действовать по ускоренному алгоритму Евклида:

```
   function speed_euklid: integer;
   begin
       while number2 > 0 do
       begin
          var n := number1 mod number2;
          number1 := number2;
          number2 := n;
       End;//While
```

```
        Result := number1;
   End;
```

В цикле *While* мы на каждой итерации находим остаток от деления первого числа на второе, значение второго числа присваиваем первому числу, а значение остатка - второму. Так мы продолжаем до тех пор, пока остаток от деления не станет равным *нулю*. Поскольку с каждым разом одно из чисел уменьшается, то рано или поздно это условие будет выполнено.

Вычисленное значение *НОД* мы печатаем в консольном окне (Рис. 13.2):

```
      writeln;
      TextColor(LightGreen);
      //выводим НОД в текстовое окно:
      writeln('НОД = ' + NOD.ToString());

      writeln;
      TextColor(Yellow);
   end;
end.
```

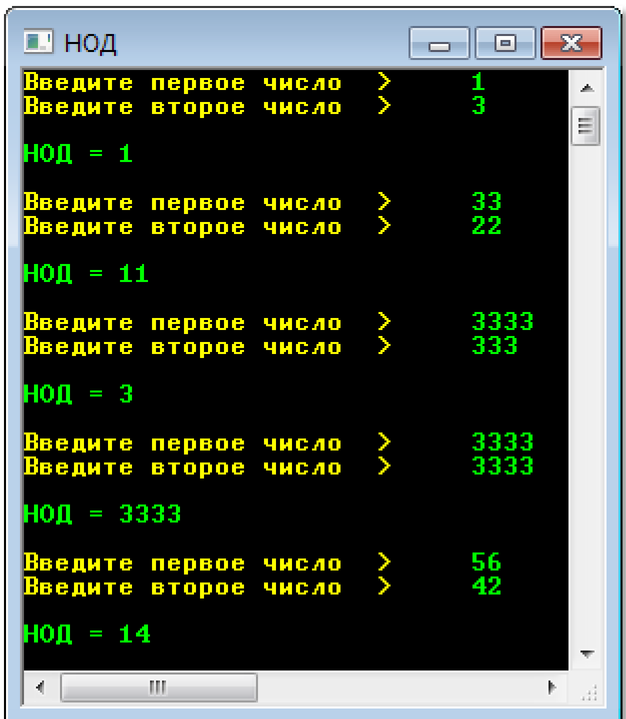


Рис. 13.2. Вычисляем *НОД*

Для большего понимания работы алгоритма давайте найдём *НОД* для чисел

```
number1 := 42;
number2 := 14;
```

Так как второе число больше нуля, то находим остаток от их деления:

```
var n := number1 mod number2;
```

И присваиваем новые значения переменным:

```
number1 := 14;
number2 := 0;
```

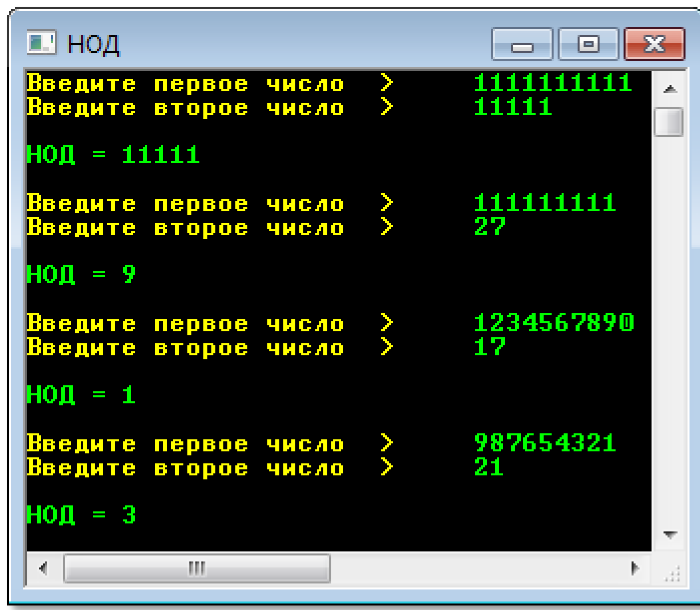
Второе число теперь равно нулю, значит, *НОД* найден – он равен второму числу, то есть 14.

Рассмотрим другой пример:

```
number1 := 56;
number2 := 42;
```

```
n := 56 mod 42;  = 14
```

```
number1 := 42;
number2 := 14;
```



После первого же цикла мы пришли к первому примеру.

Вы можете взять любые числа (в диапазоне допустимых значений для типа *integer*!) и убедиться, что алгоритм действительно работает очень быстро (Рис. 13.3)!

Рис. 13.3. Наша программа легко справляется и с большими числами!

Исходный код программы находится в папке **NOD**.

Наименьшее общее кратное (*НОК*)

Зная наибольший общий делитель двух чисел, мы очень просто вычислим *наименьшее общее кратное* по такой формуле:

НОК = число1 * число2 / НОД (число1, число2)

Возьмём за основу предыдущую программу и все вычисления *НОД* перенесем в функцию *calcNOD*:

```
//Функция для вычисления НОД двух чисел -
//n1 и n2
function calcNOD(n1, n2: integer): integer;
begin
   //если первое число меньше второго,
   //то меняем их значения:
   if n1 < n2 then
   begin
      var n:= n1;
      n1:=n2;
      n2:=n;
   End;//If

  if n2=0 then
     NOD := n1
  else
     while n2 > 0 do
     begin
        var n := n1 mod n2;
        n1 := n2;
        n2 := n;
     End;//While
   Result := n1;
End;
```

Эту функцию вы можете использовать в любых программах, в которых требуется вычислять *НОД*. Ей передаются два аргумента –

заданные числа, а функция возвращает их наибольший общий делитель.

В начале программы мы объявляем глобальные *переменные*:

```
//ВЫЧИСЛЕНИЕ НАИМЕНЬШЕГО ОБЩЕГО
//КРАТНОГО ДВУХ ЧИСЕЛ (НОК)
uses CRT;

var
   number1:=0;
   number2:=0;
   NOD:=0;
   NOK:=0;
```

Вы легко сможете обойтись только *локальными* переменными. Попробуйте!

А дальше мы просто вычисляем *НОК* по указанной выше формуле (Рис. 13.4):

```
//==========================================
//            ОСНОВНАЯ ПРОГРАММА
//==========================================
begin
   SetWindowTitle('НОК');

   while(true) do
   begin
      TextColor(Yellow);
      write('Введите первое число  >     ');
      TextColor(CRT.LightGreen);
      number1 := readInteger;
      //Если задано отрицательное число,
      //то работу с программой заканчиваем:
      if (number1 < 0) then exit;

      TextColor(Yellow);
      write('Введите второе число  >     ');
      TextColor(CRT.LightGreen);
      number2 := readInteger;

      if number1 * number2=0 then
```

```
            NOK := 0
        else
        begin
            //вычисляем NOD:
            NOD:= calcNOD(number1, number2);
            NOK:= number1 * number2 div NOD;
        End;//If
        writeln;
        TextColor(LightGreen);
        //выводим НОК в текстовое окно:
        writeln('НОК = ' + NOK.ToString());

        writeln;
        TextColor(Yellow);
    end;
end.
```

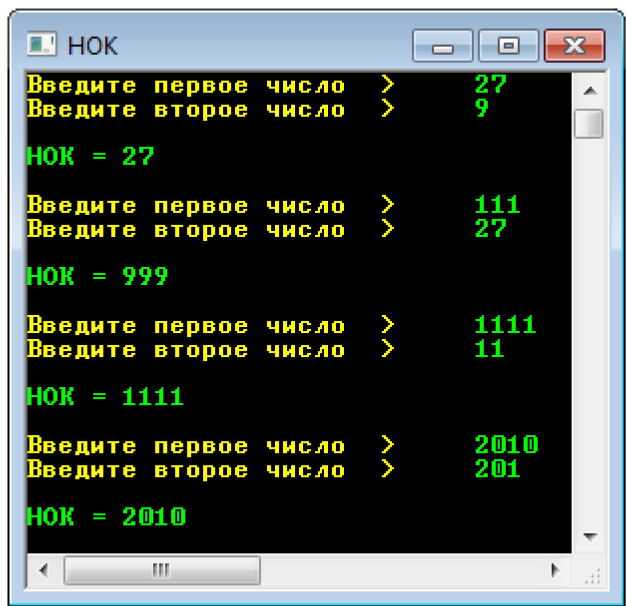


Рис. 13.4. Вычисляем *НОК*

Исходный код программы находится в папке **NOK**.

Урок 14. Графические приложения

С появлением операционной системы *Windows* основным интерфейсом пользователя стал *графический*, а текстовый вывод на консоль применяется только для быстрого получения результатов вычислений.

Вся информация – текстовая и графическая - при этой разновидности интерфейсов выводится в *графическое окно*, которое описано в модуле **GraphABC**.

Рабочий стол

Поскольку окно приложения – и консольного, и графического – «лежит» на *Рабочем столе*, то давайте сначала познакомимся с ним.

Если вы взглянете на *Рабочий стол*, то сразу поймёте, что его самые главные свойства – это *ширина, высота* (конечно, при этом имеются в виду не сантиметры, что важно для нас, а *пиксели*, с которыми компьютер дружит больше).

Ширину *Рабочего стола* сообщит нам функция *ScreenWidth:*

function ScreenWidth: integer;

А его **высоту** - функция *ScreenHeight*:

function ScreenHeight: integer;

С помощью этих функций вы только *узнаете* размеры стола, но *не сможете их изменить*. Казалось бы, размеры *Рабочего стола* вообще изменить нельзя, поскольку он существует только на экране монитора, а тот «не резиновый». На самом деле это не так: мы измеряем экран в *пикселях*, но пиксели можно сделать и больше, и меньше – в зависимости от *разрешения* экрана, а его изменить очень даже можно.

А теперь давайте напишем небольшое приложение, которое создаст *графическое окно* и напишет на нём размеры экрана:

```
//РАБОЧИЙ СТОЛ
uses GraphABC;

begin
   SetWindowTitle('Графическое окно');
   SetWindowWidth(400);
   SetWindowHeight(300);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.Green);

   SetFontSize(12);
   SetFontColor(Color.Yellow);
   SetFontStyle(fsBold);
   SetBrushStyle(bsClear);
   TextOut(10,10,'Ширина экрана = ' + ScreenWidth.ToString());
   TextOut(10,40,'Высота экрана = ' + ScreenHeight.ToString());
end.
```

Со всеми премудростями *графического окна* мы будем знакомиться дальше, а пока – запускаем приложение кнопкой *F9* и получаем интересующую нас информацию (Рис. 14.1).

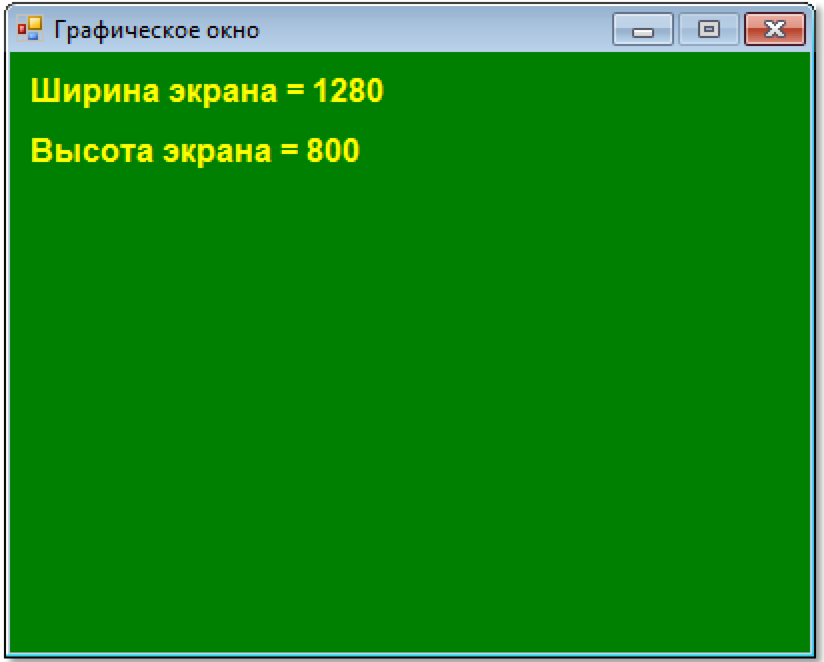


Рис. 14.1. Размер имеет значение!

Естественно, экран вашего монитора может оказаться «ширее» и «вышее».

Графическое окно

Всякая программа с графическим интерфейсом начинается с создания и настройки параметров окна приложения (Рис. 14.1). В *паскале графическое окно* создаётся *автоматически* - достаточно указать, что в программе используется модуль *GraphABC*:

```
uses GraphABC;
```

Для этого можно воспользоваться *шаблоном* кода: набираете буквы *gr*, нажимаете клавиши *Shift+ПРОБЕЛ* – и заготовка для графического приложения готова:

```
uses GraphABC;

begin

end.
```

Уже сейчас можно запустить «приложение» кнопкой *F9* и получить на экране *графическое окно* (Рис. 14.2)!

По умолчанию оно имеет размеры 640 x 480 пикселей и заголовок *GraphABC.NET*. *Графическое окно* появляется в центре экрана, а его клиентская часть окрашена в белый цвет.

Как видите, *графическое окно паскаля* – это обычное окно *Windows*, оно имеет 4 стандартные кнопки (Рис. 14.3).

Рис. 14.2. Графическое окно создано!

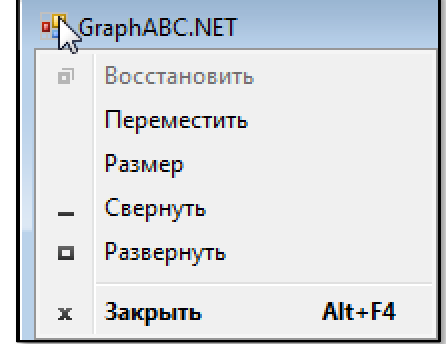


Рис. 14.3. Кнопки стандартного окна

С их помощью можно проделывать обычные «фортели»: сворачивать окно, разворачивать его, перемещать и, наконец, закрыть.

Стандартный *заголовок* окна легко заменить названием программы, передав процедуре *SetWindowTitle* нужное значение (Рис. 14.4):

```
SetWindowTitle('Графическое окно');
```

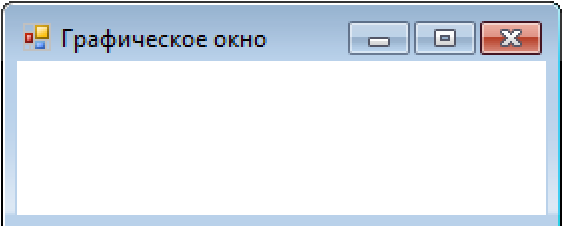


Рис. 14.4. Вот так-то лучше!

То же самое делает и процедура

procedure SetWindowCaption(s: string);

Другой способ изменения названия окна – присвоить нужное строковое значение свойству *Title* окна:

```
Window.Title:='Графическое окно';
```

Если вы забудете, что написано, в заголовке окна, то легко освежите память, считав значение этого свойства в строковую переменную (Рис. 14.5):

```
var s:= WindowCaption;
TextOut(10,10,'Заголовок окна - ' + s);
```

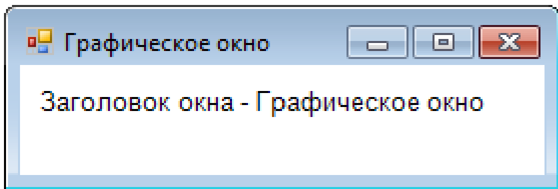
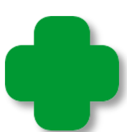


Рис. 14.5. Не забуду заголовок!



То же самое делает и функция

function WindowCaption: string;

Заголовок окна можно использовать и нестандартно, например, выводить в него число набранных в игре очков, время и другую полезную информацию. Напишем небольшое приложение **title** с таймером. Он будет срабатывать каждую секунду, а его процедура *OnTimer* напечатает число секунд, прошедших со старта приложения (Рис. 14.6):

```
uses GraphABC, Timers;

var
    time: integer;

procedure OnTimer;
begin
  time += 1;
  SetWindowTitle(time.ToString());
End;

begin
   SetWindowTitle('Графическое окно');
   SetWindowWidth(400);
   SetWindowHeight(300);
   Window.CenterOnScreen();
   Window.IsFixedSize := false;
   Window.Clear(Color.Green);

   time:=0;
   var timer := new Timer(1000, OnTimer);
   timer.Start();
end.
```

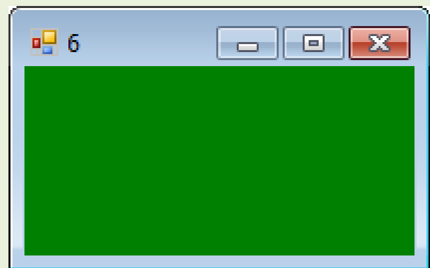


Рис. 14.6. Время пошло!

Обычно пользователь может не только перетаскивать окно по экрану, но и изменять его размеры, потянув за края или углы. Иногда это приводит к тому, что графические элементы про-

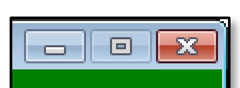
граммы перемещаются, и вид программы становится неряшливым. Поэтому вы можете пресечь подобное «вольнодумство» пользователя и запретить ему изменять размеры окна, присвоив свойству **IsFixedSize** значение *true*:

```
Window.IsFixedSize := true;
```

При этом кнопка *Развернуть* побледнеет, а в системном меню окна заблокируются некоторые команды (Рис. 14.7).

Впрочем, вы можете сменить гнев на милость и снова вернуть пользователю свободу действий:

```
Window.IsFixedSize := false;
```



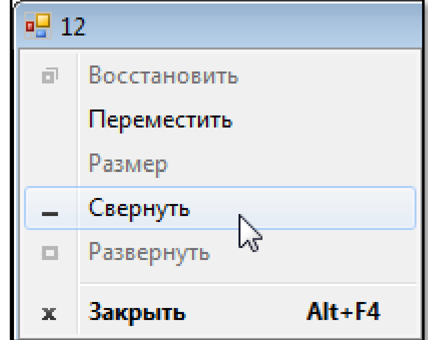


Рис. 14.7. Приложение строгих форм!

Значение *false* свойство *IsFixedSize* имеет по умолчанию, поэтому, если вы не защитите программу, то пользователь безнаказанно сможет изменять размеры её окна.

Как мы уже знаем, стандартные размеры окна - 640 на 480 пикселей, но их можно изменить, не только с помощью умелых рук пользователя, но и программно.

За *размеры окна* отвечают процедуры с понятными названиями. Передав им нужные значения

```
SetWindowWidth(240);
SetWindowHeight(120);
```

мы получим окно размером поменьше (Рис. 14.8).

Можно и просто задать свойствам окна *Width* и *Height* новые значения:

```
Window.Width:=240;
Window.Height:=120;
```

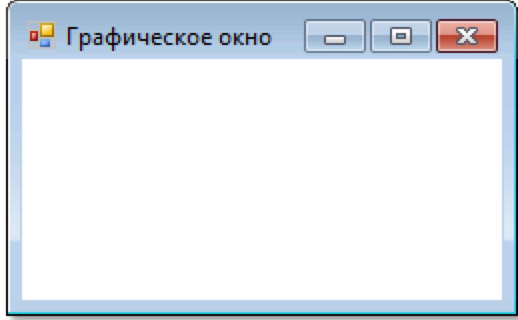


Рис. 14.8. Из окна мы сделали окошечко!

Можно сделать и «форточку» (Рис. 14.9).

```
SetWindowWidth(0);
SetWindowHeight(0);
```

Рис. 14.9. Из окошечка мы сделали форточку!

Однако даже нулевые значения этих свойств не уменьшают окно до размеров точки!

Остаются только границы окна и заголовок, клиентская часть исчезает напрочь, поэтому пользы от такого окна будет не много.

Так же легко мы можем *установить окно* в любом месте экрана, задав нужные координаты его левого верхнего угла:

```
Window.Left := 100;
Window.Top := 100;
```

По умолчанию новое окно появляется в центре экрана, но если вы переместили его в другое место или изменили размеры окна,

то снова вернуть его в центр экрана можно, вызвав метод *CenterOnScreen*:

Window.CenterOnScreen();

"Форменная" лихорадка

Давайте воспользуемся свойствами *Width* и *Height* окна и заставим его беспрерывно «ёрзать» по экрану. Сделать это проще простого. Для стимулирования окна нам потребуется объект *Таймер* (*Timer*). При его создании в конструкторе следует указать время срабатывания таймера в миллисекундах (200-400) и процедуру, которая будет выполняться при его срабатывании:

```
var timer := new Timer(200, OnTimer);
```

Изменить частоту срабатывания таймера можно в любое время, задав его свойству *Interval* новое значение:

```
timer.Interval:=200;
```

При срабатывании таймера, то есть через каждые 0,2-0,4 секунды будет вызываться процедура *OnTimer*:

```
procedure OnTimer;
begin
  if rele then begin
     Window.Left += 5;
     Window.Top += 5;
  end
  else begin
    Window.Left -= 5;
    Window.Top -= 5;
  End;
  //переключить направление перемещения формы:
  rele := succ(rele);
End;
```

В зависимости от значения переключателя *rele* окно немного сдвигается вниз-вправо, а потом возвращается на место. На ста-

тичном рисунке этого не покажешь, а в жизни бегающее окно выглядит очень забавно.

Для работы приложения с таймером необходимо добавить модуль *Timers:*

```
//"ФОРМЕННАЯ" ЛИХОРАДКА

uses GraphABC, Timers;
```

И переменную *rele* логического типа:

```
var
    rele: boolean;
```

В главной части программы мы устанавливаем параметры окна, создаём таймер и запускаем его методом *Start:*

```
begin
   SetWindowTitle('Форменная лихорадка');
   Window.Width:=260;
   Window.Height:=120;
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.Green);

   rele:= true;
   var timer := new Timer(200, OnTimer);
   timer.Start();
end.
```

Заставьте окно перемещаться по всему экрану, чтобы разозлить пользователя не на шутку!

Исходный код программы находится в папке **GraphicsWindow**.

Элементы стандартного окна

Многие из них мы уже рассмотрели:

В верхней части окна находится полоска, которую обычно называют **заголовком**. Заголовок нужен для перемещения окна мышкой, а также на нём расположены:

- **Значок формы** – по умолчанию это стандартный значок *Windows* для приложений *Windows Forms*, и мы не можем заменить его своим.

- **Заголовок формы** – по умолчанию там выводится значение свойства *Title графического окна*.

- **Кнопка сворачивания окна** (минимизации) – окно превращается в кнопку на *Панели задач*, которая из нажатого положения (активное окно) переходит в отжатое. Чтобы активизировать окно, достаточно щёлкнуть по его кнопке на *Панели задач*.

- **Кнопка разворачивания окна** (максимизации) – окно раскрывается во весь экран (за исключением *Панели задач*, если не изменены её свойства), а вместо этой кнопки появляется **кнопка восстановления окна**, при нажатии на которую окно принимает прежние размеры.

- **Кнопка закрытия окна** – окно исчезает с экрана и работа приложения завершается.

Системное меню окна – появляется, когда вы нажимаете правую кнопку мыши на заголовке окна. Оно дублирует уже рассмотренные нами команды управления окном.

Граница окна – весь контур окна позволяет изменять его размеры. Для этого нужно поставить курсор мыши на границу окна и, когда курсор примет вид двойной стрелочки , потянуть её в нужную сторону.

Клиентская область – всё остальное пространство окна.

Светоформа

По умолчанию клиентская область окна имеет *белый* цвет, но некоторые любят и другие цвета. Что ж, очень просто удовлетворить желания самых изощрённых любителей прекрасного. Для этого методу *Clear* следует передать нужный цвет (Рис. 14.10):

```
Window.Clear(Color.Green);
```

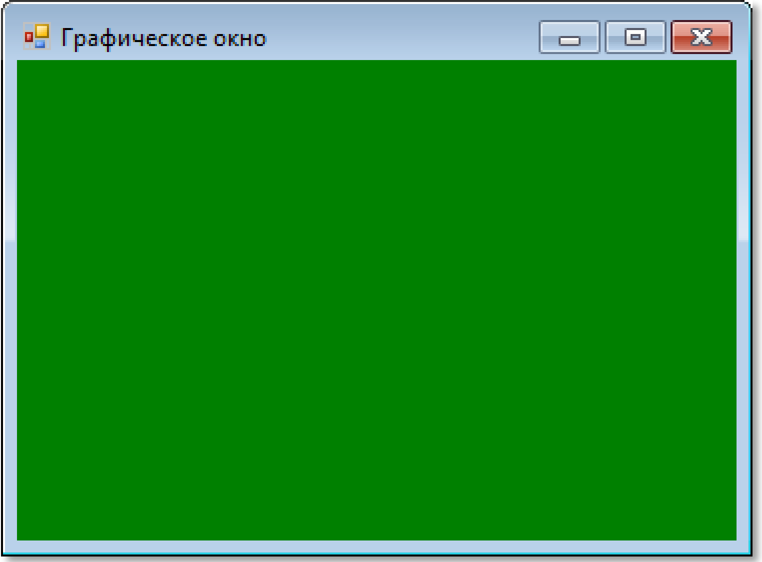


Рис. 14.10. Окно **позеленело**!

А теперь давайте превратим окно в **светофор** - чтобы оно переливалось всеми цветами радуги. Достаточно немного изменить процедуру из предыдущей программы, обрабатывающую «тиканье» таймера:

```
//МИГАЮЩЕЕ ОКНО

uses GraphABC, Timers;

//Мигание окна
procedure OnTimer;
begin
  Window.Clear(clRandom);
End;
```

После старта программы окно будет с каждым тиком таймера изменять свой цвет произвольным образом (Рис. 14.11).

Обратите внимание на новую функцию модуля *GraphABC* **clRandom,** которая возвращает «случайный» цвет. Если вам безразлична последовательность смены цветов, то это самый быстрый и простой способ получить произвольный цвет.

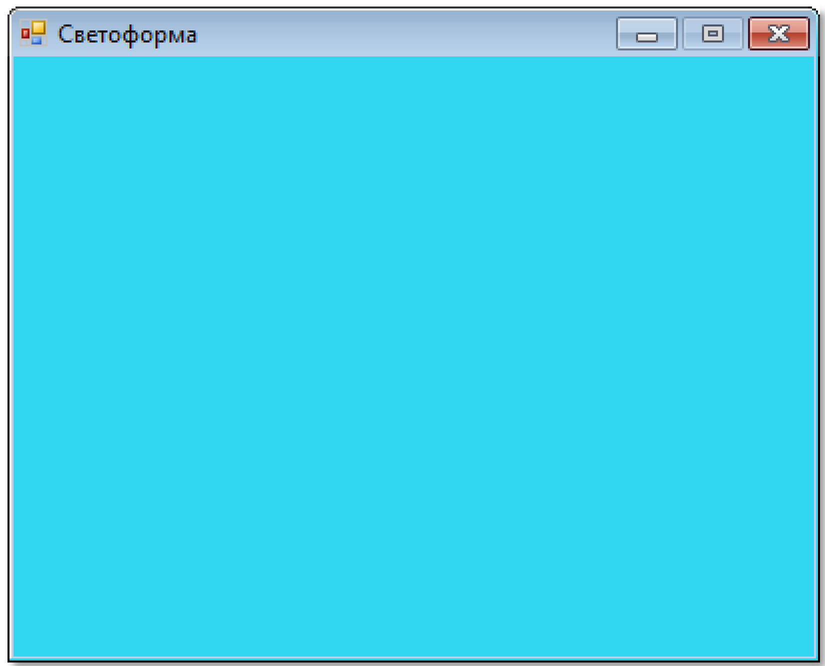


Рис. 14.11. Окно-светофор

Ещё немного изменим программу, и окно будет не только перекрашиваться, но и *изменять свои размеры*:

```
//МИГАНИЕ ОКНА И
//ИЗМЕНЕНИЕ ЕГО РАЗМЕРОВ

uses GraphABC, Timers;

var rele: integer;
    n: integer;
    clr: Color;

//Формируем случайный цвет
function GetRandomColor(): Color;
begin
  var red:=   Random(256);
  var green:= Random(256);
  var blue:=  Random(256);
  Result:= Color.FromArgb(255, red, green, blue);
End;
```

```
//Мигание и изменение размеров окна
procedure OnTimer;
begin

   if n= 100 Then begin
      rele := -rele;
      n:=0;
      //clr:= clRandom;
      clr:= GetRandomColor();
  end
  else
    n += 1;

   Window.Width += rele;
   Window.Height += rele;
   Window.Clear(clr);
   Window.CenterOnScreen();
End;

begin
   SetWindowTitle('Светоформа 2');
   SetWindowWidth(400);
   SetWindowHeight(300);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   clr:= clRandom;
   Window.Clear(clr);

   rele:= 1;
   n:=1;

   var timer := new Timer(10, OnTimer);
   timer.Start();

end.
```

Для задания цвета окна можно использовать и метод **FromARGB** структуры *Color*, который преобразует составляющие цвета (красную - **Red**, зелёную - **Green** - и синюю - **Blue**). Напишем новую функцию *GetRandomColor*, которая возвращает новый случайный цвет:

```
//Формируем случайный цвет
function GetRandomColor(): Color;
begin
  var red:=   Random(256);
  var green:= Random(256);
  var blue:=  Random(256);
  Result:= Color.FromArgb(255, red, green, blue);
End;
```

Исходный код программы находится в папке **GraphicsWindow**.

Так как *Графическое окно* имеет очень много методов и свойств, то его изучение мы продолжим на следующем уроке.

Урок 15. Текст в *графическом окне*

Забавно было выводить информацию в заголовок окна, но куда красивее текст можно напечатать в самом окне. И для этого *паскаль* предоставляет в наше полное распоряжение всё, что нужно.

Вы можете печатать текст *любым* шрифтом, установленным на вашем компьютере. Достаточно передать функции **SetFontName** название шрифта (Рис. 15.1):

```
SetFontName('Showcard Gothic');
```

Не пользуйтесь редкими шрифтами, если планируете передавать программу другим пользователям! Если у них такого шрифта не окажется, он будет заменён стандартным (Рис. 15.2).

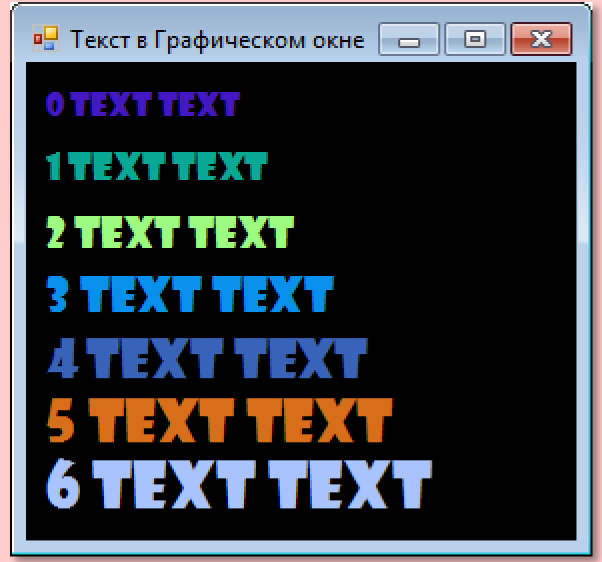


Рис. 15.1. Надпись шрифтом *Showcard Gothic*

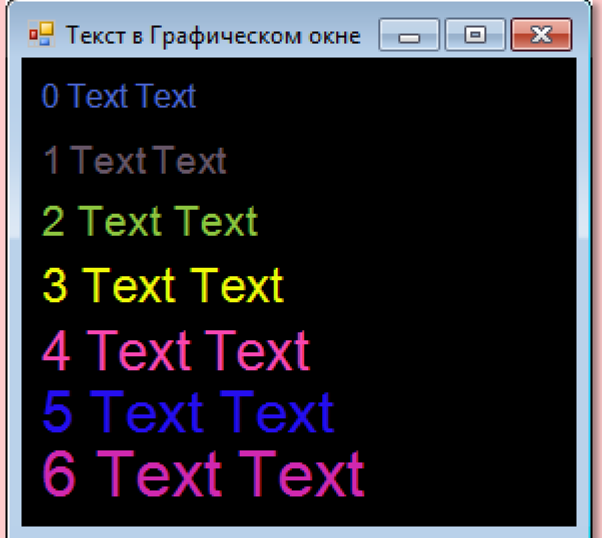


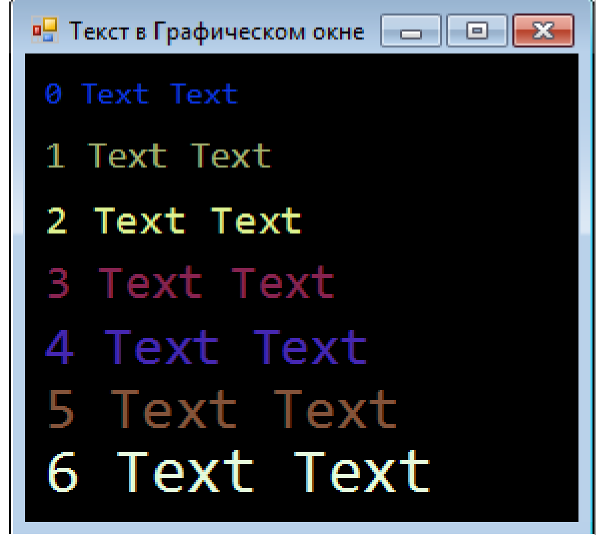
Рис. 15.2. Шрифт *Showcard Gothic* не установлен

Практически на всех компьютерах стоят такие шрифты (Рис. 15.3).

Arial

Courier New

Times New Roman



Consolas

Рис. 15.3.

А вот программа, которая напечатала все эти надписи:

```
//ТЕКСТ В ГРАФИЧЕСКОМ ОКНЕ

uses GraphABC;

begin
   SetWindowTitle('Текст в Графическом окне');
   SetWindowWidth(400);
   SetWindowHeight(300);
   Window.CenterOnScreen();
   Window.Clear(Color.Black);
```

```
  //SetFontName('Arial');
  SetFontName('Showcard Gothic');
  //SetFontName('Courier New');
  //SetFontName('Times New Roman');
  //SetFontName('Consolas');

  SetFontStyle(fsNormal);
  SetBrushStyle(bsClear);
  For var i:= 0 To 6 do
  begin
     SetFontSize(12+ i*2);
     SetFontColor(clRandom);
     TextOut(10,10+i*30, i.ToString() + ' Text Text');
  End;//For
end.
```

Как видите, *размер шрифта* задаётся процедурой **SetFontSize**, а его *цвет* – процедурой **SetFontColor**. Сам же *текст* выводится процедурой **TextOut(x, y, строка)**.

> *Первое число* в скобках – координата начала строки в пикселях, которая отсчитывается от левого края клиентской области окна.
>
> *Второе число* – координата верхней границы строки, которая отсчитывается от верхнего края клиентской области.
>
> Затем следует указать строковое выражение (*строку*) для ввода на экран.

По умолчанию все надписи выполняются нормальным шрифтом. Если вам больше по душе **жирные** буквы, то передайте процедуре *SetFontStyle* значение *fsBold* (Рис. 15.4) :

SetFontStyle(fsBold);

Буквы станут **жирными**, но вы легко вернёте им прежнюю стройность:

SetFontStyle(fsNormal);

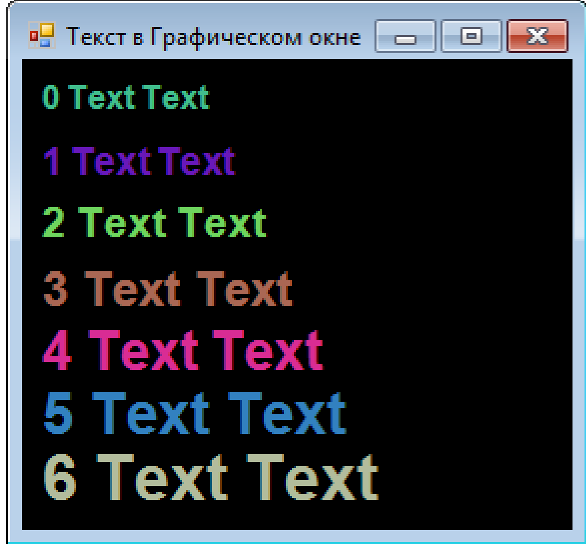


Рис. 15.4. Жирный *Arial*

А такая строка заставит *склониться* все буквы в почтении (Рис. 15.5):

SetFontStyle(fsItalic);

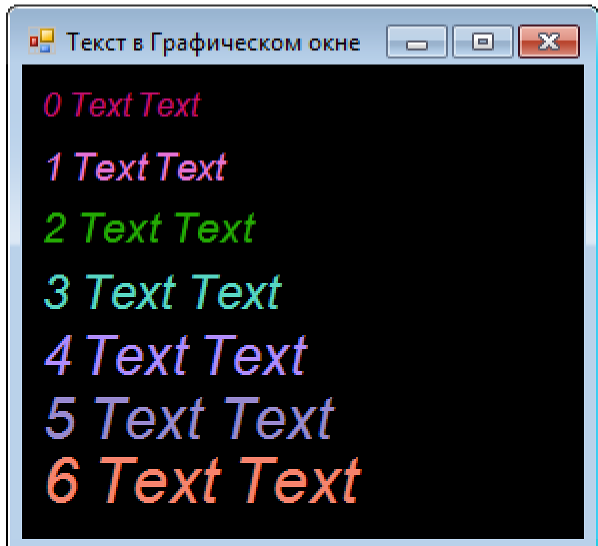


Рис. 15.5. *Курсивный* шрифт

Вот значения всех допустимых аргументов процедуры *SetFontStyle*:

fsNormal – нормальный
fsBold – жирный
fsItalic – курсив

fsBoldItalic – жирный курсив
<u>fsUnderline</u> – подчёркивание
<u>fsBoldUnderline</u> – жирный с подчёркиванием
<u>fsItalicUnderline</u> – курсив с подчёркиванием
<u>fsBoldItalicUnderline</u> - жирный курсив с подчёркиванием

Исходный код программы находится в папке **Текст в Графическом окне**.

События *Графического окна*

Операционная система *Windows* управляет всеми приложениями с помощью *сообщений*, которые она им посылает. Эти сообщения принимает окно приложения и реагирует на них. Например, если пользователь нажмёт какую-нибудь клавишу, система *Windows* посылает активному окну сообщение *WM_KEYDOWN*. Если это наше *графическое окно*, то в нём возникает событие *KeyDown*. Если нас это событие интересует, мы можем написать процедуру-обработчик для этого события и указать *графическому окну* её название:

```
//СОБЫТИЯ ГРАФИЧЕСКОГО ОКНА

uses GraphABC;

procedure KeyDown(Key: integer);
begin
   TextOut(10,10, 'Вы нажали клавишу');
End;

begin
   Window.Title:='События Графического окна';

   Window.Width:=360;
   Window.Height:=240;
   Window.CenterOnScreen();
   Window.Clear(Color.Green);
   SetFontColor(Color.Yellow);
   SetBrushStyle(bsClear);
```

```
   SetFontStyle(fsBold);
   OnKeyDown := KeyDown;
end.
```

Запускаем программу, и как только мы нажмём клавишу, так сразу же получим окно с приятным известием (Рис. 15.6).

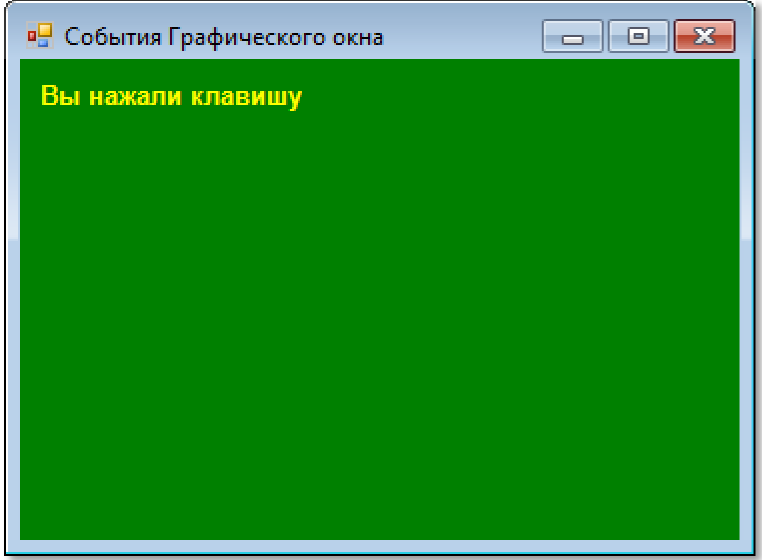


Рис. 15.6. Вам «письмо»!

Мы легко узнаем, какая именно клавиша была нажата, прочитав значение параметра *ch* в процедуре *KeyPress* (Рис. 15.7).

```
OnKeyPress:= KeyPress;

procedure KeyPress(ch: char);
begin
   writeln('Вы нажали клавишу ' + ch);
End;
```

При отпускании клавиши возникает событие *KeyUp*, которое мы также можем обработать с помощью процедуры (Рис. 15.8):

```
OnKeyUp := KeyUp;

procedure KeyUp(Key: integer);
begin
   writeln('Вы отпустили клавишу');
End;
```

События Графического окна
Вы нажали клавишу П
Вы нажали клавишу а
Вы нажали клавишу с
Вы нажали клавишу к
Вы нажали клавишу а
Вы нажали клавишу л
Вы нажали клавишу ь
Вы нажали клавишу Р
Вы нажали клавишу a
Вы нажали клавишу s
Вы нажали клавишу c
Вы нажали клавишу a
Вы нажали клавишу l

Рис. 15.7. Система знает всё!

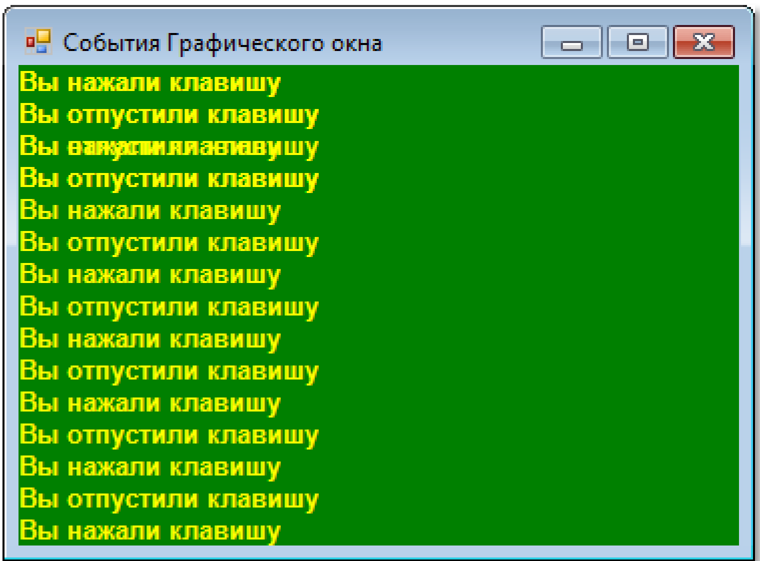


Рис. 15.8. Она и это знает!

Программа *Розыгрыш*

Обогатившись знаниями о событиях, мы можем написать небольшую программу, которая будет самым живым образом реагировать на действия доверчивого пользователя:

```
//ПРОГРАММА "РОЗЫГРЫШ"

uses GraphABC;

procedure KeyDown(Key: integer);
```

```
begin
   SetBrushStyle(bsSolid);
   Window.Clear(Color.Black);
   SetFontColor(Color.Red);
   SetBrushStyle(bsClear);
   TextOut(10,10, 'Не нажимайте клавишу, мне больно!');
End;
procedure KeyUp(Key: integer);
begin
   SetBrushStyle(bsSolid);
   Window.Clear(Color.Black);
   SetFontColor(Color.LightGreen);
   SetBrushStyle(bsClear);
   TextOut(10,10, 'Спасибо, мне стало легче!');
End;

begin
   Window.Title:='Нажмите клавишу!';

   Window.Width:=465;
   Window.Height:=60;
   Window.CenterOnScreen();
   Window.Clear(Color.Black);
   SetFontColor(Color.Yellow);
   SetFontStyle(fsBold);
   SetFontSize(18);
   OnKeyDown := KeyDown;
   //OnKeyPress:= KeyPress;
   OnKeyUp := KeyUp;
end.
```

Когда пользователь нажмёт клавишу, в окне появится жалобное сообщение (Рис. 15.9).

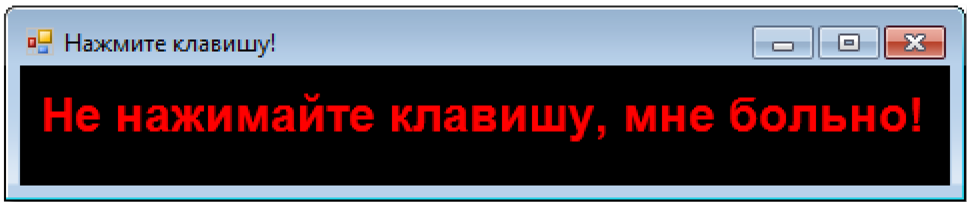


Рис. 15.9. А, может, это правда?!

Гуманный пользователь тут же отпустит клавишу, за что получит благодарность (Рис. 15.10).

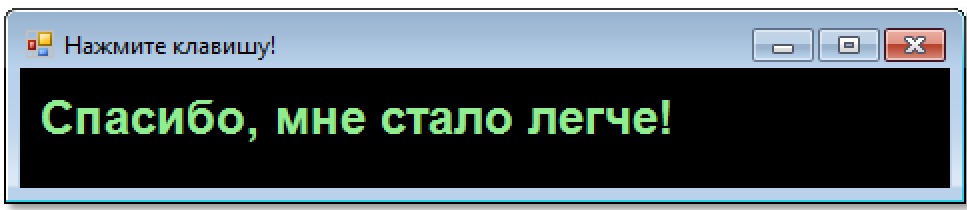


Рис. 15.10. Никогда не делайте больно!

Если пользователь долго держит клавишу нажатой и мучает *графическое окно*, значит, ему пора на урок психологии!

Обратите внимание на метод **Clear**, который очищает окно от любых надписей и рисунков, закрашивая клиентскую область в заданный цвет.

Чтобы буквы не затирали фон, мы устанавливаем *прозрачный* стиль кисти:

```
SetBrushStyle(bsClear);
```

Но такой кистью нельзя закрасить фон, чтобы стереть предыдущую надпись, поэтому перед использованием метода *Clear*, нужно вернуть кисти значение по умолчанию – *сплошная* кисть:

```
SetBrushStyle(bsSolid);
```

Как видите, ничего нового в программе нет, а получилось смешно!

Исходный код программы находится в папке **Текст в Графическом окне**.

Розыгрыши продолжаются!

Поскольку кроме клавиатуры, у пользователя всегда под рукой *мышка*, то мы вполне можем предположить, что *графическое окно*

реагирует и на мышиную «возню». И мы не ошиблись в наших ожиданиях: реагирует, да ещё как!

Мы не можем и не должны останавливаться на констатации сего приятного факта, а наоборот, мы сразу же примемся использовать мышиные события для продолжения розыгрышей.

Снова будем выводить в окно смешные надписи. При нажатии и отпускании кнопки мыши в окне одна надпись будет сменяться другой. На этот раз мы возьмём замечательные выражения из «негритянских» рассказов юмориста Лукинского.

```
//ПРОГРАММА "РОЗЫГРЫШ" 2

uses GraphABC;

procedure MouseDown(x,y,mousebutton: integer);
begin
   Window.Title:= 'Отпусти кнопку мышки!';
   SetBrushStyle(bsSolid);
   Window.Clear();

   SetFontColor(Color.Blue);
   SetBrushStyle(bsClear);
   TextOut(10,10, 'С Новым годом!');
End;

procedure MouseUp(x,y,mousebutton: integer);
begin
   Window.Title:= 'Нажми кнопку мышки!';
   SetBrushStyle(bsSolid);
   Window.Clear();

   SetFontColor(Color.Red);
   SetBrushStyle(bsClear);
   TextOut(10,10, 'Пошёл на фиг!');
End;

begin
   Window.Title:='Нажми кнопку мышки!';

   Window.Width:= 300;
   Window.Height:=60;
```

```
   Window.CenterOnScreen();
   Window.Clear(Color.White);
   SetFontColor(Color.Blue);
   SetFontStyle(fsBold);
   SetFontSize(24);

   OnMouseDown:= MouseDown;
   OnMouseUp:= MouseUp;
end.
```

Отозвавшись на просьбу окна, выраженную в его заголовке, доверчивый пользователь нажмёт кнопку мышки и получит поздравление с праздником (Рис. 15.11).

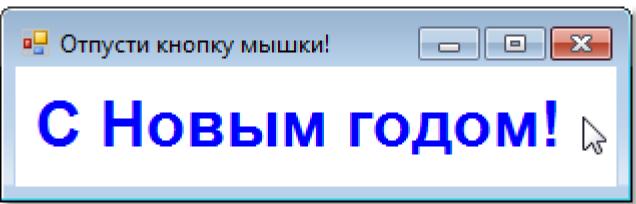


Рис. 15.11. Спасибо!

Теперь окно попросит его отпустить кнопку мышки и недобро пошлёт его - в другое приложение (Рис. 15.12).

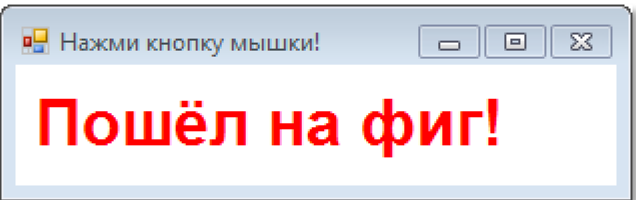


Рис. 15.12. Ну, Лукинский даёт!

Согласитесь, жить стало веселей. А вы можете повеселиться и пуще того, если заготовите множество таких фраз и подсунете их своим товарищам, близким и родственникам. Они непременно будут вам благодарны за доставленное удовольствие.

Исходный код программы находится в папке **Текст в Графическом окне.**

Прикольное окно

Попробуем сделать окно «прикольным». Прикол будет незатейливым: при попытке пользователя переместить окно, оно будет упрямо возвращаться в середину экрана (Рис. 15.13).

```
//ПРОГРАММА "ПРИКОЛЬНОЕ ОКНО"

uses GraphABC, Timers;

procedure OnTimer;
begin
   Window.CenterOnScreen();
End;

begin
   Window.Title:='Прикол';

   Window.Width:= 420;
   Window.Height:=60;
   Window.CenterOnScreen();
   Window.Clear(Color.Black);
   SetFontColor(Color.Blue);
   SetFontStyle(fsBold);
   SetFontSize(24);
   SetBrushStyle(bsClear);
   TextOut(10,10, 'Передвинь меня в угол!');

   var timer := new Timer(10, OnTimer);
   timer.Start();
end.
```

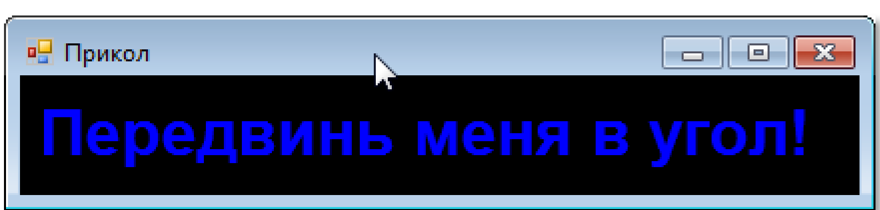


Рис. 15.13. Окно на приколе

Работает программа очень просто. Каждые 10 миллисекунд срабатывает таймер, и процедура-обработчик *OnTimer* возвращает окно на место.

Исходный код программы находится в папке **Текст в Графическом окне.**

Ввод текста в *Графическом окне*

Графическое окно реагирует ещё на одно событие, которое может пригодиться вам, например, для ввода информации с клавиатуры.

Когда пользователь нажимает клавишу, возникает событие *KeyPress*, а параметр *ch* содержит символ, соответствующий этой клавише, - букву, цифру, знак препинания и т.д. В процедуре-обработчике мы можем составить из этих символов *строку* (Рис. 15.14), а затем вывести на экран или использовать в программе как-то иначе.

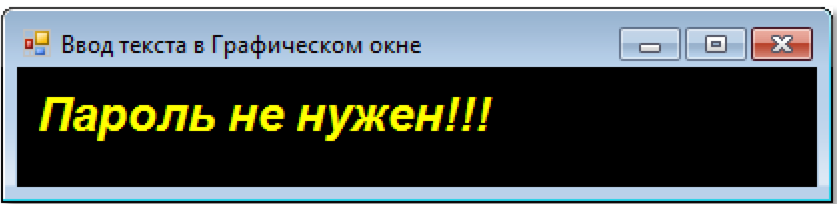


Рис. 15.14. Строка, собранная по буквам!

```
//ВВОД ТЕКСТА В ГРАФИЧЕСКОМ ОКНЕ

uses GraphABC;

var s: string;

procedure KeyPress(ch: char);
begin
   s += ch;
   SetBrushStyle(bsSolid);
   Window.Clear(Color.Black);
   SetBrushStyle(bsClear);
   TextOut(10,10, s);
```

```
End;

begin
   SetWindowTitle('Ввод текста в Графическом окне');
   SetWindowWidth(400);
   SetWindowHeight(60);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.Black);

   SetFontName('Arial');
   SetFontSize(18);
   SetFontColor(Color.Yellow);
   SetFontStyle(fsBoldItalic);

  OnKeyPress:= KeyPress;
end.
```

В данном примере после ввода каждой буквы приходится стирать всё окно, иначе надписи накладываются друг на друга, что выглядит неаккуратно. Скоро мы познакомимся с геометрическими фигурами, которые можно рисовать в *графическом окне*, и они помогут нам стирать любую часть клиентской области, не причиняя вреда окружающей среде.

Исходный код программы находится в папке **Текст в Графическом окне**.

Урок 16. Класс *Math*

В паскале над числовыми константами и переменными можно производить различные *арифметические действия*:

Знак операции	Операция
+	Сложение
-	Вычитание
*	Умножение
/	Деление
div	Целочисленное деление
mod	Остаток от целочисленного деления

Вы, наверное, заметили, что некоторые знаки арифметических операций отличаются от тех, что вы используете в школе: знак умножения в виде крестика или точки здесь заменён *звёздочкой*, а двоеточие – знак деления – *дробной чертой* (её также называют *слешем*). Это объясняется тем, что знака умножения нет на клавиатуре, а двоеточие используется для других целей.

Минус служит также для *изменения знака* выражения противоположным: 2 → -2; x → -x.

Рассмотрим *примеры*. Пусть
a := 6;
b := 3;

Действие	Значение переменной *Num*
Num := a + b;	9
Num := a - b;	3
Num := a * b;	18
Num := a / b;	2
Num := a **div** b;	2
Num := a **mod** b;	0

При вычислении выражений учитывается *приоритет* операций – точно так же, как и в математике. Сначала слева направо выполняются операции умножения и деления * / **div mod**, а затем - сложения и вычитания - +. Изменить *порядок* вычисления можно с помощью *круглых скобок* (но не квадратных и не фигурных!) – в полном согласии с законами арифметики:

Действия	Значение переменной *Num*
Num := (2 + 4 - 1) / 5 * 11;	11
Num := (3.1415 – 1.0) / 7.78 * (45 - 7) ;	10,4597686375321
Num := -2.67 * 3.14 + -3/2 – 7.01;	-16,8938
Num := 3 * 4 + 5 – 6/2;	14

Квадратные скобки нужны для обозначения элементов массива, а фигурные скобки {} в паскале применяются только в комментариях.

Между числами и знаками операций может быть сколько угодно пробелов, но можно печатать их и вплотную друг к другу.

Для «научных» вычислений в платформе *.NET* существует специальный класс **Math**, который имеет два свойства *Pi и E*, значения которых, как легко догадаться, равны *π и e*, соответственно. Зато методов у него вполне достаточно, чтобы выполнять довольно сложные расчеты.

Чтобы пользоваться классом *Math*, нужно добавить к проекту пространство имён *System*:

```
uses
   GraphABC, System;
```

Чтобы вам было удобнее пользоваться методами класса *Math*, мы сведём их в *таблицу*.

Большинство этих методов вам известны по школе, поэтому мы не будем рассматривать их подробно, а напишем программу **Класс Math** с примерами вычислений. В случае необходимости вы сможете подставить свои значения в эти методы, чтобы лучше изучить их работу.

Pi	3,14159265358979
E	2,71828182845905
Abs(число)	Возвращает *абсолютную величину* числа
Acos(косинус)	Возвращает угол в радианах, соответствующего значению *косинуса*
Asin(синус)	Возвращает угол в радианах, соответствующего значению *синуса*
Atan(тангенс)	Возвращает угол в радианах, соответствующего значению *тангенса*
Ceiling(число)	*Округляет* заданное дробное число до большего целого.
Cos(угол)	Возвращает *косинус* угла, заданного в радианах
Log(число)	Возвращает *натуральный логарифм* заданного числа
Log10(число)	Возвращает *десятичный логарифм* заданного числа
Max(число1, число2)	Возвращает *большее* из двух чисел
Min(число1, число2)	Возвращает *меньшее* из двух чисел
Pow(число, степень)	Возвращает результат возведения числа в заданную *степень*
IEEERemainder(число1, число2)	Возвращает *остаток от деления* первого числа на второе
Round(число)	*Округляет* заданное дробное число до ближайшего целого.
Sin(угол)	Возвращает *синус* угла, заданного в радианах
Sqrt (число)	Возвращает *корень квадратный* из числа
Tan(угол)	Возвращает *тангенс* угла, заданного в радианах

```
//Класс System.Math

uses
   GraphABC, System;

begin
   Window.Title := 'Класс System.Math';
   Window.Width := 360;
   Window.Height := 440;
   SetFontColor(Color.Blue);
   SetBrushStyle(bsClear);
   SetFontStyle(fsBold);
   SetFontSize(12);

   writeln('Пи = ' + Math.Pi.ToString());
   writeln('e = ' + Math.E.ToString());
   writeln('Abs(-2) = ' + Math.Abs(-2).ToString());
   writeln('Acos(0.5) = ' + Math.Acos(0.5).ToString());
   writeln('Asin(0.5) = ' + Math.Asin(0.5).ToString());
   writeln('Atan(0.5) = ' + Math.Atan(0.5).ToString());
   writeln('Ceiling(0.5) = ' + Math.Ceiling(0.5).ToString());
   writeln('Ceiling(1) = ' + Math.Ceiling(1).ToString());
   writeln('Cos(0.5) = ' + Math.Cos(0.5).ToString());
   writeln('Floor(0.5) = ' + Math.Floor(0.5).ToString());
   writeln('Floor(1) = ' + Math.Floor(1).ToString());
   writeln('Log10(100) = ' + Math.Log10(100).ToString());
   writeln('Log(100) = ' + Math.Log(100).ToString());
   writeln('Max(-1, -2) = ' + Math.Max(-1, -2).ToString());
   writeln('Min(-1, -2) = ' + Math.Min(-1, -2).ToString());
   writeln('Pow(2, 32) = ' + Math.Pow(2, 32).ToString());
   writeln('IEEERemainder(15, 4) = ' + Math.IEEERemainder(15,
4).ToString());
   writeln('Round(0.5) = ' + Math.Round(0.5).ToString());
   writeln('Round(1.6) = ' + Math.Round(1.6).ToString());
   writeln('Sin(0.5) = ' + Math.Sin(0.5).ToString());
   writeln('Sqrt(2) = ' + Math.Sqrt(2).ToString());
   writeln('Tan(0.5) = ' + Math.Tan(0.5).ToString());
end.
```

Результаты вычислений мы будем выводить в *графическое окно* (Рис. 16.1):

```
Класс System.Math
Пи = 3,14159265358979
e = 2,71828182845905
Abs(-2) = 2
Acos(0.5) = 1,0471975511966
Asin(0.5) = 0,523598775598299
Atan(0.5) = 0,463647609000806
Ceiling(0.5) = 1
Ceiling(1) = 1
Cos(0.5) = 0,877582561890373
Floor(0.5) = 0
Floor(1) = 1
Log10(100) = 2
Log(100) = 4,60517018598809
Max(-1, -2) = -1
Min(-1, -2) = -2
Pow(2, 32) = 4294967296
IEEERemainder(15, 4) = -1
Round(0.5) = 0
Round(1.6) = 2
Sin(0.5) = 0,479425538604203
Sqrt(2) = 1,4142135623731
Tan(0.5) = 0,54630248984379
```

Рис. 16.1. Все методы класса *Math* в одном окошке!

Методы класса *Math* представляют собой *функции*, которые *возвращают* результат вычислений. Это значит, что, например, выражение *Math.Sin(0.5)* равно 0.479... Обычно результат, возвращаемый функцией, *присваивают* переменной:

```
var sinus:= Math.Sin(0.5);
```

После выполнения этого оператора значение переменной *sinus* будет равно 0.479...

Иногда вызовы функций используют только для того, чтобы *сравнить* возвращаемое значение с каким-либо другим значением:

```
 if (Math.Sin(0.5) < 0.5) Then
      . . .
```

Можно сделать проверку и так:

```
var sinus:= Math.Sin(0.5);
 if (sinus(0.5) < 0.5) Then
    . . .
```

Но тогда потребуется лишняя переменная и лишняя операция присваивания. Впрочем, для удобства нередко так и делают, потому что составление длинных выражений может привести к ошибкам, которые непросто выявить.

Исходный код программы находится в папке **Класс Math**.

Однако *паскаль* имеет и собственные, встроенные математические функции (они описаны в модуле *PABCSystem*), которыми вы также можете пользоваться в своих программах:

Abs(число)	Возвращает *абсолютную величину* числа
ArcCos(косинус)	Возвращает угол в радианах, соответствующего значению *косинуса*
ArcSin(синус)	Возвращает угол в радианах, соответствующего значению *синуса*
ArcTan(тангенс)	Возвращает угол в радианах, соответствующего значению *тангенса*
Ceil(число)	*Округляет* заданное дробное число до большего целого.
Cos(угол)	Возвращает *косинус* угла, заданного в радианах
Cosh(угол)	Возвращает *гиперболический косинус* угла, заданного в радианах
DegToRad(угол)	Пересчитывает угол, заданный в градусах, в радианы
Exp(число)	Возвращает *экспоненту* числа
Floor(число)	Возвращает наибольшее целое, меньшее или равнок заданному
Frac(число)	Возвращает *дробную часть* числа
Inc(число)	Возвращает *целую часть* числа
Ln(число)	Возвращает *натуральный логарифм* заданного числа
Log2(число)	Возвращает *логарифм* по основанию 2 заданного числа

Log10(число)	Возвращает *десятичный логарифм* заданного числа
LogN(основание, число)	Возвращает *логарифм* числа по заданному основанию
Max(число1, число2)	Возвращает *большее* из двух чисел
Min(число1, число2)	Возвращает *меньшее* из двух чисел
Odd(число)	Возвращает *true*, если число *нечётно*
Power(число, степень)	Возвращает результат возведения числа в заданную *степень*
RadToDeg(угол)	Пересчитывает угол, заданный в радианах, в градусы
Round(число)	*Округляет* заданное дробное число до ближайшего целого.
Sin(угол)	Возвращает *синус* угла, заданного в радианах
Sinh(угол)	Возвращает *гиперболический синус* угла, заданного в радианах
Sign(число)	Возвращает знак числа.
Sqr (число)	Возвращает *квадрат* числа
Sqrt (число)	Возвращает *корень квадратный* из числа
Tan(угол)	Возвращает *тангенс* угла, заданного в радианах
Tanh(угол)	Возвращает *гиперболический тангенс* угла, заданного в радианах
Trunc(число)	Возвращает *целую часть* числа

Поскольку их действие принципиально не отличается от методов класса *Math*, то я предоставляю вам полную свободу самостоятельно исследовать их.

Более того, функции *паскаля* реализованы на базе класса *Math*, поэтому они просто повторяют или используют его методы.

Вычисление числа π

А теперь давайте воспользуемся методами класса *Math*, чтобы вычислить значение числа *пи* (поскольку оно выражается беско-

нечной десятичной дробью, то мы сможем найти только *приближённое* значение).

Современное обозначение этого числа греческой буквой π предложил в 1706 году английский математик У.Джонсон. Он воспользовался первой буквой греческого слова *periferia* (конечно, в оригинальном написании, а не более удобными латинскими буквами), что значит *окружность*. Но общепризнанным в научном мире этот символ стал после того как в 1736 году Леонард Эйлер использовал его в своих работах.

История вычислений числа *пи*, которое равно отношению длины окружности к её диаметру, началась много тысячелетий назад.

Так, египетские математики считали *пи* равным $(16/9)^2$, то есть примерно 3,1604938..., а индийские – $\sqrt{10}$ = 3,16227766... В третьем веке до нашей эры Архимед установил, что *пи* меньше, чем 3 1/7, и больше, чем 3 10/71 = 3,1428 … 3,1408. Среднее значение этого диапазона равно 3,14185, что больше *пи* уже в четвертом десятичном знаке. Но ещё до Архимеда, в пятом веке до нашей эры китайский математик Цзу Чунчжи нашел более точное значение - 3,1415927... В первой половине пятнадцатого века ал-Каши, астроном и математик из Самарканда вычислил *пи* с точностью до 16 знаков после запятой. В 1615 году голландский математик Лудольф ван Цейлен довёл точность вычислений до 32 знаков. В 1873 году Вильям Шенкс после 20 лет расчётов нашёл 707 знаков числа *пи*, но в 1944 году Д.Фергюсон с помощью механического калькулятора выяснил, что верны только первые 527 знаков числа Шенкса. Для своих расчетов Шенкс использовал *формулу Дж. Мачина*:

$$\frac{\pi}{4} = 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$$

А *арктангенс* вычислял по формуле

$$\arctan x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{2k+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

Сам Мачин еще в 1706 году по этой формуле вычислил 100 знаков числа *пи*.

С появлением ЭВМ скорость вычислений значительно выросла. В 1949 году электронная машина ЭНИАК за 70 часов работы вычислила более двух тысяч десятичных знаков числа *пи*. Через некоторое время были найдены 3000 знаков всего за 13 минут. В 1959 году компьютеры преодолели рубеж десяти тысяч знаков, а сейчас известно несколько десятков миллионов знаков числа *пи*. Чтобы их напечатать, потребуется несколько толстенных книг.

Формула Валлиса

$$\pi = 2\left(\frac{2}{1} \times \frac{2}{3} \times \frac{4}{3} \times \frac{4}{5} \times \frac{6}{5} \times \frac{6}{7} \times \frac{8}{7} \times \frac{8}{9} \cdots\right)$$

Джон Валлис вывел эту красивую формулу в 1655 году, когда вычислял площадь круга. Она представляет собой бесконечное произведение дробей. К сожалению, чтобы вычислить даже несколько правильных знаков числа *пи* по этой формуле, нужно затратить немало времени и сил. Однако, имея компьютер, мы можем облегчить себе задачу. Итак, пишем программу:

```
//ВЫЧИСЛЕНИЕ ЧИСЛА ПИ

uses
   GraphABC, System;

//Вычисление по формуле Валлиса
procedure Wallis;
begin
   writeln('Вычисление по формуле Валлиса:');
   var npi:=1.0;
   for var n:= 1 To 1000000 do
      npi:= npi * (4.0*n*n/(2.0*n-1)/(2.0*n+1));

  writeln('Пи = ' + (2*npi).ToString());
  writeln;
```

```
end;

begin
   Window.Title := 'Вычисление числа пи';
   Window.Width := 360;
   Window.Height := 240;
   Window.CenterOnScreen();

   SetFontColor(Color.Blue);
   SetBrushStyle(bsClear);
   SetFontStyle(fsBold);
   SetFontSize(12);

   Wallis;
   Leibniz;
end.
```

После миллиона итераций получаем ответ (Рис. 16.2).

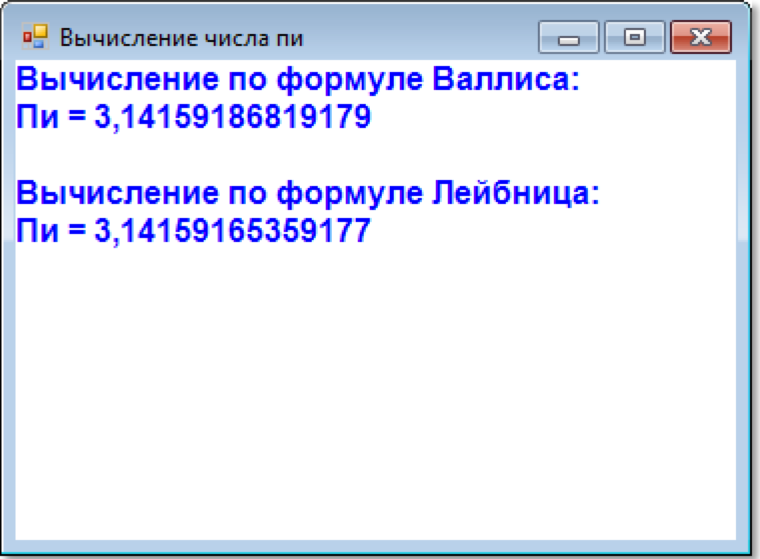


Рис. 16.2. Точности не хватает!

Да! Уже *шестой* знак после запятой неверный.

Ряд Лейбница

Попробуем зайти с другого конца и воспользуемся знакочередующимся рядом, который предложил немецкий математик Лейбниц. А ряд вот такой:

$$\pi = 4\left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \cdots\right)$$

Он хорош тем, что его легко приспособить под наши нужды:

```
//Вычисление по формуле Лейбница
procedure Leibniz;
begin
   writeln('Вычисление по формуле Лейбница:');
   var npi:=0.0;
   For var n:= 0 To 1000001 do
       npi := npi + Math.Pow(-1, n) / (2.0*n+1);

   writeln('Пи = ' + (4*npi).ToString());
   writeln;
end;
```

После миллиона итераций получаем результат (см. Рис. 16.2).

К сожалению, не лучше первого!

Всего существует не один десяток формул для вычисления *пи*, но они слишком сложны для занимательных уроков.

Исходный код программы находится в папке **Вычисление пи**.

Урок 17. Компьютерная графика

Лучше один раз увидеть,

чем сто раз услышать.

Программистская мудрость

Вся школьная геометрия построена из теорем, но мы их доказывать не будем, а просто собственными глазами убедимся, что из простых геометрических фигур с помощью компьютера можно создавать впечатляющие композиции.

Клиентская область *графического окна* выполняет ту же роль в *компьютерной графике*, что и холст в настоящей живописи. Нам уже довелось писать на ней разные словечки, а теперь мы возьмёмся за настоящее геометрическое творчество.

Пуантилизм, или Ставим точки

Мостовая пусть качнётся, как очнётся!
Пусть начнётся, что ещё не началось!
Вы рисуйте, вы рисуйте,
вам зачтётся...
Что гадать нам:
удалось - не удалось?

Булат Окуджава. *Живописцы*

Самым простым геометрическим объектом является *точка*. Она, как известно из геометрии, размеров не имеет. По той же причине она не имеет ни цвета, ни запаха. Таким образом, настоящую точку увидеть нельзя, а вот компьютерную - можно, хотя размером она примерно в четверть миллиметра. Называется такая точка **пикселем**. Пиксель мал да удал: его можно окрасить в миллионы цветов, а из множества пикселей мы сумеем нарисовать на экране монитора любую картину.

Процедура **SetPixel(x, y, color)** модуля *GraphABC* закрашивает пиксель клиентской области окна (для краткости мы будем называть её *холстом* или *канвой*) в заданный цвет. Цвет можно

задать любым способом из тех, что мы уже рассмотрели, а вот с координатами всё не так просто, как в школьной геометрии.

Расположение координатных осей на канве может показаться странным: ось *Y* (ордината) направлена вниз, а не вверх, поэтому, чем *больше* значение *Y*, тем *ниже* будет точка на экране. Положительное направление оси *X* совпадает с нашими представлениями, а вот начало координат находится в *левом верхнем* углу канвы, а вовсе не в её центре, как мы могли бы того ожидать (Рис. 17.1).

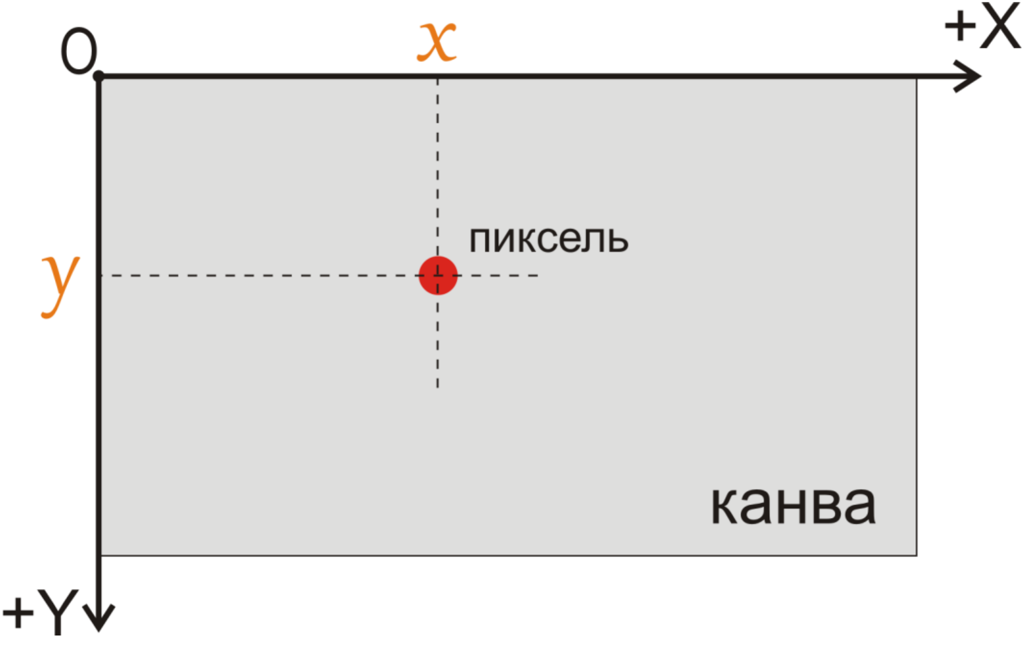


Рис. 17.1. Координатная система клиентской части окна приложения

Конечно, рисовать точками лучше в каком-нибудь графическом редакторе (*Microsoft Paint* или *Image Editor*, например, вполне годятся для этой цели), чем в программе на паскале, но поскольку все фигуры построены из точек, то давайте напишем простенькую программу, которая будет самостоятельно выводить на канву точки случайно выбранного цвета. Эстетического удовольствия мы не получим никакого, но зато хорошенько познакомимся с процедурой *SetPixel*. Программа **Пиксели** просто окрашивает точки канвы в случайные цвета, поэтому картинка получается совершенно хаотическая (Рис. 17.2).

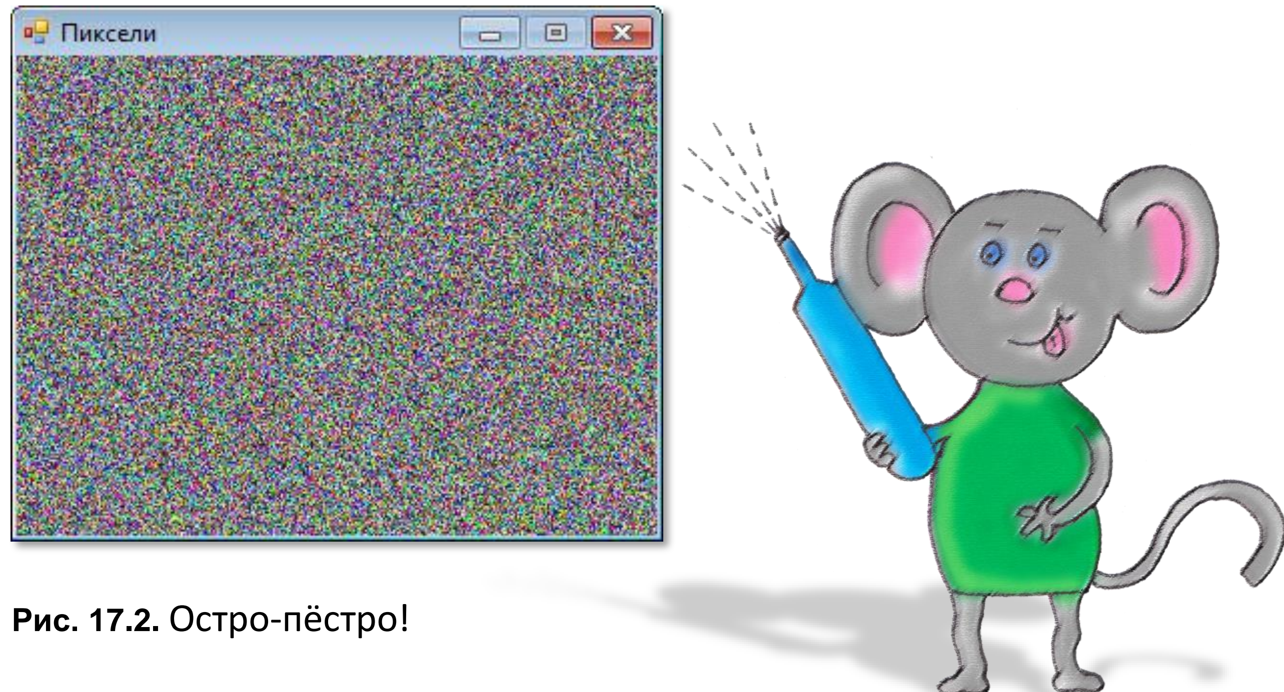


Рис. 17.2. Остро-пёстро!

Начните новый проект и сохраните его в папке **Pixel**.

Поскольку рисование точек на канве процесс довольно неспешный, то не задавайте большие размеры окна. Результат будет ничем не лучше, а времени вы потратите больше!

```
//ПИКСЕЛИ

uses
   GraphABC;

begin
   SetWindowTitle('Пиксели');
   SetWindowWidth(320);
   SetWindowHeight(240);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.Black);
```

В бесконечном цикле *While* программа последовательно перебирает все точки канвы и окрашивает их с помощью процедуры *SetPixel*:

```
  //Окрашиваем пиксели канвы в разные цвета
   var height := Window.Height;
   var width := Window.Width;
   while (true) do
```

```
   begin
      for var y := 0 to height - 1 do
      begin
         for var x := 0 to width - 1 do
         begin
            //выбираем случайный цвет для пикселя:
            var clr := clRandom;
            //и окрашиваем его:
            SetPixel(x, y, clr);
         end//For
      end//For
   end; //While
end.
```

Добавим к нашему проекту несколько строчек кода, чтобы получить интересный визуальный эффект: надпись *Пиксели* будет печататься *над* цветными точками (Рис. 17.3).

```
. . .
SetFontSize(56);
SetFontColor(Color.Red);
SetFontStyle(fsBold);
SetBrushStyle(bsClear);
. . .
   SetPixel(x, y, clr);
end; //For
//надпись:
TextOut(1, height div 2 - 40, 'Пиксели');
```

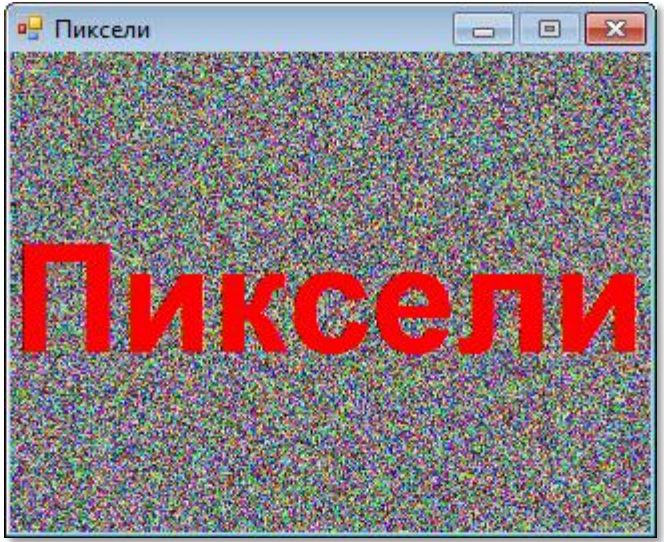


Рис. 17.3. Нестираемая надпись

Исходный код программы находится в папке **Pixel**.

«Пипетка»

Мы можем представить канву в виде двухмерного массива пикселей, каждый из которых характеризуется *координатами* (индексами массива) и *цветом*. Чтобы узнать, какой цвет имеет тот или иной пиксель канвы, достаточно обратиться к функции **GetPixel(x,y)**.

Начните новый проект и сохраните его в папке **Пипетка**.

Пипеткой в графических редакторах называют инструмент, который помогает узнать цвет пикселя под курсором. В этом режиме работы он как бы превращается в пипетку (Рис. 17.4)

Рис. 17.4. Пипетка в *Фотошопе*

```
//ПИПЕТКА

uses
   GraphABC;
begin
   SetWindowTitle('Цвет пикселя');
```

```
   SetWindowWidth(520);
   SetWindowHeight(200);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.Black);
```

Сначала мы окрашиваем все пиксели канвы в разные цвета, а затем перемещаем мышку по канве и в строке заголовка читаем координаты курсора и цвет пикселя под ним:

```
   OnMouseMove := MouseMove;

   //Окрашиваем пиксели канвы в разные цвета
   var height := Window.Height;
   var width := Window.Width;
   for var y := 0 to height - 1 do
   begin
      for var x := 0 to width - 1 do
      begin
         //выбираем случайный цвет для пикселя:
         var clr := clRandom;
         //и окрашиваем его:
         SetPixel(x, y, clr);
      end; //For
   end;//For
end.
```

Координаты курсора (точнее – его «горячей точки», которая обычно находится в верхнем левом углу) можно легко узнать по значению параметров *x* и *y* процедуры **MouseMove**, которая вызывается при наступлении события *OnMouseMove*, то есть при перемещении мышки по окну приложения.

Вся интрига этой программы заключается вот в этих скупых строках:

```
//определяем цвет пикселя под мышкой
procedure MouseMove(x, y, mousebutton: integer);
begin
   var clr := GetPixel(x, y);
   Window.Title := 'Цвет пикселя (' + x.ToString() +
                 ',' + y.ToString() + ') = ' + clr.ToString();
```

```
end;
```

Цвет пикселя под *горячей точкой* курсора (её координаты можно прочитать в заголовке окна) в виде цветных составляющих указан там же (Рис. 17.5).

К сожалению, пиксели такие маленькие, что трудно определить их цвет на глаз, поэтому добавим в программу небольшое «окошко», которое будем закрашивать цветом текущего пикселя (Рис. 17.6).

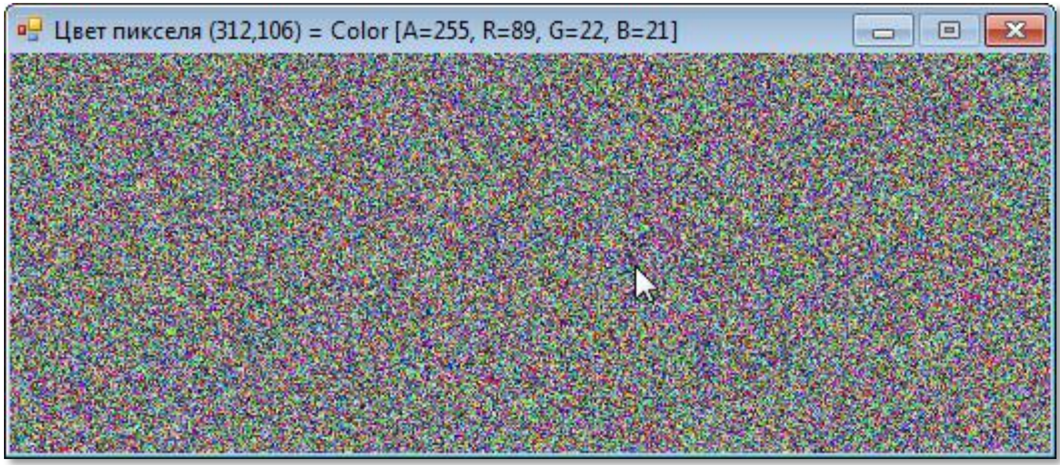


Рис. 17.5. Цветная пипетка в действии!

```
//определяем цвет пикселя под мышкой
procedure MouseMove(x, y, mousebutton: integer);
begin
   var clr := GetPixel(x, y);
   //окошко:
   SetBrushColor(clr);
   var x1 := Window.Width - 32 - 6;
   var y1 := 20;
   FillRectangle(x1, y1, x1 + 32, y1 + 32);
   Window.Title := 'Цвет пикселя (' + x.ToString() +
                  ','  + y.ToString() + ') = ' +
clr.ToString();
end;
```

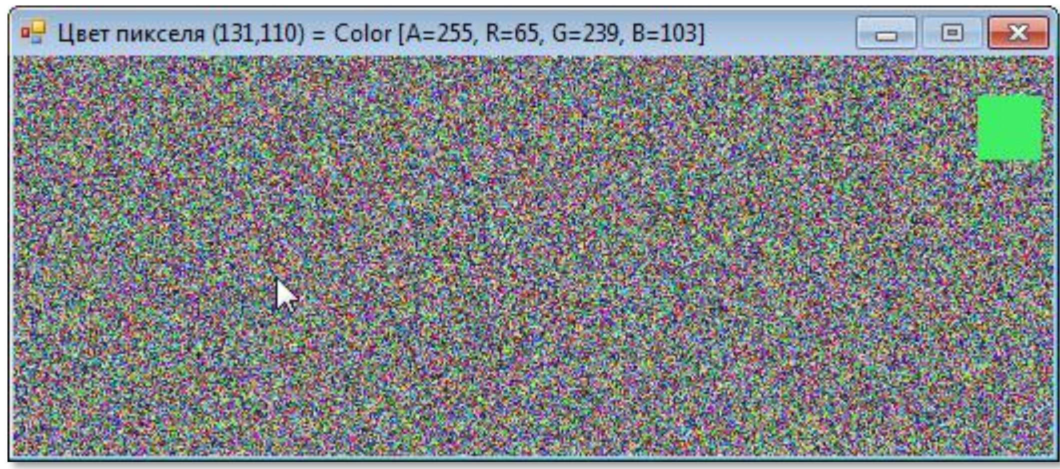


Рис. 17.6. А вот теперь каждый пиксель виден как на ладони!

Если вам понравится какой-нибудь цвет, запомните его код и применяйте в программах. Например, так:

```
var clr:= Color.FromArgb(255, 65, 239, 103);
```

Исходный код программы находится в папке **Пипетка**.

Многоточие

Раз уж компьютерные точки всё равно имеют размер, то мы сделаем их крупнее, чтобы лучше разглядеть. В этом деле нам потребуется не микроскоп и даже не увеличительное стекло, а новая программа **Многоточие**, которая умеет рисовать огромные точки, так что после её работы экран будет усыпан ими, как новогодний пол – конфетти (Рис. 17.7).

И в этом случае цвет и место точек задаются случайным образом, но теперь картина получается более «художественная».

Начало программы мы просто скопируем из предыдущих проектов:

```
//МНОГОТОЧИЕ
```

```
uses
   GraphABC;

begin
   SetWindowTitle('Многоточие');
   SetWindowWidth(640);
   SetWindowHeight(480);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.Black);

   var height := Window.Height;
   var width := Window.Width;
```

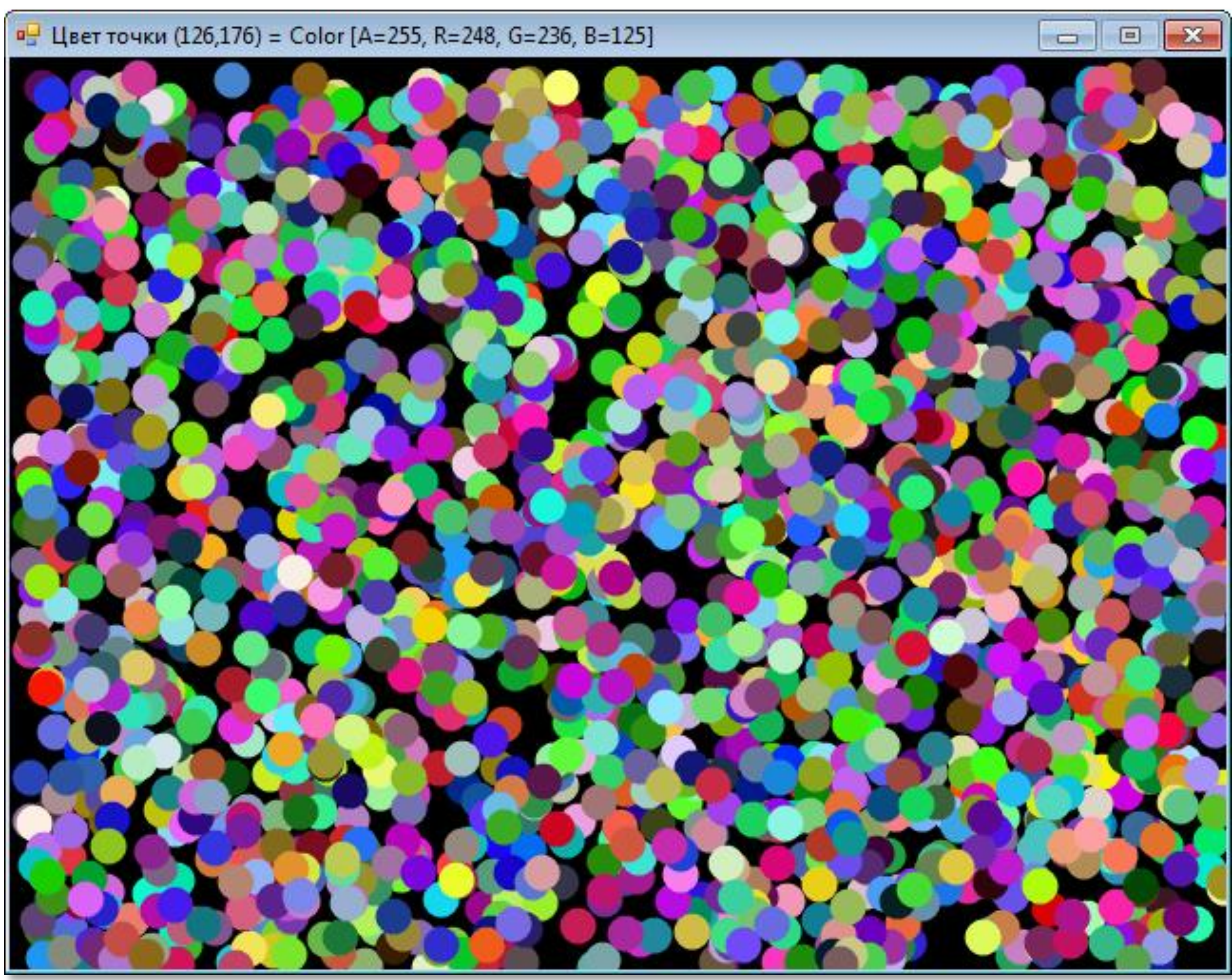


Рис. 17.7. Супер-пиксели!

Но теперь вместо отдельных пикселей мы рисуем «точки»:

```
   //радиус "точки":
   var radius:=10;
   for var i:= 1 to 2000 do //10000
   begin
      //выбираем случайные координаты  "точки":
      var x := Random(width-radius);
      var y := Random(height-radius);
      //выбираем случайный цвет для  "точки":
      var clr:= clRandom;

      Window.Title:= 'Цвет точки (' + x.ToString() +
               ',' + y.ToString() + ') = ' + clr.ToString();
      //рисуем цветную "точку":
      SetBrushColor(clr);
      FillEllipse(x, y, x+2*radius, y+2*radius);
   End;//For
end.
```

На самом деле мы, конечно, схитрили и нарисовали не точки, а окрашенные *эллипсы* (а точнее - кружочки). Для этого нам понадобилась процедура **FillEllipse(x1, y1, x2, y2).** *Первая пара* параметров – это *координаты* верхнего левого угла описанного прямоугольника, *вторая пара* – координаты его правого нижнего угла (Рис. 17.8). Цвет эллипса задаётся процедурой *SetBrushColor*.

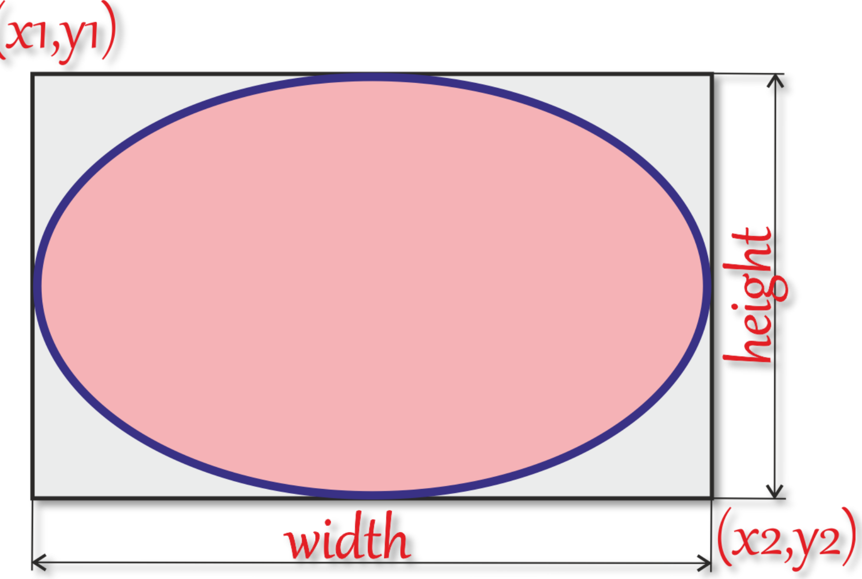


Рис. 17.8. Закрашенный эллипс

На рисунке хорошо видно, что

x2 = x1 + width-1
y2 = y1 + height-1, где

width и *height* – ширина и высота описанного прямоугольника.

Для рисования *незакрашенного* эллипса нам пригодится процедура **DrawEllipse(x1, y1, x2, y2)**. Так как он внутри пустой, то есть имеет цвет фона, то необходимо выделить *контур* эллипса, иначе мы его вообще не увидим. Цвет контура задаётся процедурой *SetPenColor*. Немного изменим код:

```
      SetPenWidth(2);
      . . .
      //рисуем цветную "точку":
      //SetBrushColor(clr);
      //FillEllipse(x, y, x+2*radius, y+2*radius);

      SetPenColor(clr);
      DrawEllipse(x, y, x+2*radius,y+2*radius);
   End;//For
```

И точки-кружочки превращаются в элегантные окружности (Рис. 17.9).

Процедура *SetPenWidth* задаёт толщину пера, то есть контура.

Поскольку точки одного и того же размера вгоняют в тоску, то давайте усеем канву точками всевозможного калибра (Рис. 17.10). Сделать это проще простого. Достаточно радиус кружков задавать случайным образом с помощью функции *Random*:

```
      radius:= Random(10)+4;
      Window.Title:= 'Цвет точки (' + x.ToString() +
               ','  + y.ToString() + ') = ' + clr.ToString();
      //рисуем цветную "точку":
      SetBrushColor(clr);
      FillEllipse(x, y, x+2*radius, y+2*radius);
```

Цвет точки (390,323) = Color [A=255, R=225, G=129, B=15]

Рис. 17.9. Цветные колечки

Цвет точки (180,48) = Color [A=255, R=167, G=121, B=172]

Рис. 17.10. Пёстренько, но со вкусом!

Исходный код программы находится в папке **Многоточие**.

Фильтруем снимки

Вам, без сомнения, не раз попадались на глаза необычные снимки, как бы составленные из цветных квадратиков или точек. Может быть, вы и сами делали такие картинки в *Фотошопе*, который имеет немало *фильтров* для придания фотографиям нового облика. Например, фильтр *Pointilize* (Рис. 17.11) может увеличивать пиксели исходного изображения в несколько раз. В итоге получается «художественное» изображение, похожее на картины, написанные в технике *пуантилизма* (цветными точками) (Рис. 17.12).

Мы не будем замахиваться на достижения искусства или даже всеми нами любимого *Фотошопа,* а напишем простенький, но зато свой фильтр, который превратит любую фотографию в «мозаику».

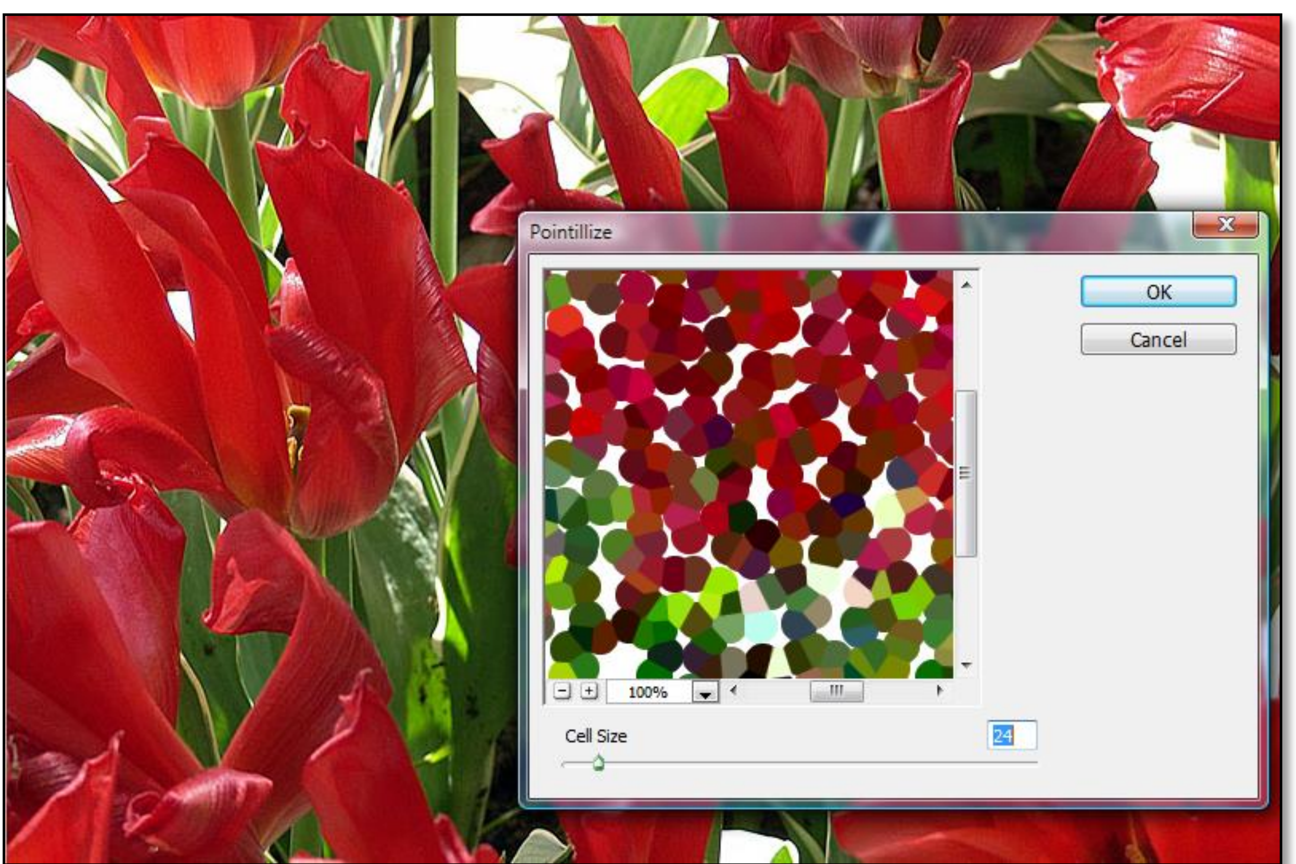


Рис. 17.11. Фильтр *Pointilize* в графическом редакторе *Фотошоп*

Для экспериментов всё-таки лучше взять не *любую* фотографию, а красочную и без мелких деталей, которые неизбежно исчезнут при *пикселизации*. Если вам приходилось видеть картины, вышитые крестиком, то смело берите пример с них.

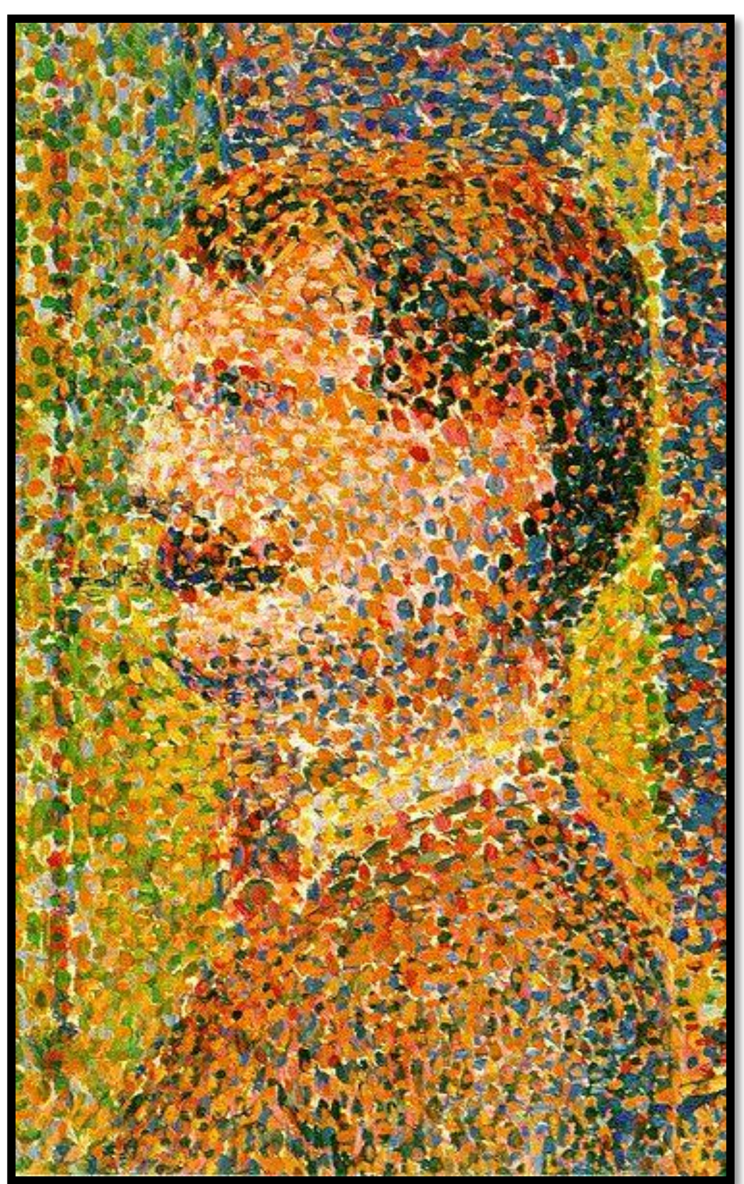


Рис. 17.12. Фрагмент картины Ж. Сёра

За основу проекта мы возьмём исходный код *Пипетки*, который сохраним в папке **Фильтр**.

В начало программы мы добавим переменную *r*, которая у нас будет отвечать за величину точек отфильтрованного изображения, и переменные *height*, *width* для хранения размеров окна. Подгоните их под выбранную вами фотографию. Поместите её в

папку с программой и при запуске программы впечатайте в канву *графического окна* методом *Fill*. В качестве примера я взял фотографию с **красными** тюльпанами, которая в *графическом окне* смотрится совсем неплохо (Рис. 17.13).

```
//ПРОГРАММА ДЛЯ ПИКСЕЛИЗАЦИИ ИЗОБРАЖЕНИЯ

uses
   GraphABC;

var
  //радиус точки:
  r:=5;
  height, width: integer;

begin
   SetWindowTitle('Фильтр');
   SetWindowWidth(800);
   SetWindowHeight(533);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   height := Window.Height;
   width := Window.Width;
   Window.Fill('тюльпаны.jpg');

   //применяем фильтр:
   Filter();
end.
```

Мы напишем *два* фильтра, которые отличаются друг от друга только тем, что в первом точки *квадратные*, а во втором - *круглые*. С точки зрения геометрии, разница небольшая, но в искусстве, как известно мелочей нет, поэтому пробуйте разные варианты, пока не добьётесь совершенства.

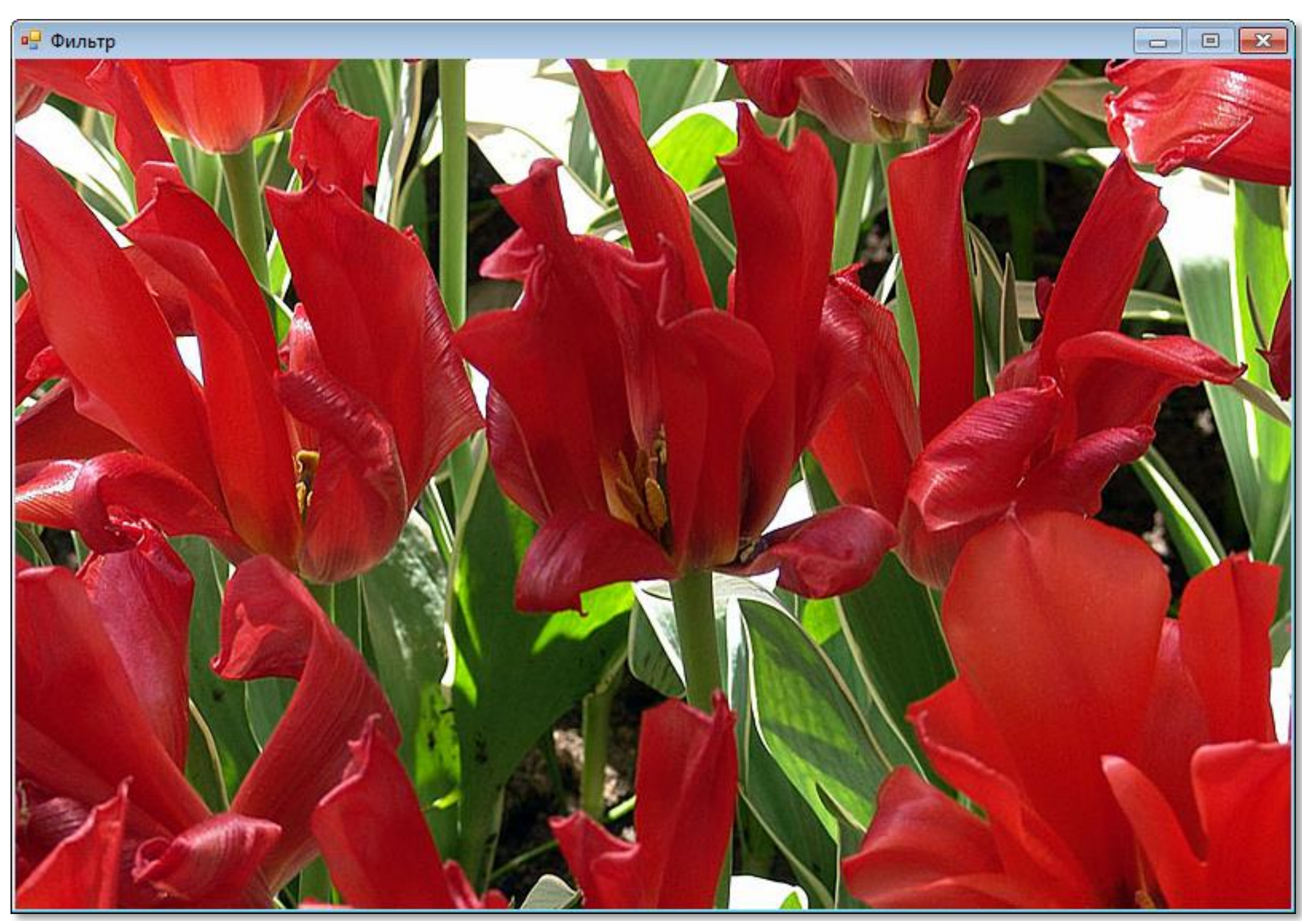


Рис. 17.13. Исходное изображение

Первый фильтр действует очень просто. Мы разбиваем все изображение на квадратики со стороной *l* пикселей и с помощью двух циклов *For* сканируем изображение. Для каждого квадратика мы находим цвет пикселя в его центре, а затем всё изображение внутри квадратика заливаем этим цветом (Рис. 17.14).

```
//Процедура фильтрации изображения
procedure Filter;
begin
  var l:= 2*r;
  For var j:= 1 To height div l do
     For var i:= 1 To width div l do
     begin
        //определяем цвет пикселя в центре точки:
        var xc := (i-1)*l + r;
        var yc := (j-1)*l + r;
        var clr:= GetPixel(xc,yc);
        SetBrushColor(clr);
        //чертим квадрат:
        var x1:= xc-r;
```

```
        var y1:= yc-r;
        FillRectangle(x1, y1, x1+l, y1+l);
    End;//For
End;
```

Рис. 17.14. Отфильтрованные тюльпаны

В результате изображение становится более грубым, как бы состоящим из огромных пикселей. Подобный эффект мы можем наблюдать при большом увеличении фотографий. Правда, в этом случае увеличиваются и размеры картинки, поэтому все пиксели исходного изображения остаются в целости и сохранности, наш же фильтр изменяет цвет большинства пикселей.

Второй фильтр более «изощрённый». Сначала мы запоминаем цвет пикселя в центре квадрата, затем закрашиваем его **чёрным** цветом.

Вы можете вообще не закрашивать квадрат, или закрашивать его другими цветом. Иногда таким простым способом удаётся получить хорошие результаты!

А уж потом, «**по-чёрному**» рисуем круг заданного цвета:

```
//Процедура фильтрации изображения
procedure Filter2;
begin
  var l:= 2*r;
  For var j:= 1 To height div l do
     For var i:= 1 To width div l do
     begin
        //определяем цвет пикселя в центре точки:
        var xc := (i-1)*l + r;
        var yc := (j-1)*l + r;
        var clr:= GetPixel(xc,yc);
        SetBrushColor(Color.Black);
        //чертим квадрат:
        var x1:= xc-r;
        var y1:= yc-r;
        FillRectangle(x1, y1, x1+l, y1+l);
        //рисуем круг:
        SetBrushColor(clr);
        FillEllipse(x1, y1, x1+l, y1+l);
    End;//For
End;
```

Если каждый цветной кружок вышить крестиком (а лучше не полениться и вышить *двойным* крестиком), то получится прекрасная картина (Рис. 17.15). Особенно если к выбору исходного изображения подойти с полной ответственностью и бездной вкуса!

Исходный код программы находится в папке **Фильтр**.

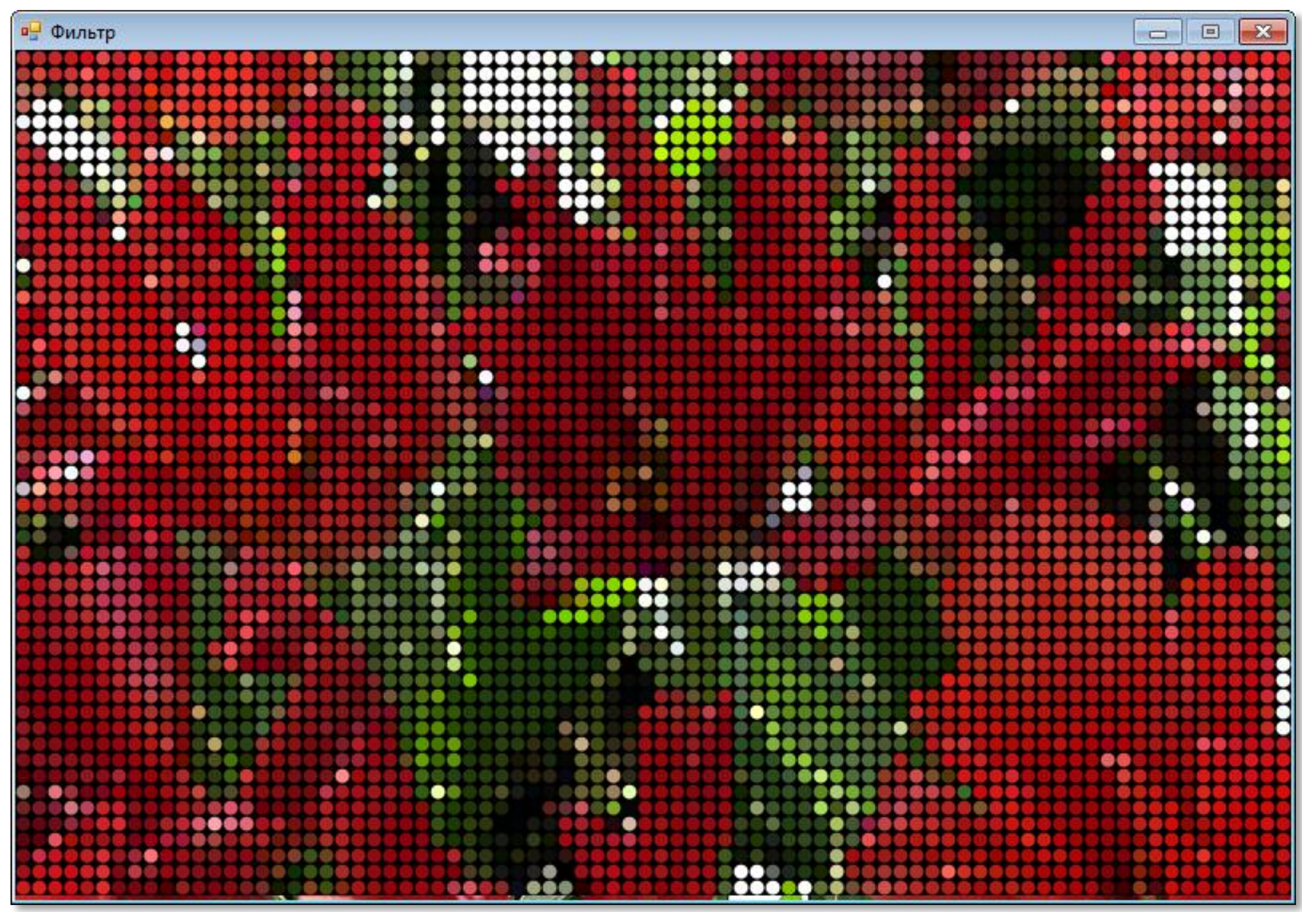


Рис. 17.15. Фильтруем тюльпаны через круглые «дырочки»

1. В программе *Фильтр* мы устанавливали цвет точки по цвету пикселя в её центре. С увеличением размеров точки отфильтрованная картинка всё больше отдаляется от оригинала, поскольку мы не учитываем цвет других пикселей этой точки. Подумайте, как *усреднить* цвет точки в фильтре.

2. Добавьте к программе процедуру рисования *сетки* после применения *квадратного* фильтра (Рис. 17.16):

```
   //применяем фильтр:
   //Filter();
   drawGrid();
   //Filter2();

//Процедура рисования сетки
procedure drawGrid;
begin
```

```
   var l:= 2*r;
   SetPenColor(Color.Black);
   SetPenWidth(1);
   //горизонтали:
   For var j:= 0 To height div l do
       Line(0, j*l, 0+width, j*l);
   //вертикали:
   For var i:= 0 To width div l do
       Line(i*l, 0, i*l, 0+height);
End;
```

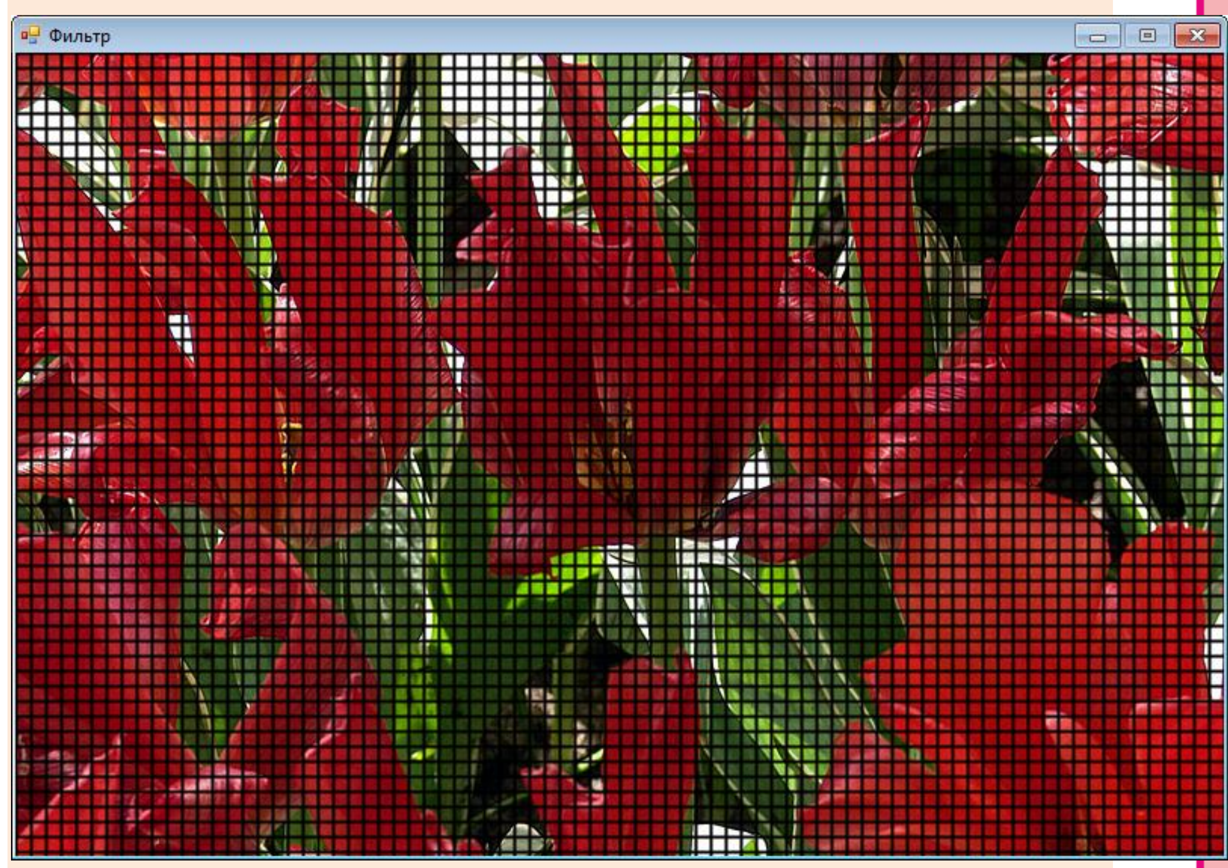


Рис. 17.16. С сеткой мозаика выглядит не в пример лучше!

Урок 18. Полярная система координат

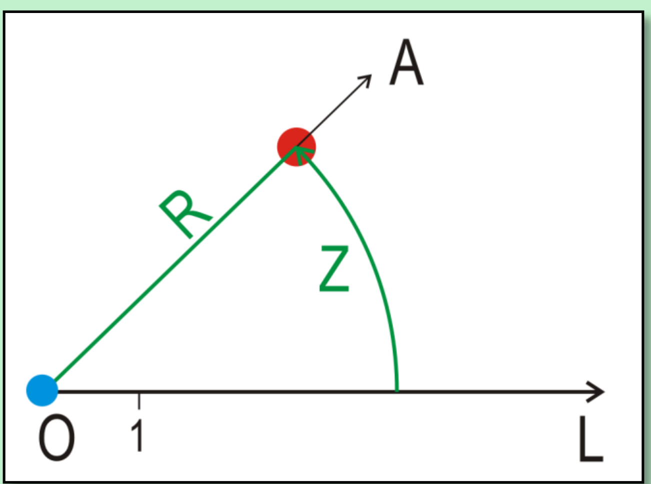
Трутся об ось медведи,-
Вертится Земля…

Песенка из комедии *Кавказская пленница*

Кроме прямоугольной системы координат, которая известна всем и каждому, существуют и другие. Если мы хотим поставить точку на плоскости, нам обязательно понадобятся *два* числа. В декартовой системе координат это абсцисса и ордината, а в **полярной**, которая нам понадобится на этом уроке, - *полярный радиус R* и *полярный угол Z*.

Полярный радиус называется также *радиус-вектором*. Длина радиус-вектора называется *модулем*.

Начало полярных координат находится, как легко догадаться, в *полюсе*. Из него мы можем провести в любом направлении луч, образующий с полярной осью *OL* угол **Z**, который отсчитывается *против* часовой стрелки. Если луч проходит через заданную точку, то её положение в полярной системе координат определяется значениями полярного угла **Z** и полярного радиуса **R**, который равен расстоянию от точки до полюса. Очень хорошо это видно на картинке (Рис. 18.1).



O – полюс
OL – полярная ось
OA – луч, проходящий через заданную точку на плоскости

R – полярный радиус точки
Z – её полярный угол

Рис. 18.1. Полярная система координат

Поскольку координаты пикселей на экране удобнее задавать в прямоугольной системе координат, то давайте выведем формулы, по которым можно пересчитать координаты точки в *полярной* системе в координаты в *прямоугольной* системе. Для этого совместим начало прямоугольной системы координат с полюсом, а ось абсцисс – с полярной осью (Рис. 18.2).

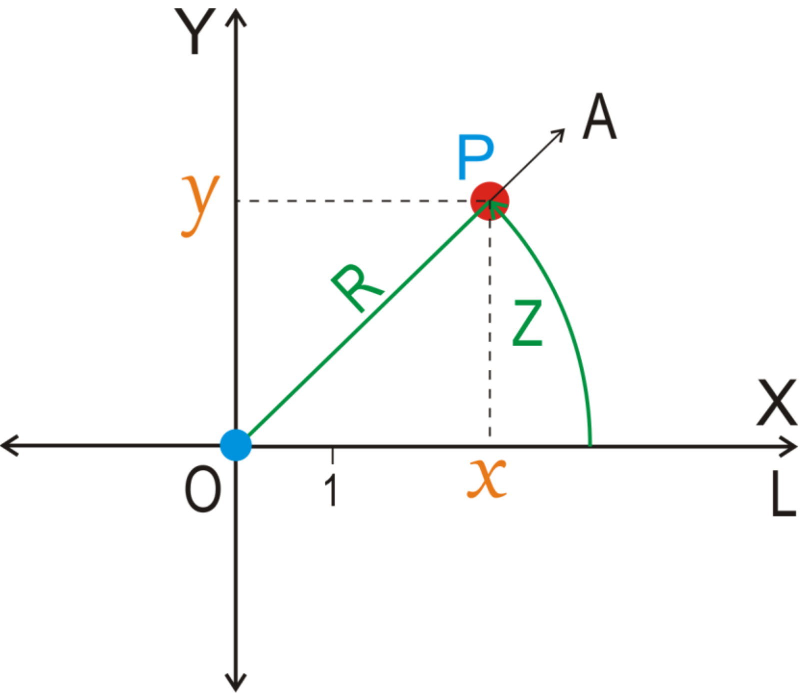


Рис. 18.2. Координаты точки в двух системах координат

Опустим перпендикуляры из точки P на оси абсцисс и ординат. Точки пересечения перпендикуляров обозначим буквами x и y и рассмотрим прямоугольный треугольник OPx, из которого легко найдём координаты точки P:

x = R * cos(Z) (1)
y = R * sin(Z) (2)

Возникает вопрос: если от полярных координат так просто перейти к прямоугольным, то для чего вообще их использовать? – Дело в том, что в полярных координатах некоторые функции задавать гораздо проще, чем в прямоугольных.

Например, *окружности* можно описать такими формулами:

$R = 1$ – для единичной окружности или
$R = r$ – для окружности радиуса r

Те же самые окружности в прямоугольных координатах имеют совсем непростой вид:

$x^2 + y^2 - 2x = 0$ и
$x^2 + y^2 - 2rx = 0$

И многие другие кривые - спирали, улитки, розы, эллипсы, кардиоиды – удобнее задавать именно в полярных координатах. Чтобы убедиться в этом, давайте построим графики нескольких знаменитых кривых, описанных в полярных координатах.

Начните новый проект **Polar** и сохраните его в новой папке.

Для построения графиков (Рис. 18.3) нам потребуются *переменные*, назначение которых понятно из комментариев:

```
//ПРОГРАММА ДЛЯ ПОСТРОЕНИЯ ГРАФИКОВ
//В ПОЛЯРНЫХ КООРДИНАТАХ

uses
   GraphABC, System;

var
   //полярный радиус:
   R := 0.0;
   //полярный угол:
   Z := 0.0;
   //ширина и высота окна:
   width, height: integer;
   //координаты центра окна:
   CX, CY: integer;
   //координаты текущей точки:
   x, y: real;

//
//   ОСНОВНАЯ ПРОГРАММА
//
begin
```

```
   SetWindowTitle('Графики в полярных координатах');
   SetWindowWidth(320);
   SetWindowHeight(320);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;

   height := Window.Height;
   width := Window.Width;
   //координаты центра окна:
   CX := width div 2;
   CY := height div 2;

   // цвет линий:
   SetPenColor(Color.Black);
   SetPenWidth(1);
```

Чтобы построить график, мы изменяем полярный угол *Z* от нуля до *360* градусов (в радианах это – *0 ... 2*Pi*). При дальнейшем увеличении полярного угла новые точки будут просто накладываться на уже существующие, поэтому такое ограничение угла вполне уместно.

Важно задать небольшой *шаг* (*0.01*) изменения угла, чтобы точки графика создавали *цельную* кривую. В некоторых случаях не помогает и мелкий шаг, поэтому мы будем ставить не отдельные точки, а проводить *линии* из каждой последующей точки в предыдущую. Так мы застрахуем себя от разрывов в графике кривой. У самой первой точки нет предыдущей, поэтому мы должны учесть это исключение.

Построение всех кривых оформлено в виде *отдельных* процедур, что, конечно, гораздо удобнее, чем записывать соответствующий код одним куском. В данном примере мы построим кривую, заданную в процедуре *ulitka*. Если мы захотим построить другой график, то просто заменим вызов процедуры *ulitka* каким-нибудь другим.

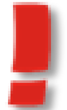
```
  //Строим график:
   //Circle(150);
   //clever(2); //12
   ulitka(1,1); //2,2
```

```
   //spiral(5);
   //star(5);
end.
```

Переменные *x* и *y* хранят координаты очередной точки кривой, пересчитанные из полярных координат в прямоугольные (оконные) по тем формулам (1) и (2), что мы вывели в начале урока. Если хотя бы одна из них выходит за границы канвы, то мы переходим к следующему углу в цикле *while*.

Обратите внимание, что некоторые формулы содержат *параметры*, при изменении которых график приобретает другой вид. Это значит, что с помощью этих формул вы можете начертить гораздо больше красивых кривых, чем показано в книге.

Например, по формуле *R= sin(2*Z)* (Рис 18.3б) будет вычерчен лист клевера, а по формуле *R= sin(12*Z)* (Рис 18.3в) – ромашка. В данном случае можно рассматривать формулу *R= sin(k*Z)* с параметром *k*, который задаёт число лепестков кривой.



Обязательно попробуйте поиграть параметрами или добавить в формулу новые члены, чтобы найти новые кривые!

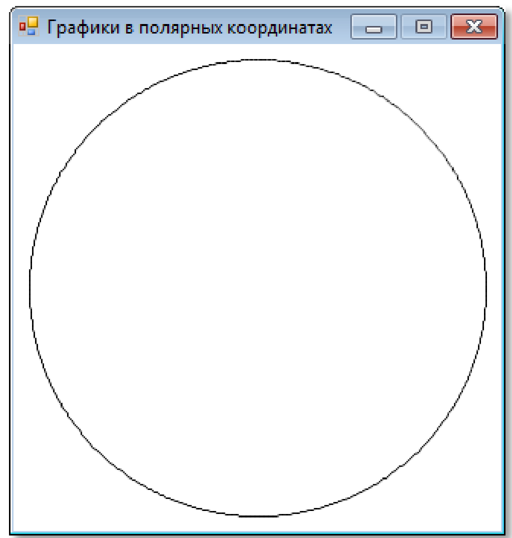
```
//Окружность
procedure circle (r: real);
begin
   //радиус окружности:
   R := r;
   var flg := false;
   var x1, y1: real;
   Z := 0;
   while Z < 2 * Math.Pi + 0.01 do
   begin
      x := CX + Cos(z) * R;
      if (x < 0) Or (x > width) Then
         continue;
      y := CY + Sin(z) * R;
      if (y < 0) Or (y > height) Then
         continue;
      //первая точка:
```

```
      if not flg then begin
         x1 := x;
         y1 := y;
         flg := true;
      end;
      //ставим "точку":
      SetDot(x, y, x1, y1);
      x1 := x;
      y1 := y;
      Z += 0.01;
   end; //while
end;

//Клевер, ромашки
procedure clever(k: integer);
begin
   //var k:=12;
   var x1, y1: real;
   var flg := false;
   Z := 0;

   while Z < 2 * Math.Pi + 0.01 do
   begin
      R := Sin(k * Z) * 160;
      x := CX + Cos(z) * R;
      if (x < 0) Or (x > width) Then
         continue;
      y := CY + Sin(z) * R;
      if (y < 0) Or (y > height) Then
         continue;
      //первая точка:
      if not flg then begin
         x1 := x;
         y1 := y;
         flg := true;
      end;
      //ставим "точку":
      SetDot(x, y, x1, y1);
      x1 := x;
      y1 := y;
      Z += 0.01;
   end; //while
end;
```

```
//Улитка Паскаля
procedure ulitka(k, l: integer);
begin
   var x1, y1: real;
   var flg := false;
   Z := 0;

   while Z < 2 * Math.Pi + 0.01 do
   begin
      //R := (1 + l * Cos(k * Z)) * 50;
      //x := CX + Math.Cos(z) * R;
      //кардиоида k=1, l=1
      R := (1 + l * Cos(k * Z)) * 100;
      x := CX + Math.Cos(z) * R - width/4;
      if (x < 0) Or (x > width) Then
         continue;
      y := CY + Sin(z) * R;
      if (y < 0) Or (y > height) Then
         continue;
      //первая точка:
      if not flg then begin
         x1 := x;
         y1 := y;
         flg := true;
      end;
      //ставим "точку":
      SetDot(x, y, x1, y1);
      x1 := x;
      y1 := y;
      Z += 0.01;
   end; //while
end;

//Спираль
procedure spiral(k: integer);
begin
   var x1, y1: real;
   var flg := false;
   Z := 0;

   while Z < 4 * Math.Pi + 0.01 do
   begin
      R := Z / k * 100;
      x := CX + Math.Cos(z) * R + 20;
```

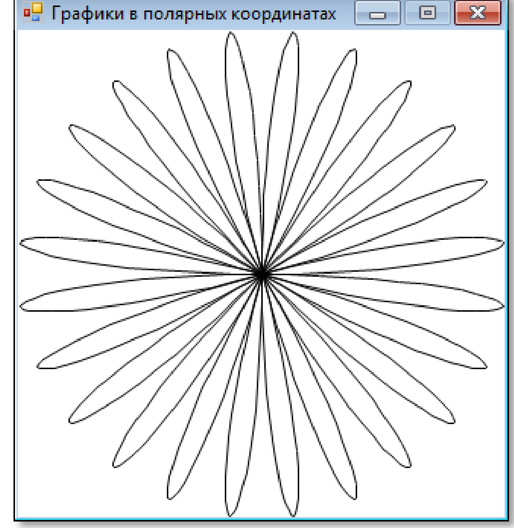
```
      if (x < 0) Or (x > width) Then
         continue;
      y := CY + Math.Sin(z) * R - 30;
      if (y < 0) Or (y > height) Then
         continue;
      //первая точка:
      if not flg then begin
         x1 := x;
         y1 := y;
         flg := true;
      end;
      //ставим "точку":
      SetDot(x, y, x1, y1);
      x1 := x;
      y1 := y;
      Z += 0.01;
   end; //while
end;

//Звёздочка
procedure star(k: integer);
begin
   for var i := 0 to 7 do
   begin
      var x1, y1: real;
      var flg := false;
      Z := 0;
      while Z < 2 * Math.Pi + 0.01 do
      begin
         var F := Z + i * Math.Pi * 2;
        R := 60 / (2 + Math.Cos(F * k + ((Math.Floor(0.1 * F) / 5 -
              Math.Floor(Math.Floor(0.1 * F) / 5))))) * 2.5;
         x := CX + Math.Cos(F) * R;
         if (x < 0) Or (x > width) Then
            continue;
         y := CY + Math.Sin(F) * R;
         if (y < 0) Or (y > height) Then
            continue;
         //первая точка:
         if not flg then begin
            x1 := x;
            y1 := y;
            flg := true;
         end;
```

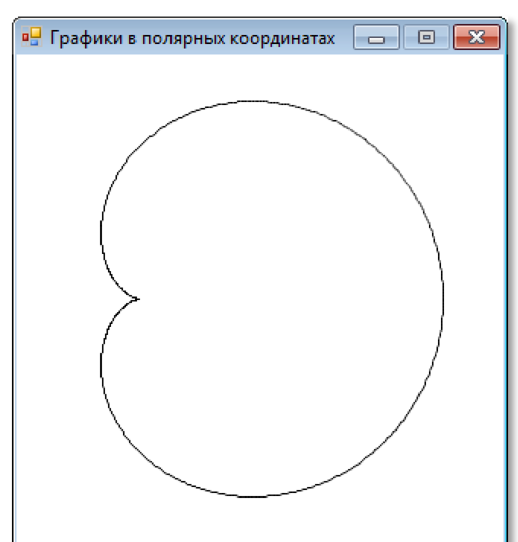
```
            //ставим "точку":
            SetDot(x, y, x1, y1);
            x1 := x;
            y1 := y;
            Z += 0.01;
        end; //while
    end; //for
end;
```



а) R = 1 - окружность

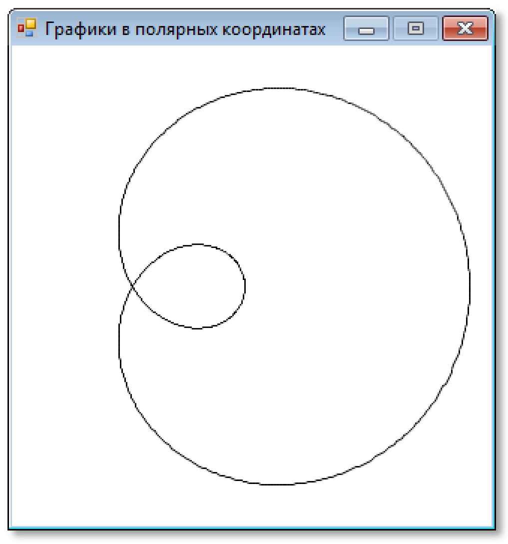
б) R= sin(2*Z) - клевер

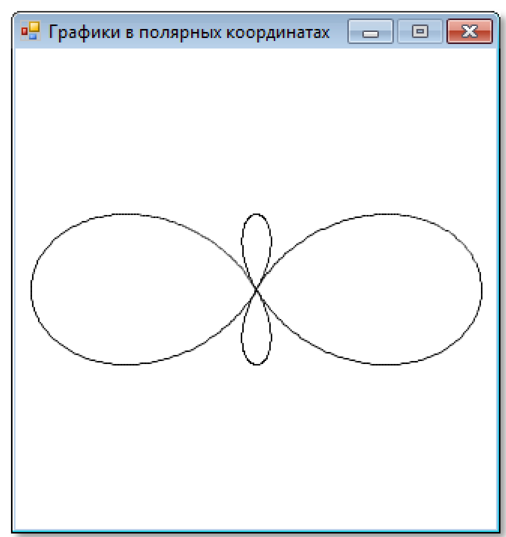




в) R= sin(12*Z) - ромашка

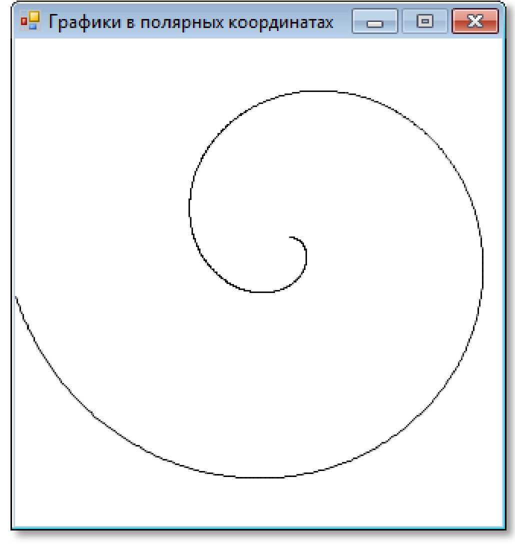
г) R= 1 + cos(Z) - кардиоида

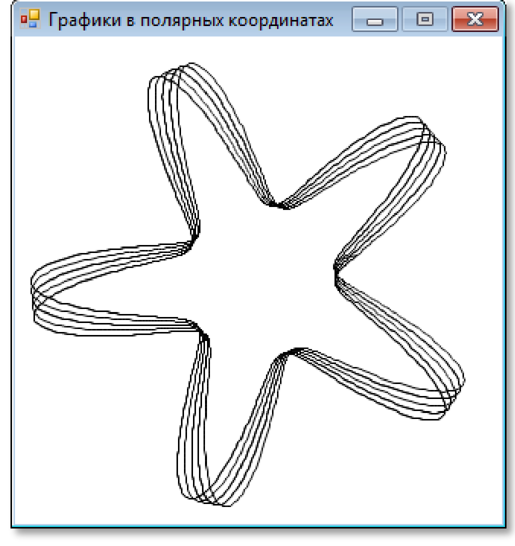




д) R= 1 + 2*cos(Z) – улитка Паскаля

е) R= 1 + 2*cos(2*Z) – петельное сцепление





ж) R= Z/k – спираль

з) Звёздочка

Рис. 18.3. Графики кривых в полярных координатах

Чтобы не приводить всякий раз вещественный тип координат к целому, который необходим функции *Line*, мы напишем короткую вспомогательную процедуру:

```
//СТАВИМ ТОЧКУ
procedure SetDot(x1, y1, x2, y2: real);
begin
   Line(Round(x1), Round(y1), Round(x2), Round(y2));
end;
```

Исходный код программы находится в папке **Polar**.

1. Установите на форме *кнопки* для выбора нужной кривой.

Кнопки мы хорошенько изучим на уроке *Элементы управления*.

2. Поэкспериментируйте с различными значениями параметров, которые входят формулы. Так, например, можно получить ромашки с разным числом лепестков или улитки Паскаля разной формы.

3. Добавьте к программе возможность автоматического масштабирования графиков под размеры клиентской области окна приложения. Для этого перед началом построения графиков следует найти максимальные значения координат *x* и *y* и вычислить коэффициент масштабирования, на который затем умножить все значения текущих координат.

Урок 19. Занимательные игры с пикселями

Случайные точки, которыми мы «баловались» на уроке *Компьютерная графика*, создают хаотичный «узор», что не очень хорошо для уроков геометрии, поэтому теперь мы будем окрашивать пиксели по строгим математическим формулам. Они могут быть и довольно простыми, но узоры при этом давать расчудесные!

Синусоидные полоски

Начнём наш новый проект и сохраним его в папке **1D sine**. Рисунок здесь будут создавать горизонтальные строки одинаково окрашенных пикселей (проще говоря, линии, которые мы рисуем отдельными точками). Цвет пикселей изменяется по высоте клиентской области окна сверху вниз по совсем простой формуле:

```
var clr:= 255*(1 + Sin(y/wl))/2;
```
(1)

Поскольку в формуле присутствует синус угла, то и проект называется *Синусные полоски*. Частота горизонтальных волн зависит от длины волны *wl*, так что вы легко получите разные картинки, если измените значение этого параметра. Нам осталось узнать о назначении множителя 255 в формуле (1). Дело в том, что выражение в скобках изменяется в диапазоне 0..2, а делённое на два – в диапазоне 0..1. Цветные составляющие пикселя должны иметь значение от 0 до 255, откуда и вытекает необходимость введения в формулу этого множителя. Цвет очередного пикселя печатается в заголовке окна. Если вам эта информация не нужна, закомментируйте строчку

```
Window.Title:= 'Цвет пикселя (y=' + y.ToString() +') = '
                              + color.ToString();
```

Остальная часть кода не должна вызвать у вас никаких вопросов:

```
//Синусоидные полоски
```

```
uses GraphABC;
begin
   SetWindowTitle('1D sine');
   SetWindowWidth(480);
   SetWindowHeight(240);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.Black);

   var height: integer:= Window.Height;
   var width: integer:= Window.Width;
   //длина волны синусоиды:
   var wl:= 10;
   for var y:=0 to height-1 do
   begin
      //цвет очередного пикселя:
      var clr:= 255*(1 + Sin(y/wl))/2;
      //серые полоски:
      var color:= RGB(Floor(clr),Floor(clr), Floor(clr));
      Window.Title:= 'Цвет пикселя (y=' + y.ToString() +') = '
                        + color.ToString();
      for var x:=0 to width-1 do
      begin
         //окрашиваем его:
         SetPixel(x,y, color);
      end
   end //For
end.
```

Картинка у нас получилась славная, но чёрно-белая (Рис. 19.1).

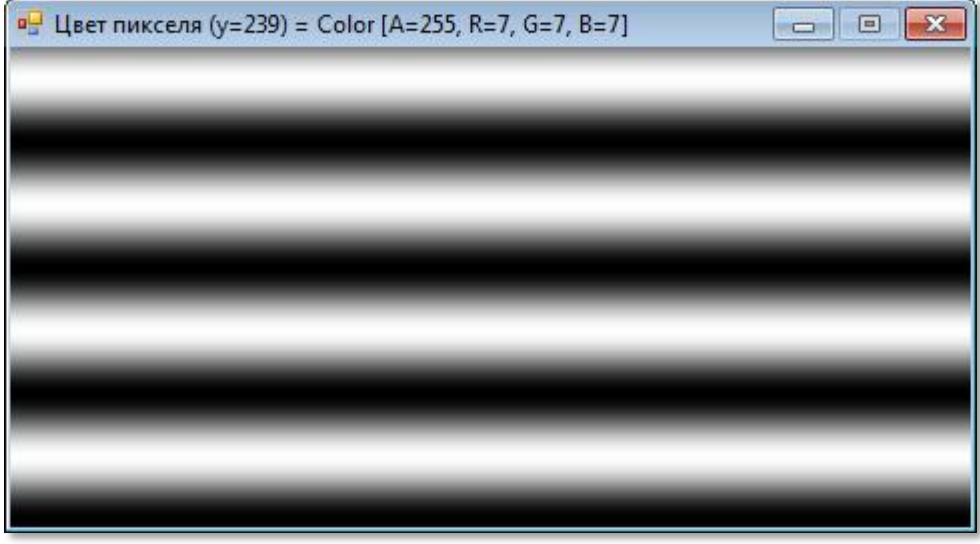


Рис. 19.1. Синусоидные полоски

Но мы без труда окрасим волны в нужный цвет, слегка изменив параметры в функции *RGB* (Рис. 19.2 и 19.3).

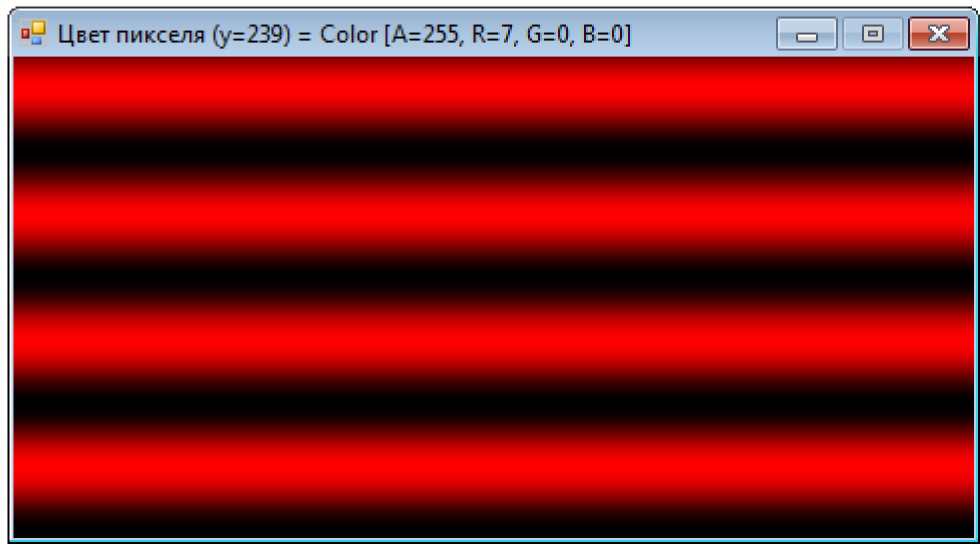


Рис. 19.2. `RGB(Floor(clr),0,0)`

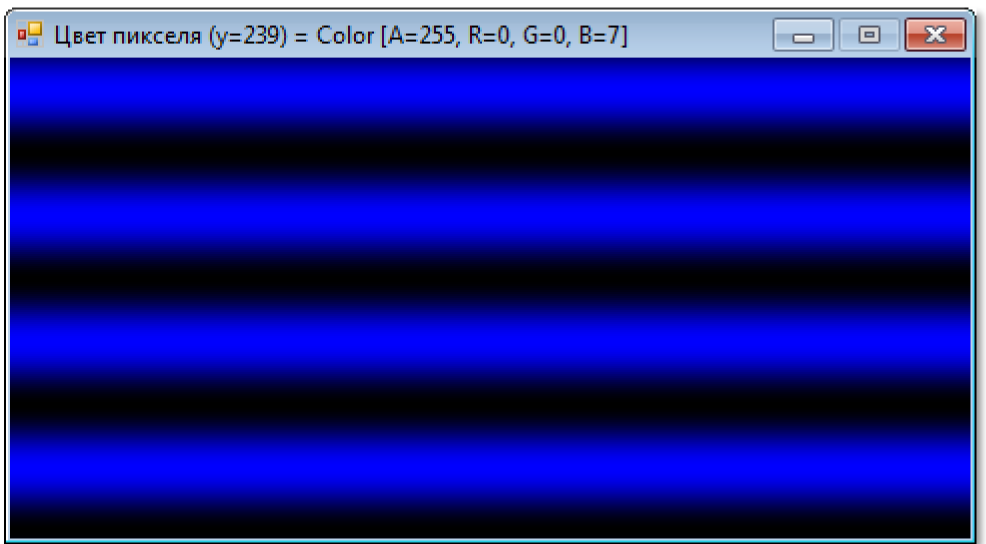


Рис. 19.3. RGB(0,0, Floor(clr))

Цветные волны ещё лучше!

Исходный код программы находится в папке **1D sine**.

Двойная волна

А теперь давайте пустим *две* волны – вертикальную и горизонтальную:

```
  //длина горизонтальной волны:
   var wX:= 20;
   //длина вертикальной волны:
   var wY:= 20;
```

Для этого в формулу для вычисления цвета пикселя добавим ещё один синус:

```
var clr:= 255*(1+Sin(x/wX)*Sin(y/wY))/2;
```

Остальная часть программы ничуть не отличается от предыдущей:

```
 for var y:=0 to height-1 do
  begin
     for var x:=0 to width-1 do
     begin
        //цвет очередного пикселя:
        var clr:= 255*(1+Sin(x/wX)*Sin(y/wY))/2;
        var color:= RGB(0,Floor(clr), 100);
        //окрашиваем его:
        SetPixel(x,y, color);
     end
  end //For
```

Получилась интересная *сетчатая* структура (Рис. 19.4).

Исходный код программы находится в папке **2D sine**.

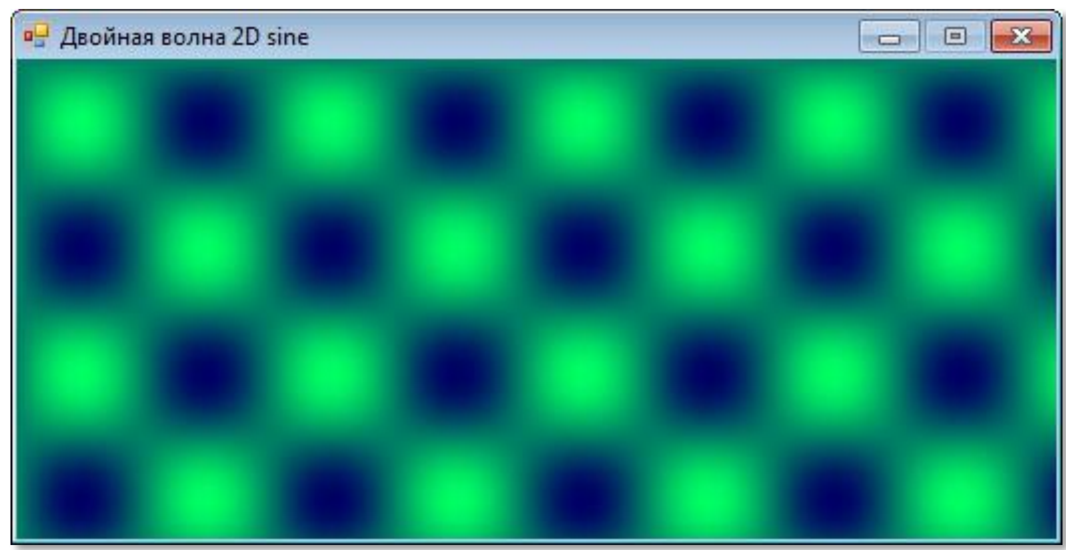


Рис. 19.4. Двойные волны

Лунки

В следующем проекте мы так изменим формулу для вычисления цвета пикселей, чтобы в ней участвовала *абсолютная величина* синусов:

```
//цвет очередного пикселя:
var clr:= 255*(1+ Abs(Sin(x/wX)) * Abs(Sin(y/wY)))/2;
```

Это приведет к тому, что все лунки приобретут *одинаковый* цвет (Рис. 19.5), в отличие от проекта *Двойные волны*, где лунки окрашивались в *разные* цвета.

```
 for var y:=0 to height-1 do
  begin
     for var x:=0 to width-1 do
     begin
        //цвет очередного пикселя:
        var clr:= 255*(1+ Abs(Sin(x/wX)) * Abs(Sin(y/wY)))/2;
        var color:= RGB(Floor(clr), 0, Floor(clr));
        //окрашиваем его:
        SetPixel(x,y, color);
     end
  end //For
```

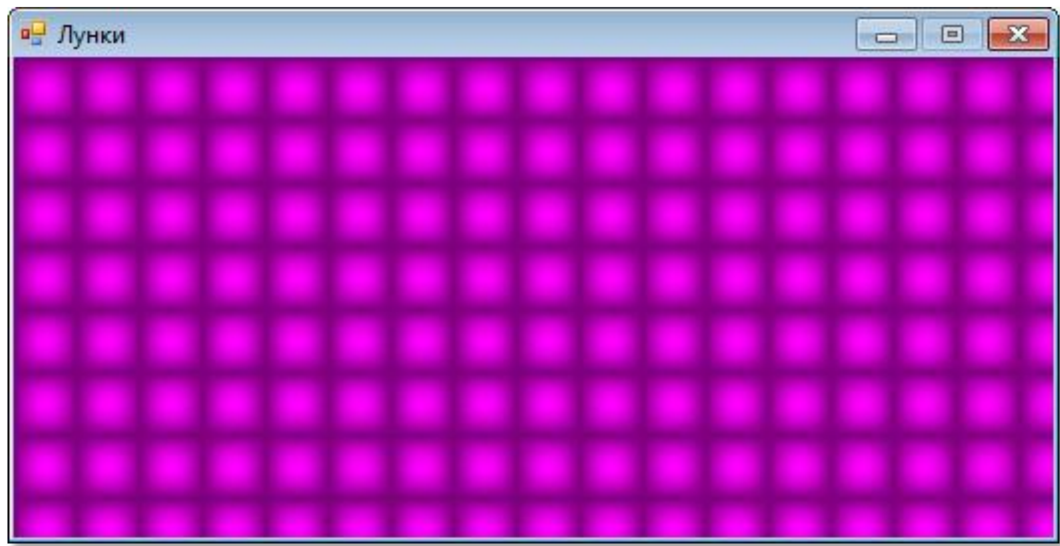


Рис. 19.5. Лунки

Исходный код программы находится в папке **Abs sine**.

Радиальные волны

Бросая в воду камешки, смотри на круги,
ими образуемые; иначе такое бросание
будет пустою забавою.

Козьма Прутков

Если вы бросали камни в воду, то, конечно, обратили внимание на то, что волны, которые разбегаются от места падения камня, имеют вовсе не форму прямых линий или лунок - они совершенно *круглые*. Чтобы загнуть волны в дугу, нам придётся перейти от прямоугольных координат к полярным, что мы и сделаем с помощью формулы:

```
//цвет очередного пикселя:
var rad:= sqrt((x-CX)*(x-CX) + (y-CY) * (y-CY)) / wl;
```

Сама же формула для цвета пикселя останется без изменений:

```
var clr:= 255*(1 + Sin(rad))/2;
```

Исходный код программы очень похож на наши прежние проекты:

```
//РАДИАЛЬНЫЕ ВОЛНЫ

uses GraphABC;

begin
   SetWindowTitle('Радиальные волны');
   SetWindowWidth(520);
   SetWindowHeight(520);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.Black);

   var height: integer:= Window.Height;
   var width: integer:= Window.Width;

   //координаты центра волн:
   var CX:= width div 2;
   var CY:= height div 2;
   //длина волны:
   var wl:= 6;

   for var y:=0 to height-1 do
   begin
      for var x:=0 to width-1 do
      begin
         //цвет очередного пикселя:
         var rad:= sqrt((x-CX)*(x-CX) + (y-CY) * (y-CY)) / wl;
         var clr:= 255*(1 + Sin(rad))/2;
         var color:= RGB(0, 0, Floor(clr));
         //окрашиваем его:
         SetPixel(x,y, color);
      end
   end //For
end.
```

Зато волны получились – как настоящие (Рис. 19.6)!

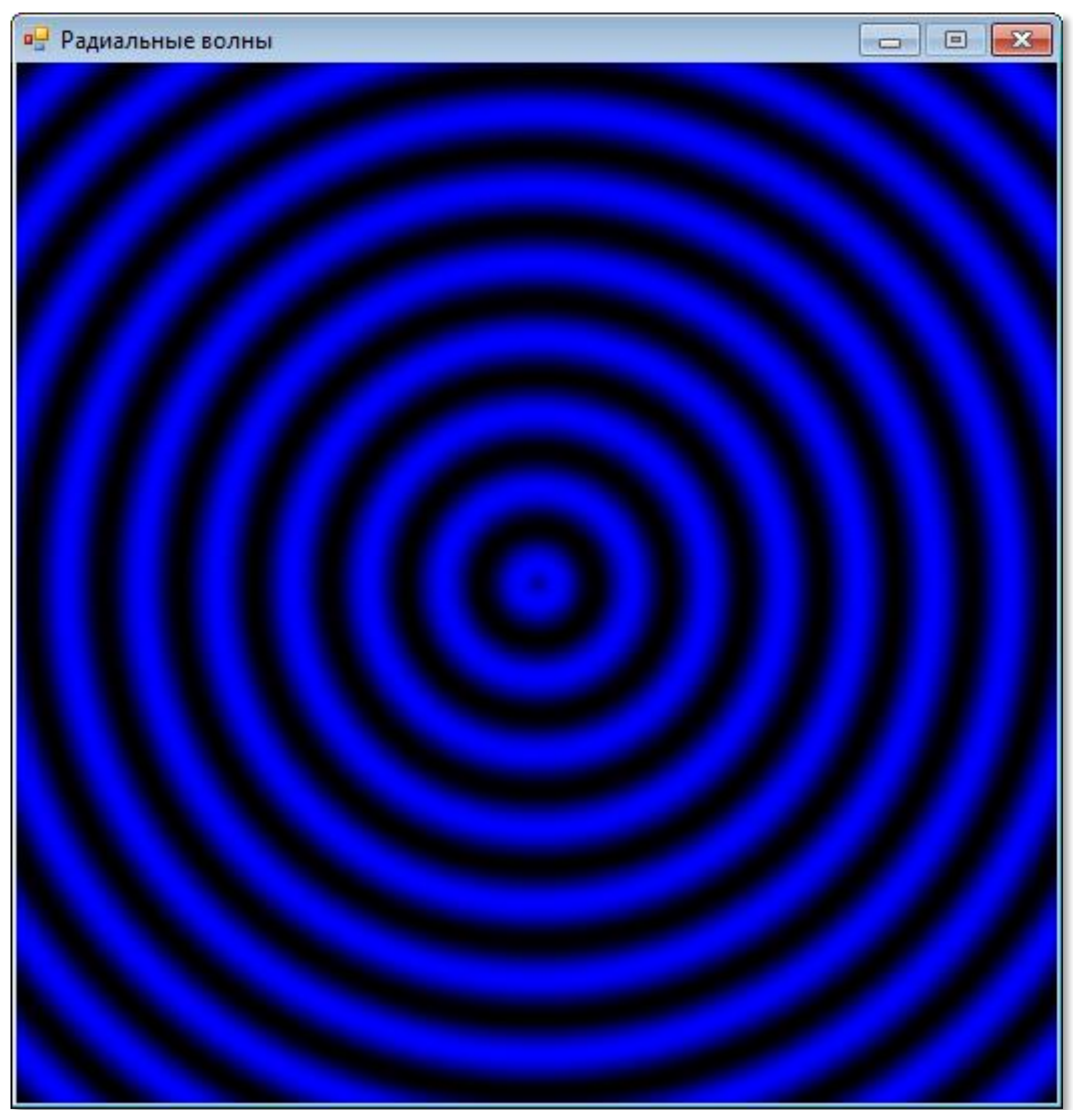


Рис. 19.6. «Полярные» волны!

Исходный код программы находится в папке **Polar sine**.

Ромбы

Применив для пикселестроения более хитроумную формулу, мы получим великолепный *ромбический узор*, от которого глаза трудно оторвать (Рис. 19.7).

```
//РОМБЫ
uses GraphABC;
```

```
begin
   SetWindowTitle('Ромбы');
   SetWindowWidth(520);
   SetWindowHeight(480);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.Black);

   var height: integer:= Window.Height;
   var width: integer:= Window.Width;

   //координаты центра волн:
   var CX:= width div 2;
   var CY:= height div 2;
   //длина волны:
   var wl:= 60;

   for var y:=0 to height-1 do
   begin
      for var x:=0 to width-1 do
      begin
         //цвет очередного пикселя:
         var clr:= 255*2*(Abs((x mod wl) -wl/2) +
                          Abs((y mod wl) -wl/2))/wl;
         var color:= RGB(Floor(clr), 0, Floor(clr));
         //окрашиваем его:
         SetPixel(x,y, color);
      end
   end //For
end.
```

Исходный код программы находится в папке **Ромбы**.

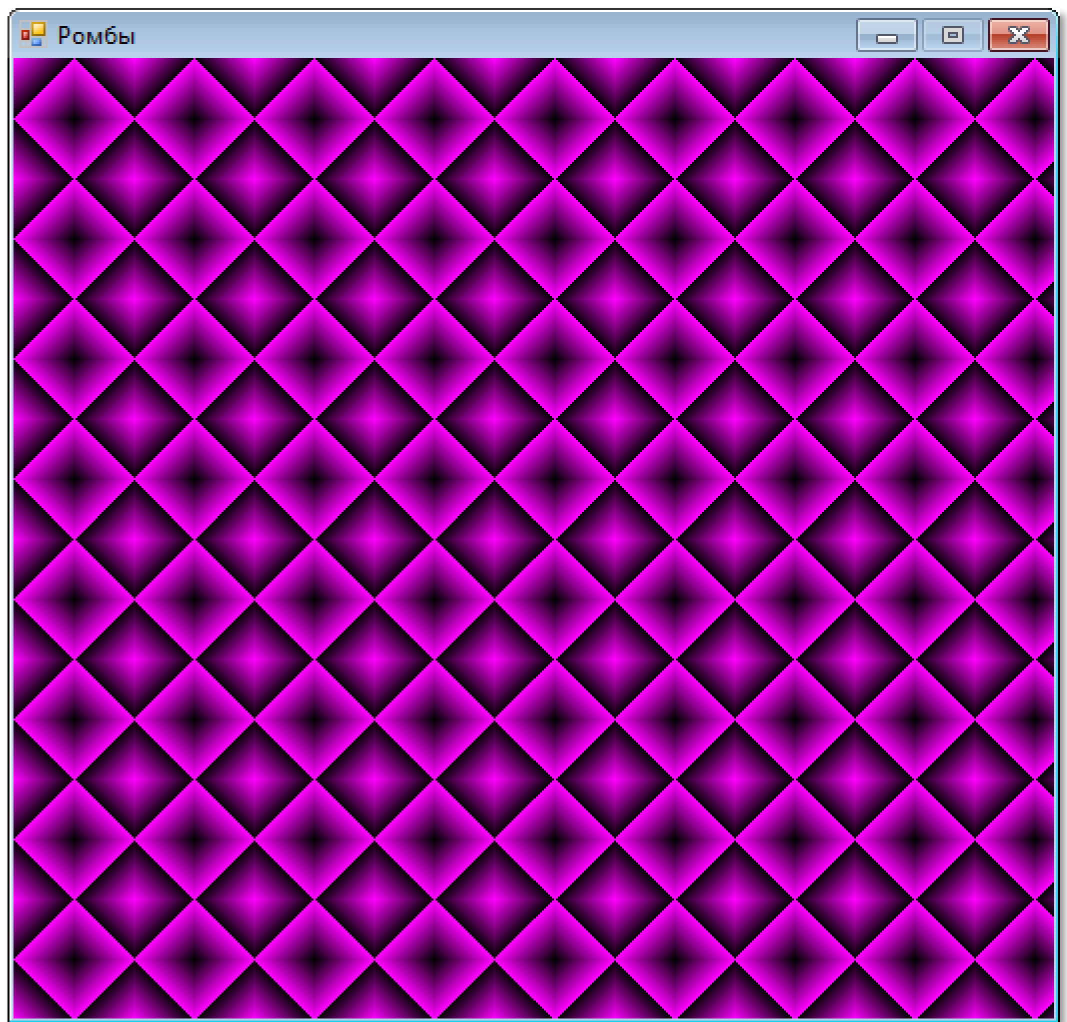


Рис. 19.7. Канва в ромбик

Синусоиды Винни-Пуха

По неведомой причине неизвестный автор этих синусоид (Рис. 19.8) посвятил их нашему любимому Винни, прославившемуся своим пыхтеньем (следствие непомерного обжорства).

В этом примере 9 параметров, входящих в формулу для вычисления цвета пикселя, выбираются случайно в начале программы:

```
//СИНУСОИДЫ ВИННИ-ПУХА

uses GraphABC;

begin
   SetWindowTitle('Синусоиды Винни-Пуха');
```

```
   SetWindowWidth(520);
   SetWindowHeight(520);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.Black);

   var height:= Window.Height;
   var width:= Window.Width;

   //длина волны:
   var wX:= 16; //10
   var wY:= 16; //10
   var a: array[0..9] of integer;

   For var i:= 0 To 9 do
      a[i]:= Random(2)+1;
```

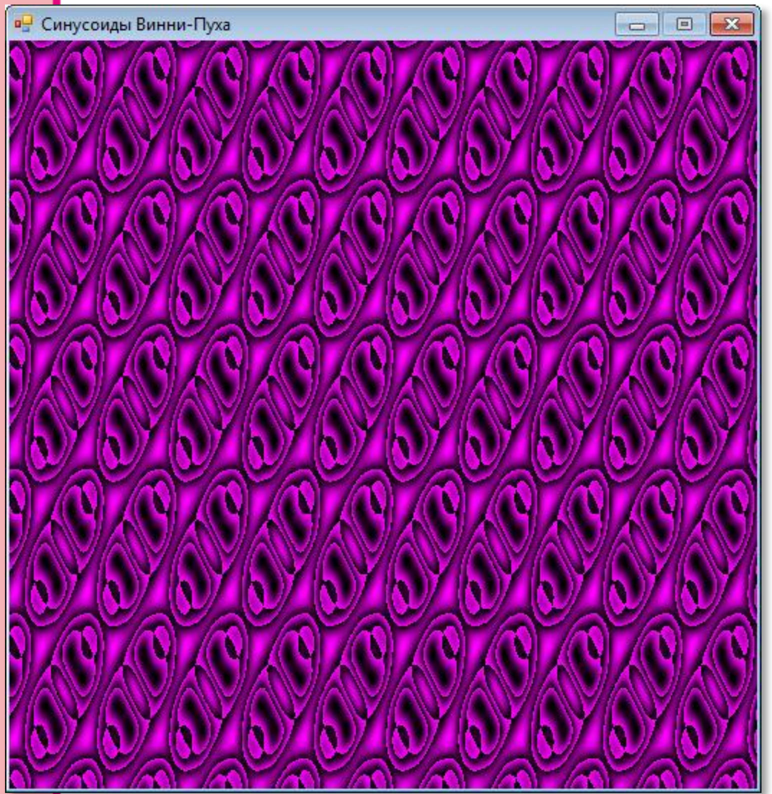
Это значит, что при каждом новом запуске программы вы будете получать *новую* картинку. Вот где простор для поиска новых узоров!

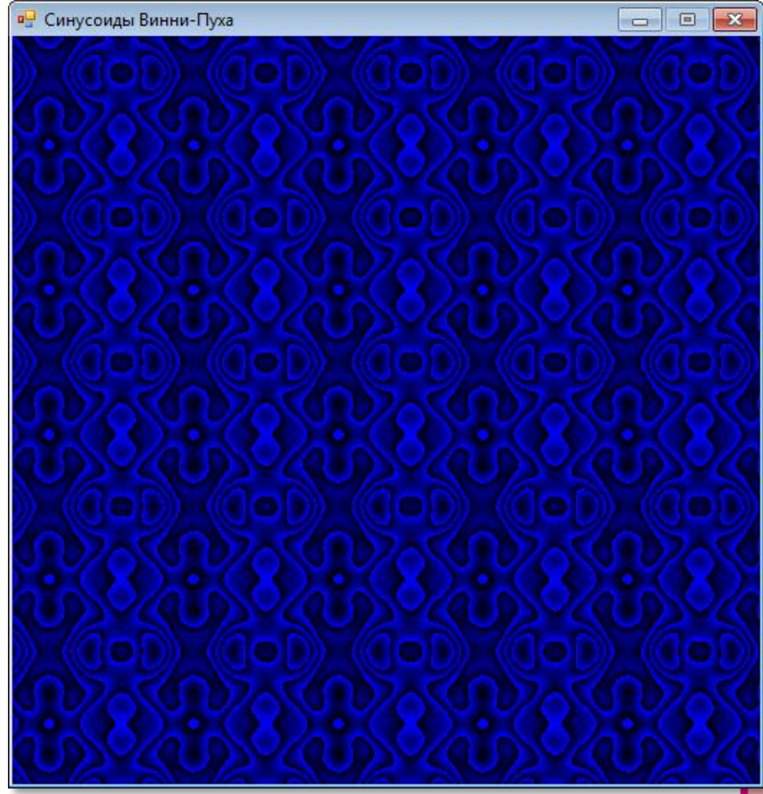
С другой стороны, *случайный поиск* не очень эффективен. Может быть, стоит попробовать подбирать коэффициенты вручную?

Формула для вычисления цвета пикселей просто ужасная, но компьютер мы ею не напугаем:

```
 for var y:=0 to height-1 do
   begin
      for var x:=0 to width-1 do
      begin
         //цвет очередного пикселя:
         var clr:= 256*(1 + (a[4]*Sin(a[0] * Sin(a[6] * x/wX) +
                            a[1]*Cos(a[7] * y/wY)) +
                            a[5]*Cos(a[2] * Cos(a[8] * x/wX) +
                            a[3]*Sin(a[9] * y/wY)))) /2;
         //жёлтая:
         var color:= RGB(Floor(clr), Floor(clr), 0);
         //синяя:
         //var color:= RGB(0, 0, Floor(clr));
         //красная:
         //var color:= RGB(Floor(clr),0, 0);
```

```
            //красная:
            //var color:= RGB(Floor(clr), 0, Floor(clr));
            //окрашиваем его:
            SetPixel(x,y, color);
        end
    end //For
end.
```





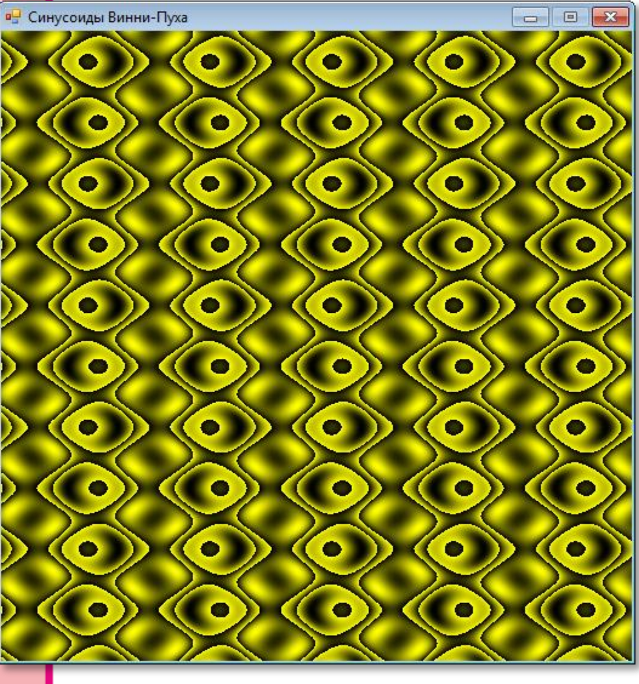


Рис. 19.8. Так как параметры для вычисления цвета пикселей выбираются случайно, то каждый раз вы будете получать другую картинку!

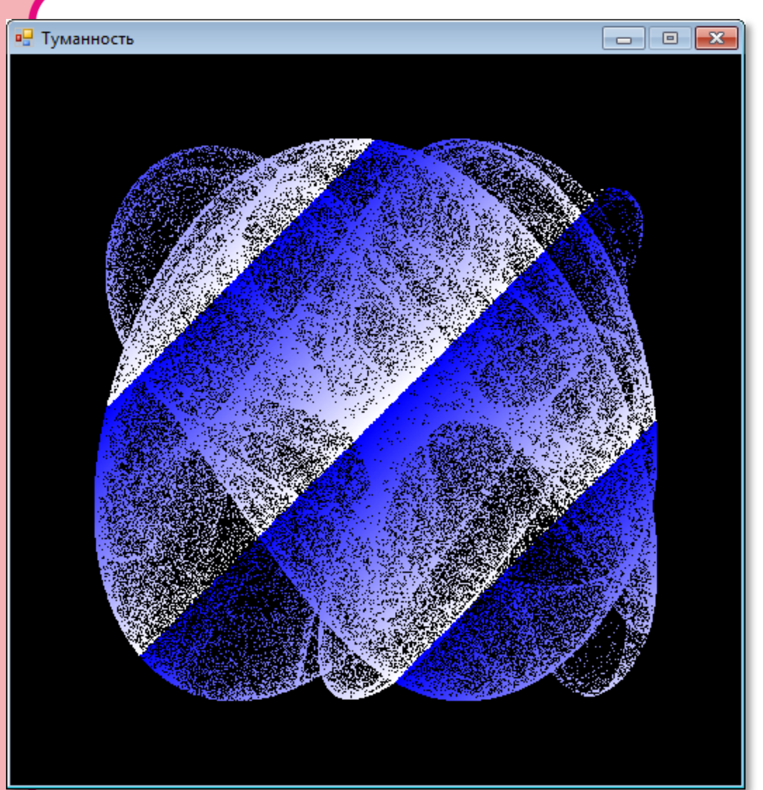
Исходный код программы находится в папке **Sine the pooh**.

Туманность

Аналогично, выбирая случайные параметры для формулы в этом примере, мы получим причудливые линии, похожие на туманности (Рис. 19.9).

```
//ТУМАННОСТИ

uses GraphABC;

begin
   SetWindowTitle('Туманность');
   . . .
   var a: array[1..6] of real;
   For var i:= 1 To 6 do
      a[i]:= PI * (1-2*Random(1000)/1000);

   While (True) do
   begin
     var x := Sin(a[1]*a[6]) - Cos(a[2]*a[5]);
     var y := Sin(a[3]*a[5]) - Cos(a[4]*a[6]);
     //цвет очередного пикселя:
     var color:= RGB(floor(128*(x+y)), floor(128*(x+y)),255);
         //окрашиваем его:
         SetPixel(floor(CX+x*100), floor(CY+y*100), color);
         a[5] := x;
         a[6] := y;
   end //while
end.
```

Исходный код программы находится в папке **Туманность.**



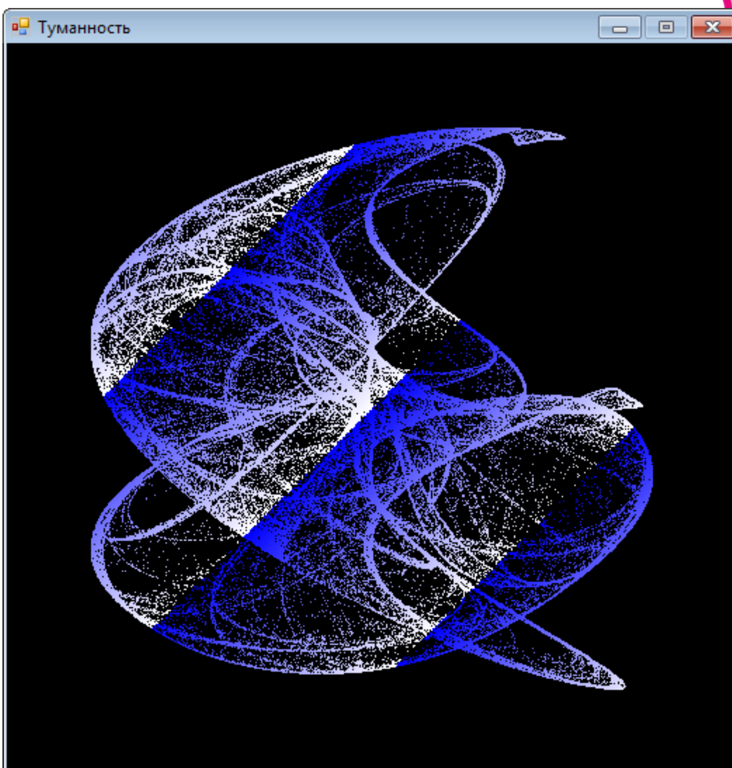


Рис. 19.9. Туманности

Урок 20. Занимательная прямолинейность

Если взять один кирпич, мало толку в нём,
Потому что из него не построишь дом.
Если пару кирпичей рядом положить,
Будет только две стены – неудобно жить.

Песенка Тыквы из мультфильма *Чиполлино*

Одна точка, как мы убедились на уроке Компьютерная графика, - унылое зрелище. А вот через *две* точки уже можно провести прямую, поэтому попробуем теперь порисовать *линиями*.

Для этого в модуле *GraphABC* припасена процедура

Line(x1, y1, x2, y2);

Точка с координатами *(x1, y1)* задаёт *начало* прямой, а точка *(x2, y2)* – её *конец*, то есть будет правильнее сказать, что эта процедура вычерчивает *отрезок* прямой (Рис. 20.1).

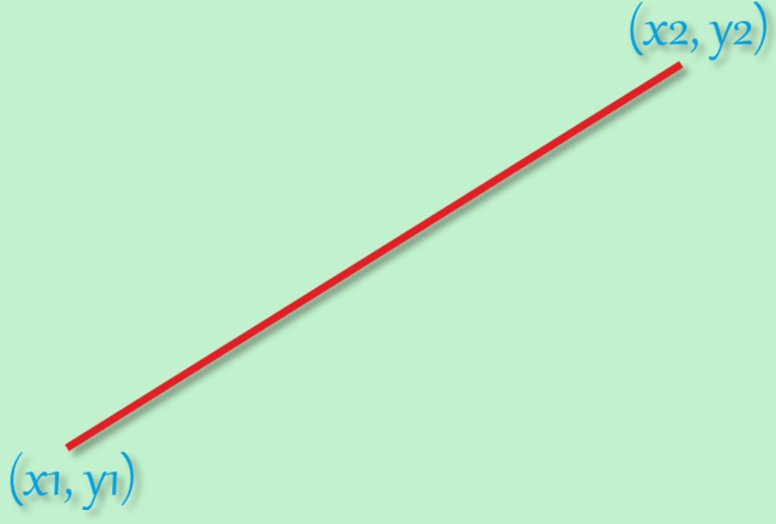


Рис. 20.1. Прямая линия

Толщина линии устанавливается процедурой *SetPenWidth*, а её *цвет* – процедурой *SetPenColor.*

Поскольку для рисования линий достаточно взять две точки, то мы начнём новый проект **Случайные линии** с того, что загрузим в *ИСР* исходный код программы *Pixel* и сохраним его в новой папке.

Сама программа будет совершенно бесхитростной, поскольку рисовать прямые линии очень просто. Нам потребуется всего одна *переменная* – для хранения толщины линий:

```
//ПРОГРАММА ДЛЯ ВЫЧЕРЧИВАНИЯ
//СЛУЧАЙНЫХ ЦВЕТНЫХ ЛИНИЙ
uses
   GraphABC;

var
   //толщина линий:
   penWidth := 2;
//
//   ОСНОВНАЯ ПРОГРАММА
//
begin
   SetWindowTitle('Случайные линии');
   SetWindowWidth(480);
   SetWindowHeight(320);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.Black);
   var height := Window.Height;
   var width := Window.Width;
```

Вместо координат одной точки, как при рисовании пикселей, нам понадобятся координаты *двух* точек, которые мы затем и соединим прямой линией, вызвав процедуру *Line* с соответствующими параметрами:

```
  //толщина линий:
   SetPenWidth(penWidth);

   //Чертим цветные линии
   while(True) do
   begin
      //задаем случайные координаты начала и конца линии -->
```

```
      //координаты первой точки:
      var x1:= Random(width - penWidth div 2);
      var y1 := Random(height - penWidth div 2);
      //координаты второй точки:
      var x2 := Random(width - penWidth div 2);
      var y2 := Random(height - penWidth div 2);
      //выбираем случайный цвет линии:
      var clr := clRandom();
      SetPenColor(clr);
      //проводим линию:
      Line(x1, y1, x2, y2);
      Sleep(30);
   end;//While
end.
```

Запускаем программу – и тут уж ничего не добавишь: картина куда краше, чем с точками (Рис. 20.2). А мало этого, увеличьте толщину линий до 20 – и смотрите сами (Рис. 20.3):

```
penWidth := 20;
```

Если линии очень **толстые**, то они больше напоминают прямоугольники.

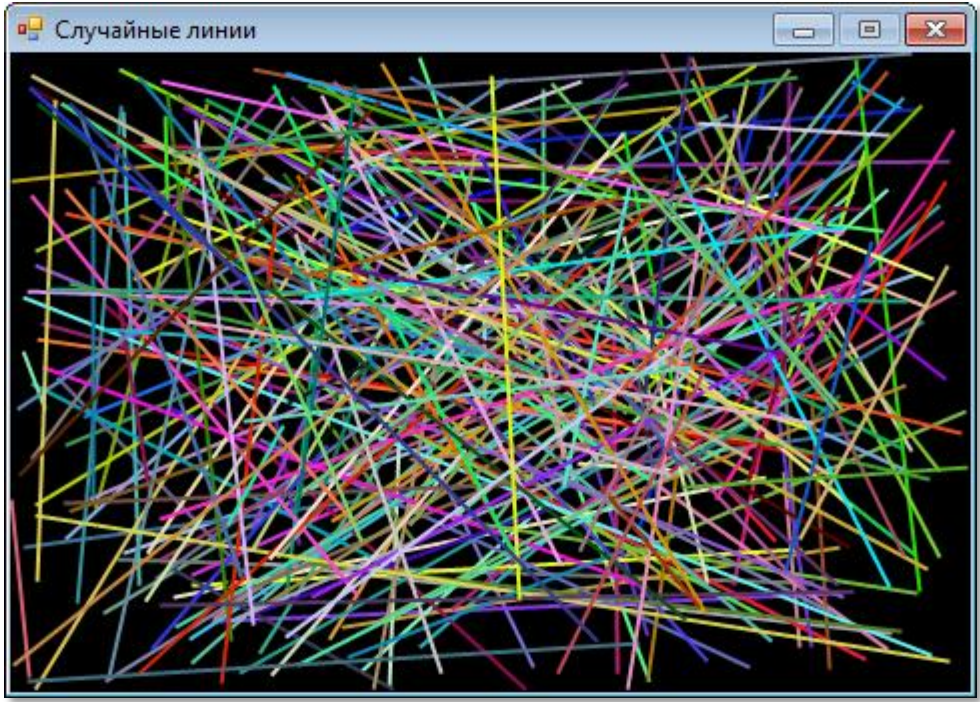


Рис. 20.2. Случайные линии

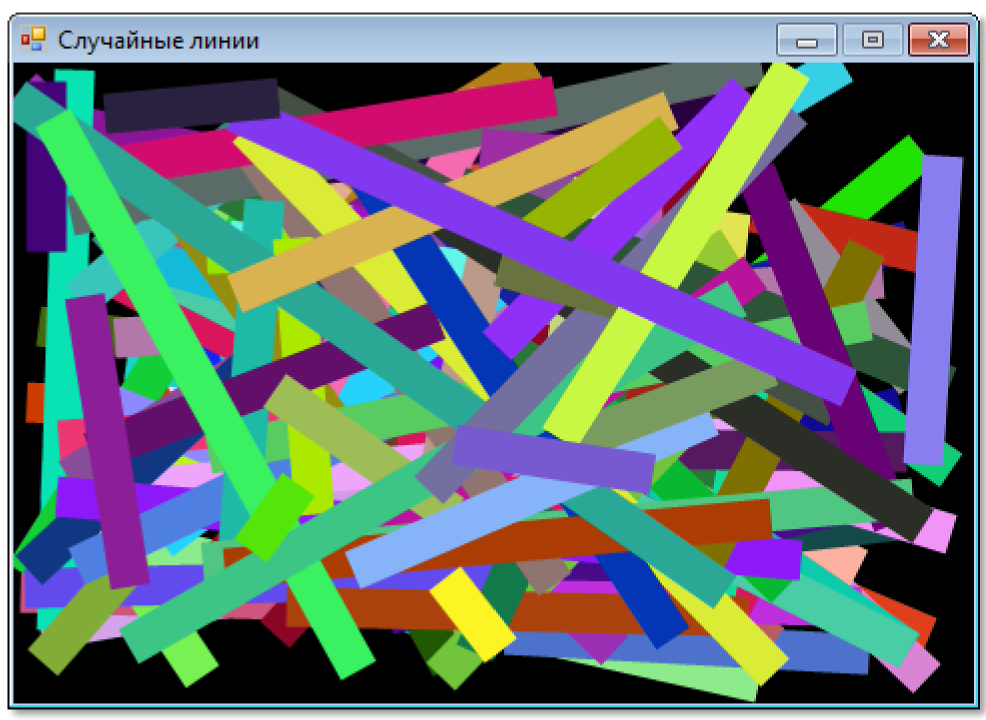


Рис. 20.3. Толстые случайные линии

Исходный код программы находится в папке **Случайные линии**.

Поскольку отрезки беспорядочно разбросаны по экрану, то глаз им не радуется. Однако есть много способов с помощью одних только прямых линий нарисовать шикарные узоры. Сейчас мы напишем совсем короткую программу, которая «намалюет» великолепную картинку (Рис. 20.4).

```
//ПРОГРАММА "ЦВЕТНЫЕ ЛИНИИ"
uses
   GraphABC;

var
   //толщина линий:
   penWidth := 2;

//
//    ОСНОВНАЯ ПРОГРАММА
//
begin
```

```
    SetWindowTitle('Цветные линии');
    SetWindowWidth(640);
    SetWindowHeight(480);
    Window.CenterOnScreen();
    Window.IsFixedSize := true;
    Window.Clear(Color.Black);
    var height := Window.Height;
    var width := Window.Width;

    //толщина линий:
    SetPenWidth(penWidth);
    //отношение высоты окна к ширине:
    var ratio:= height / width;

    var x:=0;
    //Чертим цветные линии
    while(x <= width) do
    begin
       SetPenColor(Color.Red);
       Line(0, Round(x*ratio), width-x, 0);
       SetPenColor(Color.Yellow);
       Line(0, Round((width-x)*ratio), width-x, Round(width*ratio));
       SetPenColor(Color.Blue);
       Line(width-x, Round(0*ratio), width, Round((width-x)*ratio));
       SetPenColor(Color.Green);
       Line(width-x, Round(width*ratio), width, Round(x*ratio));
       x += 10;
    end;//While
end.
```

Чтобы каждый из четырёх наборов линий имел свой цвет, перед рисованием каждой прямой мы задаём её цвет с помощью процедуры *SetPenColor*. Как изменяются координаты начала и конца линии, хорошо видно на рисунке – они скользят по границам клиентской области окна с заданным шагом 10:

```
x += 10;
```

Изменяя этот параметр, а также цвет линий, вы сможете получить и другие узоры.

Рис. 20.4. Цветные линии

Исходный код программы находится в папке **Линии**.

Градиентная заливка

Если чертить горизонтальные линии вплотную друг к другу, то можно получить картину, ласкающую взор. Сейчас мы испытаем модную нынче *градиентную* заливку, в которой один цвет *плавно* переходит в другой. Этот приём часто применяется в заставках, которые появляются на экране при установке программ, а также в компьютерной графике для раскрашивания кнопок, баннеров и других объектов.

Хорошим примером натуральной «градиентной заливки» может служить безоблачное полуденное или закатное небо!

Сохраните исходный код предыдущего проекта в папке **Градиент** и добавьте *переменные:* три - для хранения **красной**, **зелёной** и **синей** составляющих текущего цвета линий и две – для хранения размеров окна:

```
//ПРОГРАММА ДЛЯ ГРАДИЕНТНОЙ
//ЗАЛИВКИ ПРЯМОУГОЛЬНИКА

uses
   GraphABC, ABCButtons;

var
   //толщина линий:
   penWidth := 1;
   // составляющие цвета:
   r, g, b: integer;
   //Размеры окна:
   width, height: integer;
```

Всего наша программа сможет выполнить 6 различных градиентных заливок, поэтому нам потребуется именно столько *кнопок*, чтобы пользователь мог по своему желанию выбирать последовательность их просмотра.

Кнопки мы хорошенько изучим на уроке *Элементы управления*.

```
//КНОПКИ
procedure CreateButtons;
begin
   //Размеры кнопок:
   var w := 80;
   var h := 24;
   //цвет:
   var clr:= clMoneyGreen;
   //координаты:
   var x := width - 85;
   var n := 0;
```

```
   var y := 10;
   var dy := 32;

   var btnYR := new ButtonABC(x, y + dy * n, w, h, 'Ж --> К',
clr);
   n += 1;
   var btnYG := new ButtonABC(x, y + dy * n, w, h, 'Ж --> З',
clr);
   n += 1;
   var btnCG := new ButtonABC(x, y + dy * n, w, h, 'Ц --> З',
clr);
   n += 1;
   var btnCB := new ButtonABC(x, y + dy * n, w, h, 'Ц --> С',
clr);
   n += 1;
   var btnMB := new ButtonABC(x, y + dy * n, w, h, 'Л --> С',
clr);
   n += 1;
   var btnMR := new ButtonABC(x, y + dy * n, w, h, 'Л --> К',
clr);

   //процедуры-обработчики:
   btnYR.OnClick := YR;
   btnYG.OnClick := YG;
   btnCG.OnClick := CG;
   btnCB.OnClick := CB;
   btnMB.OnClick := MB;
   btnMR.OnClick := MR;
end;
```

Надписи на кнопках обозначают направление градиентного перехода, а начальный и конечный цвета указаны первой буквой названия цвета:

Ж – жёлтый

К – красный

Ц – циан

С – синий

Л – лиловый

После нажатия на любую из этих кнопок программа передаёт управление своей процедуре-обработчику, в которой и выполняется подпрограмма, закреплённая за нажатой кнопкой:

```
//
//   ОСНОВНАЯ ПРОГРАММА
//
begin
   SetWindowTitle('Градиент');
   SetWindowWidth(480);
   SetWindowHeight(320);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.Black);
   height := Window.Height;
   width := Window.Width;

   //толщина линий:
   SetPenWidth(penWidth);
   CreateButtons;
end.
```

Названия подпрограмм также обозначены первыми буквами цветов, образующих градиент, так что запутаться в них вам вряд ли удастся.

Поскольку сами процедуры, в отличие от заливок, весьма однообразны, то мы рассмотрим только две их них. Остальные вы можете посмотреть самостоятельно в исходном коде программы.

При переходе от **циана** к **синему** цвету (Рис. 20.5) *синяя* составляющая текущего цвета всегда равна 255 (максимальное значение), *красная* – нулю (отсутствует вообще), а *зелёная* составляющая постепенно уменьшает своё значение при перемещении линий сверху вниз от 255 до 0. Именно благодаря этому изменению и возникает плавный переход цвета по вертикали. Так как по горизонтали цвет остается без изменений, то достаточно провести горизонтальную линию текущего цвета:

```
//переход от циана  к синему - Cyan To Blue
procedure CB;
begin
   b := 255;
```

```
    r := 0;
    for var i := 0 to height do
    begin
        //вычисляем интенсивность зелёной составляющей цвета:
        g := Floor(255 * (1.0 - i / height));
        //задаём цвет линии:
        var clr := RGB(r, g, b);
        SetPenColor(clr);
        //проводим горизонтальную линию:
        Line(0, i, width - 90, i);
        sleep(1);
    end;//For
end;
```

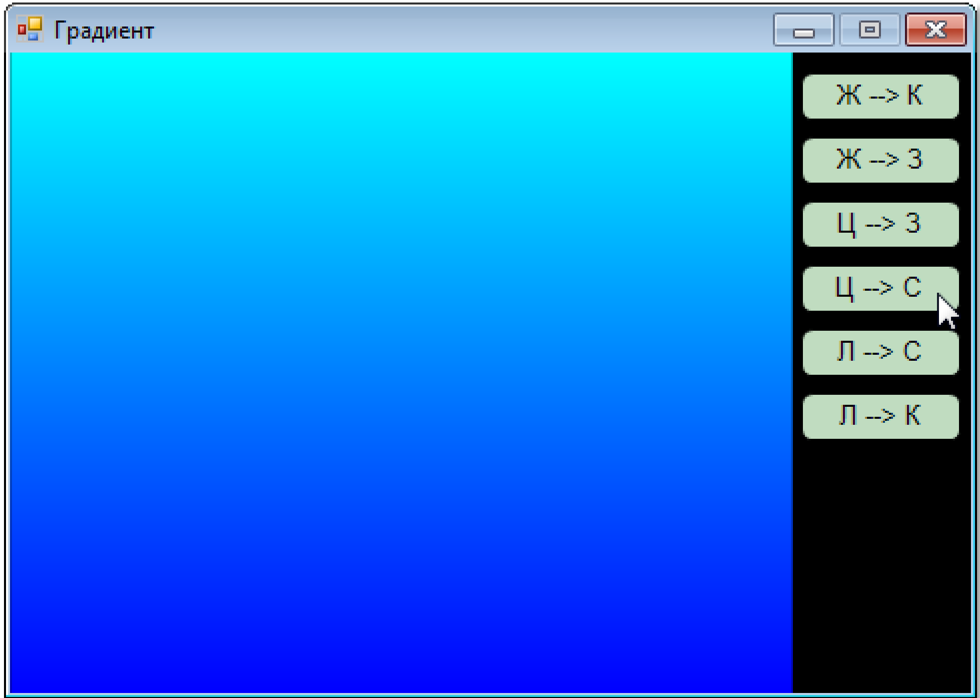


Рис. 20.5. Переход от **циана** к **синему**

Для создания градиентного перехода от **лилового** цвета к **красному** (Рис. 20.6) мы поступаем аналогично. От предыдущего градиента этот градиент отличается только составляющими цвета:

```
//переход от лилового к красному - Magenta To Red
procedure MR;
begin
    r := 255;
    g := 0;
    for var i := 0 to height do
    begin
```

```
      //вычисляем интенсивность синей составляющей цвета:
      b := Floor(255 * (1.0 - i / height));
      //задаём цвет линии:
      var clr := RGB(r, g, b);
      SetPenColor(clr);
      //проводим горизонтальную линию:
      Line(0, i, width - 90, i);
      sleep(1);
   end;//For
end;
```

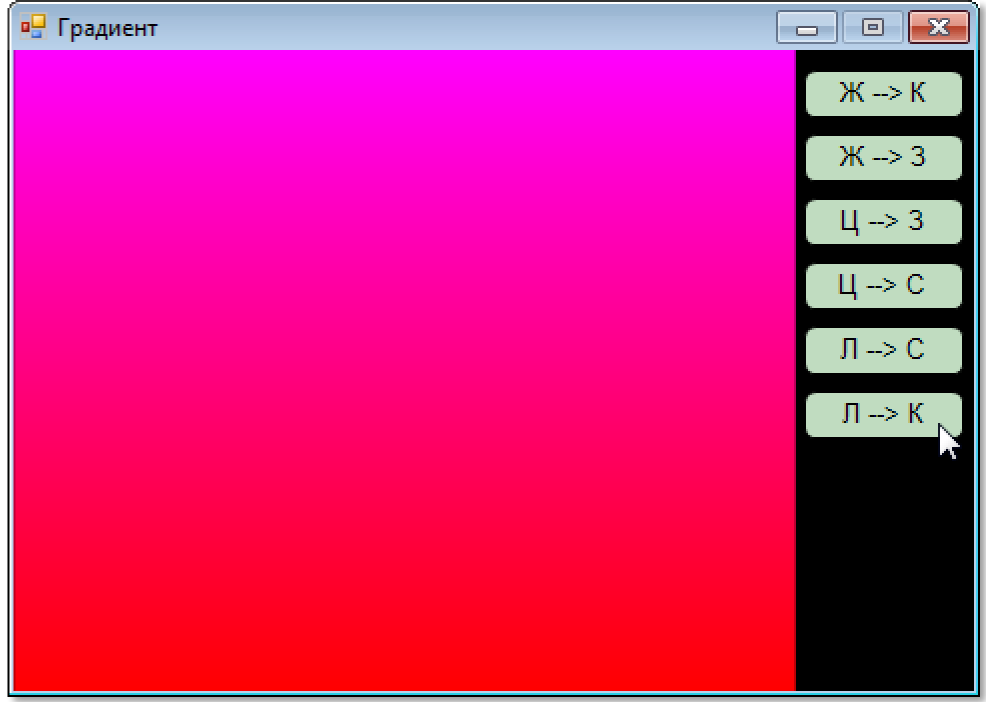


Рис. 20.6. Переход от **лилового** цвета к **красному**

Как видите, создать прямоугольную градиентную заливку совсем несложно, а результат получается впечатляющий! Запускайте программу, жмите на кнопки и наслаждайтесь **цветовыми** переходами!

Исходный код программы находится в папке **Градиент**.

1. Измените программу *Градиент* так, чтобы она чертила линии вертикально (горизонтальный градиент) (Рис. 20.7)!

2. Подумайте, как запрограммировать *двойной* (Рис. 20.8), *радиальный* (Рис. 20.9) и *квадратный* (Рис. 20.10) градиенты.

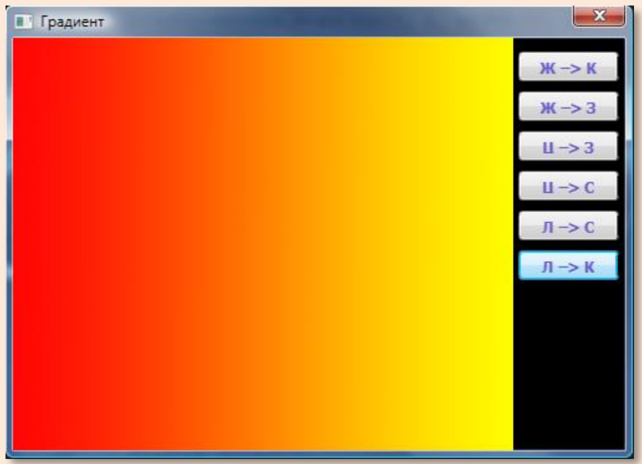


Рис. 20.7. Горизонтальный градиент

Рис. 20.8. Двойной градиент

Рис. 20.9. Радиальный градиент

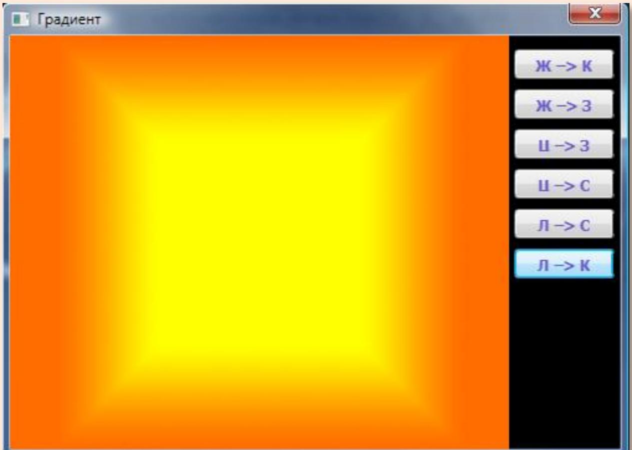


Рис. 20.10. Квадратный градиент

Урок 21. Геометрические фантазии

Из отрезков можно построить любые многоугольники, но для *прямоугольников* в модуле *GraphABC* имеются специальные процедуры.

Прямоугольники и квадраты

Полезный во всех отношениях метод

DrawRectangle(x1, y1, x2, y2);

рисует *контурный прямоугольник*, верхний левый угол которого задаётся координатами *(x1, y1*), а ширина и высота определяются параметрами *width* и *height*. Цвет контура устанавливается с помощью процедуры *SetPenColor*, а его **толщина** – с помощью процедуры *SetPenWidth* (Рис. 21.1).

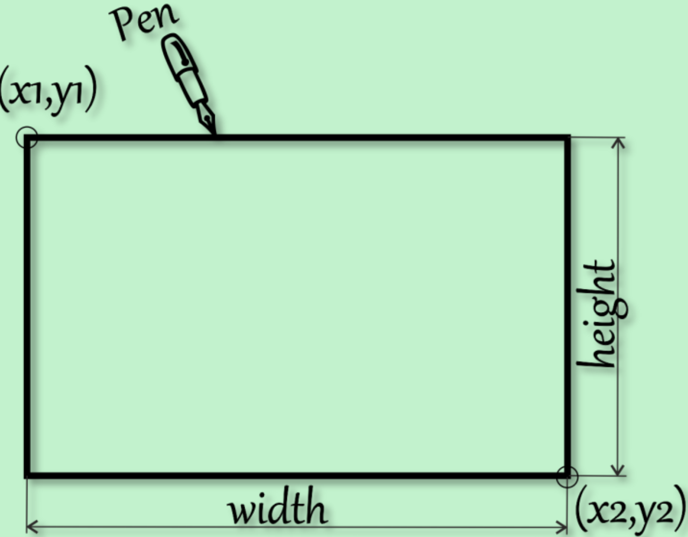


Рис. 21.1. Контурный прямоугольник

Для рисования *закрашенных прямоугольников* имеется метод

FillRectangle(x1, y1, x2, y2);

Их размеры и положение на экране задаются точно так же, как и контурных прямоугольников, но они не имеют контура, а цвет заливки определяется цветом кисти, который изменяется процедурой *SetBrushColor* (Рис. 21.2).

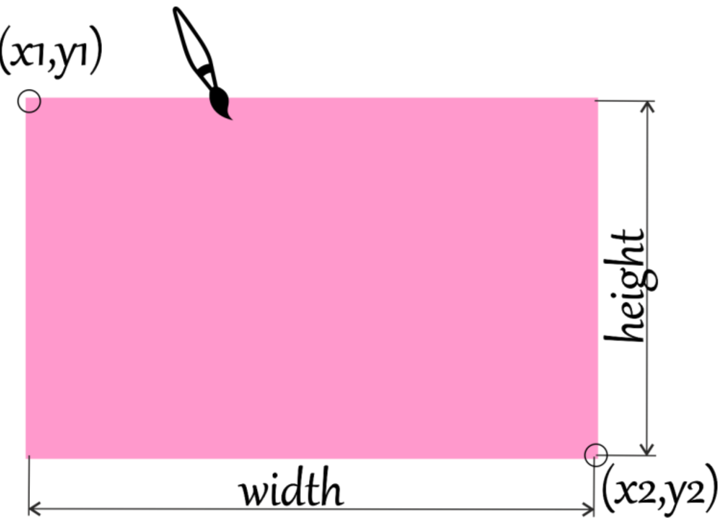


Рис. 21.2. Закрашенный прямоугольник

Если ширина прямоугольника равна высоте, то получится *квадрат*.

А теперь давайте позабавимся, рисуя случайные прямоугольники.

Сохраните исходный код программы *Случайные линии* в папке **Прямоугольники**.

Начнём мы с контурных прямоугольников. Нам придётся изменить только операторы в цикле рисования фигур. Теперь вместо прямых линий мы будем рисовать прямоугольники:

```
//ПРОГРАММА ДЛЯ ВЫЧЕРЧИВАНИЯ
//СЛУЧАЙНЫХ ПРЯМОУГОЛЬНИКОВ

uses
   GraphABC;

var
```

```
   //толщина линий:
   penWidth := 5;

//
//   ОСНОВНАЯ ПРОГРАММА
//
begin
   SetWindowTitle('Случайные прямоугольники');
   . . .
   //Чертим цветные прямоугольники
   while(True) do
   begin
      //задаем случайные координаты -->
      //верхней левой вершины:
      var x1:= Random(width - penWidth div 2);
      var y1 := Random(height - penWidth div 2);
      //нижней правой вершины:
      var x2 := Random(width - penWidth div 2);
      var y2 := Random(height - penWidth div 2);
      //выбираем случайный цвет линии:
      var clr := clRandom();
      SetPenColor(clr);
      //чертим прямоугольник:
      DrawRectangle(x1, y1, x2, y2);
      Sleep(30);
   end;//While
end.
```

Запускаем программу и визуально радуемся нашим «канвасным» достижениям (Рис. 21.3).

Совсем немного изменим программу, и вместо контурных прямоугольников на нас обрушится лавина прямоугольников, окрашенных во все **цвета радуги** (Рис. 21.4):

Рис. 21.3. Контурные прямоугольники

```
    //выбираем случайный цвет линии:
     var clr := clRandom();
     //SetPenColor(clr);
     //чертим прямоугольник:
     //DrawRectangle(x1, y1, x2, y2);
     SetBrushColor(clr);
     FillRectangle(x1, y1, x2, y2);
     Sleep(30);
   end;//While
```

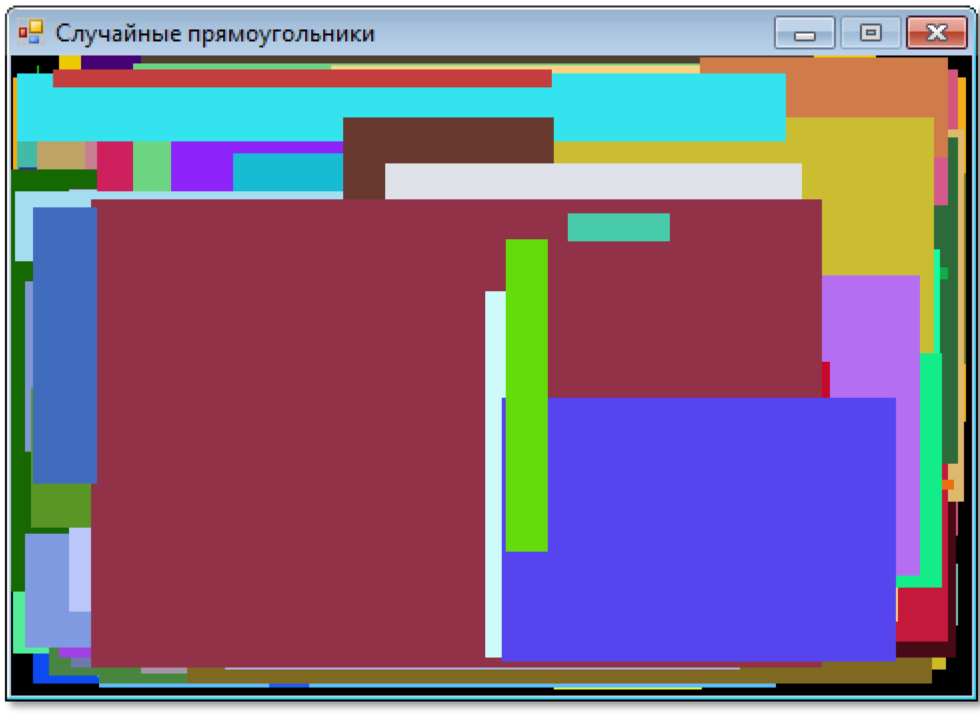


Рис. 21.4. Закрашенные прямоугольники

Эллипсы и круги

Для рисования *эллипсов* метод *GraphABC* также предоставляет нам два метода, которые почти ничем не отличаются от «прямоугольных». Единственное различие состоит в том, что у эллипса нет вершин, поэтому его положение на канве задаётся координатами описанного прямоугольника.

Метод

DrawEllipse(x1, y1, x2, y2);

рисует *контурный эллипс* (Рис. 21.5).

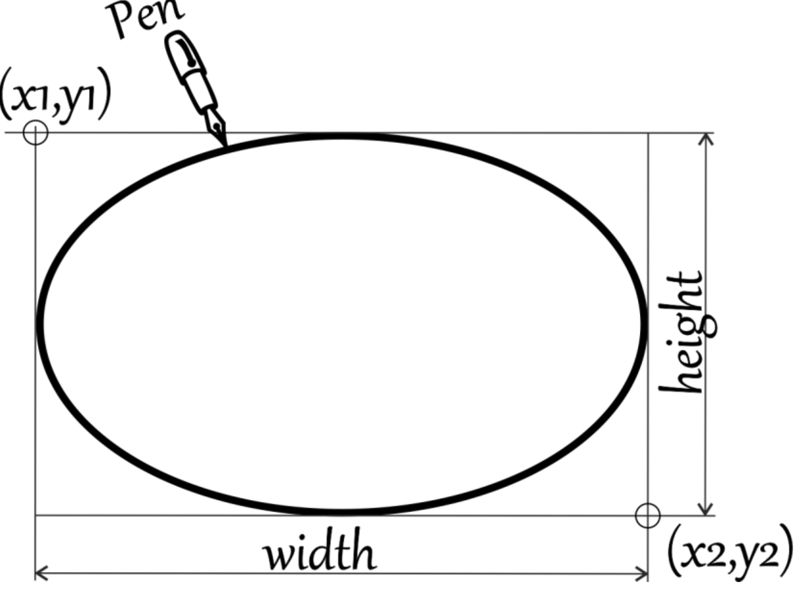


Рис. 21.5. Контурный эллипс

А второй метод -

FillEllipse(x1, y1, x2, y2);

рисует *закрашенный эллипс* (Рис. 21.6).

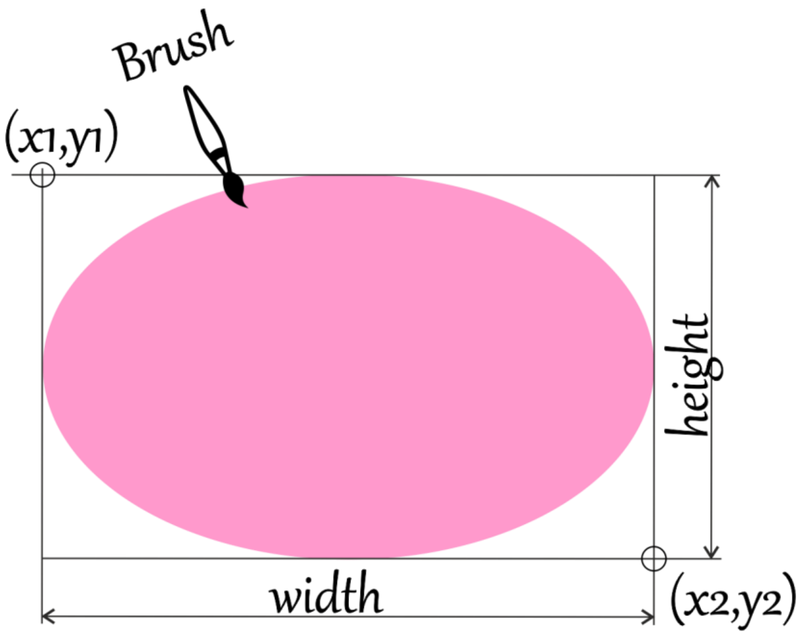


Рис. 21.6. Закрашенный эллипс

Если ширина эллипса равна высоте, то получится *круг*.

Измените обе программы для рисования прямоугольников так, чтобы они рисовали эллипсы!

Исходный код программы находится в папке **Прямоугольники**.

Треугольники

Вы, наверное, подумали, что и для вычерчивания треугольников модуль *GraphABC* также припас два метода:

```
DrawTriangle(x1, y1, x2, y2, x3, y3);

FillTriangle(x1, y1, x2, y2, x3, y3);
```

Но, к сожалению, это не так – треугольники нам придётся самостоятельно вычерчивать из трёх линий, а для этого необходимо знать координатами трёх его вершин.

Дабы лишний раз не повторяться, мы не будем рисовать случайные треугольники, а построим из маленьких треугольников один большой. Это будет свежо и экспрессивно!

Загрузите в *ИСР* проект *Прямоугольники* и сохраните его в папке **Треугольники**. Добавьте *переменную size* – для хранения *длины* сторон правильных треугольников. Также нам придётся увеличить высоту окна для гармонизации нашего произведения:

```
//ПРОГРАММА ДЛЯ ВЫЧЕРЧИВАНИЯ
//ТРЕУГОЛЬНИКА ИЗ ТРЕУГОЛЬНИКОВ

uses
   GraphABC;

var
   //толщина линий:
   penWidth := 2;
   //длина стороны треугольника:
   size:= 32;

//
//   ОСНОВНАЯ ПРОГРАММА
//
begin
   SetWindowTitle('Треугольники');
   SetWindowWidth(480);
   SetWindowHeight(440);
   . . .
```

Маленькие треугольники обращены вершиной вниз и образуют столбики, высота которых уменьшается по мере удаления столбиков от центра.

Первый столбик начинается треугольником с координатами левой вершины (*x1, y1*), а следующие столбики смещены влево и вправо на *size/4*3*j* пикселей. Кроме того, каждый следующий столбик начинается ниже предыдущего на *j*dy*1.5* пикселей.

Почему так происходит, хорошо видно на готовой картинке (Рис. 21.7)

```
   //толщина линий:
   SetPenWidth(penWidth);
   //цвет контура:
   SetPenColor(Color.Yellow);

  //координаты левого угла верхнего треугольника:
   var x1:real:= width/2- size/2;
   var y1:real:= 10;
   //высота треугольников:
   var dy:= size*Sin(DegToRad(60));
   //запоминаем начальные значения:
   var yt:=y1;
   var xt:=x1;

   //рисуем столбики из треугольников:
   For var j:= 0 to 14 do begin
     //левый столбик:
     y1:= yt+ j*dy*1.5;
     x1:= xt- size/4*3*j;
     For var i:= 1 to Floor(15-1.5*j) do begin
        drawTre(x1,y1,dy);
        y1:= y1+dy;
     end;//For
     //правый столбик:
     y1:= yt+j*dy*1.5;
     x1:= xt+ size/4*3*j;
     For var i:= 1 to Floor(15-1.5*j) do begin
       drawTre(x1,y1,dy);
       y1:= y1+dy;
     end;//For
  end;//For j
end.
```

В процедуре *drawTre* мы рисуем *один* треугольник, вычислив прежде координаты его вершин:

```
//Чертим треугольник
procedure drawTre(x, y, dy: real);
begin
   var x1:= Floor(x);
```

```
    var y1:= Floor(y);
    var x2:= x1+size;
    var y2:= y1;
    var x3:= x1+size div 2;
    var y3:= floor(y1+dy);
    Line(x1,y1,x2, y2);
    Line(x2, y2, x3, y3);
    Line(x3, y3, x1,y1);
end;
```

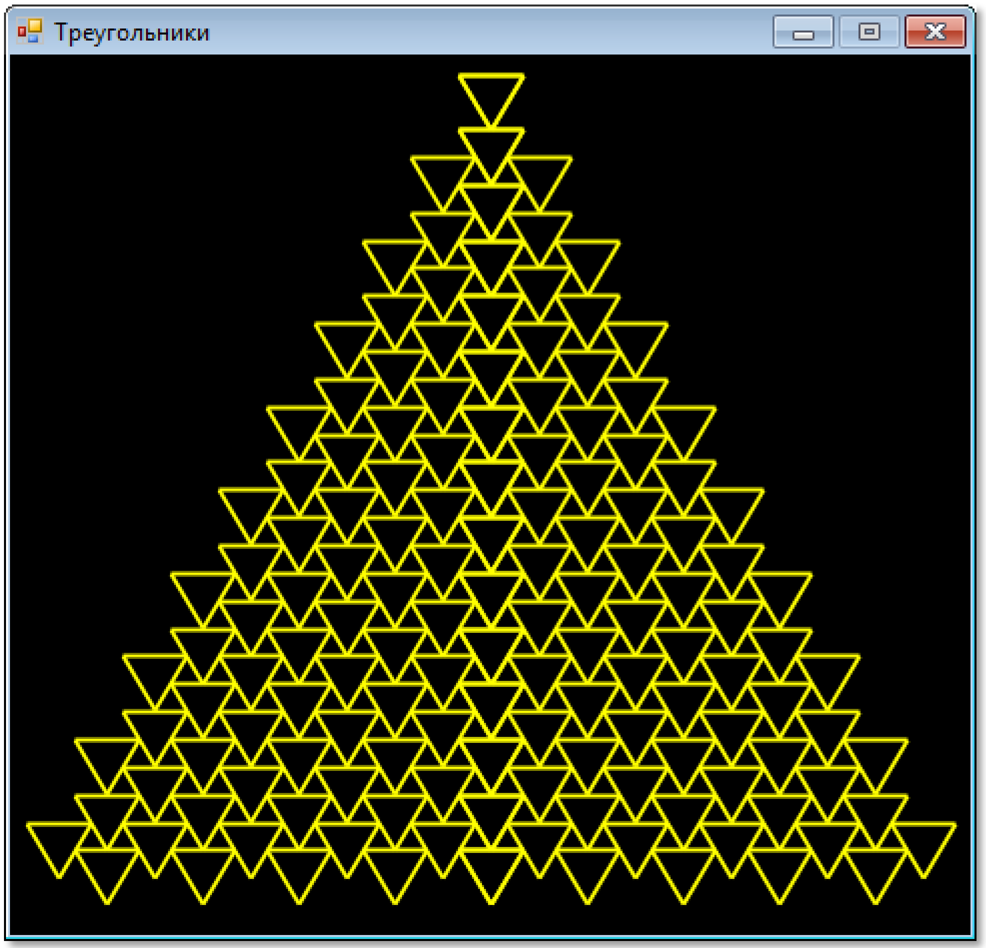


Рис. 21.7. Треугольный треугольник

Исходный код программы находится в папке **Треугольники**.

Живые картинки

До сих пор мы занимались *статическими* картинками: нарисовал их – и всё! А ведь гораздо интереснее наблюдать на экране за непрерывными изменениями геометрических фигур. Сейчас мы напишем программу, которая будет рисовать прямоугольники или эллипсы, непрерывно обновляющие картинку на экране.

Откройте проект *Прямоугольники* и сохраните исходный код в папке **Rectangular** (намёк на то, что картинка составлена из прямоугольников).

Поскольку эта программа не столько сложная, сколько заковыристая, то и переменных нам потребуется немало:

```
//ПРОГРАММА ДЛЯ ВЫЧЕРЧИВАНИЯ
//ДИНАМИЧЕСКИХ ПРЯМОУГОЛЬНИКОВ и ЭЛЛИПСОВ

uses
   GraphABC, ABCButtons;

var
   //Размеры окна:
   width, height: integer;
   //начальные координаты прямоугольника:
   x0 := 0;
   y0 := 0;
   // максимальные координаты прямоугольника:
   xmax := 0;
   ymax := 0;
   //текущие координаты прямоугольника:
   x1 := 0;
   y1 := 0;
   x2 := 0;
   y2 := 0;
   //кнопки:
   var btnRect, btnEll: ButtonABC;

//
//   ОСНОВНАЯ ПРОГРАММА
//
begin
   SetWindowTitle('Динамические прямоугольники');
```

```
   SetWindowWidth(480 + 100);
   SetWindowHeight(320);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.Black);
   height := Window.Height;
   width := Window.Width;
   //размеры области рисования:
   x0 := 0;
   y0 := 0;
   xmax := width - 100;
   ymax := height;

   CreateButtons;
end.
```

Обратите внимание, что переменные *(x1, y1)* и *(x2, y2)* задают координаты верхнего левого и правого нижнего углов прямоугольника, которые необходимы при вызове процедуры *DrawRectangle*. Рисовать мы будет прямоугольники и эллипсы, а чтобы переключаться между ними, установим пару *кнопок*:

```
procedure Rect;
begin
   DrawFig('Rect');
end;
procedure Ell;
begin
   DrawFig('Ell');
end;
//КНОПКИ
procedure CreateButtons;
begin
   //Размеры кнопок:
   var w := 80;
   var h := 24;
   //цвет:
   var clr := clMoneyGreen;
   //координаты:
   var x := width - 90;
   var n := 0;
   var y := 10;
   var dy := 48;
```

```
   var btnRect := new ButtonABC(x, y + dy * n, w, h, 'Пр-ки',
clr);
   n += 1;
   var btnEll := new ButtonABC(x, y + dy * n, w, h, 'Эллипсы',
clr);

   //процедуры-обработчики:
   btnRect.OnClick := Rect;
   btnEll.OnClick := Ell;
end;
```

После нажатия на эти кнопки мы попадём в процедуру-обработчик *Rect* или *Ell*, а из неё – в процедуру *DrawFig* с соответствующим значением параметра *figura*:

```
procedure DrawFig(figura: string);
begin
   btnEll.Visible:= false;
   btnRect.Visible:= false;

   if figura = 'Rect' then
      SetPenWidth(2)
   else
      SetPenWidth(1);

   SetBrushColor(Color.Black);
   FillRectangle(x0, y0, xmax + 4, ymax);

   //Чертим цветные прямоугольники
   for var i := 6 downto 1 do
   begin
      for var j := 0 to 5 do
      begin
         //стартовый прямоугольник:
         x1 := x0;
         y1 := y0;
         x2 := xmax;
         y2 := ymax;
         //выбираем случайный цвет контура:
         var clr := clRandom;
         SetPenColor(clr);
         while (x1 <= xmax) and (y1 <= ymax) do
```

```
         begin
            if (figura = 'Rect') Then
               //рисуем прямоугольник:
               DrawRectangle(x1, y1, x2, y2)
            Else
               //рисуем эллипс:
               DrawEllipse(x1, y1, x2, y2);
            Sleep(15);
            //новые координаты вершин прямоугольника:
            x1 := x1 + i;
            y1 := y1 + j;
            x2 := x2 - i;
            y2 := y2 - j;
         end;//While
      end;//For
   end;//For
   btnEll.Visible:= true;
   btnRect.Visible:= true;
end;
```

Здесь мы, прежде всего, стираем все предыдущие кривые, устанавливаем разную толщину контура для прямоугольников и эллипсов (вы можете выбрать свои значения) и сообщаем основной части программы, какими фигурами мы желаем рисовать.

Так как справа мы разместили кнопки, то это обстоятельство следует учесть при рисовании:

```
  //размеры области рисования:
   x0 := 0;
   y0 := 0;
   xmax := width - 100;
   ymax := height;
```

Далее мы вычерчиваем прямоугольники или эллипсы. Начинаем с самой большой фигуры, размеры которой равны области рисования, затем постепенно координаты вершин прямоугольника всё ближе и ближе приближаются к центру и, наконец, наступает момент, когда левый верхний угол занимает место правее и/или ниже правого нижнего, после чего прямоугольник «выворачивается», но процедура *DrawRectangle* рисует его правильно.

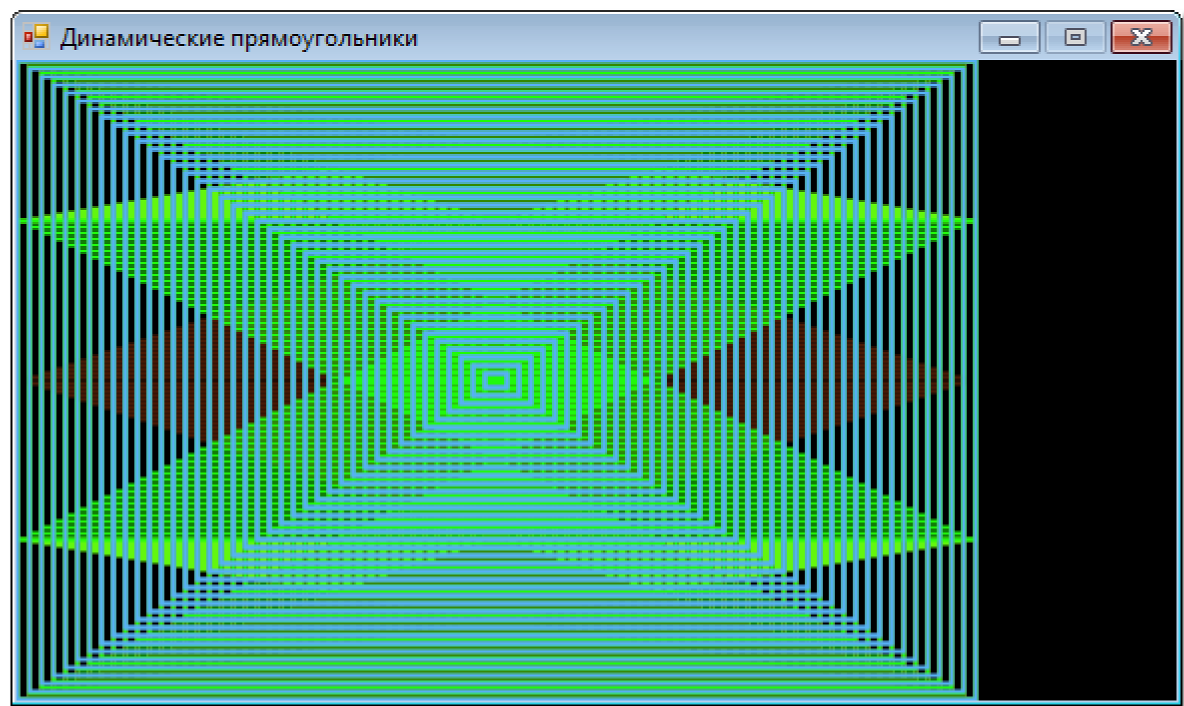


Рис. 21.8. Динамические прямоугольники

Правило преобразования фигур очень простое, но образующиеся при работе программы динамические узоры весьма любопытны, и разглядывать их можно не одну минуту (Рис. 21.8 и 21.9)!

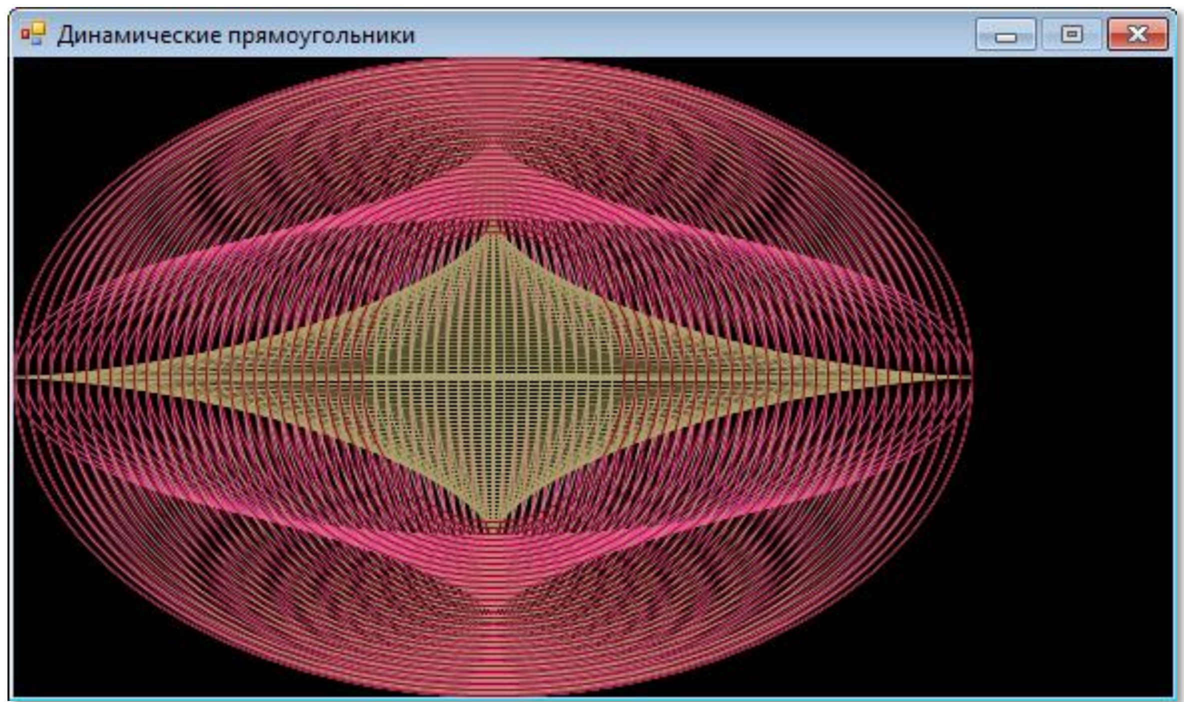


Рис. 21.9. Динамические эллипсы

Поскольку программа во время рисования не реагирует на нажатие кнопок, то их следует временно убрать. После завершения полного цикла рисования они появляются вновь, и пользователь может нажать любую из них и продолжить рисование с самого начала.

Исходный код программы находится в папке **Rectangular**.

Урок 22. Черепашья графика

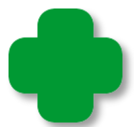
Тише едешь – дальше будешь!

Черепашье кредо

Мы уже научились рисовать картинки с помощью графических примитивов, которые предоставляет нам модуль *GraphABC*, - окружностей, эллипсов, прямоугольников, квадратов и прямых линий. Их вполне достаточно, чтобы вычертить на канве любую фигуру. Для задания координат характерных точек этих примитивов мы используем прямоугольную систему координат канвы. Однако мы знаем, что в некоторых случаях удобнее перейти к *полярным* координатам, чтобы вычертить те или иные кривые (мы с ними хорошо познакомились на уроке *Полярная система координат*). Впрочем, нам всё равно пришлось пересчитывать полярные координаты точек кривых в прямоугольные, поскольку координаты пикселей канвы мы не можем задавать иначе. А ведь было бы, наверное, зд***о***рово рисовать кривые *сразу* в полярных координатах!

Впервые такая возможность появилась в языке программирования *Лого* (*Logo*), который был разработан в 1967 году *Сеймуром Пейпертом* и *Идит Харель*. *Лого* создавался как средство для обучения детей дошкольного и младшего школьного возраста основам программирования. Главными объектами языка *Лого* были слова и предложения, что и определило выбор названия для самого языка – *лого* по-гречески значит *слово*. Но известность этому языку программирования принесла маленькая *Черепашка* (*Turtle*), которая умела выполнять несложные команды и вычерчивать линии. Именно благодаря *Черепашке* этот язык стал очень популярным в 70-80-е годы прошлого века, а *Черепашка* появилась и в других языках программирования – *LISP, SCHEME* и многих версиях бейсика. А вот в *паскале* готовой *Черепашки* пока нет. Но мы попробуем сымитировать её подручными средствами. В модуле *ABCObjects* имеется подходящий класс *PictureABC*, в который можно загрузить черепашкину фотографию и двигать её по окну приложения.

К сожалению, объекты этого класса нельзя поворачивать, а это значит, что наша *Черепашка* всегда будет смотреть вверх (или вниз – но ни в какую другую сторону), независимо от того, куда она ползёт. Это может сбить с толку, поэтому мы нарисуем на её панцире красный треугольник типа *RegularPolygonABC*, который позволяет вертеть себя во все стороны. Правда, у треугольника *три* вершины, но вы по смыслу происходящих событий легко догадаетесь, куда он «смотрит».



Вообще говоря, *Черепашка* хороша для обучения маленьких детей, поскольку она очень забавно крутится-вертится и ползает по экрану. Однако при сложных графических построениях она становится обузой и тормозит процесс рисования. В таких случаях *Черепашку* просто убирают с экрана.

Исходный код *Черепашки* находится в модуле *u_turtle*, который нужно добавить к проекту. Его мы обсуждать не будем, поскольку нас интересует только *Черепашка*, а не её реализация на языке паскаль. Что, впрочем, не мешает вам изучить и усовершенствовать код самостоятельно.

Чтобы увидеть *Черепашку* на экране, нужно её *создать*. Для этого объявите переменную **turtle** типа *CTurtle*:

```
var
   turtle: CTurtle;
```

А в основной программе создайте её, вызвав *конструктор* с указанными аргументами (Рис. 22.1):

```
//
// ОСНОВНАЯ ПРОГРАММА
//
begin
   //Подготавливаем программу:
   prepareProg();

   turtle:= new CTurtle(CX, CY, true);
```

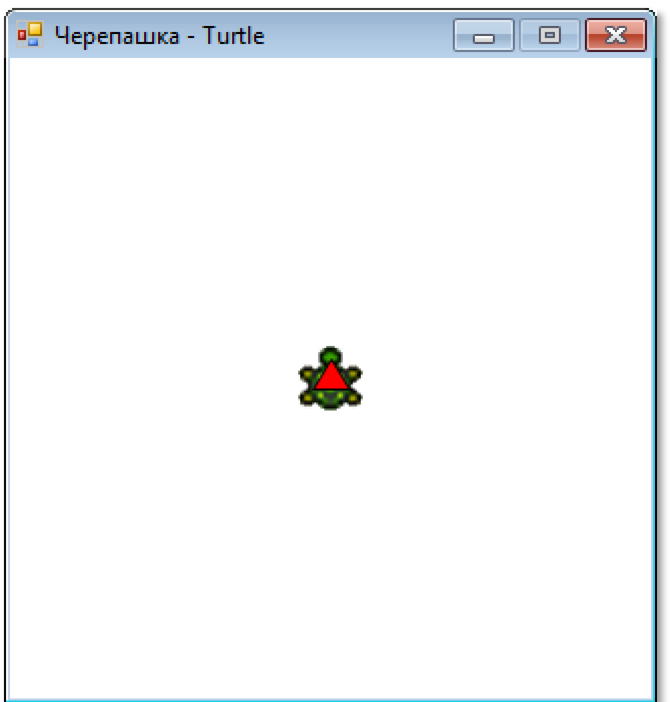
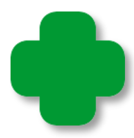


Рис 22.1. «Новорождённая» *Черепашка*



CX и *CY* – это координаты центра окна, а третий параметр отвечает за размеры *Черепашки*. Если он равен *true*, то *Черепашка* будет БОЛЬШАЯ, если *false* – маленькая, она почти целиком скрывается под треугольником (Рис. 22.2):

```
//большая Черепашка:
//turtle:= new CTurtle(CX, CY, true);
//маленькая Черепашка
turtle:= new CTurtle(CX, CY, false);
```

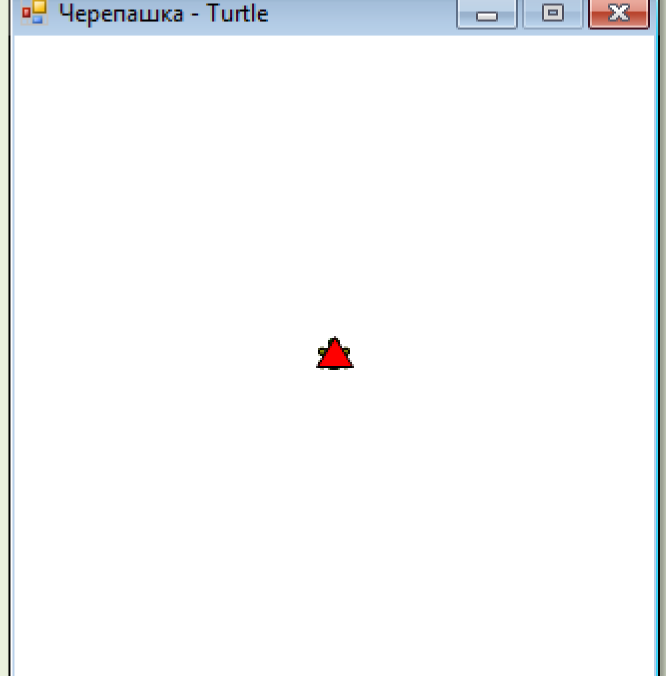


Рис. 22.2. Маленькая *Черепашка*

«Свежая» *Черепашка* всегда смотрит *вверх*. Направление её взгляда знать очень важно, потому что *Черепашка* может ползти только туда, куда глаза глядят!

Если вы хотите, чтобы *Черепашка* поползла в каком-либо направлении, сначала поверните её!

В разных системах программирования *Черепашки* отличаются по форме. Обычно это стилизованные черепахи, но встречаются и примитивные виды *Черепашек* – равносторонние треугольники и наконечники стрелок (Рис. 22.3). Их объединяет желание ползать по экрану и возможность указывать направление их перемещения.

Рис 22.3. *Черепашки* разных видов

В процедуре *prepareProg* мы создаём окно небольших размеров и записываем в глобальные переменные *CX* и *CY* координаты его центра:

```
//Подготавливаем программу
procedure prepareProg();
begin
   SetWindowTitle('Черепашка - Turtle');
   SetWindowWidth(320);
   Window.Height:= 320;
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   height := Window.Height;
   width := Window.Width;
   //координаты центра окна:
   CX:= width div 2;
   CY:= height div 2;
```

Если вы создадите Черепашку с помощью конструктора без параметров, то она будет **большой** и займёт место в левом верхнем углу клиентской области окна:

```
turtle:= new CTurtle();
```

Затем вы можете перенести её в любое место окна, например, в его центр, присвоив свойствам *X* и *Y Черепашки* нужные значения:

```
  //устанавливаем Черепашку в центре окна:
  turtle.X := CX;
  turtle.Y := CY;
```

Имейте в виду, что координаты *Черепашки* отсчитываются в системе координат канвы, то есть начало координат находится в левом верхнем углу, а ось *Y* направлена вниз!

Черепашка может выполнять несколько простых *команд*. Например, попросим её ползти вперед:

```
 turtle.Move(100);
```

Как и было обещано, *Черепашка* выполнит команду и проползёт *вперёд* (то есть *вверх*, потому что она туда смотрит!) 100 пикселей (Рис. 23.4).

А всё-таки наша *Черепашка* умеет двигаться и в противоположном взгляду направлении! Дайте ей команду

```
turtle.Move(-100);
```

и она, смотря вперёд, попятится назад (Рис. 22.5).

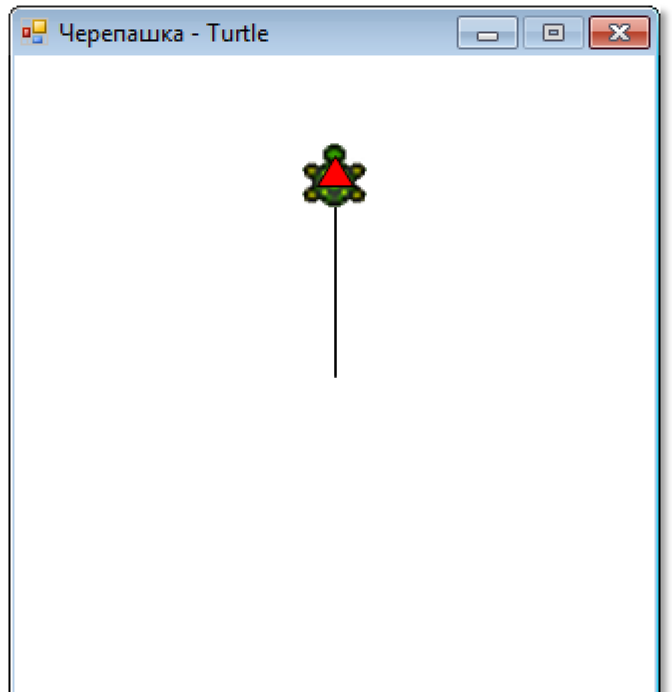


Рис. 22.4. Полный вперёд!

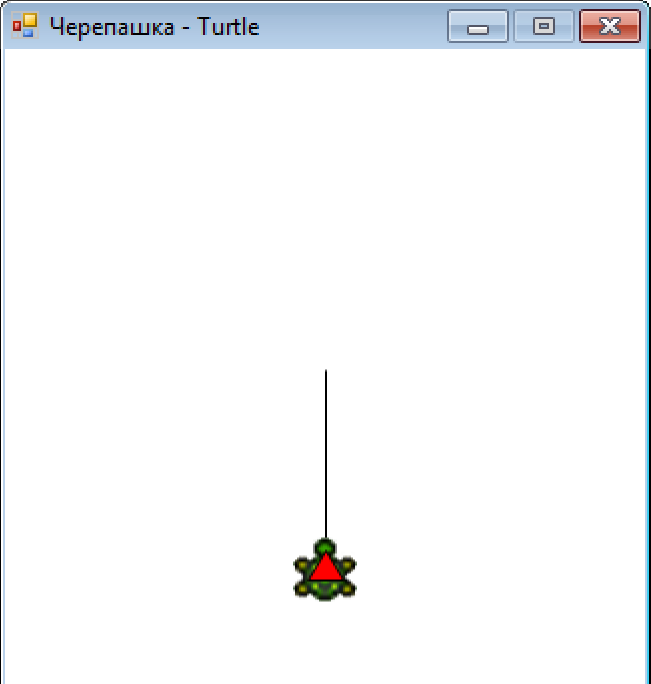


Рис. 22.5. Полный назад!

За ней остался след - тонкая прямая **чёрная** линия. Это всё потому, что наша *Черепашка* не простая, а вооруженная – карандашом! Вы можете дать *Черепашке* карандаш любого цвета, а также с грифелем большего диаметра, чтобы линии стали **толще**. Правда, для этого вам придётся обратиться к процедурам модуля *GraphABC*, который и выдаст нашей *Черепашке* нужный карандаш:

```
SetPenColor(Color.Red);
SetPenWidth(10);
turtle.Move(100);
```

Теперь *Черепашка* проведет **толстую красную** линию (Рис. 22.6).

Но *Черепашка* при ползании *не всегда* прочерчивает линию. Дело в том, что по умолчанию карандаш находится в боевом положении, поэтому от него остаётся след. За положение карандаша отвечают методы *Черепашки* **PenDown** и **PenUp**. Первый *опускает* карандаш, а второй – *поднимает* его. Соответственно этому *Черепашка* либо чертит линию, либо просто перемещается в новое положение.

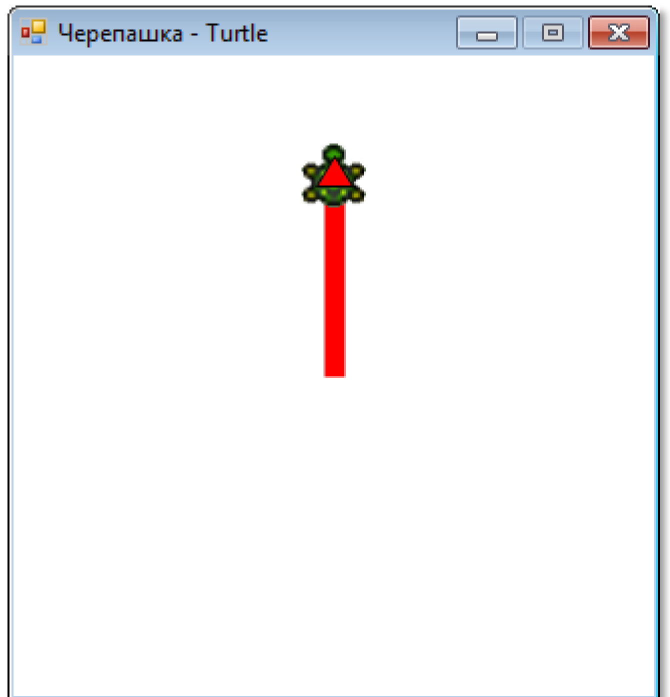


Рис. 22.6. *Черепашка* бороздит просторы *графического окна*

Немного исправим программу так, чтобы *Черепашка* чертила *пунктирную* линию (Рис. 22.7).

```
turtle.clr:= Color.Green;
turtle.w:=4;
For var i:= 1 to 10 do begin
    if (i mod 2 = 0) then
        turtle.PenUp()
    Else
        turtle.PenDown();
    turtle.Move(10);
end;//For
```

Процедуры *SetPenColor* и *SetPenWidth* действуют только разово, поэтому, чтобы сохранить настройки линий, пользуйтесь соответствующими свойствами *Черепашки*:

```
turtle.clr:= Color.Green;
turtle.w:=4;
```

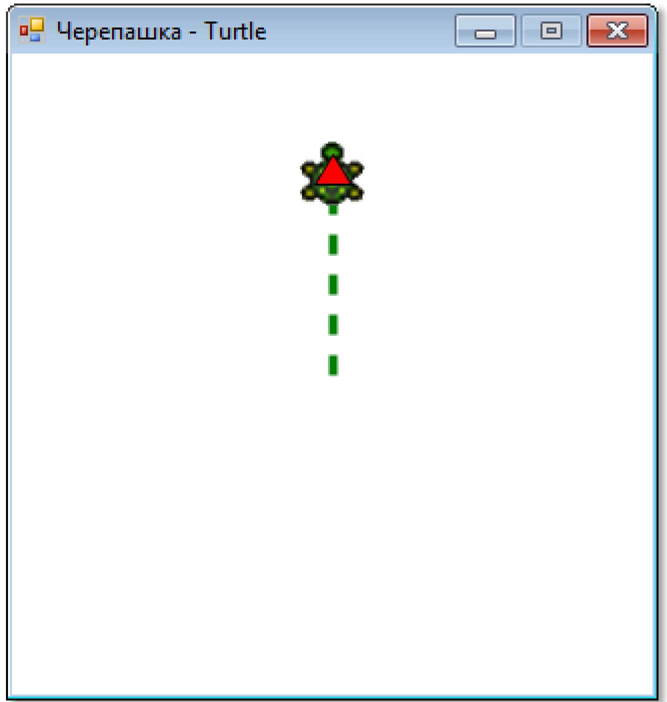


Рис. 22.7. *Черепашка* оставляет за собой следы

Иногда бывает полезно *спрятать Черепашку* после того как она выполнила свою работу, что часто бывает и в жизни: исполнители не должны заслонять руководящие кадры. Вызвав команду

turtle.Hide();,

мы превратим нашу *Черепашку* в *Черепашку-невидимку*. И что особенно приятно, она при этом не утратит способности вычерчивать линии (Рис. 22.8).

Наша *Черепашка* может перемещаться в указанную точку канвы, то есть попросту чертить линию от точки, в которой она находится, до любой другой. Для этого при вызове метода

MoveTo(x,y);

следует указать прямоугольные координаты новой точки.

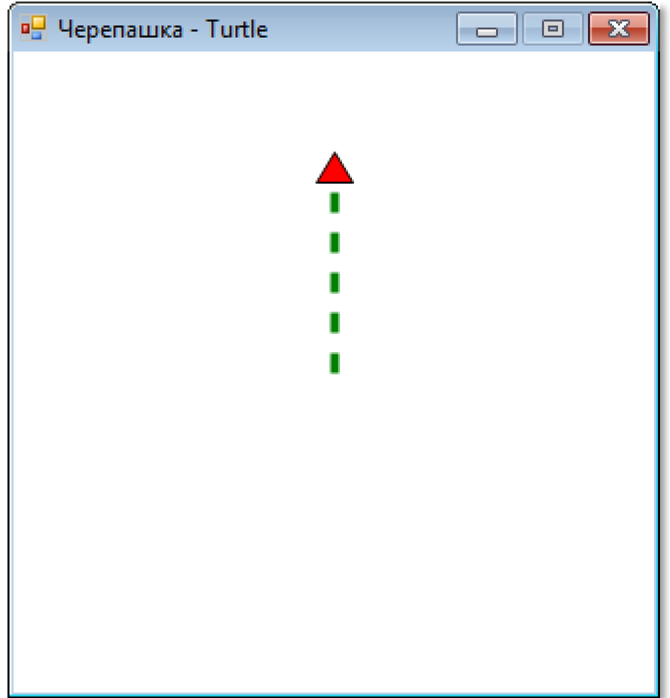


Рис. 22.8. *Черепашка* спряталась

После установки *Черепашки* в центре окна давайте-ка погоняем её по канве (Рис. 22.9):

```
turtle.Show();
 turtle.PenDown;
 turtle.clr:= Color.Green;
 turtle.w := 4;
 turtle.X := CX;
 turtle.Y := CY;
 turtle.MoveTo(300,300);
 turtle.MoveTo(20,300);
```

Таким образом, *Черепашка* неплохо ориентируется и в прямоугольной системе координат. Однако от неё было бы не много пользы, если бы она просто перемещалась от одной точки к другой в этой системе координат.

Главное достоинство *Черепашки* в том, что она может передвигаться и в *полярной* системе координат. Действительно, при запуске программы мы можем установить *Черепашку* в центре окна. Будем считать, что это *полюс* полярной системы координат.

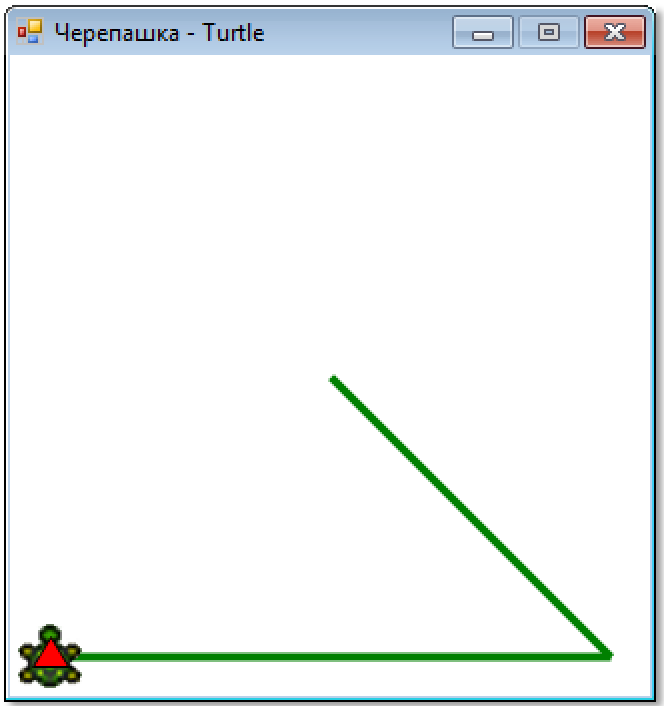


Рис. 22.9. *Черепашка* выполняет команды *MoveTo*

Три метода класса *Turtle*

TurnLeft()
TurnRight()
Turn(угол)

поворачивают *Черепашку* вокруг собственной оси. Первый метод – на 90 градусов *против* часовой стрелки, второй – на 90 градусов *по* часовой стрелке, третий – на произвольный угол. В последнем случае *Черепашка* повернётся против часовой стрелки, если угол отрицательный, и по часовой стрелке – если положительный.

Например, мы легко заставим нашу *Черепашку* танцевать брейк:

```
For var i:= 1 to 10 do begin
   turtle.TurnLeft();
   Sleep(200);
   turtle.TurnRight();
   Sleep(200);
 end;//For
```

Или так:

```
For var i:= 1 to 10 do begin
    turtle.Turn(300);
    Sleep(200);
    turtle.Turn(-320);
    Sleep(200);
  end;//For
```

С помощью этих методов мы легко сможем переместить *Черепашку* в любую точку, заданную в полярных координатах. Например, выбрав полярный угол и полярный радиус

```
 var Z:=45;
  var R:=120;
  turtle.Turn(Z);
  turtle.Move(R);,
```

мы отправим *Черепашку* в точку, координаты которой заданы в полярной системе координат (Рис. 22.10).

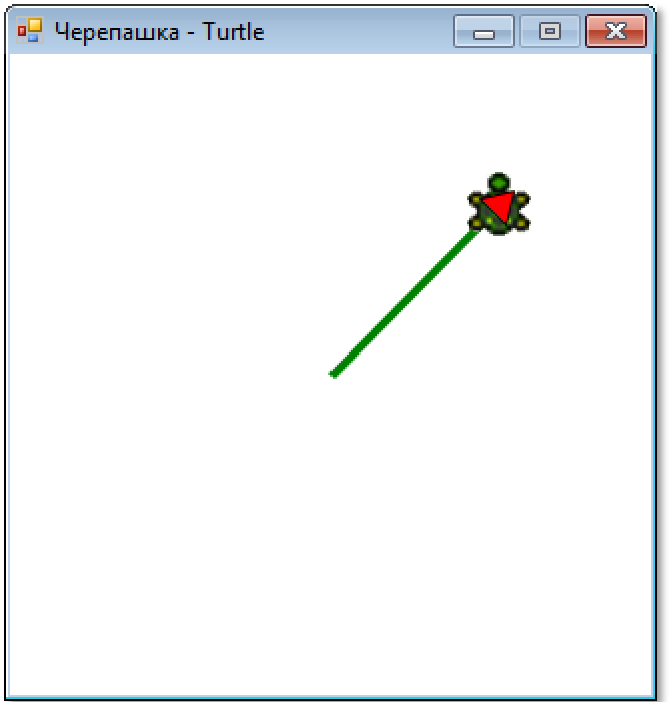


Рис. 22.10. Полярная *Черепашка*

Всякий раз возвращая *Черепашку* в начало координат и поворачивая её головой вверх, мы можем наставить сколько угодно *точек* в полярной системе координат:

```
var R:=120;
 var Z:=0;
 turtle.Turn(Z);
 For var i:= 1 to 8 do begin
    turtle.PenUp();
    turtle.Move(R-4);
    turtle.PenDown();
    turtle.Move(4);
    turtle.PenUp();
    turtle.MoveTo(CX,CY);
    turtle.Angle := Z;
    turtle.Turn(360/8* i);
 end;//For
```

Например, мы можем расставить точки в вершинах правильного восьмиугольника (Рис. 22.11).

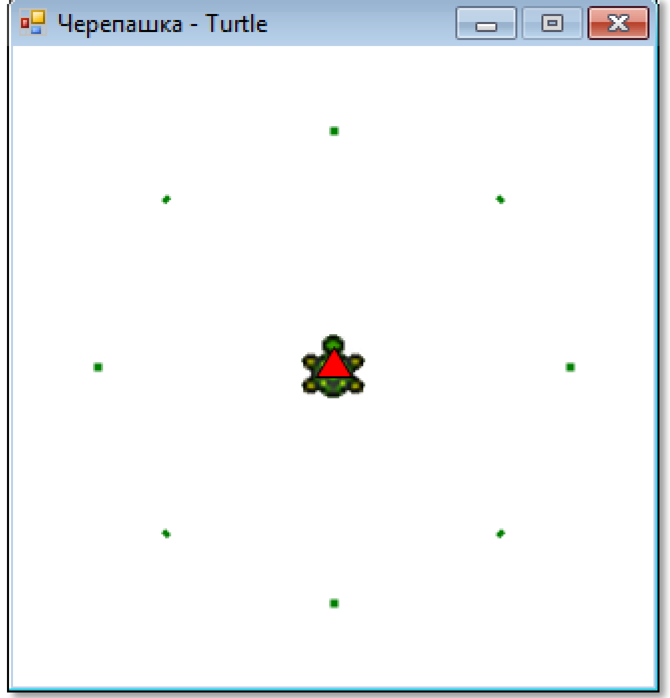


Рис. 22.11. *Черепашка* ставит точки в полярной системе координат

А что у нас получится, если мы продолжим перемещения *Черепашки*, не возвращая её в прежнее положение? – Тогда начало полярных координат переместится в точку, где находится *Черепашка*, а полярная ось повернётся по направлению взгляда *Черепашки*. Следующий поворот будет отсчитываться уже не от горизон-

тальной оси, а от *нового* положения полярной оси. Мы знаем, что все внутренние углы правильного треугольника равны 60 градусов, а внешние – 120, поэтому мы легко построим правильный треугольник (Рис. 22.12), если вычертим три его стороны, начиная каждую из конца предыдущей стороны. Так как длина всех сторон одинакова, то достаточно повернуть *Черепашку* на нужный угол:

```
var R:=120;
 turtle.TurnRight();
 For var i:= 1 to 3 do begin
     turtle.Turn(-120);
     turtle.Move(R);
 end;//For
```

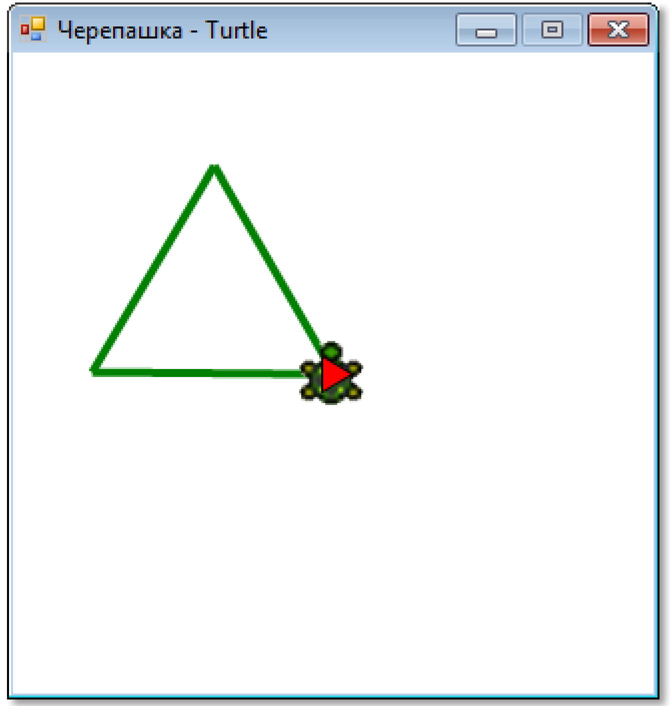


Рис. 22.12. *Черепашка* построила правильный треугольник

Мы сделаем формулу более универсальной, если введём параметр *n*, равный числу сторон правильного многоугольника:

```
turtle.X := width-100;
 turtle.Y := height-20;
 var R:=120;
 turtle.TurnRight();
 var n:= 8;//6;
```

```
   For var i:= 1 to n do begin
      turtle.Turn(-360/n);
      turtle.Move(R);
   end;//For
```

Для построения *любого* правильного многоугольника (Рис. 22.13) нам потребовалось написать всего несколько строк кода и при этом – никаких расчётов координат вершин!

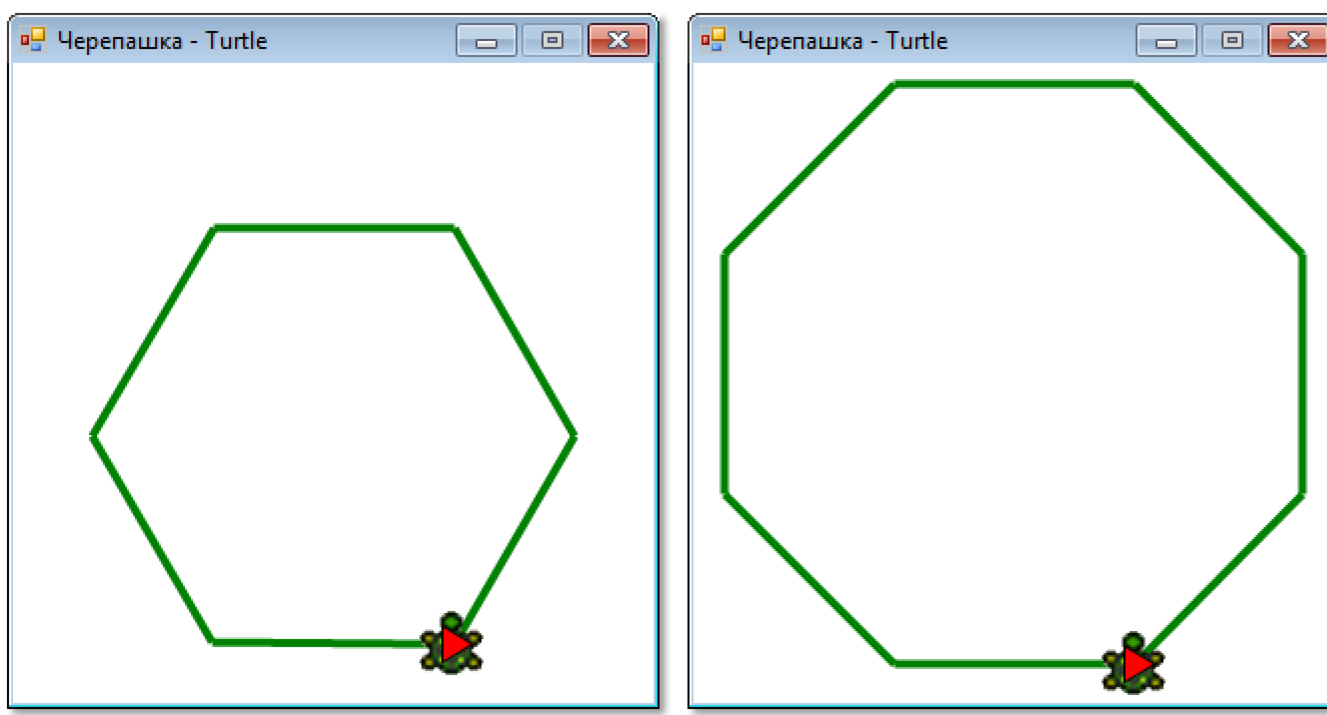


Рис. 22.13. *Черепашка* построила правильные многоугольники

Нетрудно догадаться, что если задать достаточно большое число вершин, то многоугольник превратится в *окружность* (Рис. 22.14).

```
  turtle.X := width-155;
   turtle.Y := height-12;
   var R:=12;
   turtle.TurnRight();
   var n:=80;
   For var i:= 1 to n do begin
     turtle.Turn(-360/n);
     turtle.Move(R);
   end;//For
```

Как это ни удивительно, но *Черепашке* нетрудно вычертить и *спираль* (Рис. 22.15).

```
//спираль:
turtle.X := CX;
turtle.Y := CY-10;
var R:=1.0;
turtle.TurnRight();
For var i:= 1 to 240 do begin
  turtle.Move(R);
  turtle.Turn(10);
  R += 0.1;
end;//For
```

Если при повороте *Черепашки* задать *отрицательный* угол

```
turtle.Turn(-10);
```

то вместо правой спирали мы получим *левую* (она закручена в противоположном направлении).

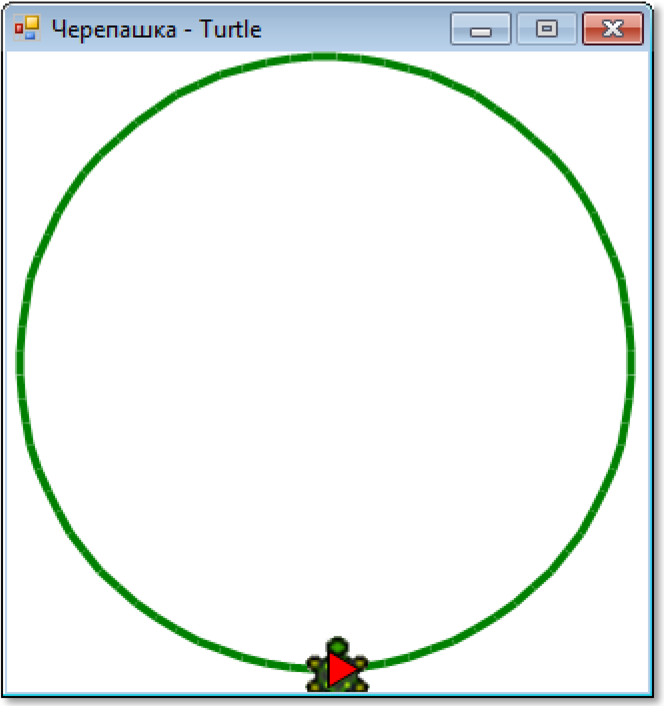


Рис. 22.14. Черепашка чертит окружность

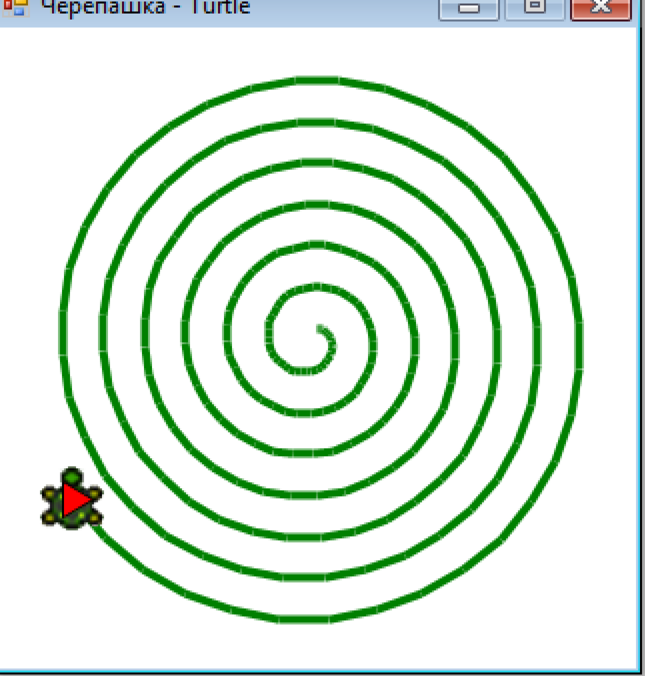


Рис. 22.15. Черепашка рисует спираль

А теперь давайте перейдём к более сложным кривым, которые наверняка убедят вас, что вы не напрасно познакомились с *Черепашкой.*

Для начала мы заставим многоугольники *вращаться* вокруг одной из своих вершин:

```
//Вращающеся фигуры
procedure polyStop();
begin
   repeat
      turtle.Move(side);
      turtle.Turn(angle1);

      _turn += angle1;
   until _turn mod 360 = 0;
End;

procedure polyRoll();
begin
   For var i:= 1 to 12 do begin
      polyStop();
      turtle.Turn(angle2);
   end;//For
End;
```

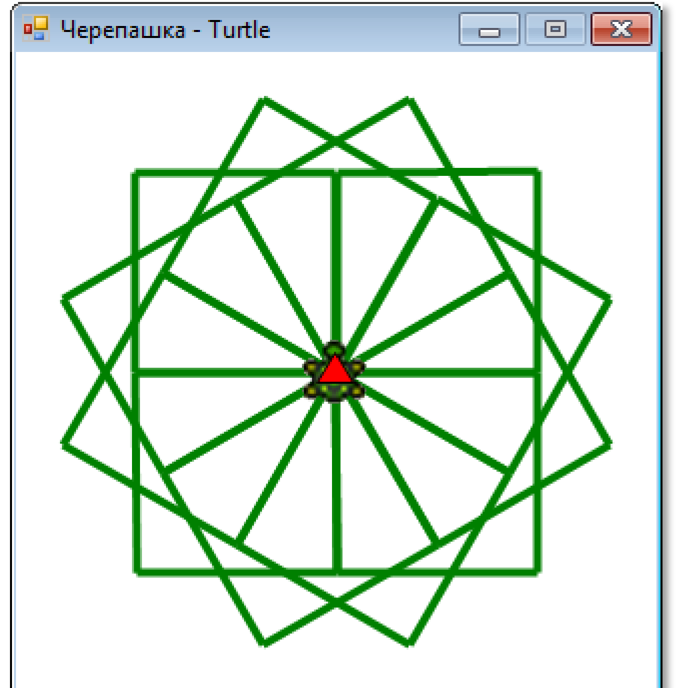
Изменяя значения переменных *angle1* и *angle2*, мы получим разные многоугольники, вращающиеся вокруг центра окна (Рис. 22.16, слева):

```
  turtle.clr:= Color.Green;
   turtle.w:=4;
   _turn:=0;
   angle1:=90;
   angle2:=30;
   side:=100;
   polyRoll();
```

И Рис. 22.16, справа:

```
  turtle.clr:= Color.Green;
   turtle.w:=4;
   _turn:=0;
```

```
   angle1:=60;
   angle2:=45;
   side:=80;
   polyRoll();
```



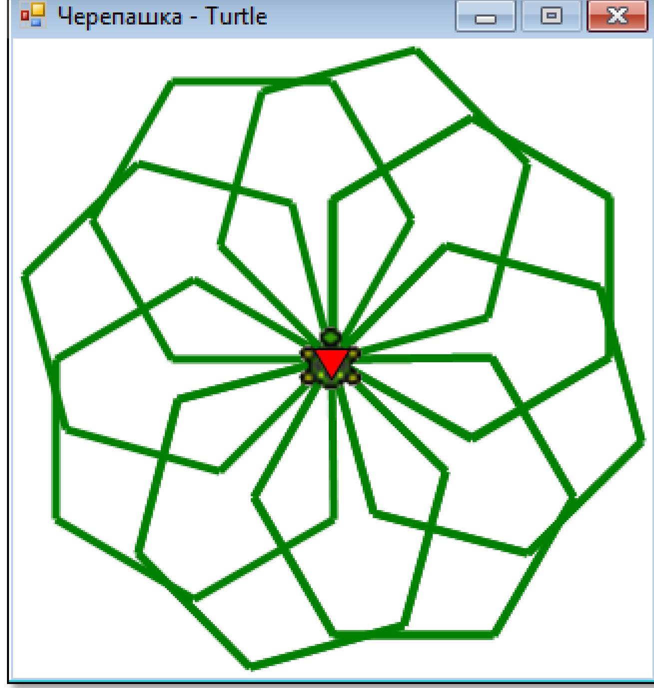


Рис. 22.16. Вращающиеся фигуры

Необыкновенно простая рекурсивная процедура *polySpi* при разных параметрах даёт потрясающее многообразие великолепных спиралей (Рис. 22.17)!

```
//Рекурсивные спирали
procedure polySpi;
begin
  If (side > maxside) Then
    exit;
  turtle.Move(side);
  turtle.Turn(angle);
  side += incr;
  polySpi();
End;
```

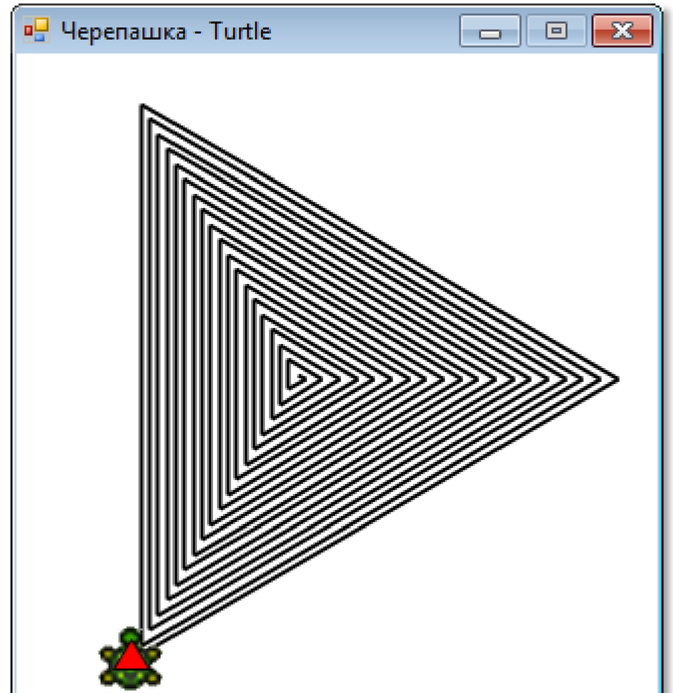
По очереди снимайте комментарии с кода и запускайте программу:

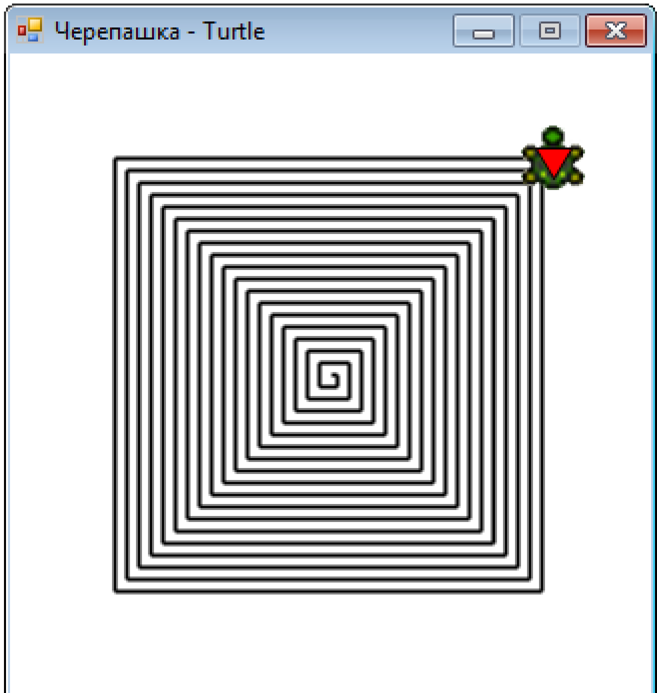
```
  turtle.clr:= Color.Black;
   turtle.w:= 2;
```

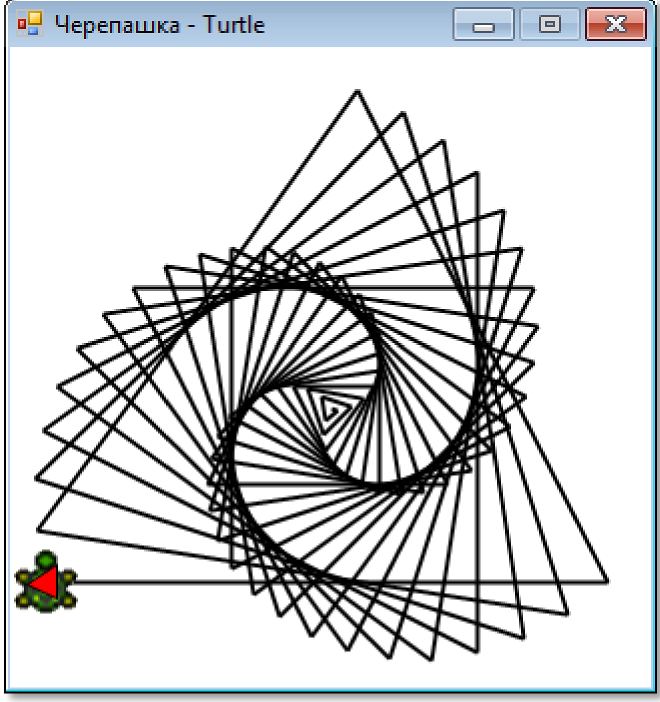
```
{
   turtle.X := CX-30;
   turtle.Y := CY+30;
   maxside:=240;
   incr:=1;
   side:=60;
   angle:=95;
   polySpi();
}
{
   //квадратная спираль:
   turtle.X := CX;
   turtle.Y := CY;
   maxside:=220;
   incr:= 3;
   angle:=90;
   //angle:=87;
   polySpi();
   }

{
   //треугольная спираль:
   incr:= 5;
   angle:=120;
   turtle.X := CX -20;
   turtle.Y := CY + 0;
   maxside:=280;
   polySpi();
   }
   {
   turtle.X := CX - 0;
   turtle.Y := CY + 20;
   incr:= 4;
   angle:=117;
   maxside:=280;
   polySpi();
   }

   turtle.X := CX - 0;
   turtle.Y := CY + 20;
   incr:= 1;
   angle:=30;
   side:= 3;
   maxside:=280;
```

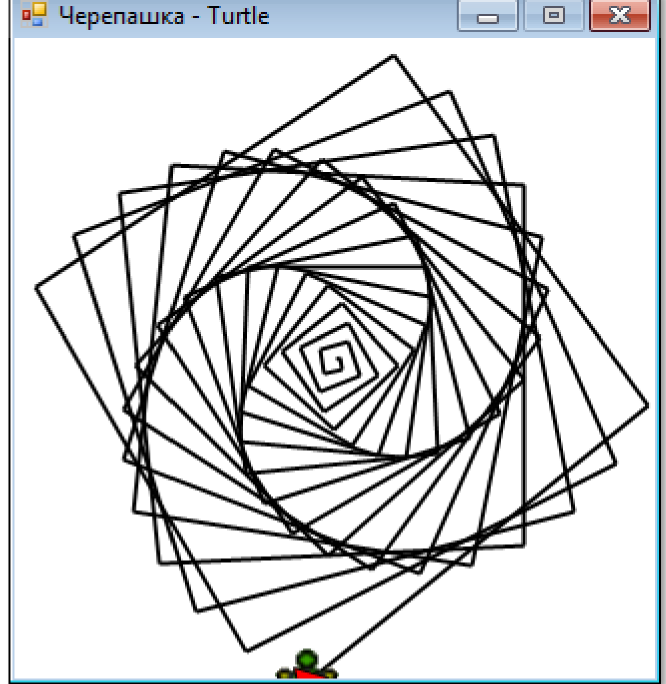
```
    polySpi();
```

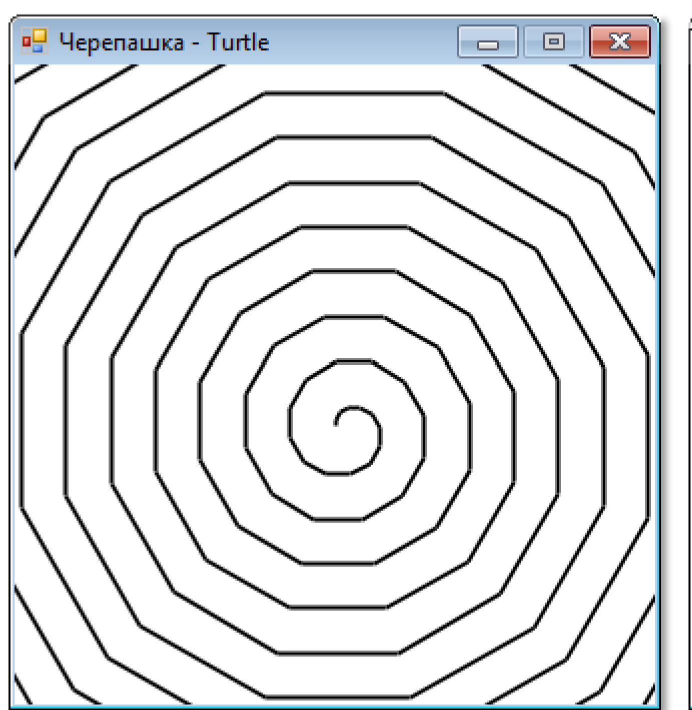
И последний проект этого урока – вычерчивание разнообразных *звёздчатых многоугольников* (Рис. 22.18).











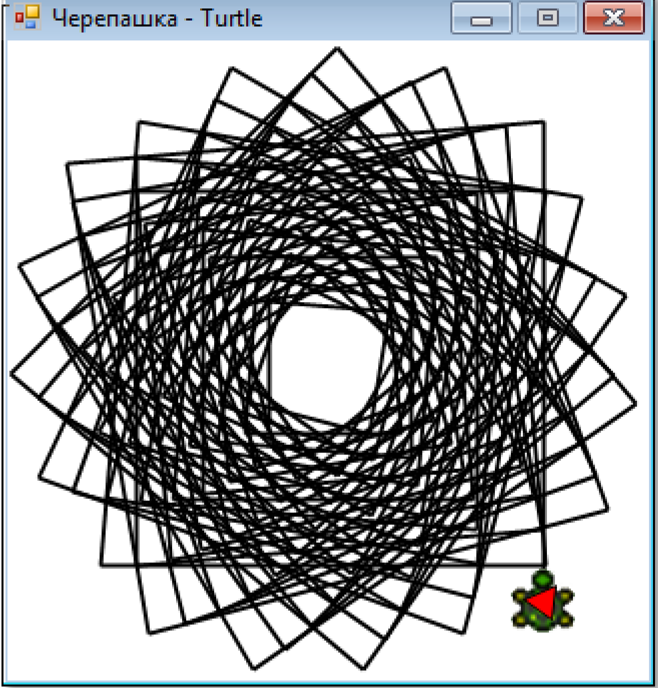
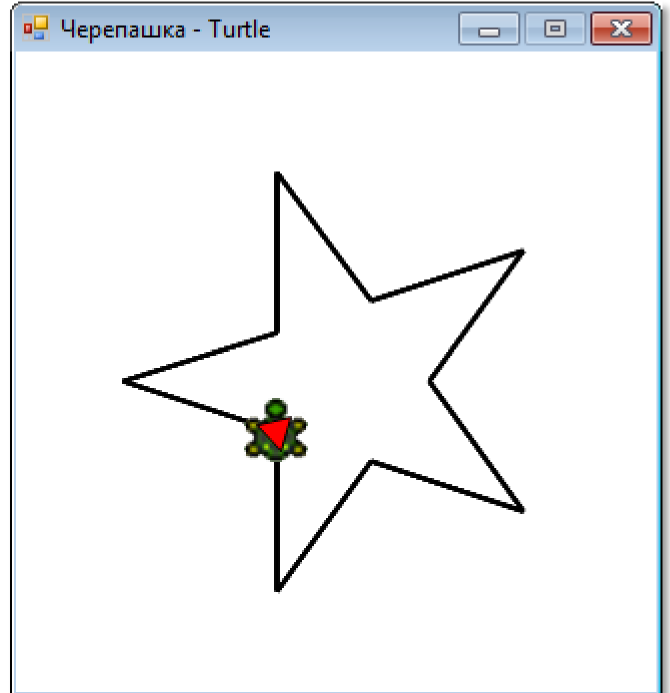


Рис. 22.17. Рекурсивные спирали

```
//Звездчатые многоугольники
procedure newPoly;
begin
  For var i:= 1 to 24 do begin
    turtle.Move(side);
    turtle.Turn(angle);
    turtle.Move(side);
    turtle.Turn(2*angle);
  end;//For
End;

  {
   turtle.X := CX-30;
   turtle.Y := CY-20;
   side:=80;
   //звезда:
   angle:=144;
   //angle:=45;
   newPoly();
  }
   //зубчатое колесо:
   turtle.X := CX-75;
   turtle.Y := CY-105;
   turtle.Angle:=0;
   side:=36;
```

```
angle:=125;
newPoly();
```

Количество зубцов определяется углом *angle*.



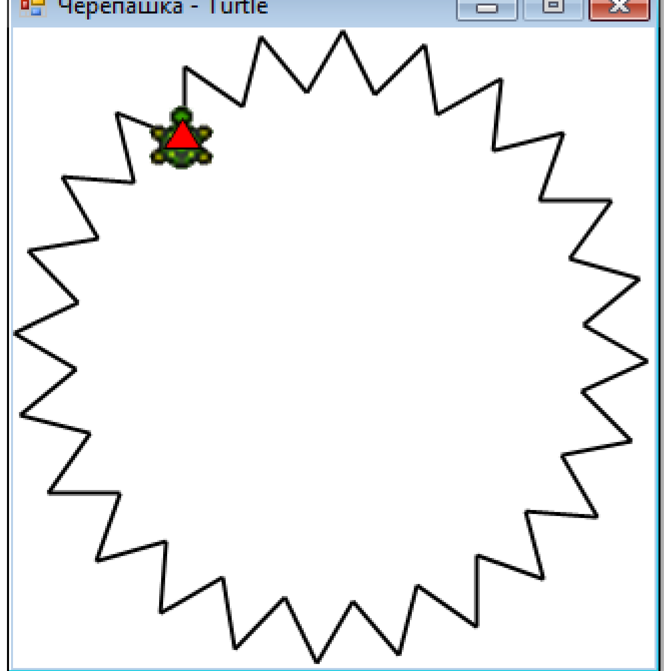


Рис. 22.18. Звёздчатые многоугольники

Исходный код программы находится в папке **Turtle**.

Напишите такие программы для *Черепашки*, чтобы она построила следующие фигуры (Рис. 22.19).

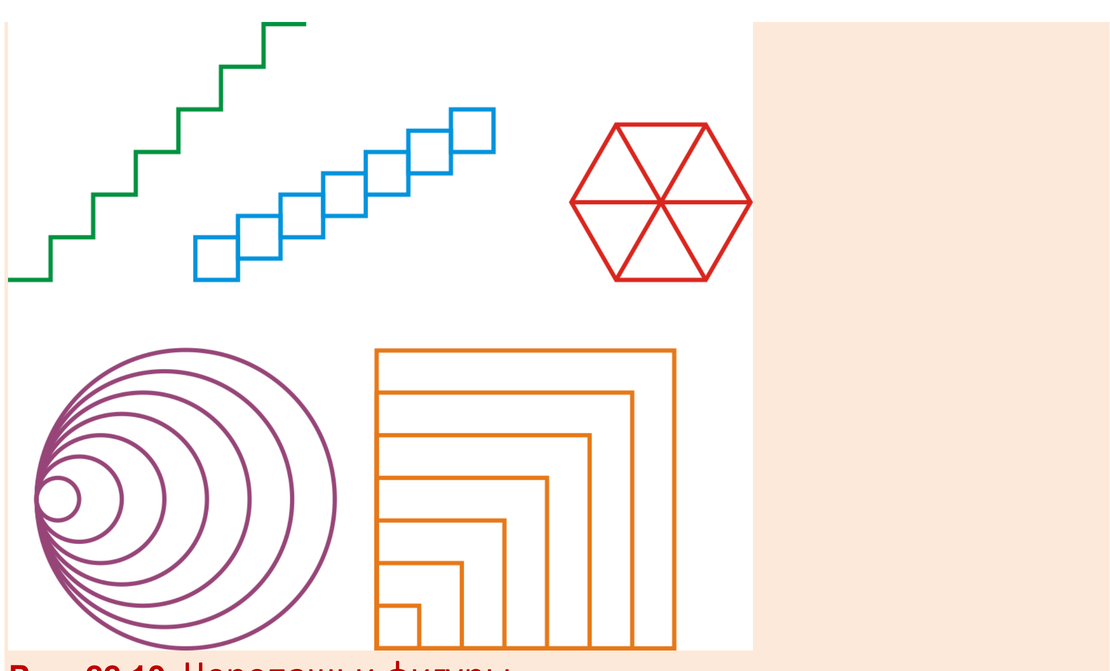


Рис. 22.19. Черепашьи фигуры

Урок 23. Фрактальная Киберчерепашка

На уроке *Черепашья графика* мы заставляли *Черепашку* выполнять наши команды, записанные на паскале. Но ведь в каждом живом существе, в его геноме записана информация о том, как оно должно расти и развиваться, а также инстинкты, управляющие поведением примитивных организмов. *Черепашка* также может выполнять не только внешнюю программу, но и внутреннюю – записанную в её геноме и реализуемую через инстинкты. Это урок мы и посвятим моделированию такой *кибернетической Черепашки*.

Начнём новый проект **Cyber-Turtle** с объявления переменных.

Черепашка рождается в условленном месте, обозначенном координатами *x0* и *y0*, а её голова направлена в сторону света, заданную начальным углом *a0*:

```
//ПРОГРАММА ДЛЯ МОДЕЛИРОВАНИЯ
//КИБЕРЧЕРЕПАШКИ

uses
   GraphABC, ABCObjects, u_turtle;

var
   turtle: CTurtle;

   //размеры окна:
   height, width: integer;
   //координаты центра окна:
   CX, CY: integer;

   //начальное положение Черепашки:
   x0, y0, a0: real;
```

Основное предназначение *Черепашки* (судьба) - перемещение по плоскости и вычерчивание линий, поэтому нам необходимо задать её основные *свойства*:

```
   //длина единичного перемещения Черепашки:
```

```
   size:=0;
   //угол единичного поворота Черепашки:
   teta:=0;
   //цвет линии:
   penColor:= Color.Red;
   //толщина линии:
   penWidth:=2;
   //список команд, которые должна выполнить Черепашка:
   instinct:='';
```

Здесь всё просто и понятно, кроме, пожалуй, *инстинкта*, который и определяет поведение *Черепашки*. Дальше мы зададим ей жизненную программу (инстинкт), и вы сможете сами программировать *Черепашку*.

Настройка *окна* приложения не должна вызвать у вас вопросов:

```
//Подготавливаем программу
procedure prepareProg();
begin
   SetWindowTitle('КиберЧерепашка - CyberTurt
   SetWindowWidth(320);
   Window.Height:= 320;
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   height := Window.Height;
   width := Window.Width;
   //координаты центра окна:
   CX:= width div 2;
   CY:= height div 2;
end;
```

Теперь мы вполне можем перейти к программированию инстинкта *Черепашки*. Пусть её жизненный путь начнётся с прямой линии. Задаём место рождения *Черепашки*:

```
turtle:= new CTurtle();
//устанавливаем Черепашку:
x0:= CX - 100;
y0:= CY + 100;
```

Её направление:

```
a0:= 0;
```

Длину «шага»:

```
size:=100;
```

И самое главное – инстинкт:

```
//прямая линия:
instinct:='F';
```

Процесс рождения *Черепашки* не зависит от самой *Черепашки*, а полностью лежит на совести её родителей, то есть на нас с вами:

```
//Черепашка занимает исходное положение
procedure start();
begin
   turtle.clr:= penColor;
   turtle.w:= penWidth ;
   turtle.PenDown();
   turtle.X:= x0;
   turtle.Y:= y0;
   turtle.Angle:= a0;
End;
```

В большинстве языков программирования, поддерживающих черепашью графику, движение *Черепашки* вперёд обозначается командой *Forward*. Первую букву этого слова мы будем использовать для той же цели. После того как *Черепашка* заняла исходное положение, она должна выполнить те действия, которые ей предписывает инстинкт. Для этого у неё имеется *мозг*, который умеет обрабатывать и исполнять команды инстинкта:

```
//Черепашка выполняет инстинкт
procedure execute;
begin
   //Черепашка считывает команды:
   For var i:= 1 To instinct.Length do begin
      //очередная команда:
      var cmd:= instinct[i];
```

```
     //двигаться вперед на один шаг:
     If (cmd='F') Then
        turtle.Move(size);
   End;//For
End;
```

Пока *Черепашка* маленькая, она может только двигаться вперёд, но и этого вполне достаточно, чтобы дать ей путёвку в жизнь:

```
  start();
  execute();
```

Запускаем программу, и *Черепашка* проводит вертикальный отрезок длиной в *size* пикселей (Рис. 23.1).

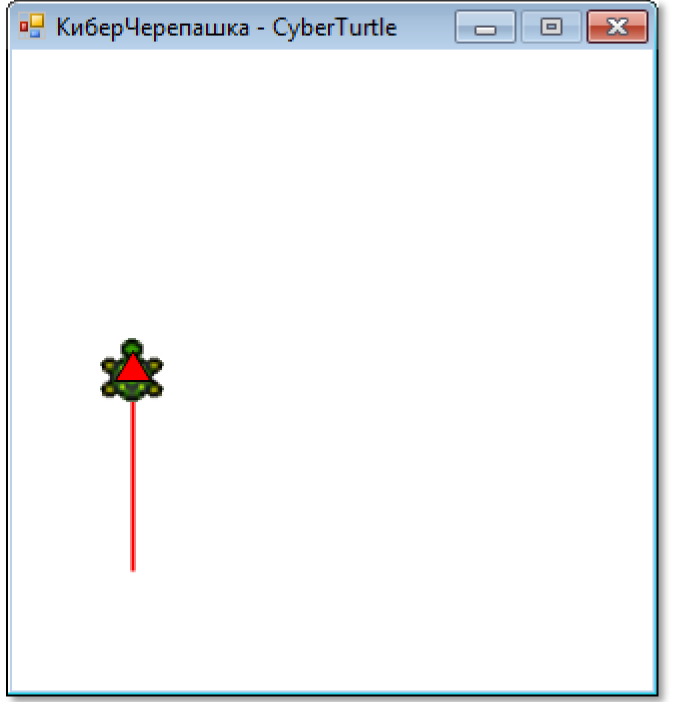


Рис. 23.1. *Черепашка* в ползунковом возрасте

На следующем этапе своей черепашьей жизни наша *Киберчерепашка* сможет выполнять команды *+* и -, то есть поворачиваться направо и налево на заданный угол *teta*. В её окрепшем мозгу появятся новые шишки для обработки более сложного инстинкта:

```
If (cmd='+') Then
    turtle.Turn(teta);
If (cmd='-') Then
```

```
    turtle.Turn(-teta);
```

Треугольник мы уже вычерчивали с помощью обычной *Черепашки*. Как вы помните, для этого *Черепашка* должна вычертить одну сторону, повернуться на угол 120 градусов, вычертить вторую сторону, опять повернуться на 120 градусов и, наконец, вычертить третью сторону. Отсюда следует, что угол единичного поворота равен 120 градусам, а инстинкт запишется строкой:

```
//треугольник:
teta:=120;
instinct:='F+F+F+';
```

Для проверки нашей гипотезы запускаем программу. *Черепашка* нас не подвела – треугольник удался на славу (Рис. 23.2)!

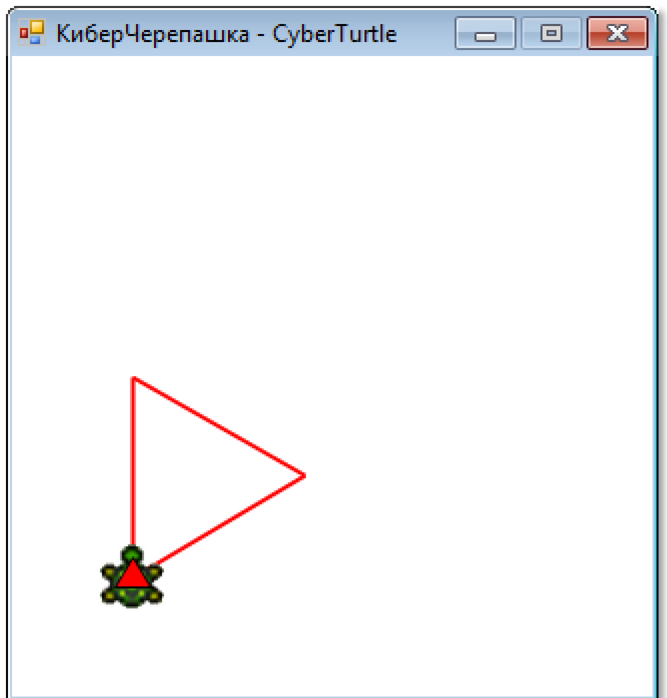


Рис. 23.2. Черепашковый треугольник

Совершенно аналогично мы можем сформировать инстинкты для вычерчивания любых правильных *многоугольников:*

```
//квадрат:
teta:=90;
instinct:='F+F+F+F+';
```

```
//пятиугольник:
teta:=72;
instinct:='F+F+F+F+F+';

//шестиугольник:
teta:=60;
instinct:='F+F+F+F+F+F+';
```

Как вы видите, инстинкт становится всё более длинным, при этом одна и та же пара команд *F+* просто повторяется *n* раз, если через *n* обозначить число вершин правильного многоугольника. Давайте научимся выращивать инстинкты!

Пусть при рождении *Черепашка* получает простейший инстинкт, который принято называть *аксиомой*, а затем, с каждым годом инстинкт развивается и становится всё более сложным благодаря *правилам* роста. Для треугольника и других правильных многоугольников аксиома самая простая:

```
axiom:='F';
```

Она означает, что *Черепашка* должна двигаться вперёд на расстояние *size*. Наша первая *Черепашка* только это и умеет делать.

Поскольку все многоугольники строятся по одному и тому же сценарию, то и правило для них одно и то же:

```
newF:='F+F';
```

А действует оно так. Через год (возраст *Черепашки* определяется переменной *iter*) каждая команда *F*, в соответствии с правилом роста, заменяется значением переменной *newF*, то есть *F+F*:

iter= 1 → `axiom:='F+F';`

Это значит, что через год (а у *Черепашек* год короткий!) наша *Черепашка* научится вычерчивать *угол* (Рис. 23.3).

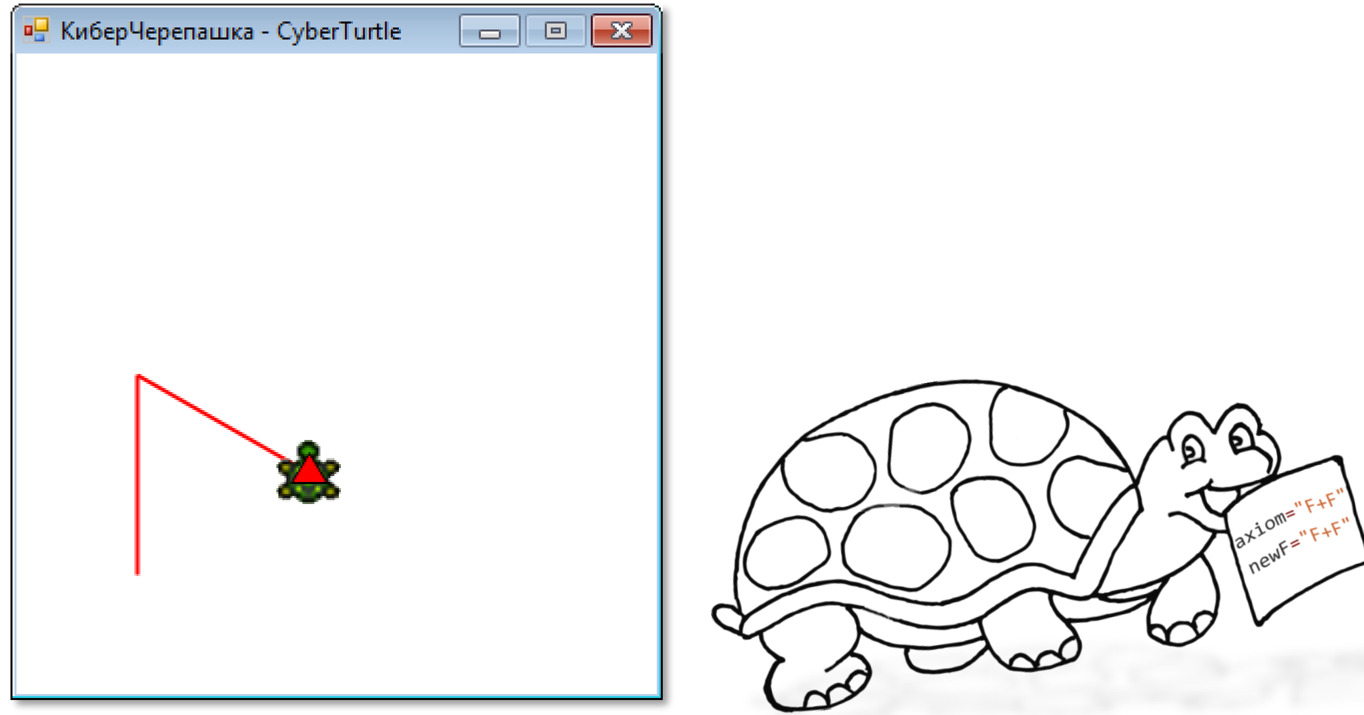


Рис. 23.3. Молодая «угловатая» *Черепашка*

На второй год с поведение *Черепашки* усложнится:

iter= 2 → `axiom='(F)+(F)'` → `'(F+F)+(F+F)'` → `'F+F+F+F'`

Опять в аксиоме каждая буква *F* заменяется выражением *F+F*. Для удобства преобразований мы воспользовались скобками, но самой *Черепашке* они не нужны.

Вот теперь *Черепашка* может вычертить настоящий *треугольник* (Рис. 23.4).

Вы, должно быть, заметили, что для построения треугольника достаточно инстинкта `'F+F+F+'`, то есть одну сторону *Черепашка* чертит дважды. Увы, все инстинкты несовершенны, но зато достаточно просты.

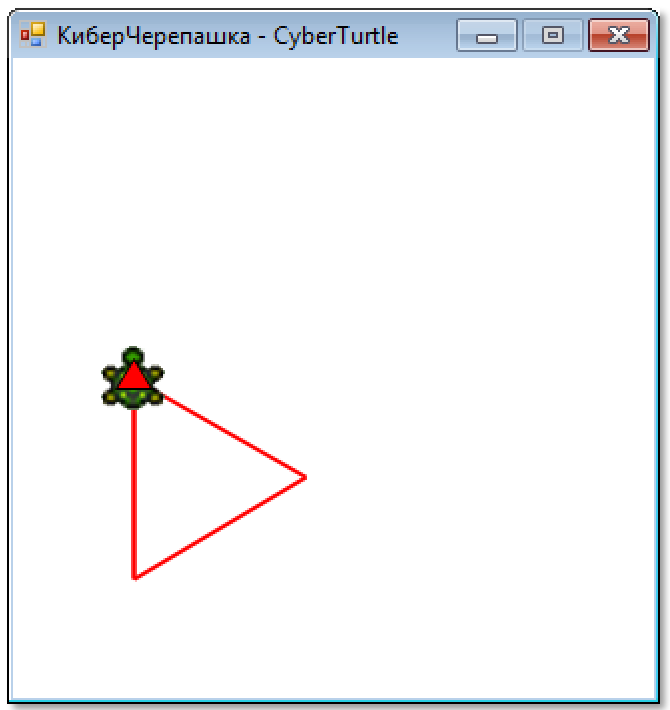


Рис. 23.4. Двухлетняя *Черепашка* умеет чертить треугольник

Инстинкт *Черепашки* формируется, конечно, не вручную - она имеет для этого в мозгу специальную *процедуру*:

```
//Черепашка формирует инстинкт
procedure createInstinct;
begin
   For var i:= 1 to iter do begin
     instinct:='';
     For var j:= 1 To axiom.Length do begin
        //очередная команда аксиомы:
        var cmd:= axiom[j];
        If (cmd='F') Then
            instinct += newF
        Else
            instinct += cmd;
     end;//For
     axiom:=instinct;
   end;//For
end;
```

Её действие мы уже разобрали. Каждая буква *F* заменяется значением переменной *newF*, а все остальные символы (команды поворота + и -) переходят в новый инстинкт без изменения. Так как мы привыкли использовать при моделировании поведения

Черепашки переменную *instinct*, то результатом работы процедуры *createInstinct* пусть будет значение этой переменной, хотя мы могли бы заменить её и переменной *axiom*.

Выделим «треугольный» инстинкт в отдельную процедуру

```
//Треугольник:
procedure Triangle;
begin
  teta:=120;
  size:=100;
  iter:= 2;
  axiom:='F';
  newF:='F+F';
end;
```

и запустим программу:

```
Triangle();
createInstinct();
start();
execute();
```

Мы научились формировать простейший инстинкт, повинуясь которому *Черепашка* рисует треугольник. Достаточно изменить значение угла *teta,* и новый инстинкт – для построения *квадрата* – готов:

```
//Квадрат:
procedure Quadrat;
begin
  teta:=90;
  size:=100;
  iter:= 2;
  axiom:='F';
  newF:='F+F';
end;
```

На третий год жизни *Черепашка* сумеет начертить пятиугольник, шестиугольник и восьмиугольник (Рис. 23.5).

```
//Пятиугольник:
procedure Pentagon;
begin
  teta:=72;
  size:=100;
  iter:= 3;
  axiom:='F';
  newF:='F+F';
end;

//Шестиугольник:
procedure Hexagon;
begin
  teta:=60;
  size:=100;
  iter:= 3;
  axiom:='F';
  newF:='F+F';
end;

//Восьмиугольник:
procedure Octagon;
begin
  teta:=45;
  size:=100;
  iter:= 3;
  axiom:='F';
  newF:='F+F';
end;
```

Мы могли бы продолжать этот процесс и дальше, но совершенно понятно, что с годами *Черепашка* сможет выстраивать любые правильные многоугольники, поэтому давайте усложним поведение *Черепашки* с помощью более изощрённых инстинктов и научим её чертить *фракталы*.

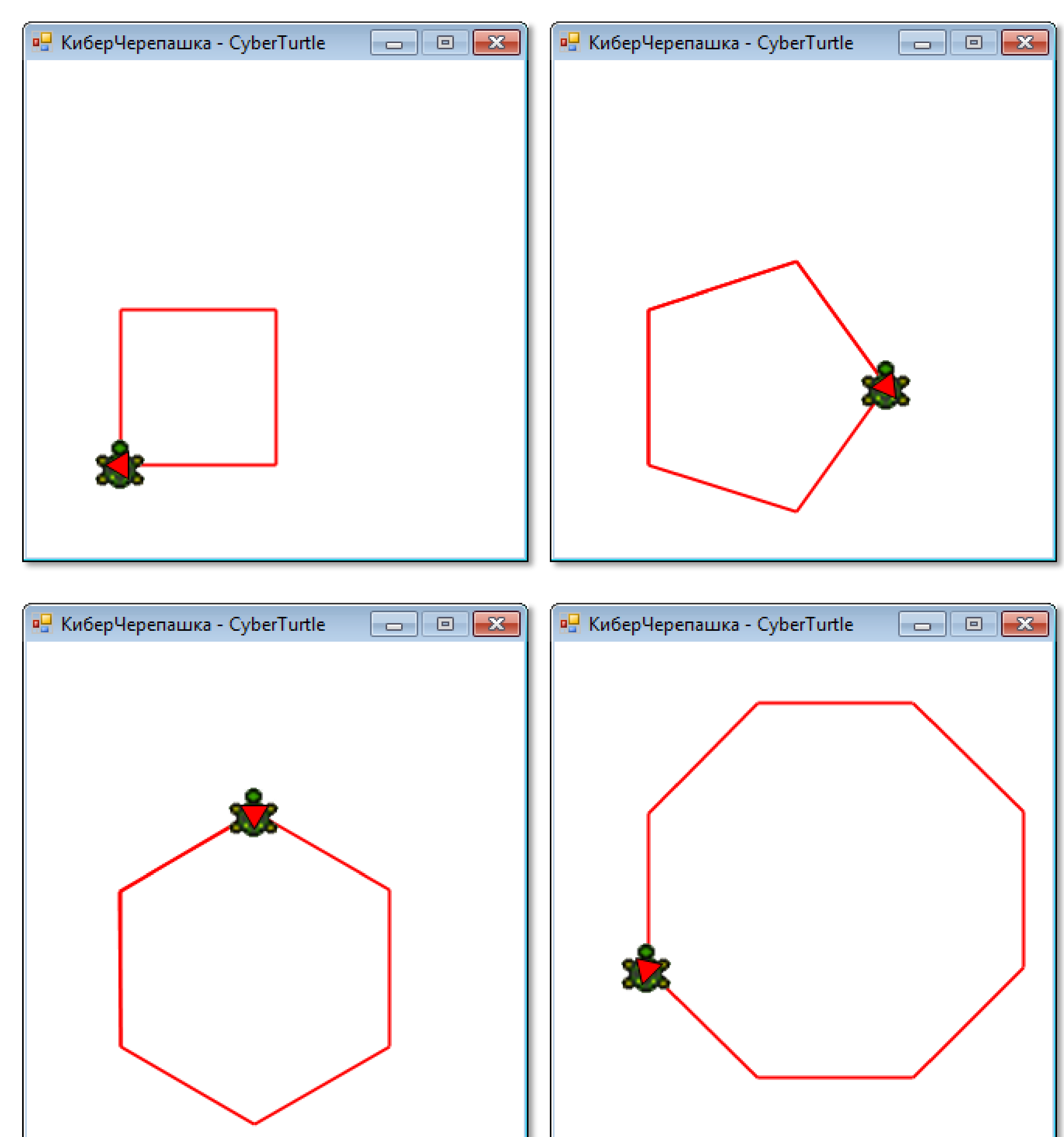


Рис. 23.5. Черепашковые многоугольники

Фракталы

Фракталы – это замечательные геометрические фигуры, составленные из точно таких же фигур, но меньшего размера. Напри-

мер, из маленьких отрезков мы можем составить отрезок большей длины, из него – еще более длинный, и так до бесконечности (Рис. 23.6).

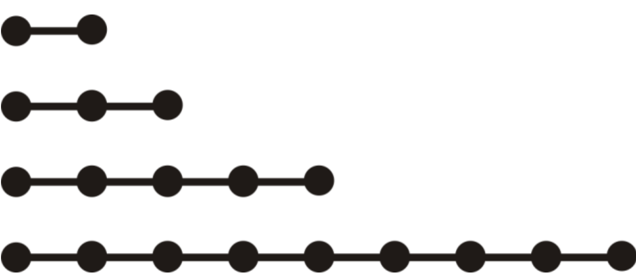


Рис. 23.6. Фрактальные отрезки

Аналогично из квадратиков мы можем построить большой квадрат (Рис. 23.7).

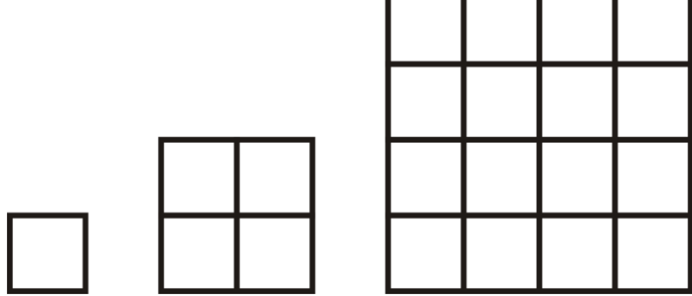


Рис. 23.7. Фрактальные квадраты

Примерами фракталов в природе могут служить: кровеносная система, береговая линия, горный рельеф, перистые облака (Рис. 23.8), разряд молнии, системы рек с притоками, трещины в почве, ветвистые растения, фьорды, прожилки на листьях (Рис. 23.9), ледяные узоры на стёклах…

Рис. 23.8. Перистые облака

Рис. 23.9. Жилкование листа

Снежинка Коха

Более сложный пример фрактальной кривой придумал в 1904 году *Гельг фон Кох*. Построение начинается с отрезка, который можно условно разделить на три равные части (Рис. 23.10а). Среднюю часть отрезка мы заменяем двумя отрезками, каждый из которых равен трети исходного отрезка (Рис. 23.10б). Затем мы повторяем эту операцию сколько угодно раз (Рис. 23.10в).

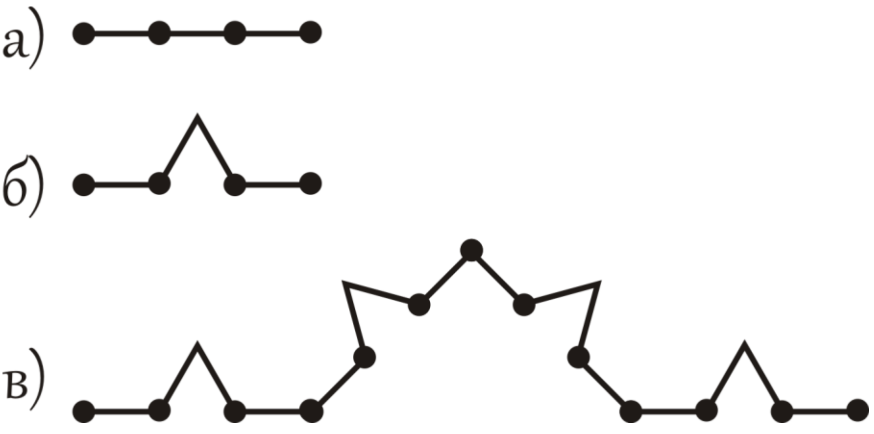


Рис. 23.10. Построение снежинки Коха

Если вы и дальше продолжите построение кривой Коха, то быстро убедитесь, что вручную её строить весьма утомительно. Поэтому лучше научить этому занятию нашу *Черепашку*, только строить она будет не кривую, а *снежинку Коха*, которая отличается от кривой только тем, что её звенья не выстраиваются в прямую линию, а поворачиваются на 60 градусов, вследствие чего кривая становится замкнутой. Итак, с углом *teta* мы определились. С аксиомой тоже можно справиться, а вот правило, порождающее инстинкт придётся позаимствовать у самого господина Коха:

```
//Снежинка Коха:
procedure Koch;
begin
  teta:=60;
  size:=40;
  iter:= 1;

  size:= 3;
  iter:= 4;
  axiom:='F++F++F';
  newF:='F-F++F-F';
end;
```

Запускаем программу:

```
x0:= CX - 70;
Koch;
createInstinct();
start();
```

На первом году жизни *Черепашка* нарисует шестиконечную звезду, в которой легко узнать кривую Коха, повторённую 3 раза (Рис. 23.12, слева).

Вообще говоря, настоящие снежинки также имеют 6 лучей, но более сложной формы (Рис. 23.11).

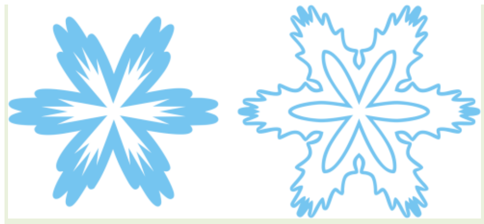
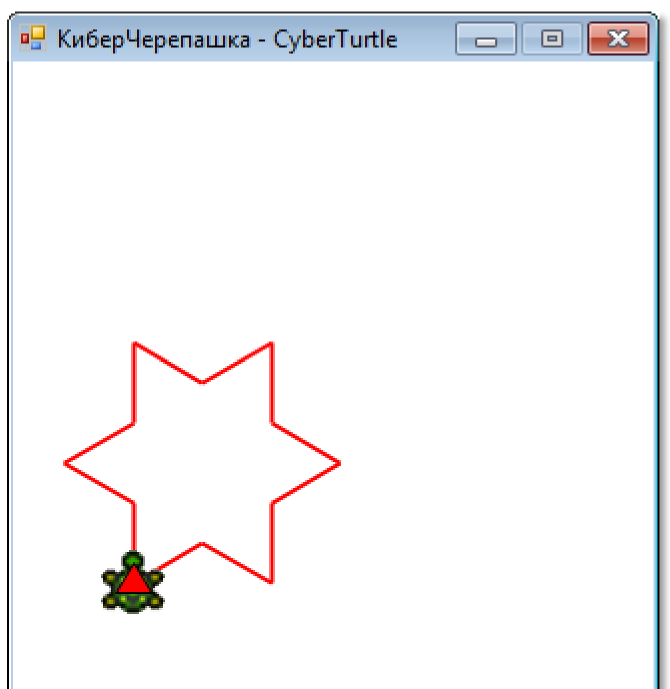


Рис. 23.11. Стилизованные снежинки



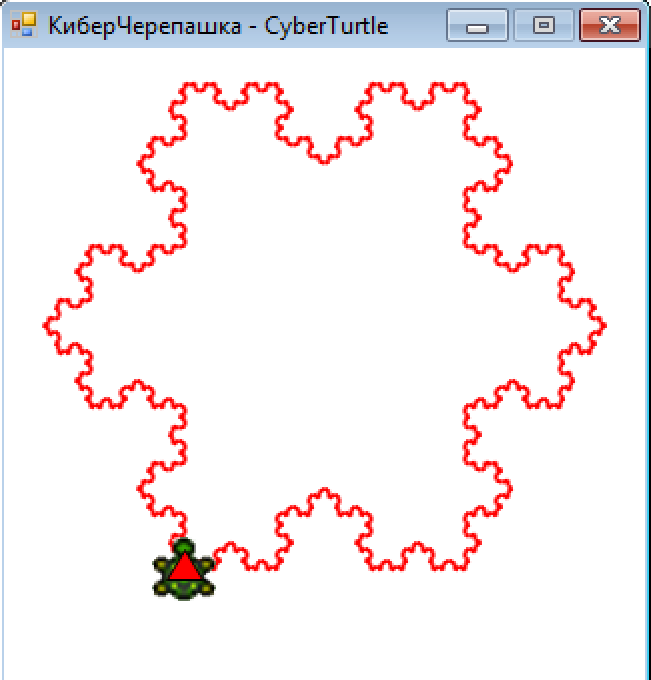


Рис. 23.12. Кривая Коха при *iter=1* и *iter=4*

С возрастом *Черепашка* рисует всё более и более сложные узоры, так что в четырёхлетнем возрасте снежинка у неё получается не хуже настоящей (Рис. 23.12, справа)!

В 1890 году *Джузеппе Пеано* построил функцию, график которой, названный *кривой Пеано*, покрывает квадратами (или, если угодно, ромбами) всю плоскость. И хотя кривая Пеано не является фракталом в полном смысле этого слова, но всё равно очень забавно наблюдать, как наша *Черепашка* её вычерчивает (Рис. 23.13). С годами (то есть с увеличением значения *iter*) она «квадрирует» все б***о***льшую и б***о***льшую поверхность.

```
//Кривая Пеано:
procedure Peano;
begin
```

```
  a0:= 45;
  teta:=90;
  size:=10;
  iter:= 3;
  axiom:='F';
  newF:='F-F+F+F+F-F-F-F+F';
end;

x0:= CX-120;
y0:= CY+80;
Peano;
```

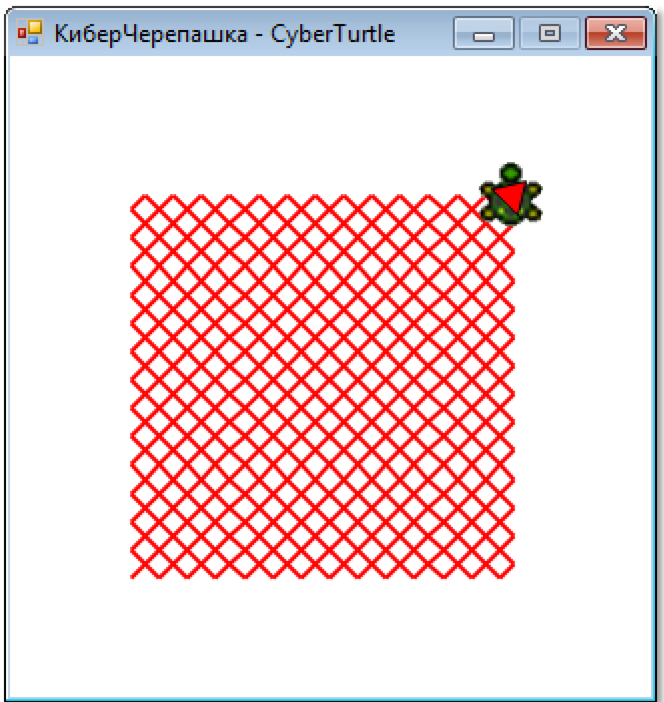


Рис. 23.13. Кривая Пеано при *iter= 3*

Ещё более усложнив инстинкт, мы научим *Черепашку* выделывать очень сложную кривую (Рис. 23.14).

```
//32-сегментная кривая:
procedure sc32;
begin
  teta:=90;
  size:= 2.5;
  iter:= 2;
  axiom:='F+F+F+F';
  newF:='-F+F-F-F+F+FF-F+F+FF+F-F-FF+FF-FF+F+F-FF-F-F+FF-F-
F+F+F-F+';
```

```
end;

Window.Width:= 400;
Window.Height:= 400;
x0:= CX - 40;
y0:= CY + 110;
penWidth:=1;
penColor:= Color.Black;
turtle.Hide;
sc32;
```

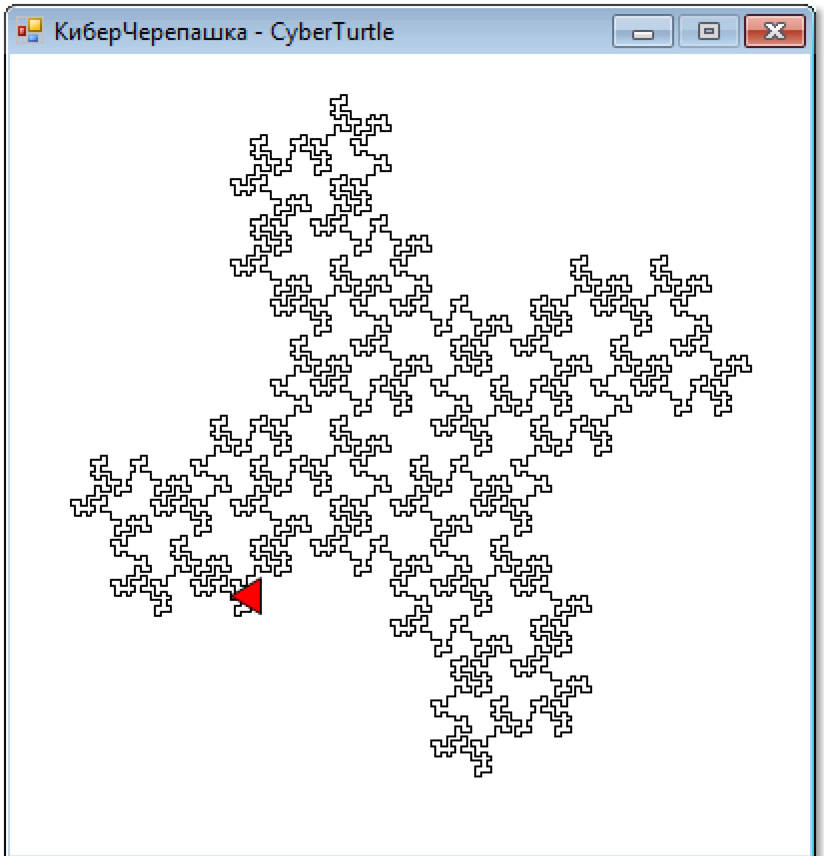


Рис. 23.14. 32-сегментная кривая при *iter= 2*

Добавим к инстинкту *Черепашки* новую команду, которая в генетическом коде обозначается буквой **b**. Получив такую команду, *Черепашка* переползает вперёд, но линию за собой не чертит.

Для обработки нового гена мы добавим в процедуру *createInstinct* пару строк:

```
. . .
else If (cmd='b') Then
```

```
     instinct += newb;
. . .
```

То есть новый ген *b* входит в состав инстинкта аналогично гену *F*. А вот в процедуре *execute Черепашка* должна сначала поднять карандаш, а после перемещения снова опустить его:

```
     else If (cmd='b') Then begin
        turtle.PenUp();
        turtle.Move(size);
        turtle.PenDown();
     End;
```

С помощью этого гена *Черепашка* сможет рисовать фигуры, состоящие из нескольких не связанных между собой частей (Рис. 23.15).

```
//Мозаика:
procedure Mozaic;
begin
  a0:= 0;
  teta:=90;
  size:= 4;
  x0:= CX+70;
  y0:= CY+70;
  iter:= 2;
  axiom:='F-F-F-F';
  newF:= 'F-b+FF-F-FF-Fb-FF+b-FF+F+FF+Fb+FFF';
  newb:= 'bbbbbb';
end;
```

Обратите внимание: каждое вхождение нового гена *b* в инстинкт заменяется значением переменной *newb*!

Мы можем добавлять к инстинкту *Черепашки* и другие гены, которые обозначаются буквами латинского алфавита. Они участвуют в формировании инстинкта, но при его выполнении игнорируются.

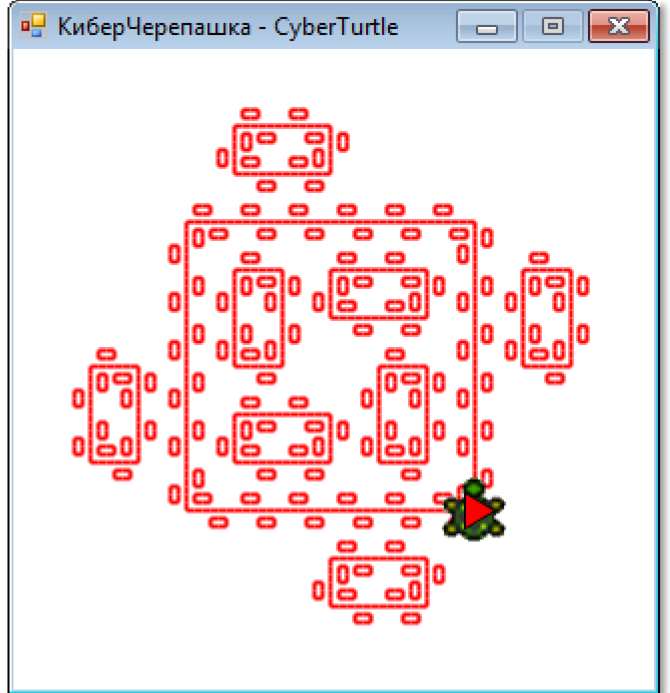


Рис. 23.15. Мозаика при *iter*= 2

Для каждого нового гена должно быть записано правило, по которому он заменяется в инстинкте другими генами. Нашей *Черепашке* будет достаточно четырёх неисполняемых генов, которые формируют инстинкт в процедуре *createInstinct*:

```
        else If (cmd='W') Then
             instinct += newW
        else If (cmd='X') Then
             instinct += newX
        else If (cmd='Y') Then
             instinct += newY
        else If (cmd='Z') Then
             instinct += newZ
```

В процедуру *execute* никаких изменений вносить не следует, поскольку эти команды *Черепашка* не выполняет. Зато новые гены могут настолько усложнить инстинкт *Черепашки*, что она начнёт вычерчивать кривые удивительной красоты!

Помните, как на уроке [*Геометрические фантазии*] мы построили *Треугольный треугольник* (см. *Рис. 21.7*)? – А теперь мы обучим этому искусству нашу *Киберчерепашку*:

```
//Треугольники:
procedure triangles;
begin
  a0:= 30;
  teta:=120;
  size:= 18;
  x0:= CX-28;
  y0:= CY+28;
  iter:= 6;
  axiom:='bX';
  newb:= 'b';
  newF:='F';
  newX:='--FXF++FXF++FXF--';
end;
```

Хотя задача оказалась непростой, но она справилась с заданием отлично (Рис. 23.16).

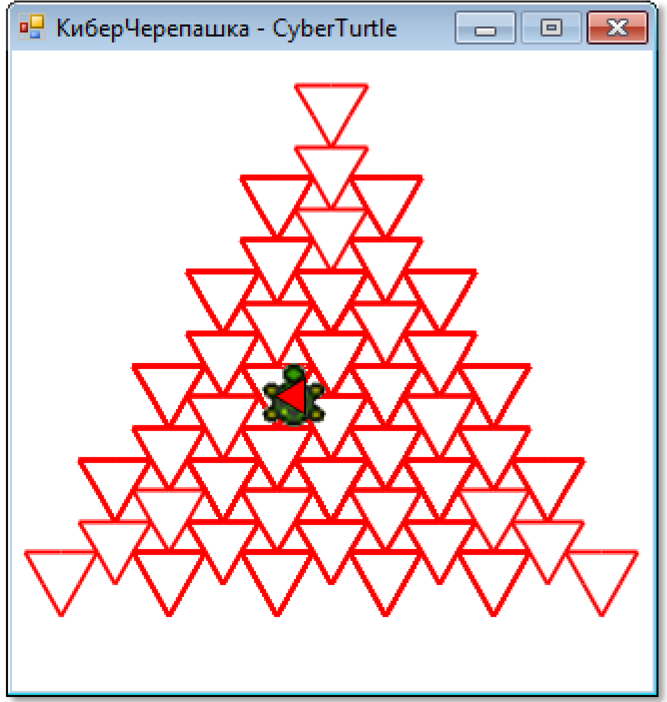


Рис. 23.16. Треугольники при *iter= 6*

С двумя новыми генами *Черепашка* построит кривую *Пеано-Госпера*, которая, как и кривая Коха, также напоминает снежинку, но заполненную внутри замысловатыми линиями (Рис. 23.17).

```
//Кривая Пеано-Госпера:
```

```
procedure PeanoGosper;
begin
  a0:= 0;
  x0:= CX-140;
  y0:= CY+58;
  teta:=60;
  size:=10;
  iter:= 4;
  axiom:='FX';
  newF:='F';
  newX:='X+YF++YF-FX--FXFX-YF+';
  newY:='-FX+YFYF++YF+FX--FX-Y';
end;
```

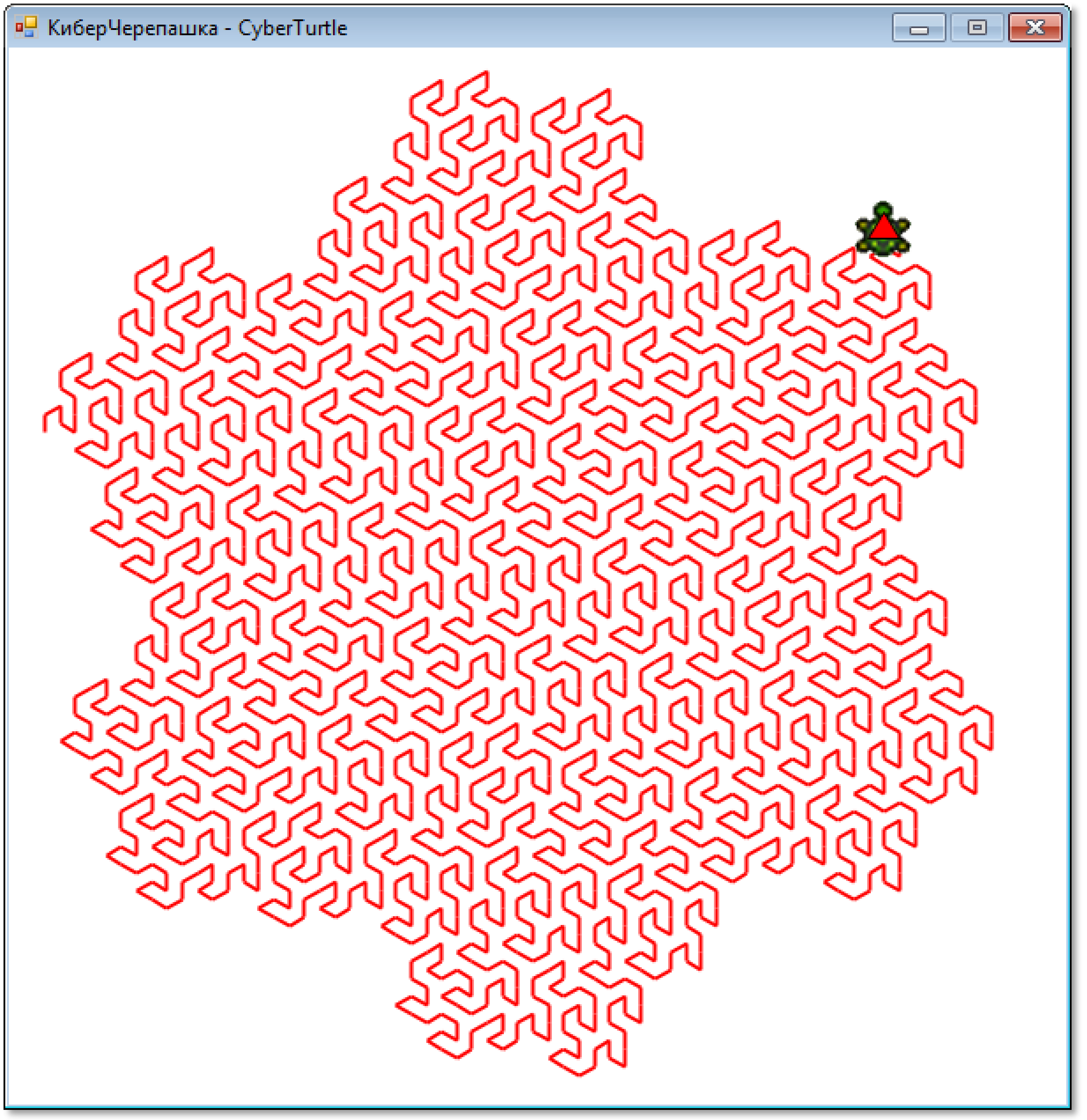


Рис. 23.17. Кривая Пеано-Госпера при *iter= 4*

А вот чтобы научить *Черепашку* рисовать почти настоящую снежинку, нам придётся наделить её *памятью*. По команде **[** она будет запоминать свои координаты и угол поворота, а по команде **]** возвращаться в это состояние.

Поскольку эти гены не влияют на формирование инстинкта, то процедура *createInstinct* останется без изменений, а в процедуру *execute* нужно добавить строчки:

```
    else If (cmd='[') Then begin
       st.Push(turtle.Angle);
       St.Push(turtle.X);
       St.Push(turtle.Y);
    End
    else If (cmd=']') Then begin
       turtle.Y:= St.Pop();
       turtle.X:= St.Pop();
       turtle.Angle:= St.Pop();
    End;
```

Как видите, память *Черепашки* устроена по принципу *стека*: *Черепашка* вспоминает *последние* запомненные события, после чего они стираются из её памяти.

В папке *c:\PABCWork.NET\Samples\MainFeatures\06_Classes* вы найдёте файл *Stack.pas*, который следует скопировать в наш проект. Затем нужно создать *переменную* типа стека:

```
st: Stack<real>;
```

И наконец, создать стек:

```
st := new Stack<real>;
```

Теперь *Черепашка* не только умеет выполнять команды инстинкта, но и обладает памятью. А вот так она строит почти настоящую снежинку (Рис. 23.18):

```
// Снежинка:
procedure Snowflake;
begin
```

```
  a0:= 0;
  size:=10;
  iter:= 2;
  teta:=60;
  x0:= CX;
  y0:= CY+100;
  axiom:='[F]+[F]+[F]+[F]+[F]+[F]';
  newF:='F[++F][-FF]FF[+F][-F]FF';
end;
```

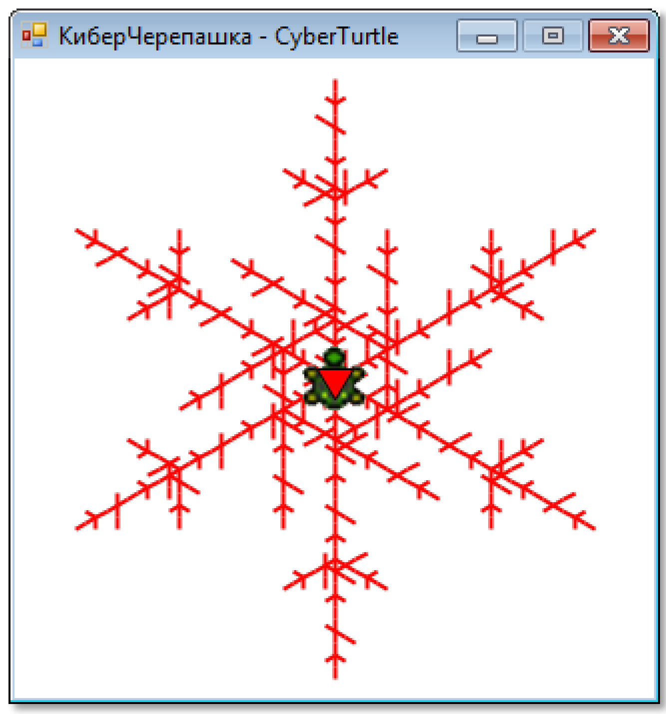


Рис. 23.18. Снежинка при *iter= 2*

С помощью очень простого инстинкта и памяти *Черепашка* нарисует очень красивую картинку, состоящую из ромбов (Рис. 23.19):

```
//Ромбы:
procedure Rhombes;
begin
  a0:= 0;
  teta:=60;
  size:=10;
  iter:= 6;
  x0:= CX;
  y0:= CY-136;
  axiom:='F';
  newF:='-F+F+[+F+F]-';
end;
```

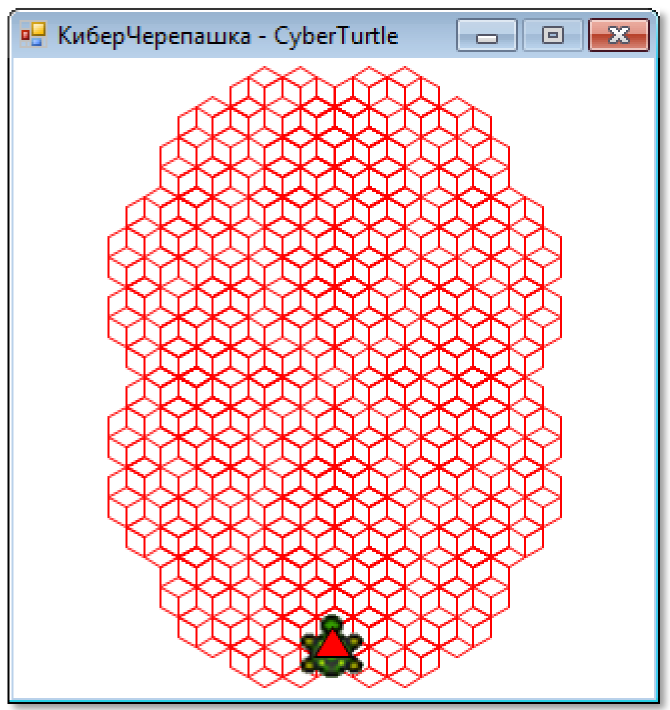


Рис. 23.19. Ромбы при *iter= 6*

Кривая Пенроуза (Рис. 23.20) также состоит из ромбов, но для её построения *Черепашке* понадобится и очень сложный инстинкт, и крепкая память:

```
//Кривая Пенроуза:
procedure Penrose;
begin
  a0:= 0;
  x0:= CX;
  y0:= CY;
  teta:=36;
  size:= 20;
  iter:= 4;
  axiom:='[Y]++[Y]++[Y]++[Y]++[Y]';
  newW:='YF++ZF----XF[-YF----WF]++';
  newX:='+YF--ZF[---WF--XF]+';
  newY:='-WF++XF[+++YF++ZF]-';
  newZ:='--YF++++WF[+ZF++++XF]--XF';
  newF:='';
end;
```

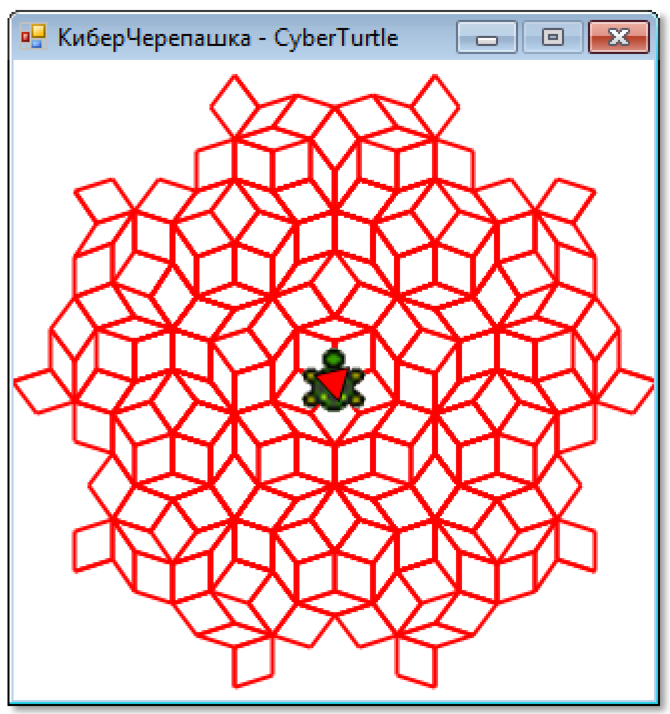


Рис. 23.20. Кривая Пенроуза при *iter= 4*

О *мозаиках Пенроуза* вы можете прочитать в книге Мартина Гарднера **[8]**.

Чтобы *выбраться* из *Лабиринта* (Рис. 23.21) также нужна хорошая память, а вот чтобы *вычертить* замысловатый лабиринт, память *Черепашке* не понадобится:

```
//Лабиринт:
procedure Lab;
begin
   a0:= 0;
   x0:= CX+230;
   y0:= CY+230;
   Teta := 90;
   Size := 6;
   iter := 3;
   axiom := '-V';
   newF := 'F';
   newV := 'VFVF+WFW+FVFV-FWF-VFV-FW+F+WF-VFV-FWFWFW+';
   newW := '-VFVFVF+WFW+FV-F-VF+WFW+FVF+WFWF-VFV-FWFW';
end;
```

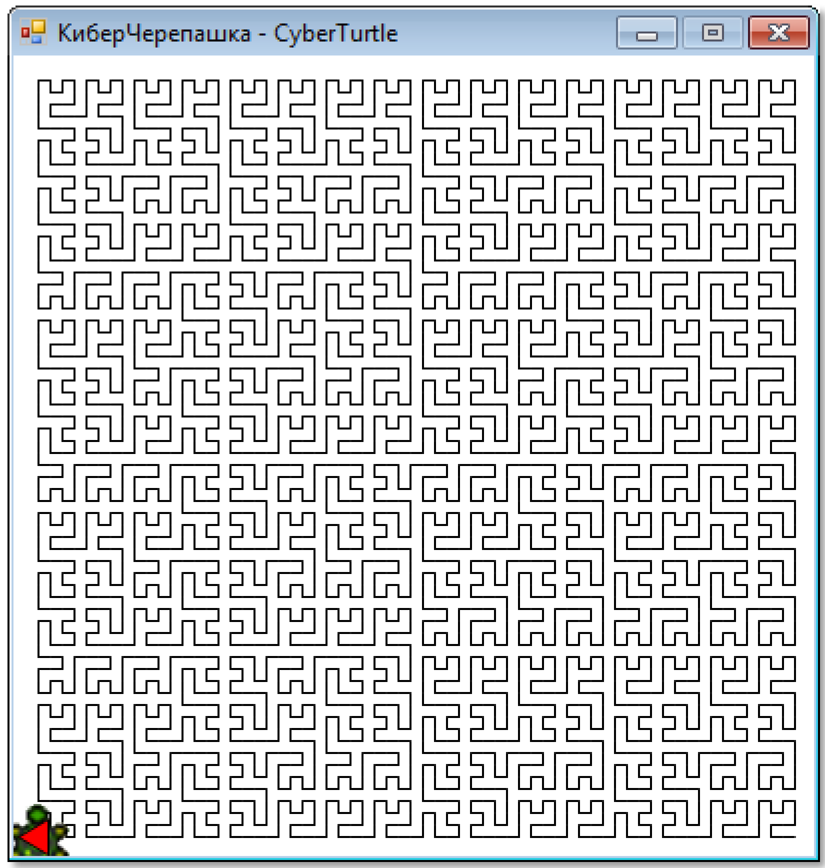


Рис. 23.21. Лабиринт при *iter*= 3

Кривая, которую мы назвали *Лабиринт*, очень похожа на знаменитую *кривую Гильберта* (*Hilbert Curve*), которая относится к семейству кривых, заполняющих всю плоскость. Если представить плоскость в виде прямоугольной сетки, то кривая Гильберта проходит через все вершины сетки размерами $(2^n - 1) \times (2^n - 1)$ клеток.

Эту кривую описал в 1891 году немецкий математик Давид Гильберт как вариант кривой Джузеппе Пеано, открытой годом ранее. В настоящее время эту фрактальную кривую используют при обработке изображений - для их сжатия и растрирования.

Поскольку кривая Гильберта очевидно *рекурсивная*, то для её построения обычно прибегают к рекурсивным процедурам, которые имеют очень небольшой размер. В книге Никлауса Вирта (который, как вы помните, разработал язык паскаль) *Алгоритмы + структуры данных = программы*, в разделе *3.3. Два примера рекурсивных программ* описан один из таких алгоритмов (он довольно длинный, но зато простой для понимания). Имеются и

нерекурсивные алгоритмы вычерчивания кривой Гильберта, но нам, естественно, хотелось бы поручить это непростое дело нашей фрактальной *Киберчерепашке*. И она справляется с ним без проблем. Её инстинкт получился очень коротким, зато кривая вышла на загляденье (Рис. 23.22)!

```
//Hilbert Curve
procedure Hilbert;
begin
   a0:= 0;
   x0:= CX+230;
   y0:= CY+230;
   Teta := 90;
   Size := 6;
   iter := 3;
   axiom := 'V';
   newF := 'F';
   newV := '-WF+VFV+FW-';
   newW := '+VF-WFW-FV+';

end;
```

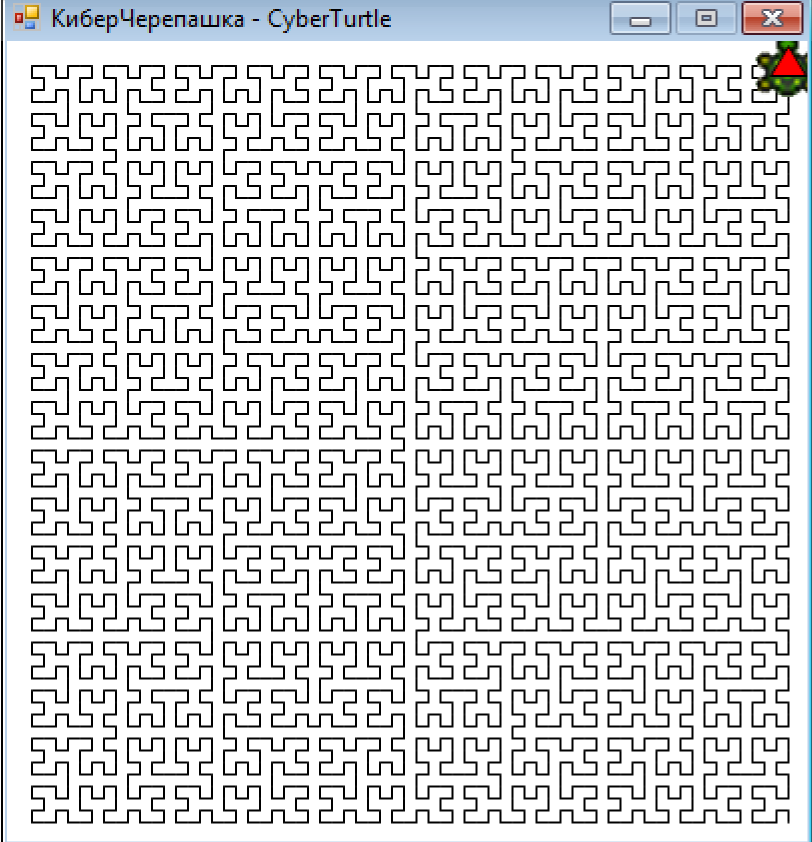


Рис. 23.22. Кривая Гильберта при *iter= 6*

И последнее, чему мы научим нашу *Черепашку*, - это строить кривую, которая очень напоминает настоящее *растение* (Рис. 23.23):

```
//Куст:
procedure Bush;
begin
  size:= 7.6;
  iter:= 4;
  teta:=180/8;
  x0:= CX+80;
  y0:= CY+160;
  axiom:='F';
  newF:='-F+F+[+F-F-]-[-F+F+F]';
end;
```

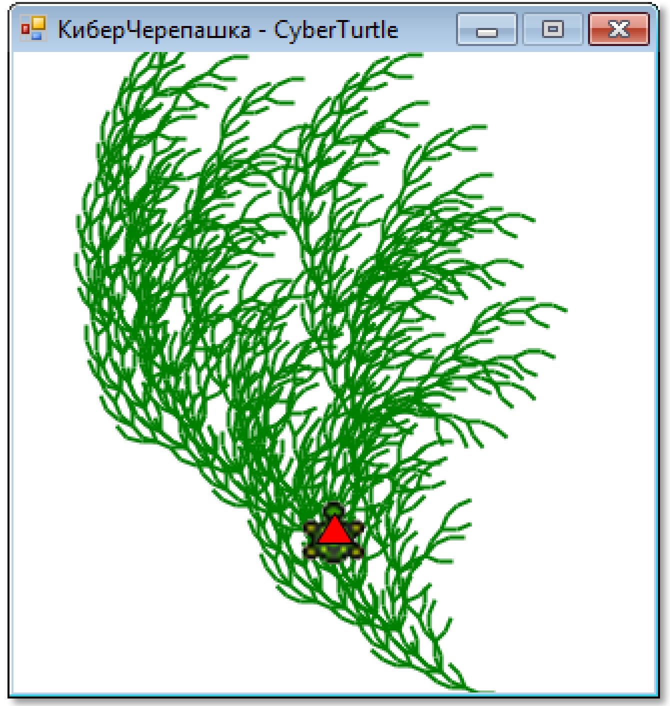


Рис. 23.23. Куст при *iter= 4*

L-системы

И вот, глядя на этот очаровательный «куст», мы можем, наконец, раскрыть тайну поведения нашей *Черепашки*. Оказывается, инстинкт *Черепашки* описывается с помощью *L-системы*, которая была предложена шведским биологом *Аристидом Линденмайе-*

ром (поэтому иначе они называются *системами Линденмайера*) для описания растений. Пример с кустом и следующий – с деревом (Рис. 23.24) - убедительно доказывают, что с помощью этой системы можно конструировать весьма правдоподобные растения (в некоторых графических редакторах они генерируются практически так же).

```
//Дерево:
procedure Tree;
begin
  size:=10;
  iter:= 4;
  teta:=180/6;
  x0:= CX;
  y0:= CY+150;
  axiom:='F';
  newF:='F[-F]F[+F][F]';
end;
```

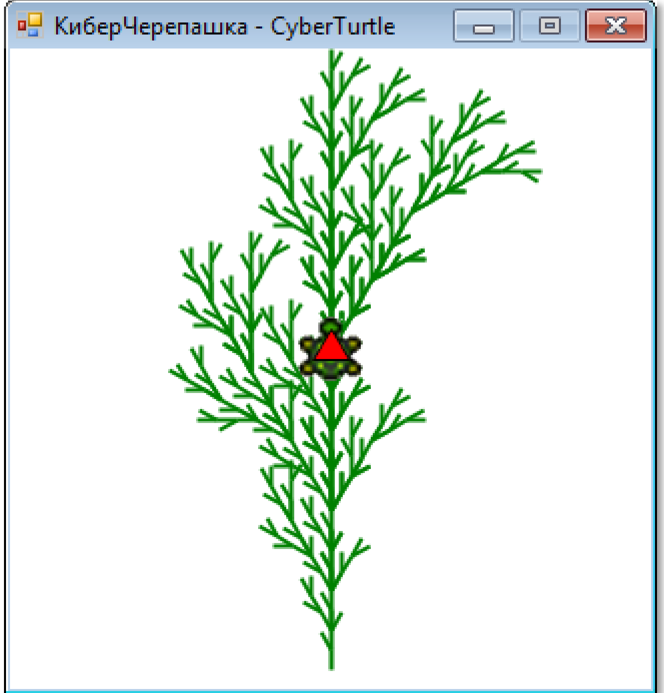


Рис. 23.24. Дерево

И ещё парочка деревьев (Рис. 23.25):

```
//Дерево2:
procedure Tree2;
begin
```

```
  size:=4;
  iter:= 4;
  teta:=180/6;
  x0:= CX;
  y0:= CY+160;
  axiom:='F';
  newF:='F[+F]F[-F]F';
end;

//Дерево3:
procedure Tree3;
begin
  a0:= 0;
  size:=6;
  iter:= 4;
  teta:=180/6;
  x0:= CX+20;
  y0:= CY+160;
  axiom:='F';
  newF:='FF-[-F+F+F]+[+F-F-F]';
end;
```

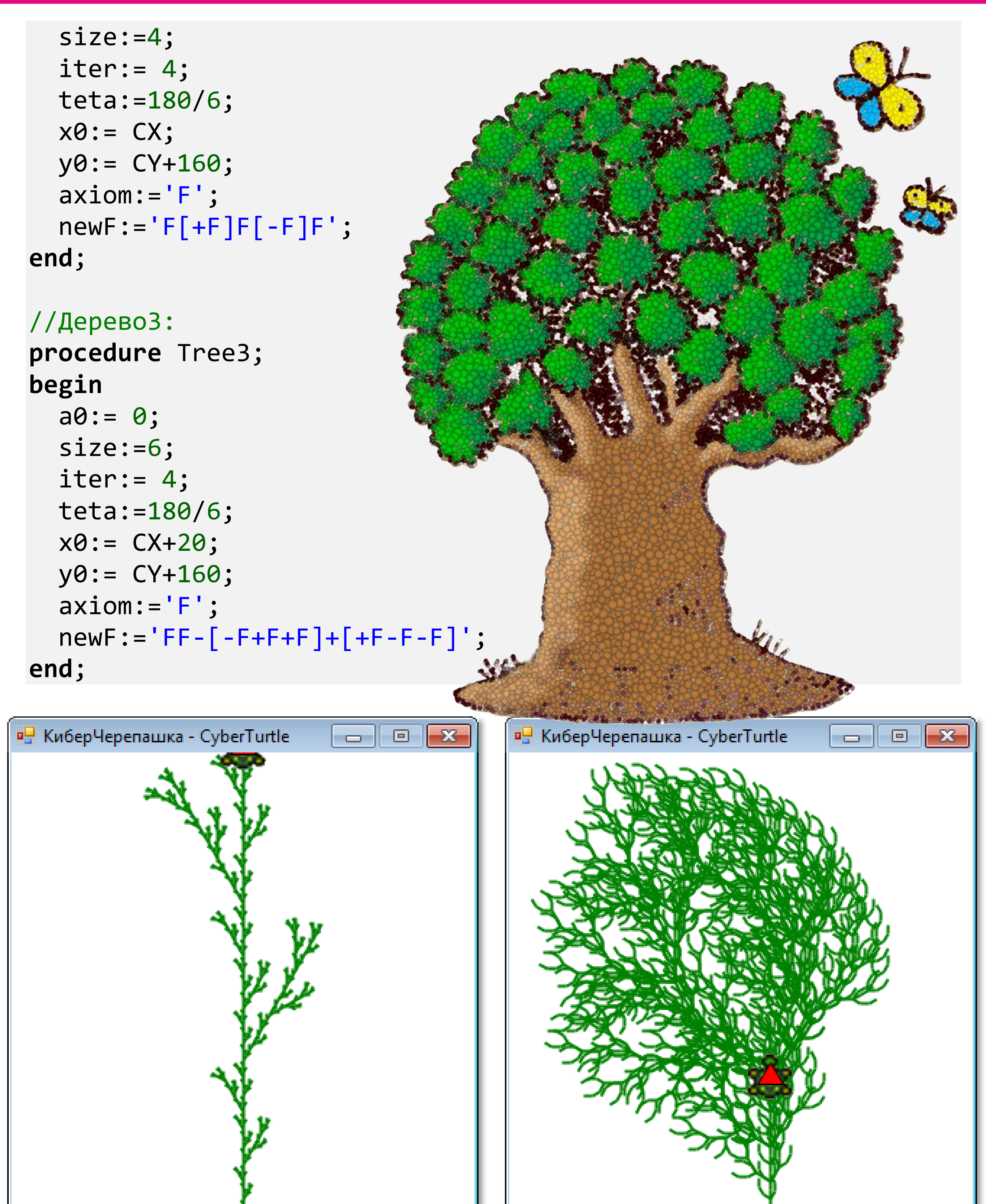


Рис. 23.25. Деревья

Все древовидные структуры, а также многие рассмотренные нами фракталы описываются аксиомой и несколькими правила-

ми, которые с увеличением числа итераций *iter* могут создать кривые любой степени детализации (в том числе и бесконечной, если у вас есть бесконечно много времени и других ресурсов). Неожиданно оказалось, что построение кривых по системе Линденмайера очень удобно реализовать с помощью черепашьей графики, поскольку *Черепашка* умеет выполнять все команды, используемые в *L-системах*.

Исходный код программы находится в папке **Cyber-Turtle.**

Урок 24. Тьюрмиты

Чебурашка - это неизвестный науке зверь,
который живёт в жарких тропических лесах.

Э. Успенский, *Крокодил Гена и его друзья*

Если вам понравились кренделя, которые выделывает *Киберчерепашка*, то вряд ли вы останетесь равнодушными и к творческим порывам другого кибернетического исполнителя наших желаний, данных ему в виде инстинктов.

Зовут его **тьюрмитом** и, как всеми любимый *Чебурашка*, это совершенно неизвестный науке зверь, поскольку в глаза его никто не видел. Поговаривают, что он похож на большого муравья, а его «научное» название *тьюрмит* лингвисты производят от двух слов: **Тьюр**инг и те**рмит**.

Если термиты известны всем любителям живой природы, то причастность Тьюринга к *тьюрмитам* нуждается в пояснении. Английский математик и кибернетик Алан Тьюринг (1912 — 1954) в 1936 году придумал абстрактную вычислительную машину, которая была названа в его честь *Машиной Тьюринга*. Она может выполнять команды, записанные на бесконечной ленте, с помощью управляющего устройства, которое последовательно принимает одно из возможных состояний. Как мы увидим дальше, тьюрмит представляет собой некий примитивный вариант машины Тьюринга.

Тьюрмит также похож и на *Киберчерепашку*, так как может выполнять команды, записанные в его инстинкте. *Команд* всего четыре:

1. Перекрасить текущую клетку в заданный цвет.
2. Повернуться направо или налево на 90 градусов.
3. Переползти в соседнюю клетку в текущем направлении.
4. Перейти в новое состояние.

Тьюрмит живёт в чистом поле, разделённом на клетки, которые первоначально окрашены в **чёрный** цвет. *Тьюрмит*, как и нейтрино, может существовать только в непрерывном движении,

которое заключается в переползании *тьюрмита* из одной клетки в другую, соседнюю с текущей по вертикали или горизонтали.

В каждый момент времени *тьюрмит* находится в каком-либо *состоянии*, обозначаемом буквами латинского алфавита. Например, при рождении *тьюрмит* находится в состоянии *A*. Также *тьюрмит* умеет определять цвет той клетки, на которой он стоит, и перекрашивать её в один из 16 цветов.

Мастерская тьюрмитов

Начнём новый проект и сохраним его в папке **Turmits**. Сразу же объявим *константы*, которые пригодятся нам в программе:

Для удобства мы введём массив *CELL_COLOR* и запишем в него все цвета, в которые могут быть окрашены клетки игрового поля:

```
const
   //число цветов:
   NUM_COLOR= 16;
   //цвета клеток:
   CELL_COLOR: array[0..NUM_COLOR-1] of  Color = (Color.Black,
                                         Color.Maroon,
                                         Color.Green,
                                         Color.Olive,
                                         Color.Navy,
                                         Color.Purple,

                                         Color.Teal,
                                         Color.Gray,
                                         Color.Silver,
                                         Color.Red,
                                         Color.Lime,

                                         Color.Yellow,
                                         Color.Blue,
                                         Color.Fuchsia,
                                         Color.Aqua,
                                         Color.White
                                         );
```

Тьюрмит может поворачиваться налево и направо, а также переползать вперёд - по направлению взгляда - в соседнюю слева, справа, сверху или снизу клетку (Рис. 24.1):

```
//направления движения тьюрмитов:
LEFT = -1;       //- поворот/движение налево
FORW = 0;        //- вперёд
RIGHT = 1;       //- поворот/движение направо
UP = FORW;       //- вверх
DOWN = 2;        //- вниз
```

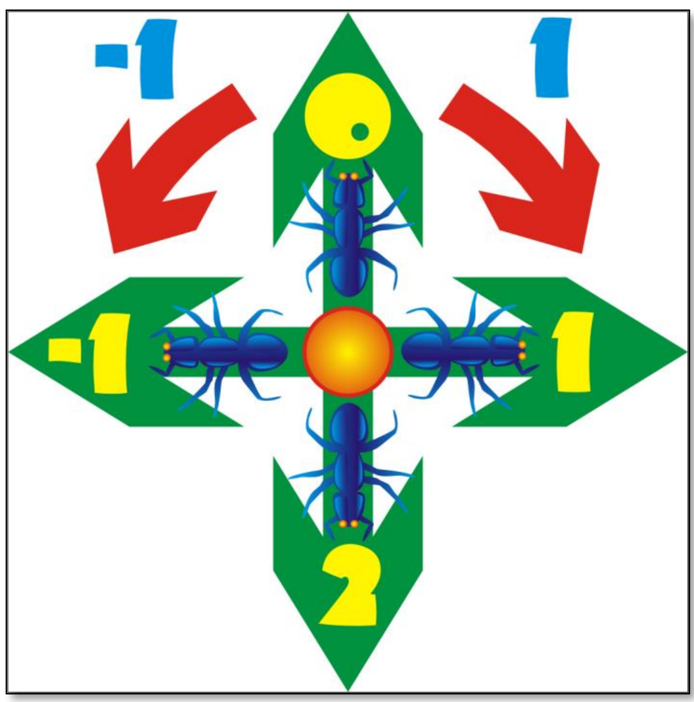


Рис. 24.1. Повороты и направления движения *тьюрмита*

Тьюрмит может переходить в одно из 26 *состояний*, которые обозначаются буквами латинского алфавита:

```
NUM_STATUS=26;  //- всего разных состояний тьюрмитов - A..Z
//число строк в сетке поля:
nRow = 65;
//число колонок в сетке поля:
nCol = 65;
//число клеток в сетке поля:
nCell = nRow*nCol;
```

Теперь мы займёмся *переменными*, которые описывают положение *тьюрмита* на поле и его состояние:

```
var
   //направление движения тьюрмита:
   Direction := FORW;
   //текущее состояние тьюрмита:
   CurStatus:= 'A';
   //x-координата клетки тьюрмита:
   xTurmit:=0;
   //y-координата клетки тьюрмита:
   yTurmit:=0;
   //число ходов тьюрмита:
   nMoves:= 0;
   //инстинкт тьюрмита:
   cmd: array[1..100] of string;
   //массив команд:
   commands: array[1..100] of TInstinct;
```

Подготовим *поле*, на котором будет жить и творить наш *тьюрмит*:

```
   //ширина клетки поля:
   wCell := 10;
   //высота клетки поля:
   hCell := 10;
   //отступ сетки поля от края окна по горизонтали:
   OffsetX:= 10;
   //отступ сетки поля от края окна по вертикали:
   OffsetY:= 60;
   //массив поля:
   masPole: array[1..nCell] of integer;
   WRAP: integer;
   nCommands: integer;

   //размеры окна:
   height, width: integer;
   //координаты центра окна:
   CX, CY: integer;

   //элементы управления:
   lblMove, lblDebug: RectangleABC;
   btnPlay: ButtonABC;
```

Полный инстинкт тьюрмита мы поместим в массив

```
commands: array[1..100] of TInstinct;
```

Каждую отдельную команду инстинкта удобно представить в виде записи *TInstinct* с пятью полями:

```
//ПРОГРАММА ДЛЯ МОДЕЛИРОВАНИЯ ТЬЮРМИТОВ

uses
   GraphABC, ABCButtons, ABCObjects;

type
   TInstinct= record
      //текущее состояние тьюрмита:
      curSost: char;
      //код цвета той клетки, в которой находится тьюрмит:
      curClr: integer;
      //код цвета той клетки, в которую тьюрмит должен перей-
ти:
      newClr: integer;
      //поворот тьюрмита:
      povorot: integer;
      //новое состояние тьюрмита:
      newSost: char;
   end;
```

Подготовку к рождению тьюрмита мы оформим в виде отдельной процедуры:

```
//
// ОСНОВНАЯ ПРОГРАММА
//
begin
   //Подготавливаем программу:
   prepareProg();
end.
```

Сначала мы выполняем все действия по созданию *окна* приложения.

Экзотичный *цвет фона* выбран только потому, что первоначально все клетки поля чёрные, но затем тьюрмит окрашивает их в разные цвета, в том числе и в чёрный, который будет не-

отличим от исходного чёрного цвета, что затруднит наши наблюдения за действиями *тьюрмита*.

```
//Подготавливаем программу
procedure prepareProg();
begin
   SetWindowTitle('Тьюрмиты');
   SetWindowWidth(OffsetX + (wCell*nCol)+ OffsetX);
   Window.Height:= OffsetY+(hCell*nRow) + 60;
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.BurlyWood);
   height := Window.Height;
   width := Window.Width;
   //координаты центра окна:
   CX:= width div 2;
   CY:= height div 2;
```

Для управления программой нам потребуются такие *элементы управления*:

```
   //КНОПКА
   //Начать:
   btnPlay := new ButtonABC(10, 10, 120, 32, 'Play Turmit!',
clMoneyGreen);
   btnPlay.FontColor:= Color.Red;
   btnPlay.FontStyle:= fsBold;
   btnPlay.TextScale:= 0.9;
   //процедура-обработчик:
   btnPlay.OnClick := btnPlay_OnClick;

   //МЕТКИ
   //Число ходов:
   lblMove:= new RectangleABC(200, 10, 120,26, Color.White);
   lblMove.FontStyle:= fsBold;
   //отладочное текстовое поле:
   lblDebug:= new RectangleABC(10, height-40,width-20,32, Col-
or.White);
   lblDebug.FontStyle:= fsBold;
```

Число клеток, которые посетил *тьюрмит*, мы будем показывать на экране:

```
   //обнуляем число ходов:
   nMoves:= 0;
   //показываем число ходов в текстовом поле:
   ShowNumMoves();
```

Жизненное пространство *тьюрмита* разбиваем на клетки:

```
   //рисуем сетку:
   drawGrid();

   //все клетки поля - чёрные:
   For var i:=1 To nCell do
       masPole[i] := 0; //Color.Black;
end;
```

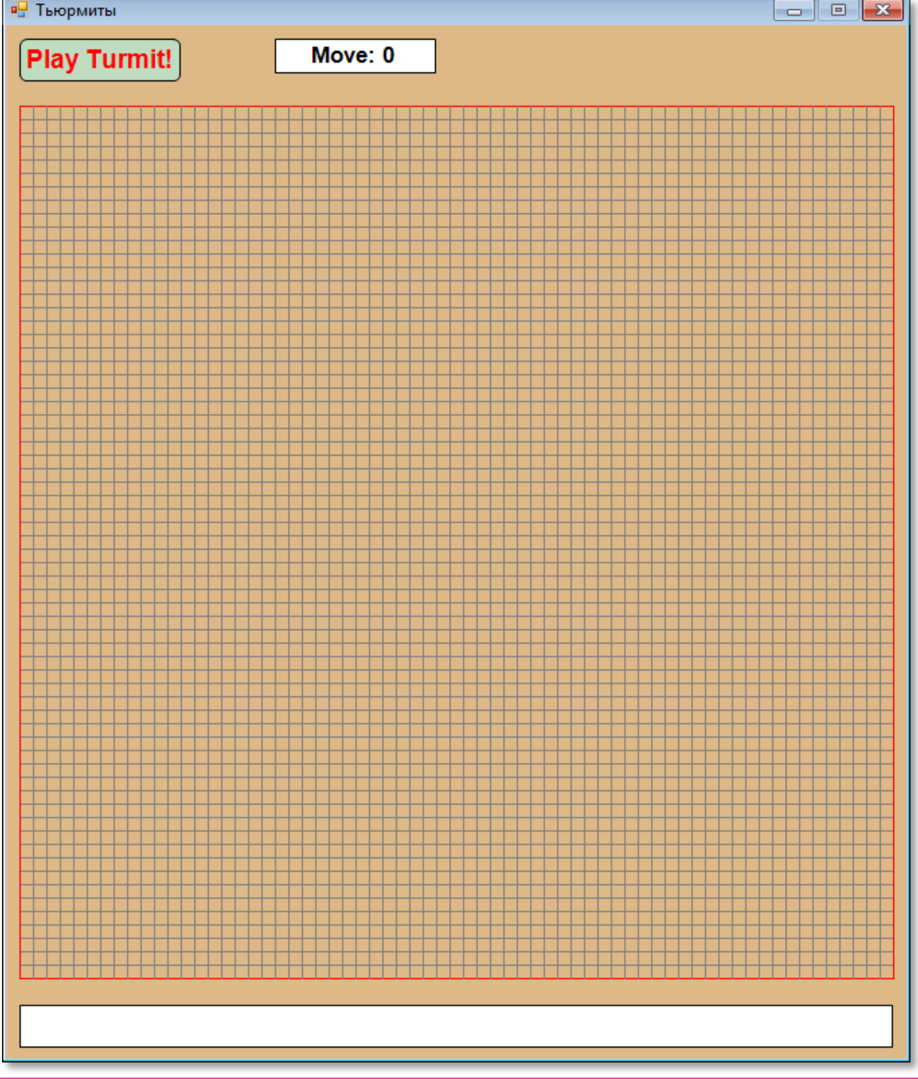
Процедура для вывода информации о числе ходов *тьюрмита*:

```
//ПЕЧАТАЕМ ЧИСЛО ХОДОВ ТЬЮРМИТА
procedure ShowNumMoves;
begin
   var s:= 'Move: ' + nMoves.ToString();
   lblMove.Text:= s;
End;
```

Следующая процедура расчерчивает поле на квадратики. Она очень простая и в комментариях не нуждается. Обратите только внимание на то, как выполняется обводка контура поля красными линиями:

```
//РИСУЕМ СЕТКУ
procedure drawGrid;
begin
  //устанавливаем размеры сетки:
  var Height := hCell*nRow;
  var Width := wCell*nCol;
  //и толщину линий:
  SetPenWidth(1);

  //чертим горизонтальные линии:
  var YPosition:= OffsetY;
  For var i:=0 to nRow do begin
     //устанавливаем цвет линий:
     If (i=0) or (i= nRow) Then
```

```
        SetPenColor(Color.Red)
     Else
        SetPenColor(Color.Gray);

     Line(OffsetX, YPosition, OffsetX+Width, YPosition);
     YPosition += hCell;
  end;//for

  //чертим вертикальные линии:
  var XPosition:= OffsetX;
  For var i:=0 to nCol do begin
     //устанавливаем цвет линий:
     If (i=0) or (i= nCol) Then
        SetPenColor(Color.Red)
     Else
        SetPenColor(Color.Gray);

     Line(XPosition, OffsetY, XPosition, OffsetY+Height);
     XPosition += wCell;
  end;//for
End;
```



Если мы сейчас запустим программу, то увидим мир до рождения *тьюрмита* (Рис. 24.2).

Рис. 24.2. Мир *тьюрмита* в начале времён

И вот в доселе пустом мире появляется *тьюрмит*!

```
//Рождение нового тьюрмита
procedure prepareTurmit;
begin
  //обнуляем число ходов:
  nMoves:= 0;
  ShowNumMoves();

  //начальные свойства тьюрмита -->

  //тьюрмит появляется в центре поля:
  xTurmit := Floor(nCol/2);
  yTurmit := Floor(nRow/2);
  //начальный статус тьюрмита - A:
  CurStatus:= 'A';

  //направление движения - вперёд:
  Direction := FORW;

  //задаем тьюрмиту инстинкт:
  Turmit1();
  readCommands();
End;
```

Весь акт творения очень прост, кроме двух последних строк, где *тьюрмит* получает инстинкт, определяющий всё его дальнейшее бренное существование.

В нашем распоряжении будет множество инстинктов, и каждый из них мы оформим в виде отдельной процедуры, чтобы затем нам было просто создать нужного нам *тьюрмита*:

```
////////////////////////////////////////////////////
//                                              //
//           ИНСТИНКТЫ ТЬЮРМИТОВ                //
//                                              //
///////////////////////////////////////////////////

procedure  Turmit1;
begin
  cmd[1]:='A 0 15 0 A';
```

```
  cmd[2]:='END';
End;
```

Как видите, инстинкт записывается в виде *строк* и хранится в массиве *cmd*. Последняя строка инстинкта всегда имеет значение *'END'* и отмечает конец списка.

Все остальные строки (Рис. 24.3) содержат *пять параметров*.

Первый параметр – буква, определяющая текущее состояние *тьюрмит*а. Этот параметр обозначается прописными буквами латинского алфавита.

Второй параметр – число от 0 до 15 – код цвета той клетки, в которой находится тьюрмит. Конкретный цвет, соответствующий этому коду, определяется значением из массива *CELL_COLOR*.

Третий параметр – число от 0 до 15 – код цвета той клетки, в которую *тьюрмит* должен перейти.

Четвёртый параметр – поворот тьюрмита. Обозначается числами:

-1 – поворот налево на 90 градусов
 0 – не поворачиваться
 1 – поворот направо на 90 градусов

Пятый параметр – буква, определяющая *новое* состояние *тьюрмита*. Также обозначается прописными буквами латинского алфавита. После выполнения строки *тьюрмит* переходит из текущего состояния в новое, которое становится текущим.

Рис. 24.3. Строка инстинкта *тьюрмита*

Сразу после рождения тьюрмит находится в состоянии *A* и занимает клетку ***чёрного*** цвета (Рис. 24.4). Таким образом, в его инстинкте первой обязательно должна быть строка, начинающаяся так: *A 0*.

Для примера рассмотрим самый простой инстинкт:

```
procedure Turmit1;
begin
  cmd[1]:='A 0 15 0 A';
  cmd[2]:='END';
End;
```

Последняя буква – *A* – совпадает с первой буквой, значит, при переходе в новую клетку состояние *тьюрмита* не изменяется. А поскольку весь инстинкт описывается единственной строкой, наш *тьюрмит всегда* будет находиться в одном и том же состоянии.

Тьюрмиты постоянно двигаются, поэтому и наш *тьюрмит* должен перейти в соседнюю клетку. Так как направление движения равно *0*, то он просто переползёт в ту соседнюю клетку, в сторону которой смотрит. Первоначально *тьюрмит* смотрит *вверх* (или, если хотите, на север), значит, и перейдёт он в соседнюю *верхнюю* клетку. Однако прежде он должен перекрасить текущую клетку в заданный цвет (Рис. 24.5). В инстинкте он обозначен числом *15*, что соответствует белому цвету.

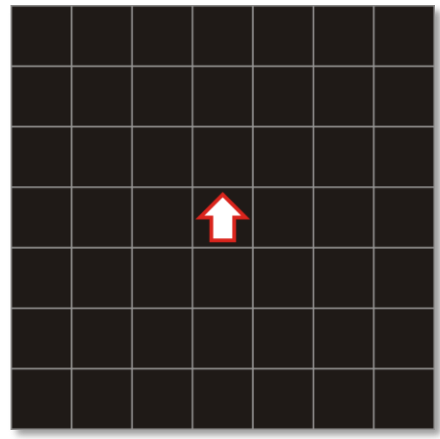


Рис. 24.4. Тьюрмит сразу после рождения

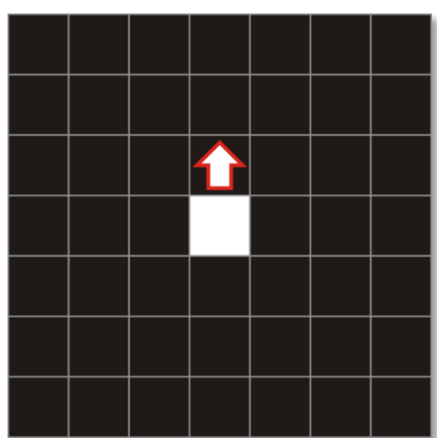


Рис. 24.5. Тьюрмит сделал первый шаг

Тьюрмит опять находится в состоянии *A* и стоит на **чёрной** клетке. Эту ситуации вновь описывает инстинкт в первой строке, то есть второй шаг *тьюрмита* будет в точности таким же, как и первый. Более того, мы можем утверждать, что тьюрмит будет бесконечно перемещаться вверх, окрашивая клетки в *белый* цвет (Рис. 24.6).

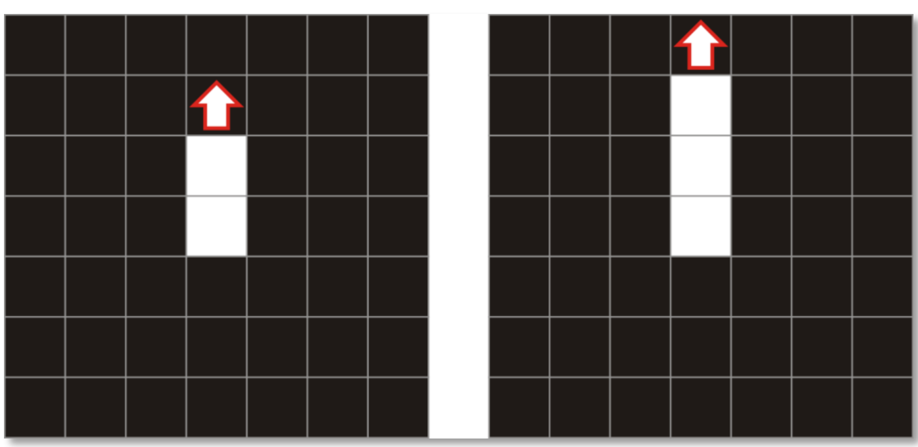


Рис. 24.6. Жизненный путь *тьюрмита*

В отличие от мира, в котором живёт *тьюрмит*, мы не можем создать ни в памяти компьютера, ни на экране монитора *бесконечный* мир, поэтому мы должны решить, чт***о*** нам делать, если *тьюрмит* покинет видимую нам часть мира.

Мы можем либо просто прекратить дальнейшие наблюдения за *тьюрмитом*, то есть прервать выполнение программы, либо вернуть его в наш мир, но уже с *обратной* стороны. В этом случае мир *тьюрмита* будет похож на поверхность *тора* (бублика) (Рис. 24.7).

Мы реализуем в программе оба сценария, но вы должны иметь в виду, что поведение *тьюрмита* в замкнутом мире будет деформировано, поскольку он может попасть на клетки, которые будут неверно истолкованы его инстинктом.

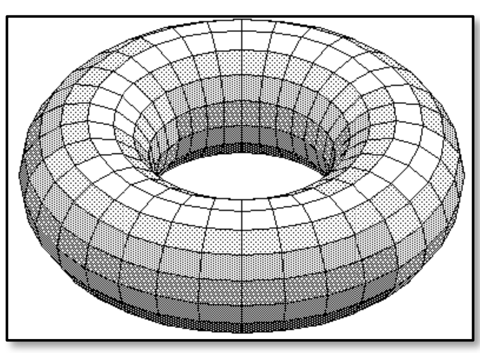
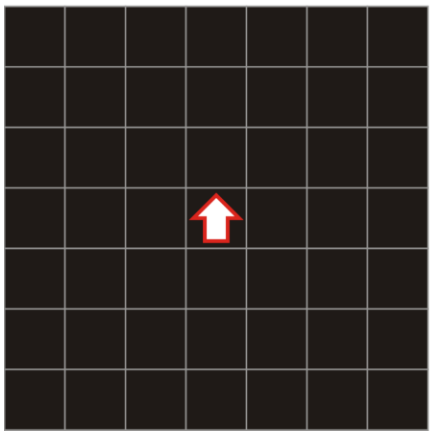


Рис. 24.7. Замкнутый «торный» мир

Ну а мы рассмотрим более сложный инстинкт, на описание которого потребовалось три строки:

```
//Зелёный квадрат 4 x 4:
procedure GreenSquare44;
begin
  cmd[1]:='A 0 2 0 B';
  cmd[2]:='B 0 2 0 C';
  cmd[3]:='C 0 2 1 A';
  cmd[4]:='END';
end;
```

Путешествие по жизни новый *тьюрмит* начнёт с первой строки. Он перекрасит текущую клетку в **зелёный** цвет (код *2*), поднимется на клетку выше (направление равно нулю) и перейдёт в состояние *B*. (Рис. 24.8).



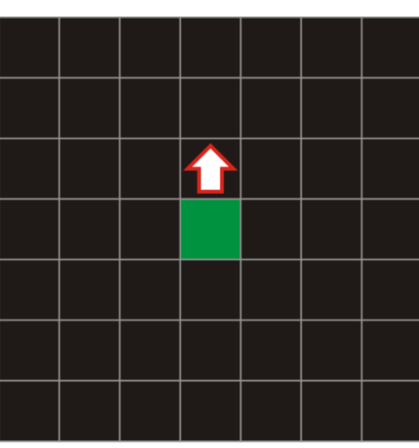


Рис. 24.8. Положение тьюрмита после первого хода

Теперь *тьюрмит* должен отыскать в своём инстинкте строку, начинающуюся с *B 0*, так как он находится в состоянии *B* и занимает клетку **чёрного** цвета. Это вторая строка:

```
cmd[2]:='B 0 2 0 C';
```

Повинуясь приказам инстинкта, *тьюрмит* и эту клетку покрасит в **зелёный** цвет и перейдёт в состояние *C* (Рис. 24.9).

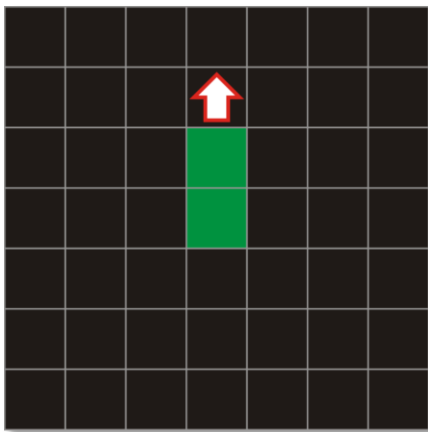


Рис. 24.9. Положение *тьюрмита* после второго хода

Далее *тьюрмит* находит строку

```
cmd[3]:='C 0 2 1 A';,
```

соответствующую его новому состоянию. Он вновь окрашивает клетку в **зелёный** цвет, но теперь переходит в клетку справа (код поворота равен единице). *Тьюрмит* возвращается в состояние *A* (Рис. 24.10).

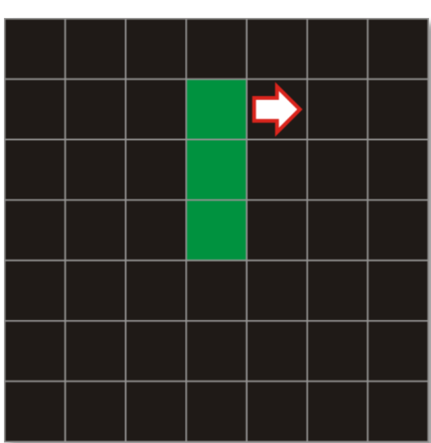


Рис. 24.10. Положение *тьюрмита* после третьего хода

Он опять должен выполнить команды, записанные в *первой* строке:

```
cmd[1]:='A 0 2 0 B';
```

Но на этот раз *тьюрмит* смотрит *вправо*, поэтому перейдёт не в верхнюю клетку, а в *правую* (Рис. 24.11).

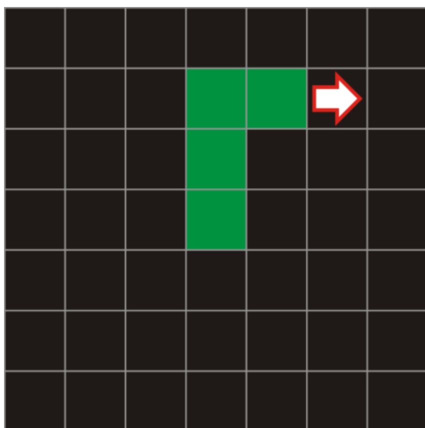


Рис. 24.11. Положение *тьюрмита* после четвёртого хода

Тьюрмит вторично выполняет *вторую* строку

```
cmd[2]:='B 0 2 0 C';,
```

при этом двигаясь вправо (Рис. 24.12).

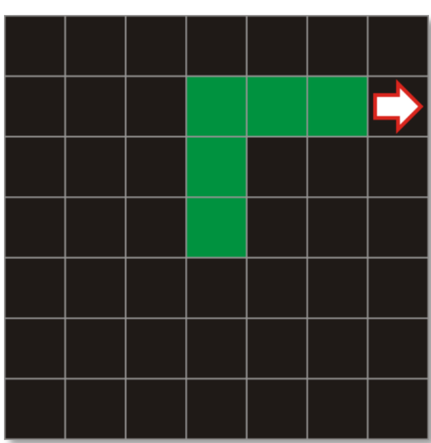


Рис. 24.12. Положение *тьюрмита* после пятого хода

Выполняя следующие команды

```
cmd[3]:='C 0 2 1 A';,
```

тьюрмит повернётся направо и перейдёт в нижнюю клетку (Рис. 24.13).

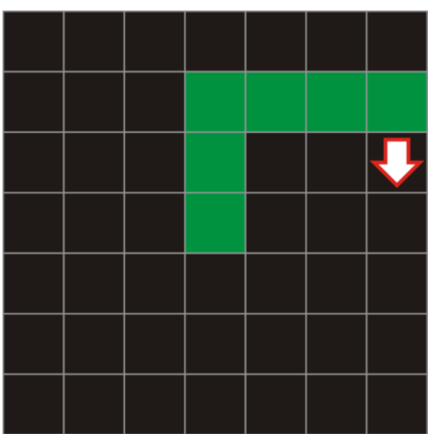


Рис. 24.13. Положение *тьюрмита* после шестого хода

Далее вы легко проследите, что тьюрмит выполнит строки

```
cmd[1]:='A 0 2 0 B';
cmd[2]:='B 0 2 0 C';
cmd[3]:='C 0 2 1 A';
cmd[1]:='A 0 2 0 B';
```

(Рис. 24.14, слева)
(Рис. 24.14, справа)
(Рис. 24.15, слева)
(Рис. 24.15, справа)

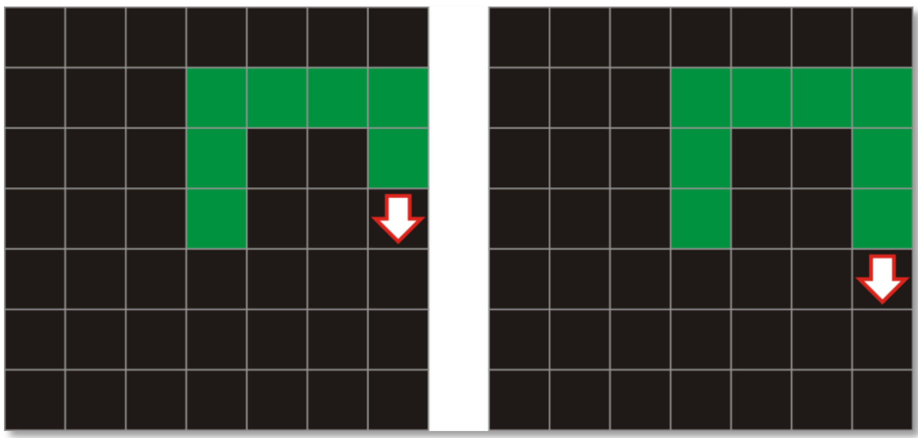


Рис. 24.14. Положение *тьюрмита* после седьмого и восьмого ходов

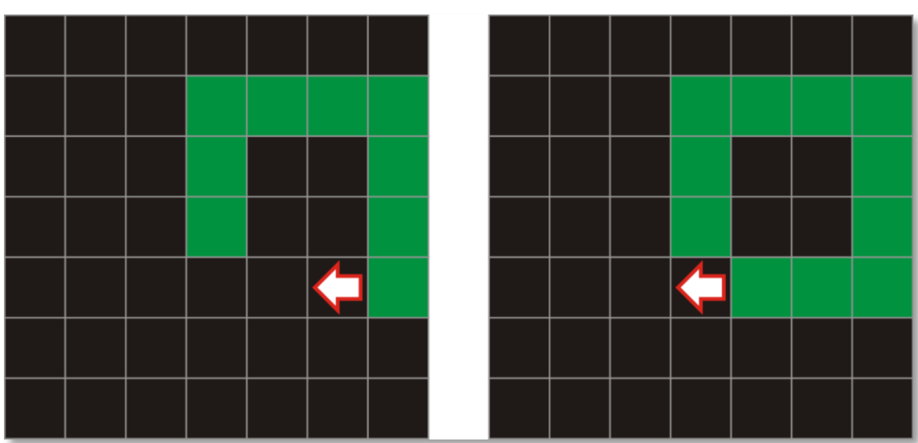


Рис. 24.15. Положение *тьюрмита* после девятого и десятого ходов

`cmd[2]:='В 0 2 0 С';` (Рис. 24.16)

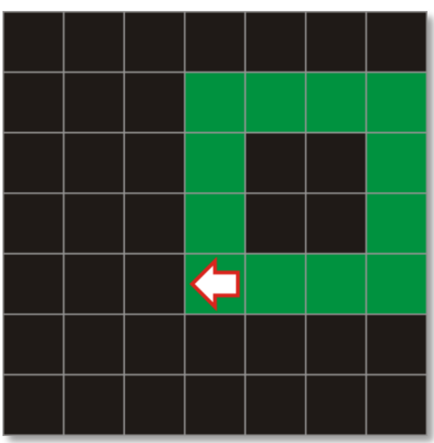


Рис. 24.16. Положение *тьюрмита* после одиннадцатого хода

В этом положении на поле *тьюрмит* находится на зелёной клетке в состоянии *С*, но в его инстинкте нет строки, начинающей с *С 2*, поэтому *тьюрмит* погибает, а программа заканчивается.

Хотя *тьюрмитом* руководят команды, записанные в одном инстинкте, нам удобнее перевести их в *массив* команд, а не просматривать каждый раз все строки инстинкта.

Оформим преобразование строк в виде отдельной процедуры:

```
//Считываем строки инстинкта в массив команд
procedure readCommands;
begin
  var i:=1;
```

```
  //обрабатываем строки, пока не достигнем конца списка:
  While (cmd[i] <> 'END') do begin
     //очередная строка инстинкта:
     var s:= cmd[i];
    //текущее состояние тьюрмита:
    commands[i].curSost:= s[1];
    //его текущий цвет:
    var n:= 3;
    var clr:=0;
    While (s[n] <> ' ') do begin
       clr:= clr*10 + integer.Parse(s[n]);
       n:= n+1;
    End;//While
    commands[i].curClr:= clr;
    //новый цвет тьюрмита:
    n:= n+1;
    clr:=0;
    While (s[n] <> ' ') do begin
       clr:= clr*10 + integer.Parse(s[n]);
       n:= n+1;
    End;//While
    commands[i].newClr:= clr;

    //направление движения тьюрмита:
    n:= n+1;
    var s2:string:= s[n];
    //отрицательное значение:
    If (s2 = '-') Then begin
       n:= n+1;
       s2:= '-'+s[n];
    End;//If
    commands[i].povorot:=  integer.Parse(s2);
    //новое состояние тьюрмита:
    commands[i].newSost:= s[s.Length];
    i := i + 1;
  End;//While
  //число строк с командами:
  nCommands:= i-1;
End;
```

Мы последовательно просматриваем строки, начиная с первой и заканчивая последней, в которой записано слово *END*.

Формат строки нам известен: текущее состояние (буква) - текущий цвет (число) – новый цвет (число) – поворот (число) – новое состояние (буква). Для облегчения обработки строк мы будем считать, что все параметры разделяет в точности *один* пробел. В этом случае небольшие сложности возникнут только при обработке кода цвета, который может быть одно- и двузначным числом, и направления движения, которое может быть положительным и отрицательным.

Игра с *тьюрмитом* (а для него это жизнь!) начинается после нажатия на кнопку *Play Thurmit* и заключена в одной-единственной процедуре:

```
//НАЧИНАЕМ ИГРУ
procedure btnPlay_OnClick;
begin
  //производим на свет нового тьюрмита:
  prepareTurmit();
```

После рождения *тьюрмит* начинает выполнять команды первой строки, которая должна содержать его текущее состояние и цвет (то есть начинаться с *A 0*):

```
  //текущая строка в списке команд:
  var curString:=1;
  var s:='';
  While (True) do begin
    //новый статус тьюрмита:
    var CurStatus:= commands[curString].newSost;
    s:= s + ' CurStatus= ' + CurStatus;
    //новый цвет клетки:
    var NewColor:= commands[curString].newClr;
    s:= s+ ' NewColor= ' + NewColor.ToString();

    //тьюрмит перекрашивает клетку в новый цвет:
    masPole[nCol*yTurmit+xTurmit]:=NewColor;
    //рисуем тьюрмита цветом клетки:
    DrawTurmit();

    //направление движения тьюрмита:
    var newDirection:= commands[curString].povorot;
```

```
    s:= s+ ' newDirection= ' + newDirection.ToString();
    lblDebug.Text:= s;
    Direction:= Direction+newDirection;
    //корректируем направление:
    If  Direction < LEFT Then
        Direction:= DOWN;
    If  Direction > DOWN Then
        Direction := LEFT;

    //клетка, в которую должен перейти тьюрмит:
    If Direction = LEFT Then begin
       xTurmit -= 1;
       yTurmit += 0;
    End;

    If Direction = UP Then begin
       xTurmit += 0;
       yTurmit -= 1;
    End;

    If Direction = RIGHT Then begin
       xTurmit += 1;
       yTurmit += 0;
    End;

    If Direction = DOWN Then begin
       xTurmit += 0;
       yTurmit += 1;
    End;

    //тьюрмит  появляется с обратной стороны поля:
    //WRAP:=1;
    //программа заканчивается, если тьюрмит выходит за границу
поля:
    WRAP:=0;
    if WRAP=0 Then begin
       s:='';
      If xTurmit = nCol Then
        s:='  Too big X!  '
      else If yTurmit = nRow Then
        s:='  Too big Y!  '
      else If xTurmit < 0 Then
        s:='  Too small X!  '
      else If yTurmit < 0 Then
```

```
        s:='  Too small Y! '
    End
    //тьюрмит  появляется с обратной стороны поля:
    else if WRAP=1 Then begin
      s:='';
      If xTurmit = nCol Then
        xTurmit := 0
      else If xTurmit < 0 Then
        xTurmit := nCol-1
      else If yTurmit = nRow Then
        yTurmit := 0
      else If yTurmit < 0 Then
        yTurmit := nRow-1;
    End;//If

    if s <> '' then begin
       lblDebug.Text:= s;
       exit;
    end;
```

Затем *тьюрмит* ищет в списке строку, соответствующую его новому состоянию и цвету текущей клетки. Если такой строки он не обнаружит, значит, он, как и мавр, сделал своё дело и должен уйти на покой:

```
    //тьюрмит ищет следующую строку:
    curString:=0;
    For var i:=1 to  nCommands do begin
      if (commands[i].curSost= CurStatus) and
         (masPole[nCol*yTurmit+xTurmit]= commands[i].curClr)
then begin
         curString:=i;
         break;
      end;
    End;//For
    if  curString=0  then begin
        //нужной строки нет --> тьюрмит погибает:
        s:= '    END of TURMIT    ';
        lblDebug.Text:= s;
        exit;
    End;//If
```

В противном случае жизнь *тьюрмита* продолжается в бесконечном цикле *While*:

```
    //тьюрмит сделал очередной ход:
    nMoves += 1;
    ShowNumMoves();
  End;//While
End;
```

И в последней процедуре проекта нам осталось только нарисовать самого героя сей жизненной драмы – *тьюрмита*:

```
//Рисуем тьюрмита в текущей позиции
procedure DrawTurmit;
begin
   var clr:= CELL_COLOR[masPole[nCol*yTurmit+xTurmit]];
   SetBrushColor(clr);
   var x:=OffsetX + xTurmit*wCell;
   var y:=OffsetY + yTurmit*hCell;
   FillEllipse(x,y,x+wCell,y+hCell);
   //Sleep(50);
end;//DrawTurmit
```

Его внешний облик представлен в программе невыразительным кружком того же цвета, что и занимаемая им клетка. Таким образом, наш герой, как и персонаж *Рассказа о неизвестном герое* Самуила Маршака, совершенно незаметен в жизни, хотя и оставляет после себя яркие воспоминания в форме прелестно раскрашенного поля. Так воздайте же *тьюрмиту* должное и изобразите его в полной красе, как и подобает всякому труженику полей.

Рассказ о *тьюрмитах* оказался довольно продолжительным, но *тьюрмит* не должен разочаровать вас в столь долгих ожиданиях прекрасного. Сейчас вы убедитесь, что не зря изучали анатомию и психологию тьюрмитов. С помощью простых инстинктов мы научим *тьюрмита* делать столь экстравагантные па, что дух захватывает!

Однако начнём мы с простых экзерсисов, чтобы вы смогли проверить, что наши бумажные манипуляции с генетическим кодом

зверушки полностью соответствуют его жизненным устремлениям. *Первый* инстинкт выполняется в полной мере и безукоризненно. *Тьюрмит* действительно уходит за верхнюю границу доступного нам мира (Рис. 24.17).

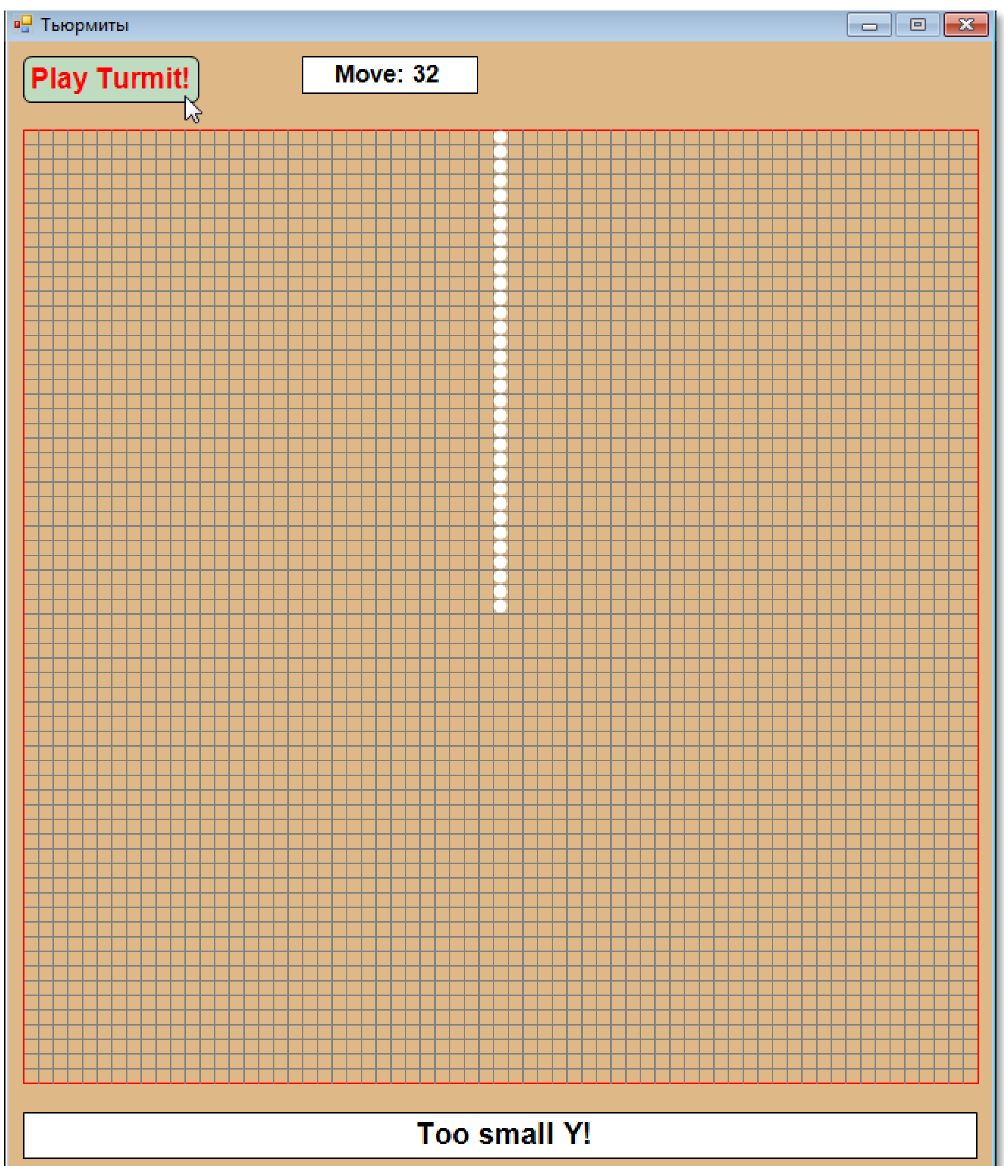


Рис. 24.17. *Тьюрмит* стремительно уходит вверх

Второй очень простой инстинкт заставляет *тьюрмита* нарисовать белый квадратик (Рис. 24.18):

```
procedure  Turmit2;
begin
  cmd[1]:='A 0 15 1 A';
  cmd[2]:='END';
End;
```

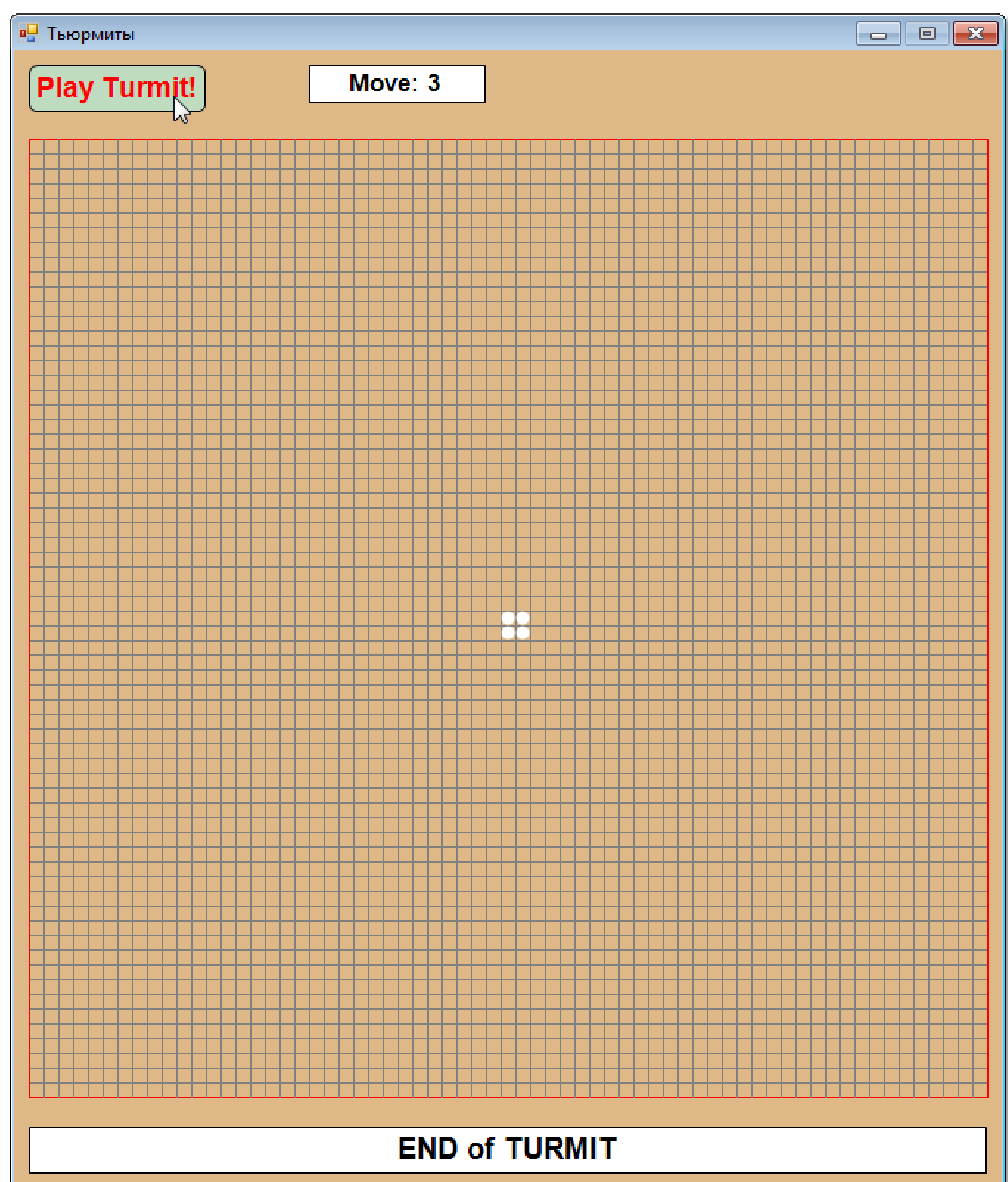


Рис. 24.18. *Тьюрмит* «оквадратился»

Далее *тьюрмит* интеллектуально прогрессирует и осиливает большие квадраты - 3 x 3 клетки (Рис. 24.19):

```
//Зелёный квадрат 3 x 3:
procedure GreenSquare33;
begin
  cmd[1]:='A 0 2 0 B';
  cmd[2]:='B 0 2 1 A';
  cmd[3]:='END';
end;
```

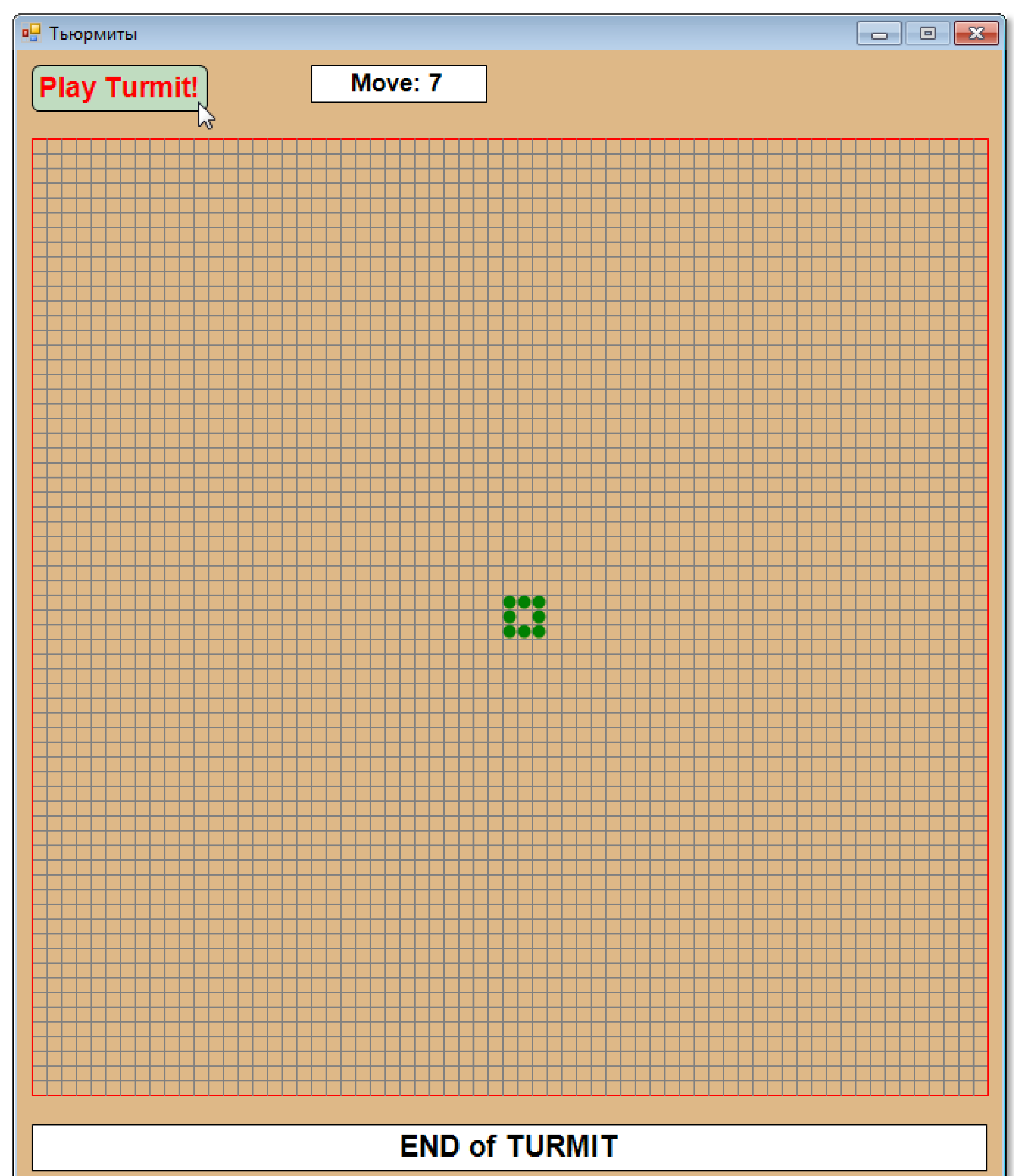


Рис. 19. Квадрат *тьюрмита* порядка 3

И 4 x 4 (этот пример мы разбирали вручную) (Рис. 24.20):

```
//Зелёный квадрат 4 x 4:
procedure GreenSquare44;
begin
  cmd[1]:='A 0 2 0 B';
  cmd[2]:='B 0 2 0 C';
  cmd[3]:='C 0 2 1 A';
  cmd[4]:='END';
end;
```

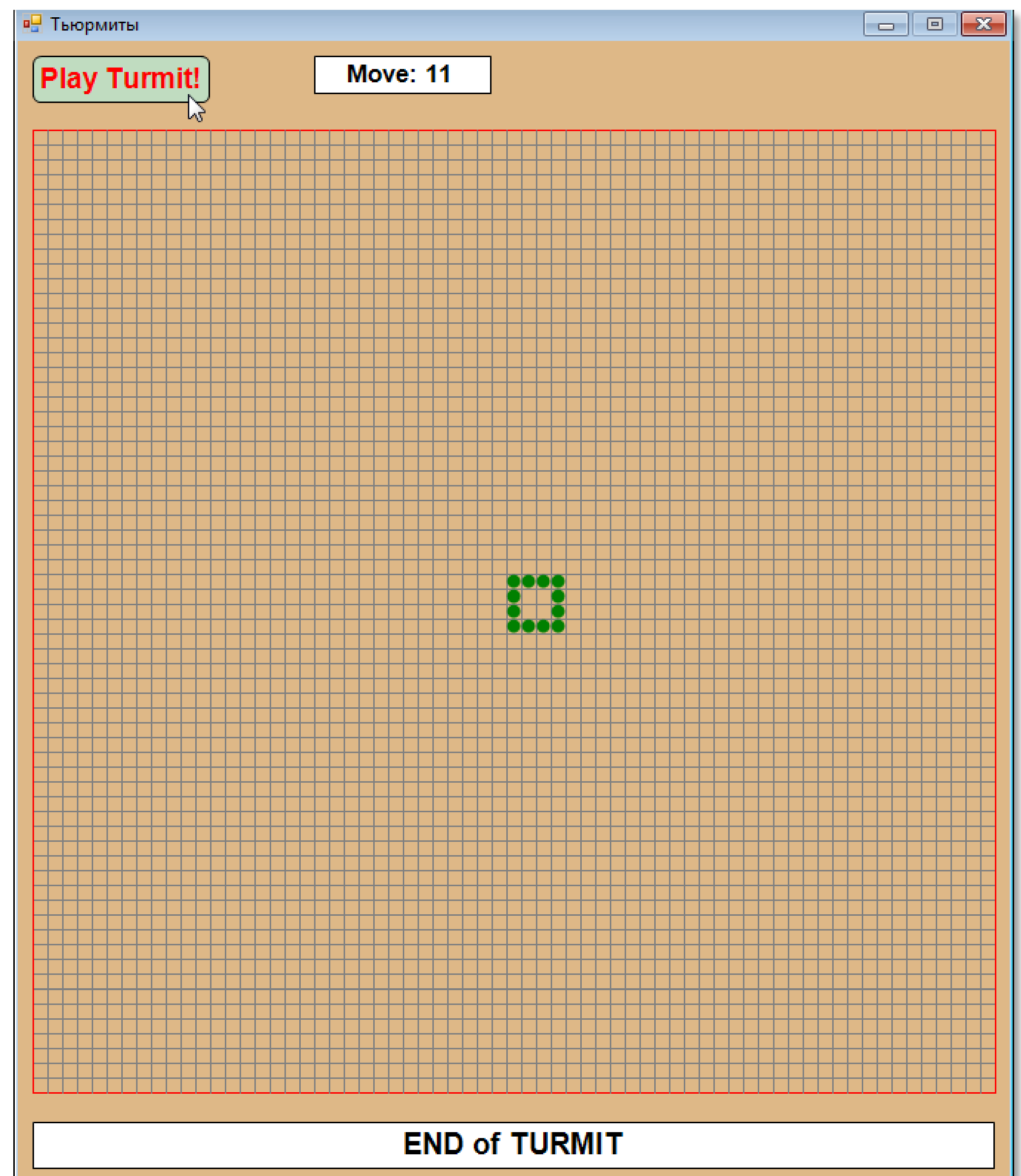


Рис. 24.20. Квадрат *тьюрмита* порядка 4

Тьюрмит может нарисовать и более сложную фигуру, например, *крест* (Рис. 24.21):

```
//Cross!
procedure Turmit5;
begin
  cmd[1]:='A 0 2 0 B';
  cmd[2]:='B 0 2 0 C';
  cmd[3]:='C 0 2 1 A';
  cmd[4]:='A 2 1 -1 A';
  cmd[5]:='END';
end;
```

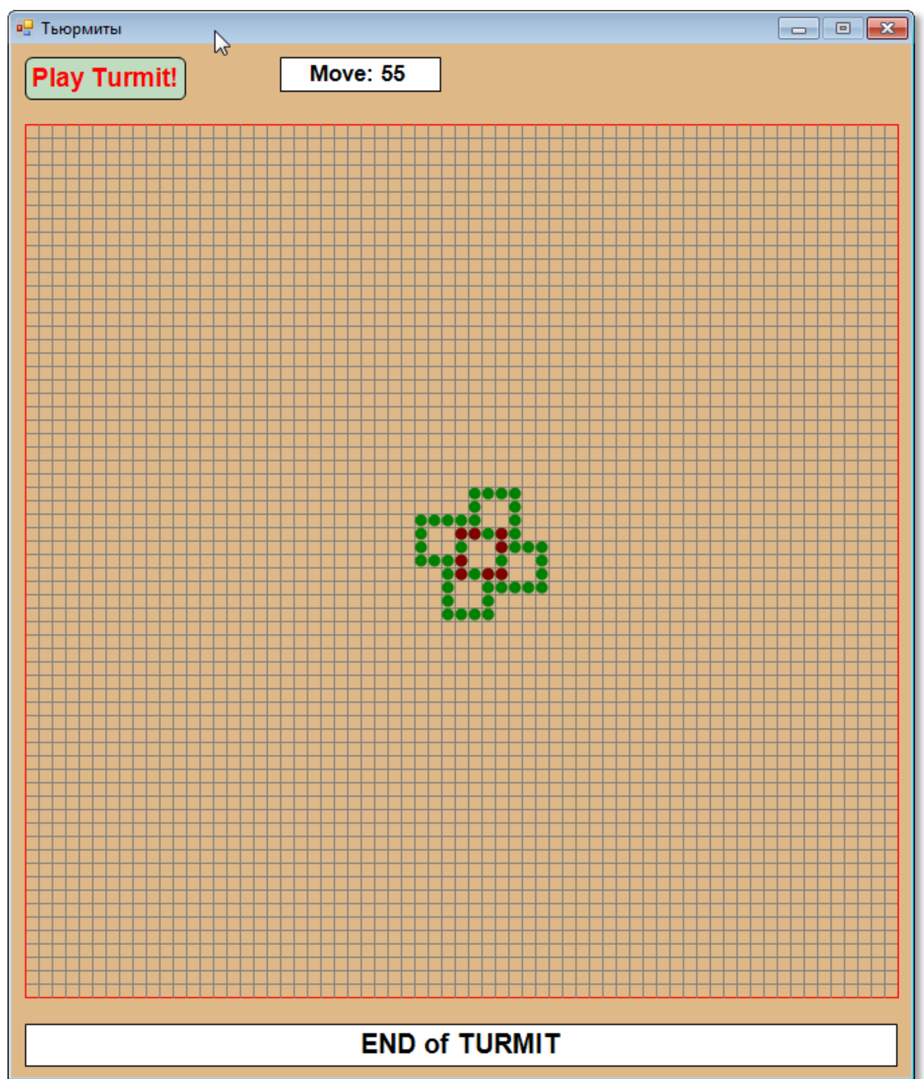


Рис. 24.21. Крест *тьюрмита*

Впрочем, статические рисунки, даже самые прекрасные, не могут сравниться с теми визуальными превращениями, которые показывает живой, полный энергии *тьюрмит*, стремительно носящийся по экрану. Наблюдать за ним – настоящее удовольствие! Поэтому достоинства двух следующих инстинктов (Рис. 24.22 и 24.23) трудно оценить по снимкам с экрана, их следует смотреть вживую.

```
//Спираль Тарка
procedure  Turmit4;
begin
  cmd[1]:='A 0 2 -1 A';
  cmd[2]:='A 2 0 0 B';
  cmd[3]:='B 0 2 1 A';
```

```
    cmd[4]:='B 2 2 1 A';
    cmd[5]:='END';
end;
```

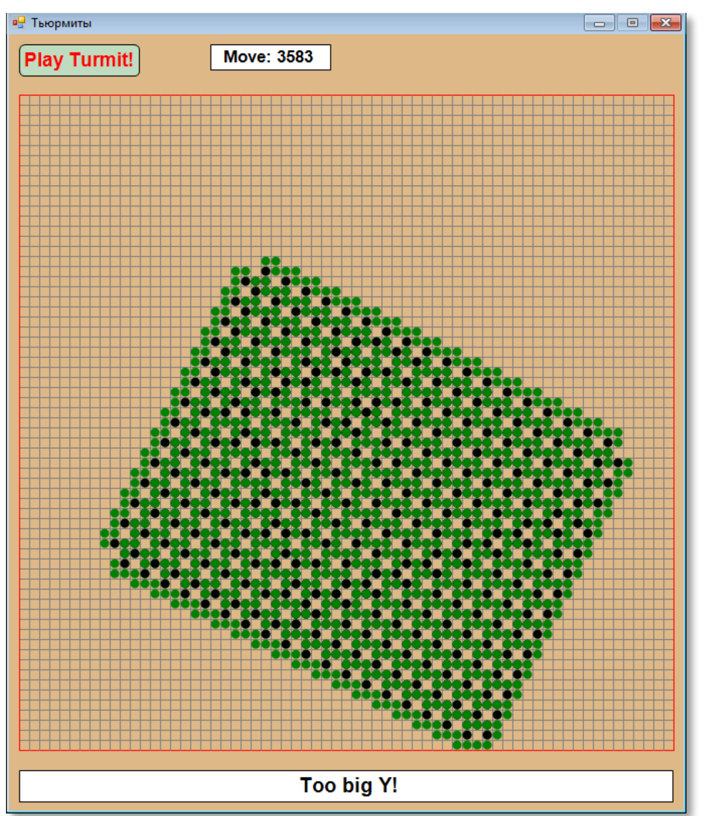


Рис. 24.22. Спираль Тарка

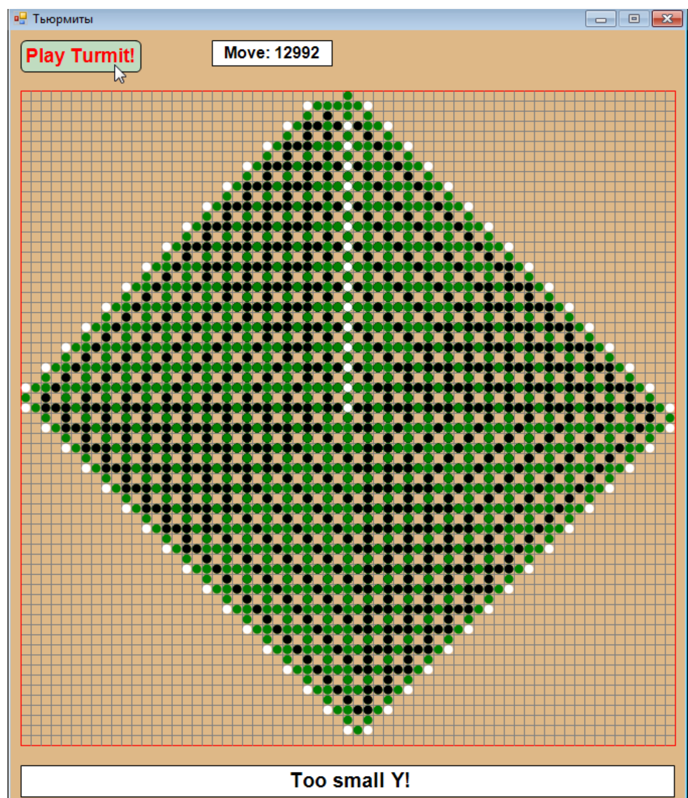


Рис. 24.23. Квадрат Кнопа

```
//Квадрат Кнопа
procedure KnopSquare;
begin
    cmd[1]:='A 0 2 0 C';
    cmd[2]:='A 2 0 0 B';
    cmd[3]:='B 2 2 1 A';
    cmd[4]:='B 15 2 1 A';
    cmd[5]:='C 2 0 -1 A';
    cmd[6]:='C 0 15 -1 A';
    cmd[7]:='C 15 2 -1 A';
    cmd[8]:='END';
End;
```

Следующий инстинкт – это более хитроумная вариация самого простого, первого инстинкта: он рисует не бесконечную белую линию, а бесконечную *шахматную доску* (Рис. 23.24):

```
//Шахматная доска
procedure ChessBoard2;
begin
  cmd[1]:='A 0 15 -1 B';
  cmd[2]:='B 0 9 0 C';
  cmd[3]:='C 0 15 0 D';
  cmd[4]:='D 0 9 0 E';
  cmd[5]:='E 0 15 0 F';
  cmd[6]:='F 0 9 0 G';
  cmd[7]:='G 0 15 0 H';
  cmd[8]:='H 0 9 1 I';
  cmd[9]:='I 0 15 1 J';
  cmd[10]:='J 0 9 0 C';
  cmd[11]:='I 15 15 1 K';
  cmd[12]:='K 9 9 1 L';
  cmd[13]:='L 15 15 1 L';
  cmd[14]:='L 9 9 -1 A';
  cmd[15]:='A 15 15 1 A';
  cmd[16]:='END';
End;
```

И, конечно, какая ж «песня» без *спирали* (Рис. 23.25)!

```
//GridSpiral
procedure GridSpiral;
begin
  cmd[1]:='A 0 14 0 J';
  cmd[2]:='J 0 13 -1 A';
  cmd[3]:='A 14 13 0 A';
  cmd[4]:='A 13 14 0 A';
  cmd[5]:='J 14 13 0 A';
  cmd[6]:='J 13 14 1 A';
  cmd[7]:='END';
End;
```

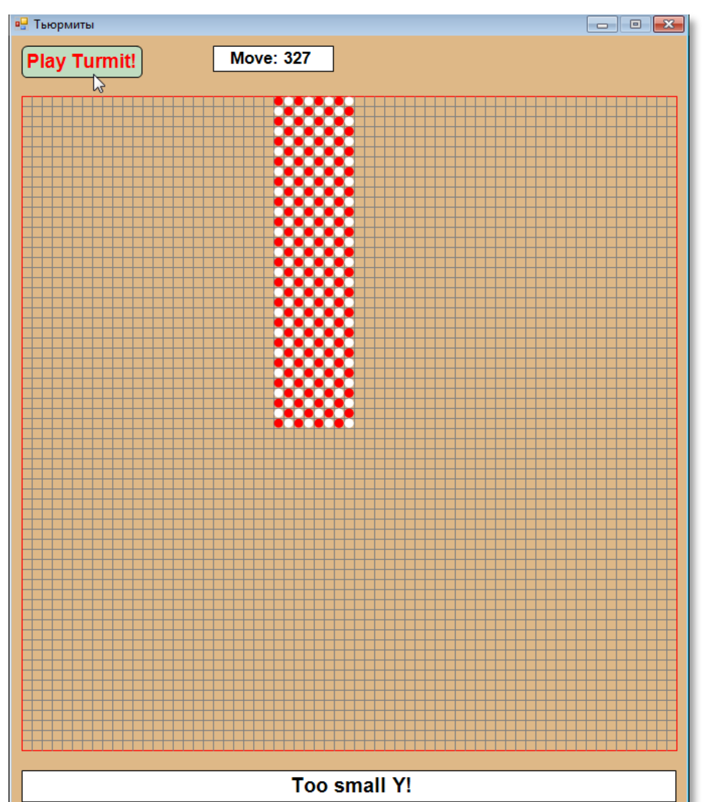


Рис. 24.24. Шахматная доска

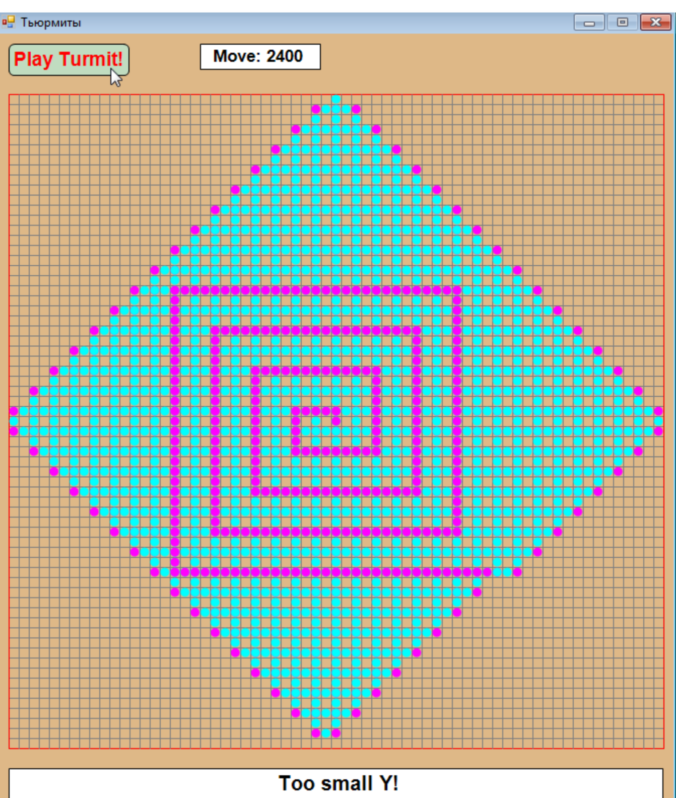


Рис. 24.25. Сетка-спираль

И последняя пара примеров весьма причудливых инстинктов - *клубок* (Рис. 24.26) и еще одна *спираль* - *ромбическая* (Рис. 24.27).

Ещё несколько инстинктов вы найдёте в исходном коде проекта.

```
//Клубок
procedure Klubok;
begin
  cmd[1]:='A 0 15 1 B';
  cmd[2]:='C 0 10 -1 A';
  cmd[3]:='B 15 15 1 C';
  cmd[4]:='A 15 15 1 B';
  cmd[5]:='B 10 10 1 C';
  cmd[6]:='B 0 15 1 A';
  cmd[7]:='C 15 15 0 B';
  cmd[8]:='END';
End;
```

```
//Ромбическая спираль
procedure RombGridSpiral;
begin
  cmd[1]:='A 0 14 0 B';
  cmd[2]:='B 0 14 0 C';
  cmd[3]:='C 0 14 0 E';
  cmd[4]:='E 0 14 0 F';
  cmd[5]:='F 0 14 0 J';
  cmd[6]:='J 0 13 -1 A';
  cmd[7]:='A 14 13 0 A';
  cmd[8]:='A 13 14 0 A';
  cmd[9]:='J 14 13 0 A';
  cmd[10]:='J 13 14 1 A';
  cmd[11]:='END';
End;
```

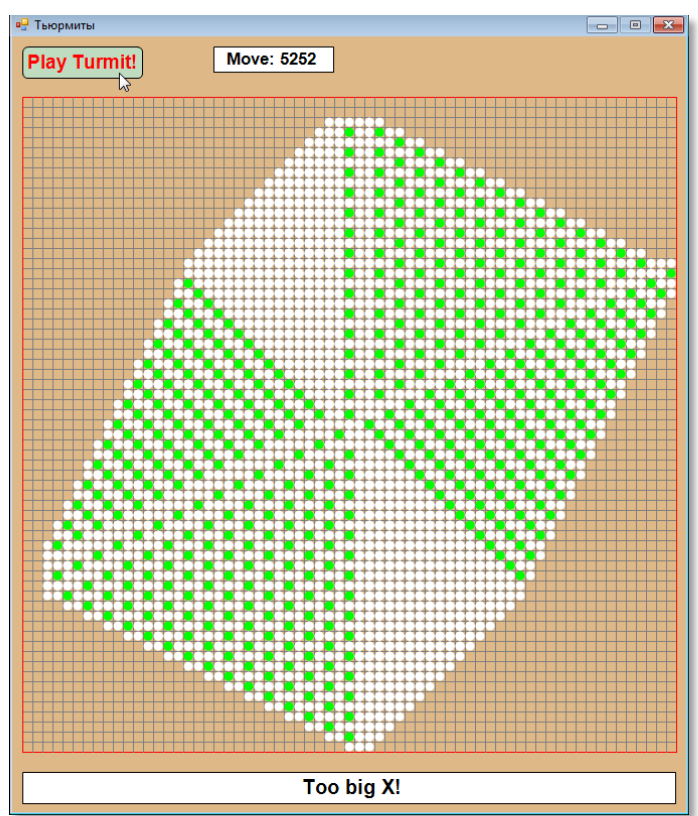


Рис. 24.26. Клубок

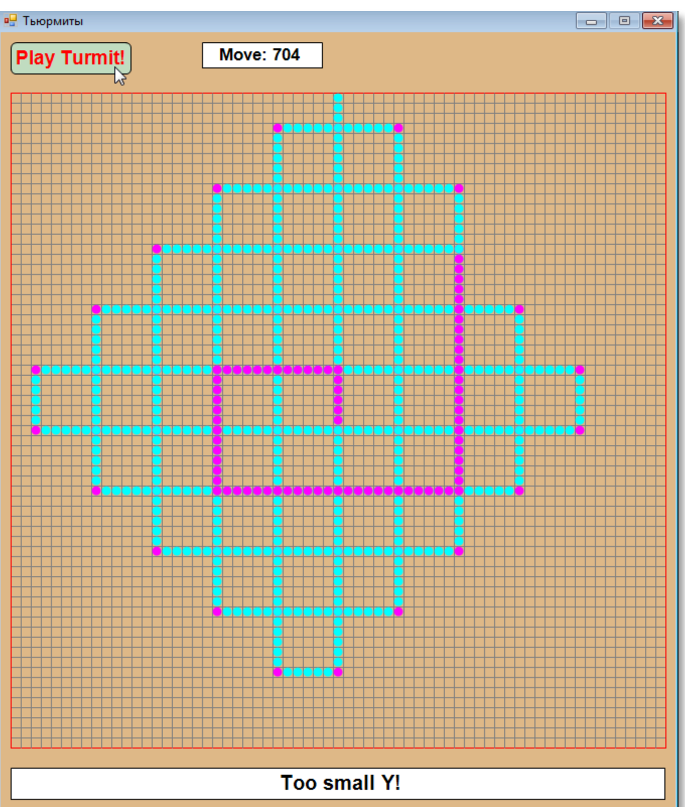


Рис. 24.27. Ромбическая спираль

Исходный код программы находится в папке **Turmits**.

1. Вы, конечно, заметили, что очень неудобно переходить от одного инстинкта к другому – нужно каждый раз делать исправления в исходном коде программы. Можно либо установить необходимое число кнопок – по одной на инстинкт, либо ограничиться одной кнопкой, но при новом нажатии на неё выводить следующую фигуру.

2. У наших тьюрмитов есть и более примитивный родственник – *муравей Лэнгтона (Langton's ant)*, названный так по фамилии его родителя Криса Лэнгтона (Chris Langton), который придумал его в 1986 году.

Он также беспрерывно ползает по бесконечной плоскости, разбитой на клетки двух цветов – белого и чёрного. За каждый ход муравей переползает в одну из четырёх соседних клеток. При этом он выполняет такие действия:

- в чёрной клетке муравей поворачивается на 90° *против часовой стрелки*, перекрашивает её в белый цвет и выполняет следующий ход;

- в белой клетке муравей поворачивается на 90° *по часовой стрелке*, перекрашивает её в чёрный цвет и выполняет следующий ход.

Ваша задача: написать программу, имитирующую бурную жизнедеятельность муравья Лэнгтона. Клетки поля можно произвольным образом выкрасить в чёрный и белый цвет, хотя, возможно, есть и более интересные раскраски, которые следует поискать (например, шахматное поле, дорожки, диагонали).

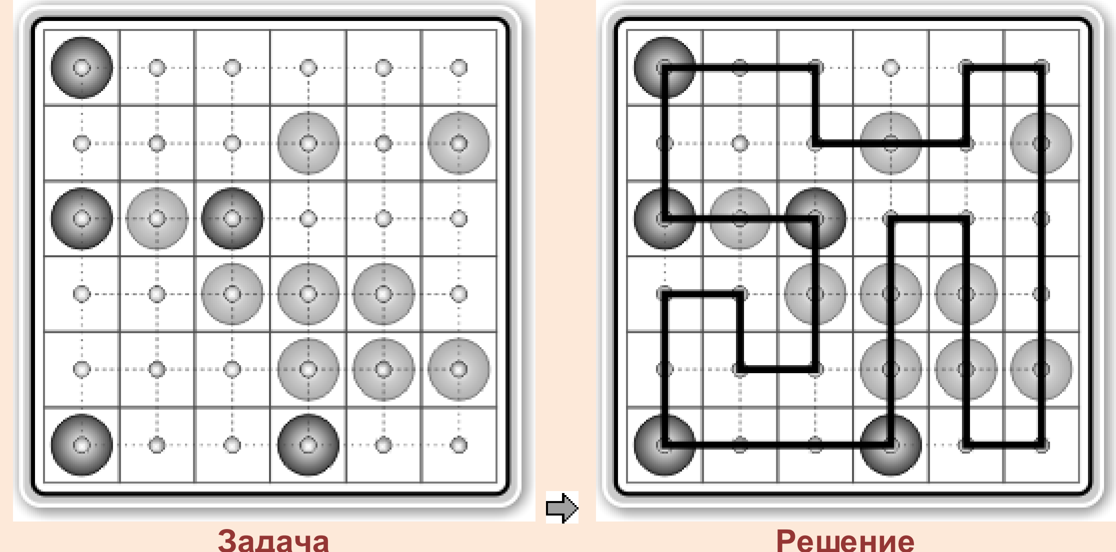
3. На сайте *Проект Эйлер* есть *Задача 349*, которая посвящена муравью Лэнгтона.

Проект Эйлер (*Project Euler*) находится на сайте *projecteuler.net*. Он посвящён решению математических задач на компьютере. Все задания представлены на английском языке, поэтому могут вызвать у вас затруднения, но вы найдёте их перевод на русский язык на сайте *euler.jakumo.org*.

Начало задачи мы пропускаем, поскольку жизненные правила муравья нам уже известны, и сразу же переходим к вопросу:

Пусть изначально поле состоит только из белых клеток. Сколько клеток муравей перекрасит в чёрный цвет после 10^{18} ходов?

4. Поведение муравья Лэнгтона весьма напоминает правила головоломки *Masyu*, в которой требуется провести через белые и чёрные кружкѝ замкнутую линию (цикл):



Задача **Решение**

Правила игры в Masyu

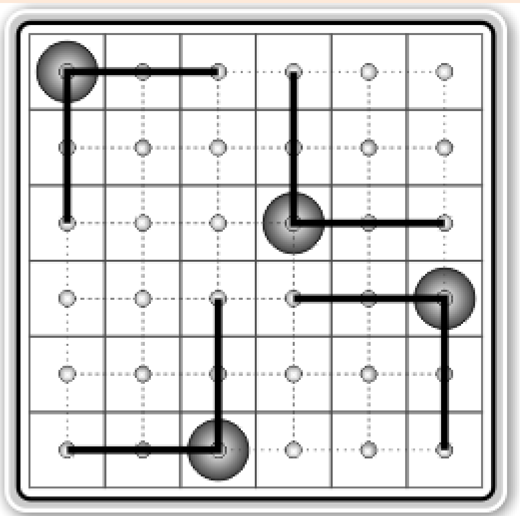
1. Искомый цикл представляет собой непрерывную замкнутую ломаную линию, состоящую из горизонтальных и вертикальных отрезков (звеньев), которые соединяют центры клеток. Он может изгибаться в центре пустых клеток или в клетках с чёрными кружками.

2. Звенья не должны пересекаться, а вершины цикла - касаться друг друга. Ни в одну клетку нельзя заходить дважды.

3. Цикл должен быть *единственным*.

4. Цикл должен пройти через *все* белые и чёрные кружки, но может не заходить в некоторые пустые клетки.

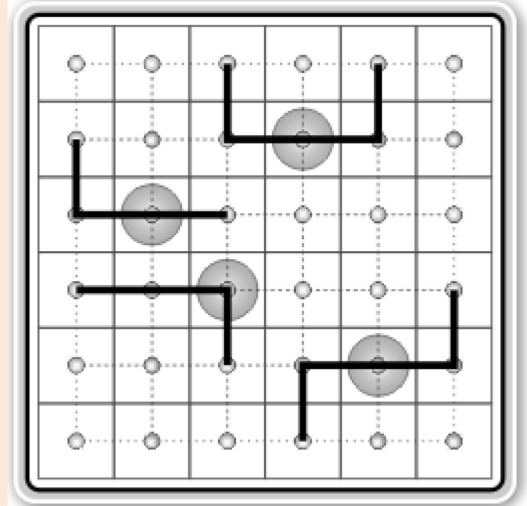
5. Проходя через клетку с чёрным кружком, цикл должен повернуть в ней на 90 градусов, а затем идти две клетки прямо в каждом направлении:



Допустимые варианты прохождения пути через чёрный кружок

Цикл также может менять направление в центре пустой клетки, если это не противоречит правилам.

6. Через белый кружок путь должен проходить прямо, но обязан повернуть, по крайней мере, в одной из двух смежных клеток:



Допустимые варианты прохождения пути через белый кружок

Напишите программу, решающую головоломку *Masyu*!

Урок 25. Элементы управления

Язык паскаль ничего не знает об *элементах управления* (*ЭУ*) – кнопках, метках, списках и прочих составных частях графического интерфейса пользователя (сокращённо – *ГУП* или *GUI, Graphical user interface*). Их необходимо создавать *самостоятельно*. К счастью, *PascalABC.NET* предоставляет в наше распоряжение несколько элементов управления, и мы можем дополнить интерфейс наших программ кнопками и другими *ЭУ*.

Например, практически любое приложение *Windows* должно иметь хотя бы одну *кнопку*, чтобы запускать и останавливать выполнение программы. Давайте создадим приложение с кнопкой.

Для этого мы воспользуемся одним из конструкторов класса *ButtonABC*:

```
constructor Create(x,y,w,h: integer; txt: string; cl: GraphABC.Color);
```

```
constructor Create(x,y,w: integer; txt: string; cl: GraphABC.Color);
```

Этот класс находится в модуле *ABCButtons*, который необходимо добавить к проекту:

```
uses
   GraphABC, ABCButtons;
```

Параметр *txt* – это надпись на кнопке. Первые 4 параметра задают *положение* кнопки в клиентской области окна и её *размеры*, а последний параметр – *цвет* кнопки.

Второй конструктор всегда создаёт кнопку высотой 30 пикселей, поэтому параметр *h* отсутствует.

Давайте начнём нашу новую программу с того, что добавим *кнопку* (Рис. 25.1):

```
//
//   ОСНОВНАЯ ПРОГРАММА
//
begin
   SetWindowTitle('Кнопки');
   SetWindowWidth(320);
   SetWindowHeight(240);
   . . .

   var btnTimer := new ButtonABC(10, Height-48, 120, 24, 'За-
пустить таймер', clMoneyGreen);
end.
```

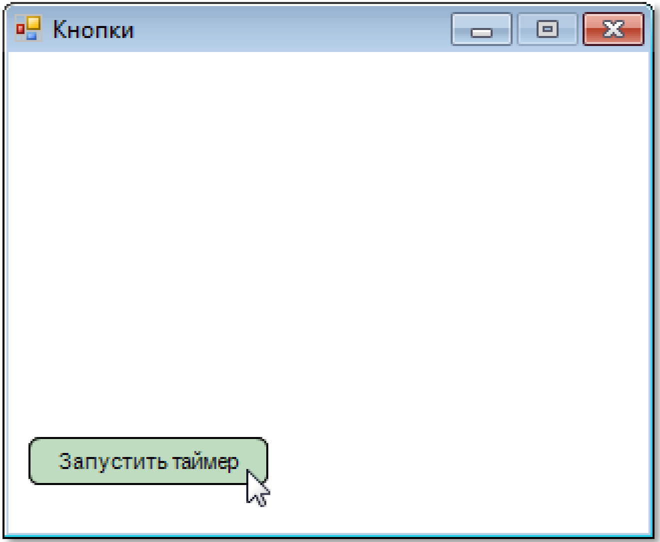


Рис. 25.1. Кнопка на месте!

Обратите внимание на то, что кнопке следует дать *имя*, по которому мы сможем потом к ней обращаться. Чтобы отличать названия кнопок от других идентификаторов программы, мы будем начинать названия кнопок с префикса *btn*. Это сокращение от английского слова *button*, что и означает *кнопка*.

Размеры кнопки можно изменить в любое время, присвоив нужные значения свойствам **Width** и **Height**. Первое отвечает за *ширину* кнопки, второе - за *высоту*. Добавим к программе пару строк:

```
btnTimer.Width:= 160;
btnTimer.Height:= 32;
```

И наша кнопка станет более солидной (Рис. 25.2).

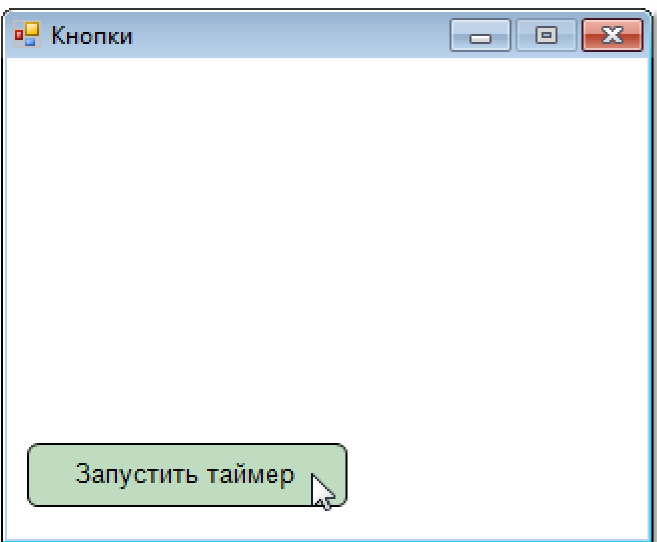


Рис. 25.2. Увеличенная кнопка

Иногда нужно *переместить* кнопку в другую позицию в окне приложения. Для этого у неё имеются свойства **Left** и **Top**:

```
btnTimer.Left:= 20;
btnTimer.Top:= Height-100;
```

Можно и *одновременно* изменить обе координаты, указав в скобках *новые* координаты левого верхнего угла элемента управления (Рис. 25.3).

```
btnTimer.Position:= new Point(20, Height-100);
```

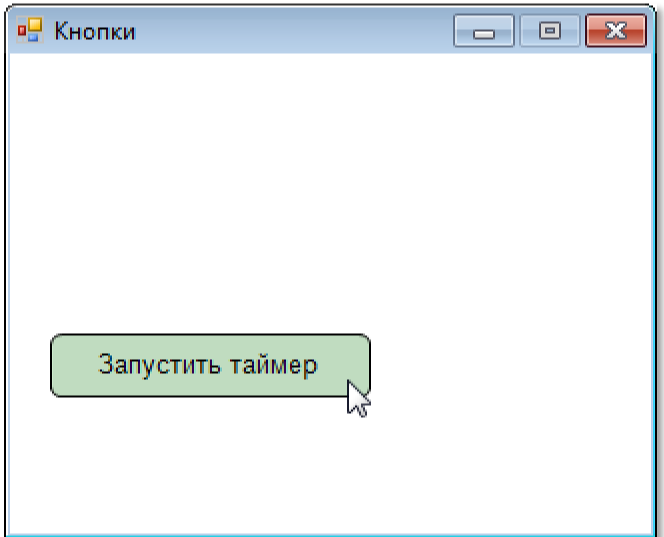


Рис. 25.3. Беглая кнопка

Такое же действие оказывает на кнопку и метод *MoveTo*, который перемещает её в заданное место окна. Воспользуемся этим методом, чтобы опять *разыграть* пользователя. Заставим кнопку беспрерывно ёрзать по экрану (Рис. 25.4):

```
var offset:=-1;
var left:=10;
var top:= Height-48;
While (True) do begin
    For var i:= 1 To 80 do begin
        left:=left-offset;
        top:=top+offset;
        btnTimer.MoveTo(left, top);
        Sleep(10);
      End;//For
      offset:= - offset;
End;//While
```

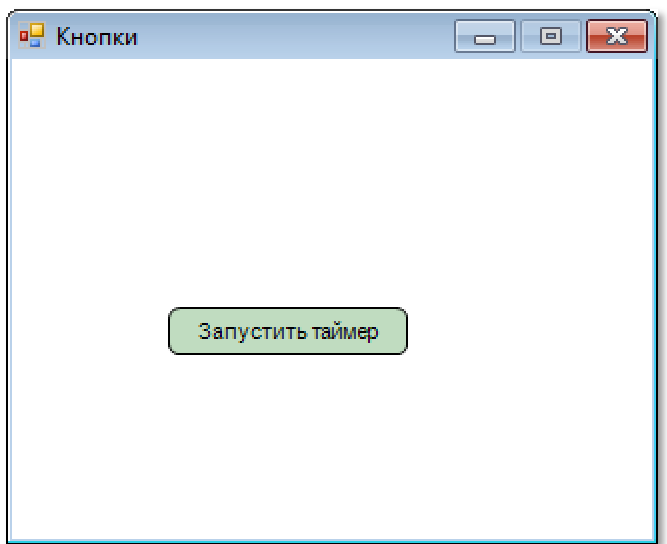


Рис. 25.4. Беспокойная кнопка

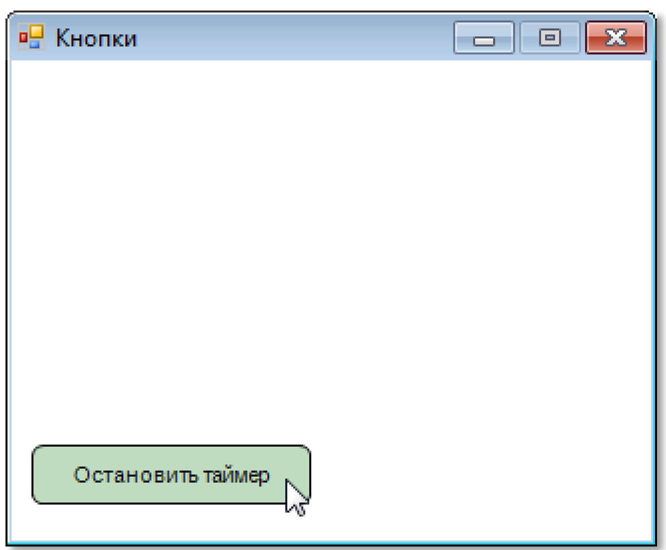
Вы уже, наверное, пробовали нажать на нашу кнопку и были раздосадованы, что таймер не запускается, несмотря на обещание, написанное на кнопке. Правильно, обещания нужно выполнять, поэтому добавим к программе процедуру-обработчик события *OnClick*, которое возникает при нажатии на кнопку:

```
btnTimer := new ButtonABC(10, Height-48, 140, 30, 'Запустить
таймер', clMoneyGreen);
//процедура-обработчик:
btnTimer.OnClick := btnTimer_OnClick;
```

С помощью свойства *Text*

```
procedure btnTimer_OnClick;
begin
   btnTimer.Text:='Остановить таймер';
end;
```

мы легко *заменим надпись* на кнопке так, чтобы она соответствовала новому состоянию программы (Рис. 25.5).



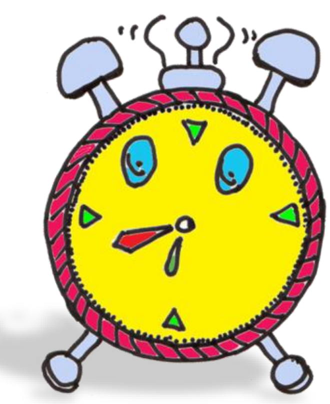


Рис. 25.5. Кнопка с новой надписью

Таким образом, нажав кнопку в *первый* раз, мы *запускаем* таймер, во *второй* раз – останавливаем его. Но вот незадача: после того как мы остановили таймер, должна вернуться первоначальная надпись, а этого не происходит. Исправляем оплошность:

```
procedure btnTimer_OnClick;
begin
   if (btnTimer.Text='Остановить таймер') then
       btnTimer.Text:='Запустить таймер'
   else
       btnTimer.Text:='Остановить таймер';
end;
```

Теперь надписи на кнопке правильные. Осталось добавить *таймер*, который также является элементом управления, но *невизуальным (невидимым).*

```
_timer:= new Timer(1000, timer_OnTick);
```

Первый параметр в конструкторе отвечает за время срабатывания таймера. В данном случае мы установили 1000 миллисекунд, что равняется одной секунде. Это значит, что через каждую секунду будет происходить событие *Tick*, которое мы сможем обработать в процедуре *timer_OnTick.*

Поскольку значение свойства *Interval* задается в *миллисекундах*, а нам нужны секунды, то придётся его поделить на 1000. Затем прошедшее после нажатия на кнопку время мы выводим в заголовке окна приложения:

```
procedure timer_OnTick;
begin
    time:= time + _timer.Interval/1000;
    Window.Title:= ' Прошло: ' + time.ToString();
end;
```

Так как мы хотим, чтобы таймер начал отсчёт времени только после нажатия на кнопку, то его нужно запустить в процедуре *btnTimer_OnClick* методом *Start:*

```
procedure btnTimer_OnClick;
begin
   if (btnTimer.Text='Остановить таймер') then
   begin
       btnTimer.Text:='Запустить таймер';
       _timer.Stop();
   end
   else begin
       btnTimer.Text:='Остановить таймер';
       _timer.Start();
   end
end;
```

Метод *Stop* останавливает таймер после его запуска методом *Start*. Вот теперь наш таймер работает как часы (Рис. 25.6)!

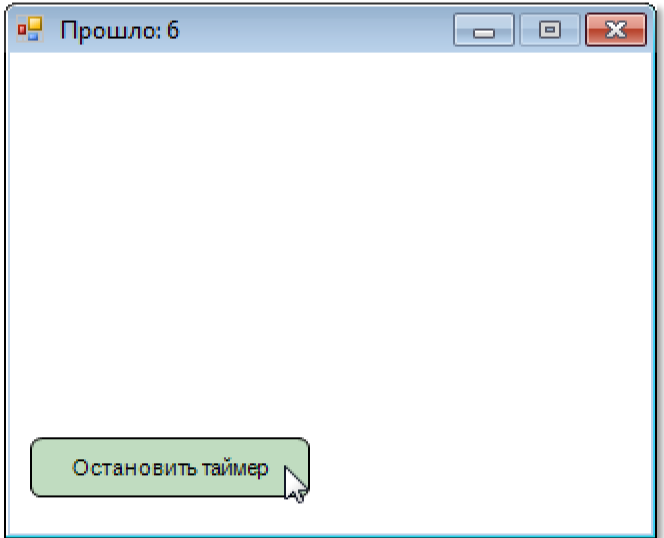


Рис. 25.6. Время пошло!

Выводить информацию, хотя бы и важную для нас, в заголовок окна вполне допустимо при отладке программы, но лучше использовать для этого элемент управления **метку**. В *паскале* такого *ЭУ* нет, но с ролью метки вполне справится прямоугольник типа *RectangleABC*, который описан в модуле *ABCObjects*, поэтому его также следует добавить к нашей программе:

```
uses
   GraphABC, Timers, ABCButtons, ABCObjects;
```

Объявите глобальную переменную типа *RectangleABC:*

```
lblTime: RectangleABC;
```

Префикс *lbl* – это сокращение от английского названия метки – *label*.

В основной программе мы создаём метку с помощью ключевого слова *new*:

```
lblTime:= new RectangleABC(10, 10, 120, 26, Color.White);
lblTime.FontStyle:= fsBold;
```

При этом вызывается конструктор

```
constructor Create(x,y,w,h: integer; cl: GColor := clWhite);,
```

который почти в точности повторяет конструктор кнопки, только без текстового параметра. Надпись на метке мы выведем в процедуре *timer_OnTick:*

```
procedure timer_OnTick;
begin
    time:= time + _timer.Interval/1000;
    Window.Title:= ' Прошло: ' + time.ToString();
    lblTime.Text:= ' Прошло: ' + time.ToString();
end;
```

Для того мы присваиваем свойству *Text* метки нужное значение (Рис. 25.7).

Помните, как нам пришлось стирать весь экран при выводе надписей в одно и то же место клиентской области окна? С *меткой* у вас таких проблем не будет.

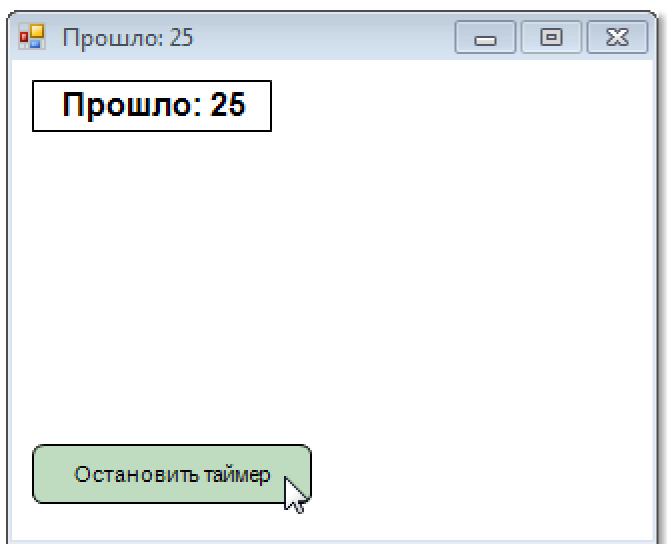


Рис. 25.7. Вот теперь совсем другое дело!

Размеры метки, а также её положение в окне приложения можно задать с помощью свойств *Width, Height, Left* и *Top*.

Заменим одну кнопку, которая отвечала за старт и остановку таймера, *двумя* (Рис. 25.8):

```
var
   btnStart, btnStop : ButtonABC;

  //КНОПКИ
```

```
   btnStart := new ButtonABC(10, Height-48, 140, 30, 'Запу-
стить таймер', clMoneyGreen);
   //процедура-обработчик:
   btnStart.OnClick := btnStart_OnClick;
   btnStop := new ButtonABC(170, Height-48, 140, 30, 'Остано-
вить таймер', clMoneyGreen);
   //процедура-обработчик:
   btnStop.OnClick := btnStop_OnClick;
```

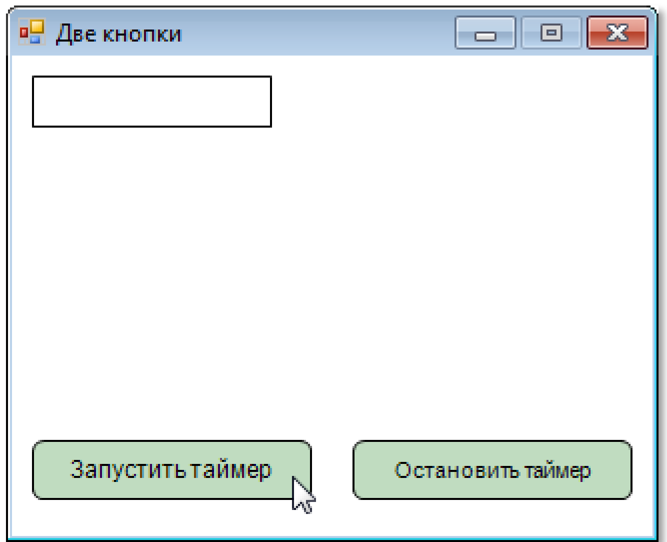


Рис. 25.8. Так удобнее?

Как говорится, одна кнопка хорошо, а две лучше. Однако правая кнопка сейчас лишняя, ведь таймер пока не запущен.

Дабы не искушать неискушенного пользователя, который обязательно нажмёт не ту кнопку и после будет роптать, уберём правую кнопку с глаз долой (рис. 25.9):

```
btnStop.Visible:= false;
```

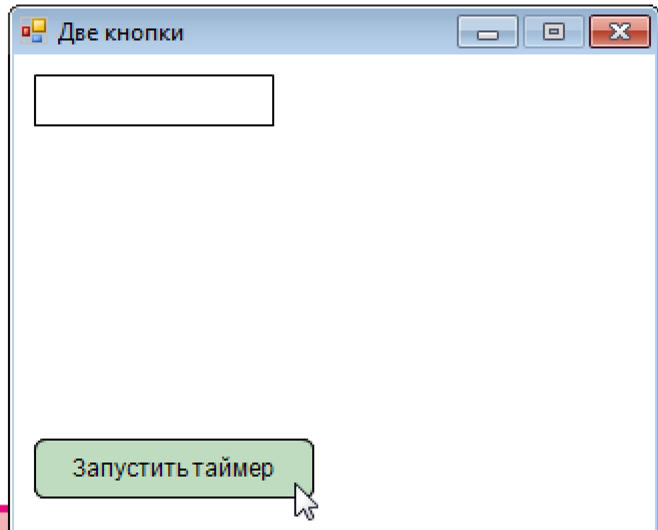


Рис. 25.9. Кнопка-невидимка

Для этого мы использовали свойство *Visible*, которое есть у кнопок. Если оно равняется *false*, то кнопка исчезает с экрана, если *true*, то снова появляется.

С двумя кнопками процедуры-обработчики стали проще:

```
procedure btnStart_OnClick;
begin
   btnStart.Visible:= false;
   btnStop.Visible:= true;
   _timer.Start();
end;

procedure btnStop_OnClick;
begin
   btnStop.Visible:= false;
   btnStart.Visible:= true;
   _timer.Stop();
end;
```

В них мы не только включаем и выключаем таймер, но и скрываем ненужную кнопку (Рис. 25.10).

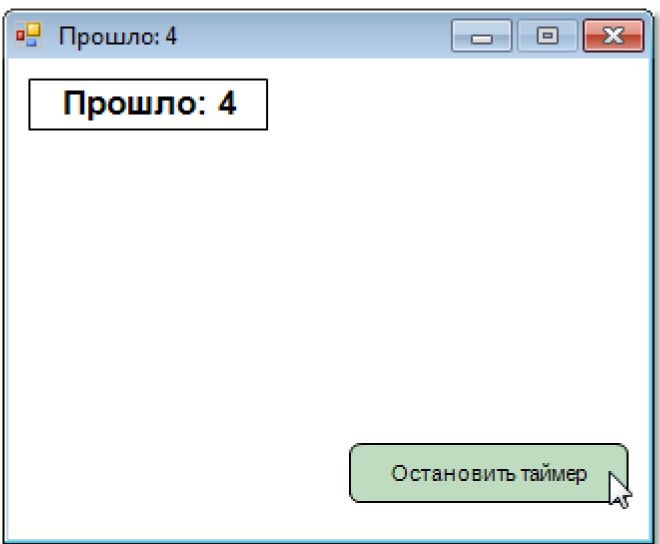


Рис. 25.10. Вторая кнопка-невидимка

Элементы управления в приложениях *Windows Forms*

На этом интерфейсный потенциал приложений с модулями *GraphABC, ABCButtons* и *ABCObjects* заканчивается. Добавление новых типов *ЭУ* - это уже весьма нелёгкая задача. А ведь для ввода и вывода информации нужны не только кнопки и метки, но и другие *ЭУ*, без которых в современных приложениях обойтись совершенно невозможно.

Приложения *FormsABC* предоставляют хороший набор *ЭУ*, но они располагаются не на форме, а на специальных панелях. Это облегчает как создание отдельных *ЭУ*, так и графического интерфейса в целом. Однако установить какой-либо *ЭУ* в нужном месте окна приложения невозможно. Этот вид приложений хорош, когда отсутствуют жёсткие требования к размещению *ЭУ*.

В приложениях *Windows Forms* доступны многочисленные *ЭУ* платформы *.NET*. Они позволяют создавать превосходный графический интерфейс пользователя.

К сожалению, в *ИСР PascalABC.NET* отсутствует визуальный конструктор формы, который имеется в *Delphi* и *Visual Studio*. Поэтому свойства *ЭУ* приходится устанавливать вслепую, что не очень удобно и значительно замедляет разработку программ. Поэтому разрабатывать приложения *Windows Forms* следует только тогда, когда вам требует качественный графический интерфейс!

Приложения *FormsABC* мы создавать не будем, поскольку они больше годятся для серьёзных, а не для занимательных проектов. А вот написать несколько приложений *Windows Forms* нам всё-таки придётся.

Чтобы начать приложение *Windows Forms* с чистого листа, можно воспользоваться шаблоном кода. Наберите буквы *wf*, а затем нажмите клавиши *Shift+ПРОБЕЛ.* В исходном коде появятся строчки:

```
{$reference 'System.Windows.Forms.dll'}
{$reference 'System.Drawing.dll'}

uses
  System, System.Windows.Forms;

begin
  Application.Run(new Form);
end.
```

Их достаточно, чтобы уже сейчас запустить приложение. Нажмите клавишу *F9* или кнопку *Выполнить*, и на экране возникнет пустое окно приложения *Windows Forms* (Рис. 25.11).

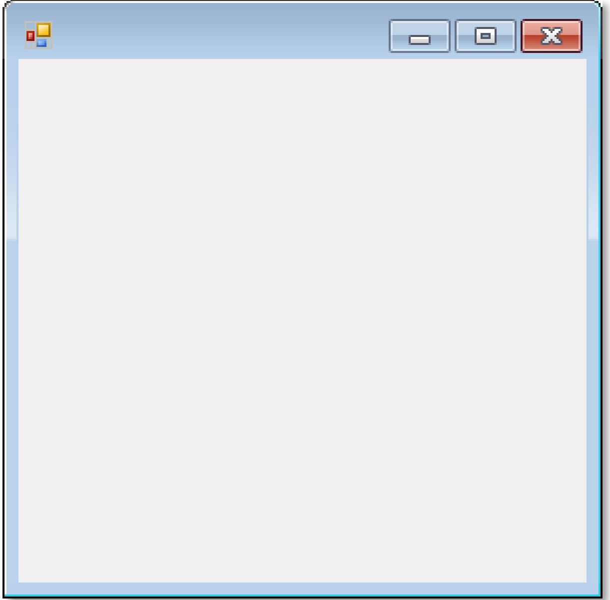


Рис. 25.11. Окно приложения *Windows Forms*

Конечно, такое безымянное окно вряд ли пригодно для дальнейшего использования, поэтому его нужно настроить, для чего мы немного изменим исходный код нашей программы:

```
//ПРИЛОЖЕНИЯ WINDOWS FORMS

{$reference 'System.Windows.Forms.dll'}
{$reference 'System.Drawing.dll'}

uses
  System, System.Windows.Forms;

var
```

```
   frmMain: Form;

begin
   frmMain := new Form;
   frmMain.Text := 'ПРИЛОЖЕНИЯ WINDOWS FORMS';
   frmMain.Width:= 400;
   frmMain.Height:= 320;
   frmMain.StartPosition:= FormStartPosition.CenterScreen;

   Application.Run(frmMain);
end.
```

Теперь окно имеет заголовок, заданные размеры и появляется в центре *Рабочего стола* (Рис. 25.12).

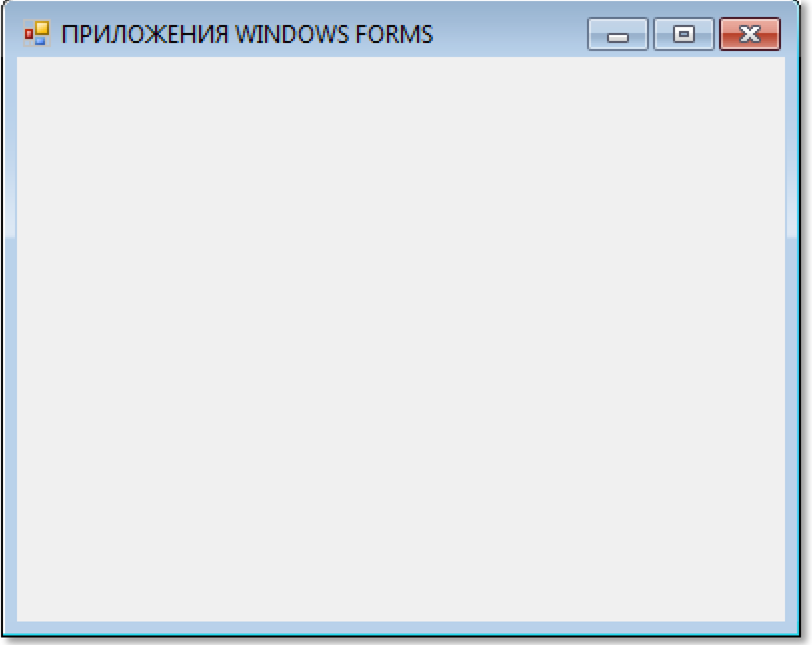


Рис. 25.12. Оформленная форма

Пришло время добавить *ЭУ*. Мы начнём с обязательной *кнопки*. Объявляем переменную типа *Button*:

```
var
   frmMain: Form;
   btnButton: Button;
```

А в основной части программы создаём новую кнопку и настраиваем её свойства:

```
  //кнопка:
   btnButton := new Button;
   btnButton.Text := 'Кнопка WF';
   btnButton.AutoSize := True;
   btnButton.Left := 10;
   btnButton.Top := frmMain.Height-70;
   frmMain.Controls.Add(btnButton);
   btnButton.Click += btnButton_Click;
```

В большинстве приложений кнопка реагирует только на нажатие, которое мы обрабатываем в процедуре *btnButton_Click*:

```
procedure btnButton_Click(sender: Object; e: EventArgs);
begin
   btnButton.Text := 'Привет!';
End;
```

Пусть наша кнопка поприветствует нас добрым словом! Запускаем приложение и видим, что кнопка на месте и готова быть кликнутой (Рис. 25.13).

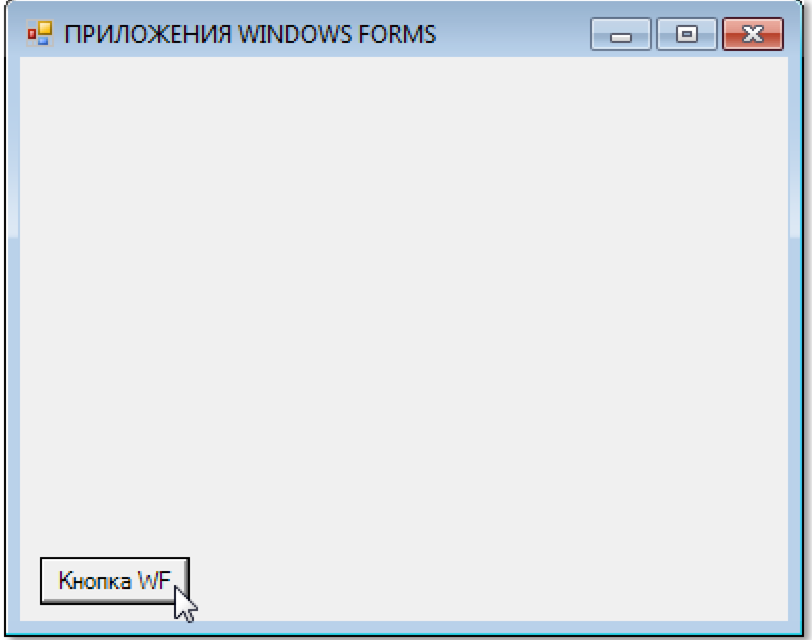


Рис. 25.13. Кнопка *Windows Forms*

Сгорая от нетерпения, жмём кнопку мышкой, на что она вежливо отзывается (Рис. 25.14).

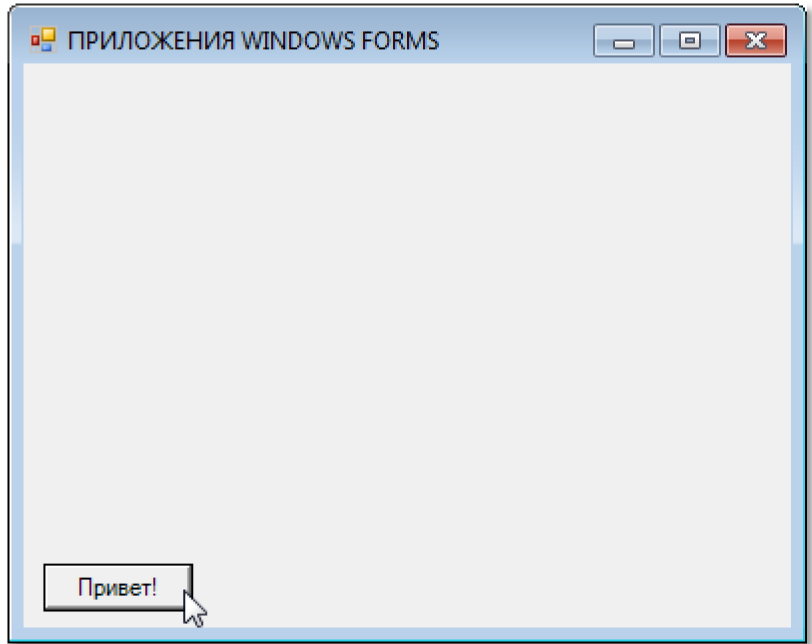


Рис. 25.14. Кнопка с приветом

Стандартные кнопки не очень красивы, поэтому вы можете придать им более привлекательный вид, присвоив свойству *FlatStyle* значение *FlatStyle.Flat* (Рис. 25.15):

```
btnButton.FlatStyle:= FlatStyle.Flat;
```

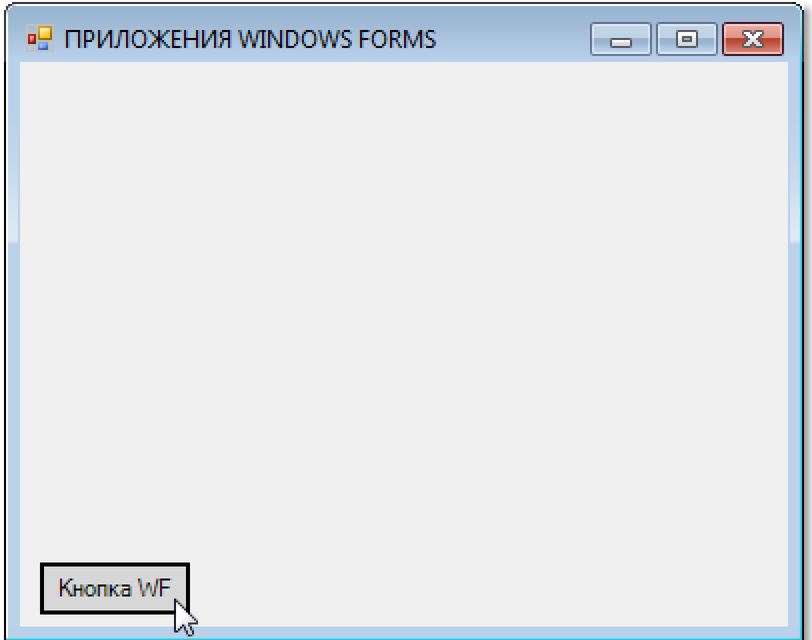


Рис. 25.15. Плоская кнопка

Пришёл черёд меток.

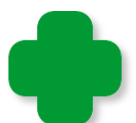
Объявляем переменную типа метки:

```
lblLabel: System.Windows.Forms.Label;
```

В данном случае необходимо дополнить тип пространством имён *System.Windows.Forms*, чтобы указать компилятору, что мы имеем в виду элемент управления метку, а не ключевое слово паскаля *label*. Иначе программа закончится с ошибкой.

В основной программе мы создаём и настраиваем новый *ЭУ*:

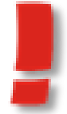
```
    //метка:
   lblLabel:= new System.Windows.Forms.Label();
   lblLabel.Left:= 10;
   lblLabel.Top:= 10;
   lblLabel.AutoSize:= true;
   lblLabel.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   lblLabel.ForeColor := Color.Red;
   lblLabel.BackColor := Color.Transparent;
   lblLabel.Text:= 'Метка WF';
   frmMain.Controls.Add(lblLabel);
```



Чтобы пользоваться цветами из структуры *Color,* подключите к проекту пространство имён *System.Drawing*:

```
uses
  System, System.Windows.Forms, System.Drawing;
```

Обычное назначение меток в приложении – вывод текста. Для этого нужно присвоить свойству *Text* строковое значение (Рис. 25.16).



Заставьте кнопку передавать нам приветы через метку!

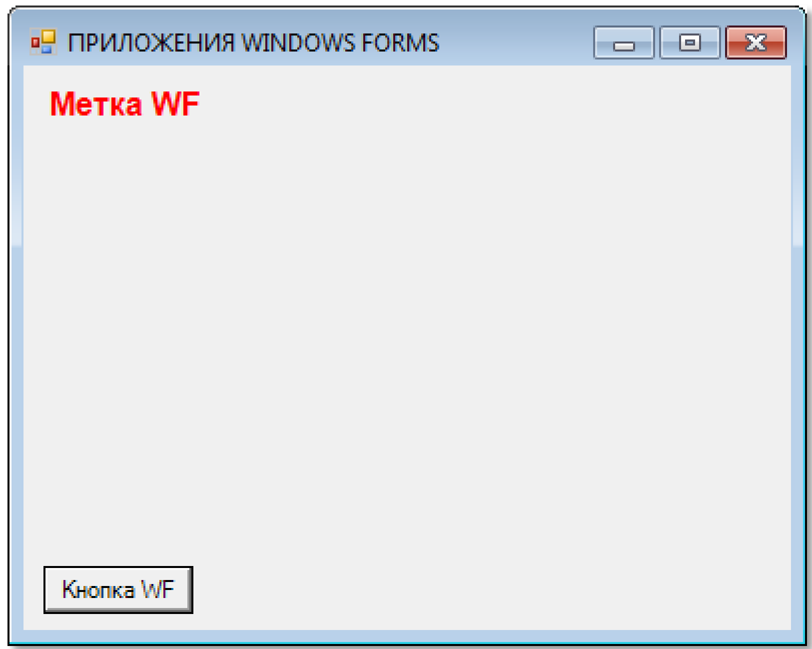


Рис. 25.16. Метка *Windows Forms*

Впрочем, кнопки и метки мы уже научились создавать и в приложениях *GraphABC*. И нам не стоило бы и огород городить только ради них. Но вот для ввода информации в программу с клавиатуры нам понадобится новый *ЭУ – текстовое поле*.

Объявляем переменную типа *TextBox*:

```
txtTextBox: TextBox;
```

В основной части программы создаём и настраиваем *текстовое поле*:

```
   //текстовое поле:
   txtTextBox:= new TextBox();
   txtTextBox.Width:= 160;
   txtTextBox.Left:= 10;
   txtTextBox.Top:= 40;
   txtTextBox.BorderStyle:= BorderStyle.FixedSingle;
   txtTextBox.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   frmMain.Controls.Add(txtTextBox);
```

После запуска программы оно возникает на форме (Рис. 25.17).

Рис. 25.17. Текстовое поле *Windows Forms*

В *текстовое поле* можно *вводить информацию* с клавиатуры (Рис. 25.18).

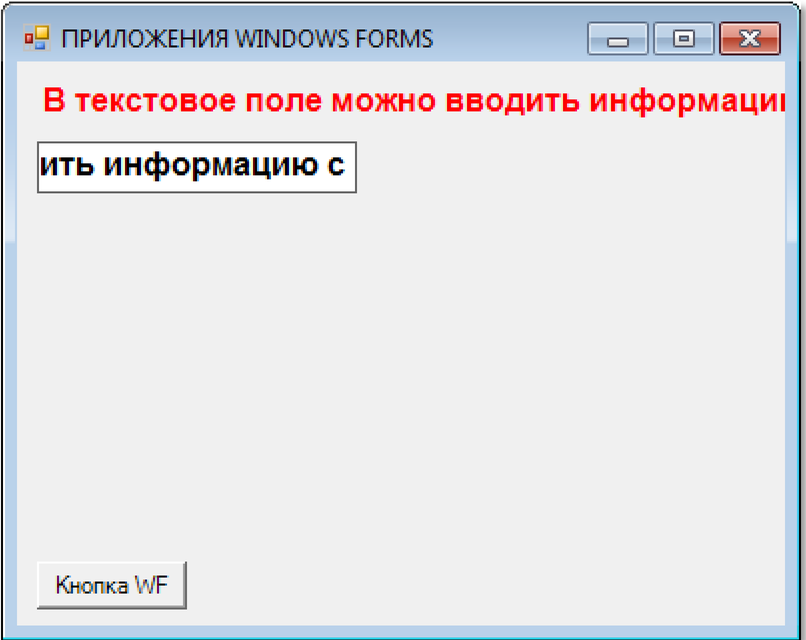


Рис. 25.18. Обратная связь с пользователем установлена!

А это очень важно, если мы хотим получить от пользователя программы какой-нибудь ответ или пароль.

Чтобы ввести текст, кликните мышкой внутри элемента управления. Так вы сделаете его *активным*, то есть передадите ему *фокус ввода*. При наборе текста возникает событие *TextChanged*, которое мы можем обработать в процедуре *txtTextBox_TextChanged*:

```
txtTextBox.TextChanged += txtTextBox_TextChanged;

procedure txtTextBox_TextChanged(sender: object; e:EventArgs);
begin
   lblLabel.Text:= txtTextBox.Text;
End;
```

Здесь мы из свойства *Text* получаем строку, записанную в *текстовом поле*, и выводим её в метку. Мы также можем присвоить это значение строковой переменной

```
var s := txtTextBox.Text;
```

и затем использовать, как нам заблагорассудится.

Текстовое поле довольно «умное»: вы можете выделять в нём текст, редактировать его, копировать и вставлять в другое текстовое поле или редактор текста!

Вы, конечно, заметили, что в *текстовом поле* можно набрать строку любой длины, но при этом видна только её часть. Чтобы прочитать длинную строку, придётся использовать клавиши со стрелками. Это не очень удобно, а если вывести длинный текст, состоящий из нескольких строк, то получится «бегущая строка». В этом случае нужно так изменить свойства *текстового поля*, чтобы оно могло печатать текст, состоящий из *нескольких* строк (мы будем называть его **многострочным текстовым полем**).

Объявим ещё одну переменную типа *TextBox*:

```
txtTextBox2: TextBox;
```

И в основной программе создадим новый *ЭУ*:

```
   //многострочное текстовое поле:
```

```
   txtTextBox2:= new TextBox();
   txtTextBox2.Multiline:= true;
   txtTextBox2.Height:= 170;
   txtTextBox2.Width:= 164;
   txtTextBox2.Left:= 10;
   txtTextBox2.Top:= 70;
   txtTextBox2.Font:= new System.Drawing.Font('Arial', 12,
System.Drawing.FontStyle.Bold);
   txtTextBox2.ScrollBars := Sys-
tem.Windows.Forms.ScrollBars.Both;
   txtTextBox2.WordWrap:= false;
   frmMain.Controls.Add(txtTextBox2);
```

Как вы видите, чтобы превратить обычное текстовое поле в *многострочное*, достаточно установить его свойство *Multiline* в *true*:

```
txtTextBox2.Multiline:= true;
```

Свойства *Width* и *Height* управляют размерами *текстового поля*. Однако часто бывает так, что текст настолько большой, что не помещается целиком в видимую часть ЭУ. Давайте-ка скопируем-ка в *текстовое поле* стихотворение *Парус* Лермонтова. На Рис. 25.19 мы видим, что у *текстового поля* появилась вертикальная *полоса прокрутки*, с помощью которой можно просмотреть весь текст, что гораздо удобнее, чем клавишами со стрелками.

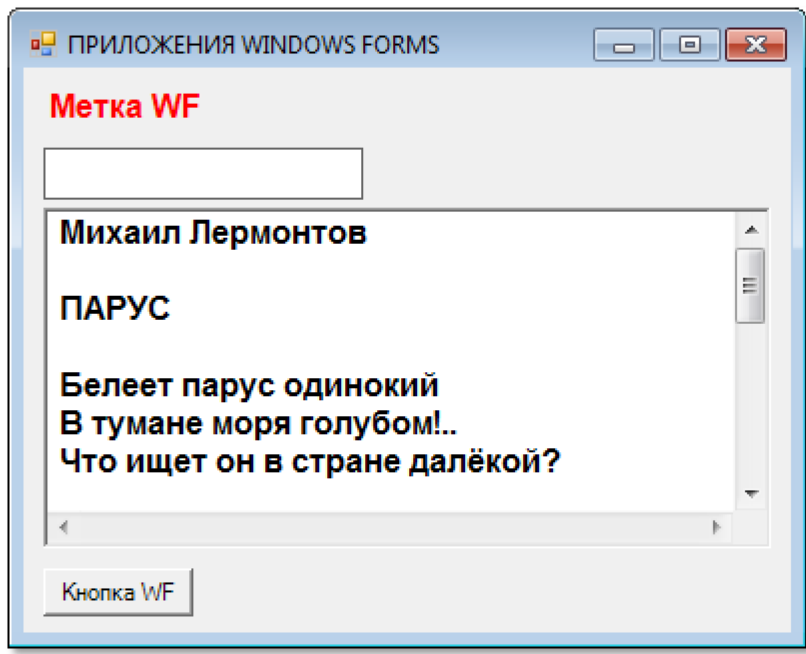


Рис. 25.19. Крути-верти!

Чтобы у текстового поля появились полосы прокрутки (вертикальная и/или горизонтальная), нужно присвоить свойству *ScrollBars* соответствующее значение. Если оно равно *ScrollBars.Both*, то - при необходимости - *ЭУ* будет обеспечен *двумя* полосами прокрутки.

И последнее свойство – *WordWrap* – отвечает за перенос длинных строк. При значении *false* строки не разрываются, а у *текстового поля* появляется горизонтальная полоса прокрутки, если длина строки больше его ширины.

Форматированный текст будет выведен, как обычный, поэтому забудьте о шрифтах, выравнивании и других изысках – здесь им места нет. Либо воспользуйтесь более сложным элементом управления *RichTextBox*.

Весь текст в многострочном *текстовом поле* хранится всё в том же свойстве *Text*, поэтому мы можем легко добраться до него, узнать его длину, а потом как нам угодно обработать строку:

```
var str:= txtTextBox.Text;
var len:= str.Length;
```

Если мы откроем *Панель элементов* в *Visual C#*, то найдём огромный список *ЭУ*, часть из которых показана на Рис. 25.20. Вы можете использовать в своих проектах любые из них. Однако нам вполне достаточно и тех, что мы рассмотрели на этом уроке.

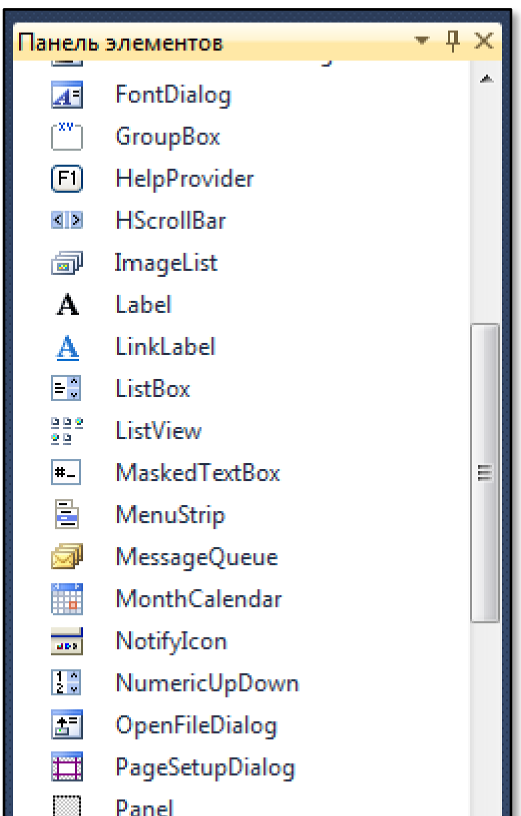


Рис. 25.20. Элементы управления платформы *.NET*

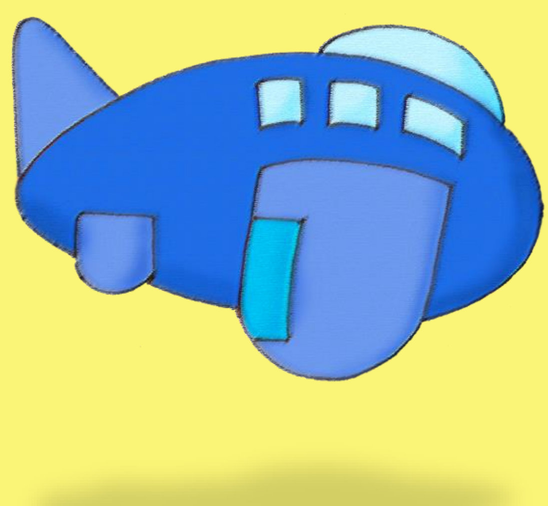
Исходный код программы находится в папке **Элементы управления**.

Урок 26. Занимательное моделирование

Природа говорит языком математики.

Галилео Галилей

Наверное, многим из вас приходилось собирать из отдельных деталей модели танков, самолётов или кораблей. Они похожи на настоящие образцы вооружений, но имеют меньший размер, сделаны из пластмассы и лишены многих полезных устройств и приспособлений, которыми обладают их прототипы. Однако эти модели умеют неплохо летать, плавать и преодолевать разнообразные препятствия на пересечённой местности. Таким образом, любая модель отражает не все, а только самые *важные* свойства и качества объекта. И чем точнее модель, чем она ближе к оригиналу, тем она сложнее и дороже (Рис. 26.1).



Подобные модели часто использовались при комбинированных съёмках кинофильмов. Несколько десятилетий назад, когда компьютеры ещё не годились для этих целей, приходилось мастерить «бутафорные» модели. Вы наверняка видели такие фильмы. В них летят под откос игрушечные составы, тонут в бассейне полуметровые корабли и горят в песочницах деревянные танки…

С давних времен модели применяются также в науке и технике. Прежде чем построить огромный мост, высоченную башню или скоростной самолёт, их уменьшенные копии трясут, мнут и обдувают потоком воздуха в аэродинамических трубах.

Но модели бывают не только реальные, сделанные из металла или пластмассы, но и виртуальные, которые существуют только

в виде формул, которые описывают саму модель и её поведение. Такие модели называют *математическими*.

Поскольку компьютер с удовольствием считает по формулам, то мы вполне можем написать компьютерную программу, которая не только выдаст огромный массив чисел, показывающих поведение модели в различных условиях, но и покажет результаты этих расчётов в виде графиков, чертежей, рисунков и анимации. Такое представление модели в памяти компьютера и на экране монитора называется *компьютерным моделированием*.

Компьютерные модели более наглядны, чем математические, и позволяют найти новые закономерности, которые трудно выявить только по числовым данным эксперимента или расчетов по формулам.

На этом и следующем уроках мы создадим несколько простых, но занимательных биологических и физических моделей.

Рис. 26.1. Макет города Амстердама с двигающимися моделями самолетов

Кролики

Кролики - это не только ценный мех,
но и 4-5 килограммов вкусного,
легко усваиваемого мяса!

Юмористические Братцы-кролики

Первую математическую модель, связанную с кроликами, разработал известный нам Леонардо Фибоначчи в трактате *Книга абака*.

Он взял пару взрослых кроликов (точнее, кролика и крольчиху) и предположил, что они могут производить на свет потомство каждый месяц. Причем у них всегда рождается пара крольчат разного пола, у которых через два месяца также рождаются крольчата.

Поскольку все кролики приходятся близкими родственниками друг другу, то такая популяция обречена на быстрое вымирание, но в математической модели этот факт не учитывается.

Фибоначчи решил подсчитать, сколько будет кроликов через год, если за это время ни один кролик не умрет.

«Эксперимент» он начал в *январе*, когда у него была только *пара* кроликов.

В *феврале* семья пополнилась парой крольчат, итого кроликов стало *две пары*.

В *марте* родительская пара принесла ещё пару крольчат, а первым крольчатам исполнился месяц. Итак, в марте семья кроликов состояла из *трёх пар* кроликов.

В *апреле* родительская пара снова произвела на свет пару крольчат, а их первое потомство достигло половозрелого возраста и внесло свой вклад в дело размножения семьи, которая теперь насчитывала *5 пар*.

Просматривая этот сериал про семейство кроликов и дальше, мы сможем составить таблицу. Её часть вы можете видеть на Рис. 26.2. Полная таблица имеет невероятные размеры, ведь мы знаем, что в июле кроликов будет 21 пара, в августе – 34 пары, в сентябре – 55 пар, в октябре – 89 пар, в ноябре – 144 пары, в декабре – 233 пары. Поскольку цыплят считают по осени, а кроликов парами, то через год в вольере счастливое семейство кроликов будет насчитывать 233 пары кроликов. У первой пары кроликов появятся 11 пар внуков, 9 пар правнуков, и так далее.

Кролики и лисицы

- Гонялся, гонялся Братец Лис
за Братцем Кроликом, и так и этак ловчился,
чтобы его поймать.
А Кролик и так и этак ловчился,
чтобы Лис его не поймал.

Джоэль Харрис,
Братец Лис и Братец Кролик

А теперь мы разработаем более правдоподобную математическую модель, также связанную с кроликами.

Как мы видели, в модели Фибоначчи кроликам всегда было достаточно травы, чтобы питаться и неограниченно размножаться. Подобный случай наблюдался в середине XIX века, когда в Австралию завезли кроликов, у которых не оказалось ни серьёзных врагов, ни пищевых конкурентов.

Мы рассмотрим модель, достаточно благоприятную для кроликов, то есть будем считать, что травы для них вполне достаточно. Не будем ограничивать и продолжительность их жизни, но поселим рядом с ними *лисиц*, которые, в свою очередь, будут питаться кроликами.

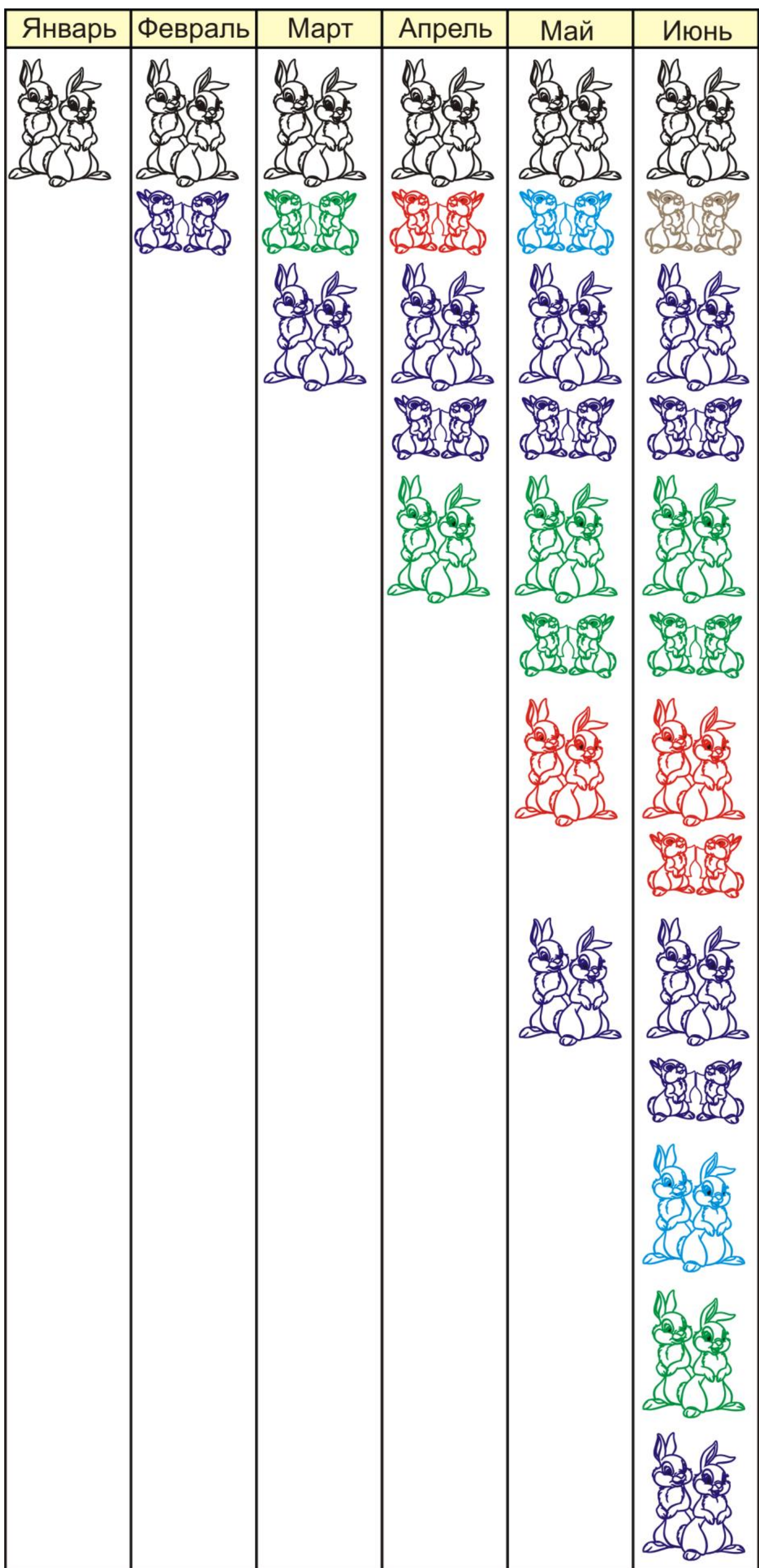


Рис. 26.2. Размножение фибоначчиевых кроликов

В начале эксперимента численность популяций кроликов и лисиц равна *nRabbit* и *nFox*, соответственно. Поскольку корма кроликам вполне хватает, они начинают усиленно размножаться, как им и предписано природой. Большое число кроликов приводит к тому, что ими может прокормиться и большее число лисиц, то есть вслед за ростом популяции кроликов начинает расти популяция лисиц. Они начинают поедать всё больше кроликов, а затем из-за недостатка «крольчатины» погибает всё больше лисиц. Но чем меньше становится лисиц, тем стремительнее размножаются кролики. Мы можем предположить, что численность популяций кроликов и лисиц будет изменяться *периодически*. Для проверки нашей гипотезы обратимся к математической модели этого процесса.

Она описывается системой дифференциальных уравнений, но это не должно вас пугать, так как компьютер решит их без проблем.

$$dR/dt = k_1GR - k_2RF$$
$$dF/dt = k_2RF - k_3F$$

Первое уравнение показывает, как со временем изменяется численность кроликов, а *второе* - лисиц.

Буквами *R*, *F* и *G* обозначено количество кроликов, лисиц и травы в неких условных единицах.

Коэффициенты k_1, k_2, k_3 определяют скорость размножения кроликов, скорость поедания кроликов лисицами и скорость вымирания лисиц из-за недостатка пищи.

В программе мы возьмём некоторые среднестатистические значения этих параметров, что ничуть не избавляет вас от необходимости проверить на практике и другие значения этих параметров.

```
//ПРОГРАММА ДЛЯ МОДЕЛИРОВАНИЯ
//ИЗМЕНЕНИЯ ЧИСЛЕННОСТИ ПОПУЛЯЦИЙ

uses
```

```
   GraphABC;

var
   height, width : integer;
   //коэффициенты:
   Q1:= 1;  //k3/k2
   Q2:= 1;  //k1/k2
   //относительное количество травы:
   Grass:= 1;
   //относительное количество кроликов и лисиц:
   nRabbit:= 0.44;
   nFox:= 0.40;
   //число циклов наблюдения:
   D:= 4000;
   //шаг по оси времени:
   T:=1;
   zoom:=100;

//
//   ОСНОВНАЯ ПРОГРАММА
//
begin
   SetWindowTitle('Кролики и Лисицы');
   SetWindowWidth(1000);
   SetWindowHeight(320);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   //Window.Clear(Color.Black);
   height := Window.Height;
   width := Window.Width;
```

После объявления переменных и настройки окна приложения мы готовим данные для изображения жизненного цикла популяций:

```
  //начальные популяции кроликов и лисиц:
   var x:= nRabbit;
   var y:= nFox;
```

А следующие переменные укажут нам *максимальное* и *минимальное* число особей каждого вида за время наблюдений:

```
  var minx:= 1000000.0;
   var maxx:=0.0;
```

```
   var miny:= 1000000.0;
   var maxy:=0.0;
```

А начнём мы их с определения количества точек, по которым будет построен график:

```
  var k:= Q1/(Q2*Grass);
   //число циклов:
   var n:= 1 + d/t;
```

Поскольку число циклов нам известно точно, то в цикле *For* мы определяем для каждого цикла численность обеих популяций. Из любопытства отслеживаем предельную численность популяций за всё время наблюдения:

```
  //циклы наблюдения:
   For var j:= 1 to Floor(n) do begin
      //вычисляем новые популяции:
      var x2:= x + (1 - y) * x * t/100;
      //dt= t/100 - приращение времени при интегрировании
      var y2:= y - (1 - x) * y * k * t/100;
      //новые популяции становится текущими:
      x:= x2;
      y:= y2;
      //печатаем текущие значения популяций:
      draw(j,x,y);

      If (x< minx) Then
         minx:=x;
      If (x> maxx) Then
         maxx:=x;
      If (y< miny) Then
         miny:=y;
      If (y> maxy) Then
         maxy:=y;
   end;//For
```

Числовые данные для наглядности мы преобразуем в график, а для его большей информативности добавляем к нему линии, соответствующие начальным популяциям кроликов и лисиц, а также вспомогательные прямые для отсчёта текущих параметров популяций:

```
  //чертим прямые, соответствующие начальным популяциям:
   SetPenWidth(3);
   SetPenColor(Color.Green);
   Line(0, Floor(height-nRabbit*zoom),
        width, Floor(height-nRabbit*zoom));
   SetPenColor(Color.Red);
   Line(0, Floor(height-nFox*zoom),
        width, Floor(height-nFox*zoom));
   //чертим прямые через 0.1 от начальной популяции:
   SetPenWidth(1);
   SetPenColor(Color.Gray);
   var i:= 0.0;
   while(i< maxy) do begin
      Line(0, Floor(height-i*zoom),
           width, Floor(height-i*zoom));
      i += 0.10;
   end;

  //печатаем макс. и мин. значения популяций:
   SetFontColor(Color.Green);
   SetBrushStyle(bsClear);
   SetFontStyle(fsBold);
   SetFontSize(12);
   Write('minx= ' + minx.ToString());
   Write('  maxx= ' + maxx.ToString());
   SetFontColor(Color.Red);
   Write('      miny= ' + miny.ToString());
   Write('  maxy= ' + maxy.ToString());
end.

// Строим график
procedure draw(a,b,c:real);
begin
   //ставим "точки":
   var x1:= Floor(a/4);
   var y1:= Floor(height-b*zoom);
   var x2:= x1;
   var y2:= Floor(height-c*zoom);
   SetBrushColor(Color.Green);
   FillEllipse(x1,y1, x1+3,y1+3);
   SetBrushColor(Color.Red);
   FillEllipse(x2,y2, x2+3,y2+3);

end;
```

Запустив программу, мы воочию убеждаемся, что наше предположение о циклическом изменении численности популяции оправдалось (Рис. 26.3). Более того, на графике хорошо видно, что кривая численности лисиц смещена вправо относительно кривой для кроликов, то есть популяция лисиц растёт и уменьшается вслед за изменением популяции кроликов с некоторым опозданием, что также правильно объяснила наша модель.

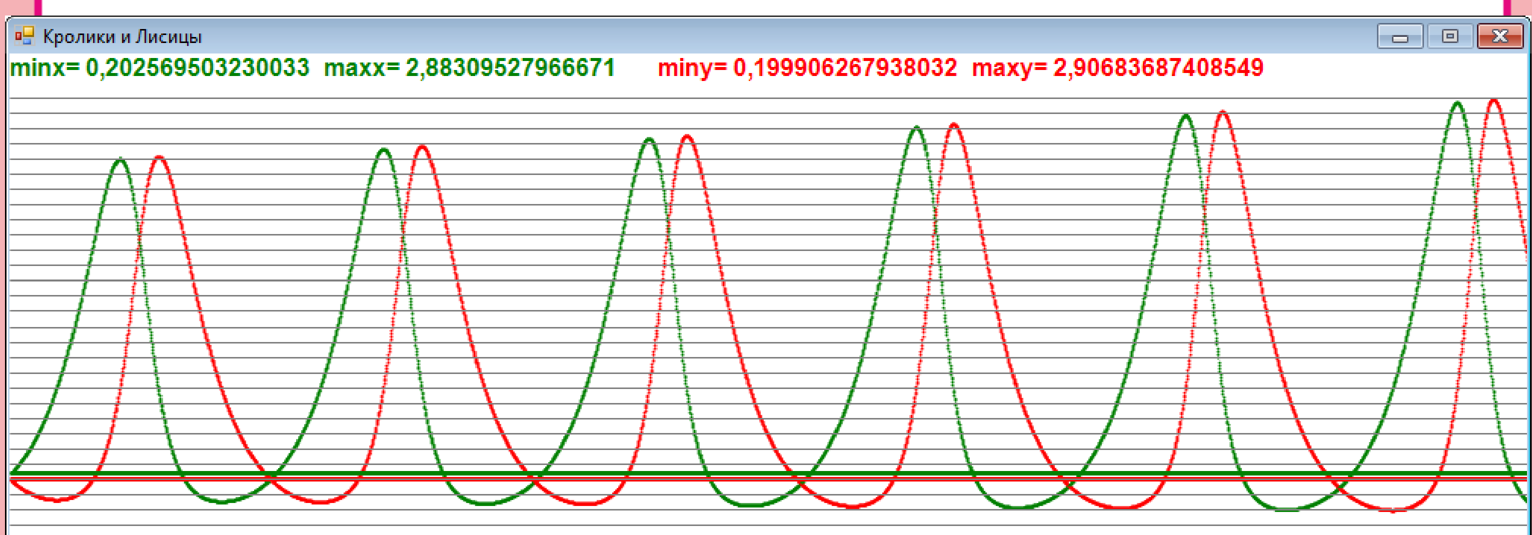
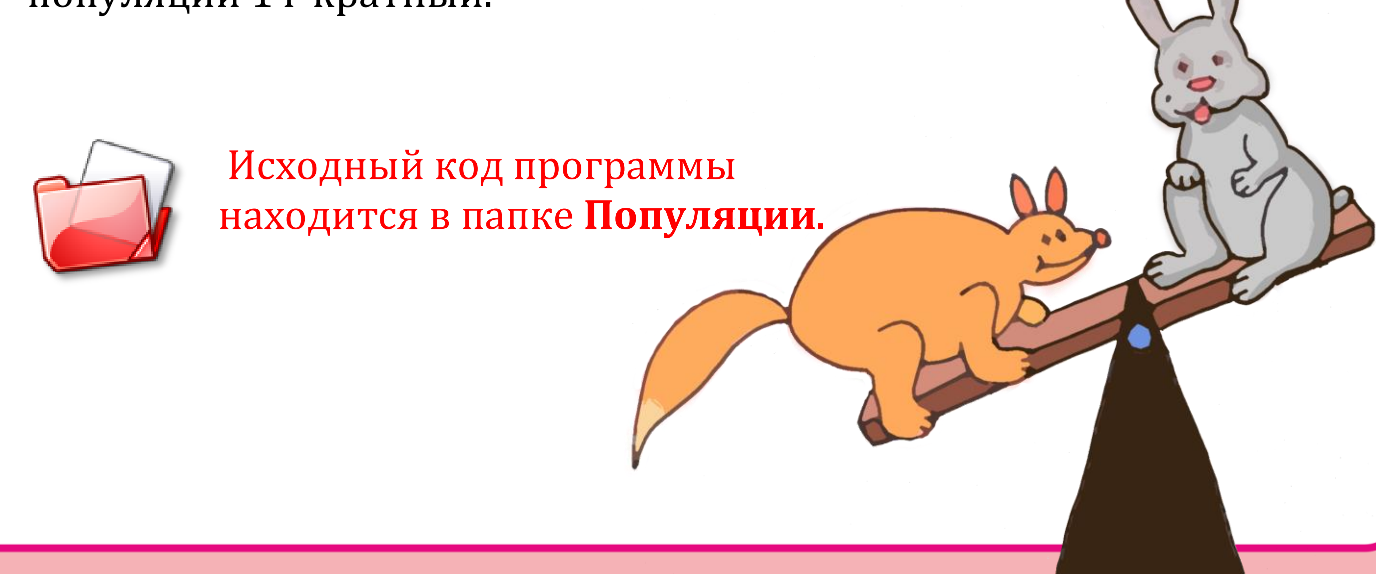


Рис. 26.3. Изменение численности кроликов (**зелёные** линии) и лисиц (**красные** линии)

Под заголовком окна мы найдём информацию о предельной численности популяций за время наблюдения. Как видите, численность и кроликов и лисиц *уменьшается* более чем в два раза, а *растёт* почти в семь раз. То есть размах колебаний численности популяций 14-кратный.



Исходный код программы находится в папке **Популяции.**

Урок 27. Занимательная физика

Вплоть до XVII в. учёные были уверены, что скорость падения тел определяется их массой, то есть тяжёлые предметы падают быстрее лёгких.

Действительно, если мы бросим с одинаковой высоты железную гирю и скомканный кусок бумаги, то убедимся в правоте этого суждения, которое, кстати говоря, высказал великий учёный древности Аристотель. Его авторитет в научной среде был настолько велик, что никому и в голову не приходило тщательно проверить его утверждение экспериментально.

Так как тела движутся в атмосфере Земли, то при падении на них действует сила сопротивления воздуха, что и приводит к таким результатам. Однако если тела имеют одинаковую форму, сила сопротивления воздуха будет сопоставима, и падать они будут практически с одинаковой скоростью.

Так продолжалось более двух тысяч лет, пока великий итальянский учёный *Галилео Галилей* не опроверг теорию Аристотеля. Согласно легенде, он одновременно сбрасывал с Пизанской башни тяжёлое пушечное ядро и гораздо более лёгкую мушкетную пулю. Оказалось, что оба предмета одновременно достигали земли, то есть скорость их падения была одинакова.

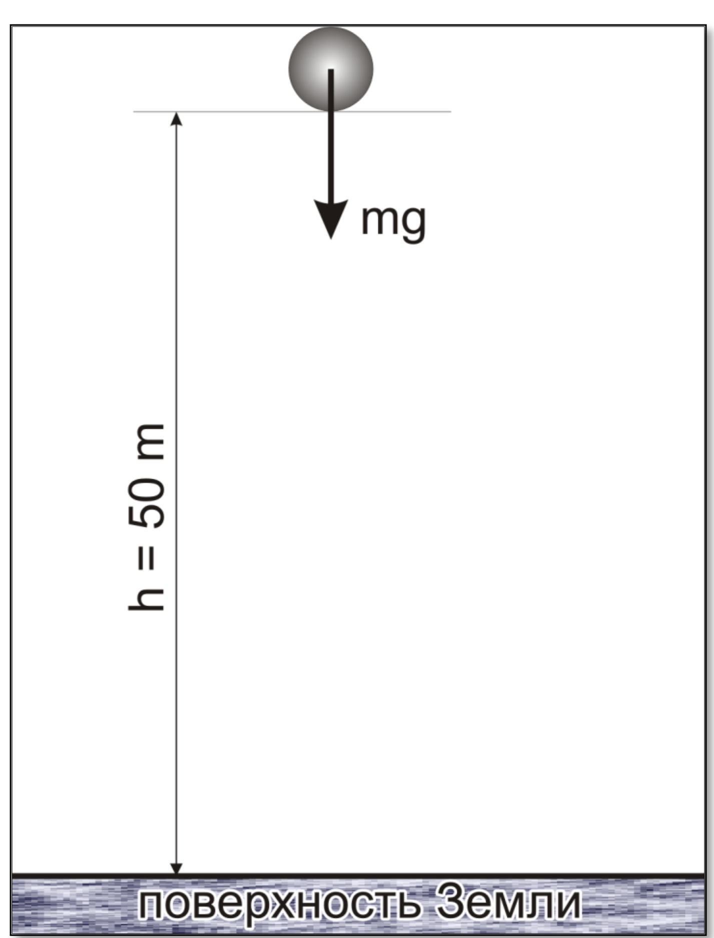
На самом деле Галилей провёл свой эксперимент гораздо «хитрее», ему не пришлось таскать тяжёлые ядра на вершину Пизанской башни. Он просто скатывал шары по наклонной доске и установил, что время скатывания шаров не зависит от их массы, причём этот вывод справедлив для разных углов наклона доски, из чего он и сделал вывод о том, что и при вертикальном падении объекты разной массы будут падать с одинаковой скоростью.

Более того, он опроверг и другое утверждение Аристотеля - что под воздействием силы тяжести тела движутся с постоянной скоростью. Его эксперименты показали, что шары катятся с *уско-*

рением. Если за первую секунду они прокатятся 1 метр (естественно, метров в то время ещё не было, но это не имеет значения), то за две - 4 метра, за три – 9 метров, и так далее, то есть скорость скатывания шаров увеличивается пропорционально времени.

Моделируем свободное падение

Мы не будем опровергать красивую легенду, а заберёмся на Пизанскую башню и сбросим оттуда шар. Общая высота башни составляет около 57 метров, но нам удобнее сбрасывать шары не с крыши, а с верхнего этажа, то есть с высоты примерно 50 метров (Рис. 27.1).



h – расстояние до поверхности Земли

mg – сила тяжести

Рис. 27.1. Шар в исходном положении

Для упрощения модели мы будем считать, что тело падает в безвоздушном пространстве, чтобы не учитывать сопротивление

воздуха при падении. Таким образом, наша модель больше годится для Луны, чем для Земли с её плотной атмосферой.

В этом случае все тела независимо от их массы должны падать с одинаковым ускорением, которое в физике принято обозначать буквой g. Его называют *ускорением свободного падения*. Вблизи поверхности Земли оно равно примерно 9,8 м/с^2.

Тогда скорость падения в зависимости от времени можно записать так:

$v_t = v_0 + gt$, где

v_0 – скорость падения тела в начале эксперимента, то есть на высоте h метров от поверхности Земли. Мы же собираемся просто выпустить шар из рук, поэтому начальная скорость будет равна нулю, а уравнение ещё больше упростится:

$v_t = gt$

Путь, пройденный телом при падении, легко рассчитать по формуле:

$S_t = v_0t + ½\ gt^2$,

а при $v_0 = 0$:

$S_t = ½\ gt^2$

Итак, в начальный момент времени $t = 0$ шар находится на расстоянии h метров от поверхности. Когда шар упадет на землю, он пролетит $S = h$ метров. Откуда

$½\ gt^2 = h$, а *время падения* составит

$t = \sqrt{2h/g}$

Для высоты 50 метров это время равно 3,19 секунды.

Модель падения шара готова, приступаем к её компьютерной реализации.

Начните новый проект и сохраните его в папке **Fall**.

Нам потребуется в программе только одна *константа* – ускорение свободного падения:

```
//ПРОГРАММА ДЛЯ МОДЕЛИРОВАНИЯ СВОБОДНОГО
//ПАДЕНИЯ ПРЕДМЕТОВ НА ЗЕМНУЮ ПОВЕРХНОСТЬ

uses
   GraphABC, ABCButtons, ABCObjects;

const
   //ускорение свободного падения:
   g= 9.8;
```

Введём также несколько *переменных*, назначение которых очевидно:

```
var
  //размеры окна:
  height, width: integer;
  //смещение шара от края окна в пикселях:
  xShar:= 322;
  //начальная высота шара в пикселях:
  hpix0:= 176;
  //начальная высота шара в метрах:
  h:= 50;
  //диаметр шара в пикселях:
  dShar:= 8;
  //уровень поверхности Земли в пикселях:
  yZemli:= 660;
  //элементы управления:
  lblTIme, lblRasst: RectangleABC;
  btnStart: ButtonABC;
  shar: CircleABC;
```

Значения в пикселях выбраны, исходя из фотографии Пизанской башни, которую мы будем использовать в качестве фона (Рис. 27.2).

Далее мы настраиваем *окно* приложения и загружаем *фоновую картинку* из файла:

```
//
// ОСНОВНАЯ ПРОГРАММА
//
begin
   SetWindowTitle('Свободное падение');
   SetWindowWidth(438+10);
   Window.Height:= 667+10;
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   //Window.Clear(Color.Black);
   height := Window.Height;
   width := Window.Width;
   //фоновая картинка:
   Window.Fill('pisa.jpg');
```

Добавим и толику *элементов управления*, чтобы придать компьютерному эксперименту б***о***льшую наглядность:

```
  //КНОПКА

   //Начать эксперимент:
   btnStart := new ButtonABC(10, Height-64, 80, 30,
'БРОСИТЬ!', clMoneyGreen);
   btnStart.FontColor:= Color.Red;
   btnStart.FontStyle:= fsBold;
   btnStart.TextScale:= 0.9;
   //процедура-обработчик:
   btnStart.OnClick := btnStart_OnClick;

   //МЕТКИ

   //Время падения:
   lblTIme:= new RectangleABC(10, 10, 100,26, Color.White);
   lblTIme.FontStyle:= fsBold;
   //Пройденное расстояние:
   lblRasst:= new RectangleABC(10, 48,100,26, Color.White);
   lblRasst.FontStyle:= fsBold;

   SetFontSize(14);
```

```
   SetFontColor(Color.Yellow);
   SetFontStyle(fsBold);
   SetBrushStyle(bsClear);
   TextOut(120, 14, '< Время падения');
   TextOut(120, 52, '< Пройденное расстояние');
```

На фотографии *уровень земли* виден не очень хорошо, поэтому прочерчиваем **толстую жёлтую линию**, которая будет обозначать поверхность Земли:

```
  //рисуем поверхность Земли:
   SetBrushStyle(bsSolid);
   SetPenWidth(5);
   SetPenColor(Color.Yellow);
   Line(0, yZemli, 480, yZemli);
```

Осталось нарисовать *шар* и запустить программу.

```
  //рисуем шар в исходном положении:
   shar:= new CircleABC(0,0,dShar, Color.Red);
   shar.Visible:=false;
   drawShar(0,0,0);
end.
```

К эксперименту мы подготовились отлично (Рис. 27.2), сам Галилей был бы доволен нашей работой!

Для рисования шара на экране мы используем фигуру *shar* – *красный* круг класса *CircleABCs*, так как его легко перерисовывать в разных положениях во время падения. Дополнительно мы будем отмечать *жёлтым кружком* текущее положение шара, чтобы нам было удобнее следить за его ускоренным падением.

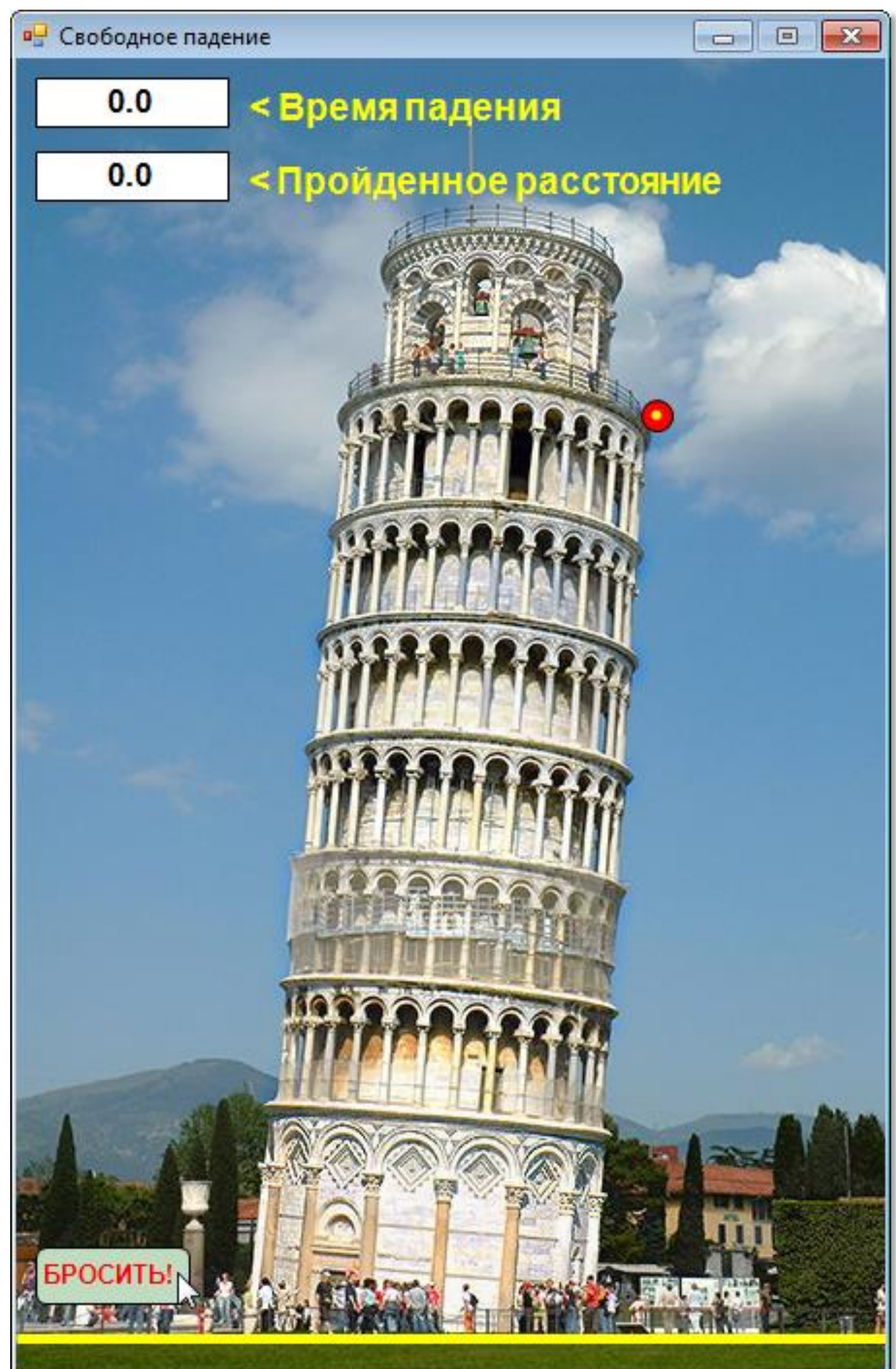


Рис. 27.2. Шар на Пизанской башне

```
procedure drawShar(n: integer; s,t: real);
begin
   //ht - текущая высота шара в метрах
   //расстоянию в 50 метров соответствуют на экране htpix пикселей:
   var htpix:= yZemli - hpix0 - dShar;
   //коэффициент пересчета метров в пиксели =
   //число пикселей, приходящихся на 1 метр:
   var kpix:= htpix/h;
   //текущая высота шара в пикселях на канве:
```

```
   var ypix:= hpix0 + kpix*s;

   //рисуем шар в текущем положении:
   shar.Position:= new Point(xShar, Floor(ypix));
   shar.Visible:= true;

   //отмечаем текущее положение шара кружком:
   If (n mod 10=0) Then begin
      SetBrushColor(Color.Yellow);
      var x1:= xShar+5;
      var y1:= ypix+5;
      FillEllipse(x1, Floor(y1), x1+6, Floor(y1)+6);
   End;//If
   //время падения шара:
   var dt:= Floor(t*100)/100;
   lblTime.RealNumber:=dt;
   //пройденное расстояние:
   var ds:= g * t * t /2;
   ds:= Floor(ds*100)/100;
   lblRasst.RealNumber:=ds;
End;
```

Единственная сложность в процедуре рисования шара – это пересчёт метров реального мира в пиксели виртуального. Кроме того, следует учесть, что при падении шара его вертикальная координата *увеличивается*, хотя расстояние до земли *уменьшается*.

Однако пора нажать на кнопку!

```
//Бросаем шар
procedure btnStart_OnClick;
begin
  fall();
End;
```

Шар начинает медленно, но с ускорением падать на землю:

```
//Падение шара
procedure fall;
begin
  var t:=0.0;
  //приращение времени в секундах:
  var dt:= 0.01;
```

```
  //текущая высота шара в метрах:
  var ht:= h;
  var n:= 0;
  While (True) do begin
     //пройденное расстояние:
     var s:= g*t*t/2;
     If (s >= h) Then begin//шар упал на землю
      s := h;
      t:= Sqrt(2*50/g);
      drawShar(n,s,t);
      //Sound.PlayClick()
      exit;
    End;//If
    drawShar(n,s,t);
    t:= t+ dt;
    Sleep(10);
    n:= n +1;
  End;//While
End;
```

Каждую сотую долю секунды мы вычисляем новое положение шара и рисуем его на экране. Из-за того что реальный шар пролетит 50 метров, а наш, нарисованный, всего несколько сотен пикселей, нам придётся отмечать его текущее положение только один раз из десяти, иначе все кружочки сольются в одну линию и полностью лишат нас научной информации о ходе эксперимента. Для этого мы ввели вспомогательную переменную-счётчик *n*. Также в конце каждого цикла мы делаем задержку на ту же одну сотую долю секунды, чтобы наш шар падал синхронно с настоящим:

```
Sleep(10);
```

Когда пройденное шаром расстояние станет равным начальной высоте шара, мы должны закончить цикл, так как ниже поверхности земли шар двигаться не может (правда, вмятину в земле он сделает, но к нашей модели это отношения не имеет).

Достаточно посмотреть на снимок экрана после завершения эксперимента (Рис. 27.3), чтобы убедиться: наша модель адекватно описывает свободное падение шара на землю.

Чтобы сымитировать звук падения шара на землю, нужно добавить к проекту пространство имён *System.Media:*

```
uses
   GraphABC, ABCButtons, ABCObjects, System.Media;
```

Объявить переменную:

```
playSound:SoundPlayer;
```

А в главной программе создать проигрыватель указанного звукового файла:

```
playSound:= new SoundPlayer('wav/buljk.WAV');
```

И наконец, в процедуре *fall* воспроизвести звук, когда шар упадёт на землю:

```
While (True) do begin
     //пройденное расстояние:
     var s:= g*t*t/2;
     If (s >= h) Then begin//шар упал на землю
      playSound.Play();
      s := h;
      t:= Sqrt(2*50/g);
      drawShar(n,s,t);
      exit;
    End;//If
```

Конечно, на картинке звук не передашь, но он совершенно очаровательный!

Во время эксперимента никто не пострадал!

Рис. 27.3. Эксперимент закончен

Исходный код программы находится в папке **Fall**.

Куда подальше!

Предположим, что мы не просто отпускаем шар в свободное падение, но и бросаем его с некоторой *горизонтальной* скоростью v_h. Поскольку вертикальная составляющая скорости при этом не изменится, то время падения шара останется прежним, то есть 3,2 секунды. Но в отсутствие воздуха шар за это время пролетит в горизонтальном направлении v_h x t метров, то есть упадёт не к подножию башни, а на расстоянии v_h x t метров от него. При этом шар будет двигаться не по вертикальной прямой, а по нисходящей ветви *параболы* (Рис. 27.4).

Перепишем исходный код предыдущего проекта в папку **Fall2** и тут же примемся за дело.

Добавим новую *переменную*, отвечающую за горизонтальную скорость шара:

```
//горизонтальная скорость шара, метров в секунду:
vh:= 10;
```

Увеличим ширину окна приложения

```
SetWindowWidth(688);
```

под новую картинку:

```
  //фоновая картинка:
  Window.Fill('pisa2.jpg');
```

В самом деле, если мы не добавим нашему шару жизненного пространства справа, то он просто улетит за пределы окна и эксперимент получится неполноценным!

Для слежения за расстоянием, которое пролетит шар в горизонтальном направлении, нам потребуется ещё одна *метка*:

```
  TextOut(120, 90, '< Пройденное расстояние по горизонтали');
```

```
  //Пройденное расстояние по горизонтали:
  lblRasstH:= new RectangleABC(10, 86,100,26, Color.White);
  lblRasstH.FontStyle:= fsBold;
```

Продлим *линию* поверхности земли вправо:

```
Line(0, yZemli, 688, yZemli);
```

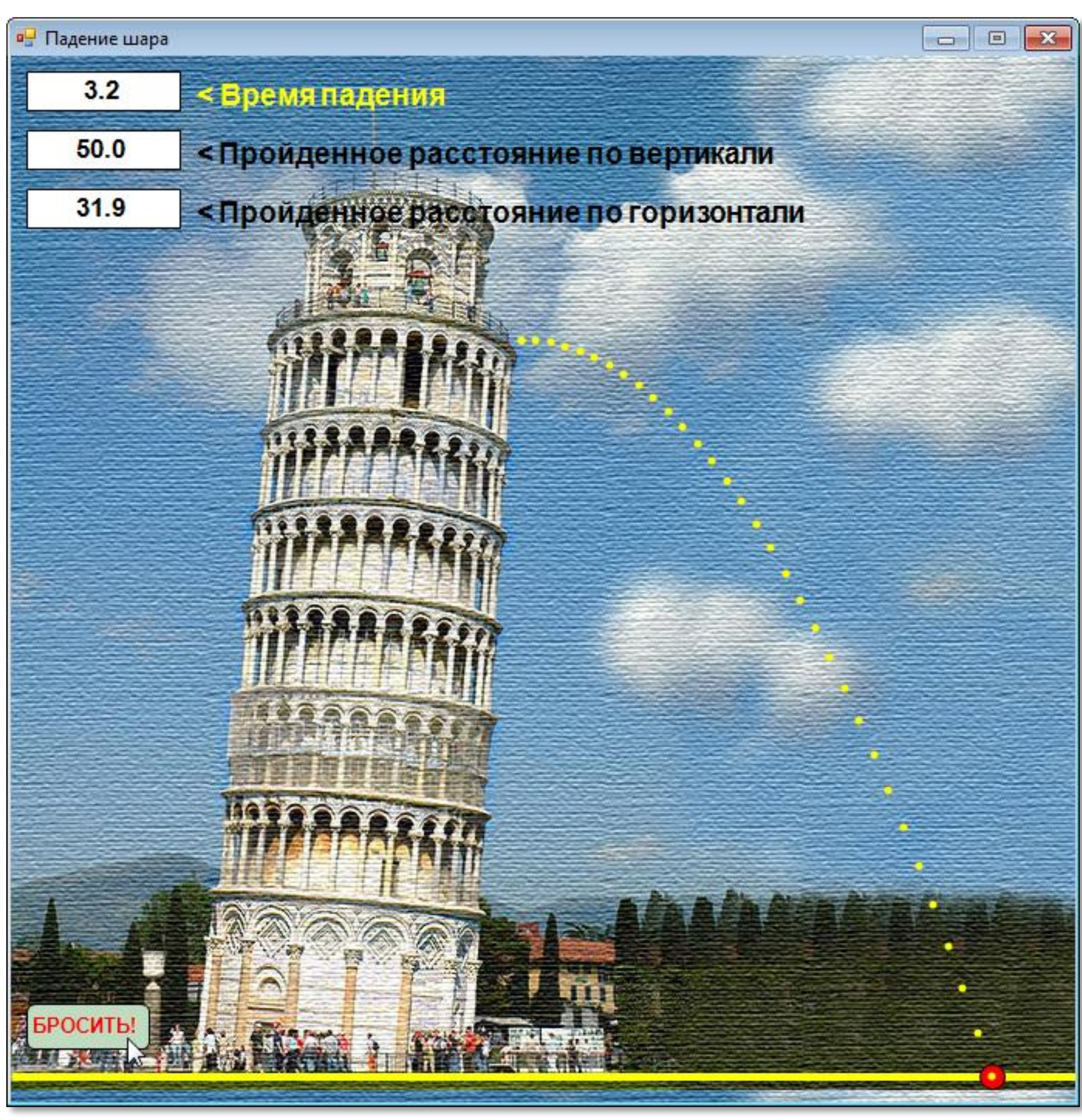


Рис. 27.4. Падение шара по параболе

И последнее, что нам следует сделать, – дополнить процедуру рисования шара:

```
procedure drawShar(n: integer; s,t: real);
   . . .

  //горизонтальная координата шара в пикселях:
   var xpix:= xShar + kpix* (vh* t);
   //рисуем шар в текущем положении:
   shar.Position:= new Point(Floor(xpix), Floor(ypix));

  //отмечаем текущее положение шара кружком:
   If (n mod 10=0) Then begin
      SetBrushColor(Color.Yellow);
      var x1:= xpix+5;
      var y1:= ypix+5;
      FillEllipse(Floor(x1), Floor(y1), Floor(x1)+6,
Floor(y1)+6);
   End;//If
  . . .
  //пройденное расстояние по горизонтали:
   var dsh:= vh * t;
   dsh:= Floor(dsh*100)/100;
   lblRasstH.RealNumber:=dsh;
End;
```

Как хотите, но теперь шар падает значительно красивее. Бросал бы его и бросал!

Исходный код программы находится в папке **Fall2**.

Стреляем ядрами

Давайте ещё более усложним нашу модель: поднатужившись, бросим шар не горизонтально, а под некоторым *углом* вверх-вправо.

В новом проекте **Fall3** добавим *переменные* для этого случая:

```
  //угол броска в градусах:
  alpha:= 60;
  //начальная скорость шара, метров в секунду:
  v0:= 15;
  alphaR:real;
  vh,vv,sv: real;
  curY: real;
```

Из прямоугольного треугольника (Рис. 27.5) мы легко найдём вертикальную и горизонтальную составляющие скорости шара в начальный момент времени:

```
   //угол броска в радианах:
   alphaR:= DegToRad(alpha);
   //горизонтальная скорость шара, м/с:
   vh:= v0 * Cos(alphaR);
   //начальная вертикальная скорость шара, м/с:
   vv:= v0 * Sin(alphaR);
   curY:= h;

   //рисуем шар в исходном положении:
   shar:= new CircleABC(0,0,dShar, Color.Red);
   shar.Visible:=false;
   drawShar(0,0,0);
end.
```

```
//Падение шара
procedure fall;
. . .
```

Поскольку теперь шар сначала поднимается вверх, а затем падает вниз с максимальной высоты, то прежняя формула расчёта вертикального пути не годится. Мы будем суммировать отдельные расстояния, которые пролетит шар по вертикали за время *dt*:

```
sv:= sv + Abs((vv - g*t) * dt);
```

принимая во внимание, что текущая вертикальная скорость равна

$vv_t = vv - gt$

```
//путь по вертикали:
sv:=0;
While (True) do begin
```

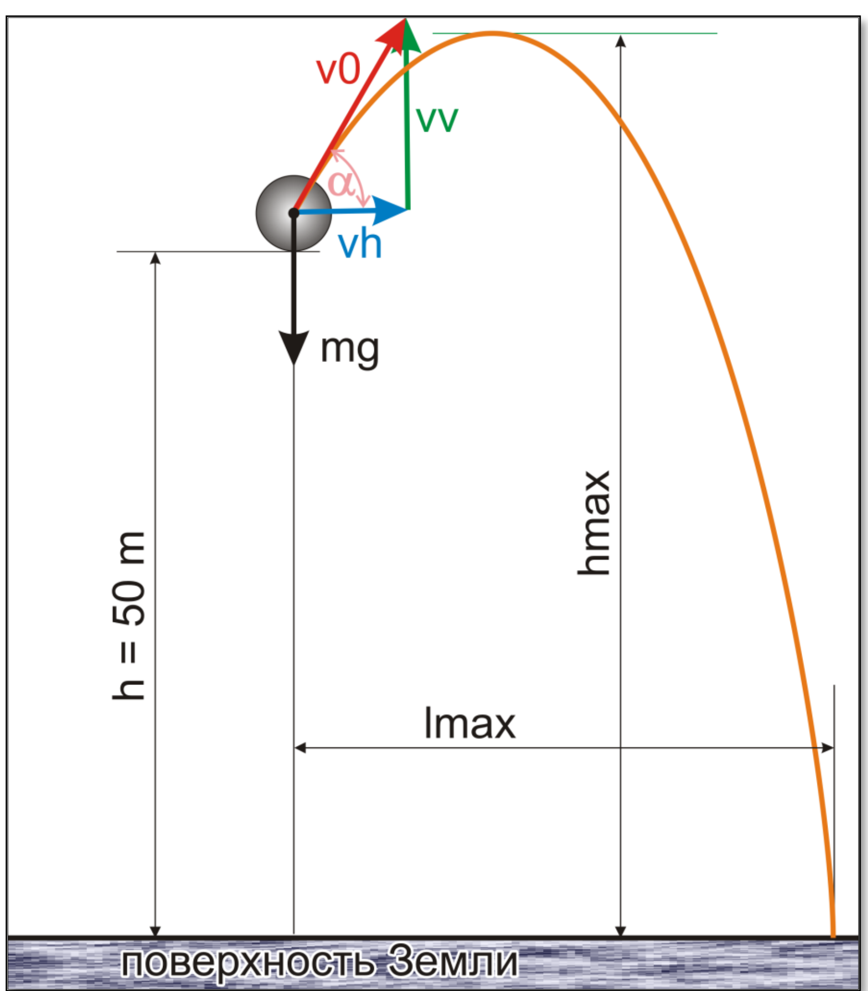


Рис. 27.5. Траектория полета шара, брошенного под углом к горизонту

Иначе придётся рассчитывать и текущую высоту шара над поверхностью земли:

```
    //текущая высота:
    curY:= h - g*t*t/2 + vv*t;
    sv:= sv + Abs((vv - g*t) * dt);
```

Условие падения шара на землю станет более естественным:

```
  If (curY <= 0) Then begin//шар упал на землю
     playSound.Play();
```

```
      curY := 0;
      drawShar(n,curY,t);
      exit;
    End;//If
    drawShar(n,curY,t);
    t:= t+ dt;
    Sleep(10);
    n:= n +1;
  End;//While
End;
```

Так как мы изменили формулу для расчёта высоты шара, нам необходимо внести поправки и в процедуру его рисования:

```
procedure drawShar(n: integer; s,t: real);
begin
   . . .
   var kpix:= htpix/h;
   //текущая высота шара в пикселях на канве:
   var ypix:= yZemli- dShar - kpix* curY;
   //горизонтальная координата шара в пикселях:
   var xpix:= xShar + kpix* (vh* t);
   //рисуем шар в текущем положении:
   shar.Position:= new Point(Floor(xpix), Floor(ypix));
   shar.Visible:= true;
   . . .

   //пройденное расстояние по вертикали:
   var ds:= sv;
   ds:= Floor(ds*100)/100;
   lblRasst.RealNumber:=ds;
   //пройденное расстояние по горизонтали:
   var dsh:= vh * t;
   dsh:= Floor(dsh*100)/100;
   lblRasstH.RealNumber:=dsh;
End;
```

Запускаем программу и убеждаемся, что модель работает верно (Рис. 27.6).

Падение шара

4.8 < Время падения

67.7 < Пройденное расстояние по вертикали

35.9 < Пройденное расстояние по горизонтали

БРОСИТЬ!

Рис. 27.6. Падение шара, брошенного под углом к горизонту

Исходный код программы находится в папке **Fall3**.

1. Ускорение свободного падения на Луне равно 1,624 м/с^2, а на Юпитере – 24,8 м/с^2. Смоделируйте падение шара на Луне, на Юпитере и на других планетах Солнечной системы!

2. Измените код программы *Fall3* так, чтобы шар можно было бы бросать с земли, а не с башни. Например, при начальных условиях

```
'угол броска в градусах:
alpha= 75
'начальная скорость шара, метров в секунду:
v0= 35
```

получится вот такая замечательная парабола (Рис. 27.7).

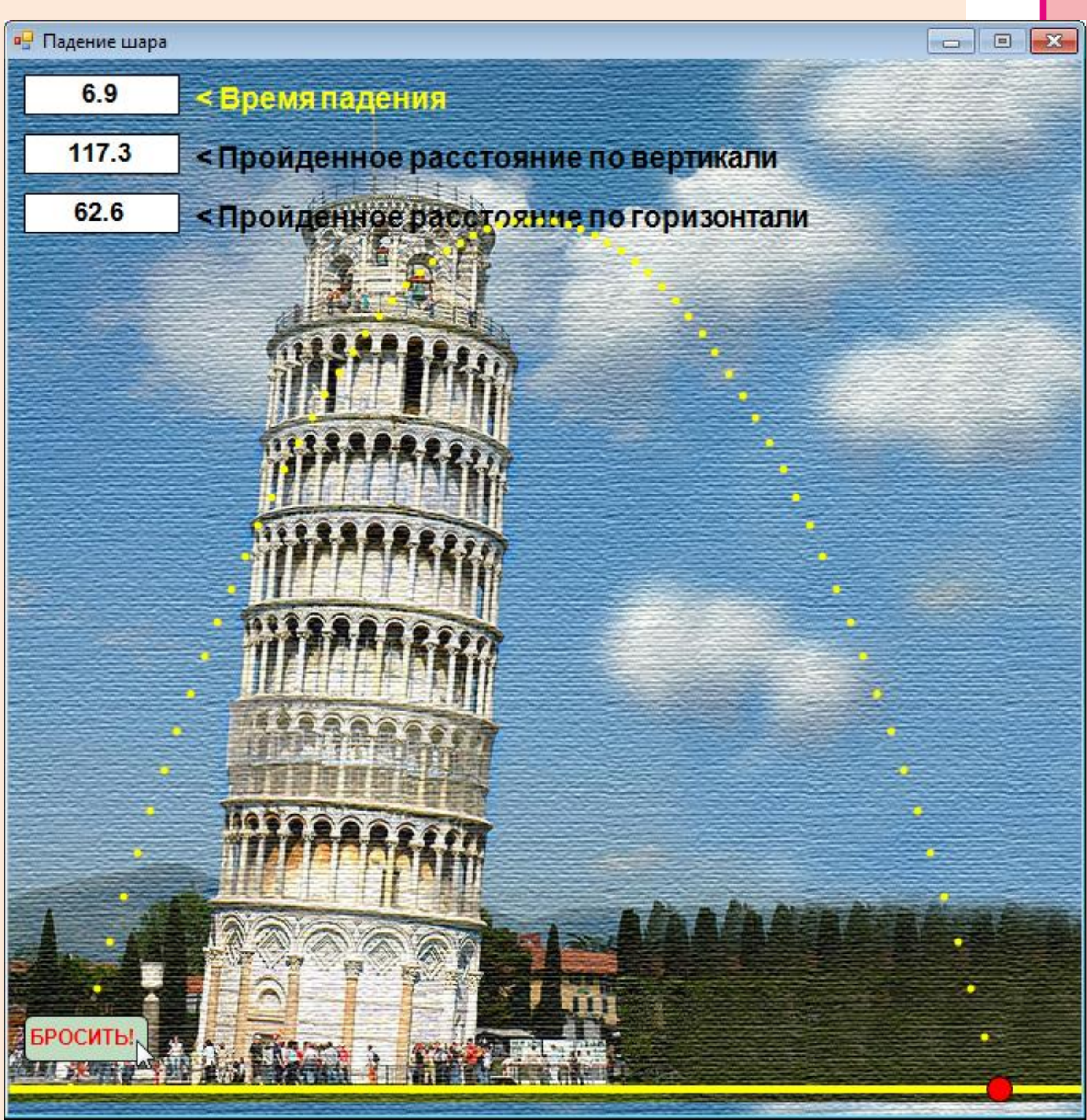
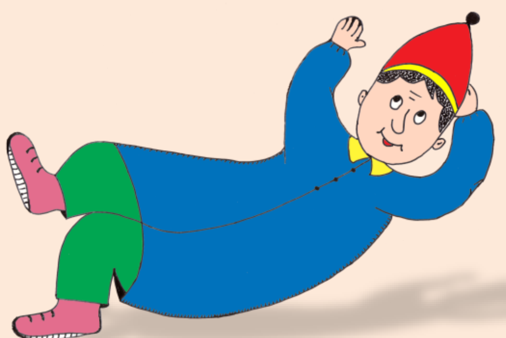


Рис. 27.7. Движение шара, брошенного с земли под углом к горизонту

Исходный код программы находится в папке **Fall3**.

3. Согласно легенде, однажды Ньютон лежал под деревом и о чём-то размышлял. Неожиданно на его голову упало яблоко, в результате чего он открыл закон всемирного тяготения. Определите, чему была равна скорость падения яблока, если оно висело в пяти метрах над землёй. Напишите программу, моделирующую падение яблока и других груш с дерева. Подумайте, какой закон открыл бы Ньютон, если бы на него упал арбуз!



4. Как вы помните, в кинокомедии *Бриллиантовая рука*, в той части, которая называется *Костяная нога*, Семён Семёныч Горбунков выпал из багажника автомобиля (Рис. 27.8). Всё бы ничего, но автомобиль был подвешен под вертолётом, который летел на высоте, будем считать, 75 метров. Определите, с какой скоростью Семён Семёныч врезался ногой в землю!

Рис. 27.8. Семён Семёныч вываливается из багажника

Урок 28. Тыблоки

Компьютерное тыблоко

Алексей Иванович Пантелеев в рассказе *Буква "ты"* учил одну маленькую девочку читать и писать. Иринушка, так звали девочку, быстро осваивала алфавит, пока они не дошли до последней буквы – *Я*, которую Иринушка упорно называла буквой *ТЫ*. Например, предложение *Якову дали яблоко* она читала так: *Тыкову дали тыблоко*. И так продолжалось до тех пор, пока Пантелеев не сказал Иринушке, что буква называется не *Я*, а *ТЫ*. Эта уловка помогла, и она стала читать все слова правильно, но мы поищем в русском языке именно *тыблоки*.

Мы возьмём за основу программу *Palindrome* и сохраним её в папке **Тыблоко** под тем же названием. Большинство переменных нам досталось по наследству, поэтому мы добавим только одну новую переменную *tybloko*:

```
//ПРОГРАММА ДЛЯ ПОИСКА ТЫБЛОК

uses CRT;

const
   MAX_WORDS = 30000;
   fileNameIn= 'OSH-W97frc.txt';
   fileNameOut='tybloki.txt';

var
   //массив-список слов:
   spisok: array [1..MAX_WORDS] of string;
   //число слов в списке:
   nWords: integer;
   f: textfile;
   //слово-тыблоко:
   tybloko: string;
```

Начало основной части программы также мало изменилось:

```
//===============================================
//            ОСНОВНАЯ ПРОГРАММА
//===============================================
begin
   SetWindowTitle('ПРОГРАММА ДЛЯ ПОИСКА СЛОВ-ТЫБЛОК');
   TextColor(LightRed);
   writeln('ИЩЕМ ТЫБЛОКИ');
   writeln;
   TextColor(Yellow);

   readFile;
   findTybloki;
   writeln();
   writeln('OK');

   writeln();
end.
```

В процедуре *readFile* мы просто заполняем массив *spisok* словами нужной нам длины. Я ограничился *пятибуквенными* словами, вы, естественно, можете расширить границы поиска тыблок:

```
//Считываем словарь в массив
procedure readFile();
begin
   nWords:=0;
   var s: string;
   assign(f, fileNameIn);
   reset(f);
   while not eof(f) do
   begin
      readln(f, s);
      if (s.length > 5) then
      break;
      writeln(s);
      inc(nWords);
      spisok[nWords]:= s;
   end;
   close(f);
end;
```

Поиск нужных нам слов я вынес в отдельную процедуру *findTybloki*:

```
//ИЩЕМ СЛОВА-ТЫБЛОКИ
procedure findTybloki();
begin
   assign(f, fileNameOut);
   rewrite(f);
   var s: string;

   //ищем слова, в которых есть буква Я:
   for var i:=1 to nWords do
   begin
      s:= spisok[i];
      //writeln(s);
      If string.Contains(s, 'Я') Then
      begin
         //нашли  слово
         //заменяем Я --> ТЫ:
         tybloko := s.Replace('Я', 'ТЫ');
         writeln(s + ' - ' + tybloko);
         writeln(f,s + ' - ' + tybloko);
      end;
   end;//For i

   close(f);
end;
```

Здесь в цикле *For* мы просматриваем все слова в списке. Если в них есть буква *Я* - а это очень просто узнать с помощью метода *Contains* класса *String*, - то мы заменяем каждую букву *Я* местоимением *ТЫ*. Готовое тыблоко теперь находится в строке *tybloko*.

Не забудьте скопировать в папку с программой фракционный словарь, иначе программа завершится с ошибкой.

Исходное слово и слово-тыблоко выводим в *консольное окно*, а также записываем в *файл* - для будущих чудачеств (Рис. 28.1).

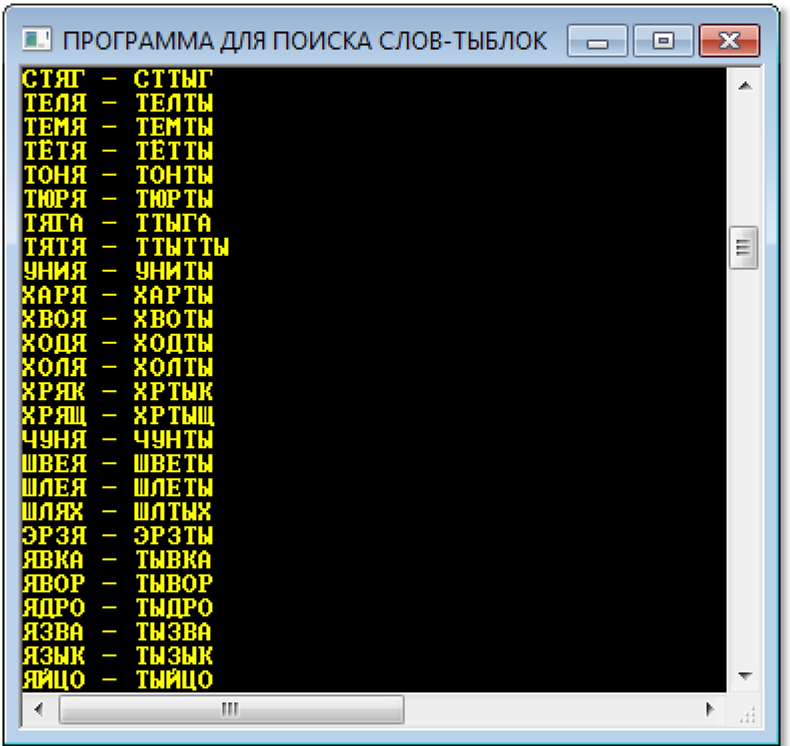


Рис. 28.1. Вот они, тыблочки наливные!

Иногда в тыблоки получаются обычными словами:

ЯК – ТЫК
ЯЛ – ТЫЛ
СОЯ – СОТЫ
БАНЯ – БАНТЫ
БУРЯ – БУРТЫ
СВАЯ – СВАТЫ
УНИЯ – УНИТЫ

Но попадаются и очень смешные тыблоки:

МЯЧ – МТЫЧ
ТУЯ – ТУТЫ
ЯЗЬ – ТЫЗЬ
БЯЗЬ – БТЫЗЬ
БЯКА – БТЫКА
ДОЯР – ДОТЫР
ДУЛЯ – ДУЛТЫ
ДЫНЯ – ДЫНТЫ
ЗАРЯ – ЗАРТЫ
ЗАЯЦ – ЗАТЫЦ
ЗМЕЯ – ЗМЕТЫ
ЗЮЗЯ – ЗЮЗТЫ

МАЯК – МАТЫК
ПЕНЯ – ПЕНТЫ
ПЛЯЖ – ПЛТЫЖ
ЯВОР – ТЫВОР
ЯДРО – ТЫДРО
ЯЗЫК – ТЫЗЫК
ЯЙЦО – ТЫЙЦО
ЯЛИК – ТЫЛИК
ЯНКИ – ТЫНКИ
ЯСЛИ – ТЫСЛИ
ЯХТА – ТЫХТА
ЯШМА – ТЫШМА

КЛЯП – КЛТЫП
КРЯЖ – КРТЫЖ

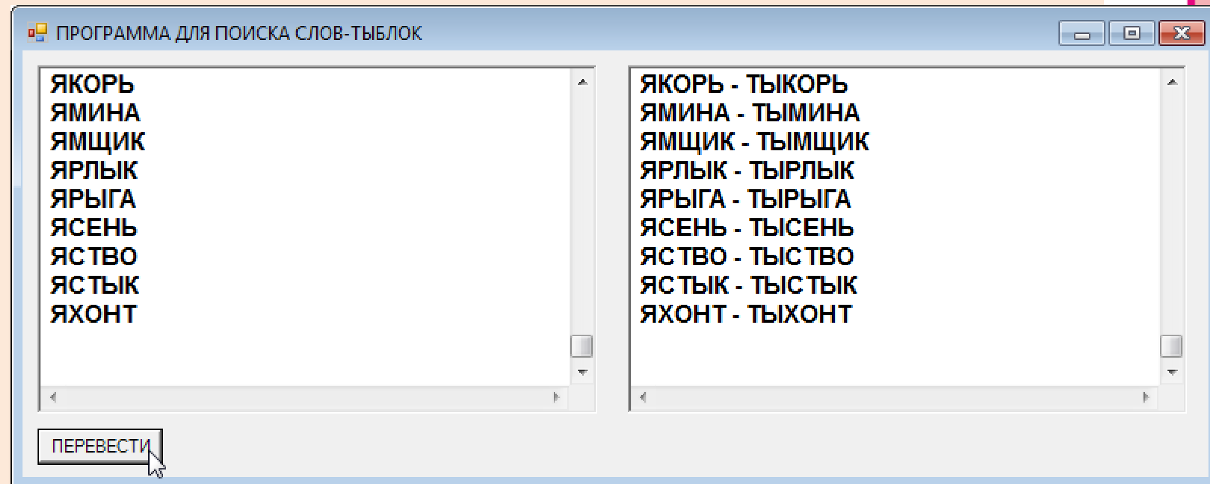
ЯЩЕР – ТЫЩЕР
ЯЩИК – ТЫЩИК

Исходный код программы находится в папке **Тыблоко**.

1. Можно заменять не только букву *Я* местоимением *ТЫ*, но и другие буквы какими-либо буквосочетаниями. Измените программу *Тыблоко* так, чтобы она искала такие слова.

2. Можно заменять не одну букву, а целое буквосочетание одной буквой или другим буквосочетанием.

3. Если взять не отдельные слова, а *текст*, в котором много слов с буквой *Я*, и заменить их *тыблоками*, то получится очень смешной юмористический рассказ на *тыблочном языке* (Рис. 28.2). В данном случае следует отказаться от *текстового окна* и воспользоваться *графическим*, так как мы можем поместить на него элемент управления *текстовое поле*, в котором легко набирать текст (или просто копировать его из других источников). Если же к нему добавить ещё одно *текстовое поле*, то в нём можно напечатать параллельный перевод текста на *тыблочный язык*.



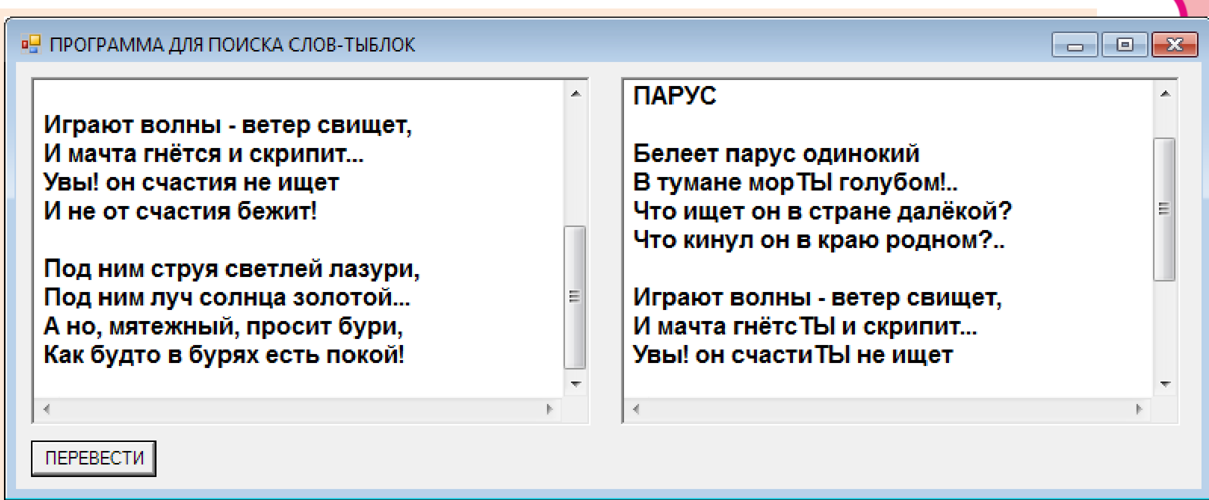


Рис. 28.2. Так может выглядеть интерфейс программы

Исходный код программы находится в папке **Тыблочный текст**.

Урок 29. Занимательная логопедия

-Февочка, скажи «ыба»
- Селёдка!

Логопедический диалог из комедии
По семейным обстоятельствам

Некоторые люди неверно произносят отдельные буквы (точнее, звуки, обозначаемые этими буквами), а один *логопеф* не выговаривал даже половину букв русского алфавита, что ничуть не мешало ему учить других людей говорить правильно. Иногда это приводило к непониманию, например, названия улиц *Кировская* и *Киевская* в его исполнении звучали совершенно одинаково. В других случаях смысл его речей был понятен, но смешон. А «смешон» - это как раз то, что нам нужно!

Логопеф везде заменял букву **Д** буквой **Ф** (конечно, он произносил звуки, а не буквы, но у нас будут как раз буквы): вместо *девочка* он говорил *февочка,* вместо *будет* – *буфет*, ну и так *фалее*.

Если вы сделали (а сейчас мы это и проверим!) самостоятельное задание из урока *Тыблоки*, то переделать тыблочную программу в логопефическую вам будет проще простого!

У кого фефект речи?

```
//ПРОГРАММА ДЛЯ ПЕРЕВОДА ТЕКСТА НА
//ЛОГОПЕФИЧЕСКИЙ ЯЗЫК

//Приложение Windows Forms
#apptype windows
#reference 'System.Windows.Forms.dll'
#reference 'System.Drawing.dll'

uses
  System,
  System.Windows.Forms,
  System.Drawing;

var
   //буква:
   chr: char;
```

```
   //логопефическое слово:
   logopef: string;
   //строка с текстом:
   str: string;
   //длина текста:
   len: integer;

   frmMain: Form;
   btnLogopef: Button;
   txtSource, txtLogopef: TextBox;
   //размеры окна:
   width, height: integer;
```

Для ввода текста и его перевода на логопефический язык нам потребуются элементы управления *текстовые поля*. Эта необходимость заставляет нас сделать нелёгкий выбор и написать приложение *Windows Forms*.

Начинаем с *главного окна* приложения и сразу же настраиваем его под свои нужды:

```
//==========================================
//          ОСНОВНАЯ ПРОГРАММА
//==========================================
begin
   frmMain := new Form;
   frmMain.Text := 'Логопефический текст';
   frmMain.Width:= 800;
   frmMain.Height:= 320;
   frmMain.StartPosition:= FormStartPosition.CenterScreen;
   width:= frmMain.Width-20;
   height:= frmMain.Height-40;
```

Добавляем к окну - по вкусу и нашим возросшим потребностям - *элементы управления* – два *текстовых поля* и одну *кнопку*:

```
  txtLogopef:= new TextBox();
   txtLogopef.Multiline:= true;
   txtLogopef.Height:= height-50;
   txtLogopef.Width:= width div 2 - 20;
   txtLogopef.Left:= width div 2 + 10;
   txtLogopef.Top:= 10;
```

```
   txtLogopef.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   txtLogopef.ScrollBars := Sys-
tem.Windows.Forms.ScrollBars.Both;
   txtLogopef.WordWrap:= false;
   frmMain.Controls.Add(txtLogopef);

   //кнопка:
   btnLogopef := new Button;
   btnLogopef.Text := 'ПЕРЕВЕСТИ';
   btnLogopef.AutoSize := True;
   btnLogopef.Left := 10;
   btnLogopef.Top := height-30;
   frmMain.Controls.Add(btnLogopef);
   btnLogopef.Click += btnLogopef_Click;

   Application.Run(frmMain);
end.
```

В *левое текстовое поле* нужно скопировать текст (или набрать его вручную - при надлежащем трудолюбии и упорстве), затем нажать кнопку *ПЕРЕВЕСТИ*, которая выполнит метод *btnLogopef_Click* и напечатает перевод сообщения в *правом текстовом поле*.

```
procedure btnLogopef_Click(sender: Object; e: EventArgs);
begin
   str:= txtSource.Text;
   len:= str.Length;

   //заменяем букву Д буквой Ф:
   logopef:='';
   For var i:= 1 to len do begin
      //очередная буква:
      chr:= str[i];
      var c := char.ToUpper(chr);
      If (c = 'Д') Then
         logopef += 'Ф'
      Else
         logopef += chr;
   End;//For
   txtLogopef.Text:= logopef;
End;
```

Важно: буква *Д* в тексте может быть и строчной, и ПРОПИСНОЙ. Для нас это одна и та же буква, а для компьютера – разные, поэтому нужно либо проверять обе буквы

```
If (chr = 'Д') Or (chr = 'д') Then  …,
```

либо переводить очередной символ в верхний регистр, тогда все буквы станут ПРОПИСНЫМИ, что облегчит проверку. Однако, если вы хотите сохранить в тексте все буквы в их исходном состоянии, то позаботьтесь об их неприкосновенности.

Наш переводчик готов к работе. Для пущего смеху лучше подобрать текст, в котором много букв *Д*. Например, в Интернете можно найти «однобуквицу» про *деда Данилыча* (автор *OJIesYA_EviL_MonKey*), которая вполне годится для наших экспериментов (Рис. 29.1).

Тексты, все слова которых начинаются на одну и ту же букву, называются *тавтограммами*. Самая известная тавтограмма: ***Ч**етыре **ч**ёрненьких **ч**умазеньких **ч**ертёнка **ч**ертили **ч**ёрными **ч**ернилами **ч**ертёж - **ч**резвычайно **ч**исто.*

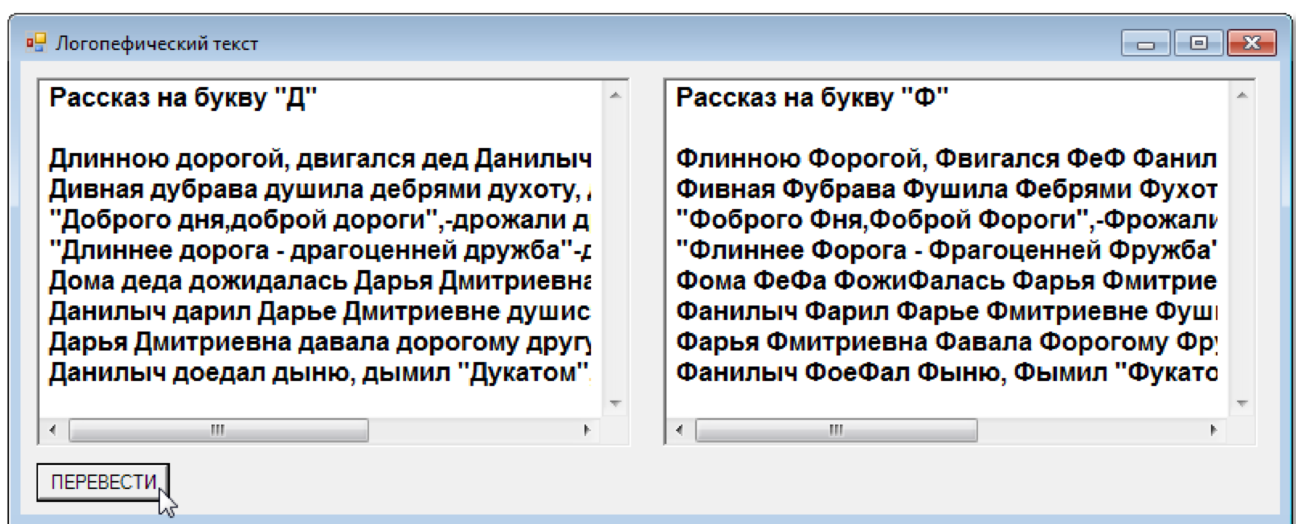


Рис. 29.1. Фолгая форога в фюнах!

Исходный код программы находится в папке **Логопеф**.

1. Подберите сами тексты с буквой *Д* и переведите их с помощью нашего *Логопефа*.

2. Попробуйте сами придумать подобный текст – с учётом того, что после перевода он должен стать очень смешным.

3. Добавьте к окну программы ещё одну кнопку – *СТЕРЕТЬ*, чтобы программой можно было бы пользоваться неоднократно.

4. Как вы помните, киношный логопеф не выговаривал букву *Р*, поэтому её тоже можно заменить буквой *Г* или *Й*. Или вообще выбросить из текста.

5. В некоторых русских говорах вместо буквы *Ц* произносят *Ч*: *цапля – чапля* (откуда и возникла русская фамилия Чаплин, которая не имеет к Чарли Чаплину никакого отношения, поскольку это наш Цаплин).

6. Писатель-юморист Семён Альтов знаменит не только своими смешными рассказами, но и особенностями их чтения: он глухие звуки произносит как звонкие, что очень смешно. Например, *канарейка* у него слала *ганарейгой*, *парикмахер – баригмакер*. Заставьте программу читать по-альтовски.

7. Как известно, москвичи (и мы вслед за ними) акают, то есть все безударные *О* произносят как *А*, то есть Москва и Россия по-московски - *МАсква* и *РАссия*. На слух мы уже привыкли к этому коверканью русской речи, но вот в напечатанном тексте это выглядит – ну, сами узнаете как, когда напишете программу для перевода русского текста на московский язык.

8. Помните песенку зайца из мультфильма *Ну, погоди*: «А ну-ка, давай-ка, плясать выходи!»? Вы легко можете ко всем (или только к некоторым) словам любого текста добавить частицу ка. Получится «заячий» текст.

9. Можно перевести текст и на «волчий» язык, если после каждого слова добавлять ну: *Ну, Заяц, ну, погоди, ну!*

Урок 30. Занимательная транслитерация

Мы уже встречались с транслитерацией на уроке *Признаки делимости*, где **транслитерация**, то есть запись русских слов латинскими буквами, была предложена как способ именования переменных программы. Давайте напишем приложение, которое будет автоматически переводить русские слова на «латынь».

Наш новый проект очень похож на программу *Тыблоко*, только теперь нам придётся заменять не только букву *Я*, но и все остальные. Как обычно, загружаем проект-шаблон *Тыблоко* и сохраняем его в папке **Транслитерация** под новым названием.

Дальше - всё просто. Заменяем одну *переменную* и – главное! - формируем *массив* для замены всех русских букв одной или несколькими латинскими:

```
//ПРОГРАММА ДЛЯ ТРАНСЛИТЕРАЦИИ РУССКИХ СЛОВ

uses CRT;

const
   MAX_WORDS = 30000;
   fileNameIn= 'OSH-W97frc.txt';
   fileNameOut='translit.txt';

   RUSALPH ='АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ';
   TRANSLIT: array[0..32] of string = ('A', 'B', 'V', 'G', 'D', 'E',
                                        'YO', 'ZH', 'Z', 'I', 'Y', 'K',
                                        'L',  'M', 'N', 'O', 'P', 'R',
                                        'S', 'T', 'U', 'F', 'KH', 'TS',
                                        'CH','SH','SHCH', '''', 'Y', 'J',
                                        'E', 'YU','YA'
                                       );
var
   //массив-список слов:
   spisok: array [1..MAX_WORDS] of string;
   //число слов в списке:
   nWords: integer;
   f: textfile;
```

А дальше – ещё проще:

```
//ПЕРЕВОДИМ СЛОВА
procedure translate();
begin
   assign(f, fileNameOut);
   rewrite(f);
   var s: string;

   //просматриваем весь список:
   for var i:=1 to nWords do
   begin
      s:= spisok[i];
      //writeln(s);
      var str :='';
      var len:= s.Length;
      For var j:= 1 to len do begin
         //очередная буква:
         var ch:= s[j];
         var id:= RUSALPH.IndexOf(ch);
         str += TRANSLIT[id];
      End;//For

      writeln(s + ' - ' + str);
      writeln(f,s + ' - ' + str);
   end;//For i

   close(f);
end;

//==========================================
//          ОСНОВНАЯ ПРОГРАММА
//==========================================
begin
   SetWindowTitle('ПРОГРАММА-ПЕРЕВОДЧИК');
   TextColor(LightRed);
   writeln('ПЕРЕВОДИМ СЛОВА');
   writeln;
   TextColor(Yellow);

   readFile;
   translate;
   writeln();
   writeln('OK');
```

```
   writeln();
end.
```

Последовательно выбираем слова из массива *spisok,* каждую букву очередного слова находим в строке *RUSALPH* и по её индексу заменяем «транслитерационными» символами. Перекодированное слово выводим в *консольное окно* и в *файл* (Рис. 30.1).

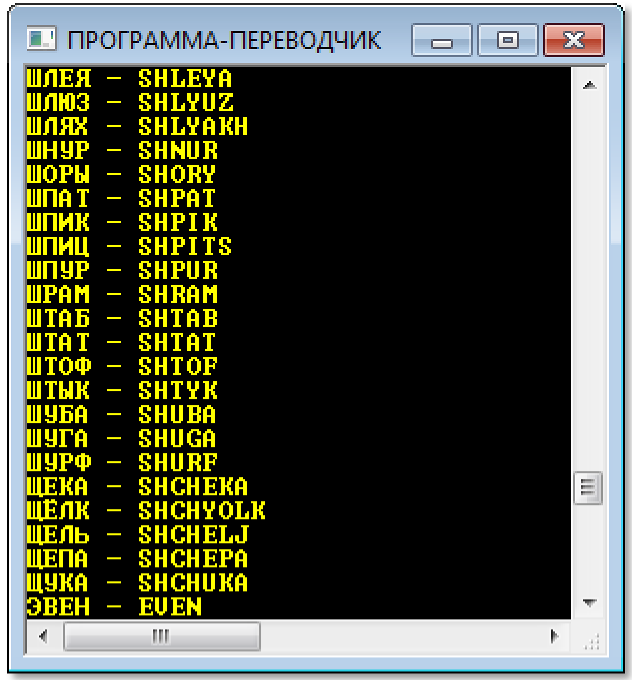


Рис. 30.1. Русско-латинский словарь

Если внимательно посмотреть на массив *translit*, то можно заметить, что некоторые буквы однозначно заменяются *транссимволами*. Например, *А, Б, В*. Другие, например, буква *Ё* требует *два* символа – *Y* и *O*, которые уже обозначают другие буквы – *Й* и *О*. Таким образом, обратный перевод (декодирование) нельзя провести однозначно.

Слово *ЙОГУРТ* при транслитерации превратится в *YOGURT*, а при переводе в обратную сторону мы получим два слова: правильное – *ЙОГУРТ* и неправильное – *ЁГУРТ*. В простых случаях достаточно просмотреть словарь и убедиться, что в нём имеется только первый вариант, который и является верным. Конечно, такая однозначность не всегда достижима.

Исходный код программы находится в папке **Транслитерация**.

Переделайте программу *Транслитерация* под *графическое окно* - так, как это сделано в программе *Занимательная логопедия*, и займитесь переводами известных произведений на «транслитерационный» язык.

Урок 31. Занимательная латиница

Moya tvoya ne ponimaet!

Из разговора в чате

Занимаясь транслитерацией, вы, конечно, не могли не обратить внимания на то, что некоторые кириллические буквы совпадают по написанию с латинскими, хотя и не обязательно обозначают один и тот же звук.

Предположим, что у нас имеются только латинские буквы. Сколько русских слов мы сможем напечатать?

Латиницей называют буквы латинского алфавита и слова, записанные такими буквами. *Кириллицей* – слова, записанные русскими буквами.

Для определённости будем считать, что только русские буквы **АВСЕНКМОРТХ** имеют аналоги в латинице.

За основу нового проекта мы возьмём программу *Тыблоки*, в которую добавим константу *LATIN*:

```
//ПРОГРАММА ДЛЯ ПОИСКА ЛАТИНСКО-РУССКИХ СЛОВ

uses CRT;

const
   MAX_WORDS = 30000;
   fileNameIn= 'OSH-W97frc.txt';
   fileNameOut='latin.txt';

   LATIN='АВСЕНКМОРТХ';

var
   . . .
```

В этой строке находятся те русские буквы, которые мы решили считать «латинскими».

Часть новой программы, которая совпадает с исходной, мы пропускаем и сразу переходим к поиску нужных нам слов:

```
//ПЕРЕВОДИМ СЛОВА
procedure translate();
begin
   assign(f, fileNameOut);
   rewrite(f);
   var s: string;

   //просматриваем весь список:
   for var i:=1 to nWords do
   begin
      s:= spisok[i];
      var len:= s.Length;
      var flg:= true;
      For var j:= 1 to len do begin
         //очередная буква:
         var ch:= s[j];
         if not string.Contains(LATIN, ch) then
         begin
            flg:= false;
            break;
         end;
      End;//For
      if flg then begin
         writeln(s);
         writeln(f,s);
      end;
   end;//For i

   close(f);
end;
```

Нам достаточно проверить, входит ли каждая буква очередного слова в строку *LATIN*. Если хотя бы одна буква «чисто русская», то мы такое слово пропускаем и переходим к следующему. В противном случае выводим найденное слово в *консольное окно* и записываем в *файл*.

Все проверки мы проводим в цикле *For* с помощью метода *Contains*:

```
string.Contains(LATIN, ch)
```

Вот конец списка, который составила наша программа (Рис. 31.1).

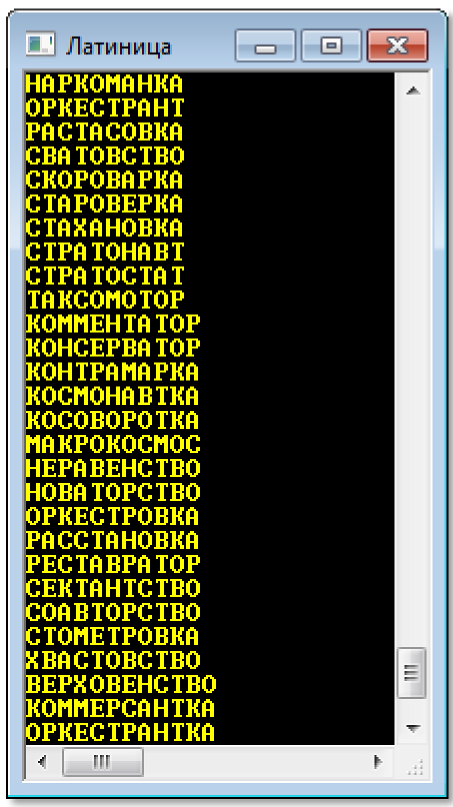


Рис. 31.1. Латинско-русский словарик составлен!

Как вы видите, самые длинные русские слова, записанные латиницей, - это ВЕРХОВЕНСТВО, КОММЕРСАНТКА и ОРКЕСТРАНТКА.

Исходный код программы находится в папке **Латиница**.

Супернаборщик

Наверное, вам известна школьная игра *Наборщик*. В ней требуется из букв какого-либо одного заданного *длинног*о слова составлять другие, более *короткие* слова, с условием, что они будут состоять только из тех букв, которые имеются в заданном слове. При этом, если в длинном слове, например, две буквы *А*, то

и в каждом составленном слове их должно быть *не больше* двух. Побеждает тот игрок, который составит более длинный список слов. Очень хорошая игра, в которую интересно играть, особенно на уроках математики.

Мы сделаем себе небольшое послабление в правилах игры – не будем требовать, чтобы число одинаковых букв в искомых словах не превышало числа этих букв в заданном слове. Иначе говоря, любую букву этого слова можно использовать сколько угодно раз. Все остальные правила мы трогать не будем.

Проект мы назовем, конечно, **Супернаборщик**:

```
//СУПЕРНАБОРЩИК

. . .

   //длинное слово:
   stroka: string;
```

Нам потребуется всего одна новая строковая переменная *stroka*, в которой мы и будем хранить заданное длинное слово.

Как и во всех других словесных программах, сначала нужно загрузить словарь. Опыт подсказывает нам, что на этот процесс потребуется некоторое время, поэтому сообщим игрокам, что им придётся секундочку подождать:

```
//========================================
//          ОСНОВНАЯ ПРОГРАММА
//========================================
begin
   SetWindowTitle('Супернаборщик');
   TextColor(LightRed);
   writeln('ЗАГРУЖАЮ СЛОВАРЬ');
   writeln;
   readFile;
```

Затем приглашаем их задать слово, из букв которого мы будем составлять более короткие слова:

```
 while(true) do
   begin
      TextColor(LightRed);
      CRT.TextBackground(Yellow);
      writeln('Введите исходное слово и нажмите клавишу ВВОД');
      writeln;
      CRT.TextBackground(Black);
      TextColor(Yellow);
      stroka:= ReadString().ToUpper();
```

Заданное слово мы сразу же переводим в *верхний* регистр, потому что все слова в наших словарях записаны БОЛЬШИМИ буквами, и сохраняем в переменной *stroka*.

Теперь нам осталось выбрать из списка такие слова, которые целиком состоят из букв длинного слова:

```
      super;

      TextColor(Yellow);
      writeln();
      writeln('OK');
      writeln();
   end;
end.
```

Для этого мы отправляемся в процедуру *super*, которая выбирает из списка только те слова, которые состоят из букв длинного слова. Нам на помощь опять приходит метод *Contains*, а всё остальное – дело техники:

```
//ИЩЕМ СЛОВА
procedure super();
begin
   var s: string;
   TextColor(LightGreen);
   //просматриваем весь список:
   for var i:=1 to nWords do
   begin
      s:= spisok[i];
      //writeln(s);
      var len:= s.Length;
```

```
      var flg:= true;
      For var j:= 1 to len do begin
         //очередная буква:
         var ch:= s[j];
         if not string.Contains(stroka, ch) then
         begin
            flg:= false;
            break;
         end;
      End;//For
      if flg then begin
         writeln(s);
      end;
   end;//For i
end;
```

Как видите, эти действия практически ничем не отличаются от тех, что мы рассмотрели в проекте *Латиница*, но теперь мы можем загадывать любые слова сколько угодно раз – пока не наиграемся!

Поиск слов даже в длинном словаре происходит очень быстро, и в *консольном окне* мы тут же получаем все подсказки (Рис. 31.2). С таким помощником вы легко станете чемпионом школы по игре в *Супернаборщика*!

1. Если вы планируете использовать эту программу «для дома, для семьи», то позаботьтесь о том, чтобы придать ей более привлекательный вид, заменив текстовый интерфейс графическим.

2. Напишите программу для игры в классического *Наборщика*! За основу вы можете взять нашу программу, но процедуру поиска слов придётся изменить, потому что буквы заданного слова необходимо считать.

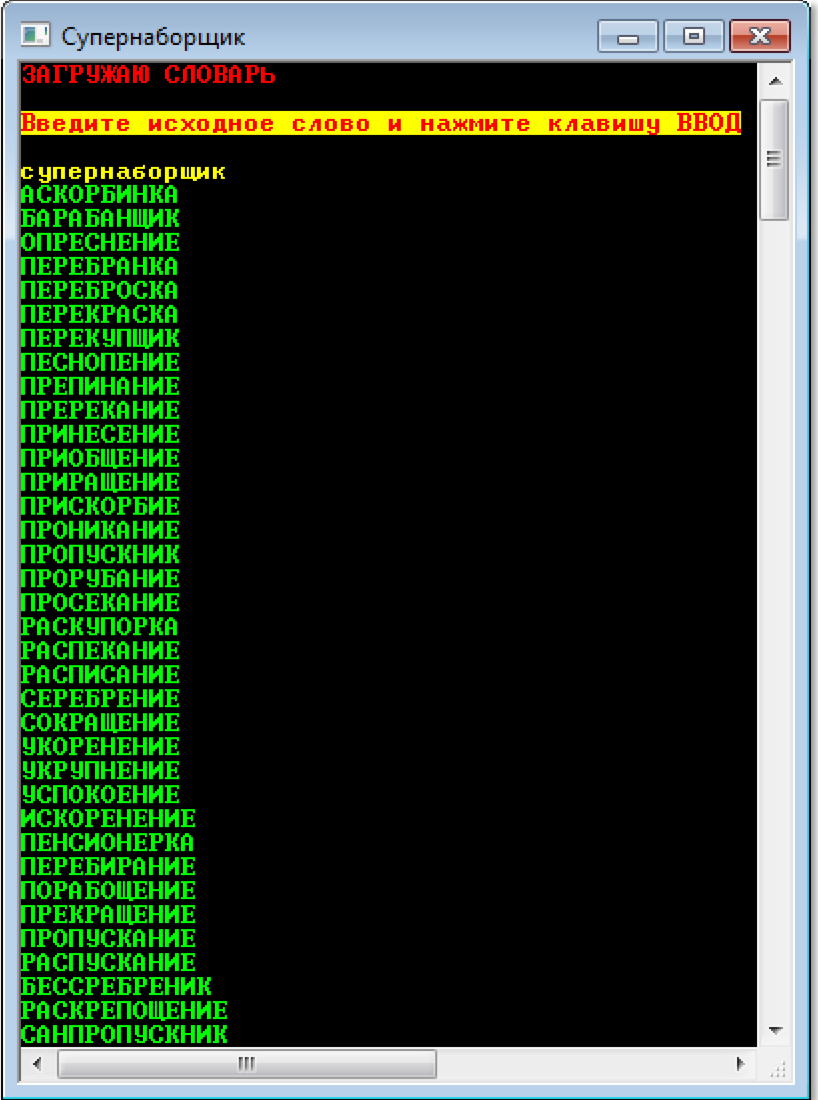


Рис. 31.2. А вы нашли бы такие слова?

Исходный код программы находится в папке **Супернаборщик**.

Урок 32. Занимательная криптография

- Бамбарбия! Киргуду!

Загадочные слова из комедии
Кавказская пленница

Криптография в переводе с греческого означает *тайнопись*. Она возникла одновременно с письменностью - для того, чтобы утаить содержание письма или другого документа. Действительно, обычный текст может прочитать любой грамотный человек, а ведь иногда просто необходимо скрыть важную информацию от посторонних лиц.

В повести Фриды Абрамовны Вигдоровой *Дорога в жизнь*, в 21-ой главе *У вас ничего не выйдет* рассказывается о криптографической проделке воспитанника детского дома Андрея Репина, который вёл в тетради протокол собрания. И вот что у него получилось:

> *Богдащоричи: беврый одвят тефувид бо гдочорой, рдовой бо трову, дведий бо чегдщике…*

Он, конечно, хотел поразить воспитателей своим шифрованным письмом, но один из них, Алексей Саввич, быстро смекнул:

> *– Да, тут есть логика. Постойте… Беврый одвят… беврый одвят… да это же первый отряд! Так… тефувид – дежурит. Понимаете, он оставил гласные, а остальной алфавит разделил пополам и поменял согласные местами: вместо п – б, вместо в – р и наоборот…*

Андрей Репин был посрамлён, но не угомонился, о чём мы поговорим дальше, а пока давайте-ка поразмыслим над его *тайнописью*.

Этот способ шифрования сообщений был известен еще в Древней Руси и называется *простой литореей* (не путайте с лотереей!). В то время букв было больше, но и наши 33 буквы вполне годятся

для шифрования. Правда, букву *Ё* придётся исключить из списка, чтобы осталось чётное число букв.

Буква *Ё* появилась в русском алфавите в 1797 году, так что возраст у неё почтенный.

Простая литорея называется также *тарабарской грамотой*. А название литорея произошло от слова *литера – буква*.

Букве *Ё* вообще не очень-то везёт в русском языке. Её уже не раз пытались исключить из алфавита, но, к счастью, пока безуспешно.

В 2005 году в городе Ульяновске был открыт памятник этой славной букве (Рис. 32.1, сверху). А в другом городе – Йошкар-Оле - установили памятник *ёшкиному коту* (Рис. 32.1, снизу). Правда, отдавая дань родному городу, мастера переименовали его в *йошкиного кота*, но мы просто проигнорируем этот грамматический факт.

Рис. 32.1. Памятник букве *Ё* и *ёшкиному коту*

Теперь расположим буквы в две строки, по 16 букв в каждой так, чтобы они образовали вертикальные пары:

А Б В Г Д Е Ж З И Й К Л М Н О П
Р С Т У Ф Х Ц Ч Ш Щ Ъ Ы Ь Э Ю Я

Чтобы закодировать слово, достаточно заменить в нём каждую букву парной в этом шифре. Например, слово *КРИПТОГРАФИЯ* превратится в *ЪАШЯВЮУАРДШП*. Пожалуй, не сразу и догадаешься, что за слово написано, хотя, как вы видите, способ шифрования очень простой. Вот только вручную заменять одни буквы другими довольно утомительно, да и ошибиться можно не раз. Поэтому давайте напишем программу, которая в два счёта зашифрует любой текст.

Загрузите проект *Логопеф* и сохраните его в папке **Литорея**.

Начинаем священнодействовать, то есть кодировать шифр!

```
//ПРОСТАЯ ЛИТОРЕЯ

//Приложение Windows Forms
```

```
#apptype windows
#reference 'System.Windows.Forms.dll'
#reference 'System.Drawing.dll'

uses
  System,
  System.Windows.Forms,
  System.Drawing;

const
   LIT: array[1..2] of string = (' АБВГДЕЖЗИЙКЛМНОП',
                                 ' РСТУФХЦЧШЩЪЫЬЭЮЯ');
var
   //строка с текстом:
   str: string;
   //длина текста:
   len: integer;
   //позиция буквы в массиве LIT:
   pos: integer;
   //литорейный текст:
   litorea: string;

   frmMain: Form;
   btnLitorea, btnClear: Button;
   txtSource, txtLitorea: TextBox;
   //размеры окна:
   width, height: integer;

//===========================================
//          ОСНОВНАЯ ПРОГРАММА
//===========================================
begin
   frmMain := new Form;
   frmMain.Text := 'Простая литорея';
   frmMain.Width:= 800;
   frmMain.Height:= 320;
   frmMain.StartPosition:= FormStartPosition.CenterScreen;
   width:= frmMain.Width-20;
   height:= frmMain.Height-40;
```

Для работы с программой нам понадобятся две *кнопки* и два *текстовых поля*:

```
  //текстовые поля:
```

```
   txtSource:= new TextBox();
   txtSource.Multiline:= true;
   txtSource.Height:= height-50;
   txtSource.Width:= width div 2 - 20;
   txtSource.Left:= 10;
   txtSource.Top:= 10;
   txtSource.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   txtSource.ScrollBars := Sys-
tem.Windows.Forms.ScrollBars.Both;
   txtSource.WordWrap:= false;
   frmMain.Controls.Add(txtSource);

   txtLitorea:= new TextBox();
   txtLitorea.Multiline:= true;
   txtLitorea.Height:= height-50;
   txtLitorea.Width:= width div 2 - 20;
   txtLitorea.Left:= width div 2 + 10;
   txtLitorea.Top:= 10;
   txtLitorea.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   txtLitorea.ScrollBars := Sys-
tem.Windows.Forms.ScrollBars.Both;
   txtLitorea.WordWrap:= false;
   frmMain.Controls.Add(txtLitorea);

   //кнопки:
   btnLitorea := new Button;
   btnLitorea.Text := 'ПЕРЕВЕСТИ';
   btnLitorea.AutoSize := True;
   btnLitorea.Left := 10;
   btnLitorea.Top := height-30;
   frmMain.Controls.Add(btnLitorea);
   btnLitorea.Click += btnLitorea_Click;

   btnClear := new Button;
   btnClear.Text := 'СТЕРЕТЬ';
   btnClear.AutoSize := True;
   btnClear.Left := 100;
   btnClear.Top := height-30;
   frmMain.Controls.Add(btnClear);
   btnClear.Click += btnClear_Click;

   Application.Run(frmMain);
```

```
end.
```

Назначение каждого из них понятно и без пояснений.

Нажатие кнопок мы обрабатываем в процедурах *btnLitorea_Click* и *btnClear_Click*.

Если была нажата кнопка *СТЕРЕТЬ*, то мы очищаем оба *текстовых поля*. Для этого достаточно вызвать метод *ResetText*:

```
procedure btnClear_Click(sender: Object; e: EventArgs);
begin
   txtSource.ResetText();
   txtLitorea.ResetText();
end;
```

Если же была нажата кнопка *ПЕРЕВЕСТИ*, то мы считываем в переменную *str* весь текст из *ЭУ txtSource*, а затем последовательно просматриваем всю строку в цикле *For*.

С помощью метода *IndexOf* мы узнаём, имеется ли такой символ в массиве *LIT*, а если имеется, то в какой из строк – в верхней или в нижней. В зависимости от результата проверки мы заменяем букву её парой. Букву *Ё* и все остальные символы (например, пробелы, латинские буквы, цифры и знаки препинания) мы просто копируем в новую строку *litorea*.

Закончив цикл, мы печатаем эту строку в правом *текстовом поле*:

```
procedure btnLitorea_Click(sender: Object; e: EventArgs);
begin
   str:= txtSource.Text;
   len:= str.Length;

   //кодируем - заменяем буквы парными:
   litorea:='';
   For var i:= 1 to len do begin
     //очередная буква:
     var chr:= str[i];
     var c := char.ToUpper(chr);
```

```
     pos:= LIT[1].IndexOf(c);
     If (pos >-1) Then // верхняя буква
        //заменяем нижней:
        litorea:= litorea + LIT[2][pos+1]
      Else begin
        pos:= LIT[2].IndexOf(c);
        If (pos >-1) Then // нижняя буква
          //заменяем верхней:
          litorea:= litorea +LIT[1][pos+1]
        else //буква Ё или другой символ:
          litorea:= litorea + chr;
      End;
   End;//For

   txtLitorea.Text:= litorea;
End;
```

Перевод проходит «без шума и пыли», но все буквы становятся ПРОПИСНЫМИ (Рис. 32.2).

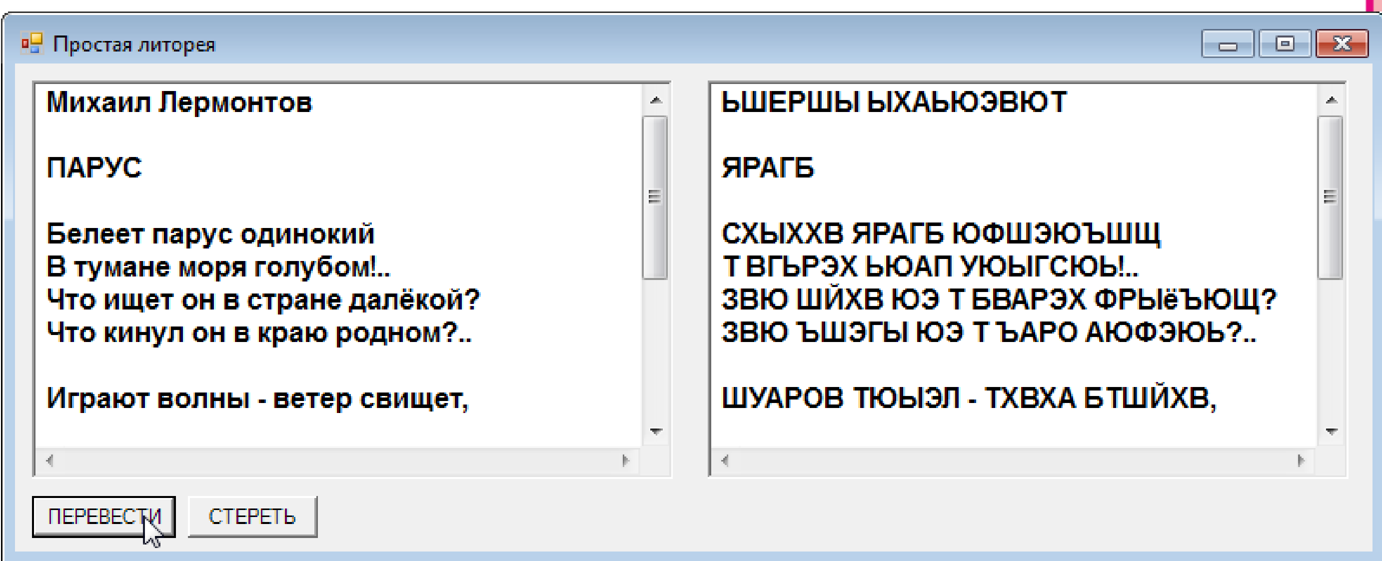


Рис. 32.2. Толковый перевод!

От этой порчи текста легко избавиться, если в массиве *LIT* завести проверочные строки и для маленьких букв.

Декодирование текста легко выполнить в той же программе. Достаточно текст из правого поля перенести в левое и нажать кнопку *ПЕРЕВЕСТИ* (Рис. 32.3).

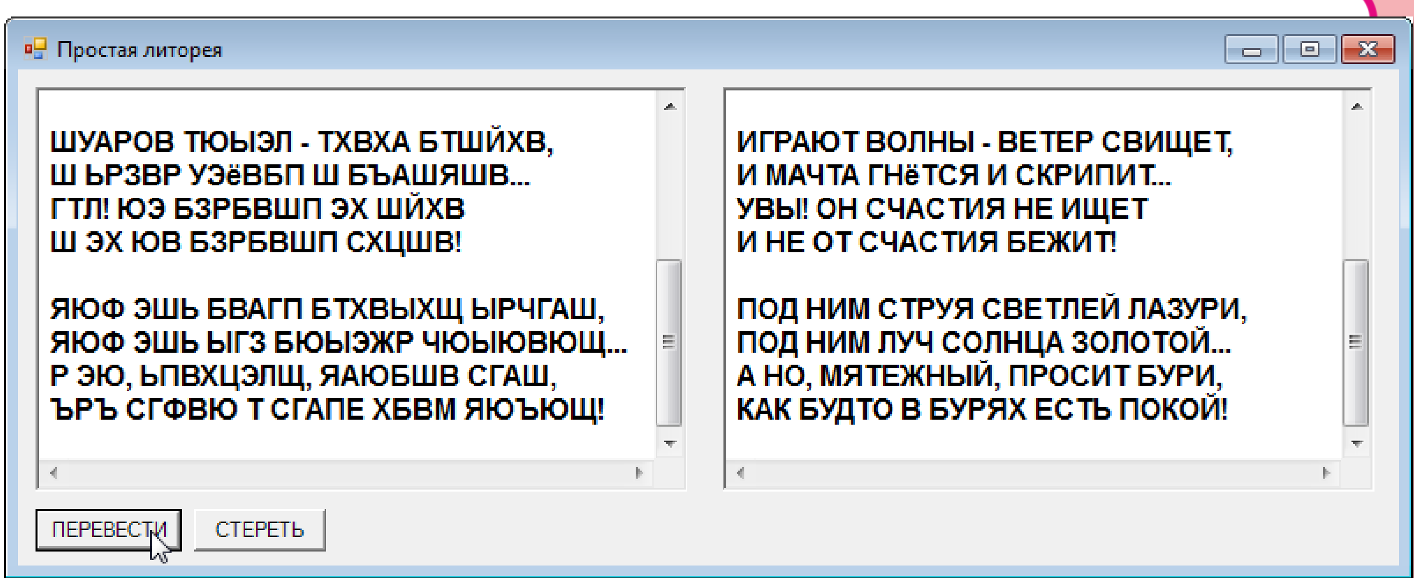


Рис. 32.3. Декодирование прошло успешно!

Исходный код программы находится в папке **Литорея**.

Литорея мудрая

В знаменитой повести А. Рыбакова *Кортик* мы также встречаемся с зашифрованной надписью, которая украшала ножны и сам кортик.

Вот что поведала нам Глава 53 *Ножны*:

> *Мальчики забежали за церковный придел, и Коровин вытащил из кармана ножны.*
> *Миша нетерпеливо выхватил их у него, повертел в руках, затем осторожно снял ободок и вывернул шарик.*
> *Ножны развернулись веером. Мальчики уставились на них, потом удивленно переглянулись...*
> *На внутренней стороне ножен столбиками были нанесены знаки: точки, чёрточки, кружки. Точно так же, как и на пластинке кортика.*

За разгадкой тайны кортика друзья обратились (Глава 56 *Литорея*) к директору школы Алексею Ивановичу, который и объяс-

нил им, что надпись на кортике и ножнах представляет собой *литорею мудрую*, и что надписи на них можно прочитать, только соединив вместе.

Так что же такое – литорея мудрая? – Давайте разбираться!

В Древней Руси 30 букв алфавита делили на три равные части, по 10 букв в каждой. В пределах первого десятка буквы последовательно обозначали **точками** (Рис. 32.4).

Рис. 32.4. Буквы-точки

Второго – **чёрточками** (Рис. 32.5), а третьего – **кружками** или крестиками (Рис. 32.6).

Как видите, знаки для каждой буквы записывались *столбиком*. А горизонтальная черта делила столбики пополам. Зная обозначение каждой буквы, мы можем составить любой текст, например, *Литорея мудрая* (Рис. 32.7).

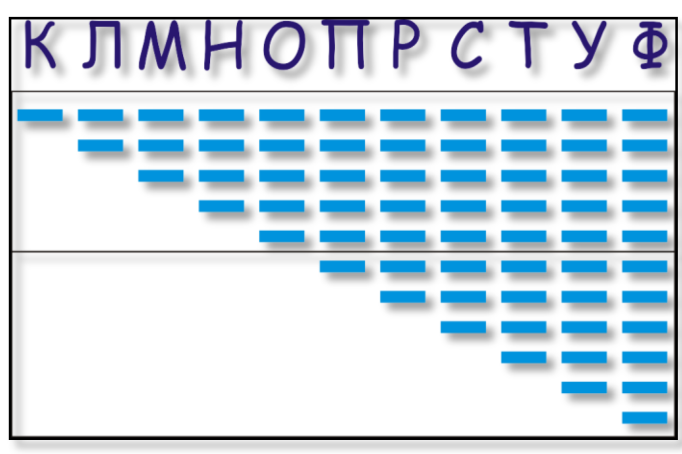


Рис. 32.5. Буквы-чёрточки

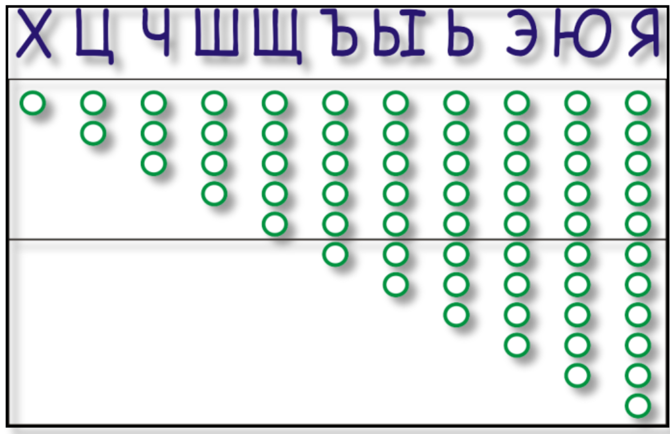


Рис. 32.6. Буквы-кружочки

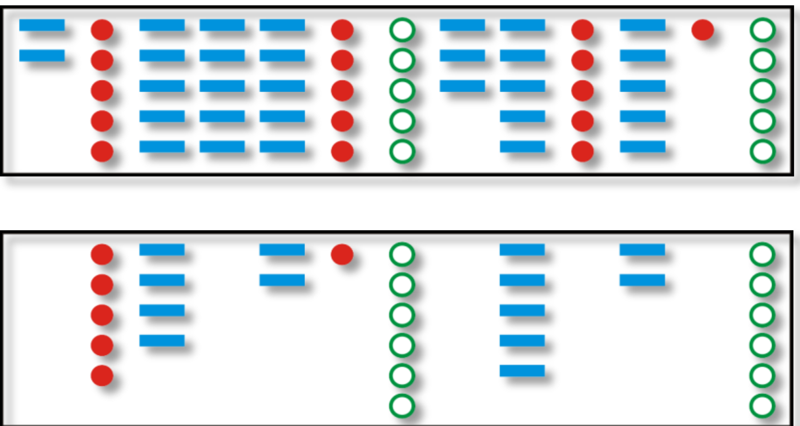


Рис. 32.7. Надпись литореей мудрой

Теперь закроем нижнюю часть шифровки и попробуем восстановить текст: **Л** Д-Й О-Ф О-Ф О-Ф Д-Й Щ-Я **М** О-Ф Д-Й О-Ф **А** Щ-Я. Удалось точно определить три буквы (они выделены **жирным** шрифтом), для остальных мы нашли только диапазон возможных значений. Да, задачка оказалась совсем непростой!

Поскольку нам нужны не 30, а все 33 буквы современного русского алфавита, то у нас в каждой группе будет не по десять, а по одиннадцать букв. По этой причине максимальная высота столбиков с символами равна 11. Число нечётное, поэтому пришлось провести горизонтальную линию так, чтобы в верхней части было до пяти символов, а в нижней – все остальные.

Осталось разрезать полоску бумаги по горизонтальной линии на две части. Теперь, чтобы прочитать текст, нужно соединить обе

части бумажки по линии разреза. Точно так же было зашифровано сообщение на кортике и ножнах. Если вы читали повесть А. Рыбакова, то помните, что на них было написано:

> *Сим гадом завести часы понеже проследует стрелка полудень башне самой повёрнутой быть.*

Несмотря на название, шифр не очень «мудрый»: некоторые буквы, на обозначение которых пошло от одного до четырех символов, остались незакодированными, что помогает расшифровать запись. Насколько помогает – определите сами.

Герои повести справились и с этой загадкой, а нам пора вернуться к тарабарской грамоте.

Тарабарская грамота

Как мы знаем, простая литорея называется также тарабарской грамотой, но есть и другие способы превратить русские тексты в тарабарщину. Этим достойным делом мы сейчас и займёмся.

Запишите исходный код программы *Литорея* в папку **Тарабарская грамота**. Давайте сразу же посмотрим, как работает программа (Рис. 32.8).

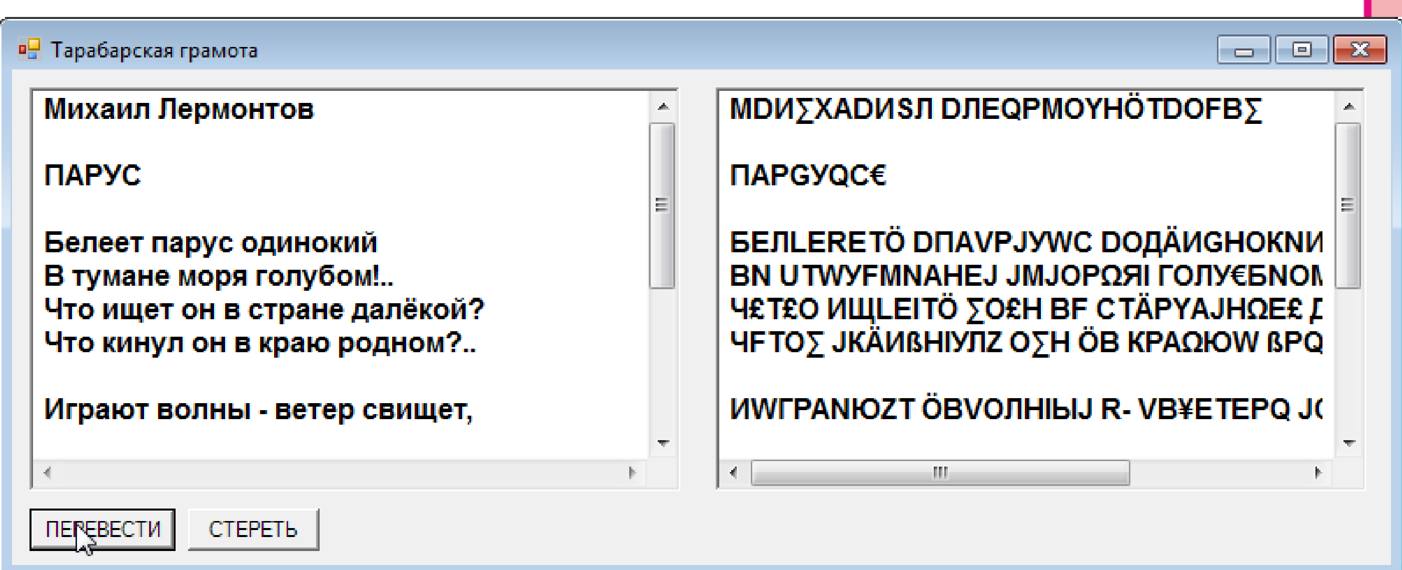


Рис. 32.8. И вправду полная тарабарщина!

Этот вид тайнописи наряду с обеими литореями применяли наши далекие предки, чтобы скрыть свои мысли и чаяния. На первый взгляд, текст справа написан не на русском языке, а на каком-то иноземном. Однако ни одна русская буква при этом гуманном эксперименте не пострадала! Их ровно столько же справа, сколько и слева, и трудно не заметить, что строчки в правом текстовом поле стали *длиннее*. Вот в этом-то весь секрет: нужно по вкусу добавить к русским буквам «тарабарские». Проще всего воспользоваться «европейскими» буквами и знаками, которых нет в русском языке.

Ну, вот, с шифром разобрались, а теперь за дело!

```
//ПРОГРАММА ТАРАБАРСКАЯ ГРАМОТА

//Приложение Windows Forms
#apptype windows
#reference 'System.Windows.Forms.dll'
#reference 'System.Drawing.dll'

uses
  System,
  System.Windows.Forms,
  System.Drawing;

const
   rus=' АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ';
   bukvy='QWRZUIUSDFGJLOA?YVN€??ΩΣ';

var
   //строка с текстом:
   str: string;
   //длина текста:
   len: integer;
   //тарабарский текст:
   tarabar: string;

   frmMain: Form;
   btnTarabar, btnClear: Button;
   txtSource, txtTarabar: TextBox;
   //размеры окна:
   width, height: integer;
```

```
//==========================================
//          ОСНОВНАЯ ПРОГРАММА
//==========================================
begin
   frmMain := new Form;
   frmMain.Text := 'Тарабарская грамота';
```

Нам потребуются также русские буквы и знак пробела. Зачем? – Мы будем вставлять инородные буквы только после русских, а для этого нам нужно находить среди всех символов текста, в котором могут быть знаки препинания, цифры и служебные символы, русские буквы, Пробел тоже пригодится, потому что он стоит перед первой буквой слова, иначе все слова будут начинаться с правильной буквы. А вот перед начальной буквой строки поставить другую букву сложнее. В этом случае стоит весь текст записать одной строкой.

Следующий фрагмент кода программы *Литорея* остался без изменений, но при желании вы можете заменить название правого поля более походящим *txtTarabar*.

В процедуре, обрабатывающей нажатие кнопок, мы перепишем только те строки, которые занимаются зашифровкой текста:

```
procedure btntarabar_Click(sender: Object; e: EventArgs);
begin
   . . .
   //кодируем -  добавляем тарабарские буквы:
   tarabar:='';
   For var i:= 1 to len do begin
     . . .
     tarabar:= tarabar + c;
     //если это буква или пробел -->
     If rus.Contains(c) Then begin
        var n:= PABCSystem.Random(101);
        //если случайное число больше заданного,
        //то добавляем к тексту тарабарскую букву:
         If (n > 33) Then begin
           n:= PABCSystem.Random(bukvy.Length)+1;
           chr:= bukvy[n];
```

```
            tarabar:= tarabar + chr;
         End;//If
      End;//If
   End;//For

   txtTarabar.Text:= tarabar;
End;
```

Если вы хотите, чтобы тарабарские буквы добавлялись чаще, то уменьшите число в выражении *n > 33*, а если реже – увеличьте его.

Исходный код программы находится в папке **Тарабарская грамота**.

1. А что же всё-таки написал в протоколе Андрей Репин? - Вы будете крепко озадачены, если раскодируете протокол в нашей программе. Вместо одной тарабарской грамоты вы получите другую:

СЮУФРЙЮАШЗШ: СХТАЛЩ ЮФТПВ ВХДГТШФ СЮ УФЮЗЮАЮЩ, АФЮТЮЩ СЮ ВАЮТГ, ФТХФШЩ СЮ ЗХУФЙШЪХ

Как же так: мы выяснили, что Андрей Репин для шифрования использовал простую литорею, а она не действует! А дело в том, что шифр Репина ещё проще, чем тот, который применяли мы. Он оставил без изменения все гласные буквы, а согласные записал в две строчки:

Б В Г Д Ж З К Л М Н
П Р С Т Ф Х Ц Ч Ш Щ

Легко заметить, что он исключил и некоторые другие буквы и знаки, но сути дела это не меняет.

Исправьте программу *Литорея*, чтобы она справилась, наконец, с дешифровкой протокола.

2. Способ Андрея Репина легко изменить так, чтобы разгадать шифровку было труднее. Например, можно записать вторую строку букв в обратном порядке:

Б В Г Д Ж З К Л М Н
П Р С Т Ф Х Ц Ч Ш Щ

3. Через некоторое время заведующий детским домом Семён Афанасьевич получил письмо вот такого содержания:

25 - 19,13 19, 2, 5, 13, 10, 13, 19 - 11, 8, 23, 9, 8 - 3, 11, 13, 18, 40 - 8, 18, 12, 2, 7, 2, 12, 40, 18, 25 -25 - 18, 1, 8, 10, 8 - 21, 14, 11, 21 - 21, 14, 11, 21, 12 - 17 - 5, 19, 8, 9, 17, 13 - 11, 10, 21, 9, 17, 13 – 25 - 11, 2, 7, 19, 8 - 4,50 - 21, 20, 13, 23 19, 8 - 5, 19, 13 - 4, 50, 23, 8 - 17, 19, 12, 13, 10, 13, 18, 19, 8 - 19, 2, 4, 23, 27, 11, 2, 12, 40 - 3, 2 – 7, 2, 5, 17 - 15, 10, 2, 7, 11, 2 – 7, 50 – 5, 19, 8, 9, 8, 9, 8 - 11, 8, 4, 17, 23, 17, 18, 40 – 19, 8 - 7, 18, 13 - 10, 2, 7, 19, 8 – 21 - 7, 2, 18 - 19, 17, 24, 13, 9, 8 - 19,13 7, 50, 14, 11, 13, 12 - 24, 13, 23, 8, 7, 13, 1 - 15, 10, 13, 6, 11, 13 - 7, 18, 13, 9, 8 - 16, 13, 19, 17, 12 - 18, 7, 8, 4, 8, 11, 21 - 18, 7, 8, 4, 8, 11, 2 - 11,23,25 - 19, 13, 9, 8 - 9, 23, 2, 7, 19, 8, 13 - 7, 50 - 22, 8, 12, 17, 12, 13 - 18, 11, 13, 23, 2, 12, 40 - 7,18,13 - 15, 8 - 11, 10, 21, 9, 8, 5, 21 - 19, 8 - 21 - 7, 2, 18 19, 17, 24, 13, 9, 8 - 19, 13 - 7, 50, 14, 11, 13, 12.

Он тут же хотел обратиться за помощью к Алексею Саввичу, разгадавшему первый шифр Репина (а вы, наверное, догадались, что письмо было именно от него), но потом решил, что и сам справится с новой головоломкой.

Заведующему очень помогла догадка: каждое число заменяет одну и ту же букву. Затем он обратил внимание на то, что некоторые числа встречаются *чаще* других, значит, им должны в русском языке соответствовать буквы, которые также встречаются в словах чаще, чем другие. Ну и наконец, иногда буквы

образуют характерные *сочетания*, по которым можно найти короткие слова, а потом …

А вот что было потом, прочитайте в повести *Дорога в жизнь*. Там вы заодно и узнаете, почему глава называется *У вас ничего не выйдет*. А ещё лучше – постарайтесь сначала самостоятельно прочитать письмо. Для решения задачи компьютер вам не понадобится, а вот *таблица частоты русских букв* вполне может пригодиться:

Пробел	0,2005		
О	0,0764	Ы	0,0143
Е	0,0732	Ь	0,0138
А	0,0629	З	0,0133
И	0,0577	Й	0,0125
Т	0,0549	Б	0,0114
Н	0,049	Ч	0,0094
Р	0,0459	Г	0,0083
С	0,0404	Ю	0,0081
В	0,0355	Ж	0,0079
П	0,033	Х	0,0048
К	0,0302	Щ	0,0042
Л	0,0299	Ф	0,0036
М	0,0275	Ш	0,0026
Д	0,0265	Э	0,0023
У	0,0222	Ц	0,0021
Я	0,0153	Ъ	0,0003

Как видите, в этой таблице буква *Ё* считается как *Е*, и вызвано это вовсе не пренебрежением к букве, а особенностями русского книгопечатания.

Вы можете отыскать и другие таблицы, в которых данные будут несколько отличаться. Это и естественно: подсчёты ведь можно проводить по-разному. Но в целом и эта таблица даёт правильное представление.

4. Разработайте проект *Ё-моё*. Из нашего словаря составьте список слов, в которые входит буква *Ё*. Задача несложная, но интересная: вы узнаете, сколько таких слов в русском языке.

5. Добавьте к программе *Тарабарская грамота* подпрограмму, *декодирующую* зашифрованный текст.

Урок 33. Занимательная биология

Поскольку биология не относится к числу точных наук, то совсем непросто подобрать интересные примеры использования компьютеров на уроках биологии, но три проекта, которые мы разработаем на этом уроке, и достаточно занимательны, и несомненно полезны, поскольку вы сможете доставить немало приятных минут своим близким, родственникам и друзьям, подвергнув их компьютерному тестированию. Не нужно относиться к его результатам излишне серьёзно, но и полностью отвергать их тоже не стоит.

Контрольное взвешивание, или Веское приложение

Хорошего человека должно быть много.

Коварное заблуждение

Сначала мы поможем себе и своим близким приблизиться к идеалу. Пока только в смысле веса.

Некоторые специалисты считают, что *идеальный вес* здорового человека можно определить по формулам:

P = (3 x A - 450 + B) x 0,25 + 45 - для мужчин,
P = (3 x A - 450 + B) x 0,225 + 40,4 - для женщин,

где под загадочными буквами *A* и *B* скрываются *рост* испытуемого (в сантиметрах) и *возраст* (в годах), соответственно.

Согласимся с ними и напишем чрезвычайно полезную для дома и семьи программу, которая позволит вам и всем окружающим оценить свои материальные (опять же только в смысле веса) достоинства.

Какой же может быть программа, основанная на этих незамысловатых формулах? - Ясно, что главную роль в ней должна играть процедура, вычисляющая идеальный вес по заданным параметрам – полу «подопытного индивидуума», его возрасту и росту. С написанием процедуры проблем не будет, ведь достаточно перевести на компьютерный язык обычные математические выражения. Чтобы процедура «знала» исходные величины, введём *переменные*, в которых будем хранить текущие значения *возраста* и *роста*:

```
//ПРОГРАММА ДЛЯ ВЫЧИСЛЕНИЯ ИДЕАЛЬНОГО ВЕСА

//Приложение Windows Forms
#apptype windows
#reference 'System.Windows.Forms.dll'
#reference 'System.Drawing.dll'

uses
  System,
  System.Windows.Forms,
  System.Drawing;


var
   ves: real;
   rost, vozrast: integer;

   frmMain: Form;
   btnM, btnW: Button;
   txtVozrast, txtRost, txtVes: TextBox;
   lblVozrast, lblRost: System.Windows.Forms.Label;
   //размеры окна:
   width, height: integer;
```

Размер *окна* и картинку для *фона* можно выбрать по своему вкусу и настроению:

```
//=========================================
//          ОСНОВНАЯ ПРОГРАММА
//=========================================
begin
   frmMain := new Form;
```

```
   frmMain.Text := 'Вес';
   frmMain.Width:= 600;
   frmMain.Height:= 390+40;
   frmMain.StartPosition:= FormStartPosition.CenterScreen;
   frmMain.FormBorderStyle := Sys-
tem.Windows.Forms.FormBorderStyle.FixedSingle;
   frmMain.BackgroundImage:= Image.FromFile('mw.jpg');
   width:= frmMain. Width-20;
   height:= frmMain.Height;
```

Нам нужно позаботиться о том, чтобы пользователь мог быстро и удобно передавать в программу необходимые значения. Так как пол может быть либо мужской, либо женский, то мы поставим на форму две *кнопки* с буквами *Эм и Жо*, как говаривал Лёлик из комедии *Бриллиантовая рука*. Пользователь выберет ту кнопку, которую пожелает (например, кто-то захочет узнать свой идеальный вес при перемене пола или больше узнать о девочках).

```
  //кнопки:
   btnM := new Button;
   btnM.Text := 'M';
   btnM.Width := 48;
   btnM.Height := 32;
   btnM.Left := 290;
   btnM.Top := height-70;
   btnM.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   btnM.FlatStyle:= FlatStyle.Flat;
   btnM.ForeColor := Color.Red;
   btnM.Cursor:= Cursors.Hand;
   frmMain.Controls.Add(btnM);
   btnM.Click += btnM_Click;

   btnW := new Button;
   btnW.Text := 'Ж';
   btnW.Width := 48;
   btnW.Height := 32;
   btnW.Left := 10;
   btnW.Top := height-70;
   btnW.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   btnW.FlatStyle:= FlatStyle.Flat;
   btnW.ForeColor := Color.Red;
```

```
   btnW.Cursor:= Cursors.Hand;
   frmMain.Controls.Add(btnW);
   btnW.Click += btnW_Click;

   Application.Run(frmMain);
end.
```

Возраст и рост могут принимать большое количество значений, поэтому их придётся вводить с клавиатуры в два *текстовых поля*:

```
  //текстовые поля:
   txtVozrast:= new TextBox();
   txtVozrast.Width:= 160;
   txtVozrast.Left:= 180;
   txtVozrast.Top:= 210;
   txtVozrast.BorderStyle:= BorderStyle.FixedSingle;
   txtVozrast.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   frmMain.Controls.Add(txtVozrast);

   txtRost:= new TextBox();
   txtRost.Width:= 160;
   txtRost.Left:= 180;
   txtRost.Top:= 248;
   txtRost.BorderStyle:= BorderStyle.FixedSingle;
   txtRost.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   frmMain.Controls.Add(txtRost);
```

Нам потребуется ещё одно *текстовое поле* – для того, чтобы предъявить пользователю его идеальный вес:

```
   txtVes:= new TextBox();
   txtVes.Multiline:= true;
   txtVes.Height:= 50;
   txtVes.Width:= 300;
   txtVes.Left:= 180;
   txtVes.Top:= 290;
   txtVes.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   txtVes.ForeColor := Color.Red;
   txtVes.WordWrap:= false;
```

```
   txtVes.BorderStyle:= BorderStyle.FixedSingle;
   frmMain.Controls.Add(txtVes);
```

Вся премудрость нашей программы заключена в процедуре *calcVes*, в которую программа попадает после того как пользователь нажмёт одну из двух кнопок:

```
procedure btnW_Click(sender: Object; e: EventArgs);
begin
   calcVes('W');
end;

procedure btnM_Click(sender: Object; e: EventArgs);
begin
   calcVes('M');
end;

//Печатаем результат тестирования
procedure calcVes(pol: char);
begin
  var s:='';
  var sv:= txtVozrast.Text;
  var sr:= txtRost.Text;

  if (sv = '') then
      s:= ' Проверьте возраст!'
  else
      vozrast:= integer.Parse(sv);
  if (sr = '') then
      s += NewLine+ ' Проверьте рост!'
  else
      rost:= integer.Parse(sr);

  If ((vozrast < 1) Or (vozrast > 120)) and (s='') Then
     s:= ' Проверьте возраст!'
  Else If ((rost < 60) or (rost > 240)) and (s='') Then
     s:= ' Проверьте рост!';

  If (s<>'') Then begin
    txtVes.Text:= s;
    exit;
  End;
```

```
  If (pol= 'M') Then //для мужчин
      ves := (3 * rost  - 450 + vozrast) * 0.25 + 45
  Else                  //для женщин
      ves := (3 * rost  - 450 + vozrast) * 0.225 + 40.4;

  //корректируем отрицательные значения:
  if ves< 0 then
    ves := 4;

   s:= ' Идеальный вес равен ' + Math.Round(ves).ToString() +
' кг';
   txtVes.Text:= s;
end;
```

Действует-злодействует (любимое занятие Бабы-Яги) она так.

Считываем введённые пользователем данные в переменные *vozrast* и *rost*, после чего проверяем биологическую грамотность пользователя или злобный юмор шутника. Если данные выходят за пределы разумения, то расчёты вести глупо, поэтому мы сразу же печатаем последнее предупреждение.

Если пользователь успешно преодолел ввод анкетных данных, то по значению символьного параметра мы определяем пол «персонажа», в соответствии с которым и выбираем одну из двух формул.

На всякий случай мы заменяем отрицательный вес более осмысленным. Это вызвано тем, что формулы для расчёта веса не универсальны, то есть действительны только для разумных сочетаний параметров, поэтому идеальный вес Кощея Бессмертного или дядьки Черномора по ней вычислять нельзя.

Найдя идеальный вес, мы округляем его до целого числа (формулы, естественно, не настолько точны, чтобы рассчитывать ещё и граммы), и выводим в *текстовое поле txtVes* строку с полезной информацией (Рис. 33.1).

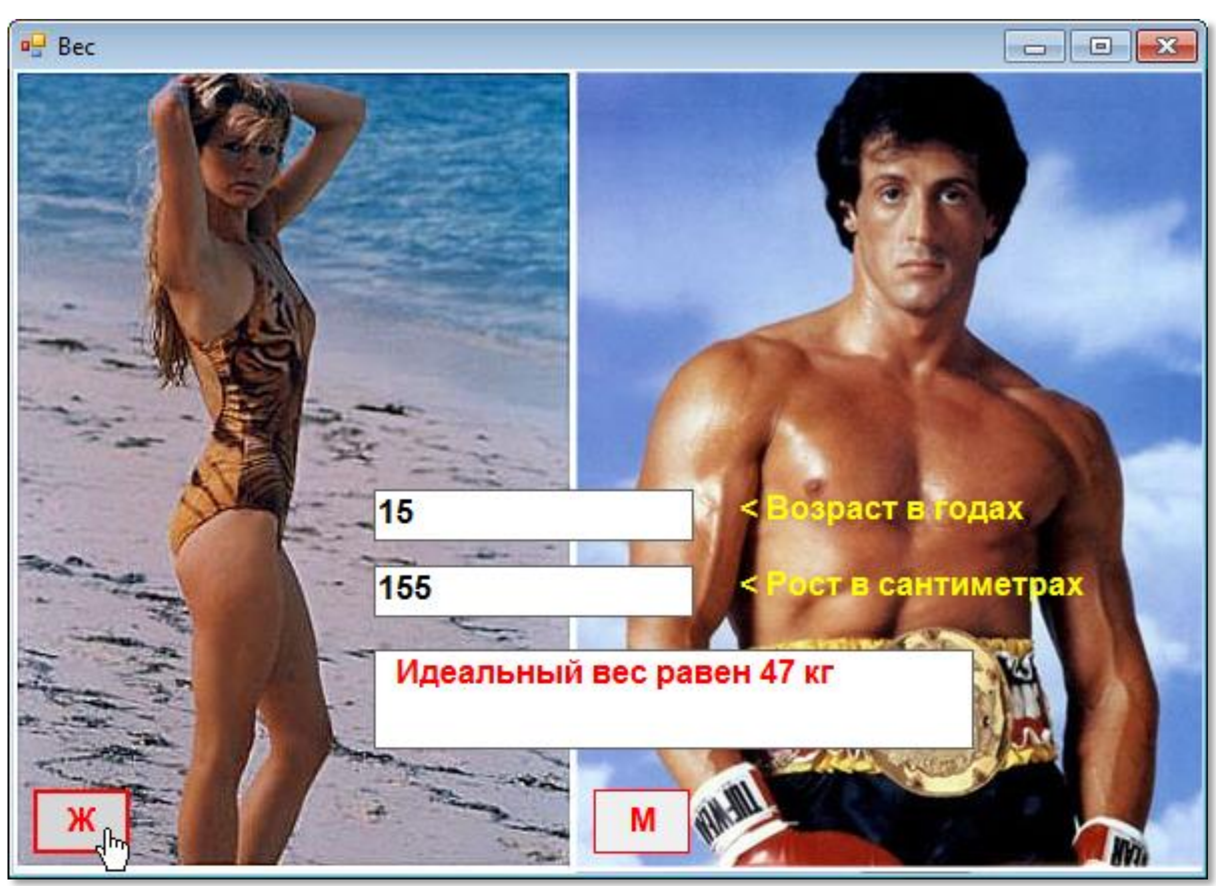


Рис. 33.1. Пора худеть?

Исходный код программы находится в папке **Вес**.

Жиропонижающее средство

90-60-90

Фигурка мирового класса

Продолжаем исследования рода человеческого с помощью компьютера. Немного подправив программу *Вес*, мы легко определим *жирность* тела любого гражданина (или гражданки).

Дотошная наука выяснила, что тело молодых здоровых мужчин содержит около 15%, а тело женщин – около 22% жира. Жирность произвольно выбранного «тела» приблизительно оценивается по формуле:

Ж = (Вес - Р) : Вес x 100 + 15 - для мужчин,
Ж = (Вес - Р) : Вес x 100 + 22 - для женщин,

где *Р* – идеальный вес, который определяется по предыдущим формулам.

Мы не будем целиком переписывать всю программу *Вес*, а адаптируем её к «новым реалиям». Для этого создадим новую папку **Жир**, в которую перепишем исходный текст программы *Вес* под новым названием.

Добавим ещё одну *переменную* – для хранения жирности тела:

```
zhir: real;
```

И *текстовое поле* – для ввода *действительного* веса пользователя. Заодно придётся подправить и другие *текстовые поля*:

```
  //текстовые поля:
   txtVozrast:= new TextBox();
   txtVozrast.Width:= 160;
   txtVozrast.Left:= 180;
   txtVozrast.Top:= 210;
   txtVozrast.BorderStyle:= BorderStyle.FixedSingle;
   txtVozrast.ForeColor := Color.Red;
   txtVozrast.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   frmMain.Controls.Add(txtVozrast);

   txtRost:= new TextBox();
   txtRost.Width:= 160;
   txtRost.Left:= 180;
   txtRost.Top:= 238;
   txtRost.BorderStyle:= BorderStyle.FixedSingle;
   txtRost.ForeColor := Color.Green;
   txtRost.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   frmMain.Controls.Add(txtRost);

   txtVes:= new TextBox();
   txtVes.Width:= 160;
```

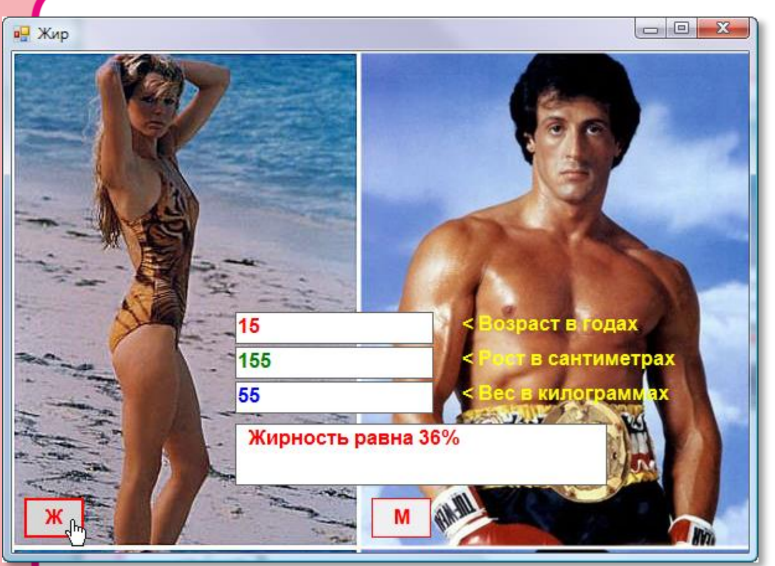
```
   txtVes.Left:= 180;
   txtVes.Top:= 266;
   txtVes.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   txtVes.ForeColor := Color.Blue;
   txtVes.BorderStyle:= BorderStyle.FixedSingle;
   frmMain.Controls.Add(txtVes);

   txtZhir:= new TextBox();
   txtZhir.Multiline:= true;
   txtZhir.Height:= 50;
   txtZhir.Width:= 300;
   txtZhir.Left:= 180;
   txtZhir.Top:= 300;
   txtZhir.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   txtZhir.ForeColor := Color.Red;
   txtZhir.WordWrap:= false;
   txtZhir.BorderStyle:= BorderStyle.FixedSingle;
   frmMain.Controls.Add(txtZhir);
```

Конечно, нужно потрудиться и над обработкой данных пользователя с учётом новых формул:

```
//Печатаем результат тестирования
procedure calcZhir(pol: char);
begin
  var s:='';
  var sv:= txtVozrast.Text;
  var sr:= txtRost.Text;
  var sves:= txtVes.Text;

  if (sv = '') then
      s:= ' Проверьте возраст!'
  else
      vozrast:= integer.Parse(sv);

  if (sr = '') then
      s += NewLine+ ' Проверьте рост!'
  else
      rost:= integer.Parse(sr);
```

```
  if (sves = '') then
      s += NewLine+ ' Проверьте вес!'
  else
      ves:= integer.Parse(sves);

  If ((vozrast < 1) Or (vozrast > 120)) and (s='') Then
     s:= ' Проверьте возраст!'
  Else If ((rost < 60) or (rost > 240)) and (s='') Then
     s:= ' Проверьте рост!'
  Else If ((ves < 10) or (ves > 240)) and (s='') Then
     s:= ' Проверьте рост!';

  If (s<>'') Then begin
    txtZhir.Text:= s;
    exit;
  End;

  If (pol= 'M') Then begin //для мужчин
      var ives := (3 * rost  - 450 + vozrast) * 0.25 + 45;
      zhir:= (ves - ives) / ves * 100 + 15
  end
  Else begin              //для женщин
      var ives := (3 * rost  - 450 + vozrast) * 0.225 + 40.4;
      zhir:= (ves - ives) / ves * 100 + 22;
  end;

  //корректируем  значения:
  if (zhir< 0) then
    zhir:= 0
  else If (zhir > 100) then
    zhir:= 100;

   s:= ' Жирность равна ' + Math.Round(zhir).ToString() + '%';
   txtZhir.Text:= s;
end;
```

Добавим проверку для введённого веса, а также расчёт жирности по формулам, и можно переходить к контрольному взвешиванию (Рис. 33.2).



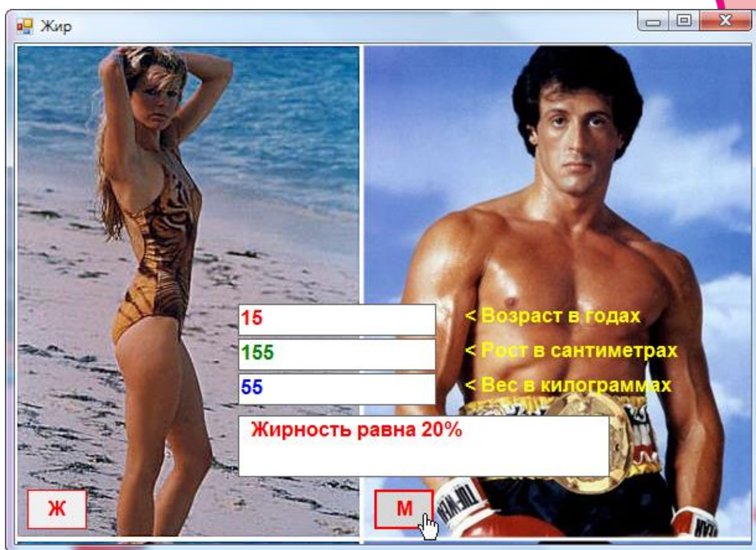


Рис. 33.2. А девочки-то «жирнее» мальчиков!

Исходный код программы находится в папке **Жир**.

Сколько кожи на человеке?

Кожа да кости!

Требование Славы Зайцева к моделям

Не лезьте из кожи!

Царевна Лягушка

Составим ещё одну познавательную программу по биологии. На этот раз мы вычислим площадь поверхности человеческого тела (то есть кожи). С точки зрения геометрии, фигура человека весьма «причудлива», поэтому не существует точных формул для определения площади тела.

Мы воспользуемся формулой *Бойде*, которая позволяет приближённо вычислить нужную нам величину (не пугайтесь - формула заковыристая):

$$S = (P \times 1000)^{(\lg(1/P)+35{,}75)\ /\ 53{,}2} \times H^{0{,}3} : 3118{,}2,$$

где

S – площадь кожи в квадратных метрах

P – вес человека в килограммах

H – его рост в сантиметрах

Так как для вычисления площади тела нужно знать *рост* и *вес* человека, то за основу нашей программы мы возьмём предыдущее наше творение - проект *Вес*, скопировав его в новую папку **Кожа**.

```
//ПРОГРАММА ДЛЯ ВЫЧИСЛЕНИЯ ПЛОЩАДИ КОЖИ ЧЕЛОВЕКА

//Приложение Windows Forms
#apptype windows
#reference 'System.Windows.Forms.dll'
#reference 'System.Drawing.dll'

uses
  System,
  System.Windows.Forms,
  System.Drawing;

var
   ves: real;
   rost: integer;

   frmMain: Form;
   btnKozha: Button;
   txtKozha, txtRost, txtVes: TextBox;
   lblVes, lblRost: System.Windows.Forms.Label;
   //размеры окна:

   width, height: integer;
```

Подправим *элементы управления*:

```
  //кнопка:
   btnKozha := new Button;
   btnKozha.Text := 'Вычислить!';
   btnKozha.Width := 120;
   btnKozha.Height := 32;
   btnKozha.Left := 290;
   btnKozha.Top := height-70;
```

```
   btnKozha.Font:= new System.Drawing.Font('Arial', 12,
System.Drawing.FontStyle.Bold);
   btnKozha.FlatStyle:= FlatStyle.Flat;
   btnKozha.ForeColor := Color.Red;
   btnKozha.Cursor:= Cursors.Hand;
   frmMain.Controls.Add(btnKozha);
   btnKozha.Click += btnKozha_Click;

   //текстовые поля:
   txtVes:= new TextBox();
   txtVes.Width:= 160;
   txtVes.Left:= 180;
   txtVes.Top:= 210;
   txtVes.BorderStyle:= BorderStyle.FixedSingle;
   txtVes.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   frmMain.Controls.Add(txtVes);

   txtRost:= new TextBox();
   txtRost.Width:= 160;
   txtRost.Left:= 180;
   txtRost.Top:= 248;
   txtRost.BorderStyle:= BorderStyle.FixedSingle;
   txtRost.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   frmMain.Controls.Add(txtRost);

   txtKozha:= new TextBox();
   txtKozha.Multiline:= true;
   txtKozha.Height:= 50;
   txtKozha.Width:= 300;
   txtKozha.Left:= 180;
   txtKozha.Top:= 290;
   txtKozha.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   txtKozha.ForeColor := Color.Red;
   txtKozha.WordWrap:= false;
   txtKozha.BorderStyle:= BorderStyle.FixedSingle;

   frmMain.Controls.Add(txtKozha);
```

Довольно странно, но формула Бойде никак не учитывает половую принадлежность испытуемого, хотя пропорции мужского и

женского тела заметно отличаются. Однако доверимся науке и уберём лишнюю кнопку.

Вычислять площадь тела мы будем в методе *btnKozha_Click*:

```
//Печатаем результат тестирования
procedure btnKozha_Click(sender: Object; e: EventArgs);
begin
  var str:='';
  var sv:= txtVes.Text;
  var sr:= txtRost.Text;

  if (sv = '') then
      str:= ' Проверьте вес!'
  else
      ves:= integer.Parse(sv);

  if (sr = '') then
      str += NewLine+ ' Проверьте рост!'
  else
      rost:= integer.Parse(sr);

  If ((ves < 10) Or (ves > 240)) and (str='') Then
     str:= ' Проверьте вес!'
  Else If ((rost < 60) or (rost > 240)) and (str='') Then
     str:= ' Проверьте рост!';

  If (str<>'') Then begin
    txtVes.Text:= str;
    exit;
  End;

  //вычисляем площадь кожи по формуле Бойде:
  var s:= (Math.Log10(1 / ves) + 35.75) / 53.2;
  s:= Math.Pow((ves * 1000), s) * Math.Pow(rost, 0.3) /
0.31182;

   str:= ' Площадь кожи равна ' + Math.Round(s).ToString() + '
кв. см';
   txtKozha.Text:= str;

end;
```

Переводим формулу Бойде на математический диалект паскалевского языка и выводим на экран площадь тела в квадратных *сантиметрах* (отбрасываем десятичные знаки с помощью метода *Round*). Как говорил Удав: *А в попугаях я гораздо длиннее!* (Рис. 33.3).

Обратите внимание, что буквой *s* мы обозначили площадь тела, как это обычно и принято в математике, поэтому идентификатор строки для вывода результатов пришлось изменить - *str*.

С помощью этой программы вы сможете вычислить площади всех доступных вам тел, к вящему удовольствию их «владельцев» (если они ещё не потеряли вкус к жизни после первых двух экспериментов).

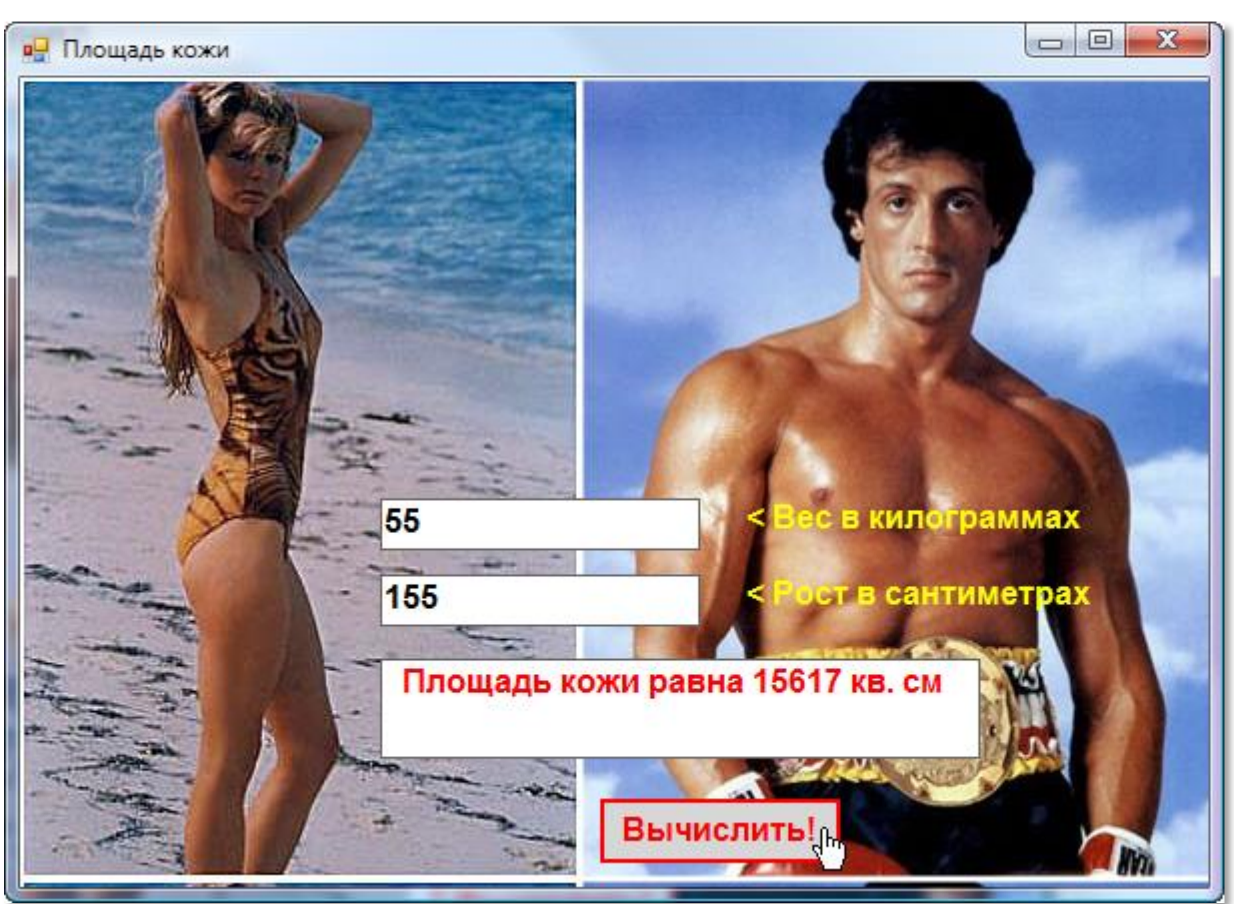


Рис. 33.3. С наукой не поспоришь!

Исходный код программы находится в папке **Кожа**.

1. Перед тем как предлагать программу *Вес* другим людям, очень полезно испытать её на себе. Запустите программу, введите данные, оцените все достоинства и недостатки интерфейса. И – непременно! – постарайтесь её улучшить.

2. Протестируйте программу *Жир* на ком только сможете. Наверняка это доставит и вам, и всем испытуемым массу удовольствия и запомнится на всю жизнь.

3. Посчитайте, сколько татуировок сможет разместить на себе Владимир Винокур, если средняя площадь татуировки равна 100 квадратным сантиметрам.

Урок 34. Занимательная психология

Очень часто в популярных тестах по психологии встречаются задания, в которых нужно быстро и правильно *сосчитать* какие-нибудь предметы: цветки, бабочек, якоря – да что угодно. Только не думайте, что это пустая забава!

В замечательной книге *Вам – взлёт!* Анатолий Маркуша рассказал о таком случае из фронтовой жизни.

Когда в нелётную погоду все лётчики полка скучали и слонялись по аэродрому, старший лейтенант Нико Ломия играл в спички: бросит несколько штук, быстро взглянет на них, сгребёт в кулак, снова бросит… И так просиживал он целые дни. Конечно, товарищи начали переживать за него – не сошёл ли он с ума от тоски? Или гадает на хорошую погоду? Дошла эта история и до командира полка. Он вызвал лейтенанта, а тот объяснил ему, что таким необычным способом он тренирует зрительную память: бросит горсть спичек и пытается с одного взгляда определить, сколько их. Он уже справлялся с дюжиной спичек, но цель у него была – 50 спичек.

А через полгода он стал лучшим воздушным разведчиком фронта. На большой скорости пролетая над вражеской территорией, он мгновенно подсчитывал и запоминал: орудий столько, самолетов – столько, танков – столько. И никогда не ошибался в своих подсчётах.

На следующей странице вы найдёте одно из таких заданий (Рис. 34.1). Вы можете просто пересчитать на время, сколько там разных цветков и листьев, но ещё лучше - сначала прикиньте, а потом проверьте, насколько вы ошиблись в своих предположениях.

Вы и сами легко можете сделать такие задачки – для себя и своих товарищей по классу. Возьмите чистый лист белой бумаги и в беспорядке нарисуйте какие-нибудь предметы. Можно использовать и красивые наклейки, тогда решать такие задачки будет ещё интереснее.

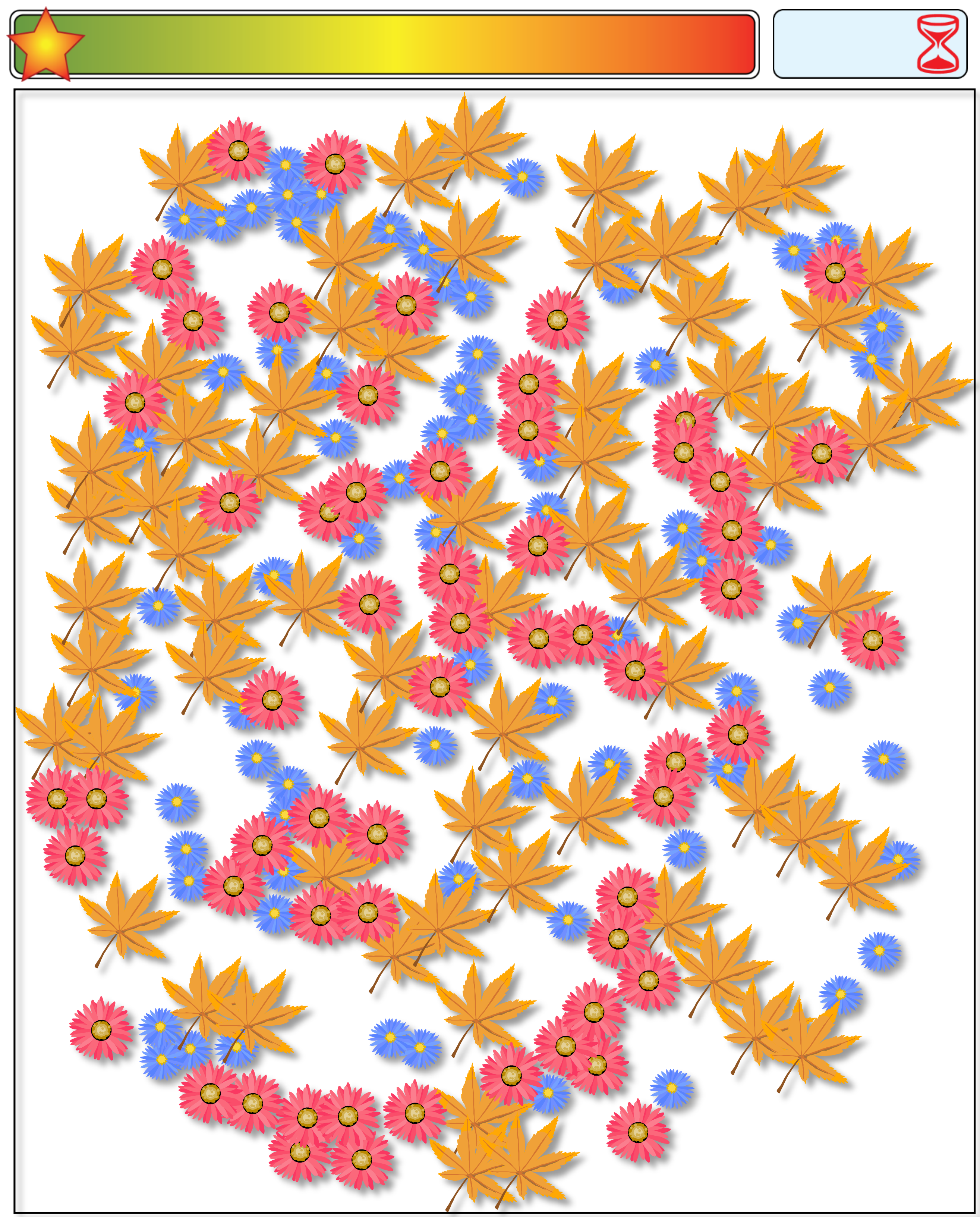


Рис. 34.1. Сосчитайте как можно быстрее, сколько здесь

1. Синих цветов ____________
2. Красных цветов ____________
3. Желтых листьев ____________

Психологическая считалка

Но самоделки - это потом, а сейчас мы поступим иначе – напишем программу, которая будет сама делать за нас такие задачки. Для почина мы будем рисовать **разноцветные** кружочки, но вы можете заменить их (или добавить к ним) квадратами или треугольниками. На другом уроке мы усовершенствуем программу так, чтобы она могла выводить на экран любые картинки, а не только простые геометрические фигуры.

А начнём мы новый проект **Психологическая считалка** с того, что уже хорошо знаем, то есть создадим приложение *Windows Forms*:

```
//ПСИХОЛОГИЧЕСКАЯ СЧИТАЛКА

//Приложение Windows Forms
#apptype windows
#reference 'System.Windows.Forms.dll'
#reference 'System.Drawing.dll'

uses
  System,
  System.Windows.Forms,
  System.Drawing;

//=========================================
//          ОСНОВНАЯ ПРОГРАММА
//=========================================
begin
   frmMain := new Form;
   frmMain.Text := 'Психологическая считалка';
   frmMain.Width:= 640;
   frmMain.Height:= 480;
   frmMain.StartPosition:= FormStartPosition.CenterScreen;
   frmMain.FormBorderStyle := Sys-
tem.Windows.Forms.FormBorderStyle.FixedSingle;
   frmMain.BackColor:= Color.FromArgb(255, $2f, $4f, $4f);
   frmMain.Paint += OnPaint;
   width:= frmMain.ClientSize.Width;
   height:= frmMain.ClientSize.Height;
```

Для определения времени, затраченного на выполнение теста, нам потребуется *таймер*, который будет срабатывать каждую секунду.

```
  _timer:= new Timer();
   _timer.Interval:=1000;
   _timer.Tick += OnTick;
   _timer.Stop();
   time:=0;
```

Процедура-обработчик выводит время в *метку*:

```
//ОТСЧИТЫВАЕМ ВРЕМЯ
procedure OnTick(sender: Object; e: EventArgs);
begin
   time += _timer.Interval/1000;
   lblTime.Text:= 'Время: ' + time.ToString();
end;
```

Её нужно создать:

```
    //метки:
   lblTime:= new System.Windows.Forms.Label();
   lblTime.Left:= 380;
   lblTime.Top:= 16;
   lblTime.AutoSize:= true;
   lblTime.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   lblTime.ForeColor := Color.Yellow;
   lblTime.BackColor := Color.Transparent;
   lblTime.Text:= 'Время: ';
   lblTime.Visible:= false;
   frmMain.Controls.Add(lblTime);
```

Не удивляйтесь комментарию «метки»: нам понадобится ещё одна метка – для вывода результата тестирования:

```
  lblInfo:= new System.Windows.Forms.Label();
   lblInfo.Left:= 40;
   lblInfo.Top:= 208;
   lblInfo.AutoSize:= true;
```

```
   lblInfo.Font:= new System.Drawing.Font('Arial', 48, Sys-
tem.Drawing.FontStyle.Bold);
   lblInfo.BackColor := Color.Transparent;
   lblInfo.Visible:= false;
   frmMain.Controls.Add(lblInfo);
```

Для ввода ответа игрока нам придётся создать ещё и одно *текстовое поле*:

```
   //текстовое поле:
   txtOtvet:= new TextBox();
   txtOtvet.Width:= 32;
   txtOtvet.Left:= 180;
   txtOtvet.Top:= 10;
   txtOtvet.BorderStyle:= BorderStyle.FixedSingle;
   txtOtvet.Font:= new System.Drawing.Font('Arial', 12,
System.Drawing.FontStyle.Bold);
   txtOtvet.Visible:= false;
   frmMain.Controls.Add(txtOtvet);
```

Не обойтись нам и без пары *кнопок*:

```
   //кнопки:
   btnStart := new Button;
   btnStart.Text := 'Начать тест!';
   btnStart.Width := 120;
   btnStart.Height := 32;
   btnStart.Left := 10;
   btnStart.Top := 10;
   btnStart.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   btnStart.FlatStyle:= FlatStyle.Flat;
   btnStart.ForeColor := Color.Yellow;
   btnStart.Cursor:= Cursors.Hand;
   frmMain.Controls.Add(btnStart);
   btnStart.Click += btnStart_Click;

   btnProv := new Button;
   btnProv.Text := 'Проверить!';
   btnProv.Width := 120;
   btnProv.Height := 32;
   btnProv.Left := 220;
   btnProv.Top := 10;
```

```
   btnProv.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   btnProv.FlatStyle:= FlatStyle.Flat;
   btnProv.ForeColor := Color. LightGreen;
   btnProv.Cursor:= Cursors.Hand;
   btnProv.Visible:= false;
   frmMain.Controls.Add(btnProv);
   btnProv.Click += btnProv_Click;
```

На этом создание нехитрого интерфейса программы закончено – можно её запускать (Рис. 34.2).

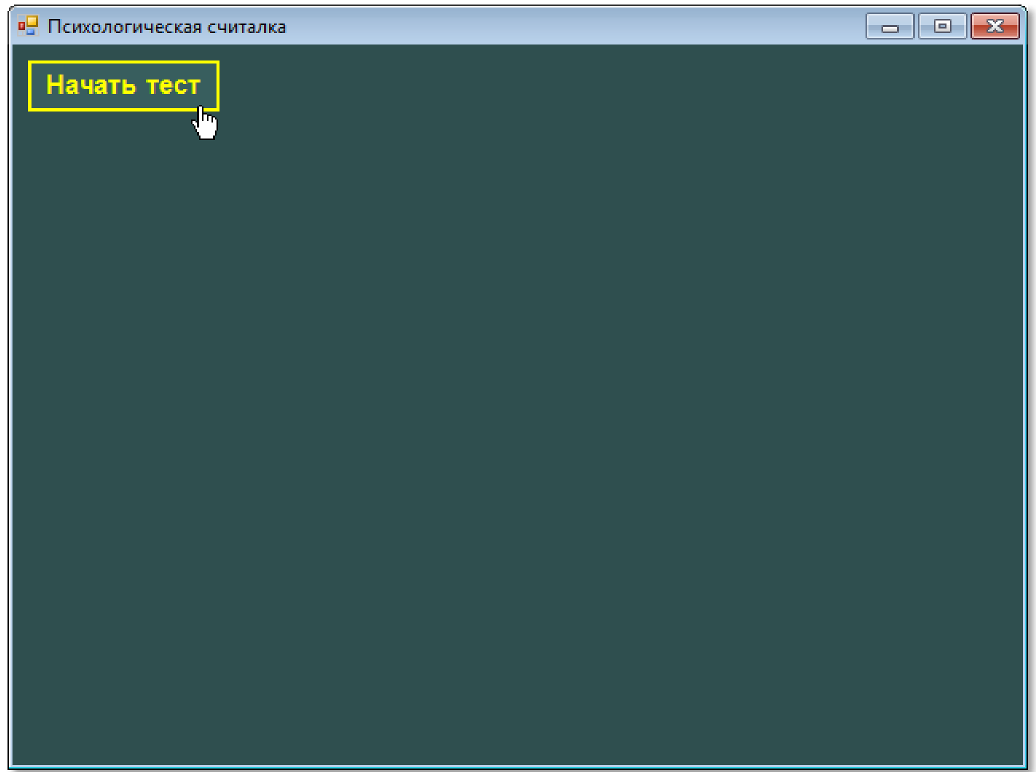


Рис. 34.2. Окно приложения

Одинокая кнопка не даст пользователю ни единого шанса запутаться в интерфейсе. Поскольку выбора у него нет, он нажмёт кнопку *Начать тест*, и, как мы видим по коду, попадёт в процедуру-обработчик нажатия на кнопку *btnStart_Click*:

```
//НАЧИНАЕМ ТЕСТИРОВАНИЕ
procedure btnStart_Click(sender: Object; e: EventArgs);
begin
   btnStart.Visible:= false;
   btnProv.Visible:= true;
   txtOtvet.Text:= '';
```

```
   txtOtvet.Visible:= true;
   lblInfo.Visible:= false;

   createTest();

   time:=0;
   _timer.Start();
   lblTime.Text:='Время: 0';
   lblTime.Visible:= true;
end;
```

Для тестирования нам нужно создать тест с помощью метода *createTest*, а также спрятать одни элементы управления и показать другие, чтобы пользователь в них не запутался.

Итак, кнопка нажата – создаём новый тест. Для этого мы просто разбрасываем по клиентской области окна **разноцветные** кружочки:

```
//СОЗДАЁМ ТЕСТ
procedure createTest;
begin
  var rand:= new Random();
  //число кружков:
  nKrug:= rand.Next(31)+ 20;
  circles:= new circle[nKrug+1];
  //отладка:
  frmMain.Text:= nKrug.ToString();
  for var i:= 1 to nKrug do begin
     //выбираем случайный радиус кружка:
     var radius:= rand.Next(16)+ 20;
     //выбираем случайные координаты  кружка:
     var x := rand.Next(width-2*radius);
     var y := rand.Next(height-2*radius-40) + 40;
     //выбираем случайный цвет для  кружка:
     var clr:= Color.FromArgb(255,rand.Next(255),
               rand.Next(255), rand.Next(255));
     circles[i].clr:= clr;
     circles[i].radius:= radius;
     circles[i].x:=x;
     circles[i].y:=y;
  end;
  frmMain.Invalidate();
```

```
End;
```

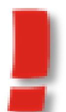
Число кружочков для каждого тестирования нужно выбирать другим, иначе будет неинтересно. В данном примере их будет от 20 до 50:

```
nKrug:= rand.Next(31)+ 20;
```

Вы легко можете выбрать другой интервал, например, для младших школьников лучше подойдет диапазон от 3 до 11.

При отладке программы очень важно знать, сколько кружков на поле, чтобы не пересчитывать их всякий раз заново, ведь программу придётся доводить до ума довольно долго:

```
//отладка:
//frmMain.Text:= nKrug.ToString();
```



Всегда вставляйте в свои программы *отладочные операторы*. А когда они станут не нужны, просто закомментируйте их.

Теперь можно нарисовать на экране заданное число кружков. Их размеры, цвет и положение мы также выбираем случайным образом, чтобы задания отличались друг от друга:

```
var radius:= rand.Next(16)+ 20;
```

Радиус кружков задайте по своему вкусу!

Выбирать место для нового кружка следует так, чтобы он целиком помещался на канве, а также не вторгался в ту область окна приложения, где расположены элементы управления:

```
     //выбираем случайные координаты  кружка:
     var x := rand.Next(width-2*radius);
     var y := rand.Next(height-2*radius-40) + 40;
```

Осталось нарисовать и показать кружки на экране (Рис. 34.3):

```
frmMain.Invalidate();
```

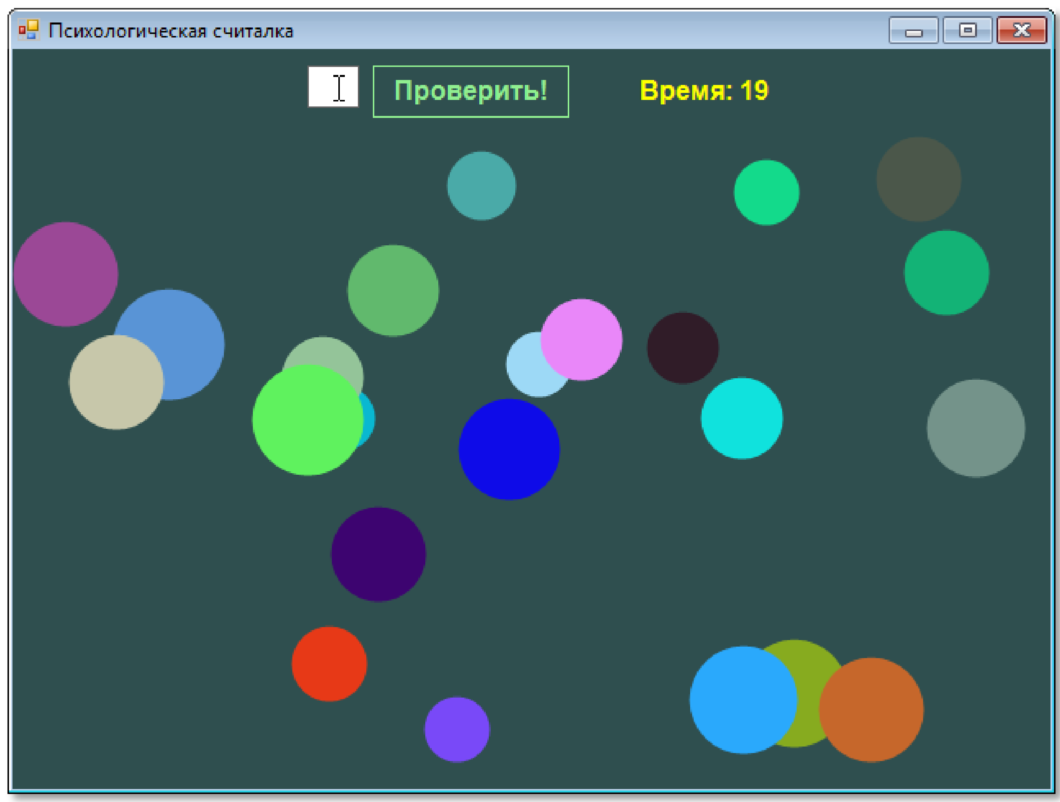


Рис. 34.3. А вот и цветные кружочки!

Канва формы не умеет хранить нарисованные на ней кружочки, поэтому мы запоминаем их свойства в динамическом массиве *circles:*

```
var
   //время:
   time: real;
   //число кружков:
   nKrug: integer;
   //кружки:
   circles: array of circle;

   frmMain: Form;
   btnStart, btnProv: Button;
   txtOtvet: TextBox;
   lblTime, lblInfo: System.Windows.Forms.Label;
   _timer: Timer;
   //размеры окна:
   width, height: integer;
```

Тип элементов массива – запись *circle:*

```
type
   circle = record
      clr: Color;
      radius: integer;
      x,y: integer;
   end;
```

Как видите, в полях записи мы сохранили *цвет* кружка, его *радиус* и *координаты*. По этим данным мы легко восстановим картинку, когда форма будет перерисовываться. А для этого она вызывает метод *OnPaint:*

```
//РИСУЕМ КРУЖКИ
procedure OnPaint(sender: object; e: PaintEventArgs);
begin
   var g := e.Graphics;
   g.SmoothingMode := Sys-
tem.Drawing.Drawing2D.SmoothingMode.HighQuality;
   var brush: SolidBrush;

   for var i:= 1 to nKrug do begin
      brush:= new SolidBrush(circles[i].clr);
      var r:= circles[i].radius;
      var x:= circles[i].x;
      var y:= circles[i].y;
      var rect := new Rectangle(x,y,2*r,2*r);
      //рисуем цветной кружок:
      g.FillEllipse(brush, rect);
   end;
end;
```

Именно здесь мы и рисуем на форме все кружки, извлекая их из массива *circles*. А вот чтобы принудить форму к перерисовке, мы после создания теста вызываем метод *Invalidate:*

```
frmMain.Invalidate();
```

Когда построение теста закончено, программа начинает отсчёт времени:

```
  time:=0;
   _timer.Start();
```

```
   lblTime.Text:='Время: 0';
   lblTime.Visible:= true;
```

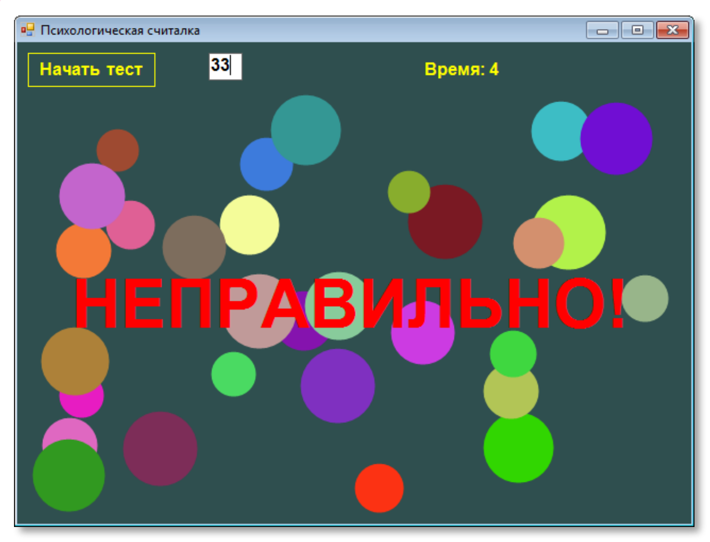
Испытуемый, подсчитав кружки на поле и напечатав ответ в *текстовом поле txtOtvet*, нажимает кнопку *Проверить*, после чего программа переходит к обработке этого события в методе *btnProv_Click*.

Таймер останавливается и показывает затраченное на тестирование время:

```
//ПРОВЕРЯЕМ ОТВЕТ
procedure btnProv_Click(sender: Object; e: EventArgs);
begin
   btnStart.Visible:= true;
   btnProv.Visible:= false;
   _timer.Stop;
```

Затем мы сравниваем число, введённое пользователем с клавиатуры в *текстовое поле txtOtvet*, с действительным числом кружков, и выдаём *результат* тестирования (Рис. 34.4):

```
   var s:= txtOtvet.Text;
   if s='' then s:='0';
   var n:= integer.Parse(s);
    If nKrug= n then begin
       s:='ПРАВИЛЬНО!';
       lblInfo.Left:= 90;
       lblInfo.ForeColor := Color.Yellow;
    end
    Else begin
      s:='НЕПРАВИЛЬНО!';
      lblInfo.Left:= 40;
      lblInfo.ForeColor := Color.Red;
    end;
    lblInfo.Text:= s;
    lblInfo.Visible:= true;
end;
```



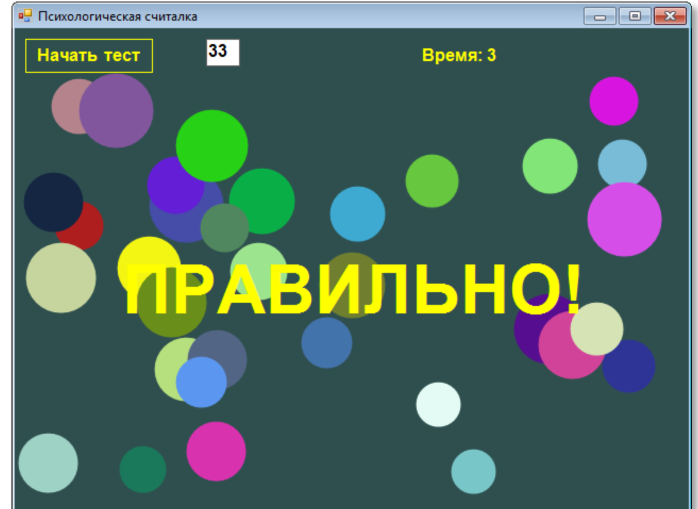


Рис. 34.4. Пользователь погорячился! Результат тренировок налицо!

На рисунках видно (и по исходному тексту тоже!), что снова появилась призывная кнопка *Начать тест* - *Психологическая считалка* снова в бою!

Исходный код программы находится в папке **Психологическая считалка**.

1. В случае неправильного ответа программа только сообщает пользователю, что он неверно подсчитал кружочки. Добавьте к строке *правильное* число кружков, чтобы пользователь мог оценить, насколько он ошибся.

2. Так как цвет кружков выбирается случайно, то он может совпасть с цветом фона *frmMain.BackColor:= Color.FromArgb(255, $2f, $4f, $4f);* и тогда будет невидим для пользователя. Вероятность такого события очень невелика, но её следует учесть. В методе *createTest* при совпадении случайного цвета *clr* с цветом фона нужно выбрать другой цвет. Либо выбирать такие цвета, которые наверняка не совпадут с цветом фона. «Миссия» кажется невыполнимой, однако здесь всё просто! Форма не поддерживает *прозрачные* цвета фона, поэтому в методе *FromArgb* первый аргумент всегда равен 255 (полная непро-

зрачность), а вот кружки можно рисовать и прозрачными цветами:

```
var clr:= Color.FromArgb(200,rand.Next(255),
            rand.Next(255), rand.Next(255));
```

Поскольку прозрачный цвет всегда отличается от непрозрачного, то они никогда не совпадут (хотя и могут быть мало отличимы друг от друга).

Этим трюком мы убиваем и второго зайца! Большой кружок может полностью закрыть маленький, который появился на экране раньше. То, что некоторые кружки частично перекрывают друг друга, это неплохо, так как это делает задачу более трудной. Однако полностью закрытый кружок вообще не видно, поэтому результат тестирования будет неверным. Однако под частично прозрачным кружком легко увидеть и все кружки, лежащие под ним (Рис. 34.5).

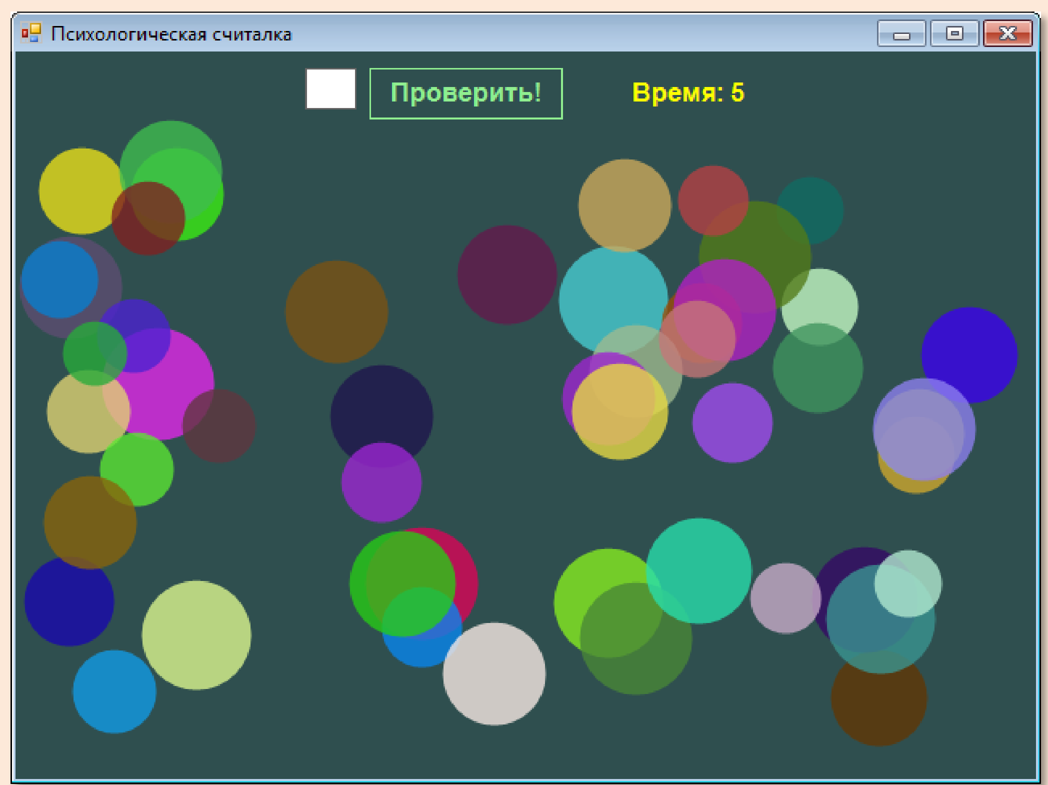


Рис.34.5. Полупрозрачные кружки

3. Другой способ избавления от перекрывающихся кружков – геометрический. Будем проверять в функции *testCircle* расстояние между центром нового кружка и центрами всех уже готовых кружков. Если хотя бы одно из них меньше суммы радиусов тестируемых кружков, значит, они частично или полностью перекрываются. Функция возвращает *false*, и процедура *createTest* продолжает попытки создать новый кружок на пустом месте канвы.

```
var n:=0;
while n < 100 do begin
   if testCircle(i,x,y,radius) then break;
   radius:= rand.Next(16)+ 20;
   x := rand.Next(width-2*radius);
   y := rand.Next(height-2*radius-40) + 40;
   Inc(n);
end;

function testCircle(n: integer; x,y: real; r: inte-
ger): boolean;
begin
   Result:= true;
   if n=1 then exit;
   //проверяем наложение на другие кружки:
   for var i:= 1 to n do begin
       //расстояние между центрами кружков:
       var dx:= circles[i].x - x;
       var dy:= circles[i].y - y;
       var dist := Math.Sqrt(dx * dx + dy * dy);
       var r2:= circles[i].radius;
       if dist < r + r2 then begin
          Result:= false;
          exit;
       end
    end;
 end;
```

Важно ограничить число попыток, иначе может сложиться такая ситуация, что новый кружок вообще невозможно будет нарисовать без перекрытия, и тогда программа зациклится!

Для этого мы ввели переменную *n*. Она подсчитывает число неудачных попыток нарисовать кружок в чистом поле. После 100 попыток кружок будет нарисован наудачу.

Рис. 34.6 показывает, что теперь кружки почти не перекрывают друг друга!

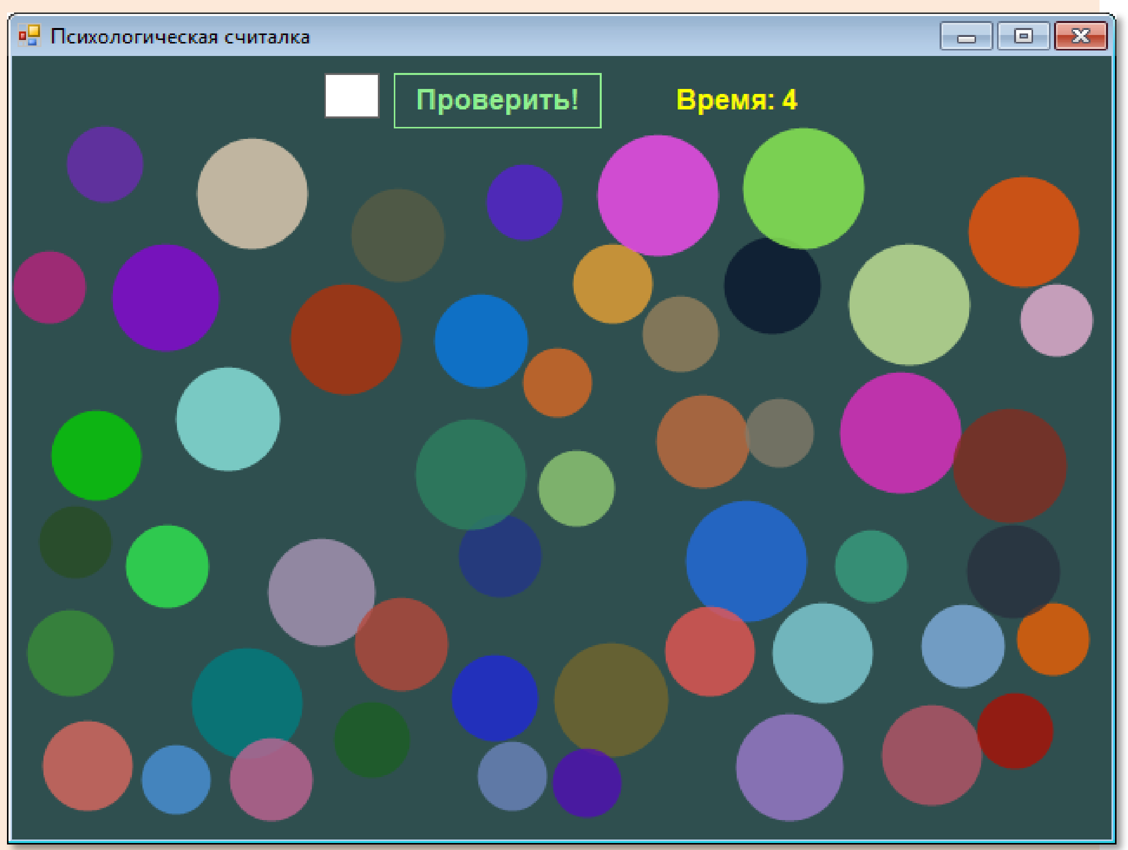


Рис. 34.6. Сохраняй дистанцию!

Исходный код программы находится в папке **Психологическая считалка**.

Урок 35. Звёздное небо

Открылась бездна, звезд полна…

М.В.Ломоносов

Трудно быть богом

Братья Стругацкие

Если бы мы жили в центре Галактики, то наше звёздное небо очень напоминало бы картинку из *Психологической считалки*. Но нам в смысле звёздной красоты повезло значительно меньше, поскольку мы живем на периферии, на забытой богом спирали. Куда ни глянь: звёзды мелкие и **жёлтые**.

На самом деле звёзды БОЛЬШИЕ и **разноцветные**, просто они нам такими кажутся из-за огромного расстояния до них. Правда, даже в самый сильный телескоп звёзды всё равно выглядят точками, хотя и **цветными**.

Если усыпать канву маленькими кружочками, то мы получим картину, вполне *напоминающую* настоящее небо.

Чтобы небо выглядело натурально, нам придётся задать координаты *всех* видимых глазом звёзд, а это несколько тысяч штук! Также нам будет необходимо учесть их цвет и блеск (звёздную величину).

Поскольку нам не нужно заботиться о перекрывании звёзд и прочих премудростях, с которыми мы столкнулись в психологическом тесте, то программа получится очень простая.

Звёзды и на самом деле закрывают друг друга, например, затменные переменные звёзды делают это периодически.

Начнём новый проект **Звёздное небо** с заголовка окна и цвета неба (Рис. 35.1, слева):

```
//ПРОГРАММА Звёздное небо

uses
   GraphABC, ABCButtons;

var
  //размеры окна:
  height, width: integer;
  //кнопка:
  btnStart: ButtonABC;
  //цвет неба:
  SkyColor: Color;
```

Если хотите, можете сделать небо совсем **чёрным** (Рис. 35.1, справа), раскомментировав строку

```
//SkyColor:= Color.Black;,
```

тогда звёзды будут казаться более яркими:

```
//
// ОСНОВНАЯ ПРОГРАММА
//
begin
   SetWindowTitle('Звёздное небо');
   SetWindowWidth(640);
   Window.Height:= 480;
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   SkyColor:= Color.MidnightBlue;
   //SkyColor:= Color.Black;
   Window.Clear(SkyColor);
   height := Window.Height;
   width := Window.Width;
```

Нам будет достаточно одной *кнопки*:

```
  //КНОПКА
   btnStart := new ButtonABC(10, 10, 100, 30, 'Новое небо!',
clMoneyGreen);
   btnStart.FontColor:= Color.Red;
   btnStart.FontStyle:= fsBold;
   btnStart.TextScale:= 0.9;
```

```
    //процедура-обработчик:
    btnStart.OnClick := btnStart_OnClick;
end.
```

Вот так, с помощью единственной кнопки, мы будем менять картину мира в галактических масштабах!

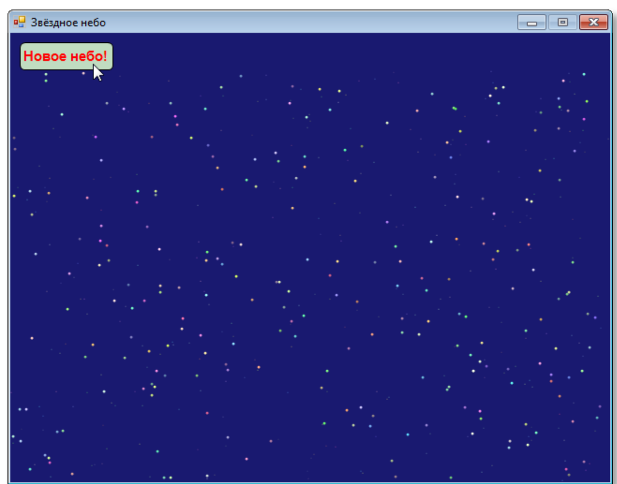


Рис. 35.1. Звёзд много и все они разные, но нет на искусственном небе наших родных созвездий Ориона, Лиры, Пегаса, Лебедя и обеих Медведиц!

Как только мы нажмём заветную кнопку творца

```
procedure btnStart_OnClick;
begin
  createSky();
End;
```

будет вызвана процедура *createSky*, где произойдет *Большой взрыв*, и небо украсится мириадами звёзд:

```
//Создаём небо
procedure createSky();
begin
    //очищаем небо:
    SetBrushColor(SkyColor);
    FillRectangle(0,40,width,height-40);
    //число звезд:
    var nStar:= Random(100)+ 400;
    for var i:= 1 to nStar do begin
        //выбираем случайный радиус звезды:
```

```
      var radius:= Random(2) + 1;
      //выбираем случайные координаты  звезды:
      var x := Random(width-2*radius);
      var y := Random(height-2*radius-40) + 40;
      //выбираем случайный цвет для  звезды:
      var r:= Random(164) + 92;
      var g:= Random(164) + 92;
      var b:= Random(164) + 92;
      var clr:= RGB(r,g,b);
      SetBrushColor(clr);
      //рисуем звезду:
      FillEllipse(x, y, x+2*radius,y+2*radius);
  End;//For
End;
```

С точки зрения программирования, ничего таинственного в этом созидательном акте нет. Единственное, на что следует обратить внимание: мы выбираем случайный цвет звезды не с помощью функции *clColor*, что было бы проще. Но нам нужны достаточно *яркие* звезды (иначе они потеряются на небе), поэтому мы задаём такие составляющие цвета *r, g, b*, чтобы они были от 93 (самые тусклые) до 255 (самые яркие), а далее формируем из них цвет в функции *RGB:*

```
var clr:= RGB(r,g,b);
```

Если вы останетесь недовольны видом созданного вами неба, смело жмите на кнопку *Новое небо*!

Исходный код программы находится в папке **Звёздное небо**.

1. Кроме одиночных и кратных звёзд, с рисованием которых наша программа неплохо справляется, на небе можно увидеть и *звёздные скопления* – группы звезд, связанных общим происхождением и силами гравитации. Они бывают *рассеянные* и *шаровые*.

Самое известное рассеянное звёздное скопление в наших широтах – это *Плеяды*, или *Стожары*. Оно находится в созвездии

Тельца и сравнительно недалеко от Солнца, поэтому многие звёзды этого скопления можно видеть невооружённым глазом.

Давайте добавим к нашему звёздному небу несколько рассеянных скоплений (Рис. 35.2). Для нас важно знать, что в них не очень много звёзд и они **белого** и **голубого** цвета.

Для создания рассеянных звездных скоплений (РЗС) нам потребуется ещё одна *кнопка*:

```
  btnAddOpenCluster:= new ButtonABC(140, 10, 120,
30, 'Добавить РЗС!', clMoneyGreen);
   btnAddOpenCluster.FontColor:= Color.Blue;
   btnAddOpenCluster.FontStyle:= fsBold;
   btnAddOpenCluster.TextScale:= 0.9;
   //процедура-обработчик:
   btnAddOpenCluster.OnClick := btnSAddOpenCluster_OnClick;
```

Потребуется написать и новую процедуру-обработчик:

```
procedure btnSAddOpenCluster_OnClick;
begin
   createOpenCluster();
end;
```

Но самое главное – в новой процедуре:

```
//СОЗДАЁМ РАССЕЯННОЕ ЗВЕЗДНОЕ СКОПЛЕНИЕ
procedure createOpenCluster;
begin
   //число звезд:
   var nStar:= Random(11)+ 7;
   //диаметр скопления:
   var rOC:= Random(21)+20;
   //левый верхний угол скопления:
   var xOC := Random(width-2*rOC);
   var yOC := Random(height-2*rOC-40) + 40;
   var clr: Color;
   for var i:= 1 to nStar do begin
      //выбираем случайный диаметр звезды:
      var diam:= Random(4)+1;
      //выбираем случайные координаты  звезды:
      var x := Random(2*rOC)+xOC;
      var y := Random(2*rOC-40) + 40 + yOC;
```

```
        //выбираем случайный цвет для  звезды:
        var nclr:= Random(4) +1;
        If (nclr= 1) Then
            clr:= Color.White
        else if (nclr= 2) Then
            clr:= Color.AliceBlue
        else if (nclr= 3) Then
            clr:= Color.LightSkyBlue
        else
            clr:= Color.DeepSkyBlue;

        SetBrushColor(clr);
        //рисуем звезду:
        FillEllipse(x, y, x+diam, y+diam);
    End;//
End;
```

Пользуясь этими подсказками, напишите новый вариант программы (Рис. 35.3)!

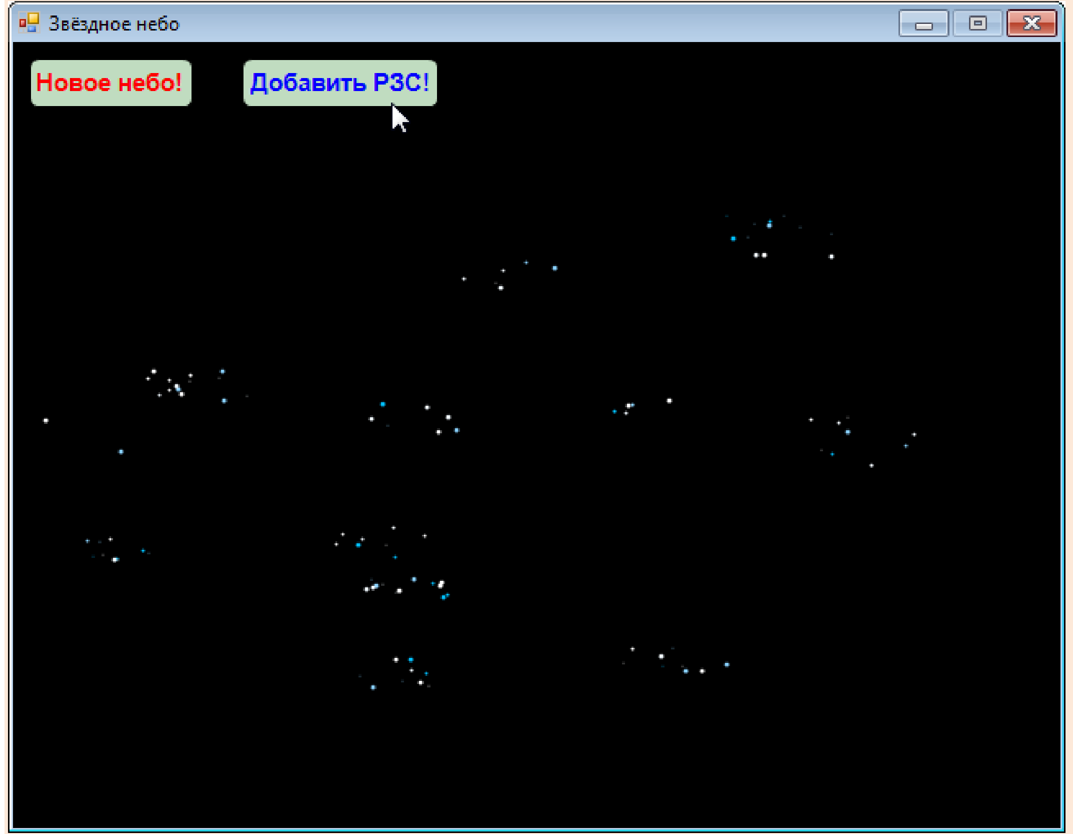


Рис. 35.2. 12 рассеянных скоплений

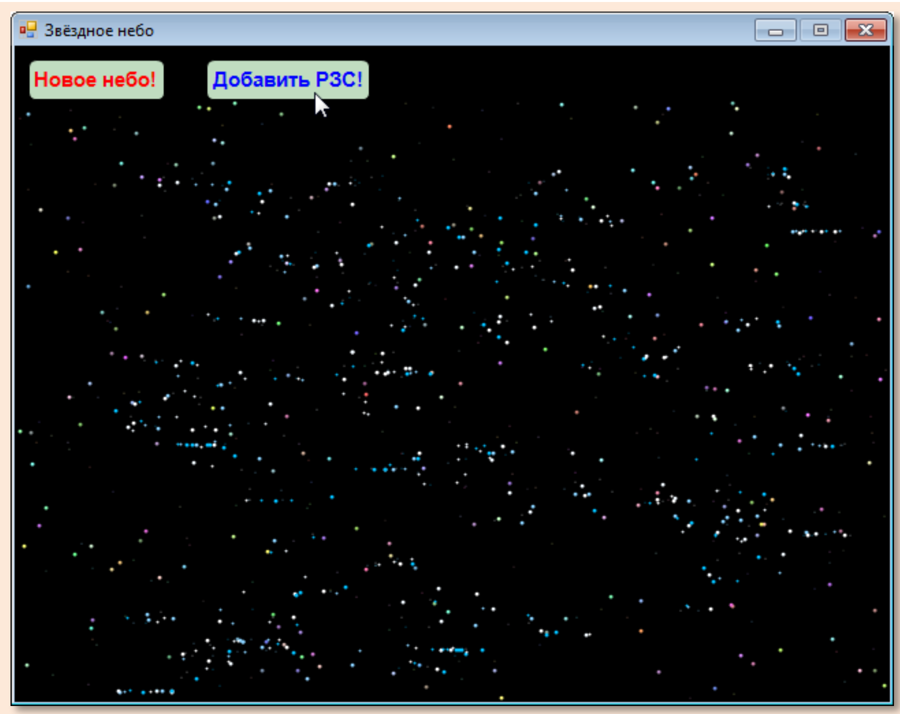
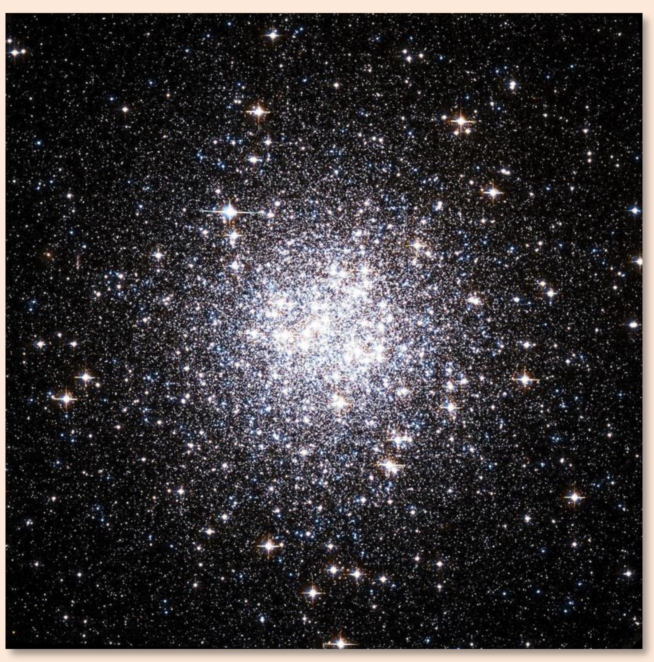


Рис. 35.3. Звёздное небо стало живописнее!

2. Любуясь звёздами, вы, конечно, не могли забыть и о *шаровых* звёздных скоплениях, в которых звёзд гораздо больше, чем в рассеянных, и форма у них сферическая, причем концентрация звёзд увеличивается к центру скопления (Рис. 35.4).



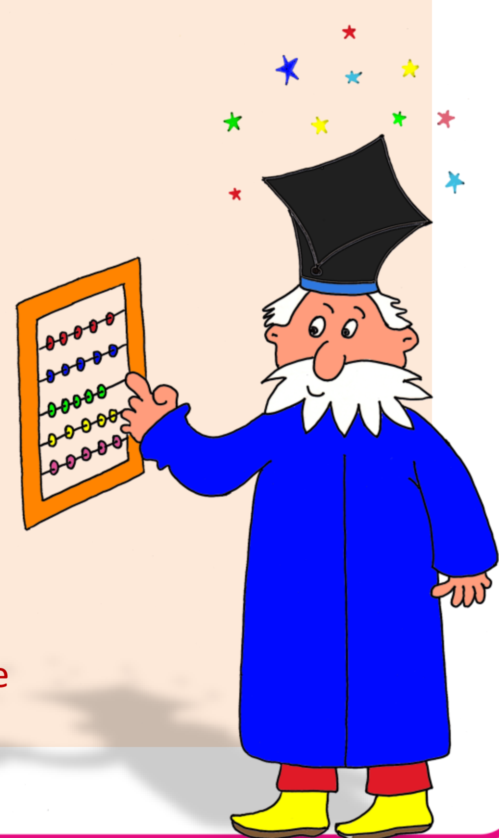


Рис. 35.4. Шаровое звёздное скопление

Цвет звёзд в таких скоплениях – от белого до **красного**, то есть можно брать такие же цвета, как для одиночных звёзд в процедуре *createSky*.

Исходный код программы находится в папке **Звёздное небо**.

Урок 36. С первого взгляда!

Когда видишь деньги,
не теряй времени!

Кинокомедия *Бриллиантовая рука*

На уроке *Занимательная психология* мы говорили о задачах, в которых требуется очень быстро, с первого взгляда определить количество предметов. Пример такого психологического теста вы найдёте на следующей странице (Рис. 36.1).

Конечно, решая задание в книге, вы можете и схитрить - подглядывать или считать монетки пальцем. Да и задание только одно – решил и сказочке конец. А ведь так хочется продлить удовольствие… И тут нам опять поможет компьютер!

Начинаем проект **С первого взгляда!**, сохраняем его в папке, объявляем *константу* и *переменные*:

```
const
   N_IMAGE=6;

type
kartinka = record
   //номер картинки:
   n: integer;
   radius: integer;
   x,y: integer;
end;

var
   //время:
   time: real;
   //число картинок:
   nImage: integer;
   //картинки:
   kartinki: array of kartinka;

   images: array of Image;
   img: array[1..N_IMAGE] of Bitmap;
```

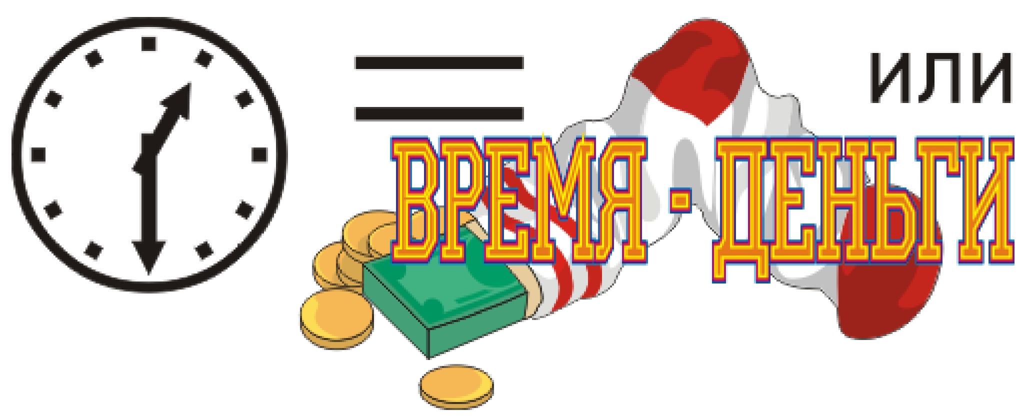


Рис. 36.1. Определите с первого взгляда, **сколько здесь монет**

Нам потребуется *таймер* для отсчёта заданного времени и *картинки* с монетами. Согласно константе *N_IMAGE*, их должно быть *шесть* штук. Хранятся они в папке с выполняемым файлом, поэтому путь к ним указывать не нужно – достаточно имени файла.

Загружаем картинки в массив *img:*

```
  for var i:= 1 to N_IMAGE do begin
     img[i]:= new Bitmap(i.ToString() + '.png');
   end;
```

Всё – картинки у нас в кармане! Теперь мы можем рисовать их на канве - где угодно и сколько угодно.

Элементы управления - точно такие же, как в программе *Психологическая считалка 2*, как и б**о**льшая часть этой программы. Выводы сделайте сами!

Методы *btnStart_Click* и *btnProv_Click* изменились незначительно:

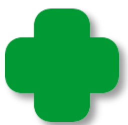
```
//НАЧИНАЕМ ТЕСТИРОВАНИЕ
procedure btnStart_Click(sender: Object; e: EventArgs);
begin
   btnStart.Visible:= false;
   //btnProv.Visible:= true;

//ПРОВЕРЯЕМ ОТВЕТ
procedure btnProv_Click(sender: Object; e: EventArgs);
begin
    . . .
    //txtOtvet.Visible:= false;
    If nImage= n then begin
       s:='ПРАВИЛЬНО - ' + nImage.ToString() + '!';
       lblInfo.Left:= 80;
       lblInfo.ForeColor := Color.Green;
    end
    Else begin
      s:='НЕПРАВИЛЬНО - '+ nImage.ToString() + '!';
      lblInfo.Left:= 40;
```

```
      lblInfo.ForeColor := Color.Red;
    end;
   . . .
end;
```

Тест формируется из загруженных *картинок*, а не из простых геометрических фигур, поэтому для программы можно использовать *любые* изображения!

```
//СОЗДАЁМ ТЕСТ
procedure createTest;
begin
  var rand:= new Random();
  //число картинок:
  nImage:= rand.Next(31)+ 20;
  kartinki:= new kartinka[nImage+1];
  //отладка:
  //frmMain.Text:= nImage.ToString();
  for var i:= 1 to nImage do begin
     //выбираем случайную картинку:
     var nImg:=rand.Next(N_IMAGE)+1;
     //"радиус" картинки:
     var radius:= Math.Max(img[nImg].Width, img[nImg].Height)
div 2;
     //выбираем случайные координаты  картинки:
     var x := rand.Next(width-img[nImg].Width);
     var y := rand.Next(height-img[nImg].Height-40) + 40;
     var n:=0;
     while n < 100 do begin
        if test(i,x,y,radius) then break;
        radius:= Math.Max(img[nImg].Width, img[nImg].Height)
div 2;
        x := rand.Next(width-img[nImg].Width);
        y := rand.Next(height-img[nImg].Height-40) + 40;
        Inc(n);
     end;

     kartinki[i].n:= nImg;
     kartinki[i].radius:= radius;
     kartinki[i].x:=x;
     kartinki[i].y:=y;
  end;
  frmMain.Invalidate();
End;
```

Здесь важно позаботиться о том, чтобы картинки не закрывали друг друга. Для этого мы 100 раз пытаемся вывести картинку на свободное место канвы и только потом бросаем монетку наудачу. При небольшом числе монет этот способ создания теста действует очень хорошо.



Поскольку все картинки *квадратные* (монеты, конечно, круглые, но они занимают только часть картинки), то радиуса у них нет. Чтобы не утруждать себя бесполезными геометрическими расчётами, мы принимаем за радиус половину длинной стороны картинки.

Время тестирования мы ограничим *десятью* секундами:

```
//ОТСЧИТЫВАЕМ ВРЕМЯ
procedure OnTick(sender: Object; e: EventArgs);
begin
   time += _timer.Interval/1000;
   lblTime.Text:= 'Время: ' + time.ToString();
   If (time>= 10) Then begin
      _timer.Stop();
      btnProv.Visible:= true;
      txtOtvet.Visible:= true;
      frmMain.Invalidate();
   End;//If
end;
```

Когда время истечёт, все монеты исчезнут с экрана, а в окне приложения появятся *кнопка* и *текстовое поле* (Рис. 36.2). Тестируемый должен ввести число – соответственно тому, сколько монет он успел насчитать, и нажать кнопку *Проверить*.

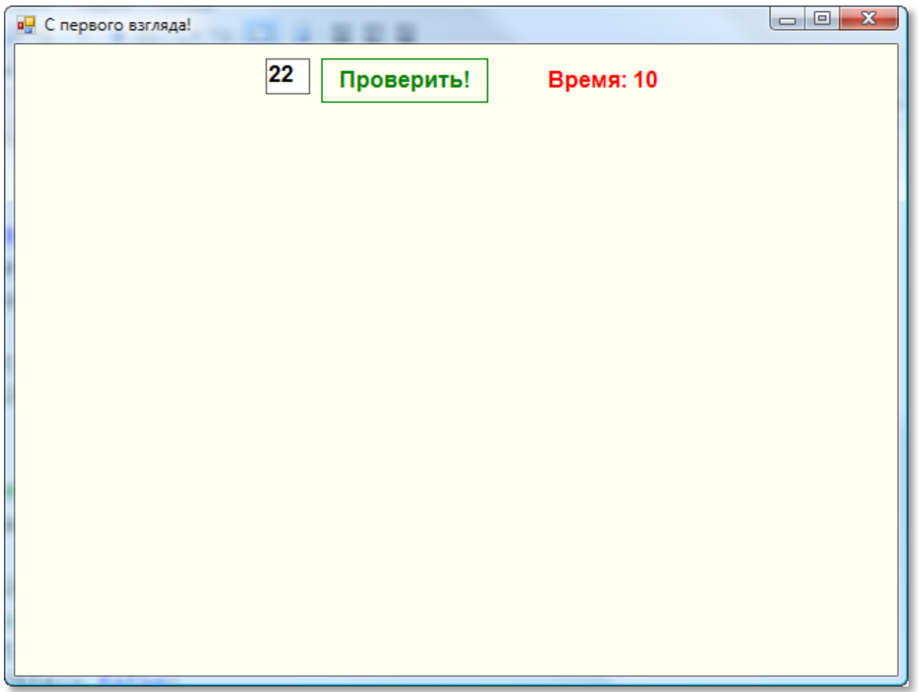


Рис. 36.2. Время вышло!

Как говорится, вердикт не заставит себя долго ждать (Рис. 36.3).

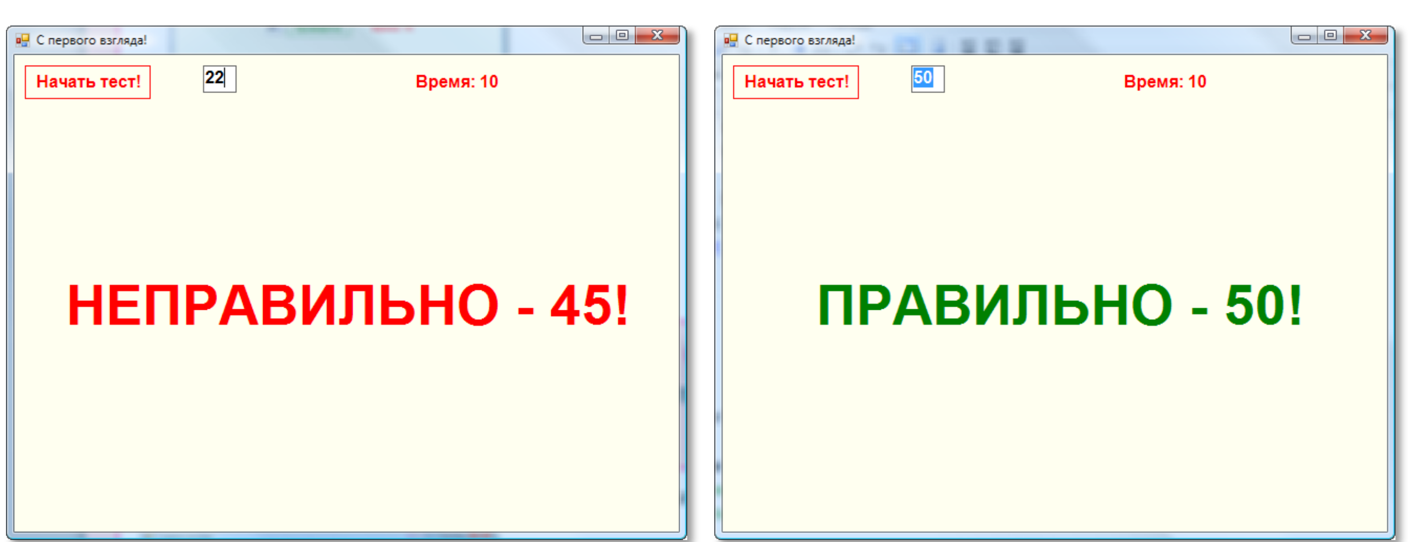


Рис. 36.3. Оттестировались по-разному!

Ну вот, к тесту мы подготовились. Запускаем программу, нажимаем кнопку *Начать тест!* и стараемся уложиться в десять секунд (Рис. 36.4).

Если вам это удаётся, вы смело можете идти в банкиры и считать чужие деньги!

Рис. 36.4. Осталось 6 секунд… Ма-ма!

Исходный код программы находится в папке **С первого взгляда!**

Урок 37. Тараканьи бега по методу Монте-Карло

Вдруг из подворотни
Страшный великан,
Рыжий и усатый
Та-ра-кан!
Таракан, Таракан, Тараканище!

Корней Чуковский, *Тараканище*

Тараканьи бега – любимое развлечение *аристократов и дегенератов*, как выразился бы Лёлик из комедии *Бриллиантовая рука,* – проводятся так.

Берут в нужном количестве больших - длиной до 10 сантиметров - тараканов с острова Мадагаскар (Рис. 37.1). Конечно, тараканам всё равно куда бежать, поэтому их помещают в направляющие желобки, избавляющие их от тяжкого бремени выбора пути. Длина забега 1,5 – 2 метра. Зрители и прочие ротозеи делают ставки на полюбившегося им «спортсмена», после чего все тараканы весело бегут к финишу, подбадриваемые дружеским свистом публики.

Рис. 37.1. И вправду тараканище!

Если вам не довелось воочию увидеть это жуткое зрелище, то вы можете прочитать его художественное описание в пьесе *Бег* Михаила Булгакова (Рис. 37.2) и в романе Алексея Толстого *Похождения Невзорова, или Ибикус*.

Рис. 37.2. Призовые гонки тараканов в фильме *Бег* (1970)

Более изощрённый вариант тараканьих бегов показан в фильме *Китайский сервиз*, где тараканы по условному сигналу подносили шулеру карты, приклеенные к спине (Рис. 37.3).

Рис. 37.3. Пляшущие тараканы из фильма *Китайский сервиз*

Наверное, вам встречались и компьютерные игры, симулирующие тараканьи бега (или лошадиные, крысиные, собачьи и поросячьи). Мы сейчас тоже напишем программу, но не игровую, а вполне серьёзную, реализующую простую *компьютерную модель* вообще любых подобных состязаний.

Чтобы не отвлекаться на анатомические подробности разных видов «спортсменов», мы будем считать их *эллипсами*. Эта фигура вполне адекватно заменяет тушки бегунов. Их количество

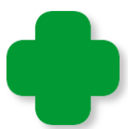
можно принять равным шести, но при желании это число можно изменить в любую сторону.

Перед стартом все участники выстраиваются в одну линию, после чего следует сигнал к началу гонок. Тараканы бегут вперёд, пока первый из них не пересечёт финишную черту. Он объявляется победителем, а все игроки, сделавшие на него ставки, получают призовые деньги.

Если бы в забегах всегда побеждал один и тот же таракан, то состязание потеряло бы всякий смысл, поскольку все зрители ставили бы на него - без всякой надежды получить выигрыш больше той суммы, что они поставили. Значит, в гонках тараканы должны бежать не с одинаковой скоростью, более того, они должны совершать рывки или отставать. На результаты гонок могут влиять и другие факторы, например, физическое состояние участников, питание и настроение.

Поскольку учесть все эти факторы невозможно, то в нашей модели мы будем считать, что скорость бега каждого таракана изменяется по дистанции *случайным* образом. Кроме того, все остальные факторы мы объединим в одну группу и обозначим их как текущее *здоровье* таракана.

Таким образом, наша компьютерная модель будет описывать случайные процессы. Неудивительно, что для этого используют *метод Монте-Карло*. Как вы знаете, в этом городе находится самый известный в мире игорный дом, а рулетку вполне можно считать хорошим генератором случайных чисел. Действительно, шарик в рулетке случайно «выбирает» одно из 37 чисел (от 1 до 36 и 0 - зеро) на колесе.



Важно, чтобы числа выпадали именно *случайно*, без всякой закономерности, ибо всякая закономерность может быть подмечена внимательным игроком. Такой случай описал Джек Лондон в рассказе *Смок и Малыш*, где героям удалось выиграть неплохую сумму денег на рулетке, которая стояла рядом с печкой и потому со временем рассохлась. Колесо крутилось неравномерно, и одни номера выпадали чаще других.

На самом деле эта история - не вымысел автора. Английский инженер Джозеф Джаггерс в 1973 году с помощниками изучил поведение рулеток одного из казино в Монте-Карло и определил, что на одном из них числа выпадают неравномерно. За четыре дня он выиграл несколько сотен тысяч долларов, после чего руководство казино догадалось проверить колесо этой рулетки. С тех пор «система Джаггерса» не действует, так как во всех казино строго следят за исправностью рулеток и время от времени опять же случайным образом меняют местами колёса.

В романе Фёдора Михайловича Достоевского *Игрок* описаны все перипетии игры на рулетке.

Метод Монте-Карло разработали в 1948 году американские математики Джон Нейман и Станислав Улам. Кстати говоря, и раньше некоторые задачи решали с помощью случайных выборок, но до появления электронно-вычислительных машин применение этого метода было очень трудоёмким.

Теперь же в любом языке программирования имеется функция, генерирующая случайные числа (точнее, *псевдослучайные*, поскольку они вычисляются по формуле). Есть такой «генератор» и в *паскале* – это функция *Random* стандартного модуля *PABCSystem*.

Теперь мы знаем достаточно для того, чтобы приступить к написанию программы.

Как мы и договорились, в забегах будут принимать участие 6 тараканов, которых мы раскрасим в разные ***цвета***:

```
//ПРОГРАММА, МОДЕЛИРУЮЩАЯ ТАРАКАНЬ БЕГА

uses
   GraphABC, ABCButtons, ABCObjects, System.Media, Timers,
System;

const
   //число тараканов:
```

```
  N_TARAKAN=6;
  //цвета тараканов:
  COLOR_TARAKAN: array[1..N_TARAKAN] of Color = (Color.Red,
                                      Color.Yellow,
                                      Color.Blue,
                                      Color.Green,
                                      Color.SandyBrown,
                                      Color.LavenderBlush
                                      );
```

Также нам понадобятся *переменные* для хранения координат тараканов и их здоровья на момент старта:

```
var
  //размеры окна:
  height, width: integer;
  //элементы управления:
  lblTIme, lblWin: RectangleABC;
  btnStart: ButtonABC;
  _timer: Timer;
  playSound:SoundPlayer;

  //координаты тараканов:
  x: array[1..N_TARAKAN] of real;
  y: array[1..N_TARAKAN] of integer;
  //здоровье таракана:
  zdor: array[1..N_TARAKAN] of real;
  tar: array[1..N_TARAKAN] of EllipseABC;
  //время старта:
  startTime: DateTime;
  //флажки начала и окончания забега:
  flgStart, flgFinish: boolean;
  //координаты стартовой позиции:
  xStart:= 10;
  yStart:= 60;
  xFinish: integer;
  //расстояние между дорожками по вертикали:
  dy:= 32;
  //размеры тараканов:
  wTar:= 32;
  hTar:= 16;
```

Размеры *окна* вы можете сделать и больше, чтобы продлить удовольствие:

```
//
// ОСНОВНАЯ ПРОГРАММА
//
begin
   SetWindowTitle('Тараканьи бега');
   SetWindowWidth(640);
   Window.Height:= 480;
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(RGB($2F,$4F,$4F));
   height := Window.Height;
   width := Window.Width;
```

Дополним интерфейс программы *элементами управления*. Каждые 10 миллисекунд (0,01 с) будет срабатывать *таймер*:

```
   _timer:= new Timer(10, OnTick);
```

После финиша раздастся гром аплодисметов:

```
  playSound:= new SoundPlayer('Победа.wav');
```

Время, прошедшее со старта, мы будем выводить в *метку lblTime* в процедуре-обработчике таймера *OnTick*:

```
//Отсчитываем время гонок:
procedure OnTick;
begin
   var t:= DateTime.Now - startTime;
   var s:= 'Время: ' + t.Seconds.ToString() + '.' +
t.Milliseconds.ToString() ;
   lblTIme.Text:= s;
End;
```

Обратите внимание, что для точного подсчёта времени мы используем свойство *Now* структуры (записи) *DateTime*.

Тараканы помчатся вперёд после нажатия на *кнопку Начать забег!*:

```
  //КНОПКА

   //Начать забег:
   btnStart := new ButtonABC(10, Height-40, 120, 30, 'Начать
забег!', clMoneyGreen);
   btnStart.FontColor:= Color.Red;
   btnStart.FontStyle:= fsBold;
   btnStart.TextScale:= 0.9;
   //процедура-обработчик:
   btnStart.OnClick := btnStart_OnClick;
```

Для вывода информации мы воспользуемся услугами двух *меток*. Первая будет показывать время, вторая – результаты забега:

```
  //МЕТКИ

   //Время забега:
   lblTIme:= new RectangleABC(340, 10, 140,26, Color.White);
   lblTIme.FontStyle:= fsBold;
   lblTIme.Visible:= false;

   //Победитель:
   lblWin:= new RectangleABC(10, 300, width-20,26, Color.White);
   lblWin.FontStyle:= fsBold;
   lblWin.Visible:= false;
```

Перед первым стартом мы сбрасываем флажки и расставляем тараканов на стартовой позиции:

```
  flgStart:= false;
   flgFinish := false;
   prepare();
```

В процедуре *prepare* мы создаём новых тараканов заданного цвета и размера. В нашей модели роль тараканов будут исполнять эллипсы. Почему эллипсы? – Во-первых, эллипсы куда приятнее для глаз, чем тараканы. Во-вторых, эллипсы более универсальны: под ними можно понимать и тараканов, и крыс, и жеребцов, что для нашей компьютерной модели тоже является плюсом. Ну и наконец, вы очень просто можете заменить эллипсы любыми

картинками (хотя бы своих одноклассников), если загрузите их из файла, и объект *EllipseABC* замените объектом *PictureABC.*

Здесь же ветврач оценивает боеготовность каждого таракана и делает соответствующие записи в массиве *zdor*.

Для большего правдоподобия мы проводим *две черты – стартовую* и *финишную*, чтобы ни у одного таракана даже не возникла мысль о фальстарте!

```
//Готовим тараканов к старту
procedure prepare;
begin
   //начинаем новый забег:
   flgFinish := false;
   lblTime.Text:= ' Время 0';
   For var i:= 1 to N_TARAKAN do begin
      //цвет таракана:
      var clr:= COLOR_TARAKAN[i];
      SetBrushColor(clr);
      SetPenColor(clr);

      //создаем новых тараканов
      //и расставляем их у стартовой черты:
      y[i]:= yStart + dy * (i-1);
      x[i]:= xStart;
      tar[i]:= new EllipseABC(Floor(x[i]),  y[i], wTar, hTar,
clr);
      //здоровье тараканов:
      zdor[i]:= (PABCSystem.Random(30)+1)/100 + 1.2;
   end;//For
   //чертим линию старта:
   SetPenWidth(2);
   SetPenColor(Color.Red);
   Line(xStart+wTar,yStart, xStart+wTar, y[N_TARAKAN]+ hTar);

   //чертим линию финиша:
   SetPenColor(Color.LightGreen);
   xFinish:= width-wTar;
   Line(xFinish, yStart, xFinish, y[N_TARAKAN]+ hTar);
   //прячем иноформационное окно:
   lblWin.Visible:= false;
```

```
end;
```

Тараканы на старте – можно начинать *игровой цикл*:

```
//игровой цикл
procedure game;
begin
   While (not flgFinish) do begin
      //нажата кнопка "Начать забег!":
      If not flgStart then exit;
      //вычисляем новые координаты тараканов:
      calcNewCoords();
      //и перемещаем их туда:
      For var i:= 1 to N_TARAKAN do
         tar[i].MoveTo(Floor(x[i]), y[i]);

      //устанавливаем скорость работы программы:
      Sleep(20);
      //проверяем, не финишировал ли таракан:
      testFinish();
   end;//While
end;
```

Он будет продолжаться до тех пор, пока один из тараканов не достигнет финиша – тогда флаг *flgFinish* будет установлен в *True*.

Однако тараканы будут послушно стоять на старте, если флаг *flgStart* равен *False*. Как вы помните, в начале программы мы задали ему именно это значение. А вот когда пользователь или другой естествоиспытатель нажмёт кнопку *Начать забег!*, ситуация резко изменится:

```
procedure btnStart_OnClick;
begin
   btnStart.Visible:= false;
   lblTime.Visible:= true;
   //обнуляем время:
   lblTime.Text:= 'Время: ' + 0.ToString();
   //запускаем таймер:
   _timer.Start();
   //засекаем время старта по системным часам:
   startTime:= DateTime.Now;
```

```
    //стреляем из стартового пистолета:
    flgStart:= True;
    game();
end;
```

В процедуре-обработчике *btnStart_OnClick* мы готовим к бою секундомер и понуждаем тараканов к бегу громким звуком.

Условие *flgStart = True* в игровом цикле выполнено, и тараканы помчались вперёд. В процедуре *calcNewCoords* мы определяем текущее положение каждого таракана.

Поскольку тараканы обязаны бежать *неравномерно*, то каждый раз мы *случайно* задаём каждому таракану то расстояние, которое он должен пробежать в данный момент. Обратите внимание на то, что мы учитываем и физическое состояние каждого таракана. Недомогающие особи будут медленнее шевелить ногами:

```
//Вычисляем новые координаты тараканов
procedure calcNewCoords;
begin
    For var i:= 1 to N_TARAKAN do begin
        //случайное перемещение:
        var dx:= (PABCSystem.Random(43)+1)/10 + 0.9;
        //новое положение таракана на дистанции:
        x[i] := x[i] + dx*zdor[i];
    end;//For
end;
```

Переместив всех тараканов в их новое положение, мы должны проверить, не пересёк ли хотя бы один из них финишную черту, иначе они один за другим скроются за пределами окна приложения, так и не выявив победителя. В том, что рано или поздно тараканы доползут до финиша, сомневаться не приходится, поскольку они бегут только вперёд.

И нам осталось рассмотреть последнюю процедуру - *testFinish*, в которой мы определяем, закончился ли забег, и если закончился, то какой таракан пришел первым. Здесь важно учесть, что финишировать могут и *несколько* тараканов. Тогда придётся прово-

дить «фотофиниш»: победителем будет признан тот таракан, который убежал дальше за линию финиша на момент проверки:

```
// Проверям финиш
procedure testFinish;
begin
   var xEnd:=0.0;
   var n:=0;
   flgFinish:= false;
   For var i:= 1 to N_TARAKAN do begin
       if (x[i] >= xFinish-wTar) Then begin
          flgFinish:= true;
          _timer.Stop();
          If (x[i] > xEnd) then begin
             //определяем номер победителя:
             n:= i;
             xEnd:= x[i];
         End;//If
      end;//if
   end;//For
   If (flgFinish= true) then begin
       playSound.Play();
       var s:= ' Победил таракан # ' + n.ToString() + '!';
       var t:= DateTime.Now - startTime;
       s:= s + ' Его время: ' + t.Seconds.ToString() + '.' +
t.Milliseconds.ToString();
       flgStart:= false;
       lblWin.Text:= s;
       lblWin.Visible:= true;

       //задержка:
       Sleep(6000);
       //уничтожаем старых тараканов:
       For var i:= 1 to N_TARAKAN do
           tar[i].Destroy;
       //готовимся к новой игре:
       prepare();
       btnStart.Visible:= true;
  end;//If
end;
```

После финиша следует объявление чемпиона, оглашение победного времени (Рис. 37.4), аплодисменты и народные гуляния. За-

тем все болельщики, пьяные от счастья и шампанского, вновь запускают свежих тараканов по скользкому пути азарта.

Рис. 37.4. Делайте ставки, господа!

Кстати говоря, наша компьютерная модель, несмотря на свою простоту, вполне адекватно отражает процесс тараканьей беготни, за которой наблюдать весьма любопытно. А если добавить к игре ещё и возможность делать ставки, то и на ипподром ходить не надо!

Исходный код программы находится в папке **Тараканьи бега.**

Электронная гадалка

Прошедший в 2010 году Чемпионат мира по футболу запомнился многим болельщикам и просто зрителям не только отвратительными африканскими дуделками (вувузелами), но и блестящими предсказаниями результатов матчей, которые делал немецкий *осьминог Пауль* (Рис. 37.5).

Рис. 37.5. Карикатурная обложка журнала *Тайм*, который признал Пауля лучшим осьминогом года

Он угадал результаты всех семи игр сборной Германии, а также финала. Итого - 8 игр. Для гадания ему предлагали два прозрачных ящичка с флагами стран-участниц очередного матча, из которых он выбирал один – открывал его щупальцами.

Поскольку встречи в группах могли закончиться и вничью, то само гадание было не совсем корректным. Впрочем, все гадания прошли успешно, потому что сборная Германии ни разу вничью не сыграла.

В Испании, чья футбольная команда завоевала Кубок мира, сделали бронзовую статую осьминога и подарили её океанариуму немецкого города Оберхаузена. Также он стал почётным гражданином испанского города Карбальино, а американец Пэрри Грипп посвятил ему песню *Осьминог Пауль*. В самой Германии, правда, из него хотели сварить суп, но, в конце концов, его простили, поскольку он выступил лучше сборной Германии, и отправили на заслуженный отдых.

Вот так осьминог Пауль *благополучно* завершил свою карьеру предсказателя, что с предсказателями бывает нечасто.

Однако давайте подсчитаем, так ли уж велики заслуги этого головастого головоногого моллюска перед мировым сообществом. Если в каждом матче выигрывает одна из двух команд, то вероятность правильного предсказания результата одной игры равна ½, двух – $(½)^2 = ¼$, а всех восьми - $(½)^8 = 1/256$. То есть из 256 осьминогов один почти наверняка угадал бы все результаты. Так что подвиг Пауля не столь уж значителен, особенно если учесть, что некоторые люди выигрывают и в Спортлото (вероятность угадывания 5 чисел из 36 равна 1/45 239 040, а 6 чисел из 49 - 1/10 068 347 520).

Поскольку результаты футбольных матчей в большой мере случайны, а учесть все факторы невозможно, то мы будем вслед за Паулем полагать, что у каждой команды *равные* шансы на победу.

И нам осталось написать простенькую программу **Осьминог Пауль**, которая моделирует простейшего *электронного оракула*.

Для ввода названий команд нам потребуются два *текстовых поля* и столько же строковых *переменных*:

```
//ПРОГРАММА ЭЛЕКТРОННЫЙ ОРАКУЛ
```

```
//Приложение Windows Forms
#apptype windows
#reference 'System.Windows.Forms.dll'
#reference 'System.Drawing.dll'

uses
  System,
  System.Windows.Forms,
  System.Drawing;

var
   //названия команд:
   team1, team2: string;

   frmMain: Form;
   btnStart: Button;
   txtTeam1, txtTeam2, txtInfo: TextBox;
   lblTeam1, lblTeam2: System.Windows.Forms.Label;
   //размеры окна:
   width, height: integer;

//===========================================
//          ОСНОВНАЯ ПРОГРАММА
//===========================================
begin
   frmMain := new Form;
   frmMain.Text := 'Осьминог Пауль';
   frmMain.Width:= 440+26;
   frmMain.Height:= 270+40;
   frmMain.StartPosition:= FormStartPosition.CenterScreen;
   frmMain.FormBorderStyle := Sys-
tem.Windows.Forms.FormBorderStyle.FixedSingle;
   frmMain.BackgroundImage:= Image.FromFile('paul.jpg');
   width:= frmMain. Width-20;
   height:= frmMain.Height;
```

Отдавая должное нашему герою Паулю, мы поместим его фото на задний план окна (Рис. 37.6).

Перейдем к *элементам управления*. Запишите рецепт: возьмите одну *кнопку,* три *текстовых поля* и две *метки*. Разместите их в окне по вкусу. Например, так, как показано на Рис. 37.6:

```
  //текстовые поля:
   txtTeam1:= new TextBox();
   txtTeam1.Width:= 140;
   txtTeam1.Left:= 10;
   txtTeam1.Top:= 10;
   txtTeam1.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   frmMain.Controls.Add(txtTeam1);

   txtTeam2:= new TextBox();
   txtTeam2.Width:= 140;
   txtTeam2.Left:= 10;
   txtTeam2.Top:= 48;
   txtTeam2.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   frmMain.Controls.Add(txtTeam2);

   txtInfo:= new TextBox();
   txtInfo.Multiline:= true;
   txtInfo.Height:= 50;
   txtInfo.Width:= 300;
   txtInfo.Left:= 10;
   txtInfo.Top:= 180;
   txtInfo.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   txtInfo.ForeColor := Color.Red;
   //txtInfo.ScrollBars := Sys-
tem.Windows.Forms.ScrollBars.Both;
   txtInfo.WordWrap:= false;
   frmMain.Controls.Add(txtInfo);

   //метки:
   lblTeam1:= new System.Windows.Forms.Label();
   lblTeam1.Left:= 160;
   lblTeam1.Top:= 10;
   lblTeam1.AutoSize:= true;
   lblTeam1.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   lblTeam1.ForeColor := Color.LightGreen;
   lblTeam1.BackColor := Color.Transparent;
   lblTeam1.Text:= '< Первая команда';
   frmMain.Controls.Add(lblTeam1);

   lblTeam2:= new System.Windows.Forms.Label();
```

```
   lblTeam2.Left:= 160;
   lblTeam2.Top:= 48;
   lblTeam2.AutoSize:= true;
   lblTeam2.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   lblTeam2.ForeColor := Color.LightGreen;
   lblTeam2.BackColor := Color.Transparent;
   lblTeam2.Text:= '< Вторая команда';
   frmMain.Controls.Add(lblTeam2);

   //кнопки:
   btnStart := new Button;
   btnStart.Text := 'ПРОГНОЗ!';
   btnStart.AutoSize := True;
   btnStart.Left := 10;
   btnStart.Top := height-70;
   btnStart.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   btnStart.ForeColor := Color.Red;
   frmMain.Controls.Add(btnStart);
   btnStart.Click += btnStart_Click;

   Application.Run(frmMain);
end.
```

Нажимаем кнопку – и получаем предсказание на интересующую нас игру:

```
//РЕЗУЛЬТАТ ГАДАНИЯ
procedure btnStart_Click(sender: Object; e: EventArgs);
begin
   var s1:= txtTeam1.Text;
   var s2:= txtTeam2.Text;
   var s: string;
   var rand:= new Random();
   If (s1='') Or (s2='') Then
       s:= ' Введите названия команд!'
   Else If (team1=s1) And (team2=s2) Then
       s:= ' Повторно не гадаю!'
   Else If (s1=s2) Then
       s:= ' Одна и та же команда!'
   Else begin
       team1:=s1;
       team2:=s2;
```

```
        If (rand.Next(2)=1) Then
            s:= ' Победит ' + team1
        Else
            s:= ' Победит ' + team2
     End;

  s:= s + NewLine + ' Давайте погадаю ещё раз!';
  txtInfo.Text:= s;
end;
```

Б*о*льшую часть процедуры занимают проверки ввода и беседа с пользователем, но зато так наш предсказатель выглядит более мудрым. А само предсказание занимает пару строк и основано на методе *Next* класса *Random*.

Рис. 37.6. И наш электронный Пауль – парень не промах!

Конечно, наша математическая модель слишком проста, но её можно улучшить, если вычислять вероятность исхода матча с учетом рейтинга ФИФА для сборных команд. Для гаданий на Чемпионат России, например, вполне уместно оценивать положение команд в турнирной таблице и статистику личных встреч. Но сколько бы мы факторов ни учитывали, а всё равно не угадаем!

Исходный код программы находится в папке **Осьминог Пауль**.

Число *пи*, или Метод научного тыка

А теперь давайте рассмотрим пример более серьёзного применения метода Монте-Карло.

Пусть нам нужно вычислить площадь S какой-либо криволинейной фигуры, которая задана графически (нарисована) или аналитически (формула).

Впишем её в квадрат (Рис. 37.7) со сторонами L сантиметров (или миллиметров, или метров) и начнём совершенно случайно «тыкать» пальцем так, чтобы все «тыки» приходились в квадрат (можно тыкать и мимо, но тогда неудачные тыки мы просто не учитываем).

Поскольку интересующая нас фигура целиком и полностью лежит *внутри* квадрата, то ей тоже достанется. И вероятность того, что тык попадёт именно в фигуру прямо пропорциональна её площади.

Если мы тыкнули N раз (**красные** и **жёлтые** точки на Рис. 37.7, справа), а в фигуру угодили P раз (**красные** точки там же), то мы можем вычислить площадь фигуры так. Площадь квадрата равна L^2, а нашей фигуры – S, значит:

$L^2 / S = N / P$, откуда находим S:

$$S = L^2 * P / N \qquad (1)$$

Нетрудно понять, что при малом количестве тыков точность вычислений по формуле (1) будет невелика, однако, как показывает практика, при больших N погрешность вычислений уменьшается.

Естественно, тысячи раз тыкать пальцем (а лучше карандашом) в чертёж, да ещё при этом надеяться на *случайное* распределение тыков по площади квадрата, было бы неразумно, поэтому мы поступим правильнее, если научим этому нехитрому занятию компьютер. Он всё сделает быстро и аккуратно.

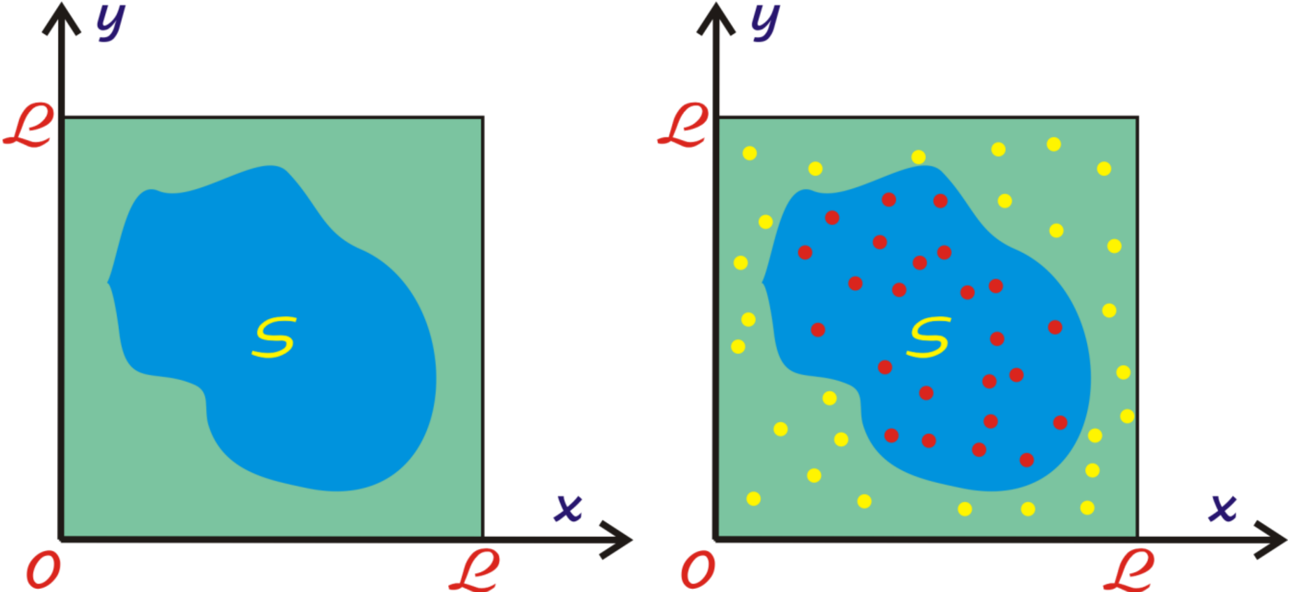


Рис. 37.7. Метод научного тыка в действии

Попробуем этим способом определить число *пи*. Мы знаем, что площадь любой плоской фигуры прямо пропорциональна квадрату её характерного размера. Для собственно квадрата это длина стороны, а для круга – *диаметр*. Коэффициент пропорциональности для квадрата равен единице, а для круга – числу *пи*, значение которого мы пока не знаем.

Нанесём по квадрату и вписанному в него кругу *N* случайных выстрелов, из которых *P* попадут в круг (Рис. 37.8).

Отношение чисел *P* и *N* пропорционально площадям фигур:

$$\frac{P}{N} = \frac{\pi D^2}{4D^2}$$

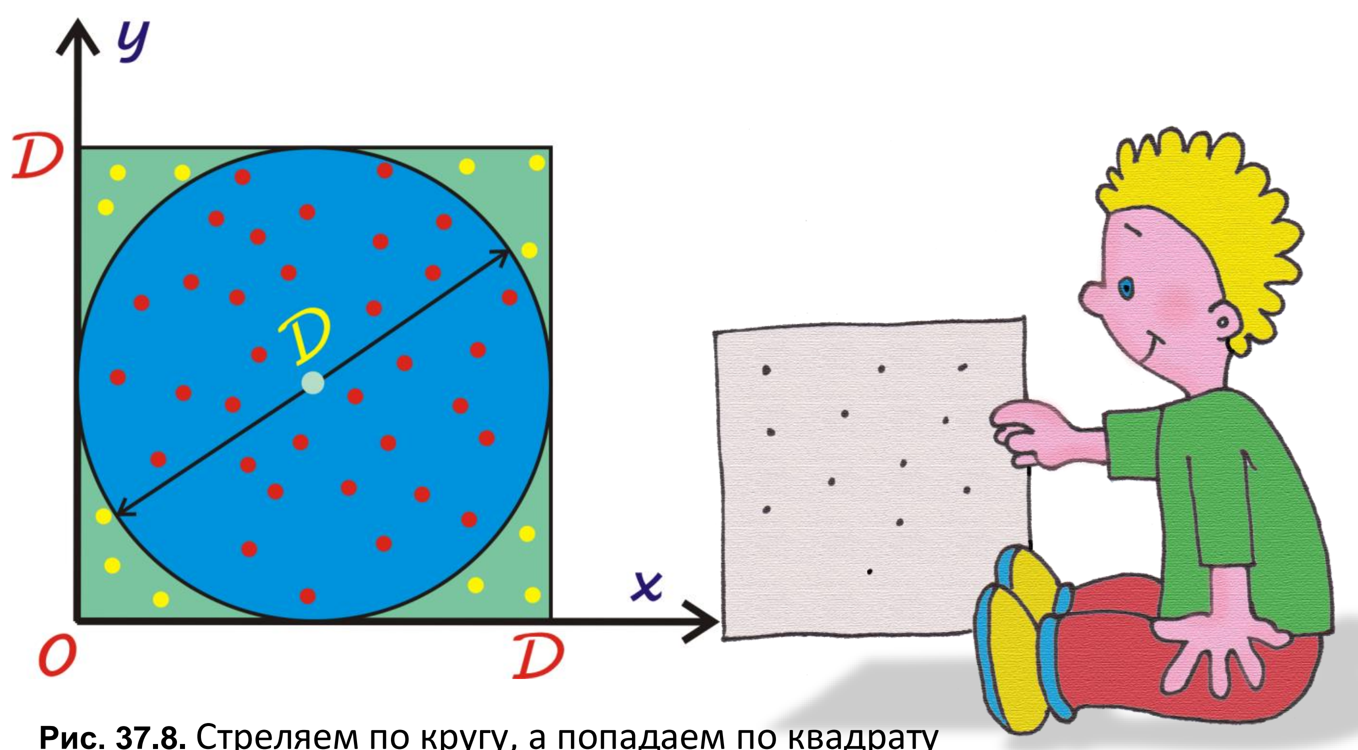


Рис. 37.8. Стреляем по кругу, а попадаем по квадрату

Из этой формулы мы легко найдем *пи*:

$$\pi = \frac{4P}{N}$$

Новую программу **Метод Монте-Карло**, как обычно, мы начнём с объявления *переменных*. После теоретической подготовки здесь трудностей у нас быть не должно:

```
//ВЫЧИСЛЕНИЕ ПИ МЕТОДОМ МОНТЕ-КАРЛО

uses
   GraphABC, ABCButtons, ABCObjects, Timers;

var
  //размеры окна:
  height, width: integer;
  //элементы управления:
  lblPi: RectangleABC;
  btnStart: ButtonABC;
  _timer: Timer;

  //смещение фигур от левого верхнего края окна:
  dx:=10;
```

```
  dy:=50;
  // общее число точек:
  N:=20000;
  //Число выстрелов:
  shot: integer;
  //число точек, попавших в круг:
  P:=0;
  //диаметр круга:
  D:=400;
  D2:= D*D;
  //радиус круга:
  R:= D div 2;
  //центр круга:
  xc:=0;
  yc:=0;
```

Задаём разумные *размеры* окна и фигур, хотя в нашем случае размер имеет значение – чем больше, тем лучше!

```
//
// ОСНОВНАЯ ПРОГРАММА
//
begin
   SetWindowTitle('Метод Монте-Карло');
   SetWindowWidth(420);
   Window.Height:= 460;
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.MidnightBlue);
   height := Window.Height;
   width := Window.Width;
```

Из *элементов управления* нам понадобится *кнопка* - чтобы начать процесс, *метка* - для вывода текущего значения числа *пи* и *таймер*, который и обеспечит нас актуальной информацией о ходе эксперимента:

```
  //КНОПКА
   //Начать эксперимент:
   btnStart := new ButtonABC(10, 10, 100, 30, 'Начать!',
clMoneyGreen);
   btnStart.FontColor:= Color.Red;
   btnStart.FontStyle:= fsBold;
```

```
   btnStart.TextScale:= 0.9;
   //процедура-обработчик:
   btnStart.OnClick := btnStart_OnClick;

   //МЕТКА
   //Пи:
   lblPi:= new RectangleABC(140, 10, 200,26, Color.White);
   lblPi.FontStyle:= fsBold;

   _timer:= new Timer(10, OnTick);
```

Рисуем на канве героев нашего эксперимента – *квадрат* и вписанный в него *круг*:

```
  //рисуем квадрат:
   SetBrushColor(Color.Green);
   FillRectangle(dx, dy, dx+D, dy+D);
   //рисуем круг:
   SetBrushColor(Color.Blue);
   FillEllipse(dx, dy, dx+D, dy+D);
   //вычисляем координаты центра круга:
   xc:= D div 2 + dx;
   yc:= D div 2 + dy;
end.
```

Для эксперимента всё готово. Без тени сомнения нажимаем кнопку *Начать!* – и *процесс пошёл*, как говаривал первый Президент СССР.

Правда, со словом *начать* у него были орфоэпические проблемы.

```
procedure btnStart_OnClick;
begin
   //запускаем таймер:
   _timer.Start();
   //ставим точки:
   P:=0;
   setPoints();
end;
```

Главные события сей научной драмы разворачиваются в процедуре *setPoints*:

```
//Ставим точки
procedure setPoints;
begin
   var clr: Color;
   for var i:= 1 to N do begin
      shot:= i;
      //выбираем случайные координаты  точки:
      var x := Random(D) + dx;// - 1
      var y := Random(D) + dy;// - 1
      var xx:= xc - x;
      var yy:= yc - y;

      //попали в круг?
      If (xx*xx + yy*yy <= R*R) Then //попали
      begin
         p:=p+1;
         clr:= Color.Red;
         end
      Else //не попали
         clr:= Color.Yellow;

      //устанавливаем цвет точки:
      SetBrushColor(clr);
      //рисуем точку:
      FillEllipse(x, y, x+2, y+2)
   End;//For
   _timer.Stop;
End;
```

Чтобы проверить, попадает ли точка с координатами *(x, y)* внутрь круга или нет, достаточно по теореме Пифагора вычислить длину гипотенузы *r* прямоугольного треугольника (Рис. 37.9):

$$r = \sqrt{xx^2 + yy^2}$$

Поскольку *r >= 0*, то обе части равенства можно возвести в квадрат:

$$r^2 = xx^2 + yy^2$$

Теперь достаточно сравнить *r* с *R*. Если *r* <= *R*, то точка находится *внутри* круга, иначе – *снаружи*.

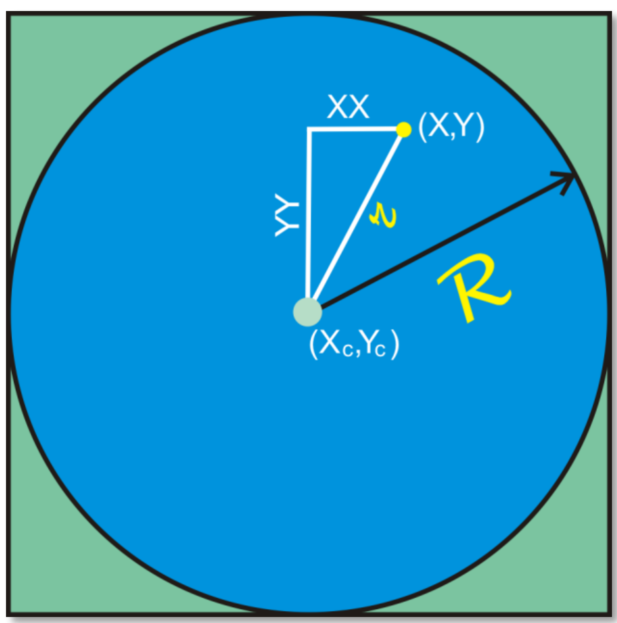


Рис. 37.9. Геометрия – это наука!

В *компьютерной* модели проверку можно проводить и иначе. Мы знаем, что круг окрашен в **синий** цвет, поэтому достаточно проверить цвет пикселя в точке с координатами *(x,y).* Если он **синий**, то точка находится внутри круга, в противном случае – снаружи.

Однако, после того как в круге появятся точки **красного** цвета, нужно будет учитывать и их. Поскольку мы окрашиваем точки вне круга в **жёлтый** цвет, то проверка будет очень простой.

Вопреки теоретическим изысканиям, наша модель не уточняет значения *пи* с увеличением числа испытаний, что хорошо видно на Рис. 37.10. Зато генератор случайных чисел в *паскале* работает совсем неплохо, поскольку на правом рисунке почти все точки закрашены (при его идеальной работе были бы закрашены все 200 000 точек, а это не так).

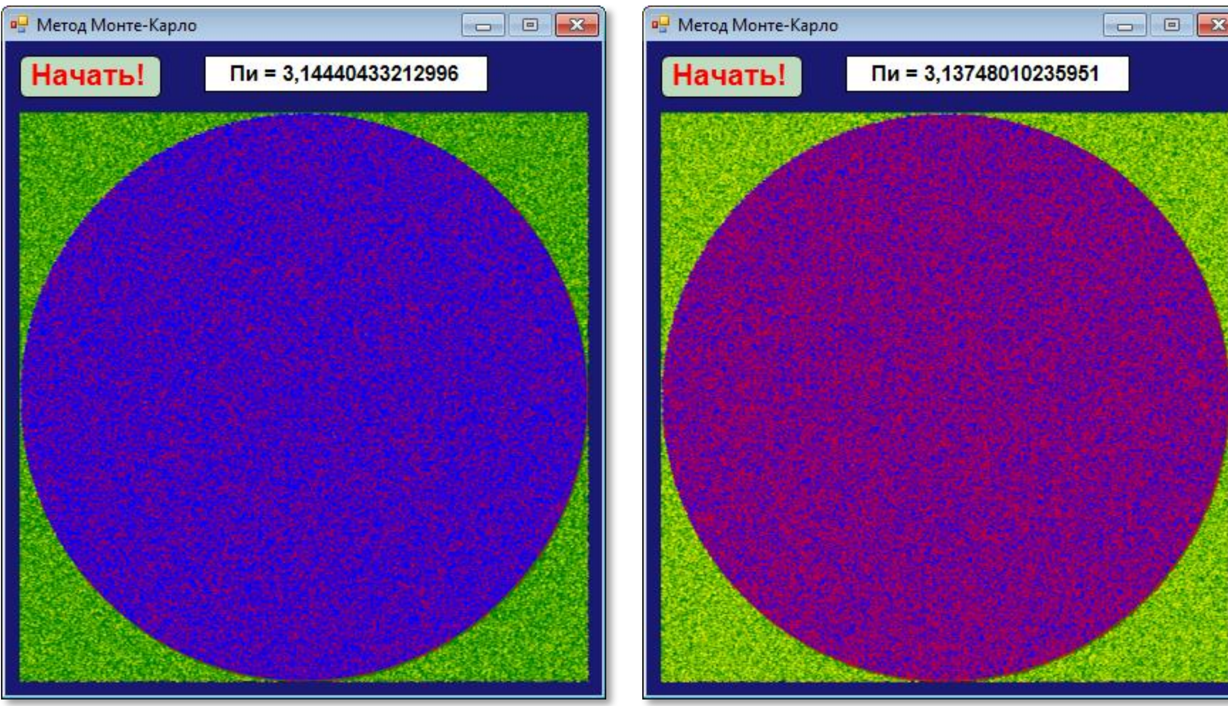


Рис. 37.10. Результаты эксперимента при N= 20 000 и N= 200 000

И хотя в *метке* мы видим множество цифр после запятой, точность вычислений ограничивается уже *вторым* знаком. Несколько улучшить ситуацию можно, если провести серию испытаний, а затем вычислить среднее значение. Принципиальный же источник ошибок здесь в том, что реальные фигуры состоят из *бесконечного* числа точек, а мы закрашиваем пиксели, которых совсем немного. Конечно, если отказаться от визуализации процесса и вычислять значение *пи* по формулам, то эта проблема будет решена. Другая проблема состоит в том, что все случайные числа, которые генерирует функция *Random*, - *целые*, а координаты точек внутри фигур скорее рациональные, чем целые. Можно генерировать последовательности цифр после запятой, чтобы получать рациональные случайные числа, но это уже детали реализации математической модели, которые не могут поколебать её основ.

Исходный код программы находится в папке **Метод Монте-Карло.**

1. Напишите программу для бросания монеты. Для этого достаточно слегка изменить проект *Осьминог Пауль*.

2. Немного труднее научить программу бросать кубик, вытаскивать карту из колоды или бочонок из мешка в игре лото.

3. Экстремальным вариантом рулетки можно считать *русскую рулетку* (иначе она называется *гусарской рулеткой*). «Играют» в неё так. В барабан шестизарядного револьвера вставляется один патрон (из соображений гуманности и здравомыслия будем считать, что *холостой*). Затем барабан прокручивается несколько раз так, чтобы «игроки» не знали его положения относительно ствола пистолета. Мы, не обсуждая все разновидности этой игры, будем считать, что после каждого спуска крючка барабан прокручивается вновь. Таким образом, вероятность выстрела будет постоянной на протяжении всей игры и равна 1/6. Соответственно, вероятность благоприятного исхода после первого выстрела равна 1 - 1/6 = 5/6 = 0,833. После второго $(5/6)^2$ = 0,694, после шестого - $(5/6)^6$ = 0,335, и так далее, приближаясь к *нулю* (вот почему так важно учить теорию вероятностей!). Математическая модель этой игры ничем не отличается от моделирования бросания игрального кубика, у которого в точности шесть граней.

4. *Слот-машина*, больше известная под названием *однорукий бандит*, представляет собой три или пять барабанов, которые вращаются независимо друг от друга, наподобие колеса рулетки. На боковую поверхность барабанов нанесены различные символы, комбинация которых после остановки колёс и определяет сумму выигрыша или проигрыша игрока, сделавшего ставку (Рис. 37.11). Справедливое прозвище этих автоматов *однорукий бандит* восходит к тому времени, когда колёса раскручивались рычагом на боковой поверхности автомата. Этот рычаг и придавал ему такой запоминающийся облик.

Сейчас больше распространены видео-слоты, которые обходятся без механических барабанов, поэтому рычаг им не нужен. Поскольку в новых машинах комбинация символов задаётся генератором псевдослучайных чисел, то достаточно просто написать программу, имитирующую их бандитскую деятельность.

Рис. 37.11. Удачная комбинация!

Урок 38. Перебор с возвратами

Мы рассмотрим сначала простую задачу: *найти все варианты расстановки восьми ладей на шахматной доске так, чтобы они не били друг друга*. Когда речь идёт о поиске *всех* вариантов, то обычно для решения задачи используют *перебор вариантов*, или *метод перебора с возвратами* (по-английски *backtracking*).

Обычная шахматная доска имеет 8 клеток в ширину и столько же в высоту, что довольно много для предварительных изысканий, поэтому мы начнём с самой маленькой доски – 1 х 1 клетку (Рис. 38.1). Ясно, что на одну клетку можно поставить *единственную* ладью, которая других ладей бить не сможет - из-за отсутствия соперников. Итак, на такую доску можно поставить одну ладью одним способом.

Рис. 38.1. Ладья на поле 1 х 1

Мы будем рассматривать только *квадратные* доски - на прямоугольных досках мы не найдём ничего нового.

Возьмём доску побольше – 2 х 2 клетки. Первую ладью можно поставить на первую горизонталь, в самую первую клетку (Рис. 38.2).

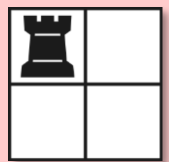


Рис. 38.2. Ладья на поле 2 х 2

Поскольку других ладей на поле нет, то никакие проверки не нужны. Из шахматных правил следует, что ладья бьёт все поля,

находящиеся с ней на одной вертикали и горизонтали. Для убедительности закрасим битые поля (Рис. 38.3).

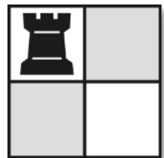


Рис. 38.3. Ладья бьёт две клетки

Теперь ясно видно, что на каждой горизонтали может стоять только *одна* ладья, потому что она держит под контролем *всю* горизонталь, в которой находится. Учтём это на будущее и попробуем поставить *вторую* ладью.

Первая горизонталь уже занята, поэтому поставим вторую ладью в первую клетку второй горизонтали. Проверка показывает, что на этой вертикали уже находится первая ладья, значит, такая позиция невозможна (Рис. 38.4).

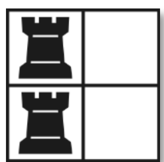


Рис. 38.4. Первая ладья бьет вторую

Передвигаем вторую ладью **вправо** (Рис. 38.5).

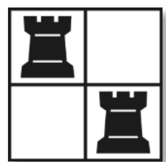


Рис. 38.5. Вторая ладья уходит из-под боя

Теперь всё нормально, обе ладьи заняли допустимые позиции, а мы нашли *первый* вариант расстановки. Но, возможно, есть и другие - ведь нам нужно найти *все* варианты. Мы не можем вторую ладью передвинуть еще раз вправо, потому что горизонталь закончилась. Нам остаётся убрать вторую ладью, вернуться назад

к первой ладье и попробовать *её* передвинуть вправо. Справа от неё располагается пустая клетка, так что такая операция возможна (Рис. 38.6).

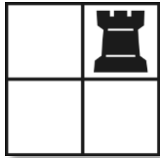


Рис. 38.6. Первая ладья идет вправо

Ищем место для второй ладьи от начала второй горизонтали. Подходит уже первая клетка (Рис. 38.7).

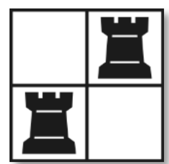


Рис. 38.7. Второе решение найдено!

Мы нашли *второй вариант* решения задачи. А есть ли ещё решения? – Для проверки снова передвигаем вторую ладью вправо (Рис. 38.8).

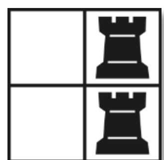


Рис. 38.8. Вторая ладья опять под боем!

В этой позиции вторая ладья оказывается под боем первой ладьи. Передвинуть её ещё вправо невозможно, поэтому снимаем вторую ладью с доски и пытаемся передвинуть первую ладью вправо. Это также невыполнимая операция. Убираем первую ладью с поля и убеждаемся, что на поле не осталось больше ни одной ладьи. Значит, мы нашли *все* расстановки двух ладей на поле 2 х 2 клетки. Всего таких расстановок *две*.

Если у вас есть желание и терпение, вы аналогично можете исследовать доску 3 x 3 клетки, но мы уже знаем *алгоритм поиска* всех решений, поэтому нам ничего не стоит перевести его на язык паскаль, а компьютер найдёт нам решения всех ладейных задач.

Начнём новый проект **Ладьи** и сразу же введём все *переменные* и *константы* программы:

```
//ПРОГРАММА ДЛЯ РАССТАНОВКИ ЛАДЕЙ

uses CRT;

const
  DIM = 8;
  //ширина поля в клетках (число вертикалей):
  COLS = DIM;
  //высота поля в клетках (число горизонталей):
  ROWS = DIM;
  //число ладей:
  NUM_LAD = DIM;

var
   //число уже выставленных ладей:
   n := 0;
   //текущая вертикаль:
   v := 0;
   //число найденных вариантов расстановки:
   nVar :=0;
   //номер вертикали, на которой стоит ладья:
   vert: array[1..ROWS] of integer;

label nextLad, nextVert, back, writeVar, finish;
```

В первую очередь, нам понадобятся константы *COLS* и *ROWS* для хранения размеров шахматного поля и константа *NUM_LAD,* в которой мы будем хранить число ладей, выставляемых на поле. В нашем примере, все они равны восьми, но вы легко можете изменить как размеры поля, так и число ладей, присвоив константам нужные значения. Например, отладку программы следует проводить на поле 2 x 2, чтобы убедиться в правильной работе программы на таком поле. Затем можно постепенно довести разме-

ры поля до 8 x 8. Так по ходу отладки мы подсчитаем варианты расстановки ладей на квадратных полях разных размеров.

Буквой *n* мы обозначим переменную для хранения номера выставляемой на поле ладьи. Во всех горизонталях должна стоять *единственная* ладья, при этом каждая из них занимает свою вертикаль. Номер вертикали, на которой стоит та или иная ладья, мы будем хранить в массиве *vert*. Например, первая ладья занимает вертикаль *vert[1]* в первой горизонтали (всегда считаем горизонтали сверху вниз!). Вторая - *vert[2]* во второй горизонтали, и так далее. Пока место для очередной ладьи не найдено, мы запоминаем номер вертикали *v*, на которую мы хотим её поставить.

И последняя переменная – *nVar* – послужит нам для подсчёта найденных вариантов.

Начинаем кодировать наш алгоритм.

В исходном положении ни одна ладья не выставлена на поле:

```
//========================================
//          ОСНОВНАЯ ПРОГРАММА
//========================================
begin
   SetWindowTitle('Ладьи');

   //ни одна ладья еще не выставлена:
   n:=0;
```

Ставим *следующую* ладью. После нулевой это будет, конечно, *первая*. Если одну ладью мы уже поставили, то следующей будет вторая, и так далее. Поскольку ладей нужно выставить несколько, то обозначим меткой *nextLad* строку кода, в которой мы выставляем следующую ладью:

```
nextLad:
   n += 1;
```

Если выставлены уже *все* ладьи, то найден очередной вариант расстановки, который нужно записать в файл или вывести на экран:

```
  If (n > NUM_LAD) Then
        Goto writeVar;
```

Каждую новую ладью мы пытаемся выставить на первую вертикаль, затем на вторую, и так до тех пор, пока не выйдем за границу поля. Как вы помните, эта операция проводится многократно для каждой ладьи, поэтому нужно запомнить начало кода для движения ладьи *вправо*, чтобы в нужный момент перейти к нему:

```
  v:=0;
nextVert:
   v += 1;
   If (v > cols) Then
      Goto back;
```

Если места для очередной ладьи не нашлось, значит, нужно вернуться *назад*, к предыдущей выставленной ладье. Эти строки кода обозначены меткой *back*.

```
  If not test() then
     Goto nextVert;

   vert[n]:= v;
   Goto nextLad;
```

Итак, очередную ладью *n* мы хотим поставить на вертикаль *v*. Есть ли у нас право сделать это, мы узнаём по результатам проверки в функции *test*. Если нет (функция *test* вернула *false*), то мы возвращаемся на строку *nextVert*, чтобы передвинуть ладью вправо. Если же проверка прошла успешно и функция *test* вернула *true*, то мы уверенно ставим ладью *n* на эту вертикаль, а само приятное для нас событие отмечаем в массиве *vert*. Далее мы возвращаемся на метку *nextLad*, чтобы выставить следующую ладью.

Найденное решение мы выводим в *консольное окно*:

```
writeVar:
   nVar := nVar + 1;
   writePos;

procedure writePos;
begin
   TextColor(LightRed);
   writeln('Вариант # ' + nVar.ToString());
   TextColor(Yellow);
   For var i:= 1 To NUM_LAD do
   begin
      For var j:= 1 to cols do
         if (vert[i] <> j) Then
            Write('.')
         Else
            Write('X');
      writeln;
   end;//For
   writeln;
   writeln;
end;
```

Для поля 4 х 4 клетки «протокол» выглядит так (Рис. 38.9).

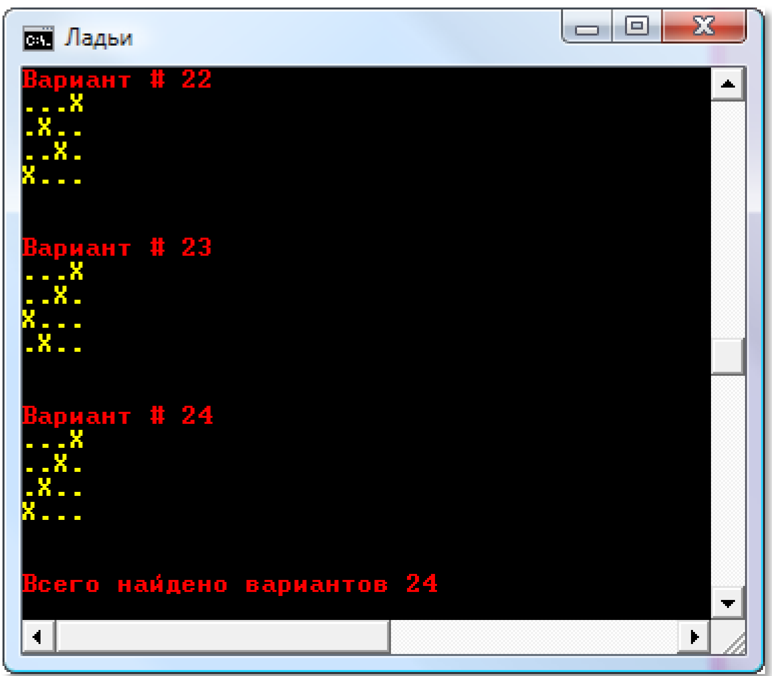


Рис. 38.9. Все решения для доски 4 х 4

Можно даже красиво нарисовать решение в *графическом окне*. Однако для больших полей решений окажется слишком много, чтобы вырисовывать каждое из них на экране. Правильнее - сохранять все решения в *файле*.

Если для очередной ладьи места на поле не нашлось, значит, нужно *вернуться назад*, к предыдущей ладье и передвинуть её вправо:

```
back:
   n -= 1;
   If (n = 0) Then
      Goto finish;

   v:= vert[n];
   Goto nextVert;
```

Как только программа вернётся к *нулевой* ладье, мы можем рапортовать о завершении работы – поиск вариантов закончен. Сообщаем пользователю число расстановок и ждём дальнейших указаний:

```
finish:
   TextColor(LightRed);
   WriteLn('Всего найдено вариантов ' + nVar.ToString());
   WriteLn;
   TextColor(Yellow);

end.
```

Нам осталось написать проверочную функцию *test*:

```
//Проверка: можно ли поставить очередную
//ладью n на текущую вертикаль v
function test(): boolean;
begin
   Result:= true;
   If (n = 1) Then
       exit;

   For var i:= 1 To n-1 do
    If (vert[i]=v) then
```

```
    begin
       Result:= false;
       exit;
    End;//If
End;
```

Она очень простая. Сначала мы присваиваем переменной *Result* значение *true* и, если выставляется *первая* ладья, то сразу же завершаем функцию, поскольку первую ладью *всегда* можно выставить на любую вертикаль. Для каждой ладьи, начиная со второй, следует проверить, не занята ли вертикаль *v* одной из предыдущих ладей. Если занята, мы возвращаем значение *false*. В противном случае мы возвращаем значение *true*.

Запустив программу при разных размерах поля, мы получим такие результаты:

Размер	**Число решений**
1 x 1	1
2 x 2	2
3 x 3	6
4 x 4	24
5 x 5	120
6 x 6	720
7 x 7	5040
8 x 8	40320

Если эти числа вам ничего не напоминают, то возвратитесь к уроку, на котором мы подсчитывали факториалы. Действительно, если размеры поля равны *n x n*, то первую ладью можно выставить на любую из *n* вертикалей, для второй останется *(n-1)* вертикаль, для третьей - *(n-2)*... и, наконец, для последней - только одна-единственная. Общее число расстановок ладей мы легко найдем, перемножив эти числа, то есть, другими словами, найдя факториал *n!* Вот так неожиданно факториал оказался связан с шахматными ладьями!

Повторяющиеся сходные действия, из которых и состоит перебор с возвратами, можно оформить в виде *рекурсии*. Программа станет насколько короче, настолько же и труднее для понимании.

Исходный код программы находится в папке **Ладьи**.

Напишите программу, которая генерирует одну *произвольную* (случайную) расстановку ладей на поле.

Урок 39. Занимательная Гауссиана

Кроме ладей, можно расставлять на шахматной доске и другие фигуры так, чтобы они не били друг друга. Самая известная из таких шахматных задач, безусловно, *головоломка о восьми ферзях*.

Впервые задача о расстановке восьми ферзей появилась в 1848 году в немецкой шахматной газете *Schachzeitung* (бьюсь об заклад, что вы никогда не догадаетесь, чт***о*** значит её название!). Автором задачи был Макс Беццель. В 1854 году в этой же газете были напечатаны 40 решений задачи. Однако ещё 21 сентября 1850 года в другой немецкой газете Франц Н***а***ук опубликовал 92 решения. Правда, сам он тогда ещё не был уверен, что это *все* возможные решения.

Впрочем, эта задача стала одной из самых известных головоломок мира только потому, что она связана с именем великого немецкого математика Карла-Фридриха Гаусса, который летом 1850 года нашел 76 решений этой задачи.

Он придумал очень интересный способ поиска решений. Прежде всего, Гаусс установил, что в каждой вертикали и горизонтали может стоять только один ферзь. Это очевидно вытекает из условия задачи. Таким образом, все решения можно представить в виде *перестановки* чисел *1 2 3 4 5 6 7 8*. Действительно, если ставить ферзей на доску, начиная с первой горизонтали, то каждый из них займет в ней место, отличное от мест других ферзей. Поскольку всего ферзей восемь, и все они стоят на разных вертикалях и горизонталях, то все решения можно записать в виде перестановки восьми чисел.

Количество перестановок восьми разных чисел нам хорошо известно: 8! = 40320. Но, в отличие от ладей, ферзи не могут занимать одну и ту же *диагональ*, поэтому из общего числа перестановок следует убрать те, при которых на одной и той же диагонали окажутся два ферзя или больше.

Гаусс придумал остроумный *перестановочный метод* для проверки диагоналей. Для простоты возьмём доску 5 х 5 клеток и любую перестановку из пяти чисел, например, *1 3 4 5 2*. Сразу и не скажешь, бьют ли ферзи друг друга по диагонали или нет, поэтому подпишем под перестановкой числа от 1 до 5 (число горизонталей доски) в прямом и в обратном порядке и вычислим *сумму* каждой вертикальной пары чисел:

1 3 4 5 2   **1 3 4 5 2**
1 2 3 4 5   **5 4 3 2 1**
2 5 7 9 7   **6 7 7 7 3**

Красным цветом отмечены *равные* суммы для каждого из двух случаев (у нас они совпали, но это необязательно).

В первой расстановке совпали *третья* и *пятая* суммы, это значит, что ферзи в соответствующих горизонталях стоят на ***восходящей*** диагонали. Во второй расстановке мы имеем три одинаковые суммы, поэтому ферзи на *третьей, четвертой* и *пятой* горизонталях расположены на ***нисходящей*** диагонали. Эту позицию легче проследить по рисунку (Рис. 39.1).

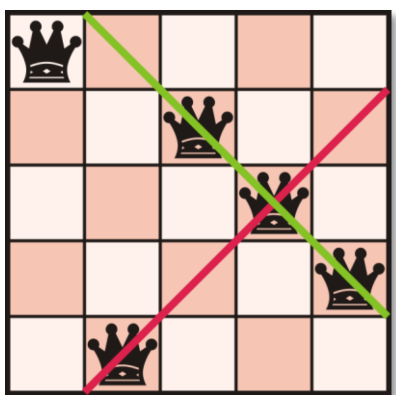


Рис. 39.1. Позиция на доске, соответствующая расстановке *1 3 4 5 2*

Как видите, способ Гаусса чисто механический. Достаточно выписать все перестановки чисел 1..8 (или 1..5, если мы хотим решить задачу на доске 5 х 5 клеток), затем подписать под ними номера горизонталей в прямом и обратном порядке, найти суммы и

сравнить их между собой. Задача осложняется только тем, что для доски 8 х 8 клеток эти действия придётся повторить несколько десятков тысяч раз.

А вот для компьютера подобные задачи – самые простые. В самом деле, если одна и та же операция совершается многократно, а сама по себе она несложная, то и программу для компьютера можно написать за несколько минут. Мы, естественно, не будем кодировать способ Гаусса, который ничего не знал о компьютерах, а просто приспособим программу для расстановки ладей под свои нужды.

Решаем задачу Гаусса

Дописав к нашей программе *Ладья* всего несколько строк, мы решим знаменитую задачу о расстановке ферзей.

Для нас ферзи отличаются от ладей только тем, что они бьют не только вертикаль и горизонталь, но и *диагонали*, проходящие через клетку с ферзем. Это легко учесть в проверочной функции *test:*

```
//Проверка: можно ли поставить очередного
//ферзя n на текущую вертикаль v
function test(): boolean;
begin
   Result:= true;
   If (n = 1) Then
       exit;

   //проверяем вертикаль:
   For var i:= 1 To n-1 do
   begin
      If (vert[i]=v) then
      begin
         Result:= false;
         exit;
      End;//If

      //проверяем диагонали:
```

```
      If (vert[i]-i = v-n) then
      begin
        Result:= false;
        exit;
      End;//If
      If (vert[i]+i = v+n) then
      begin
         Result:= false;
         exit;
      End;//If
   end;
End;
```

Проверка стала сложнее из-за того, что ферзь более «убойная» фигура, чем ладья. Нам нужно проверить, не бьёт ли новый ферзь других ферзей, выставленных ранее, не только по горизонтали и вертикали, но и по восходящей и нисходящей диагоналям. Постарайтесь разобраться, как проходит проверка, потому что в задачах нередко приходится проверять диагонали, а это несколько труднее, чем вертикали и горизонтали.

Запустив нашу программу для поиска расстановок ферзей на стандартной шахматной доске, мы быстро получим список из всех 92-х вариантов (Рис 39.2).

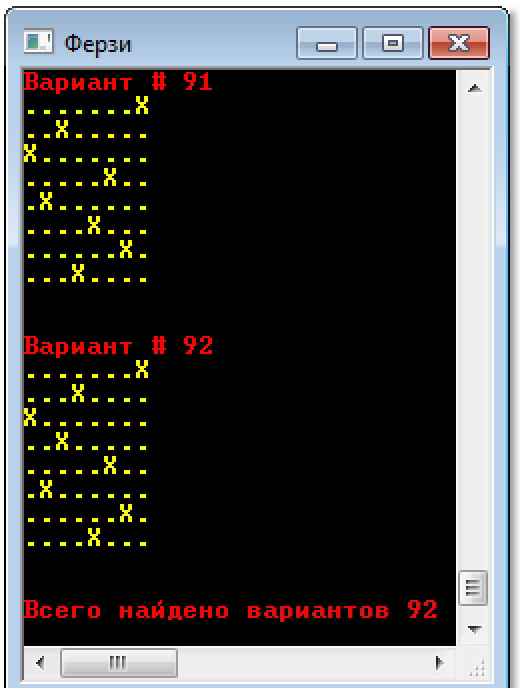


Рис. 39.2. Все расстановки ферзей получены!

Как и в случае с ладьями, вы можете исследовать доски других размеров, даже 10 x 10 клеток и больше:

Размер	Число расстановок
1 x 1	1
2 x 2	0
3 x 3	0
4 x 4	2
5 x 5	10
6 x 6	4
7 x 7	40
8 x 8	92
9 x 9	352
10 x 10	724

Как видите, для ферзей нет такой простой формулы для подсчёта числа решений, как для ладей.

Исходный код программы находится в папке **Ферзи**.

Из 92 расстановок ферзей принято выделять 12 *уникальных*, то есть таких, которые невозможно получить из других расстановок путем поворотов на 90 градусов и зеркальных отражений доски (Рис. 39.3).

Из 10-ой расстановки можно получить только 3 новых решения, из остальных – по 7. С учётом уникальных решений как раз и получаются все 92 расстановки ферзей.

Возьмём, для примера, *первую* расстановку. Поворачивая доску каждый раз на 90 градусов по часовой стрелке, мы получим ещё три новых решения (Рис. 39.4, верхний ряд). Переворачивая (отражая) доску по горизонтали и вертикали, мы получим ещё 4 новых решения (Рис. 39.4, нижний ряд). Вместе с исходным они и дадут 8 расстановок.

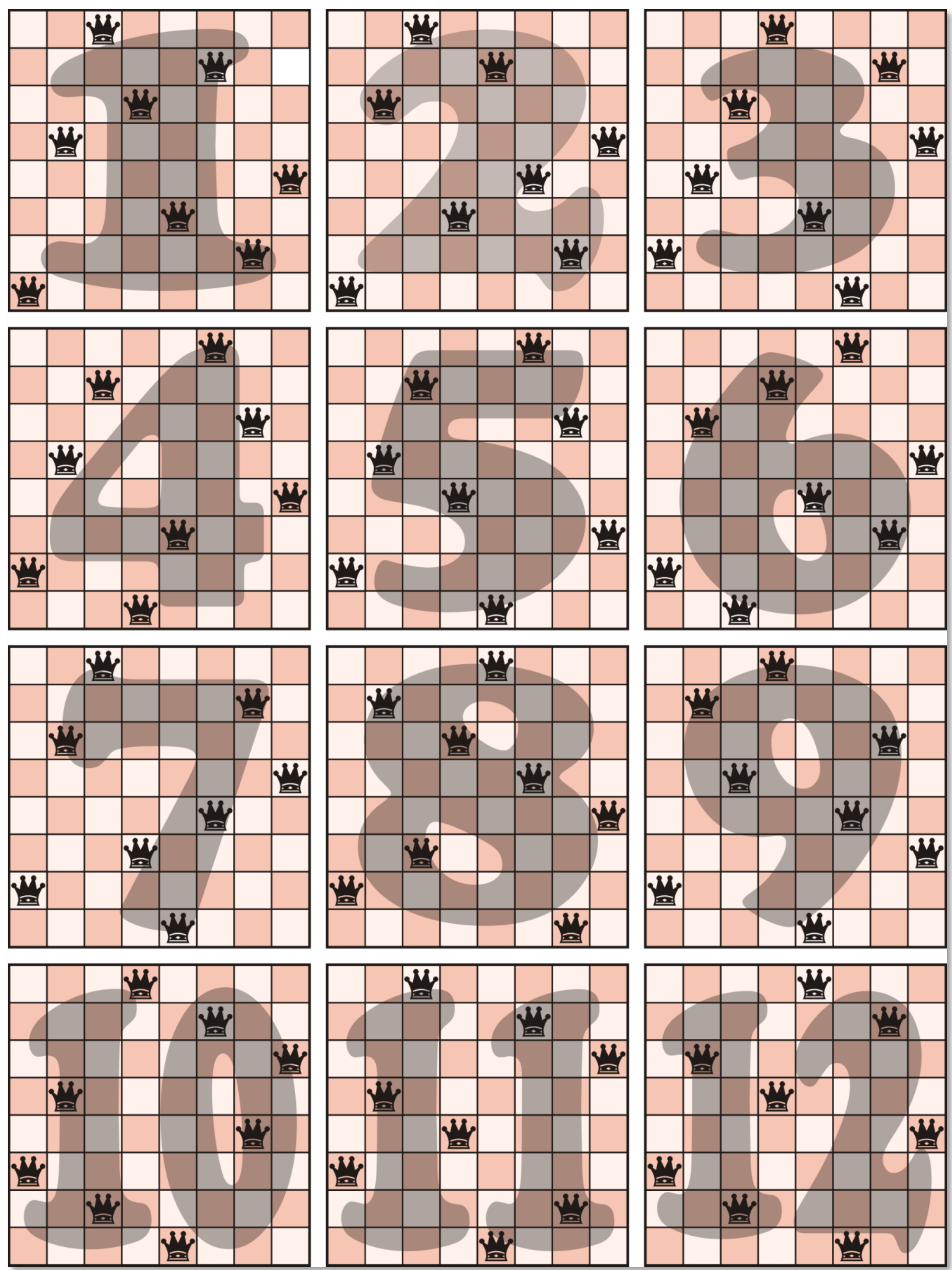


Рис. 39.3. 12 уникальных расстановок ферзей.

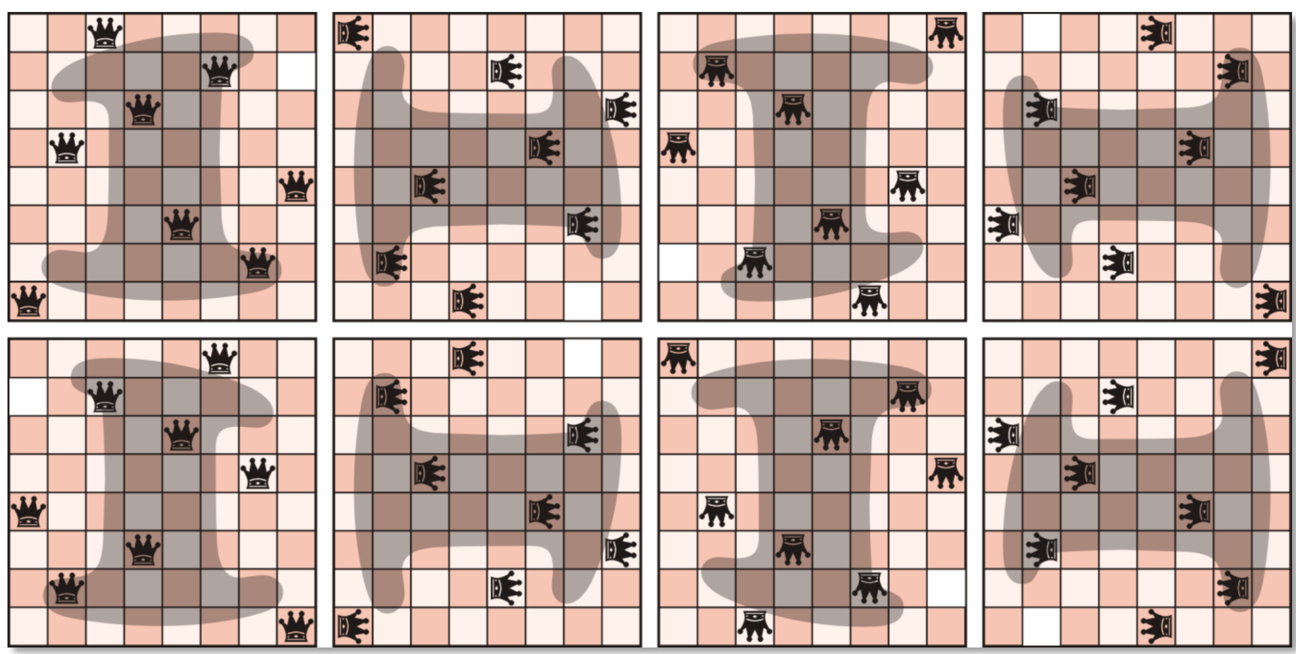


Рис. 39.4. Все расстановки ферзей, полученные из первого решения

На рисунках и ферзи, и номер уникальной расстановки оставлены в том положении, в котором они оказались после поворотов и отражений, потому что так легче проследить, как они были получены.

Исследуя все решения, английский изобретатель головоломок Генри Дьюдени нашёл, что среди 12 уникальных решений, имеется только одно, в котором ни через какую тройку ферзей нельзя провести прямую линию (естественно, не диагональ!). Это 11-ая расстановка на Рис. 39.3.

Напишите программу, которая генерирует одну *произвольную* (случайную) расстановку ферзей на поле.

Урок 40. Полный перебор

Рассматривая задачи о расстановке ладей и ферзей на шахматной доске, мы прибегли для их решения к методу *перебора с возвратами.* Этот метод требует предварительных размышлений, чтобы построить быстрый алгоритм. Однако представим себе такую ситуацию, что вам нужно решить какую-либо задачу – и забыть о ней. В жизни такие задачи встречаются очень часто. Например, вы хотите решить вот такой *числовой ребус* на сложение двух чисел, зашифрованных буквами:

```
ОДИН
 ДВА
----
АНОД
```

Вообще говоря, *числовые ребусы* (*числобусы, арифметические ребусы, криптарифмы*) предназначены для *логического* решения, но практически ни из них не решается без перебора вариантов.

В этом случае было бы неоправданной роскошью выстраивать хороший алгоритм – ведь он нам больше никогда не понадобится. Если задача это позволяет, то её следует решить **методом полного перебора**. Английское название *brute force*, что в переводе значит *грубая сила*, более эмоционально выражает суть этого метода программирования. Коли нам нужно единожды решить задачу, то решим её *самым простым* способом. Конечно, алгоритм наверняка получится очень медленным, но зато мы сэкономим время на разработке самого алгоритма. И всё бы хорошо, но нередко число всех возможных вариантов решения задачи столь велико, что экономия времени при написании программы совершенно ничтожна по сравнению со временем поиска решений, которое может достигать многих сотен лет.

По-английски метод называют также *exhaustive search - исчерпывающий поиск*.

Вернёмся к задаче о расстановке ферзей. Общее число возможных способов размещения 8 ферзей на 64-клеточной доске равно 64*63*62*61*60*59*58*57 = 178 462 987 637 760. Действительно, первого ферзя можно поставить на любое из 64 полей, для второго ферзя остается 63 поля, и так далее. Конечно, совершенно немыслимо перебирать все эти варианты, поэтому метод *полного* перебора к этой задач неприменим.

А вот числовые ребусы вполне успешно можно решать методом полного перебора. В *криптарифме* каждая цифра заменена буквой, но, поскольку цифр всего десять, то каждая буква может принимать только 10 разных значений - от 0 до 9. В любом ребусе не более десятка разных букв, то есть в худшем случае нам придётся проверить 10 000 000 000 вариантов. Современным компьютерам это вполне по силам. Впрочем, так бездумно компьютер не используют, поэтому даже при полном переборе следует разумно ограничивать число вариантов. Обычно сделать это очень просто, поскольку некоторые способы лежат на поверхности и не требуют глубоких размышлений. В случае с ферзями достаточно заметить, что на одной горизонтали может стоять только *один* ферзь, чтобы число проверяемых вариантов расстановки уменьшилось до 8*8*8*8*8*8*8*8 = 16 777 216, что совсем немного.

В ребусах также легко найти простой ограничитель числа вариантов – достаточно учесть тот очевидный факт, что все буквы должны иметь *разное* значение. Это следует из условия самой головоломки: разным буквам соответствую разные цифры. Вот теперь можно смело решать любой числовой ребус. Но мы, конечно, решим не любой, а тот самый, что озадачил нас в самом начале урока.

Программа, как это обычно и бывает при полном переборе, очень простая. Записываем столько вложенных циклов *For,* сколько

разных букв в ребусе. В нашем примере всего 6 разных букв, поэтому и циклов тоже будет шесть. Начиная со второй буквы, проверяем, чтобы её цифровое представление не совпало с предыдущими буквами. Найдя значение каждой буквы, проверяем условие:

```
1000 * O + 100 * D + 10 * I + n + 100 * D + 10 * V + A <> 1000
* A + 100 * N + 10 * O + D
```

Если оно выполняется (то есть левая часть выражения *не равна* правой), то продолжаем поиск решения, в противном случае выписываем найденное решение в *консольном окне* и ищем другие решения:

```
//ПРОГРАММА ДЛЯ РЕШЕНИЯ ЧИСЛОВОГО РЕБУСА
//МЕТОДОМ ПОЛНОГО ПЕРЕБОРА

uses CRT;

var
   //число найденных вариантов решения:
   nVar: integer;
   O, D, I, N, V, A: integer;

//Печатаем решение ребуса
procedure writeSolution;
begin
   nVar += 1;
   TextColor(LightRed);
   writeln('Вариант # ' + nVar.ToString());
   TextColor(Yellow);

   var s := '';
   s += O.ToString();
   s += D.ToString();
   s += I.ToString();
   s += N.ToString();
   writeln(s);
   s := ' ';
   s += D.ToString();
   s += V.ToString();
   s += A.ToString();
```

```
   writeln(s);
   s := '';
   s += A.ToString();
   s += N.ToString();
   s += O.ToString();
   s += D.ToString();
   writeln(s);

   writeln;
   writeln;
end;

//===========================================
//          ОСНОВНАЯ ПРОГРАММА
//===========================================
begin
   CRT.SetWindowTitle('Числовой ребус');

   nVar := 0;
   for A := 1 to 9 do
   begin

      for V := 0 to 9 do
      begin
         if (V = A) Then
            continue;

         for D := 1 to 9 do
         begin
            if (D = A) Or (D = V) Then
               continue;

            for I := 0 to 9 do
            begin
               if (I = A) Or (I = V) Or (I = D) Then
                  continue;

               for N := 0 to 9 do
               begin
                  if (N = A) Or (N = V) Or (N = D) Or (N = I)
Then
                     continue;

                  for O := 1 to 9 do
```

```
                  begin
                     if (O = A) Or (O = V) Or (O = D) Or (O =
I) Or (O = N) Then
                        continue;

                     if (1000 * O + 100 * D + 10 * I + n + 100
* D + 10 * V + A <> 1000 * A + 100 * N + 10 * O + D) Then
                        continue
                     Else
                        writeSolution();
                  end
               end
            end
         end
      end
   end;

   TextColor(LightRed);
   writeln('Всего найдено вариантов ' + nVar.ToString());
   writeln;
   TextColor(LightGreen);
end.
```

В итоге мы найдём все *четыре* решения этого ребуса (Рис. 40.1). Для учебной задачи это вполне допустимо, но правильно составленный ребус должен иметь *единственное* решение!

В программе учтено условие, что первая цифра числа не может быть нулём, поэтому циклы для букв *A, D, O* начинаются с *единицы*. Если немного подумать, то можно ещё более сузить круг поисков. Например, очевидно, что *D > 4*, а *H <> 0*. Нетрудно и дальше улучшать алгоритм, но тогда мы потеряем все преимущества метода полного перебора, который как раз для того и нужен, чтобы решить задачу без долгих раздумий.

```
Числовой ребус
Вариант # 1
1785
 732
2517

Вариант # 2
1735
 782
2517

Вариант # 3
3571
 564
4135

Вариант # 4
3561
 574
4135

Всего найдено вариантов 4

Программа завершена, нажмите любую клавишу
```

Рис. 40.1. Задача решена!

Исходный код программы находится в папке **Alphametics.**

Взлом паролей

- Пароль!
- Яблоко!
- Не яблоко, а груша! Проходи.

Анекдотичный пример взлома пароля

Некоторые задачи невозможно решить иначе, как только методом полного перебора вариантов, причем сократить перебор принципиально невозможно. Одна из таких задач – *взлом паролей.*

Мы, конечно, не будем взламывать чужих паролей. Наоборот, мы постараемся защитить свои пароли от посягательств.

Наверняка вам много раз приходилось выбирать себе пароль при регистрации на разных сайтах. Обычно для пароля разрешено использовать все 10 цифр и буквы латинского алфавита в верхнем и нижнем регистрах. Итого получается 10 + 26 + 26 = 62 символа. Из них можно составить 62 односимвольных пароля. Естественно, такой пароль разгадать очень просто. Попробуем составить пароль из двух символов: 62 х 62 = 3844 – это уже лучше, но всё равно мало, поэтому минимальная длина пароля должна составлять 6 символов, что дает уже 62^6= 56 800 235 584 варианта. Максимальная длина пароля также ограничивается – 16 символами. Поскольку 62^{16} = 4 767 240 170 682 353 345 026 330 816, то его невозможно отгадать за разумное время.

Почему же тогда хакеры так часто подбирают чужие пароли? – На самом деле пароли чаще крадут, чем взламывают. Если вы храните пароли в файле на компьютере, а антивирусная программа у вас не установлена или плохого качества, то получить доступ к вашей информации совсем нетрудно, а, значит, и ваши пароли могут украсть.

Что касается взлома паролей, то, как мы убедились, длинный пароль, составленный из *случайной* последовательности символов, разгадать нельзя. Но ведь такой пароль, - например, *A8xGzOAgm7* – и запомнить очень сложно, поэтому пользователи нередко выбирают пароли попроще – их запомнить легче.

Предположим, что некий нехороший человек, хакер Редиска решил взломать ваш сайт, то есть получить доступ к файлам на сервере провайдера, подобрав пароль. Редиска знает, что длинный хаотичный пароль от не разгадает никогда, поэтому он полагается на вашу беспечность при выборе пароля. Он очень быстро перебирает все короткие пароли.

Хакер не знает не только пароля, но чаще всего даже его длину.

Если ни один из них не подошёл, значит, ваш пароль состоит из большего числа символов. Полный перебор длинных паролей не

годится, но ведь вы вполне могли в качестве пароля выбрать какое-либо осмысленное слово. Например, свою фамилию, имя, кличку собаки, город. Наконец, просто русское слово, записанное латинскими буквами. При всем богатстве выбора так можно загадать не более миллиона паролей, которые легко перебрать в считанные секунды.

Естественно, хакер должен иметь список подходящих слов, но его несложно составить, просеяв тексты в Интернете.

Итак, знание метода полного перебора вариантов помогает нам сделать правильный выбор пароля: он должен состоять не менее чем из шести символов, взятых в случайном порядке. Другой хороший способ выбора пароля – шифрование осмысленных слов одним из методов, рассмотренных нами на уроке *Занимательная криптография*. Тогда и пароль вы легко запомните, и разгадать его будет невозможно.

Естественно, для серьёзного шифрования наши методы не годятся!

А теперь попробуйте составить несколько паролей, беря наугад произвольные буквы и цифры. Довольно утомительное занятие! Все неприятные дела следует поручать компьютеру, поэтому мы сейчас научим его составлять пароли.

Откройте любой проект, в котором есть кнопки и текстовые поля и запишите его в папку **Пароль**.

Так как программа очень простая, то нам понадобятся две *константы* и одна *переменная*. В строке *SYMBOLS* мы будем хранить все символы, из которых можно составлять пароли. Если хотите, можете добавить в неё и что-нибудь от себя.

```
//ПРОГРАММА ДЛЯ ГЕНЕРИРОВАНИЯ ПАРОЛЕЙ

//Приложение Windows Forms
#apptype windows
#reference 'System.Windows.Forms.dll'
```

```
#reference 'System.Drawing.dll'

uses
  System,
  System.Windows.Forms,
  System.Drawing;

const
   SYMBOLS =
'1234567890abcdefghijklmnopqrstuvxyzABCDEFGHIJKLMNOPQRSTUVXYZ';
   MAX_LEN= 16; //    макс. длина пароля

var
   //пароль:
   pass: string;

   frmMain: Form;
   btnGen: Button;
   txtLen, txtPass: TextBox;
   lblInfo, lblPass: System.Windows.Forms.Label;
   //размеры окна:
   width, height: integer;

//===========================================
//          ОСНОВНАЯ ПРОГРАММА
//===========================================
begin
   frmMain := new Form;
   frmMain.Text := 'Генератор паролей';
   frmMain.Width:= 320+26;
   frmMain.Height:= 140+40;
   frmMain.StartPosition:= FormStartPosition.CenterScreen;
   frmMain.FormBorderStyle := Sys-
tem.Windows.Forms.FormBorderStyle.FixedSingle;
   frmMain.BackColor := Color.Black;
   width:= frmMain. Width;
   height:= frmMain.Height;
```

Для удобства пользования программой мы украсим её интерфейс *элементами управления – кнопкой,* двумя *текстовыми полями* и двумя *метками*:

```
  //текстовые поля
```

```
   //длина пароля:
   txtLen:= new TextBox();
   txtLen.Width:= 30;
   txtLen.Left:= 10;
   txtLen.Top:= 10;
   txtLen.Text:= '6';
   txtLen.Font:= new System.Drawing.Font('Arial', 12,
System.Drawing.FontStyle.Bold);
   frmMain.Controls.Add(txtLen);

   //пароль:
   txtPass:= new TextBox();
   txtPass.Width:= 180;
   txtPass.Left:= 10;
   txtPass.Top:= 48;
   txtPass.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   txtPass.Text:= '***************';
   frmMain.Controls.Add(txtPass);

   //метки:
   lblInfo:= new System.Windows.Forms.Label();
   lblInfo.Left:= 70;
   lblInfo.Top:= 12;
   lblInfo.AutoSize:= true;
   lblInfo.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   lblInfo.ForeColor := Color.Green;
   lblInfo.Text:= ' Введите длину пароля 1..' +
MAX_LEN.ToString();
   frmMain.Controls.Add(lblInfo);

   lblPass:= new System.Windows.Forms.Label();
   lblPass.Left:= 200;
   lblPass.Top:= 48;
   lblPass.AutoSize:= true;
   lblPass.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   lblPass.ForeColor := Color.Green;
   lblPass.Text:= '< Пароль';
   frmMain.Controls.Add(lblPass);

   //кнопка:
```

```
   btnGen := new Button;
   btnGen.Text := 'Составить!';
   btnGen.AutoSize := True;
   btnGen.Left := 10;
   btnGen.Top := 100;
   btnGen.Font:= new System.Drawing.Font('Arial', 12, Sys-
tem.Drawing.FontStyle.Bold);
   btnGen.ForeColor := Color.Red;
   btnGen.FlatStyle:= FlatStyle.Flat;
   frmMain.Controls.Add(btnGen);
   btnGen.Click += btnGen_Click;

   Application.Run(frmMain);
end.
```

Прежде всего, «шифровальщик» должен задать *длину* пароля. По умолчанию она равна шести символам, но можно выбрать и другую – от одного до шестнадцати. Нажав кнопку *Составить!*, пользователь в нижнем *текстовом окне* получает свежий пароль. Если он не годится или нужны и другие пароли, то опять же следует нажать эту кнопку.

А вот так просто наша программа составляет пароли заданной длины:

```
//Генерируем пароль
procedure btnGen_Click(sender: Object; e: EventArgs);
begin
   var s:= txtPass.Text;

   // длина пароля:
   var len:= Integer.Parse(txtLen.Text);
   // проверить:
   if (len < 1) Then
       len := 1;
   if (len > MAX_LEN) Then
       len := MAX_LEN;

   txtLen.Text:= len.ToString();

   pass:='';
   var rand:= new Random();
   For var i:= 1 To len do begin
```

```
      // выбираем случайную букву из строки символов:
      var n:= rand.Next(SYMBOLS.Length)+1;
      var ch:= SYMBOLS[n];
      pass += ch;
  End;//For
  txtPass.text:= pass;
end;
```

После очевидных проверок ввода пользователя процедура *btnGen_Click* последовательно выбирает из строки символов *случайный* и добавляет её к строковой переменной *pass*. Когда длина пароля достигнет заданной, он будет напечатан в *текстовом поле* (Рис. 40.2).

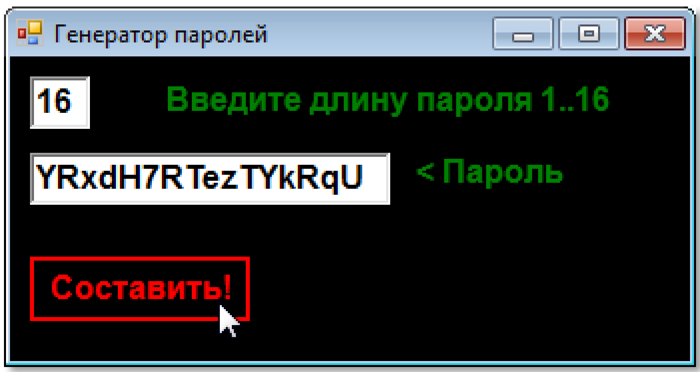


Рис. 40.2. Такой пароль хакеру не по зубам!

Исходный код программы находится в папке **Пароль.**

Решите самостоятельно такие ребусы:

BOR • JOD = ARGON

РАМА • 6 = ОКНО

ЖЕЛЕЗО + ЖЕЛЕЗО = МЕТАЛЛ

Каждый ребус имеет *единственное* решение.

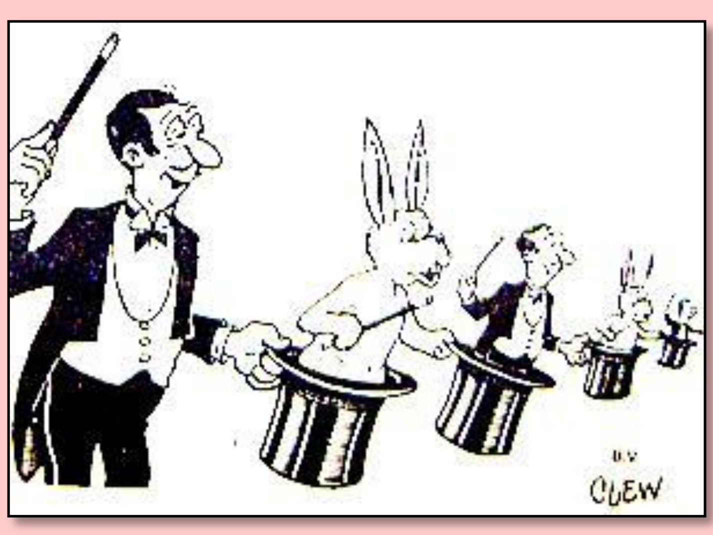
Ответы для проверки решения:

1. 283 x 189 = 53487
2. 1343 x 6 = 8058
3. 304072 + 304072 = 608144

Урок 41. Рекурсия, или Сказочка про белого бычка

- Сказать ли тебе сказку про белого бычка?
- Скажи.
- Ты скажи, да я скажи,
да сказать ли тебе сказку про белого бычка?
- Скажи.
- Ты скажи, да я скажи, да чего у вас будет,
да докуль это будет!
Сказать ли тебе сказку про белого бычка?

Докучная сказка



В теле процедур и функций могут быть записаны вызовы других процедур функций, то есть из одной подпрограммы можно вызвать другую подпрограмму. Но наиболее интересный случай таких вызовов – это когда подпрограмма вызывает саму себя.

Этот приём программирования называется **рекурсией**.

Хорошим примером рекурсии в *литературе* могут служить *докучные сказки*, которые находчивый русский народ придумал для того, чтобы изводить ими своих врагов, поскольку они никогда не заканчиваются (и те, и другие). Такую рекурсию называют *бесконечной*. В программах она вызывает «зависание» компьютера до тех пор, пока не будут полностью исчерпаны его ресурсы. Естественно, такое поведение приложения совершенно недопустимо, поэтому для рекурсивных подпрограмм необходимо предусмотреть *условие выхода* из рекурсии.

Попробуем запрограммировать другую докучную сказку – про то, как поп убил собаку.

Число повторений сказки будет ограничивать переменная *n*. Она просто указывает в качестве передаваемого в процедуру *pop* аргумента, сколько раз мы хотим услышать эту душещипательную историю. Это и будет тот счётчик повторений, который избавит нас от бесконечной рекурсии.

А вот и сама сказочка:

```
//РЕКУРСИВНАЯ ПРОГРАММА ДЛЯ РАССКАЗЫВАНИЯ СКАЗОЧЕК

uses CRT;

//Процедура печати сказочки в консольном окне
procedure pop(n:integer);
begin
   //n - число повторов

   if n <= 0 then
      exit
   else begin
      TextColor(Yellow);
      WriteLn('У попа была собака, он её любил,');
      WriteLn('Она съела кусок мяса, он её убил.');
      WriteLn('В землю закопал и надпись написал:');
      WriteLn;
      pop(n-1);
   end;
End;

//========================================
//          ОСНОВНАЯ ПРОГРАММА
//========================================
begin
   SetWindowTitle('Рекурсивная сказочка');
   TextColor(LightRed);
   writeln('Сказочка про попа и его собаку');
   writeln;

   TextColor(Yellow);
   //n - число повторов сказочки;
   var n:= 3;
   pop(n);
```

```
   TextColor(CRT.LightGreen);
   writeln;
end.
```

В начале процедуры как раз и находится обязательное условие, гарантирующее завершение сказочки. Как только переменная *n* станет равной *нулю*, процедура закончится. Но до этого сказочка будет напечатана 1 раз, после чего процедура *pop* снова вызовет себя со значением переменной *n*, уменьшенным на единицу:

```
pop(n-1);
```

Таким образом, процедура *pop* будет выполнена 3 раза (если вы не изменили начальное значение), а значение параметра *n* всякий раз будет уменьшаться на единицу, пока не достигнет *нуля*, после чего вызовы прекратятся:

n = 3
n = 2
n = 1
n = 0

На последнем вызове сработает условие *n <= 0*, и страдания читателя сих милых строк закончатся (Рис. 41.1).

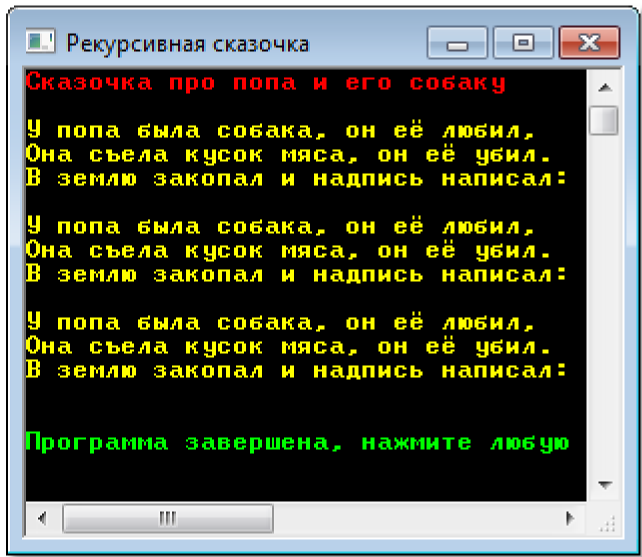


Рис. 41.1. Рекурсивная сказочка при n=3

Действие процедуры *pop* напоминает действиве оператора цикла *While*, поэтому давайте напишем *нерекурсивную* процедуру, что иногда бывает полезно:

```
//НЕРЕКУРСИВНАЯ ПРОГРАММА ДЛЯ РАССКАЗЫВАНИЯ СКАЗОЧЕК

uses CRT;

//Процедура печати сказочки в консольном окне
procedure popWhile(n:integer);
begin
   //n - число повторов

   while n > 0 do
   begin
      TextColor(Yellow);
      WriteLn('У попа была собака, он её любил,');
      WriteLn('Она съела кусок мяса, он её убил.');
      WriteLn('В землю закопал и надпись написал:');
      WriteLn;
      n -= 1;
   end;
End;
```

Заменим вызов процедуры *pop* вызовом процедуры *popWhile*:

```
//===========================================
//          ОСНОВНАЯ ПРОГРАММА
//===========================================
begin
   . . .
   var n:= 3;
   popWhile(n);
   . . .
end.
```

Результат рассказывания сказки будет точно такой же, как и в рекурсивном варианте программы (Рис. 41.2).

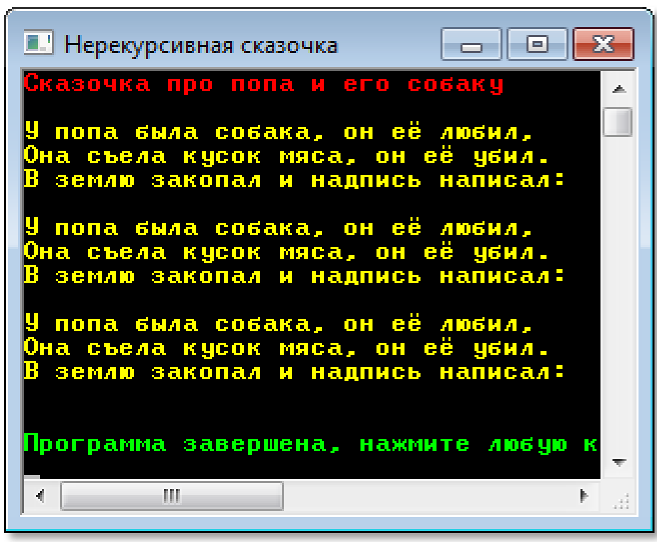


Рис. 41.2. Нерекурсивная сказочка тоже докучная!

Конечно, не всегда бывает так просто избавиться от рекурсии, как в этом примере.

Исходный код программы находится в папке **Pop**.

Передача параметров через стек

При вызове процедур и функций вместо параметров нужно указать в скобках соответствующие им аргументы. Например, вызов процедуры *pop* выглядит так:

```
var n:= 3;
pop(n);
```

Можно поступить ещё проще и записать в скобках числовой литерал:

```
pop(3);
```

При вызове процедуры *pop(3)* значение параметра *n* (то есть тройка в нашем случае) помещается на стек, а в самой процедуре значение снимается со стека и присваивается локальной переменной процедуры. Какое бы имя она ни имела, она не может из-

менить значение переменной с таким же именем вне процедуры (в данном примере имя параметра и имя переменной совпадают, но это простая случайность). Все локальные переменные процедуры уничтожаются при её завершении.

Рекурсивные программы

Перейдём теперь к несказочной рекурсии – к классическому примеру вычисления *факториала*.

Интерфейс программы мы позаимствуем у проекта *FactorialBig*, в котором мы вычисляли его *обычным* способом. Начало программы мы оставим без изменений:

```
//ПРОГРАММА ДЛЯ НАХОЖДЕНИЯ
//ФАКТОРИАЛА ЗАДАННОГО ЧИСЛА
//МЕТОДОМ РЕКУРСИИ
{$reference 'System.Numerics.dll'}

uses CRT, System.Numerics;

//variables
var number:=0;
    fact: BigInteger;

//=========================================
//          ОСНОВНАЯ ПРОГРАММА
//=========================================
begin
   SetWindowTitle('Большие факториалы');

   while(true) do
   begin
      repeat
         TextColor(Yellow);
         write('Введите число 0..2000 >  ');
         TextColor(CRT.LightGreen);
         number:= readInteger;

         //Если задано отрицательное число,
         //то работу с программой заканчиваем:
         if (number < 0) then exit;
```

```
      until ((number >= 0) and (number <= 2000));

      writeln;
      //Вычисляем и выводим в консольное окно
      //факториал заданного числа number:
      fact:= factorial(number);

      //печатаем факториал:
      writeln(number.ToString() + '! = ' + fact.ToString());
      writeln;
      TextColor(Yellow);
   end;
end.
```

Заданное пользователем число *number* мы передаём функции *Factorial*, которая возвращает вычисленное значение факториала. Присваиваем его переменной *fact* и печатаем результат на экране. Затем пользователю будет предложено ввести новое число (Рис. 41.3):

```
function factorial(n: integer): BigInteger;
begin
   //если число равно нулю,
   //то его факториал равен 1:
   If (n = 0) Then
   begin
       Result := 1;
       exit;
   end
   Else begin
      //иначе вычисляем факториал по формуле
      //n! = n * (n-1)!:
      Result:= n * factorial(n-1);
   end;
end;
```

Функция *Factorial* снимает число со стека и помещает его в локальную переменную *n*. Анализируем её значение: если оно равно *нулю*, то мы возвращаем единицу и заканчиваем вычисления. Если *единице*, то вызываем функцию *Factorial* со значением *n-1*.

Вызовы функции будут продолжаться до тех пор, пока параметр *n* не примет значение, равное нулю. К этому времени значение переменной *Result* станет равным *n!*, так что наша функция работает верно (Рис. 41.3).

Большие факториалы

```
Введите число 0..2000 >  0
0! = 1
Введите число 0..2000 >  1
1! = 1
Введите число 0..2000 >  2
2! = 2
Введите число 0..2000 >  5
5! = 120
Введите число 0..2000 >  27
27! = 10888869450418352160768000000
Введите число 0..2000 >  100
100! = 93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000
```

Рис. 41.3. Рекурсивное вычисление факториала

Вместо переменной *Result* вы можете использовать имя функции:

```
function factorial(n: integer): BigInteger;
begin
   If (n = 0) Then
   begin
       factorial := 1;
       exit;
   end
   Else begin
      factorial:= n * factorial(n-1);
   end;

end;
```

Результат от этого не изменится.

Исходный код программы находится в папке **FactorialR**.

Ханойские башни

Другой классический пример рекурсивных программ – решение головоломки *Ханойские башни*.

Эту головоломку придумал французский математик Эдуар Люка (Edouard Lucas) в 1883 году. Тогда же её начали продавать как игрушку, а её изобретателем был назван некий профессор Клаус (Prof. Claus) из коллежа Li-Sou-Stian. Впрочем, люди, сведущие в анаграммах, быстро «вычислили» настоящего автора – профессора Люка из Сен-Луи (Prof. Lucas, Saint Louis).

Историю этой головоломки описал непревзойдённый Мартин Гарднер в книге *Hexaflexagons and Other Mathematical Diversions: The First Scientific American Book of Puzzles and Games*, обложку которой и украшает Ханойская башня (Рис. 41.4).

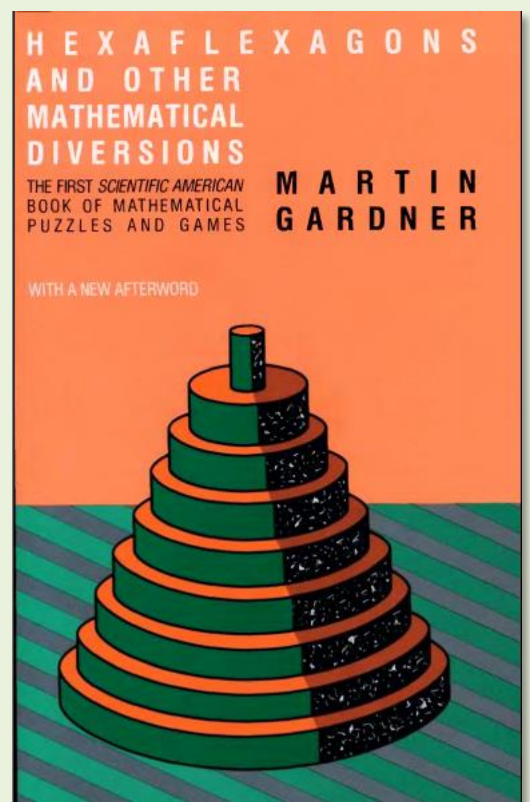


Рис. 41.4. Ханойская башня на обложке книги Мартина Гарднера

Впрочем, это только фрагмент головоломки. На картинке отсутствуют ещё два стрежня без колец. Целиком она представлена на странице 58 (Рис. 41.5).

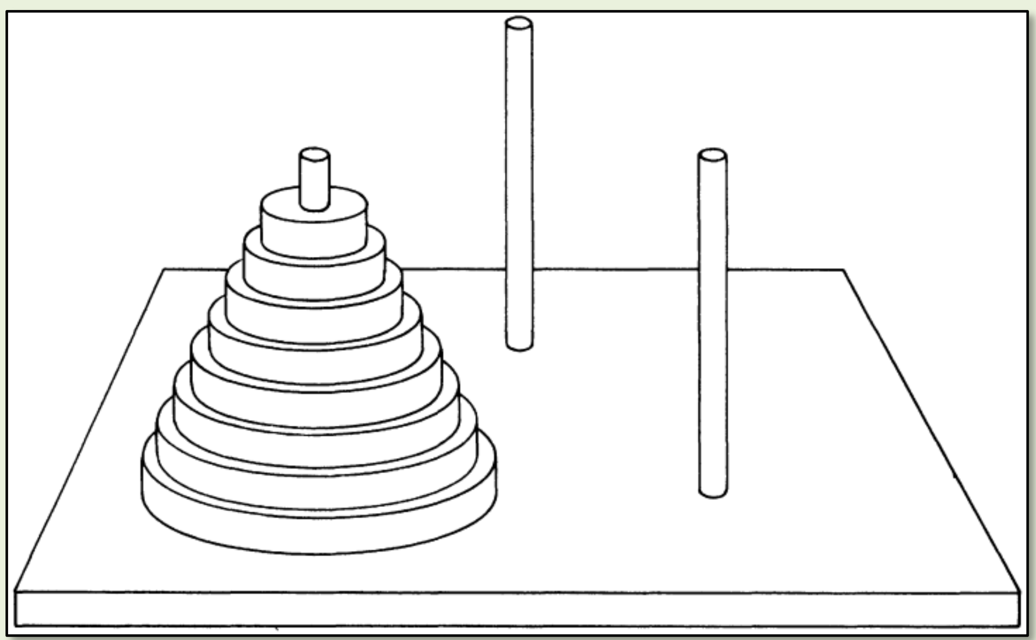


Рис. 41.5. Головоломка в полном виде

Как вы видите, *Ханойская башня* – это набор из восьми колец, нанизанных на стержень. Кольца имеют разный диаметр, и в начальном положении самое большое кольцо находится в основании башни, а самое маленькое – на его вершине. Все остальные кольца располагаются согласно их диаметру.

Цель решения головоломки состоит в том, чтобы переложить за *минимальное* число ходов все кольца с одного стержня на другой так, чтобы их порядок сохранился. Совершенно очевидно, что при двух стержнях задачу решить невозможно, поэтому имеется и ещё один - *вспомогательный* - стержень. За каждый ход разрешается снять одно кольцо с какого-либо стержня и переместить его на один из двух оставшихся. При этом следует неукоснительно выполнять правило: *нельзя на маленькое кольцо класть большое*.

Методом индукции нетрудно доказать, что при *n* кольцах минимальное число ходов равняется 2^n-1, то есть для перемещения восьми колец потребуется $2^8-1 = 255$ ход.

Но давайте вернёмся к головоломке Люка, в которой он продолжил свои мистификации и в инструкции к игре указал, что *Ханойские башни* – это уменьшенный вариант мифической *Башни Брамы*, которая находится в замке индийского города Бенарес. Эта башня построена из 64 дисков, которые жрецы должны переложить по описанным выше правилам. Когда они закончат свою работу, замок превратится пыль, а мир исчезнет.

Как остроумно замечает Мартин Гарднер, за мир можно быть спокойным, поскольку на $2^{64}-1 = 18\ 446\ 744\ 073\ 709\ 551\ 615$ ходов потребуется несколько миллиардов лет. Что касается первого утверждения, то замок превратится в пыль гораздо раньше означенного срока.

Для ручного решения головоломки кольца-диски можно изготовить из бумаги либо замерить их игральными картами – от туза до восьмёрки. Но нас, естественно, больше интересует компьютерное решение. Чтобы не утомлять себя многочисленными перемещениями колец, мы ограничим их число, что ничуть не изменит самой сути рекурсивного процесса.

Итак, начните новый проект с объявления констант и переменной:

```
//ПРОГРАММА ДЛЯ РЕШЕНИЯ ГОЛОВОЛОМКИ
//ХАНОЙСКИЕ БАШНИ

uses CRT;

type
   peg = (peg1, peg2, peg3);

const
   NAME: array[peg1..peg3] of string = ('Стержень1',
                              'Стержень2', 'Стержень3');
   //число дисков:
```

```
   NUM_DISC = 5;

var
   //число ходов:
   nMoves := 0;
```

Основная часть программы очень простая. Мы настраиваем окно приложения, вызываем рекурсивную процедуру *solve*, а затем печатаем число ходов, которые сделала программа:

```
//=========================================
//          ОСНОВНАЯ ПРОГРАММА
//=========================================
begin
   SetWindowTitle('ХАНОЙСКИЕ БАШНИ');
   TextColor(LightRed);
   Write('Решаем головоломку ');
   TextColor(Yellow);
   WriteLn('Ханойские башни');
   writeln;

   nMoves := 0;
   solve(NUM_DISC, peg1, peg2, peg3);

   TextColor(LightRed);
   WriteLn('Всего сделано ходов ' + nMoves.ToString());
   WriteLn;
   TextColor(Yellow);
end.
```

Поскольку процедура *solve* – рекурсивная, то она совсем короткая, но непростая. Чтобы лучше понять, как она действует, нужно внимательно проследить за всеми перемещениями дисков и обратить внимание, что первоначальное назначение стержней - *исходный, промежуточный* и *конечный* – изменяется по ходу решения головоломки:

```
//РЕШАЕМ ЗАДАЧУ ДЛЯ nDisks ДИСКОВ
procedure solve(nDisks: integer; source, inter, dest: peg);
begin
   if ndisks > 1 then
   begin
      solve(nDisks - 1, source, dest, inter);
```

```
      MoveDisk(source, dest);
      solve(ndisks - 1, inter, source, dest);
   end
   else
      MoveDisk(source, dest);
end;
```

Нам поможет в этом процедура *MoveDisk*, которая усердно печатает номера ходов и стержней, которые участвуют в них:

```
//ПЕРЕНОСИМ ДИСК СО СТЕРЖНЯ source
//НА СТЕРЖЕНЬ dest
procedure MoveDisk(source, dest: peg);
begin
   TextColor(LightGreen);
   nMoves += 1;
   //ход:
   write('Ход ');
   if nMoves < 10 then
      write(' ');
   write(nMoves.ToString() + '. ');
   //откуда:
   write(NAME[source] + ' > ');
   //куда:
   writeln(NAME[dest]);
end;
```

Как и предписано теорией, на решение понадобится 31 ход (Рис. 41.6).

Пользуясь этим протоколом, вы легко решите задачу с настоящими стержнями и кольцами, вот только хорошо было бы иметь перед глазами не только список ходов, но и все промежуточные *позиции*, как это обычно делают при разборе шахматных партий. Для этого нам нужно представить каждый стержень в виде *стека*. С этим классом мы уже познакомились при изучении Киберчерепашки. Скопируйте из проекта *Cyber-Turtle* класс *Stack<T>* и давайте-ка изучим работу стека.

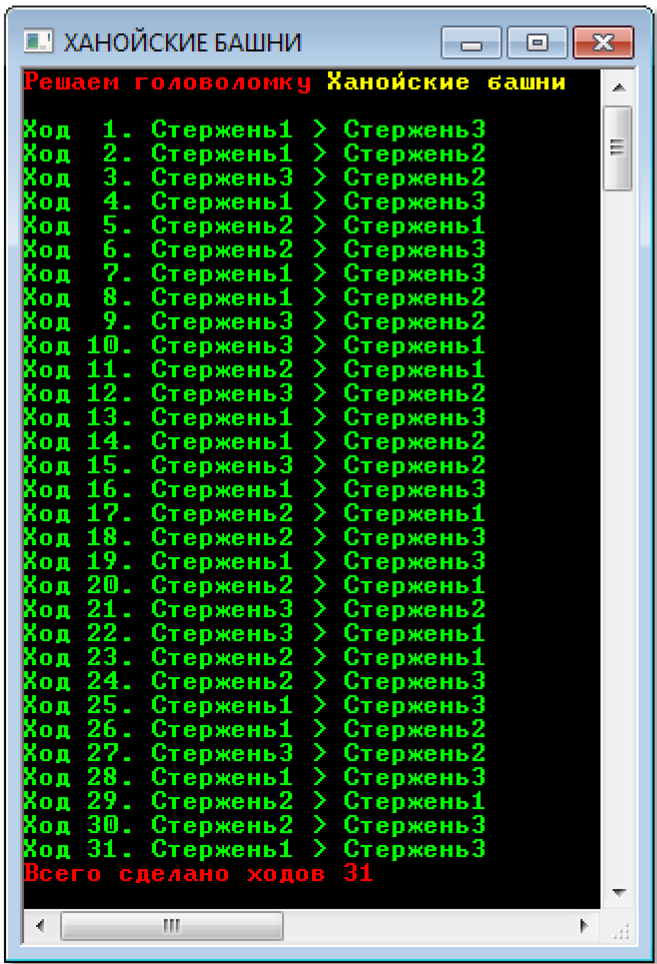


Рис. 41.6. Задача решена!

Класс Stack<T>

Как мы знаем, при передаче аргументов в процедуры и функции они сначала кладутся на стек, а затем снимаются с него. Этот стек называется *системным*. Он используется программой также для хранения адресов возврата после окончания работы подпрограмм и некоторых других целей. Поскольку этот стек целиком и полностью принадлежит программе, то мы не можем пользоваться им для своих собственных нужд. А как показывает практика, иногда такая структура данных просто необходима.

В пространстве имён *System.Collections.Generic* имеется мощный класс *Stack*, но для наших скромных программ вполне достаточно и того крохотного класса, что мы добавили к своему проекту.

Класс **Stack<T>** служит для хранения различных объектов «*стопкой*». Тип объектов обозначен буквой *T* в угловых скобках. Его и нужно указывать при объявлении и создании стека.

Рассмотрим работу стека на примере с книгами.

Сначала стол пустой, на нём книг нет. Затем мы кладём на него первую книгу (Рис. 41.7).

Рис. 41.7. На столе лежит *одна* книга

Затем на первую книгу – вторую, и так далее (Рис. 41.8-11).

Рис. 41.8. В стопке *две* книги

Рис. 41.9. В стопке *три* книги

Рис. 41.10. В стопке *четыре* книги

Рис. 41.11. В стопке *пять* книг

Эти действия можно продолжать до тех пор, пока в доме не закончатся книги или стопка не развалится.

Компьютер не может выстраивать стопки из книг или других материальных объектов. Вместо них он оперирует данными - числами, строками и другими «виртуальными» объектами. В остальном аналогия полная.

Мы можем раскладывать книги в *несколько* стопок (например, по каждому школьному предмету отдельно), и паскаль позволяет нам создавать любое число стеков.

В переводе с английского *stack* как раз и значит *стопка*. По-русски и сам класс, и объекты этого класса называют *стеком*.

Переменные типа *Stack* объявляются точно так же, как и другие переменные, но дополнительно в угловых скобках нужно указать тип элементов, которые будут храниться в стеке. Например, это могут быть *целые числа*:

```
var
   _stack: Stack<integer>;
```

Перед употреблением стек необходимо *создать* с помощью ключевого слова *new*:

```
_stack:= new Stack<integer>;
```

Вначале стек, как и наш стол, пустой, в чем легко убедиться, если распечатать значение поля *last,* которое хранит число элементов на стеке (Рис. 41.12):

```
writeln('Число элементов на стеке = ' + _stack.last.ToString);
```

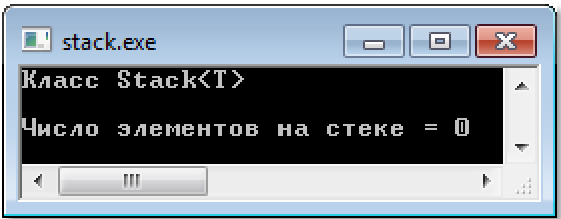


Рис. 41.12. Пока пусто!

Чтобы поместить какой-нибудь объект на стек, нужно вызвать метод *push* с нужным аргументом:

push(i: T);

Тип аргумента должен совпадать с типом элементов стека.

Например, положим на наш стек число 1:

```
_stack.push(1);
```

Теперь он имеет один элемент, который находится на *вершине* стека.

Аналогично мы можем добавить к стеку сколько угодно объектов:

```
For var i:= 1 to 12 do
   _stack.push(i);

writeln('Число элементов на стеке = ' + _stack.last.ToString);
_stack.WriteStack();
```

В цикле *for* мы последовательно помещаем на стек числа от 1 до 12 (Рис. 41.13).

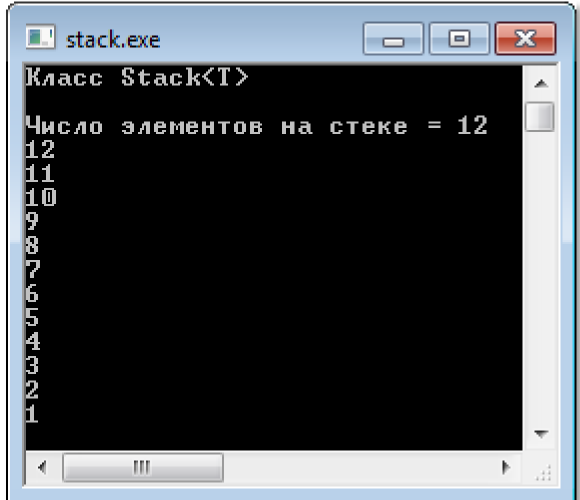


Рис. 41.13. В стеке – 12 чисел

Важно помнить, что последний объект, который мы кладём на стек, всегда находится на *вершине* стека – точно так же, как и книги на столе. Таким образом, при добавлении объектов стек растёт *вверх*, что хорошо видно на Рис. 41.13. Последнее число 12 лежит выше других, а первое число – единица – «прозябает» в самом низу стека.

Из стопки мы можем взять только ту книгу, которая лежит *наверху* (если не хотим уподобиться Трусу из комедии *Операция Ы и другие приключения Шурика*, который из «стека» с горшками вытянул нижний, отчего весь стек рухнул со страшным звоном), из компьютерного стека также нельзя вытянуть объект, не лежащий на вершине. А чтобы снять со стека верхний объект, достаточно воспользоваться методом *pop*:

function pop: T;

Он вернёт тот объект, который находится на вершине стека. При этом число элементов в стеке уменьшится на единицу, объект будет удален из стека, а его место на вершине займёт тот объект, который был вторым сверху. Проясним эту ситуацию примером (Рис. 41.14):

```
 var n:= _stack.pop;
  writeln('Сняли элемент # ' + (_stack.last+1).ToString +
          '= '+ n.ToString);
  writeln('Число элементов на стеке = ' + _stack.last.ToString);
  _stack.WriteStack();
  writeln();
```

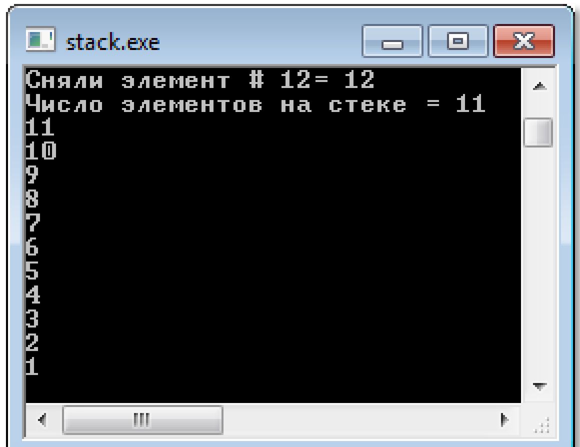


Рис. 41.14. Сняли со стека число 12

Ханойские башни 2

Теперь давайте вернёмся к *Ханойской башне*. После знакомства с устройством стека вы без труда обнаружите полное сходство стека с дисками на стержне. Действительно, со стержня можно снять только верхний диск и положить его также разрешается исключительно на верхний диск другого стержня.

Итак, каждый стержень мы вполне можем представить стеком, в котором каждый диск - это его номер. Самый большой диск имеет номер 5 (*NUM_DISC*), а самый маленький – номер 1. Поскольку у нас *три* стержня, то и стеков нам потребуется столько же. Очень удобно объединить их в массив:

```
pegs: array[peg1..peg3] of Stack<integer>;
```

В основной программе мы создаём стеки-стержни:

```
  //создаём стержни:
  for var i:= peg1 to peg3 do
      pegs[i]:= new Stack<integer>;
```

И укладываем на стек числа 5,4,3,2,1, которые обозначают кольца соответствующих размеров:

```
  //надеваем кольца на первый стержень:
  for var i:= NUM_DISC downto 1 do
      pegs[peg1].push(i);
```

Теперь можно поинтересоваться образной картиной происходящих событий:

```
 nMoves := 0;
 writePegs;
```

Процедура *writePegs* показывает нам, что головоломка приняла исходное положение (Рис. 41.15):

```
procedure writePegs;
begin
```

```
    writeln;
    for var n:= NUM_DISC downto 1 do begin
       for var i:= peg1 to peg3 do begin
           if (pegs[i].last -n >=0) then begin
              TextColor(LightRed);
              write(s[i] + pegs[i].a[n-1].ToString());
           end
           else begin
             TextColor(Yellow);
             write(s[i]+':');
           end;
       end;
       writeln;
    end;
end;
```

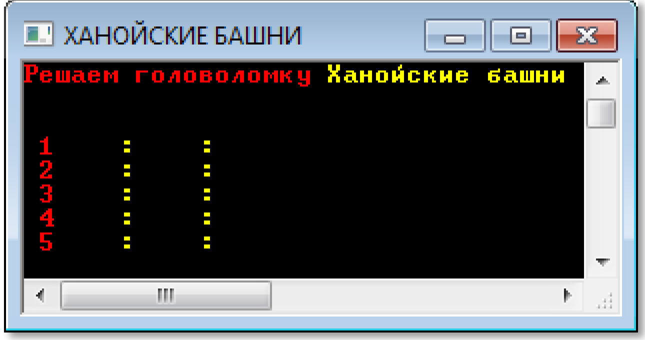


Рис. 41.15. Головоломка готова!

Приступаем к решению задачи:

```
solve(NUM_DISC, peg1, peg2, peg3);
```

Естественно, на процедуру *solve* никак не повлияли наши разноцветные картинки на экране, поэтому переходим к процедуре *MoveDisk*, которая дополнилась строчками, отвечающими за перекладывание дисков на стеках:

```
procedure MoveDisk(source, dest: peg);
begin
   . . .
   //откуда:
   write(NAME[source] + ' > ');
   var nSource:= pegs[source].pop();
   //куда:
   writeln(NAME[dest]);
```

```
    pegs[dest].push(nSource);
    writePegs;
    writeln();
end;
```

Мы снимаем верхнее число (диск) с исходного стержня *source* и переносим его на целевой стержень *dest*. Именно так мы бы и поступили при решении головоломки вручную. С помощью процедуры *writePegs* мы распечатываем позицию после очередного хода.

Итогом нашей бурной деятельности служит *наглядный* отчёт обо всех погрузочно-разгрузочных операциях (Рис. 41.16).

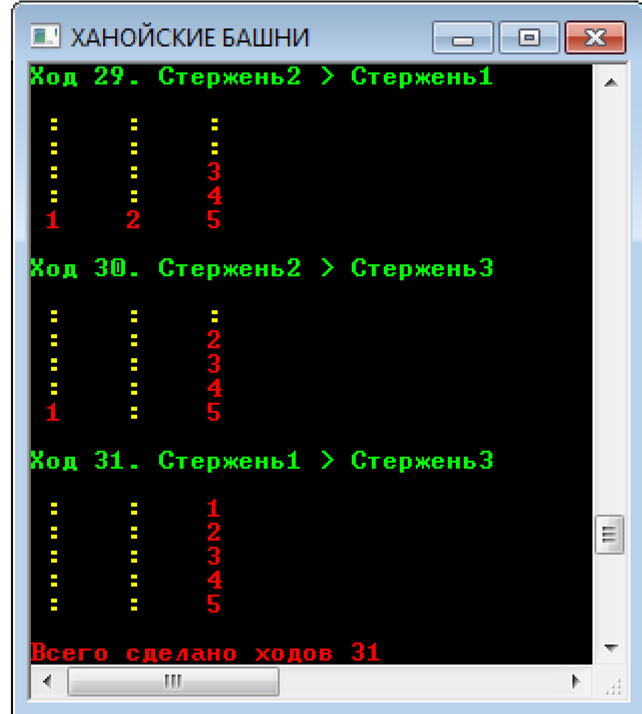


Рис. 41.16. Красота решения – страшная сила!

Я думаю и надеюсь, что теперь вы без труда проследите за всеми рекурсивными хитросплетениями решения этой непростой задачи!

Исходный код программы находится в папке **Hanoi**.

1. Переделайте программу для подсчёта заданного числа *Фибоначчи* рекурсивным методом. Вычисляйте только заданное число, а не все числа, не превосходящие заданного. Поскольку глубина рекурсии в этом случае большая, то скорость программы очень низкая, поэтому рекурсивный способ получения чисел Фибоначчи не имеет практического значения.

Исходный код программы находится в папке **FibonacciR**.

2. Попробуйте изобразить процесс решения головоломки *Ханойские башни* в *графическом окне*. Основание, стержни и диски легко представить прямоугольниками разных цветов. Процедуру *writePegs* следует переделать в *drawPegs*, которая и будет рисовать текущую позицию. Задача непростая, но вполне посильная.

Но можно пойти ещё дальше и *анимировать* перемещение колец с одного стержня на другой. Вот это уже интересно!

Урок 42. Занимательная география

Снятся людям иногда
Их родные города -
Кому Москва, кому Париж…

Песня Эдуарда Хиля *Голубые города*

Помните, как Доцент и его команда в кинокомедии *Джентльмены удачи* играли в *Города*? – Впрочем, игра всем известна с детства и без кинокомедии. Суть её заключается в том, чтобы последовательно называть города так, чтобы первая буква каждого следующего слова совпадала с последней буквой последнего названного. Проигрывает тот участник, который не сможет на своём ходу назвать походящего слова.

Вот пример такой игры: *Москв**а*** – ***А**рхангельс**к*** – ***К**иев* – ***В**олгогра**д*** – ***Д**ел**и*** – ***И**ркутс**к*** *- …* Поскольку городов на земле очень много, то игра может и затянуться. В этом случае можно ограничить выбор городов какой-нибудь страной или частью света.

Можно предложить и такой вариант этой игры, который более «географичен», а потому и более полезен. Возьмём карту мира и совершим вояж по городам, придерживаясь основного принципа игры, то есть названия городов должны составлять такую же цепочку слов, как и в основном варианте игры.

Если карта мира небольшая по размерам (а для игры она подойдет, конечно, лучше, чем карта в полстены!), то на ней будут отмечены только *большие* города. Например, мы можем воспользоваться вот такой таблицей и для определённости считать, что путешествовать мы будем только по ним.

Город	**Население**	**Площадь** (км^2)	**Страна**
Мехико	18 841 916	1 485	Мексика
Шанхай	16 492 800	289,44	КНР
Карачи	18 140 670	3530	Пакистан
Стамбул	8 566 823	1538,9	Турция
Токио	8 742 995	621,9	Япония
Мумбаи	13 922 125	603	Индия
Буэнос-Айрес	13 356 715	4000	Аргентина

Дакка	12 725 000	815	Бангладеш
Манила	12 285 000	636	Филиппины
Дели	11 954 217	1483	Индия
Москва	10 562 099	1081	Россия
Сеул	10 421 782	605,4	Республика Корея
Киншаса	10 076 099	10 550	ДР Конго
Сан-Паулу	10 037 593	1520	Бразилия
Лагос	9 360 883	999	Нигерия
Джакарта	8 576 788	660	Индонезия
Нью-Йорк	8 140 993	1214	США
Лима	8 057 397	804,3	Перу
Каир	7 947 121	210	Египет
Пекин	7 712 104	1370	КНР
Лондон	7 581 052	1580	Великобритания
Богота	7 137 849	1590	Колумбия
Гонконг	7 102 354	1104	КНР
Тегеран	6 897 547	660	Иран
Лахор	6 747 238	1010	Пакистан
Багдад	6 431 839		Ирак
Рио-де-Жанейро	6 193 265	1180	Бразилия
Бангкок	5 698 435	1100	Таиланд
Бангалор	5 180 533	230	Индия
Сантьяго	5 090 824		Чили
Калькутта	5 021 458		Индия
Сингапур	4 974 232	699	Сингапур
Янгон	4 886 305		Мьянма
Эр-Рияд	4 606 888		Саудовская Аравия
Санкт-Петербург	4 600 310	1439	Россия
Ченнай	4 562 843		Индия
Чунцин	4 406 788		КНР
Сиань	4 305 536		КНР
Ухань	4 262 236	400	КНР
Александрия	4 247 414	2680	Египет
Ибадан	4 149 487		Нигерия
Сидней	4 114 710	1212	Австралия
Кано	4 082 050		Нигерия
Хайдарабад	3 980 938	170	Индия
Лос-Анджелес	3 975 590	1290	США
Анкара	3 882 639	2500	Турция
Чэнду	3 915 259		КНР
Абиджан	3 900 546		Кот-д’Ивуар
Пусан	3 899 140	771	Республика Корея
Ахмадабад	3 867 336	190	Индия
Тяньцзинь	3 682 177		КНР
Омдурман	3 667 982		Судан
Берлин	3 599 000		Германия
Чикаго	3 573 721		США
Мельбурн	3 528 690		Австралия

Если вы с этим выбором не согласны, ничто не мешает вам отобрать города по собственному желанию. Для нас важно только одно – найти такой маршрут, чтобы посетить *как можно больше городов*, поскольку совершенно очевидно, что все города в одну цепочку уложить не удастся. Задача сама по себе непростая, но ещё труднее доказать, что найденная цепочка городов и есть самая длинная из всех возможных. Так или иначе, но лучше решение найти (или только проверить) с помощью компьютера.

Компьютерный навигатор

Прежде всего, нам потребуется список городов, по которым мы и проложим наш маршрут. Названия городов из выделенной колонки (см. Табл.) мы переведём в ВЕРХНИЙ регистр (это можно сделать и в самой компьютерной программе, и в текстовом редакторе), затем скопируем в *Блокнот* и сохраним файл *города.txt* в кодировке *UTF-8*.

Теперь можно браться за программу **Города**, которая и составит нам цепочку слов. Поскольку мы сохранили список названий городов в файле (а это гораздо удобнее, чем присваивать значения элементам массива в исходном тексте программы!), то этот список нам нужно загрузить с диска. Эта необходимость подталкивает нас взять за основу нового проекта любой из уже готовых, в котором мы загружали словарь. Например, подойдёт *Латиница*.

Часть *переменных*, отвечающих за загрузку файла в массив *spisok*, мы оставим без изменения:

```
//ПРОГРАММА ДЛЯ ПОИСКА ЦЕПОЧЕК СЛОВ

uses CRT;

const
   MAX_WORDS = 1000;
   fileNameIn= 'города.txt';

var
   //массив-список названий городов:
```

```
   spisok: array [1..MAX_WORDS] of string;
   //число слов в списке:
   nWords: integer;
   f: textfile;
```

Однако нам их маловато будет – только загрузить список, поэтому добавим *переменные* конкретно для нашей программы:

```
   //если флаг равен True, значит, слово стоит в цепочке:
   flgChain: array [1..MAX_WORDS] of boolean;
   //массив начальных букв слов:
   begLetter: array [1..MAX_WORDS] of string;
   //массив конечных букв слов:
   endLetter: array [1..MAX_WORDS] of string;
   //число уже выставленных слов:
   n: integer;
   //макс. длина цепочки:
   maxChain: integer;
   //номер слова в списке, которое стоит в цепочке:
   idWord: array [1..MAX_WORDS] of integer;
   //текущий номер слова в списке для слова,
   //которое мы ставим в цепочку:
   id: integer;

label nextWord, nextId, back, finish;
```

Об их назначением мы подробно поговорим по ходу написания программы, которую начнём с *загрузки* списка слов в массив *spisok*:

```
//===========================================
//          ОСНОВНАЯ ПРОГРАММА
//===========================================
begin
   SetWindowTitle('ИЩЕМ ЦЕПОЧКИ ГОРОДОВ');
   TextColor(LightRed);
   writeln('ИЩЕМ ЦЕПОЧКИ ГОРОДОВ');
   writeln;
   TextColor(Yellow);

   readFile;
```

Процедуру *readFile* мы обсуждать не будем, потому что мы уже много раз использовали её в других проектах. Нам только важно помнить, что теперь все слова находятся в массиве *spisok*, а их число надёжно хранится в переменной *nWords*.

Дальше мы поступим хитро и обзаведёмся двумя *массивами* - для первых *begLetter* и последних *endLetter* букв всех слов в списке. Действительно, слова в цепочке цепляются друг за друга так, что первая буква следующего слова должна быть точно такой же, как и последняя буква предыдущего (у первого слова цепочки предшественника нет, поэтому его можно ставить в цепочку безбоязненно). Можно каждый раз извлекать эти буквы из слов и сравнивать их между собой, но эти действия потребуют больше времени, чем в случае с двумя массивами (да и программа с двумя массивами букв становится понятнее!). Остальные буквы слов нас вообще не интересуют, так что для составления цепочки нам будет достаточно только этих массивов. Их заполнение – дело совершенно незамысловатое, но мы должны учесть специфику нашей программы: каждое слово можно использовать в цепочке только *один* раз. Значит, мы как-то должны отмечать *цепочечные* слова. Проще всего для этого завести массив *flgChain* и заносить для каждого слова значение *False*, если слово в цепочке отсутствует, и *True* - в противном случае.

До построения цепочки все слова, естественно, находятся вне цепочки:

```
  //заполняем массивы начальных и конечных букв:
  for var i:=1 to nWords do
  begin
    begLetter[i]:= spisok[i][1];
    Write(i + ' ' + begLetter[i] + ' - ');
    var len:= spisok[i].Length;
    endLetter[i]:= spisok[i][len];
    //WriteLn(endLetter[i] + ' len= ' + len.ToString());
    //ни одно слово не стоит в цепочке:
    flgChain[i] := false;
  End;//For
```

Сначала в цепочке нет ни одного слова:

```
   //цепочка пустая:
   maxChain:=0;
   //ни одно слово еще не выставлено:
   n:=0;
```

Вспомним условие задачи: построить цепочку слов *максимальной* длины. Для того чтобы определить, какая цепочка самая длинная, мы должны составить *все* возможные цепочки и выбрать из них нужную. Это типичная задача *перебора с возвратами*, поэтому в построении алгоритма мы будем опираться на *ладейную* головоломку, но с учётом того, что слова следует выставлять в *одну* цепочку, а не на *разные* горизонтали шахматного поля.

Начало – традиционное для переборных программ: ставим *следующее* слово в цепочку: первое (оно тоже следующее – для пустой цепочки), второе и так далее, насколько возможно. Аналог слова, которое мы добавляем к цепочке, - это новая ладья.

```
//ставим следующее слово в цепочку:
nextWord:
   n += 1;
```

Итак, нам нужно выбрать слово из списка, которое отсутствует в цепочке. Начинаем поиск очередного слова для цепочки от начала массива *spisok*. По смыслу это действие совпадает с перемещением ладьи вправо по горизонтали. Если список закончился, а ни одного нового слова в цепочку поставить не удалось, то нужно возвращаться назад. В этой программе мы, конечно, вернёмся не к предыдущей выставленной ладье, а к предыдущему слову в цепочке.

```
   //начинаем поиск очередного слова для
   //цепочки от начала списка:
   idWord[n]:=0;
   id:=0;

nextId:
   id += 1;
   If (id >= nWords) Then
      Goto back;
```

Опять мы поступаем аналогично ладейной программе: проверяем, возможно ли присоединить очередное слово из списка к цепочке. Если нет, то переходим к следующему слову в списке:

```
  if not test() then begin
     Goto nextId;
   End;//If
```

Что касается функции проверки, то здесь она совершенно другая, поскольку именно проверка существенно зависит от особенностей конкретной задачи, в то время как алгоритм перебора изменяется не столь радикально. Но, как и в случае с ладьями или ферзями, первое слово можно ставить в цепочку без дополнительной проверки, так как оно не зависит от других слов. Нельзя поставить в цепочку уже выставленное слово, и, наконец, главная проверка – на совпадение букв:

```
//Проверка: можно ли поставить очередное
//слово в цепочку
function test(): boolean;
begin
  Result:= true;
  If (n = 1) Then
    exit;

  //слово уже стоит в цепочке:
  If (flgChain[id]) Then begin
     Result:= false;
     exit;
  End;//If

  //первая буква очередного слова должна совпадать
  //с конечной буквой предыдущего слова:
  If (begLetter[id] <> endLetter[idWord[n-1]]) Then
  begin
     Result:= false;
     exit;
  End;//If
end;
```

В результате проверки функция возвращает значение *False*, если слово не подходит, и *True* – если годится.

Первое слово в нашем списке – *Мехико*. Оно и начнёт цепочку. В программе это действие выполняется так. В массиве *idWord* мы запоминаем номер слова из списка, которое дополнило цепочку, а также отмечаем его в массиве *flgChain* как использованное, чтобы у программы не было соблазна использовать его вторично. Нам в этой программе не нужны вообще все цепочки, которые можно составить из слов списка. Нет – нам нужна только *самая длинная*. Её длина хранится в переменной *maxChain*, которую мы обнулили в самом начале программы. Естественно, уже первое слово создаст цепочку *единичной* длины. Поэтому максимальная длина станет равной единице, и мы сможем напечатать все слова, из которых цепочка состоит, после чего постараемся удлинить её ещё на одно слово.

```
   //поставили слово в цепочку:
   idWord[n]:=id;
   flgChain[idWord[n]]:= true;
   //проверяем длину цепочки:
   If (n > maxChain) Then begin
      maxChain:= n;
      writeChain();
   End;//If

Goto nextWord;
```

По условию *n > maxChain* будет напечатана только *первая* цепочка максимальной длины. Иногда таких цепочек может оказаться и *несколько*. Если вы хотите лицезреть их все, то исправьте условие на *n >= maxChain*.

В процедуре печати мы выводим в *консольное окно* все *n* слов цепочки из списка. Номера слов в списке легко узнать по значению элемента массива *idWord[i]*, соответствующего *i*-тому слову в цепочке.

```
procedure writeChain;
begin
  TextColor(LightRed);
  WriteLn('Длина цепочки = ' + n.ToString());
  TextColor(Yellow);
  For var i:= 1 To n do
```

```
    WriteLn(spisok[idWord[i]]);

  WriteLn;
  WriteLn;
  Read;
end;
```

Поставив слово *МЕХИКО* в цепочку, мы переходим к поиску второго слова: *n=2*. Просматриваем список городов с самого начала: *id=1*. Проверка возвращает отрицательный результат: слово *МЕХИКО* уже выставлено. Переходим к следующему слову: *id= 2*. Это *ШАНХАЙ*. Это слово не использовано, но первая буква – *Ш* – не совпадает с последней буквой – *О* – предыдущего слова. Результат опять отрицательный. И так мы будем просматривать весь список, пока не дойдём до слова *ОМДУРМАН* в самом конце списка. Добавляем его к цепочке и ищем третье слово. Легко находим, что это слово *НЬЮ-ЙОРК*. Затем присоединяем *КАРАЧИ – ИБАДАН*, и стоп: больше ни одного слова, начинающегося на букву *Н* в списке нет. Нужно возвращаться *назад*!

```
back:
   //переходим к предыдущему слову:
   n -= 1;
   If (n = 0) Then
     Goto finish;

   id:= idWord[n];
   //убираем слово  n из цепочки:
   flgChain[idWord[n]]:= false;
   Goto nextId;
```

Почти дословно эта процедура повторяет *ладейную*: переходим к предыдущему выставленному слову и, если слов в цепочке больше не осталось, то наши поиски закончены на самом интересном месте, иначе убираем его из цепочки (поэтому его снова можно использовать!) и продолжаем поиски со следующего слова в списке.

Наша программа, как и большинство других переборных программ, совсем несложная, но неминуемо находит самую длинную

цепочку, какую только можно составить из слов заданного списка. Но если список очень длинный, то придётся запастись недюжинным терпением!

Для нашего виртуального путешествия самый длинный маршрут состоит из 15 городов (Рис. 42.1).

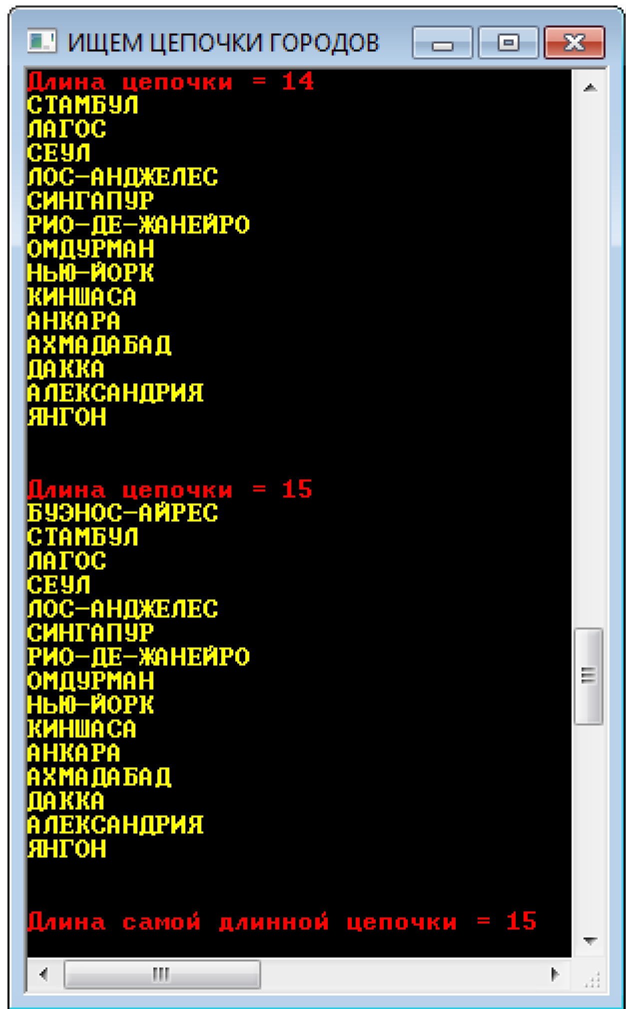


Рис. 42.1. А вы сумели проложить такой маршрут?

Исходный код программы находится в папке **Города**.

Чайнворды

Полученная нами цепочка городов в *головоломковедении* называется *чайнвордом*, а это значит, что мы без всяких переделок

можем использовать нашу программу и для составления чайнвордов. Но лучше начать новый проект под названием **Chainword** и сохранить в папке под соответствующим именем нашу программу.

Если у вас на примете есть подходящий словарик, то вы можете использовать его, а нет – так попробуйте из моих запасов - *морской.txt*.

Наша программа составит цепочку немалой длины (больше ста слов!), причём можно в**ы**резать из цепочки кусок так, чтобы первая буква первого слова совпала с последней буквой последнего слова (Рис. 42.2).

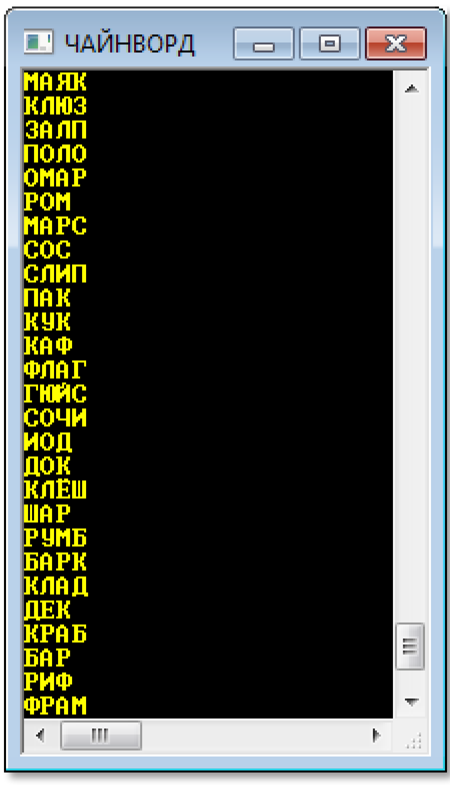


Рис. 42.2. Цепь замкнулась!

Мы легко соединим первое и последнее звенья цепочки и получим *замкнутый чайнворд*.

Не пытайтесь повторить этот трюк с нашими городами!

К сожалению, каждый раз мы будем получать одну и ту же цепочку, потому что наш алгоритм перебирает слова в том самом порядке, в котором они записаны в словаре. Но мы легко изменим этот порядок, если *перемешаем* слова:

```
//Перемешиваем слова в списке
procedure shuffle;
begin
   For var i:= 1 to nWords do
   begin
      var n1:= Random(nWords)+1;
      var n2:= Random(nWords)+1;
      var s:= spisok[n1];
      spisok[n1]:= spisok[n2];
      spisok[n2]:= s;
   end;//for
end;
```

Осталось после загрузки файла вызвать процедуру *shuffle*:

```
//===========================================
//          ОСНОВНАЯ ПРОГРАММА
//===========================================
begin
   SetWindowTitle('ЧАЙНВОРД');
   TextColor(LightRed);
   writeln('СОСТАВЛЯЕМ ЧАЙНВОРД');
   writeln;
   TextColor(Yellow);

   readFile;
   shuffle;
   writeln;
```

Теперь после каждого запуска программы вы будете получать *новый* чайнворд (Рис. 42.3).

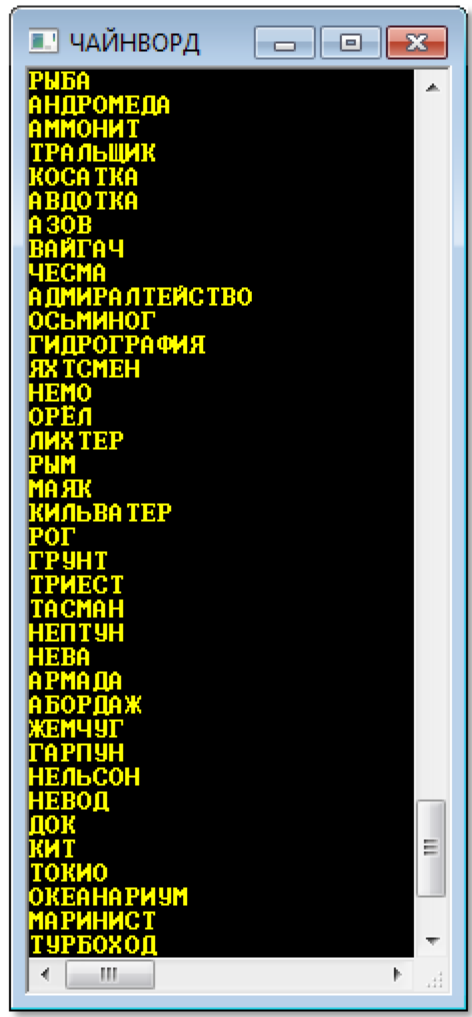


Рис. 42.3. Морской чайнворд

Вот так просто вы можете составлять сколько угодно чайнвордов – если, конечно, предварительно приготовите словарик с интересными толкованиями слов. Вообще говоря, составить чайнворд совсем нетрудно, поэтому красивое оформление сетки и неожиданные толкования – вот что может сделать вашу задачу по-настоящему занимательной!

Поскольку в реальном словаре может оказаться очень много слов, то составление всех цепочек может растянуться на всю оставшуюся жизнь, поэтому при достижении желаемой длины нужно прекратить дальнейшие поиски:

```
//ставим следующее слово в цепочку:
nextWord:
   n += 1;
```

```
   //ограничиваем длину цепочки:
   If (n > 100) Then
      Goto finish;
```

Исходный код программы находится в папке **Chainword**.

Играем в города

А теперь вернёмся к началу урока и научим компьютер играть в *Города*. Его противником будет пользователь, который всегда называет первое слово.

Поскольку игра серьёзная, то и *переменных* нам понадобится немало. Большинство из них вам уже хорошо известны, а названия новых говорят сами за себя.

```
//ПРОГРАММА ДЛЯ ИГРЫ В ГОРОДА

uses CRT;

const
   MAX_WORDS = 10000;
   fileNameIn= 'города3.txt';

var
   //массив-список названий городов:
   spisok: array [1..MAX_WORDS] of string;
   //число слов в списке:
   nWords: integer;
   f: textfile;

   //если флаг равен True, значит, слово стоит в цепочке:
   flgChain: array [1..MAX_WORDS] of boolean;
   //конечная буква:
   endLetter: string;
   //число уже выставленных слов:
   n: integer;
   //номер слова в списке, которое стоит в цепочке:
   idWord: array [1..MAX_WORDS] of integer;
```

```
   //очередной город:
   gorod: string;
   //длина цепочки:
   lenChain: integer;

label startGame, hodIgroka, endGame, c_yes;
```

Перед началом игры нужно загрузить список с названиями городов. В файле *города3.txt* записаны больше тысячи российских городов. Отброшены те города, названия которых состоят из двух и более отдельных слов. Обычно такие слова не используют в словесных головоломках и играх. К сожалению, этот список имеет пробелы, поэтому вы можете дополнить его новыми названиями.

Здесь нужно обратить внимание на то, что название файла мы передаём в процедуру как аргумент:

```
//==========================================
//           ОСНОВНАЯ ПРОГРАММА
//==========================================
begin
   SetWindowTitle('ИГРАЕМ В ГОРОДА');
   TextColor(LightRed);
   writeln('ИГРАЕМ В ГОРОДА');
   writeln;
   TextColor(Yellow);
   readFile(fileName);
```

В самой процедуре *readFile* имя файла – это локальная переменная *fn*:

```
//Считываем словарь в массив
procedure readFile(fn: string);
begin
   nWords:=0;
   var s: string;
   assign(f, fn);
   reset(f);
   while not eof(f) do
   begin
      readln(f, s);
      writeln(s);
```

```
        inc(nWords);
        spisok[nWords]:= s;
    end;
    close(f);
end;
```

Не забываем *перемешать* слова в списке, чтобы компьютер играл всякий раз по-другому:

```
 //Начало игры
startGame:
    shuffle;

    //ни одно слово не стоит в цепочке:
    for var i:=1 to nWords do
    begin
        flgChain[i] := false;
    End;//For

    //длина цепочки равна нулю:
    lenChain:=0;
```

Как мы и договаривались, первый ход делает игрок. Для этого он просто вводит с клавиатуры название любого города:

```
//ХОД ИГРОКА
hodIgroka:
    TextColor(Yellow);
    WriteLn;
    Write('Напишите город > ');
    gorod:= ReadString().ToUpper();
```

При этом ему не нужно заботиться о регистре букв, потому что его слово мы тут же переводим в *верхний* регистр, ведь все слова в нашем списке записаны ПРОПИСНЫМИ буквами.

Естественно, мы не можем слепо доверять игроку, поэтому должны проверить, есть ли названный город в нашем списке (вы можете дополнить список новыми городами, если вам этого недостаточно). Если он отсутствует, то игроку придётся повторить свой ход.

Если название города имеется в списке, но он уже назван (такая ситуация очень часто возникает в этой игре), то игрок должен назвать другой город.

И наконец, первая буква названия города должна совпадать с последней буквой предыдущего города – таковы суровые правила игры!

Обратите внимание, что для *первого* города сделано исключение в этой проверке:

```
//проверка:
 var flg:= False;
 var i: integer;
 For i:=1 to nWords do
 begin
    If gorod = spisok[i] Then begin
       flg:= True;
       break;
    End;//If
 end;//for


If (not flg) then begin
    TextColor(LightRed);
    Writeln('Такого города нет!');
    goto hodIgroka;
 end
Else If (flgChain[i]) then begin
    TextColor(LightRed);
    Writeln('Такой город уже назван!');
    goto hodIgroka;
 end
Else If (lenChain > 0) and (spisok[i][1] <> endLetter) Then
 begin
    TextColor(LightRed);
    Writeln('Неверная первая буква!');
    goto hodIgroka;
 End;
```

Итак, долго ли коротко, но игрок сделает правильный ход, что мы и должны зафиксировать в программе. Город игрока мы от-

мечаем в списке, чтобы его нельзя было назвать повторно. Увеличиваем длину цепочки и определяем последнюю букву слова *endLetter* для проверки следующего хода.

Обычно в игре действуют так. Если последняя буква слова *Ы* или *Ь* (наверное, можно добавить к ним *Й*), с которых русские слова не начинаются, то последней считают *предпоследнюю* букву. Для игры это хорошо, но при составлении чайнвордов такие послабления недопустимы.

После соблюдения всех необходимых процедур слово игрока печатается на экране ПРОПИСНЫМИ буквами:

```
  //удачный ход:
   flgChain[i]:= true;
   lenChain += 1;
   var len:= spisok[i].Length;
   endLetter:= spisok[i][len];
   //если слово заканчивается на "плохие" буквы,
   //то берем предыдущую:
   While (len >= 2) And ((endLetter= 'Ы') Or (endLetter= 'Ь')) do
   begin
      len -= 1;
      endLetter:= spisok[i][len];
   end;//while

   WriteLn;
   TextColor(Yellow);
   Write('Вы назвали город #' + lenChain.ToString() + ' ');
   TextColor(LightGreen);
   WriteLn(gorod);
```

Ответный ход за *компьютером*. Ему, конечно, ход сделать проще, поскольку весь список слов в его полном распоряжении.

Он просматривает список от начала до конца и проверяет каждое слово в функции *test*, которой мы передаём номер слова в списке.

```
  //ХОД КОМПЬЮТЕРА

   //просматриваем список слов с начала:
```

```
   var ic: integer;
   For ic:= 1 to nWords do
   begin
      //подходит ли оно?
      If test(ic) then
         Goto c_yes; //компьютер нашёл слово
   end;//for
   //компьютер не нашел продолжения:
   Goto endGame;
```

Проверки в функции *test* ничем не отличаются от проверок слова игрока, за маленьким исключением: компьютер всегда выбирает слова из списка, поэтому одна проверка была бы лишней:

```
//Проверка: можно ли поставить очередное
//слово в цепочку
function test(n: integer): boolean;
begin
  //n - Номер слова в списке:
  Result:= true;

  //слово уже стоит в цепочке:
  If (flgChain[n]) Then begin
    Result:= False;
    exit;
  End;//If

  //первая буква проверяемого слова должна совпадать
  //с конечной буквой предыдущего слова:
  If (lenChain > 0) and (spisok[n][1] <> endLetter) Then
  begin
    Result:= False;
    exit;
  End;//If
end;
```

Если проверка завершилась неудачно, значит, в списке не осталось подходящих слов. Компьютер сообщает о своем поражении, после чего начинается новая игра:

```
  //игра закончена:
endGame:
```

```
   TextColor(LightRed);
   WriteLn;
   WriteLn('Признаю своё поражение!');
   WriteLn;
   //начинаем новую игру:
   Goto startGame;
end.
```

Но, скорее всего, этого не случится, и компьютер сделает свой ход:

```
  //компьютер выполняет ход:
c_yes:
   gorod:= spisok[ic];
   flgChain[ic]:= true;
   lenChain += 1;
   len:= spisok[ic].Length;
   endLetter:= spisok[ic][len];
   //если слово заканчивается на "плохие" буквы,
   //то берем предыдущую:
   While (len > 2) And ((endLetter= 'ы') Or (endLetter= 'ь'))
do
   begin
      len -= 1;
      endLetter:= spisok[ic][len];
   end;//while

   Write;
   TextColor(Yellow);
   Write('Я называю город  #' + lenChain.ToString() + ' ');
   TextColor(LightGreen);
   WriteLn(gorod);
```

Последствия хода компьютера ничем не отличаются от последствий, вызванных ходом игрока, вот только теперь ход следует передать игроку:

```
   //теперь ход игрока:
   goto hodIgroka;
```

А теперь – за игру (Рис. 42.4)! Надо признать, компьютер играет великолепно, но есть *уловки*, которые приведут его к поражению.

Подумайте и отыщите их. Кстати говоря, они вам пригодятся и при игре с настоящими соперниками, а не только с компьютерными.

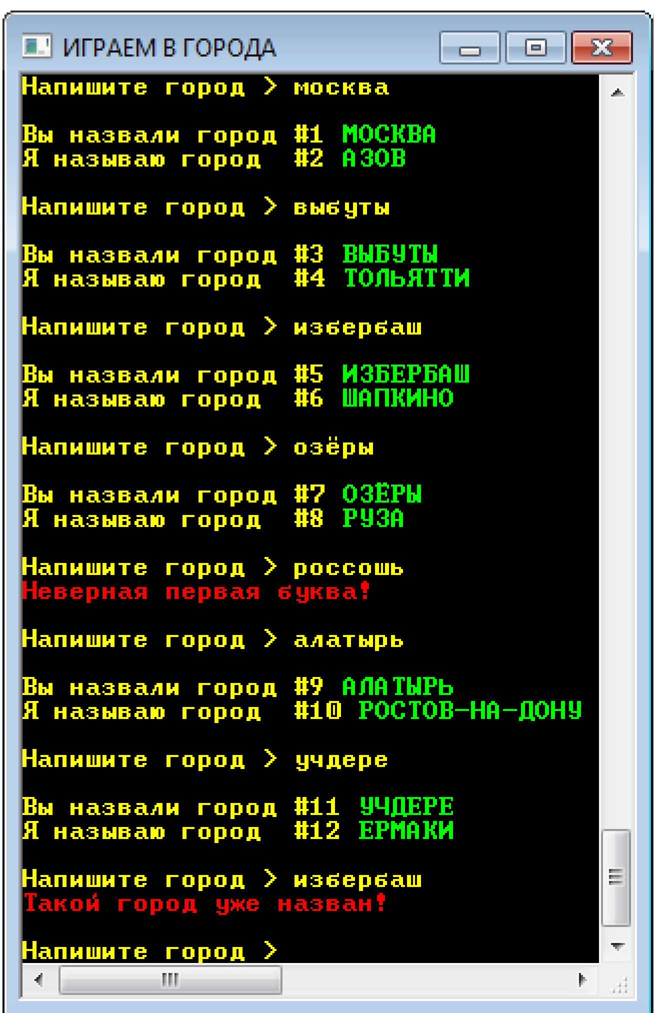


Рис. 42.4. Партия в самом разгаре.

Исходный код программы находится в папке **Города2**.

1. Вместо названий городов России можно взять европейские города или вообще любые города мира. Более того, совсем не обязательно играть только в города, это могут быть названия фруктов, овощей, птиц, названия химических элементов, иностранные слова – любые *тематические* наборы слов. Такие

игры очень помогают обогатить свой словарный запас и могут пригодиться на любых уроках.

2. Компьютер признаёт свое поражение, если не может найти продолжения цепочки. Предусмотрите в программе возможность и для игрока признать своё поражение.

3. Перепишите программу так, чтобы игра велась в *графическом окне*, что и более красиво, и более удобно.

4. Поместите в *графическом окне* многострочное *текстовое поле*, в которое можно вывести список всех слов, чтобы игрок мог выбирать следующее слово из списка.

5. Дайте игроку возможность выбирать, будет ли начинать игру он или компьютер.

6. Некоторые действия игрока и компьютера практически совпадают, поэтому выделите их в отдельные процедуры.

7. Замените компьютер вторым игроком, чтобы играть в города с друзьями-товарищами.

Урок 43. Магические квадраты

Магический квадрат – это таблица, которая состоит из равного числа строк и столбцов. Во всех её клетках записано по одному числу. Если обозначить буквой *n* число строк (столбцов) в таблице (это число называется *порядком* магического квадрата), то в обычном магическом квадрате записаны последовательные числа от *1* до n^2. Например, в самом простом магическом квадрате порядка 3 (то есть размером 3 x 3 клетки) мы найдём все числа от 1 до $3^2 = 9$ (Рис. 43.1).

Магические квадраты иначе называют *волшебными*.

Таким образом, мы всегда знаем, какие числа следует записать в таблицу. Сделать это в *произвольном* порядке, конечно нетрудно, но ведь квадрат не зря называется *магическим*! В нём сумма чисел в каждой строке, столбце и в каждой из двух главных диагоналей должна быть *одинаковой*. Эта *сумма* называется *магической* и обозначается буквой *S*.

Магическая сумма называется также *константой* магического квадрата.

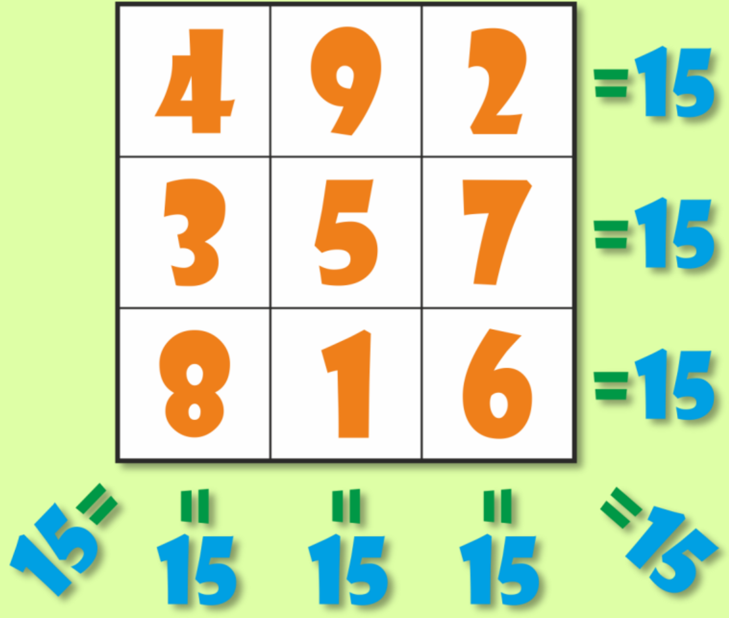


Рис. 43.1. Магический квадрат третьего порядка

Мы легко найдём магическую сумму любого квадрата, если подсчитаем сумму всех чисел в квадрате и разделим её на порядок квадрата. Последовательные натуральные числа, которые находятся в магическом квадрате, образуют *арифметическую прогрессию*, сумму членов которой мы умеем находить.

Первый член прогрессии равен *1*.
Последний член прогрессии равен n^2.
Число членов прогрессии равно n^2.

Сумму всех членов прогрессии можно вычислить по формуле:

$$sum = \frac{1 + n^2}{2} \cdot n^2 = \frac{n^2(1 + n^2)}{2}$$

А *магическую сумму* – по формуле:

$$S = \frac{n(1 + n^2)}{2}$$

История магических квадратов

Составление магических квадратов – одна из первых головоломок в истории человечества. Первый магический квадрат порядка 3 составили китайцы (Рис. 43.2) и назвали его *Ло-шу*. Легенда гласит, что люди впервые увидели его на панцире черепахи, которая выползла из реки Ло. Случилось это задолго до нашей эры, хотя дотошные историки утверждают, что квадрат Ло-шу появился не раньше четвёртого века до нашей эры.

Китайцы придавали математическим особенностям квадрата *Ло-шу* мистические свойства и считали его символом, объединяющим предметы, людей и вселенную. Чётные числа представляли женскую сущность Инь, а нечётные – мужскую – Ян (Рис. 43.3).

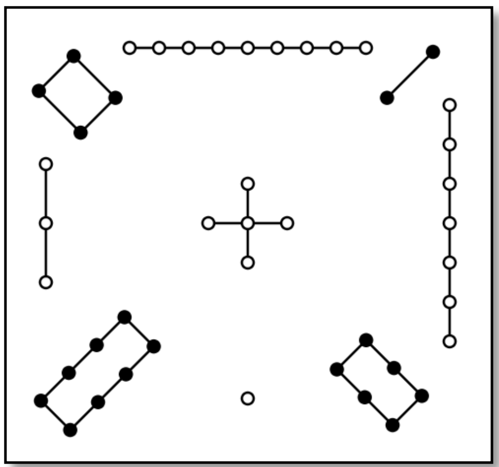


Рис. 43.2. Магический квадрат *Ло-шу* на панцире черепахи

Рис. 43.3. Знаменитый символ Инь-Ян

В центре *Ло-шу* находится пятёрка, которая символизирует Землю (Рис. 43.4). Вокруг неё располагаются элементы (стихии). Металл, представленный числами 4 и 9, вода – 1 и 6, огонь – 2 и 7, дерево 3 и 8. Легко заметить, что каждый элемент содержит мужское и женское начала *Инь* и *Ян*.

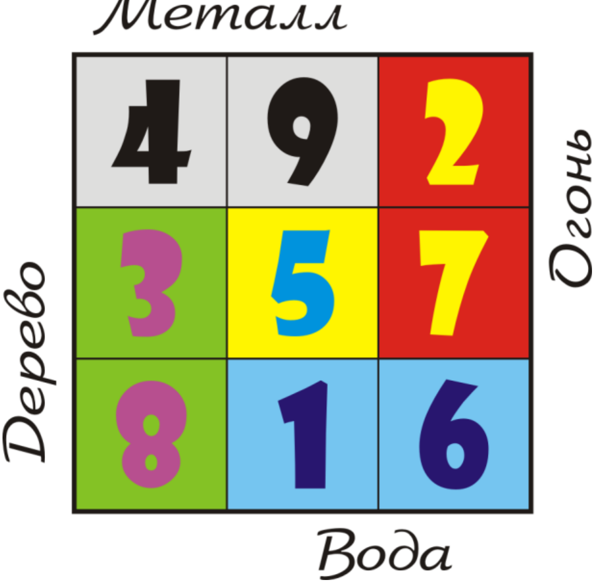


Рис. 43.4. Элементы природы в квадрате *Ло-шу*

Из Китая магические квадраты попали в Индию, а затем - через арабские страны – в Европу. В эпоху Ренессанса немецкий математик Генрих Корнелиус Агриппа (1486-1536) построил магические квадраты от третьего до девятого порядка и дал им астрономическое толкование.

Наименьшим магическим квадратом считается *Ло-шу*, поскольку квадрат, состоящий всего из одной клетки, трудно назвать магическим, а магических квадратов второго порядка вообще не существует, в чём вы легко сможете убедиться сами, если попробуете их составить.

Они представляли собой семь известных тогдашней науке «планет»: Сатурн, Юпитер, Марс, Солнце, Венеру, Меркурий и Луну.

Но создание первого «европейского» магического квадрата приписывают немецкому художнику, уроженцу города Нюрнберга, современнику Леонардо да Винчи Альбрехту Дюреру. На знаменитой гравюре *Меланхолия* (Рис. 43.5) в правом верхнем углу он поместил магический квадрат четвёртого порядка. Причём Дюрер составил его так ловко, что два средних числа в нижней строке образовали год создания произведения - **1514** (Рис. 43.6).

После *Ло-шу* это самый известный магический квадрат в мире!

Конечно, Дюрер украсил свою гравюру магическим квадратом не только для того, чтобы указать год. Дело в том, что в то время квадраты четвёртого порядка считались хорошим терапевтическим средством, и астрологи «прописывали» их для амулетов против меланхолии.

Необычным свойствам магических квадратов люди с давних времен находили применение в мистике и религии. Вырезанные на дереве, камне или металле магические квадраты служили амулетами (в этом качестве они до сих пор находят применение в восточных странах). Ещё в 16-17 веках люди верили, что выгравированный на серебряной пластине магический квадрат может защитить их от чумы.

Арабские астрологи более девяти веков назад использовали магические квадраты при составлении гороскопов.

Рис. 43.5. Дюрер. *Меланхолия* **Рис. 43.6.** Магический квадрат крупным планом

А сколько всего магических квадратов?

Мало ли в Бразилии Педров? -
И не сосчитаешь!

Фраза тётушки Чарли
из комедии *Здравствуйте, я ваша тётя*

Мы уже знаем, что магические квадраты первого и третьего порядков существуют в единственном числе, а квадратов второго порядка вообще нет.

Мы считаем только *уникальные* квадраты, которые нельзя получить из других с помощью поворотов и отражений.

Французский математик Бернар де Бесси (Bernard Frénicle de Bessy, 1605 – 1675) подсчитал, что магических квадратов четвёртого порядка существует 880 штук.

Долгое время учёные оценивали число магических квадратов пятого порядка в 13 миллионов, пока в 1973 году американский программист Ричард Шрёппель (Richard Schroeppel) с помощью компьютера не нашёл их точное число - 275 305 224.

Сколько существует квадратов шестого порядка, до сих пор неизвестно, но их примерно 1.77 x 10^{19}. Число огромное, поэтому нет никаких надежд пересчитать их с помощью полного перебора, а вот формулы для подсчёта магических квадратов никто придумать не смог.

Как составить магический квадрат?

Существует очень много способов построения магических квадратов. Проще всего смастерить магические квадраты *нечётного порядка*. Мы воспользуемся очень простым методом, который предложил французский учёный XVII века А. де ла Лубер (De La Loubère). Он основан на *пяти* правилах, действие которых мы рассмотрим на самом простом магическом квадрате 3 x 3.

Правило 1. Поставить 1 в среднюю колонку первой строки (Рис. 43.7).

	1	

Рис. 43.7. *Первое* число

Правило 2. Следующее число поставить, если возможно, в клетку, соседнюю с текущей по диагонали правее и выше (Рис. 43.8).

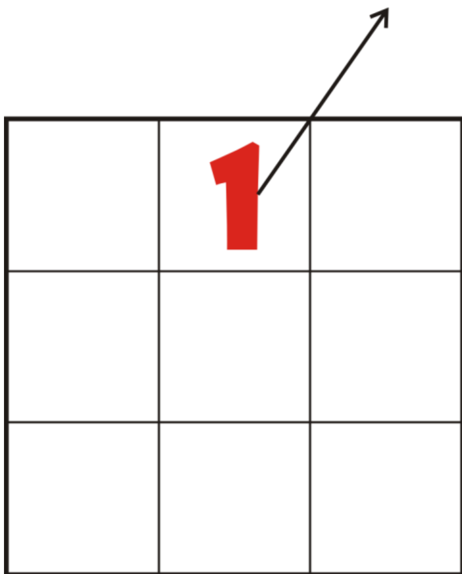


Рис. 43.8. Пытаемся поставить второе число

Правило 3. Если новая клетка выходит за пределы квадрата *сверху*, то число записывается в самую нижнюю строку и в следующую колонку (Рис. 43.9).

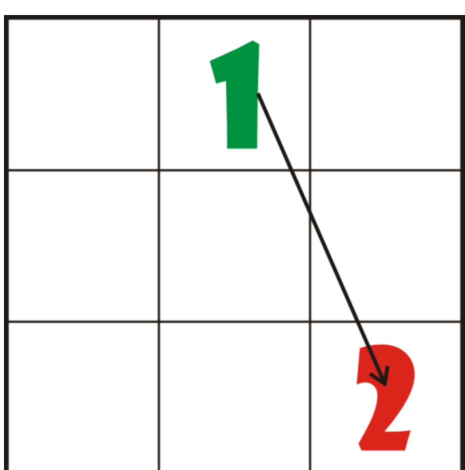


Рис. 43.9. Ставим *второе* число

Правило 4. Если клетка выходит за пределы квадрата *справа*, то число записывается в самую первую колонку и в предыдущую строку (Рис. 43.10).

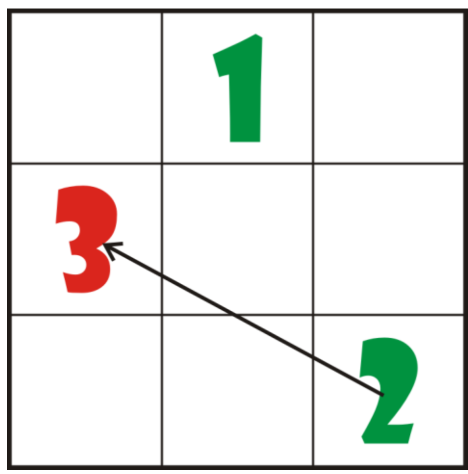


Рис. 43.10. Ставим *третье* число

Правило 5. Если в клетке уже стоит число, то очередное число записывается под текущей клеткой (Рис. 43.11).

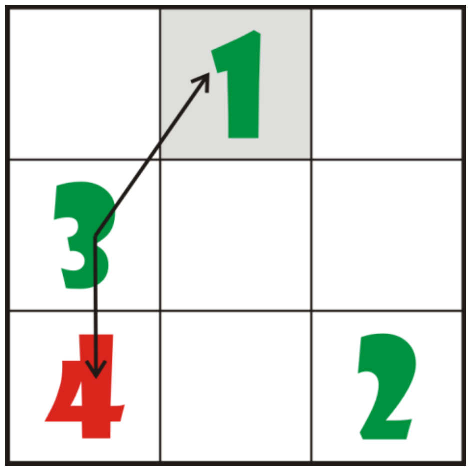


Рис. 43.11. Ставим *четвёртое* число

Далее переходим к *Правилу 2* (Рис. 43.12).

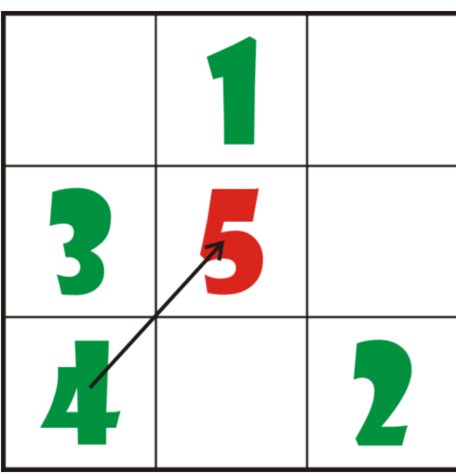


Рис. 43.12. Ставим *пятое* и *шестое* число

Снова выполняем *Правила 3, 4, 5*, пока не составим весь квадрат (Рис. 43.13).

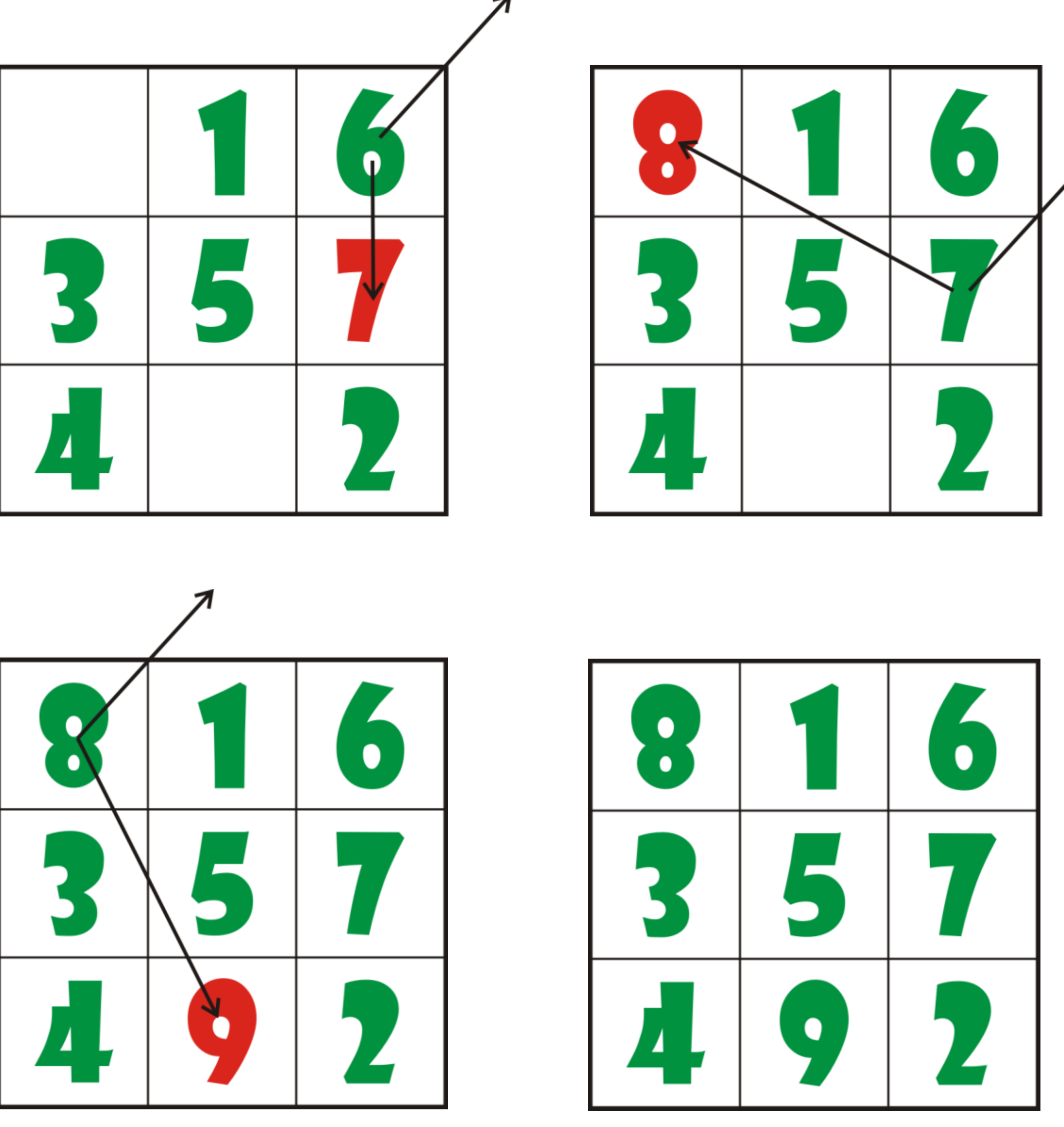


Рис. 43.13. Заполняем квадрат следующими числами

Не правда ли, правила очень простые и понятные, но всё равно довольно утомительно расставлять даже 9 чисел. Однако, зная алгоритм построения магических квадратов, мы легко можем поручить компьютеру всю рутинную работу, оставив себе только творческую, то есть написание программы.

Набор *переменных* для программы **Магические квадраты** совершенно очевиден:

```
//ПРОГРАММА ДЛЯ ГЕНЕРИРОВАНИЯ
//НЕЧЁТНЫХ МАГИЧЕСКИХ КВАДРАТОВ
```

```
//ПО МЕТОДУ ДЕ ЛА ЛУБЕРА

uses CRT;

var
   //порядок квадрата:
   n:=0;
   //текущее число для записи в квадрат:
   number:=0;
   //текущая колонка:
   col:=0;
   //текущая строка:
   row:=0;
   //магический квадрат:
   mq: array[,] of integer;
```

Метод де ла Лубера годится для *нечётных* квадратов *любого* размера, поэтому мы можем предоставить пользователю самостоятельно выбрать порядок квадрата. Однако, дав ему свободу, мы должны проверить результаты его деятельности («доверяй, но проверяй!»).

Физические размеры квадратов большого порядка превышают возможности *консольного окна* по ширине, поэтому подумайте, как организовать вывод готовых квадратов в *файл* или в *графическое окно*.

Проверки очень простые, но необходимые:

```
//==========================================
//          ОСНОВНАЯ ПРОГРАММА
//==========================================
begin
   SetWindowTitle('Нечётные магические квадраты');

   while(true ) do
   begin
      repeat
         TextColor(Yellow);
         write('Введите порядок квадрата 3..27 > ');
         TextColor(CRT.LightGreen);
         n := readInteger;
```

```
        //Если задан нуль,
        //то работу с программой заканчиваем:
        if (n = 0) then exit;
    until ((n >= 3) and (n <= 27));

    if (n mod 2 <> 1) then
    begin
        TextColor(LightRed);
        writeln('Число должно нечётным!');
        writeln;
        continue;
    End;//If
```

Если пользователь попался сознательный, то в переменной *n* мы получим *порядок* магического квадрата.

В *паскале* при создании нового динамического массива все его элементы *обнуляются* автоматически. В других реализациях паскаля и в других языках программирования этот «сервис» может и отсутствовать. Поэтому обязательно проверяйте содержимое нового массива, иначе в массиве окажутся совершенно случайные числа, которые погубят всю программу. Более того, при каждом запуске программы содержание массива будет другим, поэтому иногда вы будете получать правильные результаты, иногда – неправильные. Вот и попробуйте найти ошибку!

В данной программе я распечатал массив с помощью процедуры *writeMQ*, а потом закомментировал её.

```
    //Введено правильное число -->

    //создаём массив:
    mq:= new integer[n+1, n+1];
    //writeMQ();
```

Начинаем действовать по правилам де ла Лубера и записываем первое число – единицу – в среднюю клетку первой строки квадрата (или массива, если угодно):

```
     //первое число:
      number:=1;
 rule1:
    //колонка для первого числа - средняя:
    col:= Floor(n/2) + 1;
    //строка для первого числа - первая:
    row:=1;
    //заносим его в квадрат:
    mq[row,col]:= number;
    //writeMQ();
```

Теперь последовательно пристраиваем в клетки остальные числа – от двойки до $n * n$:

```
//переходим к следующему числу:
nextNumber:
   number:= number+1;
```

Запоминаем на всякий случай координаты актуальной клетки

```
  var tc:=col;
   var tr:=row;
```

и переходим в следующую клетку по диагонали:

```
  col:= col+1;
   row:= row-1;
```

Проверяем выполнение *третьего* правила:

```
rule3:
   If (row < 1) Then
      row:= n;
```

А затем *четвёртого*:

```
rule4:
    If (col > n) then
    begin
      col:=1;
      Goto rule3;
    end;//If
```

И *пятого*:

```
rule5:
   If (mq[row,col] <> 0) Then
   begin
      col:=tc;
      row:=tr+1;
      Goto rule3;
   End;//If
```

Как мы узнаем, что в клетке квадрата уже находится число? – Очень просто: мы предусмотрительно записали во все клетки *нули*, а все числа в готовом квадрате *больше нуля*. Значит, по содержимому ячейки массива мы сразу же определим, пустая она или с числом! Обратите внимание, что здесь нам понадобятся те координаты клетки, которые мы запомнили перед поиском клетки для следующего числа.

Попробуйте иначе организовать проверку допустимости перехода в новую клетку!

Рано или поздно мы найдём подходящую клетку для числа и запишем его в соответствующую ячейку массива:

```
   //заносим его в квадрат:
   mq[row,col]:= number;
   //writeMQ();
```

Если это число было последним, то программа свои обязанности выполнила, иначе она добровольно переходит к обеспечению клеткой следующего числа:

```
   //если выставлены все числа, то
   //квадрат составлен,
   //иначе переходим к следующему числу:
   If (number < n*n) Then
      Goto nextNumber;
```

И вот квадрат готов! Вычисляем магическую сумму и распечатывает квадрат в *консольном окне*:

```
      //построение квадрата закончено:
       writeln;
       writeln('Магическая сумма = ' + ((n*n*n +n) div 2).ToString());
       writeMQ();

       writeln;
       TextColor(Yellow);
    end;
end.
```

Напечатать элементы массива очень просто, но важно учесть выравнивание чисел разной «длины», ведь в квадрате могут быть одно-, дву- и трёхзначные числа:

```
   //Печатаем готовый квадрат
   procedure writeMQ;
   begin
      writeln;
      TextColor(LightGreen);
      //печатаем магический квадрат:
      For var i:= 1 To n do
      begin
         var s:='';
         For var j:= 1 To n do
         begin
            If (n*n > 10) And (mq[i,j] < 10) Then
                s := s + ' ';
            If (n*n > 100) And (mq[i,j] < 100) Then
                s := s + ' ';
            s += mq[i,j] + ' ';
         End;//For
         writeln(s);
      End;//For

      writeln;
   end;
```

Запускаем программу – а ведь недаром мы трудились: квадраты получаются быстро и на загляденье (Рис. 43.14).

Нечётные магические квадраты

```
Введите порядок квадрата 3..27 > 19

Магическая сумма = 3439

192 213 234 255 276 297 318 339 360   1  22  43  64  85 106 127 148 169 190
212 233 254 275 296 317 338 359  19  21  42  63  84 105 126 147 168 189 191
232 253 274 295 316 337 358  18  20  41  62  83 104 125 146 167 188 209 211
252 273 294 315 336 357  17  38  40  61  82 103 124 145 166 187 208 210 231
272 293 314 335 356  16  37  39  60  81 102 123 144 165 186 207 228 230 251
292 313 334 355  15  36  57  59  80 101 122 143 164 185 206 227 229 250 271
312 333 354  14  35  56  58  79 100 121 142 163 184 205 226 247 249 270 291
332 353  13  34  55  76  78  99 120 141 162 183 204 225 246 248 269 290 311
352  12  33  54  75  77  98 119 140 161 182 203 224 245 266 268 289 310 331
 11  32  53  74  95  97 118 139 160 181 202 223 244 265 267 288 309 330 351
 31  52  73  94  96 117 138 159 180 201 222 243 264 285 287 308 329 350  10
 51  72  93 114 116 137 158 179 200 221 242 263 284 286 307 328 349   9  30
 71  92 113 115 136 157 178 199 220 241 262 283 304 306 327 348   8  29  50
 91 112 133 135 156 177 198 219 240 261 282 303 305 326 347   7  28  49  70
111 132 134 155 176 197 218 239 260 281 302 323 325 346   6  27  48  69  90
131 152 154 175 196 217 238 259 280 301 322 324 345   5  26  47  68  89 110
151 153 174 195 216 237 258 279 300 321 342 344   4  25  46  67  88 109 130
171 173 194 215 236 257 278 299 320 341 343   3  24  45  66  87 108 129 150
172 193 214 235 256 277 298 319 340 361   2  23  44  65  86 107 128 149 170


Введите порядок квадрата 3..27 >
```

Рис. 43.14. Изрядный квадратище!

Исходный код программы находится в папке **Магические квадраты.**

Оператор безусловного перехода *goto*

В программе мы не раз пользовались оператором *goto*, который может передать управление любому оператору, которому предшествует *метка*, однако область его действия ограничивается той подпрограммой (или основной программой), в которой он находится. Невозможно перейти из одной подпрограммы на метку в другой подпрограмме.

Метка – это любой допустимый идентификатор, после которого в коде программы ставится *двоеточие*. Она указывает на тот оператор, к которому перейдёт управление после выполнения оператора *goto метка;*

Именем метки может быть и любое натуральное число. Очевидно, это дань ранним версиям бейсика, в которых номера строк использовались в операторах *goto*.

В нашей программе мы объявили все необходимые метки в разделе описаний после переменных, предварив их ключевым словом *label* (по-английски - *метка*):

```
  //метки:
  label rule1, rule3, rule4, rule5, nextNumber;
```

А затем расставили их в нужных местах программы.

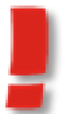
Обычно использование операторов *goto* в программе не приветствуется, поскольку они нарушают логику работы программы, перескакивая в произвольное место кода.

Когда таких переходов много, то получается запутанный код, который программисты остроумно называют *спагетти*. Подобную картину можно часто наблюдать в программах, написанных на классическом бейсике.

Почему же мы нарушили это неписаное правило в своей программе? – Ответ простой: всему виной алгоритм составления магических квадратов по методу де ла Лубера. Он описан так, что его проще всего реализовать на паскале с помощью операторов *goto*. Естественно, наша программа получилась ничуть не запутаннее самого алгоритма, поскольку, по сути, она – всего лишь другая форма записи алгоритма.

При этом программа получилась ясной для понимания и отлично работающей. Это значит, что не следует слепо отказываться от применения оператора *goto* в своих программах. Кстати говоря, в большинстве современных языков программирования он присутствует, что свидетельствует о его необходимости. Но использовать его нужно правильно и осторожно! Так, для выхода из процедур и функций используйте оператор *exit*, а для выхода из одиночных циклов - оператор *break*. И только для выхода из

вложенных циклов лучше применить оператор *goto*. Но никогда не пытайтесь запрыгнуть извне *внутрь* цикла!



Попробуйте переписать программу *Магические квадраты* без операторов *goto*!

Магические ферзи

Магический квадрат седьмого порядка, составленный по методу де ла Лубера (Рис. 43.15), как показал американец Клиффорд Пиковер, непостижимым образом связан с другой знаменитой задачей – расстановкой ферзей на шахматном поле.

Поскольку метод де ла Лубера позволяет строить только нечётные квадраты, а шахматная доска имеет чётный порядок, то придётся взять квадрат, наиболее близкий к шахматной доске, то есть 7 х 7 клеток. Мы его легко построим с помощью нашей программы.

```
Нечётные магические квадраты

Введите порядок квадрата 3..27 > 7

Магическая сумма = 175

30 39 48  1 10 19 28
38 47  7  9 18 27 29
46  6  8 17 26 35 37
 5 14 16 25 34 36 45
13 15 24 33 42 44  4
21 23 32 41 43  3 12
22 31 40 49  2 11 20
```

Рис. 43.15. Квадрат де ла Лубера

Следующий шаг: заменяем все числа в магическом квадрате остатками от их деления на семь. При этом нулевой остаток будем считать семёркой (Рис. 43.16).

30	39	48	1	10	19	28
38	47	7	9	18	27	29
46	6	8	17	26	35	37
5	14	16	25	34	36	45
13	15	24	33	42	44	4
21	23	32	41	43	3	12
22	31	40	49	2	11	20

2	4	6	1	3	5	7
3	5	7	2	4	6	1
4	6	1	3	5	7	2
5	7	2	4	6	1	3
6	1	3	5	7	2	4
7	2	4	6	1	3	5
1	3	5	7	2	4	6

Рис. 43.16. Преобразование квадрата де ла Лубера

Новый квадрат в каждой строке и паре нисходящих диагоналей содержит решение для задачи с ферзями. Например, из первой строки вытекает такое решение (Рис. 43.17).

2	4	6	1	3	5	7
3	5	7	2	4	6	1
4	6	1	3	5	7	2
5	7	2	4	6	1	3
6	1	3	5	7	2	4
7	2	4	6	1	3	5
1	3	5	7	2	4	6

Рис. 43.17. Строки дают решение задачи с ферзями

Одна из главных диагоналей квадрата - нисходящая. Мы можем использовать её для решения задачи о ферзях, но мы рассмотрим другой пример - когда нисходящая диагональ является *побочной*, поэтому должна быть *продолжена* так, чтобы в ней оказалось ровно семь клеток (Рис. 43.18).

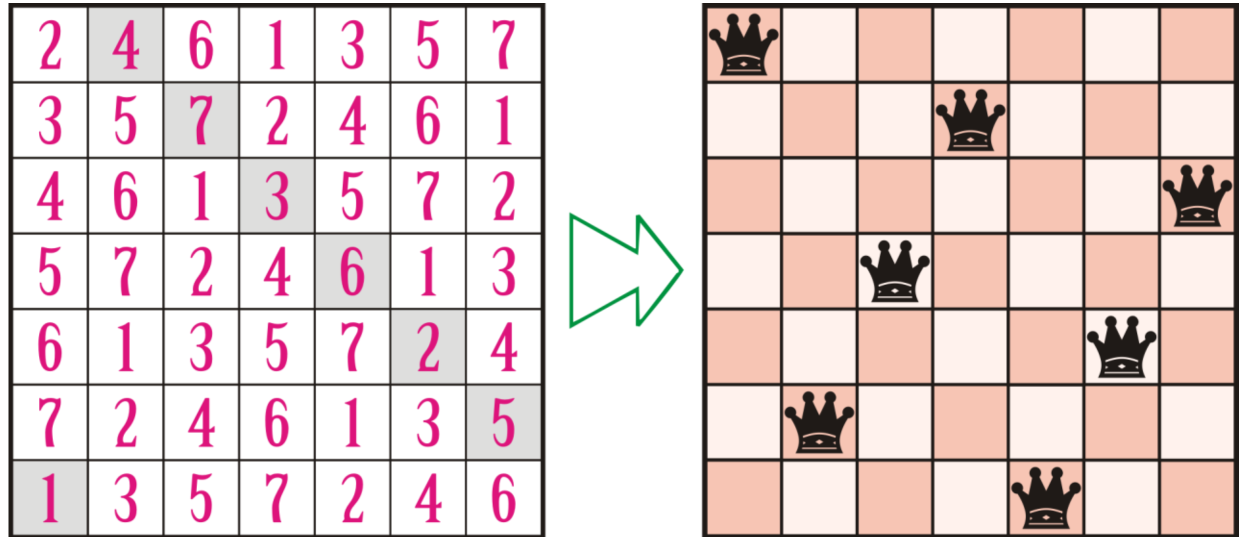


Рис. 43.18. Нисходящие диагонали дают решение задачи с ферзями

Магические квадраты чётного порядка

Теперь давайте составим магический квадрат четвёртого порядка. Один из таких квадратов и изображён на гравюре Дюрера.
Для удобства нарежьте из бумаги 16 квадратиков и напишите на них числа от 1 до 16. Это самая трудная и ответственная часть задания! Дальше будет легче, особенно если вам приходилось складывать картинки из пазлов.
Разложите бумажки квадратиком так, чтобы числа следовали друг за другом по ранжиру, то есть соблюдая субординацию (Рис. 43.19).

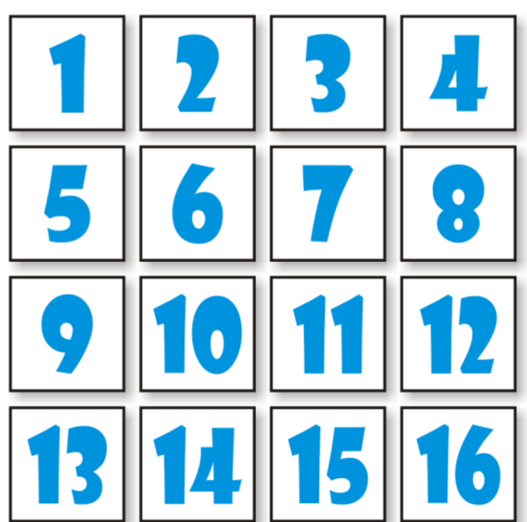


Рис. 43.19. Приняли исходное положение

Шаг 1. Поменяйте местами *вторую* и *третью* строки (Рис. 43.20).

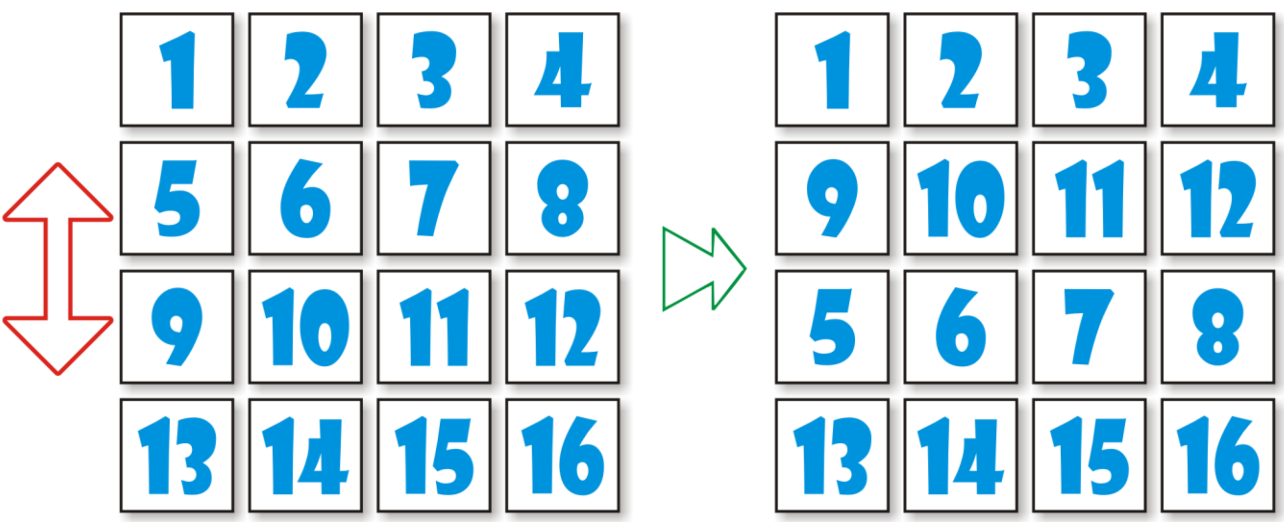


Рис. 43.20. Первый шаг

Шаг 2. Переложите числа во *второй* и *четвёртой* строках в обратном порядке (Рис. 43.21).

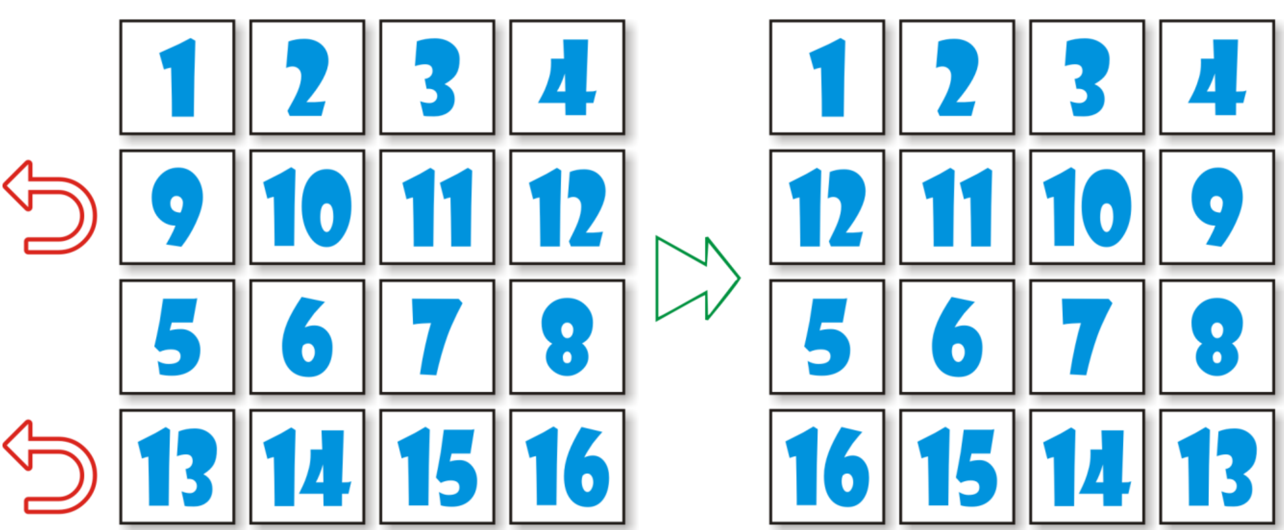


Рис. 43.21. Второй шаг

Шаг 3. Переложите числа во *втором* и *третьем* столбцах в обратном порядке (Рис. 43.22).

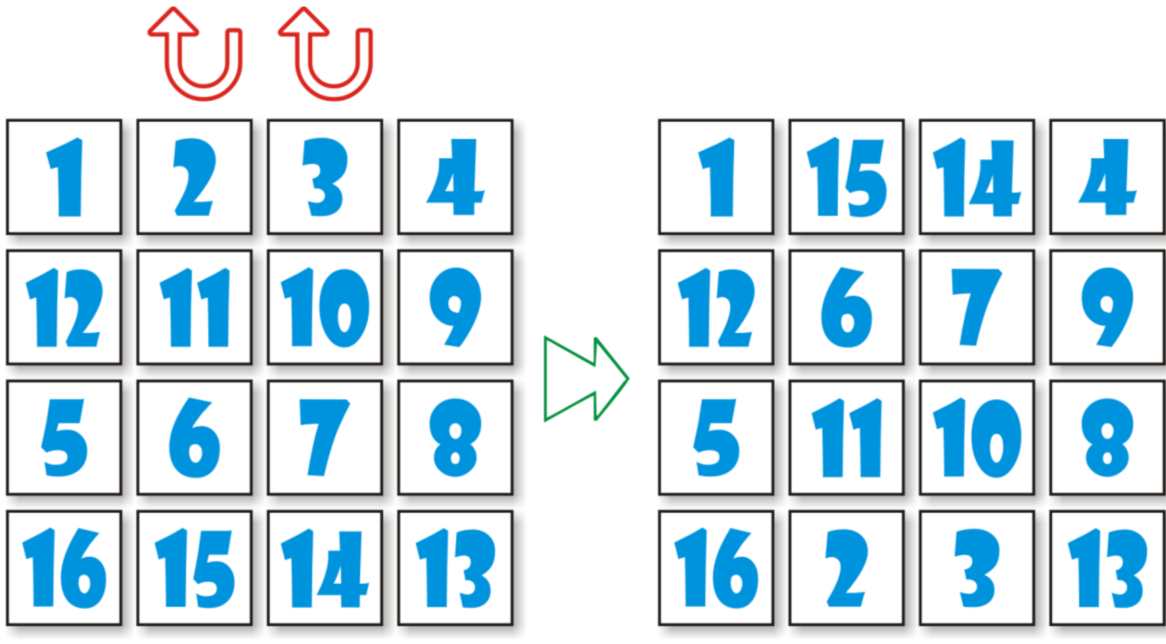


Рис. 43.22. Третий шаг

Шаг 4. Переложите числа в *третьей* и *четвёртой* строках в обратном порядке (Рис. 43.23).

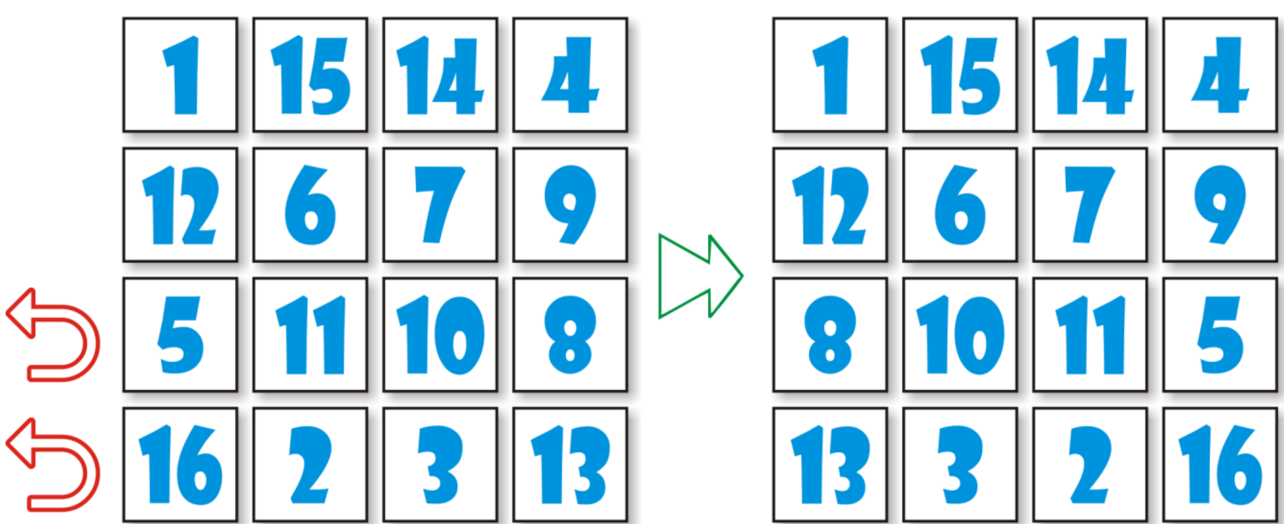


Рис. 43.23. Четвёртый шаг

Магический квадрат готов! И почти, как у Дюрера, - только год стоит не в последней, а в первой строке. Впрочем, вы можете перевернуть квадрат вверх ногами, чтобы дата оказалась внизу. Осталось переставить *первый* и *четвёртый* столбцы – и наш квадрат превращается в «дюреровский» (Рис. 43.24).

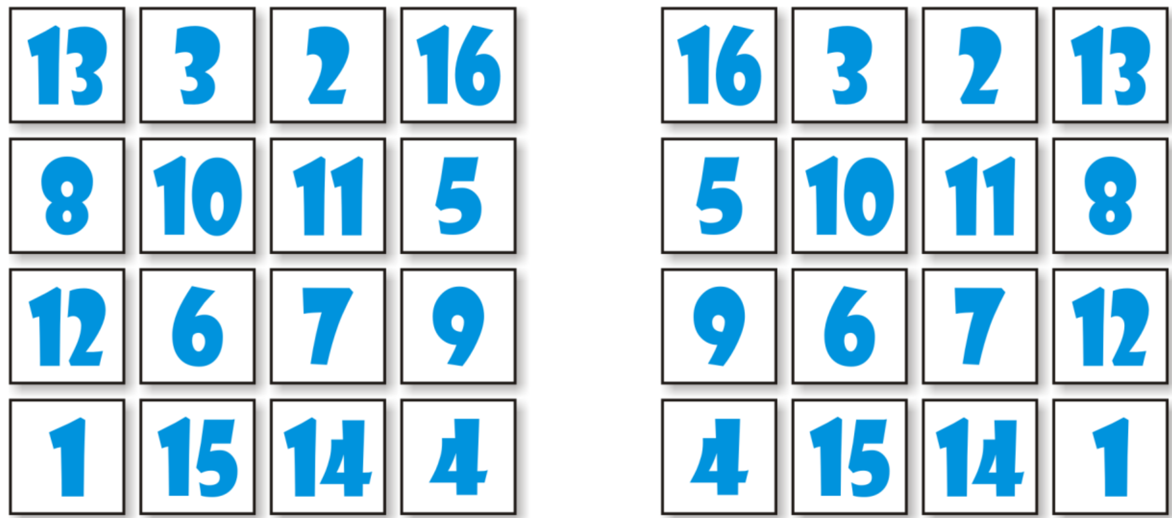


Рис. 43.24. Смастерили Дюреровский квадрат!

Мы не знаем, как именно построил Дюрер свой магический квадрат, но, возможно, и «нашим» способом.

Кстати говоря, в магическом квадрате, который мы построили, *больше* магических сумм, чем «требуется». Магическая сумма, которая в квадратах четвёртого порядка равна 34, повторяется не только в четырёх строках, четырёх столбцах и двух диагоналях, но и

- в пяти квадратиках 2 х 2 клетки (Рис. 43.25).

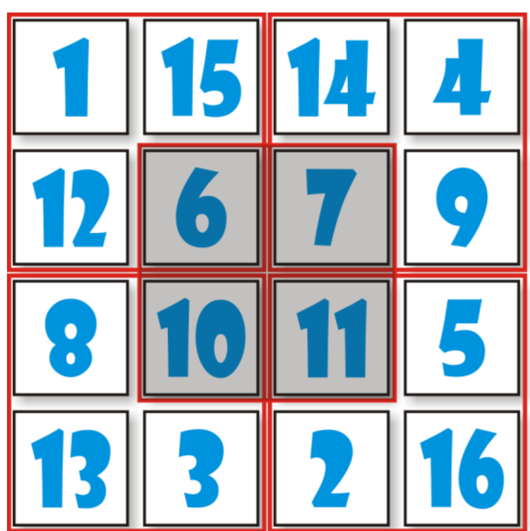


Рис. 43.25. Дополнительные магические квадратики 2 х 2

- в углах четырёх квадратов 3 х 3 клетки (Рис. 43.26).

Рис. 43.26. Дополнительные магические квадраты 3 х 3

- в углах двух прямоугольников 2 х 4 клетки (Рис. 43.27).

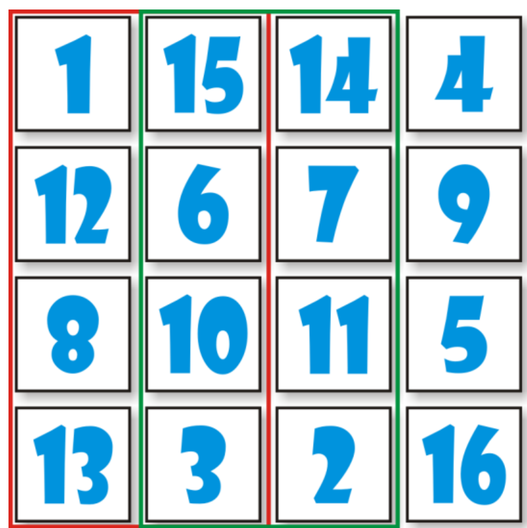


Рис. 43.27. Дополнительные магические прямоугольники 2 х 4

- в углах самого квадрата 4 х 4 клетки (Рис. 43.28).

Итого – *22* раза. Но, оказывается, и это не предел. До предела мы дойдём, если выполним ещё два шага.

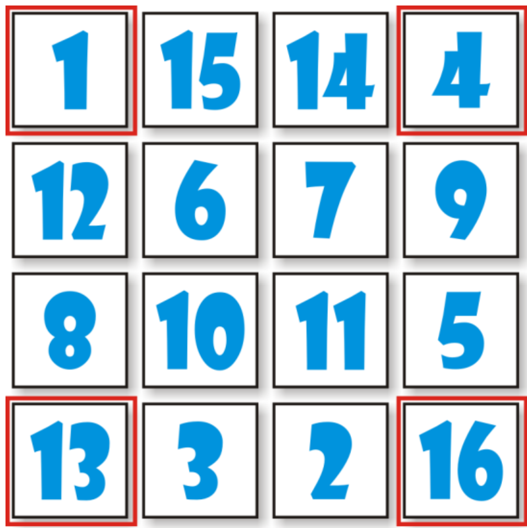


Рис. 43.28. Дополнительные магические прямоугольники 2 х 4

Шаг 5. Временно уберите *третью* и *четвёртую* строки. Переложите *вторую* строку на место *четвёртой*, а затем верните *третью* и *четвёртую* строки на свободные места. То есть *третья* строка займёт в квадрате место *второй*, а *четвёртая* – *третьей* (Рис. 43.29).

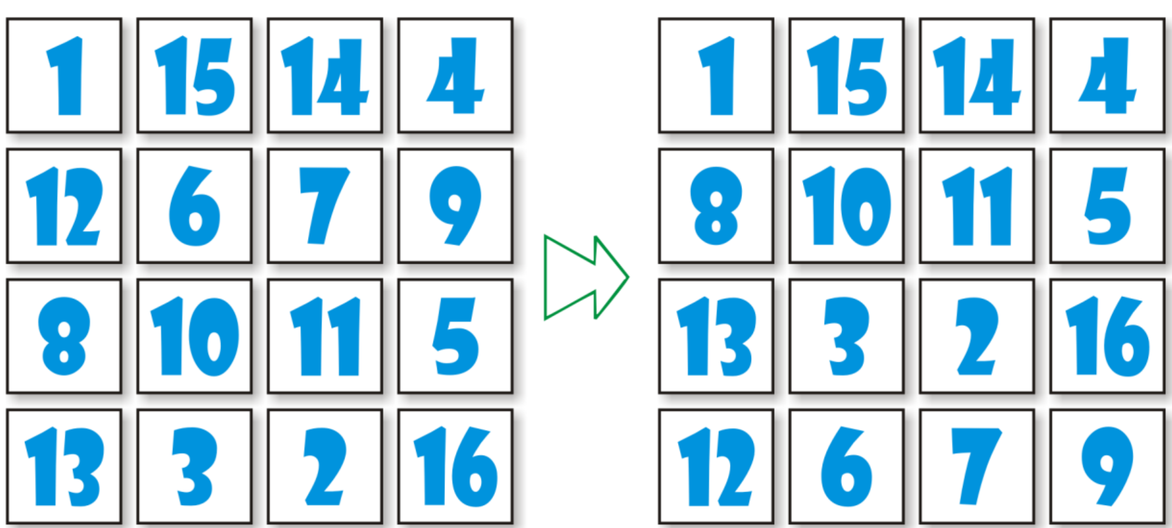


Рис. 43.29. Пятый шаг

Шаг 6. Поменяйте местами *третий* и *четвёртый* столбцы (Рис. 43.30).

Легко заметить, что все числа, кроме первых двух, заняли в квадрате *другие* места, и теперь магическая сумма повторяется уже *30*

раз. Конечно, в этом квадрате сохранились все 22 магические суммы прежнего квадрата, но к ним добавились еще 11:

- в углах девяти (а не пяти) квадратов 2 x 2 клетки;
- в углах шести (а не двух) прямоугольников 2 x 4 клетки.

Найдите самостоятельно *все* магические суммы!

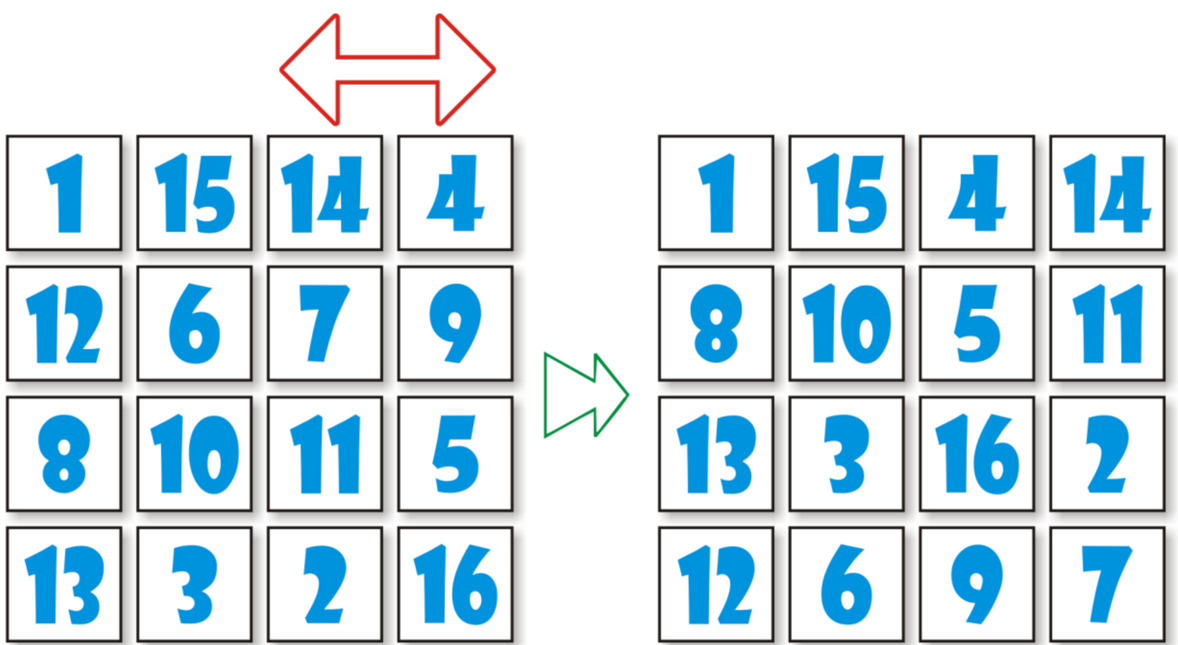


Рис. 43.30. Шестой шаг

1. С помощью программы *Магические квадраты* вы можете составить квадраты любого размера, но только по одному для каждого порядка. Увы, все подобные алгоритмы действуют именно так. Но мы всё-таки можем добавить некоторое разнообразие нашим квадратам. Например, если прибавить или вычесть одно и то же число (или умножить на одно и то же число), то квадрат останется магическим, а его магическая сумма изменится. Научите программу составлять такие квадраты.

2. Можно также поворачивать квадрат на 90 градусов и отражать его относительно горизонтальной и вертикальной осей, как мы это делали с ферзями. Поскольку эти операции проводятся над уже готовыми квадратами, то это задание совсем несложное.

Урок 44. (PascalABC.NET or C#) or (PascalABC.NET and C#)

end. – программе конец!

Программистская пословица

Не покидая среды разработки *PascalABC.NET*, вы можете писать приложения и на профессиональном языке программирования *C#* (*Си-шарп*), который лежит в основании всей платформы *.NET*!

Для примера мы переведём с паскаля на Си-шарп несколько программ из этой книги. Надеюсь, они убедят вас, что это совсем несложно.

Начните новый проект и сохраните его в папке **Rome** с расширением *.cs*, по которому *ИСР* определит, что программа написана не на паскале, а на Си-шарпе.

Для проектов на Си-шарпе я завёл новую папку *_ProjectsCS*, что и вам советую сделать незамедлительно.

По названию проекта вы, конечно, догадались, что наш первый проект связан с *римскими числами*.

Мы рассмотрим только *консольные* приложения, так как приложения *Windows Forms* более сложны и не имеют никаких преимуществ при изучении языка Си-шарп.

Все консольные приложения начинаются с выполнения главного метода *Main*.

Си-шарп различает ПРОПИСНЫЕ и строчные буквы, поэтому идентификаторы *Main* и *main* – разные! В паскале это не так, поэтому будьте внимательны!

Модификатор *static* в заголовке метода *Main* означает, что этот метод *статический*. Все методы, константы и переменные в консольном приложении также должны быть статическими.

Любой метод, в том числе и *Main* должен принадлежать какому-то классу, поэтому мы должны определить класс *Rome* и описать в нём все методы, константы и переменные, которые называются *членами* класса:

```
//ПРОГРАММА ДЛЯ ПЕРЕВОДА АРАБСКИХ
//ЧИСЕЛ В РИМСКИЕ

using System;

class Rome
{
   //const
   static readonly  string[] ROME = {"M", "CM", "D", "CD",
"C", "XC",
                                     "L", "XL", "X", "IX",
"V", "IV", "I"};
   static readonly  int[]  ARABIC = {1000, 900, 500, 400, 100,
90, 50,
                                      40, 10, 9, 5, 4, 1};

   //var
   static int number=0;
   static string sNumber="";
   static int n=0;
```

При объявлении *переменных* (полей) в Си-шарпе ключевое слово *var* указывать не нужно, а тип переменной *предшествует* идентификатору. Будьте внимательны с оператором присваивания – в Си-шарпе он состоит только из знака равенства, без двоеточия!

Перед идентификатором констант следует записывать модификатор *const*, но если вы взглянете на описание массивов констант, то они начинаются с ключевого слова *readonly*. А всё дело в том, что в Си-шарпе запрещено объявлять *массивы* констант с ключевым словом *const*.

Как видите, массивы в Си-шарпе также объявляются иначе, чем в паскале, а строковые литералы необходимо брать в *двойные* кавычки.

Переходим к основной части программы, которая находится в методе *Main*:

```
   //==========================================
   //           ОСНОВНАЯ ПРОГРАММА
   //==========================================
   public static void Main()
   {
      Console.Title = "Перевод арабских чисел в римские";
      while(true)
      {
         do
         {
            Console.ForegroundColor = ConsoleColor.Green;
            Console.WriteLine("Введите арабское число
1..3999");
            Console.ForegroundColor = ConsoleColor.Yellow;
            number = Convert.ToInt32(Console.ReadLine());
            //Если задан нуль, то работу с программой заканчиваем:
            if (number == 0) return;
         }
         while ((number < 1) || (number > 3999));

         //Формируем строку с римским числом,
         //равным заданному числу number:
         n = 0;
         sNumber = "";
         while (number > 0)
         {
            while (ARABIC[n] <= number)
            {
               sNumber += ROME[n];
               number -= ARABIC[n];
            }
            ++n;
         }
         Console.ForegroundColor = ConsoleColor.Red;
         Console.WriteLine(sNumber);
         Console.WriteLine();
      }
   }
}
```

Операции ввода-вывода производятся с помощью методов статического класса *Console*, который находится в пространстве имён *System*. Пространства имён подключаются к проекту аналогично модулям в паскале, но с ключевым словом *using*:

```
using System;
```

Поскольку консольные приложения на *паскале* также используют этот класс платформы *.NET*, то останавливаться на нём мы не будем.

Вы, безусловно, заметили множество фигурных скобок, которые играют в Си-шарпе ту же роль, что и операторные скобки *begin-end* в паскале – они заключают в себе блок операторов:

```
{        - begin

}        - end
```

В Си-шарпе мы найдём те же три оператора циклов – *while, for* и *do-while* (аналог *repeat-until*). Но все условия выполнения циклов должны быть заключены в круглые скобки, а при составлении сложных логических выражений используют менее наглядные знаки, чем в паскале:

```
||       - or
&&       - and
```

Все методы в Си-шарпе – это *функции*, поэтому должны возвращать значение заданного типа. Оно указывается после ключевого слова *return*. Процедурам паскаля в Си-шарпе соответствуют функции, которые возвращают значение типа *void*, то есть ничего. Метод *Main* имеет этот тип, поэтому из него можно выйти с помощью оператора *return* без возвращаемого значения:

```
if (number == 0) return;
```

Запускать программу на выполнение нужно клавишами *Shift+F9*, то есть без связи с оболочкой. Рис. 44.1 ясно показывает, что, не-

смотря на разные языки программирования, приложение работает точно так же, как и раньше!

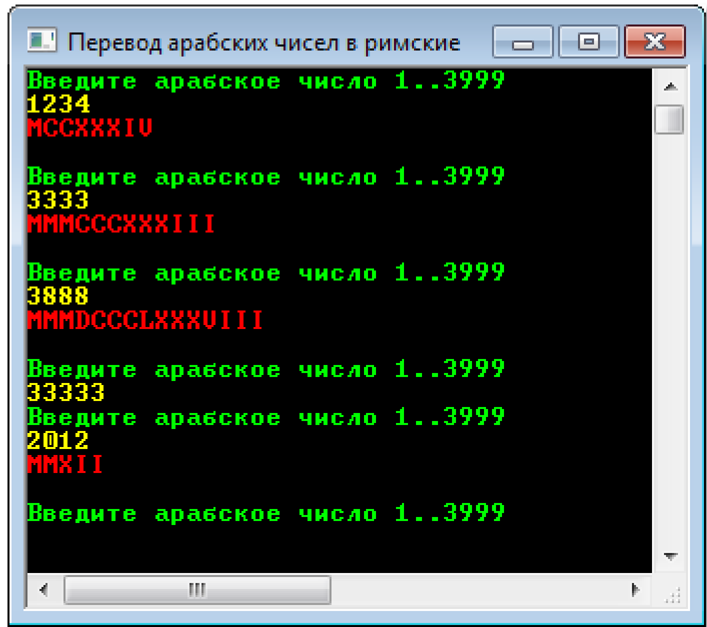


Рис. 44.1. Переводим арабские числа в римские в Си-шарпе!

Исходный код программы находится в папке **Rome**.

Переходим к следующей программе – для вычисления *наибольшего общего делителя* двух чисел:

```
using System;

namespace NOD
{
    //ПРОГРАММА ДЛЯ НАХОЖДЕНИЯ НАИБОЛЬШЕГО
    //ОБЩЕГО ДЕЛИТЕЛЯ ДВУХ НАТУРАЛЬНЫХ ЧИСЕЛ (НОД)
    class NOD
    {
        static void Main(string[] args)
        {
            //заголовок окна:
            Console.Title = "НОД";

           //бесконечный цикл ввода данных -
           //пока пользователь не закроет программу:
           do
```

```
        {
            Console.WriteLine("");
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine("НОД двух чисел");
            Console.WriteLine("");

            Console.ForegroundColor = ConsoleColor.Yellow;
            Console.BackgroundColor = ConsoleColor.Black;

           Console.Write("Введите первое число >  ");
            Console.ForegroundColor = ConsoleColor.Green;
            long number1 = Convert.ToInt64(Console.ReadLine());
            Console.ForegroundColor = ConsoleColor.Yellow;
            Console.Write("Введите второе число >  ");
            Console.ForegroundColor = ConsoleColor.Green;
            long number2 = Convert.ToInt64(Console.ReadLine());
            //если первое число меньше второго,
            //то меняем их значения:
            if (number1 < number2)
            {
                long n= number1;
                number1=number2;
                number2 = n;
            }
            long nod = 0;

            //находим НОД:
            if (number2==0)
                nod = number1;
            else
                //nod = euklid(number1, number2);
                nod = speed_euklid(number1, number2);

            //печатаем НОД:
            Console.ForegroundColor= ConsoleColor.Green;
            Console.WriteLine("НОД = (" + number1 + ","
                                  + number2 + ") = " + nod);
            Console.WriteLine("");
            Console.ForegroundColor= ConsoleColor.Yellow;
        }
        while (true);
    } //Main
```

```
        //алгоритм Евклида
        static long euklid(long n1, long n2)
        {
            while (n2 != n1)
            {
              if (n1 >= n2) n1 -= n2;
              else n2 -= n1;
            }
            return n1;
         }

        //быстрый алгоритм Евклида
        static long speed_euklid(long n1, long n2)
        {
            while (n2 > 0){
              long n = n1 % n2;
              n1 = n2;
              n2 = n;
            }
            return n1;
        }
    }
}
```

Здесь мы ввели собственное пространство имён

```
namespace NOD
```

Для небольших проектов это излишне, поэтому вы вполне можете закомментировать эту строку – но только вместе с открывающей (самой первой) и закрывающей (самой последней) фигурными скобками!

Здесь мы добавили пару новых методов для вычисления *НОД*:

```
        //быстрый алгоритм Евклида
        static long speed_euklid(long n1, long n2)
        {
            while (n2 > 0){
              long n = n1 % n2;
              n1 = n2;
              n2 = n;
            }
```

```
            return n1;
        }
```

В паскале это была бы функция, возвращающая значение типа *int64*. В Си-шарпе этот тип следует записывать с заглавной буквы - *Int64 (*или *long)*!

Оператор % означает остаток от деления *mod*, а оператор неравенства != в паскале известен под именем <>. Других сюрпризов в этом коде нет, поэтому смело запускаем приложение и обнаруживаем полное сходство с его паскалевским предшественником (Рис. 44.2).

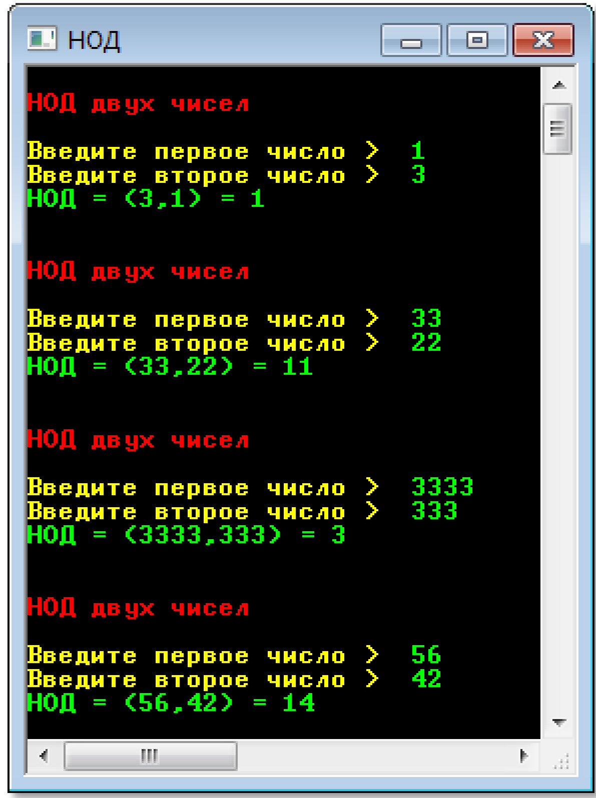


Рис. 44.2. Вычисляем *НОД* на Си-шарпе

Исходный код программы находится в папке **NOD**.

Следующей жертвой наших экспериментов я предлагаю сделать программу *Решето Эратосфена*:

```
using System;
```

```
using System.Text;
using System.IO;

namespace Resheto
{
    //РЕШЕТО ЭРАТОСФЕНА
    class Resheto
    {
        static void Main(string[] args){
            //заголовок окна:
            Console.Title = "Решето Эратосфена";

            Console.ForegroundColor = ConsoleColor.Red;
            Console.BackgroundColor = ConsoleColor.Yellow;
            Console.WriteLine("Решето Эратосфена");
            Console.WriteLine("");

            //бесконечный цикл ввода данных -
            //пока пользователь не закроет программу
            //или не введет 0:
            do
            {
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.BackgroundColor = ConsoleColor.Black;
                Console.Write("Введите конец  диапазона  >
");
                int end = Convert.ToInt32(Console.ReadLine());

                //если конец диапазона равен 0,
                //то программу закрываем:
                if (end == 0) return;
                //конец диапазона не меньше двойки:
                if (end < 2) end = 2;
                //ищем простые числа:
                primes(end);
            }
            while (true);
        } //Main

        //ИЩЕМ ПРОСТЫЕ ЧИСЛА
        static void primes(int end)
        {
            Console.WriteLine("");
            Console.ForegroundColor = ConsoleColor.Red;
```

```
            Console.WriteLine("Простые числа в заданном диапа-
зоне:");
            Console.ForegroundColor = ConsoleColor.Green;

            //создаём массив натуральных чисел 2..end:
            int[] number = new int[end+1];
            for (int i = 2; i <= end; ++i)
                number[i] = i;

            int prime = 2;

            do
            {
              for (int i = prime * prime; i <= end; i +=
prime)
                number[i] = 0;

                //ищем следующее простое число:
                prime++;
                while (prime <= Math.Sqrt(end) && num-
ber[prime] == 0)
                    prime++;
            } while (prime <= Math.Sqrt(end));

            //создаем новый файл для записи результатов писка:
            using (FileStream fs = new
FileStream("primes.txt", FileMode.Append))
            {
              using (StreamWriter w = new StreamWriter(fs, En-
coding.UTF8))
              {
                 w.WriteLine("Простые числа в заданном диапа-
зоне:");
                 //печатаем простые числа в консоли
                 //и записываем в файл:
                 for (int i = 2; i <= end; ++i)
                      if (number[i] != 0)
                      {
                          Console.Write(i + " ");
                          w.WriteLine(i);
                      }
                  w.WriteLine("");
              }
            }
```

```
            Console.WriteLine("");
            Console.WriteLine("");
        } // primes
    }
}
```

Здесь нас должна заинтересовать процедура записи найденных простых чисел в файл на диске, а всё остальное даётся нам без труда (Рис. 44.3).

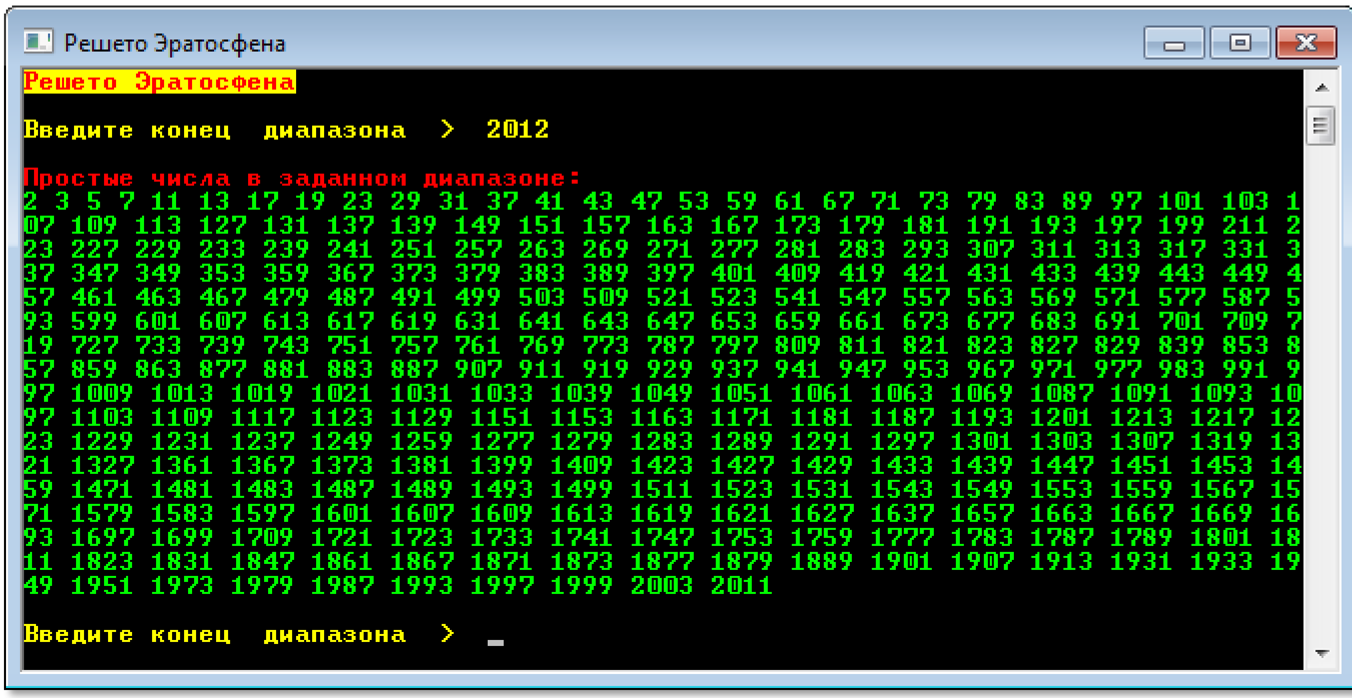


Рис. 44.3. Просеиваем числа на Си-шарпе

Исходный код программы находится в папке **Resheto**.

Изрешетив числа, мы переходим к просеиванию слов на предмет поиска *палиндромов*:

```
using System;
using System.IO;

namespace Palindrome
{
```

```
//ПРОГРАММА ДЛЯ ПОИСКА ПАЛИНДРОМОВ
class Palindrome
{
  static void Main(string[] args)
  {
     //заголовок окна:
     Console.Title = "Палиндромы";
     //имя словаря:
     string fileName="OSH-W97.txt";
     //ищем палиндромы:
     palindromes(fileName);
     Console.Read();
  }//Main

//ИЩЕМ ПАЛИНДРОМЫ В СЛОВАРЕ
static void palindromes(string fn)
{
  Console.WriteLine("");
  Console.ForegroundColor = ConsoleColor.Red;
  Console.WriteLine("Палиндромы:");
  Console.ForegroundColor = ConsoleColor.Green;
  Console.WriteLine("");
  //открываем файл словаря для чтения:
  using (StreamReader r = File.OpenText(fn))
  {
     string word = null;
     //считываем слова из файла:
     while (true)
     {
        //считываем очередное слово из файла:
        word= r.ReadLine();
        //словарь закончился!
        if (word == null) break;
        //печатаем очередное слово на экране:
        //Console.WriteLine(word);
        //длина слова:
        int len = word.Length;
        bool flg = true;
        //слово-палиндром?
        for (int i = 0; i < len / 2; ++i)
        {
           char ch1 = word[len - i - 1];
           char ch2 = word[i];
           if (ch1 != ch2)
```

```
                {
                    //не палиндром:
                    flg = false;
                    break;
                }
            }
            //печатаем найденный палиндром:
            if (flg) Console.WriteLine(word);
        } //while
    }
  }//palindromes
  }//class
}//namespace
```

Благодаря этой программе мы научились считывать словарный запас из файла, а заодно и отыскали все русские палиндромы (Рис. 44.4).

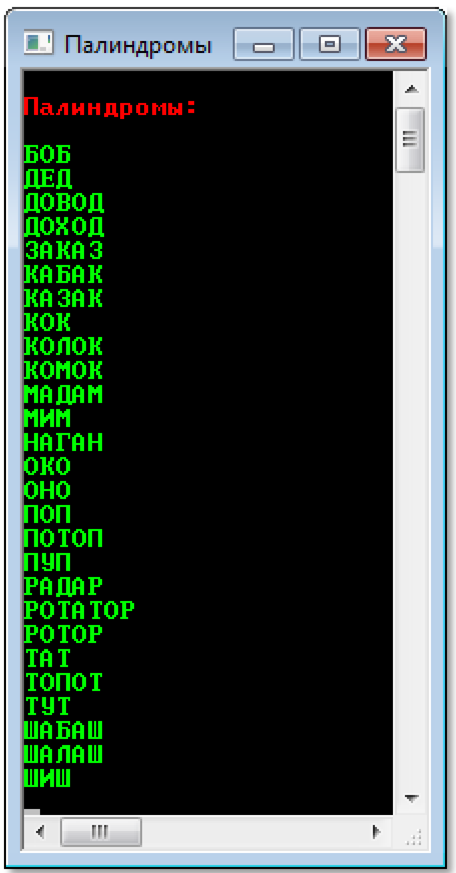


Рис. 44.4. От БОБа до ШИШа все на месте!

Исходный код программы находится в папке **Palindrome**.

Наше последнее приложение будет юмористическим. Если вы выполнили домашнее задание в конце урока Занимательная логопедия (в чём я не сомневаюсь), то вы легко перепишете *задание 6* на Си-шарп:

> **6**. Писатель-юморист Семён Альтов знаменит не только своими смешными рассказами, но и особенностями их чтения: он глухие звуки произносит как звонкие, что очень смешно. Например, *канарейка* у него слала *ганарейгой*, *парикмахер* – *баригмакер*. Заставьте программу читать по-альтовски.

Самое сложное здесь – это подобрать парные согласные. Я предлагаю поместить их в два константных строковых поля:

```
using System;

namespace Альтов
{
    //ПРОГРАММА ДЛЯ ПЕРЕВОДА ТЕКСТА
    //НА АЛЬТОВСКИЙ ЯЗЫК
    class Альтов
    {
        const string lit1 = "ПФКШСТХ";
        const string lit2 = "БВГЖЗДК";
```

В строку *lit1* мы собрали *глухие* согласные (вы можете подредактировать этот набор, если он противоречит вашим представлениям о глухоте), а в строку *lit2* – *звонкие.*

В методе *Main* пользователь вводит «нормальные» слова, а метод *Altov* переводит их на альтовский язык:

```
      static void Main(string[] args)
        {
            //заголовок окна:
            Console.Title = "Альтов";

            //бесконечный цикл ввода данных -
            //пока пользователь не закроет программу:
```

```
            do
            {
                Console.WriteLine("");
                Console.ForegroundColor = ConsoleColor.Red;
                Console.WriteLine("Альтов");
                Console.WriteLine("");

                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.Write("Ваш текст >  ");
                //исходный текст:
                string text = Console.ReadLine();
                //переводим буквы в верхний регистр:
                text = text.ToUpper();
                //литорейный текст:
                string altov = Altov(text);
                //печатаем перевод:
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine(altov);
                Console.WriteLine("");
            }
            while (true);
        }
```

Сделать это очень просто. Мы просматриваем строку *text* буква за буквой и, если найдём в ней глухую букву, то заменяем её звонкой. Остальные буквы мы оставляем без изменений:

```
        //ПЕРЕВОДИМ ТЕКСТ
        static string Altov(string text)
        {
            //альтовский текст:
            string res = "";
            //кодируем - заменяем буквы парными:
            foreach (char ch in text)
            {
                int n = lit1.IndexOf(ch);
                //глухая буква:
                if (n > -1)
                    //заменяем звонкой:
                    res += lit2[n];
                //остальные буквы не изменяем:
                else
                    res += ch;
            }
```

```
                return res;
            } //Altov
        }
    }
```

Новый цикл *foreach* теперь имеется и в *паскале*, он позволяет извлекать все элементы любой коллекции, например, символы из строки.

Поскольку строки *lit1* и *lit2* мы составили так, что парные согласные имеют *одинаковые* индексы, то перевод слов на альтовский язык получился очень простым.

Запускаем приложение и вводим слова, в которых *много* глухих согласных, а наш переводчик тут же выдаёт эти слова по-альтовски (Рис. 44.5).

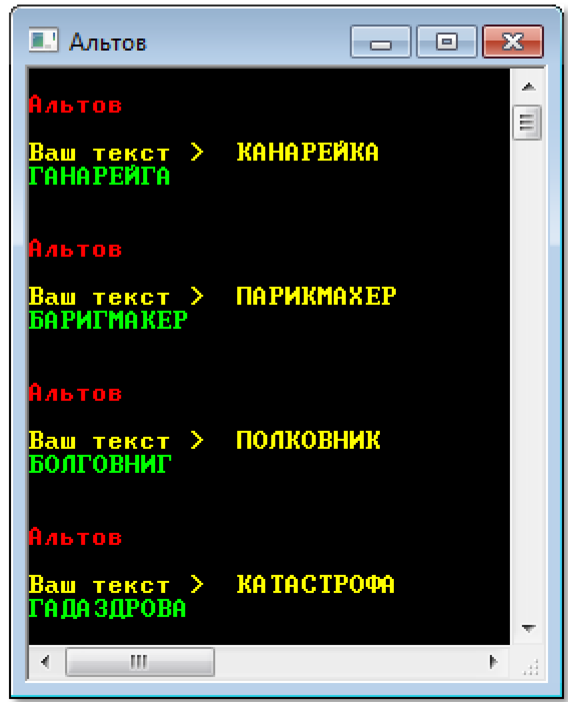
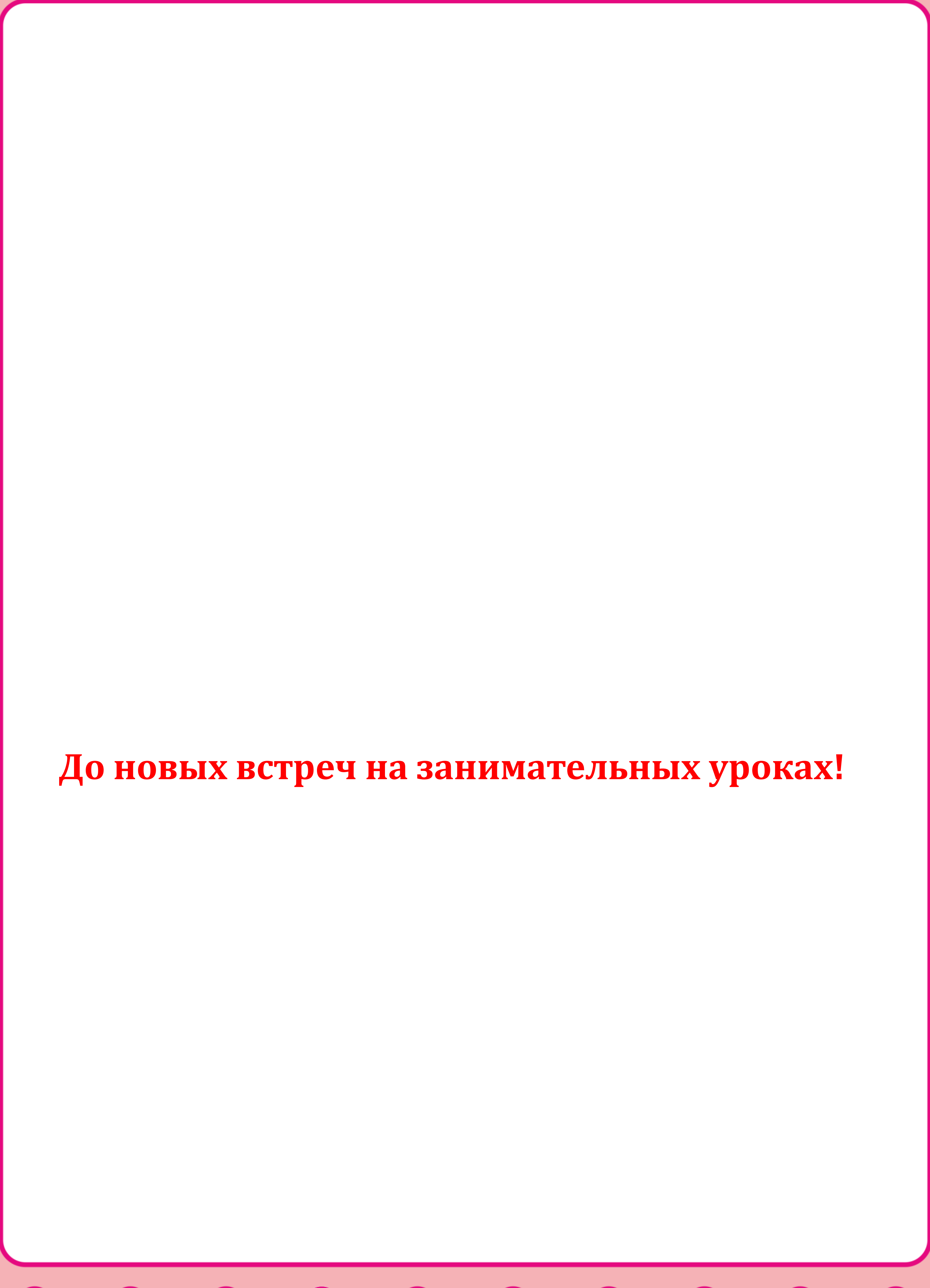


Рис. 44.5. Змежно болучилозь!

Исходный код программы находится в папке **Альтов**.

Подведём итоги

PascalABC.NET – это два языка в одном флаконе! Вы можете одновременно и параллельно писать программы и на паскале, и на Си-шарпе. А выучить два языка программирования, конечно, гораздо лучше (приятнее и полезнее), чем один. Но – для программ на Си-шарпе отсутствуют подсказки и встроенный отладчик. И если люди, хорошо знающие Си-шарп, обойдутся и без того, и без другого, то начинающим программистам это создаст дискомфорт и принесёт немало проблем. А выход такой: параллельно с изучением паскаля изучайте и Си-шарп, но используйте для этого средства разработки *Visual C# Express 2010* или *Visual Studio Express 2012*, которые можно бесплатно скачать с сайта фирмы *Майкрософт*.


До новых встреч на занимательных уроках!

Литература

Список литературы, использованной в книге

1. *Анатолий Маркуша*. **Вам - взлёт**! Издательство детской литературы МП РСФСР, 1982. – 238 с.

2. *Коснёвски Ч.* **Занимательная математика и персональный компьютер**. - Мир, 1987. – 192 с.

3. *Соболь И.М.* **Метод Монте-Карло**. - Наука, 1978. – 64 с.

4. *Мозговой М.В.* **Занимательное программирование: Самоучитель**. - Питер, 2005. – 208 с.

5. *Студенецкий Н.* **Мастерская головоломок**. – Детская литература, 1964. – 256 с.

6. *Кроновер Р.* **Фракталы и хаос в динамических системах**. – М.: Постмаркет, 2000. – 354 с.

7. *Гарднер Мартин.* **От мозаик Пенроуза к надёжным шифрам**. – М.: Мир, 1993. – 417 с.

8. *Эткинс П.* **Порядок и беспорядок в природе**. – М.: Мир, 1987. – 224 с.

9. *Гарднер Мартин*. **Математические досуги** - М.:Мир, 1972. – 495 с.