ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
Государственное образовательное учреждение высшего профессионального образования
**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

С.А. Рыбалка, Г.И. Шкатова

C++ BUILDER
ЗАДАЧИ И РЕШЕНИЯ

*Рекомендовано УМО по образованию в области телекоммуникаций
в качестве учебного пособия для студентов высших учебных заведений,
обучающихся по направлению подготовки дипломированных специалистов
210400 «Телекоммуникации»*

Издательство
Томского политехнического университета
2010

УДК 681.3.06:800.92(076)
ББК 32.973-018.1я73
Р93

Рыбалка С.А.

Р93 С++ Builder. Задачи и решения: учебное пособие / С.А. Рыбалка, Г.И. Шкатова; Национальный исследовательский Томский политехнический университет. – Томск: Изд-во Томского политехнического университета, 2010. – 479 с.

ISBN 978-5-98298-641-2

В учебном пособии изложены алгоритмы и примеры решения задач в среде Borland C++ Builder. Подача материала по нарастанию сложности позволяет овладеть секретами мастерства решения задач. Для работы с пособием необходимо иметь базовые знания языка С++.

Предназначено для студентов, обучающихся по направлению подготовки дипломированных специалистов 210400 «Телекоммуникации».

УДК 681.3.06:800.92(076)
ББК 32.973-018.1я73

Рецензенты

Доктор физико-математических наук, профессор ТГАСУ
Б.М. Шумилов

Кандидат технических наук
ведущий программист Института сильноточной электроники СО РАН
О.С. Колобов

ISBN 978-5-98298-641-2

© ГОУ ВПО «Национальный исследовательский Томский политехнический университет», 2010
© Рыбалка С.А., Шкатова Г.И., 2010
© Оформление. Издательство Томского политехнического университета, 2010

Введение

В настоящее время происходит стремительное развитие процесса информатизации общества. Новые автоматизированные информационные технологии проникают практически во все сферы социальной практики и становятся неотъемлемой частью жизни человечества. Появились новые понятия: «информационное общество» и «информационная культура». Информационное общество – это общество, в котором большинство работающих занято производством, хранением, переработкой и реализацией информации, в особенности высшей ее формы – знаний [16]. Очевидно, что такие страны, как США, Япония, Англия, Германия и некоторые другие, имеют наиболее развитую информационную индустрию. В этих государствах частью внутренней политики является поддержка инноваций и инвестирование в информационную индустрию, в развитие компьютерных технологий.

Для ориентации в потоке информации человек должен обладать информационной культурой как одной из составляющих общей культуры. Под информационной культурой понимается умение целенаправленно работать с информацией и использовать ее для получения знаний, и обрабатывать ее. Все это связано с компьютерными информационными технологиями, современными технические средствами и методами, которые изучаются в том или ином объеме в рамках дисциплины «Информатика».

Как и при использовании традиционных видов ресурсов и продуктов, человек должен знать: где находятся информационные ресурсы, сколько они стоят, как они создаются, кто ими владеет, кто в них нуждается, насколько они доступны. Представляется принципиально важным, чтобы перспективные информационные технологии, которые широко используются обществом уже в начале двадцать первого века, были бы изначально ориентированы на человека, учитывали бы его способности к восприятию информации и формированию на ее основе новых знаний.

Из всего сказанного можно сделать вывод, что дисциплина «Информатика» занимает весьма важное место при подготовке инженера независимо от того, какой деятельностью он предполагает заниматься.

В настоящем учебнике представлен один из самых сложных и самых значимых разделов из множества разделов дисциплины «Информатика» – это раздел «Программирование». Задача этого раздела – научить студентов создавать современные программные продукты, предназначенные для решения инженерных и математических задач.

Авторы убеждены в том, что в любом образовательном процессе очень важным элементом является самостоятельная работа. Только самостоятельная работа над проблемой позволяет получить глубокие знания. Одним из способов чему-нибудь «научиться» является ознакомление с уже готовыми решениями. Но если поставить себе цель: научиться решать задачи, то следует овладеть секретами мастерства решения задач. Именно мастерству создания программ с использованием современных образцов программирования учит данная книга.

Для работы с книгой необходима предварительная подготовка, а именно: базовые знания языка *C++*, общие сведения об объектно-ориентированном и модульном программировании, а также знания о визуальном проектировании программ в среде *C++ Builder*.

Книга представлена как задачник. Весь материал систематизирован по главам. Всего таких глав 18. Если Вы знаете алгоритмический язык *C++* и имеете практические навыки в визуальном программировании, то решение можно начинать с любой из задач по требованию. Если практических навыков нет, то рекомендуется начинать самообразование с решения тех задач, которые представлены в первой главе. Причем, выполнять их нужно последовательно друг за другом, так как по замыслу авторов именно первая глава позволяет на примитивных вычислительных алгоритмах погрузиться в *C++ Builder* и проникнуть в его идеологию.

В каждом разделе даны: информация о спектре применения поставленной задачи; программная постановка задачи; анализ задачи; рассуждения, связанные с особенностями решения представленного класса задач; план решения; решение задачи в соответствии с плановым. К простым алгоритмам даны подробные комментарии. Сложные алгоритмы представлены демонстрационными примерами, подробностями описания используемых вычислительных приемов, а также текстами программ, обеспечивающих вычисления.

Большинство задач выполнено в стиле объектно-ориентированного программирования. Это тесно связано с такими понятиями, как «класс» или «абстрактный тип данных». Хотя термины «тип данных» (или просто тип) и «абстрактный тип данных» звучат похоже, они имеют различный смысл. В языках программирования тип данных (переменной) определяет множество значений, которые может принимать эта переменная. Абстрактный тип данных (АТД) определяется как математическая модель с совокупностью операторов, определенных в рамках этой модели. Таким образом, классы могут служить простым примером АТД. Также примером АТД могут служить и структуры. В модели АТД операторы могут использовать в качестве операндов не только данные, определенные АТД, но и данные других типов: базовые типы языка программирования или определенные в других АТД. Результат действия оператора также может иметь тип, отличный от АТД, определенного в данной модели. Но предполагается, что, по крайней мере, один операнд или результат любого оператора имеет тип данных, определенный в рассматриваемой модели АТД [14].

В основу учебного пособия положен многолетний опыт чтения лекций и проведения лабораторных занятий по языкам программирования и современным информационным технологиям. Авторы надеются, что данный учебник послужит хорошим подспорьем студентам в приобретении навыков разработки проектов и реализации их в среде *C++ Builder*.

Авторы выражают благодарность коллегам и сотрудникам кафедры прикладной математики Национального исследовательского Томского политехнического университета за поддержку, а также родным и близким за проявленное долготерпение и понимание.

Общие рекомендации

Подготовительный этап

Практическую часть следует начинать с подготовительного[1] этапа.

1. Создать отдельную папку для проекта.
2. Запустить *C++ Builder*. Сохранить «пустой» проект в подготовленной папке.
3. Расположить на форме (или на формах) выбранные визуальные компоненты.
4. Задать значения свойств визуальных компонентов, влияющие на внешний вид и функционирование.

Разработка классов

Именно вопросам разработки классов или вопросам по созданию АТД требуется максимальное время при создании программы. Проекты, использующие принципы объектно-ориентированного программирования (ООП) и предназначенные для решения математических задач, рекомендуется разрабатывать в следующем порядке[2].

1. Анализ задачи.

На этом этапе выделяются математические объекты, с помощью которых можно описать решение задачи. Между ними устанавливаются отношения.

Создавая математическую модель для решения задачи, нужно

- сформулировать предположения, на которых будет основываться математическая модель;
- определить, что считать исходными данными и результатами;
- записать математические соотношения, связывающие результаты и исходные данные.

2. Программная постановка задачи.

Устанавливает соответствие между возможностями программы и требованиями пользователя.

3. Разработка интерфейсов классов.

При создании программы, как и при проектировании классов, разработчик должен изначально исходить из предположения, что предложенная задача со временем может претерпевать изменения, связанные с возрастанием и изменением запросов пользователя. Поэтому необходимо предусмотреть механизм, который бы позволял мобильно изменять интерфейсы классов, не затрагивая их реализацию. С этой целью имеет смысл сразу проектировать семейства классов.

В *интерфейс* вписываются поля и прототипы методов. Для них устанавливаются правила доступа – *public, private, protected.* Методы можно разделить на две группы:

- стандартные методы – конструкторы, в том числе конструктор копирования, деструкторы, свойства, унарные и бинарные операторы;
- специфичные методы, которые следуют из логики задачи.

[1] Использовать формулу: новая задача – новый проект.

[2] Этому порядку следуют авторы при решении задач, связанных с разработкой классов.

Особое внимание необходимо уделить конструкторам – методам конструирования объектов. Конструкторы класса требуют тщательной проработки. Их может быть много – «на все случаи жизни».

4. Формализация методов класса.

Формализация методов класса – это переход от спецификации метода, представляющего собой словесное описание назначения метода, к разработке алгоритма, обеспечивающего выполнение соответствующих действий.

Этап *«формализации методов класса»* включает в себя разработку алгоритмов и запись программной реализации этих алгоритмов. Если эти алгоритмы сложные, то их разработку следует производить в соответствии с технологией *нисходящего проектирования программ*: задача разбивается на несколько «крупных» подзадач, затем каждая из подзадач разбивается на новые подзадачи и так далее до тех пор, пока на очередном уровне решение подзадач не становится очевидным. В соответствии с такой технологией интерфейсы классов дополняются новыми методами со своими правилами доступа. Формализация методов классов – важнейший элемент программирования.

Разработка алгоритма начинается с его словесного описания. В непростых случаях это словесное описание в учебнике сопровождается демонстрационным примером. Возможным шагом может быть написание блок-схемы алгоритма. В простом случае можно ограничиться комментариями в описании кода метода. Иногда спецификация может быть выражена простой фразой в контексте интерфейса класса. В сложных случаях в учебнике спецификация выводится из контекста интерфейса и описывается более подробно.

Разработка интерфейса приложения и его дизайна

Подразумевается, что соответствие интерфейса задачам пользователя является неотъемлемым его свойством. Существует четыре основных критерия оценки качества любого интерфейса, а именно:

- скорость работы пользователей;
- количество человеческих ошибок;
- скорость обучения;
- субъективное удовлетворение [8].

Скорость выполнения работы является важным критерием эффективности интерфейса. Длительность выполнения работы пользователем состоит из длительности восприятия исходной информации, длительности интеллектуальной работы (пользователь думает, что он должен сделать), длительности физических действий пользователя и длительности реакции системы. Как правило, длительность реакции системы является наименее значимым фактором.

Согласно Дональду Норману, взаимодействие пользователя с системой (не только компьютерной) состоит из семи шагов: формирование цели действий; определение общей направленности действий; определение конкретных действий; выполнение действий; восприятие нового состояния системы; интерпретация состояния системы; оценка результата [8]. Согласно этому списку процесс размышления занимает почти все время, в течение которого пользователь работает с компьютером. Во всяком случае, шесть из семи этапов полностью заняты умственной дея-

тельностью. Соответственно, повышение скорости этих размышлений приводит к существенному улучшению скорости работы.

К сожалению, существенно повысить скорость собственно мышления пользователей невозможно. Тем не менее, уменьшить влияние факторов, усложняющих процесс мышления, вполне возможно. Пользователь должен знать:

- что он хочет получить на выходе (решение задачи);
- как минимум одну последовательность действий, приводящую к успешному результату;
- где ему найти все объекты, участвующие в процедуре решения;
- как определять пригодность объектов для их использования;
- как управляться с объектами.

Список, как видим, довольно внушительный. И если с первым пунктом проблем обычно не возникает, то остальные требуют определенных усилий. А помочь разобраться в том должен интерфейс со встроенной системой подсказок действий. Следовательно, должен быть продуман механизм *управления программой через элементы интерфейса.*

Обеспечение функциональности приложения

Обеспечение функциональности приложения связано с разработкой кода программы, предназначенного для

- операций с объектами класса, обеспечивающих их взаимодействие и согласованную работу;
- управления программой через элементы интерфейса.

Тестирование приложения

Тестирование созданного программного продукта следует начинать с разработки плана тестирования. При этом следует помнить, что весь процесс тестирования можно разделить на три этапа

- Проверка в нормальных условиях. Предполагает тестирование на основе данных, которые характерны для реальных условий функционирования программы.
- Проверка в экстремальных условиях. Тестовые данные включают граничные значения области изменения входных переменных, которые должны восприниматься программой как правильные данные. Типичными примерами таких значений являются очень малые или очень большие числа и отсутствие данных. Еще один тип экстремальных условий — это граничные объемы данных, когда массивы состоят из слишком малого или слишком большого числа элементов.
- Проверка в исключительных ситуациях. Проводится с использованием данных, значения которых лежат за пределами допустимой области изменений.

Известно, что все программы разрабатываются в расчете на обработку ограниченного набора данных. Поэтому важно получить ответ на следующие вопросы:

- что произойдет, если в программе, не рассчитанной на обработку отрицательных и нулевых значений переменных, в результате какой-либо ошибки придется иметь дело как раз с такими данными?

- как будет вести себя программа, работающая с массивами, если количество их элементов превысит величину, указанную в объявлении массива?
- что произойдет, если числа будут слишком малыми или слишком большими?

Наихудшая ситуация складывается тогда, когда *программа воспринимает неверные данные как правильные и выдает неверный, но правдоподобный результат*. **Хорошая программа должна сама отвергать любые данные, которые она не в состоянии обрабатывать правильно.**

Следует попытаться представить возможные типы ошибок, которые пользователь способен допустить при работе с Вашей программой и которые могут иметь неприятные для нее последствия. Нужно не забывать, что способ мышления пользователя отличается от способа мышления программиста. Нужно предусмотреть обработку ошибок типа: отсутствие нужного файла, неправильные форматы данных и т.д. Список действий, которые могут привести к неправильному функционированию программы, довольно длинный и зависит от того, что делает приложение в данный момент времени.

Для тестирования можно изготовить подпрограмму, которая будет вызывать выполняемую процедуру и следить за ходом ее выполнения. Эта подпрограмма должна содержать: подготовку входных данных, которые должны быть прочитаны из файла для запуска тестируемой программы; сравнение выходных данных отдельных подпрограмм с контрольными значениями, которые также извлекаются из файла.

Вычисление значений функции по формулам 1

Эта глава позволяет получить начальные практические навыки визуального программирования в *C++ Builder*. Предложенные в данной главе задачи – практически однострочные вычисления, не представляющие собой алгоритмической сложности. Различие задач основано только на особенностях построения интерфейса и функциональных возможностях программы.

Общая постановка задачи

Постановка задачи вычисления значений функции по формулам содержит функцию $Y = F(\bar{x}) = F(x_1, x_2 \ldots x_n)$, определенную на *n*-мерном евклидовом пространстве R^n.

Требуется разработать функциональное приложение, в котором задаются значения $x_1, x_2 ... x_n$ и на основании заданных значений производятся вычисления функции $F(\bar{x})$. Результаты решения должны быть отражены на экране.

Стратегия решения задачи

Производится *анализ* задачи. На этом этапе анализируется вычислительная сторона задачи, определяются входные и выходные данные. В соответствии с этим формируется набор компонент и внешний вид или *интерфейс* программы. Затем на форму[1] выставляются визуальные компоненты. В перечне компонент обязательно должны быть такие, которые предоставляют пользователю возможность задавать вручную исходные данные и просматривать результаты вычислений. На этапе обучения эти компоненты рекомендуется располагать в том виде, как это показано на рисунках, сопровождающих решение задачи. С помощью инспектора объектов устанавливаются *свойства* этих компонентов. После чего *формируются тексты обработчиков* событий с целью обеспечения функциональности приложения. На завершающем этапе производится тестирование работоспособности приложения и исправляются обнаруженные ошибки.

Задача 1.1. Требуется вычислить значения функции $F(k, x, n) = k + x + n$ для заданных вещественных значений *k*, *x*, *n*.

Решение

1. Анализ задачи.

Вычислительная сторона получения значений функции проста и не вызывает вопросов. Следует только определиться с типом переменных. Так как по условию задачи *k*, *x*, *n* – вещественные, то, соответственно, и результат – вещественный.

[1] Предварительно выполняется подготовительный этап, как это изложено во введении.

В соответствии с условием задачи пользователю нужно предоставить возможность вводить вручную значения исходных данных *k*, *x*, *n* и отражать результат вычислений – значение $y = F(k, x, n)$.

2. Разработка интерфейса и задание свойств компонентов.

Для ввода/вывода данных можно использовать визуальный компонент – однострочный редактор `Edit`. Очевидно, что пользователь должен иметь подсказку, какое из окон однострочных редакторов соответствует конкретному входному и выходному значению. Для отображения поясняющего статического текста будем использовать визуальный компонент `Label`. Программа должна быть функциональной. Она должна принять от пользователя, по крайней мере, две команды:

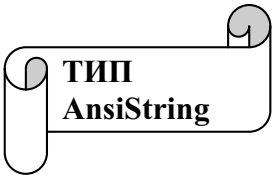
- произвести вычисления;
- прекратить выполнение программы.

Для того чтобы пользователь мог задать эти команды можно воспользоваться визуальными компонентами-кнопками `Button` (событие `OnClick`).



Появляется всегда при нажатии левой или правой кнопки манипулятора типа «мышь».

Все перечисленные компоненты можно расположить на одной форме. Расположение кнопок может быть таким, как это показано на рис. 1.1, *а*. Исходные данные и результат вычислений будут отображаться в области редактирования компонента `Edit`, доступ к которой обеспечивается свойством `Text` редактора. Это поле представляет собой стандартную строку на языке *C++* с классовым типом `AnsiString`.



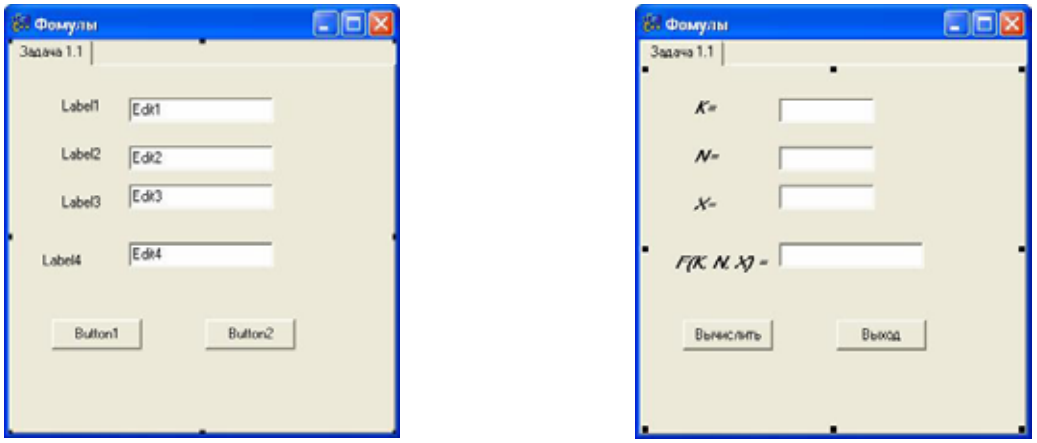
В *C++ Builder* тип строк `AnsiString` реализован как класс, представляющий строки с нулевым символом в конце. При объявлении переменные типа `AnsiString` инициализируются пустыми строками. Для `AnsiString` определены операции отношения ==, !=, >, <, >=, <= а также операции присваивания =, += и операция склеивания строк (конкатенации) +. Определена также операция индексации []. Индексы начинаются с 1. Например, если строка `S1 = "Привет"`, то `Sl[l]` вернет `'П'`, `Sl[2]` вернет `'р'` и так далее. Также класс `AnsiString` имеет множество других методов и свойств.

Все переменные программы хранятся во внутреннем формате, поэтому использование компонента `Edit` при вводе данных потребует преобразования строки `AnsiString` в вещественное число, а при выводе – преобразования числа в строку `AnsiString`. Для этого можно использовать следующие функции преобразования:

- `extended __fastcall StrToFloat(const AnsiString S);` – преобразование строки `S` в вещественное число;
- `AnsiString __fastcall FloatToStrF(Extended Value, TFloatFormat Format, int Precision, int Digits);` – преобразование вещественного числа

`Value` в строку типа `AnsiString`, используя заданный формат `Format`, точность `Precision` и число цифр `Digits`. Здесь возможны следующие типы формата `Format`:

- `ffExponent` – научный формат; используется для отображения очень больших или очень малых чисел, например число $1.3 \cdot 10^5$ в виде 1.3e5;
- `ffFixed` – формат с фиксированной запятой – для отображения небольших чисел в виде 12.718;
- `ffGeneral` – обобщенный формат – для отображения очень больших или очень малых величин в научном формате, а небольших – в формате с фиксированной запятой;
- `ffNumber` – в отличие от `ffFixed`, имеет символ-разделитель тысяч;
- `ffCurrency` – денежный формат.



а б

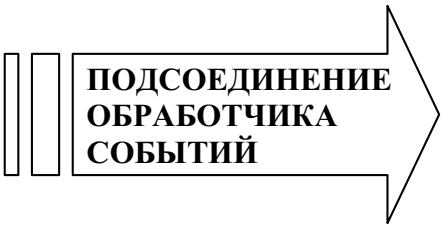
Рис. 1.1. Интерфейс программы для вычисления по формулам: а) до задания свойств компонентов; б) после задания свойств компонентов

СВОЙСТВА КОМПОНЕНТОВ

- Компонент `Label1` (статический текст): `Caption` (заголовок метки) – «X= »; `Name` (наименование) – «XLabel».
- Компонент `Label2`: `Caption` – «K= »; `Name` – «KLabel».
- Компонент `Label3`: `Caption` – «N= »; `Name` – «NLabel».
- Компонент `Label4`: `Caption` – «F(K,N,X)= »; `Name` – «YLabel».
- Компонент `Edit1` (однострочный редактор): `Text` (поле ввода, значение по умолчанию) – « »; `Name` (наименование) – «XEdit».
- Компонент `Edit2`: `Text` – « »; `Name` – «KEdit».
- Компонент `Edit3`: `Text` – « »; `Name` – «NEdit».
- Компонент `Edit4`: `Text` – « »; `Name` – «YEdit».
- Компонент `Button1` (кнопка): `Caption` (текст на кнопке) – «&Вычислить»; `Name` (наименование) – «CalcButton».
- Компонент `Button2`: `Caption` – «В&ыход»; `Name` – «CloseButton».

3. Подготовка кодов программ.

После установки свойств следующим шагом будет подготовка текстов для обработчиков событий `OnClick` под кнопки, предназначенные обеспечить функциональность приложения.



Выделяется соответствующий компонент, в окне *Object Inspector* открывается страница `Events`. В свободном поле строки `onClick` производится двойной щелчок левой кнопкой мыши. После этого активизируется окно *Code Editor*. В него будет вставлена заготовка для процедуры, которую следует заполнить соответствующим кодом.

Процедура-обработчик события `OnClick` имеет один параметр `Sender` объектового типа `TObject`.



является стандартным входным параметром процедуры-обработчика событий. Он указывает на компонент, который является инициатором события, подлежащего обработке. Если процедура-обработчик связана с несколькими компонентами, то перед описанием действий следует проанализировать, какой из компонентов находится в фокусе.

Алгоритм решения задачи (событие «Вычислить») сводится к следующим шагам.

1. Из окон однострочных редакторов (свойство `Text`) читаются исходные данные. При чтении необходимо преобразование информации из типа `AnsiString` в тип `float`.
2. Производится вычисление по формуле.
3. После оператора вычисления производится запись результата в поле `Text` соответствующего редактора с предварительным преобразованием в тип `AnsiString` из типа `float`.

Тексты обработчиков событий представлены в листингах 1.1 и 1.2.

Листинг 1.1. Описание действий команды «Вычислить»

```
void __fastcall TForm2::CalcButtonClick(TObject *Sender)
{
  float k, n, x;  // описание входных переменных
  float y;  // описание переменной - результата
  k = StrToFloat (Edit1->Text); // из строки редактора Edit1
      // взять текст, преобразовать его  в число
      // и передать в k
  n = StrToFloat (Edit2->Text);
  x = StrToFloat (Edit3->Text);
  y = k+n+x; // произвести вычисления
  Edit4->Text = FloatToStrF (y, ffFixed, 6, 3);
            // преобразовать число y в строку и
            // записать результат в Edit4
}
```

Листинг 1.2. Описание действий команды «Выход»

```
void __fastcall TForm2::CloseButtonClick(TObject *Sender)
{ Close(); // закрыть форму и приложение
}
```

Примечание

При вводе числовых вещественных данных в строку `Edit` для последующего преобразования в число, *дробную часть от целой следует* отделять не точкой, а запятой. В общем случае функции преобразования строки текста в число используют знак отделения «,» или «.» в зависимости от настроек *Windows*. Чтобы эти функции использовали единый символ при работе на разных компьютерах, следует в начале работы программы, например в обработчике события `OnCreate` формы, задать глобальную переменную `DecimalSeparator`. Чтобы символом разделения была «точка», достаточно выполнить присваивание: `DecimalSeparator = '.'`.

Задача 1.2. Вычислить значения функции $F(k,x,n) = k \cdot \sqrt{|x|} + x \cdot n$. Проект реализовать в соответствии с интерфейсом, представленным на рис. 1.2.

Решение

В отличие от предыдущей задачи все исходные значения для ввода данных располагаются в поле одного редактора `Edit`. Поэтому решение будет отличаться только способом ввода данных и, конечно, вычислительной формулой.



Рис. 1.2. Интерфейс приложения для задачи 1.2

Для чтения данных из строки можно воспользоваться функцией `sscanf`, которая расположена в стандартной библиотеке ввода/вывода (необходимо подсоединить модуль `<stdio.h>`).

ФУНКЦИЯ SCANF

Функция `scanf` выполняет форматированный ввод из потока `stdin`. Синтаксис прототипа этого оператора следующий:

```
int scanf (const char *format [,adress, ...]);
```

Посредством функции `scanf` просматривают входные поля, символ за символом, считывая их из потока `stdin`. Затем каждое поле формати-

руется в соответствии со спецификацией формата, передаваемой `scanf` аргументом `format`. После этого она записывает отформатированный ввод по адресам, задаваемым аргументами, следующими за форматной строкой. Число спецификаций формата должно соответствовать числу адресов.

Как это следует из синтаксиса, функция `scanf` предназначена для ввода данных из заранее сформированной строки с типом `char*`. Так как строка редактора `Edit` имеет тип `AnsiString`, а функция `sscanf` требует строку с типом `char*`, необходимо выполнить преобразование типа `AnsiString` в `char*`, Для этого следует воспользоваться функцией класса `AnsiString – c_str()`.

И, поскольку в формуле требуется вызов функций из библиотеки `<math.h>`, следует подсоединить и эту библиотеку. Тогда действия, которые необходимо выполнить в соответствии с постановкой задачи, можно описать следующими шагами.

1. Читаются исходные данные из одной строки `Edit`. В этой строке числа располагаются друг за другом и разделены пробелами.
2. Производится вычисление по формуле.
3. После оператора вычисления производится запись результата в поле `Text` соответствующего редактора с предварительным преобразованием.

Код процедуры представлен в листинге 1.3.

Листинг 1.3. Описание действий команды «Вычислить»

```
void __fastcall TForm2::Button3Click(TObject *Sender)
{
  float k, n, x, y;
  char* Str; // вспомогательная переменная для строки чисел
  Str = Edit6->Text.c_str(); // переписать текст из строки
   // редактора в строку Str с преобразованием типов
   // AnsiString в char*
  sscanf (Str, "%f %f %f ", &k, &n, &x);
  y = k*sqrt(fabs(x))+x*n;
  Edit6->Text = FloatToStrF (y, ffFixed, 6, 3);
 }
```

Примечание

Следует отметить, что в строке ввода `Edit` *дробная и целая части числовых* значений должны отделяться точкой, так как эта строка предназначена для использования функцией `sscanf`.

Задача 1.3. Вычислить значения функции $F(k,x,n) = k \cdot \sqrt{|x|} + \cos(n)$. Проект реализовать в соответствии с интерфейсом, представленном на рис. 1.3.

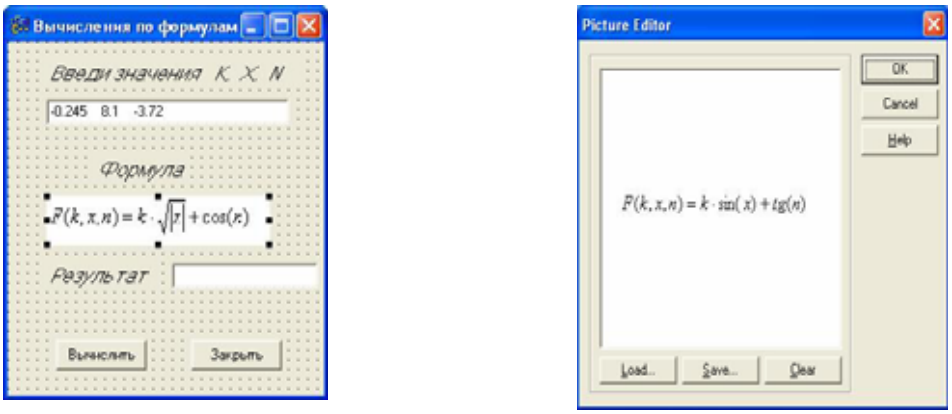
Решение

Данная задача 1.3 отличается от задачи 1.2 тем, что направлена на вычисление другой функции. Более того, в интерфейсе данного приложения предусмотрено поле, в котором отображается вычисляемая формула.

Один из способов решения задачи отображения формулы – воспользоваться компонентом `Image`. Этот компонент имеет свойство `Picture`, которое задает изображение в графическом формате, размещенное внутри этого компонента.

ДЛЯ ОТОБРАЖЕНИЯ ФОРМУЛЫ

требуется предварительно получить ее графическое изображение. Сделать это можно, например, так: с помощью редактора *MathType* набрать формулу в естественном математическом виде и сохранить в формате *.wmf*.



а б

Рис. 1.3. Элементы проекта для задачи 1.3: а) интерфейс приложения с выделенным компонентом Image; б) подключение картинки к полю Picture компонента Image посредством специализированного редактора

Задача 1.4. Вычислить значения функции $F(k,x,n)=k\cdot\sin(x)+\mathrm{tg}(n)$. Проект реализовать в соответствии с интерфейсом, представленном на рис. 1.4.

Решение

Решение данной задачи отличается только оформлением результата: *в окно вывода помимо числового значения* помещается формула, в которую на место параметров помещаются введенные числовые значения. Поэтому в обработчике события `OnClick` для кнопки «Выполнить» после вычисления функции необходимо внести действия, связанные с построением строки вывода. Эта строка формируется сборкой фрагментов, то есть состоит из «кусочков» вычислительной формулы и строк-чисел со значениями исходных данных и результата вычислений.

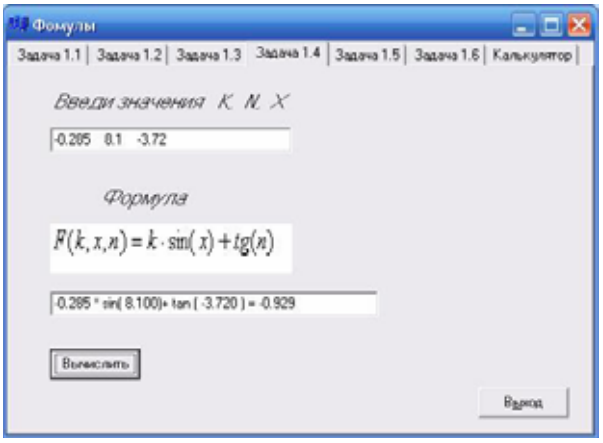


Рис. 1.4. Интерфейс приложения для задачи 1.4

В алгоритме используются две вспомогательные переменные `Str` и `Ptr`. Переменная `Str` используется в двух смыслах: для чтения данных; для формирования строки вывода. Сначала в `Str` копируется строка текста из редактора `Edit` с преобразованием типа `AnsiString` в тип `char*`, из которой читаются данные с помощью функции `sscanf`. А затем в нее заносятся фрагменты с числами – исходными данными. Переменная `Ptr` используется для сборки строки, содержащей результат вычислений. Текст обработчика события вычисления представлен в листинге 1.4.

Листинг 1.4. Описание действий команды «Вычислить»

```
void __fastcall TForm1::Button5Click(TObject *Sender)
{ char Str [60]; // вспомогательная переменная
  float k, x, n, y;
  // копирование строки из Edit с преобразованием в char*
  strcpy (Str,Edit1->Text.c_str());
  // чтение значений k, x, n из строки Str
  sscanf (Str,"%f %f %f",&k, &x, &n);
  y = k*sin(x)+tan(n);  // вычисление функции
            // сборка результата
  AnsiString Ptr; // объявление строки-результата
  sprintf (Str,"%3.3f",k); // запись в Str числа k
  Ptr = (AnsiString) Str; // занести в Ptr
  Ptr = Ptr + " * sin( "; // добавить к Ptr фрагмент
  sprintf(Str,"%3.3f",x); // запись в Str числа x
  Ptr = Ptr + (AnsiString)Str; // добавить к Ptr значение x
  Ptr = Ptr + ")+ tan ( "; // добавить к Ptr фрагмент
  sprintf(Str,"%3.3f",n); // запись в Str числа n
  Ptr = Ptr + (AnsiString)Str; // добавить к Ptr значение n
  Ptr = Ptr + " ) = "; // добавить к Ptr фрагмент
  sprintf(Str,"%3.3f",y); // запись в Str числа y
  Ptr = Ptr + (AnsiString)Str; // добавить к Ptr значение y
  Edit2->Text = Ptr; // занести результат в Edit2
}
```

Задача 1.5. Вычислить значения функции

$$y = \begin{cases} 2 + \operatorname{tg} k + x^3 e^{-x^2}, & x < 2 \\ \dfrac{1}{7}\sin(2x) + \sqrt{\left|e^k\right|}, & 2 \le x \le 7 \\ \sqrt[7]{\operatorname{tg}(2k)} + x + n, & x > 7 \end{cases}.$$

Приложение реализовать в соответствии с интерфейсом, представленным на рис. 1.5.

Решение

Согласно рис. 1.5 в центре формы размещается графическое изображение формулы. С технологией подсоединения изображения аналитического вида формулы мы уже имели дело в предыдущих задачах. В задаче пользователю предлагается ввести значения исходных данных. И в зависимости от местоположения аргумента `x` на числовой оси, производится выбор формулы для вычисления значения `y` посредством условного оператора `if`.

Для того чтобы показать, какая из формул выбрана, слева от изображения аналитического вида функции расположены 3 кнопки компонента `RadioButton` (с име-

нем `RadioButton1`). Соответственно, в тексте обработчика события `OnClick` кнопки «Выполнить» после команд выбора формулы и вычисления функции следует поставить команду, которая будет активизировать нужную кнопку `RadioButton`. И для того, чтобы эффект демонстрации выбора программой нужной формулы был более ярким, *создадим отдельный компонент* `Image` (с именем `MyImage`) *во время работы приложения*. В этот компонент будем на этапе выполнения загружать соответствующую формулу и накладывать это новое изображение на изображение общее. Это создаст дополнительно к выделению кнопки `RadioButton` эффект выделения конкретной формулы в общей формуле.

ДЛЯ СОЗДАНИЯ КОМПОНЕНТА ВО ВРЕМЯ РАБОТЫ ПРИЛОЖЕНИЯ

Для создания компонента во время работы приложения требуется:

- объявить указатель на переменную с типом соответствующего компонента;
- создать экземпляр компонента с помощью операции *new*;
- присвоить его свойству `Parent` адрес того элемента, на фоне которого этот компонент будет отображаться;
- установить требуемые по условию задачи значения его свойств.

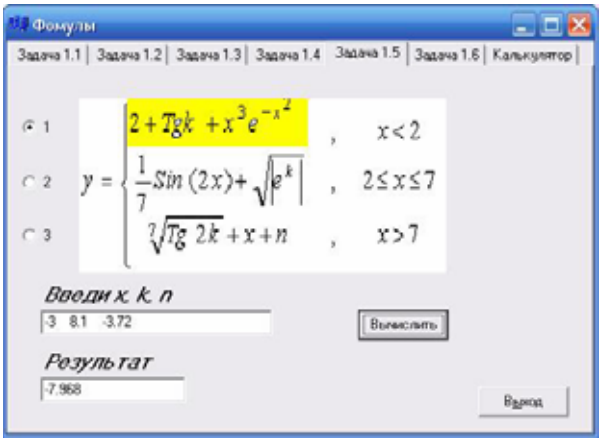


Рис. 1.5. Интерфейс приложения для задачи 1.5

Алгоритм решения сводится к следующим шагам:

1) создается компонент `Image`, если он не был создан ранее;
2) читаются из строки редактора значения переменных;
3) проверяется с помощью условного оператора принадлежность значения переменной `x` одному из интервалов и в зависимости от этого:
   а) устанавливается отметка на соответствующем компоненте `RadioButton`;
   б) устанавливаются размеры и местоположения `Image` для отдельной формулы;
   в) загружается графическая формула;
   г) вычисляется значение функции в соответствии с формулой;
4) отображается полученный результат в соответствующее окно редактора.

Код процедуры представлен в листинге 1.5.

Листинг 1.5. Описание действий команды «Вычислить» для задачи 1.5

```
  // MyImage – динамическая переменная для хранения формулы
TImage* MyImage = NULL; // объявляется переменная
void __fastcall TForm2::Button6Click(TObject *Sender)
{ T1 = Time();
  if (MyImage == NULL)
   {MyImage = new TImage(Form2); // создается экземпляр
    MyImage->Parent = tsTask1_5; // задается родитель
   }
  float k, n, x, y;
    // ввод исходных данных
  char* Str =  EnterEdit->Text.c_str( );
  sscanf(Str,"%f %f %f", &x, &k, &n);
  if (x<2) // если меньше 2
   {RadioButton1->Checked = true; //отметку на первую кнопку
      // установка размеров и местоположения изображения
    MyImage->Height = 40;
    MyImage->Width = 150;
    MyImage->Left = Image1->Left+40;
    MyImage->Top = Image1->Top;
    MyImage->Picture->LoadFromFile("F5_1.bmp");
                                    // загружается формула
    MyImage->Stretch = true;
    MyImage->Visible = true;
    y = 2*tan(k)+x*x*x*exp(-x*x);
   }
  else
    if (x>7)
     {RadioButton3->Checked = true;
      MyImage->Height = 40;
      MyImage->Width = 150;
      MyImage->Left = Image1->Left+40;
      MyImage->Top = Image1->Top+100;
      MyImage->Picture->LoadFromFile("F5_3.bmp");
      MyImage->Stretch = true;
      MyImage->Visible = true;
      y = pow(tan(2*k),(float)1/7)+x+n;
     }
    else
     {RadioButton2->Checked = true;
      MyImage->Height = 60;
      MyImage->Width = 150;
      MyImage->Left = Image1->Left+40;
      MyImage->Top = Image1->Top+40;
      MyImage->Picture->LoadFromFile("F5_2.bmp");
      MyImage->Stretch = true;
      MyImage->Visible = true;
      y = (float)1/7*sin(2*x)+sqrt(fabs(exp(k)));
     }
  ResultEdit->Text = FloatToStrF (y, ffFixed, 6, 3);
}
```

Задача 1.6. Вычислить значения функции $F(k,x,n)=\sqrt[7]{\mathrm{tg}(2k)}+x+n$. Проект разработать в соответствии с интерфейсом, представленном на рис. 1.6.

Решение

Особенность данной задачи в том, что в строке с исходными данными, помимо числовых значений, *содержатся поясняющие символы*. Это не позволяет воспользоваться функцией `sscanf`, как в предыдущих задачах. Здесь требуется решить дополнительную задачу: отделить символы-цифры от обычных символов. Кроме этого, на форме не предусмотрено поле для отображения результатов вычисления. Следовательно, необходимо создать такое поле (компонент *Edit*) *динамически на этапе выполнения* программы.

Рассмотрим следующий подход к выделению чисел из строки символов. Очевидно, что каждое число представляет собой группу из подряд идущих друг за другом цифр, среди которых может быть/не быть символ «.», который отделяет дробную часть числа от его целой части. Если предположить, что цифры поступают последовательно слева направо, то формирование целой части числа может производиться с помощью формулы:

$$S = S \cdot 10 + ц, \qquad (1)$$

где S – начальное значение равно 0, а $ц$ – очередная цифра. Например, число 245 получается за три итерации: $S = 0 \cdot 10 + 2$; $S = 2 \cdot 10 + 4 = 24$; $S = 24 \cdot 10 + 5 = 245$. Формирование дробной части следует производить по другой формуле:

$$S = S + ц \cdot (0.1)^h, \quad \text{где } h = 1, 2, 3... \qquad (2)$$

Например, число 0.6391 получается за 4 итерации: $S = 0 + 6 \cdot (0.1)^1 = 0.6$; $S = 0.6 + 3 \cdot (0.1)^2 = 0.63$; $S = 0.63 + 9 \cdot (0.1)^3 = 0.639$; $S = 0.639 + (0.1)^4 = 0.6391$. Окончательное числовое значение получается в результате сложения целой и дробной частей.

Вопрос с определением знака числа можно решить с помощью переменной-признака, которая будет принимать значение ноль/один в зависимости от того, является ли число положительным. Знак можно «снять» в процессе просмотра символов строки, проанализировав символ, расположенный слева от цифровой группы.

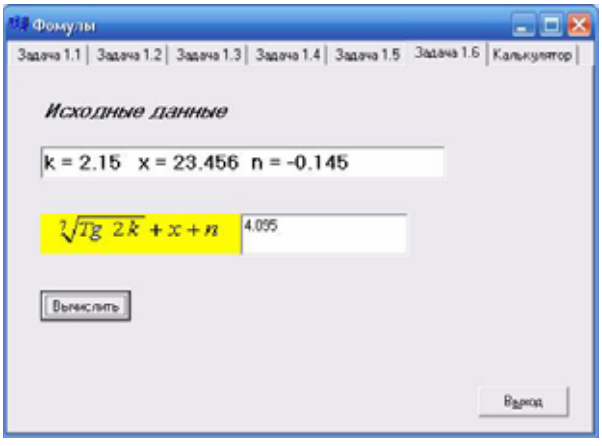


Рис. 1.6. Главное окно для решения задачи 1.6

Алгоритм выделения чисел из строки символов *основан на повторении двух циклов*: один цикл находит начало очередного числа; второй – конец числа. В первом цикле перебираются символы до тех пор, пока не встретится символ-цифра.

Этот факт будет говорить о том, что найдено начало группы цифр, то есть начало числа. Второй цикл будет перебирать символы до тех пор, пока не закончится цифровая группа (число). Внутри этого второго цикла следует производить «сборку» цифр в число в соответствии с описанными выше правилами для сборки целых и дробных частей. После сборки сформированное число записывается в подготовленный массив для этих целей.

Для простоты алгоритма является целесообразным оформить в виде отдельной процедуры проверку символа на то, является ли этот символ цифрой или нет. Назовем эту процедуру именем `IsDigit`.

Проверка символа на «цифру»

Работа алгоритма функции `IsDigit` – проверки символа на «цифру» основана на известном факте, что *символы-цифры в кодовом представлении, располагаются непосредственно друг за другом*: «1» после «0», «2» после «1» и так далее. Этим символам соответствуют порядковые значения в системе кодирования: 48(0), 49(1) …

Функция `IsDigit` имеет единственный входной параметр `T` – символ, который нужно проверить. В качестве результата функция возвращает одно из двух значений: `true` (истина, если `T` – цифра) или `false` (ложь, если `T` – не цифра). Описание этой процедуры приведено в листинге 1.6.

Листинг 1.6. Описание функции, которая определяет, является ли заданный символ цифрой

```
Boolean IsDigit (char T)
// если T – цифра, то функция возвращает true иначе false
{ if (T >= '0' && T <= '9') return true;
  else return false;
}
```

В алгоритме «вычисления по формуле», в соответствии с условием задачи, должны быть предусмотрены следующие действия:

- извлечение числа из строки символов;
- передача полученных значений в области памяти `k`, `x`, `n`;
- вычисление значения `y` по формуле;
- создание поля редактора динамически;
- помещение в него результата вычислений.

Для реализации алгоритма используются следующие вспомогательные переменные:

- `Str` – строка читаемых символов;
- `Value` – вспомогательный массив для хранения трех чисел;
- `Len` – длина строки `Str`;
- `S` – вспомогательная переменная для хранения очередного числа;
- `Flag` – признак перехода от целой части к дробной;
- `Pr` – признак наличия знака минус у числа;
- `h` – значение степени дробной части;
- `MyEdit` – динамическая переменная для создания экземпляра редактора.

И тогда алгоритм подробно можно описать следующими шагами.

1) Копируется входной текст из строки редактора во вспомогательную переменную `Str`.
2) Определяется `Len` – длина строки.
3) Создается динамическая переменная `MyEdit` типа «Однострочный редактор» для ввода исходных данных.
4) Устанавливается родитель для создаваемого однострочного редактора (страница `TabSheet5`).
5) Инициализируется индекс для массива `Value`.
6) Выполняется цикл по индексу символа в строке `Str`:
   а) выполняется цикл для пропуска символов, не относящихся к цифрам;
   б) устанавливается значение признака `Pr`, указывающего на знак числа;
   в) устанавливается начальное значение степени `h` дробной части;
   г) инициализируется начальное значение переменной `S` в которую собирается число;
   д) выполняется цикл по перебору символов, относящихся к цифрам:
      - если символ – «точка», то устанавливается значение переменной `Flag`, определяющий факт перехода через «точку»;
      - в зависимости от значения `Flag` производится «сборка» отдельно целой части числа по формуле (1) или дробной части по формуле (2);
   е) производится «сборка» всего числа с учетом знака с записью значения в элемент массива `Value`.
7) Переписываются значения из массива `Value` в переменные `x`, `k`, `n`.
8) Устанавливаются свойства однострочного редактора для помещения результата.
9) Вычисляется значение функции.
10) Помещается результат вычислений в однострочный редактор.

Текст обработчика события `OnClick` для кнопки «Вычислить» представлен в листинге 1.7.

Листинг 1.7. Описание действий команды «Вычислить» задачи 1.6

```
TEdit* MyEdit = NULL; // объявление динамической переменной
void __fastcall TForm2::BitBtn1Click(TObject *Sender)
{ char Str[40];
      // скопировать из редактора текст в строку Str
  strcpy (Str, Edit7->Text.c_str());
  float Value[3];
      // определение длины строки/числа символов
  int Len = strlen (Str);
  if (MyEdit == NULL)
   {MyEdit = new TEdit(Form2); // создание экземпляра
    MyEdit->Parent = TabSheet5; // установить родителя
   }
  int j=0; // инициализировать индекс для массива Value
  for (int i=0; i<Len; i++) // делать цикл
    {// пока символ - не цифра, переход к следующему
      while (!IsDigit(Str[i]) && i<Len)i++;
      int Pr; // признак наличия знака минус у числа
       int h=1; // начальное значения степени дробной части
      float S=0; // инициализация начального значения числа
      bool Flag = true; // признак - цифры до точки
             // определение знака числа
      if (Str[i-1]=='-') Pr=1; else Pr=0;
```

```
            // цикл пока внутри числа
        while ((IsDigit(Str[i])||Str[i]=='.') && i<Len)
          {   // если точка, то смена значения признака
            if (Str[i]=='.') {Flag = false; i++; continue;}
            if (Flag) // если целая часть
              S = S*10 + (int)Str[i] -48;
            else    // если дробная часть
              S = S+((int)Str[i] - 48)*pow(0.1,h++);
            i++; // переход к следующему символу
          }
              // запись суммарного значения в массив
         if (Pr) *(Value + j++)= -S; else *(Value + j++)= S;
      }
        // определить исходные данные
    float x = Value[0];
    float k = Value[1];
    float n = Value[2];
        // установить свойства динамического редактора
    MyEdit->Width = 140;
    MyEdit->Height = 35;
    MyEdit->Top = 120; MyEdit->Left = 190;
        // вычислить функцию
    y = pow(tan(2*k),(float)1/7)+x+n;
        // записать в окно редактора
    MyEdit->Text = FloatToStrF (y, ffFixed, 4, 3);
}
```

Задача 1.7. Создать приложение, имитирующее работу простейшего калькулятора, позволяющего выполнить набор операций над операндами целого и веществом типа.

Решение

Главная форма, в нашем случае она является единственной, должна содержать средства:

- ввода числовых значений операндов;
- отображения результата вычисления;
- команды для инициации вычислений/действий калькулятора.

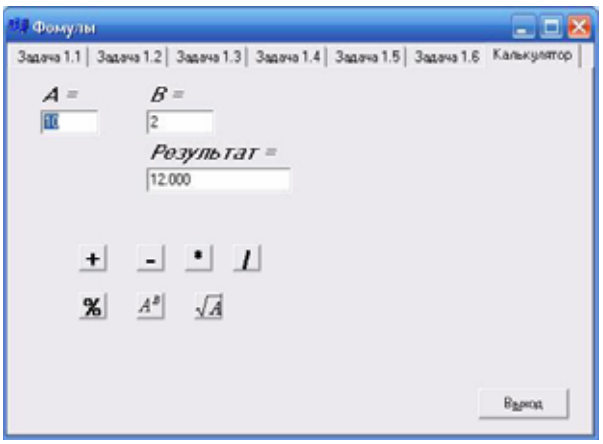


Рис. 1.7. Окно программы «Калькулятор»

Предположим, что интерфейс программы «Калькулятор» имеет вид, как на рис. 1.7. Для реализации такого интерфейса потребовалось бы три компонента `TLabel` – для фиксации поясняющего текста и три компонента `TEdit` – для ввода и вывода данных. Но в данном проекте использована группа всего лишь из трех компонентов `TLabeledEdit`, совмещающая пару этих компонентов, которые мы назовем: `AEdit`, `BEdit` – для ввода значений исходных данных (операндов) и `ResEdit` – для вывода результата.

Для обеспечения функциональности приложения, а именно: выполнения процедур вычисления в соответствии с выбранным действием и закрытия программы будут использоваться кнопки `TSpeedButton` (7 компонентов). Для решения задачи следует подготовить форму, установить свойства, которые очевидны и следуют из рис. 1.7, а затем сформировать обработчики событий. Обработчики событий, обеспечивающих действия калькулятора, просты и не требуют дополнительных пояснений. Коды реализации обработчиков событий приведены в листинге 1.8.

Листинг 1.8. Тексты обработчиков событий, обеспечивающих работу калькулятора

```
void __fastcall TForm2::SpeedButton1Click(TObject *Sender)
// выполняет сложение
{ float a, b;
  float y;
  a = StrToFloat(AEdit->Text); // взять первое число из Edit
  b = StrToFloat(BEdit->Text); // взять второе число из Edit
  y = a + b; // сложение
  ResEdit->Text = FloatToStrF (y, ffFixed, 6, 3);
}
void __fastcall TForm2::SpeedButton5Click(TObject *Sender)
  // остаток от деления
{ float a, b;
  int y;
  a = StrToFloat(AEdit->Text); // взять первое число из Edit
  b = StrToFloat(BEdit->Text); // взять второе число из Edit
  y = (int)a % (int)b; // целочисленное деление
  ResEdit->Text = IntToStr(y);
}
void __fastcall TForm2::SpeedButton6Click(TObject *Sender)
  // возведение в степень
{
  float a, b;
  float y;
  a = StrToFloat(AEdit->Text); // взять первое число из Edit
  b = StrToFloat(BEdit->Text); // взять второе число из Edit
  y = pow (a, b); // возвести в степень
  ResEdit->Text = FloatToStrF(y, ffFixed, 6, 3);
}
void __fastcall TForm2::SpeedButton7Click(TObject *Sender)
  // корень квадратный
{ float a;
  float y;
  a = StrToFloat(AEdit->Text); // взять число из Edit
  y = sqrt(a); // извлечь корень
  ResEdit->Text = FloatToStrF (y, ffFixed, 6, 3);
}
```

Примечание

Здесь не приведены обработчики событий для кнопок, инициализирующих операции вычитания, умножения и деления. Эти обработчики полностью совпадают с обработчиком операции сложения и отличаются только строкой выполнения соответствующей операции.

Задача 1.8. Создать приложение, имитирующее работу стандартного калькулятора, принятого в обиходе и имеющего вид, как на рис. 1.8.



Рис.1.8. Внешний вид главной формы программы «Калькулятор»

Решение

Интерфейсы калькуляторов, представленных на рис. 1.7 и 1.8 отличаются: во втором случае для отображения чисел предусмотрена отдельная строка – поле вывода. А сами цифры, из которых складываются числа, набираются с помощью предусмотренных для этих целей «кнопок-цифр» или с помощью клавиш клавиатуры.

Порядок действий при вычислении на таком калькуляторе следующий:

- набирается число – операнд_1;
- «нажимается» знак операции;
- набирается число – операнд_2 (при этом старое значение стирается);
- нажимается знак «=» (команда «Вычислить»).

После чего в строке вывода отражается результат вычислений.

В качестве операндов могут быть использованы значения стандартных констант π и e. Эти константы выставлены на форме отдельными кнопками. Кнопка «CE» очищает поле вывода. Кнопка «C» стирает последний символ в строке вывода.

Для построения интерфейса, представленного на рис. 1.8, потребуются компоненты всего двух видов: 21 компонентов-кнопок типа `TBitBtn` и один компонент типа `TStaticText` – для строки чисел.

СВОЙСТВА КОМПОНЕНТОВ

- Компоненты: `BtnBtn1` – `BtnBtn21` (кнопки): `Caption` (заголовки) – «0», «1», «2», «3», «4», «5», «6», «7», «8», «9», «+»; «–»; «*», «/», «=», «+/–», «.», «π», «e», «C», «CE»; `Name` (наименования) – «bb0»…«bb9», «bbPlus», «bbMinus», «bbMult», «bbDiv», «bbEquel», «bbSign», «bbComma», «bbDelLast», «bbPi», «bbClear», «bbE».

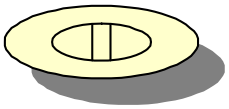
- Компонент `StaticText1` (статический текст): `Caption` (заголовок метки) – «0»; `Name` (наименование) – «stDisplay».

Для того чтобы обеспечить требуемые действия калькулятора, в программе предусмотрено 11 процедур, каждая из которых оформлена в виде метода класса `Form1`, причем 9 из этих процедур прописаны, как обработчики событий. Эти процедуры в своей работе опираются на *общие/глобальные переменные*, которые целесообразно включить также в поля класса формы в раздел **`private`**. Описания этих переменных и их назначение представлены в листинге 1.9.

Листинг 1.9. Описание переменных общего назначения, которые включены в поля класса формы

```
Boolean Fract; // признак присутствия знака «точка» в числе

            // рабочие переменные для выполнения операций
long double Register1, // хранит первый операнд
              Register2; // хранит второй операнд
int FractDigit;  // кол-во знаков после запятой
int Sign;  // знак числа (положительное или отрицательное)
char OperSign;  // знак операции
char RegText [51];  // строка для сборки числа
char DecSep [2]; // DecimalSeparator – разделители ".","
```



ПРОЦЕДУРЫ И ОБРАБОТЧИКИ СОБЫТИЙ:

- `void __fastcall TForm1::DigitalClick (TObject *Sender);` – вызывается при нажатии на кнопки «0»…«9». Она использует указанный явным образом стандартный параметр `Sender` – объект-отправитель события;
- `void __fastcall TForm1::bbCommaClick (TObject *Sender);` – устанавливает знак «,» – отделение дробной части;
- `void __fastcall TForm1::bbEquelClick(TObject *Sender);` – обработчик события `OnClick` для кнопки «=». Обеспечивает выполнение соответствующей операции;
- `void __fastcall TForm1::Display (void);` – функция, предназначенная для формирования строки, составляющей число с учетом знака;
- `void __fastcall TForm1::FormCreate (TObject *Sender);` – обработчик события `OnCreate` главной формы. Служит для установки начальных значений переменных. Используется функция `Display`;
- `void __fastcall TForm1::bbDelLastClick (TObject *Sender);` – обработчик события `OnClick` для кнопки «C» – удалить последний введенный символ;
- `void __fastcall TForm1::bbClearClick (TObject *Sender);` – обработчик события `OnClick` для кнопки «CE» – очистить строку ввода;
- `void __fastcall TForm1::bbOperateClick(TObject *Sender);` – обработчик событий `OnClick` для кнопок, предназначенных для выполнения операций: «+»; «–»; «*», «/»;
- `void __fastcall TForm1::bbSignClick(TObject *Sender);` – обработчик событий `OnClick` для кнопки «+/–» – знак числа. Переключает знак числа, переключая его между значениями положительное/отрицательное;

- `void __fastcall TForm1::bbConstClick(TObject *Sender);` – обработчик событий `OnClick` кнопок «π», «*e*».

Как следует из описания, некоторым кнопкам, например кнопкам «0»…«9», предоставлена возможность пользоваться одним и тем же обработчиком событий. Определение требуемых действий в этом случае реализуется с помощью параметра `Sender`.

Общую логику работы калькулятора можно описать следующим образом:

- нажатие кнопок с цифрами пополняет строку дисплея соответствующей цифрой;
- по нажатию кнопок с константами произойдет замена набранного числа на значение этой константы;
- нажатие кнопки «,» дополняет текст числа знаком, отделяющим дробную часть;
- нажатие кнопки смены знака меняет знак перед числом с плюса на минус или наоборот;
- нажатие кнопок операций приводит к фиксации набранного числа как первого аргумента, сохранению кода операции и подготовки переменных приложения к набору второго аргумента;
- нажатие кнопки «=» приводит к выполнению соответствующей операции и отображению результата;
- нажатие кнопки «CE» приводит к начальной установке всех переменных и отображению на дисплее стартового значения «0»;
- нажатие кнопки «C» стирает последний символ. При удалении очередного последнего символа он анализируется и в соответствии с этим выполняются установки переменных – полей класса.

Установка начальных значений полей класса

Алгоритм, реализованный в обработчике события `OnCreate` главной формы, содержит операторы присваивания глобальным переменным начальных значений (для переменной с типом `char*` – вызов функции копирования). Код обработчика представлен в листинге 1.10.

Листинг 1.10. Тексты обработчика события `OnCreate` главной формы

```
void __fastcall TForm1::FormCreate (TObject *Sender)

{   // инициализация
      // начальное значение кол-ва знаков после запятой
  FractDigit = 0;
  Fract  = False;
  Sign = 1; // начальное значение знака
  strcpy (RegText, ""); // обнуление числового поля
  Display ();// формирование строки ввода
  DecSep [0] = DecimalSeparator;
  DecSep [1] = '\0';
// чтобы форма смотрела код клавиши ДО компонента на форме
  KeyPreview = true;
}
```

СВОЙСТВО KeyPreview

Свойство `KeyPreview` определяет, будут ли события, связанные с нажатием клавиши, предварительно обрабатываться формой как объектом или эти события будут обра-

батываться только компонентом, на котором установлен фокус, минуя соответствующий обработчик формы.

Подсоединение разделителя «точка» к строке ввода данных

Алгоритм включения символа разделителя дробной части, реализованный в обработчике события `OnClick` для кнопки «,», можно описать следующими действиями:

- установить признак того, что число содержит точку в положение «истина»;
- подсоединить знак «точка» к строке-числу;
- запретить нажимать кнопку «точка» повторно;
- вызвать функцию для формирования строки.

Код обработчика представлен в листинге 1.11.

Листинг 1.11. Тексты обработчика события `OnClick` для кнопки «,»

```
void __fastcall TForm1::bbCommaClick(TObject *Sender)
{ Fract = True; // признак присутствия точки
  strcat (RegText, DecSep); // присоединить ”.”
      // запретить при наборе вторую точку
  bbComma->Enabled = False;
  Display(); // вызвать функцию для формирования строки
}
```

Очистка строки ввода

Алгоритм обработчика события `OnClick` для кнопки «CE» сводится к выполнению операторов присваивания, которые устанавливают значения переменных в стартовые позиции. В нем также очищается строка, предназначенная для сборки числа. Код обработчика представлен в листинге 1.12.

Листинг 1.12. Тексты событий, обеспечивающих `OnClick` для кнопки «CE» – очистить строку ввода

```
void __fastcall TForm1::bbClearClick (TObject *Sender)
{ StrCopy (RegText, ""); // очищается строка
  Fract = False; // признак, что число целое
  FractDigit = 0; // число знаков в дробной части
  Sign = 1; // знака у числа нет
  Display (); // вызвать функцию для формирования строки
  bbComma->Enabled = false; // запрет на знак ”,”
  bb0->Enabled = false; // запрет на ”0”
}
```

Удаление последнего введенного символа

Алгоритм обработчика события `OnClick` для кнопки «C» должен удалять последний введенный символ:

- если стираемый символ – разделитель, то кнопку «,» сделать доступной;
- сместить в строке для сборки числа символ конца строки `'\0'` на одну позицию влево;
- если строка для сборки числа пуста, то кнопки «C» и «,» сделать недоступными;
- вызвать функцию для формирования строки.

Код обработчика представлен в листинге 1.13.

Листинг 1.13. Тексты обработчика событий `OnClick` для кнопки «С» – удалить последний введенный символ

```
void __fastcall TForm1::bbDelLastClick(TObject *Sender)
{ if (RegText [StrLen (RegText) - 1] == DecSep [0])
    bbComma->Enabled = True;

  RegText [StrLen (RegText) - 1] = '\0';

  if ( ! StrLen (RegText))
   {
    bbDelLast->Enabled = false;
    bbComma->Enabled = false;
    bb0->Enabled = false;
   }

  Display ();
}
```

Обработчик события ввода очередной цифры числа

Эта процедура-обработчик связана с несколькими компонентами (кнопками с цифрами) – отправителями события. Следовательно, перед выполнением действий следует уточнить, какая из кнопок-цифр находится в фокусе. Проверяется адрес компонента-отравителя события (параметр `Sender`) на соответствие кнопке-цифре, затем такая цифра и добавляется к строке, предназначенной для сборки символов. Например: `if (Sender == bb5) StrCat (RegText, "5");`. Такой оператор просто приписывает цифру «5» к тексту числа, если нажата кнопка «5». Небольшая «хитрость» в том, что на старте приложения (еще при разработке) кнопки «CE», «,», «0» запрещены, а после ввода первого символа они разрешаются, так как, когда набран хотя бы один символ, его можно стереть, отделить от дробной части или набрать другой нуль. Код обработчика представлен в листинге 1.14.

Листинг 1.14. Описание обработчика событий `OnClick` для кнопок «0»…«9»

```
void __fastcall TForm1::DigitalClick (TObject *Sender)
{ if (Sender == bb0) StrCat (RegText, "0");
  if (Sender == bb1) StrCat (RegText, "1");
  if (Sender == bb2) StrCat (RegText, "2");
  if (Sender == bb3) StrCat (RegText, "3");
  if (Sender == bb4) StrCat (RegText, "4");
  if (Sender == bb5) StrCat (RegText, "5");
  if (Sender == bb6) StrCat (RegText, "6");
  if (Sender == bb7) StrCat (RegText, "7");
  if (Sender == bb8) StrCat (RegText, "8");
  if (Sender == bb9) StrCat (RegText, "9");

  Display (); // вызывается процедура
      // устанавливаются значения переменных
  if (StrLen (RegText) == 1)
   {bbDelLast->Enabled = true;
    bbComma->Enabled = true;
    bb0->Enabled = true;
   }
}
```

Метод отображения числа `Display`

В алгоритме используется вспомогательная переменная `buff`, предназначенная для хранения первого операнда. Проверяется, набрано ли число (его значение временно хранится в глобальной переменной `RegText`). Если число введено, то оно копируется во вспомогательную переменную. При этом к ней добавляется знак, если есть необходимость. Если число не введено, то в `buff` помещается 0. Код реализации данного метода представлен в листинге 1.15.

Листинг 1.15. Описание метода `Display`, предназначенного для формирования строки, составляющей число с учетом знака

```
void __fastcall TForm1::Display (void)
{ char buff [52];
  if (StrLen (RegText)) //если строка сборки числа не пуста
    if (Sign > 0) // если число положительное
                  // то поместить его в строку ввода
      stDisplay->Caption=RegText;
    else // если число отрицательное
     {
       StrCopy (buff, "-"); // в buff поместить знак "-"
            // поместить знак в buff и добавить к строке ввода
       stDisplay->Caption = StrCat (buff, RegText);
     }
  else  // если пустая строка то в строку ввода поместить
    stDisplay->Caption = "0";  // просто 0
}
```

Реакция калькулятора на знак «=»

В результате выполнения события `OnClick` для кнопки «=», набранное (второй операнд) число помещается в `Register2`, после чего обеспечивается выполнение соответствующей операции, которая определяется знаком, записанным в переменную `OperSign`, а затем общие переменные переводятся в «старт». Код обработчика представлен в листинге 1.16.

Листинг 1.16. Текст обработчика события `OnClick` для кнопки «=»

```
void __fastcall TForm1::bbEquelClick(TObject *Sender)
{
// переписать операнд из строки RegText в число Register2
  Register2 = StrToFloat (RegText);

      // выбрать действия по знаку операции
  switch (OperSign)
   {case '+' : Register2 += Register1; break;
    case '-' : Register2 = Register1 - Register2; break;
    case '*' : Register2 *= Register1; break;
    case '/' : Register2 = Register1 / Register2; break;
   }
      // результат поместить в поле ввода/вывода
  stDisplay->Caption = FloatToStr (Register2);
  Register1 = 0; // очистить переменную
  StrCopy (RegText, ""); // очистить строку
      // установить начальные значения общих переменных
  Fract = False;
```

```
  FractDigit = 0;
  Sign = 1;
  bbComma->Enabled = false;
  bb0->Enabled = false;
}
```

Реакция калькулятора на ввод символа операции

Действия обработчика событий `OnClick` для кнопок, предназначенных для сообщения знака операций: «+», «–», «*», «/», очень просты:

- переписывается введенное число (первый операнд) из строки `RegText` в переменную `Register1`;
- устанавливаются стартовые значения переменных для приема второго операнда;
- знак операции заносится в переменную `OperSign`.

Код обработчика представлен в листинге 1.17.

Листинг 1.17. Описание обработчика событий `OnClick` для кнопок, предназначенных для выполнения операций: «+», «–», «*», «/»

```
void __fastcall TForm1::bbOperateClick(TObject *Sender)
{ Register1 = StrToFloat (RegText);
  StrCopy (RegText, "");
  Fract = False;
  FractDigit = 0;
  Sign = 1;
  bbComma->Enabled = false;
  bbDelLast->Enabled = false;
  bb0->Enabled = false;
  if (Sender == bbPlus) OperSign = '+';
  if (Sender == bbMinus) OperSign = '-';
  if (Sender == bbMult) OperSign = '*';
  if (Sender == bbDiv) OperSign = '/';
}
```

Реакция калькулятора на команду смены знака

Алгоритм этого обработчика прост: у соответствующей переменной `Sign` меняется знак и вызывается функция визуализации числа. Код его реализации приведен в листинге 1.18.

Листинг 1.18. Тексты обработчика событий `OnClick` для кнопки «+/–» – знак числа

```
void __fastcall TForm1::bbSignClick(TObject *Sender)
{ Sign = -Sign;
  Display ();
}
```

Реакция калькулятора на ввод констант «π», «e»

Ввод специальных значений – констант «π», «e» – выполняется единой подпрограммой по простому алгоритму: анализируется адрес кнопки и в переменную `RegText` вводится текст соответствующий значению той или иной константе. Реализация этой процедуры обработки приведена в листинге 1.19.

Листинг 1.19. Тексты обработчика событий `OnClick` кнопок «π», «e»

```
void __fastcall TForm1::bbConstClick(TObject *Sender)
```

```
// берет значения стандартных констант из библиотеки
// <math.h> и копирует их с преобразованием в RegText
{  if (Sender == bbPi)
      StrCopy (RegText, FloatToStr (M_PI).c_str ());
  if (Sender == bbE)
      StrCopy (RegText, FloatToStr (exp (1)).c_str ());

  Display ();
}
```

Реакция калькулятора на событие `KeyUp`

Для удобства работы пользователя ему, помимо кнопок на форме, предоставляется возможность использовать клавиатуру. Все клавиши клавиатуры имеют индивидуальный код. Например, клавиша «0»…«9» имеют коды 0x0030…0x0039 или 0x0060…0x0069. Диапазон кодов зависит от того, на какой клавиатуре нажали клавишу – на основной или дополнительной. Аналогично есть коды и у клавиш со знаками операций. Клавиш с таким знаком на клавиатуре тоже несколько.



Событие OnKeyUp наступает при отпускании пользователем любой клавиши, включая как функциональные, так и вспомогательные, такие как `Shift`, `Alt` и `Ctrl`. Параметр `Sender` указывает на компонент, который отправил сообщения о событии. Параметр `Key` определяет клавишу клавиатуры, которая отпускается. Параметр `Shift` является множеством, включающим элементы: `ssShift` (клавиша `Shift`), `ssAlt` (клавиша `Alt`), `ssCtrl` (клавиша `Ctrl`). Это позволяет определить удерживалась ли соответствующая служебная клавиша в момент нажатия основной.

Алгоритм работы этого обработчика достаточно прозрачен. Производится анализ кода нажатой клавиши и в соответствии с кодом клавиши просто вызывается обработчик соответствующей кнопки формы. Ведь нажата ли кнопка на форме или клавиша, редактор должен выполнить те же самые действия. Анализ клавиш разделен на две группы в зависимости от того, нажата ли модифицирующая клавиша `Shift` или нет. Дело в том, что знаки «+» и «*» можно набрать на основной клавиатуре только с использованием модификатора `Shift`. Чтобы форма перехватывала это событие и вызывала этот обработчик, необходимо на стадии разработки или в начале работы программы свойству `KeyPreview` формы присвоить значение `true`. В качестве действий, соответствующих тем или иным клавишам, вызывается одна из ранее разработанных подпрограмм. Ведь помимо того, что обработчик имеет жестко определенный список параметров и подключается к соответствующему указателю на событие, он так и остается просто методом класса. А следовательно, может и вызываться как обычный метод. Так, например, при нажатии на клавиатуре цифровых клавиш обработчик `KeyUp` вызывает процедуру `DigitalClick (TObject *Sender)` с указателем `bb0` или с другим указателем на кнопку формы, имитируя нажатие такой кнопки на форме, а самое главное, корректно выполняя необходимые при этом действия. Аналогично обрабатываются и клавиши операций процедурой `bbOperateClick(TObject *Sender)`. При нажатии клавиши с точкой или запятой вызывается метод `bbCommaClick (NULL)` с произвольным, а лучше с пустым указателем NULL,

так как известно, что в нем не делается анализ указателя. Текст обработчика события KeyUp формы приведен в листинге 1.20.

Листинг 1.20. Текст событий KeyUp, обеспечивающих ввод данных для калькулятора с клавиатуры

```
void __fastcall TForm1::FormKeyUp(TObject *Sender,
                                  WORD &Key, TShiftState Shift)
{ if (Shift.Contains (ssShift))   // нажата клавиша Shift
   {switch (Key)
     {case 0x0038 : bbOperateClick(bbMult); break;
      case 0x00BB : bbOperateClick(bbPlus); break;
     }
   }
  else   // нет клавиши Shift
    switch (Key)
     {case 0x0030 : case 0x0060 :
                if (bb0->Enabled) DigitalClick (bb0); break;
      case 0x0031 : case 0x0061 : DigitalClick (bb1); break;
      case 0x0032 : case 0x0062 : DigitalClick (bb2); break;
      case 0x0033 : case 0x0063 : DigitalClick (bb3); break;
      case 0x0034 : case 0x0064 : DigitalClick (bb4); break;
      case 0x0035 : case 0x0065 : DigitalClick (bb5); break;
      case 0x0036 : case 0x0066 : DigitalClick (bb6); break;
      case 0x0037 : case 0x0067 : DigitalClick (bb7); break;
      case 0x0038 : case 0x0068 : DigitalClick (bb8); break;
      case 0x0039 : case 0x0069 : DigitalClick (bb9); break;

      case 0x006E :  // точка или запятая
      case 0x00BC :
      case 0x00BE :
           if (bbComma->Enabled) bbCommaClick (NULL); break;

      case 0x00BB : bbEquelClick (NULL); break;
                                          // знак равенства
                // операции умножения и сложения
      case 0x006A : bbOperateClick(bbMult); break;
      case 0x006B : bbOperateClick(bbPlus); break;
                // операции вычитания и деления
      case 0x00BD : case 0x006D:
                           bbOperateClick(bbMinus); break;
      case 0x00BF : case 0x006F:
                           bbOperateClick(bbDiv); break;

      case 0x0008 : bbOperateClick (bbDelLast); break;
                                        // клавиша BackSpace

//  default : MessageDlg (IntToHex (Key, 4), mtInformation,
//                        TMsgDlgButtons() << mbOK, 0);
     }
```

В приведенном тексте, в теле обработчика, в операторе switch оставлен оператор default, но закомментирован. Если убрать комментарий с этого оператора, то во время работы программа будет выдавать коды клавиш, которые еще не включены в работу. Дополнив текст обработчика анализом этих кодов можно расширить возможности калькулятора.



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. В заданном целом четырехзначном числе поменять местами 1 и 2, 3 и 4 цифры.

Рекомендации: сначала найти цифры числа, воспользовавшись операциями: «%» – остаток от деления и «/» – деления целых, а затем собрать цифры в число в требуемом порядке. Например, в числе x = 8564, применение операции x%10 →4 (выделит левую крайнюю цифру числа), а применение операций x/100%10→5 (выделит вторую цифру от начала).

2. Даны три числа. Наибольшему из этих чисел присвоить сумму двух остальных.
3. Определить квадрант декартовой системы, в котором находится точка A(a1, a2).
4. Определить, принадлежит ли точка A(a1, a2, a3) фигуре, полученной удалением из шара с центром в точке (1, 2, 3) и радиусом 2, шара с центром в точке (1, 2, 3) и радиусом 1.
5. Определить, лежат ли две точки A(a1, a2) и B(b1, b2) по одну сторону от прямой $y = k \cdot x + b$.
6. Определить, лежат ли две точки A(a1, a2) и B(b1, b2) на одном расстоянии от прямой $A \cdot x + B \cdot y + C = 0$.
7. Определить, пересекаются ли две прямые $A1 \cdot x + B1 \cdot y + C1 = 0$ и $A2 \cdot x + B2 \cdot y + C2 = 0$?
8. Прямая $A \cdot x + B \cdot y + C = 0$ пересекает окружность с центром, расположенным в начале координат, и радиусом R. Внутри окружности взята точка. Определить, расположена ли эта точка относительно прямой с той же стороны, что и центр окружности?
9. Определить, могут ли три заданные точки A(a1, a2), B(b1, b2) и C(c1, c2) быть вершинами треугольника?
10. Даны три вещественных числа, из которых два отличаются друг от друга целой частью. Найти это третье число.
11. Заданы два прямоугольника со сторонами, параллельными осям координат. Определить, имеют ли они общую область.

Алгоритмы обработки строковых данных методами компонентов C++ Builder, содержащих списки строк

2

Система *C++ Builder* обеспечивает возможность визуального программирования с помощью библиотеки визуальных компонентов *VCL* (*Visual Components Library*). Эта библиотека содержит большое число классов, предназначенных для быстрой разработки приложений. По своей сути все компоненты – это классы, которые располагаются на определенном уровне иерархии и образуют дерево классов.

Работу почти всех компонентов, содержащих элементы работы со строковыми данными, обеспечивает общий для них класс-предок `TStrings`. Практическое применение имеет также класс `TStringList` – наследник класса `TStrings`.

КЛАССЫ TStrings и TStringList

Методы этих классов позволяют: хранить списки строк; добавлять и удалять строки в указанных позициях списка строк; перестраивать и упорядочивать последовательность строк; получать доступ к конкретным строкам; читать и записывать списки строк в файлы и потоки; связывать с каждой строкой некоторый объект. В класс `TStringList` включены дополнительные возможности, такие как автоматический контроль повторяемости или дублирования строк, сортировка строк. В качестве элемента массива класса `TStringList` используется структура, которая объединяет *строку и объект*.

Методы классов `TStrings` и `TStringList` доступны из программного кода проекта, поэтому логично было бы прописывать код решения задачи с учетом максимального использования этих методов как составной части компонентов интерфейса.

Задача обработки строковых данных, как и практически любая задача, может иметь несколько решений. Предположим, требуется расположить некоторый заданный список строк в алфавитном порядке. Можно решить эту задачу традиционным способом: загрузить список строк, например из файла, в оперативную память, организовав некую область памяти строковых данных; отсортировать их, пользуясь одним из классических методов сортировки; затем сохранить их в другом файле.

В *C++ Builder* можно эту же задачу решать, пользуясь непосредственно методами компонентов, которые были заложены при разработке визуальных компонентов-списков, например, таких как `TListBox` и `TComboBox`, которые, в свою очередь, используют классы `TStrings` и `TStringList`.

Потомки абстрактного класса `TStrings` играют основную роль в компонентах списках. Основой, вокруг которой строится вся логика списка, является свойство, обеспечивающее хранение элементов списка:

```
__property AnsiString Strings[int Index];
```

Обращение, например, к отдельному элементу списка `ListBox` для присваивания значения может осуществляться через это свойство следующим образом:

```
ListBox->Items->Strings[1] = " ";
```

При помощи такого простого присваивания можно задавать новые значения только тогда, когда элемент списка уже создан. Для добавления элемента используются методы `Add` и `AddString`.

Примечание

Попытка прямого использования в приложении класса `TStrings` вызовет ошибку использования абстрактного класса на этапе выполнения программы, а именно при попытке заполнить список значениями. В проектах используются только *потомки этого абстрактного класса*. Для большинства приложений достаточно применения класса `TStringList`.

Помимо работы со списками, *C++ Builder* предоставляет возможность работы с таблицами строковых данных. Для редактирования табличных данных часто применяют компонент `StringGrid`. Центральным свойством компонента `StringGrid` является `Cells` – двухмерный массив ячеек, каждая из которых может содержать произвольный текст. Конкретная ячейка определяется парой чисел – номером столбца и номером ряда, на пересечении которых она находится (нумерация начинается с нуля). Свойство `Cells` имеет тип `AnsiString`, поэтому программа может легко прочитать или записать содержимое нужной ячейки в текстовом виде. Например,

- чтение строки из ячейки `StringGrid`, расположенной в строке `Row` и в столбце `Col`, в соответствующий элемент матрицы. Если элементы матрицы целочисленные, то предварительно необходимо преобразовать строку в число, а уже затем пересылать:

```
a[Row][Col] = StrToInt(StringGrid->Cells[Col][Row]);
```

- запись в `StringGrid` элемента матрицы, расположенного в строке 3 и в столбце 4:

```
StringGrid->Cells[4][3] = IntToStr(a[3][4]);
```

Среди свойств `StringGrid` в первую очередь следует выделить два – `Cols` и `Rows`, – имеющие описание:

```
__property Classes::TStrings* Cols[int Index];
__property Classes::TStrings* Rows[int Index];
```

Значения свойства `Cols` представляет собой список строк, содержащих значения всех ячеек столбца с номером `Index`. Значения свойства `Rows` представляет список строк, содержащих значения всех ячеек строки с номером `Index`.

Примечание

Все предложенные к рассмотрению задачи из главы 2 выполнены в рамках одной формы с использованием компонента `PageControl`. Решения строятся таким образом, что с каждой задачей связана отдельная страница блокнота `PageControl`.

Задача 2.1. В текстовом файле содержится массив строк, причем в каждой строке расположено одно слово. Требуется расположить строки в алфавитном порядке. Переупорядоченные строки сохранить в отдельном файле.

Решение

Для решения поставленной задачи можно не разрабатывать специальных алгоритмов, а лишь правильно воспользоваться возможностями, заложенными в компоненты, например `TStringList`. Для этого достаточно просто задать свойство `Sorted` компонента `TStringList`. Следовательно, интерфейс программы нужно формировать из компонентов, которые имеют в своем составе поле класса `TStringList`.

Предлагается следующая стратегия решения задачи.

1. Определяется, какие визуальные компоненты нужно использовать для того, чтобы можно было воспользоваться методами класса `TStringList`, предназначенными для упорядочения строк.
2. Определяется последовательность действий пользователя, необходимых для решения задачи.
3. Разрабатывается интерфейс приложения.
4. Подготавливаются специальные процедуры для обеспечения действий пользователя, направленных на решение задачи.

Подготовка решения в соответствии с описанной стратегией предполагает уяснение и получение ответов на следующие вопросы.

1. Для того чтобы отсортировать строки, содержащиеся в файле, целесообразно воспользоваться визуальными компонентами – списками `TListBox`, поскольку этот компонент включает поле класса `TStringList`.
2. Пользователю следует предоставить возможность выполнить действия:
   - прочитать информацию с исходными данными из файла;
   - выполнить команду «Сортировать»;
   - получить отсортированные данные;
   - записать отсортированную информацию в новый файл данных.
3. Интерфейс приложения может быть таким, как это показано на рис. 2.1. Для отображения данных используются два компонента `ListBox` с именами, например: `ListBoxSource2_1`(для исходных/неотсортированных строк) и `ListBoxSolve2_1` (для отсортированных строк). Предусмотрены три кнопки для выполнения действий пользователя: «Читать данные» (кнопка `btLoad2_1`), «Сортировать» (кнопка `btSort2_1`), «Сохранить данные» (кнопка `btSave2_1`). На форме помимо кнопок, предназначенных для обеспечения действий пользователя, представлены два невизуальных компонента `TOpenDialog` и `TSaveDialog`. Они нужны для того, чтобы вызвать диалоговые окна и выбрать имя файла *для чтения* исходных строк, и задать имя файла *для записи* упорядоченных строк.
4. Для обеспечения действий пользователя необходимо подготовить три процедуры-обработчика события `OnClick` кнопок управления:
   - `btLoad2_1Click (TObject *Sender);` – обработчик события `OnClick` кнопки «Читать данные»;
   - `btSort2_1Click (TObject *Sender);` – обработчик события `OnClick` кнопки «Сортировать»;
   - `btSave2_1Click (TObject *Sender);` – обработчик события `OnClick` кнопки «Сохранить данные».

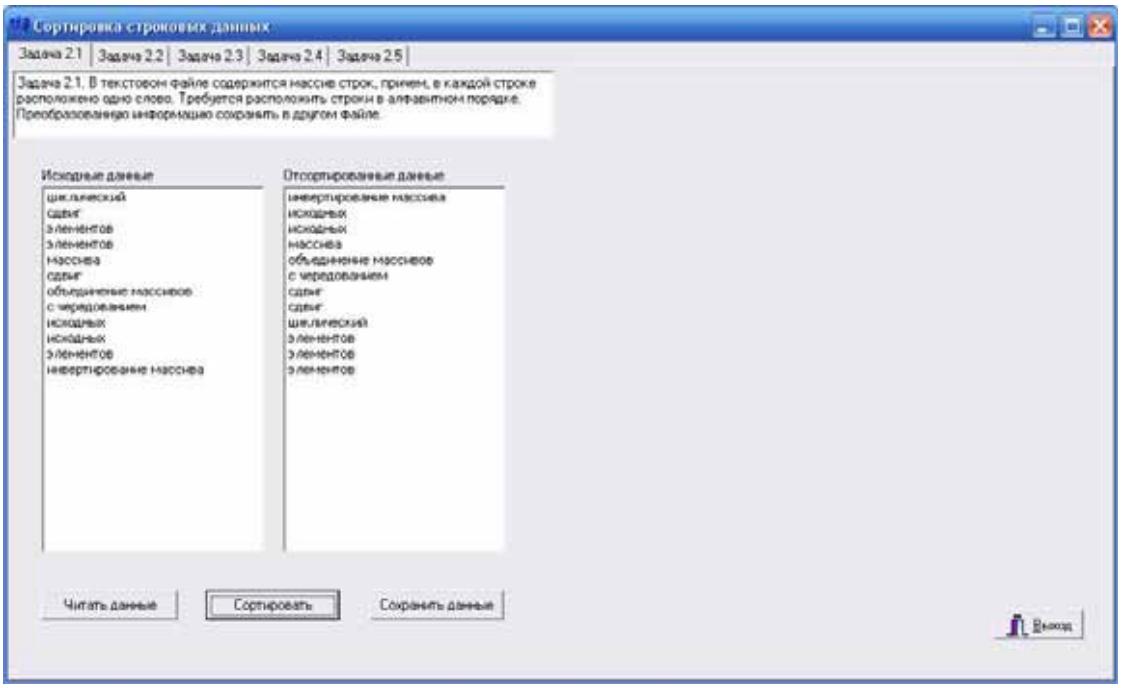


Рис. 2.1. Интерфейс приложения для задачи 2.1

Описание действий команды «Взять из файла»

Для чтения строк из файла целесообразно воспользоваться методом `LoadFromFile` класса `TStrings`, предназначенным для импорта набора строк:

```
void TStrings::LoadFromFile(const AnsiString aFile);
```

Здесь `aFile` – имя файла. И тогда решение сводится к выполнению стандартного диалога `OpenDialog` и вызову метода `LoadFromFile`. Код обработчика событий приведен в листинге 2.1.

Листинг 2.1. Описание действий команды «Взять из файла».

```
void __fastcall TfoMain::btLoad2_1Click (TObject *Sender)
{
  if (OpenDialog->Execute())
    ListBoxSource2_1->Items->
                        LoadFromFile (OpenDialog->FileName);
}
```

Описание действий команды «Отсортировать»

В результате выполнения этой команды список строк должен переместиться из `ListBoxSource2_1` в отведенное окно списка `ListBoxSolve2_1` в отсортированном виде. Для сортировки строк следует для списка `ListBoxSolve2_1` установить свойство `Sorted` в положение `True`, а затем скопировать все строки из `ListBoxSource2_1` в `ListBoxSolve2_1`. Код этого обработчика события приведен в листинге 2.2.

Листинг 2.2. Описание действий команды «Отсортировать»

```
void __fastcall TfoMain::btSort2_1Click (TObject *Sender)
{
  ListBoxSolve2_1->Sorted = True;
  ListBoxSolve2_1->Items = ListBoxSource2_1->Items;
}
```

Описание действий команды «Сохранить данные»

Для записи строк в файл целесообразно воспользоваться методом `SaveToFile` класса `TStrings`, предназначенным для импорта набора строк:

```
void TStrings::SaveToFile(const AnsiString aFile);
```

И тогда решение сводится к вызову и выполнению стандартного диалога `SaveDialog` и вызову метода `SaveToFile`. Код реализации обработчика событий приведен в листинге 2.3.

Листинг 2.3. Описание действий команды «Сохранить данные»

```
void __fastcall TfoMain::btSave2_1Click (TObject *Sender)
{
  if (SaveDialog->Execute())
    ListBoxSolve2_1->Items-> SaveToFile (SaveDialog->FileName);
}
```

Задача 2.2. В файле находятся строковые данные по одному слову в строке. Часть из этих данных требует корректировки: из заданного набора слов удалить слова, начинающиеся с заданной буквы. Требуется поместить модифицированную отсортированную информацию в новый файл.

Решение

Для решения задачи можно воспользоваться стратегией и интерфейсом, описанным в задаче 2.1. Следует только заменить значения полей `Name`, а также название и содержательную подкладку одной из трех кнопок: вместо кнопки «Отсортировать» поставить кнопку «Решить задачу» (с именем `btSolve2_2`) и подготовить обработчик событий под эту кнопку в соответствии с условием задачи. Интерфейс задачи представлен на рис. 2.2. Для того чтобы задать начальную букву для удаления слов, на форму выставляется дополнительный компонент `Edit` (с именем `edFirstLetter2_2`). Причем для `Edit` должно быть установлено свойство `MaxLength = 1`, чтобы можно было вводить только одну букву.

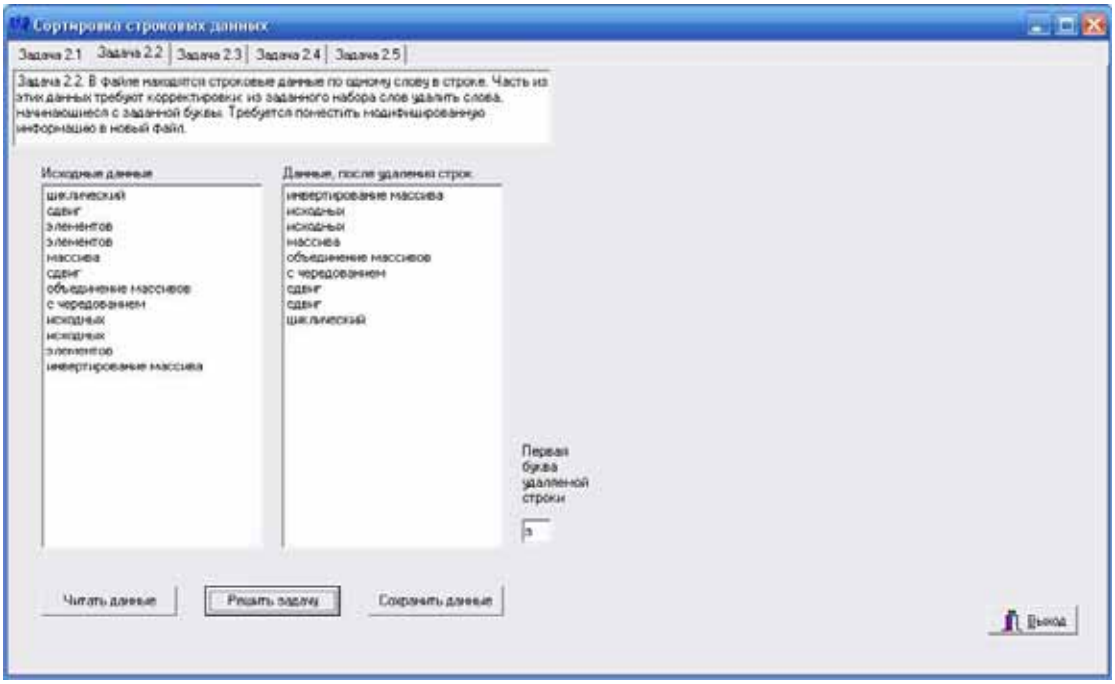


Рис. 2.2. Интерфейс приложения для задачи 2.2

Описание действий команды «Решить задачу»

Решение задачи можно свести просто к переносу строк из списка `ListBox` с исходными строками (`ListBoxSource2_2`) в список `ListBox` с удаленными строками (`ListBoxSolve2_2`), в которых первая буква НЕ совпадает с буквой, указанной в `edFirstLetter2_2`.

Алгоритм удаления строк, начинающихся с заданной буквы, сводится к выполнению следующих шагов:

1) устанавливается свойство `Sorted` в положение `true`;
2) очищается окно списка `ListBoxSolve2_2`;
3) выполняется цикл (по числу строк в `ListBoxSource2_2`), в котором проверяется первая буква текущей строки на совпадение с буквой из `edFirstLetter2_2`, и если первая буква НЕ совпадает с буквой из `edFirstLetter2_2`, то эта строка добавляется к списку `ListBoxSolve2_2`.

Код обработчика событий команды «Сохранить данные» представлен в листинге 2.4.

Листинг 2.4. Описание действий команды «Сохранить данные»

```
void __fastcall TfoMain::btSolve2_2Click(TObject *Sender)
{
  ListBoxSolve2_2->Sorted = true;
  ListBoxSolve2_2->Clear();
  for (int ind = 0; ind < ListBoxSource2_2->Count; ind++)
    if (ListBoxSource2_2->Items->Strings[ind][1]
          != edFirstLetter2_2->Text[1])
      ListBoxSolve2_2->Items->
                  Add (ListBoxSource2_2->Items->Strings[ind]);
}
```

Примечание

Следует напомнить, что строки компонента-списка `ListBox` и строка `Edit` имеют тип `AnsiString`. Индексы литер в строке начинаются с 1, а не с 0, как это принято в массивах символов, например, типа `char St1[10]`, `char* St2` (см. глава 1).

Задача 2.3. В двух текстовых файлах хранится в виде строк некоторая информация, которая может повторяться неоднократно (как внутри одного файла, так и дублироваться в обоих файлах). Переписать эту информацию в отсортированном виде в отдельный файл, исключив избыточную (повторяющуюся) информацию.

Решение

Предположим, что интерфейс приложения будет таким, как это показано на рис. 2.3. Его также можно создать путем копирования элементов интерфейса предыдущей задачи и исправлением значений полей `Name` этих компонентов. А также добавить недостающие элементы.

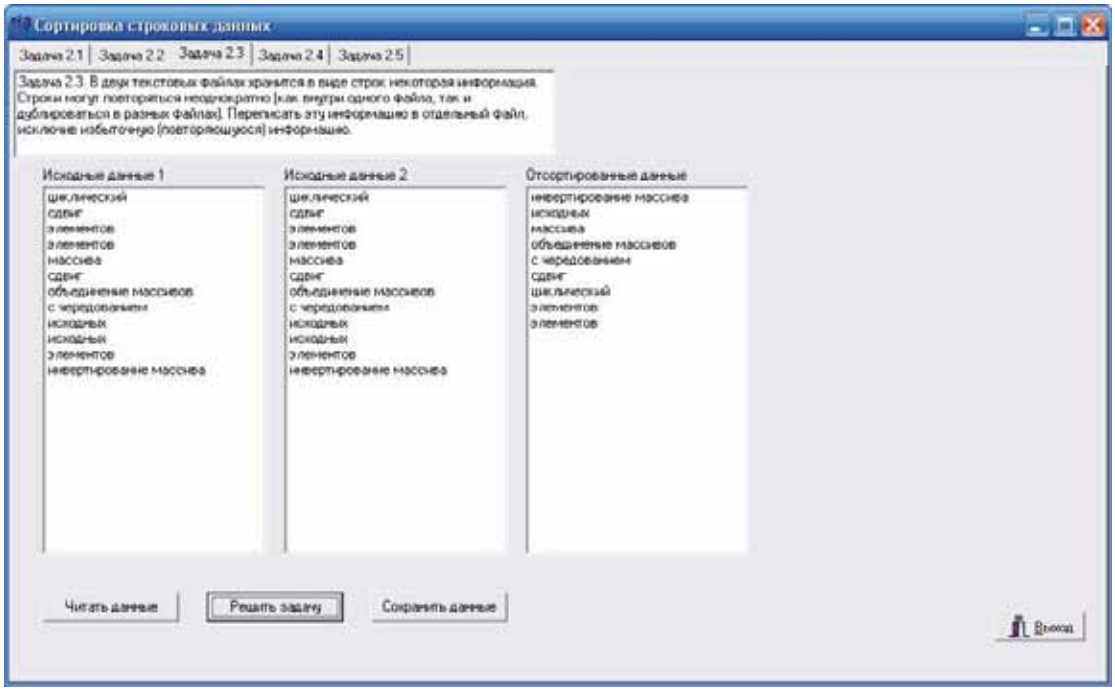


Рис. 2.3. Окно, отображающее результат работы программы для задачи 2.3

Пользователю для решения задачи предлагается выполнить два действия: прочитать данные из файлов с помощью команды «Читать данные» (кнопка `btLoad2_3`) в окна «Исходные данные 1» и «Исходные данные 2» (компоненты `ListBoxSource2_3_1` и `ListBoxSource2_3_2`); получить информацию без дублирующих строк с помощью команды «Решить задачу» (кнопка `btSolve2_3`) в окно «Отсортированные данные» (компонент `ListBoxSolve2_3`). И сохранить в файле отредактированный текст с помощью команды: «Сохранить данные» (кнопка `btSave2_2`). В результате выполнения первой команды будет прочитана информация, содержащаяся в двух входных файлах с отображением в полях компонентов `ListBox` (с именами `ListBoxSource2_3_1`, `ListBoxSource2_3_2`). В результате нажатия второй кнопки в окно третьего `ListBox` (с именем `ListBoxSolve2_3`) будет

помещена сборная информация без дублирующих строк. Третья кнопка «Сохранить данные» записывает содержимое ListBoxSolve2_3 в файле. Приведем тексты обработчиков событий под первые две кнопки.

Описание действий команды «Взять из файла»

Алгоритм действий под кнопку «Взять из файла» прост и содержит следующие шаги:

- выполнить OpenDialog, взять из него имя первого файла и сохранить это имя во вспомогательной строке str1;
- выполнить OpenDialog, взять из него имя второго файла и сохранить это имя во вспомогательной строке str2;
- загрузить строковые данные из файла с именем str1 в компонент ListBoxSource2_3_1;
- загрузить строковые данные из файла с именем str2 в компонент ListBoxSource2_3_2.

Код процедуры-обработчика представлен в листинге 2.5.

Листинг 2.5. Описание обработчика события под кнопку «Взять из файла»

```
void __fastcall TfoMain::btLoad2_3Click (TObject *Sender)
  // чтение исходных данных из двух файлов в два списка
{
  if (OpenDialog->Execute())
    {
      AnsiString str1 = OpenDialog->FileName;
      if (OpenDialog->Execute())
        {
          AnsiString str2 = OpenDialog->FileName;
          ListBoxSource2_3_1->Items->LoadFromFile (str1);
          ListBoxSource2_3_2->Items->LoadFromFile (str2);
        }
    }
}
```

Описание действий команды «Решить задачу»

Для реализации алгоритма «Решить задачу» вводится вспомогательная переменная (назовем ее str) классового типа TStringList. Использование такой переменной с типом TStringList позволит использовать свойство Duplicate.

Алгоритм слияния списков с исключением дублирующей информации можно описать следующими шагами.

1. Создать переменную str (породить объект: список строк).
2. Присвоить свойству Duplicate для созданной переменной str значение dupIgnore (не вставлять дубликаты).
3. Установить для созданной переменной str свойство Sorted - «отсортировать список» в положение true.
4. Передать строки, помещенные в ListBoxSource2_3_1, в созданную переменную str.
5. Добавить к списку str строки из ListBoxSource2_3_2.

6. Переписать преобразованный список строк в другой список – в компонент `ListBoxSolve2_3`.
7. Уничтожить созданный список `str`.

Код процедуры-обработчика представлен в листинге 2.6.

Листинг 2.6. Описание обработчика события под кнопку «Взять из файла»

```
void __fastcall TfoMain::btSolve2_3Click(TObject *Sender)
{
            // породить объект: список строк
  TStringList *str = new TStringList;
  str->Duplicates = dupIgnore;  // не вставлять дубликаты
  str->Sorted = true;
            // дополнить первым списком
  str->AddStrings (ListBoxSource2_3_1->Items);
            // дополнить вторым списком
  str->AddStrings (ListBoxSource2_3_2->Items);
            // очистка списка результирующего
  ListBoxSolve2_3->Clear();
            // копирование в список результирующий
  ListBoxSolve2_3->Items = str;
  delete str;  // уничтожить объект: список строк
}
```

Задача 2.4. Требуется создать таблицу, ячейки которой заполнены случайным текстом. При выделении ячейки программа должна сообщать, имеются ли в выделенном столбце сетки ячейки со совпадающими значениями.

Решение

Подготовка решения предполагает уяснение и проработку следующих вопросов.

1. Пользователь должен предоставить информацию:
   а) о количестве строк и столбцов сетки, которую требуется заполнить случайным текстом (будет задаваться на форме);
   б) о минимальном и максимальном числе символов в тексте, который будет располагаться в отдельной ячейке сетки (будет задаваться локально внутри программы);
   в) о диапазоне, в котором берутся символы текста из алфавита (будет задаваться внутри программы);
   г) о номере столбца, для которого нужно решить задачу о наличии совпадающих значений (будет отмечаться мышкой на сетке).

   Указанные значения будут определять параметры сетки и выходные значения приложения.
2. Пользователю следует предоставить возможность выполнить следующие действия:
   а) задать исходные данные, определяющие размеры таблицы и условия генерации текста;
   б) выполнить команду «Сгенерировать текст», предназначенную для получения таблицы значений. Результат должен быть представлен сеткой значений соответствующего размера;

в) выполнить команду «Ручной размер ячеек» для включения/отключения режима подгона размера ячеек сетки по ширине, посредством манипулятора «мышь»;
г) выполнить команду «Выровнять по ширине текста» для изменения/подгона ширины столбцов сетки под размер текста автоматически, то есть посредством выполнения соответствующей процедуры;
д) выполнить команду «Искать совпадения в колонках», позволяющую
- выделить мышкой нужный столбец;
- принять номер выделенного столбца;
- определить, есть ли в этом столбце ячейки с совпадающими значениями.

В результате выполнения этой команды, в указанном столбце *совпадающие ячейки должны выделяться отдельным* цветом и появляться диалоговое окно с сообщением о результатах проверки.

3. Интерфейс приложения может быть как на рис. 2.4. Согласно рис. 2.4 интерфейс формы включает следующие визуальные компоненты:
   а) метки – для поясняющего текста;
   б) однострочные редакторы `SpinEdit: seRowCount2_4`, `seColCount2_4` – для задания размеров матрицы;
   в) однострочный редактор `edCellNumber2` – для фиксации координат выделенной ячейки;
   г) однострочный редактор `edEqualCount` – для сообщения о количестве одинаковых ячеек в выбранном столбце сетки;
   д) сетка `StringGrid2_4` – для отображения табличных данных;
   е) кнопки, обеспечивающие действия пользователя:
   - `bbGenerate2_4` – сгенерировать текст;
   - `bbMakeWidth2_4` – выровнять по ширине текста;
   - `sbManual2_4` – ручной размер ячеек (необходимо настроить свойства `AllowUp`, `GroupIndex`);
   - `sbSeachEqual` – искать совпадения в колонках.
4. Для обеспечения действий пользователя необходимо подготовить описания обработчиков событий, связанных с каждой из команд:
   - `bbGenerate2_4Click(TObject *Sender);` – обработчик события `OnClick` кнопки «Сгенерировать текст»;
   - `bbMakeWidth2_4Click(TObject *Sender);` – обработчик события `OnClick` кнопки «Выровнять по ширине текста»;
   - `sbManual2_4Click(TObject *Sender);` – обработчик события `OnClick` кнопки «Ручной размер ячеек»;
   - `sbSeachEqualClick (TObject *Sender);` – обработчик события `OnClick` кнопки «Искать совпадения в колонках» (вызывается процедура `DrawCell`);
   - `StringGrid2_4SelectCell(TObject *Sender, int ACol, int ARow, bool &CanSelect);` – обработчик события `OnSelectCell` для сетки, вызывается каждый раз, когда пользователь позиционирует ячейку. Здесь он используется для подсчета числа совпадающих строк внутри отмеченного столбца, в момент выбора ячейки (вызывается процедура `SearchEqualStrings`);

- `void DrawCell (System::TObject *aSender, int aCol, int aRow, const Types::TRect &aRect, TGridDrawState aState);` – процедура для закрашивания отдельной ячейки сетки `StringGrid2_4;`
- `SearchEqualStrings (int aCol);` – процедура поиска первого набора совпадающих ячеек в колонке `aCol`.

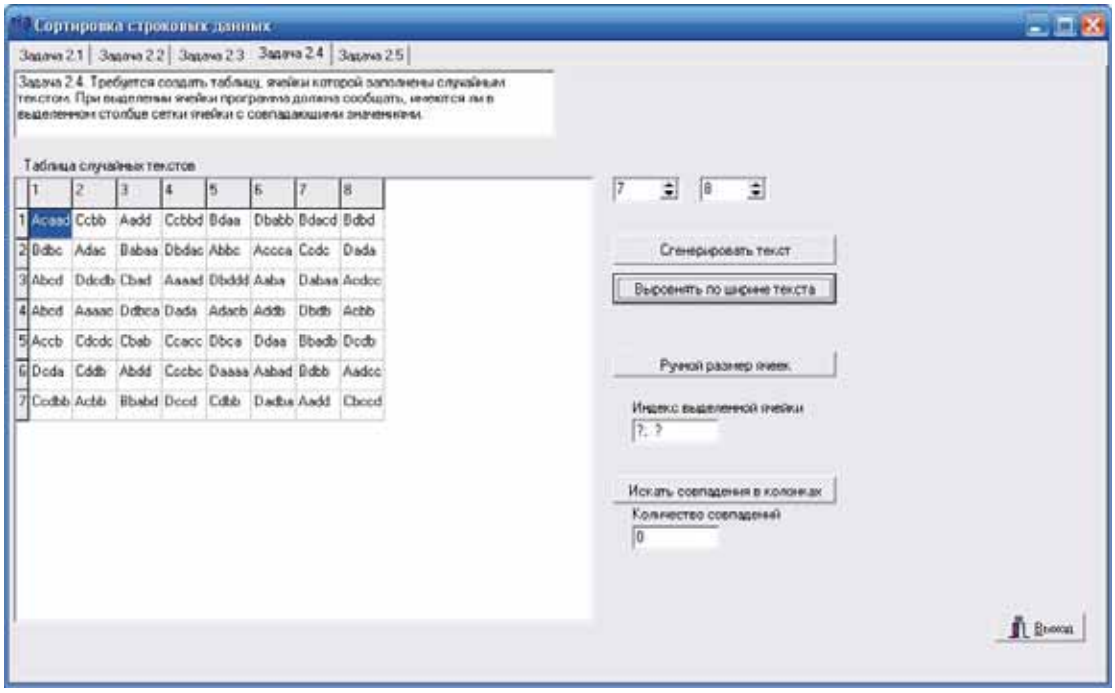


Рис. 2.4. Окно, отражающее результат работы программы для задачи 2.4

В предлагаемой программе используются следующие глобальные переменные:

```
int FEqualCount,  // кол-во совпадений;
          FCol;  // номер выделенной колонки
int FEquals[coMaxCount2_4];  // массив индексов с совпадениями
```

Примечание

Описания глобальных переменных целесообразно поместить в поля класса главной формы (наследника `TForm`), в разделе `protected`.

Генерация случайного текста и заполнение им ячеек `StringGrid`

Для реализации алгоритма используются следующие *внутренние* константы и переменные:

- `coMinNum`, `coMaxNum` – для хранения максимального и минимального количества символов в строке;
- `coLastSimbBig`, `coLastSimbSmall` – для задания заглавного и прописного символов, ограничивающих выбор символа в алфавите символов;
- `cou` – для хранения сгенерированного размера текущей строки;
- `str` – для формирования текущего значения строки.

Алгоритм генерации случайного текста и заполнения им ячеек сетки StringGrid2_4 можно свести к выполнению следующих действий:

1) устанавливаются значения внутренних констант;
2) устанавливаются размеры сетки StringGrid2_4;
3) выполняется цикл (по числу строк) для того, чтобы выставить номера строк в фиксированной области сетки StringGrid2_4;
4) выполняется цикл (по числу столбцов) для того, чтобы выставить номера столбцов в фиксированной области сетки StringGrid2_4;
5) выполняется перебор ячеек таблицы StringGrid2_4 (двойной цикл по числу строк и по числу столбцов сетки StringGrid2_4), в котором:
   а) генерируется целое число cou, которое будет определять длину строки текста, помещаемую в ячейку сетки;
   б) подготавливается строка str к заполнению;
   в) выполняется цикл для генерации символов в строке str;
   г) сформированная строка заносится в очередную ячейку.

Генерация случайного символа из алфавита

Прием, который используется для генерации случайного символа, взятого из символов английского алфавита, основывается на том факте, что коды символов, расположенных в алфавитном порядке, также имеют порядковые значения, которые отличаются друг от друга на единицу. При этом наименьший код соответствует букве «A», наибольший – букве «Z», а код «B» отличается от кода «A» на единицу. И тогда случайный символ str можно получить по формуле:

```
str = (char)(random ('Z' - 'A' + 1) + 'A');
```

Следует обратить внимание, что выражение в скобках ('Z' - 'A' + 1) в результате вычисления будет преобразовано к типу int и, соответственно, будет определять диапазон для выбора случайного кода. Число будет сгенерировано в диапазоне от 0 до 26. Затем сгенерированное случайное число *подвигается* на код буквы «A». Конечно, начальный и конечный символы алфавита можно устанавливать произвольно в соответствии с требованием задачи.

Код процедуры генерирования текста представлен в листинге 2.7.

Примечание

Следует обратить внимание, что при записи номеров ячеек в фиксированную область сетки преобразование индекса в текст производится оператором класса AnsiString.

Листинг 2.7. Описание обработчика события под кнопку «Сгенерировать текст»

```
void __fastcall TfoMain::bbGenerate2_4Click(TObject *Sender)
  /* Генерация случайного текста и заполнение ячеек StringGrid */
{
            // минимальное и максимальное количество символов в тексте
  const coMinNum = 3, coMaxNum = 5;
  const char coLastSimbBig = 'D', coLastSimbSmall = 'd';
            // установка размера
  StringGrid2_4->RowCount = seRowCount2_4->Value + 1;
```

```
  StringGrid2_4->ColCount = seColCount2_4->Value + 1;

  for (int iRow = 1; iRow < StringGrid2_4->RowCount; iRow++)
    StringGrid2_4->Cells[0][iRow] = iRow;
  for (int iCol = 1; iCol < StringGrid2_4->ColCount; iCol++)
    StringGrid2_4->Cells[iCol][0] = iCol;

  for (int iRow = 1; iRow < StringGrid2_4->RowCount; iRow++)
    for (int iCol = 1; iCol < StringGrid2_4->ColCount; iCol++)
     {
        int cou = random (coMaxNum - coMinNum) + coMinNum;
        AnsiString str = "";
        str = (char)(random (coLastSimbBig - 'A' + 1) + 'A');
        for (int ind = 0; ind < cou; ind++)
        str = str + (char)(random (coLastSimbSmall - 'a'
                                                   + 1) + 'a');
        StringGrid2_4->Cells [iCol][iRow] = str;
     }
}
```

Подгон ширины столбцов сетки под ширину текста, расположенного в ячейках сетки

Для реализации алгоритма используются следующие вспомогательные переменные: `coDelta` определяет добавку к ширине текста; `maxWidth` применяется для хранения максимального размера строки, помещаемой в ячейку сетки; `aux` – для размера текста текущей ячейки. Константа `coDelta` используется, чтобы рассчитать размер ячейки больше длины помещаемого текста. Тогда текст не будет сливаться с границей ячейки.

Алгоритм подгона ширины столбцов под ширину текста подробно можно описать следующим образом.

1) Устанавливается величина добавки `coDelta` к ширине текста.
2) Выполняется цикл по числу столбцов:
   а) инициализируется начальное значение переменной `maxWidth` – максимального размера строки текста;
   б) выполняется цикл по числу строк:
      - определяется текущее значение ширины ячейки;
      - текущее значение проверяется на максимум `maxWidth`. Если ширина больше текущего максимума, то максимум `maxWidth` переопределяется;
   в) устанавливается ширина столбца в соответствии с максимальным размером и с учетом добавки `coDelta`.

Код процедуры-обработчика события под кнопку «Выровнять по ширине текста» представлен в листинге 2.8.

Листинг 2.8. Описание обработчика события под кнопку «Выровнять по ширине текста»

```
void __fastcall TfoMain::bbMakeWidth2_4Click (TObject *Sender)
{
  const coDelta = 4;  // добавка к ширине текста
  for (int iCol = 0; iCol < StringGrid2_4->ColCount; iCol++)
    {
       int maxWidth = 0;
       for (int iRow = 0; iRow < StringGrid2_4->RowCount; iRow++)
```

```
          {
                // спрашиваем ширину текста в ячейке
            int aux = StringGrid2_4->Canvas->
                TextWidth(StringGrid2_4->Cells [iCol][iRow]);
            maxWidth = (maxWidth > aux) ? maxWidth : aux;
          }
        StringGrid2_4->ColWidths [iCol] = maxWidth + coDelta;
    }
}
```

Включение/выключение ручной настройки ширины столбцов сетки

Назначение кнопки «Ручной размер ячеек» состоит в том, чтобы обеспечить/отключить возможность изменять размеры ячеек сетки `StringGrid2_4` вручную, а именно: если кнопка нажата, то можно изменять размеры ячеек вручную; если кнопка не нажата, то нельзя изменять размеры ячеек вручную. В зависимости от нажатия кнопки, программно устанавливается соответствующее значение свойства `Options` сетки `StringGrid2_4`.



Для добавления соответствующего значения из предусмотренных значений к свойству `Options` используется переопределенная операция <<, а для исключения соответствующего значения из списка свойств – операция >>.

Алгоритм переключения режима ручной настройки прост и сводится к проверке, нажата ли кнопка: если кнопка нажата, то значение `goColSizing` исключается из списка `Options`, иначе – включается. Код процедуры-обработчика события под кнопку «Ручной размер ячеек» представлен в листинге 2.9.

Листинг 2.9. Описание обработчика события под кнопку «Ручной размер ячеек»

```
void __fastcall TfoMain::sbManual2_4Click(TObject *Sender)
{
  if (sbManual2_4->Down)// если кнопка нажата
    StringGrid2_4->Options =
                  StringGrid2_4->Options << goColSizing;
  else  // если кнопка не нажата
    StringGrid2_4->Options =
                  StringGrid2_4->Options >> goColSizing;
}
```

Обработчик события «Искать совпадения в колонках»

В задачу обработчика события «Искать совпадения в колонках» входит:

- обнулить счетчик и список совпадений;
- подключить собственную процедуру `DrawCell` для закраски совпадающих ячеек цветом, отличным от цвета выделенного столбца (массив совпадений уже получен в обработчике `OnSelectCell`). Или, наоборот, отключить ее, оставив только стандартную процедуру;
- запустить перерисовку таблицы.

В предложенном алгоритме требуется закрашивать или прорисовывать ячейки цветом, отличным от цвета, предложенного по умолчанию.

СОБЫТИЕ OnDrawCell

Каждый раз, когда надо прорисовать какую-нибудь ячейку, возникает событие `OnDrawCell` и вызывается обработчик этого события. Если прорисовывается вся сетка, то обработчик вызывается для каждой ячейки отдельно. Процедура-обработчик этого события имеет 5 параметров:

- `Sender` – передается указатель на объект, который сгенерировал событие;
- `ARow` – номер строки ячейки, которую необходимо прорисовать;
- `ACol` – номер столбца ячейки, которые необходимо прорисовать;
- `Rect` – структура, в которой указаны *относительные* размеры и положение ячейки на холсте `Canvas` компонента `StringGrid`;
- `State` – текущее состояние ячейки: выделена, имеет фокус ввода, фиксирована, нет таких значений.

СТРУКТУРА Rect

Структура `Rect` содержит четыре параметра – `Left`, `Top`, `Right`, `Bottom` – для задания левой, верхней, правой и нижней позиций или координат. Величина, соответствующая координатам, задается в пикселях. Так, в параметре `Right` будет указано расстояние в пикселях от левого края *сетки* до правого края *ячейки*, а в параметре `Bottom` будет расстояние от верхнего края *сетки* до нижнего края *ячейки*. То есть мы получаем размеры *относительно самой сетки*, точнее его холста `Canvas` компонента `StringGrid`, а не всей формы.

Поскольку для решения требуется нестандартная прорисовка совпадающих ячеек, в коде этого обработчика предлагается следующее решение:

1) инициализируются начальные значения для количества совпадений;
2) проверяется, нажата/включена кнопка «Искать совпадения в колонках»;
3) если кнопка нажата, то переопределяется адрес обработчика события `OnDrawCell` – он заменяется адресом подготовленной новой процедуры с именем `DrawCell`, предназначенной для нестандартной прорисовки совпадающих ячеек в выделенном столбце;
4) если кнопка отключена, то
   а) отказ от услуг собственного обработчика `OnDrawCell`;
   б) вызывается полная перерисовка компонента таблицы стандартной процедурой.

Код процедуры-обработчика под кнопку «Искать совпадения в колонках» представлен в листинге 2.10.

Листинг 2.10. Описание обработчика события под кнопку «Искать совпадения в колонках»

```
void __fastcall TfoMain::sbSeachEqualClick (TObject *Sender)
{
  FEqualCount = 0;
  edEqualCount->Text = '0';
  if (sbSeachEqual->Down) StringGrid2_4->OnDrawCell = DrawCell;
  else
   {
    StringGrid2_4->OnDrawCell = NULL;
```

```
    StringGrid2_4->Invalidate();  // перерисовать
   }
}
```

Закрашивание отдельной ячейки сетки

Процедура `DrawCell` будет вызываться в обработчике «Искать совпадения в колонках» для закрашивания ячеек сетки, определенных условием. Она имеет пять параметров, назначение которых описано в представлении обработчика события `OnDrawCell` сетки.

Примечание

Так как данный метод `DrawCell` подменяет стандартный обработчик события `OnDrawCell`, он должен иметь строго определенный список параметров, ПОЛНОСТЬЮ совпадающий с параметрами `OnDrawCell`.

Предложенный алгоритм процедуры `DrawCell`, предназначенной для закрашивания отдельных ячеек сетки, использует значения глобальных переменных `FEqualCount`, `FCol` и `FEquals[coMaxCount2_4]` – массива индексов с совпадениями сводится к выполнению следующих действий.

1) Проверяется условие, нужно ли перекрашивать ячейку (совпадают ли номер столбика текущей ячейки со значением, записанным в глобальную переменную `FCol`).
2) Если номер столбика совпадает с выделенным столбиком `FCol`, то
   а) в цикле проверяется, содержится ли номер строчки текущей ячейки в массиве индексов `FEquals`. Если это так, то
      - выставляется цвет кисти `Brush`(желтый);
      - закрашивается установленным цветом прямоугольник заданного размера, соответствующего размеру ячейки сетки;
      - повторяется процедура вывода текста в ячейку.
      - выход;
   б) если совпадений не найдено, то
      - выставляется цвет, как у выделенного столбца;
      - закрашивается установленным цветом прямоугольник заданного размера, соответствующего размеру ячейки сетки;
      - повторяется процедура вывода текста в ячейку.

ПРОЦЕДУРА TextRect

Процедура `TextRect` предназначена для вывода текста в ячейку. Подробности использования изложены в главе 13.

Процедуру `DrawCell` целесообразно включить в методы класса «Форма». Код реализации процедуры-обработчика представлен в листинге 2.11.

Листинг 2.11. Описание метода закрашивания отдельной ячейки

```
void __fastcall TfoMain::DrawCell (System::TObject *aSender,
  int aCol, int aRow, const Types::TRect &aRect, TGridDrawState aState)
  /* Метод который будет подключен к OnDrawCell */
{
```

```
  if (FCol == aCol)  // если номер колонки выделен
    {
      for (int ind = 0; ind < FEqualCount; ind++)
        {
                  // ячейка в списке совпадений
         if (FEquals [ind] == aRow)
           {
                  // выставляем цвет
             ((TStringGrid *)aSender)->Canvas->Brush->
                                                 Color = clYellow;
            // красим прямоугольник соответствующего размера
        ((TStringGrid *)aSender)->Canvas->FillRect(aRect);

           // повторяем процедуру вывода текста в ячейку
        ((TStringGrid *)aSender)->Canvas->
            TextRect(aRect,aRect.Left+2, aRect.Top+2,
              ((TStringGrid *)aSender)->Cells [aCol][aRow]);
        return;
       }
      }
     if (aRow > 0)
      {
   // совпадений не найдено; красим как выделенный столбец
                                          // выставляем цвет
      ((TStringGrid *)aSender)->Canvas->Brush->Color = clSilver;
      // красим прямоугольник соответствующего размера
      ((TStringGrid *)aSender)->Canvas->FillRect(aRect);
         // повторяем процедуру вывода текста в ячейку
      ((TStringGrid *)aSender)->Canvas->TextRect (aRect,
                             aRect.Left+2, aRect.Top+2,
           ((TStringGrid *)aSender)->Cells [aCol][aRow]);
     }
   }
}
```

Поиск первого набора совпадающих ячеек в заданной колонке

Функция `SearchEqualStrings` имеет один входной параметр `aCol`, задающий номер столбца, в котором осуществляется поиск ячеек с совпадающими значениями. Результатом работы является формирование значений глобальных переменных `FEqualCount`, `FEquals[coMaxCount2_4]`.

Алгоритм поиска совпадающих ячеек можно описать следующими шагами:

1) выполняется цикл по индексам строчек в колонке, где следует искать совпадающие значения:
   а) инициализируется счетчик совпадений;
   б) заносится первый индекс в массив, который будет содержать индексы совпадающих ячеек;
   в) выполняется цикл по номерам ячеек, с которыми следует проверить совпадение и, если ячейки совпадают, то
      - индекс заносится в массив индексов;
      - счетчик совпадений увеличивается на 1;
2) если счетчик совпадений больше 1, то
   а) формируется глобальная переменная `FEqualCount`;
   б) выход;

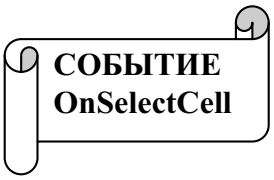
3) иначе глобальная переменная `FEqualCount` = 0.

Код процедуры, предназначенной для поиска совпадающих по значениям ячеек, представлен в листинге 2.12.

Листинг 2.12. Описание функции поиска первого набора совпадающих ячеек

```
void __fastcall TfoMain::SearchEqualStrings (int aCol)
{
  for (int ind = 1; ind < StringGrid2_4->RowCount - 1; ind++)
   {
    int iFree = 1;  // индекс свободной ячейки
    FEquals [0] = ind;  // первый, возможно совпадающий
    for (int jnd = ind + 1; jnd < StringGrid2_4->RowCount; jnd++)
     {
        if (StringGrid2_4->Cells[aCol][ind] ==
                              StringGrid2_4->Cells[aCol][jnd])
        if (iFree < coMaxCount2_4)  // проверяем ограничение
          FEquals [iFree++] = jnd;
     }
    if (iFree > 1)  // нашлись совпадения
     {
        FEqualCount = iFree;
        return;
     }
   }
  FEqualCount = 0;  // все просмотрели -- нет совпадений
}
```

Обработчик события `OnSelectCell` сетки `StringGrid`



Каждый раз, когда выделяется ячейка в сетке `StringGrid`, вызывается обработчик события `OnSelectCell`. Процедура-обработчик события имеет 4 параметра:

- `Sender` – передается указатель на объект, который сгенерировал это событие;
- `ACol` – номер столбца выделенной ячейки;
- `ARow` – номер строки выделенной ячейки;
- `CanSelect` – определяет, может ли ячейка быть выделенной.

Алгоритм, реализованный в обработчике события, сводится к выполнению последовательности действий:

- координаты выделенной ячейки заносятся в окно редактора `edCellNumber2_4;`
- проверяется, нажата ли кнопка «Искать совпадения в колонках». Если это так, то вызывается процедура `SearchEqualStrings` для формирования массива совпадений;
- выполняется перерисовка всей таблицы.

Код обработчика события `OnSelectCell` сетки `StringGrid` представлен в листинге 2.13.

Листинг 2.13. Описание обработчика события `OnSelectCell` выделения ячейки сетки `StringGrid`

```
void __fastcall TfoMain::StringGrid2_4SelectCell(TObject *Sender,
                                int ACol, int ARow, bool &CanSelect)
```

```
{ // выбор/выделение ячейки в сетке строк
  edCellNumber2_4->Text = ARow;
  edCellNumber2_4->Text = edCellNumber2_4->Text + ";  ";
  edCellNumber2_4->Text = edCellNumber2_4->Text + ACol;

  if (sbSeachEqual->Down)
   {
    FCol = ACol;
    SearchEqualStrings (ACol);
    edEqualCount->Text = FEqualCount;
    StringGrid2_4->Invalidate();
   }
}
```

Задача 2.5. В файле хранится информация о содержимом библиотеки следующего вида: автор; название; год издания; цена (как это показано на рис. 2.1). На рисунке параметры литературных изданий отделяются друг от друга символом *. Требуется считать эту информацию, внести при необходимости корректировку в данные: исправить ошибочную информацию; удалить информацию, связанную с утерянными книгами; добавить данные, связанные с новыми поступлениями. Измененную информацию следует занести в новый файл.

```
Суворов В.*       Беру свои слова обратно*      2006*       92.40
Булгаков М.А.*    Ранняя проза*                 1990*       196
Бунин И.А.*       Рассказы*                     1986*       212
```

Рис. 2.5. Пример файла с информацией о содержимом библиотеки

Решение

Решение данной задачи должно выполняться в несколько шагов.

Шаг I. Определяется информация, необходимая для обеспечения работы приложения.

1) Входная:
   а) имя файла, в котором содержатся исходные данные (задается пользователем);
   б) имя файла, в который следует поместить исправленные данные (задается пользователем).
2) Рабочая:
   а) сетка, в которую должны быть занесены данные для просмотра на предмет исправления;
   б) отдельные окна (типа списков `ComboBox`) для хранения полей отдельных строк сетки с целью коррекции.

Шаг II. Пользователю следует предоставить возможность выполнить следующие действия:

1) Задать имя файла, содержащего исходные данные, а также имя файла, куда следует поместить результат коррекции. Здесь следует воспользоваться компонентами: `OpenDialog`, `SaveDialog`. С этими компонентами следует связать два отдельных действия под кнопки, например, «Загрузить из файла» и «Записать в файл». При выполнении «Загрузить из файла» ин-

формация из файла должна отразиться в сетке, отдельно от которой должны быть предоставлены поля ComboBox для возможной коррекции данных.
2) Выполнить команду изменения/подгона ширины столбцов сетки под размер текста.
3) Выполнить команду «Корректировать», позволяющую исправлять данные.
4) Возможность для исправления данных по схеме: пользователь в списке ComboBox выбирает ФИО автора. При этом выборе в ComboBox отражаются соответствующие поля таблицы и одновременно выделяется цветом редактируемая строка в сетке. Внесенные исправления *синхронно* исправляются в соответствующих ячейках сетки.

Интерфейс приложения может быть таким, как это показано на рис. 2.6. Согласно рисунку интерфейс формы включает следующие визуальные компоненты:
1) метки для поясняющих текстов;
2) сетку StringGrid2_5 для отображения данных;
3) списки типа ComboBox с именами:
   а) cbAuthor – для коррекции списка авторов;
   б) cbName – для коррекции списка наименований;
   в) cbYear – для коррекции списка дат изданий;
   г) cbPrice – для коррекции списка цен;
4) кнопки со следующими именами:
   а) btLoad2_5 – «Загрузить из файла»;
   б) btEqualWidth2_5 – «Выровнять по ширине текста»;
   в) sbCorrect2_5 – «Корректировать»;
   г) btSave2_5 – «Записать в файл».

Для обеспечения действий пользователя необходимо подготовить описания следующих процедур и обработчиков событий:

- btLoad2_5Click(TObject *Sender); – обработчик события OnClick кнопки «Загрузить из файла»; вызывает процедуру AddToCombo2_5 для загрузки данных в списки ComboBox;
- btEqualWidth2_5Click(TObject*Sender); – обработчик события OnClick кнопки «Выровнять по ширине текста»;
- sbCorrect2_5Click(TObject *Sender); – обработчик события OnClick кнопки «Корректировать»;
- btSave2_5Click(TObject *Sender); – обработчик события OnClick кнопки «Записать в файл»;
- StringGrid2_5SelectCell(TObject *Sender, int ACol, int ARow, bool &CanSelect); – обработчик события OnSelectCell сетки StringGrid2_5;
- AddToCombo2_5(); – функция для записи столбцов сетки в списки ComboBox;
- cbCombo2_5Exit(TObject *Sender); – обработчик события OnExit списков ComboBox; подключается один ко всем спискам.

Групповой обработчик события

Часто в приложениях встречается ситуация, когда над различными компонентами одного типа по однотипному событию необходимо выполнять единые действия или почти одинаковые. Тогда можно разработать один обработчик события, а на странице *Events* подсоединить его к соответствующему

указателю. Это можно сделать одновременно и к целой группе визуальных компонентов следующим образом: выделяется группа компонентов, а на странице *Events* один раз выполняется подключение обработчика к соответствующему указателю.

В предлагаемой программе используются следующие глобальные переменные:

```
const coSectionNumOf = 4; // число столбцов
int   FRow;  // номер строки
int   FiAuthor,  FiName, FiYear,  FiPrice; // вспом. индексы
```

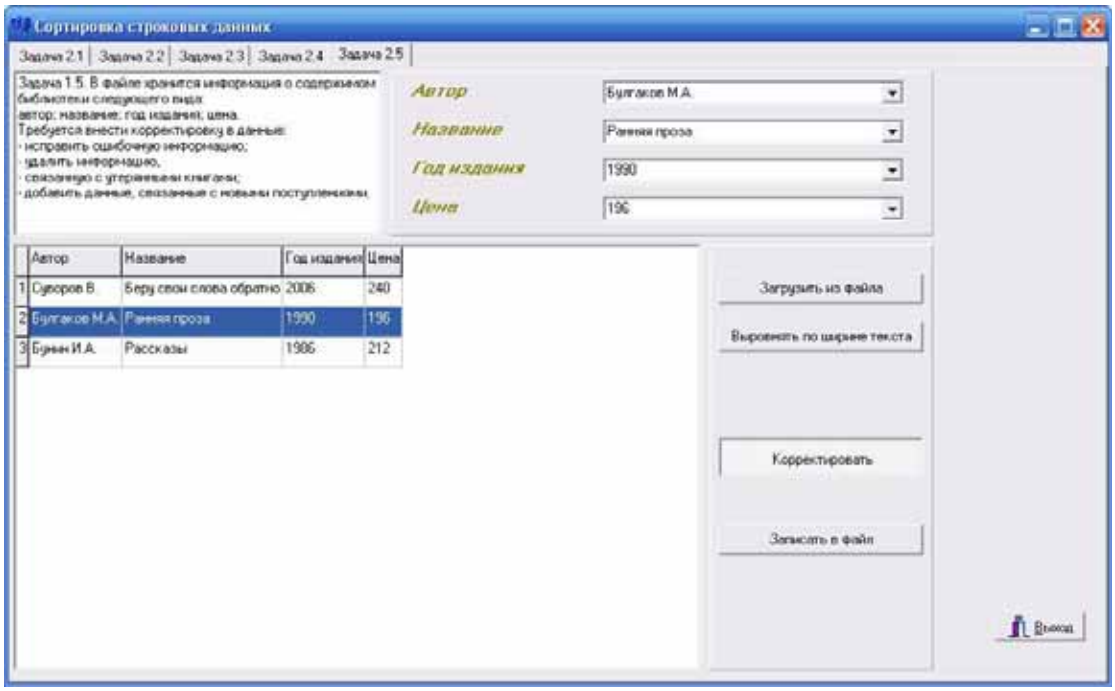


Рис. 2.6. Окно, отражающее результат работы программы для задачи 2.5

Обработчик события «Загрузить из файла»

В соответствии с условием задачи, в файле хранится информация о литературных источниках по строчкам: одна строчка – одна книга. Причем в каждой строчке файла располагается информация об одной книге таким образом, что параметры книги (автор, наименование, год издания, цена) отделяются друг от друга символом «*». Использование в качестве символа разделителя знака «*», а не «пробела» обусловлено тем, что в написании значений параметров присутствует «пробел» как символ.

Особенность загрузки из файла в данном исполнении в том, что требуется не только прочитать все строки файла, но и из каждой прочитанной строки требуется извлечь подстроки, расположенные между разделителями, которые представляют собой параметры книги. Следует отметить, что подстроки (отдельные параметры книги) выделяются, начиная с *последней подстроки*. Это соответствует колонке «Цена» в таблице, затем – «Год» и так далее. В программу вводятся вспомогательные переменные: локальный массив `names`, в котором сохраняются названия столбцов таблицы (эти названия пойдут в «шапку» таблицы); список `list`, куда загружа-

ются все строки из файла; aux – для временного хранения отдельной строки списка (из этой строки будут выделяться параметры книги и заносится в сетку).

В соответствии с изложенными выше пояснениями, алгоритм чтения данных из файла и загрузки его содержимого в сетку StringGrid2_5 можно описать следующими действиями.

1) Если диалоговое окно OpenDialog завершилось по кнопке «OK», то
   а) в список list загружаются строки из файла;
   б) устанавливается номер последней колонки в таблице StringGrid2_5;
   в) устанавливается число строк в таблице StringGrid2_5;
   г) заголовки столбцов из массива names копируются в цикле в «шапку» – фиксированную область таблицы;
   д) номера строк в цикле заносятся в фиксированную область – первый столбец;
   е) выполняется цикл (по числу строк):
      - копируется текущая строка во вспомогательную переменную aux;
      - выделяются из строки aux значения отдельных параметров книги по схеме:
        - находится номер ind последнего разделителя (после этого разделителя расположена последняя подстрока – параметр);
        - выполняется цикл (пока номер разделителя ind > 0):
          - копируется последняя подстрока в сетку StringGrid2_5 в установленную колонку с коррекцией (удалением пробелов слева и справа);
          - стирается в aux последняя подстрока вместе со знаком *;
          - номер колонки уменьшается на 1;
          - находится позиция последнего разделителя (после этого разделителя расположена очередная последняя подстрока); если такого разделителя нет, то переходим к обработке следующей строки.
2) Высвобождается локальный список.
3) Выравниваются столбцы по ширине.
4) Производится загрузка значений из столбцов в списки ComboBox.

Код процедуры-обработчика события OnClick кнопки «Загрузить из файла» представлен в листинге 2.14.

Листинг 2.14. Описание обработчика события OnClick кнопки «Загрузить из файла»

```
void __fastcall TfoMain::btLoad2_5Click(TObject *Sender)
{
 // чтение данных о книгах из файла
  AnsiString names [coSectionNumOf] =
                  {"Автор", "Название", "Год издания", "Цена"};
  TStringList *list;
  list = new TStringList;  // рабочая переменная 'список'
  char *ptr;
  char strChar [80];
           // из файла загрузить строки в StringGrid2_5
  if (OpenDialog->Execute())
   {
     list->LoadFromFile (OpenDialog->FileName);
                 // количество колонок
     StringGrid2_5->ColCount = coSectionNumOf + 1;
```

```
                // количество строк
     StringGrid2_5->RowCount = list->Count + 1;
                // Заголовки в шапку таблицы
     for (int col = 0; col < coSectionNumOf; col++)
     {
       StringGrid2_5->Cells[col+1][0] = names [col];
     }
      // Номера - в первую колонку таблицы
    for (int row = 1; row < StringGrid2_5->RowCount; row++)
      StringGrid2_5->Cells[0][row] = row;

      // Выделение отдельных блоков и запись в ячейки
                              // цикл по числу строк
    for (int iStr = 0; iStr < list->Count; iStr++)
     {
            AnsiString aux = list->Strings [iStr];
                // ищем последний разделитель '*'
          int ind = aux.LastDelimiter ("*"),
          col = StringGrid2_5->ColCount - 1;
             while (ind > 0)
              {
                // подсчитываем длину посл. подстроки
                int len = aux.Length() - ind;
          // копируем последнюю подстроку и подчищаем
          // начальные и конечные пробелы
                StringGrid2_5->Cells[col][iStr + 1] =
                      aux.SubString (ind + 1, len).Trim();
            // стираим последнюю подстроку вместе с '*'

              aux.Delete (ind, len + 1);
              col--;  // на предыдущую колонку
                        // ищем последний разделитель '*'
              ind = aux.LastDelimiter ("*");
            }
          // остатки в первую колонку
         StringGrid2_5->Cells[1][iStr + 1] = aux.Trim();
     }
   }
  delete list;  // высвобождение локального объекта 'список'
            // выравниваем ширину колонок сетки
  btEqualWidth2_5Click (NULL);
  AddToCombo2_5();
}
```

Обработчик события «Выровнять по ширине текста»

Алгоритм обработчика события «Выровнять по ширине текста» описан в задаче 2.4. Решение сводится к перебору всех колонок и определению наибольшего размера из размеров текстов, расположенных в каждой ячейки соответствующей колонки, после чего устанавливается ширина этой колонки `StringGrid`. Код представлен в листинге 2.15.

Листинг 2.15. Описание обработчика события `OnClick` кнопки «Выровнять по ширине текста»

```
void __fastcall TfoMain::btEqualWidth2_5Click
                                          (TObject *Sender)
{
  const coDelta = 4;  // Добавка к ширине текста
```

```
  for (int iCol = 0; iCol < StringGrid2_5->ColCount; iCol++)
   {
    int maxWidth = 0;
    for (int iRow = 0; iRow < StringGrid2_5->RowCount; iRow++)
     {
       int aux = StringGrid2_5->Canvas->  // опрос длины
              TextWidth(StringGrid2_5->Cells [iCol][iRow]);
       maxWidth = (maxWidth > aux) ? maxWidth : aux;
     }
     StringGrid2_5->ColWidths [iCol] = maxWidth + coDelta;
   }
}
```

Обработчик события «Корректировать»

В алгоритме обработчика события `OnClick` кнопки «Корректировать» выполняется проверка, нажата ли кнопка «Корректировать». В зависимости от положения этой кнопки выполняются следующие действия:

1. Если кнопка нажата, то
   - в поля редактирования списков `ComboBox` заносятся те строки (параметры), которые соответствуют выделенной строке сетки `StringGrid`;
   - сохраняются индексы выбранного текста; панель со списками `ComboBox` становится доступной, а сетка `StringGrid` становится недоступной.
2. Если же кнопка «Корректировать» не нажата, то отредактированные данные опять заносятся в сетку в соответствующее место; сетка становится доступной к выделению, а панель – недоступной.

Код обработчика события `OnClick` кнопки «Корректировать» представлен в листинге 2.16.

Листинг 2.16. Описание обработчика события `OnClick` кнопки «Корректировать»

```
void __fastcall TfoMain::sbCorrect2_5Click(TObject *Sender)
{
  if (sbCorrect2_5->Down)// если нажата кнопка
   {
   // загрузить данные о конкретной книге для редактирования
    cbAuthor->ItemIndex = cbAuthor->Items->
                          IndexOf(StringGrid2_5-Cells[1][FRow]);
    cbName->ItemIndex = cbName->Items->
                          IndexOf (StringGrid2_5->Cells[2][FRow]);
    cbYear->ItemIndex = cbYear->Items->
                          IndexOf (StringGrid2_5->Cells[3][FRow]);
    cbPrice->ItemIndex =  cbPrice->Items->
                          IndexOf (StringGrid2_5->Cells[4][FRow]);
                          // индекс выбранного текста
    FiAuthor = cbAuthor->ItemIndex;
                 // синхронизация индексов по ComboBox
    FiName = cbName->ItemIndex;
    FiYear = cbYear->ItemIndex;
    FiPrice = cbPrice->ItemIndex;

    Panel2_5->Enabled = true;
    StringGrid2_5->Enabled = false;
   }
  else  // обратно
```

```
    {
      // загрузить отредактированные данные
      // о конкретной книге обратно в таблицу
      // только если были изменения
     if (FiAuthor != cbAuthor->ItemIndex)
       StringGrid2_5->Cells[1][FRow] = cbAuthor->Text;
     if (FiName != cbName->ItemIndex)
       StringGrid2_5->Cells[2][FRow] = cbName->Text;
     if (FiYear != cbYear->ItemIndex)
       StringGrid2_5->Cells[3][FRow] = cbYear->Text;
     if (FiPrice != cbPrice->ItemIndex)
       StringGrid2_5->Cells[4][FRow] = cbPrice->Text;
     Panel2_5->Enabled = false;
     StringGrid2_5->Enabled = true;
    }
}
```

Обработчик события «Записать в файл»

В данном случае решается задача, которую можно назвать обратной по отношению к задаче «Загрузить из файла». Если в предыдущем случае требовалось из файловой строки выделить подстроки и записать их в ячейки сетки, то здесь нужно взять подстроки из ячеек сетки, собрать их в строки, которые потом нужно будет сохранить в файле.

Примечание

В отличие от структуры исходного файла, описанной ранее и принятой при чтении данных, для выходного файла изменим структуру: введем дополнительную строку в начале файла, в которую занесем параметры таблицы: число строк и число столбцов. Кроме этого предлагается в качестве разделителя использовать символ табуляции `'\t'`, а не символ `'*'`. Хотя это всегда можно поправить по усмотрению пользователя.

Предложенный алгоритм записи таблицы данных в файл можно детализировать следующими шагами:

1) создать вспомогательную переменную `strList` для хранения всех строк списка;
2) создать динамический объект `SaveDialog`;
3) настроить параметры `SaveDialog`;
4) если выполняется `SaveDialog`, то
   а) в список `strList` поместить число строк и столбцов (это будет первая информационная строка в файле);
   б) выполнить цикл по числу строк:
      - инициализировать вспомогательную строку `str`;
      - сформировать строку `str` (выполнить цикл по числу столбцов, в котором к `str` добавляется очередная ячейка таблицы + разделитель `'\t'` (знак табуляции));
      - добавить к списку строку `str`;
   в) список сохраняется в файле;
5) освободить динамические переменные.

Код процедуры-обработчика события `OnClick` кнопки «Записать в файл» представлен в листинге 2.17.

Листинг 2.17. Описание обработчика события `OnClick` кнопки «Записать в файл»

```
void __fastcall TfoMain::btSave2_5Click(TObject *Sender)
{
            // записать в файл обновленный список книг
  TStringList *strList;  // вспомогательный список
  strList = new TStringList;

  int i, j;
  AnsiString str;
                // диалоговое окно 'Сохранить как ...'
  TSaveDialog *sDialog;
                // динамическое создание
  sDialog = new TSaveDialog (foMain);
                // настройка фильтра
  sDialog->Filter ="Файлы с литературой|*.lit";
  sDialog->DefaultExt = "lit";

  if (sDialog->Execute())  // формирование списка литературы
    {
        strList->Add (IntToStr(StringGrid2_5->RowCount-1)
            + " " + IntToStr(StringGrid2_5->ColCount - 1));
        for (i = 0; i < StringGrid2_5->RowCount; i++)
          {
            str = "";
            for (j = 0; j < StringGrid2_5->ColCount; j++)
               str = str + StringGrid2_5->Cells[j][i] + '\t';
            strList->Add (str);
          }
                              // сохранение
        strList->SaveToFile (sDialog->FileName);
    };
  strList->Free();  // высвобождение памяти от объектов
  sDialog->Free();
}
```

Отдельно следует отметить особенность данного кода: компонент диалоговое окно типа `TSaveDialog` создается динамически и уничтожается в конце процедуры.

Формирование списков `ComboBox`

Процедура `AddToCombo2_5` не имеет параметров. Формирование списков сводится к копированию соответствующих столбиков таблицы `StringGrid2_5` в отведенные им списки. Списки `cbAuthor`, `cbName`, `cbYear`, `cbPrice` формируются одинаково по следующей схеме:

- производится очистка соответствующего списка;
- в цикле по числу строчек из соответствующего столбца берется содержимое очередной ячейки, которое добавляется к списку.

Код процедуры представлен в листинге 2.18.

Листинг 2.18. Описание метода, предназначенного для формирования списков `ComboBox` из столбцов таблицы `StringGrid2_5`

```
void __fastcall TfoMain::AddToCombo2_5()
{
  cbAuthor->Clear();
```

```
  for (int ind = 1; ind < StringGrid2_5->RowCount; ind++)
        cbAuthor->Items->Add(StringGrid2_5->Cells[1][ind]);

  cbName->Clear();
  for (int ind = 1; ind < StringGrid2_5->RowCount; ind++)
   cbName->Items->Add (StringGrid2_5->Cells[2][ind]);

  cbYear->Clear();
  for (int ind = 1; ind < StringGrid2_5->RowCount; ind++)
   cbYear->Items->Add (StringGrid2_5->Cells[3][ind]);

  cbPrice->Clear();
  for (int ind = 1; ind < StringGrid2_5->RowCount; ind++)
      cbPrice->Items->Add (StringGrid2_5->Cells[4][ind]);
}
```

Обработчик события `OnSelectCell` сетки `StringGrid`

Такой обработчик автоматически вызывается системой при выборе пользователем ячейки сетки. Здесь этот обработчик используется для настройки глобальной переменной `FRow` и старта процесса перерисовки сетки, с учетом этого нового значения. Алгоритм работы данного обработчика события прост:

- координата (номер строчки) выделенной ячейки заносится в глобальную переменную `FRow`.
- перерисовывается таблица `StringGrid2_5`.

Код обработчика события `OnSelectCell` сетки `StringGrid` представлен в листинге 2.19.

Листинг 2.19. Описание обработчика события `OnSelectCell` сетки `StringGrid`

```
void __fastcall TfoMain::StringGrid2_5SelectCell
      (TObject *Sender, int ACol,int ARow, bool &CanSelect)
{
  FRow = ARow;
  StringGrid2_5->Invalidate();
}
```

Обработчик группового события `OnExit` списков `ComboBox`

В процедуру обработки события вводится вспомогательная переменная `aux`, которой присваивается адрес компонента, передавшего событие. Проверяется, есть ли в списке этого компонента элемент, совпадающий с исправленным текстом. Если нет, то текст добавляется к списку. Код реализации такого обработчика приведен в листинге 2.20.

Листинг 2.20. Описание обработчика события `OnExit` списков `ComboBox`

```
void __fastcall TfoMain::cbCombo2_5Exit(TObject *Sender)
{
      // для сокращения записей дальнейших операторов
    TComboBox *aux = (TComboBox *)Sender;
                                // такого еще нет в списке
  if (aux->Items->IndexOf (aux->Text) == -1)
    aux->Items->Add (aux->Text);
}
```

Задача 2.6. О студентах группы, распределенных на практику в город Тюмень, нужно подать в виде таблицы следующие сведения: название предприятия, на котором студент будет проходить практику; ФИО студента; рейтинг за весь период обучения (округленный до целого значения); домашний адрес. Всего в город Тюмень направляется 5 человек. Для подготовки такой информации предоставлен шаблон таблицы с одним заполненным столбцом, в котором вписаны названия предприятия. Требуется выполнить редактирование таблицы, вписав в нее недостающие сведения. Для редактирования таблицы использовать *встроенные редакторы других визуальных* компонентов.

Решение

По условию задачи задан шаблон таблицы с заготовкой данных. Эту таблицу разумно представить компонентом `StringGrid`. В предыдущих примерах для изменения содержимого ячейки `StringGrid` нами использовался встроенный в компонент `StringGrid` однострочный текстовый редактор.

Но могут быть ситуации, когда информация отображается в текстовом виде, а редактировать ее необходимо каким-либо «экзотическим» способом, например выбирать из списка готовых строк (как в компоненте `ComboBox`) или редактировать как целое число (как в компоненте `CSpinEdit`). Разработать специальный компонент, подобный `StringGrid`, но такой, чтобы отдельные ячейки его редактировались бы специфично, – очень сложная задача. Но если информация сохраняется исключительно в текстовом виде, то в этом нет необходимости.

Примечание

ОС Win32 имеет графический интерфейс. В графике часто используют прием «обмана»: строится изображение несколько иное или другим способом. Важным является лишь то, чтобы у пользователя возникало необходимое впечатление. Здесь это эффект особенного редактирования. Используя этот принцип можно поступать следующим образом: над ячейкой компонента `SringGrid` отображать необходимый однострочный редактор соответствующего размера. При этом у пользователя будет *возникать впечатление*, что в ячейку компонента `SringGrid` встроен некий другой редактор.

Для решения задачи необходимо пользователю обеспечить следующие возможности:

- загрузить шаблон – заготовку таблицы;
- выполнить редактирование этой таблицы;
- установить необходимую ширину ячеек;
- отредактировать данные, пользуясь другими встроенными редакторами;
- занести отредактированные данные в ячейки сетки компонента `SringGrid`.

При этом, в соответствии с заданием, редактирование содержимого ячеек следует производить разными редакторами, например:

- колонка 1 (предприятие) – встроенным редактором самого компонента `TStringGrid`;
- колонка 2 (ФИО) – выбором фамилии из выпадающего списка `TComboBox`;
- колонка 3 (рейтинг) – редактором целочисленных данных `TCSpinEdit`;

- колонка 4 (адрес) – редактором компонента `TEdit`.

Интерфейс такого приложения в действии может иметь вид, как на рис. 2.7.

ПРАВИЛА ИСПОЛЬЗОВАНИЯ сторонних редакторов для исправления данных в сетке **StringGrid**

При разработке проекта с редактированием ячеек компонента `StringGrid` с привлечением редакторов, связанных с другими компонентами, необходимо соблюдать следующие принципы:

1. Компонент-редактор пока не производится редактирование, не должен быть виден.
2. При редактировании ячейки редактор должен отображаться «точно в ячейке», правильнее говоря, он должен отображаться точно **над ячейкой**.
3. Перед началом редактирования текущее значение требуемой ячейки должно быть внесено в компонент-редактор в соответствующем виде. Редактор проявится над требуемой ячейкой.
4. После завершения редактирования данные должны попасть в ячейку в текстовом виде, объект-редактор должен исчезнуть.
5. Между входом в редактор (проявление и настройка объекта-редактора на ячейку `StringGrid`) и выходом из него (то есть момент внесения отредактированного значения в ячейку) проходит большое количество времени (происходят разные вызовы подпрограмм), поэтому для запоминания нужно использовать отдельную глобальную переменную (например, `FLastCombo` – для использования редактора `ComboBox`);
6. *Право на редактирование* ячеек `StringGrid` предоставляет отдельный компонент `CheckBox` (редактировать). По его состоянию задается значение свойства `StringGrid->Options`.

Согласно рис. 2.7 для создания приложения потребуются следующие компоненты: `StringGrid` (с именем `StringGrid2_6`) – для отображения таблицы со сведениями о студентах; кнопка «Загрузить шаблон» (с именем `DoTable);` компонент `CheckBox` (с именем `cbEdit`) – для предоставления прав на редактирование ячеек; компонент `CheckBox` (с именем `cbColSizing`) – для разрешения изменять ширину ячеек таблицы.

Помимо этого для редактирования потребуются компоненты `Edit` (с именем `Edit`), `ComboBox` (с именем `ComboBox`), `CSpinEdit` (с именем `CSpinEdit`) – для предоставления своих редакторов таблице `StringGrid2_6`. Эти компоненты невидимы на этапе выполнения программы, поэтому местоположение компонентов `Edit`, `ComboBox`, `CSpinEdit` **несущественно**.

Существенным является следующее:

- компоненты-редакторы – `Edit`, `ComboBox`, `CSpinEdit` – должны быть помещены на форму позже, чем `StringGrid`. Или в ходе работы проекта эти компоненты должны перемещаться на передний план, например командой: `Edit->BringToFront();`. Тогда при перерисовке эти компоненты-редакторы будут видны на фоне `StringGrid`;

- нет необходимости создавать/уничтожать объекты-редакторы. Достаточно, используя свойство `Visible`, регулировать *видимость/невидимость* этих объектов-редакторов. Причем эти редакторы должны быть невидимыми на старте программы, чтобы пользователь не догадывался об их существовании.

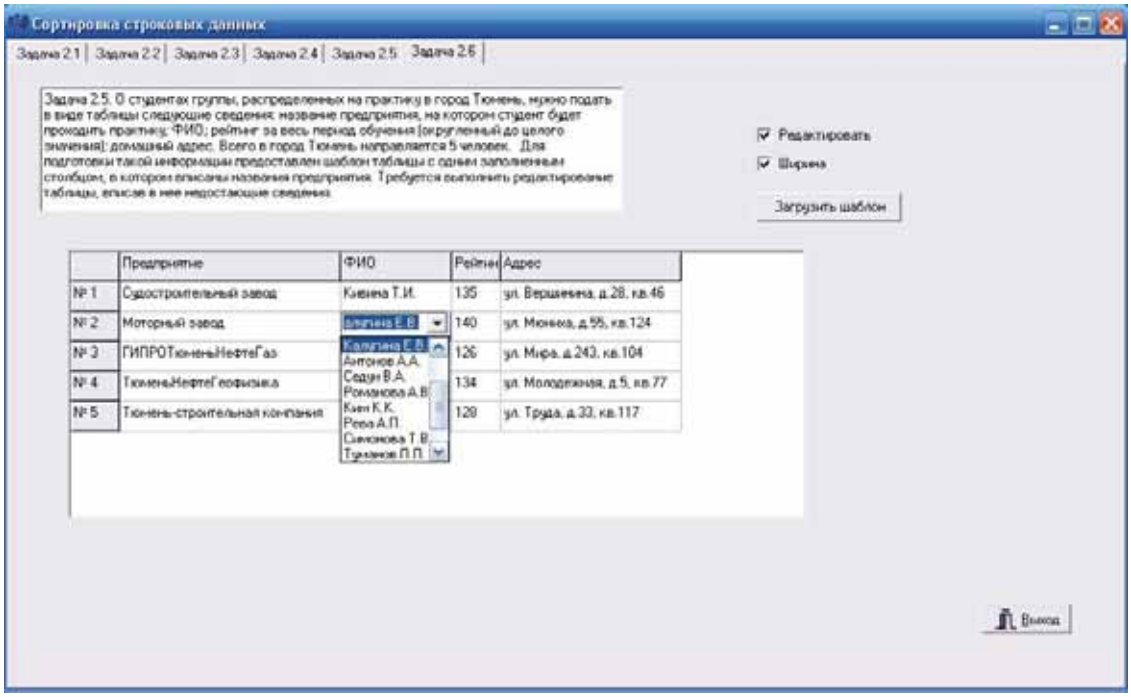


Рис. 2.7. Интерфейс приложения на этапе редактирования табличных данных

ТРУДНОСТИ

В некоторых случаях при перемещении ячеек `StringGrid` эти ячейки могут отображаться частично. Требуется решать вопрос, а какого размера будут объекты-редакторы: соответствующие полной ширине этой колонки или видимой части. Если их размер будет полным, то они будут «выглядывать» за `StringGrid` и, возможно, отображаться на фоне соседних объектов. Как должен отображаться объект-редактор – решает разработчик конкретного приложения. Эта проблема в задаче решается так:

- сравниваются размеры, опрошенные разным способом;
- *если они совпадают, то ячейка видна полностью* – продолжаем, делаем чего надо;
- если не совпадают, то выход из подпрограммы; редактор не включаем.

РАЗМЕРЫ ячеек редактирования

Узнать размеры выбранной ячейки можно:

а) из свойств `StringGrid->ColWidths [ACol]` и `StringGrid->RowHeights [ARow];`

б) из метода `TRect aux = StringGrid->CellRect (ACol, ARow);`. Этот метод возвращает значение типа `TRect`, в котором хранятся все габариты видимой части ячейки.

При этом свойство `ColWidths[ACol]` определяет *полный размер* – ширину колонки. Метод `CellRect(ACol, ARow)` возвращает *размер видимой части* ячейки. При сов-

падении размеров этих двух опросов *можно говорить о полной видимости* ячейки. Аналогично для высоты.

Для обеспечения функциональности приложения, предназначенного для решения задачи, необходимо подготовить следующие обработчики событий:

- `StringGrid2_6SelectCell(TObject *Sender, int ACol, int ARow, bool &CanSelect);` – обработчик события `OnSelectCell` сетки `StringGrid2_6` предназначен для включения или выключения режима редактирования. В качестве входного параметра имеет и адрес выбираемой ячейки: `ACol`, `ARow`;
- `EditClick(TObject *Sender);` – обработчик события `OnClick` редактора `Edit` предназначен для переноса результатов редактирования в ячейки сетки `StringGrid2_6`;
- `cbColSizingClick(TObject *Sender);` – обработчик события `OnClick` кнопки «Ширина» (предоставляет право менять/не менять вручную ширину ячеек таблицы);
- `DoTableClick(TObject *Sender);` – обработчик события `OnClick` кнопки «Загрузить шаблон».

Обработчик события `OnClick` кнопки «Загрузить шаблон»

В задачу события «Загрузить шаблон» входит подготовка шаблона таблицы в следующей последовательности.

1. Выставляются подписи к колонкам и строчкам.
2. Заполняется колонка с названиями предприятий (эти названия считаются заданными).
3. Заполняются остальные колонки произвольным текстом (этот произвольный текст будет полностью отредактирован):
   а) во 2-ую колонку выводятся буквы латиницы с цифрой;
   б) во 3-ую колонку выводятся трехзначные числа вида: 111, 222, 333 …
   в) во 4-ую колонку выводятся буквы кириллицы с цифрой;
   Это, конечно, вольный текст, демонстрирующий необходимость и приемы формирования начального заполнения ячеек таблицы.
4. Загружается список `ComboBox` с фамилиями студентов группы для редактирования;
5. Выставляются значения глобальных переменных.

Следует отметить, что в соответствии с условием задачи размеры шаблона таблицы фиксированы и устанавливаются на этапе подготовки интерфейса в свойствах таблицы `StringGrid2_6`.

Код обработчика события `OnClick` кнопки «Загрузить шаблон» представлен в листинге 2.21.

Листинг 2.21. Описание обработчика события `OnClick` кнопки «Загрузить шаблон»

```
void __fastcall TfoMain::DoTableClick(TObject *Sender)
{
  AnsiString Title[4] = {"Предприятие", "ФИО","Рейтинг","Адрес"};
  AnsiString TitleFirms[5] = {"Судостроительный завод",
                  "Моторный завод", "ГИПРОТюменьНефтеГаз",
      "ТюменьНефтеГеофизика","Тюмень-строительная компания"};
```

```
            // подписывание StringGrid

                  // в нулевую строку -> номер колонки
  for (int iRow = 1; iRow < StringGrid2_6->RowCount; iRow++)
   {
     AnsiString aux = "№ ";
     aux += iRow;
     StringGrid2_6->Cells [0][iRow] = aux;
   }
            // в первый столбец -> названия фирм
  for (int iRow = 1; iRow < StringGrid2_6->RowCount; iRow++)
   {
    StringGrid2_6->Cells [1][iRow] = TitleFirms[iRow-1];
   }
            // в нулевую строку -> заголовки столбцов
  for (int iCol = 1; iCol < StringGrid2_6->ColCount; iCol++)
   {
     AnsiString aux = "К ";
     aux += iCol;
     StringGrid2_6->Cells [iCol][0] = Title[iCol-1];
   }

   // во вторую колонку -> букву латиницы с цифрой
  for (int iRow = 1; iRow < StringGrid2_6->RowCount; iRow++)
   {
    char aux = 'A' + iRow - 1;
    AnsiString aux2 = aux;
    aux2 += iRow;
    StringGrid2_6->Cells [2][iRow] = aux2;
   }

      // в четвертую колонку -> букву кириллицы с цифрой
  for (int iRow = 1; iRow < StringGrid2_6->RowCount; iRow++)
   {
      char aux = 'A' + iRow - 1;
      AnsiString aux2 = aux;      // и во вторую колонку
      aux2 += iRow;
      StringGrid2_6->Cells [4][iRow] = aux2;
   }

//в третью колонку -> трехзначные числа вида: 111,222,333, …
  for (int iRow = 1; iRow < StringGrid2_6->RowCount; iRow++)
   {
     AnsiString aux = 100 * iRow + 10 * iRow + iRow;
     StringGrid2_6->Cells [3][iRow] = aux;
   }
            // загрузка ComboBox
  ComboBox->Items->LoadFromFile("Task2_6.txt");
  FLastEdit = 1;
  FLastCombo = 1;
  FLastSpinEdit = 1;
}
```

Обработчик события `OnSelectCell` сетки `StringGrid2_6`

Такой обработчик, как уже упоминалось, вызывается системой автоматически при выборе пользователем ячейки сетки. Здесь этот обработчик используется для

включения/выключения объекта-редактора. Эта функция в качестве входного параметра имеет адрес выбираемой ячейки: `ACol`, `ARow`.

В обработчике решаются две ситуации, связанные с редактированием: в начальном блоке закрывается объект-редактор, если таковой открыт; во втором блоке, если включена галочка, разрешающая редактировать, соответствующий редактор открывается. Одним словом, если редактировался какой-то элемент, то его значение нужно передать в ячейку `StringGrid` и «спрятать» элемент.

Для отображения объекта-редактора необходимо и достаточно:

- проявить объект: свойству `Visible` присвоить значение `true`;
- установить параметры ячейки редактирования;
- перенести в поле редактора текущее значение из сетки `StringGrid2_6`;
- переключить фокус на объект-редактор;
- запомнить номер редактируемой строки;
- проверить, если было произведено редактирование, то содержимое из поля редактора перенести в ячейку `StringGrid`;
- спрятать объект-редактор.

Примечание

Для правильной работы проекта целесообразно в области памяти *отдельной переменной* фиксировать, какая ячейка (какая строка) подвергается редактированию.

Пояснение. Случаи бывают разные:

- *переходим на редактирование новой ячейки, при этом открыт объект-редактор для другой ячейки*. Тогда по отдельной переменной, например `FLastCombo`, определяем номер ячейки, которая ныне редактируется; из объекта-редактора вносим значение в ячейку `StringGrid`, и прячем этот редактор;
- *включаем объект-редактор, то есть начинаем редактировать как бы изначально*. Просто проявляем объект-редактор в нужной позиции и заносим в него данные из требуемой ячейки.

В соответствии с приведенными выше пояснениями, алгоритм обработчика события `OnSelectCell` сетки `StringGrid2_6` с включением/выключением элементов редактирования можно детализировать следующими шагами.

1. Если объект-редактор `Edit` виден, то необходимо выполнить следующее:
   а) скопировать его поле редактирования в текущую ячейку сетки `StringGrid2_6`;
   б) сделать объект-редактор `Edit` невидимым.
2. Если объект-редактор `ComboBox` виден, то необходимо выполнить следующее:
   а) скопировать его поле редактирования в текущую ячейку сетки `StringGrid2_6`;
   б) сделать объект-редактор `ComboBox` невидимым.
3. Если объект-редактор `CSpinEdit` виден, то необходимо выполнить следующее:
   а) скопировать его поле редактирования в текущую ячейку сетки `StringGrid2_6`;
   б) сделать объект-редактор `CSpinEdit` невидимым.
4. Если было разрешено редактирование – запретить.

5. Получить размер прямоугольника выделенной ячейки и сохранить его во вспомогательной переменной `aux`.
6. Если включен режим редактирования и редактируется ячейка из 4-го столбца, то необходимо выполнить следующее:
   а) устанавливаются свойства окна редактирования `Edit`;
   б) в свойство Text занести текущее значение из ячейки `StringGrid2_6`;
   в) сохраняется номер редактируемой ячейки в глобальной переменной `FLastEdit`;
   г) переключается фокус на объект-редактор `Edit`;
   д) завершается подготовка к редактированию.
7. Если включен режим редактирования и редактируется ячейка из 2-го столбца, и если ячейка видна полностью, то необходимо выполнить следующее:
   а) устанавливаются свойства окна редактирования `ComboBox`;
   б) в свойство Text занести текущее значение из ячейки `StringGrid2_6`;
   в) сохраняется номер редактируемой ячейки в глобальной переменной `FLastCombo`;
   г) переключается фокус на объект-редактор `ComboBox`;
   д) завершается подготовка к редактированию.
8. Если включен режим редактирования и редактируется ячейка из 3-го столбца, и ячейка видна полностью, то необходимо выполнить следующее:
   а) устанавливаются свойства окна редактирования `CSpinEdit`;
   б) в свойство Text занести текущее значение из ячейки `StringGrid2_6`;
   в) сохраняется номер редактируемой ячейки в глобальной переменной `FLastSpinEdit`;
   г) переключается фокус на объект-редактор `CSpinEdit`;
   д) завершается подготовка к редактированию.

Код обработчика события `OnSelectCell` таблицы `StringGrid2_6` представлен в листинге 2.22.

Листинг 2.22. Описание обработчика события `OnSelectCell` таблицы `StringGrid2_6`

```
void __fastcall TfoMain::StringGrid2_6SelectCell(TObject *Sender, int
ACol, int ARow, bool &CanSelect)
{
  if (Edit->Visible)
    {
        StringGrid2_6->Cells[4][FLastEdit] = Edit->Text;
        Edit->Visible = false;
    }
  if (ComboBox->Visible)
    {
       StringGrid2_6->Cells[2][FLastCombo] = ComboBox->Text;
       ComboBox->Visible = false;
    }
  if (CSpinEdit->Visible)
    {
     StringGrid2_6->Cells[3][FLastSpinEdit]
                                          = CSpinEdit->Text;
       CSpinEdit->Visible = false;
    }
        // если было разрешено редактирование -- запретить
```

```
  if (StringGrid2_6->Options.Contains(goEditing))
     StringGrid2_6->Options >> goEditing;

    // опрашиваем размер прямоугольника выделенной ячейки
  TRect aux = StringGrid2_6->CellRect (ACol, ARow);
     // выставляем элемент редактирования в окно ячейки
      //StringGrid2_6; запоминаем какую ячейку редактируем
     // загружаем текущее значение ячейки в элемент
      // в нужном формате
  if (cbEdit->Checked)
    switch (ACol)
     {
           case 4 : Edit->Left =
                          aux.Left + StringGrid2_6->Left + 1;
            Edit->Top = aux.Top + StringGrid2_6->Top + 1;
            Edit->Visible = true;
            Edit->Width = StringGrid2_6->ColWidths [4];
            Edit->Height = aux.Height();
            FLastEdit = ARow;
            Edit->Text = StringGrid2_6->Cells [4][FLastEdit];
            Edit->SetFocus();
            break;
         case 2 :
                     // только если ячейка видна полностью
              if (StringGrid2_6->ColWidths[2]
                          != aux.Width()) break;
             ComboBox->Left = aux.Left +
                          StringGrid2_6->Left + 1;
             ComboBox->Top = aux.Top +
                          StringGrid2_6->Top + 1;
             ComboBox->Visible = true;
             ComboBox->Width =
                          StringGrid2_6->ColWidths [2];
             ComboBox->Height = aux.Height();
             FLastCombo = ARow;
             ComboBox->Text =
                     StringGrid2_6->Cells [2][FLastCombo];
             ComboBox->SetFocus();
             break;
         case 3 :
                     // только если ячейка видна полностью
              if (StringGrid2_6->ColWidths[3]
                     != aux.Width()) break;
             CSpinEdit->Left = aux.Left +
                     StringGrid2_6->Left + 1;
             CSpinEdit->Top = aux.Top +
                     StringGrid2_6->Top + 1;
             CSpinEdit->Width = aux.Width() + 2;
             CSpinEdit->Height = aux.Height();
             CSpinEdit->Visible = true;
             FLastSpinEdit = ARow;
             CSpinEdit->Text =
                     StringGrid2_6->Cells [3][FLastSpinEdit];
               CSpinEdit->SetFocus(); break;
                          // включить редактирование
          default: StringGrid2_6->Options << goEditing;
     }
}
```

Обработчик события «разрешить/запретить» изменять ширину ячейки

Алгоритм обработчика события `OnClick` кнопки «Ширина» предназначен для переключения режима ручной настройки. Он прост и сводится к проверке, нажата ли кнопка: если кнопка нажата, то значение `goColSizing` включается в список `Options`, иначе – исключается.

Код обработчика события `OnClick` кнопки «Ширина» представлен в листинге 2.24.

Листинг 2.24. Описание обработчика события `OnClick` кнопки «Ширина»

```
void __fastcall TfoMain::cbColSizingClick(TObject *Sender)
{
 if (cbColSizing->Checked)
  StringGrid2_6->Options =
                        StringGrid2_6->Options << goColSizing;
 else
  StringGrid2_6->Options =
                        StringGrid2_6->Options >> goColSizing;
}
```



Сетка `StringGrid` может иметь много колонок. Тогда внизу проявляется скроллер с ползунком. Если на этапе редактирования (когда висит объект-редактор) «продернуть» скроллер, то ячейки `StringGrid` переместятся, а объект-редактор останется на месте.



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. В задачи 2.4 значения полей о минимальном и максимальном числе символов в тексте, располагаемом в отдельной ячейке сетки и о диапазоне, в котором берутся символы текста, задается внутри программы. Доработать интерфейс программы, чтобы и эти параметры задавались пользователем.
2. В заданном списке слов есть одинаковые. Определить их порядковые номера.
3. Из заданного списка слов исключить слова, совпадающие с заданным словом.
4. В заданном списке слов поменять местами слова с максимальной и минимальной длинами.
5. Найти слова, имеющие наибольшее число употреблений в заданном списке слов.
6. Имеется таблица, в которой содержится список рабочих, их месячный оклад, количество трудодней. Требуется каждому рабочему начислить премию в размере 30 % от зарплаты. Подсчитать общую сумму премии.
7. Дан список участников соревнования, для каждого указаны время старта и финиша (ч, мин, с). Требуется упорядочить список по убыванию результатов, найти время прохождения дистанции для 3-х лучших призеров.
8. Имеются сведения о товарах, находящихся на складе: наименование, изготовитель, объем партии, дата поступления на склад, стоимость единицы товара. Найти объем партии заданного вида товара.

3 Табулирование функций Построение графиков таблично заданных функций

Задача получения таблицы значений функций является почти классической задачей в программировании. Полученные значения функции могут служить исходными значениями для множества задач: отделение корней; поиска экстремальных значений функции; графического анализа функции. Функции могут быть представлены по-разному. В задачах данной главы функции представляются либо простыми формулами, либо в виде суммы бесконечного ряда. Бесконечные ряды нашли широкое применение в технических расчетах. В виде бесконечного ряда могут быть представлены как простые тригонометрические функции, так и функции сложной формы.

Если разрабатываемая программа содержит фрагмент табулирования функции, то ее интерфейсная часть должна, как минимум, предоставлять пользователю возможность задавать границы интервала, на котором необходимо выполнять табулирование функции, а также шаг табулирования, определяющий расстояние между отсчетами аргумента.

Примечание

Интерфейс программы не обязательно должен содержать элементы для задания пользователем этих значений. Но перед выполнением самого процесса табулирования необходимо получить такие значения или непосредственно – из соответствующих окон интерфейса, или косвенно – посредством обработки каких-то текущих значений переменных проекта.

Конечно же, программа должна предоставлять область, в которой будут отображаться значения аргументов и функции. Это могут быть элементы, отображающие значения в текстовом виде или в виде графиков. И, конечно, кнопки или другие управляющие элементы, обеспечивающие функциональность.

3.1. Общая постановка задачи и основные положения

Прежде чем приступить к разработке конкретных приложений рассмотрим общие положения свойственные для задач данного класса.

Общая постановка задачи

Математическая постановка задачи табулирования функции формулируется следующим образом. Задана функция $Y = F(X)$, определенная на пространстве R^1. Получить значения функции $y_i = F(x_i)$, вычисленные в точках абсциссы $x_0, x_1 \ldots x_n$, где $x_0 < x_1 < \ldots < x_n$. В общем случае значения абсцисс x_i формируют возрастающую последовательность. Но часто необходимо проводить табулирование, когда абсциссы располагаются равномерно с шагом h на интервале $x \in [x_0, x_n]$, то есть $h = x_{i+1} - x_i$ для любого i, $0 \le i \le n-1$.

Программная постановка задачи такого приложения выглядит так.

Требуется разработать функциональное приложение, в котором формируется массив значений функции $y_i = F(x_i)$ на множестве $x \in R^1$. Абсциссы $x_0, x_1 \ldots x_n$ бе-

рутся на заданном интервале $[c, d] \in R^1$ с равномерным шагом h: $x_i = c + i \cdot h$, где $i = 0 \ldots n$, где $n = (d - c) / h$.

Математический анализ решения задачи

Напомним, что набор равномерно отстоящих значений $x_i \in [c, d]$ классически формируют двумя способами:

$$\begin{array}{ll} \text{а)} & x_i = c + i \cdot h; \\ \text{б)} & x_{i+1} = x_i + h;\ x_0 = c. \end{array} \qquad (3.1)$$

Интервал табулирования $[c, d]$ в приложениях напрямую или косвенно определяется пользователем. Но в различных приложениях пользователем может задаваться или h, или n. А второй параметр определяться по формулам:

$$\begin{array}{ll} \text{а)} & n = (d - c) / h; \\ \text{б)} & h = (d - c) / n. \end{array} \qquad (3.2)$$

Здесь необходимо отметить, что эти простые формулы таят в себе некоторые вычислительные особенности. Рассмотрим каждый случай в отдельности.

1. Пусть заданы интервал $[c, d]$ и шаг h. Но по формуле (3.2, *а*) мы получим $\tilde{n}$, не равное целому числу. То есть в общем случае отрезок шириной h не обязательно укладывается в отрезок $[c, d]$ целое количество раз. Тогда приняв $n = [\tilde{n}]$ или $n = [\tilde{n}] + 1$, где $[\tilde{n}]$ целая часть от $\tilde{n}$, и организовав вычисления по $i = 0 \ldots n$ по формуле (3.1, *а*) получим $x_n < d$ или $x_n > d$.

   Если же вычислять абсциссы по формуле (3.1, *б*), а в качестве критерия останова задать условие $x_i \le d$, мы получим ту же картину: или x_n будет значительно меньше d, или значительно больше. Если условия задачи не позволяют вычислять функцию в точках $x_i > d$, то процесс вычисления, конечно же, останавливаем, если условие $x_i \le d$ нарушается. Но x_n может находиться от d почти на расстоянии h. Если же «дозволено залезть» за пределы интервала $[c, d]$, можно минимизировать отклонение x_n от d. Для этого в качестве критерия останова достаточно указать $x_i \le d + h/2$. Тогда конечная абсцисса x_n будет находиться в интервале $d - h/2 < x_i \le d + h/2$. На рис. 3.1 представлен пример, демонстрирующий возможное расположение точек на заданном интервале с заданным шагом.

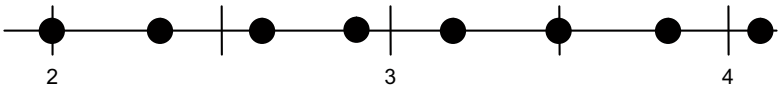


Рис. 3.1. Расположение точек на интервале [2; 4] с шагом 0,3

2. Пусть задан интервал $[c, d]$ и количество интервалов n. Математически мы получаем шаг h, который целое количество n раз укладывается в отрезок $[c, d]$. Тогда пользуясь формулами (3.1, *а*) или (3.1, *б*) получаем необходимый набор абсцисс x_i.

Примечание

Необходимо помнить, что операции с вещественными переменными выполняются с погрешностями, хотя и с высокой точностью. То есть математически и формула (3.1, *а*) и формула (3.1, *б*) дадут один и тот же набор значений x_i. При вычислениях же на компьютере мы получим два разных набора абсцисс, то есть $x_i = c + i \cdot h \ne x_i = x_{i-1} + h$. При использовании разных типов вещественных перемен-

ных *float* или *long double* эти различия между наборами абсцисс будут весьма различны.

Отклонение от точного математического значения, как правило, пренебрежительно мало, а в случае использования переменных типа `long double` тем более, и все же это способно повлиять на результаты вычислений:

а) если процесс формирования набора абсцисс организован в цикле с критерием $0 \le i \le n$, то будет сформирован набор из $(n + 1)$ значений x_i, каждое из которых близко к точному математическому;
б) если критерий останова $x_n \le d$, то набор абсцисс может содержать значение n или $(n + 1)$. То есть последний отсчет будет равен или $(d - \varepsilon) \le x_{n-1} \le (d + \varepsilon)$, или $(d - \varepsilon) \le x_n \le d$, где ε – некая очень малая величина. Важно здесь понимать, что если при вычислении x_n за счет погрешности вычислений окажется, что $d < x_n \le (d + \varepsilon)$, то цикл завершит работу, не вычисляя значение функции в точке x_n.

Стратегия программного решения задачи

Начинать решения следует с *анализа* задачи. На этом этапе следует определить, какие значения следует задавать пользователю в качестве исходных данных, и в каком виде должен быть представлен результат решения задачи. После этого прорисовывается интерфейс главной формы. Следует отметить, что все задачи, за исключением первой задачи из данной главы, выполнены в рамках одного проекта и на единственной форме. Необходимое пространство для отражения результатов решения обеспечено страницами одного блокнота, который растянут почти на все пространство формы. На соответствующую страницу выставляются визуальные компоненты и устанавливаются *свойства* этих компонент.

После чего можно переходить к задаче создания *обработчиков событий*, то есть к формированию текстов обработчиков событий. Рекомендуется действия обработчиков событий сводить, где это возможно, к вызову процедур, а сами процедуры оформлять либо методами класса `TForm`, либо в обособленном виде.

ПОРЯДОК ПОДГОТОВКИ КОДОВ ПРОГРАММ

Рекомендуется разрабатывать тексты, содержащие коды программы, в следующей последовательности:

- выделить подзадачи, которые следует оформить в виде методов, необходимых для решения задачи;
- определиться с *местоположением* этих процедур;
- разработать алгоритмы и подготовить тексты процедур;
- подготовить тексты обработчиков событий, содержащих вызовы методов;
- вписать тексты в соответствующие пространственные области редактора кода.

3.2. Табулирование функций, заданных аналитическим выражением

Рассмотрим несколько задач, связанных с табулированием функций.

Задача 3.1. Получить таблицу значений функции $Y = e^{-\frac{\pi}{3} \cdot x^2}$ на интервале [0,1; 0,4] с шагом 0,05.

Решение

1. Анализ задачи.

Алгоритм решения задачи включает в себя следующие шаги:

- по формуле (3.2, *а*) определяется *n* – количество точек на интервале с заданным шагом; здесь интервал [0,1; 0,4] содержит целочисленное количество интервалов длиной 0,05;
- в цикле, с пределами от 0 до *n* последовательно вычисляются значения (x_i , y_i).

Для вычисления x_i можно использовать формулу (3.1, *б*). Начальное значение x_0 до цикла следует положить равным *c*, здесь это 0,1. Тип данных для абсцисс и ординат лучше выбрать `double` или `long double`, так как работа с трансцендентными функциями требует большой точности.

2. Разработка интерфейса и задание свойств компонентов.

Для ввода исходных данных (*c*, *d*, *h*) воспользуемся тремя однострочными редакторами `Edit`, для отображения результатов – многострочным редактором `Memo` и компонентами `Label` для пояснений к окнам ввода исходных данных и вывода результатов. С целью отображения формулы в привычном математическом виде воспользуемся компонентом `Image` с меткой `Label` для пояснений.

Программа должна принять от пользователя, по крайней мере, две команды: произвести вычисления и прекратить выполнение программы. Для задания этих команд воспользуемся двумя кнопками `Button` (с именами `bbTable` и `bbExit`).

Все эти компоненты можно расположить на одной форме. Расположение элементов может быть таким, как на рис. 3.2.



Рис. 3.2. Интерфейс программы для задачи 3.1

3. Разработка программы.

Для того чтобы предоставить пользователю возможность выполнения действий в соответствии с описанным порядком и особенностей решения задачи редактирования, целесообразно реализовать следующие подпрограммы:

- `Double Func (double t);` – для вычисления значения функции в заданной точке `t`;

- `void DoTable ();` – для получения таблицы значений абсциссы и ординаты с размещением этих значений в `Memo`;
- `bbTableClick (TObject *Sender);` – обработчик события `OnClick` кнопки «Вычислить»;
- `bbExitClick (TObject *Sender);` – обработчик события `OnClick` кнопки «Выход».

4. Определение местоположения подпрограмм.

Прототипы функций `Func` и `DoTable` разумно описать в заголовочном файле `Unit1.h`, в интерфейсе класса `TForm1`, а в `Unit1.cpp` – дать описание реализации этих методов. Здесь же разместятся обработчики событий.

Вычисление значения функции в заданной точке

Функция `Func` возвращает результат вычисления функции $Y = e^{-\frac{\pi}{3}\cdot x^2}$ в точке `t`. Описание `Func`, предназначенной для вычисления значения функции в заданной точке, представлено в листинге 3.1.

Листинг 3.1. Метод класса *TForm* для вычисления значения функции в заданной точке

```
double TForm1::Func (double t)
{
  return (exp(-Pi/3*t*t));
}
```

Формирование таблицы значений функции $Y = e^{-\frac{\pi}{3}\cdot x^2}$
на заданном интервале с заданным шагом

Функция `DoTable` не имеет параметров. Она создает таблицу значений функции `Func` и размещает ее в `Memo1`. Из поля `Text` компонентов `Edit` читаются строки, которые преобразуются в вещественные числа и записываются в области памяти переменных `c`, `d`, `h`. Вводится две вспомогательные переменные `x` и `y` для хранения текущих значений координат. А также вводится переменная `Str` для хранения текущих координат. Алгоритм сводится к выполнению цикла.

До цикла:

- определяется количество точек `n`, для которых требуется получить табличные значения;
- инициализируется начальное значение координаты `x`.

В цикле (по числу точек):

- вычисляется `y` – текущее значение функции `Func`;
- `x` и `y` преобразуется в строку и помещаются в `Str`;
- `Str` – строка добавляется к `Memo`;
- изменяется текущее значение координаты `x` на величину шага `h`.

Описание процедуры формирования таблицы значений представлено в листинге 3.2.

Листинг 3.2. Описание метода класса `TForm`, предназначенного для формирования таблицы значений функции *F(x)*

```
void TForm1:: DoTable ()
{
```

```
  double c = StrToFloat (Edit1->Text);
  double d = StrToFloat (Edit2->Text);
  double h = StrToFloat (Edit3->Text);

  Memo1->Lines->Clear();// Очистка Memo1

      // Определяется количество точек на интервале
  int n = (d-c)/h + 1;

  double y;
  double x = c; // Задается начальное значение x
  for (int i = 0; i<n; i++)// Делать цикл
   {
    y = Func (x); // Вычислить значение функции
    AnsiString Str; // Вспомогательная переменная
         // Записать в Str значения x и y
    Str = FloatToStrF (x, ffFixed, 5, 2) + " " +
                          FloatToStrF (y, ffFixed, 7, 4);

    Memo1->Lines->Add (Str); // Добавить к Memo1
    x = x+h;//Увеличить значение x на величину шага h
   }
}
```

Обработчик события `OnClick` команды «Вычислить»

Содержит всего одну команду – вызвать процедуру `DoTable`, предназначенную для построения таблицы значений. Описание представлено в листинге 3.3.

Листинг 3.3. Описание действий под кнопку «Вычислить»

```
void __fastcall TForm1::bbTableClick(TObject *Sender)
{
    DoTable();
}
```

Задача 3.2. Требуется построить графики функций $F(x) = 2 \cdot x^3 + 4 \cdot x - 7$ и ее производной $F'(x)$, которые должны быть представлены своими значениями на интервале $[c, d]$ с заданным шагом h.

Решение

1. Предварительные рассуждения.

Как и в общем случае, то есть при решении практически любой задачи, в соответствии с технологией нисходящего проектирования необходимо постараться разделить задачу на независимые части, которые были бы компактны и в меру универсальны, и которые бы при этом хорошо взаимодействовали между собой.

Задача 3.2 похожа на задачу 3.1, но и имеет отличия. Здесь требуется:

- получить таблицу значений не одной функции, а двух – собственно для функции $F(x)$ и для ее производной $F'(x)$;
- помещать таблицы значений в `Memo` не требуется;
- построить графики функций с использованием компонента `Chart`.

Естественно, как и в задаче 3.1, будет логично для вычисления функций $F(x)$ и $F'(x)$ разработать отдельные подпрограммы.

В зависимости от проекта такие подпрограммы могут быть *обычными* функциями либо *методами класса*, либо *членами структуры*. При этом есть два принципиально разных способа вызова: непосредственный вызов такой подпрограммы или такая подпрограмма передается через входные параметры в другую подпрограмму. Конечно, та или иная реализация имеет свои преимущества и недостатки.

Примечание

- *Обычная функция*.

Вариант удобен для вычисления простых математических функций с фиксированными параметрами, заданными в виде констант. *Преимущества*: может вызываться другими функциями непосредственно или передаваться им как параметр. *Недостатки*: задавать дополнительные параметры для работы такой подпрограммы необходимо или через глобальные переменные, или через дополнительные параметры. Второй способ делает заголовок такой функции уникальным, так как другая функция может регулироваться другими параметрами. А использование глобальных переменных всегда нежелательно.

- *Метод класса*.

Такой вариант требуется, если при вычислении функции используются некоторые поля класса, то есть отдельные значения некоторых коэффициентов хранятся в полях класса. Тогда, естественно, что такая подпрограмма должна быть членом класса, чтобы иметь доступ к полям класса, тем более, если поля окажутся в разделах `private` или `protected` класса; доступ к таким элементам извне, как известно, невозможен. Да и если подпрограмма зависит от полей класса, логично было бы, чтобы эта подпрограмма сама была членом класса. *Преимущества*: может быть функцией, которая очень тонко настраивается полями класса; может вызываться другими функциями непосредственно или передаваться им как параметр. *Недостатки*: при расширении набора требуемых функций необходимо корректировать описание класса или формировать наследника, который имеет дополнительные методы с таким же заголовком.

- *Функция – член структуры*.

По шкале удобства, наверное, такую технологию можно расположить между двумя первыми. Такой способ организации вычисления значений функции это организация специальной структуры содержащей поля, значения которых участвуют как параметры при вычислении значения функции, и подпрограмму вычисления функции. То есть вычислением такой функции легко управлять – изменить значения полей структуры, если требуется, а затем вызывать подпрограмму – член структуры, для вычисления значений функции при этих параметрах. *Преимущества*: может быть функцией, которая очень тонко настраивается полями структуры; может вызываться другими функциями непосредственно или передаваться им как параметр. *Недостатки*: перед использованием такой подпрограммы необходимо настраивать элементы структуры, то есть производить копирование данных уже хранящихся в других переменных.

Конечно, можно использовать не только такие технологии, но и комбинации из них, например, массивы указателей на функции или «встраивать» одни функции в тело других, чтобы иметь функцию с требуемым заголовком, например, с единственным вещественным параметром – значением абсциссы, для которого требуется

вычислить значение. А такая «функция-обертка» выполняет необходимые подготовительные вычисления и выполняет вызов функции с уникальным заголовком или открытый метод класса и так далее.

Примечание

Способ вызова подпрограммы также влияет на гибкость программы. Когда некоторая *подпрограмма_1* передается как входной параметр в *подпрограмму_2*, повышается гибкость программы, так как в этом случае непосредственно при вызове *подпрограммы_2* можно указывать, а какая подпрограмма вида *подпрограммы_1* должна использоваться в ходе работы *подпрограммы_2*. То есть в программе можно определить несколько подпрограмм с единообразным заголовком типа_1 и указатель на функцию с таким же заголовком. А *подпрограмма_2*, использующая другую подпрограмму, должна содержать в качестве формального параметра такой указатель на подпрограмму типа_1, и тело содержать вызов подпрограммы типа_1.

2. Анализ задачи.

В общем случае вычисленные значения функции (таблица значений) могут подвергаться различной дальнейшей обработке. В данном проекте требуется их отобразить в текстовом виде на компоненте *Memo* и в виде графика на компоненте *Chart*. Если функция вычисляется сложно и каждое вычисление требует больших затрат, например, времени и памяти, то, возможно, будет легче настроить параметры вычисления, диапазон и шаг табулирования и произвести однократное вычисление значений функции, и сохранить их в оперативной памяти, например, в динамическом массиве. А затем использовать эти элементы массива для дальнейших расчетов.

Примечание

Проектируя максимальную независимость процедуры построения графика функции от конкретного компонента `Chart`, можно провести следующие рассуждения. Компонент `Chart` отображает графики на компонентах `ChartSeries`, составляющих массив, и являются частью компонента `Chart`. Тогда логично разрабатывать подпрограмму построения графика так, чтобы в качестве параметра передавать указатель на компонент `ChartSeries` и параметры, определяющие порядок построения графика. Такая структура заголовка подпрограммы позволяет воспользоваться такой подпрограммой как для построения графиков на одном и том же компоненте `Chart`, так и на нескольких. Это возможно потому, что подпрограмма имеет дело не с компонентом `Chart`, а всего лишь с компонентом `ChartSeries`.

3. Разработка интерфейса и задание свойств компонентов.

В соответствии с анализом задачи на главной форме программы целесообразно разместить исходные данные, то есть аналитические выражения для функций (компоненты `Image`), начало и конец интервала табулирования, а также шаг табулирования. Для этого можно воспользоваться компонентами `LabeledEdit` с именами `le_3_2_C`, `le_3_2_D` и `le_3_2_H`, соответственно. А для визуализации графиков потребуется один компонент `Chart` с именем `ChartSeries`. Предлагаемый интерфейс приложения для задачи 3.2 может иметь вид, как на рис. 3.3.

Согласно рисунку все пространство главной формы поделено на две неравные части. Большую часть занимает блокнот `PageControl`. С каждой страницей блокно-

та будет в дальнейшем связываться определенная задача. Справа на форме на подготовленной панели (компонент `Panel`) расположены кнопки (`Button`) для решения. Для удобства пользователя, чтобы «не заблудиться», страницы и кнопки, обеспечивающие выполнение соответствующей задачи, имеют одинаковые наименования.

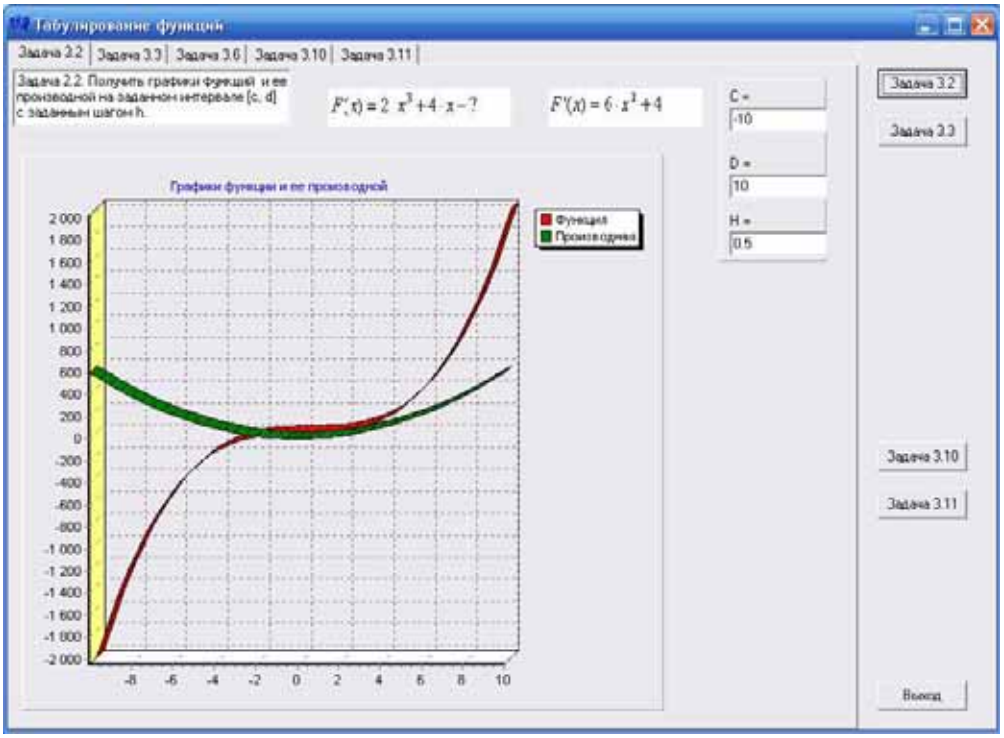


Рис. 3.3. Представление задачи на странице «Задача 3.2»

4. Разработка списка процедур.

Для решения этой задачи целесообразно разработать набор из шести подпрограмм:

- `double Func (double aX);` – подпрограмма вычисления значений функции. Имеет один входной параметр – абсцисса, а через заголовок функции будет возвращаться посчитанная ордината;
- `double DeriveFunc (double aX);` – подпрограмма вычисления значений производной от функции. Имеет один входной параметр – абсцисса, а через заголовок функции будет возвращаться посчитанная ордината;
- `int DoTable_3_2 (double (*aFun)(double), double aC, double aD, double aH, double *&aY);` – подпрограмма для выполнения табулирования функции `aFun` на интервале `[aC,aD]` с шагом `aH`. Набор полученных ординат помещается в массив `aY`;
- `void DoPlot (TChartSeries *aS, TColor aCol, double *aArr, int aN, double aC, double aH, bool aClear);` – подпрограмма для построения графика функции. Входные параметры: `aS` – указатель на график `ChartSeries`, `aCol` – цвет графика, `aArr` – массив ординат и `aN` – его длина, `aC` – начало интервала и `aH` – шаг табулирования, `aClear` – признак, показывающий, требуется ли «очищать» график;
- `void DoTask_3_2();` – подпрограмма решения задачи 3.2. Эта подпрограмма из интерфейса проекта получает интервал и шаг табулирования, вызывает подпро-

граммы `Func` и `DeriveFunc` для вычислений массивов ординат, вызывает метод `DoPlot` для построения графиков, освобождает память, отведенную для массивов ординат;

- `Button_3_2Click (TObject *Sender);` – обработчик события `OnClick` кнопки `Button_3_2`, который запускает метод `DoTask_3_2` и открывает страницу с решением и графиками для задачи 3.2.

Примечание

Расчет количества отсчетов и процесс расчета ординат функции и производной логично сконцентрировать в одной подпрограмме – в теле метода `DoTable_3_2`, представленного в листинге 3.6. Для этого список формальных параметров включает в себя значения интервала и шага табулирования *c*, *d*, *h*. Следовательно, отведение памяти под массив ординат *aArr* и, как следствие этого, формирование адреса массива будут выполнены внутри тела этой процедуры. Тогда, чтобы функция возвращала таблицу вычисленных значений в виде массива, необходимо также передавать по ссылке указатель на массив.

5. Определение местоположения процедур.

Как это следует из вышеописанного, все процедуры, кроме двух, которые предназначены для вычисления значения функции, и ее производной в заданной точке следует разместить в классе формы: добавить прототипы в *h*-файл, а описания – в *cpp*-файл. А две функции `Func`, `DeriveFunc` описать *независимо* от класса формы и разместить их в *cpp*-файле.

Вычисление значения функции в заданной точке

Функция `Func` возвращает значение функции $F(x) = 2 \cdot x^3 + 4 \cdot x - 7$ в точке `aX`. Описание функции представлено в листинге 3.4.

Листинг 3.4. Вычисление функции $F(x) = 2 \cdot x^3 + 4 \cdot x - 7$ в заданной точке

```
double Func (double aX)
// aX – координата точки
{
  return (2 * pow (aX, 3) + 4 * aX - 7);  // результат
}
```

Вычисление значения производной в заданной точке

Функция `DeriveFunc` возвращает значение производной от функции $F(x) = 2 \cdot x^3 + 4 \cdot x - 7$ в точке `aX`. Описание функции представлено в листинге 3.5.

Листинг 3.5. Вычисление производной от функции $F'(x) = 6 \cdot x^2 + 4$ в заданной точке

```
double DeriveFunc (double aX)
// aX – координата точки
{
  return (6 * pow (aX, 2) + 4);// результат
}
```

Вычисление таблицы значений функции
на заданном интервале с заданным шагом

Функция `DoTable_3_2`, предназначенная для вычисления таблицы значений функции, имеет 4 входных параметра и два выходных. С помощью входных пара-

метров передаются: aFun – указатель на функцию для вычисления значений; aC, aD – границы интервала табулирования; aH – шаг табулирования; aY – выходной массив, представляющий таблицу значений (память под массив отводится внутри программы, поэтому указатель на массив передается по ссылке). Выходным значением – результатом работы алгоритма – является также количество вычисленных значений функции на заданном интервале. Это значение передается через заголовок функции.

Вычисление сводится к одному циклу, внутри которого последовательно вычисляются координаты точки на заданном интервале и значение функции aFun в этой точке. Число повторений цикла определяется количеством точек. Эта величина определяется до входа в цикл. А вычисление координат точки производится по рекуррентной формуле от начального значения aC с шагом aH. Описание функции DoTable_3_2 приведено в листинге 3.6.

Листинг 3.6. Описание метода, предназначенного для вычисления таблицы значений функции aFun на интервале с шагом

```
int  __fastcall TfoMain::DoTable_3_2 (double (*aFun)(double), double aC,
                                      double aD, double aH, double *&aY)
// x - вспомогательная переменная для хранения текущей
//  координаты очередной точки
// n - количество точек
  {
    int n = (aD - aC) / aH + 1; // Подсчет числа точек
    aY = new double [n]; // Отведение памяти под массив
    double x = aC; // Установка начального значения
    for(int i = 0; i<n; i++) // Делать цикл
      {
        aY [i] = aFun (x);// Вычислить значение
        x += aH; // Изменить шаг
      }
    return n; // Вернуть размер
  }
```

Построение графика функции, заданной массивом значений

В методе DoPlot, предназначенном для построения графиков функции, используется визуальный компонент TChart системы программирования *C++ Builder*.

КОМПОНЕНТ Chart

Компонент Chart является панелью, на которой можно создавать диаграммы и графики различных типов. Свойства Chart удобно устанавливать специальным *Редактором диаграмм*, вызываемым из *Инспектора объектов* или двойным щелчком на компоненте Chart.

Компонент является контейнером объектов Series типа TChartSeries – серий данных, характеризующихся различными стилями отображения. Для задания значений, отображаемых на графике, используются различные методы серий Series, среди которых можно отметить AddXY.

МЕТОД AddXY

Метод AddXY позволяет добавить новую точку (с координатами X и Y) в график функции. Прототип этой функции имеет следующий вид:

```
int AddXY (const double aX, const double aY,
                    const AnsiString aStr, TColor aClr);
```

Здесь параметры:
`aX`, `aY` – значения абсциссы и ординаты, соответственно;
`aStr` – текст подписи к этой отметке;
`aClr` – цвет точки на графике.

Всего метод `DoPlot` имеет 7 входных параметров: `aS` – ссылка на `TChartSeries` из компонента `TChart`; `aArr` – ссылка на массив; `aCol` – цвет графика; `aN` – количество значений в массиве; `aC`, `aH` – начало интервала и шаг табулирования; `aClear` – признак очистить/не очистить график предварительно.

Алгоритм формирования графика сводится к одному циклу, внутри которого последовательно вычисляются координаты точек, которые добавляются к графику с помощью метода `AddXY`. Причем координата `x` вычисляется внутри цикла по рекуррентной формуле от начального значения `aC` с шагом `aH`. А вторая координата `y` передается из входного массива `aArr` в соответствии с ее номером. Описание метода построения графиков функций приведено в листинге 3.7.

Листинг 3.7. Описание метода построения графиков функций

```
void _fastcall TfoMain::DoPlot (TChartSeries *aS,
                  TColor aCol, double *aArr, int aN, double aC,
                                          double aH, bool aClear)
{
  // функция вычерчивания графика для вещественного массива

  double x = aC;// Установить начальное значение
  if (aClear) aS->Clear (); // Очистить график
  for (int i = 0; i<aN; i++) // Делать цикл
   {
              // Добавить точку к графику
    aS->AddXY (x, aArr [i],"", aCol);
    x += aH; // Изменить шаг
   }
}
```

Описание метода решения задачи 3.2

Функция `DoTask_3_2` не имеет параметров. Алгоритм сводится к следующим шагам:

1. Читаются из строк редакторов `le_3_2_C`, `le_3_2_D`, `le_3_2_H` входные значения `c`, `d`, `h`.
2. При чтении проверяется корректность записи значений:
   а) если преобразование одного из текстов из окон редактирования в число привело к ошибке, то
      - выдается сообщение;
      - прекращается вычисление;
   б) иначе –
      - вычисляются таблицы значений;
      - строятся графики.

Описание функции `DoTask_3_2` приведено в листинге 3.8.

Листинг 3.8. Описание метода решения задачи

```
void __fastcall TfoMain::DoTask_3_2()
{
  // указатели на массивы для значений функции и производной
  double *y, *y1;
  double c, d, h;  // интервал табулирования и шаг
         // если преобразование ошибочно
  if (!(TryStrToFloat (le_3_2_C->Text, c)
        && TryStrToFloat (le_3_2_D->Text, d)
        && TryStrToFloat (le_3_2_H->Text, h)))
    {
           // то сообщение об ошибке
     MessageDlg ("Одно из окон содержит не число!",
                       mtError, TMsgDlgButtons() << mbOK, 0);
     return; // и выход из метода
    }
       // если значения пределов или шага не соответствующие
  if (   c >= d
      || h <= 0.0
      || h > d - c)
    {      // то сообщение об ошибке
     MessageDlg ("Неправильное соотношение
                            пределов интервала и шага!",
                   mtError, TMsgDlgButtons() << mbOK, 0);
     return; // и выход из метода
    }
           // расчет значений функции и количества точек
  int n = DoTable_3_2 (Fun c, c, d, h, y);
           // расчет точек производной функции
  DoTable_3_2 (DeriveFunc, c, d, h, y1);
           // отображение графика функции
  DoPlot (Chart_3_2->Series [0], clRed, y, n, c, h);
           // отображение графика производной функции
  DoPlot (Chart_3_2->Series [1], clGreen, y1, n, c, h);
  delete[] y, y1; // высвобождение памяти
}
```

Обработчик события `OnClick` кнопки `Button_3_2` – «Решить задачу»

Алгоритм содержит всего два действия: вызов метода решения задачи; сделать активной страницу блокнота, на котором расположено решение. Код обработчика события `OnClick` кнопки `Button_3_2` – «Решить задачу» приведен в листинге 3.9.

Листинг 3.9. Описание обработчика события `OnClick` кнопки `Button_3_2` – «Решить задачу»

```
void __fastcall TfoMain::Button_3_2Click(TObject *Sender)
// инициализация решения задачи 3.2
{
  DoTask_3_2 ();// вызвать метод решения задачи
           // активизировать страницу блокнота
  PageControl->ActivePage = TabSheet_3_2;
}
```

Задача 3.3. Известно, что функция $Y = F(x)$ является унимодальной на заданном интервале. Найти $\max\limits_{x\in[t1,t2]} Y = D\cdot\sin(A\cdot x^B + C)$ с погрешностью **ε** методом сканирования. Коэффициенты *A*, *B*, *D*, *C* имеют заданные значения.

Решение

1. Анализ задачи.

Чтобы правильно спроектировать данное приложение необходимо напомнить математические определения некоторых терминов.

УНИМОДАЛЬНАЯ ФУНКЦИЯ

Унимодальная функция – это функция, имеющая только единственный экстремум в области допустимых значений.

Метод сканирования заключается в следующем.

1. Вычисляется таблица значений функции на исходном интервале с крупным шагом. Находится отрезок, в котором функция принимает наибольшее значение. Этот отрезок выделен цветом на рис. 3.4.
2. Затем этот отрезок табулируется с более мелким шагом. Определяется новый отрезок, внутри которого находится уточненное значение максимума.
3. Процедура повторяется до тех пор, пока величина отрезка, содержащего максимальное значение, не будет меньше заданной погрешности **ε**.

x	*y*
0,6	0,57
0,9	0,68
1,2	0,71
1,5	0,49
1,8	0,38
2,1	0,34
2,4	0,29
2,7	0,28
3	0,25
3,3	0,2
3,6	0,31
3,9	0,43
4,2	0,46

Наименьшее значение – 0,2
Отрезок, содержащий это наименьшее значение – [3; 3,6]

Рис. 3.4. Таблица значений функции $Y = D\cdot\sin(A\cdot x^B + C)$. Цветом выделено наименьшее значение и выбранный интервал, содержащий найденное минимальное значение

Из описания метода следует, что производится многократное повторение решения задачи поиска отрезка, внутри которого находится уточненное значение максимума. Останов происходит, когда отрезок достаточно мал. Эта задача чуть сложнее, чем прежняя 3.2, но может также решаться одним методом – `DoTask_3_3`. Эта подпрограмма будет вводить значения параметров функции, интервал поиска максимума и количество интервалов сканирования на каждом из этапов.

Но такой метод сканирования, в отличие от `DoTask_3_2`, вызывает методы табулирования и построения графика несколько раз, пока не будет достигнута задан-

ная точность **ε**. На каждой итерации новый интервал будет разбиваться на заданное пользователем количество интервалов табулирования, определяться шаг табулирования и выполняться табулирование функции. В сформированном массиве значений будет определяться текущий максимум и новый подходящий интервал, а также осуществляться переход на новую итерацию.

Примечание

Метод табулирования `DoTable_3_3` для этой задачи подобен соответствующему методу для задачи 3.2. Но в данном случае удобнее модифицировать метод так, чтобы в качестве входного параметра задавалось количество интервалов табулирования *N*, а через заголовок возвращался шаг табулирования *H*, посчитанный в ходе работы метода. Этот шаг потом используется методом `DoTask_3_3` для организации новых итераций и проверки на окончание процесса сканирования.

Другая особенность этой задачи в том, что на каждой итерации количество точек табулирования одинаковое. Тогда метод `DoTable_2_3` следует разработать так, что он будет выделять память под этот массив только на первой итерации; на последующих итерациях – использовать уже отведенную память. Это можно реализовать следующим образом: указатель на массив ординат на старте метода `DoTask_2_3` инициализировать значением NULL, а в методе `DoTable_2_3` выделение памяти осуществлять только тогда, когда этот указатель равен NULL. Это обеспечит однократное выделение памяти достаточного объема на старте очередного решения данной задачи. И, конечно, в конце метод `DoTask_3_3` должен освободить эту память.

Для построения графика в данной задаче удобно воспользоваться особенностью метода `AddXY`, который позволяет подавать точки в любом порядке. Этот метод сам упорядочивает точки по возрастанию абсцисс. Тогда, если выполнить очистку серии точек перед первым построением графика, а на последующих итерациях только пополнять новыми отсчетами, то после завершения расчетов получим следующую картину. График функции будет отображен на всем исходном интервале. При этом крайние участки графика будут отображаться ломаной линией с крупным шагом, а участки, примыкающие к точке максимума, с уменьшающимся шагом. Такой график будет содержать все посчитанные точки и адекватно отображать весь процесс сканирования функции с постепенным уменьшением шага сканирования.

Примечание

Недостаток изложенного метода, как и многих численных методов, заключается в возможности пропуска «острого» локального максимума.

По условию задачи функция $Y = F(x)$ этой задачи зависит от параметров *A*, *B*, *D*, *C*.

С одной стороны, подпрограмма вычисления функции *может получать* значение абсциссы *x* и значения параметров через список формальных параметров. Но тогда подпрограммы построения графика и решения задачи, использующие подпрограмму вычисления ординат, должны быть специфичными – они должны передавать и абсциссу, и текущие значения параметров. С другой стороны, *большое количество параметров делает вызов подпрограммы более медленным*. Следует заметить также, что за время решения данной задачи параметры функции *A*, *B*, *D*, *C* не изменя-

ются. Эти два обстоятельства позволяют предположить, что в данном случае целесообразно передавать параметры через *глобальные переменные* или *поля класса*.

Примечание

Так как подпрограмма вычисления функции не является методом класса, то доступ к полям класса затруднителен. Поэтому предлагается в заголовочном файле *Unit1.h* описать структуру `TParams`, поля которой являются параметрами функции, и глобальную переменную `gParam_3_3` этого типа. Описание структуры представлено в листинге 3.10.

Листинг 3.10. Описание структуры для параметров и глобальной переменной для задачи 3.3

```
struct TParams
{
  double A, B, C, D; //параметры
} gParam_3_3;
```

2. Разработка интерфейса и задание свойств компонентов.

Интерфейс решения задачи 3.3 представлен на рис. 3.5. Согласно рисунку он похож на интерфейс задачи 3.2. Различие состоит только в числе параметров, выставленных на форму: добавлены компоненты `LabeledEdit` с именами `le_3_3_A`, `le_3_3_B`, `le_3_3_C`, `le_3_3_D`, `le_3_3_Eps` – для ввода и отражения данных: `A`, `B`, `C`, `D`, `eps`, а также `le_3_3_AT1` и `le_3_3_T2`, `le_3_3_N` – для отражения результатов вычисляемых значений `t1`, `t2` и `N`. Свойства остальных компонент очевидны.

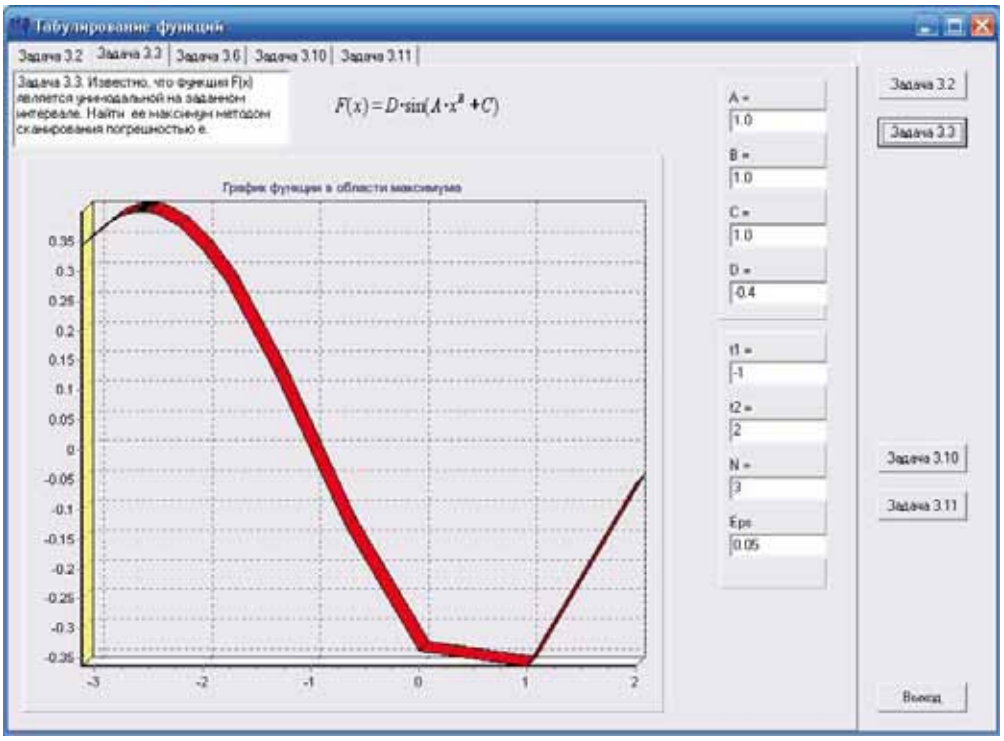


Рис. 3.5. Представление задачи на странице «Задача 3.3»

3. Разработка программы.

Для решения задачи в соответствии с интерфейсом, представленным на рис. 3.5, целесообразно разработать набор из шести подпрограмм:

- `double FuncScan (double aX);` – подпрограмма вычисления значений функции. Имеет один входной параметр – абсцисса, а через заголовок функции будет возвращаться посчитанная ордината;
- `double DoTable_3_3 (double (*aFun)(double), double aC, double aD, int aN, double *&aY);` – подпрограмма вычисления таблицы значений. Входные параметры: `aFun` – имя функции; `aC, aD` - границы интервала; `aN` - число точек на интервале. Выходной параметр: `aY` - массив значений;
- `void DoTask_3_3 ();` – метод решения задачи;
- `void Button_3_3Click(TObject *Sender);` - обработчик события под кнопку «Решить задачу».

Как и для задачи 3.2, процедуры, представленные в разделе анализа, включены в методы класса за исключением процедуры, предназначенной для вычисления значения функции $F(x) = D \cdot \sin(A \cdot x^B + C)$ в заданной точке. Эта процедура описана независимо от класса формы.

Вычисление значения функции в заданной точке

Функция `FuncScan`, предназначенная для вычисления $F(x) = D \cdot \sin(A \cdot x^B + C)$, в заданной точке имеет один входной параметр `aX` – абсциссу задаваемой точки. Параметры `A`, `B`, `C`, `D` берутся из структуры `gParam_3_3`. Описание функции `FuncScan` приведено в листинге 3.11.

Листинг 3.11. Вычисление функции $F(x) = D \cdot \sin(A \cdot x^B + C)$ в заданной точке

```
double FuncScan (double aX)
{
  return gParam_3_3.D * (gParam_3_3.A * sin (pow (aX,
                                gParam_3_3.B) + gParam_3_3.C));
}
```

Вычисление таблицы значений на заданном интервале

Процедура `DoTable_3_3` имеет пять параметров. Входные параметры: `aFun` – имя функции; `aC`, `aD` – границы интервала; `aN` - число точек на интервале. Выходной параметр: `aY` – массив значений. Введена локальная переменная `x` – текущее значение координаты точки. В программе высчитываются значения функции для точек, количество которых на одну больше чем `aN`. Алгоритм этой процедуры состоит из следующей последовательности действий.

1. Определяется величина шага табулирования.
2. Отводится память под выходной массив (поэтому указатель передается по ссылке).
3. Инициализируется текущее значение координаты.
4. В цикле (по числу точек):
   а) вычисляется очередное значение функции;
   б) изменяется текущее значение координаты на величину шага.

Описание процедуры `DoTable_3_3` представлено в листинге 3.12.

Листинг 3.12. Вычисление таблицы значений функции

```
double __fastcall TfoMain::DoTable_3_3 (double (*aFun)(double),
                           double aC, double aD, int aN, double *&aY)
```

```
{
    double h = (aD - aC) / aN;  // величина шага

            // память отводится только если указатель NULL
    if (aY == NULL) aY = new double [aN + 1];

    double x = aC;  // установка начального значения
    for(int i = 0; i <= aN; i++)  // делать цикл
      {
        aY [i] = aFun (x);  // значение в очередной точке
        x += h;  // увеличение координаты на величину шага
      }
    return h;  // вернуть шаг
  }
```

Решение задачи 3.3

Процедура не имеет параметров. Внутренние вспомогательные переменные: `y` – указатель на массив, предназначенный для хранения значений функции; `t1`, `t2` – интервал табулирования; `h` – шаг; `eps` – погрешность; `n` – количество интервалов сканирования; `needClear` – параметр очистки.

Последовательность действий.

1. Исходные данные читаются из соответствующих окон однострочных редакторов с проверкой на ошибку преобразования данных.
2. Если при чтении возникла ошибочная ситуация, то необходимо выполнить следующее:
   а) сообщить об ошибке;
   б) выйти.
3. Если ошибок нет, то необходимо выполнить следующее:
   а) читается с проверкой на ошибку значение, задающее количество точек и условие на ограничение (количество точек не должно быть меньше 3 и больше 20);
   б) если при чтении возникла ошибочная ситуация, то сообщить об ошибке и выйти;
   в) если ошибок нет, то выполняются следующие действия:
      - проверяются границы интервала и точность на корректность;
      - если параметры некорректны, то сообщить об ошибке и выйти.
4. Инициализируется параметр очистки.
5. В цикле (пока величина шага превышает точность)
   а) производится расчет значений функции и шага сканирования; строится график функции;
   б) сменяется значение признака очистки;
   в) выполняется цикл с целью поиска наибольшего значения и его номера;
   г) устанавливаются новые значения границ табулирования (это две точки – левее и правее максимума).

Код реализации процедуры приведен в листинге 3.13.

Листинг 3.13. Описание метода решения задачи

```
void __fastcall TfoMain::DoTask_3_3 ()
{
  double *y = NULL;  // начальное значение указателя
```

```
  double t1, t2, h, eps;
  int n;
// если преобразование одного из текстов из
// окон редактирования в число привело к ошибке,
// то сообщаем и завершаем программу
  if (!(TryStrToFloat (le_3_3_A->Text, gParam_3_3.A)
        && TryStrToFloat (le_3_3_B->Text, gParam_3_3.B)
        && TryStrToFloat (le_3_3_C->Text, gParam_3_3.C)
        && TryStrToFloat (le_3_3_D->Text, gParam_3_3.D)
        && TryStrToFloat (le_3_3_T1->Text, t1)
        && TryStrToFloat (le_3_3_T2->Text, t2)
        && TryStrToFloat (le_3_3_Eps->Text, eps)))
    {
     MessageDlg ("Одно из окон данных содержит не число!",
                          mtError, TMsgDlgButtons() << mbOK, 0);
     return; // выход
    }
// если преобразование одного из текстов из
// окон редактирования в число привело к ошибке,
// то сообщаем и завершаем программу
  if (!(TryStrToInt (le_3_3_N->Text, n)
        && (n >= 3 || n <= 20)))
    {
     MessageDlg ("Окно N содержит не целое число!
                           Или оно не в диапазоне 3<=N<=20",
                     mtError, TMsgDlgButtons() << mbOK, 0);
     return; // выход
    }
// если значения пределов или шага не соответствующие,
// то завершение программы
  if ( t1 >= t2 || eps < 1e-5 )
    {
     MessageDlg ("Неправильное соотношение пределов
                             интервала и погрешность мала!",
                     mtError, TMsgDlgButtons() << mbOK, 0);
     return; // выход
    }
  bool needClear = true; // инициализация параметра очистки
  do // делать цикл
   {
            // расчет значений функции и шага сканирования
    h = DoTable_3_3 (FuncScan, t1, t2, n, y);
            // отображение графика функции
    DoPlot (Chart_3_3->Series [0], clRed, y, n + 1, t1, h, needClear);
    needClear = false; // смена значения признака очистки

            // поиск максимального из полученных значений
    int    imax = 0;
    double max = y [imax];
    int i;
    for (i = 1; i < n; i++)
      if (max < y [i])
       { max = y [i];
        imax = i;
       }

// начальной границей интервала  становится точка,
// расположенная перед максимальной
```

```
    t1 = t1 + (imax - 1) * h;
// конечной становится точка после максимальной
    t2 = t1 + 2 * h;
   } while (2 * h > eps);// пока величина превышает точность

  delete[] y;// высвободить память
}
```

Обработчик события `OnClick` кнопки `Button_3_3` – «Решить задачу»

Алгоритм содержит всего два действия: вызов метода решения задачи; сделать активной страницу блокнота, на котором расположено решение. Код обработчика события `OnClick` кнопки `Button_3_3` – «Решить задачу» приведен в листинге 3.14.

Листинг 3.14. Описание обработчика события `OnClick` кнопки `Button_3_3` – «Решить задачу»

```
void __fastcall TfoMain::Button_3_3Click(TObject *Sender)
// инициализация решения задачи 3.3
// активизировать страницу
{
  DoTask_3_3 ();
  PageControl->ActivePage = TabSheet_3_3;
}
```

3.3. Табулирование функций, заданных в виде суммы бесконечного ряда

Как известно, не всегда функции задаются аналитическим выражением. Иногда они задаются в виде ряда. Значения таких функций могут быть вычислены на заданном интервале с различной точностью.

Задача 3.4. Известно, что функция арктангенс гиперболический может быть представлена в виде суммы бесконечного ряда $\mathrm{Arth}(x) = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \ldots + \frac{x^{2n+1}}{2n+1} + \ldots,$ для $|x| < 1$. Требуется:

1. Построить график частичных сумм членов ряда в зависимости от устанавливаемой точности ε.
2. Построить график зависимости количества членов ряда в зависимости от устанавливаемой точности ε.

Решение

1. Анализ задачи.

Данная задача имеет одно, но очень существенное отличие от предыдущих задач. Если в предыдущих случаях функции были представлены аналитическими выражениями, то в данном случае функция представлена в виде суммы бесконечного числового ряда.

Вычисление математических функций не всегда возможно путем простых вычислений с использованием операций сложения, умножения и тому подобное. Процесс вычисления многих функций достаточно сложен и поэтому значения таких функций закладываются в таблицы[1]. Это относится даже к таким, на первый взгляд,

[1] Например, такие широко известные таблицы, как таблицы Брадиса.

элементарным функциям, как тригонометрические: *sin*, *cos* и тому подобное. На современном этапе развития вычислительной техники процесс вычисления многих таких функций заложен в математические пакеты или даже реализован на аппаратном уровне – на уровне команд процессора и в калькуляторы.

Одним из стандартных математических приемов вычисления сложных функций является разложение функции в ряд, например ряд Тейлора, и вычисление суммы членов такого ряда с заданной точностью. Разложение в ряд производится в окрестности некоторой точки. И, пользуясь правилом формирования очередного члена ряда, можно вычислять сумму ряда, и значение суммы принять за значение функции.

Из математического анализа известно, что сходимость бесконечного ряда определяется величиной его остатка. Остаток представляет собой разность между суммой всего ряда и суммой первых его членов. Поэтому остаток ряда по абсолютной величине показывает, какую ошибку мы допускаем, когда заменяем сумму бесконечного ряда суммой конечного количества первых его членов.

ВЫЧИСЛЕНИЕ СУММЫ РЯДА

Из теорем математического анализа известно, что сумма отброшенных членов сходящегося знакопеременного ряда не будет превышать величину последнего оставленного члена ряда. То есть по первому отброшенному члену ряда можно *оценить* остаток. Для ряда с положительными членами такой оценки нет, но, останавливая вычисления на некотором члене, малом по величине, считается, что остаточная сумма также достаточно мала. Поэтому процесс вычисления суммы можно продолжать сколь угодно долго, пока члены ряда по величине не станут меньше заданного ε. Как только очередной член станет меньше заданного ε процесс суммирования можно прекращать.

2. Разработка интерфейса и задание свойств компонентов.

Интерфейс программы представлен на рис. 3.6. Он реализован с помощью того же набора компонентов, что и в предыдущих задачах. Исключением являются два компонента `RadioButton`, которые играют роль переключателей, меняющих вид графика:

а) «Сумма ряда» – отражает зависимость суммы ряда от точности вычислений.
б) «Количество членов» – показывает, как меняется количество слагаемых в зависимости от изменения точности.

Свойства очевидны и следуют из интерфейса. В соответствии с интерфейсом пользователю предлагается следующий порядок решения задачи:

а) задать значения исходных данных;
б) выбрать нужный переключатель: «Сумма ряда» или «Количество членов»;
в) решить задачу.

3. Разработка программы.

Для решения задачи в соответствии с интерфейсом, представленным на рис. 3.6, и определенным порядком действий, целесообразно разработать набор из шести подпрограмм:

- `double FuncArth (double aT, double aEps, int &aN)` – функция вычисления значений функции. Имеет два входных параметра – абсцисса `aT`, `aEps` – точ-

ность вычислений. И один выходной `aN` – количество членов ряда. Через заголовок функции будет возвращаться посчитанная ордината;

- `int DoTable_3_10 (double (*aFun)(double, double, int &), double aX, double aBeg, double aEnd, double aH, double *& aY, int *& aCou);` – предназначен для решения задачи 3.10. Передаются значения: имя функции; `aX` – аргумент; `aBeg`, `aEnd`, `aH` – границы интервала и шаг. Выходные значения: `aY` – выходной массив значений функции `aFun`; `aCou` – выходной массив значений, определяющих число членов ряда;
- `void TfoMain::DoTask_3_10 ();` – предназначен для решения задачи 3.10;
- `void RadioButton_3_10Click (TObject *Sender);` – обработчик события `OnClick` переключателя `RadioButton` для активации графиков;
- `Button_3_10Click(TObject *Sender);` – обработчик события `OnClick` кнопки `Button_3_10`, который запускает метод `DoTask_3_10` и открывает страницу с решением и графиками для задачи 3.10.

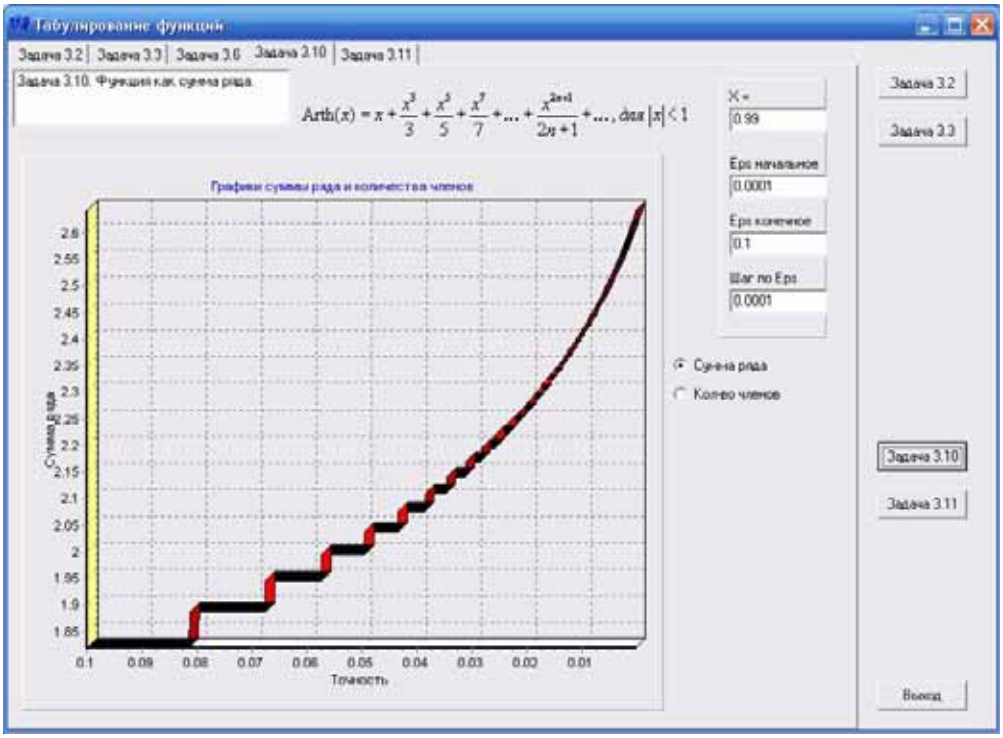


Рис. 3.6. Представление решения задачи на странице «Задача 3.10»

Вычисление значения функции *Arth(x)*, представленной в виде бесконечного ряда

Функция `FuncArth` будет принимать два значения: `aT` – координату точки, для которой нужно вычислить значение функции; `aEps` – заданную точность вычисления значения функции. Возвращать в качестве своего результата `FuncArth` будет также два значения: через заголовок – значение функции; `aN` – для получения количества слагаемых, необходимых для получения значения функции с установленной точностью. Второй выходной параметр введен, так как в задании требуется провести исследование на количество членов ряда. Он является и входным, и выходным.

В алгоритм введены вспомогательные переменные: `term` – в ней будет храниться очередное слагаемое; `sum` – в ней накапливается сумма. Для суммирования членов ряда используется оператор цикла *while*. Окончанием цикла служит условие, что очередной член ряда станет меньше заданного ε.

Алгоритм вычисления суммы ряда сводится к следующим шагам:

1. Инициализируется начальное значение числа слагаемых.
2. Выполняется цикл (пока не достигнута точность вычисления):
   2.1. Вычисляется значение числителя очередного слагаемого (сводится к циклу вычисления произведения).
   2.2. Формируется текущее значение слагаемого.
   2.3. Текущее значение добавляется к сумме.
   2.4. Увеличивается количество слагаемых на 1.

Вычисление значения функции `FuncArth` в заданной точке и с заданной точностью представлено в листинге 3.15.

Листинг 3.15. Вычисление значения функции $\mathrm{Arth}(x) = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \ldots + \frac{x^{2n+1}}{2n+1} + \ldots$ в заданной точке

```
double FuncArth (double aT, double aEps, int &aN)
{
            // инициализация очередного члена ряда
   double term = aT;
   double sum = term;// начальное значение суммы
   aN = 1; //начальное значения числа слагаемых
   while (fabs (term) > aEps)//пока не достигнута точность
     {
       term = 1;
       for (int i = 1; i <= 2 * aN + 1; i++)
          term = term * aT;
       term = term / (2 * aN + 1);
       sum = sum + term;
       aN = aN + 1;
     }
   return sum;
}
```

Примечание

Прямое вычисление каждого члена ряда, как это реализовано в листинге 3.15, не всегда эффективно. Так, например, если при вычислении очередного члена ряда используется факториал, то вычисление каждого нового члена ряда становится все более затратным, так как с ростом номера члена ряда для вычисления факториала необходимо все больше умножений. Как правило, в формулу очередного члена ряда входит факториал от номера, а то и несколько факториалов, например от индекса и от утроенного индекса. Поэтому для повышения эффективности вычисления суммы членов ряда, необходимо преобразовать порядок вычисления очередного члена ряда.

Стандартным приемом повышения эффективности вычислений является преобразование процесса прямого вычисления очередного члена ряда в процесс вычисления *очередного* члена ряда через *предшествующий* член ряда. В более сложных случаях может быть эффективнее вычислять *нечетные* члены ряда через *предшествующий нечетный* член ряда, а *четный* – через *предшествующий четный* член.

Более того, эффективно вычислять *часть от очередного члена ряда*, через такую же *часть предыдущего члена ряда*. И только потом «собирать» очередной член ряда и добавлять его к сумме предыдущих членов ряда.

Эти приемы позволяют сократить число операций при вычислении суммы ряда и, как следствие, уменьшить величину погрешности.

Примечание

Если расписывать процесс вычисления суммы ряда более детально, то можно порекомендовать следующий порядок.

1. В *простых случаях* производится вычисление очередного члена и его значение добавляется к сумме ряда. Выполняется проверка, не стало ли значение очередного члена ряда меньше заданного ε. Если да, то процесс вычисления суммы ряда прекращается.
2. Если *вычисление* каждого члена ряда является очень *громоздким* и *затратным*, то поступают следующим образом:
   а) разбивают формулу очередного члена ряда на составные части, например, вычисляют значение числителя и знаменателя по отдельности. Возможно, что и числитель (да и знаменатель) удобнее вычислять несколькими частями. Критерием удобства является положительный ответ на следующий вопрос: «Эта выделенная часть достаточно ли просто вычисляется через свое прежнее значение, посчитанное для предыдущего члена ряда?»;
   б) вычисляют каждую из составных частей; по их значениям вычисляют значение члена ряда и сумму ряда. Выполняют проверку, а не стало ли значение очередного члена ряда меньше заданного ε. Если да, то процесс вычисления суммы ряда прекращается.
3. Иногда несколько первых членов ряда не могут быть вписаны в общую схему алгоритма. Тогда такие члены ряда вычисляют по отдельному правилу до основного цикла. То есть основной цикл суммирования начинается некоторой ненулевой суммой – суммой первых членов ряда.

Построение таблицы, отражающей зависимости числа членов ряда от точности вычислений

В функцию `DoTable_3_10` передаются значения: имя функции; `aX` – аргумент; `aBeg`, `aEnd`, `aH` – границы интервала и шаг. Выходные значения: `aY` – выходной массив значений функции `aFun`; `aY`, `aCou` – выходные массивы значений, определяющие значения функции и число членов ряда. Вводятся вспомогательные переменные: `n` – для количества точек на интервале; `eps` – для текущего значения абсциссы.

Алгоритм построения таблицы, отражающей зависимости числа членов ряда от точности вычислений, можно описать следующими шагами.

1. Вычисляется количество точек на интервале в соответствии с заданными значениями.
2. Отводится память под массивы.
3. Инициализируется текущее значение абсциссы.
4. Выполняется цикл (по числу точек), в котором:
   а) вызывается функция для вычисления очередного значения функции, и ее результаты заносятся в формируемые массивы;
   б) изменяется текущее значение абсциссы на величину шага.

Описание функции `DoTable_3_10` представлено в листинге 3.16.

Листинг 3.16. Вычисление таблицы значений, предоставляющей зависимость числа членов ряда от точности вычислений

```
int __fastcall TfoMain::
         DoTable_3_10 (double(*aFun)(double, double, int &),
                       double aX, double aBeg, double aEnd, double aH,
                       double *& aY, int *& aCou)
  {
            // вычисляется количество точек
    int n = (aEnd - aBeg) / aH + 1;
    aY = new double [n];// отводится память
    aCou = new int [n]; // отводится память
    double eps = aBeg; // начальное значение eps
    for(int i = 0; i < n; i++)// делать цикл
      {
            // вызвать функцию для вычислений
        aY [i] = aFun (aX, eps, aCou [i]);
        eps += aH;// изменить eps на шаг
      }
    return n;// вернуть число точек в таблице
  }
```

Решение задачи 3.10

В процедуре `DoTask_3_10` используются вспомогательные переменные: `x` – вспомогательная переменная для хранения текущего значения; `eps1`, `eps2`, `epsH` – для хранения значений границ интервала табулирования и шага; указатели: `sums` – на массивы для значений сумм, `counts` – на массивы для количества членов ряда. Алгоритм решения задачи.

1. Отрабатываются стандартные действия по проверке корректности входных данных при их чтении из окон редакторов.
2. Вызывается процедура получения таблицы значений.
3. Вызывается процедура построения графика функции.
4. Вызывается процедура построения графика производной.

Описание метода решения задачи представлено в листинге 3.17.

Листинг 3.17. Описание метода решения задачи 3.10

```
void __fastcall TfoMain::DoTask_3_10 ()
{
  double *summs;
  int    *counts;
  double x, eps1, eps2, epsH;
      // если преобразование одного из текстов из окон
      // редактирования в число привело к ошибке,
      // то выдается сообщение и выход
  if (!(TryStrToFloat (le_3_10_X->Text, x)
        && TryStrToFloat (le_3_10_Eps1->Text, eps1)
        && TryStrToFloat (le_3_10_Eps2->Text, eps2)
        && TryStrToFloat (le_3_10_EpsH->Text, epsH)))
    {
       MessageDlg ("Одно из окон содержит не число!",
                   mtError, TMsgDlgButtons() << mbOK, 0);
       return;// выход
    }
      // если значения пределов или шага не соответствующие
  if (   x <= -0.999 || x >= 0.999
```

```
      || eps1 < 0.00001 || eps2 < 0.00001
      || eps1 >= eps2
      || epsH <= 0.000001
      || epsH > eps2 - eps1)
    {
           // сообщение
     MessageDlg (”Неправильное значение абсциссы”+
           ” или соотношение пределов интервала и шага!”,
                    mtError, TMsgDlgButtons() << mbOK, 0);
     return;// выход
    }
      // вызывается процедура вычисления таблицы значений
      // для получения значений сумм ряда, количества членов
      // ряда и количества точек табулирования
  int n = DoTable_3_10 (FuncArth, x, eps1, eps2, epsH, summs, counts);
      // вызывается процедура отображения графика функции
  DoPlot (Chart_3_10->Series [0], clRed, summs, n, eps1, epsH);

      // вызывается процедура отображение графика производной
  DoPlot (Chart_3_10->Series [1], clGreen, counts, n, eps1, epsH);

  delete[] summs, counts;  // высвобождение памяти
}
```

Примечание

Так как значения функции и количество членов ряда слишком разнятся, такие графики лучше строить на отдельных компонентах `Chart` или на одном, но тогда отображать их по отдельности, переключая их, например, с помощью радиокнопок. Помимо этого *следует инвертировать отображение нижней оси*, то есть, чтобы с ростом точности (а следовательно, уменьшением ε) было видно, что растет количество членов ряда, да и значение функции в данном случае, так как ряд с положительными членами. Такую настройку можно выполнить так:

- двойным щелчком по компоненту `Chart` вызвать окно *Editing Chart*;
- выбрать закладку *Axis*;
- выбрать нижнюю ось – радиокнопка `Bottom`;
- установить галочку в кнопке выбора `Inverted`.

Конечно, инвертировать ось не обязательно, но тогда необходимо понимать, что по графику точность вычисления выше, если мы ближе к оси.

Обработчик события `OnClick` кнопки `RadioButton` для активации графика

В процедуре-обработчике события `OnClick` кнопки `RadioButton` выполняется активация тех графиков, для которых включены переключатели `RadioButton`. В зависимости от выбора переключателя активируется построение выбранного графика, отражающего либо зависимость числа членов ряда от заданной точности, либо от суммы ряда, и корректируются подписи осей.

Описание процедуры-обработчика события `OnClick` переключателя представлено в листинге 3.18.

Листинг 3.18. Текст обработчика события `OnClick` переключателя

```
void __fastcall TfoMain::RadioButton_3_10Click (TObject *Sender)
```

```
// выбор отображаемого графика
{     // если выбор переключателя - «Сумма ряда», то
      // активируется первый график
  Chart_3_10->Series [0]->Active = RadioButton_3_10_Y->Checked;
      // если выбор переключателя - «Кол-во членов», то
      // активируется второй график
  Chart_3_10->Series [1]->Active = !RadioButton_3_10_Y->Checked;
      // поправляется надпись на оси 0y
  if (RadioButton_3_10_Y->Checked)
    Chart_3_10->LeftAxis->Title->Caption = "Сумма ряда";
  else
    Chart_3_10->LeftAxis->Title->Caption = "Количество членов ряда";
}
```

Обработчик события `OnClick` кнопки `Button_3_10` – «Решить задачу»

Алгоритм обработчика события `OnClick` кнопки «Решить задачу» содержит всего два действия:

- вызвать метод решения задачи;
- сделать активной страницу блокнота, на котором расположено решение.

Код обработчика события `OnClick` кнопки `Button_3_10` – «Решить задачу» приведен на листинге 3.19.

Листинг 3.19. Описание обработчика `OnClick`

```
void __fastcall TfoMain::Button_3_10Click(TObject *Sender)
// инициация решения задачи 3.10
// активация соответствующей страницы
{
  DoTask_3_10 (); // вызвать метод решения
  PageControl->ActivePage = TabSheet_3_10;
}
```

Задача 3.5. Написать процедуру вычисления функции *arcos(x)*, представленной в виде суммы бесконечного ряда, с заданной точностью **ε**:

$$\arccos(x) = \frac{\pi}{2} - \left(x + \frac{x^3}{2 \cdot 3} + \frac{1 \cdot 3 \cdot x^5}{2 \cdot 4 \cdot 5} + \frac{1 \cdot 3 \cdot 5 \cdot x^7}{2 \cdot 4 \cdot 6 \cdot 7} + \ldots \right), \text{ для } x^2 < 1.$$

Решение

1. Анализ задачи.

При решении предыдущей задачи стало очевидно, что вычислять степень абсциссы для каждого члена ряда неэффективно. То есть надо было позаботиться о том, чтобы на последующем шаге использовать уже вычисленную степень x, и только домножать ее на x^2, чтобы получить степень абсциссы для нового члена ряда. Да и при вычислении знаменателя можно было обойтись без умножения.

Когда члены ряда вычисляются более громоздко, необходимо более тщательно подходить к организации их вычисления. После исследования очередного члена ряда для определения значения функции arccos(x) можно рекомендовать следующий порядок действий:

- суммировать члены ряда до заданной точности, а уже потом вычесть эту сумму из константы $\pi/2$;

- инициализировать сумматор значением переменной x до суммирования ряда, так как этот член ряда не укладывается в общую схему вычисления очередного члена ряда;
- сохранить значение x^2 в отдельной переменной до вычисления ряда, так как степень абсциссы в числителе изменяется на эту величину x^2.

Как правило, числитель и знаменатель вычисляют по отдельности и только потом производят деление. В этой задаче вычисления числителя и знаменателя выполняются рекуррентно, а порядок вычисления очередного члена ряда рекомендуется следующий.

- Числитель вычислять по следующей накопительной схеме: значение предыдущего числителя домножать на очередное нечетное число и x^2.
- Знаменатель вычислять по следующей накопительной схеме:
  - отдельно накапливать произведение четных чисел;
  - отдельно подсчитывать очередное нечетное число. Оно, кстати, будет востребовано на следующем шаге при вычислении числителя.
- Окончательно очередной член ряда вычислять так: числитель разделить на произведение четных чисел и разделить на нечетное число.

Примечание

Следующая схема соответствует типовой: числитель и знаменатель вычислять по отдельности. Каждый из них, возможно, также вычислять по частям.

Но члены данного ряда можно вычислять по другой схеме. Очередной член ряда можно интерпретировать как значение предыдущего члена, помноженный на следующее нечетное число, x^2 и деленный на следующее четное число. И только одно нечетное число «предательски» перепрыгивает из знаменателя в числитель. Поэтому можно вести накопление некоторой переменной выполняя умножение на нечетное число, x^2 и деление на четное число. А окончательно очередной член ряда вычислять делением на нечетное число. Хотя выигрыша в количестве операций здесь, похоже, не получается.

2. Решение задачи.

Решение задачи здесь совершенно аналогично задаче 3.4. Разница только в содержимом подпрограммы вычисления суммы ряда.

Вычисление значения функции *Arccos(x)*, представленной в виде бесконечного ряда

Функция `FuncArcCos` имеет два входных параметра: абсцисса `aT`, `aEps` – точность вычислений. И один выходной `aN` – количество членов ряда. Через заголовок функции будет возвращаться посчитанная ордината. Вычисления сводятся к одному циклу, в котором

- формируется очередное слагаемое по описанному правилу;
- слагаемое добавляется к сумме;
- изменяется число слагаемых.

Описание функции `FuncArCos` представлено в листинге 3.20.

Листинг 3.20. Вычисление значения функции в заданной точке

```
double  FuncArCos (double aT, double aEps, int &aN)
// вычисление арккосинуса для x^2 <1
// типовые константы с π располагаются в math.h
{
   double numerator = aT;  // числитель
   double evenprod = 1;  // произведение четных
   double odd = 1, even = 2;  // очередное нечетное и четное
   double mult = aT * aT;  // множитель x^2
   aN = 1;  // подсчет количества членов ряда
   double term = aT;  // очередной член ряда
   double sum = aT;  // сумма членов ряда
   while (fabs (term) > aEps) // пока точность не достигнута
     {
           // подсчет числителя
       numerator = numerator * odd * mult;
           // подсчет произведения четных
       evenprod = evenprod * even;
           // изменение номеров на следующие значения
       odd = odd + 2; even = even + 2;
           // вычисление очередного члена
       term = numerator / (evenprod * odd);
       sum = sum + term;  // и накопление суммы
       aN = aN + 1; // изменение числа слагаемых
     }
           // окончательно: половина π без суммы
   return M_PI_2 - sum;
}
```



Показанные здесь приемы разработки подпрограмм, предназначенных для вычисления суммы ряда, являются базовыми и достаточными. Но при разработке таких подпрограмм есть опасность, что пользователь задал значение точки *x* в недопустимых пределах или по ошибке программиста, тогда члены ряда вычисляются неверно. Это может привести к бесконечному вычислению суммы ряда и аварийному завершению работы подпрограммы по переполнению.

С проблемой неверно заданных данных лучше бороться в той части программы, которая отвечает за работу с интерфейсом. То есть заданные пользователем данные должны проверяться не только на конвертируемость в целое или вещественное число, но и на допустимый диапазон.

Можно бороться с бесконечными вычислениями и в процедуре подсчета суммы ряда. Для этого достаточно ее текст дополнить проверкой, а не превысило ли количество членов ряда заданное ограничение. Такое ограничение можно задать константой в теле подпрограммы или передавать как входной параметр. Наверное, такую проверку лучше использовать только на этапе отладки, так как она тоже увеличивает время работы подпрограммы, хотя и достаточно мало.



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. Разработать приложение, предназначенное для получения таблицы значений функции на заданном интервале с заданным шагом, при условии, что значения функции вычисляются как сумма ряда с заданной точностью. Построить график функции.

Рекомендации: при решении задачи целесообразно совместить разработанные приложения к задачам 3.3 и 3.4.

2. Расширить задачу следующим образом: подпрограмму вычисления суммы ряда реализовать три раза, но на различных типах данных: `float`, `double` и `long double`. Построить два графика:

   а) график разности между функциями, вычисленными с использованием типов `long double` и `float`;

   б) график разности между функциями, вычисленными с использованием типов `long double` и `double`.

3. Известно, что физическое, эмоциональное и умственное состояние человека изменяется со дня его рождения циклически с периодом 23, 28 и 33 дня, соответственно. Написать программу построения графиков биоритмов на заданный интервал времени.

Рекомендации: для определения состояния человека для *i*-дня со дня рождения можно воспользоваться формулой: $Y = Sin(x)$, где $x = \left(\frac{i}{D} - \left[\frac{i}{D} \right] \right) \cdot 2\pi$, где $D = 23, 28, 33$, соответственно. Для получения графика, начиная со дня, определяемого заданной датой на *S* дней, нужно определить, каким днем со дня рождения является начальный день (с учетом того, что каждый четвертый год – високосный), и далее организовать цикл по номеру дня от 1 до *S* от заданной даты.

Класс «одномерный массив» 4

Огромное количество программных проектов выполняют обработку данных, размещенных в одномерных массивах. Некоторые проекты полностью нацелены на обработку единственного массива или даже набора массивов, другие проекты включают в себя фрагменты, которые обрабатывают массивы. При этом назначение данных, хранимых в массивах, может быть самым разнообразным: это могут быть однородные данные, полученные в результате измерений, например, температуры воздуха в течение суток или значения функции табулированной с постоянным шагом на заданном интервале.

Очевидно, что одномерный массив является той структурой данных, которая наилучшим образом способна представлять такой *математический объект* как векторы, а программы, предназначенные для решения задач векторной алгебры, по сути, являются подмножеством программ обработки одномерных массивов.

4.1. Разработка классов одномерных массивов

Общая постановка задачи

Создание приложений с использованием технологий объектно-ориентированного программирования (ООП) начинается с разработки набора классов, которые в последующем и будут составными элементами этих приложений.

Задача 4.1. Требуется разработать семейство классов-векторов, предназначенных для решения следующих задач:

- умножать вектор на заданное число (коэффициент);
- вычислять скалярное произведение двух векторов;
- определять длину вектора;
- вычислять сумму векторов;
- вычислять разность векторов;
- сменить знак вектора на противоположный;
- находить наименьшую (наибольшую) из координат.

Стратегия решения

Разработка класса – это самая трудоемкая и самая важная часть при решении задач с использованием технологий ООП. Начинают процесс решения с анализа данных. На данном этапе выделяются основные *математические объекты* и определяются их характеристики и операции, задающие методы поведения. Все это влияет на формирование структуры классов и их реализацию. Сначала прорисовываются и продумываются *интерфейсы классов*, и только после этого *формализуются методы* классов. Подготовленная система классов оформляется в виде отдельного модуля проекта. После чего для решения задач создается *интерфейс приложения* и обеспечивается его *функциональность*.

Решение

Весь процесс решения задачи необходимо разделить на несколько шагов. Результаты предыдущего шага будут основой для следующего.

1. Выделение математических объектов.

В рамках поставленной задачи фундаментом для разрабатываемого семейства классов является математический объект – многомерный вектор, принадлежащий пространству R^n. Одна часть перечисленных задач связана с преобразованием координат отдельного вектора. Другая часть – затрагивает взаимодействие двух векторов.

2. Разработка классов.

Разработку классов рекомендуется проводить в следующем порядке: разработка структуры данных; разработка модели поведения математического объекта; разработка схемы иерархии классов; разработка интерфейсов классов и алгоритмов для методов классов.

Разработка структуры данных

Конкретный вектор $\bar{a}$ определяется двумя характеристиками: *размерностью*, то есть размерностью пространства R^n, которому принадлежит вектор, и индивидуальным *набором координат*, определяющим именно этот вектор. Все остальные свойства такого вектора следуют однозначно из этих двух его характеристик.

Следует отметить, что элементы массива хранятся в одномерном массиве, память для которого должна будет выделяться автоматически в момент создания объекта типа класса. Именно благодаря такому решению можно будет создавать объекты – массивы с любым количеством элементов. Ключевой момент в том, что одномерный массив в *C++* в таком случае должен быть описан как *указатель*, например, типа `int*`. Эти две основные характеристики массива и будут определять поля класса.

Кроме этого, при выборе исходных данных для исследования поведения того или иного алгоритма, часто возникает необходимость знать такие характеристики объекта, которые позволяли бы получить представление о диапазоне изменения данных – *максимальное и минимальное значения* координат. Если это реальные массивы данных с известной или неизвестной структурой, то максимальное и минимальное значения потребуется вычислить. Если это искусственные массивы, создаваемые с помощью датчика случайных чисел, то максимальное и минимальное значения будут служить исходными данными для генерации самих значений элементов. Следовательно, в поля класса следует включить также максимальное и минимальное значения координат вектора.

Для удобства пользователей в классе можно предусмотреть возможность присваивания конкретному объекту-массиву (переменной типа созданного класса) *содержательного имени*. Имя будет храниться в виде переменной типа «массива символов» с размерностью, не превышающей количество символов, составляющих максимальное по числу символов имя объекта. В процессе решения задач это имя можно «поправлять», присоединяя к нему по мере продвижения вычислений знаки операций. Тогда результаты решения задач можно представлять на языке, близком к языку алгебры.

Разработка модели поведения математического объекта

Под моделью поведения (терминология принятая в технологии ООП) принято понимать совокупность методов, необходимых для решения определенного набора задач. Класс должен предоставлять пользователям следующие возможности: решать задачи, заявленные в условии; задавать исходные данные – размер массива и значения; генерировать значения по какому-либо правилу; формировать файл с данным вектором, с целью повторного использования; извлекать данные из файла данных и заполнять вектор; отображать вектор в различных видах как информацию. Сформулированные выше возможности будут определять поведение объектов класса и должны быть заложены в методы класса.

Естественно, что, кроме заявленных выше методов, класс-вектор, как и любой другой класс, должен иметь несколько конструкторов и деструктор.

Разработка схемы иерархии классов

При проектировании классов, как и при создании любой программы, разработчик должен изначально исходить из предположения, что предложенная задача со временем может претерпевать изменения, связанные с возрастанием и изменением запросов пользователя. Поэтому необходимо предусмотреть механизм, который бы позволял мобильно изменять интерфейсы классов, не затрагивая их реализацию. С этой целью имеет смысл сразу проектировать семейства классов.

Проектируя семейство классов, необходимо распределить, какие методы будут реализованы в базовом классе и какие из них будут абстрактными, сколько необходимо потомков и каким набором методов и свойств они должны обладать.

В рамках поставленной задачи целесообразно выделить семейство из *трех классов*. В *базовый класс* поместить методы, общие для перечисленных задач, в класс *наследник первой очереди* поместить методы решения этих перечисленных задач. А в третий класс *наследник от наследника* поместить методы, связанные с вводом и отображением данных. В таком представлении первые два класса будут мобильными, то есть они практически без исправлений могут компилироваться под любой системой программирования для *C++* и для различных операционных систем. Третий класс будет содержать методы с учетом работы в конкретной системе программирования. То есть он будет дополнять предыдущие классы методами, которые привязаны к типовым элементам, таким как классы и компоненты, предопределенные в нашем случае в среде *C++ Builder*.

Перечень задач, решаемых над массивами, гораздо шире списка задач над векторами. Поэтому класс типа вектор является частным случаем класса одномерный массив. Следовательно, целесообразно разработать класс типа одномерный массив, а уже от него наследовать базовый класс типа вектор.

Часть методов класса типа вектор, таких как сложение, умножение, смена знака и так далее, целесообразно оформить в виде операторных функций. Тогда применительно к векторам можно будет применять операции, естественные для записи алгебраических выражений. Все такие операторы, за исключением оператора присваивания, не следует включать в методы класса. Это поможет снять ограничения на построение выражений. Объявление оператора не функцией-членом класса требует предоставления такому оператору привилегированных прав доступа к данным класса. Это достигается предписанием `friend`.

Для некоторых операторов и функций желательно, чтобы один или оба параметра были бы объявлены с использованием ключевого слова *const.* Это будет гарантировать, что в процессе работы переданные значения (параметры) не будут изменены. Тогда попытки изменения будут выявлены на этапе трансляции как ошибки.

Введем следующие обозначения:

- базовый класс «одномерный массив» `TArray`;
- наследник `TArray`, базовый класс-вектор `TVector`;
- наследник `TVector`, вектор-класс с расширенными возможностями `TVectorEx;`
- наследник `TVectorTask` – с методами, привязанными к элементам, предопределенным в *C++ Builder*.

Описание базового класса



Базовый класс обычно не используется для создания экземпляров. Такие классы содержат элементы, общие для членов иерархии, и связывают иерархию классов воедино. Часто базовые классы содержат абстрактные методы.

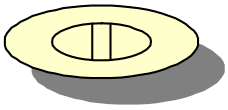
В интерфейс базового класса `TArray`, в соответствии с изложенными выше рассуждениями, будут включены:

- *поля:* длина массива, указатель на массив;
- конструкторы;
- деструктор;
- *свойства:* длина массива, индексированное свойство доступа к элементам, указатель на массив.

Поскольку в основную задачу конструктора входит отведение памяти и инициализация полей, исходя из различных входных данных, то для конструирования объектов типа класса `TArray` предусмотрено четыре конструктора. Интерфейс класса представлен в листинге 4.1.

Листинг 4.1. Интерфейс базового класса `TArray`

```
class TArray
 {
     protected:
       int *FData;   // указатель на координаты вектора
       int  FCount;  // размер векторного пространства
     public:
       TArray ();    // конструктор по умолчанию
       TArray (char* NameFile); // конструктор
       TArray (const TArray &aVect); // конструктор
        TArray (int aInitN, bool aZero = true); // конструктор
      ~TArray (); // деструктор
 };
```



НАЗНАЧЕНИЕ МЕТОДОВ КЛАССА `TArray`:

- `TArray (const TArray &aVect);` – конструктор имеет один параметр `aVect` с типом ссылки на класс. Необходим для передачи параметров, которые имеют тип класса и передаются «по значению»;

- `TArray ();` – конструктор *по умолчанию*. Его также называют пустым конструктором. Он задает «нулевые» значения полей, что обозначает объявление массива, в котором не содержится ни одного элемента;
- `TArray (char* NameFile);` – конструктор имеет один параметр `NameFile`. Через этот параметр передается имя текстового файла, в котором содержится информация для инициализации полей класса: число элементов, а также числовые значения координат вектора;
- `TArray (int aInitN, bool aZero = true);` – в этом конструкторе предусмотрено два параметра. Через параметр `aInitN` передается размер массива, в соответствии с этим размером отводится память. Второй параметр `aZero` может принимать два значения: `true` или `false`. Если передается значение `true`, то значения элементов массива обнуляются;
- `TArray (const TArray &aVect);` – конструктор *копирования*. Имеет один параметр `aVect`, который имеет тип ссылки на класс. Вызывается автоматически. Но в отдельных случаях конструктор копирования можно вызвать для конструирования экземпляра класса;
- `~TArray ();` – деструктор. Разрушает объект – высвобождает динамическую память, отведенную конструктором под массив.

В соответствии с результатами проектирования помимо методов, которые обеспечивают поведение классовых объектов, требуется разработать несколько конструкторов *для создания экземпляров* класса.

Задачи КОНСТРУКТОРА

Любой конструктор класса предназначен решить следующие задачи:
- принять информацию о способе конструирования;
- инициализировать поля класса;
- отвести память под массив.

В данном случае, простейший базовый класс `TArray` имеет только конструкторы и деструктор.

Конструктор без параметров

Этот конструктор создает «пустой» массив – массив, в котором нет элементов. Такой конструктор целесообразно использовать в тех случаях, когда параметры экземпляра класса не известны заранее, то есть их значения определяются в процессе работы. Описание такого конструктора представлено в листинге 4.2.

Листинг 4.2. Описание конструктора «по умолчанию» базового класса `TArray`

```
TArray::TArray ()
{  // Конструктор без параметров – «по умолчанию»
  FCount = 0;
  FData = NULL;
}
```

Конструктор с параметром «размер массива»

Описываемый конструктор имеет два параметра. Первый параметр – `aInitN` – определяет размер массива. В соответствии с этим параметром инициализируются

поля класса: инициализируется поле FCount и в соответствии с его значением отводится память под массив; и, как следствие этого, инициализируется второй параметр – указатель на массив. Второй параметр – aZero – логического типа. Он предоставляет пользователю возможность заполнить элементы массива нулевыми значениями, если aZero – истина (по умолчанию), и не делать этого, если aZero – ложь. Этот параметр может принимать значение «по умолчанию» – true. То есть при вызове этого конструктора можно указывать лишь один, тогда второй будет true. Описание конструктора представлено в листинге 4.3.

Листинг 4.3. Описание конструктора, который принимает размер массива для конструирования объекта

```
TArray::TArray (int aInitN, bool aZero)
// Конструктор массива: aInitN – длина массива
// aZero – параметр для указания
// «выполнять/не выполнять» обнуление данных
{
 FCount = aInitN; // инициализация размерности
 if (FCount == 0) FData = NULL; //если количество равно нулю
 else // если количество не равно нулю
  {
   FData = new int [FCount]; // отвести память под элементы
   if (aZero)  // если значение признака – истина
                 // то выполнить обнуление
     for (int i = 0; i < FCount; i++)
       FData [i] = 0; // присвоить нулевые значения
  }
}
```

Конструктор с одним параметром «имя файла»

Этот конструктор формирует экземпляр класса на основании той информации, которая содержится внутри файла, имя которого передается через параметр конструктора. Следовательно, прежде чем воспользоваться услугами такого конструктора, следует создать файл и подготовить его соответствующим образом. В данном случае – это текстовый файл, в котором содержатся числа, представляющие собой значения координат вектора. Причем в самом начале этого файла записано целое число, предопределяющее количество следующих значений, то есть размерность вектора. Обычно это количество располагают в отдельной строчке.

Алгоритм конструктора с параметром «имя файла» построен по следующей схеме:

а) обнуляются поля класса на случай, если произойдут сбои при работе с файлом;
б) открывается файл для чтения;
в) проверка, если файл не существует, то – выход из этой процедуры;
г) проверка, если файл пуст, то – выход из этой процедуры;
д) читается размер и инициализируется поле FCount;
е) отводится память под массив и инициализируется указатель FData;
ж) в цикле по числу элементов читаются из файла данные в массив;
з) закрывается файл.

Описание конструктора представлено в листинге 4.4.

Листинг 4.4. Описание конструктора, формирующего объект на основе информации, содержащейся в файле данных

```
TArray::TArray(char* NameFile)
// Конструктор массива с параметром «имя файла»
{
      // на случай, если ошибка чтения (нет файла)
   FCount = 0; FData = NULL;
      // открыть файл NameFile для чтения
   ifstream iFin(NameFile, ios::in);

   if(!iFin)return;  // если нет файла, то – выход
   if(iFin.eof())return; // если файл пуст, то - выход
   iFin>>FCount;     // прочитать число элементов
   FData = new int[FCount];  // отвести память под массив

                  // прочитать данные из файла
   for (int i = 0; i < FCount; i++) iFin>>FData[i];

   iFin.close();  // закрыть файл
}
```

Конструктор копирования

Следует обратить особое внимание на *конструктор копирования*, который добавлен к методам класса. Этот конструктор узнается компилятором по специфике его заголовка – один параметр с типом ссылки на класс. Причем этот параметр объявлен *const*, что говорит о том, что значение параметра не должно изменяться внутри описания тела конструктора.

КОНСТРУКТОР КОПИРОВАНИЯ

Конструктор копирования является неотъемлемой частью методов класса, в которых имеются поля типа указатели. Это связано с тем, что семантика вызова параметров «по значению» требует, чтобы локальная копия параметра создавалась и инициализировалась от значения выражения, переданного как фактический параметр. Компилятор предоставляет такой конструктор по умолчанию. Он производит копирование путем почленной инициализации. Для классов с полями-указателями такой способ не эффективен, так как указатель является адресом объекта, удаляемого при выходе из контекста. Это удаление объекта воздействует на другие экземпляры все еще предполагающие, что объект существует. Вызывается этот конструктор автоматически.

Конструктор копирования создает новый экземпляр класса по образу и подобию своего параметра `aVect` того же класса. Внутри методов к полям и методам именно этого экземпляра обращаться можно с помощью указателя `this`.

Указатель this

Указатель `this` связан с полями и методами текущего или «этого» класса. Обычно он не указывается в явном виде, то есть берется «по умолчанию». Но иногда без явного указания `this` не обойтись. Каждая функция класса имеет доступ к тому объекту, для которого она вызвана через указатель `this`. Поэтому этот параметр мы будем считать входящим в список формальных параметров и описывать его таковым. Типом `this` является тип *указателя на класс*.

Из параметра aVect конструктор копирования берет значения для инициализации своих полей. А затем копирует в this значения элементов массива. Эти действия выполняются в следующей последовательности:

а) из поля aVect значение копируется в поле, определяющее размер массива;
б) выполняется проверка. Если размер нулевой, тогда указатель на массив инициализируется NULL. Иначе отводится память под массив this;
в) выполняется поэлементное копирование элементов в цикле по числу элементов.

В листинге 4.5 дано описание конструктора копирования класса TArray.

Листинг 4.5. Описание конструктора копирования базового класса TArray

```
TArray::TArray (const TArray &aVect)
// Конструктор копирования. Порождается объект по образу
// и подобию входного объекта aVect
{
            // Читается размер массива из объекта aVect
 FCount = aVect.FCount;

 if (FCount == 0) FData = NULL;// если количество равно нулю
  else  // если количество не равно нулю
   {FData = new int [FCount]; // отвести память
    for (int i = 0; i < FCount; i++) // выполнить цикл для
      FData [i] = aVect.FData [i]; // копирования элементов
   }
}
```

Деструктор

Назначение деструктора – освободить память, отведенную под массив. Описание деструктора класса TArray представлено в листинге 4.6.

Листинг 4.6. Описание деструктора базового класса TArray

```
TArray::~TArray () // деструктор
{
  delete []FData; //освободить память, занятую массивом
}
```

Описание класса TVector

В интерфейс класса TVector, наследника класса TArray, помимо конструкторов и деструктора, необходимых для вызова конструкторов и деструктора класса-предка, размещены операторные функции и методы, реализующие выполнение математических операций над данным вектором или над парой векторов, перечисленных в условии задачи. Интерфейс класса TVector представлен в листинге 4.7.

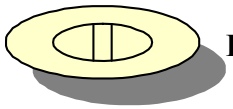
Листинг 4.7. Интерфейс класса TVector, потомка первой очереди от класса TArray

```
class TVector : public TArray       // АТД "Вектор"
{
 protected:
 public:
  TVector ();   // конструктор по умолчанию
  TVector (char* NameFile); // конструктор
  TVector (const TVector&); // конструктор копирования
  TVector (int aInitN, bool aZero = true);// конструктор
  ~TVector ();                  // деструктор
```

```
  int& operator [](int aInd);  // оператор индексации
  TVector& operator =(const TVector &aVect); // оператор
                                             // присваивания
  TVector& operator +(TVector &aVect); // оператор сложения
  float operator *(TVector &aVect); //скалярное произведение
  friend void Add (TVector aVect1, TVector aVect2,
                        TVector &aVect3);  // сумма векторов
  int  GetCount () {return FCount;}  // вернуть размер
  void SetCount (int aCount); // ввести-изменить размер
                              // вектора
  int  GetValue (int aInd);    // выдать значение
  void SetValue (int aInd, int aValue); // задать значение
};
```

Принятые обозначения в изложении

При описании методов, которые определены в классах, с целью упрощения описаний действий над классовыми переменными будем использовать сокращенную запись, которая, тем не менее, будет отражать суть действий. Например, запись `<aVect> + <bVect>`, если операнды описаны типом какого-либо векторного класса, будут обозначать: элементы вектора, определенного экземпляром класса `aVect` сложить с элементами вектора, определенного экземпляром класса `bVect`.



НАЗНАЧЕНИЕ МЕТОДОВ КЛАССА `TVector`:

- `TVector ();`
  `TVector (char* NameFile);`
  `TVector (const TVector &);`
  `TVector (int aInitN, bool aZero = true);` – конструкторы, включенные в описание класса-наследника для того, чтобы передать параметры, если они есть, классу-предку и инициировать выполнение конструктора предка;
- `~TVector ();` – деструктор. Вызывает деструктор предка, а тот, в свою очередь, высвобождает динамическую память;
- `int& operator [](int aInd);` – оператор индексации. Позволяет взять/вернуть значение элемента массива с номером `aInd`;
- `TVector& operator =(const TVector &aVect);` – оператор присваивания: `<*this> = <aVect>`;
- `TVector& operator +(TVector &aVect);` – оператор сложения векторов: `<*this> + <aVect>`;
- `float operator *(TVector &aVect);` – скалярное произведение векторов: `<*this> * <aVect>`;
- `friend void Add (TVector aVect1, TVector aVect2, TVector &aVect3);` – сложение векторов: `<aVect3> = <aVect1> + <aVect2>`;
- `int  GetCount () {return FCount;}` – вернуть длину вектора;
- `void SetCount (int aCount);` – ввести/изменить размер вектора на `aCount`, то есть установить новые размеры для вектора и переопределить память под значения элементов;
- `void SetValue (int aInd, int aValue);` – задать значение элемента массива с номером `aInd`: `aData[aInd] = aValue`.

Конструкторы и деструктор класса `TVector`

Следует обратить внимание, что в теле конструкторов и деструкторов потомка `TVector` в данном случае не должно быть никаких действий – они пусты. Но им обязательно необходимо «сказать» своему предку, что ему требуется выполнить свой конструктор. Если конструктору предка нужны параметры, то эти параметры потомок должен ему передать. Если у конструктора предка нет параметров, нужно «сказать», что требуется конструктор предка без параметров. С деструктором дело обстоит проще. Деструктор потомка автоматически вызывает выполнение деструктора предка. Описания конструкторов и деструктора класса `TVector` представлены в листинге 4.8.

Листинг 4.8. Описание конструкторов и деструкторов класса `TVector`

```
TVector::TVector () : TArray ()
// Выполнить конструктор предка
{
}
TVector::TVector (char* NameFile) : TArray (NameFile)
// Выполнить конструктор предка
{
}
TVector::TVector (const TVector &aVect) : TArray (aVect)
// Выполнить конструктор предка
{
}

TVector::~TVector ()
// Выполнить деструктор предка
{
}
```

Примечание

Следует иметь в виду, что при порождении объекта класса-потомка, конструктор базового класса будет вызываться автоматически перед выполнением операторов конструктора производного класса. В случае деструктора имеет место обратное: вначале выполняются операторы деструктора производного класса, а затем вызывается деструктор базового класса.

Оператор индексации

Данный оператор переопределяет операцию, известную нам по работе с обычными массивами: вернуть значение элемента массива по его номеру (например, `a[7]` – взять седьмой элемент в массиве `a`). Только эта операция в данном контексте будет определена уже для классовой переменной, точнее для массива, соотнесенного с классовой переменной. Данная операция является бинарной: один операнд `this` представляет массив; а второй операнд – индекс, он передается через параметр `aInd` операторной функции.

Схема алгоритма проста: проверяется индекс на допустимость. Если он находится в приемлемом диапазоне, то функция возвращает элемент массива с заданным номером. Описание оператора индексации представлено в листинге 4.9.

Листинг 4.9. Описание оператора индексации

```
int& TVector::operator [](int aInd)
// Оператор индексации
{
        // проверка индекса на допустимые пределы
  if (aInd < 0 || aInd > (FCount-1))

    {   // если индекс не укладывается в границы
     return (-10000);
    }
  return (FData[aInd]);
}
```

Оператор присваивания

Операция присваивания является бинарной. Первый операнд – указатель `this`. В операции присваивания он предполагается слева от знака «=». Второй операнд передается через параметр `aVect`. Поскольку операция присваивания связана с копированием элементов из массива в массив, необходимо, прежде всего, убедиться, что размеры этих массивов совпадают. Если размеры разные, возможны варианты действий: взять в качестве размера – результата тернарной операции – наименьшее из значений, и выполнить копирование соответствующего количества элементов; отказаться от операции с сообщением о несоответствии. В данном случае выбран первый вариант решения. В результате, после выполнения действий функции, возвращается ссылка на первый операнд. Описание операторной функции представлено в листинге 4.10.

Листинг 4.10. Описание оператора присваивания

```
TVector& TVector::operator =(const TVector& aVect)
// оператор присваивания
{
     // проверка размеров векторов – операндов на совпадение
  int cou = (FCount < aVect.FCount) ? FCount : aVect.FCount;
  for (int i = 0; i < cou; i++) FData [i] = aVect.FData [i];
                                                  // присваивание
  return (*this); // вернуть значения
}
```

Оператор сложения двух векторов

Операция также является бинарной. Первый операнд также является указателем `this`. Второй операнд передается через параметр `aVect`. При выполнении операции сложения двух векторов необходимо решить вопрос о том, куда поместить результат сложения? В реализации данной операции принято следующее решение: результат суммирования заносится в область памяти первого операнда в определяемой операции. Сокращенно эту операцию можно сформулировать: `<*this> = <*this> + <aVect>`. И перефразировать, например, так: «Операция предназначена для увеличения элементов *своей* матрицы на величину, соответствующую значениям элементов другой матрицы».

Операция присваивания сводится к выполнению цикла с суммированием соответствующих элементов. В том, что количества элементов в векторах-слагаемых совпадают, следует убедиться до применения этой операции, то есть выполнить в алгоритмах программ. После выполнения действий этой функции, результат воз-

вращается через первый операнд this как ссылка на текущий объект. Описание операторной функции сложения представлено в листинге 4.11.

Листинг 4.11. Описание оператора сложения двух векторов

```
TVector& TVector::operator+(TVector &aVect)
// оператор сложения: *this = (*this)+aVect
{
  for (int i = 0; i < FCount; i++)
     FData [i] = FData [i] + aVect.FData [i];
  return (*this);
}
```

Оператор «Скалярное произведение векторов»

Операция тоже бинарная и первый операнд тоже указатель this. Второй операнд aVect передается через параметр. Алгоритм сводится к процедуре накопления суммы, где каждое слагаемое представлено в виде произведения соответствующих векторных элементов. Полученная сумма возвращается как результат умножения. Проверку на совпадение размеров также рекомендуется производить в программе до вызова этой операции. Описание операторной функции, предназначенной для вычисления скалярного произведения векторов, представлено в листинге 4.12.

Листинг 4.12. Описание оператора умножения двух векторов

```
float TVector::operator *(TVector &aVect)
// оператор: скалярное произведение векторов (*this) * aVect
{
  float tmp = 0;
  for (int i = 0; i < this->FCount; i++)
    tmp = tmp + aVect.FData [i] * this->FData [i];
  return tmp;
}
```

Функция сложения векторов

Данный метод Add реализован в виде дружественной функции класса.

ФУНКЦИЯ-ДРУГ
friend

Дружественная функция – это специальная функция, не являющаяся членом класса, но которая имеет те же самые привилегии доступа, что и члены класса. Используется, если *среди параметров имеется классовая* переменная.

Следует обратить внимание, что в описываемой функции Add три классовых параметра. Два параметра aVect1 и aVect2 – входные, они передаются «по значению», а один aVect3 – выходной, он передается по ссылке.

Примечание

При выполнении функции Add дважды будет выполнен автоматический вызов конструктора копирования: для передачи значения aVect1 и для передачи значения aVect2.

С помощью метода определяется операция: <aVect3> = <aVect1> + < aVect2>. Операция реализуются с помощью цикла, в котором выполняется поэлементное сложение. Описание метода Add приведено в листинге 4.13.

Листинг 4.13. Описание метода, предназначенного для поэлементного сложения векторов

```
void Add (TVector aVect1, TVector aVect2, TVector &aVect3)
// функция суммирования векторов: aVect3 = aVect1 + aVect2
// векторы одной длины и все три уже определены
{
  for (int i = 0; i < aVect1.FCount; i++)
    aVect3.FData [i] = aVect1.FData [i] + aVect2.FData [i];
}
```

Функция взятия элемента по индексу

Действия, связанные с выполнением этой функции, уже описаны в представлении операции индексации. Функция введена для удобства. Пользователю предоставляется возможность выбора: воспользоваться описанным ранее оператором индексации или обратиться к этому методу. Возможно, при написании программы разработчик сочтет, что явный вызов этого метода класса является более удобным вариантом. Также метод можно использовать при описании свойства, которое будет возвращать значение элемента с индексом `aInd`. Описание метода представлено в листинге 4.14.

Листинг 4.14. Описание метода, возвращающего значение элемента по его номеру

```
int TVector::GetValue (int aInd)  // выдать значение
{
  return FData [aInd];
}
```

Функция изменения элемента с заданным индексом

Метод имеет два входных параметра. Первый параметр задает индекс элемента, значение которого нужно изменить. А второй параметр содержит новое значение, которое должен принять этот элемент. Решение сводится к одному действию. Описание метода, изменяющего элемент с заданным индексом, представлено в листинге 4.15.

Листинг 4.15. Описание метода, изменяющего значение элемента с заданным номером

```
void TVector::SetValue (int aInd, int aValue)
{  // задать значение
 FData [aInd] = aValue;
}
```

Функция переопределения длины вектора

Этот метод используется в том случае, когда в процессе преобразований или обработки данных изменяются размеры массива: либо происходит увеличение числа элементов, либо уменьшение. Тогда для корректной работы с экземпляром класса нужно переопределить его параметры. Метод имеет один параметр, который передает новый размер массива. Схема действий алгоритма по переопределению полей класса можно описать следующими шагами:

а) проверяется размеры нового и старого массивов на совпадение;
б) если размеры совпадают, то ничего менять не нужно;
в) если размеры не совпадают, то
   - создается вспомогательный массив;
   - копируются данные из старого массива во вспомогательный массив с целью их сохранения;

- освобождается память, занятая старым массивом;
- указатель на массив переназначается на вспомогательный;
- количество элементов устанавливается равным новому значению, переданному через входной параметр.

Описание метода `SetCount`, предназначенного для переопределения параметров класса, представлено в листинге 4.16.

Листинг 4.16. Описание метода `SetCount`, предназначенного для переопределения длины вектора

```
void TVector::SetCount (int aCount)
// Ввести-изменить размер вектора на aCount
// Если размеры новый и старый не совпадают,
// то выделяется новая память
{
  if (aCount <= 0)
   {FCount = 0;
    delete []FData; FData = NULL;
   }
  if (FCount != aCount)
   {
    int *aux = new int [aCount];
    for (int i = 0; i < min (FCount, aCount); i++)
      FData [i] = aux [i];
    delete[] FData;
    FData = aux;  // адрес нового массива в FCount
    FCount = aCount;
   }
  return;
}
```

Описание класса `TVectorEx`

В расширенный вектор-класс `TVectorEx`, который является наследником класса `TVector`, добавлены поля: *значение левой и правой границ* координатных значений; *имя экземпляра* класса; *имя файла* с векторными данными. В связи с появлением дополнительных полей, добавляются и новые конструкторы класса. То есть помимо тех конструкторов, которые предназначены для наследования конструкторов классов-предков, в новых конструкторах добавлены параметры для инициализации значений введенных полей.

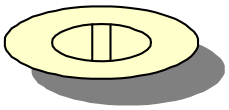
С целью соблюдения свойства инкапсуляции в интерфейс класса введены методы доступа к новым полям. Для ввода и отображения данных включены следующие методы: вернуть размер массива; сгенерировать значения координат вектора заданного диапазона с помощью датчика случайных чисел; вернуть значение элемента массива по его номеру.

Кроме этого, в интерфейс класса включены методы, которые позволяют задать левую и правую границы значений вектора, если его значения задаются датчиком случайных чисел (ДСЧ), или определять такие границы, если значения элементов вектора получены иным способом.

В листинге 4.17 представлен интерфейс класса `TVectorEx`.

Листинг 4.17. Интерфейс класса `TVectorEx` – расширенного класса `TVector`

```
class TVectorEx : public TVector
{
     protected:
   int  FLeft;        // левая граница координатных значений
   int  FRight;   // правая граница координатных значений
   char FNameObj [20];   // имя экземпляра класса
   char FNameFile [60];  // имя файла с векторными данными
     public:
  TVectorEx (); // конструктор по умолчанию
  TVectorEx(char* NameFile); // конструктор
  TVectorEx (const TVectorEx &aVect); // конструктор
  TVectorEx (int aInitN, char *aInitNameObj,
                       bool aZero = true);  // конструктор
  TVectorEx (char *File, char *aInitNameObj);// конструктор
  TVectorEx (int Initn, int InitLeft, int InitRight,
                  char *InitNameObj);  // конструктор
 ~TVectorEx ();  // деструктор
  int& operator [](int aInd);     // оператор индексации
  TVector& operator=(const TVectorEx& aVect);//присваивание
  TVector& operator +(TVectorEx& aVect);  // сложение
  float operator *(TVectorEx& aVect); //произведение
  void EnterDataRand ();  // задать значения ДСЧ
  void PrintAll ();
  void EnterLeft (); // ввести левую границу значений
  void EnterRight (); // ввести правую границу значений
  int  ReCalcLeft ();  // пересчитать значение левой границы
  int  ReCalcRight ();// пересчитать значение правой границы
  char* GetNameObj () {return FNameObj;} // вернуть имя
  void ReadFile ();
};
```



НАЗНАЧЕНИЕ МЕТОДОВ КЛАССА `TVectorEx`:

- `TVectorEx();`
  `TVectorEx (char* NameFile);`
  `TVectorEx (const TVector &);`
  `TVectorEx (int aInitN, bool aZero = true);`
  `TVectorEx (const TArray &aVect);` – конструкторы, включенные в описание класса-наследника для того, чтобы передать параметры, если они есть, конструкторам классов предков;
- `TVectorEx (int aInitN, char *aInitNameObj, bool aZero = true);` – конструктор принимает размер `aInitN` и значение `aZero`, которые передаются предкам для инициализации, а также имя экземпляра `aInitNameObj` – для инициализации собственного поля имени;
- `TVectorEx (char *File, char *aInitNameObj);` – конструктор принимает имя файла `File` и имя экземпляра класса `aInitNameObj`. Файл с именем `File` содержит значения исходных данных для полей класса. Этот параметр нужен для инициализации собственного поля `FNameFile`. А через `aInitNameObj` передается имя созданного экземпляра. Этот параметр нужен также для инициализации собственного поля `FNameObj`;

- `TVectorEx (int Initn, int InitLeft, int InitRight, char *InitNameObj);` – конструктор принимает размер вектора `Initn`, левую `InitLeft` и правую `InitRight` границы значений и имя объекта `InitNameObj`. Значение входного параметра `Initn` передается предку, а последние три параметра используется для инициализации собственных полей класса;
- `~TVectorEx();` – деструктор. Вызывает деструктор предка;
- `int& operator [](int aInd);` – оператор индексации. Возвращает значение элемента с номером `aInd`;
- `TVector& operator =(const TVectorEx& aVect);` – оператор присваивания: `*this = aVect`;
- `TVector& operator +(TVectorEx& aVect);` – оператор сложения: `<*this> + <aVect>`;
- `float operator *(TVectorEx& aVect);` – скалярное произведение векторов: `<*this> * <aVect>`;
- `void EnterDataRand();` – задать значения координат вектора датчиком случайных чисел;
- `void PrintAll();` – вывод всех элементов;
- `void EnterLeft (int InitLeft);` – ввести левую границу значений в поле `FInitLeft`;
- `void EnterRight (int InitRight);` – ввести правую границу значений в поле `FInitRight`;
- `int ReCalcLeft();` – пересчитать значение левой границы;
- `int ReCalcRight();` – пересчитать значение правой границы;
- `char* GetNameObj() {return FNameObj;}` – вернуть имя экземпляра.

Конструкторы и деструктор класса `TVectorEx`

Описания конструкторов и деструкторов класса представлены в листингах 4.18–4.24. Из описаний конструкторов следует, что в тех случаях, когда потомок имеет свои поля, в работе конструктора можно выделить две части: отдельные параметры передаются конструктору предка, а другие используются для настройки своих полей. Алгоритмы для описания методов класса настолько просты, что они не нуждаются в отдельном разборе. Достаточно познакомиться с комментариями в тексте программ.

Листинг 4.18. Описания конструктора без параметров класса `TVectorEx`

```
TVectorEx::TVectorEx () : TVector ()
// конструктор пустой с заданием имени
// инициализирует поле FNameObj с именем «по умолчанию»
{
  strcpy (FNameObj, "NoNamed");
}
```

Листинг 4.19. Описания конструктора класса `TVectorEx` с параметром – именем файла

```
TVectorEx::TVectorEx (char* NameFile) : TVector (NameFile)
// конструктор пустой с заданием имени файла данных
// передает файл классу – предку
// инициализирует поле FNameObj
```

```
{
  strcpy(FNameObj, "NoNamed");
}
```

Листинг 4.20. Описания конструктора класса `TVectorEx` с параметрами – размером массива и именем объекта

```
TVectorEx::TVectorEx (int aInitN, char* aInitNameObj, bool aZero) :
                                                    TVector (aInitN, aZero)
// конструктор принимает размер aInitN, имя объекта -
// aInitNameObj и признак для обнуления/не обнуления aZero.
// aInitN, aZero - передает файл классу - предку
{
 strcpy(FNameObj,aInitNameObj); // инициализирует
                                // поле FNameObj
}
```

Листинг 4.21. Описания конструктора класса `TVectorEx` с параметрами – именем файла и именем объекта

```
TVectorEx::TVectorEx (char *aFile, char *aInitNameObj): TVector ()
// Инициализация полей класса
{
  strcpy (FNameFile, aFile);
  strcpy (FNameObj, aInitNameObj);
  ReadFile (); // чтение данных из файла
}
```

Листинг 4.22. Описания конструктора класса `TVectorEx`, которому передаются: размер массива, значения левой и правой границ данных, имя объекта

```
TVectorEx::TVectorEx (int aInitN, int aInitLeft, int aInitRight,
                                  char *aInitNameObj) : TVector (aInitN)
// значение параметра aInitN передается предку
{
      // Инициализация полей класса
  FLeft = aInitLeft;
  FRight = aInitRight;
  strcpy (FNameObj, aInitNameObj);
}
```

Листинг 4.23. Описания конструктора копирования класса `TVectorEx`

```
TVectorEx::TVectorEx (const TVectorEx &aVect): TVector (aVect)
// Конструктор копирования.
// Значение параметра aVect передается предку
{
   // Копирование собственных полей
  FLeft  = aVect.FLeft;
  FRight = aVect.FRight;
  strcpy (FNameObj, aVect.FNameObj);
  strcpy (FNameFile, aVect.FNameFile);
}
```

Листинг 4.24. Описания деструктора класса `TVectorEx`

```
TVectorEx::~TVectorEx ()
{
}
```

Функция чтения вектора из файла

Чтобы воспользоваться методом `ReadFile`, нужно заранее создать текстовый файл, в котором расположить *следующую информацию*: число элементов в массиве (предпочтительно, чтобы на отдельной строчке); значения элементов массива в нужном количестве (числа должны друг от друга отделяться пробелами). Имя этого файла должно быть известно на этапе конструирования экземпляра класса.

Алгоритм сводится к следующим шагам:

- читается размер массива в поле `FCount`;
- отводится память под массив `FData`;
- в цикле читаются данные из файла в элементы массива.

Описание метода `ReadFile` представлено в листинге 4.25.

Листинг 4.25. Описания метода `ReadFile` класса `TVectorEx`, предназначенного для чтения данных из файла

```
void TVectorEx::ReadFile ()
//
{
  ifstream fin;
  fin.open (FNameFile, ios::in);
  fin >> FCount;
  delete []FData;
  FData = new int [FCount];
  for (int i = 0; i < FCount; i++)
    fin >> FData [i];
  fin.close ();
}
```

Создание вектора с помощью датчика случайных чисел

Функция `EnterDataRand` не имеет параметров. Она предназначена для заполнения значений вектора датчиком случайных чисел. Случайные числа задаются в диапазоне, заданном параметрами класса `FRight` и `FLeft`. Описание метода, предназначенного для создания векторных значений датчиком случайных чисел, представлено в листинге 4.27.

Листинг 4.27. Описания метода `EnterDataRand` класса `TVectorEx`, предназначенного для создания векторных значений датчиком случайных чисел

```
void TVectorEx::EnterDataRand ()
{
  for (int i = 0; i < FCount; i++)
    this->FData [i] = random (FRight - FLeft) + FLeft + 1;
}
```

Установка левой границы векторных значений

Функция `EnterLeft` имеет один входной параметр, через который задается/изменяется левая граница, ограничивающая диапазон, в котором изменяются значения вектора. Описание метода `EnterLeft`, предназначенного для установки левой границы векторных значений, представлено в листинге 4.28.

Листинг 4.28. Описания метода `EnterLeft` класса `TVectorEx`, предназначенного для установки левой границы

```
void TVectorEx::EnterLeft (int InitLeft)
```

```
// Инициализировать левую границу вектора
{
  FLeft = InitLeft;
}
```

Установка правой границы векторных значений

Функция имеет один параметр, через который задается/изменяется правая граница, ограничивающая диапазон, в котором изменяются значения вектора. Описание метода `EnterRight`, предназначенного для установки правой границы векторных значений, представлено в листинге 4.29.

Листинг 4.29. Описания метода `EnterRight` класса `TVectorEx`, предназначенного для установки правой границы

```
void TVectorEx::EnterRight (int InitRight)
// Инициализировать правую границу вектора
{
  Fright = InitRight;
}
```

Определение левой границы векторных значений

Алгоритм процедуры `ReCalcLeft`, предназначенной для определения левой границы векторных значений, сводится к поиску наименьшего из векторных значений по схеме:

1. В `FLeft` заносится первый элемент массива `FData[0]`.
2. В цикле (по числу элементов) производится сравнение значения `FLeft` с текущим элементом массива `FData`. Если текущий элемент меньше, чем `FLeft`, то значение переменной `FLeft` заменяется на это меньшее значение.

Описание алгоритма, определяющего левую границу векторных значений, представлено в листинге 4.30.

Листинг 4.30. Описания метода класса `TVectorEx`, определяющего левую границу

```
int TVectorEx::ReCalcLeft ()
// Вернуть левую границу значений вектора
{
  int i;
  for (i = 1, FLeft = FData [0]; i < FCount; i++)
    if (FData[i] < FLeft) FLeft = FData [i];
  return FLeft;
}
```

Определение правой границы векторных значений

Алгоритм метода `ReCalcRight` сводится к поиску наибольшего из векторных значений. Описание представлено в листинге 4.31.

Листинг 4.31. Описания метода класса `TVectorEx`, определяющего правую границу

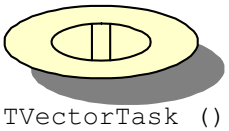
```
int TVectorEx::ReCalcRight ()
// Вернуть правую границу значений вектора
{
  int i;
  for (i = 1, FRight = FData [0]; i < FCount; i++)
    if (FData [i] > FRight) FRight = FData [i];
  return FRight;
}
```

Описание класса `TVectorTask`

В класс `TVectorTask` помещены такие методы, которые оперируют с параметрами, имеющими типы, указывающими на визуальные компоненты *C++ Builder*. Интерфейс класса `TVectorTask` представлен в листинге 4.32.

Листинг 4.32. Интерфейс класса `TVectorTask` – потомка от `TVectorEx`

```
class TVectorTask : public TVectorEx
 {
  protected:
  public:
  TVectorTask ();     // конструктор по умолчанию
  TVectorTask (const TVectorTask &aVect); // конструктор
  TVectorTask(char*aInitNameObj):TVectorEx(aInitNameObj) {};
                                  // конструктор
  TVectorTask (int InitN, char *aInitNameObj, bool aZero);
                     // конструктор с обнулением
  ~TVectorTask (); // деструктор
  void ReadFromStrList (TStrings *aSList);
                    // чтение вектора из TStringList
  void SaveToStrList (TStrings *aSList, bool aClear = true);
         // запись вектора в TStringList, с очисткой или нет
 };
```



НАЗНАЧЕНИЕ МЕТОДОВ КЛАССА `TVectorTask`:

- `TVectorTask ();`
  `TVectorTask (char* NameFile);`
  `TVectorTask (const TVector &aVect);`
  `TVectorTask (int aInitN, bool aZero = true);` – конструкторы, включенные в описание класса-наследника для того, чтобы передать параметры конструкторам предков;
- `TVectorTask(char *aInitNameObj);` – конструктор. Передает имя экземпляра `aInitNameObj` конструктору класса предка;
- `TVectorTask (int InitN, char *aInitNameObj, bool aZero);` – конструктор принимает размер `aInitN`, значение параметра `aZero`, имя экземпляра `aInitNameObj`. Эти значения передаются конструкторам предков для инициализации;
- `~TVectorTask ();` – деструктор. Вызывает деструктор предка;
- `void ReadFromStrList (TStrings *aSList);` – производит чтение вектора из компонента `aSList` типа `TStringList`;
- `void SaveToStrList (TStrings *aSList, bool aClear = true);` – производит запись вектора в компонент `aSList` типа `TStringList` с предварительной очисткой/не очисткой полей компонента.

Конструкторы и деструктор класса `TVectorTask`

Описания конструкторов и деструкторов класса приведены в листингах 4.33–4.36. Они не нуждаются в дополнительном описании, поскольку используются

только для того, чтобы передать параметры конструкторам классов-предков и заставить их выполниться.

Листинг 4.33. «Пустой» конструктор класса `TVectorTask`

```
TVectorTask::TVectorTask () : TVectorEx ()
// конструктор по умолчанию
{
};
```

Листинг 4.34. Конструктор копирования класса `TVectorTask`

```
TVectorTask::TVectorTask (const TVectorTask &aVect) : TVectorEx (aVect)
{
};
```

Листинг 4.35. Конструктор класса `TVectorTask` – потомка от `TVectorEx`

```
TVectorTask::TVectorTask (int aInitN, char *aInitNameObj, bool aZero) :
                                TVectorEx (aInitN, aInitNameObj, aZero)
// конструктор с обнулением
{
};
```

Листинг 4.36. Деструктор класса `TVectorTask` – потомка от `TVectorEx`

```
TVectorTask::~TVectorTask ()
{
}
```

Функция чтения векторных значений из списка

Метод чтения значений элементов вектора из списка строк `ReadFromStrList` имеет один входной параметр `aSList` типа `TStrings`. Этот параметр определяет список, в котором находятся данные массива.

Примечание

Следует отметить, что фактическим параметром в таком случае может быть как соответствующее *поле* компонента типа `TMemo*`, так и любого другого. Важно, чтобы входным параметром был класс, являющийся наследником класса `TStrings`.

На основании данных, значения которых представлены в списке `aSList`, нужно инициализировать поля класса. При этом может оказаться так, что класс уже создан и содержит ненулевой указатель на массив. Алгоритм метода `ReadFromStrList` предписывает:

а) определить количество строк в списке и занести в поле класса `FCount`;
б) проверить, является ли указатель на массив `FData` нулевым;
в) если указатель не NULL, то высвободить память, занятую предыдущим массивом;
г) отвести память под массив `FData` в соответствии со значением `FCount`;
д) в цикле (по числу элементов в списке) прочитать очередное значение из списка с преобразованием в число и занести в поле текущего элемента массива.

Описание метода `ReadFromStrList` представлено в листинге 4.37.

Листинг 4.37. Описание метода `ReadFromStrList` класса `TVectorTask`, предназначенного для чтения данных из списка строк

```
void TVectorTask::ReadFromStrList (TStrings *aSList)
// чтение вектора из TStringList
{
  FCount = aSList->Count;
  if (FData != NULL) delete []FData;
  FData = new int [FCount];
  for (int i = 0; i < FCount; i++)
    FData [i] = StrToInt (aSList->Strings [i]);
}
```

Запись векторных значений в список

Метод записи значений элементов в список строк `SaveToStrList` имеет два входных параметра. Первый параметр `aSList` определяет список, в который нужно поместить элементы массива. Второй параметр `aClear` является признаком, который определяет, нужно ли очистить/не очистить список до того, как занести в него новые значения. Вначале проверяется значение признака `aClear`. Если его значение – истина (`true`), то список очищается. После чего выполняется цикл, в котором перебираются последовательно элементы массива, преобразуются в строку типа `AnsiString`, добавляющуюся к списку. Описание метода `SaveToStrList` представлено в листинге 4.38.

Листинг 4.38. Описание метода `SaveToStrList` класса `TVectorTask`, предназначенного для записи данных в список

```
void TVectorTask::SaveToStrList (TStrings *aSList,
                                 bool aClear)
// запись вектора в TStringList
{
  if (aClear) aSList->Clear ();  // почистили список
  for (int i = 0; i < FCount; i++)
    aSList->Add (IntToStr (FData[i]));
}
```

4.2. Управление классом через элементы интерфейса

После создания классов и обустройства формы, необходимо предоставить пользователю возможность работы с этими классами, то есть необходимо *обеспечить доступ к свойствам и методам* класса. Для этого необходимо создать переменную типа класс – экземпляр класса.

Описание переменной типа КЛАСС

Описание переменной типа класс, то есть создание экземпляра класса, в общем случае отличается от описания обычной переменной. Это связано с тем, что классовая переменная *должна создаваться с помощью конструктора*, которому при необходимости следует *передать параметры*. Возможны два варианта описания:

а) `<имя_класса> <имя_переменной> (<список параметров>);`

б)
```
<имя_класса>* <имя_переменной>;
<имя_переменной> = new <имя_класса> (<список параметров>);
```

Здесь в списке параметров содержатся параметры, предназначенные конструктору класса для создания экземпляра.

Напоминание Именно по списку параметров система *C++* определяет, какой конструктор следует вызвать.

При описании программ более предпочтительным является второй вариант. В этом случае используются переменные типа «указатель на класс», например

```
TVectorTask *FVector_1, *FVector_2;
```

Где расположить объявление переменной с типом класса?

1. Можно предложить вариант: в .h файле (в заголовочном файле) формы в разделе `private` класса формы.
2. А можно поместить объявление в `cpp`-файле формы, расположив объявление рядом с объявлением переменной типа класса формы.

Объявленная таким образом классовая переменная становится глобальной для формы (редактора кода), где она объявлена. И, следовательно, ее можно конструировать с помощью того или иного конструктора, а затем вызывать методы класса для решения задач.

4.3. Разработка интерфейса приложения для решения задач с классом «Одномерный массив»

Приложения решения задач векторной алгебры, использующие разработанные классы, как правило, требует в качестве исходных данных задания значений одного или двух векторов. В свою очередь, выходным значением или результатом решения будет либо число, либо вектор, значения которых можно использовать для изучения требуемого свойства математического объекта.

В основе разработки интерфейса решения любой задачи обычно исходят из желания сделать так, чтобы наработки, полученные в результате создания этого приложения, можно было в дальнейшем активно использовать для других задач с минимумом переделок. Это, своего рода, вклад в будущее. Поэтому в нашем случае решение задачи как бы разделено на два отдельных этапа. Один – для того, чтобы задать исходные данные. Второй – для того, чтобы решить задачу. Причем первый этап будет являться общим для всех задач, решаемых с использованием класса. Отсюда становится очевидной целесообразность тщательной его проработки с включением всех возможных защит от ошибок.

На рис. 4.1 представлена главная форма приложения. На главной форме (модуль `FMain.cpp`) отражено, что процесс решения задачи состоит из двух этапов. Решение каждого из этапов оформлено в виде «блокнота» – визуального компонента `PageControl`. Эти блокноты занимают практически все поле главной формы. Левый блокнот служит для подготовки исходных данных, необходимых для вычислений. Правый блокнот – для решения задачи и отображения результатов. Такой вариант для решения данной многофункциональной задачи оказался наиболее предпочтителен из-за предоставляемой им возможности на одном поле независимо располагать решения различных задач. Тогда для решения новой задачи требуется только добавить новую страницу к блокноту решений и оформить на ней решение (компоненты

+ свойства + функциональность). Каждая страница блокнота справа позволяет решить задачу из некоторой группы задач. А наименование страницы, по существу, будет являться характеристикой этой группы.

Кроме этого, для удобства пользователя, предусмотрена вспомогательная *модальная* форма (модуль `FEditData.cpp`), которая предназначена для возможного редактирования данных. На форме имеется также главное меню, обеспечивающее выполнение общих задач приложения.

МОДАЛЬНОЕ ОКНО

Модальное окно по определению – это всплывающее окно, обычно отображающее или запрашивающее информацию. Это окно не может быть первичным окном в разрабатываемом приложении. Во время отображения такого окна остальные окна приложения недоступны, но не окна других приложений *Windows*.

Особенности или свойства модальных окон:

- может присутствовать кнопка закрытия окна;
- заголовок окна объясняет его функцию;
- нет строки состояния;
- может иметься кнопка «Что это?» для получения справки.

Примечание

Ниже представлена разработка только этапа задания исходных данных, реализованного на странице блокнота с именем «Загрузка векторов». Второй этап завязан на конкретную задачу и потому рассматривается и разрабатывается отдельно в рамках соответствующей задачи.

Страница «Загрузка векторов»

Страница «Загрузка векторов» предоставляет возможность:

- загрузить для решения задачи данные из файлов (кнопка «Загрузить»);
- выполнить контроль данных (кнопка «Контроль»);
- при необходимости исправить их значения (кнопка «Исправить»).

По кнопке «Загрузить» пользователю предоставляется возможность в режиме диалога выбрать нужный файл, содержащий значения векторов. Эти значения из файла переносятся в список `ListBox`. Высвечивается количество данных в поле компонента `StaticText`. Одновременно в списке `ListBox`, помеченном меткой «Индексы», проставляются индексы элементов. Поле «Индексы» позволяют не ошибиться при сравнении значений, так как индекс, соответствующий выбранному набору, выделяется цветом.

Если возникла ошибочная ситуация или необходимость в проверке данных, то нажимается кнопка «Исправить». В таком случае открывается модальная форма `foEditData`, интерфейс которой представлен на рис. 4.2. На форме `foEditData` предусмотрен компонент `Memo`, в который «автоматически» загружаются данные, а также высвечивается количество данных в поле компонента `StaticText`. Данные можно проверить и исправить. После чего следует выполнить команду «`OK`».

Для того чтобы обезопасить пользователя от возможных ошибок, предусмотрены дополнительные проверки, а именно: при закрытии формы проверяется, за-

крывается ли она с помощью кнопки «OK». Если это так, тогда проверяется, сделан ли контроль и дал ли он хороший результат. При возникновении ошибки программа не разрешает закрывать форму. При нажатии пользователем кнопки Cancel программа разрешает закрыть форму. Если форму foEditData закрыли с помощью кнопки «OK», то форма foMain перегружает новые данные в список значений вектора поля ListBox.

Таким образом, на странице «Загрузка векторов» располагаются: 3 компонента ListBox для отображения векторных значений и индексов, имеющие имена Name – lbVector_1, lbVector_2, lbIndexes; 5 компонентов TLabel для записи поясняющего текста; 6 кнопок BitBtn с именами BitBtn1…BitBtn6 для управления решением, со свойствами Caption: «OK», «Загрузить», «Загрузить», «Контроль», «Исправить», «Исправить», соответственно.

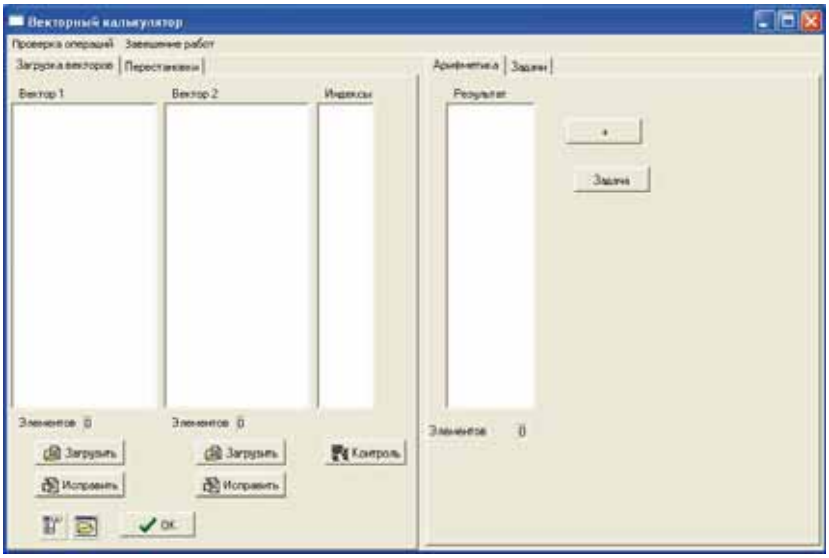


Рис. 4.1. Главная форма приложения с открытыми страницами «Загрузка векторов» и «Арифметика»

Кроме этого, на форме предусмотрены два компонента StaticText для отражения количества элементов в первом и втором списках, с именами stNumberOf_1 и stNumberOf_2, соответственно. И, конечно, невизуальные компоненты: главное меню MainMenu и стандартное диалоговое окно OpenDialog, со значением полей: Name – dlgOpenFile; Options: OfPathMustExist, OfFileMustExist – установленных в значение True. Такая настройка параметров поля Options заставляет окно dlgOpenFile ограничивать пользователя в выборе: пользователь может выбрать только существующий файл в существующей папке.

Чтобы обеспечить в рамках страницы «Загрузка векторов» описанную технологию, нужно подготовить соответствующие тексты для пяти обработчиков событий, предназначенных для пяти кнопок.

Примечание

Поскольку, по замыслу авторов, на этапе «Загрузка векторов» подготавливаются исходные данные для работы, необходимо ввести описания этих *классовых* данных.

Придадим им смысловые имена и свяжем их с последним классом-потомком: TVectorTask *FVector_1, *FVector_2.

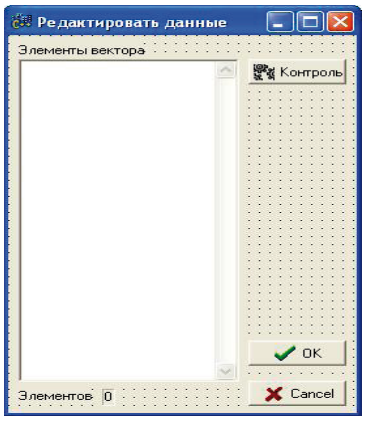


Рис. 4.2. Модальная форма для редактирования данных

Рекомендация

Так как использование этих переменных начинается сразу после запуска программы, имеет смысл создать эти экземпляры в обработчике события `OnCreate` главной формы.

Помимо этого *для удобства пользователя* целесообразно сделать так, чтобы по щелчку мыши на строке индексов (событие `OnMouseUp`) выделялись элементы в `ListBox` с соответствующими номерами. Это не позволит ошибиться при сравнительном анализе данных.

Таким образом, для решения задачи в соответствии с интерфейсом, представленным на рис. 4.1, целесообразно разработать набор из шести подпрограмм – обработчиков событий:

1) `void BitBtn1Click (TObject *Sender);` – обработчик события `OnClick` под кнопку «Загрузить» для первого вектора;
2) `void BitBtn2Click (TObject *Sender);` – обработчик события `OnClick` под кнопку «Загрузить» (для второго вектора);
3) `void BitBtn5Click(TObject *Sender);` – обработчик события `OnClick` под кнопку «Исправить» (для первого и второго векторов);
4) `void BitBtn4Click(TObject *Sender);` – обработчик события `OnClick` под кнопку «Контроль»;
5) `void lbIndexesMouseUp (TObject *Sender, TMouseButton Button, TShiftState Shift, int X, int Y);` – обработчик события `OnMouseUp`, то есть «Нажать кнопку мыши» на списке «Индексы»;
6) `void FormCreate(TObject *Sender);` – обработчик события `OnCreate`, то есть «На создание формы» для главной формы проекта.

Обработчик события `OnClick` кнопки «Загрузить»

Алгоритм действий под кнопку «Загрузить» сводится к следующим шагам:

1) проверяется, произведен ли выбор файла средствами компонента `OpenDialog`;
2) если файл выбран «удачно», тогда
   а) загрузить данные из файла в список `lbVector_1`;
   б) записать количество строк в `stNumberOf_1` и `stNumberOf_2`;
   в) проверить на совпадение количество индексов, уже отображенных в окне, содержащем списки индексов, с числом строк, взятых из файла;

г) если количества не совпадают, то список индексов очищается, и создается новый список.

Описание алгоритма, предназначенного для загрузки первого вектора, представлено в листинге 4.39. Текст процедуры загрузки второго вектора совершенно аналогичен и отличается лишь именем компонента `lbVector_2`.

Листинг 4.39. Описание действий обработчика события `OnClick` кнопки «Загрузить»

```
void __fastcall TfoMain::BitBtn2Click (TObject *Sender)
{
  if (dlgOpenFile->Execute()) // если файл выбран
            // то загрузить из файла первый вектор
    {
     lbVector_1->Items->LoadFromFile
                                  (dlgOpenFile->FileName);
            // записать количество строк в StaticText
     stNumberOf_1->Caption = lbVector_1->Count;
            // если число строк в списке «Индексы»
            // не совпадает с числом строк, взятых из файла
     if (lbIndexes->Count != lbVector_1->Count)
         {
                  // очистить список «Индексы»
            lbIndexes->Clear();
                  // создать новый список «Индексы»
            for (int i = 0; i < lbVector_1->Count; i++)
              lbIndexes->Items->Add (IntToStr (i + 1));
         }
    }
}
```

Обработчик события `OnClick` кнопки «Исправить»

Алгоритм действий обработчика события `OnClick` кнопки «Исправить» реализован по следующей схеме:

1) очищается поле `Memo` для данных на второй форме `foEditData`;
2) загружаются в освобожденное поле `Memo` значения вектора;
3) отображается окно «Редактировать данные». В это время пользователь видит окно `foEditData` с данными на форме, как на рис. 4.1, и имеет возможность внести исправления, и нажать кнопку «OK»;
4) если отработала команда «OK», то на главной форме, на странице «Загрузка векторов», выполняется следующее:
   а) стираются прошлые значения;
   б) загружаются новые значения;
   в) сверяются размеры с размерами индексного списка и, если на то есть необходимость, индексы перезаписываются.

Описание обработчиков событий `OnClick` кнопок «Исправить» представлено в листингах 4.40–4.41 для первого и второго векторов, соответственно.

Листинг 4.40. Описание обработчика события `OnClick` кнопки «Исправить» первого вектора данных

```
void __fastcall TfoMain::BitBtn5Click(TObject *Sender)
// позволяет исправить данные, расположенные в поле
// формы редактирования, и перезаписать
// в список значений для Vector_1 главной формы
```

```
{
      // очистка списка «Элементы вектора»
  foEditData->meEditData->Clear();
      // загрузка 1-го вектора
  for (int i = 0; i < lbVector_1->Count; i++)
    foEditData->meEditData->Lines->
                        Add (lbVector_1->Items->Strings [i]);
      // если нажата кнопка «Ok»
  if (foEditData->ShowModal () == mrOk)
    {
           // «очистить» прошлые значения вектора
      lbVector_1->Clear();
           // загрузить вектор из отредактированного списка
      for (int i = 0; i < foEditData->meEditData->
                                              Lines->Count; i++)
        lbVector_1->Items->Add (foEditData->meEditData->
                                              Lines->Strings [i]);

           // если не совпадают списки «Индексы»
      if (lbIndexes->Count != lbVector_1->Count)
        {
                // очистить и перезаписать
          lbIndexes->Clear();
                // загрузка 2-го вектора

          for (int i = 0; i < lbVector_1->Count; i++)
            lbIndexes->Items->Add (IntToStr (i + 1));
        }
           // перезаписать количество элементов
      stNumberOf_1->Caption = lbVector_1->Count;
   };
}
```

Листинг 4.41. Описание обработчика события `OnClick` кнопки «Исправить» для второго вектора данных

```
void __fastcall TfoMain::BitBtn6Click(TObject *Sender)
// позволяет исправить данные, расположенные в поле
// формы редактирования, и перезаписать
// в список значений для Vector_2 главной формы
{
      // очистка списка «Элементы вектора»
  foEditData->meEditData->Clear();
     // загрузка 2-го вектора
  for (int i = 0; i < lbVector_2->Count; i++)
    foEditData->meEditData->Lines->Add (lbVector_2->
                                              Items->Strings [i]);
      // если нажата кнопка «Ok»
  if (foEditData->ShowModal () == mrOk)
   {
      // «очистить» прошлые значения вектора
    lbVector_2->Clear();
      // загрузить вектор из отредактированного списка
    for (int i = 0; i < foEditData->meEditData->
                                              Lines->Count; i++)
      lbVector_2->Items->Add (foEditData->meEditData->
                                              Lines->Strings [i]);

           // если не совпадают списки «Индексы»
```

```
    if (lbIndexes->Count != lbVector_2->Count)
     {
           // очистить и перезаписать
       lbIndexes->Clear();
       for (int i = 0; i < lbVector_2->Count; i++)
                  lbIndexes->Items->Add (IntToStr (i + 1));
      }
      // перезаписать количество элементов
    stNumberOf_2->Caption = lbVector_1->Count;
   };
}
```

Обработчик события `OnClick` кнопки «Контроль»

Кнопка «Контроль» служит для старта процедуры проверки корректности данных: векторные значения должны совпадать по количеству; данные в строках `ListBox` должны быть представлены таким образом, чтобы эти строки можно было преобразовать в числовое значение. Для простоты в построение алгоритма введена локальная переменная `flag` – для фиксации факта успеха/неуспеха в проверке корректности данных. Тогда алгоритм можно свести просто к последовательности проверок:

1) проверить соответствие размеров;
2) если размеры совпали, то перейти к пункту 4;
3) если размеры не совпали, то сообщить об этом факте и выйти;
4) установить значение `flag = true`;
5) выполнить цикл с целью проверки успешности попытки преобразовать строки в числа (функцией `StrToInt`) для элементов первого вектора;
6) если все попытки оказались удачны, то перейти к пункту 8;
7) если обнаружена ошибка преобразования, то `flag = false`, выдать сообщение и выйти;
8) установить значение `flag = true`;
9) выполнить цикл с целью проверки успешности попытки преобразовать строки в числа (функцией `StrToInt`) для элементов второго вектора;
10) если все попытки оказались удачны, то перейти к пункту 12;
11) если обнаружена ошибка преобразования, то `flag = false`, выдать сообщение и выйти;
12) загрузить первый вектор в вектор `FVector_1`;
13) загрузить второй вектор в вектор `FVector_2`.

Описание обработчика события `OnClick` кнопки «Контроль» представлено в листинге 4.42.

Листинг 4.42. Описание обработчика события `OnClick` кнопки «Контроль»

```
void __fastcall TfoMain::BitBtn4Click(TObject *Sender)
{
  if ((lbVector_1->Count != lbVector_2->Count) ||
      (lbVector_1->Count != lbIndexes->Count) ||
      (lbVector_2->Count != lbIndexes->Count))
                  // контроль на длины векторов и индексов
   {
    ShowMessage
          ("Длины векторов или списка индексов НЕ равны!");
    return;
```

```
   }

  int aux = 0;
  int flag = true;
           // контроль корректности данных первого вектора
  for (int i = 0; i < lbVector_1->Count; i++)
    if (!TryStrToInt (lbVector_1->Items->Strings [i], aux))
      {flag = false; break;};
  if (!flag)
   {
    ShowMessage "Первый вектор имеет неправильные данные!");
    return;
   }

  flag = true;
           // контроль корректности данных второго вектора
  for (int i = 0; i < lbVector_1->Count; i++)
    if (!TryStrToInt (lbVector_1->Items->Strings [i], aux))
     {flag = false; break;};
  if (!flag)
   {
    ShowMessage ("Второй вектор имеет неправильные данные!");
    return;
   }
    // загружаем векторы
  FVector_1->ReadFromStrList(lbVector_1->Items);
  FVector_2->ReadFromStrList(lbVector_2->Items);
}
```

Обработчик события для выравнивания списков

Содержит действия по синхронизации отображения списков и выделения элементов в них. Сводится к выполнению операторов присваивания соответствующим полям компонентов. Описание обработчика события `OnClick` кнопки «Выровнять списки» приведено в листинге 4.43.

Листинг 4.43. Описание обработчика события `OnClick` кнопки «Выровнять списки»

```
void __fastcall TfoMain::lbIndexesMouseUp(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
  lbVector_1->TopIndex = lbIndexes->TopIndex;
  lbVector_2->TopIndex = lbIndexes->TopIndex;

  lbVector_1->ItemIndex = lbIndexes->ItemIndex;
  lbVector_2->ItemIndex = lbIndexes->ItemIndex;
}
```

Обработчик события «На создание формы»

Этот обработчик содержит два действия: создать экземпляры классов, то есть отвести память под векторные данные. Описание обработчика события `OnCreate` главной формы приведено в листинге 4.44.

Листинг 4.44. Описание обработчика события `OnCreate` главной формы

```
void __fastcall TfoMain::FormCreate(TObject *Sender)
// Создать экземпляры классов. Используется конструктор
// с одним параметром – «Имя объекта»
{
```

```
  FVector_1 = new TVectorTask ("Первый");
  FVector_2 = new TVectorTask ("Второй");
}
```

4.4. Решение задач векторной алгебры с использованием класса «Одномерный массив»

Предложенные к рассмотрению задачи решаются в рамках разработанной системы классов. Решение задачи оформляется в виде отдельного метода класса. Это может быть метод нового, созданного класса-потомка в построенной иерархии классов, а может быть методом уже разработанного класса, например `TVectorTask`.

В соответствии с предложенной концепцией интерфейса, при подготовке решения задачи можно поступить следующим образом:

- либо добавить к уже разработанной странице новую кнопку, если есть все компоненты, которые необходимы, чтобы отразить результат, и вписать текст обработчика события «нажатие на кнопку». По сути, написать процедуру, решающую задачу;
- либо добавить к блокноту новую страницу, и на этой странице организовать решение.

Примечание

При изложении раздела «Решение задач» разработка интерфейса для решения будет предложена только для решения первой задачи. Для всех последующих – только разработка алгоритмов, так как задача разработки интерфейса во всех случаях практически одинакова.

Задача 4.2. Даны два вектора, значения которых хранятся в файлах данных. Требуется найти сумму этих двух векторов.

Решение

Среди методов класса имеется метод, который позволяет выполнить сложение векторов. Векторные значения хранятся в текстовых файлах. Вывод: можно воспользоваться разработанным интерфейсом, представленным на рис. 4.1, с соответствующей ему функциональной поддержкой. Для решения задачи и получения результата суммирования к блокноту, расположенному в правой части формы, добавим страницу, которую назовем «Арифметика» и обеспечим решение задачи в соответствии со следующей технологией.

Технология подготовки решения с уже имеющимся методом

1. Расположить на форме в отведенной области соответствующие компоненты, необходимые для решения задачи: `ListBox` – для отражения результата, то есть векторных значений; `StaticText` – для фиксации размера вектора; `Label` – для поясняющего текста; кнопку `Button` – для обозначения действия «Выполнить сложение».
2. Задать свойства компонентов с помощью *Инспектора объектов*.

3. Поместить описания классовых переменных (если они еще не объявлены) в разделе `private` класса главной формы приложения. В данном случае они уже описаны под именами: `FVector_1`, `FVector_2`.
4. Подготовить и написать текст обработчика события «Выполнить сложение».

Обработчик события «Выполнить сложение» содержит вспомогательную локальную переменную `result` классового типа. Для ее создания вызывается конструктор, которому передаются три параметра:

- размер;
- название объекта;
- требование сделать предварительную очистку данных.

Размер определяется с помощью метода `GetCount`, вызванного для вектора `FVector_1`.

Алгоритм решения задачи сводится к следующим шагам.

1. Вызывается метод `Add` для сложения векторов.
2. Результирующий вектор `result` сохраняется в списке.
3. Если список с индексами не соответствует по размерности списку, предоставленному `result`, то список индексов очищается и формируется заново с требуемым количеством значений.

Описание метода приведено в листинге 4.45.

Листинг 4.45. Описание метода под кнопку «+»

```
void __fastcall TfoMain::bbAddClick(TObject *Sender)
{
  TVectorTask result (FVector_1->GetCount (),
                                "Результат : сумма", true);

  Add (*FVector_1, *FVector_2, result);

  result.SaveToStrList (lbVector_Res->Items, true);

  if (lbIndex_Res->Count != lbVector_Res->Count)
   {lbIndex_Res->Clear();
    for (int i = 0; i < lbVector_Res->Count; i++)
      lbIndex_Res->Items->Add (IntToStr (i + 1));
   }
  stNumberOf_Res->Caption = lbVector_Res->Count;
}
```

Задача 4.3. Требуется сгенерировать множество неповторяющихся случайных целых чисел из заданного диапазона значений.

Решение

Для решения данной задачи предлагается следующая стратегия.

1) Последовательно, одно за другим, генерируются случайные числа. Получаемые случайные числа пересылаются в массив.
2) Числа, начиная со второго, сравниваются с теми, которые получены ранее. Если такое число уже есть в массиве, то будет генерироваться новое число, пока в итоге не получится требуемое количество случайных чисел.

Имеет смысл осуществить *проверку сгенерированного числа на повторяемость* в уже созданной части массива. А процесс проверки *выделить в отдельную задачу и оформить в виде метода класса*. Тогда алгоритм решения задачи может быть описан с помощью двух циклов. Внешний цикл служит для формирования значений элементов по следующей схеме.

а) Генерируется очередное число заданного диапазона.
б) Вызывается процедура для проверки того факта, есть ли сгенерированное число среди чисел, созданных ранее.
в) Если такое число имеется, генерируется другое число, и происходит возврат к пункту б.
г) Если такого числа нет, то сгенерированное и проверенное число добавляется к сформированному списку элементов.

Пункты б) и в) выполняются в цикле внутреннем. Описание методов решения представлены в листингах 4.46 и 4.47.

Листинг 4.46. Проверка числа на повторяемость в сформированной части массива

```
int TVectorTask::Detect (int R, int k)
// R – число, для которого выясняется факт присутствия его
// в массиве FData, содержащем элементы с номерами от 0 до k
// включительно. Функция возвращает значение 1, если число
// обнаружено в массиве. И возвращает 0, если числа нет.
{
  for(int i=0; i<=k; i++) // делать цикл
    if (R == FData[i]) return 1;// если есть совпадение
  return 0; // если нет факта совпадения
}
```

Листинг 4.47. Генерация различных чисел из заданного диапазона

```
void TVectorTask::F4_1()
// Используются значения полей класса, определяющие
// допустимые границы изменения чисел
{
   for(int i=0; i<FCount; i++) // делать цикл
     {
       int R = random(FRight-FLeft)+FLeft; // создать число
       while (Detect(R, i)) // пока «число присутствует»
         R = random(FRight -FLeft)+FLeft; // пересоздать число
       FData[i] = R; // занести в массив
     }
}
```

Задача 4.4. Требуется расположить элементы заданного массива по возрастанию методом «плавающего пузырька».

Решение

Для решения данной задачи предлагается следующий алгоритм «плавающего пузырька»:

1. В последовательности чисел сравниваются поочередно 2 соседних числа `FData[i]` и `FData[i+1]`, где `0 ≤ i < FCount – 1`.
2. Если `FData[i] ≤ FData[i+1]`, то осуществляется переход к сравнению следующих членов последовательности, то есть `FData[i+1] ≤ FData[i+2]` и так далее, что означает движение **вправо** по последовательности.

3. А иначе, если `FData[i] > FData[i+1]`, то делается перестановка сравниваемых членов и далее осуществляется проверка для всех элементов от *i* до 0 (то есть движение **влево** по последовательности), а не надо ли делать перестановку и, возможно, ни одну среди них.
4. Движение влево продолжается до тех пор, пока либо переставляемый элемент не займет своего места, либо не будет обнаружен левый конец массива данных.
5. Если при движении вправо не достигнут правый конец массива, то переходим к пункту 2. Иначе, алгоритм завершает работу.

Описание метода представлено в листинге 4.48.

Листинг 4.48. Описание метода «плавающего пузырька»

```
void TVectorTask::SortSwim()
{
  for(int i = 0; i < FCount-1; i++)// движение вправо
    for (int k = i; k >= 0 && FData[k] > FData[k+1]; k--)
        {
          int Buf = FData[k];
          FData[k] = FData[k+1];
          FData[k+1] = Buf;
        }
}
```

Задача 4.5. Требуется расположить элементы заданного массива по возрастанию с использованием метода быстрой сортировки.

Решение

Для решения данной задачи предлагается применить алгоритм К. Хоора, использующий принцип обмена. В основу алгоритма положен тот факт, что для достижения наибольшей эффективности желательно производить обмены элементов, которые расположены на больших расстояниях друг от друга. Суть алгоритма:

а) выбирается случайным образом элемент внутри массива. Это может быть любой элемент, но обычно берут его в середине. Назовем его `xmid`;
б) просматривается массив слева направо, пока не отыщется `FData[i] > xmid`;
в) просматривается массив справа налево, пока не отыщется `FData[j] > xmid`;
г) меняются местами `FData[i]` и `FData[j]`;
д) продолжается выполнение пунктов б)–г) до тех пор, пока индекс `i` не превышает значения индекса `j`. В результате массив разделится на две части, слева – меньшие `xmid`, а справа – большие `xmid`;
е) процедуру разделения с обеими частями рекурсивно повторяют до тех пор, пока каждая часть не будет содержать только один элемент.

Описание реализации метода представлено в листинге 4.49.

Листинг 4.49. Описание метода быстрой сортировки

```
void TVectorTask::QuickSort( int nl, int nr)
{
  int i, j, xmid;
  i = nl;  // индекс для движения справа налево
  j = nr;  // индекс для движения слева направо
  xmid = FData[(nl+nr)/2]; // элемент в середине массива
  while ( i <= j ) // пока индексы не встретятся
    {
```

```
      while (FData[i] < xmid) i++; // пока не встретился
                                   // элемент, больше чем xmid
      while (FData[j] > xmid) j--; // пока не встретился
                                   // элемент, меньший xmid
      if (i <= j)
       {
         int z = FData[i]; // перестановка элементов
         FData[i] = FData[j];
         FData[j] = z;
         i++; j--; // изменение индексов
       }
    }
  if (nl < j) QuickSort(nl,j);
  if (i < nr) QuickSort(i,nr);
}
```

Примечание

Следует обратить внимание, что во внутренних циклах условие завершения цикла не завязано на переменную `i`. Но цикл всегда будет конечным, так как внутри массива есть значение `xmid`, и при попадании на это значение срабатывает условие выхода из цикла.

Задача 4.6. Требуется расположить элементы заданного массива в обратном порядке.

Решение

По условию задачи требуется поменять местами: первый элемент массива с последним элементом, второй элемент – с предпоследним элементом и так далее. Если количество элементов в массиве `FCount`, то число таких перестановок будет равно `FCount/2`. Для решения этой задачи предлагается алгоритм: задаются начальные значения индексов переставляемых элементов. По одной из этих переменных организуется цикл, в котором элементы переставляются. Цикл повторяется, пока индексы переставляемых элементов не встретятся. Описание метода представлено в листинге 4.50.

Листинг 4.50. Описание алгоритма обращения массива

```
void TVectorTask::Reverse()
// Обозначим: i - начальный индекс массива,
// j – конечный индекс массива
// цикл - пока начальный и конечный индексы не встретятся
{
  int j;
  for(int i=0, j=FCount-1; i<j; i++, j--)
    {
       int z = FData[i];
       FData[i] = FData[j];
       FData[j] = z;
    }
}
```

Задача 4.7. В массиве есть повторяющиеся элементы. Сформировать новый массив, в котором не будет дубликатов.

Решение

Предположим, что элементы массива имеют вид: 2, 3, 4, 45, 2, 5, 6, 45, 18, 5, 67. Предлагается следующая стратегия: просматриваются элементы массива по порядку от начала до конца – это внешний цикл. Для каждого элемента проверяется, есть ли элемент с таким значением после/правее – это внутренний цикл. Если такой элемент есть, то внутренний цикл прерывается и берется для анализа следующий элемент во внешнем цикле. Если правее нет элемента с таким же значением, то элемент записывается в формируемый массив. В соответствии с предложенным алгоритмом в новый массив будут отправлены те элементы и в том порядке, как они отмечены в исходном массиве. Первый элемент 2 пропускается, так как справа от него есть элемент 2. Элемент 3 отправляется в новый массив, так как он единственный и справа от него нет 3; по этой же причине передается в новый массив 4; элемент 45 пропускается, так как справа есть 45; элемент 2 записывается в новый массив (правее нет 2, записывается последний дубль) и так далее. Описание метода приведено в листинге 4.51.

Листинг 4.51. Формирование массива без дублирующих элементов

```
void TVectorTask::Duplicate(TVectorTask& aVect)
// aVect – классовая переменная, в нее заносится
// результирующий массив
{
  int* A = new int[FCount]; // вспомогательный массив
  int j, k = 0;
  for(int i=0; i<FCount; i++) // перебираются элементы
   {
    for (j=i+1; j<FCount; j++) // сравниваются с
                                  // элементами справа
      if (FData[i]==FData[j]) break;
      if (j==FCount) A[k++] = FData[i];
    }
  aVect.SetCount(k);//устанавливаются новые размеры aVect
  for(int i=0; i<k; i++) aVect.FData[i] = A[i];
  delete []A; // высвобождение памяти
}
```

Задача 4.8. Выполнить циклический сдвиг элементов массива.

Решение

Задачу циклического сдвига элементов массива можно сформулировать следующим образом.

Требуется переместить элементы массива, состоящего из N элементов, вправо/влево на M позиций, при этом M элементов из конца массива перемещаются в начало. Например, результат циклической перестановки $A = \{a_0, a_1, a_2, a_3, a_4\}$ вправо на 2 позиции будет $A = \{a_3, a_4, a_0, a_1, a_2\}$.

Рассмотрим следующий алгоритм. Во вспомогательную переменную пересылается последний элемент массива *A*. Затем все элементы массива сдвигаются вправо на одну позицию и на место первого элемента перемещается содержимое вспомогательной переменной. Эта процедура повторяется *M* раз. Описание метода сдвига элементов массива вправо на *M* позиций представлено в листинге 4.52.

Листинг 4.52. Циклический сдвиг элементов массива

```
void TVectorTask::DisPlace(int M)
{
  for(int i=0; i < M; i++)
    {
      int P = FData[FCount-1]; // Запоминается последний
                               // элемент в P
      // Сдвиг массива на одну позицию
      for (int j=FCount-1; j>=1; j--)
        FData[j] = FData[j-1];
      FData[0] = P; // На место первого помещается число из P
    }
}
```

Задача 4.9. Из заданного массива выбрать неповторяющиеся элементы и переписать их в новый массив.

Решение

Для решения этой задачи предлагается следующий алгоритм. Берется элемент массива и выполняется проверка, есть ли элементы с таким же значением, расположенные слева и справа от проверяемого элемента. Если такого элемента нет, то он – единственный в массиве и его следует переписать в новый массив. Такая процедура повторяется `FCount` раз (по числу элементов). Новый массив определяется через параметр метода с типом классовой переменной. Описание метода представлено в листинге 4.53.

Листинг 4.53. Выбор неповторяющихся элементов

```
void TVectorTask::NotRepeat(TVectorTask& aVect)
 {
  int* A = new int[FCount]; // Вспомогательный массив
  int Flag; // Признак повторения
  int k=0;
    // Цикл для перебора элементов с целью проверки
  for(int i=0; i<FCount; i++)
   {
    Flag = 1; // Начальное значение признака

    // Цикл проверяет на повторяемость элемент FData[i]
    for (int j=0; j<FCount; j++)
      if (j != i && FData[i]==FData[j])
       {
        Flag = 0; // Если повторился, то меняется признак
        break; // Выход из цикла проверки
       }
      if(Flag) // Если признак не изменился
       {
        A[k++] = FData[i]; // В новый массив
       }
   }
  aVect.SetCount(k); // Установка новых параметров
    // Копирование элементов в класс
  for (int i=0; i<k; i++) aVect.FData[i] = A[i];
  delete []A; // Высвободить память
}
```

Согласно листингу 4.53 для реализации алгоритма введены вспомогательные переменные: массив `A` – в него заносятся неповторяющиеся элементы, и переменная `Flag`. Переменная `Flag` служит признаком повторяемости. Если переменная не повторяется, то `Flag = 1`, если повторяется, то `Flag = 0`. После того, как будет сформирован требуемый массив и, соответственно, определена его длина, вызывается метод `SetCount` для результирующей переменной классового типа `aVect`. Этот метод переопределяет параметры экземпляра класса. После чего значения элементов массива `A` переносятся в поля векторных значений `aVect`.

Задача 4.10. В данном массиве найти подпоследовательность, наибольшую по длине и состоящую из одинаковых элементов, расположенных подряд.

Решение

Предлагается алгоритм, в основе которого лежит повторение двух циклов. Один из этих циклов находит начало подпоследовательности из одинаковых элементов. В этом цикле сравниваются соседние элементы между собой до тех пор, пока не встретятся одинаковые соседи. Эту одинаковость следует положить в основу критерия выхода из цикла. Второй цикл будет находить конец подпоследовательности из одинаковых элементов. В нем также сравниваются соседи, но сравниваются на проверку их одинаковости. В основу критерия выхода из цикла следует положить различие. Конечно, нужно проверять, не закончились ли элементы в массиве при выполнении операции сравнения. В результате выполнения этих двух циклов определяется длина участка из одинаковых элементов. Эта длина проверяется на максимум.

Описанные два цикла и проверка на максимум повторяются до тех пор, пока в последовательности данных не будут просмотрены все элементы. Следует отметить, что длина очередного участка вычисляется по-разному в зависимости от того, расположен ли участок в конце списка или нет. Если все элементы различные, то `Beg` и `Size` останутся равными нулю. Описание реализации метода представлено в листинге 4.54.

Листинг 4.54. Выбор наибольшей по длине подпоследовательности из одинаковых элементов

```
void TVectorTask::MaxLengthPart(int& Beg, int& Size)
// Beg - начало участка из одинаковых элементов
// Size - длина участка из одинаковых элементов
// iBeg, iSize - переменные для хранения промежуточных
// значений
{
 int iBeg, iSize, i=0;
   // Установка/инициализация начальных значений
 Size = 0;
 Beg = 0;
   // Делать цикл, пока есть элементы
 do {
     // цикл, пока элементы разные
   while (FData[i]!=FData[i+1]&& i<FCount) i++;
     // если элементы не закончились
   if(i<FCount)
   iBeg=i; // запомнить начало участка
      // цикл, пока элементы равны
   while(FData[i]==FData[i+1]&&i<FCount) i++;
```

```
    // определить и запомнить длину участка
   if (i==FCount)iSize = i-iBeg;
   else iSize = i-iBeg+1;
   if (Size < iSize) // проверить на максимум
    {Size = iSize; Beg = iBeg;}
   } while(i<FCount);
}
```

Задача 4.11. Объединить два упорядоченных массива в один упорядоченный.

Решение

В алгоритм вводятся три вспомогательные переменные – для задания индексов массивов. Индексы массивов будем менять независимо. Если элемент из первого массива будет меньше, чем элемент из второго массива, то в результирующий массив записывают элемент из первого массива. При этом индекс у первого массива увеличивается на 1, а индекс у второго не меняется. Все это выполняется в цикле до тех пор, пока очередной элемент первого массива не станет больше элемента 2-го массива. Тогда в результирующий массив начинается цикл по пересылке элементов из второго массива (индекс у второго массива увеличивается на 1, а индекс у первого не меняется) пока условие снова не нарушится. Оба цикла будут повторяться в объемлющем цикле до тех пор, пока в каком-либо из массивов не будут исчерпаны все элементы. Если один из массивов будет исчерпан, то хвостовые элементы другого массива пересылаются, элемент за элементом, без каких-либо проверок в результирующий массив. Описание метода представлено в листинге 4.55.

Листинг 4.55. Слияние возрастающих массивов в один возрастающий

```
void TVectorTask::Merger(TVectorTask& aVect,TVectorTask&
                                        // aRes)
// Сливаются в один массив два вектора: FData – текущий
// из класса и вектор FData из классовой переменной aVect
// Результат слияния помещается в классовую переменную aRes
// i – индекс вектора из aVect
// j – индекс вектора из текущего объекта
// k – индекс результирующего вектора из aRes
{
   int = 0; int = 0; int = 0;
      // цикл – пока есть элементы в обоих массивах
   while (i<aVect.FCount && j<FCount)
     {
            // пока элементы из aVect меньше текущего
        while (aVect.FData[i]<=FData[j]&& i<aVect.FCount)
          { aRes.FData[k++] = aVect.FData[i]; i++;}

            // пока элементы из текущего меньше из aVect
        while (i<aVect.FCount && j<FCount &&
                                    aVect.FData[i]>FData[j])
          { aRes.FData[k++] = FData[j]; j++;}
     }
            // дописывается остаток из aVect
    while (i<aVect.FCount)
      { aRes.FData[k++] = aVect.FData[i]; i++;}
            // дописывается остаток из текущего
    while (j<FCount)
      { aRes.FData[k++] = FData[j]; j++;}
}
```



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. Проверить, является ли заданный вектор единичным.
   Пояснение: вектор *e* называется единичным, если одна его координата равна 1, а остальные равны 0.
2. Дана последовательность чисел. Определить, является ли эта последовательность периодической (то есть может ли она быть получена повторением некоторой своей начальной части). Из всех периодов указать наименьший.
3. Найти 10 натуральных чисел, каждое из которых можно представить как сумму квадратов двух неотрицательных чисел, представленную двумя способами, то есть $M = I^2 + J^2 = K^2 + L^2$, где *I*, *J*, *K*, *L* – разные числа.
4. В обращении имеются монеты: 5-, 10-, 20-, 50-копеечного достоинства. Написать программу, определяющую сдачу наименьшим количеством монет.
5. Задается список упорядоченных элементов. Программа должна расположить в их в случайном порядке.

Рекомендации: решение можно реализовать с помощью алгоритма, немного похожего на сортировку выбором. Для каждого номера из списка алгоритма случайным образом выбирается элемент, при этом рассматриваются только элементы из еще не помещенных на свое место. Затем выбранный элемент меняется местами с элементом, стоящим в данной позиции.

Разработка алгоритмов, предназначенных для обработки битовых строк 5

Представление информации в виде битовой строки находит широкое применение при создании алгоритмов решения различных задач. В качестве примеров можно привести алгоритмы, связанные с упаковкой/сжатием данных, с построением и использованием дескрипторов, генетические алгоритмы.

Сжатие данных имеет смысл, когда необходимо избавиться от избыточности текста/информации с целью экономии памяти компьютера. Дескрипторы представляют собой некоторые лексические единицы (слова), которые используются для компактного описания характеристик различных объектов. И в том, и другом случаях встают задачи, связанные с кодированием/декодированием информации на уровне битового доступа к машинным словам.



Машинное слово интерпретируется как вектор, состоящий из отдельных битов или разрядов. Биты нумеруются слева направо, причем крайний слева (младший бит) в разрядной сетке имеет номер 0. Длины байта, слова и двойного слова равны 8, 16, 32 и 64 бит, соответственно. Разрядная сетка может иметь длины: 8, 16, 32 и 64.

5.1. Разработка классов «Биты»

Общая постановка задачи

Базовая задача. Требуется разработать класс, предназначенный для обработки битовых строк, способный решать следующие задачи:

- выделять отдельный бит в машинном слове;
- выдавать содержимое разрядной сетки слова;
- строить маску;
- вычислять количество единичных битов в слове.

Решение

В общем случае, при постановке задач «на биты», количество используемых битов может быть не ограничено рамками одного слова. Поэтому при разработке классов целесообразно считать, что битовой обработке подвергается массив слов. Следовательно, за базовый класс можно принять класс, например, «Одномерный массив» (`TVectorEx`), разработанный ранее. А разработку битового класса вести, как разработку потомка от класса `TVectorEx`.

Для реализации алгоритмов, связанных с обработкой битовых участков разрядной сетки с позиции внутреннего представления данных, широкое применение нашли, так называемые, «маски». Маски очень удобны при работе с *отдельными участками* разрядной сетки числа.



Маска строится таким образом, чтобы в ней разрядам, соответствующим *выделяемым* битам, ставились в соответствие 1, а битам, свободным от выделения, ставились в соответствие 0. Тогда побитовое умножение (операция &) разрядной сетки числа на маску приведет к обнуле-

нию тех битов, которые соответствуют нулям маски, и оставит без изменения те биты, которые соответствуют единичным значениям маски. Предположим, что требуется выделить в 8-ми разрядном числе 01011010 биты с 4-го по 6-й. Для выделения битов потребуется маска вида 00011100. Тогда выделение самих битов можно произвести с помощью операции побитового умножения: 01011010 & 00011100. В результате примененной операции в числе 01011010 будут оставлены без изменения биты с 4-го по 6-й, а остальные биты будут равны 0: 00011000.

Существуют различные подходы к построению маски. Рассмотрим два способа.

1) Можно строить маску, пользуясь знанием того, каким 16-ричным числом (десятичным) представляется требуемое количество единиц в битовой сетке: $1_{10} = 1_{16} = 1_2$, $3_{10} = 3_{16} = 11_2$, …, $15_{10} = F_{16} = 1111_2$.
2) Маску можно построить с помощью накопления суммы сдвигаемых единиц по итерационной формуле. На каждом шаге итерации формируется очередное слагаемое, которое представляет собой единицу (1), передвинутую влево по разрядной сетке. Позиция, на которую перемещается 1, зависит от номера итерации. Перемещение единиц в нужную позицию битовой сетки можно выполнить с помощью обычного сдвига (операция «<<»);

Пример построения маски из четырех 1, расположенных в конце разрядной сетки, приведен на рис. 5.1. В первой колонке указан номер итерации. Во второй колонке «скользящая» 1 получается умножением 1 на число 2. В третьей колонке такая же скользящая 1 получается с использованием операции сдвига (операции «<<»).

Номер итерации *i*	`c = c*2`	`c = 1<<i`	`S = S+c`
0	00000001	00000001	00000001
1	00000010	00000010	00000011
2	00000100	00000100	00000111
3	00001000	00001000	00001111

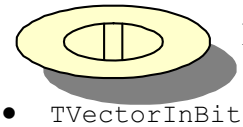
Рис. 5.1. Построение маски S из четырех 1, расположенных в конце разрядной сетки

Интерфейс класса «Биты» (назовем его `TVectorInBit`) может быть представлен в виде, показанном в листинге 5.1.

Листинг 5.1. Интерфейс класса «Биты» – «`Работа с вектором на битовом уровне`»

```
class TVectorInBit: public TVectorEx
{
  protected:
  public:
   TVectorInBit ();  // конструктор по умолчанию
   TVectorInBit (const TVectorTask &aVect);  // конструктор
   TVectorInBit(char *File, char *aInitNameObj); // конструктор
   ~TVectorInBit  ();
   void PutInversInGrid(TStringGrid *aSGrid);
```

```
    void SaveGrid (TStringGrid *aSGrid); // запись вектора
    unsigned int DoMask2(int n1, int n2); // создать маску
    unsigned int DoMask(int n1, int n2);  // создать маску
    void DoTest0(unsigned t1, unsigned& t2); //
    int bitbit(unsigned int x,int i);  // вернуть бит
    int bit(unsigned long int x,int i);// вернуть бит
    AnsiString word(unsigned long d);  // вернуть сетку
    int bit(unsigned  int x,int i); // вернуть бит из
    AnsiString word(unsigned int d); // вернуть сетку
    int invers(unsigned int a, int n1, int n2); // инвертировать
};
```



НАЗНАЧЕНИЕ МЕТОДОВ КЛАССА `TVectorInBit`:

- `TVectorInBit();` – конструктор по умолчанию. Его также называют пустым конструктором. Он задает «нулевые» значения полей, что обозначает, что объявляется массив, в котором не содержится ни одного элемента;
- `TVectorInBit (char *File, char *aInitNameObj);` – в этом конструкторе предусмотрено два параметра. Через параметр `File` передается имя файла, в котором располагаются исходные данные. Второй параметр `aInitNameObj` используется для передачи имени объекта;
- `TVectorInBit (const TVectorInBit &aVect);` – конструктор копирования;
- `~TVectorInBit ();` – деструктор. Разрушает объект – высвобождает динамическую память, отведенную конструктором;
- `void PutInversInGrid (TStringGrid *aSGrid);` – запись в инверсном порядке вектора в сетку `aSGrid`;
- `void SaveGrid (TStringGrid *aSGrid);` – запись вектора в сетку `aSGrid`;
- `unsigned int DoMask2 (int n1, int n2);` – построить маску с единичными битами в позициях от `n1` до `n2`;
- `unsigned int DoMask (int n1, int n2);` – построить маску с единичными битами в позициях разрядной сетки от `n1` до `n2`;
- `void DoTest0 (unsigned t1, unsigned& t2);` – в `t2` переписать `t1` с обнуленным крайним единичным битом;
- `int bitbit (unsigned int x, int i);` – вернуть значение `i`-бита в числе `x`;
- `int bit (unsigned long int x, int i);` – вернуть значение `i`-бита в числе `x`;
- `AnsiString word (unsigned long d);` – вернуть разрядную сетку слова `d`;
- `AnsiString word (unsigned int d);` – вернуть разрядную сетку слова `d`;
- `int invers (unsigned int a, int n1, int n2);` – инвертировать биты, начиная с номера `n1` до номера `n2`.

Построение маски с использованием операции умножения

Функции `DoMask` и `DoMask2`, предназначенные для построения маски, имеют два входных параметра: `n1` и `n2`. Эти параметры задают начальное и конечное значения номеров единичных битов, составляющих маску. Сама маска возвращается через заголовок.

Функции `DoMask` и `DoMask2` отличаются друг от друга тем, что первая функция для смещения положения единицы в очередном слагаемом использует операцию умножения на 2, а вторая – операцию «<<».

Алгоритм получения маски с единичными битами в разрядах машинного слова, начиная с разряда `n1` по `n2`, сводится к следующим действиям:

- определяется количество 1 (единиц) в маске (переменная `k`);
- строится маска (переменная `cs`), содержащая нужное количество единиц, которые подтянуты (расположены) к концу разрядной сетки;
- определяется величина смещения (переменная `k1`), которая говорит о том, на сколько разрядов нужно подвинуть влево группу единичных битов маски, чтобы они заняли требуемое положение в разрядной сетке числа;
- возвращается подвинутое значение маски как результат функции.

В алгоритм введена также вспомогательная переменная `c`, содержащая 1 в текущем разряде, значение которого переопределяется в алгоритме. Алгоритм `DoMask2` отличается от алгоритма `DoMask` только одной операцией: умножение заменяется сдвигом. Коды процедур построения маски приведены в листингах 5.1 и 5.2.

Листинг 5.1. Описание метода построения маски в 16-битовой разрядной сетке первым способом – умножением на 2

```
unsigned int DoMask(int n1, int n2)
{
  int c = 1,   // Начальное значение слагаемого
       cs = 0;  // Подготовка суммы
   int k = n2-n1+1;
   int k1 = 16-n2;
     // настройка маски
   for( int i = 0; i<k; i++) // цикл по числу 1
     {
       cs = cs +c;
       c = c*2;
     }
  return (cs<<k1);
}
```

Листинг 5.2. Описание метода построения маски в 16-битовой разрядной сетке вторым способом – использованием операции сдвига

```
unsigned int DoMask2(int n1, int n2)
{
   int c = 1, // Начальное значение слагаемого
       cs = 0; //Подготовка суммы
           // Определение числа 1, необходимых для маски
   int k = n2-n1+1;
     // Определение величины для смещения последовательности
   int k1 = 16-n2;
   for( int i = 0; i<k; i++)// цикл по числу 1
     {
       cs = cs + (1<<i);
     }
  return (cs<<k1);
}
```

Получение значения бита с заданным номером разрядной сетки

Функции `bitbit` и `bit` имеют два входных параметра: число `x` и `i` – номер выделяемого бита. Функция возвращает либо 0, либо 1 в зависимости от значения разряда с номером `i` в числе `x`.

В качестве примера предположим, что в 8-битовом числе 01011011 нужно определить значение 5-го бита. Функции `bitbit` и `bit` должны вернуть в качестве результата значение 1.

Реализуют два разных варианта решения задачи.

а) Вариант 1, используемый в функции `bitbit`.

Определяем, сколько битов расположено справа от `i` ? (8 – `i` = 3). Сдвигаем этот бит и, соответственно, все биты перед ним вправо на 3 бита (в результате требуемый бит в разрядном слове займет крайнее правое место). Сдвигаем этот бит влево на 7 битов.

Эти три шага приведут к следующим преобразованиям:
01011011 => 00001011 => 10000000.

В результате этих преобразований обнуляются все биты за исключением требуемого бита. Правда, при этом он расположился в самой левой позиции, но это не играет роли в установлении факта, является ли он 0 или 1. Алгоритм получения значения бита с заданным номером в разрядной сетке можно описать следующими шагами:

- входное число `x` копируется во вспомогательную переменную `f`, которая будет изменена для решения задачи;
- выполняется операция сдвига `f` вправо, в результате которой выделяемый бит займет в `f` правую крайнюю позицию;
- выполняется операция сдвига `f` влево на 31 бит, в результате бит займет в *f* левую крайнюю позицию, а все, что правее будет обнулено;
- проверяется значение `f` на ноль. Если оно равно нулю, то – возврат 0 (разряд равен `0`), иначе – `1`.

Описание процедуры `bitbit` представлено в листинге 5.3.

Листинг 5.3. Вернуть бит по его номеру

```
int bitbit (unsigned int x,int i)
// Функция bitbit возвращает значение i-бита в числе x с
// разрядной сеткой из 32 битов
{
  unsigned int f;  // вспомогательная переменная
// копируется исходное число x во вспомогательную переменную
  f = x;
// сдвиг i-бита вправо, а потом влево до конца
  f >>=32-i;  f <<=31;
// если i-бит равен 0, то возврат в качестве результата 0,
  if(f==0)return 0;
  else return 1;    // иначе возврат 1
}
```

б) Вариант 2, используемый в функции `bit`.

Известно, что битовая операция умножения `&` над разрядами выполняется следующим образом: `1&1 => 1`, `1&0 => 0`.

Так для того, чтобы получить значение 5-го бита в слове, можно это слово сложить побитово с маской, в которой все биты равны 0, за исключением 5-го бита, который равен 1:

```
01011011 & 00001000 => 00001000.
```

Если бы 5-й бит был 0, то в результате побитового умножения был бы получен 0.

В алгоритме строится маска с одной единственной единицей, предназначенная для выделения i-бита в заданном числе x. Маска будет содержать одну единицу (1), помещенную в позицию, указанную параметром i. Если результат побитового умножения маски на число равен 0, то возвращается 0, иначе – 1 (единица). Алгоритм получения значения бита с заданным номером в разрядной сетке можно описать следующими шагами:

- задается начальное значение маски (1 – в конце разрядной сетки);
- маска корректируется, то есть 1, содержащаяся в маске, сдвигается на требуемую позицию разрядной сетки;
- если результат побитового умножения маски на число равен 0, то возвращается 0, иначе возвращается 1.

Описание процедуры приведено в листинге 5.4.

Листинг 5.4. Описание функции bit, предназначенной для выделения значения бита по его порядковому номеру

```
int bit(unsigned int x,int i)
// Функция bit возвращает значение i-бита в числе
{
  unsigned long int f=1;  // маска устанавливается в 1
            // 1 устанавливается в i-позицию
  f <<= 32-i;
         // если результат побитового умножения
  if((x&f)==0)return 0;
     // маски и числа 0, то возврат 0
  else return 1;   // иначе возврат 1
}
```

Получение/распечатка содержимого разрядной сетки числа

Функция word имеет один входной параметр – беззнаковую целую переменную d. Возвращает через заголовок значение типа AnsiString, представляющее содержимое разрядной сетки числа.

В алгоритме используется самый простой способ решения – воспользоваться услугами функции bit (или bitbit), предназначенной для выделения значения бита по его порядковому номеру. В таком случае решение задачи будет сведено к циклу, в котором последовательно определяются значения битов со сборкой их в единое целое в виде строки. Алгоритм получения разрядной сетки числа можно описать таким образом:

1) инициализируется вспомогательная переменная Ptr, предназначенная для сборки разрядных значений числа;
2) выполняется цикл (по числу разрядов):
   а) если значение bit(d,i)==0, то к Ptr добавляется 0;
   б) если значение bit(d,i)==1, то к Ptr добавляется 1;
3) возвращается значение Ptr как результат работы функции.

Описание функции word представлено в листинге 5.5.

Листинг 5.5. Описание функции word, возвращающей битовое представление числа

```
AnsiString TVectorInBit::word(unsigned int d)
// Функция word возвращает значение всех битов числа d
{
      // Определяется размер битовой сетки
```

```
  int Size = sizeof(unsigned int )*8;
      // Вспомогательная переменная для сборки
  AnsiString Ptr = "";
            // цикл по перебору номеров битов
  for(int i = 1; i<=Size;i++)
            // если значение бита 0, то добавляется 0
    if(bit(d,i)==0)Ptr = Ptr+"0";
            // иначе добавляется 1
    else  Ptr = Ptr+"1";
  return Ptr; // возврат строки
}
```

Подсчет количества единичных битов в разрядной сетке числа

Функция `DoTest` имеет один входной параметр `t1` – переменная, в разрядной сетке которой нужно подсчитать количество единичных битов. Подсчитанное количество возвращается в качестве результата работы функции.

Для решения задачи используется формула `x & (x - 1)`, которая обнуляет крайний справа единичный бит (например, применение этой операции к числу `x = 01011000` приводит к значению `01010000)`. Алгоритм подсчета количества единичных битов описывается шагами:

1) выполняется проверка исходного числа `t1` на 0:
   а) если `t1` равно 0, то число единичных битов равно 0 и функция возвращает 0;
   б) выход;
2) счетчик числа (k) единичных битов устанавливается в 1;
3) выполняется цикл (пока есть единичные биты в соответствии с применением формулы, описанной выше), в котором счетчик битов увеличивается на 1;
4) возвращается значение счетчика как результат работы функции.

Следует заметить, что *изменение входного параметра* `t1` внутри функции не отразится на значении фактического параметра, так как параметр передается «по значению». Код алгоритма `DoTest` представлен в листинге 5.6.

Листинг 5.6. Описание функции DoTest для определения количества единичных битов в разрядной сетке

```
int TVectorInBit::DoTest(unsigned t1)
{
  if (t1==0) return 0;
  int k = 1; // счетчик битов
  while ( t1 = t1&(t1-1)) // пока есть единичные биты
    k++; // увеличивается счетчик
  return k;
}
```

Инвертирование заданной подпоследовательности битов внутри одной ячейки

Функция `invers` имеет 4 входных параметра. Параметр `a` определяет переменную, внутри которого требуется произвести инвертирование битовой подпоследовательности. Параметры `n1`, `n2` задают номера битов, определяющих начало и конец подпоследовательности для инвертирования. Параметр `size` определяет длину разрядной сетки (размер ячейки, отведенной под элемент `a`, в битах). Функция через свой заголовок возвращает преобразованное инвертированием значение элемента `a`.

Алгоритм такой процедуры может быть следующим: из разрядной сетки элемента `a` извлекаются биты один за другим, начиная с последнего бита по порядку. Извлеченные биты перемещаются (складываются) в новую ячейку также в направлении от конца к началу. При перемещении биты либо инвертируются, либо нет, в зависимости от того, подпадают ли они в интервал, предназначенный для инвертирования.

Пример

```
a = 11.011000.11101101
```

Знаки «.» в разрядную сетку не включаются. Они использованы для наглядности.
Пусть границы интервала (начиная от 1) для инвертирования: `n1=3`; `n2=8`.
Пересчитываются границы: `n1=14`; `n2=8`. В пересчитанном виде `n1`, `n2` будут определять границы перехода участков от процесса инвертирования к не инвертированию и наоборот (если начинать от конца).
Первая часть – без изменения бит за битом, начиная с последнего бита:

```
ps => 00000000.11101101
```

Вторая – с инверсированными битами:

```
ps => 00.100111.11101101
```

Третья часть – без изменения бит за битом:

```
ps => 11.001110.1101101
```

В алгоритм вводятся вспомогательные переменные: `ps` – для накопления битов, `c` – для хранения маски с единицей в соответствующей позиции. Каждая из частей реализуется с помощью цикла, внутри которого «подстраивается маска», с помощью которой либо берется очередной бит без изменения его значения, либо в инвертированном виде. Описание функции `invers` представлено в листинге 5.7.

Листинг 5.7. Описание функции инвертирования заданного интервала битов в битовой сетке

```
int invers(unsigned int a, int n1, int n2, int size)
{
  unsigned int c;
  int ps, i;
  c = 1; ps = 0;
  n1 = size - n1 +1;
  n2 = size - n2+1;
  for(i=1; i<n2; i++)
    {
      ps = ps + (a&c);
      c = c*2;
    }
   for(i=n2; i<=n1; i++)
    {
      ps = ps + ((a&c)^c);
      c = c*2;
    }
   for(i=n1+1; i<=size; i++)
    {
      ps = ps + (a&c);
      c = c*2;
    }
   return ps;
}
```

5.2. Решение задач с использованием класса «Биты»

Задача 5.1. Произвести кроссинговер двух хромосом, а именно: из двух хромосом, каждая из которых содержит по 32 гена, получить две новые хромосомы рекомбинацией или обменом двух участков генов (с номеров *k1* по *k2* и с номеров *k3* по *k4*).

Решение

Хромосома – опорное понятие при разработке генетических алгоритмов. Структура данных генетического алгоритма представляется набором хромосом. Хромосома, как правило, отображается в виде битовой строки, так что термин «строка» часто заменяет понятие «хромосома». Каждая хромосома (строка), таким образом, является последовательным объединением ряда подкомпонентов (битов), которые называются генами.

Гены расположены в различных позициях (или локусах) хромосомы и принимают значения, называемые аллелями – это биологическая терминология. В представлении хромосомы бинарной строкой, *ген* является битом этой строки, *локус* – есть позиция бита в строке, а *аллель* – это значение гена, 0 или 1. Биологический термин «генотип» относится к полной генетической модели особи и соответствует структуре в генетическом алгоритме. Термин «фенотип» относится к внешним наблюдаемым признакам и соответствует вектору в пространстве параметров задачи. С хромосомами выполняют множество операций, таких как мутация, кроссинговер.

Под *мутацией* понимается преобразование хромосомы, случайно изменяющее один или несколько генов. Термин кроссинговер (скрещивание или рекомбинация) обозначает порождение из двух хромосом двух новых путем обмена генами.

Алгоритм решения задачи продемонстрируем на следующем примере.

Даны две хромосомы:

`x1={0110.1010.1011.1111.1010.1011.0000.0100} и`

`x2={1111.1101.0010.1000.1110.1111.1110.1011},`

`k1 = 3, k2 = 12, k3 = 24, k4 = 29.`

Пусть с этих участков гены переносятся из хромосомы `x2` в хромосому `x1`. В результате такого кроссинговера должна получиться хромосома вида:

`x3 = {0111.1101.0010.1111.1010.1011.1110.1100}.`

Аналогично получается вторая хромосома:

`x4 = {1110.1010.1011.1000.1110.1111.0000.0011}.`

Шаг 1. Строится «маска» для выделения генов – битов, которые будут переданы в новую хромосому из хромосомы `x1` в виде:

`Mask1 = {1100.0000.0000.1111.1111.1110.0000.0111}.`

Шаг 2. С помощью операции побитового умножения `(t3 = x1 & Mask1)` выделяются передаваемые из `x1` биты. В результате

`t3 = {0100.0000.0000.1111.1010.1010.0000.0100}.`

А те биты, которые будут заменяться, обнулятся.

Шаг 3. Строится «маска» для выделения генов – битов, которые будут переданы в новую хромосому из хромосомы `x2` в виде:

`Mask2 = {0011.1111.1111.0000.0000.0001.1111.1000}.`

Шаг 4. С помощью операции побитового умножения `(t2 = x2 & Mask1)` выделяются передаваемые биты. В результате

`t2 = {0011.1101.0010.0000.0000.0001.1110.1000}.`

Шаг 5. Складываются побитово `t1` и `t2: x3 = t1|t2`.

Шаг 6. Повторяются шаги со 2-го по 5-й, но гены переносятся из `x1` в хромосому `x2`.

Предложенный алгоритм реализуется в процедуре-обработчике события `OnClick` для кнопки «Рекомбинация», который представлен в листинге 5.8.

Рекомбинация битов

В теле обработчика события сразу формируется выходная сетка `StringGrid2`, как на рис. 5.2.

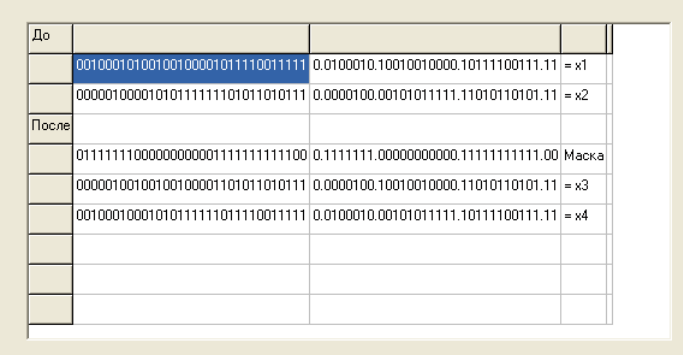


Рис. 5.2. Результат рекомбинации битов

Функция – обработчик события `OnClick` кнопки «Рекомбинация» предназначена для кроссинговера хромосом `x1` и `x2` рекомбинацией битов, расположенных на двух участках: (`k1...k2`) и (`k3...k4`). Эти значения задаются внутри обработчика. Здесь `x3` и `x4` – хромосомы, полученные в результате рекомбинации. В функции, в свою очередь, вызывается функция `PointIns`, которая «отмечает» точками те участки в хромосоме, которые подлежат рекомбинации. Это совершается для удобства пользователя.

Примечание

В обработчике событий считается определенным экземпляр `BitArr` битового класса `TVectorInBit`. Более того, будем считать, что известны и заданы значения базового класса `TVector`. В качестве значений хромосом `x1` и `x2` в предложенном обработчике события берутся два первых по порядку элемента из вектора. При этом для получения значения `x1` используется метод базового класса `GetValue`, а для получения `x2` – оператор индексации.

Листинг 5.8. Описание метода, предназначенного для инвертирования заданного интервала битов в битовой сетке

```
void __fastcall TfoMain::Button5Click(TObject *Sender)
{
  StringGrid2->Cells[0][0] = "До";
  StringGrid2->Cells[0][3] = "После";
  StringGrid2->Cells[3][1] = "= x1";
  StringGrid2->Cells[3][2] = "= x2";
  StringGrid2->Cells[3][4] = "Маска";
  StringGrid2->Cells[3][5] = "= x3";
  StringGrid2->Cells[3][6] = "= x4";
  unsigned int x1, x2, x3, x4, Mask1, Mask2, Mask;
  int k1 = 2, k2 = 8, k3 = 20, k4 = 30;
  x1 = BitArr->GetValue(0);  // в x1 взять 0-й элемент вектора
  x2 = (*BitArr)[1]; // в x2 взять 1-й элемент вектора
      // в Ptr записать разрядную сетку x1
```

```
  AnsiString Ptr = BitArr->word(x1);
      // скопировать значение Ptr в ячейку сетки StringGrid2
  StringGrid2->Cells[1][1] = Ptr;
      // Ptr занести в строку Str
  char *Str = Ptr.c_str();
  int n = strlen(Str); // определить n - длину строки
      // создать строку из n+5 символов
  char* Str1 = new char[n+5];
// скопировать Str в Str1 с добавлением разделительных точек
  PointIns(Str,Str1,n,k1,k2,k3,k4);
// скопировать значение Str1 в ячейку сетки StringGrid2
  StringGrid2->Cells[2][1] = (AnsiString) Str1;
      // в Ptr записать разрядную сетку x2
  Ptr = BitArr->word(x2);
// скопировать значение Ptr в ячейку сетки StringGrid2
  StringGrid2->Cells[1][2] = Ptr;
      // Ptr занести в строку StrF
  char *StrF = Ptr.c_str();
      // создать строку из n+5 символов
  char* Str1F = new char[n+5];
           // скопировать Str1F в StrF с добавлением
           // разделительных точек
  PointIns (StrF,Str1F,n,k1,k2,k3,k4);
// скопировать значение StrF в ячейку сетки StringGrid2
  StringGrid2->Cells[2][2] = (AnsiString) Str1F;
// создать маску с единичными битами в позициях k1-k2
  Mask1 = BitArr->DoMask(k1,k2);
// создать маску с единичными битами в позициях k3-k4
  Mask2 = BitArr->DoMask(k3,k4);
// побитово сложить маски
  Mask = Mask1|Mask2;
// получить разрядную сетку сформированной маски
  Ptr = BitArr->word(Mask);
// поместить в сетку StringGrid2
  StringGrid2->Cells[1][4] = Ptr;
  Str = Ptr.c_str();  // преобразовать в строку
      // дополнить разделительными точками
  PointIns(Str,Str1F,n,k1,k2,k3,k4);
      // поместить в сетку StringGrid2
  StringGrid2->Cells[2][4] = (AnsiString)Str1F;
      // выполнить кроссинговер и получить x3
  x3 = x1&(~Mask)|x2&Mask;
      // получить разрядную сетку
  Ptr = BitArr->word(x3);
      // записать в ячейку StringGrid2
  StringGrid2->Cells[1][5] = Ptr;
  Str = Ptr.c_str();  // преобразовать в строку
      // дополнить разделительными точками
  PointIns (Str,Str1F,n,k1,k2,k3,k4);
      // поместить в сетку StringGrid2
  StringGrid2->Cells[2][5] = (AnsiString)Str1F;
      // выполнить кроссинговер и получить x3
  x4 = x2&(~Mask)|x1&Mask;
      // получить разрядную сетку
  Ptr = BitArr->word(x4);
      // поместить в сетку StringGrid2
  StringGrid2->Cells[1][6] = Ptr;
  Str = Ptr.c_str();  // преобразовать в строку
```

```
      // дополнить разделительными точками
  PointIns (Str,Str1F,n,k1,k2,k3,k4);
      // поместить в сетку StringGrid2
  StringGrid2->Cells[2][6] = (AnsiString)Str1F;
}
```

Добавление к строке разделительных точек

Функция `PointIns` предназначена для добавления точек к строке битов. Она имеет 7 параметров: `Str` – входная строка; `Str1` – выходная строка; `n` – целая переменная которая указывает длину строк; `k1`, `k2`, `k3`, `k4` – целые переменные, которые указывают местоположения точек, определяющих участки рекомбинации. Алгоритм преобразования строки сводится к циклу по перебору символов исходной строки, в котором проверяется номер символа на совпадение с местоположением разделительной «.» и формированием символов новой строки с учетом добавления «.». После цикла добавляется символ конца строки «\0». Описание функции `PointIns`, предназначенной для добавления к строке разделительных точек, представлено в листинге 5.9.

Листинг 5.9. Описание метода, предназначенного для добавления к строке разделительных точек

```
void PointIns(char* Str, char* Str1, int n,
                  int k1, int k2, int k3, int k4)
{
  int j =0;
  for(int i=0;i<n;i++)
    {
      if(i==(k1-1)||i==k2||i==(k3-1)||i==k4)Str1[j++]='.';
      Str1[j++]=Str[i];
    };
  Str1[j] = '\0';
}
```

Задача 5.2. Память массива переменных типа `unsigned int` рассматривается как последовательность битов. Реализовать алгоритм инвертирования подпоследовательности битов, задаваемой порядковым номером `p` и размером `n`. Допускается, что подпоследовательность битов может накладываться на границу переменных `unsigned int`.

Решение

В соответствии с условием задачи каждый элемент массива рассматривается с позиции его внутреннего представления. На рис. 5.3 приведен битовый расклад элементов массива. Поскольку тип элемента массива `unsigned int`, речь будет идти только о положительных значениях.

a_0	a_1		a_{n1}		a_{n2}		a_{n-1}
32 бита	32 бита		32 бита		32 бита		32 бита
	64 бита		$32\times(n1+1)$ битов		$32\times(n2+1)$ битов		$32\times n$ битов
Всего $32\times n$ битов							

Рис. 5.3. Битовый расклад элементов массива

Разрабатываемый алгоритм требует доступа к отдельным битам массива данных. Получить такой доступ возможно только на уровне отдельных элементов массива. Поэтому первым шагом к решению задачи будет являться задача определения тех элементов в массиве, значения которых требуют доступа с целью преобразования битов в соответствии с заданным условием. А второй – в просмотре элементов с целью их инверсии.

Например, если массив состоит из 10 элементов (что соответствует 320 битам), а $p = 123$ и $n = 105$, определить, какие элементы массива и какие биты подлежат преобразованию, можно по следующей схеме.

1) Определить первый элемент, начиная с которого требуется выполнять инверсию. Для этого определим, сколько полных элементов (по 32 бита) содержатся в $p = 123$. Число полных элементов – это то количество элементов от начала массива, которые сохраняются неизменными. Это число можно получить, выполнив целочисленное деление $\frac{123}{32} = 3$ *целых* $(27 - остаток)$. Вывод: преобразование нужно начинать с 4-го по порядку элемента массива, то есть с a_3, так как индексация начинается от 0.
2) Определить номер начального бита для преобразования внутри этого элемента. Для этого воспользуемся величиной, равной остатку от деления числа *p* на 32. Вывод: бит, начиная с которого следует производить инвертирование, с номером 27. Заметим, что если остаток равен 0, то бит, начиная с которого следует производить инвертирование, с номером 0.
3) Определить номер последнего элемента массива, в котором будет производиться инверсия. Для этого определим сначала номер последнего инвертируемого бита сложением: $p + n = 123 + 105 - 1 = 227$. А затем повторим пункт 1 применительно к полученной сумме: $\frac{227}{32} = 7$ *целых* $(3 - остаток)$. Вывод: преобразование нужно завершать в 7-м по порядку элементе массива, то есть в a_7, так как индексация начинается от 0.
4) Определить номер последнего бита в 7-м элементе, на котором останавливается инверсия. Для этого воспользуемся величиной остатка от деления – числом 3. Вывод: 3 – номер бита, последнего для инвертирования. Заметим, что если остаток равен 0, то в последнем элементе массива нет битов для инвертирования.

Эти действия можно положить в основу алгоритма функции, предназначенной для решения поставленной задачи.

Алгоритм инвертирования последовательности битов в массиве

Решение задачи инвертирования может осуществляться по следующему алгоритму:

1) Вычисляется номер последнего инвертируемого бита `nFinish = iStart+iLen-1`.
2) Вычисляется индекс первого изменяемого элемента массива `k1 = (iStart-1) /Size`.
3) Вычисляется индекс последнего изменяемого элемента массива `k2 = (nFinish-1)/Size`.

4) Вычисляется номер первого инвертируемого бита в первом изменяемом элементе массива `nLeft = iStart-k1*Size.`
5) Вычисляется номер последнего инвертируемого бита в последнем изменяемом элементе массива `nRight = nFinish-k2*Size.`
6) Выполняется цикл по номерам изменяемых элементов, в котором
   а) для каждого изменяемого элемента сначала рассчитываются значения начального и конечного битов, определяющие границы инвертирования внутри элемента;
   б) вызывается процедура инвертирования – `invers` (описанная в классе, которая будет инвертировать биты внутри взятого элемента массива).

Для реализации алгоритма создается функция с именем `DoInvers`. Эта функция имеет 5 параметров: `fa` – входной массив, в котором нужно инвертировать битовую подпоследовательность; `fb` – выходной массив с инвертированной подпоследовательностью битов; `iStart` – номер начального бита инвертирования; `iLen` – количество инвертируемых битов; `Size` – размер памяти в битах, отводимой под отдельный элемент массива `fa`. Код функции `DoInvers`, предназначенной для инвертирования битов внутри массива, представлен в листинге 5.10.

Листинг 5.10. Описание функции `DoInvers`, предназначенной для инвертирования битов внутри массива

```
void DoInvers(int* fa, int*fb, int iStart, int iLen, int Size)
{
   int n1, n2;
            // Последний инвертируемый бит
   int nFinish = iStart+iLen-1;
   int k1 = (iStart -1)/Size;  // Первое изменяемое слово
            // Нач. бит в первом изменяемом слове
   int nLeft = iStart - k1*Size;
   int k2 = (nFinish - 1)/Size; // Последнее изменяемое слово

      // Конечный бит в последнем изменяемом слове
   int nRight = nFinish - k2*Size;
   for(int i = k1; i<=k2; i++)
   {
      n1 =1; n2 = Size; // границы для инверсии внутри слова
            // переопределение левой границы в первом слове
      if (i==k1) n1=nLeft;
            // переопределение правой границы в последнем слове
      if (i==k2) n2=nRight;
      fb[i] = invers((unsigned int)fa[i],n1,n2,Size);
   }
}
```

Задача 5.3. Вычислить значение V^K. При вычислении использовать битовое представление показателя степени *K*.

Решение

Идея вычисления A^n заключается в следующем: если степень $n = 2 \cdot k$ (четная), то $A^n = (A^2)^k$. Если же $n = 2 \cdot k + 1$ (нечетная), то $A^n = (A^2)^k \cdot A$. Степень с показателем *k* вычисляется из этих же соображений. В соответствии с этим, надо разделить *n* на 2 с остатком: *n = 2k + 1*. Затем то же проделать с *k* и так далее. Эти дей-

ствия, как известно, приводят к построению двоичной записи числа n. Алгоритм, основанный на этой идее, состоит в том, что последовательно вычисляются: $A^{\alpha 0}, A^{\alpha 1 \alpha 0}, A^{\alpha 2 \alpha 1 \alpha 0} \ldots$ где $\alpha_l \alpha_{l-1} \ldots \alpha_2 \alpha_1 \alpha_0$ – двоичная запись числа *n*. Для реализации этого алгоритма вычисляется, цифра за цифрой, двоичная запись числа и параллельно степень за степенью $A^{2^0}, A^{2^1}, A^{2^2} \ldots$ При этом подсчитывается произведение тех степеней, для которых соответствующая цифра в двоичном представлении равна 1. Например, запись 9 в двоичной системе есть 1001 (9 = 1·8 + 0·4 + 0·2 + 1); для вычисления A^9 достаточно найти: A^2, A^4, A^8 (3 умножения), а затем определить $A \cdot A^8$ (одно умножение).

Преимущество этого алгоритма, по сравнению с обычным умножением, состоит в том, что обычный алгоритм требует числа умножений, растущего как линейная функция от n, а здесь число умножений, грубо говоря, пропорционально количеству цифр двоичного числа или логарифму n.

Работу предлагаемого алгоритма продемонстрируем на следующем примере: пусть требуется вычислить 5^{83}.

1) Анализируем битовое представление степени числа 5. Целое число 83 представляется (в 16-разрядной сетке):

0000.0000.0101.0011.

Определяем номер первого по порядку единичного бита. Это бит с номером 10.

Вывод: при накоплении степени числа достаточно произвести (16 – 10 + 1) операций умножения, то есть всего 7 операций.

2) Готовим ячейку для накопления произведения (P = 1). Готовим ячейку для очередного сомножителя (q = 5), значения которого будут пересчитываться от итерации к итерации.

3) Делаем цикл на 7 итераций, перебирая разряды с 10 по 16, в направлении от 16 бита к 10-му. В теле цикла формируем очередное значение сомножителя умножением на самого себя. И накапливаем произведение, то есть умножаем произведение на сомножитель, причем только в случае единичного разряда.

Последовательность вычислений представлена в таблице 5.1. А функция `Multiply` (возведения в степень) представлена в листинге 5.11. Следует отметить, что в теле функции, в соответствии с алгоритмом, вызывается функция `bit`, описанная в классе и предназначенная для определения значения заданного бита в разрядной сетке числа.

Листинг 5.11. Описание функции возведения в степень целого числа (v^K) на основании анализа битовой сетки степени числа

```
float TVectorInBit::Multiply(int v, int k)
{
  int Size = sizeof(int)*8; // длина разрядной сетки
  float q = v,
  P = 1;  // Подготовка ячейки для накопления произведения
  int i,
    int t1 = 0;  // начальное значение t1

     // просматриваются биты, начиная с первого,
      // пока не встретится первый по порядку единичный бит
  for (i = 1; t1 == 0 && i<=Size;i++)
```

```
// в t1 заносится значение очередного по порядку бита
    t1 = bit ((unsigned)k, i);

  int k1= i-1;
  for ( i = Size; i>=k1 ; i--)
    {
       if (bit ((unsigned)k, i)==0) q = q*q;
       else
         {
           P = P*q;
           q =q*q;
         }

    }
  return P;
}
```

Таблица 5.1

Демонстрация работы алгоритма возведения в степень

Номер разряда	Битовое значение разряда	P = 1	q = 5
16	1	$P = P \cdot q = 5$	$q = q \cdot q = 5 \cdot 5 = 5^2$
15	1	$P = P \cdot q = 5 \cdot 5^2 = 5^3$	$q = q \cdot q = 5^2 \cdot 5^2 = 5^4$
14	0		$q = q \cdot q = 5^4 \cdot 5^4 = 5^8$
13	0		$q = q \cdot q = 5^8 \cdot 5^8 = 5^{16}$
12	1	$P = P \cdot q = 5^3 \cdot 5^{16} = 5^{19}$	$q = q \cdot q = 5^{16} \cdot 5^{16} = 5^{32}$
11	0		$q = q \cdot q = 5^{32} \cdot 5^{32} = 5^{64}$
10	1	$P = P \cdot q = 5^{19} \cdot 5^{64} = 5^{83}$	

Примечание

Такой алгоритм позволяет существенно экономить время работы программы, особенно в тех случаях, когда велико значение степени. Пользуясь разрядной сеткой можно возводить в степень не только числа, но и, например, матрицы.



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. Реализовать алгоритм восстановления значения целого отрицательного числа по его дополнительному двоичному коду. Значение восстанавливаемого отрицательного числа вводится с клавиатуры.

Рекомендации: задать положительное число; применить операцию инвертирования «~»; добавить 1 к последнему разряду.

2. Из заданного числа получить новое с помощью записи его битов в обратном порядке.

3. Реализовать алгоритм подсчета числа нулевых битов в значении целого числа без знака.
4. Реализовать алгоритм подсчета пар несовпадающих битов в двух переменных типа `unsigned long`.
5. Реализовать алгоритм упаковки шестнадцатеричных чисел в диапазоне от 0 до F в массиве переменных типа `unsigned long`. Предполагается упаковка шестнадцатеричных чисел в одном элементе массива. Число упаковываемых чисел, а также сами упаковываемые числа вводятся с клавиатуры. Результирующий массив выводится на экран в шестнадцатеричном виде. Предусмотреть процедуру для «распаковки» чисел.
6. Память массива переменных `long` рассматривается как последовательность битов. Реализовать алгоритм переписи подпоследовательности битов размером `n` с позиции `p` в позицию `k`. Предполагается, что подпоследовательность битов не пересекает границы переменной `long`. Параметры `p`, `k`, `n` вводятся с клавиатуры. Предусмотреть отображение данных в шестнадцатеричном и битовом представлениях.
7. Представить распечатку разрядной сетки вещественного числа.

Рекомендации: использовать структуру типа `union`.

Класс «Строка» 6

Существует широкий класс алгоритмов, в которых информацию удобно представлять в символьном виде, например в виде слов. В языках программирования объекты, которые являются последовательностями символов из некоторого алфавита, принято называть строками.

СТРОКИ В С++

Строки в С++ как специальный тип не определены. Но в качестве такого специального типа можно рассматривать массивы символов, в которых информативная часть оканчивается нулевым символом (`'\0'`). Строка доступна через указатель на первый символ в строке. Значением переменной типа «строка» является адрес ее первого символа. Таким образом, можно сказать, что в *C++* переменная типа «строка» является указателем на строку и на первый элемент строки – первый символ. В этом смысле строки подобны массивам, потому что переменная типа «массив» тоже является указателем на свой первый элемент. Необходимо, конечно, помнить, что переменная при этом является адресом-константой, то есть хранящийся в ней адрес строки не может быть изменен.

Для работы со строками, как с массивом символов, в *C++* разработана целая библиотека подпрограмм, прототипы которых перечислены в модуле <string.h>. А в *C++ Builder*, чтобы стандартизировать работу с визуальными компонентами, предоставляющими широкие возможности для редактирования и отображения текстовых данных, подобный тип строк расширен и реализован в виде класса с именем `AnsiString`. В этом классе содержится множество методов, предназначенных для обработки текстовой информации. С простейшими приемами работы с классом `AnsiString` мы познакомились при изучении главы 2.

Но потребности различных приложений гораздо шире тех возможностей, которые заложены в подпрограммы стандартной библиотеки и в класс `AnsiString`. Поэтому при реализации проектов приходится разрабатывать собственные классы, в том числе и над строками, которые нацелены на решение задач проекта.

Факт наличия стандартных процедур по работе со строками, несомненно, необходимо учитывать. Прежде чем приступать к разработке и реализации методов собственного класса, описывающего математический объект со строковыми данными, следует выяснить, нет ли в стандартной библиотеке подпрограмм, которые целиком реализуют алгоритм метода, либо могут быть использованы в качестве операторов при реализации алгоритма метода.

6.1. Разработка классов «Строка»

Общая постановка задачи

Базовая задача. Требуется разработать семейство классов «Строка», методы которых решали бы следующие задачи:

- создавать строки;
- выделять из строки слова;
- сортировать слова по алфавиту;

- присваивать строки;
- складывать/соединять строки;
- вводить/выводить строковые данные в файл.

Решение

Математическим объектом в рамках поставленной задачи является строка – массив символов, то есть массив переменных типа `char`. Строку можно охарактеризовать *указателем на массив символов* и *числом символов*. Эти характеристики будут определять поля базового класса. Назовем его `Stroka`.

Поскольку многие задачи обработки текстовой информации, так или иначе, приводят к работе со словами, взятыми из текста по какому-либо правилу, то целесообразно в поля класса-потомка `StrokaEx`, специализированного под задачу работы со словами, включить *массив для слов*, взятых из текста, и, как следствие, *количество таких слов*. И, соответственно, дополнить этот класс методами выделения слов и их сортировки. А в класс `StrokaTask`, потомка класса `StrokaEx`, традиционно размещаются методы, связанные с вводом и отображением данных. Состав списка методов базового класса `Stroka` следует из списка задач, перечисленных в условии, но с рядом особенностей:

1. Для сложения строк предусмотрим несколько вариантов: сложение двух классовых переменных; сложение классовой переменной с массивом символов; сложение классовой переменной с отдельным символом.
2. Реализуем два варианта оператора присваивания: присваивание значения одной классовой переменной другой классовой переменной; присваивание классовой переменной числового значения (такой оператор позволит числовое значение представить строкой символов).
3. Сделаем так, чтобы можно было взять из строки символ (оператор доступа к элементу) по его номеру, пользуясь не только квадратными скобками, но и круглыми.
4. Ввод/вывод в файл реализуем с помощью переопределения потоковых операций >> и <<, которые оформим друзьями класса.
5. И введем две вспомогательные операции приведения типа: строка преобразуется в `int` (это целое – длина строки); строка преобразуется в `char*`. Последние операторы будут выполнять соответствующие преобразования из классового типа в другой определенный тип.

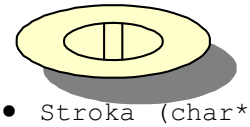
Описание базового класса `Stroka`

Этот класс разрабатывается исходя из рассуждений проведенных выше. В листинге 6.1 представлен интерфейс базового класса.

Листинг 6.1. Интерфейс базового класса

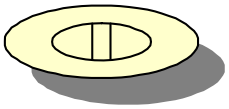
```
class Stroka
{
  protected:
    char* s;      // указатель на строку-текст
    int SLength;   // длина строки
  public:
   Stroka(char* Inits);  // Конструктор с параметром строка
   Stroka(const Stroka&);  // Конструктор копирования
```

```
    Stroka(){ s = NULL;}  // Пустой конструктор
    ~Stroka(){delete s;}
    int TakeLenS(){ return (strlen(s));}
    char operator[](int);  // Взять символ в строке по номеру
    char& operator()(int);  // Взять символ в строке по номеру
    Stroka& operator=(int);  // Присваивание
    Stroka& operator=(Stroka&);  // Присваивание
    Stroka& operator=(char* aS); // Присваивание
    operator int();   // операция приведения типа (int)
    Stroka& operator+(char); // сложение с символом
    Stroka& operator+(char*); // сложение со строкой
    Stroka& operator+(Stroka&); // операция сложения
    char* word(int n); // взять слово с номером n
    friend ifstream& operator >>(ifstream& aIn, Stroka& aS);
    friend ofstream& operator <<(ofstream& aOut, Stroka& aS);
};
```



НАЗНАЧЕНИЕ МЕТОДОВ КЛАССА `Stroka`:

- `Stroka (char* Inits);` – конструктор с одним параметром `Inits`. В соответствии со значением `Inits` инициализируются поля класса;
- `Stroka (const Stroka& aStr);` – конструктор копирования;
- `Stroka (){ s = NULL;}` – конструктор без параметров. Создает пустую строку;
- `~Stroka (){delete s;}` – деструктор – высвобождает память;
- `int TakeLenS (){ return (strlen(s));}` – метод класса – возвращает длину строки;
- `char operator [](int aNum);` – операторная функция: взять символ с номером aNum в строке `s`;
- `char& operator ()(int aNum);` – операторная функция: взять символ с номером `aNum` в строке `s`;
- `Stroka& operator =(int aNum);` – операторная функция: `<Stroka> = <число aNum>`;
- `Stroka& operator =(char* aS);` – операторная функция: `<Stroka> = <массив символов aS>`;
- `Stroka& operator =(Stroka& aStr);` – операторная функция: `<эта Stroka> = <Stroka aStr>`;
- `operator int();` – операция приведения типа `(int)`;
- `Stroka& operator +(char aCh);` – операторная функция: `<эта Stroka> = <эта Stroka>+<символ aCh>`;
- `Stroka& operator +(char* aChar);` – операторная функция: `<эта Stroka> = <эта Stroka>+<массив aChar>`;
- `Stroka& operator +(Stroka& aStr);` – операторная функция: `<эта Stroka> = <эта Stroka>+<строка aStr>`;
- `char* word(int n);` – метод класса: взять из `s` слово с номером `n`.



ДРУЗЬЯ КЛАССА `Stroka`:

- `friend ifstream& operator >>(ifstream& aIn, Stroka& aS);` – операция ввода значений `aS` из файла `aIn`;

- `friend ofstream& operator <<(ofstream& aOut, Stroka& aS);` – операция вывода `aS` в файл `aOut`.

Конструктор с параметром «массив символов»

Конструктор имеет один входной параметр `src`. Этот параметр используется для того, чтобы сформировать значение поля `s`, а также определить размер строки и этим значением инициализировать поле `SLength`. Следует отметить, что для поля `S` отводится место, на 1 символ превышающее размер входной строки. Это «лишнее» место – для признака конца строки `'\0'`. Алгоритм действий конструктора с параметром «массив символов» можно описать следующими шагами:

- определить длину входной строки;
- отвести память под поле класса;
- скопировать входной параметр в поле класса.

Описание конструктора приведено в листинге 6.2.

Листинг 6.2. Описание конструктора

```
Stroka:: Stroka(char* src)
// Конструктор - передает строку через параметр
{
  SLength = strlen(src); // вычислить длину
  S = new char[SLength+1]; // отвести память
  strcpy(s,src); // скопировать значение входного параметра
}
```

Конструктор копирования

Конструктор имеет один стандартный параметр, по которому он опознается компилятором. Копируется вся информация из классового параметра `src` в `this`. В алгоритм конструктора копирования входят следующие действия:

- отвести память под строку класса;
- инициализируется параметр – длина строки;
- копируется значение строки параметра в поле класса.

Описание конструктора копирования представлено в листинге 6.3.

Листинг 6.3. Описание конструктора копирования класса `Stroka`

```
Stroka::Stroka(const Stroka& src)
// Конструктор копирования
{
      // отводится память под строку this
  s = new char[strlen(src.s)+1];
      // инициализируется параметр - длина строки
  SLength = strlen(src.s);
      // копируется информация
  strcpy(s,src.s);
}
```

Оператор доступа к элементу «взять по индексу»

Операция «взять по индексу» – бинарная. Первый операнд – `this`. А второй передается через параметр – это номер индекса `n`. Выполняется проверка на невыход индекса за правую границу. Возвращает символ в соответствии со значением `n`

или символ пробел «». Описания операторных функций приведены в листингах 6.4–6.5.

Листинг 6.4. Описание оператора «взять по индексу»

```
char  Stroka::operator[](int n)
// Вернуть символ по его номеру
{
  if(n>=strlen(s))return(' ');
  return(s[n]);
}
```

Листинг 6.5. Описание оператора «взять по индексу»

```
char& Stroka::operator()(int n)
// Вернуть символ по его номеру
{
  if(n>=strlen(s))return(s[0]);
  return(s[n]);
}
```

Оператор присваивания для классовых переменных

Операция присваивания является бинарной. Первый операнд – `this`. А второй передается через параметр – это классовая переменная `aS`. Алгоритм сводится к выполнению процедуры копирования строк `strcpy`. Описание операторной функции приведено в листинге 6.6.

Листинг 6.6. Оператор присваивания

```
Stroka& Stroka::operator=(Stroka& aS)
// Присваивание строки
{
  strcpy(s, aS.s);
  return *this;
}
```

Операторная функция для присваивания классовой переменной целого числа

Операция – бинарная. Первый операнд – `this`. А второй – передается через параметр – это переменная целого типа `src`. Использование этого оператора позволяет присвоить классовой переменной числовое значение, записанное в виде строки символов. Алгоритм строится по схеме:

- объявляется рабочая строка;
- очищается строка из класса `this`;
- читается в рабочую строку число, передаваемое параметром `src`;
- инициализируется новый размер и отводится память под строку из класса `this`;
- копируется вспомогательная переменная в строку `this`.

Описание операторной функции приведено в листинге 6.7.

Листинг 6.7. Оператор присваивания

```
Stroka& Stroka::operator=(int src)
// Оператор присваивания, но присваивает число, которое
// преобразуется в строку
{
```

```
  char ss[80];    // рабочая строка
  if(s!=NULL)delete s;   // очистка строки - поля
      // в рабочую строку заносится число src
  sprintf(ss,"%d",src);
      // устанавливается новая длина строки  - поля
  s=new char[strlen(ss)+1];
  strcpy(s,ss);  //  копируется число
  return(*this);    // возвращается результатом поля
}
```

Оператор приведения к типу `int`

Очень простая унарная операция из одного действия: вернуть длину строки. Описание операторной функции приведено в листинге 6.8.

Листинг 6.8. Оператор приведения к типу `int`

```
Stroka::operator int()
// Приводит строку к целому типу - длине строки
{
  return(strlen(s));
}
```

Оператор приведения к типу `char*`

Унарная операция. Алгоритм описывается следующими действиями:

- объявляется вспомогательная переменная `p` типа `char*` соответствующего размера;
- в переменную `p` копируется строка класса `this`;
- значение `p` возвращается как результат работы операторной функции.

Описание операторной функции приведено в листинге 6.9.

Листинг 6.9. Оператор приведения к типу `char*`

```
Stroka::operator char*()
// Приводит строку к строке
{
  char* p = new char [ strlen(s)+1];
  strcpy(p,s);
  return(p);
}
```

Операторная функция для сложения классовых переменных

Операция – бинарная. Первый операнд – `this`, второй `t` передается через список параметров. Алгоритм описывается следующими действиями:

- объявляется вспомогательная переменная `p` для хранения строки – результата – и отводится память суммарного размера, необходимого для хранения соединенных строк;
- в эту переменную `p` копируется строка из `this`;
- к `p` добавляется строка, взятая из классового параметра;
- переопределяется значение в `this`;
- возвращается результат сложения.

Описание операторной функции приведено в листинге 6.10.

Листинг 6.10. Описание оператора сложения

```
Stroka& Stroka::operator +(Stroka& t)
//операция сложения строк
{
  char*p = new char[strlen(s)+strlen(t.s)+1];
  strcpy(p,s); // копирование
  strcat(p,t.s); // добавление
  if(s!=NULL)delete s;
  s=p; // переопределение
  return(*this);
}
```

Операторная функция для сложения классовой переменной с массивом символов

Алгоритм описывается следующими действиями:

- объявляется вспомогательная переменная `p` для хранения строки – суммарного результата;
- в эту переменную `p` копируется строка из `this;`
- к `p` добавляется строка, взятая из параметра типа указателя на массив символов;
- переопределяется значение в `this;`
- возвращается результат сложения.

Описание операторной функции приведено в листинге 6.11.

Листинг 6.11. Оператор добавления к классу массива символов

```
Stroka& Stroka:: operator+(char* t)
//операция сложения
{
  char*p = new char[strlen(s)+strlen(t)+1];
  strcpy(p,s);
  strcat(p,t);
  if(s!=NULL)delete s;
  s=p;
  return(*this);
}
```

Операция ввода строковых данных из файла

Операция является бинарной. Так как операторная функция является дружественной, то эти два параметра содержатся в списке параметров. Первый параметр `Fin` – ссылка на файловую переменную типа `ifstream`. Вторая переменная – `aS` – классового типа. Алгоритм по назначению должен выполнять следующие действия: читать из файла `Fin` текстовую информацию и помещать ее в классовую переменную `aS`. Поскольку информация в файле может состоять из символьных строк разной длины, а в классе предусмотрена одно строковое поле, решается задача последовательного чтения строк с элементом сборки этих строк в единое целое. Решение этой задачи описывается следующими шагами:

- объявляется вспомогательная переменная для временного хранения строки-результата;
- объявляется вспомогательная переменная для временного хранения очередной прочитанной из файла строки;

- пока не дотигнут конец файла выполнять цикл:
  - читать очередную строку;
  - добавлять эту строку к строке-результату;
  - добавлять пробел (чтобы не сливался текст);
- прочитанную строку присвоить параметру `aS`.

Следует отметить, что для реализации последнего шага будет использоваться, в свою очередь, переопределенная операция присваивания вида: `<Stroka> = <char*>`. Описание этой операторной функции приведено в листинге 6.12.

Листинг 6.12. Оператор «взять из файла»

```
ifstream& operator >>(ifstream& Fin, Stroka& aS)
// полагаем, что размер в 300 символов предостаточен
{
  char* Ptr =new char[300];
  int LenFull = 0;
  char P[2] =" ";
  strcpy(Ptr,"");
  char aBuf[80];
  while (!Fin.eof()) // пока файл не пуст
    {
      Fin.getline(aBuf, 80); // взять строку
      strcat(Ptr,aBuf); // подсоединить
      LenFull = strlen(Ptr);
      strcat(Ptr,P); // дополнить пробелом
    }
  Fin.close();
  LenFull = strlen(Ptr);
  aS = Ptr; // присваивание
  return Fin;
}
```

Операция вывода строки из классовых данных в файл

Операция является бинарной. Так как операторная функция является дружественной, то эти два операнда содержатся в списке параметров. Первый параметр `FOut` – ссылка на файловую переменную типа `ofstream`. Вторая переменная – `aS`, соответственно, классового типа. Алгоритм по назначению должен выполнять следующие действия: прочитать строку из переменной `aS` и поместить ее в файл `FOut`.

В принципе, это несложная процедура. Исходная строка может быть достаточно длинной, что читать ее в файле не очень удобно – она вытягивается в одну линию. Поэтому принято решение помещать информацию в файл таким образом, чтобы в одной строчке файла количество символов не превышало заданного числа, например 50. Для решения задачи потребуются вспомогательные переменные:

- строковая переменная `Ptr` – для хранения строки классовой переменной `aS` (параметра);
- строковая переменная `P`, в которую будет читаться очередная порция данных;
- указатель на строку `P1` – для позиционирования места/начала чтения очередной порции данных;
- переменная целого типа `i`, которая ограничивает количество взятых символов размером строки.

Решение этой задачи описывается следующими шагами:

- объявляются и инициализируются вспомогательные переменные. При этом используется операция: <char*> = <Stroka> (Ptr = aS), которая, в свою очередь автоматически вызывает операцию приведения классовой переменной к типу char*;
- пока не достигнут конец строки символов выполнять цикл:
  - скопировать порцию символов в P;
  - завершить P символом «переход на следующую строку»;
  - передать порцию в файл;
  - изменить P1, чтобы позиционировалась непрочитанная часть данных;
  - пересчитать количество прочитанных символов;
- закрыть файл.

Описание операторной функции приведено в листинге 6.13.

Листинг 6.13. Оператор «записать в файл»

```
ofstream& operator <<(ofstream& FOut, Stroka& aS)
//  оператор вывода строки
{
  char* Ptr = aS; // в строку переписать строку из класса
  char P[60], *P1;
  int i=0;
  P1 = Ptr; // позиционировать начало строки
  while (i<aS.SLength ) // пока есть символы
    {
      strncpy(P,P1,50); // скопировать 50 символов
      P[50] = '\n'; // завершить «концом строки»
      FOut<<P; // записать в файл
      P1 = P1+50; // передвинуть указатель
      i+=50; // пересчитать количество прочитанных символов
    }
  FOut.close();
  return FOut;
}
```

Взять из строки слово по его порядковому номеру

Метод имеет один входной параметр n, который задает номер того слова, значение которого нужно получить и вернуть в качестве результата работы функции. В предлагаемом алгоритме номер слова находится по порядковому номеру разделителя, отделяющего одно слово от другого в массиве символов. Для поиска слова используется функция strtok из стандартной библиотеки <string.h>:

```
char *strtok(char*s1,char*s2);
```

Функция strtok

В функции strtok предполагается, что входная строка s1 состоит из фрагментов, разделенных одним или многосвязными разделителями из строки s2. Рассматривается ситуация, когда одно слово от другого отделяется только знаком «пробел» (' '). При первом обращении к strtok выдается указатель на первый символ первого фрагмента строки s1. Последующие вызовы с заданием нуля (NULL) вместо первого аргумента будут выдавать адреса дальнейших фрагментов из строки s1 до тех пор, пока фрагментов не останется. Следует иметь в виду, что функция strtok после об-

наружения разделителя выставляет в соответствующее ему место признак конца строки – '\0'. То есть строка s1 «портится».

Поэтому при построении алгоритма используется вспомогательная строка с именем str. В эту строку копируется строка s из класса всякий раз при обращении к методу класса. Вводится вспомогательная переменная k – счетчик выделенных слов. Основу алгоритма составляет цикл, в котором выделяется очередное слово до тех пор, пока счетчик k не достигнет заданного номера n. Для того чтобы возможно было выделить в строке, в том числе и последнее слово, на начальном этапе к концу строки добавляется знак пробела. Описанный алгоритм представлен в листинге 6.14.

Листинг 6.14. Описание метода, предназначенного для выделения слова из строки по его номеру

```
char* Stroka::word(int n)
// Взять слово с номером n
// str - вспомогательный массив
// k - счетчик слов
// t - вспомогательный массив для хранения очередного слова
{
  char str[100];
  int k=1;  // Инициализация счетчика слов
  strcpy (str, s); // Исходное предложение копируется в str
  strcat (str, " "); // Добавить пробел к концу строки
  char* t=new char[20]; // Место под t
      // Выделение первого слова и запись его в t
  t = strtok (str, " ");
      // Пока есть слова в предложении  и номер меньше, чем n
  while (t != NULL && k<n)
    {
       t=strtok(NULL," "); // Занести в t очередное слово
       k++; // Счетчик слов увеличить на 1
    }
  return t; // Вернуть t в качестве результата функции
}
```

Описание класса StrokaEx – потомка первой очереди

В поля расширенного класса включены новые характеристики: массив слов (aWrds) и, соответственно, количество слов (FCounts). Предполагается, что длина слова не может превышать числа 30 по величине. Поэтому массив слов описывается как указатель на char [30]. Поскольку по условию требуется включение метода сортировки слов по алфавиту, к полям класса добавляется массив указателей (m), в котором будут храниться адреса слов, следующих в алфавитном порядке. В результате такого устройства класса слова можно брать и в том порядке, как они располагаются в строке, и в алфавитном порядке. Интерфейс класса представлен в листинге 6.15.

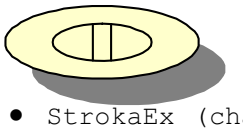
Листинг 6.15. Интерфейс класса StrokaEx

```
class StrokaEx: public Stroka
{
  protected:
    char** FWrds;
    char(*aWrds)[30]; // указатель на слова
    int FCounts;  // число слов
    char **m; // адреса «по алфавиту»
  public:
```

```
    StrokaEx(char* InitStr);
    StrokaEx();
    int CalcInitWrds();

    void DoWords(); // выделить слова
    void SortStr(); // отсортировать слова

};
```



НАЗНАЧЕНИЕ МЕТОДОВ КЛАССА `StrokaEx`:

- `StrokaEx (char* InitStr);` – конструктор с одним параметром `InitStr`. Передает значение `InitStr` классу-предку;
- `StrokaEx (const Stroka& aStr);` – конструктор копирования;
- `StrokaEx();` – конструктор без параметров. Обеспечивает выполнение конструктора-предка;
- `~StrokaEx();` – деструктор;
- `void DoWords();` – метод для выделения всех слов из строки – поля класса;
- `void SortStr();` – метод для сортировки слов по алфавиту;
- `int CalcInitWrds();` – метод определяет число слов и инициализирует поля.

Конструктор с параметром «массив символов»

Поскольку в классе-потомке объявлены новые поля, в задачу конструктора входит задача не только передать значение параметра классу-предку, но и произвести инициализацию своих собственных полей. Чтобы решить эту задачу, необходимо определить количество слов в строке класса. Для этого вызывается процедура `CalcInitWrds`. Описание конструктора представлено в листинге 6.16.

Листинг 6.16. Описание конструктора класса

```
StrokaEx::StrokaEx(char* InitStr): Stroka(InitStr)
{
   CalcInitWrds(); // инициализация и место под массив слов
}
```

Определение количества слов и инициализация полей класса

Процедура `CalcInitWrds` не имеет параметров. Будем исходить из того, что слова разделены пробелами, причем этих пробелов может быть много. Это означает, что количество слов может быть много меньше, чем количество пробелов. Этот факт следует учитывать при подсчете количества слов. Алгоритм должен выполнять следующее:

а) подсчитать число пробелов, причем пробелы, идущие друг за другом, следует считать за один пробел (тогда число пробелов можно связать с числом слов);
б) инициализировать поля и отвести память.

Подсчет количества слов/пробелов сводится к двойному циклу. Алгоритм можно описать следующими шагами:

1) выполнить внешний цикл (пока есть символы в строке):
   а) если встретился символ «пробел», то счетчик числа слов увеличивается на 1;
   б) выполнить внутренний цикл (выполнять цикл пока текущий символ – «пробел» и пока не достигнут конец строки);

2) отвести память под массивы слов.

В задачу внутреннего цикла входит задача – пропустить «пробелы». Описание метода `CalcInitWrds`, предназначенного для определения количества слов и инициализации полей класса, представлено в листинге 6.17.

Листинг 6.17. Вычисление количества слов в строке класса

```
int StrokaEx::CalcInitWrds()
{
   int p=0;
   for (int i=0;i<SLength; i++)
      {
        if (s[i]==' ')p++;
        while(s[i]==' '&& i<SLength)i++;
      }
   FCounts = p;
   m=NULL;
   aWrds = new char[ FCounts][30];
   FWrds = new char*[ FCounts];
   for (int i=0;i< FCounts; i++) FWrds[i]= new char[30];
   return p;
}
```

Выделение слов из строки

Процедура `DoWords` не имеет параметров. Так как в базовом классе описан метод выделения слова по его номеру, решение задачи сводится к простому циклу с одной операцией: выделить очередное слово и скопировать его в массив слов. Описание метода `DoWords` представлено в листинге 6.18.

Листинг 6.18. Формирование массива слов из строки класса

```
void StrokaEx::DoWords()
{
  for(int i=0; i<FCounts; i++)
    {
      Strcpy (aWrds[i], word(i+1));
    }
}
```

Сортировка слов по алфавиту

Метод `SortStr` не имеет параметров. Задача алгоритма сводится к получению значений `m`, каждый элемент которого должен будет указывать на адрес очередной строки/слова, следующего в алфавитном порядке. По сути, мы имеем дело с обычной сортировкой элементов массива, только элементом является слово, а роль операндов в операции «меньше» играют позиции сравниваемых слов в алфавите. Так как исходный массив слов сохраняется в неизменном виде, в алгоритм вводится вспомогательный массив (`ind`), исполняющий роль признака: взято или нет (то есть свободно или нет) соответствующее слово в упорядоченную часть: 0 – слово не взято; 1 – слово взято. Первоначально все элементы массива-признака равны 0. Решение сводится к повторению этих шагов, то есть выполняется цикл – пока есть слова:

а) находится очередной наименьший элемент из числа свободных и запоминается его номер;
б) адрес (или номер) найденного слова запоминается в массиве `m`;

в) слово переводится в категорию «занятое».

Описание метода `SortStr`, предназначенного для сортировки слов, представлено в листинге 6.19.

Листинг 6.19. Сортировка слов в алфавитном порядке

```
void StrokaEx::SortStr()
// ind – признак того, занято или нет слово
// p – очередное «наименьшее» значение первого символа
// в слове; g – номер «наименьшего» слова
{
  int i, j, g, *ind;
  ind = new int[FCounts];
      // инициализация массива-признака
  for(i = 0; i<FCounts; ind[i]=0,i++);

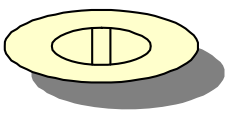
  m = new char*[FCounts];
  char p;
  for(i = 0; i<FCounts;i++) // делать цикл по числу слов
     {
           // найти минимум
       p = 'z'; // последняя буква в алфавите
       g = 0; // инициализация номера
       for(j = 0; j<FCounts;j++)
            // если слово «меньше» и свободно
          if (aWrds[j][0] <= p && ind[j] == 0)
            {
                 // запомнить «координаты»
              p = aWrds[j][0]; g=j;
            }
       ind[g] = 1; // пометить слово, как занятое
       m[i] = &aWrds[g][0]; // запомнить адрес
     }
}
```

Описание класса `StrokaTask`

Класс дополняет расширенный класс `StrokaEx` методами ввода и вывода строк в поля визуальных компонент *C++ Builder*. Интерфейс класса представлен в листинге 6.20.

Листинг 6.20. Интерфейс класса `StrokaEx`

```
class StrokaTask: public StrokaEx
{
  public:
      StrokaTask(char*);
      StrokaTask():StrokaEx(){};
      void PrintWrds(TListBox* FBox, Boolean Flag);
      void TakeS(TEdit*);  // Вывод в Edit
      void TakeS(TListBox*);  // Вывод в ListBox
};
```



НАЗНАЧЕНИЕ МЕТОДОВ КЛАССА `StrokaTask`:

- `StrokaTask (char* Inits);` – конструктор с одним параметром `Inits`. Передает `Inits` конструктору класса предка;

- `StrokaTask (const Stroka& aStr);` – конструктор копирования;
- `StrokaTask(): StrokaEx(){};` – конструктор без параметров;
- `~StrokaTask (){};` – деструктор;
- `void PrintWrds(TListBox* FBox, Boolean Flag);` – метод класса, предназначенный для вывода слов в список `FBox`. `Flag` – признак, определяющий порядок слов: `true` – в порядке следования; `false` – в алфавитном порядке;
- `void TakeS(TEdit* aEd);` – метод класса, предназначенный для вывода строки класса в окно редактора `aEd`;
- `void TakeS(TListBox* aLsb);` – метод класса, предназначенный для вывода строки класса в окно списка `aLsb`.

Примечание

Написание данных конструкторов и деструктора не представляет алгоритмической сложности, потому тексты кодов для них не приводятся.

Вывод строки в окно редактора

Метод `TakeS` имеет один параметр, через который передается наименование редактора для отображения строки класса. Описание метода приведено в листинге 6.21.

Листинг 6.21. Вывод строки в окно редактора

```
void StrokaTask::TakeS(TEdit* LBox)
// Печать строки
{
   AnsiString Ptr = s;
   LBox->Text = Ptr;
}
```

Вывод строки в окно списка

Метод `TakeS` имеет один параметр, через который передается наименование списка для отображения строки класса. Описание метода приведено в листинге 6.22.

Листинг 6.22. Вывод строки в окно списка

```
void StrokaTask::TakeS(TListBox* LBox)
// Печать строки
{
   AnsiString Ptr = s;
   LBox->Items->Add( Ptr);
}
```

Вывод слов в окно списка

Метод класса позволяет выводить слова, сформированные из строки класса как в порядке их следования, определенном в процессе выделения, так и в алфавитном (отсортированном) порядке. Какой из списков следует предоставить в выводе, определяется значением параметра `Flag`. Если `Flag` = 1, то порядок обычный, если `Flag` = 0, то передается упорядоченный список слов. В алгоритм вводится вспомогательная переменная `Buf`. В эту переменную копируются адреса слов для вывода в зависимости от признака `Flag`. Описание метода приведено в листинге 6.22.

Листинг 6.22. Вывод слов в список типа ListBox

```
void StrokaTask::PrintWrds(TListBox* FBox, Boolean Flag)
{
   char *Buf[30];
   if (Flag)
     for(int i=0; i < FCounts; Buf[i]=aWrds[i],i++);
   else
     for(int i=0; i < FCounts; Buf[i]=m[i],i++);
   for(int i=0; i < FCounts;i++)
     {
       FBox->Items->Add(Buf[i]);
     }
}
```

6.2. Решение задач с использованием классов «Строка»

Задача 6.1. В текстовом файле содержится набор некоторых слов, разделенных пробелами. Требуется найти самое длинное слово. Если таких слов несколько, то взять первое по порядку следования.

Решение

Решение целесообразно оформить в виде метода класса `StrokaEx`. Для решения задачи потребуется:

а) сформировать массив слов;
б) найти номер слова с максимальной длиной;
в) вернуть слово с найденным номером.

Решение задачи оформлено в виде метода с именем `LongWords` класса `StrokaEx`. Описание метода `LongWords` представлено в листинге 6.23.

Листинг 6.23. Описание метода поиска самого длинного слова в заданном предложении

```
char* StrokaEx::LongWords()
{
  int MaxLen = 0, iMax;
  DoWords(); // сформировать массив слов
  for(int i=0; i<FCounts; i++)
    {
      if(strlen(aWrds[i]) > MaxLen)
        {
          MaxLen =  strlen(aWrds[i]);
          iMax =i;
        }
    }
  return aWrds[iMax];
}
```



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. Записать слова предложения в обратном порядке.
2. Удалить из предложения слова с заданными номерами.

3. Найти слова, имеющие наибольшее число вхождений в предложении.
4. В предложении найти слова, состоящие из тех же символов, что и заданное слово, если таковые есть.
5. Из двух предложений удалить одинаковые слова, если таковые есть.
6. Даны два предложения, причем второе состоит из слов первого, записанных в другом порядке. Найти этот порядок.
7. В предложении все слова начинаются с различных букв. Напечатать, если можно, слова предложения в таком порядке, чтобы последняя буква каждого слова совпадала с первой буквой следующего.
8. Найти множество всех слов, которые встречаются в каждом из двух заданных предложений.
9. В заданном предложении найти пару слов, из которых одно является обращением другого.
10. Дано множество слов одинаковой длины, два слова из этих слов выделены. Построить цепь от первого выделенного слова ко второму так, чтобы все слова этой цепочки были только из заданного множества. Соседние слова построенной цепи должны отличаться только одной буквой.

Разработка классов «Матрица», предназначенных для решения задач линейной алгебры 7

Процесс решения сложной проблемы с использованием матричных вычислений сводится к последовательному решению более простых задач линейной алгебры и является в настоящее время одним из важнейших направлений развития вычислительной математики.

Главным математическим объектом в задачах линейной алгебры является матрица (двумерный массив). А программы, предназначенные для решения задач линейной алгебры, по сути, являются подмножеством программ обработки двумерных массивов.

7.1. Разработка классов «Матрица»

Общая постановка задачи

Базовая задача. Требуется разработать семейство классов «Матрица», предназначенных для выполнения следующих действий:

- умножать матрицу на заданное число (коэффициент);
- вычислять произведение двух матриц;
- определять норму матрицы;
- вычислять сумму матриц;
- вычислять разность матриц;
- вычислять обратную матрицу;
- вычислять определитель матрицы;
- транспонировать матрицу;
- находить наименьший (наибольший) из элементов.

Стратегия решения

Производится *анализ* задачи: выделяются математические объекты, определяются характеристики и операции, позволяющие моделировать поведение матрицы. Все это распределяется по классам, то есть, производится *разработка классов*. При этом сначала проектируются и продумываются интерфейсы классов, а после этого формализуются методы классов в рамках технологии *«нисходящего проектирования»*. Такой подход позволяет свести программу решения отдельной сложной задачи к решению последовательности простых задач. Процедуры решения простых задач добавляются к методам класса, обычно в область *protected*. Подготовленная система классов оформляется в виде отдельного модуля проекта. Затем создается интерфейс приложения и обеспечивается его функциональность.

Решение

а) Выделение математических объектов

В рамках поставленной задачи фундаментом для разрабатываемого семейства классов является математический объект – матрица, принадлежащая пространству $R^n \times R^m$. Перечисленный набор действий связан либо с преобразованием элементов

отдельной матрицы, либо затрагивает взаимодействие двух матриц и они укладываются в поведение выделенного математического объекта.

б) Разработка классов

Разработка структуры данных

Конкретная матрица A определяется тремя характеристиками: размерностью, то есть числом строк и числом столбиков, а также индивидуальным набором элементов, определяющим эту матрицу. Следует отметить, что элементы матрицы, например типа *double*, хранятся в двумерном массиве, память для которого должна будет выделяться автоматически в момент создания экземпляра типа класса. Именно благодаря такому решению можно будет создавать объекты-матрицы с любым количеством элементов, например, типа *double*. Ключевой момент в том, что матрица в *C++* в таком случае должна быть описана как массив указателей на указатели, то есть типа *double***.

Поля класса следует объявить в секции `protected`, поскольку они предназначены для использования наследниками. Такое решение позволяет также выявлять ошибки типа нарушения прав доступа к данным еще на этапе компиляции программы.

Кроме этого, для удобства использования в классе можно предусмотреть возможность присваивания конкретному объекту-матрице содержательного имени. Имя будет храниться в виде переменной типа «массива символов» с размерностью, не превышающей количество символов, составляющих максимальное по числу символов имя объекта. Это имя можно «поправлять», присоединяя к нему по мере продвижения вычислений некоторый поясняющий текст, например знаки операций.

Все остальные свойства матрицы следуют однозначно из этих характеристик, которые определяют поля базового класса.

Разработка модели поведения математического объекта

Действия, перечисленные в постановке задачи, определяют основной список методов класса.

Но помимо этого класс должен предоставлять разработчикам программ следующие возможности: задавать исходные данные – размеры матрицы и значения элементов; генерировать значения элементов по какому-либо правилу; формировать файл с данными значениями с целью повторного использования; извлекать данные из файла данных и заполнять матрицу; отображать матрицу как информацию в различных видах. Сформулированные выше возможности будут определять поведение объектов класса и должны быть также заложены в методы класса.

Естественно, что кроме заявленных выше методов, класс-матрица, как и любой другой класс, должен иметь несколько конструкторов и деструктор. В *теле конструктора* должна быть, как минимум, реализована следующая последовательность действий: инициализация полей класса; выделение памяти под массив элементов; при необходимости инициализация элементов матрицы. В *деструкторе* происходит обратный процесс – занимаемая память высвобождается.

Примечание

Часть методов класса, таких как сложение, умножение, смена знака и так далее, целесообразно оформить в виде операторных функций. Тогда применительно к матрицам можно будет тоже использовать формат записи операций, естественный для записи алгебраических выражений для переменных базовых типов.

Все операторы, за исключением *оператора присваивания*, следует сделать дружественными (`friend`), а не включать в методы класса. Это поможет снять ограничения на построение выражений с типом класса.

Примечание

Оператор присваивания может быть только методом класса.

Для некоторых операторов и функций желательно, чтобы один или оба параметра были бы объявлены с использованием ключевого слова *const*. Это будет гарантировать, что в процессе работы переданные значения (параметры) *не будут изменены*. Попытки изменения будут выявлены на этапе трансляции как ошибки.

Разработка схемы иерархии классов

В рамках поставленной задачи целесообразно выделить семейство из трех классов. В базовый класс поместить методы, общие для перечисленных задач, в том числе операторные функции.

В классе-наследнике первой очереди следует поместить методы решения задач, перечисленных в условии и не описанных в базовом классе. А в третий класс-наследник от наследника поместить методы, связанные с вводом и отображением данных. В таком представлении первые два класса будут мобильными, то есть они практически без исправлений могут компилироваться под любой системой программирования для *C++* и для различных операционных систем. Третий класс будет содержать методы с учетом работы в конкретной системе программирования, то есть дополнять предыдущие классы методами, которые привязаны к типовым элементам, таким как классы и компоненты, предопределенные в нашем случае в среде *C++ Builder*.

В соответствии с представленной концепцией разработки классов, для решения поставленной задачи создается три класса.

Введем следующие обозначения:

- `TMatrix` – базовый класс «матрица»;
- наследник `TMatrixEx` – класс «матрица» с расширенными возможностями;
- наследник `TMatrixTask` – класс, дополненный методами, связанными с компонентами *C++ Builder*.

Описание базового класса TMatrix

На основании анализа задачи в базовый класс `TMatrix` целесообразно включить:

- *поля*

- FData – указатель на массив;
- FRowCount, FColCount – число строк и столбцов матрицы;

- *методы*
  - конструкторы;
  - деструктор;
  - операторные функции;
  - методы транспонирования;
  - методы задания элементов матрицы числовыми значениями;
- *свойства*
  - длина массива;
  - индексированное свойство доступа к элементам;
  - указатель на массив.

СВОЙСТВА КЛАССА
property

Свойства класса, по существу, являются полями класса с большими возможностями. Понятие «свойства» объединяет понятие «поле данных» и «функция/метод» его записи и/или чтения. В интерфейсной части описания класса сами поля объявляются, как правило, в разделах *private* или *protected*. Если в объявлении свойства после ключевого слова read/write записано просто имя поля, то это означает, что специальная функция чтения/записи отсутствует, и компилятор строит код непосредственной работы со значением поля.

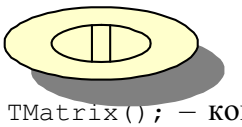
В листинге 7.1 представлен интерфейс разработанного базового класса TMatrix.

Листинг 7.1. Интерфейс базового класса «Матрица вещественных чисел»

```
class TMatrix
{
  protected:
    int FRowCount; // число строк
    int FColCount; // число столбцов
    double** FData;  // указатель на матрицу
  public:
    TMatrix (int aInitRow, int aInitCol); // конструктор
    TMatrix (const TMatrix&); // конструктор копирования
    TMatrix(); // конструктор по умолчанию
    ~TMatrix ();  // деструктор
    // получить размеры матрицы
    void GetMatrixSize (int& aRow, int& aCol);

    // вернуть значение элемента, расположенного
    // в строке iRow и столбце iCol
    double GetData (int iRow, int iCol)
                                  {return(FData[iRow][iCol]);}
    // получить значения ДСЧ в диапазоне [Left,Right]
    void InitRandom (int Left, int Right);
    // выполнить транспонирование «на месте»
    TMatrix& Transpose();
    // транспонирование в новую матрицу
    TMatrix& Transpose (TMatrix&);
    void InitNul(); // обнуление матрицы
```

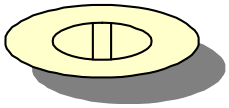
```
    // оператор присваивания и др.
    TMatrix operator = (const TMatrix&);
    TMatrix& operator ! ();   // смена знака
    friend TMatrix& operator + (const TMatrix &aMatr,
                 const TMatrix &bMatr); // оператор сложения
    friend TMatrix& operator += (TMatrix &aMatr,
            const TMatrix& BMatr); // оператор увеличения на
    friend TMatrix& operator - (const TMatrix &aMatr,
               const TMatrix &bMatr); // оператор вычитания
    // оператор перемножения двух матриц
    friend TMatrix operator * (const TMatrix &aMatr,
                                    const TMatrix &bMatr);
    // оператор умножения числа на матрицу
    friend TMatrix operator * (double aX,
                                       const TMatrix &bMatr);
    // оператор умножения матрицы на число
    friend TMatrix operator * (const TMatrix &aMatr,
                                                  double aY);
    // оператор сравнения матриц
    friend TMatrix& operator == (const TMatrix &aMatr,
                                    const TMatrix &bMatr);
    friend TMatrix& operator - (TMatrix &aMatr); //оператор
    void GetFromFile (char*);
         /* свойства */
      // кол-во строк
 __property int RowCount = {read = FRowCount};
      // кол-во столбцов
 __property int ColCount = {read = FColCount};
      // кол-во столбцов - используется метод GetData
 __property double Data[int iR][int iC] = {read = GetData};
};
```



НАЗНАЧЕНИЕ МЕТОДОВ КЛАССА `TMatrix`:

- `TMatrix();` – конструктор без параметров или, как его называют, «по умолчанию». Использовать такой конструктор можно, когда размеры матрицы будут определяться в процессе работы. Создавать матрицу в таком случае можно, например, в диалоговом режиме: конструктор объекта-матрицы запросит у пользователя необходимые данные;
- `TMatrix (int aInitRow, int aInitCol);` – конструктор с двумя параметрами. Через параметр `aInitRow` передается число строк матрицы, а через параметр `aInitCol` передается число столбцов матрицы. В соответствии с этими размерами отводится необходимая память под динамический двумерный массив `FData`;
- `TMatrix (const TMatrix& aMatr);` – конструктор копирования. Конструктор имеет один параметр `aMatr` с типом ссылки на класс. Используется автоматически при вызове процедур с параметрами классового типа, передаваемыми «по значению»;
- `~TMatrix ();` – деструктор. Высвобождает динамическую память, отведенную под матрицу данных `FData`;

- `void GetMatrixSize (int& aRow, int& aCol);` – метод класса. Имеет два выходных параметра. Через параметр `aRow` передается число строк, а через параметр `aCol` передается число столбцов матрицы `FData`;
- `double GetData (int iRow, int iCol);` – метод класса. Имеет два входных параметра, которые указывают на местоположение (`iRow` – номер строки; `iCol` – номер столбца) элемента внутри матрицы `FData`, значение которого нужно вернуть через заголовок. Описан в *inline*;
- `void InitRandom (int Left, int Right);` – метод класса. Присваивает элементам матрицы `FData` значения, сгенерированные датчиком случайных чисел (ДСЧ) из диапазона `[Left, Right]`;
- `TMatrix& Transpose();` – метод предназначен для транспонирования матрицы `FData`. В результате матрица преобразуется в транспонированную матрицу. Возвращает ссылку на себя. Можно использовать в том случае, если матрица квадратная;
- `TMatrix& Transpose (TMatrix& aMatr);` – метод предназначен для транспонирования матрицы `FData`. Имеет один выходной параметр `aMatr`, куда заносится транспонированная матрица. Возвращает ссылку на `aMatr`. Можно использовать в том случае, если матрица прямоугольная;
- `void InitNul();` – метод присваивает всем элементам матрицы `FData` нулевые значения;
- `TMatrix operator = (const TMatrix& aMatr);` – операторная функция. Определяет операцию присваивания для переменных классового типа. Здесь `aMatr` – второй операнд в операции. Операция возвращает в качестве результата присвоенное значение (значение первого операнда);
- `TMatrix& operator ! ();` – унарная операция. Предназначена для смены знака у элементов матрицы `FData`;
- `void GetFromFile (char* NameFile);` – метод класса. Имеет один входной параметр `NameFile` – имя файла. В данном случае предлагается использовать текстовый файл. В нем расположены числовые данные, которые нужно записать в матрицу. При записи данных в файл дробная и целая части отделены друг от друга точкой. В начале файла, обычно в его первой строке, размещены два числа – число строк и число столбцов. Эти числа считываются в соответствующие поля класса, отводится память под данные, а затем данные читаются в ячейки матрицы.



ДРУЗЬЯ КЛАССА `TMatrix`:

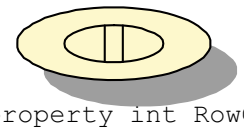
- `friend TMatrix& operator + (const TMatrix &aMatr, const TMatrix &bMatr);`

  определяет операцию сложения двух матриц. Имеет два входных параметра `aMatr` и `bMatr`, которые передаются по ссылке. Эти параметры определяют слагаемые. Результат сложения возвращается через заголовок функции;
- `friend TMatrix& operator += (TMatrix &aMatr, const TMatrix &bMatr);`

  имеет два параметра: `bMatr` – входной, `aMatr` – входной и выходной. Определяет операцию сложения двух матриц, точнее, увеличения значений матрицы `aMatr`

на значения элементов матрицы `bMatr`, которые передаются по ссылке. Эти параметры определяют слагаемые. Результат сложения записывается в `aMatr` и возвращается и через формальный параметр, и через заголовок функции;

- `friend TMatrix& operator - (const TMatrix &aMatr, const TMatrix &bMatr);`
  определяет операцию вычитания двух матриц. Имеет два входных параметра `aMatr` и `bMatr`, которые передаются по ссылке. Эти параметры определяют уменьшаемое и вычитаемое, соответственно. Результат вычитания возвращается через заголовок функции;
- `friend TMatrix operator * (const TMatrix &aMatr, o const TMatrix &bMatr);`
  определяет операцию умножения двух матриц. Имеет два входных параметра `aMatr` и `bMatr`, которые передаются по ссылке. Эти параметры определяют матрицы-сомножители. Произведение возвращается через заголовок функции;
- `friend TMatrix operator * (double aX, const TMatrix &bMatr);`
  определяет операцию умножения числа (параметр `aX`) на матрицу (параметр `bMatr`). Произведение возвращается через заголовок функции;
- `friend TMatrix operator * (const TMatrix &aMatr, double aY);`
  определяет операцию умножения матрицы (параметр `bMatr`) на число (параметр `aY`). Произведение возвращается через заголовок функции;
- `friend int operator == (const TMatrix &aMatr, const TMatrix &bMatr);`
  операторная функция. Сравнивает поэлементно две матрицы, определенные параметрами `aMatr` и `bMatr`. Если элементы матриц совпадают, то возвращается 1, иначе возвращается 0.



СВОЙСТВА КЛАССА `TMatrix`:

- `__property int RowCount = {read = FRowCount};` – количество строк;
- `__property int ColCount = {read = FColCount};` – количество столбцов;
- `__property double Data[int iR][int iC] = {read = GetData};` – значение элемента с заданными номерами строки и столбца.

Примечание

Очевидно, что число описанных операторных функций не претендует на полноту, оно может быть расширено такими действиями, как деление элементов на число, сложение матрицы с числом и так далее. Для этого достаточно добавить прототипы таких функций к интерфейсу класса и включить описание кода в раздел реализации.

Как это следует из интерфейса, для задания значений матрицы здесь предусмотрены следующие варианты: взять данные из файла; сгенерировать значения датчиком случайных чисел; задать элементы матрицы нулевыми значениями.

Для транспонирования матрицы, а эта операция одна из самых «ходовых» в линейной алгебре, предусмотрены также два варианта: транспонирование на месте, когда транспонируемая матрица заменяется матрицей, полученной в результате транспонирования (это допускается в данном варианте исполнения только для квадратных матриц); транспонирование матрицы с записью результата в новую матрицу (эта операция возможна и для прямоугольных матриц, пользователю в этом случае

не следует забывать о размерах – они у матриц исходной и результирующей противоположны).

Основные приемы, используемые в алгоритмах обработки матриц

Особенность любой задачи по обработке матрицы связана с тем, что для доступа к какому-либо элементу матрицы, следует задать две его «координаты» – номер строки и номер столбца, которые указывают, где находится данный элемент. Если посмотреть на запись элемента `a[i][j]`, то его первая координата `i` задает номер строки матрицы, а `j` задает номер столбца. Большинство «матричных» задач связано с обработкой элементов, которая заключается в том, что для решения задачи требуется организовать просмотр/изменение всех элементов матрицы.

Порядок, в котором берутся элементы матрицы, определяется порядком изменения индексов элементов. Для изменения индексов, а их два, необходимо организовать два цикла, а точнее – двойной цикл или цикл в цикле. Один цикл будет перебирать первый индекс, то есть менять номер строки, а второй цикл – изменять второй индекс, то есть менять номер столбца. Если внешний цикл организован по номеру строки, а внутренний – по номеру столбца, то перебор элементов матрицы будет в порядке – по строчкам. Иначе – по столбикам. Так, в конструкции вида

```
for (int i=0; i<FRowCount; i++) // перебор строк
  for (int j=0; j<FColCount; j++) // перебор столбцов
     {
          // обработка a[i][j]
     }
```

реализован доступ к элементам матрицы «по строкам», так как здесь второй индекс меняется быстрее, чем первый. А в конструкции вида

```
for (int j=0; j<FColCount; j++) // перебор столбцов
  for (int i=0; i<FRowCount; i++) // перебор строк
     {
          // обработка a[i][j]
     }
```

реализован доступ к элементам матрицы «по столбцам».

В приведенных ниже программных кодах практически вся обработка построена на подобных конструкциях. К более сложным алгоритмам прилагаются дополнительные пояснения по ходу изложения.

Конструктор класса «Матрица» с параметрами-размерами

Конструктор имеет два входных параметра: `aInitRow` – определяет количество строк, `aInitCol` – определяет количество столбцов в матрице. Эти параметры используются для инициализации размеров матрицы и выделения памяти. Процедура выделения памяти под матрицу сводится к двум шагам:

- отводится память под массив указателей на строки матрицы;
- отдельно в цикле (по числу строк) отводится память под каждую строку.

Код реализации конструктора с параметрами, задающими размер матрицы, приведен в листинге 7.2.

Листинг 7.2. Конструкторы класса «матрица» с параметрами-размерами

```
TMatrix::TMatrix (int aInitRow, int aInitCol)
{
  FColCount = aInitCol; // инициализировать поле FColCount
  FRowCount = aInitRow; // инициализировать поле FRowCount

      // отвести память под массив указателей на строки
  FData = new double* [FRowCount];
      // отвести память под строки
  for (int i=0;  i<FRowCount;  i++)  // перебор строк
      FData[i] = new double [FColCount];
}
```

Конструктор копирования класса «Матрица»

Конструктор имеет один *классовый* параметр `aMtr`, откуда берет значения своих полей. В соответствии с этими значениями отводится память под матрицу `FData`. И в эту матрицу копируются данные из матрицы `aMtr`. Алгоритм работы конструктора следующий:

- значения размеров матрицы из параметра `aMtr` присваиваются полям-размерам матрицы `this`;
- в соответствии с установленными размерами отводится память под матрицу `this`;
- значения элементов матрицы, взятой из параметра `aMtr`, присваиваются элементам матрицы `this`.

Код реализации конструктора копирования приведен в листинге 7.3.

Листинг 7.3. Конструкторы копирования класса `TMatrix`

```
TMatrix :: TMatrix (const TMatrix &aMtr)
{
   // читается из aMtr число строк и столбиков в this
  FColCount = aMtr.FColCount;
  FRowCount = aMtr.FRowCount;
   // отводится память под элементы this
  FData = new double* [FRowCount];
  for (int i=0; i<FRowCount; i++) // перебор строк
    FData[i] = new double [FColCount];
   // копируется матрица из aMtr в матрицу this
  for (int i=0; i<FRowCount; i++) // перебор строк
    for (int j=0; j<FColCount; j++) // перебор столбцов
      FData[i][j] = aMtr.FData[i][j];
}
```

Деструктор класса «Матрица»

Деструктор класса `TMatrix` высвобождает динамически отведенную память. Алгоритм деструктора можно описать двумя шагами:

- в цикле высвобождается память, отведенная под строки матрицы;
- высвобождается память, занятая массивом указателей на строки.

Код реализации деструктора класса `TMatrix` приведен в листинге 7.4.

Листинг 7.4. Деструктор класса `TMatrix`

```
TMatrix::~TMatrix()
```

```
{
    // освободить память, занятую строками
  for (int i=0; i<FRowCount; i++)
    delete[] FData[i];
// освободить память, отведенную для массива указателей
  delete[] FData;
}
```



Следует обратить внимание на *порядок*, в котором отводится память под матрицы конструктором и, соответственно, высвобождается деструктором. Высвобождение памяти идет в порядке, обратном по отношению к отведению памяти.

Операция присваивания переменных классового типа

Операция присваивания классовых переменных – бинарная. В операции присваивания вида <*TMatrix*> = <*TMatrix*> два операнда:

- первый операнд, расположенный слева – текущий объект класса «матрица», на который ссылается указатель `*this`;
- второй операнд, расположенный справа и играющий роль выражения в операторе присваивания – матрица, определенная параметром `aMatr`.

Указатель на текущую матрицу `this` возвращается как результат работы оператора присваивания.

Порядок действий этого оператора, описывается следующим алгоритмом:

а) проверяются размеры операндов на соответствие, то есть сравниваются число строк и число столбцов;
б) если у первого операнда размеры не соответствуют второму, то перейти нужно к пункту *в*, иначе – к пункту *е*;
в) высвобождается память, занятая матрицей;
г) заменяются размеры матрицы-приемника на размеры матрицы-операнда;
д) отводится память в соответствии с новыми размерами;
е) выполняется копирование данных.

Описание операции присваивания приведено в листинге 7.5.

Листинг 7.5. Описание операции присваивания, если операнды одного типа: <*TMatrix*> = <*TMatrix*>

```
TMatrix TMatrix::operator = (const TMatrix& aMatr)
{
      // проверка размеров на соответствие
  if (FRowCount != aMatr.FRowCount || FColCount !=
                                              aMatr.FColCount)
   {
    // размеры не совпадают, освободить прошлую память
     for (int i=0; i<FRowCount; i++)
       delete[] FData[i];
     delete[] FData;

       // отвести новую память
     FRowCount = aMatr.FRowCount;
     FColCount = aMatr.FColCount;
     FData = new double* [FRowCount];
```

```
     for (int i=0; i<FRowCount; i++)
       FData[i] = new double [FColCount];
    }
      // скопировать данные
  for (int i=0; i<FRowCount; i++)
    for (int j=0; j<FColCount; j++)
      FData[i][j] = aMatr.FData[i][j];
   return *this;  // вернуть результат
}
```

Оператор смены знаков у элементов матрицы

Операция, предназначенная для смены знака, является унарной. Так как операция представлена как метод класса, то параметры отсутствуют. Используется только один операнд `this` – «эта» матрица. В матрице с помощью двойного цикла перебираются все ее элементы матрицы с преобразованием этих элементов путем смены знака. Описание операторной функции приведено в листинге 7.6.

Листинг 7.6. Операции смены знаков элементов матрицы

```
TMatrix& TMatrix::operator ! ()
// изменяет знак у элементов матрицы класса
{
     for( int i=0; i<FRowCount; i++)
        for( int j=0; j<FColCount; j++)
           this->FData[i][j] = -FData[i][j];
     return *this;
}
```

Дружественная функция смены знаков элементов матрицы

И эта операция – унарная, но функция определена как дружественная. Поэтому востребованный один классовый операнд `aMatr` передается через параметры. Параметр будет и входным, и выходным, так как операция изменяет его значения. Следовательно, этот операнд передается по ссылке. В матрице `aMatr` с помощью двойного цикла перебираются элементы матрицы с преобразованием элементов путем смены знака. Описание дружественной операторной функции смены знака у элементов матрицы приведено в листинге 7.7.

Листинг 7.7. Операции смены знаков элементов матрицы

```
friend TMatrix& operator - (TMatrix &aMatr)
// дружественная функция для изменения знаков
// операнда  aMatr
{
     for (int i=0; i<aMatr.FRowCount; i++)
        for (int j=0; j<aMatr.FColCount; j++)
             aMatr.FData[i][j] = -aMatr.FData[i][j];
      return aMatr;
}
```

Операторная функция для перемножения матриц

Операция перемножения двух матриц – бинарная. Функция реализации этой операции определена как дружественная. Поэтому оба операнда – `aMatr` и `bMatr` – являются входными и представляют перемножаемые матрицы или сомножители. Для того чтобы сохранить результат умножения матриц, создается вспомогательная переменная `MBuf` классового типа.

В начале работы процедуры определяется соответствие размеров матриц, то есть определяется можно ли перемножать такие матрицы. Известно, что можно перемножать матрицы только в том случае, если число столбцов в первой матрице, то есть левом сомножителе, равно числу строк во второй матрице, то есть правом сомножителе.

Формулу для получения элемента, расположенного в `i`-строке и `j`-столбце матрицы-произведения можно записать в виде:

$$MBuf_{i,j} = \sum_{k=0}^{FRowCount-1} aMatr_{i,k} \cdot bMatr_{k,j} .$$

Для программирования всего процесса вычисления всех элементов результирующей матрицы необходимо использовать три цикла: два из них перебирают индексы элементов матрицы-произведения, а третий – реализует вычисление формулы. Результат перемножения передается через заголовок функции. Описание операторной функции, предназначенной для перемножения матриц, приведено в листинге 7.8.

Листинг 7.8. Дружественная операторная функция, предназначенная для перемножения двух матриц

```
TMatrix operator * (const TMatrix &aMatr,
                                    const TMatrix &bMatr)
{
  if (bMatr.FRowCount != aMatr.FColCount)
          throw ("Матрицы нельзя умножить! - Размеры!!!");
            // Создать вспомогательную матрицу
  TMatrixTask MBuf (aMatr.FRowCount, bMatr.FColCount);
            // Выполнить умножение
  for (int i=0; i<MBuf.FRowCount; i++)
    for (int j=0; j<MBuf.FColCount; j++)
     {
      MBuf.FData[i][j] = 0;
      for (int k=0; k<aMatr.FColCount; k++)
         MBuf.FData[i][j] +=
                    aMatr.FData[i][k]*bMatr.FData[k][j];
        }
   return MBuf; // Вернуть результат
}
```

Операторная функция для умножения числа на матрицу

Операция умножения матрицы на число – бинарная. Функция определена как дружественная. Поэтому оба операнда `aX` и `bMatr`, которые представляют перемножаемые число и матрицу, являются входными. Для того чтобы сохранить результат умножения числа на матрицу, создается вспомогательная переменная `MBuf` классового типа. Результат перемножения передается через заголовок функции. Описание операторной функции приведено в листинге 7.9.

Листинг 7.9. Дружественные операторные функции, предназначенные для умножения матрицы на число слева

```
TMatrix operator * (double aX, const TMatrix &bMatr)
// дружественная функция для умножения числа aX
// на матрицу bMatr
// MBuf – вспомогательная переменная для хранения
// результатов умножения
```

```
{
       // Создать вспомогательную матрицу
  TMatrixTask MBuf (bMatr.FRowCount, bMatr.FColCount);
      // Выполнить умножение
  for (int i = 0; i < MBuf.FRowCount; i++)
    for (int j = 0; j < MBuf.FColCount; j++)
      MBuf.FData[i][j] = aX * bMatr.FData[i][j];
  return MBuf;    // Возврат результата
};
```

Операторная функция для умножения матрицы на число

Операция умножения числа на матрицу – бинарная. Функция определена как дружественная. Поэтому оба операнда `aY` и `bMatr`, которые представляют перемножаемые число и матрицу, являются входными. Для того чтобы сохранить результат умножения числа на матрицу, создается вспомогательная переменная `MBuf` классового типа. Результат перемножения передается через заголовок функции. Описание операторной функции приведено в листинге 7.10.

Листинг 7.10. Дружественные операторные функции, предназначенные для умножения матрицы на число справа

```
TMatrix operator * (const TMatrix &aMatr, double aY)
// дружественная функция для умножения матрицы aMatr
// на число aY
// MBuf – вспомогательная переменная для хранения
// результатов умножения
{
     // Создать вспомогательную матрицу
  TMatrixTask MBuf (aMatr.FRowCount, aMatr.FColCount);
      // Умножение
  for (int i = 0; i < MBuf.FRowCount; i++)
       for (int j = 0; j < MBuf.FColCount; j++)
          MBuf.FData[i][j] = aMatr.FData[i][j] * aY;
  return MBuf; // Возврат результата
};
```

Транспонирование матрицы «на месте»

Операция транспонирования заключается в том, что строки матрицы заменяются ее столбцами, а столбцы – строками. Операция транспонирования «на месте» с преобразованием себя в транспонированную матрицу возможна только, если транспонируется квадратная матрица.

В процедуре транспонирования создается вспомогательная классовая переменная `bufMatr`. Эта матрица используется для временного хранения транспонированной матрицы. При объявлении этой переменной конструктору для ее создания передаются два параметра: число строк и число столбцов, взятые от исходной транспонируемой матрицы, на которую указывает `*this`.

Получение транспонированной матрицы сводится к поэлементному копированию, посредством двойного цикла, исходной матрицы во вспомогательную матрицу по типу «столбик – в строчку». После чего используется оператор присваивания классовых переменных для копирования вспомогательной матрицы в исходную матрицу. В качестве результата операции возвращается значение транспонированной матрицы. Код реализации метода `Transpose` приведен в листинге 7.11. Следует обратить внимание на оператор `(*this) = bufMatr;`, представленный в листин-

ге 7.11. Этот оператор допустим, так как операция присваивания классовых переменных определена в классе.

Листинг 7.11. Описание метода транспонирования матрицы по месту

```
TMatrix& TMatrix::Transpose()
// Если матрица неквадратная, то - ошибка.
// bufMatr - вспомогательная матрица
// Транспонированная матрица формируется сначала в bufMatr
// а затем – из bufMatr копируется в себя
{
    // Создать вспомогательную матрицу
  TMatrix bufMatr (FColCount, FRowCount);

    // скопировать транспонированную матрицу в bufMatr
  for (int i=0; i<FRowCount; i++)
    for (int j=0; j<FColCount; j++)
       bufMatr.FData[j][i] = FData[i][j];

    // скопировать вспомогательную bufMatr в себя
  (*this) = bufMatr;  // используется операция присваивания
  return (*this); // вернуть результат
}
```

Получение транспонированной матрицы

Метод имеет только один выходной параметр `aMatr`. Этот параметр классового типа, в нем получается транспонированная матрица. В отличие от операции «транспонирование на месте» в данном исполнении транспонированию может подлежать и прямоугольная матрица. В программе нет необходимости создавать вспомогательную переменную. Получение транспонированной матрицы сводится к поэлементному копированию, посредством двойного цикла, исходной матрицы в выходную матрицу по типу «столбик – в строчку». В качестве результата операции возвращается значение транспонированной матрицы.

Следует обратить внимание на тот факт, что в программе не проводится проверка на соответствие размеров матрицы. Такую проверку следует делать до вызова метода транспонирования. Описание функции `Transpose` приведено в листинге 7.12.

Листинг 7.12. Описание методов транспонирования матриц

```
TMatrix& TMatrix::Transpose (TMatrix &aMatr)
// транспонирование с формированием результата в новой
// матрице aMatr
{
  for (int i=0; i<FRowCount; i++)
     for (int j=0; j<FColCount; j++)
        aMatr.FData[j][i] = FData[i][j];
  return (aMatr);
}
```

Примечание

При анализе описаний реализаций методов транспонирования, представленных на листингах 7.11 и 7.12, следует обратить внимание на три факта.

1. Оба метода имеют одинаковые имена. Это допустимо, так как методы отличаются параметрами.

2. В теле метода «транспонирование на месте» используется операция присваивания. Это нормально, так как в любом классе методы могут вызывать друг друга.
3. Во втором методе транспонирования не потребовалась вспомогательная переменная, так как транспонированная матрица напрямую передается в выходной параметр – результат транспонирования.

Описание класса TMatrixEx

В этот расширенный класс `TMatrixEx` – потомок класса `TMatrix` добавлено поле `aTitle`, в котором можно хранить наименование объекта. Если пользователь не посчитает это целесообразным, то его значение задается «по умолчанию» – `"Temp"`.

В классе `TMatrixEx` предусмотрено также три конструктора. В данном случае они призваны для обеспечения корректной работы предка, поскольку именно ему необходимо передать параметры для конструирования объекта.

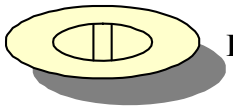
К числу операций добавлена операция присваивания. Эта операция необходима в тех случаях, когда требуется работать с операндами разных типов: `<TMatrixEx> = <TMatrix>`.

К методам класса добавлены следующие методы: вычисление определителя; получение обратной матрицы.

Интерфейс класса `TMatrixEx` представлен в листинге 7.13.

Листинг 7.13. Интерфейс расширенного класса – потомка от класса «Матрица вещественных чисел»

```
class TMatrixEx : public TMatrix
{
   protected:
     AnsiString aTitle;  // название объекта
   public:
     TMatrixEx (int, int, AnsiString ="Temp"); // конструктор
     TMatrixEx (const TMatrixEx &);//конструктор копирования
     TMatrixEx(); //пустой конструктор
     ~TMatrixEx(); // деструктор
     double DetMatr();  // возвращает значение определителя
     double Verify_Val (int aK);
     TMatrixEx& operator =(TMatrix );//оператор присваивания
     double InversMatr();  // вычисление обратной матрицы
 };
```



НАЗНАЧЕНИЕ МЕТОДОВ КЛАССА `TMatrixEx`:

- `TMatrixEx (int aInitRow, int aInitCol, AnsiString ="Temp");` – конструктор. Имеет три параметра. Два из них – число строк `aInitRow` и число столбцов `aInitCol` – предназначены для конструктора-предка. Третий параметр позволяет передать/инициализировать имя объекта или взять его по умолчанию `"Temp"`;
- `TMatrixEx (const TMatrixEx &aMatr);` – конструктор копирования. Конструктор имеет один параметр `aMatr` с типом ссылки на класс. Используется автоматически при вызове процедур с параметрами классового типа, передаваемыми «по значению»;

- `TMatrixEx();` – конструктор без параметров или, как его называют «по умолчанию». Задает матрицу с нулевыми размерами;
- `~TMatrixEx();` – деструктор. Высвобождает память, занятую объектом;
- `double DetMatr();` – метод класса. Вычисляет и возвращает значение определителя/детерминанта матрицы;
- `double Verify_Val (int aK);` – метод класса предназначен для поиска ненулевого диагонального элемента матрицы;
- `TMatrixEx& operator =(TMatrix);` – оператор присваивания;
- `double InversMatr();` – метод класса. Вычисляет обратную матрицу, которая помещается на место исходной матрицы `FData`. Через заголовок функции возвращает значение определителя исходной матрицы.

Конструктор копирования класса `TMatrixEx`

Поскольку в классе-потомке нет дополнительных полей, то этот конструктор только передает параметр конструктору предка. Описание конструктора копирования представлено в листинге 7.14.

Листинг 7.14. Конструкторы копирования класса-наследника от класса «Матрица расширенная»

```
TMatrixEx::TMatrixEx (const TMatrixEx &InitMatr) :
                                                TMatrix (InitMatr)
 // конструктор копирования
{
}
```

Конструктор без параметров класса `TMatrixEx`

Конструктор «по умолчанию» не имеет параметров. Поскольку в классе-потомке нет дополнительных полей, конструктор используется только для того, чтобы активировать конструктор предка. Описание конструктора представлено в листинге 7.15.

Листинг 7.15. Конструктор без параметров класса «Матрица расширенная»

```
TMatrixEx::TMatrixEx(): TMatrix ()
 // пустой конструктор
{
}
```

Конструктор класса `TMatrixEx` с параметрами-размерами

Этот конструктор имеет два параметра, значения которых передаются конструктору класса-предка. Описание конструктора представлено в листинге 7.16.

Листинг 7.16. Конструктор с параметрами класса «Матрица расширенная»

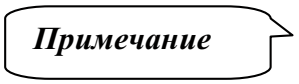
```
TMatrixEx::TMatrixEx (int aInitRow, int aInitCol,
            AnsiString aTitle): TMatrix (aInitRow, aInitCol)
// конструктор
{
}
```

Деструктор класса `TMatrixEx`

Деструктор используется только для того, чтобы активировать деструктор предка. Описание деструктора представлено в листинге 7.17.

Листинг 7.17. Деструктор класса «Матрица расширенная»

```
TMatrixEx::~TMatrixEx()
// деструктор
{
}
```



Как видно из описаний, представленных в листинге 7.9, у конструкторов в теле нет операторов. Их назначение – только передать свои параметры конструктору предка и активизировать его. Деструктор не имеет параметров и не содержит явную связь с деструктором предка. Он вызывается автоматически.

Оператор присваивания

Оператор присваивания представляет операцию присваивания, если операнды разных типов: `<TMatrixEx> = <TMatrix>`. Операция является бинарной. Первый операнд – текущая матрица, на которую ссылается указатель `this`, второй операнд – классовая переменная `aMatr`. Матрица текущего класса возвращается как результат функции. Алгоритм выполнения такой операции над матрицами сводится к следующему:

а) проверяются размеры матриц-операндов на соответствие, то есть сравниваются число строк и число столбцов;
б) если у первого операнда размеры не соответствуют второму, то перейти нужно к пункту *в*, иначе к пункту *е*;
в) высвобождается память, занятая под текущую матрицу;
г) заменяются размеры матрицы-приемника на размеры матрицы второго операнда;
д) отводится память в соответствии с новыми размерами;
е) выполняется копирование данных.



Следует отметить, что для *доступа к полям базового* класса при выполнении копирования данных (пункта *е*) в теле операции присваивания используются свойства класса!

Описание реализации оператора присваивания приведено в листинге 7.18.

Листинг 7.18. Описание оператора присваивания <TMatrixEx> = <TMatrix>

```
TMatrixEx& TMatrixEx::operator = (TMatrix aMatr)
{
      // проверка размеров
  if (FRowCount != aMatr.RowCount || FColCount !=
                                              aMatr.ColCount)
   {
// если размеры не совпадают, то освободить прошлую память
    for (int i=0; i<FRowCount; i++)
      delete[] FData[i];
    delete[] FData;

      // установить новые размеры матрицы - приемника
    FRowCount = aMatr.RowCount;
    FColCount = aMatr.ColCount;
```

```
      // отвести новую память
    FData = new double* [FRowCount];
    for (int i=0; i<FRowCount; i++)
      FData[i] = new double [FColCount];
    }
     // скопировать данные
  for (int i=0; i<FRowCount; i++)
    for (int j=0; j<FColCount; j++)
      FData[i][j] = aMatr.Data[i][j];

  return *this; // вернуть результат
}
```

Метод поиска ненулевого диагонального элемента `Verify_Val`

Этот метод используется при вычислении определителя матрицы в методе `DetMatr` для поиска ненулевого диагонального элемента. Имеется один входной параметр `k`, который задает номер диагонального элемента матрицы с целью проверки его на нулевое значение. Алгоритм сводится к следующему: проверяется указанный диагональный элемент на нулевое значение:

1) если диагональный элемент не равен нулю, то производится выход из процедуры с возвратом единичного значения;
2) если диагональный элемент равен нулю, то в цикле производится поиск первого по порядку ненулевого элемента в последующих строчках матрицы:
   а) если такая строка найдена, то найденная строка добавляется к строке с номером `k`; производится выход из процедуры с возвратом единичного значения;
   б) если такая строка не найдена, то производится выход из процедуры со значением 0.

Описание метода `Verify_Val` приведено в листинге 7.19.

Листинг 7.19. Описание метода, предназначенного для поиска ненулевого диагонального элемента матрицы

```
double TMatrixEx::Verify_Val (int k)
{
  int n = FColCount;
  if (FData[k][k]!=0) return 1.0;
  else
    for (int j=k+1; j<n; j++)
     {
      if (FData[j][k] != 0)
       {
         for (int i=0; i<n; i++)
           FData[k][i] += FData[j][i];
         return 1.0;
       }
      else continue;
     }
  return 0;
}
```

Вычисление определителя матрицы

Метод `DetMatr` не имеет параметров. Он используется для вычисления определителя матрицы. Полученное значение определителя возвращается в качестве результата через заголовок метода. В алгоритме используется вспомогательная пере-

менная `bufMatr` с типом класса. Использование вспомогательной переменной позволяет сохранить неизменность исходной матрицы, так как по логике работы алгоритма текущая матрица в процессе вычисления определителя подлежит преобразованию. Описание алгоритма приведено в листинге 7.20. Подробности описания алгоритма можно посмотреть в книге Касьянова и Сабельфельда [12].

Листинг 7.20. Описание метода вычисления определителя матрицы

```
double TMatrixEx::DetMatr()
{
  // Объявление вспомогательной переменной.
  TMatrixEx  bufMatr(*this);

  int k, i, j , n=FColCount;
  double el1, el2, sign;
  double det = 1;
  for (k=0; k<n-1; k++)
   {
    if ((sign=Verify_Val(k))==0)
      return 0;
    else
       det *= bufMatr.FData[k][k]*sign;

       el1 = bufMatr.FData[k][k];
       for (i=k; i<n; i++)
         bufMatr.FData[i][k]/=el1;
       for (j=k+1; j<n; j++)
         {
           el2 = bufMatr.FData[k][j];
           for (i=k; i<n; i++)
             bufMatr.FData[i][j] -= bufMatr.FData[i][k]*el2;
         }
    }
  det *= bufMatr.FData[n-1][n-1];
  return det;
}
```



Вспомогательная переменная `bufMatr`, используемая в описании метода `DetMatr`, создается *с помощью конструктора копирования*, а именно: непосредственным его вызовом. Это в данном случае является удобным, так как конструктор не только решает задачу отведения памяти, но и копирует значения элементов (из `*this`) в `bufMatr`.

Вычисление обратной матрицы

Данный метод не имеет параметров. Алгоритм построен таким образом, что в результате его работы исходная матрица преобразуется в матрицу обратную. Поэтому, если в работе требуется использование одновременно исходной матрицы и обратной матрицы, то *рекомендуется перед вызовом метода сохранить значение исходной матрицы во вспомогательной* переменной классового типа. Вычисленное значение определителя матрицы возвращается через заголовок функции. Описание метода `InversMatr` – вычисления обратной матрицы – приведено в листинге 7.21. С алгоритмическими подробностями можно познакомиться в книге Касьянова и Сабельфельда [12].

Листинг 7.21. Описание метода вычисления обратной матрицы

```
double TMatrixEx::InversMatr()
{
  int n = FColCount;
  int i, j, k, ij, fl, *index;
  double r, d, w, eps = 1e-15;
  index = new int [n];
  r=1; d=1;
  for (i=0; i< n; i++)
   {
    fl=0;
    for (j=i; j< n; j++)
     {
      if (fabs (FData[j][i]) > eps)
       {
        fl=1;
        for (k=0; k< n; k++)
         {
          double Z = FData[i][k];
          FData[i][k] = FData[j][k];
          FData[j][k] = Z;
         }
        index[i] = j;
        break;
       }
     }
    if (FData[i][j] == 0) return 0;
    FData[i][i] = 1.0/FData[i][i];
    for (k=0; k< n; k++)
     {
      if (k != i)
      FData[i][k] *= FData[i][i];
     }
    for (ij=0; ij< n; ij++)
     {
      if (ij != i)
       {
        w = -FData[ij][i];
        FData[ij][i]=0;
        for (k=0; k< n; k++)
        FData[ij][k] += w*FData[i][k];
       }
      }
    if (fl==0)
      return 0;
    d = d/FData[i][i];
   }
  for (i=n-1; i>=0; i--)
   {
    if (index[i] != i)
     {
      r = -r;
      for (k=0; k< n; k++)
       {
        double Z = FData[k][i];
        FData[k][i] = FData[k][index[i]];
        FData[k][index[i]] = Z;
       }
```

```
      }
    }
  delete[] index;
  return (r*d);
}
```

Описание класса TMatrixTask

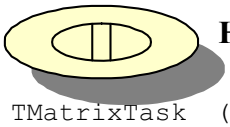
Класс-наследник `TMatrixTask` специализирован под среду *C++ Builder* и содержит методы, связанные с вводом и отображением информации в поля визуальных компонент.

В число конструкторов, помимо тех, которые необходимы для работы классов-предков, включен конструктор, который позволяет создать матрицу на основе содержимого сетки – компонента `StringGrid`. Добавлена также операция присваивания и два метода: загрузить значения из сетки `StringGrid` в матрицу; загрузить значения из матрицы в сетку `StringGrid`. Интерфейс класса представлен в листинге 7.22.

Листинг 7.22. Интерфейс класса-потомка `TMatrixTask`

```
class TMatrixTask: public TMatrixEx
{
  public:
   TMatrixTask(int, int, AnsiString="Temp");  // конструктор
   TMatrixTask(const TMatrixTask&);  // конструктор копирования
   TMatrixTask();  // пустой конструктор
   TMatrixTask(TStringGrid*);  // конструктор
   ~TMatrixTask();  // деструктор
   TMatrixTask& operator = (TMatrix&);  // присваивание
   void LoadFromGrid (TStringGrid*);  // взять данные из сетки

   void SaveToGrid (TStringGrid *, int = 6, int = 3);
                                 // перенести из матрицы в сетку
};
```



НАЗНАЧЕНИЕ МЕТОДОВ КЛАССА `TMatrixTask`:

- `TMatrixTask (int aInitRow, int aInitCol, AnsiString = "Temp");` – конструктор. Имеет три параметра. Два из них – число строк `aInitRow` и число столбцов `aInitCol` – предназначены для передачи в конструктор предка. Третий параметр – имя объекта – также передается предку;
- `TMatrixTask (const TMatrixTask& aMatr);` – конструктор копирования. Имеет один параметр – ссылку на `aMatr`, который передается конструктору предка;
- `TMatrixTask();` – пустой конструктор. Необходим для инициирования выполнения соответствующего конструктора предка;
- `TMatrixTask (TStringGrid* aGrid);` – конструктор. Имеет один параметр `aGrid` – указатель на визуальный компонент-сетку. Позволяет взять размеры матрицы из сетки и передать их в качестве параметров (число строк/столбцов) конструкторам предка;
- `~TMatrixTask();` – деструктор. Вызывает деструкторы предков;
- `TMatrixTask& operator = (TMatrix& aMatr);` – операторная функция присваивания;

- `TMatrixTask& operator = (TMatrix);` – операторная функция присваивания;
- `void LoadFromGrid (TStringGrid* aGrid);` – метод класса. Имеет единственный параметр: указатель на визуальный компонент типа «сетка». Предназначен для того, чтобы взять данные из сетки `aGrid` и занести их в область памяти матрицы `FData`;
- `void SaveToGrid (TStringGrid * aGrid, int aPrec = 6, int aDig = 3);` – метод класса. Имеет три параметра: один параметр `aGrid` – это указатель на визуальный компонент типа «сетка». Он предназначен для того, чтобы взять значения элементов матрицы `FData` и занести их в сетку `aGrid`. Два других: `aPrec` – определяет количество разрядов, отведенных под число при записи его в ячейку сетки; `aDig` – количество знаков после запятой. Значения последних параметров можно взять/не взять по умолчанию.

Конструктор класса `TMatrixTask` с параметрами, определяющими размеры матрицы

Листинг 7.23. Описание конструктора `TMatrixTask`

```
TMatrixTask::TMatrixTask (int aInitRow, int aInitCol,
        AnsiString aTitle): TMatrixEx (aInitRow, aInitCol)
// конструктор
{
}
```

Деструктор класса

Листинг 7.24. Описание деструктора

```
TMatrixTask::~TMatrixTask()
// деструктор
{
}
```

Чтение матрицы из сетки

Метод `LoadFromGrid` предназначен для чтения значений элементов матрицы из сетки `aGrid` типа `TStringGrid`. Описание метода представлено в листинге 7.26.

Листинг 7.26. Описание метода, предназначенного для чтения значений элементов матрицы `FData` из сетки `aGrid`

```
void TMatrixTask::LoadFromGrid (TStringGrid *aGrid)
// загрузка данных из сетки aGrid в область памяти FData
{
 for (int i=0; i<FRowCount; i++)
   for (int j=0; j<FColCount; j++)
     FData[i][j] = StrToFloat (aGrid->Cells[j+1][i+1]);

}
```

Запись матрицы в сетку

Метод `SaveToGrid` предназначен для противоположной операции и имеет три параметра. Первый параметр `aGrid` определяет сетку, куда записываются значения элементов матрицы. Второй и третий параметры используются в функции `FloatToStrF`, предназначенной для преобразования числовых значений элементов матрицы в строку `AnsiString` для передачи преобразованных значений в ячейки сетки. Параметр `aPrec` – определяет общее количество разрядов, а `aDig` – количество знаков после запятой. Код реализации метода представлен в листинге 7.27.

Листинг 7.27. Описание метода, предназначенного для записи значений элементов матрицы `FData` в ячейки сетки `aGrid`

```
void TMatrixTask::SaveToGrid (TStringGrid *aGrid, int aPrec, int aDig)
// отображение матрицы в StringGrid
{
  for (int i=0; i<FRowCount; i++)
    for (int j=0; j<FColCount; j++)
      aGrid->Cells[j+1][i+1] = FloatToStrF (FData[i][j],
                                    ffFixed, aPrec, aDig);
}
```

7.2. Разработка интерфейса приложения для решения задач с классом «Матрица»

Основная идея, положенная в основу построения интерфейса данного приложения, исходит от желания тратить как можно меньше усилий на перестройку/модификацию главной формы при добавлении к программе решения очередной задачи. Потому принята стратегия – на главной форме создать окно, в котором бы отражались в виде списка названия всех созданных вторичных форм проекта *по принципу*: добавили новую форму для решения новой задачи – автоматически в окно добавилась новая строка. И далее остается лишь реализовать механизм, который бы позволял выделить в этом списке нужную форму таким образом, чтобы сфокусированное в списке название формы (название отражает смысл задачи) позволяло бы активизировать/открыть эту форму по щелчку на кнопке «Открыть». Общий вид главной формы представлен на рис. 7.1.

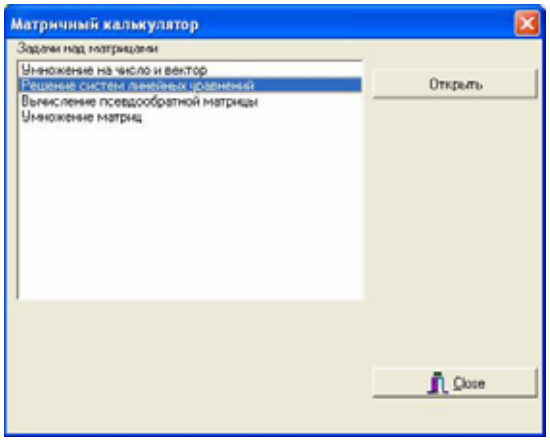


Рис. 7.1. Главная форма, представляющая приложение для решения задач с использованием классов «Матрицы»

Согласно рис. 7.1 на главной форме с именем `foMain` приложения всего 4 компонента:

1) компонент `Label` – для поясняющего текста;
2) компонент-список `ListBox` с именем `lbTasks` – для хранения списка задач, то есть заголовков вторичных форм;
3) 2 компонента-кнопки *`BitBtn`*:
   а) с именем `bbTask` – для перехода к решению выбранной задачи;
   б) с именем `bbClose` – чтобы закрыть приложение.

Для того чтобы приложение могло функционировать в рамках описанной стратегии, нужно предусмотреть обработчики для следующих событий:

- `void __fastcall TfoMain::FormShow(TObject *Sender);` – обработчик события `OnShow` главной формы. Позволяет получить список задач/форм и занести его в `lbTasks` сразу при запуске программы, непосредственно перед тем, как показать главную форму;
- `void __fastcall TfoMain::lbTasksDblClick(TObject *Sender);` – обработчик события `OnDblClick` компонента `lbTasks`. Позволяет выделить строку в списке `lbTasks`, связать ее с вторичной формой, которая соответствует сфокусированной задаче, а также открыть ее двойным щелчком мыши;
- `void __fastcall TfoMain::bbTaskClick (TObject *Sender);` – обработчик события `OnClick` компонента `bbTask`. Позволяет перейти к решению выбранной задачи: выполнить команду «Открыть», которая откроет вторичную форму, соответствующую выделенной строке в списке и предназначенную для решения этой задачи.

Обработчик события показа формы

Обработчик события `OnShow` главной формы создается для выполнения необходимых действий перед отображением ее на экране. Здесь этот обработчик используется для формирования списка вторичных форм с целью его отображения на форме. Для того чтобы достигнуть эффекта отображения списка вторичных форм в окне *`lbTasks`* главной формы, необходимо выполнение следующих шагов.

1. Опрашиваются ресурсы/компоненты проекта. Каждый из ресурсов (с именем `Components[i]`) описан в *C++ Builder* соответствующим классовым типом. Число таких ресурсов/компонентов определяется из свойства `ComponentCount`, характеризующего приложение/проект. Само приложение имеет стандартное имя – `Application`.
2. Выбираются из компонентов/ресурсов те, которые являются компонентами типа форма (потомками класса `TForm`), но в то же время не являются главной формой. На основании такого выбора и определяется, является ли компонент вторичной формой, и если это так, то заголовок формы добавляется к списку `lbTasks`.

В алгоритме используется вспомогательная переменная `classRef`, куда заносится тип класса компоненты. Описание кода обработчика события приведено в листинге 7.28.

Листинг 7.28. Описание обработчика события `OnShow` главной формы, предназначенного для опроса ресурсов и формирования `lbTasks`

```
void __fastcall TfoMain::FormShow(TObject *Sender)
// Application->ComponentCount – общее количество
// компонентов классовых типов в приложении;
{
      // выполнить цикл по числу компонент
  for (int i = 0; i < Application->ComponentCount; i++)
   {
    TClass classRef = Application->Components[i]->ClassType();

// Выполняется проверка, есть ли такой объект и является
// ли его родителем класс TForm и если это так, то
```

```
// уточняется, что это не сама главная форма (this – эта)
    if (classRef != NULL
        && (classRef = classRef->ClassParent())->
                                                ClassNameIs("TForm")
        && Application->Components[i] != this)
// Если так, то заголовок
// (Caption) компоненты добавить к lbTask (имя ListBox)
      lbTasks->
            Items->AddObject (((TForm *)Application->
                               Components[i])->Caption,
                              Application->Components[i]);
   }
}
```

Обработчик события для перехода к вторичной форме через список окон

В алгоритме этого обработчика события проверяется, имеется ли список вторичных форм (или список задач). Если список не пуст, то он показывает объект-форму с номером, соответствующим выделенной строке списка. Описание обработчика события приведено в листинге 7.29.

Листинг 7.29. Описание обработчика события `OnDblClick` компонента главной формы – `ListBox`

```
void __fastcall TfoMain::lbTasksDblClick(TObject *Sender)
{ if (lbTasks->ItemIndex >= 0)
   {((TfoTask1 *)lbTasks->Items->Objects [lbTasks->ItemIndex])->Show();
   }
}
```

Обработчик события для перехода к вторичной форме по кнопке

В алгоритме обработчика события проверяется, имеется ли выделенная строка в списке. Косвенно этим же определяется, имеется ли список вторичных форм (или список задач). Если список не пуст и есть выделенная строка, то данный обработчик показывает объект-форму с номером, соответствующим выделенной строке списка. Код приведен в листинге 7.30.

Листинг 7.30. Описание обработчика события `OnClick` «открыть» компонента `Button`.

```
void __fastcall TfoMain::bbTaskClick (TObject *Sender)
{ if (lbTasks->ItemIndex >= 0)
   {((TfoTask1 *)lbTasks->Items->Objects
                              [lbTasks->ItemIndex])->Show();
   }
}
```

Описанная стратегия позволяет каждую задачу проекта решать на отдельной форме. Причем интерфейс и функциональность этой формы разрабатываются независимо как друг от друга, так и от главной формы.

ДЛЯ РЕШЕНИЯ НОВОЙ ЗАДАЧИ

В данном приложении для решения новой задачи требуется *подсоединить к проекту* отдельную форму. Эта форма как бы будет вторичной, она должна представлять пользователю интерфейс для решения задачи. Название этой формы автоматически отразится в списке

задач – окне компонента `lbTask`. А выбор на этапе выполнения элемента списка автоматически предоставит пользователю созданную пользователем форму для работы.

7.3. Решение задач линейной алгебры с использованием типа «Матрица»

Задача 7.1. Требуется разработать программу «Калькулятор», которая должна предоставлять пользователю возможность выполнения следующих операций над матрицами: умножать матрицу на число слева/справа, умножить матрицу на вектор слева/справа.

Решение

Решение задачи не требует разработки дополнительных методов. В числе методов класса есть методы умножения матриц и умножения матрицы на число. А задачу умножения матрицы на вектор можно рассматривать как частный случай перемножения матриц, когда один из операндов представляет собой матрицу с одним столбиком/строкой. Поэтому для данной задачи наиболее трудоемкой частью будет *разработка надлежащего интерфейса* и обеспечение его функциональности.

На форме для решения задачи *пользователю нужно предоставить возможность* вводить исходные данные: элементы матрицы, элементы вектора, число. Для этого на форме должны быть размещены соответствующие поля – визуальные компоненты. Требуются также *компоненты для отображения результата.* Желательно предусмотреть «инструменты», которые помогли бы пользователю *избежать возможных ошибок*. Это касается как ошибок, связанных с несоответствием размеров перемножаемых матриц, так и ошибок при наборе данных. На рис. 7.2 представлено рабочее окно программы «Калькулятор». Для построения этого окна потребуется:

1) 4 компонента `Label` – для написания поясняющих текстов;
2) 3 компонента – кнопки `SpeedButton` с именами (свойство `Name`):
   а) `sbHandle` – «Ручной ввод»;
   б) `sbLeft` – «Слева»;
   в) `sbRight` – «Справа»;
3) 3 компонента – кнопки `BtnBtn` с именами:
   а) `bbSetSize` – «Установить размеры»;
   б) `bbControlValue` – «Проверить значения»;
   в) `bbMiltiply` – «Умножить»;
4) компонент `Edit` с именем `edValue` для ввода числа;
5) 2 компонента `CSpinEdit` с именами: `cseColCount`, `cseRowCount` – для задания целочисленных значений – размеров матрицы;
6) 3 компонента – переключателя `RadioButton` с именами:
   а) `rbValue` – для включения режима «Множитель»;
   б) `rbVectorRow` – для включения режима «Вектор-строка»;
   в) `rbVectorCol` – для включения режима «Вектор-столбец»;
7) 4 компонента – сетки `StringGrid` с именами:
   а) `sgVectorRow` – для задания вектора-строки;
   б) `sgVectorCol` – для задания вектора-столбца;
   в) `sgMatrixSource` – для задания исходной матрицы;

г) `sgMatrixResult` – для отображения результирующей матрицы.

Кнопки обеспечивают функциональность приложения.

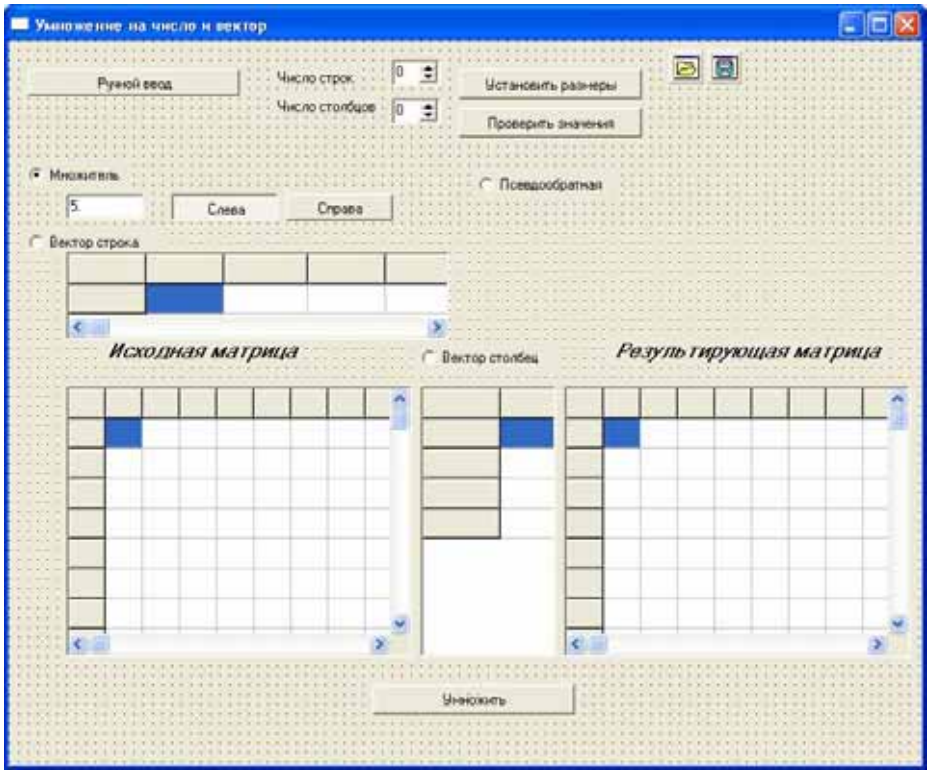


Рис. 7.2. Главная форма, представляющая приложение для работы с матрицами

При разработке приложения будем придерживаться стратегии, в соответствии с которой пользователю будет разрешено перейти к выполнению очередного шага только в том случае, если предыдущий шаг произведен корректно.

Замысел

Замысел состоит в том, чтобы пользователя последовательно провести по этапам:

а) установить размеры;
б) проверить значения;
в) умножить.

При этом элементы на форме будут переходить из состояния в состояние следующим образом.

1. На этапе выполнения при выборе данной формы в начальном ее состоянии все кнопки окна за исключением кнопки `sbHandle` «Ручной ввод» *будут не активны*.
2. Очередная кнопка становится *активной* только в том случае, если выполнилась предыдущая команда, причем корректно.
3. После того, как пользователь выполнит команду `sbHandle` «Ручной ввод», *активизируется* кнопка `bbSetSize` «Установить размеры», а также элементы `cseColCount`, `cseRowCount` – для установки числа строк и столбцов исходной матрицы. На этом этапе пользователю следует сделать выбор варианта для

умножения. Выбор этот обеспечивается с помощью активации нужной кнопки RadioButton:

а) умножить на число (слева или справа) – кнопка rbValue «Множитель»;

б) умножить на вектор-строку – кнопка rbVectorRow «Вектор-строка»;

в) умножить на вектор-столбец – кнопка rbVectorCol «Вектор-столбец».

Установка размеров и выбора типа умножения создает начальные условия для команды bbSetSize – «Установить размеры». Эта команда на основании установленных значений приведет в соответствие размеры результирующей и исходной матриц и вектора.

4. После этого становится доступной команда bbControlValue «Проверить значения». Но перед тем как выполнить эту команду, необходимо в сетки *занести значения* элементов, а если выбрана команда «Умножение на число», то в соответствующее окно следует вписать нужное значение множителя.
5. Команда bbControlValue «Проверить значение» проверяет, правильно ли пользователем вписаны значения в соответствующие поля. То есть проверяется все ли значения, вписанные в ячейки сеток, являются корректной записью числа.
6. Если нет ошибок в исходных данных, активизируется кнопка bbMiltiply «Умножить».

Чтобы обеспечить приложению функциональность такого рода, целесообразно ввести глобальные переменные:

FMatrixSource – для хранения исходной матрицы;

FMatrixResult – для хранения результирующей матрицы;

FVectorRow, FVectorCol – для хранения размеров матрицы.

А также необходимо подготовить следующие обработчики событий:

- `void __fastcall TfoTask1::FormCreate(TObject *Sender);` – обработчик события OnCreate формы для приведения кнопок и компонентов в неактивное стартовое состояние.
- `void __fastcall TfoTask1::sbHandleClick (TObject *Sender);` – обработчик события OnClick кнопки «Ручной ввод».
- `void __fastcall TfoTask1::bbSetSizeClick(TObject *Sender);` – обработчик события OnClick кнопки «Установить размеры».
- `void __fastcall TfoTask1::bbControlValuesClick(TObject *Sender);` – обработчик события OnClick кнопки «Проверить значения».
- `void __fastcall TfoTask1::bbMultipleClick(TObject *Sender);` – обработчик события OnClick кнопки «Умножить».

Обработчик события OnCreate формы, предназначенного инициализировать стартовые значения полей

Как это следует из описанного выше плана, на старте программы в *активном состоянии должна находиться только одна кнопка* sbHandle «Ручной ввод», а остальные кнопки должны быть неактивны.

Алгоритм сводится к инициализации глобальных переменных и приведению кнопок в неактивное состояние. Описание обработчика события приведено в листинге 7.31.

Листинг 7.31. Текст обработчика события «на создание формы»

```
void __fastcall TfoTask1::FormCreate(TObject *Sender)
// Инициализация полей класса
// Приведение кнопок и компонентов в неактивное состояние
{
     FMatrixSource = NULL;
     FVectorRow = NULL;
     FVectorCol = NULL;
     FMatrixResult = NULL;
     cseRowCount->Enabled = sbHandle->Down;
     cseColCount->Enabled = sbHandle->Down;
     bbSetSize->Enabled = sbHandle->Down;
     bbControlValues->Enabled = sbHandle->Down;
     bbMultiple->Enabled = false;
}
```

Обработчик события `OnClick` кнопки «Ручной ввод»

Алгоритм сводится к активизации кнопки `bbSetSize` «Установить размеры», а также к активации элементов `cseColCount`, `cseRowCount` для установки числа строк и столбцов исходной матрицы. Описание обработчика события `OnClick` кнопки «Ручной ввод» приведено в листинге 7.32.

Листинг 7.32. Текст обработчика события на нажатие кнопки «Ручной ввод»

```
void __fastcall TfoTask1::sbHandleClick (TObject *Sender)
// команда ручной ввод
{
  cseRowCount->Enabled = sbHandle->Down;
  cseColCount->Enabled = sbHandle->Down;
  bbSetSize->Enabled = sbHandle->Down;
  rbValue->Enabled = true;
  rbVectorRow->Enabled = true;
  rbVectorCol->Enabled = true;
  bbMultiple->Enabled = false;
}
```

Обработчик события `OnClick` кнопки «Установить размеры»

В тексте обработчика устанавливаются размеры исходной матрицы и вектора (значения берутся из визуальных компонент). Определяются размеры матрицы – результата в зависимости от выбора операции. Активизируется кнопка `bbControlValue` «Проверить значения». Описание обработчика события приведено в листинге 7.33.

Листинг 7.33. Текст обработчика события на нажатие кнопки «Установить размеры»

```
void __fastcall TfoTask1::bbSetSizeClick(TObject *Sender)
{
       // матрица исходная
  sgMatrixSource->RowCount = cseRowCount->Value + 1;
  sgMatrixSource->ColCount = cseColCount->Value + 1;

       // вектор
  sgVectorRow->ColCount = cseColCount->Value;
  sgVectorCol->RowCount = cseRowCount->Value;

        // матрица результирующая
  if (rbValue)
```

```
   {
     sgMatrixResult->RowCount = cseRowCount->Value + 1;
     sgMatrixResult->ColCount = cseColCount->Value + 1;
   }
  else
   {
     if (rbVectorRow)
      {
        sgMatrixResult->RowCount = 1 + 1;
        sgMatrixResult->ColCount = cseColCount->Value + 1;
      }
     else
       if (rbVectorCol)
        {
          sgMatrixResult->RowCount = cseRowCount->Value+1;
          sgMatrixResult->ColCount = 1 + 1;
        }
   }

  cseRowCount->Enabled = false;
  cseColCount->Enabled = false;
  sbHandle->Enabled = false;
  bbSetSize->Enabled = false;
  bbControlValues->Enabled = true;
}
```

Обработчик события `OnClick` кнопки «Проверить значения»

Кнопка `bbControlValue` «Проверить значения» может быть нажата только после того, как будут введены числовые значения исходных данных в отведенные им поля. Контроль правильности введенных значений производится в следующем порядке:

1) проверяются значения элементов матрицы: если ошибка, то – выход;
2) проверяется множитель: если ошибка, то – выход;
3) проверяется вектор-строка или вектор-столбец: если ошибка, то – выход;
4) если ошибок не обнаружено, то
   а) создаются экземпляры глобальных переменных;
   б) отводится память под глобальные переменные, и присваиваются значения элементам;
   в) активизируется кнопка `bbMiltiply` «Умножить».

Описание обработчика события `OnClick` кнопки «Проверить значения» приведено в листинге 7.34.

Листинг 7.34. Текст обработчика события на нажатие кнопки «Проверить значения»

```
void __fastcall TfoTask1::bbControlValuesClick (TObject *Sender)
// проконтролировать значения матриц, заполненных вручную
{
 double aux;
    // матрица исходная
 for (int irow = 1; irow < sgMatrixSource->RowCount; irow++)
   for (int icol = 1; icol < sgMatrixSource->ColCount; icol++)
                  //если попытка преобразовать строку
                  // ячейки в число неудачна
     if (!TryStrToFloat (sgMatrixSource->Cells [icol][irow], aux))
        { // выдать сообщение о неудаче
```

```
        sgMatrixSource->Row = irow;
        sgMatrixSource->Col = icol;
        MessageDlg ("Матрица содержит некорректное данное!",
                    mtError, TMsgDlgButtons() << mbOK, 0);
        return; // вернуться
      }
         // если установлен режим умножения на число
if (rbValue->Checked)
               // если попытка преобразовать строку
               // ячейки в число неудачна
  if (!TryStrToFloat (edValue->Text, aux))
    {// выдать сообщение о неудаче
      MessageDlg
          ("Множитель содержит некорректное данное!",
                  mtError, TMsgDlgButtons() << mbOK, 0);
      edValue->SetFocus(); // фокус на редакторе
      return; // вернуться
    }
          // если установлен режим умножения на вектор
if (rbVectorRow->Checked)
  for (int icol = 1; icol < sgVectorRow->ColCount; icol++)
               // если попытка преобразовать строку
               // ячейки в число неудачна
    if (!TryStrToFloat
                     (sgVectorRow->Cells [icol][1], aux))
      {// выдать сообщение о неудаче
         sgVectorRow->Row = 1;
        sgVectorRow->Col = icol;
        MessageDlg
          ("Вектор-строка содержит некорректное данное!",
                    mtError, TMsgDlgButtons() << mbOK, 0);
       return; //вернуться
      }
if (rbVectorCol->Checked)
  for (int irow = 1; irow < sgVectorCol->RowCount; irow++)
    if (!TryStrToFloat
                     (sgVectorCol->Cells [1][irow], aux))
      {
       sgVectorCol->Row = irow;
       sgVectorCol->Col = 1;
       MessageDlg
          ("Вектор-столбец содержит некорректное данное!",
                   mtError, TMsgDlgButtons() << mbOK, 0);
       return;
      }

 // все нормально; порождаем и заполняем матрицу и векторы

  // матрица исходная
if (FMatrixSource != NULL) delete FMatrixSource;
FMatrixSource = new TMatrixTask
    (sgMatrixSource->RowCount - 1,
                                  sgMatrixSource->ColCount - 1);
FMatrixSource->LoadFromGrid (sgMatrixSource);
  // простое число
if (rbValue->Checked)
  FMultiply = StrToFloat (edValue->Text);
  // векторы
```

```
  if (rbVectorRow->Checked)
   {
     if (FVectorRow != NULL) delete FVectorRow;
     FVectorRow = new TMatrixTask
                                    (1, sgVectorRow->ColCount - 1);
     FVectorRow->LoadFromGrid (sgVectorRow);
   }
  if (rbVectorCol->Checked)
    {
     if (FVectorCol != NULL) delete FVectorCol;
     FVectorCol = new TMatrixTask (sgVectorCol->RowCount - 1, 1);
     FVectorCol->LoadFromGrid (sgVectorCol);
    }
  bbControlValues->Enabled = false;
  rbValue->Enabled = false;
  rbVectorRow->Enabled = false;
  rbVectorCol->Enabled = false;

  bbMultiple->Enabled = true;
}
```

Обработчик события `OnClick` кнопки «Умножить»

Алгоритм определяется следующей последовательностью действий: создается экземпляр класса с соответствующими размерами – матрица `FMatrixResult`; уточняется тип выполняемой операции путем анализа: какая из кнопок нажата, и, в зависимости от выбора, вызывается нужный метод умножения, то есть требуемая операция; результат умножения заносится в матрицу результата и в сетку, предназначенную для отображения результата. Код реализации обработчика приведен в листинге 7.35.

Листинг 7.35. Текст обработчика события `OnClick` на нажатие кнопки `bbMiltiply` «Умножить»

```
void __fastcall TfoTask1::bbMultipleClick(TObject *Sender)
  // выполнение операции умножения
{
  if (FMatrixResult != NULL) delete FMatrixResult;
  FMatrixResult = new TMatrixTask (sgMatrixResult->
                  RowCount - 1, sgMatrixResult->ColCount - 1);
            // если режим «Множитель»
  if (rbValue->Checked)
   {double aux = StrToFloat (edValue->Text);//взять число
    if (sbLeft->Down) // если режим «Слева»
     {
                  // умножить на переменную слева
       *FMatrixResult = aux * (*FMatrixSource);
     }
    else // иначе
     {
              // умножить на переменную справа
      *FMatrixResult = (*FMatrixSource) *aux;
     }
   }
     // если режим «Вектор-строка»
  if (rbVectorRow->Checked)
   {
            // умножить строку на матрицу
    *FMatrixResult = *FVectorRow * (*FMatrixSource);
```

```
   }
      // если режим «Вектор-столбец»
  if (rbVectorCol->Checked)
   {
      // умножить матрицу на столбец
    *FMatrixResult = *FMatrixSource * (*FVectorCol);
   }
      // установить размеры сетки - результата
  sgMatrixResult->RowCount = FMatrixResult->RowCount + 1;
  sgMatrixResult->ColCount = FMatrixResult->ColCount + 1;
      // загрузить матрицу-результат в сетку
  FMatrixResult->SaveToGrid (sgMatrixResult);
}
```

Задача 7.2. Требуется разработать приложение для решения задачи перемножения матриц.

Решение

Так как в числе методов класса имеется метод, обеспечивающий перемножение матриц, для решения задачи необходимо всего три шага: добавить к проекту новую форму; разработать ее интерфейс; обеспечить функциональность таким образом, чтобы пользователь мог задавать элементы матриц-сомножителей, перемножать заданные матрицы и получать результирующую матрицу.

Примечание

В данной задаче, как и во многих других, разумно воспользоваться тем замыслом, который был описан в предыдущей задаче: провести пользователя последовательно по этапам: установить размеры; проверить значения; умножить. Объединим эти этапы общим названием «настройка решения».

Тогда на форму, предназначенную для решения задачи, помимо элементов, связанных с «настройкой решения», требуется поместить компоненты для ввода/вывода матриц. С целью разнообразить дизайн и желанием сэкономить/уменьшить рабочее пространство, воспользуемся компонентом `PageControl`. Предусмотрим в `PageControl` две страницы, на каждой из которых будем отображать элементы матриц-сомножителей. Интерфейс решения задачи представлен на рис. 7.3.

Как это показано на рис. 7.3, элементы «Настройки решения» некоторым образом перестроились по отношению к предыдущему проекту. Часть элементов была удалена за ненадобностью, часть элементов добавилась: элемент `SpinEdit` для задания размерности матриц. Выведенные размерности предоставляют пользователю возможность уточнить размеры перемножаемых матриц, а на основании этих значений определяются размеры для матрицы-результата.

Выполнение действий по решению задачи или функциональность приложения обеспечивается 5 событиями, 4 из которых подложены под кнопки: «Ручной ввод», «Установить размеры», «Проверить значения», «Умножить». И одно событие на `OnCreate` формы предназначено для того, чтобы выполнить стартовые установки. Тексты обработчиков представлены в листингах 7.36–7.40.

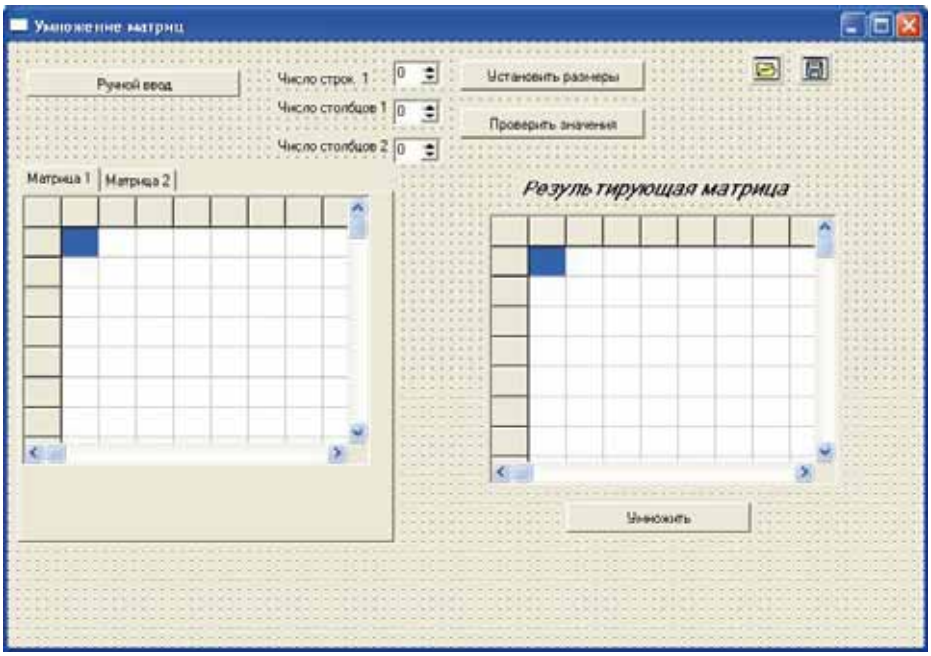


Рис. 7.3. Главная форма, представляющая приложение для умножения матриц

Замечание

Ниже приведены только тексты обработчиков событий. Обозначения данных такие же, как в предыдущей задаче 7.1. Там же можно посмотреть подробные пояснения.

Листинг 7.36. Текст обработчика события `OnCreate` формы

```
void __fastcall TfoTask2::FormCreate(TObject *Sender)
{ FMatrix1 == NULL;
  FMatrix2 == NULL;
  FMatrixRes == NULL;
  cseRowCount1->Enabled = sbHandle->Down;
  cseColCount1->Enabled = sbHandle->Down;
  cseColCount2->Enabled = sbHandle->Down;
  bbSetSize->Enabled = sbHandle->Down;
  bbControlValues->Enabled = sbHandle->Down;
}
```

Листинг 7.37. Текст обработчика события `OnClick` кнопки «Ручной ввод»

```
void __fastcall TfoTask2::sbHandleClick(TObject *Sender)
{   // команда ручной ввод
  cseRowCount1->Enabled = sbHandle->Down;
  cseColCount1->Enabled = sbHandle->Down;
  cseColCount2->Enabled = sbHandle->Down;
  bbSetSize->Enabled = sbHandle->Down;
}
```

Листинг 7.38. Текст обработчика события `OnClick` кнопки «Установить размеры»

```
void __fastcall TfoTask2::bbSetSizeClick(TObject *Sender)
{  // установить размеры матриц
    // матрица 1
  sgMatrix1->RowCount = cseRowCount1->Value + 1;
  sgMatrix1->ColCount = cseColCount1->Value + 1;
```

```
    // матрица 2
  sgMatrix2->RowCount = cseColCount1->Value + 1;
  sgMatrix2->ColCount = cseColCount2->Value + 1;
    // матрица 3
  sgMatrixResult->RowCount = cseRowCount1->Value + 1;
  sgMatrixResult->ColCount = cseColCount2->Value + 1;
    // установить активность компонентов
  cseRowCount1->Enabled = false;
  cseColCount1->Enabled = false;
  cseColCount2->Enabled = false;
  sbHandle->Enabled = false;
  bbSetSize->Enabled = false;
  bbControlValues->Enabled = true;
}
```

Листинг 7.39. Текст обработчика события `OnClick` кнопки «Проконтролировать значения»

```
void __fastcall TfoTask2::bbControlValuesClick(TObject *Sender)
{ // проконтролировать значения матриц, заполненных вручную
  double aux;
    // матрица 1
  for (int irow = 1; irow < sgMatrix1->RowCount; irow++)
    for (int icol = 1; icol < sgMatrix1->ColCount; icol++)
      if (!TryStrToFloat
                (sgMatrix1->Cells [icol][irow], aux))
        {
            sgMatrix1->Row = irow;
          sgMatrix1->Col = icol;
          PageControl->ActivePage = tsMatrix1;
          MessageDlg
                ("Матрица 1 содержит некорректное данное!",
                      mtError, TMsgDlgButtons() << mbOK, 0);
         return;
        }
    // матрица 2
  for (int irow = 1; irow < sgMatrix2->RowCount; irow++)
    for (int icol = 1; icol < sgMatrix2->ColCount; icol++)
      if (!TryStrToFloat
                    (sgMatrix1->Cells [icol][irow], aux))
           {
           sgMatrix1->Row = irow;
             sgMatrix1->Col = icol;
           PageControl->ActivePage = tsMatrix2;
           MessageDlg
                 ("Матрица 1 содержит некорректное данное!",
                       mtError, TMsgDlgButtons() << mbOK, 0);
             return;
         }
    // все нормально; порождаем и заполняем матрицы
    // матрица 1
  if (FMatrix1 != NULL) delete FMatrix1;
  FMatrix1 =
      new TMatrixTask
        (sgMatrix1->RowCount - 1, sgMatrix1->ColCount - 1);
  FMatrix1->LoadFromGrid (sgMatrix1);
    // матрица 2
  if (FMatrix2 != NULL) delete FMatrix2;
  FMatrix2 =
      new TMatrixTask
```

```
        (sgMatrix2->RowCount - 1, sgMatrix2->ColCount - 1);
  FMatrix2->LoadFromGrid (sgMatrix2);

  bbControlValues->Enabled = false;
}
```

Листинг 7.40. Текст обработчика события `OnClick` кнопки «Умножить»

```
void __fastcall TfoTask2::bbMultiplyClick(TObject *Sender)
{
      // освободить прошлую память
  if (FMatrixRes != NULL) delete FMatrixRes;
      // отвести память под матрицу - результат
  FMatrixRes = new TMatrixTask
            (sgMatrixResult->RowCount - 1,
                                  sgMatrixResult->ColCount - 1);
            // умножение матриц
      (*FMatrixRes) = (*FMatrix1) * (*FMatrix2);
            // матрицу - в сетку sgMatrixResult
  sgMatrixResult->RowCount = FMatrixRes->RowCount + 1;
  sgMatrixResult->ColCount = FMatrixRes->ColCount + 1;
  FMatrixRes->SaveToGrid(sgMatrixResult);
}
```

Задача 7.3. Получить матрицу, псевдообратную по отношению к заданной прямоугольной матрице A и вычисленную по формуле: $A^+ = A \cdot (A^T \cdot A)^{-1}$. Для проверки правильности решения следует убедиться в том, что $A^T \cdot A^+ = E$, где E – единичная матрица.

Решение

Эта задача может служить примером для разработки алгоритмов, использующих матричные операции на базе созданного класса.

В списке методов класса нет метода, который напрямую позволил бы решить эту задачу. Поэтому метод решения нужно создать. Оформим решение задачи как метод созданного класса `TMatrixEx`. Назовем этот метод `PCode`:

```
void TMatrixEx::PCode(TMatrixEx &aMatr);
```

Тогда в качестве исходной матрицы можно будет взять матрицу объекта *this*, а результирующую матрицу получить через параметр метода `aMatr`. Следует отметить, что размеры псевдообратной матрицы будут обратными по отношению к размерам матрицы исходной, то есть $A^+(m \times n) \Rightarrow A(n \times m)$. Для простоты вычислений будем сохранять промежуточные результаты во вспомогательных матрицах. Тогда процедура получения псевдообратной матрицы может быть сведена к цепочке вычислений: $A \Rightarrow A^T \Rightarrow (A^T \cdot A) \Rightarrow (A^T \cdot A)^{-1} \Rightarrow A \cdot (A^T \cdot A)^{-1}$. Описание метода представлено в листинге 7.41.

Листинг 7.41. Текст метода `PCode`, предназначенного для вычисления псевдообратной матрицы

```
void TMatrixEx::PCode(TMatrixEx &aMatr)
// вычисляет псевдообратную матрицу
// aMatr - получит результат псевдообращения
// buf1Matr - вспомогательная переменная для хранения
// транспонированной матрицы
// buf2Matr - вспомогательная переменная для хранения
// результата промежуточного умножения исходной матрицы на
```

```
// обратную матрицу
{
   TMatrixEx buf1Matr (FColCount, FRowCount);
   TMatrixEx buf2Matr (FRowCount, FRowCount);
   Transpose (buf1Matr); // Транспонирование в buf1Matr (A^T)
   buf2Matr = buf1Matr*(*this);  // (A^T · A)
   double det = buf2Matr.DetMatr();  // Вычислить детерминант
   if(det == 0) return;// Если детерминант равен 0 - выход
   buf2Matr.InversMatr();   // Вычислить обратную (A^T · A)^-1
   aMatr = (*this)*buf2Matr; // Получить A·(A^T · A)^-1
 }
```

Задача 7.4. Требуется решить систему линейных уравнений, заданную в матричном виде $A \cdot x = B$.

Решение

Для решения данной задачи воспользуемся стандартным выводом решения в линейной алгебре, представленным на рис. 7.4. Здесь A – матрица коэффициентов; x – вектор неизвестных; B – вектор свободных членов.

Запись СЛУ в матричной форме	$A \cdot x = B$
Умножение уравнения слева на обратную матрицу	$A^{-1} \cdot A \cdot x = A^{-1} \cdot B$
Использование свойства $A^{-1} \cdot A = E$, где E – единичная матрица	$E \cdot x = A^{-1} \cdot B$
Использование свойства $E \cdot x = x$	$x = A^{-1} \cdot B$

Рис. 7.4. Схема вывода решения СЛУ

Исходными данными в задаче будут две матрицы: матрица коэффициентов и матрица из одного столбца, представляющая вектор свободных членов. Оформим решение как метод созданного класса `TMatrixEx`. Тогда в качестве матрицы коэффициентов можно взять матрицу объекта `this`, а матрицу, представляющую вектор свободных членов передать через параметр метода. И также через параметр метода получить результат, так как это тоже будет матрица с одним столбцом. Решение системы уравнений сводится к следующим шагам:

- *вычислить* определитель матрицы коэффициентов;
- если *определитель* равен 0, то необходимо выйти – решения нет;
- если *определитель* не равен нулю, то необходимо проделать следующее:
  - вычислить обратную матрицу;
  - умножить ее на матрицу свободных членов.

Описание метода `SLEqv` приведено в листинге 7.42.

Листинг 7.42. Текст метода `SLEqv`, предназначенного для решения СЛУ

```
void TMatrixEx::SLEqv(TMatrixEx &bMatr, TMatrixEx &xMatr)
 // решение СЛУ
{
      // Вспомогательная переменная  для хранения
      // исходной и обратной матрицы
```

```
    TMatrixEx buf2Matr (*this);

    double det = DetMatr();  // Вычислить детерминант
             // Если  детерминант равен 0, то выход
    if(det == 0) return;
    buf2Matr.InversMatr();   // Вычислить обратную
    xMatr = buf2Matr*bMatr;   // Получить  решение
}
```

Рекомендация

Для получения формы к решению задачи обращения матрицы можно воспользоваться копией формы «Умножение матриц» и исправить соответствующие обработчики.



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

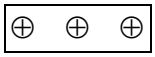
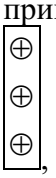
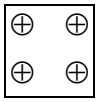
1. Элемент матрицы называется седловой точкой, если он является одновременно наименьшим в своей строке и наибольшим в своем столбце. Выяснить, имеются ли седловые точки в данной матрице, а если имеются, то указать их координаты.
2. На квадратном листе клетчатой бумаги заданного размера нарисовано несколько прямоугольников. Каждый прямоугольник состоит из клеток. Различные прямоугольники не накладываются друг на друга и не соприкасаются. Дана целочисленная матрица (по числу клеток), в которой элемент равен 0, если соответствующая клетка принадлежит прямоугольнику, и отличен от нуля в противном случае. Определить число прямоугольников.
3. Дана целочисленная матрица, каждый элемент которой равен 0, 1, 2 или 3. Определить количество четверок $a_{i,j}, a_{i+1,j}, a_{i,j+1}, a_{i+1,j+1}$, в каждой из которых все элементы различны.
4. Дана матрица. Исключить совпадающие строки среди строк с большими номерами.
5. Сложить, умножить треугольные или симметричные матрицы.
6. Задача «Жизнь»: Жизнь – это многоклеточное сообщество, населяющее пустыни Флатландии. Пустыня представляет собой квадратную решетку, каждая ячейка которой вмещает одну клетку Жизни. Мерой течения времени служит смена поколений Жизни, приносящая в колонию смерть и рождение. Требуется написать программу, моделирующую колонию Жизни [23].

Рекомендации:

Чтобы проследить за историей развития колонии, разместим в пустыне клетки жизни в их начальном положении. Смена поколений будет происходить по следующим правилам:

а) соседями клетки будут считаться все элементы, расположенные в восьми ячейках рядом с данными по горизонтали, вертикали или диагонали;
б) если у некоторой клетки меньше двух соседей, то она погибает от одиночества;
в) если у клетки больше трех соседей, то она погибает от тесноты;

г) если рядом с пустой клеткой окажется ровно три соседние клетки Жизни, то в этой клетке рождается новая клетка;

д) гибель и рождение происходит в момент смены поколений. Таким образом, гибнущая клетка может способствовать рождению новой клетки, но рождающаяся клетка не может воскресить гибнущую, и гибель одной клетки, уменьшив локальную плотность населения, не может предотвратить гибель другой. Например, колония  в следующем поколении превращается в колонию  , а колония  никогда не меняется.

7. Исключить из матрицы нулевые и дублирующие столбцы (строки).
8. Проверить, является ли матрица единичной.
9. Выделить базисные столбцы.
10. Получить сопряженную матрицу.
11. Найти след матрицы.
12. Найти собственные значения матрицы.

Использование элементов комбинаторной математики в решении задач 8

Среди объектов, изучаемых в дискретной математике, да и не только, одними из важнейших являются перестановки конечных множеств.

ГЕНЕРАТОР ПЕРЕСТАНОВОК

Генератор перестановок – программа, которая генерирует все возможные перестановки элементов некоторого множества.

Общее количество перестановок для множества мощностью *n* вычитается по формуле: *n*!.

Рассматривая задачу порождения перестановок множества элементов $(a_1, a_2 \ldots a_n)$ без ограничения общности, будем полагать, что для любого *i* выполняется $a_i = i$, то есть элементами множества являются числа от *1* до *n*. Тогда для произвольного набора данных $R_1, R_2 \ldots R_n$ новая перестановка будет сводиться к вычислению индексов: $R_{a_1}, R_{a_2} \ldots R_{a_n}$.

При генерации перестановок обычно придерживаются следующей стратегии, которую можно свести к выполнению трех шагов:

а) построение начальной перестановки (эта перестановка становится текущей перестановкой);
б) преобразование текущей перестановки в следующую перестановку;
в) проверка условия окончания.

Причем первый шаг можно считать подготовительным.

С точки зрения объема памяти, необходимого для процесса порождения перестановок, порождать всю последовательность сразу и хранить ее в памяти машины *не является целесообразным*. В таком смысле более предпочтительным является подход, который основывается на итеративном порождении очередной перестановки.

Задача 8.1. Создать перестановку, следующую за данной перестановкой в лексикографическом порядке.

Решение

Решение задачи поиска перестановки, следующей за данной перестановкой, в *лексикографическом* порядке будем оформлять в виде независимой процедуры, которую назовем `NextPermutation`.

Процедура `NextPermutation` будет иметь два параметра: `aX` – массив, в котором задается начальная перестановка, туда же после работы процедуры помещается найденная новая перестановка; `aN` – число элементов в массиве `aX`. В результате работы процедуры будет возвращаться `True` (если процедура успешно завершена и в массив `aX` будет помещена следующая перестановка) или `False` (если `aX` является последней перестановкой или процедуре переданы неверные параметры, например элементы массива не являются числами от 1 до *N*).

Примечание

Говорят, что перестановки $f =< a_1, a_2 \ldots a_n)$ и $g =< b_1, b_2 \ldots b_n)$ записаны в лексикографическом порядке ($f < g$), если существует такое $k \geq 1$, что $a_k \leq b_k$ и (при $k > 1$) выполняются условия $a_g = b_g$ для всех $g < k$.

На рис. 8.1 представлено множество перестановок из 4-х элементов, записанных в лексикографическом порядке. Как видно из рисунка, в самой первой перестановке элементы располагаются в возрастающей последовательности, а в последней перестановке – в убывающей. Всего перестановок – $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$.

1. <1, 2, 3, 4>	7. <2, 1, 3, 4>	13. <3, 1, 2, 4>	19. <4, 1, 2, 3>
2. <1, 2, 4, 3>	8. <2, 1, 4, 3>	14. <3, 1, 4, 2>	20. <4, 1, 3, 2>
3. <1, 3, 2, 4>	9. <2, 3, 1, 4>	15. <3, 2, 1, 4>	21. <4, 2, 1, 3>
4. <1, 3, 4, 2>	10. <2, 3, 2, 4>	16. <3, 2, 4, 1>	22. <4, 2, 3, 1>
5. <1, 4, 2, 3>	11. <2, 4, 1, 3>	17. <3, 4, 1, 2>	23. <4, 3, 1, 2>
6. <1, 4, 3, 2>	12. <2, 4, 3, 1>	18. <3, 4, 2, 1>	24. <4, 3, 2, 1>

Рис. 8.1. Последовательность перестановок из 4-х элементов, полученных в лексикографическом порядке

Работу предлагаемого к рассмотрению алгоритма продемонстрируем на следующем примере.

Пример. Пусть требуется записать перестановку, которая расположена за перестановкой 15-го порядка вида:

<15, 2, 4, 3, 1, 13, 7, 10, 14, 12, 11, 9, 8, 6, 5>.

Шаг 1. Определим, так называемый, *хвост* перестановки. Под *хвостом* будем понимать максимально возможную убывающую последовательность значений элементов перестановки, расположенную на позициях в конце этой перестановки:

<15, 2, 4, 3, 1, 13, 7, 10, 14, 12, 11, 9, 8, 6, 5>.

Очевидно, это будет подпоследовательность 14, 12, 11, 9, 8, 6, 5.

Шаг 2. Фиксируем элемент перед хвостом. Непосредственно перед началом хвоста находится элемент со значением 10.

<15, 2, 4, 3, 1, 13, 7, 10, 14, 12, 11, 9, 8, 6, 5>.

Шаг 3. Находим в хвосте первый от конца элемент, который больше, чем число 10. Это число 11.

<15, 2, 4, 3, 1, 13, 7, 10, 14, 12, 11, 9, 8, 6, 5>.

Шаг 4. Меняем местами 10 и 11. Получим

<15, 2, 4, 3, 1, 13, 7, 11, 14, 12, 10, 9, 8, 6, 5>.

Шаг 5. Переписываем хвост в обратном порядке:

<15, 2, 4, 3, 1, 13, 7, 11, 5, 6, 8, 9, 10, 12, 14>.

В алгоритм вводится вспомогательная булева переменная `result` для того, чтобы сообщить об успешности/неуспешности завершения процедуры. Тогда алго-

ритм генератора очередной перестановки в лексикографическом порядке (при заданных параметрах) можно представить в виде следующих шагов:

1. Проверяются, есть ли элементы в исходном массиве `aX`. Если `aN` ≤ 0, то необходимо выполнить:
   а) `result = False`;
   б) выход.
2. Проверяется утверждение, что в исходном массиве `aX` представлены все значения (и только) натурального ряда от 1 до `aN`. Для этого выполняется цикл (по числу элементов в массиве `aX`). Для каждого элемента выполняется проверка и если элемент меньше 1 или больше `aN`, то необходимо выполнить:
   а) `result = False`;
   б) выход.
3. Выделяется *хвост* текущей перестановки. Для этого в цикле, начиная от конца массива `aX`, производится сравнение соседних элементов до тех пор, пока не будет нарушен порядок, определяющий убывание элементов. Начало *хвоста* фиксируется.
4. Если *хвост* не совпадает со всей перестановкой, то в нем находится первый от конца элемент, который по величине больше, чем элемент перестановки, расположенный непосредственно перед ее хвостом.
5. Если перестановка совпадает со своим хвостом, то она является заключительной во всей итерации:
   а) `result = False`;
   б) выход.
6. Хвост преобразуется, а именно: меняется местами элемент, найденный в пункте 4, с элементом, расположенным непосредственно перед хвостом.
7. Производится инвертирование преобразованного хвоста (переставляются элементы в обратном порядке).
8. Выполняются завершающие действия: `result = true`; выход.

Описание генератора перестановок представлено в листинге 8.1.

Листинг 8.1. Создание перестановки, следующей за данной перестановкой в лексикографическом порядке

```
bool NextPermutation (int *aX, const int &aN)
{
    bool result;
    int k;
    int t;
    int y;
    //  Проверка
    if (aN <= 0)
     {
        result = false;
        return result;
     }
    for(k = 1; k <= aN; k++)
     {
        if (aX [k] <1 || aX [k] > aN)
        {
            result = false;
            return result;
        }
```

```
     }
    // Process
                // поиск «хвоста»
    k = aN - 1;
    while(k>0) // пока не просмотрены все элементы
     {
        if (aX [k] <= aX [k+1]) // если хвост найден
         {
            break;      // выход из цикла
         }
        k = k-1; // изменение индекса просмотра
     }
// если хвост не совпадает со всей исходной перестановкой
    if( k!=0 )
     {
            // ищется элемент для перестановки с элементом,
            // расположенным непосредственно перед хвостом
        t = k+1;
        while (t < aN)
         {
            if( aX [t+1] <= aX [k]) // если элемент найден
             {
                break; // выход из цикла
             }
            t = t+1; // изменение индекса просмотра
         }
        y = aX [k]; // перестановка
        aX [k] = aX [t];
        aX [t] = y;
                  // инверсия измененного хвоста
        t = 0;
        while (t < (aN-k)/2)
         {
            y = aX [aN-t];
            aX [aN-t] = aX [k+1+t];
            aX [k+1+t] = y;
            t = t+1;
         }
        result = true;
     }
    else // если хвост совпадает с перестановкой
     {
        result = false;
     }
    return result;
}
```

Задача 8.2. Создать итеративный генератор перестановок некоторого множества из *n*-элементов в антилексикографическом порядке.

Решение

Алгоритм для генерации перестановок в антилексикографическом порядке может базироваться на двух утверждениях.

1. Если множество перестановок *P* упорядочено антилексикографически, то начальная и конечная перестановки – это, соответственно, (1, 2 … *n*) и (*n*, (*n* – 1) … 1).

2. Упорядоченная антилексикографически последовательность перестановок по значению последнего элемента в них может быть разбита на n-блоков длины $(n-1)!$. При этом q блок на последнем месте имеет элемент, равный $(n-q+1)$, а первые $n-1$ позиций этого блока определяют последовательность перестановок множества $\{1, 2 \ldots n\}\backslash\{q\}$ в антилексикографическом порядке.

На рис. 8.2 представлено множество перестановок в антилексикографическом порядке.

3!	**4!**			
1 2 **3**	1 2 3 **4**	1 2 4 **3**	1 3 4 **2**	2 3 4 **1**
2 1 **3**	2 1 3 **4**	2 1 4 **3**	3 1 4 **2**	3 2 4 **1**
1 3 **2**	1 3 2 **4**	1 4 2 **3**	1 4 3 **2**	2 4 3 **1**
3 1 **2**	3 1 2 **4**	4 1 2 **3**	4 1 3 **2**	4 2 3 **1**
2 3 **1**	2 3 1 **4**	2 4 1 **3**	3 4 1 **2**	3 4 2 **1**
3 2 **1**	3 2 1 **4**	4 2 1 **3**	4 3 1 **2**	4 3 2 **1**

Рис. 8.2. Последовательность перестановок из 3-х и 4-х элементов, полученных в антилексикографическом порядке

Для того чтобы реализовать описанный алгоритм, предусмотрены следующие подпрограммы:

- `void swap (int *a, int *b);` – для перестановки элементов, расположенных по адресам a и b;
- `void reverse (int * P, int m);` – для записи элементов массива `P` в обратном порядке; здесь `m` – число элементов в массиве;
- `void antilex (int * P, int m, TListBox* Box);` – для генерации всех перестановок в антилексикографическом порядке и записи их в список `Box`;
- `AnsiString Print (int* P);` – для записи элементов массива `P` в строку типа `AnsiString`;
- `void DoMain (TListBox* Box);` – главная процедура, предназначенная для вызова процедуры `antilex`.

Перестановка двух элементов

Процедура `swap` имеет два параметра `a` и `b`, которые являются адресами переставляемых параметров. Алгоритм перестановки сводится к трем операциям:

- первое число из `a` запоминается во вспомогательной переменной `t`;
- по адресу `a` заносится число, расположенное по адресу `b`;
- в `b` заносится число из `a`, предварительно сохраненное в `t`.

Код программы приведен в листинге 8.2.

Листинг 8.2. Перестановка двух элементов, расположенных по адресам `a`, `b`

```
void swap(int *a, int *b)
{ int t;

  t=*a;
  *a=*b;
  *b=t;
}
```

Запись элементов массива в обратном порядке

Процедура `reverse` имеет два параметра `P` и `m`. Массив `P` является и входным, и выходным. А переменная `m` задает *номер последнего элемента* в массиве. Алгоритм «переворачивания» массива описывается следующими шагами:

- во вспомогательных переменных `i` и `j` устанавливаются начальное и конечное значения индексов;
- выполняется цикл (пока начальные и конечные значения индексов не встретятся):
  - меняются элементы со значениями индексов `i` и `j`;
  - индекс начального элемента `i` увеличивается на 1;
  - индекс конечного значения `j` уменьшается на 1.

Описание процедуры приведено в листинге 8.3.

Листинг 8.3. Запись элементов в массиве P в обратном порядке

```
void reverse(int * P, int m)
{
   int i=0, j=m;
   while(i<j)
   {
     swap(&P[i], &P[j]);
     ++i;
     --j;
   }
}
```

Создание перестановок из заданного массива, следующих в антилексикографическом порядке

Процедура `antilex`, предназначенная для создания перестановок из заданного массива, следующих в антилексикографическом порядке, является рекурсивной. Она имеет три параметра: массив `P` является и входным, и выходным (в нем хранится текущая перестановка); `m` задает размер массива; список `Box` определяет поле, в которое последовательно *помещаются все созданные перестановки*. Алгоритм создания перестановок в антилексикографическом порядке можно описать следующими действиями.

1. Если получена очередная перестановка, то необходимо выполнить следующие шаги:
   а) этот массив – перестановка заносится во вспомогательную строку;
   б) строка помещается в список `Box`.
2. Иначе выполняется цикл по числу блоков, в котором строится очередная перестановка по схеме:
   а) меняются в очередном блоке первый и последний элементы;
   б) переворачивается начало блока.

Описание процедуры `antilex` приведено в листинге 8.4.

Листинг 8.4. Создание перестановок в антилексикографическом порядке

```
void antilex (int *P, int m, TListBox *Box)
{
  int i;
  if(m == 0)
```

```
  {
            // перестановка упаковывается в строку
     AnsiString Ptr =  Print(P);
            // добавляется к списку перестановок
     Box->Items->Add(Ptr);
  }
  else
  {
    for(i=0; i<=m; ++i)
    {
      antilex (P,m-1, Box);
      if(i<m)
      {
        swap (&P[i], &P[m]);
        reverse (P,m-1);
      }
    }
  }
}
```

Запись элементов целочисленного массива в строку `AnsiString`

Процедура принимает на входе через список параметров массив `P` определенной размерности, а через заголовок возвращает результат. Для того чтобы не усложнять восприятие алгоритма в целом, принято решение количество элементов задавать внутри программы. В примере установлено число 4. При изменении размера массива это значение нужно подправлять. Алгоритм записи элементов целочисленного массива в строку можно описать следующими шагами:

- инициализируется вспомогательная переменная `Ptr` для приема чисел;
- выполняется цикл по числу элементов, в котором к вспомогательной переменной `Ptr` добавляется число и символ-пробел для отделения одного числа от другого.

Описание процедуры `Print` приведено на листинге 8.5.

Листинг 8.5. Запись элементов массива `P` в строку

```
AnsiString Print (int *P)
{
   AnsiString Ptr ="";
   for (int i=0; i<4; ++i)
      Ptr = Ptr + "  "+ IntToStr(P[i]);
   return Ptr;
}
```

Создание всех перестановок из заданного числа элементов в антилексикографическом порядке

Процедура имеет только один входной параметр – для задания списка, в поле которого будут помещены сгенерированные перестановки. Алгоритм создания перестановок можно свести к двум шагам:

- создание начального образа массива, элементы которого будут переставляться в соответствии с логикой алгоритма;
- вызов процедуры `antilex` – генерации перестановок.

В процедуре начальные значения массива представлены порядковыми номерами. Это не обязательно. Элементы массива могут быть представлены разным набором чисел, в том числе *неупорядоченным* и с *совпадающими* элементами.

На рис. 8.3 представлены перестановки с разными представлениями начальных значений исходного массива (первая строка в каждом столбце таблицы). Код программы приведен в листинге 8.6.

Листинг 8.6. Создание перестановок в антилексикографическом порядке

```
void DoMain (TListBox* Box)
{
  int i;
  int N = 4;
  int*P = new int[N];

  for(i=0; i<N; ++i)
    P[i] = i+1;

  antilex(P, N-1, Box);
}
```

1 2 3 4	4 1 3 2	43 12 43 1
2 1 3 4	1 4 3 2	12 43 43 1
1 3 2 4	4 3 1 2	43 43 12 1
3 1 2 4	3 4 1 2	43 43 12 1
2 3 1 4	1 3 4 2	12 43 43 1
3 2 1 4	3 1 4 2	43 12 43 1
1 2 4 3	4 1 2 3	43 12 1 43
2 1 4 3	1 4 2 3	12 43 1 43
1 4 2 3	4 2 1 3	43 1 12 43
4 1 2 3	2 4 1 3	1 43 12 43
2 4 1 3	1 2 4 3	12 1 43 43
4 2 1 3	2 1 4 3	1 12 43 43
1 3 4 2	4 3 2 1	43 43 1 12
3 1 4 2	3 4 2 1	43 43 1 12
1 4 3 2	4 2 3 1	43 1 43 12
4 1 3 2	2 4 3 1	1 43 43 12
3 4 1 2	3 2 4 1	43 1 43 12
4 3 1 2	2 3 4 1	1 43 43 12
2 3 4 1	1 3 2 4	12 43 1 43
3 2 4 1	3 1 2 4	43 12 1 43
2 4 3 1	1 2 3 4	12 1 43 43
4 2 3 1	2 1 3 4	1 12 43 43
3 4 2 1	3 2 1 4	43 1 12 43
4 3 2 1	2 3 1 4	1 43 12 43

Рис. 8.3. Последовательности перестановок из 4-х элементов, полученных в антилексикографическом порядке

Задача 8.3. Требуется подсчитать количество сочетаний из n по m элементов.

Решение

Алгоритм использует стандартную формулу: $C_n^m = \frac{n!}{m! \cdot (n-m)!}$. В этой формуле число n определяет общее количество элементов (исходя из этого количества, производится выбор), а число m – количество элементов, объединяемых в сочетании. В соответствии с этим изначально должно выполняться условие: $n \geq m$.

Логику алгоритма поясним на примере следующих вычислений:

$$C_7^5 = \frac{7!}{5!\cdot(7-5)!} = \frac{7!}{5!\cdot 2!} = \frac{(1\cdot 2)\cdot 3\cdot 4\cdot 5\cdot 6\cdot 7}{1\cdot 2\cdot 3\cdot 4\cdot 5\cdot (1\cdot 2)} = \frac{3\cdot 4\cdot 5\cdot 6\cdot 7}{1\cdot 2\cdot 3\cdot 4\cdot 5}.$$

Как видно из примера, вычисления сводятся к подсчету двух произведений, каждое из которых содержит $m = 5$ сомножителей. Заметим, что числитель начинается от сомножителя $3 = (n - m + 1)$ и заканчивается значением n. Вычисления оформлены в виде процедуры с именем `CombiNM`, которая имеет два входных параметра `aN` и `aM`, как это требуется для формулы.

Алгоритм процедуры `CombiNM` для подсчета количества сочетаний из `aN` элементов по `aM` элементов можно описать следующими шагами:

1) определяется `p` – число сомножителей;
2) программируется формула для получения числа перестановок (количество перестановок записывается в `r`):
   а) инициализируется `r` – начальное значение сомножителя в числителе (в знаменателе сомножитель определяется значением `i`);
   б) выполняется цикл (по числу сомножителей), в котором формируется произведение.
3) Возвращается `r` как результат функции.

Описание процедуры `CombiNM` представлено в листинге 8.7.

Листинг 8.7. Подсчет количества сочетаний из `aN` по `aM` элементов

```
long int __fastcall CombiNM (int aN, int aM)
{
 long int p, r, i;
      // переопределяем меньшее
 p = (aM <= aN - aM) ? aM : aN - aM;
 aM = aN - p;
 r = (p == 0) ? 1 : aM + 1;
 for (i = 2; i <= p; i++)
   r = (aM + i) * r / i;
 return r;
}
```

Задача 8.4. Требуется получить все сочетания из n по m элементов.

Решение

Идею алгоритма можно пояснить на примере получения всех возможных сочетаний из 5-ти (n) элементов по 3 (m) элемента в порядке, как показано на рис. 8.4.

Исходных элементов всего пять: 1, 2, 3, 4, 5. Согласно рис. 8.3 первое сочетание соответствует первым по порядку трем числам, взятым из исходного множества: 1, 2, 3. Следующие два сочетания отличаются последним элементом (4, 5). Отметим, что число 5 является предельным элементом в исходном множестве. После этого изменяется предпоследнее число на 1, и снова меняется последний элемент до своего конечного значения (числа 5), и так далее до тех пор, пока не будет получено последнее сочетание, которое соответствует последним по порядку трем числам, взятым из исходного множества: 3, 4, 5.

Оформим алгоритм вычисления сочетаний в виде отдельной процедуры. Процедура (назовем ее `SimpleCombination`) должна иметь три входных параметра: общее количество сочетаний (`aLen`); количество базовых элементов, из которых выбираются сочетания (`aN`); количество элементов, объединяемых в сочетании (`aM`).

И один выходной параметр – двумерный массив, содержащий все сочетания (aCombi). Например, для задачи, следующей из рисунка 8.4, массив aCombi должен быть определен размером 10×3.

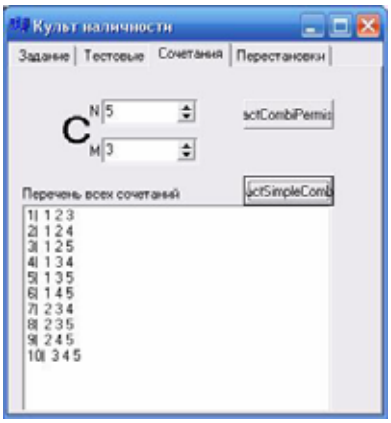


Рис. 8.4. Всевозможные сочетания из 5-ти элементов по 3 элемента

Алгоритм получения всех возможных сочетаний из заданного множества чисел можно описать следующими шагами.

1. генерируется первое сочетание (это сочетание будет первой строкой в выходном массиве);
2. выполняется цикл по i для всех сочетаний (начиная от номера 1, так как первое уже сформировано):
   а) копируется предыдущее сочетание в текущее/формируемое (оно будет исправляться на новое значение);
   б) фокусируется последнее число (aCombi [i][j]) в текущем сочетании (установкой индекса j);
   в) увеличивается это последнее число на 1;
   г) инекс j устанавливается на элемент, следующий за переполненными элементами;
   д) выстраиваются конечные элементы в текущем сочетании – это новое сочетание.

Описание процедуры SimpleCombination представлено в листинге 8.8.

Листинг 8.8. Получение сочетаний из aN элементов по aM элементов

```
void __fastcall SimpleCombination (int **aCombi, int aLen,
                                                    int aN, int aM)
{
  for (int j = 0; j < aM; j++)  // первое сочетание
    aCombi [0][j] = j + 1;

  for (int i = 1; i < aLen; i++)
   {
      // копируем предыдущее сочетание
      for (int j = 0; j < aM; j++)
      aCombi [i][j] = aCombi [i - 1][j];

    int _shi = 0;
    int j = aM - 1;
do_next:   // проматываем переполненные элементы
    aCombi [i][j]++;
```

```
    if (j > 0 && (aCombi [i][j] + _shi) % aN == 1)
      {
        j--;
       _shi++;
        goto do_next;
      }
    for (j++; j < aM; j++)  // выстраиваем конечные элементы
      aCombi [i][j] = aCombi [i][j - 1] + 1;
  }  // for i
};
```

Примечание

Если решать задачу в соответствии с интерфейсом, представленным на рис. 8.4, то действия пользователя можно описать шагами:

- установить значения *N* и *M*;
- нажать кнопку «Получить сочетания»;
- в результате в окне, названном «Перечень всех сочетаний», должны отобразиться все сочетания в описанном порядке со своим порядковым номером (который отделяется от сочетания вертикальной линией).

Целесообразно ввести глобальную переменную, в которой поместить общее количество сочетаний, а также глобальный массив, в котором поместить все сочетания:

```
int **FCombi; // указатель на массив сочетаний
long int FNComb; // количество сочетаний
```

И тогда алгоритм обработчика события `OnClick` для кнопки «Получить сочетания», в свою очередь, можно описать последовательностью действий:

- взять значения `N` и `M` из отведенных полей (компонентов `SpinEdit` с именами `cseN` и `cseM`);
- подсчитать `FNComb` – общее количество сочетаний;
- отвести необходимую память под массивы (`FCombi`), предназначенные для хранения сочетаний;
- получить множество сочетаний (вызвать процедуру `SimpleCombination`);
- отобразить полученное множество в окне компонента-списка (с именем `lbCombi`) интерфейса.

Код обработчика представлен в листинге 8.9.

Листинг 8.9. Обработчик события `OnClick` для кнопки «Получить сочетания»

```
void __fastcall TfoMain::actSimpleCombiExecute
                                          (TObject *Sender)
{
  int nN, mM;
  int iComb, ind;  // индексы

  nN = cseN->Value;
  mM = cseM->Value;

  // подсчет количества сочетаний из N по M
  FNComb = CombiNM (cseN->Value, cseM->Value);

  // может высвобождение памяти
```

```
  if (FCombi) delete []FCombi;
  // заведение памяти
  FCombi = new int * [FNComb];
  for (iComb = 0; iComb < FNComb; iComb++)
    FCombi [iComb] = new int [mM];

  // заполнение сочетаниями
  SimpleCombination (FCombi, FNComb, nN, mM);

  lbCombi->Clear();
  for (iComb = 0; iComb < FNComb; iComb++)
   {
    AnsiString aux = IntToStr (iComb + 1) + "|  ";
    for (int j = 0; j < cseM->Value; j++)
      aux = aux + IntToStr (FCombi [iComb][j]) + " ";
    lbCombi->Items->Add (aux);
   }
  actCombiPermit->Enabled = true;
}
```

Задача 8.5. Задан набор – множество букв. Требуется создать список всех возможных слов с заданными длинами. Причем для проверки правильности слов использовать `SpellChecker`, например, из *Word for Windows*.

Решение

Предлагается следующая стратегия решения задачи.

1. Определяется, какую информацию следует принять от пользователя.
2. Определяется, какую информацию следует предоставить пользователю.
3. Определяется последовательность действий, которые должен выполнить пользователь для того, чтобы решить задачу.
4. Разрабатывается интерфейс приложения.
5. Подготавливаются специальные процедуры для обеспечения действий пользователя.

Подготовка решения в соответствии с описанной стратегией.

1. Пользователь должен задать следующие параметры:
   а) количество букв, из которых складываются слова;
   б) максимальную и минимальную длины слов;
   в) набор букв для составления слов.
2. Пользователю требуется предоставить перечень возможных слов из заданного набора подходящей длины.
3. Предлагается обеспечить следующую последовательность действий с приложением: задать параметры; нажать кнопку «Найти слова»; получить окно с результатом.
4. Для решения задачи может быть предложен интерфейс программы, как на рис. 8.5.

Согласно рис. 8.5, для решения задачи, расположенной на странице блокнота «Задание» необходимы следующие компоненты:

1. `Label` – для поясняющих текстов.
2. Три компонета `CSpinEdit`:
   а) `cseLetterCount` – для задания числа букв;

б) cseMinCount – для задания минимального числа символов в слове;
в) cseMaxCount – для задания максимального числа символов в слове.

3. Компонент типа «список» TListBox (lbPossibleWords) – для хранения отобранных слов.
4. Компонент типа TStringGrid – сетка для задания самих букв;
5. Четыре компонента StaticText:
   а) StaticText – для отображения информации о завершении решения задачи;
   б) stLetterCount – число букв в слове;
   в) stCompose – число сочетаний;
   г) stWordCount – число подготовленных слов.
6. Кнопка «Решить задачу» с именем bbPosibleWord.

Кроме этого, предлагается подготовить страницу «Тестирование», как показано на рис. 8.6. На этой странице представлены окна – списки для отображения в них:

а) множества сочетаний (lbTestCombi);
б) множества перестановок всех сочетаний (lbTestCombu);
в) множества слов (lbTestLetter).

Эта страница позволит оценить правильность работы алгоритма.

Для того чтобы предоставить пользователю возможность выполнения действий в соответствии с описанным порядком и особенностей решения задачи получения слов целесообразно создать следующие подпрограммы:

- `long int __fastcall CombiNM (int aN, int aM);` – функция для вычисления количества сочетаний из aN по aM;
- `void __fastcall SimpleCombination (int **aCombi, int aLen, int aN, int aM);` – функция для получения массива сочетаний (параметр aCombi) из aN по aM (длиной aLen);
- `bool NextPermutation (int *aX, const int &aN);` – функция для вычисления перестановки, следующей за данной перестановкой, хранящейся в массиве aX, в лексикографическом порядке; здесь aN – длина массива; новая перестановка также записывается в массив aX;
- `void __fastcall TfoMain::FormCreate(TObject *Sender);` – обработчик события OnCreate главной формы решения; он содержит действия, связанные с подключением *Word for Windows*;
- `void __fastcall TfoMain::FormClose(TObject *Sender, TCloseAction &Action);` – обработчик события OnClose главной формы решения; он предназначен для высвобождения памяти, занятой глобальными массивами, и завершения сеанса с *Word for Windows*;
- `void __fastcall TfoMain::bbFindWordsClick (TObject *Sender);` – обработчик события onClick кнопки «Найти слова»;
- `cseLetterCountChange (TObject *Sender);` – обработчик события OnChange компонента cseLetterCount, отображающего число букв.

Первые три процедуры подробно описаны в задачах:

- в задаче 8.1 – функция NextPermutation;
- в задаче 8.3 – функция CombiNM;
- в задаче 8.4 – функция SimpleCombination.

Рекомендация

В связи с использованием перечисленных процедур целесообразно воспользоваться глобальными переменными `FCombi`, `FNComb`.

В соответствии с условием задачи из всех возможных слов, которые можно собрать, пользуясь сочетаниями заданного множества букв, следует выбрать те, которые могут трактоваться как *правильные* слова в русском языке. С этой целью в соответствии с условием задачи рекомендуется воспользоваться *Word for Windows* и встроенным в *Word for Windows* редактором `SpellChecker`.

Для связи с *Word* в программе используется OLE-технология, которая подробно описана в главе 18. При использовании OLE-технологии потребуется глобальная переменная, которую назовем `FWord`:

```
Variant  FWord;  // переменная для связи с Word по OLE
```

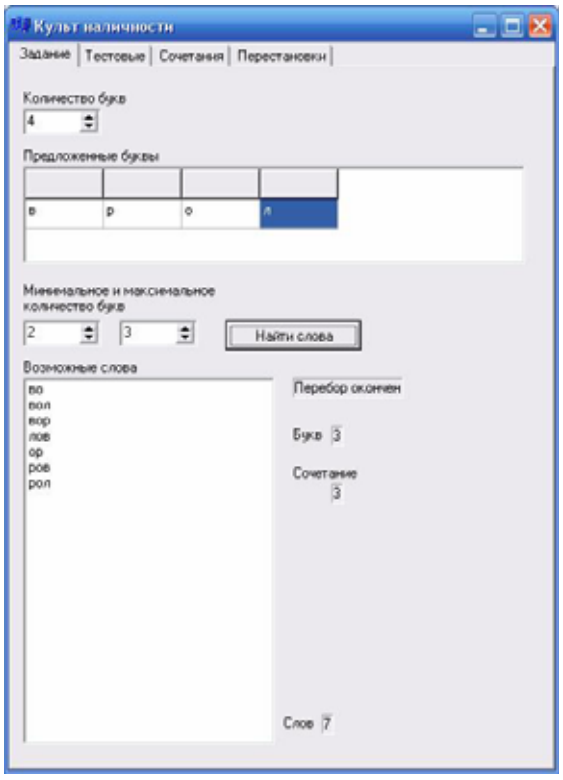


Рис. 8.5. Интерфейс программы, предоставляющей множество слов ограниченной длины

Подключение *Word for Windows*

В задачу обработчика события `OnCreate` формы входит установка связи с *Word*. Для этого предусмотрены действия:

- инициализируется глобальная переменная `FWord`, предназначенная для связи с *Word*;
- поскольку вся работа с *Word* должна проходить как бы «за кадром», приложение *Word* делается невидимым (устанавливается соответствующее OLE-свойство *Word*);
- если приложение *Word* не запущено, то выдается сообщение.

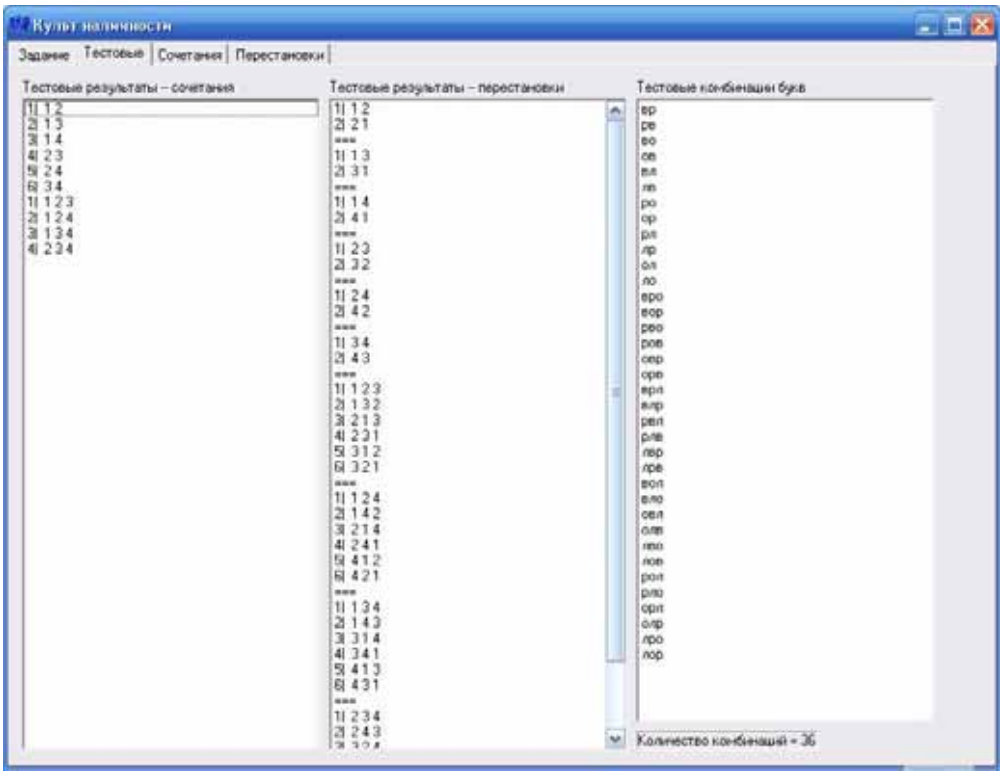


Рис. 8.6. Интерфейс страницы «Тестовые»

Код обработчика события `OnCreate` («на создание формы») представлен в листинге 8.10.

Листинг 8.10. Обработчик события «на создание формы»

```
void __fastcall TfoMain::FormCreate(TObject *Sender)
{
  FCombi = NULL;
    // открыть сессию Word for Windows
  FWord = Variant::CreateObject ("Word.Application");
  FWord.OlePropertySet ("Visible", false);
  if (FWord.IsEmpty())  // если Word все же не стартовали
    ShowMessage ("Приложение Word не удалось стартовать!
                              Программа НЕ готова к работе!");
}
```

Поиск слов, составленных из заданного множества букв

Основная идея, положенная в основу алгоритма, предназначенного для поиска слов (обработчик события `OnClick` под кнопку «Найти слова») сводится к следующему:

- генерируются все возможные сочетания из заданного множества символов соответствующей длины (от минимальной длины до максимальной длины по порядку);
- для каждого сочетания генерируются все возможные перестановки (так как значение *слова изменяется от перестановки* букв);
- каждой перестановке из очередного сочетания ставится в соответствие определенное этим сочетанием слово;

- это слово проверяется на правильность и, если оно правильное, добавляется к массиву правильных слов.

Примечание

Для того чтобы от перестановки перейти к слову, достаточно из сетки `StringGrid` взять содержимое ячеек, расположенных в первой строчке, *номер колонки которых совпадает* со значениями элементов перестановки.

Подробности алгоритма обработчика события `OnClick` под кнопку «Найти слова» можно описать следующими шагами.

1. Проверяется связь с *Word*. Если *Word* не запустилось – выход.
2. Устанавливается общее количество (`nN`) букв в слове.
3. Выполняется цикл по переменной (`mM`), принимающей значения количества сочетаемых букв (от минимально возможного до максимально возможного по порядку):
   а) подсчитывается количество сочетаний из `nN` по `mM`;
   б) фиксируется количество букв в слове в `stLetterCount`;
   в) отводится память под массив `ptrCombi` – всего множества сочетаний из `nN` по `mM` элементов;
   г) отводится память под вспомогательный массив `arrPerm` для хранения отдельной перестановки;
   д) находятся все сочетания из `nN` по `mM` элементов, которые записываются в массив `ptrCombi`;
   е) выполняется цикл по перебору всех комбинаций-сочетаний из массива `ptrCombi`:
      i) текущее сочетание помещается во вспомогательную строку `auxC` вместе с порядковым номером этого сочетания и с пробелом-разделителем;
      ii) сформированная строка добавляется к списку сочетаний `lbTestCombi`;
      iii) для текущего сочетания выполняется цикл для создания всех возможных перестановок из этого сочетания:
         (1) «взять» перестановку во вспомогательную строку `auxP` в виде номеров букв;
         (2) добавить `auxP` в список перестановок `lbTestCombi`;
         (3) сформировать на основе перестановки слово и занести его в строку `auxW`; при этом текущая перестановка помещается во вспомогательную строку `auxW` в виде символов с соответствующими порядковыми номерами;
         (4) добавить `auxW` в список слов `lbTestLetter`;
         (5) установить правильность слова с помощью OLE-функции `CheckSpelling`;
         (6) если слово правильное, то оно помещается в окно списка правильных слов `lbPosibleWord`; длина слова отражается в поле `stWordCount`;
         (7) создается новая перестановка в массиве `arrPerm`;
      iv) количество полученных комбинаций записать в `stTestLettersCount`;
      v) об окончании работы сообщить в `StaticText`.

Код обработчика события `OnClick` для кнопки «Найти слова» представлен в листинге 8.11.

Листинг 8.11. Обработчик события `OnClick` для кнопки «Найти слова»

```
void __fastcall TfoMain::bbFindWordsClick (TObject *Sender)
{
  StaticText->Caption = "Идет перебор слов";
  if (FWord.IsEmpty())  // если Word все же не стартовали
   {
      ShowMessage
             ("Приложение Word не присоединено!!! Увы!?");
      return;
   }

  int nN, mM, nComb;
  int iComb, ind;  // индексы
  int **ptrCombi = NULL;
  int *arrPerm = NULL;
           // очистка списков
  lbTestCombi->Clear();
  lbTestCommu->Clear();
  lbPosibleWord->Clear();

  nN = cseLetterCount->Value;  // общее кол-во букв
      // перебор количеств элементов в сочетании
      // от минимального значения к максимальному
  for (mM = cseMinCount->Value; mM <= cseMaxCount->Value; mM++)
   {
           // подсчет количества сочетаний из nN по mM
    nComb = CombiNM (nN, mM);
    stLetterCount->Caption = mM;  // количество букв в слове

           // может быть, требуется высвобождение памяти?
    if (ptrCombi) delete []ptrCombi;
    if (arrPerm) delete []arrPerm;

           // отведение памяти
    ptrCombi = new int * [nComb];
    for (iComb = 0; iComb < nComb; iComb++)
      ptrCombi [iComb] = new int [mM];
    arrPerm = new int [mM];

           // заполнение массива сочетаниями
    SimpleCombination (ptrCombi, nComb, nN, mM);

       // перебор текущего набора комбинаций
    for (iComb = 0; iComb < nComb; iComb++)
     {
                 // текущий номер сочетания
      stCompose->Caption = iComb;
                 // отображение текущего набора сочетаний
      AnsiString auxC = IntToStr (iComb + 1) + "|  ";
      for (int j = 0; j < mM; j++)
         auxC = auxC + IntToStr (ptrCombi [iComb][j]) + " ";
                 // добавление к списку
      lbTestCombi->Items->Add (auxC);
          // формирование перестановок и отображение их
```

```
      int num = 1;
      for (int i = 0; i < mM; i++) arrPerm [i] = i;
      do
        {
                        // заносим номер
           AnsiString auxP = IntToStr (num++) + "|  ";
                        // тестовая комбинация букв
           AnsiString auxW = "";
           for (int j = 0; j < mM; j++)
           {
             auxP=auxP+IntToStr
                  (ptrCombi[iComb][arrPerm [j]]) + " ";
          auxW = auxW + sgLetters->Cells
                       [ptrCombi [iComb][arrPerm [j]]-1][1];
           }
           if (lbTestCommu->Count < 512)
                  lbTestCommu->Items->Add (auxP);
           if (lbTestLetters->Count < 512)
                  lbTestLetters->Items->Add (auxW);
            // проверяем слово на правильность
           bool test = FWord.OleFunction
                        ("CheckSpelling", auxW.c_str());
         if (test)
         {
             lbPosibleWord->Items->Add (auxW);
          stWordCount->Caption = lbPosibleWord->Count;
          Application->ProcessMessages();
         }
        } while (NextPermutation0 (arrPerm, mM));
       lbTestCommu->Items->Add ("===");
      }  // iComb
    }  // for mM
  stTestLettersCount->Caption = lbTestLetters->Count;
  stTestLettersCount->Caption =
                        "Количество комбинаций = " +
                                  stTestLettersCount->Caption;

  StaticText->Caption = "Перебор окончен";
}
```

Обработчик события «на закрытие формы»

В действия обработчика события `OnClose` формы входят два действия: освободить память, занятую глобальными массивами; завершить сеанс с *Word*. Код обработчика `OnClose` формы представлен в листинге 8.12.

Листинг 8.12. Обработчик события «на закрытие формы»

```
void __fastcall TfoMain::FormClose (TObject *Sender,
                                              TCloseAction &Action)
{
  // освобождение памяти
  for (int iComb = 0; iComb < FNComb; iComb++)
    delete []FCombi [iComb];
  delete []FCombi;

  if (!FWord.IsEmpty())  // если Word стартовали
    FWord.OleProcedure ("Quit");  // останавливаем его
}
```

Обработчик события `OnChange` компонента `cseLetterCount`

Этот обработчик предназначен для установки размеров сетки. Код его реализации представлен в листинге 8.13.

Листинг 8.13. Установка размера сетки `sgLetters` на основании значения `cseLetterCount`

```
void __fastcall TfoMain::cseLetterCountChange
                                          (TObject *Sender)
{
    sgLetters->ColCount = cseLetterCount->Value;
}
```

Задача 8.6. Требуется получить все перестановки всех возможных сочетания из *n* по *m* элементов.

Решение

На начальном этапе решения предлагается воспользоваться результатами работы процедуры `SimpleCombination`, которая формирует глобальный массив `FCombi` размерностью в `FNComb` элементов, предоставляющий собой список всех возможных сочетаний из `nN` элементов по `mM` элементов. То есть будем считать, что массив `FCombi` всех возможных сочетаний уже сформирован. Тогда решение задачи будет сводиться к повторению с каждым сочетанием следующих действий:

- «взять» очередное сочетание;
- сгенерировать из этого сочетания все возможные перестановки.

Решение задачи можно построить как продолжение задачи 8.4, интерфейс которой представлен на рис. 8.4. Начальными условиями для выполнения задачи «Получить перестановки всех сочетаний» в таком случае будет решение задачи «Получить сочетания».

Предположим, что все перестановки всех сочетаний должны быть помещены в список `lbPermit`. На листингах 8.14 и 8.15 приведены тексты процедур для решения задачи: обработчик события «Получить перестановки всех сочетаний» (листинг 8.14); процедура `CombPermit` – для поиска перестановок всех сочетаний (эта процедура вызывается в обработчике события «Получить перестановки всех сочетаний»).

Листинг 8.14. Обработчик события под кнопку «Получить перестановки всех сочетаний»

```
void __fastcall TfoMain::actCombiPermitExecute(TObject *Sender)
{
  PageControl->ActivePage = tsPermit;
  int nPermiss, mM = cseM->Value;
  int *arrPerm;
  arrPerm = new int [mM];
  lbPermit->Clear ();

  CombPermit( arrPerm,  mM,  FNComb);
  delete[] arrPerm;
}
```

Листинг 8.15. Описание процедуры, отображающей перестановки всех сочетаний в списке lbPermit типа TListBox

```
void __fastcall TfoMain::CombPermit( int *arrPerm, int mM, int FNcomb)
{
  // все перестановки из сочетаний
  int num = 1;
  lbPermit->Items->Add ("Старт");
  for (int iComb = 0; iComb < FNComb; iComb++)
   {
    for (int i = 0; i < mM; i++) arrPerm [i] = i;
    do
     {
           // заносим номер
      AnsiString aux = IntToStr (num++) + "|  ";
      for (int j = 0; j < mM; j++)
        aux=aux+IntToStr (FCombi [iComb][arrPerm [j]])+" ";
      lbPermit->Items->Add (aux);
     } while (NextPermutation0 (arrPerm, mM));
   }
  lbPermit->Items->Add ("Финиш");
}
```

Задача 8.7. Создать латинский квадрат на базе заданного целочисленного массива, элементы которого представлены различными случайными числами из заданного диапазона.

Решение

Прежде чем рассмотреть решение задачи следует вспомнить понятие *латинского квадрата*.

ЛАТИНСКИЙ КВАДРАТ

Представляет собой квадратную матрицу порядка *n*, каждая строка которой и каждый столбец являются *перестановками* конечного множества *S*, состоящего из *n* элементов.

Предлагается следующий порядок решения:

- задать размер латинского квадрата;
- сгенерировать массив из *различных случайных* чисел – он будет служить основой для построения латинского квадрата;
- построить латинский квадрат;
- отобразить квадрат в поле списка.

Введем следующие обозначения переменных, которые объявим как глобальные:

```
int **FLatin; // латинский квадрат
int* FBasaList; // базовый массив для латинского квадрата
int FCount; // размер
```

Для создания массива FBasaList из неповторяющихся чисел можно создать отдельную процедуру на базе решения задачи 4.2, описанной в гл. 4 «Одномерный массив». Для построения латинского квадрата можно воспользоваться формулой:

$$FLatin[i][j] = FBasaList[i + j - 1(\text{mod } n)].$$

А само построение латинского квадрата можно реализовать, например, в виде обработчика события `OnClick` кнопки `lbGetLat`. Код обработчика события представлен в листинге 8.16. А описание процедуры `DoRandom` и используемой в ней процедуры `Detect` представлено в листингах 8.17 и 8.18.

Листинг 8.16. Описание процедуры-обработчика события `OnClick` кнопки `lbGetLat`, предназначенной для построения латинского квадрата

```
void __fastcall TfoMain::lbGetLatClick(TObject *Sender)
{
  FCount = seSizeLat->Value;
    // отведение памяти
  FLatin = new int * [ FCount];
  for (int iComb = 0; iComb <  FCount; iComb++)
    FLatin [iComb] = new int [FCount];
  FBasaList = new int [ FCount];
      // создание базового массива
  DoRandom (28, 156);
      // создание латинского квадрата - FLatin
  int i, j;
  for (i=0;i<FCount;i++)
    for(j=0; j<FCount;j++)
       FLatin[i][j] = FBasaList[(i+j-1)%5];
      // печать в списке lbLat
   for (i=0;i<FCount;i++)
     {
       AnsiString Str =""; // подготовка строки вывода
            // добавление элементов к строке вывода
       for(j=0; j<FCount;j++)
         Str = Str+FLatin[i][j]+"   ";
       lbLat->Items->Add(Str); // добавление к списку
     }
}
```

Листинг 8.17. Описание процедуры, предназначенной для генерации неповторяющихся случайных чисел из заданного интервала (создание базового массива)

```
void __fastcall TfoMain::DoRandom(int FRight, int FLeft)
// FBasaList - глобальная переменная, в которую помещается
// базовый массив для латинского квадрата
// Fright,FLeft - входные параметры, определяющие
// допустимые границы изменения случайных чисел
{
   for(int i=0; i<FCount; i++) // делать цикл
     {
       int R = random(FRight -FLeft)+FLeft; // создать число
       while (Detect(R,i)) // пока "число присутствует"
         R = random(FRight -FLeft)+FLeft; // пересоздать число
       FBasaList[i] = R; // занести в массив
     }
}
```

Листинг 8.18. Описание процедуры, предназначенной для проверки того факта, есть ли число `R` в созданном массиве `FBasaList` из `k` элементов

```
int __fastcall TfoMain::Detect (int R, int k)
// R - число, для которого выясняется факт присутствия его
// в сформированной части массива FBasaList, содержащей
```

```
// элементы с номерами от 0 до k включительно.
// Функция возвращает значение 1, если число
// обнаружено в массиве. И возвращает 0, если числа нет.
{
  for(int i=0; i<=k; i++) // делать цикл
    if (R == FBasaList[i]) return 1; // если есть совпадение
  return 0; // если нет факта совпадения
}
```



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. Проверить, являются ли заданные два латинских квадрата ортогональными.

Ортогональные латинские квадраты

Два латинских квадрата называются ортогональными, если $(a_{i,j}, b_{i,j}) \neq (a_{k,l}, b_{k,l}), \forall\ (i,j) \neq (k,l),\ i,j,k,l \in S = \{1 \ldots n\}$.

2. Проверить, являются ли заданные два латинских квадрата эквивалентными.

Эквивалентные латинские квадра-

Два латинских квадрата называются эквивалентными, если один из другого получается перестановкой строк, столбцов и переименованием элементов.

3. Имеется *n* предметов, вес которых равен, соответственно, $a_1, a_2 \ldots a_n$. Разделить эти предметы на две группы так, чтобы общий вес двух групп был максимально близок друг к другу.
4. Найти все такие расстановки восьми ферзей на шахматной доске, при которых ферзи не бьют друг друга.
5. «Задача о рюкзаке». Имеется *n* различных предметов, известен вес каждого из предметов и его стоимость. Определить, какие предметы надо положить в рюкзак, чтобы общий вес не превышал заданной границы, а общая стоимость была максимальна.
6. Даны целые числа: $a_1, a_2 \ldots a_n$. Определить, являются ли они перестановкой чисел 1, 2 … *n*.

Класс «Список» 9

Фундаментальными типами данных языка *C++* являются переменные, структуры и массивы из таких элементов. Массивы позволяют обрабатывать последовательно расположенные данные и предоставляют возможность прямого доступа к любому элементу по его индексу. Это дает возможность легко читать значение любого элемента в массиве или записывать в любой элемент новое значение. Но такая организация данных требует больших затрат при выполнении операций вставки или удаления элементов из массива [22].

Другая организация данных, обеспечивающая последовательное расположение элементов – это связные списки. Использование связных списков не требует изменения места хранения других элементов при решении задач, связанных с включением и исключением. Существует множество разновидностей списков: *линейные односвязные списки*, *стеки*, *очереди*, *деки*, *деревья*, *хеш-таблицы*, *двусвязные списки.*

Любую задачу с последовательным хранением данных можно привести к связному хранению данных, организованных в виде списков. Но есть такие задачи, при решении которых без списков трудно обойтись. Каждый представитель такого множества занимает свою нишу при решении задач определенного класса. Например, стеки чаще всего возникают в связи с алгоритмами, имеющими явно или неявно рекурсивный характер. Очереди обычно используются в организации промежуточных буферов, деки – при разработке редакторов, деревья и хеш-таблицы – при построении компиляторов.

Списки, в отличие от массивов, могут иметь переменное количество элементов. Простейшим представителем такой организации данных являются односвязные списки. Односвязный список позволяет также легко осуществлять вставку целого списка, как и вставку одного элемента. И удалить целый подсписок.

9.1. Разработка классов «Линейный односвязный список»

Элементы списка, в отличие от фундаментального типа «массив», принято называть *узлами* списка. И, если в случае массива, мы имеем дело с последовательностью однородных элементов, занимающий *непрерывный* участок памяти, то список представляет собой последовательность элементов/узлов с *произвольным расположением* в памяти. Связь каждого узла списка с соседними узлами производится через ссылку, которая должна содержаться в каждом узле списка. То есть узлы, образно говоря, как бы нанизаны на единую нитку и следуют последовательно друг за другом или выстроены в единую линейку. Поэтому списки часто называют линейными списками, подчеркивая это обстоятельство.

Помимо ссылки на соседний узел, каждый узел должен содержать информационное поле, ведь иначе в списке нет смысла. Информационное поле может иметь произвольную структуру и наполняться любым содержанием.

Одним из распространенных видов списков являются такие списки, каждый узел которых связан с соседними узлами только одной ссылкой – ссылкой на следующий элемент. Такие списки носят название *линейные односвязные списки*. На рис. 9.1 приведено схематическое изображение односвязного списка.

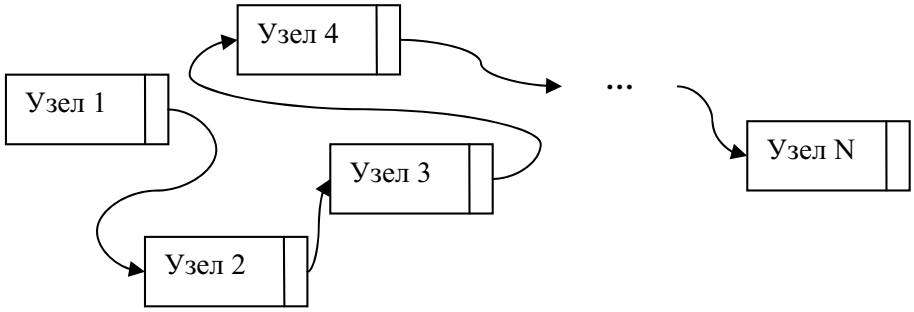


Рис. 9.1. Схематическое изображение линейного односвязного списка

Общая постановка задачи

Базовая задача. Требуется разработать семейство классов «Линейный односвязный список» со следующим набором операций:

- создать список;
- вставить элемент в список;
- удалить элемент из списка;
- добавить элемент в начало списка;
- добавить элемент в конец списка;
- изменить порядок следования узлов на обратный;
- отсортировать узлы списка;
- получить доступ к узлу списка для чтения его содержимого;
- вставить новый узел после узла с заданным адресом;
- определить количество узлов в списке.

Стратегия решения

Стратегия решения укладывается в стандартную стратегию, применяемую при разработке любого класса: разрабатывается иерархия классов; разрабатываются методы классов; создается *интерфейс приложения* и обеспечивается его *функциональность* в рамках поставленной задачи.

Решение

Перечисленные задачи связаны с преобразованием либо одного отдельного списка, либо затрагивают взаимодействие двух списков. Такие задачи также укладываются в поведение выделенного математического объекта «Линейный односвязный список». Можно специфицировать математический объект «Линейный односвязный список» и реализовать соответствующий класс.

Очевидно, что один список от другого может отличаться только содержимым информационной части/поля узла, поскольку весь механизм операций со списками обеспечивается посредством манипуляций со ссылками, связывающими узлы между собой. Потому на этапе разработки класса в качестве информационной составляющей узла можно взять переменную целого типа. Но, чтобы обеспечить независимость списка от заданного типа переменной, следует ввести для этого типа некоторое абстрактное имя, например `TDataType`. И если в данном случае с типом

TDataType будет связана простая переменная целого типа, в дальнейшем, при решении других задач, этот тип можно будет привязать к любой другой структуре.

Примечание

Введение типа TDataType в дальнейшем позволит просто переориентировать работу созданного списка на узел с другим составом информационного поля. Достаточно будет описать типом TDataType структуру узла с набором операций и переопределить методы, связанные с обработкой информационной части узлов.

Начинать разработку списка следует с описания структуры его узла. С учетом вышеизложенного, структура может быть описана так, как это показано в листинге 9.1.

Листинг 9.1. Описание структуры узла списка

```
typedef int TDataType; // тип узловых значений

struct TKnot  // узел стека
{
  TDataType Data; // информационное поле
  TKnot* Next; // адрес – указатель следующего узла
};
```

Примечание

Следует обратить внимание на тот факт, что в описании структуры TKnot присутствует поле Next, тип которого является указателем на описываемую структуру. На первый взгляд есть противоречие – структура Knot еще не описана, но уже появляется ссылка на TKnot. Такая конструкция разрешена в языке *C++*.

Как следует из определения списка и его схематического представления на рис. 9.1, доступ к элементам/узлам списка может быть осуществлен только по ссылкам, которые содержатся в каждом узле списка, начиная от его «заглавного» узла – начала списка. Для описания списка *как математического объекта* нужны две характеристики: *описание структуры узла* и описание переменной, предоставляющей *адрес начала списка* – начального узла. Ссылку на начальный узел назовем FTop. Эта информация позволит достигнуть любого узла списка и выполнить любую операцию из перечисленных операций в постановке задачи.

При решении некоторых задачах удобно обращаться к элементу связного списка по его индексу или порядковому номеру, как это имеет место в массивах. Тогда для проверки, находится ли индекс в допустимых пределах, можно использовать соответствующую переменную, которую назовем FCount – количество узлов в списке. Это аналог длины массива.

Таким образом, односвязный список можно определить тремя характеристиками: структурой узла, адресом начального узла, количеством узлов. Эти характеристики и будут использованы при описании класса «Линейный односвязный список».

По аналогии с рассуждениями, приведенными в анализе задач разработки классов, создаваться будут сразу три класса: класс базовый; класс расширенный; класс под задачу с элементами ввода/вывода. Введем следующие обозначения:

- базовый класс TDataList – класс «Список»;

- наследник `TListEx` – класс с расширенными возможностями;
- наследник `TListTask` от наследника `TListEx` – класс, дополненный методами, связанными с компонентами *C++ Builder*.

Примечание

При разработке классов необходимо помнить, что всегда приходится балансировать между «интеллектуальностью» класса и его простотой. Поэтому к методам базового класса, помимо конструкторов, деструктора и базовых методов, добавляются методы для выполнения вспомогательных операций – дополнительно к тем, которые требуются по условию. Эти простые операции будут использоваться в качестве «инструментов» при построении сложных алгоритмов.

Описание класса `TDataList`

Ввиду вышеизложенного, если подвергнуть анализу задачи, заявленные в постановке и стоящие перед классом, то очевидно, что некоторые алгоритмы со списками просты и их реализация не вызывает вычислительных трудностей. Например, легко добавлять новые узлы перед первым узлом. Но для добавления нового узла после последнего элемента необходимо пройти весь список, чтобы добраться до последнего элемента и только после этого удается добавить новый элемент. Для упрощения выполнения этой операции нужно хранить не только указатель на первый, но и на последний элемент. Не так просто решить задачу, связанную с сортировкой данных списка.

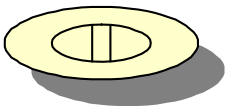
Поэтому в число методов базового класса `TDataList` следует включить дополнительные методы и поля – инструменты, не входящие напрямую в перечень задач, требуемых по условию. Это могут быть следующие методы: установить указатель списка в заданную позицию; передвинуть указатель списка вперед/назад; добавить элемент до указателя/за указателем узла списка; взять указатель элемента списка до заданного указателя/за указателем; переставить значения в узлах списка; удалить элемент списка до указателя/за указателем.

Эти методы/операции будут использоваться при построении алгоритмов сортировки узлов списка, при поиске и удалении узлов с заданными параметрами и др. Интерфейс класса `TDataList` представлен в листинге 9.2.

Листинг 9.2. Интерфейс базового класса `TDataList` «односвязный список»

```
class TDataList  // базовый класс "Список"
{
  protected:
    TKnot* FTop;  // указатель начала списка
    int    FCount;  // число узлов
  public:
    TDataList();
    ~TDataList();
    void   Do_List (char* FileName); // из файла FileName
                  // вставить aX в позицию по адресу aPtr
    void   Insert (TDataType aXNew, TKnot* aPtr);
           // добавить узел со значением aX к началу списка
    void   Insert (TDataType aX);
           // вернуть адрес узла со значеним aX в списке
    TKnot* Locate (TDataType aX);
```

```
      // вернуть значение X из узла с адресом aPtr
    TDataType Retrieved (TKnot* aPtr);
         // удалить узел с адресом aPtr списка
    void   Delete (TKnot* aPtr);
           // получить адрес узла, следующего за узлом
           // по адресу aPtr
    TKnot* NextKnot (TKnot* aPtr);
           // получить адрес предыдущего узла
           // по отношению к узлу  с адресом aPtr в списке
    TKnot* PrevKnot (TKnot* aPtr);
    void   MakeNull(); // сделать список пустым
           // показать значение X узла с номером aInd
    TDataType Show_Item (int aInd);
           // получить значение верхнего элемента
    TDataType GetTop() {return FTop->Data;}
           // получить значение нижнего элемента
    TDataType GetBottom() {return FTop->Data;}
           // удаляет элемент из вершины списка
    void Del_From_Top();
           // удаляет элемент из конца списка
    void Del_From_Bottom();
    int  GetQuantity(); // количество элементов в списке
};
```



МЕТОДЫ КЛАССА `TDataList`:

- `TDataList();` – конструктор без параметров. Устанавливает поля класса в «нулевые» значения;
- `~TDataList(){};` – деструктор;
- `void Do_List(char* FileName);` – метод класса. Создает список с данными, взятыми из файла `FileName`;
- `void Insert(TDataType aXNew, Knot* aPtr);` – метод класса с назначением – вставить `aXNew` в позицию по адресу `aPtr`;
- `void Insert (TDataType aX);` – метод класса. Добавляет узел со значением `aX` к началу списка;
- `Knot* Locate(TDataType aX);` – метод класса. Возвращает адрес узла со значением `aX` в списке;
- `int Retrieved(TKnot* aPtr);` – метод класса. Возвращает значение информационного поля узла `X`, расположенного по адресу `aPtr`;
- `void Del(TKnot* aPtr);` – метод класса. Удаляет из списка узел, расположенный по адресу `aPtr`;
- `Knot* NextKnot(Knot* aPtr);` – метод класса с назначением – получить адрес узла, следующего за узлом, расположенным по адресу `aPtr`;
- `TKnot* PrevKnot (TKnot* aPtr);` – метод класса с назначением – получить адрес узла, предыдущего по отношению к узлу, расположенного по адресу `aPtr`;
- `void MakeNull();` – метод класса. Делает список пустым;
- `TDataType Show_Item (int aInd);` – метод класса с назначением – показать значение узла с номером `aInd`;
- `TDataType GetTop();` – метод класса с назначением – получить значение верхнего элемента;

- `TDataType GetBottom() {return FTop->Data;}` – метод класса с назначением – получить значение нижнего элемента;
- `void Del_From_Top();` – метод класса. Удаляет элемент из вершины списка;
- `void Del_From_Bottom();` – метод класса. Удаляет элемент из конца списка;
- `int  GetQuantity();` – метод класса. Возвращает количество элементов в списке.

Описание конструктора класса `TDataList`

В классе `TDataList` всего один конструктор без параметров. Он устанавливает начало списка в `NULL` и счетчик узлов `FCount` в значение 0. Описание конструктора представлено в листинге 9.3.



Важно понимать, что в работе со списками доступ к узлам/элементам списка осуществляется только по ссылкам, которые содержатся в каждом узле, начиная от самого первого. Если какой-то узел существует, но на него нет ссылки из какого-либо узла списка, то такой узел является недоступным.

Листинг 9.3. Описание конструктора класса `TDataList`

```
TDataList::TDataList()  // Конструктор
{
  FCount = 0;
  FTop = NULL;
}
```

Описание деструктора класса `TDataList`

Действия деструктора сводятся к вызову метода `MakeNull`, назначение которого – сделать список пустым. Описание деструктора представлено в листинге 9.4.

Листинг 9.4. Описание деструктора класса `TDataList`

```
TDataList::~TDataList()  // Деструктор
{
  MakeNull ();
}
```

Создание списка

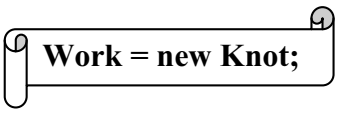
Метод `Do_List` имеет один входной параметр `aFileName` «имя файла». Для создания списка предусмотрен всего один метод, который берет данные из файла. Поэтому, если решается задача «создать список», то нужно выполнить подготовительную работу: предварительно создать текстовый файл, в который занести соответствующие числовые значения.

Предлагается алгоритм, в котором данные последовательно читаются из файла и заносятся в список таким образом, что первое прочитанное число становится последним элементом в списке, а последнее прочитанное – первым в списке. Адрес последнего элемента, помещенного в список, будет определять начало списка. Описание алгоритма, предназначенного для создания списка.

1. Устанавливаются начальные значения параметров списка на случай, если файла нет или в списке нет данных.

2. Открывается файл для чтения.
3. Выполняется проверка наличия файла с заданным именем. Если файла нет, то необходимо выйти из процедуры.
4. Производится проверка, что файл содержит данные. Если файл пустой, то необходимо выйти из процедуры.
5. Выполнить в цикле следующее, пока в файле есть данные.
   5.1. Прочитать из файла число.
   5.2. Создать новый узел.
   5.3. Заполнить поля созданного узла.
   5.4. Переопределить начало списка.
   5.5. Увеличить число узлов.

Для реализации такого алгоритма целесообразно ввести дополнительные переменные: для чтения очередного числа из файла (`Value`) и для временного хранения узла (`Work, Work1`). Для создания `Work`-узла необходимо воспользоваться операцией `new`.



В результате этой операции в динамической памяти выделяется область (резервируется место) размером, необходимым для структуры типа `Knot`, а в указателе `Work` запоминается адрес этой области.

Описание метода создания списка `Do_List` приведено в листинге 9.5.

Листинг 9.5. Описание метода `Do_List`, предназначенного для создания списка со значениями элементов, взятыми из файла с именем a`FileName`

```
void TDataList::Do_List (char* aFileName)
{ fstream fin (aFileName, ios::in);
  if (fin == NULL) return;
  FTop = NULL;
  int value;
  TKnot *work, *work1;
  while (!fin.eof())
   {fin >> value;
    work = new TKnot;  // порождаем элемент
    work->Next = NULL;  // и заполняем его
    work->Data = value;
    FCount++;
           // при порождении первого элемента
    if (FTop == NULL) {FTop = work; work1 = FTop;}
            // при порождении очередного
    else {work1->Next = work; work1 = work;}
   }
  fin.close();
}
```

Вставить значение по заданному адресу

Метод `Insert` имеет два входных параметра. Параметр `aXNew` предоставляет значение, которое нужно поместить в список. Параметр `aPtr` предоставляет адрес в списке, куда нужно поместить это значение. Логика этой задачи заключается в следующем. Пусть в списке есть узел с заданным адресом. Этот адрес, по существу, определяет место, где должен быть «поставлен» новый элемент. При этом все прежние элементы, а также их порядок должны остаться неизменными. То есть задачу

можно переформулировать: «Вставить в список новый элемент, расположив его до заданного указателя».

Цикл, который лежит в основе алгоритма, организуется по ссылочной переменной. В цикле производится переход от одного узла к другому до тех пор, пока ссылка узла, расположенного за текущим узлом, не совпадет с заданным адресом. Это позволит остановиться/найти узел, после которого должен быть поставлен новый элемент. После чего остается выполнить шаги:

а) запомнить ссылку от узла предшествующего, так как этой ссылкой будет пользоваться новый элемент;
б) создать новый узел; заполнить поля нового узла: в поле данных поместить заданное значение, а в поле ссылки сохраненную ссылку для этих целей;
в) изменить ссылку предшествующего узла на адрес созданного элемента.

Код программы представлен в листинге 9.6.

Листинг 9.6. Описание метода `Insert`, предназначенного для добавления нового узла к списку и размещения его по адресу `aPtr`

```
void TDataList::Insert (TDataType aXNew, TKnot* aPtr)
  // вставляет объект aXNew в позицию по адресу aPtr
{ if (FTop == NULL || aPtr == NULL) return;
  TKnot* work, *work1, *work2;
  work = FTop;
  while (work->Next->Next != aPtr)
    work = work->Next;
  work2 = work->Next;
  work1 = new TKnot;
  work1->Data = aXNew;
  work1->Next = work2;
  work = work1;
  FCount++;
}
```

Вставить значение в начало списка

Метод `Insert` имеет один параметр `aXNew`, который предоставляет значение для помещения его в начало списка. Текст алгоритма представлен в листинге 9.7.

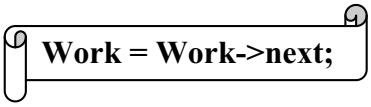
Листинг 9.7. Описание метода `Insert`, предназначенного для добавления нового узла к списку

```
void TDataList::Insert (TDataType aXNew)
  // добавить узел со значением aX к началу списка
{ if (FTop == NULL) return;
  TKnot* work = new TKnot;
  work->Data = aXNew;
  work->Next = FTop->Next;
  work = FTop;
  FCount++;
}
```

Поиск узла с заданным числовым значением

Метод `Locate` имеет один входной параметр `aX`, адрес которого требуется найти в списке и вернуть как результат работы. Алгоритм поиска узла с требуемым значением информационного поля прост. Список просматривается от начала до конца с использованием вспомогательной переменной ссылочного типа – ссылка на узел (например, с именем `Work`). При просмотре значение параметра сравнивается со

значением узла списка. Для перехода к следующему узлу списка используется операция `work = work->Next;`



Эта операция выполняется для присваивания указателю `Work` в качестве нового значения ссылки на следующий узел в списке.

Значение, расположенное в поле структуры с именем `x`, при просмотре сравнивается с заданным значением. Этот процесс заканчивается в двух случаях:

а) очередной узел списка содержит требуемое число; в этом случае процедура возвращает адрес найденного узла;
б) просмотрен весь список, а узел с требуемым значением не найден; в этом случае процедура возвращает `NULL`.

Текст с описанием этой процедуры представлен в листинге 9.8.

Листинг 9.8. Описание метода `Locate`, предназначенного вернуть адрес узла с заданным значением `Newx` в его информационном поле

```
TKnot* TDataList::Locate (TDataType aX)
  /* Функция возвращает адрес aX в списке */
  // Если в списке aX встречается несколько раз,
  // то возвращается адрес первого от начала
  // Если aX нет в списке, то возвращается NULL
{ TKnot* work;
  work = FTop;
  while (work != NULL && work->Data != aX)
    work = work->Next;
  return (work);
}
```

Получить значение узла по заданному адресу

Метод `Retrieved` имеет один входной параметр `aPtr`, задающий адрес узла в списке, значение которого требуется найти и вернуть как результат работы. Алгоритм поиска узла с заданным адресом с целью вернуть числовое значение этого узла содержит один цикл. В этом цикле просматриваются узлы списка от начала до конца с использованием переменной типа ссылки на узел. При просмотре адрес очередного узла списка сравнивается с заданным адресом. Этот процесс заканчивается в двух случаях:

а) найден узел с указанным адресом; в этом случае процедура возвращает значение из структуры узла;
б) список исчерпан, а узел с нужным адресом не найден; в этом случае процедура возвращает – `MaxInt` (условное число – сигнал). Результат не определен, если нет элемента с таким адресом.

Текст с описанием этой процедуры представлен в листинге 9.9.

Листинг 9.9. Функция `Retrieved` возвращает значение информационного поля элемента, который стоит по адресу `p` в списке

```
TDataType TDataList::Retrieved (TKnot* aPtr)
{ TKnot* work;
  work = FTop;
     // пока не aPtr и не конец списка
```

```
  while (work != NULL && work != aPtr)
    work = work->Next; // переход к следующему узлу
     // если объекта нет в списке
  if (work == NULL) return -MaxInt;
  return (work->Data);
}
```

Удаление узла по указанному адресу

Метод `Delete` имеет один входной параметр `aPtr`, определяющий позицию в списке для удаления узла. Для исключения элемента из списка достаточно изменить ссылку у предшествующего ему узла, причем в качестве новой ссылки у этого элемента надо принять ссылку, которая содержится в исключаемом элементе. Таким образом, задача может быть сведена к поиску узла, предшествующего узлу удаляемому.

Основу алгоритма составляет цикл, в котором используется ссылочная переменная. Поскольку удаление узла связано с операцией, в которой требуется изменить ссылку в узле, предыдущему по отношению к удаляемому, в алгоритм вводится дополнительная переменная типа ссылки. Эта переменная будет хранить адрес предыдущего узла. Список просматривается от начала до конца. При этом ссылочная переменная сверяется с адресом узла, который нужно удалить. Этот процесс заканчивается в двух случаях:

а) найден узел с указанным адресом; в этом случае ссылка предыдущего узла изменяет свое значение на адрес последующего узла, а количество узлов уменьшается на 1;
б) список исчерпан, а узел с нужным адресом не найден; в этом случае список не изменяется.

В самом начале следует проверить, не совпадает ли искомый узел с началом списка. Если так, то решение задачи сводится к поправке начала списка и процедура заканчивается. Удаление не выполняется, если в списке нет позиции `aPtr` или `aPtr = NULL`. Текст с описанием представлен в листинге 9.10.

Листинг 9.10. Описание метода `Delete`, предназначенного для удаления узла с заданным адресом

```
void TDataList::Delete (TKnot* aPtr)
  //
{ if (FTop == aPtr) // Если элемент в вершине
   {TKnot *aux = FTop;
    FTop = FTop->Next;
    delete aux;
   }
  TKnot *work, // рабочая переменная для просмотра списка
         *work1; // хранит позицию предыдущего узла

  work = FTop;    // встать в начало списка
  // пока адрес не найден и не конец списка
  while (work != aPtr && work != NULL)
   {
    work1 = work; // запомнить адрес
    work = work->Next;// перейти к следующему узлу
   }
  if (work != NULL)// если адрес найден
   {
     // удалить узел
```

```
    work1->Next = aPtr->Next;
    delete work;
   }
  if (FCount > 0) FCount--;// уменьшить число узлов
}
```



Следует обратить внимание, что при удалении элемента из списка не производится операция высвобождения памяти, занимаемой удаляемым элементом. Удаляемая ячейка просто «остается за бортом» полезной памяти. Такое стремление к быстродействию породило недостатки. В приведенном варианте реализации метода добавлен оператор `delete`, чтобы высвободить память.

Поиск узла, расположенного после заданного указателя

Метод `NextKnot` имеет один входной параметр `aPtr`, определяющий позицию в списке для поиска адреса узла, следующего за ним. Этот адрес будет возвращаться через заголовок метода.

Список просматривается в цикле от начала до *предпоследнего* элемента. Если искомый узел – последний в цикле, то после него искомого элемента нет. При проходе цикла ссылочная переменная сверяется с адресом узла, который нужно найти. Этот процесс заканчивается в двух случаях:

а) найден узел с указанным адресом; в этом случае ссылка возвращает адрес следующего узла;
б) список исчерпан, а узел с нужным адресом не найден; в этом случае возвращается `NULL`.

Текст с описанием метода `NextKnot` представлен в листинге 9.11.

Листинг 9.11. Описание метода `NextKnot`, предназначенного для определения адреса узла, на который ссылается элемент узла списка, расположенный по заданному адресу

```
TKnot* TDataList::NextKnot (TKnot* aPtr)
  // Если aPtr - последняя позиция в списке, то
  // NEXTKnot(p) = NULL.
  // Результат функции NEXTKnot не определен, когда
  // aPtr = NULL.
{ TKnot* work;
  work = FTop; // встать на вершину
      // пока узел не найден и не конец списка
  while (work->Next != NULL && work != aPtr)
    work = work->Next; // переход к следующему узлу
  return work->Next; // вернуть адрес
}
```

Поиск узла, расположенного до заданного указателя

Метод `PrevKnot` имеет один входной параметр `aPtr`, определяющий позицию в списке для поиска адреса узла, расположенного перед ним. Этот адрес будет возвращаться через заголовок метода.

Решение сводится к циклу, в котором используется ссылочная переменная. Вводится также дополнительная ссылочная переменная. Эта переменная будет хранить *адрес предыдущего узла*. Список просматривается от начала до конца. При

этом ссылочная переменная сверяется с адресом узла, который нужно найти. Этот процесс заканчивается в одном из двух случаев:

а) найден узел с указанным адресом; в этом случае возвращается адрес предшествующего узла;
б) список исчерпан, а узел с нужным адресом не найден; в этом случае возвращается `NULL`.

В самом начале следует проверить, не совпадает ли искомый узел с началом списка. Если так, то возвращается начало списка и процедура заканчивается. Текст с описанием представлен в листинге 9.12.

Листинг 9.12. Описание метода, который возвращает указатель, предыдущий по отношению к заданному

```
TKnot* TDataList::PrevKnot (TKnot* aPtr)
  // Возвращает предыдущий указатель перед указателем aPtr
  // в списке
  // Результат функции PREVIOUS не определен,
  // если aPtr совпадает с вершиной или aPtr = NULL.
{
  if (FTop == aPtr) return NULL; // если адрес - вершина
  TKnot *work;
  work = FTop;
// если искомый узел - следующий от начала списка
  while (work->Next != aPtr && work->Next != NULL)
    work = work->Next;
  if (work->Next == NULL) return NULL;
  return work;
}
```

Сделать список пустым

Метод `MakeNull` не имеет параметров. Назначение метода – высвободить память, отведенную под узлы списка. Алгоритм сводится к циклу. В цикле просматриваются все узлы списка. Как только получен адрес следующего узла (`work`), предыдущий узел (`work1`) удаляется (высвобождается память для последующего использования). Текст с описанием представлен в листинге 9.13.

Листинг 9.13. Описание метода, предназначенного для высвобождения памяти, занятой узлами списка

```
void TDataList::MakeNull()
{
  TKnot *work, *work1;
  work = work1 = FTop; // встать в начало
  if (work == NULL) return;  // список и так пуст
  do        // делать цикл
   {
    work = work->Next;  // на следующий элемент
    delete work1;       // удаляем предыдущий
    work1 = work;       // тоже на следующий
   } while (work != NULL); // пока не конец списка
            // обнулить поля класса
  FTop = NULL;
  FCount = 0;
}
```

Удаление узла, расположенного на дне списка

Метод `Del_From_Bottom` не имеет параметров. Алгоритм содержит цикл, в котором используется ссылочная переменная. Эта переменная служит для перехода от узла к узлу – от начала списка до его предпоследнего элемента. В найденном предпоследнем элементе изменяется ссылка. Если раньше она указывала на последний элемент, то теперь должна указывать на `NULL`. После чего последний узел уничтожается. Текст алгоритма представлен в листинге 9.14.

Листинг 9.14. Описание метода `DelLast`, предназначенного для удаления узла, расположенного в конце списка

```
void TDataList::Del_From_Bottom()
{
  TKnot *work;
  work = FTop; // встать на вершину
      // пока не предпоследний элемент
  while (work->Next->Next != NULL)
    work = work->Next; // переход к следующему узлу
  delete (work->Next); // удалить элемент
  work->Next = NULL;
  if (FCount > 0) FCount--; // уменьшить число узлов
}
```

Вернуть количество узлов списка

Метод `GetQuantity` не имеет параметров. Алгоритм сводится к одному оператору `return`. Код программы представлен в листинге 9.15.

Листинг 9.15. Описание метода, предназначенного вернуть число узлов в списке

```
int TDataList::GetQuantity ()  // Возвращает количество узлов
{
  return FCount;
}
```

Удаление узла, расположенного в вершине списка

Метод `Del_From_Top` не имеет параметров. Алгоритм сводится к двум действиям: изменения адреса вершины и удаления «верхнего» узла. Текст алгоритма представлен в листинге 9.16.

Листинг 9.16. Описание метода, предназначенного для удаления узла, расположенного в вершине списка

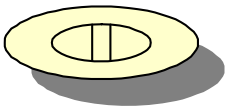
```
void TDataList::Del_From_Top()
{
  TKnot* work = FTop;
  FTop = FTop->Next; // изменение адреса
  delete work; // удаление узла
}
```

Описание класса `TDataListEx`

Следует отметить, что на этапе разработки расширенный класс-потомок `TDataListEx` содержит всего несколько методов (листинг 9.3). Этот класс по замыслу авторов должен служить своего рода заготовкой, куда будут помещаться новые методы, предназначенные для решения задач с использованием классов «Линейный односвязный список». Интерфейс класса представлен в листинге 9.17.

Листинг 9.17. Интерфейс расширенного класса «Список»

```
class TDataListEx : public TDataList
{
  protected:
  public:
    TDataListEx();
    void Purge();  // удаление совпадающих элементов
    int  TheSame (TDataType aX, TDataType aY);  // сравнение
                      // меняет местами значения
    void Replace (TKnot* aPtr1,  TKnot* aPtr2);
                      // меняет местами элементы
    void ReplacePtr (TKnot* aPtr1,  TKnot* aPtr2);
    void Sort(); // упорядочивает по Data
};
```



МЕТОДЫ КЛАССА `TDataListEx`:

- `TDataListEx();` – конструктор без параметров. Инициализирует поля класса;
- `void Purge();` – метод для удаления совпадающих элементов;
- `int  TheSame (TDataType, TDataType aY);` – метод для сравнение элементов `aX` и `aY`;
- `void Replace (TKnot* aPtr1, TKnot* aPtr2);` – метод класса. Меняет местами значения в элементах с адресами `aPtr1` и `aPtr2`;
- `void ReplacePtr (TKnot* aPtr1, TKnot* aPtr2);` – метод класса. Меняет местами элементы в адресах `aPtr1` и `aPtr2`;
- `void Sort();` – метод класса. Упорядочивает список по возрастанию значений.

Удаление совпадающих элементов

Метод `Purge` не имеет параметров.

Алгоритм реализуется с помощью двух вложенных циклов. Внешний цикл с помощью ссылочной переменной обеспечивает переход от одного узла списка к другому. Значение этой ссылочной переменной проверяется на повторяемость с помощью внутреннего цикла. Для этого значение переменной внешнего цикла последовательно сравнивается (функцией `TheSame`) со всеми значениями (функция `Retrieved`) внутреннего цикла, и если наблюдается совпадение, то узел, определенный внутренней переменной, удаляется (функцией `Del`) из списка.

В алгоритм включены две вспомогательные переменные: `ptr` – для просмотра элементов списка; `qtr` – для просмотра элементов, следующих за позицией `ptr`. Порядок действий.

1. В начальное значение – вершину списка – устанавливается `ptr`.
2. Выполняется цикл, пока в списке есть элементы:
   а) `qtr` устанавливается в начальное значение – принимает адрес элемента, следующего за элементом `ptr`;
   б) выполняется цикл, пока правее элемента `ptr` есть другие элементы:
      i) если совпадают значения узлов `ptr` и `qtr`, то
         (1) запомнить адрес следующего элемента;
         (2) удалить совпадающий;
         (3) установить новое значение `qtr`;

(4) если не совпадают значения узлов `ptr` и `qtr`, то переход к следующему узлу за узлом `qtr`;

ii) перейти к следующему узлу за узлом `ptr`.

Текст алгоритма представлен в листинге 9.18.

Листинг 9.18. Удаление совпадающих элементов

```
void TDataListEx::Purge()
{
  TKnot *ptr,
        *qtr;
  ptr = FTop; // встать на вершину
  while (ptr != NULL) // пока список не пуст
   {
    qtr = ptr->Next;  // начинаем цикл со следующего за ptr
    while (qtr != NULL) // пока не конец списка
     {
            // если элементы совпадают
      if (TheSame (Retrieved (ptr), Retrieved (qtr)))
                // запоминаем адрес следующего
       {
         TKnot *aux = qtr->Next;
         Delete (qtr);
         qtr = aux;  // устанавливаем адрес,
                   // с которого будем продолжать
       }
      else
        qtr = qtr->Next; // к следующему узлу
     }
    ptr = ptr->Next; // к следующему узлу
   }
}
```

Проверка значений на совпадение

Метод `TheSame` имеет два входных параметра: `aX` и `aY`. Возвращает значение `true` (истина), если `aX` и `aY` «одинаковые», и `false` (ложь) – в противном случае. Текст алгоритма представлен в листинге 9.19.

Листинг 9.19. Описание метода `TheSame`, проверяющего значения, переданные через параметры, на совпадение

```
int TDataListEx::TheSame (TDataType aX, TDataType aY)

{ if (aX == aY) return 1;
  else return 0;
}
```

Перестановка значений элементов списка с заданными адресами

Метод `Replace` имеет два входных параметра: `aPtr1` и `aPtr2`. Алгоритм сводится к трем операциям присваивания с использованием вспомогательной переменной. Описание реализации метода приведено в листинге 9.20.

Листинг 9.20. Описание метода `RePlace`, предназначенного для перестановки элементов с заданными адресами

```
void TDataListEx::Replace (TKnot* aPtr1,  TKnot* aPtr2)
  // Меняет местами значения элементов с адресами aPtr1 и aPtr2
```

```
{ TDataType aux = aPtr1->Data;
  aPtr1->Data = aPtr2->Data;
  aPtr2->Data = aux;
}
```

Перестановка элементов списка с заданными адресами

Метод `ReplacePtr` имеет два входных параметра: `aPtr1` и `aPtr2`. Если адреса переставляемых элементов совпадают, то переставлять нечего, тогда достаточно просто выйти из подпрограммы. Иначе выполняется перестановка, при которой используются две вспомогательные переменные: `work1` и `work2`. Адрес предыдущего узла по отношению к узлу `aPtr1` хранит `work1`. Адрес предыдущего узла по отношению к узлу `aPtr2` хранит `work2`. Алгоритм сводится к следующим шагам:

1) если `aPtr1` – не вершина списка, то в `work1` заносится адрес предыдущего узла;
2) иначе, если `aPtr1` – вершина списка, то в `work1` заносится `NULL`;
3) если `aPtr2` – не вершина списка, то в `work2` заносится адрес предыдущего узла;
4) иначе, если `aPtr2` – вершина списка, то в `work2` заносится `NULL`;
5) если `work1` – не `NULL`, то в адресное поле узла записывается `aPtr2`;
6) иначе, если `work1` – `NULL`, то вершиной списка становится `aPtr2`;
7) если `work2` – не `NULL`, то в адресное поле узла записывается `aPtr1`;
8) иначе, если `work2` – `NULL`, то вершиной списка становится `aPtr1`;
9) меняются значения адресных полей в узлах `aPtr1` и `aPtr2`.

Описание метода `ReplacePtr` приведено в листинге 9.21.

Листинг 9.21. Описание метода перестановки элементов с заданными адресами

```
void TDataListEx::ReplacePtr (TKnot* aPtr1,  TKnot* aPtr2)
  // Меняет местами элементы в адресах aPtr1 и aPtr2
{ if (aPtr1 == NULL || aPtr2 == NULL) return;
  TKnot *work1, *work2;
  if (aPtr1 != FTop) work1 = PrevKnot (aPtr1);
  else work1 = NULL;
  if (aPtr2 != FTop) work2 = PrevKnot(aPtr2);
  else work2 = NULL;
  if (work1 != NULL) work1->Next = aPtr2;
  else FTop = aPtr2;
  if (work2 != NULL) work2->Next = aPtr1;
  else FTop = aPtr1;
  TKnot *aux = aPtr1->Next; aPtr1->Next = aPtr2->Next;
                            aPtr1->Next = aux;
}
```

Сортировка данных в списке

Функция Sort располагает значения в узлах списка в порядке возрастания. Для сортировки списка используется метод, основанный на поиске наименьшего элемента.

Сортировка начинается с того, что в списке находится узел с наименьшим числовым значением. Это значение помещается в первый узел списка. Для того чтобы не потерялось значение, находящееся в первом узле, оно помещается в найденный узел с наименьшим числовым значением. Таким образом производится перестановка значений. Затем процедура повторяется, но список просматривается не от начала, а от второго по порядку следования узла списка. Процесс продолжается до тех пор,

пока не сформируется предпоследний элемент списка. Тогда последний узел примет автоматически свое минимальное значение.

Для реализации этого алгоритма потребуется два цикла с использованием ссылочных переменных. Один цикл – внешний (по `Work1`) – перебирает узлы, в которые нужно поместить найденное очередное минимальное значение. Второй цикл – внутренний (по `Work2`) – используется для поиска наименьшего значения. Вспомогательная переменная `Work3` используется для хранения адреса минимального элемента. Описание метода представлено в листинге 9.22.

Листинг 9.22. Описание метода `Sort`, предназначенного для сортировки данных списка

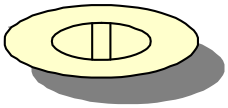
```
void TDataListEx::Sort()
{
  TKnot *work1, *work2, *work3; // вспомогательные переменные
  AnsiString ptr;
  work1 = FTop; // инициализация переменной внешнего цикла
  while (work1->Next != NULL) // пока в списке есть узлы
   {
    work2 = work1; // адреса минимума
    work3 = work2->Next;
    while (work2 != NULL)
     {
      if (work2->Data < work3->Data) // проверка на минимум
        work3 = work2; // запомнить адрес минимума
            // переход к следующему узлу во внутреннем цикле
      work2 = work2->Next;
     };
    ptr = IntToStr (work1->Data) + "  " +
                                    IntToStr (work3->Data);
    Replace (work1, work3); // перестановка значений
            // переход к следующему узлу во внешнем цикле
    work1 = work1->Next;
 };
}
```

Описание класса `TDataListTask`

В интерфейс класса включены методы, связанные с отображением данных в окна визуальных компонентов. Описание интерфейса класса представлено в листинге 9.23.

Листинг 9.23. Интерфейс класса `TDataListTask` – потомка класса `TDataListEx`

```
class TListTask : public TDataListEx
{ protected:
  public:
    TListTask();  // Пустой конструктор
    TListTask (char *);  // Данные читаются из файла
            // отображать по индексу
    void Show_By_Index (TListBox *, bool = true);
            // отображать по индексам, в обратном порядке
    void Show_BackOrder (TListBox *, bool = true);
            // показать на ListBox
    void Show (TListBox *, bool = true);
            // показать список номеров
    void Show_Nums (TListBox *, bool = true);
};
```



МЕТОДЫ КЛАССА `TListTask`:

- `TListTask();` – пустой конструктор;
- `TListTask (char * FileName);` – конструктор. Через параметр `FileName` передается имя файла, из которого читаются данные;
- `void Show_By_Index (TListBox *aLBx, bool aClear);` – метод класса. Предназначен отображать список по индексу. Параметр `aLBx` определяет имя списка типа `TListBox`, в который нужно поместить значения сформированного в классе списка. Параметр `aClear` – признак, указывающий чистить/не чистить `aLBx`;
- `void Show_BackOrder (TListBox *, bool = true);` – метод класса. Предназначен отображать список по индексу в обратном порядке. Параметр `aLBx` определяет имя списка типа `TListBox`, в который нужно поместить значения сформированного в классе списка. Параметр `aClear` – признак, указывающий чистить/не чистить `aLBx`;
- `void Show (TListBox *, bool = true);` – метод класса. Предназначен отображать список. Параметр `aLBx` определяет имя списка типа `TListBox`, в который нужно поместить значения сформированного в классе списка. Параметр `aClear` признак, указывающий чистить/не чистить `aLBx`;
- `void Show_Nums (TListBox *, bool = true);` – метод класса. Предназначен отображать список номеров. Параметр `aLBx` определяет имя списка типа `TListBox`, в который нужно поместить значения сформированного в классе списка. Параметр `aClear` – признак, указывающий чистить/не чистить `aLBx`.

Отображать по индексу

Метод `Show_By_Index` имеет два параметра. Параметр `aLBx` определяет имя списка типа `TListBox`, в который нужно поместить значения сформированного в классе списка. Второй параметр `aClear` логического типа служит указанием, следует ли предварительно чистить/не чистить `aLBx`. Алгоритм содержит один цикл (по числу элементов в списке), в котором

- берется значение `i` элемента;
- это значение преобразуется в строку `AnsiString`, которая добавляется к списку `aLBx`.

Описание метода представлено в листинге 9.24.

Листинг 9.24. Описание метода, предназначенного для отображения списка в прямом порядке по его индексу

```
void TListTask::Show_By_Index (TListBox *aLBx, bool aClear)
{
  if (aClear) aLBx->Clear ();  // если надо -- чистим
  for (int i = 0; i < GetQuantity(); i--)
   {TDataType temp = Show_Item (i);
    AnsiString Ptr = IntToStr (temp);
    aLBx->Items->Add (Ptr);
   }
            // на первую строку
  if (aLBx->Count > 0) aLBx->ItemIndex = 0;
}
```

Отображать по индексам в обратном порядке

Метод `Show_BackOrder` имеет два параметра. Параметр `aLBx` определяет имя списка типа `TListBox`, в который нужно поместить значения сформированного в классе списка. Второй параметр `aClear` логического типа служит указанием, следует ли предварительно чистить/не чистить `aLBx`. Отличие от метода, представленного в листинге 9.24, только в порядке выбора номера элемента в списке класса. Описание метода представлено в листинге 9.25.

Листинг 9.25. Описание метода, предназначенного для отображения списка в обратном порядке по его индексу

```
void TListTask::Show_BackOrder (TListBox *aLBx, bool aClear)
{ if (aClear) aLBx->Clear ();  // если надо -- чистим
  for (int i = GetQuantity()-1; i >= 0; i--)
   {TDataType temp = Show_Item (i);
    AnsiString Ptr = IntToStr (temp);
    aLBx->Items->Add (Ptr);
   }
      // на первую строку
  if (aLBx->Count > 0) aLBx->ItemIndex = 0;
}
```

Показать на `ListBox`

Метод `Show` имеет два параметра. Параметр `aLBx` определяет имя списка типа `TListBox`, в который нужно поместить значения сформированного в классе списка. Второй параметр `aClear` логического типа служит указанием, следует ли предварительно чистить/не чистить `aLBx`. Отличие от метода, представленного в листинге 9.24, состоит только в способе доступа к очередному элементу: используется прием `work = work->Next;`. Описание метода представлено в листинге 9.26.

Листинг 9.26. Описание метода, предназначенного для отображения списка в прямом порядке по его индексу

```
void TListTask::Show (TListBox *aLBx, bool aClear)
{ if (aClear) aLBx->Clear ();  // если надо -- чистим
  TKnot *work;
  work = FTop;
  while (work != NULL)
   {aLBx->Items->Add (IntToStr (work->Data));
    work = work->Next;
   };
            // на первую строку
  if (aLBx->Count > 0) aLBx->ItemIndex = 0;
}
```

Показать список номеров

Метод `Show_Nums` имеет два параметра. Параметр `aLBx` определяет имя списка типа `TListBox`, в который нужно поместить значения номеров элементов, сформированного в классе списка. Второй параметр `aClear` логического типа служит указанием, следует ли предварительно чистить/не чистить `aLBx`. Алгоритм содержит один цикл (по числу элементов), в котором номера (то есть параметр цикла) преобразуется в строку и добавляется к `aLBx`. Описание метода представлено в листинге 9.27.

Листинг 9.27. Описание метода, предназначенного для показа номеров списка, созданного в классе «Список»

```
void TListTask::Show_Nums (TListBox *aLBx, bool aClear)
  // показать список номеров
{ if (aClear) aLBx->Clear ();  // если надо -- чистим
  for (int i = 1; i <= GetQuantity(); i++)
    aLBx->Items->Add (IntToStr (i));
}
```

9.2. Разработка классов «Стеки»

Помимо линейных списков общего вида, широкое распространение получили списки, в которых операции вставки, удаления узлов и доступа к данным осуществляется только через первый или последний узел. Одним из часто применяемых типов организации данных является *стек магазинного типа.* Обычно такую организацию данных называют коротко – *стек*. Узлы пополняют такой список или покидают его, подчиняясь правилу «последним пришел, первым ушел», или LIFO (*last-in*, *first-out*). Формирование такого списка подобно стопке тетрадей с контрольными работами: ученики складывают их друг на друга, формируя стопку, а учитель потом при проверке берет их, начиная с верхней тетради. Итак, получаем порядок, при котором тетрадь, сданная на проверку последней, подвергается проверке первой. На рис. 9.2. представлена схема, иллюстрирующая механизм формирования стека.

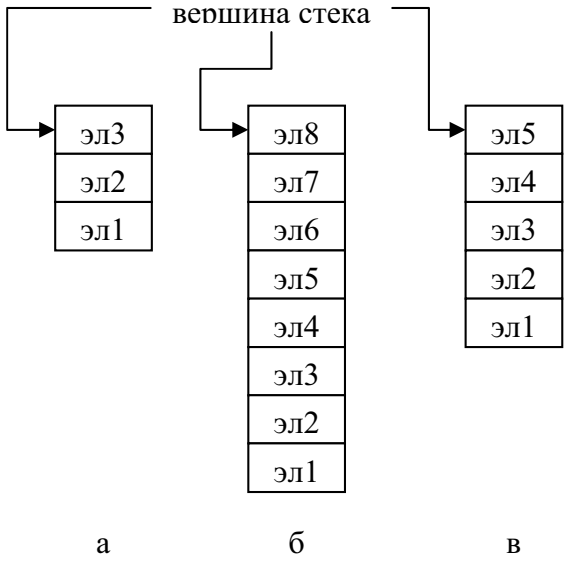


Рис. 9.2. Состояние стека на разных фазах работы: а) некое начальное состояние, три элемента в стеке; б) после вставки пяти элементов; в) после удаления трех элементов

Класс, реализующий стек на базе линейного односвязного списка, является частным случаем от класса «Список». Но здесь допустимы следующие операции: создание стека; инициализация пустого списка, то есть стека; добавление элемента перед первым элементом и удаления первого элемента. Начальный узел стека называют *вершиной стека*. И все перечисленные операции выполняются только над этим элементом. Базовый класс стека может быть представлен интерфейсом, как это показано в листинге 9.28.

Листинг 9.28. Интерфейс базового класса «Стек»

```
class TDataStack
{ protected:
    TKnot* FTop;  // левый узел
    int FCount;  // число узлов
  public:
    TDataStack();
   ~TDataStack();
    void Do_Stack (char* aFileName); // из файла aFileName
    TDataType GetTop() {return FTop->Data;}
    void Add_To_Top (TDataType aValue);  // добавить элемент
    void Del_From_Top();  // удалить элемент из стека
    void MakeNull();  // сделать стек пустым
    int  GetQuantity();  // количество элементов в стеке
};
```

Примечание

В обозначении данных, заявленных в интерфейсе класса `TDataStack`, изменены только названия методов, а их реализация *полностью совпадает* с реализацией соответствующих методов класса «Линейный односвязный список».

9.3. Разработка классов «Очереди»

На линейном односвязном списке также легко реализовать тип данных с другой организацией – *очередь*. Этот тип данных характеризуется отношением «первым пришел, первым ушел», или FIFO (*first-in*, *first-out*). Такую организацию данных проще реализовывать над односвязным списком со ссылками на начальный и конечный элементы `FTop` и `FBottom`. Тогда легко реализовать добавление нового элемента в конец списка с исправлением ссылки `FBottom` и удалением первого элемента с исправлением ссылки `FTop`. Эта структура данных используется для моделирования реальных очередей с целью определения их характеристик, например: средняя длина очереди; время пребывания заказа в очереди; вероятность отказа в постановке в очередь при ограничении ее максимальной длины.

Над очередью определяются всего две операции: занесение элемента в очередь и выбор элемента из очереди. При этом выбранный элемент исключается из очереди.

При реализации модели на основе списка, организованного по типу «Очередь», поступают следующим образом:

- очередной элемент, который необходимо подвергнуть обработке, добавляют в очередь;
- прежде чем обработать очередной элемент его извлекают из очереди; извлекается, конечно, первый элемент в очереди; затем этот элемент, уже исключенный из очереди, подвергают обработке.

Для полной аналогии живой очереди в магазине с «компьютерной очередью» можно считать, что очередь – это те покупатели, которые стоят в очереди в кассу, но еще не расплачиваются; а когда касса освобождается, то покупатель, который является первым в очереди, отделяется (извлекается из нее) и оплачивает покупку (подвергается обработке).

При организации данных в список, организованный по типу «Очередь», нужно поступить следующим образом: очередной элемент, который необходимо подвергнуть обработке, добавляют в очередь. При необходимости обработать очередной элемент, извлекают этот элемент (он должен быть первым) из очереди и подвергают обработке.

В отличие от линейного *односвязного списка*, в *очереди*, в соответствии с ее определением, должны быть доступны две позиции:

- начало – из этой позиции удаляется элемент из очереди;
- конец – в эту позицию помещается новый элемент, заносимый в очередь.

Этот факт вносит свою лепту в описании математического объекта «Очередь». А именно, к полям класса (в отличие от предыдущих классов) добавляется поле – указатель конца очереди. И, как следствие, изменяются алгоритмы, обеспечивающие выполнение операций над списками типа «очередь». В листинге 9.29 представлен интерфейс базового класса «Очередь».

Листинг 9.29. Интерфейс базового класса «Очередь»

```
class TDataQueue
{ protected:
    TKnot* FLeft;  // левый узел
    TKnot* FRight;  // правый узел
    int FCount;  // число узлов
  public:
    TDataQueue();
   ~TDataQueue();
      // создать очередь с данными из файла aFileName
    void Do_Queue (char* aFileName);

    TDataType GetLeft() {return FLeft->Data;}
    TDataType GetRight() {return FRight->Data;}
    void Del_From_Left();  // удалить элемент слева
    void MakeNull();  // сделать очередь пустой
           // добавить элемент справа
    void Add_To_Right (TDataType aValue);
    int  GetQuantity();  // количество элементов в стеке
};
```

Согласно листингу 9.29, в число полей базового класса, по сравнению с предыдущими списками, добавлено одно поле и, соответственно, откорректированы названия полей с учетом их смысловой нагрузки. А в числе методов *оставлены только те, которые следуют из определения очереди*. В алгоритмы методов внесены соответствующие поправки.

Изменился конструктор: в тело конструктора класса «Очередь» добавлен оператор, который инициализирует добавленное поле. Описание конструктора базового класса представлено в листинге 9.30.

Листинг 9.30. Описание конструктора базового класса «Очередь»

```
TDataQueue::TDataQueue()
{
  Left = Right = NULL;
  FCount = 0;
}
```

Создание очереди

В методе `Do_Queue`, предназначенном для создания очереди, предусмотрен единственный параметр `aFileName` – имя текстового файла. Как это было описано в разделе 9.1 «Линейные односвязные списки», данные предварительно следует разместить в файле. При обращении к этому файлу из программы, в общем случае целесообразно предусмотреть следующие ситуации:

а) файл не открывается (не найден); в этом случае очередь пуста;
б) в файле нет данных; в этом случае очередь также пуста;
в) в файле одно число; в этом случае очередь состоит из одного элемента, а, следовательно, начало и конец очереди совпадают;
г) в файле множество данных (больше одного).

Все эти ситуации отражены в алгоритме `Do_Queue`, описывающем создание очереди. В предложенном алгоритме формирование начального состояния очереди производится по схеме, уже описанной в алгоритме создания линейного списка. Поэтому при создании очереди первый элемент из файла поступает в конец очереди, а последний встает в ее начало. Отличие данного алгоритма в том, что при поступлении первого значения фиксируется правая граница очереди и соответствующая ссылка заносится в область памяти параметра класса `Right` – конца очереди. Описание метода представлено в листинге 9.17.

Листинг 9.17. Создание очереди

```
void TDataQueue::Do_Queue (char* aFileName)
  // создать очередь с данными из файла aFileName
{
      // инициализация начальных значений
  FLeft = FRight = NULL;
  FCount = 0;
      // открыть файл для чтения
  fstream fin (aFileName, ios::in);
  if (fin == NULL) return;// если файла нет
  if (fin.eof() )return; // если файл пуст
  TDataType value; // для чтения текущего значения
  TKnot* work; // текущий узел
      // очередь из одного элемента
  fin >> value; // чтение числа
  work = new TKnot; // создание узла
  work->Next = NULL;
  work->Data = value;
  FLeft = FRight = work;
      // если есть еще элементы
  while (!fin.eof()) // пока файл не пуст
   {
    fin >> value; // чтение числа
    work = new TKnot; // создание узла
    work->Next = FLeft;
    work->Data = value;
    FLeft = work; // переопределение левой границы
    FCount++; // увеличения количества
   };
  fin.close(); // закрыть файл
}
```

Удаление элемента из очереди

Метод `Del_From_Left` не имеет параметров. Исключение элемента по определению должно производиться из начала (со стороны левой границы) очереди. Поэтому для исключения элемента из очереди достаточно изменить адрес начала очереди `FLeft`. В качестве нового начала нужно воспользоваться ссылкой (принять ссылку), которая содержится в исключаемом – первом элементе очереди. Описание метода представлено в листинге 9.18.

Листинг 9.18. Удаление элементов из очереди

```
void TDataQueue::Del_From_Left()  // удалить элемент слева
{ FLeft = FLeft->Next;
  FCount--;
}
```

Добавление элемента к очереди

Метод `Add_To_Right` содержит один входной параметр – значение, которое нужно добавить к очереди. По логике организации очереди этот элемент должен быть добавлен к концу (правой границе) очереди. Поэтому в результате работы алгоритма должно быть изменено значение правой границы очереди. Алгоритм сводится к следующим шагам:

а) отводится место под новый узел;
б) формируются поля структуры узла; в информационное поле помещается заданное число, а в поле ссылки – `NULL` (элемент будет последним в очереди);
в) изменяется ссылка в правом узле очереди, если раньше это была ссылка `NULL`, то теперь она будет указывать на новый элемент;
г) поправляется правая граница (она изменяет свое значение на адрес нового элемента).

Описание алгоритма представлено в листинге 9.19.

Листинг 9.19. Добавление элемента `aValue` к очереди

```
void TDataQueue::Add_To_Right (TDataType aValue)
{
  TKnot* work; // вспомогательная переменная
  work = new TKnot; // создание нового узла
  work->Data = aValue;
  work->Next = NULL;
  FRight->Next = work; // подсоединение к очереди
  FRight= work;// поправка правой границы
  FCount++; // увеличение счетчика элементов
}
```

9.4. Разработка классов «Деки»

Рассмотренные выше однонаправленные списки предусматривают жесткий порядок перебора элементов. Но иногда возникают ситуации, когда необходимо не только двигаться вперед по списку, но и возвращаться назад, например, для того, чтобы посмотреть, а возможно и изменить предыдущие элементы. Для этого предусмотрена более сложная организация данных – это *дек*. Упрощенно *дек* можно представлять как *стек*, но применяемый к обоим концам списка, то есть и добавление, и удаление производится как над началом, так и над концом списка.

Такой список можно представить некоторой цепочкой, в которой каждый узел содержит ссылку не только на следующий узел, но также и на узел, предшествующий ему. Схематически дек может быть представлен, как на рис. 9.4.

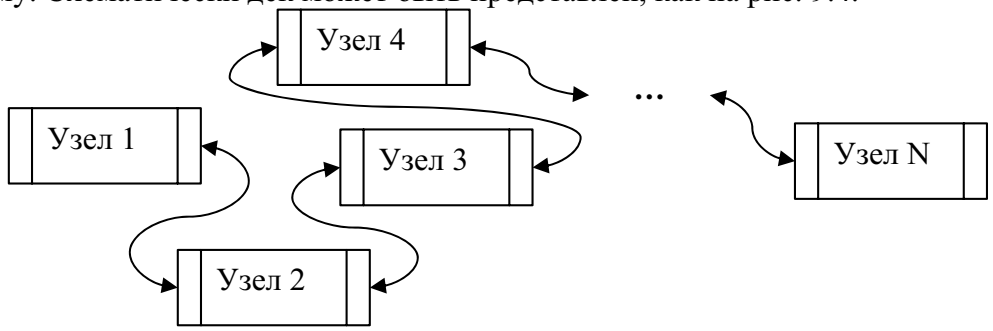


Рис. 9.4. Схематическое изображение двусвязного списка «Дек»



Очевидно, что добавление новой связи к структуре узла, *существенным образом отражается на разработке* методов класса, несмотря на общую для всех списков идеологию. Для того чтобы разобраться со спецификой дека, предложим к рассмотрению, как некоторый вариант разработки, интерфейс некоторого базового класса `DataDeck` и разработку методов этого класса.

Представим структуру узла дека в виде, показанном в листинге 9.20.

Листинг 9.20. Описание структуры узла дека

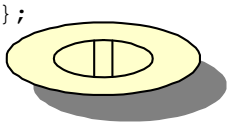
```
struct TNode  // Узел дека
{
  TDataType Data; // значение узла
  TNode *Next; // ссылка на следующий узел
  TNode *Pred; // ссылка на предшествующий узел
};
```

Как это имело место при создании классов «Очередь», математический объект, определенный нами как дек, можно охарактеризовать тремя параметрами: началом дека, концом дека и числом узлов. Эти параметры будут составлять поля класса «Дек». А в методы класса включим набор методов, стандартный для стека: создать дек; удалить элемент слева/справа; добавить элемент слева/справа. Интерфейс базового класса представлен в листинге 9.21.

Листинг 9.21. Интерфейс базового класса «Дек»

```
class TDataDeck
{ protected:
    TNode *FLeft, *FRight;// начало и конец дека
    int  FCount; // количество узлов
  public:
    TDataDeck(); // конструктор без параметров
    void Do_Deck (char* aFileName);// создать дек
    void AddToLeft (TDataType aValue);// добавить узел
    void AddToRight (TDataType aValue); // добавить узел
    void DelFromLeft();// удалить узел
    void DelFromRight();// удалить узел
    TDataType  Show_LeftItem (); // показать от начала
```

```
    TDataType  Show_RightItem ();// показать от конца
    int  GetQuantity(); // вернуть количество
    void MakeNull();  // сделать дек пустым
};
```



МЕТОДЫ КЛАССА `TDataDeck`:

- `TDataDeck ();` – конструктор без параметров. Устанавливает поля класса в «нулевые» значения;
- `~TDataDeck ();` – деструктор;
- `void Do_Deck (char* FileName);` – метод класса. Создает список с данными, взятыми из файла `FileName`;
- `void AddToLeft (TDataType aValue);` – метод класса. Добавляет узел со значением `aValue` к началу списка;
- `void AddToRight (TDataType aValue);` – метод класса. Добавить узел со значением `aValue` к концу списка;
- `void DelFromLeft();` – метод класса. Удаляет из списка левый узел;
- `void DelFromRight();` – метод класса. Удаляет из списка правый узел;
- `TDataType  Show_LeftItem ();` – метод класса. Показывает значение левого узла;
- `TDataType  Show_RightItem ();` – метод класса. Показывает значение правого узла;
- `int  GetQuantity();` – метод класса. Возвращает количество элементов в списке;
- `void MakeNull();` – метод класса. Делает список пустым.

Описание конструктора

Конструктор обнуляет поля класса. Описание конструктора представлено в листинге 9.22.

Листинг 9.22. Описание «пустого» конструктора базового класса «Дек»

```
Deck::Deck()
{
  FCount = 0;
  Left = Right = NULL;
}
```

Создание дека

Метод `Do_Deck` имеет один входной параметр `aFileName` (имя файла) и предназначен для создания дека. Данные для создания дека должны быть предварительно размещены в текстовом файле, как это было принято при организации работ над списками. При создании дека рассматриваются следующие случаи:

а) файл не открывается (не найден); в этом случае дек пуст;
б) в файле нет данных; в этом случае дек также пуст;
в) в файле одно число; в этом случае дек состоит из одного элемента, а следовательно, начало и конец дека совпадают;
г) в файле множество данных (больше одного).

Все эти ситуации должны быть отражены в алгоритме. В предложенном к рассмотрению алгоритме, формирование начального состояния дека производится аналогично схеме, уже описанной в алгоритме создания очереди. Особенность данного алгоритма состоит в том, что в процессе создания формируются две ссылки узла: левая и правая. Следует заметить, что в начальном узле дека в поле `pred` фигурирует пустая ссылка `NULL`, так как у начального узла нет предыдущего элемента. Также и у последнего узла – в поле `next` фигурирует пустая ссылка `NULL` (для него нет последующего). Описание метода `Do_Deck` представлено в листинге 9.23.

Листинг 9.23. Создание дека с данными, взятыми из файла

```
void TDataDeck::Do_Deck (char* aFileName)
{
            // начальные значения границ
  FLeft = NULL;
  FRight = NULL;
            // открыть файл для чтения
  fstream fin (aFileName, ios::in);
  if (fin == NULL) return; // если нет такого файла
  if (fin.eof()) return;  // если нет данных в файле
  TNode *work; // вспомогательная переменная для узла
  TDataType value; // вспомогательная переменная для числа
  fin >> value; // прочитать число из файла
  FCount++; // увеличить счетчик
      // создать дек из одного узла
  work = new TNode;
  work->Next = NULL;
  work->Data = value;
  work->Pred = NULL;
  FLeft = work;
  FRight = work;
      // добавлять узлы к деку, пока есть данные в файле
  while (!fin.eof())
   {
    fin >> value; // читать число
    work = new TNode; // новый узел
    work->Next = FLeft; // заполнить адрес «направо»
    work->Next->Pred = work; // заполнить адрес «налево»
    work->Data = value; // заполнить числовое поле
    FLeft = work; // переопределить левую границу
    FCount++; // увеличить счетчик узлов
   };
      // заполнить адрес «налево» у начального узла дека
  FLeft->Pred = NULL;
  fin.close(); // закрыть файл
}
```

Добавление элемента к концу дека

Метод `AddToRight` имеет один параметр. Включение в дек нового элемента `aValue`, значение которого передается через параметр метода, производится по следующей схеме: создается новый узел; заносится значение в соответствующее ему поле структуры; формируются ссылки созданного узла; корректируются ссылки параметров дека – конечного узла, поскольку элемент добавляется к концу дека. Метод добавления нового узла к концу дека приведен в листинге 9.24.

Листинг 9.24. Добавление элемента к концу дека

```
void TDataDeck::AddToRight (TDataType aValue)
{ TNode *work;
  work = new TNode;  // отвести память
  work->Data = aValue; // заполнить числовое поле
  work->Next = NULL; // заполнить адрес «направо»
  work->Pred = FRight; // заполнить адрес «налево»
      // переопределить адрес «направо» в крайнем узле
  FRight->Next = work;
  FRight = work; // переопределить правую границу
  FCount++; // увеличить счетчик узлов
}
```

Добавление элемента к началу дека

Метод `AddToLeft` имеет один параметр. Включение в дек нового элемента `aValue`, значение которого передается через параметр метода, производится по следующей схеме: создается новый узел; заносится значение в соответствующее ему поле структуры; формируются ссылки созданного узла; корректируются ссылки параметров дека – начального узла, поскольку элемент добавляется к началу дека. Добавление нового узла к началу дека приведено в листинге 9.25.

Листинг 9.25. Добавление элемента к началу дека

```
void TDataDeck::AddToLeft (TDataType aValue)
{
  TNode *work;
  work = new TNode;  // отвести память под узел
  work->Data = aValue; // заполнить числовое поле
  work->Next = FLeft; // заполнить адрес «направо»
  FLeft->Pred = work; // заполнить адрес «налево»
        // переопределить адрес «налево» в начальном узле
  work->Pred = NULL;
  FLeft = work; // переопределить левую границу
  FCount++; // увеличить счетчик узлов
}
```

Удаление элемента из начала дека

Метод `DelFromLeft` не имеет параметров. Исключение элемента просто сводится к изменению ссылки, которая определяет начало дека, так как исключается элемент первый от начала. Описание метода приведено в листинге 9.26.

Листинг 9.26. Удаление первого элемента от начала дека

```
void TDataDeck::DelFromLeft()
{ TNode *aux = FLeft;  // запоминание адреса первого узла
  FLeft = FLeft->Next; // переопределение границы
  FLeft->Pred = NULL; // переопределение адреса «налево»
  delete aux; // удаление первого узла
  FCount--; // уменьшение счетчика узлов
}
```

Удаление элемента от конца дека

Метод `DelFromRight` не имеет параметров. Исключение элемента просто сводится к изменению ссылки, которая определяет конец дека, поскольку удаляется элемент, первый от конца дека. Описание метода приведено в листинге 9.27.

Листинг 9.27. Удаление первого элемента от конца дека

```
void TDataDeck::DelFromRight()
{ TNode *aux = FRight; // запоминание адреса крайнего узла
  FRight = FRight->Pred; // переопределение границы
  FRight->Next = NULL; // переопределение адреса «направо»
  delete aux; // удаление крайнего узла
  FCount--; // уменьшение счетчика узлов
}
```

Сделать список пустым

Метод `MakeNull` не имеет параметров. Назначение метода – высвободить память, отведенную под узлы списка. Алгоритм сводится к циклу. В цикле просматриваются все узлы списка. Как только получен адрес следующего узла `work`, предыдущий узел `work1` удаляется – перенастраивается указатель на текущий элемент и высвобождается память для последующего использования. Текст с описанием метода представлен в листинге 9.28.

Листинг 9.28. Описание метода, предназначенного «опустошить» дек

```
void TDataDeck::MakeNull()
  // Эта функция делает дек пустым
{ TNode *work, *work1; // вспомогательные переменные
  work = work1 = FLeft;
  if (work == NULL) return;  // список и так пуст
  do  // делать цикл
   {work = work->Next;  // на следующий элемент
    delete work1;       // удаляем текущий
    work1 = work;       // тоже на следующий
   } while (work != NULL); // пока есть узлы
            // обнуление полей класса
  FLeft = FRight = NULL;
  FCount = 0;
}
```

Примечание

Ниже представлено в качестве примера описание интерфейса и решение задач только для класса «Списки». Решение задач с использованием других описанных выше классов выполняется аналогично, поэтому демонстрировать их не является целесообразным.

9.5. Разработка интерфейса приложения для решения задач с классом «Список»

Решение задач со списковым типом данных выполнено в рамках одной формы так, как это показано на рис. 9.3. Как это видно из рисунка, все рабочее поле формы занято блокнотом `PageControl`, в котором предусмотрены страницы для каждого из разработанных списков.

На странице «Списки» расположены следующие компоненты:

- 2 компонента `ListBox`:
  - `lbList` – для отображения числовых значений списка;
  - `lbListNums` – для отображения порядковых номеров узлов списка;
- 3 компонента `Edit`:
  - `edListNumOf` – для хранения количества узлов;

- edListElem – для задания числового значения по условию задачи;
- edInfo – для отображения значения, расположенного в начале списка.

Перечень возможных операций над списком представлен в списке компонента RadioGroup с именем rgList. В данном случае это единственный компонент, который *обеспечивает функциональность* приложения, решая задачи, назначения которых следуют из названия.

Такая блокнотная организация удобна, поскольку для добавления новых задач, всегда можно открыть новую страницу и оформить на ней решение независимо от других страниц.

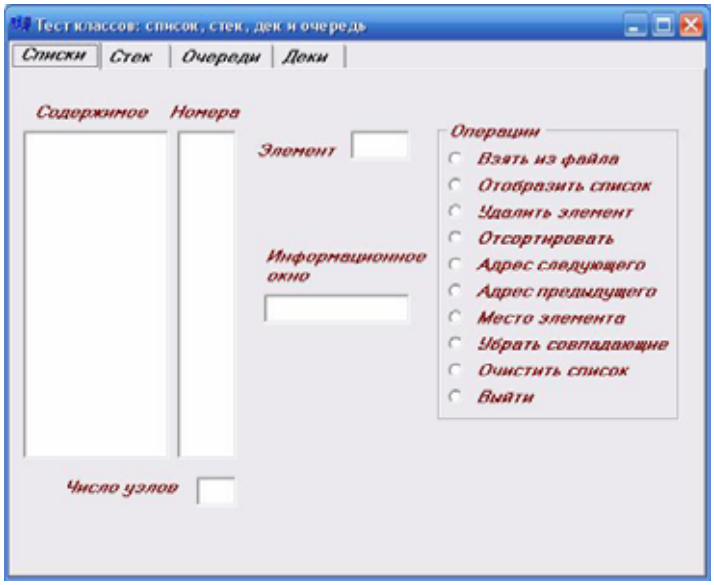


Рис. 9.3. Интерфейс приложения, представленного страницей блокнота для работы с классом «Список»

В данном проекте показано нестандартное применение компонента RadioButton(rgList). Здесь RadioButton не только регулирует/определяет, какую команду необходимо выполнить, но и является инициатором этой команды. Помимо этого, он же играет роль «протокола». По отмеченному пункту в RadioButton видно, какая из представленных операций была выполнена последней.

Отметим

Отдельно отметим, что на старте программы одному из пунктов компонента RadioButton принято устанавливать отметку. Здесь же отметка не устанавливается и показывает тот момент, что еще ни одной команды на этой закладке не исполнялось.

В представленном приложении используются следующие глобальные переменные классового типа:

```
TListTask  *gLWork;  // для списка
TStackTask *gSWork; // для стека
TDeckTask  *gDWork; // для дека
TQueueTask *gQWork; // для очереди
```

В листинге 9.29 представлен текст обработчика события OnClick компонента rgList, обеспечивающего выполнение перечисленных в нем действий над списка-

ми. В предложенном алгоритме процедуры-обработчика события, выбор действий, связанных с операцией, производится с помощью переключателя. С этой целью в алгоритм вводится вспомогательная переменная `temp`. В эту переменную `temp` по команде `OnClick` компонента `rgList` заносится номер выделенной строки/операции. А переключатель, в зависимости от принятого значения, уже направляет к выполнению соответствующих действий.

Листинг 9.29. Описание обработчика события `OnClick` компонента `rgList`

```
void __fastcall TfoLiStDeQue::rgListClick(TObject *Sender)
{
            // занести в temp номер операции
  int temp = rgList->ItemIndex;
  switch (temp)  // выполнить переключение по temp
   {
      // если в temp содержится номер 0 – «Взять из файла»
      case 0:
      gLWork = new TListTask; // создать экземпляр класса
      if (dlgOpenFile->Execute()) // выполнить OpenDialog
            // создать список из файла
        gLWork->Do_List (dlgOpenFile->FileName.c_str());
      else delete gLWork; // если с файлом неудача
      break;
      // если в temp содержится номер 1- «Отобразить список»
    case 1:
     {
      gLWork->Show (lbList);  // отобразить список
                  // заполнить список номеров
      gLWork->Show_Nums (lbListNums);
      edListNumOf->Text = gLWork->GetQuantity(); // количество
                  // запомнить значение в вершине списка
      TDataType aux = gLWork->GetTop();
      edInfo->Text = IntToStr (aux); // отобразить значение
      lbInfo->Caption = "Первый элемент списка";
      break;
     }
      // если в temp содержится номер 2 – «Удалить элемент»
    case 2:
     {
            // запомнить номер выделенного в списке элемента
      AnsiString aux = lbList->Items->
                                    Strings[lbList->ItemIndex];
      TDataType newX = StrToInt (aux);
      TKnot *ptr = gLWork->Locate (newX); // получить адрес

      lbInfo->Caption = "Удален элемент";
      edInfo->Text = ptr->Data;  // заслать число;
                        // конвертирование автоматическое
      gLWork->Delete (ptr);  // удалить элемент из списка

      gLWork->Show (lbList); // показать измененный список
      gLWork->Show_Nums(lbListNums); // показать номера
                                     // отобразить количество
      edListNumOf->Text = IntToStr (temp);
      break;
     }
      // если в temp содержится номер 3 – «Отсортировать»
    case 3:
```

```
      {
       gLWork->Sort(); // отсортировать
       gLWork->Show (lbList); // отобразить отсортированный
       gLWork->Show_Nums (lbListNums); // показать номера
       break;
      }
       // если в temp содержится номер 4 – «Адрес следующего»
     case 4:
      {
            // запомнить номер выделенного в списке элемента
       TDataType newX = StrToInt (edListElem->Text);
       TKnot *ptr = gLWork->Locate (newX); // взять адрес
       TKnot *ptr1;
       ptr1 = gLWork->NextKnot (ptr); // взять адрес следующего
       lbInfo->Caption = "Следующий элемент";
       if (ptr1 != NULL) // если следующий элемент есть
         edInfo->Text = IntToStr (ptr1->Data);
       else edInfo->Text = "Нет такого!!!";
       break;
      }
       // если в temp содержится номер 5 – «Адрес предыдущего»
     case 5:
      {
       // запомнить номер выделенного в списке элемента
       TDataType newX = StrToInt (edListElem->Text);
       TKnot *ptr = gLWork->Locate (newX); // взять адрес
                        //взять адрес предыдущего
       TKnot *ptr1 = gLWork->PrevKnot (ptr);
       lbInfo->Caption = "Предыдущий элемент";
       if (ptr1 != NULL) // если предыдущий есть в наличии
         edInfo->Text = IntToStr (ptr1->Data);
       else edInfo->Text = "Нет такого!!!";
       break;
      }
       // если в temp содержится номер 6 – «Место элемента»
     case 6:
      {
       // запомнить номер выделенного в списке элемента
       TDataType newX = StrToInt(edListElem->Text);
       TKnot *ptr = gLWork->Locate (newX); // взять адрес
       newX = gLWork->Retrieved (ptr); // получить значение
       lbInfo->Caption = "Текущий элемент";
       if (newX != -MaxInt) // если адрес в списке есть
         edInfo->Text = IntToStr (newX); // отобразить
       else edInfo->Text = "Нет такого!!!";
       break;
      }
// если в temp содержится номер 7 – «Убрать совпадающие»
     case 7:
      {
       gLWork->Purge(); // убрать совпадающие
       gLWork->Show (lbList); // показать исправленный список
       gLWork->Show_Nums (lbListNums); // показать номера
                  // показать новое количество
       edListNumOf->Text = gLWork->GetQuantity();
       break;
      }
// если в temp содержится номер 8 – «Очистить список»
```

```
    case 8:
     {
      gLWork->MakeNull(); // обнулить список
      gLWork->Show (lbList); // показать список
      gLWork->Show_Nums (lbListNums); // номера
      edListNumOf->Text = gLWork->GetQuantity(); // количество
      break;
     }
    default: // в остальных случаях
      Close(); // выход
   }
}
```

9.6. Решение задач с использованием классов «Список»

Представленные к решению задачи опираются на разработанную систему классов. Решение задачи в том случае, если оно не может быть напрямую реализовано одним из методов класса, оформляется в виде отдельного метода класса. Рекомендуется включить этот метод в расширенный класс, например в класс `FListEx`.

Примечание

В дальнейшем при пояснении задачи будет представлена к рассмотрению только та часть решения, которая *относится к разработке метода решения*. И не будет затрагиваться интерфейсная сторона разработки приложения. Эта сторона решения задачи настолько проста, что не нуждается в дополнительных объяснениях. Для решения очередной задачи можно подготовить новую страницу в блокноте формы, а можно воспользоваться уже созданной страницей, добавив к ней соответствующую команду «Решить задачу…».

Задача 9.1. Смоделировать работу магазина самообслуживания с одним кассовым аппаратом.

Решение

Работу магазина самообслуживания с одним кассовым аппаратом упрощенно можно смоделировать двумя списками, точнее списком `A` и очередью `B`. Список `A` имитирует приход покупателя в зал и уход из зала к кассе. В список будут помещаться клиенты, представляемые набором временных характеристик: временем захода в магазин и временем пребывания в магазине до того момента, когда клиент займет свою очередь у кассы.

Список пополняется в произвольный момент времени. Предположим, что известна вероятность появления нового клиента в течение единицы времени, и эта вероятность равна `p`. Очевидно, что в один и тот же момент времени не может появиться более одного клиента. Время пребывания элемента в магазине имитирует время, которое покупатель тратит на выбор товара и наполнение корзины.

Очередь `B` строится в порядке возрастания временных оценок, которые получены на основании: времени захода в магазин; времени выбора товара. Следует отметить, что время «встать в очередь к кассе» и время «получить расчет у кассира» может сильно отличаться – это зависит от времени подхода к кассе других покупа-

телей и времени, потраченным кассиром на обслуживание предыдущих клиентов. Очевидно, что для определения того времени, которое потратил покупатель с номером N на приобретение товара, необходимо найти разницу между временем его выхода из магазина и временем захода в магазин.

Для создания списка покупателей и очереди покупателей у кассы можно воспользоваться классами «Список» и «Очередь».

Поскольку задача достаточно сложная и требует множество специальных методов, для решения задачи целесообразно создать новый класс – наследник классов «Список» и «Очередь».



При использовании классов «Список» и «Очередь» *необходимо переопределить тип* T, который связан с описанием информационной части узла и представляет ее описание. В структуре T, предназначенной для описания клиентов магазина самообслуживания, следует представить их временные характеристики.

Предлагается следующий порядок решения задачи.

1. Разрабатывается структура T для описания характеристик клиентов магазина самообслуживания.
2. Разрабатываются методы класса FListTask, необходимые для ввода и отображения списка покупателей.
3. Определяется список действий пользователя, необходимых для решения задачи.
4. Разрабатывается интерфейс приложения.
5. Подготавливаются обработчики событий для обеспечения функциональности приложения в соответствии с выстроенной схемой.

Из перечисленных пунктов решения задачи наибольшую сложность представляет пункт, связанный с разработкой структуры T. Разработка остальных вопросов укладывается в рамки изложенного материала. А в качестве примера будет представлено описание всего лишь трех процедур: две процедуры для отображения списка покупателей; обработчик события под кнопку «Создать список покупателей».

Разработка структуры T

Каждому покупателю для его идентификации присвоим порядковый номер. При этом заметим, что увеличение номера покупателя не будет говорить о возрастании времени его захода в магазин. Так как время захода будет разыгрываться датчиком случайных чисел. Помимо номера покупатели будут обладать следующими характеристиками: временем захода в магазин; временем, затраченным на приобретение товара; временем подхода к кассе.



Кроме этого, следует отметить, что при использовании методов классов «Список» и «Очередь» со структурами узлов выполняются различные операции. И если в базовых классах с типом T был связан стандартный базовый тип, который обладал базовым набором операций, то для представляемой структуры этот набор операций следует определить ДЛЯ ТОГО, ЧТОБЫ ОПЕРАЦИИ В БАЗОВЫХ КЛАССАХ ВЫПОЛНЯЛИСЬ КОРРЕКТНО.

Поэтому в структуру с типом T, предназначенную для описания информационной составляющей узла списка, помимо полей, *следует добавить набор операций*. Описание структуры представлено в листинге 9.30.

Листинг 9.30. Описание структуры T

```
struct T    // информационная часть узла
    {
       int FNumber; // Номер покупателя
       TDateTime ForEnter;   // Время захода в магазин
       TDateTime ForSelect; // Время выбора
       TDateTime ForKasse;  // Время подхода к кассе
       T operator=(T);  // копирование/присваивание
       T operator=(int);  // формирование узла
       TDateTime  operator+();  // вернуть значение узла
       Boolean operator!=(T);// сравнение узлов
       Boolean operator<(T);// сравнение узлов
    };
```

Оператор присваивания структурных данных

Бинарная операция имеет два операнда структурного типа для присваивания вида: <T> = <T>. Первый операнд *this – эта структура. Второй операнд Val передается через параметр. В теле операции помещены операции присваивания полям «этой» структуры полей структуры Val. Код приведен в листинге 9.31.

Листинг 9.31. Описание оператора присваивания вида: <T> = <T>.

```
T T::operator = (T Val)
{
      // присваивание полей
    FNumber = Val.FNumber;
    ForEnter = Val.ForEnter;
    ForSelect  = Val.ForSelect;
    ForKasse  = Val.ForKasse;
    return *this;
}
```

Оператор присваивания, введенный для задания исходных данных покупателя

Бинарная операция имеет два операнда для выполнения операции присваивания вида: <T> = <int>. Отличие от первого оператора определяется другим его назначением – он используется для задания исходных данных покупателя. Первый операнд *this – указатель на структуру, поля которой формируются. Второй операнд Val определяет номер клиента, информационные данные для которого нужно подготовить. В процедуре используется стандартная функция EncodeTime.

Функция EncodeTime

Функция EncodeTime имеет описание прототипа:

```
TDateTime EncodeTime (Word Hour, Word Min, Word Sec,
                                                Word Sec);
```

Эта функция возвращает значение даты и времени типа TDateTime, сформированное из отдельных значений часов Hour, минут Min, секунд Sec, миллисекунд MSec.

В соответствии с предложенной моделью, временные характеристики покупателей случайны и ограничиваются только временным диапазоном: покупатели мо-

гут посетить магазин в отведенный период времени длительностью в 5 часов; время для выбора покупки – 20 минут. Поэтому количество часов `Hour` и минут `Min` генерируются датчиком случайных чисел в диапазонах, определенных в рамках описанной модели, а `Sec`, `MSec` не учитываются. Операция используется для задания/генерации полей клиента с номером `Val`. Описание операции приведено в листинге 9.32.

Листинг 9.32. Описание оператора присваивания

```
T T::operator=(int Val)
// задать параметры покупателя с номером Val
{
     // Val изменить на неповторяющийся номер
  FNumber = Val; //random(Val); // присвоить номер
     // сгенерировать время захода в магазин
  ForEnter = EncodeTime(random(5),random(60),0,0);
      // сгенерировать время на выбор товара
  ForSelect = EncodeTime(0,random(20),0,0);
      // определить время для подхода к кассе
  ForKasse = ForEnter + ForSelect;
  return *this;
}
```

Унарная операция «+» для структуры T

Введение этой операции изменяет, в том числе, и стандартный смысл этой операции. Ее назначение – вернуть время подхода к кассе. Описание операторной функции приведено в листинге 9.33.

Листинг 9.33. Описание оператора «+»

```
TDateTime  T::operator+()  // вернуть значение узла
{
  return  ForKasse;
}
```

Операция «!=» для структуры T

Имеет два операнда. Первый операнд `*this` – указатель на структуру. Второй структурный операнд `Val` передается через параметр. Сравниваются время подхода к кассе этого покупателя `*this` и покупателя, определенного структурой `Val`. Если значения не равны, то возвращается `true`, иначе – `false`. Описание операторной функции приведено в листинге 9.34.

Листинг 9.34. Описание оператора «!=»

```
Boolean T::operator!=(T Val) // сравнение узлов
{

  if(ForKasse != Val.ForKasse) return true;
  return false;
}
```

Операция « < » для структуры T

Имеет два операнда. Первый операнд `*this` – указатель на структуру. Второй структурный операнд `Val` передается через параметр. Сравниваются время подхода к кассе операнда `*this` и операнда `Val`. Если значение первого меньше второго, то

возвращается `true`, иначе – `false`. Описание операторной функции приведено в листинге 9.35.

Листинг 9.35. Описание оператора «<»

```
Boolean T::operator<(T Val) // сравнение узлов
{
  if(ForKasse< Val.ForKasse) return true;
  return false;
}
```

Операция приведения типа для структуры `T`

Операция включена в класс `FList` для того, чтобы использовать поля структуры в операциях с классовыми типами. Описание операторной функции приведено в листинге 9.36.

Листинг 9.36. Описание оператора приведения типа «T»

```
FList::operator T()
{
  T M = Top->x;
  return M;
}
```

Примеры описания процедур, рекомендуемых для решения задачи

Какой бы интерфейс не был предложен для разработки программы, программист неизбежно столкнется с необходимостью *отобразить список покупателей магазина* с указанием их характеристик. И, конечно, возникнет необходимость в создании *списка покупателей (например, в обработчике события* под кнопку «Создать список покупателей»*)*.

Отображать список данных о покупателях лучше всего в сетке `StringGrid`. Для решения такой задачи можно разработать две процедуры.

1. Одна процедура (назовем ее `PutInTopGrid`) будет создана для отображения «шапки» таблицы – подписи к колонкам таблицы покупателей. Эту процедуру можно расположить вне класса.
2. Вторая процедура (назовем ее `List_s`) будет предназначена для «распечатки» списка покупателей в сетку `StringGrid`. Эту процедуру можно включить в методы класса, разрабатываемого для решения задачи (например, класса `FListTask`).

Предположим, что внешний вид таблицы с отображением списка покупателей будет таким, как это показано на рис. 9.4.

Вывод «шапки» таблицы (сетки)

Процедура `PutInTopGrid` описывается вне класса `FListTask` и предназначена для вывода шапки, то есть заголовков таблицы. В качестве параметра она принимает имя сетки `FGrid`, в которую следует поместить список покупателей. Реализация процедуры не представляет алгоритмической сложности. Описание процедуры приведено в листинге 9.37.

Листинг 9.37. Вывод заголовков полей таблицы в фиксированную область сетки `StringGrid`

```
void PutInTopGrid(TStringGrid* FGrid)
{
  FGrid->Cells[0][0] = "Номер";
  FGrid->Cells[1][0] = "Приход в магазин";
  FGrid->Cells[2][0] = "Время выбора";
  FGrid->Cells[3][0] = "Подход к кассе";
}
```



Номер	Приход в магазин	Время выбора	Подход к кассе
9	4:39:00	0:05:00	4:44:00
8	2:43:00	0:08:00	2:51:00
7	2:31:00	0:17:00	2:48:00
6	0:46:00	0:01:00	0:47:00
5	3:39:00	0:18:00	3:57:00
4	2:40:00	0:03:00	2:43:00
3	1:21:00	0:04:00	1:25:00
2	4:56:00	0:06:00	5:02:00

Рис. 9.4. Внешний вид таблицы покупателей, представленной с помощью процедур `PutInTopGrid` и `List_s`

Вывод списка покупателей в сетку

Метод `List_stek` включен в класс `FListTask` для вывода созданного списка в сетку `FGrid`. В программе используется адрес вершины списка, взятый из поля базового класса `Top`. Это можно делать, так как метод располагается в классе – потомке от класса «Список». Алгоритм метода прост: в цикле (пока есть элементы в списке) читаются из узлов значения параметров (полей структуры `T`), описывающих покупателей, и переносятся в сетку. Описание метода приведено в листинге 9.38.

Листинг 9.38. Вывод списка в сетку `StringGrid`

```
void FListTask::List_s(TStringGrid* FGrid)
{
  Knot* Work; // рабочая переменная
  Work = Top; // встать на вершину
  int i=1; // инициализация номера строки в сетке
  while (Work != NULL) // пока в списке есть узлы
     {
           // скопировать значения в сетку
       FGrid->Cells[0][i] = IntToStr(Work->x.FNumber);
       FGrid->Cells[1][i] = TimeToStr(Work->x.ForEnter);
       FGrid->Cells[2][i] = TimeToStr(Work->x.ForSelect);
       FGrid->Cells[3][i] = TimeToStr(Work->x.ForKasse);
       i++; // увеличить номер строки
       Work = Work->next;// перейти к следующему узлу
     };
}
```

Примечание

Очевидно, что для работы со списками возникает необходимость в *использовании классовых* глобальных переменных. Предположим, что имя классовой переменной, которая используется для создания списка покупателей, – `Work`:

```
FListTask* Work;
```

Такая переменная должна быть полем класса форма или глобальной переменной в модуле.

Обработчик события `OnClick` на кнопку «Создать список покупателей»

В алгоритме обработчика всего три операции: создать экземпляр класса; создать список; поместить список покупателей в сетку `StringGrid`. Текст обработчика события приведен в листинге 9.39.

Листинг 9.39. Описание обработчика события «Создать список покупателей»

```
void __fastcall TForm2::Button1Click(TObject *Sender)
{
      // создать экземпляр класса с инициализацией поля
      // «количество элементов в списке» (10 покупателей)
  Work = new FListTask(10);
  Work->DoList();// сгенерировать список из 10 покупателей
      // отобразить список в сетку StringGrid1
  StringGrid1->RowCount = Work->GetQuantity()+1;
  Work->List_stek(StringGrid1);
}
```



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. Разработать и реализовать алгоритм сравнения полиномов одной переменной с целочисленными коэффициентами.
2. Вычесть два полинома.
3. Получить производную от заданного полинома.
4. Переписать элементы списка в обратном порядке.
5. Соединить две очереди в одну.
6. Сравнить две очереди между собой.
7. В стеке есть два одинаковых элемента. Найти их номера.
8. Элементы очереди расположены по возрастанию за исключением одного элемента. Найти его и поставить на место.
9. Из списка удалить наибольший элемент.
10. Проверить, имеются ли лица, фигурирующие в нескольких списках (например, в списке покупателей иномарок; в списке участников круиза; в списке нарушителей закона).
11. Реализовать как вариант моделирования следующее.

В список «Зал самообслуживания» элементы попадают по таймеру (очередной покупатель входит в интервале времени *In1-In2*). Ему приписывается *время выхода* = *время входа* + *длительность выбора товара* (*Se1-Se2*). Если время выхода равно текущему, то покупатель попадает в очередь. Из очереди пользователь выходит после того, как он пробудет на кассе отведенное время обслуживания (*Ka1-Ka2*). (Возможные ограничения: количество покупателей в зале; длина очереди в кассе; время работы магазина.)

Запись алгебраического выражения в виде обратной польской нотации 10

Для представления арифметических и логических выражений часто используется польская запись [9], которая просто и точно указывает порядок выполнения операций. Кроме того, она не требует скобок. В этой записи, впервые примененной польским логиком Яном Лукашевичем, операторы следуют непосредственно за операндами. Поэтому ее иногда называют суффиксной или постфиксной записью. Например, выражение $A*(B+C/D)$ представляется как $ABCD/+*$.

Есть правила, касающиеся польской записи, которые помогают это делать вручную:

а) идентификаторы в польской записи следуют в том же порядке, что и в инфиксной записи;
б) операторы в польской записи следуют в том порядке, в каком они должны вычисляться (слева направо);
в) операторы располагаются непосредственно за своими операндами.

В данной главе показывается программная реализация записи выражения в виде обратной польской нотации, которая основывается на использовании списков, организованных по типу стеков. Исходное выражение для последующего представления в виде обратной нотации изначально помещается в строку символов. Строку следует дополнить еще одним вспомогательным символом – операцией/знаком «=». Эта операция будет иметь самый низкий приоритет и будет использоваться в алгоритме для завершения процесса построения обратной нотации.

Алгоритм использует два стека, которые мы назовем *X* и *Y*. В стеке *X* формируется обратная запись выражения. А стек *Y* играет вспомогательную роль. Формирование выражения в стеке *X* происходит по следующей схеме.

1. Из строки, содержащей выражение, читается символ за символом слева направо. Операнды складываются в стек *X*, левые скобки и операции – в стек *Y*.
   1.1. Если встречается правая скобка, то чтение выражения приостанавливается, а вместо этого в стеке *Y* отыскивается соответствующая ей левая скобка. При поиске левой скобки все символы, расположенные в стеке *Y* до левой скобки (сверху), выталкиваются из стека *Y* и вталкиваются в стек *X*.
   1.2. Не помещается в стек *Y* очередная операция, если только нижележащая операция не имеет более низкий приоритет или не оказывается левой скобкой. В таком случае чтение выражения приостанавливается. Вместо этого символы из *Y* выталкиваются в *X* до тех пор, пока не встретится левая скобка или дно стека.
2. Символы из стека *X* переписываются в отдельную строку – это результат.

Для реализации описанного алгоритма разрабатывается отдельный класс. Этот класс подобен классу «Стек» из главы 9.2, но адаптирован под сформулированную задачу с учетом ее особенностей.

В качестве значений стеков *X* и *Y* выступают символы. Поэтому узел стека может быть представлен структурой, описание которой представлено в листинге 10.1.

Листинг 10.1. Структура узла

```
struct FRec
```

```
{
   char Data;  // символ: операнд/операция
   FRec* next; // адрес следующей вершины/узла
};
```

В соответствии с логикой описанного алгоритма, основными операциями в нем являются: поместить символ в стек; взять символ с вершины стека. Причем операция «поместить символ в стек» используется в двух вариантах: поместить в стек *X*; поместить в стек *Y*. Поэтому имя стека должно быть включено в список параметров этой операции.

Так как поведение алгоритма определяется, в том числе, приоритетами операций, необходимо включить метод для задания приоритета.

Примечание

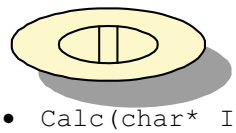
Самый низкий приоритет назначается операции «=». Эта операцию *необходимо добавить* к концу строки, содержащей исходное выражение, из которого формируется обратная нотация.

И, конечно, в классе должен быть метод, который будет из стека *X* извлекать символы для получения обратной нотации выражения в виде строки.

Назовем класс для формирования польской обратной нотации именем `Calc`. В поля класса включена всего одна переменная `Str`, она представляет собой строку (массив символов) фиксированного размера из 80 символов. При необходимости этот размер можно изменить. Интерфейс класса `Calc`, предназначенного для решения задачи, представлен в листинге 10.2.

Листинг 10.2. Интерфейс класса `Calc`

```
class  Calc
{
  char Str[80];  // строка, содержащая исходное выражение
   public:
    Calc(char*); // конструктор
    ~Calc(){}; // деструктор
    void PushTail(FRec* &Ptr, char c); // в стек
    char PopTail(FRec* &Ptr); // взять из стека Ptr
    int Prec(char c); // задать операции приоритет
    void Push( FRec* &Ptr, char c); // создать узел
    char Pop(FRec* &Ptr); // взять из стека Ptr
    void DoCalc();// вычислить
    AnsiString DoNext(); // вернуть польскую нотацию
};
```



НАЗНАЧЕНИЕ МЕТОДОВ КЛАССА `Stroka`:

- `Calc(char* Inits);` – конструктор с одним параметром `Inits`. В соответствии со значением `Inits` инициализируются строка класса;
- `~Calc(){};` – деструктор;
- `void PushTail(FRec* &Ptr,char c);` – метод класса с назначением: поместить в стек `Ptr` символ `c`;
- `char PopTail(FRec* &Ptr);` – метод класса с назначением: вернуть значение элемента, расположенного в вершине стека по адресу `Ptr`;

- `int Prec(char c);` – назначить операции со значением, равным `c`, номер в соответствием с ее приоритетом;
- `void Push( FRec* &Ptr, char c);` – создать новый узел с адресом `Ptr` и поместить в него символ `c`;
- `char Pop(FRec* &Ptr);` – вернуть значение элемента, расположенного по адресу `Ptr`;
- `AnsiString DoNext();` – вернуть строку/обратную нотацию. Этот метод является основным. В нем реализован описанный алгоритм решения задачи.

Конструктор класса `Calc`

В конструкторе выполняется всего одна операция – скопировать значение параметра в поле `Str` класса, то есть инициализировать поле класса. Описание конструктора приведено в листинге 10.1.

Листинг 10.1. Описание конструктора класса `Calc`

```
Calc::Calc(char* Init)
{
  strcpy(Str,Init);
}
```

Добавление элемента к стеку

Метод класса имеет два параметра. Один параметр – `Ptr` – определяет имя стека, а второй – `c` – определяет значение, которое нужно поместить в стек. Указатель стека `Ptr` передается «по ссылке». Такая необходимость обусловлена тем, что создание нового узла, предназначенного для хранения значения `c`, производится внутри метода, и, следовательно, адрес `Ptr` является не только входным, но и выходным. Описание метода в листинге 10.2.

Листинг 10.2. Добавление символа к вершине стека

```
void Calc:: PushTail(FRec* &Ptr, char c)
  // поместить (добавить) символ с в стек
{
  if (Ptr == NULL)      // если стек был пуст
    {
       Ptr = new FRec; // отвести память под узел
       Ptr -> Data = c;// записать символ
       Ptr->next = NULL;
    }
  else    // иначе добавить к вершине непустого стека
    Push(Ptr->next, c);
}
```

Извлечение элемента из стека

Метод класса имеет один параметр `Ptr`, который определяет адрес вершины стека (имя стека). Указатель стека `Ptr` передается «по ссылке». Такая необходимость обусловлена тем, что при «чтении» узла, расположенного в вершине стека, помимо того, что его значение возвращается через заголовок метода, производится также удаление прочитанного узла. И, как следствие этого, изменяется адрес вершины стека `Ptr`, то есть указатель является как входным, так и выходным. Описание метода в листинге 10.3.

Листинг 10.3. Извлечение элемента из стека

```
char Calc::PopTail(FRec* &Ptr)
// взять элемент с вершины стека Ptr
{
  char P = '0';
  if (Ptr != NULL)   // если не дно, т.е. стек не пустой
     {
           // если за Ptr больше нет элементов
       if (Ptr-> next == NULL)
          P = Pop(Ptr);
            // если за Ptr есть еще элемент
       else
          P = PopTail(Ptr->next);
     }
    return P;
}
```

Установление ранга для операции

Метод класса имеет один входной параметр – символ `c`, который сообщает знак операции. В методе этой операции присваивается приоритетный номер, значение которого возвращается через заголовок. Описание метода в листинге 10.4.

Листинг 10.4. Установление ранга для операции

```
int Calc::Prec(char c)
// присвоить операции приоритет, определяющий порядок
{
  switch (c)
    {
      case '*' : return (3); // самый высокий приоритет
      case '/' : return (3); // самый высокий приоритет
      case '+':
      case '-': return (2);
      case '(': return (1);
      case '=': return (0); // самый низкий приоритет
    }
}
```

Добавление элемента к стеку

Метод класса имеет два параметра. Один параметр – `Ptr` – определяет имя стека, а второй – `c` – определяет значение, которое нужно поместить в стек. Указатель стека `Ptr` передается «по ссылке». Такая необходимость обусловлена тем, что формирование нового узла, предназначенного для хранения значения `c`, производится внутри метода, и, следовательно, адрес `Ptr` является не только входным, но и выходным. Описание метода в листинге 10.5.

Листинг 10.5. Добавление элемента к стеку

```
void Calc:: Push( FRec* &Ptr, char c) // добавить узел
// с адресом Ptr к стеку и поместить в него символ c
{
  FRec* P;  // вспомогательная переменная
  P = new FRec;  // отвести память
  P->Data = c;  // поместить символ c
  P->next = Ptr; // определить адресное поле
  Ptr = P; // определить значение Ptr
}
```

Извлечение элемента из стека

Метод класса имеет один параметр `Ptr`, который определяет адрес вершины стека (имя стека). Указатель стека `Ptr` передается «по ссылке». Такая необходимость обусловлена тем, что при «чтении» узла, расположенного в вершине стека, помимо того, что его значение возвращается через заголовок метода, производится также удаление прочитанного узла. И, как следствие этого, изменяется/поправляется адрес `Ptr` – вершины стека, то есть указатель является как входным, так и выходным. Описание метода – в листинге 10.6.

Листинг 10.6. Извлечение элемента из стека

```
char Calc::Pop(FRec* &Ptr) // вытолкнуть элемент из стека Ptr
{
  char P=0;
  if (Ptr != NULL)
    {
      P = Ptr->Data;
      Ptr = Ptr -> next;
    }
  return P; // вернуть значение
}
```

Запись обратной нотации в строку

В алгоритме используются две вспомогательные переменные `X` и `Y`. Они обозначают стеки, описанные в схеме решения задачи. А также переменная `Qtr` – для хранения обратной нотации. Алгоритм сводится к следующим шагам:

1) инициализируются вершины стеков;
2) выполняется цикл (пока есть символы в исходной строке):
   а) если символ – буква, то она/он помещается в стек `X`;
   б) если символ – левая круглая скобка, то она помещается в стек `Y`;
   в) если символ – правая круглая скобка, то выполняется цикл (пока не встретится левая скобка):
      - взять символ из стека `Y`;
      - поместить символ в стек `X`;
   г) если символ – знак операции (не буква и не круглая скобка), то выполняется цикл до тех пор, пока `Flag` – *Истина*:
      - инициализировать переменную `Flag`, которая будет прекращать выполнение цикла;
        - ♦ если стек `Y` не пуст и ранг анализируемой операции меньше, чем ранг операции, расположенной в вершине стека `Y`, то:
          - из *Y* элементы выталкиваются в стек `X`;
          - значение `Flag` устанавливается в значение *Ложь*;
      - занести знак операции в стек `Y`;
3) инициализировать индекс строки `Qtr`;
4) делать цикл (пока не пуст стек `X`):
   а) переписать символ из стека *X* в строку `Qtr`;
   б) изменить индекс строки `Qtr` на 1;
5) добавить к строке `Qtr` символ `'\0'` – конец строки;
6) вернуть результат.

Описание метода – в листинге 10.7.

Листинг 10.7. Формирование из выражения обратной польской нотации

```
AnsiString Calc:: DoNext()
// построить и вернуть польскую нотацию
{
  char Qtr[80]; // вспомогательная переменная
  FRec * X,  // стек X
       *Y;  // стек Y
      // инициализация стеков
  X = NULL;
  Y = NULL;
// пока строка - выражение не пуста (пока есть символы)
  for (int i = 0; Str[i] != '\0'; i++)
   {
         char Ch = Str[i]; // очередной символ
                   // если очередной символ - операнд
         if ( 'A' <= Ch && Ch <= 'Z')
                        // то символ помещается в стек X
                    Push (X,Ch);
                   // если очередной символ – круглая скобка '('
         if (Ch == '(' )
                   // то символ помещается в стек Y
           Push(Y,Ch);
                   // если очередной символ – круглая скобка ')'
         if (Ch == ')' )
            {
            // пока в Y не встретится левая '('
              while (Y->Data != '(')
                   // из Y элементы выталкиваются в стек X
                Push(X,Pop(Y));

                   //(читается верхняя операция из Y)
             Ch = Pop(Y);

            }
           Boolean Flag;
                  // анализ верхнего символа на знак операции
           switch (Ch)
           {
             case '+':
             case '-':
             case '*':
             case '/':
             case '=':
               {
                  do {
                       Flag = true;
                       if (Y != NULL)
                        // если ранг меньше, чем верхний в Y
                          if (Prec(Ch) < Prec(Y->Data) )
                            {
                        // из Y элементы выталкиваются в стек X
                               Push(X, Pop(Y));
                               Flag = false;
                            }
                    }while (!Flag);
              Push(Y,Ch);
```

```
            }
        }
      }
    int j = 0;
    while(X != NULL)Qtr[j++] = PopTail(X);
    Qtr[j]='\0';
    return ( (AnsiString )Qtr);
}
```



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. В класс `Calc` включить метод для вычисления значения алгебраического выражения, записанного в виде обратной польской нотации.

Рекомендация: ввести таблицу символов для хранения данных.

2. Расширить класс `Calc`, дополнив его возможностями для выполнения операторов вида:

   goto A

   if < *выр* > *then* < *инстр*$_1$ > *else* < *инстр*$_2$ >

Рекомендация: следует пополнить польскую запись новыми операторами. Метка `A` в операторе безусловного перехода должна быть представлена адресом соответствующего элемента из таблицы символов. Для условного оператора результат вычисления <*выр*>, равный 0, означает «истину», а не равный 0, означает «ложь». Если результат «истина» – то безусловный переход на <*инстр*$_1$>, иначе – на <*инстр*$_2$>.

3. В запись алгебраического выражения входит переменная `A`. Требуется получить таблицу значений этого алгебраического выражения в зависимости от `A`, если известно, что `A` меняется на заданном интервале с заданным шагом.
4. Заданы два алгебраических выражения, операнды в которых представлены одинаковым списком переменных. Требуется проверить эти выражения на совпадение результатов при заданных наборах исходных данных.

Класс «Двоичное дерево» 11

Один из фундаментальных вопросов программирования – это вопрос о представлении сложных объектов, а также вопрос об эффективной реализации операций над подобными объектами. В данной главе рассматриваются такие часто используемые структуры, которые принадлежат большому семейству «Деревья». Речь будет идти о бинарных или двоичных деревьях.

Такой динамической структурой данных, организованной в виде двоичного или бинарного дерева, особенно удобно пользоваться, когда поиск и включение элементов в списки данных могут происходить одинаково часто. Двоичное дерево схематически можно изобразить так, как это показано на рис. 11.1.

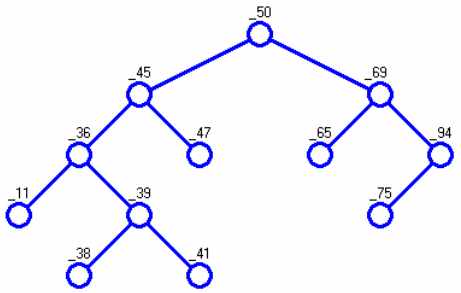


Рис. 11.1. Бинарное двоичное дерево с узлами, содержащими одно целочисленное информационное поле

Дерево представлено набором вершин, соединенных стрелками. Из каждой вершины исходит не более двух стрелок. Отсюда и название дерева – *бинарное*. Можно построить *тернарное* дерево (из каждой вершины исходит три стрелки) и дерево с *произвольным числом дочерних узлов*, выходящих из каждой вершины.

Верхняя вершина, в которую не входит никакая стрелка, называется *корнем* дерева. В остальные вершины входит по одной стрелке. Вершины дерева, не имеющие преемников, называются *листьями*. Каждая некорневая вершина бинарного дерева, в свою очередь, может служить *корнем для поддерева* – преемника.

Каждый элемент/узел, описывается некоторой структурой. В полях этой структуры в бинарном дереве, помимо информационной части (значение узла), должны быть предусмотрены две ссылки. Одна ссылка указывает *на левое поддерево*, другая – *на правое поддерево*. Эти ссылки задают «сцепления» по левой и правой ветвям, выходящим из этой вершины. Примеры бинарных деревьев: генеалогическое дерево человека, отображение истории теннисного турнира.

ДЕРЕВО ПОИСКА

Следует обратить внимание, что в дереве, изображенном на рис. 11.1, вершины связываются так, что для каждой вершины левый нисходящий узел имеет значение меньшее, а правый нисходящий – большее, нежели значение информационного поля предка. Это упорядоченное дерево. Его называют *деревом поиска.* Дерево поиска рекомендуется строить при решении задач, связанных с поиском определенных значений, связанных условием.

ИДЕАЛЬНО СБАЛАНСИРОВАННОЕ ДЕРЕВО

Для того чтобы достичь минимальной высоты бинарного дерева (не дерева поиска) при заданном количестве узлов, нужно располагать максимально возможное число узлов на всех уровнях, кроме самого нижнего. Это можно сделать очень просто, если распределять все поступающие узлы поровну: слева и справа от каждого узла. В результате получится дерево, которое называют *идеально сбалансированным*.

Возможные варианты двоичных идеально сбалансированных деревьев с числами узлов k, где $k = 4, 5, 6, 7$, представлены на рис. 11.2.

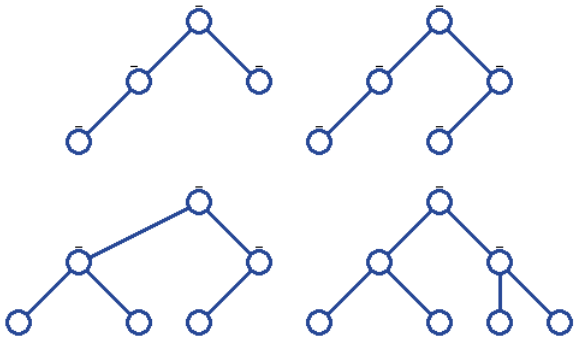


Рис. 11.2. Виды двоичных идеально сбалансированных деревьев с числами узлов k, где k = 4, 5, 6, 7

11.1. Разработка классов «Бинарное, или двоичное, дерево»

Общая постановка задачи

Базовая задача. Требуется разработать семейство классов «Бинарное дерево» со следующим набором операций [6]:

- создать бинарное сбалансированное дерево;
- создать бинарное упорядоченное дерево (или дерево поиска);
- включить в дерево поиска новый элемент;
- удалить элемент из дерева;
- просмотреть узлы дерева.

Стратегия решения

Стратегия решения укладывается в стандартную стратегию, применяемую при разработке любого класса: выделяются основные *математические объекты* и определяются их характеристики и операции, задающие методы поведения. Все это распределяется по классам. Сначала прорисовываются и продумываются *интерфейсы классов*, и только после этого *формализуются методы* классов. Подготовленная система классов оформляется в виде отдельного модуля проекта. После чего для решения задач создается *интерфейс приложения* и обеспечивается его *функциональность*. При решении задач используются алгоритмы, разработанные Н.Виртом [6].

Примечание

Практически все процедуры, связанные с обработкой деревьев, являются *рекурсивными*!

Решение

В рамках поставленной задачи фундаментом для разрабатываемого семейства классов является математический объект – бинарное дерево. Бинарное дерево характеризуется тремя признаками: *структурой отдельного узла*, указателем на *корень дерева* и *количеством* таких узлов. Именно эти признаки будут определять поля базового класса, который будет называться в данной задаче `Tree`. В число методов базового класса следует включить следующие компоненты: конструктор – для создания пустого дерева; деструктор; а также методы для получения значений, характеризующих поля класса.

В потомок первого уровня, класс `TreeEx`, помимо конструкторов и деструкторов, включаются методы, перечисленные в рамках поставленной задачи. Часть этих методов в процессе своей работы используют, так называемый, *барьерный узел*, поэтому является целесообразным к полям класса добавить новое поле – «барьер» с типом структуры узла. Для того чтобы сделать процедуры создания деревьев более универсальными и не зависящими от способа получения информации, необходимой для заполнения информационных полей узлов, принято решение брать информацию из файлов данных (входных файлов), а результаты просмотра деревьев помещать в выходной файл. С такой целью в поля класса включаются *две файловые переменные* с типами `ifstream` и `ofstream`, соответственно. Кроме этого, в поля введена переменная типа `AnsiString`. Эта *переменная используется для передачи из методов текстовых* комментариев или иной текстовой информации. Использование этих дополнительных полей класса требует введения соответствующего конструктора для их инициализации.

В третий класс помещаются методы, связанные, в том числе, с обходом дерева, а также вводом и выводом значений.

Предположим, что дерево состоит из узлов с простейшей структурой информационного поля, как это показано в листинге 11.1.

Листинг 11.1. Описание узла бинарного дерева

```
struct Ref    // Узел дерева
{
  int x; // информационное поле
  Ref * Left, * Right; // ссылки на левое и правое поддерево
};
```

Описание базового класса `Tree`

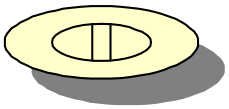
В свете вышеизложенного интерфейс базового класса может быть таким, как в листинге 11.2.

Листинг 11.2. Интерфейс базового класса «Бинарное дерево»

```
class  Tree
{
  protected:
    Ref* Top;  // Указатель на вершину дерева
    int FCounter;  // Число узлов

  public:
   Tree(); // Конструктор. Создает пустое дерево
   ~Tree(); // Деструктор
```

```
   Ref* TakeTop(){return Top;} // Вернуть вершину
   int Quantity(); // Вернуть количество
};
```



МЕТОДЫ КЛАССА `Tree`:

- `Tree();` – конструктор без параметров. Устанавливает поля класса в «нулевые» значения;
- `~Tree();` – деструктор;
- `Ref* TakeTop();` – возвращает адрес корня дерева;
- `int Quantity();` – возвращает количество узлов дерева.

Конструктор без параметров класса `Tree`

Конструктор используется для создания пустого дерева. Код конструктора представлен в листинге 11.3.

Листинг 11.3. Конструктор без параметров

```
Tree :: Tree()
{
  FCounter = 0;
  Top = NULL;
}
```

Деструктор класса `Tree`

Деструктор используется для обнуления полей класса. Код деструктора представлен в листинге 11.4.

Листинг 11.4. Деструктор

```
Tree::~Tree()
{
  if (Top !=NULL) delete Top;
}
```

Вернуть число узлов в дереве

Метод `Quantity` класса `Tree` не имеет параметров. Он возвращает количество узлов дерева. Код метода представлен в листинге 11.5.

Листинг 11.5. Взять число узлов в дереве

```
int Tree::Quantity()
{
  return FCounter;
}
```

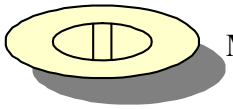
Описание расширенного класса `TreeEx`

В расширенный класс, потомок от класса `Tree`, включены новые поля: ссылочное поле `sTop` – для хранения барьерного значения (используется некоторыми методами); строковая переменная `Ptr` – в нее можно помещать информационный текст или использовать в иных целях; файловые переменные `Fin` и `Fout`, используемые для ввода/вывода данных в файл. Также в класс включены все методы, заявленные в условии задачи. Причем некоторые методы включены с разными вариан-

тами исполнения. Помимо этого добавлены методы, играющие роль инструментов для упрощения логики сложных алгоритмов. Интерфейс класса представлен в листинге 11.6.

Листинг 11.6. Интерфейс расширенного класса, потомка класса «Бинарное дерево»

```
class TreeEx: public Tree
{
   protected:
     Ref* sTop; // Барьер
     AnsiString Ptr;  // Для комментариев
     ifstream Fin;  // Входной файл данных
     ofstream Fout; // Выходной файл
   public:
      TreeEx(); // Пустой конструктор
                        // передать имена файлов
      TreeEx(char* NameFile, char* NameFileOut);
      ~TreeEx();
                        // Добавление узла к дереву поиска
      void Search(int value, Ref* &Work);
                        // Поиск узла  со значением NewX
      Ref* Location(int NewX, Ref*T);
                        // Поиск узла  NewX с use барьера
      Ref* LocWithHarrier(int NewX, Ref*T);
      // Поместить значение узла с адресом t в выходной файл
      void P(Ref* t);
      AnsiString GetText(){return Ptr;} // Взять текст
      void DoHarrier(){ sTop = new Ref;}// Создать барьер
      void DelRefFromTree(int NewX,Ref* &P);//
      void DelRef(Ref* &R, Ref* q);//
      // Удалить узел со значением NewX из дерева поиска
     // void Del(int NewX);
            // Построение сбалансированного дерева из n узлов
      Ref* Do_tree(int n);
            // Построение дерева поиска (упорядоченного дерева)
      void DoTree();
};
```



МЕТОДЫ КЛАССА `TreeEx`:

- `TreeEx();` – конструктор без параметров;
- `TreeEx(char* NameFile, char* NameFileOut);` – конструктор, имеющий два входных параметра. Эти параметры задают имена файлов: `NameFile` задает имя файла, из которого берутся числа для создания дерева; `NameFileOut` – имя файла, куда заносятся результаты просмотра дерева;
- `~TreeEx();` – деструктор класса;
- `void Search(int value, Ref* &Work);` – метод класса, предназначенный для добавления узла к дереву поиска: `value` – добавляемое значение, `Work` – адрес узла, где помещено значение `value`;
- `Ref* Location(int NewX, Ref*T);` – метод класса, предназначенный для поиска узла со значением `NewX` в информационном поле; `T` – адрес найденного узла;

- `Ref* LocWithHarrier(int NewX, Ref *T);` – метод класса, предназначенный для поиска узла со значением `NewX` в информационном поле; `T` – адрес найденного узла. Метод использует барьерный узел;
- `void P(Ref* t);` – метод, имеющий один входной параметр. Решает задачу: поместить значение узла с адресом `t` в выходной файл;
- `AnsiString GetText();` – метод, возвращающий текстовую строку, определенную в полях класса;
- `void DoHarrier();` – метод класса, создающий барьерный узел;
- `void DelRefFromTree(int NewX, Ref* &P);` – метод класса, предназначенный для удаления узла со значением `NewX`;
- `void DelRef(Ref* &R, Ref* q);` – метод класса, предназначенный для удаления узла с двумя потомками;
- `void Del(int NewX);` – функция удаления узла со значением `NewX` из дерева поиска;
- `Ref* Do_tree(int n);` – метод построения сбалансированного дерева из `n` узлов;
- `void DoTree();` – метод построения дерева поиска (упорядоченного дерева).

Следует также обратить внимание на функцию с именем `P`, включенную в методы класса. В этом методе будет содержаться только одна операция – операция записи значения узла в файл.

Конструктор класса `TreeEx`

Конструктор класса имеет два параметра: `NameFileIn` – имя файла с исходными данными; `NameFileOut` – имя файла с данными, являющимися результатами. Предку не передаются параметры. В теле конструктора инициализируются поля класса и открываются файлы для чтения и записи. Код конструктора представлен в листинге 11.7.

Листинг 11.7. Конструктор класса `TreeEx`

```
TreeEx::TreeEx(char* NameFileIn, char* NameFileOut):Tree()
{
  sTop = NULL; // для барьера
  Ptr = ""; // для комментариев
  Fin.open(NameFileIn,ios::in); // для чтения
  Fout.open(NameFileOut,ios::out);// для записи
}
```

Деструктор класса `TreeEx`

В теле деструктора закрываются открытые файлы. Код деструктора представлен в листинге 11.8.

Листинг 11.8. Деструктор расширенного класса

```
TreeEx::~TreeEx()
{
  Fin.close();
  Fout.close();
}
```

Построение сбалансированного дерева

Рекурсивная функция `Do_tree`, предназначенная для построения сбалансированного дерева, имеет один параметр `n`. Этот параметр определяет количество чисел/узлов, которые нужно прочитать из файла `ifstream` с целью размещения их в узлы дерева. Файл с данными должен быть предварительно сформирован.

Каждая отдельная активация (рекурсивный вызов) `Do_tree` должна содержать обязательный набор действий: прочитать из файла очередное число, отвести место под узел/структуру, сформировать поля этой структуры, вернуть ссылку на узел через заголовок. Этих активаций будет столько, сколько узлов в дереве, то есть `n` штук.

Кроме этого, взятые из файла значения по условию задачи должны равномерно распределяться по узлам дерева. Потому в тело функции следует ввести элементы проверок и пересчета узлов. И тогда список действий несколько расширяется и его можно описать следующей схемой:

- проверить значение входного параметра `n`. Если узлов больше нет (`n=0`), то вернуть адрес `NULL`;
- если узлы есть (`n > 0`), то один узел взять в качестве очередного корня дерева (оставшиеся (`n - 1`) узлы далее следует разместить поровну в левом и правом поддеревьях);
- отвести место под узел/структуру;
- прочитать очередное значение из файла;
- так как самый первый узел по логике алгоритма должен стать корнем/вершиной всего дерева, здесь нужно предусмотреть проверку. В случае если прочитанный из файла элемент – первый по порядку, то в классовом поле `Top` запомнить его адрес;
- заполнить поля структуры созданного узла:
    - заполнить информационное поле созданного узла;
    - определить левую ссылку узла; эта ссылка определится в результате *рекурсивного* вызова этой же процедуры для построения левого поддерева с *nl = n/2* узлами; вычисленный адрес поместить в ссылочное (левое) поле;
    - определить правую ссылку узла; эта ссылка определится в результате *рекурсивного* вызова этой же процедуры для построения правого поддерева с *nr = n-nl-1* узлами; вычисленный адрес поместить в ссылочное (правое) поле;
- вернуть адрес созданного узла как результат работы процедуры.

Следует заметить, что получение адреса в качестве результата работы рекурсивной процедуры позволяет в алгоритме формировать ссылки узлов с помощью одного действия – оператора присваивания.

В листинге 11.10 представлено описание функции/метода `Do_tree` построения сбалансированного бинарного дерева.

Листинг 11.10. Построение сбалансированного дерева

```
Ref* TreeEx::Do_tree(int n)
// n – число узлов в дереве;
// value – вспомогательная переменная;
// nl – количество узлов, передаваемых в левое поддерево;
// nr - количество узлов, передаваемых в правое поддерево;
```

```
// Work – рабочая переменная для создания узла;
{
   Ref* Work;
   int value, nl, nr;
   if (n == 0) return NULL; // если узлов нет
   else // иначе, если узлы еще есть
     {
       nl = n/2;   // число узлов налево
       nr = n-nl-1;   // число узлов направо
       Fin>>value; // прочитать из файла очередное число
            // отвести место под узел и адрес поместить в Work
       Work = new Ref;
            // если это первый узел, то запомнить его в Top
       if (FCounter == 0) Top = Work;
        Work->x = value; // поместить значение в узел дерева
       FCounter++; // увеличить счетчик узлов на 1
            // Рекурсивное обращение
                        // построить левое поддерево
       Work->Left = Do_tree(nl);
                       // построить правое поддерево
       Work->Right = Do_tree(nr);
       return Work; // вернуть адрес узла
     }
}
```

Построение дерева поиска

Работу предлагаемого алгоритма построения дерева поиска продемонстрируем на следующем примере. Пусть в дерево, которое на начальном этапе будет пустым, требуется разместить следующие числа: 9, 2, 33, 44, 55, 6, 7, 1, 5 … Разумеется, что эти числа, в соответствии с обозначенной стратегией, должны предварительно располагаться в файле `ifstream`.

Первое число – 9 – помещаем в корень дерева. Значение следующего числа (это число 2) является меньшим, нежели число 9, стоящее в корне дерева, поэтому формируемая для элемента 2 вершина будет направлена в сторону левой ссылки от узла 9. Следующее число (33) по величине больше, чем 9, поэтому формируемая для элемента 33 вершина будет направлена в сторону правой ссылки от узла 9. Число 44 больше, чем 9, поэтому проверяется правая ссылка узла 9, а эта ссылка показывает на число 33. Следовательно, число 44 будет направлено в правую ссылку от числа 33. Аналогичная проверка производится для числа 55. Оно помещается по направлению правой ссылки от числа 44. Число 6 меньше, нежели корень дерева 9, поэтому проверяется левая ссылка, где записано число 2. Сравнение 6 и 2 показывает, что 6 должно быть направлено в правую ссылку от числа 2 и так далее. В результате подобных рассуждений дерево принимает вид, как на рис. 11.3.

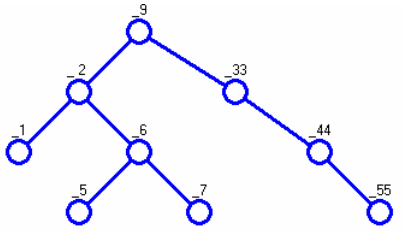


Рис.11.3. Иллюстрация формирования дерева поиска с узлами: 9, 2, 33, 44, 55, 6, 7, 1, 5 ...

В представленном к рассмотрению примере все числа, поступающие в дерево поиска, различные. А как быть, если среди чисел есть одинаковые?

Примечание

В настоящем случае в дерево поиска просто не включаются совпадающие значения. Если потребуется все же учитывать дублирующие элементы, то можно поступить следующим образом: в структуру узла включить еще одно поле – число повторений.

В основу процедуры `DoTree` положен один цикл (пока есть данные в файле), в котором повторяются всего две операции:

а) прочитать числовое значение из файла;
б) добавить к дереву поиска узел с прочитанным значением.

Вторая операция, в свою очередь, реализована с помощью метода класса `Search`. В листинге 11.11 представлено описание дерева поиска.

Листинг 11.11. Построение дерева поиска

```
void TreeEx::DoTree()
// Функция строит бинарное дерево поиска, значениями узлов
// которого  будут все числа, находящиеся в файле ifstream
// Work – рабочая переменная, хранит адрес очередного
// создаваемого узла; value – вспомогательная переменная;
{
   Ref* Work = NULL; // подготавливается рабочая переменная
   int value;
   while (!Fin.eof()) // пока не конец файла
     {
        Fin>>value;  // читается из файла значение
        Search(value, Work); // добавляется узел с value
        FCounter++;       // увеличивается число узлов на 1
     }
}
```

Добавление к дереву поиска нового узла

Функция `Search`, описание которой представлено в листинге 11.12, имеет два параметра. Один параметр – `value` – входной, через него передается значение для очередного узла дерева. Второй параметр – `Work` – является и входным, и выходным. Поэтому этот указатель `Work` передается по ссылке.

Примечание

Функция `Search` хороша тем, что ее можно использовать не только в процедуре создания дерева поиска, как это предполагается в данном случае, но и тогда, когда требуется добавить новый узел к ранее созданному дереву поиска. Если дерево создается, то на вход `Work` передается `NULL`. Если дерево уже построено, то на вход передается адрес вершины.

В результате работы через этот параметр возвращается адрес узла, куда функция `Search` поместила значение `value`. Входные значения параметров поступают из функции `DoTree`. Первоначальное значение `Work = NULL`, потому что на начальном этапе создания дерева оно пустое.

Функция Search построена таким образом, что в общем случае значение Work = NULL будет сигналом того, что в дереве поиска найдено место (левая или правая ссылка), по которому следует направить значение value. Тогда производится формирование узла: отводится память, заносится value и обнуляются исходящие ссылки, с целью подготовки для приема следующих значений. И, соответственно, уже ненулевой адрес Work возвращается в функцию DoTree. И его выходное значение (сигнал – узел занят) передается на вход процедуры при повторном обращении. Поэтому алгоритм всегда начинается с проверки того факта, свободно ли место для очередного значения или нужно спускаться дальше по дереву. Если Work = NULL, то место найдено, иначе его следует найти.

Поиск места связан с процедурой сравнения значения value со значением, расположенным в узле Work. В зависимости от результатов сравнения, производится спуск по дереву влево или вправо с помощью рекурсивного вызова функции Search, и продолжается до тех пор, пока не обнаружится пустая ссылка. Алгоритм функции можно описать следующими шагами:

- если найдено (Work = NULL) место для узла с value, то необходимо:
  - отвести место под узел;
  - адрес первого узла записать в Top (это будет корень/вершина всего дерева);
  - заполнить структуру узла:
    - заполнить информационное поле, записав в него value;
    - обнулить левую и правую ссылки;
- если не найдено место для value, то необходимо:
  - если value меньше значения узла с адресом Work, то рекурсивный «спуск» с вызовом Search – в левое поддерево;
  - если value больше значения информационного поля узла с адресом Work, то рекурсивный «спуск» с вызовом Search – в правое поддерево.

Описание метода представлено в листинге 11.12.

Листинг 11.12. Добавление узла к дереву поиска

```
void TreeEx::Search(int value, Ref* &Work)
// Метод Search включает узел со значением value в дерево
// поиска с сохранением порядка – по возрастанию.
// Work – адрес узла, предназначенного для хранения
// или размещения значения value.
{
   if(Work == NULL) // если найдено место
                    // для value в дереве поиска
     {
       Work = new Ref; // отвести память под узел
                       // и связать ее с Work
       if (FCounter == 0) Top = Work; // запомнить адрес
                                      // вершины/корня
       Work->x = value; // занести в узел значение value
       Work->Left = NULL;  // обнулить левую ссылку
       Work->Right = NULL; // обнулить правую ссылку
     }
    else // поиск места для value
      if (value < Work->x) Search(value, Work->Left);
      else
         if (value>Work->x) Search(value, Work->Right);
}
```

Поиск узла с заданным значением информационного поля

По условию задачи требуется найти адрес вершины узла с заданным значением информационного поля. Функция `Location`, предназначенная для поиска узла с заданным значением, имеет два входных параметра. Через первый параметр передается значение, по которому требуется найти узел, содержащий это значение. А второй параметр сообщает начальный адрес для поиска. Если поиск производится по всему дереву, то передается адрес его вершины. В результате своей работы функция `Location` возвращает адрес узла с заданным значением или `NULL`, если узла с таким значением в дереве нет.

Основу алгоритма составляет цикл, в котором производится переход от одного узла дерева к другому, начиная от его вершины, влево или вправо, в зависимости от результатов сравнения: заданное значение меньше – перемещение влево, больше – вправо до тех пор, пока не будет совпадения, или не будет установлен факт отсутствия узла с таким значением.

В алгоритм подпрограммы введена вспомогательная переменная `Found`, которая будет играть роль признака того факта, найден/не найден узел с заданным значением. Первоначальное значение `Found = false`. Если узел с указанным значением будет обнаружен, то `Found` примет значение `true`. В таком случае поиск следует прекратить – завершить цикл по поиску. Описание метода `Location` для поиска узла представлено в листинге 11.13.

Листинг 11.13. Описание метода `Location`, предназначенного для поиска узла в дереве поиска

```
Ref* TreeEx::Location (int NewX, Ref*T)
// NewX – искомое значение;
// T – адрес узла, первоначально его значение совпадает
// с адресом вершины, затем его значение заменяется адресом
// узла, следующего в связке (изменяется)
// Found – вспомогательная переменная - признак
{
      // на тот случай, что значения NewX не будет в дереве
   Boolean Found = false;
            // NewX не найдено и пока не конец дерева
   while (T != NULL && !Found)
    {
      if (T->x == NewX) // проверка на совпадение
          Found = true; // изменение Found, если узел найден
      else // иначе спуск
         if (T->x > NewX) T = T->Left; // влево
         else T = T->Right; // вправо
    }
  return T; // вернуть адрес
}
```

Поиск узла с заданным значением информационного поля с использованием барьерного узла

Логика метода организована так же, как в методе `Location`, но используется вспомогательный узел – барьер, в который помещается искомое значение `NewX`. Барьер обычно используется для того, чтобы упростить сложность условия окончания цикла поиска. В таком случае в конце списка помещается барьер и все терминальные узлы (с нулевыми ссылками) связываются с одним и тем же барьером. По-

лученная структура дерева – это не просто дерево, а дерево, все листья которого прикреплены внизу к одному якорю.

Функция `LocWithHarrier` возвращает адрес узла со значением `NewX` или `NULL`, если такого значения в дереве нет. Описание представлено в листинге 11.14.

Листинг 11.14. Описание функции `LocWithHarrier`, предназначенной для нахождения узла в дереве поиска с использованием барьера

```
Ref* TreeEx::LocWithHarrier (int NewX, Ref*T)
{
  sTop->x = NewX; // Задается поле узла - барьера
  while (T -> x!= NewX)
    {
      if (T->x > NewX) T = T-> Left;
      else T = T->Right;
    }
  Ptr = IntToStr(T->x);
  return T;
}
```

Удаление из дерева узла с заданным значением информационного поля

Задача удаления узла из дерева – самая сложная из задач, связанных с обработкой такой структуры данных. В общем случае исключение узла сводится к выполнению следующих двух действий.

1. Поиск узла с заданным значением информационного поля.
2. Исключение найденного узла.

При этом возможны три ситуации:

1. Из исключаемого звена не выходит ни одной ветки, то есть удаляется лист. Тогда исключение сводится к уничтожению ссылки на это звено.
2. Из исключаемого звена выходит только одна ветвь. Тогда удаление узла сводится просто к исправлению ссылки у узла – предшественника исключаемого узла. Ее новое значение будет определять адрес преемника исключаемого узла. Новая ветвь будет идти в ту же сторону, куда шла ветвь к исключаемой вершине так, как это показано на рис. 11.4.
3. Из исключаемого узла исходит две ветви. Если узел имеет двух потомков, то возникает определенная трудность, так как в удаляемую вершину входит одна ссылка, а выходит две. В этом случае нужно найти подходящее звено дерева, которое можно было бы вставить на место удаляемого, причем это подходящее звено должно легко перемещаться. Такое звено всегда существует: это либо *самый правый элемент левого поддерева*, либо *самый левый элемент правого поддерева*. Чтобы найти его, нужно идти по дереву от исключаемого узла один раз налево, а потом все время направо. А можно наоборот. И как только окажется, что идти некуда, то это звено будет годиться для подстановки в дерево.

Если выбран вариант – взять *самый правый элемент левого поддерева*, то необходимо перейти в следующую вершину по левой ветви; а затем переходить только по правой ветке до тех пор, пока очередная ссылка не будет `NULL`.

Во втором случае для достижения искомого звена, необходимо перейти в следующую вершину по правой ветви, а затем переходить только по левой ветке до тех пор, пока очередная ссылка не будет `NULL`.

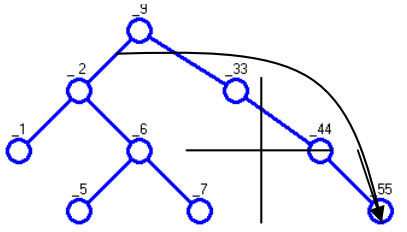


Рис. 11.4. Удаление узла с одной ссылкой

На рис. 11.5 рассматривается случай, когда требуется удалить узел со значением 36. В качестве подходящего звена, которое можно было бы вставить вместо удаляемого, берется число из узла со значением 35 – это самый правый элемент левого поддерева (он соответствует самому большому элементу слева). Удаление сводится к тому, что

а) находится адрес числа 35 (на замену);
б) число 36 меняется на число 35;
в) обнуляется правая ссылка у числа 33.

Вид дерева после удаления узла 36 показан на рис. 11.6. Очевидно, что при такой замене получается правильное дерево поиска. В нем у любой вершины все прямые и косвенные преемники по левой ветви имеют значения, меньшие по величине. А по правой – большие по величине.

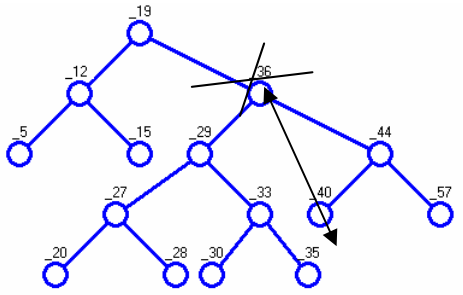


Рис. 11.5. Выбор значения-претендента на место удаляемого

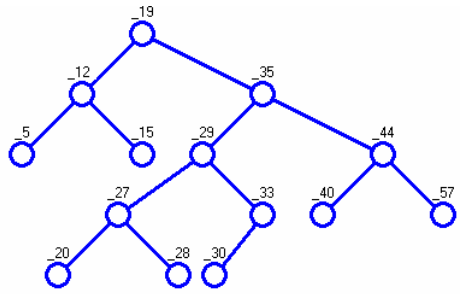


Рис. 11.6. Вид дерева после удаления числа 36

Удаление узла с заданным значением информационного поля реализовано с помощью рекурсивной функции `DelRefFromTree`. Эта функция имеет два параметра. Один параметр – `NewX` – входной. Через него передается значение для удаления его из дерева. Второй параметр `P` является как входным, так и выходным. Вход в процедуру происходит с заданным адресом начального узла, а в результате работы процедуры происходит спуск по дереву к следующему узлу – и заданный ранее адрес пе-

реопределяется на адрес нового значения – претендента на замену. При повторном вызове выходная ссылка становится входной и так далее.

Та часть алгоритма, которая касается поиска узла, уже представлена в описании функции `Location`. Возможны два выхода из описанной рекурсивной процедуры:

- если узла для удаления нет: выдается сообщение и нужно выйти;
- если узел найден: удалить найденный узел.

При удалении узла, в свою очередь, рассматриваются ситуации, когда

- узел не имеет потомков;
- имеет только одного потомка;
- имеет двух потомков.

Если узел не имеет потомков или имеет только одного потомка, то удаление узла реализуется просто: для этого достаточно откорректировать соответствующую ссылку у вершины предшественника, как это показано на рис. 11.4. Если удаляемый узел имеет двух потомков, то вызывается отдельная процедура удаления такого узла – `DelRef`. Описание функции `DelRefFromTree` представлено в листинге 11.15.

Листинг 11.15. Описание функции `DelRefFromTree`, предназначенной для поиска узла с заданным значением с целью удаления

```
void TreeEx::DelRefFromTree(int NewX, Ref* &P)
// NewX – значение удаляемого узла;
// P – адрес начального узла для поиска (переопределяется
// при спуске по дереву на адрес следующего узла)
// q - вспомогательная переменная;
{
  Ref* q;
  if (P == NULL) // если конец просмотра и узел не найден
                 // подготовить сообщение
     Ptr = "Узла со значением нет в дереве";
  else
            // определиться с направлением поиска
    if(NewX < P->x) // если значение меньше
// рекурсивный спуск влево с переопределенным начальным
// адресом
      DelRefFromTree( NewX,P->Left);
    else
      if(NewX > P->x) // если значение больше
// рекурсивный спуск вправо с переопределенным начальным
// адресом
        DelRefFromTree( NewX,P->Right);
      else  // если узел найден
                  // произвести удаление узла
        {
          q = P; // запомнить P
          if (q->Right == NULL) // если правый потомок - NULL
             P = q->Left; // исправить P
          else
            if(q->Left == NULL) // если левый потомок - NULL
               P = q->Right; // исправить P
            else DelRef(q->Left, q);// удалить узел
        }
}
```

Удаление из дерева узла, имеющего двух потомков

Для удаления из дерева узла с двумя потомками предназначена функция `DelRef`. Она вызывается из функции `DelRefFromTree` лишь в третьем случае процедуры исключения. Функция `DelRef` имеет два параметра – `R` и `q`. На входе оба параметра имеют одно и то же значение – адреса удаляемого узла. Первый параметр `R` является входным и выходным: при выходе ссылка изменяет свое значение – она показывает адрес узла со значением для замены.

В `DelRef` параметр `R` используется для «спуска» до самого правого звена левого поддерева удаляемого элемента `q` для того, чтобы найти элемент для замены. Затем заменяются поле `x` и ссылка в `q` на соответствующее поле и ссылку звена `R`. После этого звено, на которое указывает `R`, можно удалить (это делается оператором присваивания). Описание метода представлено в листинге 11.16.

Листинг 11.16. Удаление из дерева узлов с двумя потомками

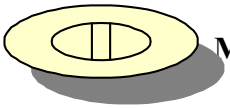
```
void TreeEx::DelRef (Ref* &R, Ref* q)
// запоминает
{
  if(R->Right !=NULL) // если правая ссылка узла R не NULL
       DelRef(R->Right,q); // спуск вправо вниз до конца
  else // если правая ссылка NULL (найдена ссылка для замены)
    {
      q->x = R->x; // изменение поля и ссылки
      q = R;
      R = R->Left;  // удаление звена
    }
}
```

Описание класса `TreeTask`

В расширенный класс `TreeTask` – потомок от класса `TreeEx` – включены все методы, связанные с вводом и выводом данных, а именно: с вводом/выводом в файлы. Интерфейс класса представлен в листинге 11.17.

Листинг 11.17. Интерфейс класса `TreeTask`

```
class TreeTask: public TreeEx
{
  public:
   TreeTask(); // Пустой конструктор
   TreeTask(char* NameFile, char* NameFileOut); // Конструктор
   ~TreeTask(); // Деструктор
            // Создание выходного файла
   void InFile(void(__closure *aMethod)(Ref*), Ref*);
            // Обход дерева сверху вниз
   void List_preorder(Ref* t);
            // Обход дерева снизу вверх
   void List_postorder(Ref* t);
             // Обход дерева слева направо
   void List_inorder(Ref* t);
};
```



МЕТОДЫ КЛАССА `TreeTask`:

- `TreeTask();` – конструктор без параметров;
- `TreeTask(char* NameFileIn, char* NameFileOut);` – конструктор, имеющий два параметра. Через параметр `NameFileIn` передается имя файла, из которого берутся значения для построения дерева. Через параметр `NameFileOut` передается имя файла, в котором сохраняются значения построенного дерева;
- `~TreeTask();` – деструктор;
- `void InFile(void(__closure *aMethod)(Ref*), Ref* aTop);` – создание выходного файла. Параметр `aMethod` – имя метода обхода дерева. Параметр `aTop` – вершина дерева;
- `void List_preorder(Ref* t);` – обход дерева сверху вниз. Параметр `t` – узел дерева;
- `void List_postorder(Ref* t);` – обход дерева снизу вверх. Параметр `t` – узел дерева;
- `void List_inorder(Ref* t);` – обход дерева слева направо. Параметр `t` – узел дерева.

Конструктор без параметров класса `TreeEx`

Предназначен для того, чтобы вызвать конструктор предка. Описание этого конструктора класса представлено в листинге 11.18.

Листинг 11.18. Описание конструктора без параметров

```
TreeTask::TreeTask():TreeEx()
{
}
```

Деструктор класса `TreeEx`

Предназначен для того, чтобы вызвать деструктор предка. Описание деструктора класса представлено в листинге 11.19.

Листинг 11.19. Описание деструктора класса `TreeEx`

```
TreeTask::~TreeTask()
{
}
```

Конструктор с параметрами

Предназначен для того, чтобы вызвать соответствующий конструктор предка, которому передаются имена входного и выходного файла. Описание этого конструктора класса представлено в листинге 11.20.

Листинг 11.20. Конструктор с параметрами – именами файлов

```
TreeTask::TreeTask (char* NameFileIn, char* NameFileOut):
                            TreeEx(NameFileIn, NameFileOut)
{
}
```

Процедуры полного обхода дерева

Во многих задачах по обработке древовидных структур приходится решать задачу обхода дерева – такого последовательного прохождения по узлам дерева, при котором каждый узел встречается ровно один раз. Для обхода бинарного дерева можно воспользоваться одним из трех способов: проходить узлы в префиксном порядке, инфиксном и в постфиксном порядке. Все способы реализуются похожим образом. Подробно коснемся одного из них – префиксного.

Префиксный порядок обхода дерева определяет последовательность посещаемых узлов в соответствии со следующей схемой действий.

1. В качестве очередной вершины взять корень дерева.
2. Произвести обработку очередной вершины в соответствии с требованиями задачи. Перейти к пункту 3.
3. Если очередная вершина:
   а) имеет обе ветви, то в качестве очередной новой вершины взять ту вершину, на которую ссылается левая ветвь, а вершину, на которую ссылается правая ветвь, поставить в очередь/стек;
   б) является конечной, то необходимо:
      - если очередь не пуста, то выбрать в качестве новой очередной вершины – вершину из очереди и перейти к пункту 2;
      - если очередь пуста, то это означает, что обход дерева окончен, следует перейти к пункту 4;
   в) имеет только одну ветвь, то в качестве новой очередной вершины выбрать ту, на которую указывает эта ветвь, и перейти к пункту 2.
4. Конец алгоритма.

Если посмотреть на то дерево, которое представлено на рис. 11.1, то, в соответствии с описанным алгоритмом, узлы будут следовать в порядке: 50, 45, 36, 11, 39, 38, 41, 47, 69, 65, 94, 75, который в соответствии с логикой алгоритма будет определен следующими действиями.

1. Берется в качестве текущего очередного узла корень дерева – 50.
2. От узла 50 отходят две ветви (45 и 69), следовательно, текущим становится левый узел 45, а правый (69) становится в очередь/стек.
3. От узла 45 отходят две ветви (36 и 47), следовательно, текущим становится левый узел 36, а правый (47) становится в очередь/стек.
4. От узла 36 отходят две ветви (11 и 39), следовательно, текущим становится левый узел 11, а правый (39) становится в очередь/стек.
5. Узел 11 не содержит ссылок, но очередь/стек не пуста. Следовательно, вершина 39 из очереди становится текущей.
6. От узла 39 отходят две ветви (38 и 41), следовательно, текущим становится левый узел 38, а правый (41) становится в очередь/стек.
7. Узел 38 не содержит ссылок, но очередь/стек не пуста. Следовательно, вершина 41 из очереди/стека становится текущей.
8. Узел 41 не содержит ссылок, но очередь/стек не пуста. Следовательно, вершина 47 из очереди/стека становится текущей.
9. Узел 47 не содержит ссылок, но очередь/стек не пуста. Следовательно, вершина 69 из очереди/стека становится текущей.
10. От узла 69 отходят две ветви (65 и 94), следовательно, текущим становится левый узел 65, а правый (94) становится в очередь/стек.

11. Узел 65 не содержит ссылок, но очередь/стек не пуста. Следовательно, вершина 94 из очереди/стека становится текущей.
12. Узел 94 имеет только одну ветвь. Поэтому 75 – узел по этой ветви становится текущим.
13. Список пуст.

Следует заострить внимание на втором пункте алгоритма. В нем речь идет об обработке очередной вершины в соответствии с требованием задачи.

Примечание

Для того чтобы сделать решение максимально независимым от способа такой обработки, в алгоритмах обхода дерева обработка реализуется просто вызовом процедуры `P`. Изменения самих процедур обхода дерева в этом случае не потребуется. В общем случае тело процедуры `P` может содержать любой набор операций по обработке и в любом количестве.

Метод `P` вызывается во всех процедурах обхода деревьев. В листинге 11.21 представлено описание метода, предназначенного для префиксного порядка обхода дерева.

Листинг 11.21. Описание префиксного порядка обхода дерева

```
void List_preorder(Ref* t)
{
  if (t != NULL)
  {
   P(t);
   List_preorder(t->Left);
   List_preorder(t->Right);
  };
}
```

Инфиксный порядок обхода дерева определяется следующим образом. Если дерево пустое, список узлов пуст. Если дерево непустое, то инфиксный порядок – это

- узлы левого поддерева в инфиксном порядке;
- корень дерева;
- узлы правого поддерева в инфиксном порядке.

Можно убедиться, что узлы дерева, которое представлено на рис. 11.1, при инфиксном обходе будут следовать в порядке: 11, 36, 38, 39, 41, 45, 47, 50, 65, 69, 75, 94. В листинге 11.22 представлено описание метода, предназначенного для инфиксного порядка обхода дерева.

Листинг 11.22. Описание метода, предназначенного для инфиксного обхода дерева

```
void List_inorder(Ref* t)
{
  if (t != NULL)
  {
   List_inorder(t->Left);
   P(t);
   List_inorder(t->Right);
  };
}
```

Постфиксный порядок обхода дерева определяется следующим образом. Если дерево пусто, список узлов пуст. Если дерево не пустое, то постфиксный порядок включает:

- узлы левого поддерева в постфиксном порядке;
- узлы правого поддерева в постфиксном порядке;
- корень дерева.

Можно убедиться, что узлы дерева, которое представлено на рис. 11.1, при постфиксном обходе будут следовать в порядке: 11, 38, 41, 39, 36, 47, 45, 65, 75, 94, 69, 50. В листинге 11.23 представлено описание метода, предназначенного для постфиксного порядка обхода дерева.

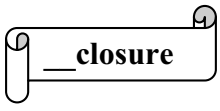
Листинг 11.23. Описание методов, предназначенных для суффиксного обхода дерева

```
void List_postorder(Ref* t)
{
  if (t != NULL)
  {
  List_postorder(t->Left);
  List_postorder(t->Right);
  P(t);
  };
}
```

Создание выходного файла из значений дерева

Алгоритм процедуры `InFile` в силу своего назначения очень прост: просматриваются все узлы дерева; из каждого узла читается значение и заносится в файл данных, предназначенный для вывода. Для того чтобы метод `InFile` можно было использовать с любым из вариантов обхода дерева, в качестве одного из параметров взят указатель на функцию обхода. Таким образом, процедура `InFile` имеет два параметра: первый параметр `aMethod` – указатель на функцию обхода, а второй `aTop` – передает ссылку на корень дерева. Первый параметр `aMethod` позволяет при формировании файла задать нужную процедуру обхода дерева.

В данном случае в процедуре `P` (см. листинг 11.25), которая вызывается в методе обхода дерева, будет всего одна операция – запись значения в файл. В листинге 11.24 представлено описание функции `InFile`.



Это определение, связанное с указателем на функцию, разрешает использовать в качестве параметра метода класса, в свою очередь, указатель на метод этого же класса.

Листинг 11.24. Описание функции `InFile`, предназначенной для формирования файла значений, взятых из узлов дерева

```
void TreeTask::InFile(void (__closure *aMethod)(Ref*), Ref* aTop)
{
  aMethod(aTop);  // вызов метода для обхода дерева
  Fout<<endl;
}
```

Листинг 11.25. Описание функции `P`

```
void TreeEx::P(Ref* t)
{
```

```
  Fout<<t->x<<"  ";
}
```

11.2. Решение задач с использованием классов «Бинарное, или двоичное, дерево»

Представленные к решению задачи опираются на разработанную систему классов. Решение задачи в том случае, если оно не может быть напрямую реализовано одним из методов класса, оформляется в виде отдельного метода класса. Рекомендуется включить этот метод в расширенный класс.

Задача 11.1. Во внешнем файле задана информация в виде наименований файлов. Отдельно снимается информация о дате последнего обращения к каждому из файлов. Требуется представить данные в виде бинарного упорядоченного дерева. Разработать программу, которая обходит дерево и удаляет из него все файлы, последнее обращение к которым происходило до определенной даты.

Стратегия решения

Предлагается воспользоваться разработанной системой классов «Двоичное дерево». При этом необходима подготовительная работа: адаптировать класс для работы с новой структурой узла. Для того чтобы принять разработанные методы классов без изменения, нужно в новую создаваемую структуру узла помимо полей, требуемых по условию задачи, *включить описание операций* над этими полями. После соответствующей подготовки класса решение задачи будет сведено к построению процедуры удаления из дерева узлов в соответствии с условием задачи. Эта процедура может быть оформлена в виде метода класса, а может быть решена, например, «под кнопку».

Предложенная стратегия определяет следующую последовательность действий для решения задачи.

1. Определить поля, которые должны характеризовать узел дерева, а именно: его информационную часть.
2. Подготовить перечень тех операций, которые используются при описании методов класса «Двоичное дерево».
3. На основании пунктов 1–2 определить интерфейс структуры, определяющей значение узла. Заметим, что имя этой структуры должно быть T (такой тип принят в созданной системе классов).
4. Разработать алгоритмы и подготовить описания операторных функций структуры.
5. Определить, где следует расположить процедуру для решения задачи.
6. Определить последовательность действий для пользователя.
7. Разработать интерфейс, обеспечивающий функциональность приложения в соответствии с описанными действиями пользователя.
8. Подготовить обработчики событий.

Решение

Для решения поставленной задачи самым трудоемким элементом является реализация пунктов 1–4, описанных в стратегии решения задачи. Если определить

структуру узла и ОПЕРАЦИИ НАД СТРУКТУРОЙ, то решение не будет содержать алгоритмических сложностей, и будет сводиться к вызову методов класса «Дерево».

Поэтому решение задачи ограничим описанием структуры узла дерева.

1. В соответствии с условием задачи в узле должна храниться следующая информация: наименование файла (берется из файла); дата последнего обращения к файлу (будем задавать датчиком случайных чисел); порядковый номер.
2. В методах класса используются следующие операции: < , >, !=, == , =. Помимо этого для удобства определим дружественные операторные функции >> и << – для ввода/вывода структуры в файл.

На основании пунктов 1 и 2, описание структуры может быть таким, как это показано в листинге 11.26.

Листинг 11.26. Интерфейс структуры, определяющий тип `T` для задачи 11.2.

```
// К задаче "О файлах"
 struct T
    {
       int FNumber; // порядковый номер
       AnsiString NFile;   // имя файла
       TDateTime ForSelect; // дата последнего обращения
       T operator=(T);  // оператор присваивания
       T operator=(int);  // оператор присваивания
       TDateTime  operator+();  //
       Boolean operator!=(T);// оператор
       Boolean operator==(T);// оператор
       Boolean operator<(T);// оператор
       Boolean operator>(T);// оператор
       friend ifstream& operator>>(ifstream& aIn, T& aT);
       friend ofstream& operator<<(ofstream& aIn, T& aT);
       operator TDateTime(){return ForSelect;}
    };
```

Заметим, что структура описания самого узла при этом не изменилась. Она имеет вид:

```
struct Ref    // Узел дерева
  {
     T x;
     Ref* Left,* Right;
  };
```

Описание операторных функций может быть следующим.

Листинг 11.27. Описание операторной функции присваивания `<T>=<T>`

```
T T::operator=(T Val)
{
      // копируются поля из параметра Val в this
    FNumber = Val.FNumber;
    NFile = Val.NFile;
    ForSelect  = Val.ForSelect;
    return *this;// вернуть *this
}
```

Листинг 11.28. Описание операторной функции присваивания вида `<T>=<int>`

```
T T::operator=(int Val)
// оператор присваивания даты и номера
```

```
{
   FNumber = Val;
      // сгенерировать дату создания
   TDateTime T =
          EncodeDate(2008,random(6)+1,random(27)+1);
   ForSelect = T.DateString(); // занести в поле
   return *this; // вернуть *this
}
```

Листинг 11.29. Описание операции «+»

```
TDateTime  T::operator + ()
{
  return  ForSelect; // вернуть дату
}
```

Листинг 11.30. Описание операции «!=»

```
Boolean T::operator!=(T Val)
{
  if(ForSelect != Val.ForSelect) return true;
  return false;
}
```

Листинг 11.31. Описание операции сравнения «==»

```
Boolean T::operator==(T Val)
{

  if(ForSelect == Val.ForSelect) return true;
  return false;
}
```

Листинг 11.32. Описание операции «<»

```
Boolean T::operator<(T Val)
{

  if(ForSelect< Val.ForSelect) return true;
  return false;
}
```

Листинг 11.33. Описание операции «>»

```
Boolean T::operator>(T Val)
{
  if(ForSelect>Val.ForSelect) return true;
  return false;
}
```

Листинг 11.34. Описание операции «>>», предназначенной для чтения структурных данных из файла

```
ifstream& operator>>(ifstream& aIn, T& aT)
{
     char H[20];
     aIn>> H;
     aT.NFile = H;
     return aIn;
}
```

Листинг 11.35. Описание операции «<<», предназначенной для записи структурных данных в файл

```
ofstream& operator<<(ofstream& aOut, T& aT)
```

```
{
   AnsiString Qtr = aT.ForSelect;
   aOut<< Qtr.c_str();
   Qtr = aT.NFile;
   aOut<< "  "<<Qtr.c_str()<<endl;
   return aOut;
}
```

Листинг 11.36. Переопределение функции `IntToStr`

```
AnsiString IntToStr(T aT)
{
  return aT.ForSelect.DateString();
}
```



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. Собрать листья дерева в цепочку символов. Определить количество листьев.
2. В древовидной структуре содержится некоторая информация рекламного толка о товарной продукции, изготавливаемой на предприятии. Количество узлов в дереве определяется из расчета: один узел – один товар. Определяется частота обращений к каждому узлу для получения информации о спросе на товар. Например, определяется число посещений каждого узла за определенный промежуток времени, и эта частота приписывается к соответствующему узлу.
3. Реорганизовать дерево, построенное для задачи 2, таким образом, чтобы элементы в нем были поставлены в порядке убывания счетчиков частоты обращений. Сравнить длину пути до узла с заданным товаром до и после реорганизации.

Алгоритмы поиска кратчайшего пути 12

Задача поиска наилучшего пути относится к самым фундаментальным задачам комбинаторной оптимизации. Известно, что важнейшим свойством любой информационной модели *является структура* этой модели. Под структурой информационной модели в общем случае понимается совокупность бинарных отношений, построенных на наборах элементарных единиц данных и действий. Эти структуры данных и порядок действий составляют сущность программ и обрабатываемой ими информации. Изображать такие структуры принято в виде графов.

ГРАФ

Граф – это пара (V, E), где V – конечное непустое множество вершин, а E – множество неупорядоченных пар (u, v) вершин из V, называемых ребрами. Говорят, что ребро $s = <u,v>$ соединяет вершины u и v. Вершины, соединенные ребрами, называются смежными. Упорядоченные пары вершин соединяются дугами (в отличие от ребер). Ребро и любая из его подсоединенных вершин называются *инцидентными*. Граф с неупорядоченными парами вершин называется неориентированным, а с упорядоченными – ориентированным графом. Пример неориентированного графа представлен на рис. 12.1, *а*.

В виде графа удобно изображать *системы дорог*. При этом вершины графа соответствуют городам, а ребра – дорогам (односторонним дорогам соответствуют дуги). Отличительная черта такого графа в том, что одна и та же пара вершин может быть связана более чем одним ребром. Понятие «двусторонние дороги» совпадает с обыденным пониманием и означает, что из одного города в другой можно доехать по одной и той же дороге, по маршруту «туда/обратно». То есть при перемещении между двумя городами, связанными дорогой, требуется проезжать одно и то же расстояние.

Разработаны различные способы задания или описания графов. Например, с помощью *матрицы смежности* или *матрицы инцидентности*.

МАТРИЦА СМЕЖНОСТИ

Под матрицей смежности понимается матрица

$$A = \{a_{i,j}\}, i = 1 \ldots n;\ j = 1 \ldots m,$$

в которой

$$\begin{cases} a_{i,j} = \textit{числу ребер, соединяющих вершины } u_i \textit{ и } v_j; \\ a_{i,j} = 0, \textit{если вершины } u_i \textit{ и } v_j \textit{ не смежные.} \end{cases}$$

МАТРИЦА ИНЦИНДЕНТНОСТИ

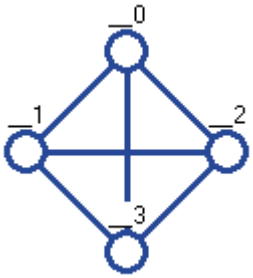
Под матрицей инцидентности понимается матрица

$$A = \{a_{i,j}\}, i = 1 \ldots n;\ j = 1 \ldots m,$$

в которой

$$\begin{cases} a_{i,j} = 1, & \textit{если вершина } u_i \textit{ инцидентна ребру } e_j; \\ a_{i,j} = 0, & \textit{если вершины } u_i \textit{ не инцидентна ребру } e_j. \end{cases}$$

Предположим, что система двусторонних дорог, связывающая между собой четыре города, описана в виде графа, как это показано на рис. 12.1, *а*. Тогда матрица инцидентности будет иметь вид, как на рис. 12.1, *б*. При этом следует заметить, что нумерация городов, предложенная здесь, начинается не от 1, а от 0, как это принято в *C++*.



а

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

б

Рис. 12.1. Представление системы двусторонних дорог, связывающей четыре города: а) изображение в виде графа; б) матрица инцидентности

Существуют различные типы задач о кратчайшем пути:

- найти кратчайший путь между двумя заданными вершинами;
- найти кратчайший путь между двумя заданными вершинами из путей, проходящих через одну или несколько заданных вершин (присутствуют дополнительные условия и ограничения);
- найти кратчайший путь между данной вершиной и всеми остальными;
- найти кратчайший путь между каждой парой вершин.

Задачи не выглядят сложными, если вариантов для выбора немного и если не учитывать различные факторы: расстояние; время прохождения с учетом ограничений скорости; дорожные условия; плотность движения; число городов, которые следует посетить и др.

Предлагаются разные подходы к решению таких задач, основанные на разном представлении графа в оперативной памяти. Один подход – представление графа дорог в *виде матрицы инцидентности.* Этот подход отличается простотой. Так, операция объединения графов, представленных матрицей инцидентности, превращается в операцию сложения соответствующих матриц. Второй подход – представление графа *в виде динамической структуры данных*.

Примечание

Очевидно, что матрица инцидентности может описываться двумерным массивом с элементами логического типа – типа `bool`.

Каждый из таких подходов имеет свои достоинства и свои недостатки. Нижеследующие задачи позволяют посмотреть на задачи поиска пути с позиции разных технологических решений и определиться со своими предпочтениями.

Задача 12.1. Дано множество городов, которые связаны между собой системой дорог так, как это показано на рис. 12.2. Требуется разработать алгоритм «*с возвратом*», позволяющий найти все возможные пути (маршруты) из одного заданного города, например, `Lex` в другой заданный город, например, `Bed` при условии, что в

один и тот же город нельзя заходить дважды. По каждому маршруту необходимо предоставить информацию о длине пути.

Решение

Алгоритм «с возвратом», упомянутый в условии задачи, зачастую связывают с решением задач из области искусственного интеллекта. Этот алгоритм ищет решение не по фиксированным правилам вычисления, а методом *проб и ошибок.* Обычно процесс вычисления методом проб и ошибок разделяется на отдельные подзадачи. Часто эти подзадачи более естественно описывать с помощью *рекурсии* [6].

МЕТОД ПРОБ И ОШИБОК

Основная стратегия метода *проб и ошибок* может быть описана следующими действиями: а) выбрать очередной ход; б) попытаться сделать этот ход; в) если ход оказался неудачным, то стереть результаты этого хода (то есть вернуться назад) и сделать другую попытку. Поэтому такой алгоритм и носит название «Алгоритм с возвратом».

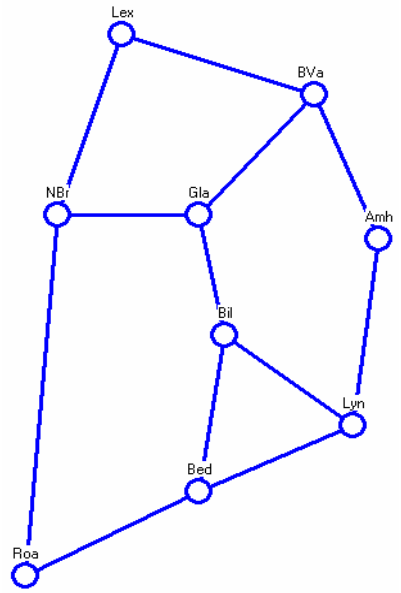


Рис. 12.2. Граф со взвешенными ребрами, описывающий систему дорог для задачи 12.1

$$X = \begin{pmatrix} 0 & 7 & 10 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 9 & 27 & 0 & 0 & 0 & 0 \\ 10 & 0 & 0 & 8 & 0 & 0 & 0 & 0 & 31 \\ 0 & 9 & 8 & 0 & 0 & 11 & 0 & 0 & 0 \\ 0 & 27 & 0 & 0 & 0 & 0 & 15 & 0 & 0 \\ 0 & 0 & 0 & 11 & 0 & 0 & 15 & 17 & 0 \\ 0 & 0 & 0 & 0 & 15 & 15 & 0 & 21 & 0 \\ 0 & 0 & 0 & 0 & 0 & 17 & 21 & 0 & 32 \\ 0 & 0 & 31 & 0 & 0 & 0 & 0 & 32 & 0 \end{pmatrix}$$

Рис. 12.3. Матрица смежности для множества городов, представленных на рис. 12.2

Ход поиска решения методом *проб и ошибок* можно рассматривать в общем виде как поисковый процесс, который постепенно строит, просматривает, а также обрезает дерево подзадач. Во многих случаях такие поддеревья растут очень быстро, обычно экспоненциально, в зависимости от заданного параметра.

Если посмотреть на граф, представленный на рис. 12.2, узлы графа соединены в тех случаях, когда между соответствующими городами проложена дорога, в противном случае соединения, ребра, нет. Числа, расположенные на графе рядом с линиями ребер, предоставляют информацию о расстоянии между соединенными городами. Так как при решении задачи необходимо учитывать не только наличие дороги между двумя городами, но и расстояние между ними, к ребрам графа приписывают необходимую информацию или, как принято говорить, придают вес.

Для описания графа дорог построим матрицу инцидентности, которую обозначим $X = \{x_{i,j}\}, i = 1 \ldots n;\ j = 1 \ldots n$. В данном случае система дорог представлена 9-ю городами. Для построения такой матрицы следует пронумеровать города в порядке, начиная с номера 0 до номера 8. Нумерацию будем вести по порядку: слева направо и сверху вниз, что соответствует: Lex – 0, Bva – 1, Nbr – 2, Gla – 3, Amh – 4, Bil – 5, Lyn – 6, Bed – 7, Roa – 8.

Из определения следует, что матрица инцидентности в общем случае должна фиксировать лишь факт наличия дороги, связывающей два города. Например, если дорога между городами i и j есть, то элемент $x_{i,j} = 1$. Если же таковой дороги нет, то $x_{i,j} = 0$. При этом перемещение из города i в себя же не рассматривается. То есть считается, что перемещаться так нельзя. Поэтому диагональные элементы, как правило, в матрице инцидентности равны 0.

В случае данной задачи матрицу инцидентности можно сделать более информативной и сформировать по-другому по следующему правилу: если два города не связаны между собой, то за значение соответствующего элемента матрицы принимают равным 0, а если два города связаны между собой, то за значение соответствующего элемента матрицы принимают значение, равное величине *расстояния* между этими городами. Это *расстояние* будет использоваться для *определения длины* маршрута. Тогда матрица инцидентности, описывающая граф, представленный на рис. 12.2, принимает вид, как на рис. 12.3.

Так как по условию, принятому для данной задачи, все дороги двусторонние, то матрица инцидентности является симметричной. Города `Lex` и `Bed`, расстояния между которыми нужно найти, не соединены между собой дорогой непосредственно. Но добраться из одного города в другой можно, проехав через другие города. И таких маршрутов несколько. Здесь требуется рассмотреть их и определить тот, который будет наименьшей длины. Изберем следующую стратегию.

1. Как только город включается в маршрут, он (его номер) помечается как занятый. Это делается для того, чтобы в дальнейшем выборе городов повторно не включать его в маршрут. Начальный город сразу становится занятым.
2. Из *одного очередного* города будем всегда переходить в другой, первый по порядку свободный город, указанный в матрице инцидентности. Этот город будет выбираться из той строки матрицы, которая соответствует номеру *этого очередного* города. Переход возможен в том случае, если город не был включен в маршрут ранее (свободен), и если к этому городу проложена дорога (элемент ненулевой).
3. При выборе очередного города для включения в маршрут возможны варианты:

а) найден свободный город, и его номер совпадает с номером города, который является конечным в маршруте;
б) найден свободный город, но его номер не является конечным в маршруте;
в) нет свободного города, который можно было бы добавить к маршруту.

4. Если выпадает вариант (а), то это означает, что найден путь между заданными городами, и этот путь следует добавить к множеству путей, определяющих решение задачи. После чего нужно перейти к поиску нового маршрута: вернуться на шаг назад (к предыдущему городу) и сделать от него другой ход.
5. Если выпадает вариант (б), то это означает, город нужно включить в маршрут, а затем нужно сделать следующий шаг в направлении поиска пути.
6. Если выпадает вариант (в), то это означает, что никакой ход из очередного города невозможен. Следовательно, нужно вернуться на шаг назад (к предыдущему городу) и сделать от него другой ход.

В соответствии с описанной стратегией, пути от города *0 (Lex)* до города *7 (Bed)* будут определяться следующими шагами:

1. Начальный город *0 (Lex)* помечаем как «занятый».
2. *Делаем попытку хода.* Выбираем ход от города *0 (Lex).* Он соответствует элементу с ненулевым значением 7. *Ход принят.* Номер этого элемента – *1(BVa).* Помечаем *1(BVa)* как «занятый».
3. *Делаем попытку хода.* Выбираем ход от города *1(BVa).* Он соответствует элементу с ненулевым значением 9. *Ход принят.* Номер этого элемента – *3(Gla).* Помечаем *3(Gla)* как «занятый».
4. *Делаем попытку хода.* Выбираем ход от города *3(Gla).* Он соответствует элементу с ненулевым значением 8. *Ход принят.* Номер этого элемента – *2(Nbr).* Помечаем *2(Nbr)* как «занятый».
5. *Делаем попытку хода.* Выбираем ход от города *2(Nbr).* Он соответствует элементу с ненулевым значением 31. *Ход принят.* Номер этого элемента – *8(Roa).* Помечаем *8(Roa)* как «занятый».
6. *Делаем попытку хода.* Выбираем ход от города *8(Roa).* Он соответствует элементу с ненулевым значением 32. *Ход принят.* Номер этого элемента – *7(Bed).* Это *конечный* пункт. Путь найден: $0 \Rightarrow 1 \Rightarrow 3 \Rightarrow 2 \Rightarrow 8 \Rightarrow 7$ или $Lex \Rightarrow BVa \Rightarrow Gla \Rightarrow Nbr \Rightarrow Roa \Rightarrow Bed$. Маршрут заносится во множество решений.
7. *Делаем возврат назад* с целью поиска нового маршрута. Для этого стираем предыдущий ход. Возвращаемся в город *2(Nbr).*
8. *Делаем попытку хода.* Но из *2(NBr)* нет хода, так как от 8 – мы отказались, а *0(Lex)* и *3(Gla)* – «заняты».
9. *Делаем возврат назад*. Стираем предыдущий ход. Возвращаемся в город *3(Gla).*
10. *Делаем попытку хода.* Выбираем ход к городу *5(Bil).* Он соответствует элементу с ненулевым значением 11. *Ход принят.* Помечаем *5(Bil)* как «занятый».
11. *Делаем попытку хода.* Выбираем ход к городу *6(Lin).* Он соответствует элементу с ненулевым значением 15. *Ход принят.* Помечаем *6(Lin)* как «занятый».
12. *Делаем попытку хода.* Выбираем ход к городу *7(Bed).* Он соответствует элементу с ненулевым значением 21. *Ход принят.* Это *конечный* пункт. Путь найден: $0 \Rightarrow 1 \Rightarrow 3 \Rightarrow 5 \Rightarrow 6 \Rightarrow 7$ или $Lex \Rightarrow BVa \Rightarrow Gla \Rightarrow Bil \Rightarrow Lin \Rightarrow Bed$. Маршрут заносится во множество решений.

13. Процесс продолжается, пока не будут исчерпаны все возможные варианты. Если ход принимается, расстояние до этого города добавляется к общему расстоянию в маршруте. Можно проверить, что в результате работы алгоритма, множество возможных путей будет представлено следующим набором маршрутов (в скобках указана длина пути):
    - $0 \Rightarrow 1 \Rightarrow 3 \Rightarrow 2 \Rightarrow 8 \Rightarrow 7$ (87)
    - $0 \Rightarrow 1 \Rightarrow 3 \Rightarrow 5 \Rightarrow 6 \Rightarrow 7$ (63)
    - $0 \Rightarrow 1 \Rightarrow 3 \Rightarrow 5 \Rightarrow 7$ (44)
    - $0 \Rightarrow 1 \Rightarrow 4 \Rightarrow 6 \Rightarrow 5 \Rightarrow 3 \Rightarrow 2 \Rightarrow 8 \Rightarrow 7$ (146)
    - $0 \Rightarrow 1 \Rightarrow 4 \Rightarrow 6 \Rightarrow 5 \Rightarrow 7$ (81)
    - $0 \Rightarrow 1 \Rightarrow 4 \Rightarrow 6 \Rightarrow 7$ (70)
    - $0 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1 \Rightarrow 4 \Rightarrow 6 \Rightarrow 5 \Rightarrow 7$ (101)
    - $0 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1 \Rightarrow 4 \Rightarrow 6 \Rightarrow 7$ (90)
    - $0 \Rightarrow 2 \Rightarrow 3 \Rightarrow 5 \Rightarrow 6 \Rightarrow 7$ (65)
    - $0 \Rightarrow 2 \Rightarrow 3 \Rightarrow 5 \Rightarrow 7$ (46)
    - $0 \Rightarrow 2 \Rightarrow 8 \Rightarrow 7$ (73)



Решение задачи целесообразно разделить на 2 части:
1) разработать класс с методами для поиска путей;
2) решить задачу с использованием методов класса.

Сделаем так, что номера городов, между которыми требуется найти расстояние, *определены как глобальные переменные*. На начальном этапе построения рекурсивной процедуры это будет облегчать восприятие (понимание) алгоритма. При желании в дальнейшем при переработке алгоритма эти переменные можно включить в поля класса и инициализировать конструкторами.

Разработка классов

Для решения задачи разработки классов, как это принято в объектно-ориентированном программировании, следует подготовить систему из трех классов: `Poisk`, `PoiskEx`, `PoiskTask`. Для описания объекта «дороги» и построения алгоритма решения задачи, предназначенной для поиска путей, в интерфейсы классов целесообразно ввести следующие поля:

- `x` – матрица инцидентности – динамический двумерный массив;
- `n` – размерность матрицы инцидентности, которая определяется по числу городов;
- `Pr` – вектор, хранящий информацию о «занятости» городов: соответствующий элемент равен 0, если город свободен, и равен 1, если город занят;
- `Distans` – вектор, хранящий список городов, включенных в маршрут;
- `l` – количество городов в маршруте.

Первые два поля предназначены для описания объекта, поэтому их следует оставить в базовом классе `Poisk`, а остальные, используемые в решении поисковой задачи, следует поместить в `PoiskEx`, расширенный класс-потомок. В третий класс `PoiskTask` поместим методы, связанные с вводом/выводом.

Примечание

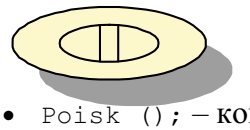
Поскольку все подробности разработки классов и описания конструкторов/деструкторов уже неоднократно изложены в решениях задач, некоторые подробности в изложении будут опущены. В таком случае считается, что достаточно ограничиться короткими комментариями.

Описание базового класса

В базовый класс, помимо матрицы инцидентности и количества городов, целесообразно также включить файловую переменную `fOut`, которая будет использоваться для создания выходного файла с полученными маршрутами. Выходной файл имеет определенное наименование: «zz.txt». Интерфейс класса представлен в листинге 12.1.

Листинг 12.1. Интерфейс базового класса, предназначенного для решения задач поиска маршрута по заданной системе дорог

```
class Poisk
  {
     protected:
     int** x;  // матрица инцидентности
     int  n;   // число городов
     ofstream fOut; // файловая переменная
       public:
       Poisk (int Initn); // конструктор
       Poisk(); // конструктор
       ~Poisk(); // деструктор
  };
```



МЕТОДЫ КЛАССА `Poisk`:

- `Poisk ();` – конструктор без параметров;
- `Poisk (int Initn);` – конструктор, который принимает для инициализации количество городов – `Initn`;
- `~Poisk();` – деструктор.

Конструктор базового класса `Poisk`

Конструктор базового класса `Poisk` имеет один параметр `Initn`, который задает количество городов. В соответствии с этим значением инициализируются поля класса, и отводится память. А также открывается файл «`zz.txt`» для записи. Таким образом, файловая переменная `fOut` связывается с именем файла. Код конструктора представлен в листинге 12.2.

Листинг 12.2. Описание конструктора

```
Poisk::Poisk (int Initn)
{
  n = Initn;
  x = new int* [n];
  for( int i=0; i<n; i++)
     x[i] = new int[n];
      // файл для вывода маршрутов
  fOut.open("zz.txt",ios::out);
}
```

Деструктор базового класса `Poisk`

Назначение деструктора – высвободить память, отведенную под массивы и закрыть выходной файл. Код деструктора представлен в листинге 12.3.

Листинг 12.3. Описание деструктора

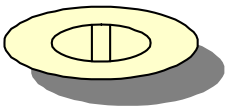
```
Poisk::~Poisk()
{
  for (int i=0; i<n; i++)
    delete x[i];
  delete x;
  fOut.close();
}
```

Описание расширенного класса

В расширенном классе помещается метод решения задачи в соответствии с описанным алгоритмом. Потому в его поля включаются массивы, необходимые для получения и хранения маршрута, а также переменная, содержащая длину маршрута: `Distans` – для фиксации маршрута в виде списка порядковых номеров городов; `l` – длина маршрута; `City` – для хранения списка наименований городов в форме буквенных обозначений; `aCity` – для хранения наименований городов, представленных условием задачи; `Pr` – для хранения признака занятости города в текущем маршруте. Как видно из интерфейса класса, наименования городов представлены в двух формах: в виде буквенных обозначений и в виде списка реальных названий. В действительности можно ограничиться только одним списком. Интерфейс класса представлен в листинге 12.4.

Листинг 12.4. Интерфейс расширенного класса

```
class PoiskEx: public Poisk
{
  protected:
     int l; // число городов в маршруте
     int* Distans;  // маршрут
     int* Pr;    // признаки занятости в маршруте
         public:
       PoiskEx (int Initn); // конструктор
       PoiskEx(); // конструктор
       ~PoiskEx(){}; // деструктор
      void Way (int&, int);
};
```



МЕТОДЫ КЛАССА `PoiskEx`:

- `PoiskEx ();` – конструктор без параметров;
- `PoiskEx (int Initn);` – конструктор, принимающий для инициализации количество городов – `Initn`;
- `~ PoiskEx ();` – деструктор;
- `void Way (int& S, int i);` – метод класса, предназначенный для поиска всех маршрутов из одного заданного города в другой, где `S` – выходной параметр, ко-

торый определяет длину маршрута, а `i` – входной параметр, который задает номер города для включения его в маршрут.

Конструктор класса-потомка `PoiskEx`

Передает значение параметра (количество городов) конструктору предка и инициализирует собственные поля. Первоначально все элементы массива-признака `Pr` равны 0, значит города не включены в маршрут. Следует отметить, что в данном случае матрица инцидентности хранится в глобальном массиве с именем `C`. При желании ее можно предварительно разместить в файле. Код конструктора представлен в листинге 12.5.

Листинг 12.5. Описание конструктора расширенного класса

```
PoiskEx::PoiskEx(int Initn):Poisk(Initn)
{
  l=0; // инициализируется число городов в маршруте
       // отводится память
  Pr = new int[n];
  Distans = new int[n];
  aCity = new char[n][4];
  City = new char[n];
// создается массив буквенных наименований, инициализируются
// массивы: массив-признак и массив-маршрут
  for (int i=0; i<n; i++)
    {
      City[i] = (char)(97+i); Pr[i]=0; Distans[i] = 0;
    }
// копируется матрица инцидентности
  for( int i=0; i<n; i++)
    for( int j=0; j<n; j++)
      x[i][j]=c[i][j];
}
```

Пустой конструктор расширенного класса-потомка

Пустой конструктор расширенного класса-потомка сообщает о том, что необходимо выполнить конструктор предка, и инициализирует собственные поля. Код конструктора представлен в листинге 12.6.

Листинг 12.6. Описание конструктора

```
PoiskEx::PoiskEx():Poisk()
{
    l=0;
    Pr = new int[n];
    City = new char[n];
    Distans = new int[n];
    for(int  i=0; i<n; i++)
      {
       City[i] = (char)(97+i); Pr[i]=0; Distans[i] = 0;
      }
    for( int i=0; i<n; i++)
      for( int j=0; j<n; j++)
         x[i][j]=c[i][j];
}
```

Поиск всех маршрутов от одного города до другого

Метод класса `Way` предназначен для поиска всех маршрутов из одного заданного города в другой, имеет `S` – выходной параметр, который определяет длину маршрута, и `i` – входной параметр, который задает номер города для включения его в маршрут.

Как это принято в технологии метода *проб и ошибок* описанная выше стратегия поиска реализована в виде рекурсивной функции. Каждый рекурсивный вызов делает попытку одного шага в схеме дорог. Функция имеет два параметра. Первый параметр `s` является и входным, и выходным. Через него передается в функцию текущее значение длины маршрута, а возвращается откорректированное значение с учетом принятого в маршрут очередного города. Второй параметр `i` – входной. Он сообщает номер очередного города, из которого делается попытка хода. При первом обращении значение `S` задается равным 0, а `i` – исходный город маршрута.

В функции предоставляется возможность получить маршрут не только в виде списка номеров, но и в виде списка наименований городов. Для этого применяется следующий механизм. Глобально для класса формируется строковая константа с именем `Nam`, в которую вписываются наименования городов в изначально определенном порядке:

```
char Nam[] = "Lex\0BVa\0Nbr\0Gla\0Amh\0Bil\0Lin\0Bcd\0Roa\0";
```

Каждое из наименований заканчивается символом `'\0'` – признаком конца строки. Поскольку под каждое имя в строке `Nam` отводится 4 позиции, всегда можно связать номер позиции в строке `Nam` (определяющий начало имени города) с номером города из списка: имя города с номером 0 – от позиции 0; имя города с номером 3 – от позиции 4*3 = 12 и так далее. Алгоритм поиска всех маршрутов из одного города в заданный можно описать следующими шагами:

1) пометить город как «занятый»;
2) включить город в маршрут;
3) проверить, не является ли он конечным в маршруте: если да, то цель достигнута и тогда:
   - записать маршрут в файл маршрутов в двух видах: в виде номеров и в виде наименований городов;
   - записать расстояние или длину маршрута;
4) сделать попытку следующего возможного хода из множества возможных ходов:
   а) если ход приемлем, то
      - увеличить длину маршрута;
      - рекурсивно повторить выполнение функции `Way`, но уже с новым городом;
   б) если ход неприемлем, то уменьшить длину маршрута;
   в) стереть предыдущий ход с целью выполнить ход заново (возврат назад).

Описание данной функции представлено в листинге 12.7.

Листинг 12.7. Описание функции для определения возможных маршрутов

```
void PoiskEx::Way(int& s, int i)
{
      // запись хода
   Pr[i] = 1; // пометить город как «занятый»
   Distans[l++] = i; // включить город в маршрут
```

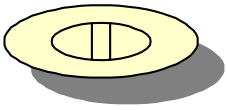
```
   if(i==7) // если город - цель достигнута
     {
           // запись маршрута – списка номеров в файл
        for( int k=0; k<l; fOut<<" "<<Distans[k],k++);
          // запись маршрута - наименований городов в файл
        for( int k=0; k<l;
                              fOut<<" "<<Nam + 4*Distans[k],k++);
           // запись длины маршрута в файл
        fOut<<" .. "<<s<<" ..    "<<endl;
     }
           // переход к следующему ходу
   for( int j=0; j<n; j++)
       {
           // выбрать следующий ход
        if(i!=j && Pr[j] == 0 && x[i][j] != 0 )
          {
           // если ход приемлем, увеличить длину маршрута
            s += x[i][j];
                // рекурсивно повторить Way, но из города j
         Way(s ,j);
           // если ход неудачный, то уменьшить длину маршрута
             s -=x[i][j];     }
          }
    Pr[i]=0; l--; // стереть предыдущий ход
}
```

Описание класса `PoiskTask`

Назначение этого класса – предоставить по требованию информацию, связанную с отображением данных. Помимо конструкторов и деструктора в нем три метода: прочитать из файла матрицу инцидентности; напечатать список номеров городов маршрута и список наименований городов маршрута. Интерфейс класса представлен в листинге 12.8.

Листинг 12.8. Интерфейс класса `PoiskTask`

```
class: public PoiskEx
{
   public:
       PoiskTask( int Initn);
       PoiskTask();
       ~PoiskTask(){};
            // взять из файла матрицу смежности
       void Get_net(char* NameFile);
       void Put_net(TListBox* aLb);
       void Put_way(TListBox* aLb);
};
```



МЕТОДЫ КЛАССА `PoiskTask`:

- `PoiskTask ();` – конструктор без параметров;
- `PoiskTask (int Initn);` – конструктор, принимающий для инициализации количество городов `Initn`;
- `~PoiskTask ();` – деструктор;

- `void Get_net(char* NameFile);` – метод класса, предназначенный для чтения из файла матрицы инцидентности;
- `void Put_net(TListBox* aLb);` – метод класса, предназначенный для вывода в список `aLb` матрицы инцидентности;
- `void Put_way(TListBox* aLb);` – метод класса, предназначенный для вывода маршрута в виде списка наименований городов в список `aLb`.

Конструктор класса-потомка `PoiskTask`

Предназначен лишь для вызова предыдущего конструктора.

Листинг 12.9. Описание конструктора

```
PoiskTask::PoiskTask():PoiskEx()
{
}
```

Чтение из файла матрицы инцидентности

Метод класса `Get_net` имеет один входной параметр `NameFile`, через который передается имя файла, содержащего матрицу инцидентности, которая предварительно в нем записана. Алгоритм работы метода:

1) открыть файл для чтения:
   а) если неудача при открытии, то необходимо выйти из процедуры;
   б) если файл пустой, то необходимо выйти из процедуры;
2) прочитать матрицу.

Описание метода представлено в листинге 12.10.

Листинг 12.10. Описание метода чтения матрицы инцидентности из файла

```
void PoiskTask::Get_net (char* NameFile)
{
  ifstream fIn (NameFile,ios::in);
  if (fIn==NULL) return;
  if (fIn.eof()) return;
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      fIn>>x[i][j];
}
```

Вывод в список матрицы инцидентности

Метод класса `Put_net` имеет один входной параметр `Box`, через который передается имя списка для вывода. В алгоритме используется вспомогательная строковая переменная `Ptr`, предназначенная для «сборки» текущей строки матрицы. Алгоритм работы метода заключается в выполнении цикла по числу строк, в котором

а) инициализируется вспомогательная строка;
б) выполняется цикл по перебору элементов текущей строки, в котором на каждом шаге элемент добавляется к строке в паре с «пробелом», необходимым для отделения одного элемента от другого;
в) строка добавляется к списку.

Описание метода представлено в листинге 12.11.

Листинг 12.11. Вывод в список матрицы инцидентности

```
void PoiskTask::Put_net (TListBox* Box)
{
  Box->Items->Add (" Матрица инцидентности ");
  AnsiString Ptr;
  for (int i=0; i<n; i++)
    {
       Ptr = "";
       for (int j=0; j<n; j++)
           Ptr = Ptr + " "+ IntToStr (x[i][j]);
       Box->Items->Add (Ptr);
    }
}
```

Вывод в список очередного маршрута

Метод класса имеет один входной параметр `Box`, через который передается имя списка для вывода. В алгоритме используется вспомогательная строковая переменная `Ptr`, предназначенная для «сборки» текущей строки матрицы. Алгоритм работы метода заключается в выполнении цикла по числу строк, в котором

а) инициализируется вспомогательная строка;
б) выполняется цикл по перебору элементов текущей строки, в котором элемент добавляется к строке в сумме с «пробелом», необходимым для отделения одного элемента от другого;
в) строка добавляется к списку.

Описание метода представлено в листинге 12.12.

Листинг 12.12. Описание метода, предназначенного для печати маршрута

```
void PoiskTask::Put_way (TListBox* Box)
{
  Box->Items->Add ("Ways - ");
  AnsiString Ptr = "";
  for (int j=0; j<n; j++)
    Ptr = Ptr + " " + (AnsiString)City[j];
  Box->Items->Add(Ptr);
}
```

Решение задачи с использованием методов класса PoiskTask

Решение задачи, если разработан класс, можно реализовать в обработчике события, например, `OnClick` для кнопки «Найти все маршруты» и описать следующими шагами:

- установить начальное расстояние в положение 0;
- создать экземпляр класса `WrCity`;
- отобразить матрицу инцидентности; найти все возможные пути из города *A*;
- отобразить решение в списке типа `ListBox`; разрушить экземпляр класса.

Описание обработчика события представлено в листинге 12.13.

Листинг 12.13. Описание метода, предназначенного для печати маршрута

```
void __fastcall TForm1::Button4Click (TObject *Sender)
{
  int s=0; // инициализация длины маршрута
  PoiskTask* WrCity;
```

```
  WrCity = new PoiskTask; // создание экземпляра класса
  WrCity->Put_net(ListBox1); // печать матрицы инцидентности
  int A = 0; // задать начальный город
  WrCity->Way (s, A); // найти все пути
  WrCity->Put_way (ListBox1); // печать в ListBox
  delete WrCity; // разрушение экземпляра
}
```

В листинге 12.14 представлено содержимое файла с именем «zz.txt», в который вписаны все маршруты из города 0 (*Lex*) до города 7 (*Bcd*), определенные для системы дорог, показанной на рис. 12.2.

Листинг 12.14. Пример содержимого выходного файла с маршрутами

0 1 3 2 8 7 Lex BVa Gla Nbr Roa Bcd .. 87 ..
0 1 3 5 6 7 Lex BVa Gla Bil Lin Bcd .. 63 ..
0 1 3 5 7 Lex BVa Gla Bil Bcd .. 44 ..
0 1 4 6 5 3 2 8 7 Lex BVa Amh Lin Bil Gla Nbr Roa Bcd .. 146 ..
0 1 4 6 5 7 Lex BVa Amh Lin Bil Bcd .. 81 ..
0 1 4 6 7 Lex BVa Amh Lin Bcd .. 70 ..
0 2 3 1 4 6 5 7 Lex Nbr Gla BVa Amh Lin Bil Bcd .. 101 ..
0 2 3 1 4 6 7 Lex Nbr Gla BVa Amh Lin Bcd .. 90 ..
0 2 3 5 6 7 Lex Nbr Gla Bil Lin Bcd .. 65 ..
0 2 3 5 7 Lex Nbr Gla Bil Bcd .. 46 ..
0 2 8 7 Lex Nbr Roa Bcd .. 73 ..

Задача 12.2. Дано множество городов, которые связаны между собой системой дорог. Требуется найти кратчайший путь между двумя заданными городами.

Решение

Рассмотрим два варианта решения задачи. Первый вариант построен на использовании матрицы смежности и известен как алгоритм Дейкстры [10]. Во втором варианте схема дорог описывается с помощью множества вершин/городов и ребер/связей.

Вариант 1

Работу алгоритма Дейкстры продемонстрируем на примере системы дорог, представленной на рис. 12.2 и при условии, что требуется найти наименьший путь от города 0 (*Lex*) до города 7 (*Bed*). Алгоритм заключается в следующем:

1. Определяются непосредственные расстояния (то есть длиной в одно ребро) от заданной вершины (*Lex*) до всех остальных вершин.
2. Отбирается наименьшее из них в качестве «постоянного наименьшего расстояния» (ребро *Lex* и *Bva* – 7).
3. Добавляется это наименьшее расстояние 7 к длинам ребер от новой вершины *Bva* до всех остальных: (7 + 27), (7 + 9).
4. Сравнивается эта сумма с предыдущим расстоянием от *Lex* до всех остальных вершин и заменяется прежнее расстояние, если новое меньше.
5. Затем новая вершина *Bva* удаляется из списка вершин, до которых еще не определены кратчайшие расстояния. В данном примере последняя вершина

этого списка *Roa* заменяет вновь определенную вершину *Bva* и список укорачивается на один элемент.

6. Затем весь процесс повторяется, присоединяется новое кратчайшее расстояние к списку, пока конечная вершина *Red* не окажется соединенной с *Lex* маршрутом из выделенных ребер.

Алгоритм Дейкстры также опирается на матрицу инцидентности. Но в отличие от предыдущей задачи, если города не связаны между собой, то соответствующий элемент считается равным бесконечности, а не нулю. Такой прием *при поиске минимума* позволит алгоритму отказаться от «такой» большой дороги. Но поскольку числа «бесконечность» нет в языке программирования, то роль бесконечности будет играть заведомо большое число, например 100 000.

Функция `Dijkst`, созданная для реализации метода, имеет 8 параметров следующего назначения.

Входные параметры:

`A` – матрица инцидентности;

`M` – число городов;

`V0` – номер начального города;

`W` – номер конечного города;

Выходная информация будет состоять из последовательностей упорядоченных троек, характеризующих маршрут, элементы которых расположены в трех массивах:

`From` – для указания города, от которого берется связь;

`To` – для указания города, до которого делается шаг;

`Leng` – для указания расстояния между городами;

`EdGes` – для указания числа связей на настоящий момент.

Для работы алгоритма требуются также вспомогательные переменные:

- `NumUn` – число невыбранных вершин/городов. На начальном этапе из общего количества городов определенной является только одна вершина – *V0*. Остальные – неопределенные, то есть не выбрано (`M - 1`) городов;
- `UndEt` – массив из невыбранных городов. На начальном этапе в этом массиве перечислены номера всех городов за исключением одного – начального города. В процессе работы алгоритма *массив будет уменьшаться*, но *начало отсчета в нем всегда будет идти от 0*. Для того чтобы не потерять вершину, на место вершины, исключаемой из списка неопределенных вершин, будет помещаться последняя невыбранная вершина;
- `VerTex` – вершины, лежащие на кратчайшем пути. На начальном этапе все элементы этого массива соответствуют номеру города, от которого ищется путь;
- `Dist` – массив, каждый `i`-элемент которого будет содержать наименьшее расстояние между невыбранной (из `UndEt`) `i`-вершиной и некоторой выбранной вершиной (из `VerTex`). Первоначально в массив заносятся расстояния от вершины `V0`;
- `Next` – очередная вершина/город, добавляемый к кратчайшему пути. Первоначально `Next = V0`.

Предложенный алгоритм можно описать следующими шагами:

1. Инициализируются переменные: список неопределенных вершин; массив расстояний до неопределенных вершин.

2. Пока не будет достигнута конечная вершина:
   а) определяется та вершина из числа неопределенных вершин, которая находится на наименьшем расстоянии от вершин, определенных ранее;
   б) найденная неопределенная вершина добавляется к дереву кратчайших путей;
   в) найденная неопределенная вершина исключается из списка неопределенных вершин;
   г) обновляется массив кратчайших расстояний до всех неопределенных вершин.

Описание метода представлено в листинге 12.15.

Листинг 12.15. Поиск кратчайшего пути методом Дейкстры

```
void Dijkst(int A[9][9], int M, int V0, int W, int From[9], int To[9],
int Leng[9], int& EdGes)
//     Формальные параметры
// A(i,j) - матрица связности. Если ребро отсутствует, то A(i,j)=100000
// M - вершины пронумерованы 1, 2, ..., M  2<=M<=100
// V0 - начальная вершина
// W - конечная вершина
// From(i) - содержит i-ребро в дереве кратчайших путей
//           от вершины From(i)
// To(i) - к вершине To(i) длиной Leng(i)
// Leng(i) -
// EdGes - число ребер в дереве кратчайших путей на настоящий момент
//
//          Внутренние переменные
// Dist(i) - кратчайшее расстояние от UndEt(i)
//           до частичного дерева кратчайших путей
// Next - очередная вершина, добавленная к дереву кратчайших путей
// NumUn - число неопределенных вершин
// UndEt(i) - список неопределенных вершин
// VerTex(i) - вершины частичного дерева кратчайших путей, лежащие
//             на кратчайшем пути от UndEt(i) до V0
{
  int j, jk, L, k;
  int Dist[9],UndEt[9],VerTex[9];
//       Инициализация переменных
   EdGes = -1;
   int Next = V0;
   int  NumUn = M-1;
   fOut<<"Инициализация переменных "<<endl;
   for(int i=0; i<M; i++)
     {
       UndEt[i]=i;
       Dist[i] = A[V0][i];
       VerTex[i] = V0;
       fOut<<UndEt[i]<<"  "<<Dist[i]<<"  "<<VerTex[i]<<endl;
     }
   UndEt[V0] = M-1;
   Dist[V0] = Dist[M-1];
   goto A350;
// Обновление кратчайшего расстояния до всех неопределенных
//     вершин
   A200:   for (int i=0; i<=NumUn; i++)
             {
                j = UndEt[i];
```

```
                jk = L+A[Next][j];
                if (Dist[i] <= jk) continue;
                VerTex[i] = Next;
                Dist[i] = jk;
              }
//  Определение кратчайшего расстояния до неопределенных
// вершин
   A350:   k = 0;
           L = Dist[0];
           for(int i=0; i<=NumUn; i++)
             {
               if(Dist[i] >= L) continue;
               L = Dist[i];
               k = i;
             }
// Добавление ребра к дереву кратчайших путей
   EdGes  = EdGes +1;
   From[EdGes] = VerTex[k];
   To[EdGes] = UndEt[k];
   Leng[EdGes] = L;
   Next = UndEt[k];
// Достигнут ли конечный пункт W ?
   if(Next == W) // если достигнут
      {
            // записать в маршрут
         int Way[9];
         int j = EdGes;
         Way[0] = To[j];
         Way[1] = From[j];
         int U = 1;
         while (j>=0)
          {
            while (To[j]!= Way[U] && j>=0) j--;
            if (j>=0) Way[++U] = From[j];
          }
         return; // выход
      }
      // если конечный город не достигнут, то
      // исключение вновь определенной вершины из списка
      // неопределенных
   Dist[k] = Dist[NumUn];
   UndEt[k] = UndEt[NumUn];
   VerTex[k] = VerTex[NumUn];
// если остались неопределенные вершины
   NumUn = NumUn -1;
   goto A200;
}
```

Можно убедиться, что в результате применения алгоритма к системе городов, представленной на рис. 12.2, получен маршрут, который описывается следующими значениями:

From	0	0	1	3	1	2	2	5	5
To	1	2	3	5	4	8	8	6	7
Leng	7	10	16	27	34	41	41	42	44

Последовательность городов, входящих в маршрут (от его конца к началу), может быть определена по значениям этого массива следующим образом:

- берутся последние значения у `From` и `To (7,5);`
- находится в `To` место `5` (включается соответствующее ему `3`);
- находится в `To` место `3` (включается соответствующее ему `1`);
- находится в `To` место `1` (включается соответствующее ему `0`).

Получается следующая цепочка городов: `7`, `5`, `3`, `1`, `0` с расстоянием в `44`. Можно сравнить с результатами, полученными методом *проб и ошибок*, и убедиться в их совпадении.

Вариант 2

Предлагается другой способ решения. В данном случае связь между городами/вершинами описывается не с помощью матрицы смежности, а с помощью соответствующего описания вершин сети. В структуру, описывающую вершину/узел, входят следующие поля:

- номер предыдущего узла. К узлу может подходить несколько ребер, поэтому этот номер будет уточняться по ходу алгоритма поиска наилучшего пути;
- указатель на список ребер, выходящих от этого узла. Формируется по типу стека;
- лучшее время для узла. Уточняется и определяется в процессе работы алгоритма;
- состояние узла. Принимает одно из двух значений: `true` – узел не обработан; `false` – узел обработан.

Описание структуры узла приведено в листинге 12.16.

Листинг 12.16. Описание структуры узла в схеме дорог

```
struct TopRec
  {
     int Previous; // номер предыдущего узла
     FReb* Rebs;  // список ребер, выходящие из узла
     float BestTime; // лучшее время/расстояние
     Boolean Go;  // состояние вершины
  };
```

Структура ребра, используемая в описании, в свою очередь, характеризуется:

- номером вершины/узла начального;
- адресом вершины/узла следующего;
- величиной времени/расстояния между соединенными городами.

Описание структуры ребра представлено в листинге 12.17.

Листинг 12.17. Описание структуры ребра в схеме дорог

```
struct FReb
  {
     int NTop;  // номер вершины
     FReb* next; // адрес следующей вершины
     float TimeReb; // время до вершины
  };
```

Последовательность действий в соответствии с алгоритмом может быть описана следующими шагами:

1) инициализировать вершины/узлы;
2) ввести исходные данные, на основании которых к каждому узлу «прицепляется» список исходящих ребер;

3) найти кратчайший путь по схеме алгоритма.

Продемонстрируем работу алгоритма на примере решения задачи с системой городов, представленной на рис. 12.2. На схеме представлено 9 городов/узлов. Количество ребер равно 12. Тогда порядок следующий:

1. Инициализируются узлы начальными значениями. Начальное состояние узла показано на рис. 12.3.

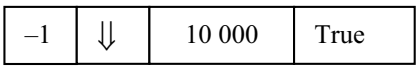


Рис. 12.3. Начальное состояние узла

2. К каждому узлу привязываются соответствующие ребра. В результате вершины принимают вид, как на рис. 12.4.

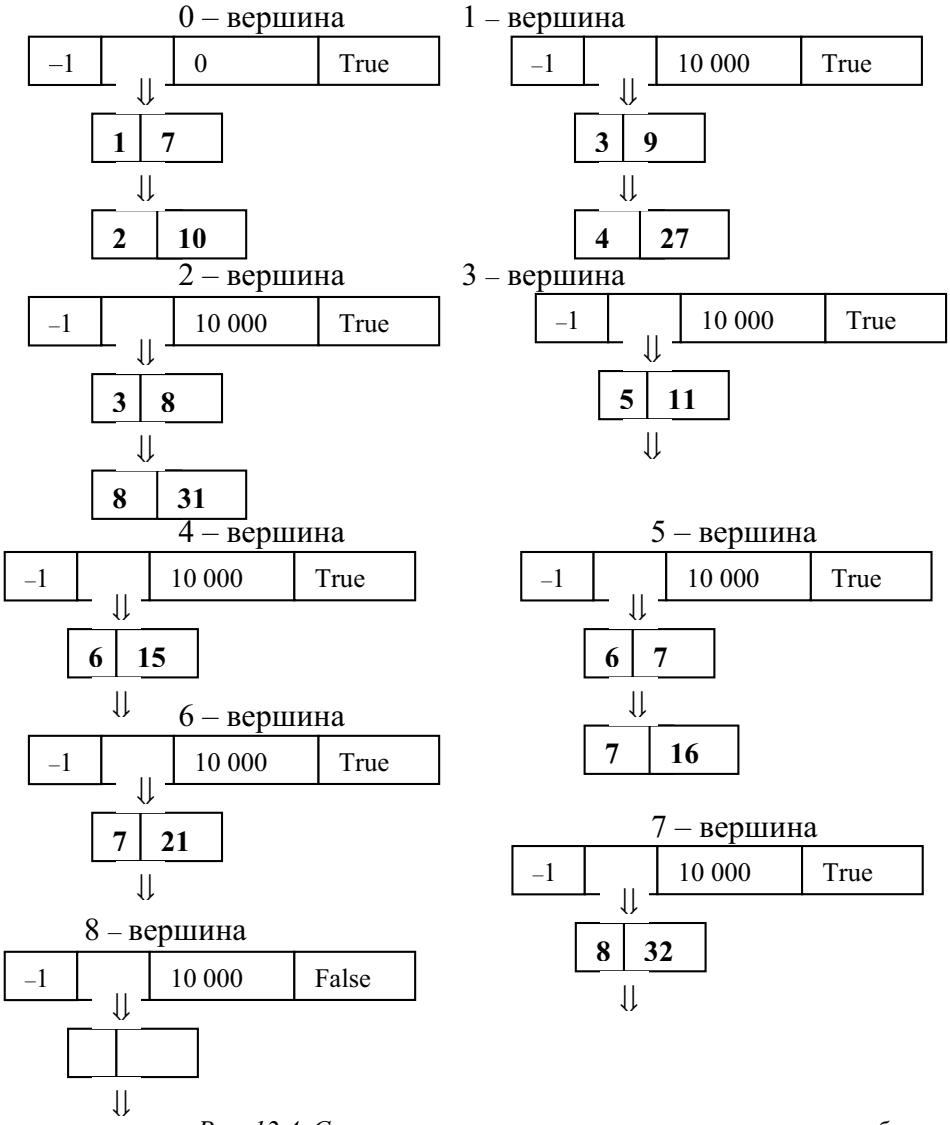


Рис. 12.4. Состояние структуры узлов с прицепленными ребрами

3. От начального узла – это узел с номером 0, находится самый короткий путь до каждой из вершин. Информация, какую вершину взять следующей, идет от вер-

шин, которые находятся на концах соединяющих ребер. По мере прохождения вершин, накопленное (так как идет переход от старта через вершины), наилучшее время записывается в структуру и вписывается номер предыдущего узла:

а) узел с номером 0 остается неизменным. Это *стартовый* узел. Наименьшее расстояние до самого себя равно 0. Возможен ход до 1-й и до 2-й вершины.
б) узел с номером 1. Наименьшее расстояние от узла 0 до этой вершины есть 7 (0 + 7 – берутся из узла 0);
в) узел с номером 2. Наименьшее расстояние от 0 вершины до вершины 2 есть 10 (0 + 10 – берутся из узла 0);
г) узел с номером 3. Наименьшее расстояние от 1 (начиная от стартовой 0) вершины до вершины 3 есть 16: (7 + 9 от 1 вершины) < (10 + 8 от 2 вершины);
д) узел с номером 4. Наименьшее расстояние от 1 (начиная от стартовой 0) вершины до вершины 3 есть 34 (7 + 27);
е) узел с номером 8. Наименьшее расстояние от 2 (начиная от стартовой 0) вершины до вершины 8 есть 41 (10 + 31);
ж) узел с номером 5. Наименьшее расстояние от 3 (начиная от стартовой 0) вершины до вершины 5 есть 27 (11 + 16);
з) узел с номером 6. Наименьшее расстояние от 4 (начиная от стартовой 0) вершины до вершины 8 есть 49 (34 + 15);
и) узел с номером 7. Наименьшее расстояние от 5 вершины (начиная от стартовой 0) до вершины 7 есть 44 (27 + 17);
к) узел с номером 6. Наименьшее расстояние от 5 вершины до вершины 6 есть 42 (27 + 15).

В результате структуры узлов изменили свои значения и приняли вид, как на рис. 12.5.

4) Осталось выбрать путь. Производим этот выбор, начиная от конечного узла, то есть от узла с номером 7, производя переход от очередного предыдущего до следующего предыдущего.

Интерфейс класса для решения задачи может быть представлен, как показано в листинге 12.18.

Листинг 12.18. Интерфейс класса для решения задачи с использованием списков

```
class  WayNet
{
    TopRec* Tops; // список указателей на узлы
    int Start; // начальный узел
    int Finish;  // конечный узел
    int  NumberTops;  // число узлов
    int NumberReb;  // число ребер
  public:
     WayNet(int InitStart, int InitFinish,
           int InitNumberTops, int InitNumberReb);
     void InitTops(); // инициализация вершин для старта
     void PrintTops(TMemo*); // печатать
     void Do_stek(char* NameFile); // Взять данные из файла
     void DoMethod(); // метод
 };
```

Путь из 7 до 0:
7 ..... 5 ..... 3 ..... 1 0
Требуемое время 44

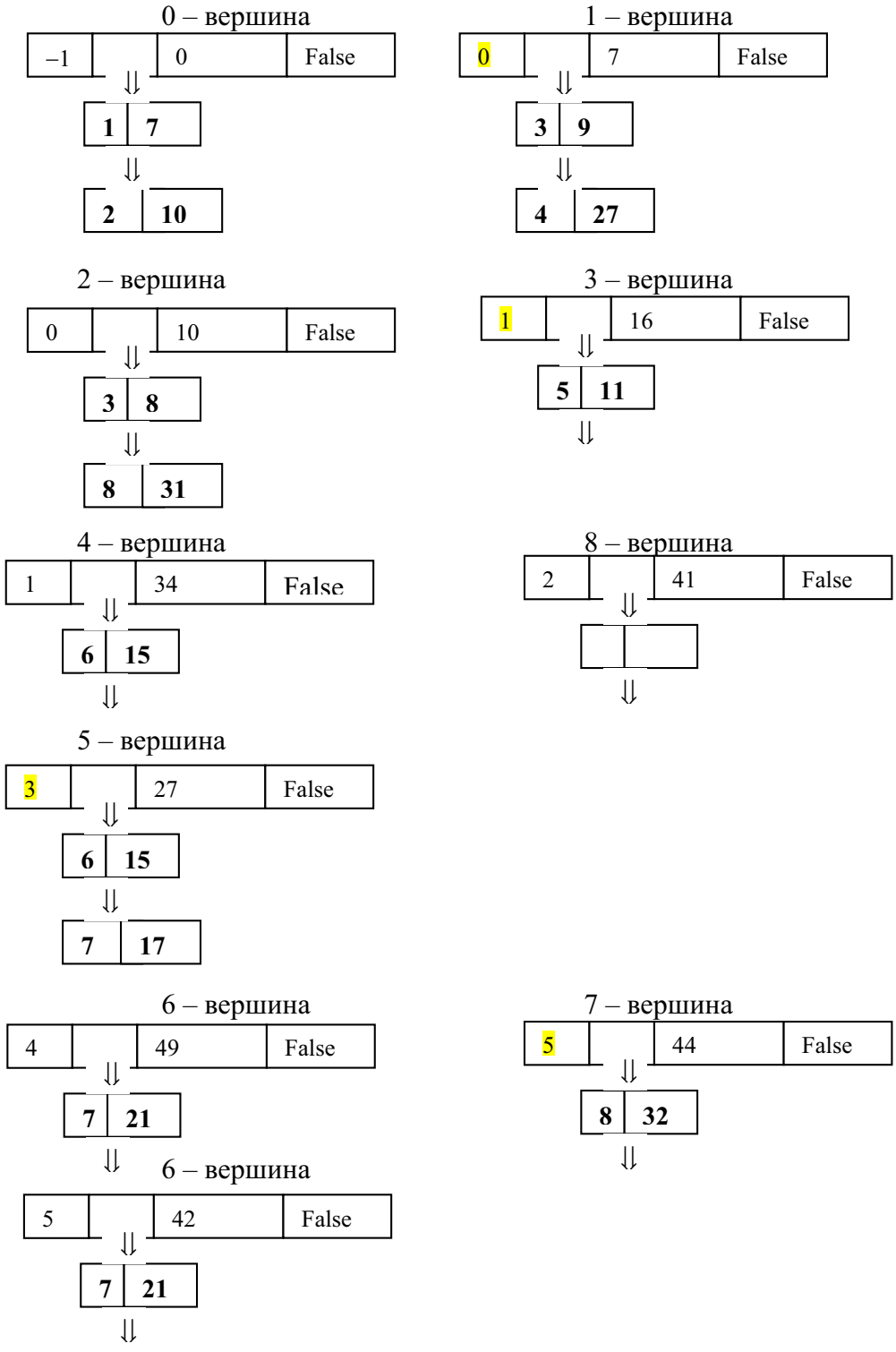


Рис.12.5 Состояние структуры узлов после пересчета расстояний от стартового узла

Конструктор класса

Конструктор имеет 4 входных параметра, которые задают начальные значения полей класса. Помимо этого в нем вызывается процедура, предназначенная для инициализации узлов, описывающих представленную систему дорог. Описание конструктора представлено в листинге 12.19.

Листинг 12.19. Описание конструктора класса `WayNet`

```
WayNet::WayNet (int InitStart, int InitFinish,
                  int InitNumberTops, int InitNumberReb)
{
    Start = InitStart;
    Finish = InitFinish;
    NumberTops = InitNumberTops;
    NumberReb = InitNumberReb;
    InitTops();
}
```

Инициализация вершин

Метод `InitTops` не имеет параметров. В нем отводится память под массив узлов. И поля структуры `TopRec`, которая описывает отдельный узел, инициализируется начальными значениями в соответствии с принятыми и представленными на рис. 12.3. Описание метода представлено в листинге 12.20.

Листинг 12.20. Описание метода класса, предназначенного для инициализации вершин/узлов, описывающих систему дорог

```
void  WayNet::InitTops ()
{
   Tops = new TopRec[NumberTops];
   for( int i = 0; i<NumberTops; i++)
      {
            Tops[i].Rebs = NULL;
            Tops[i].BestTime = 10000;
            Tops[i].Go = True;
            Tops[i].Previous = -1;
      }
   Tops[Start].BestTime = 0;
}
```

Распечатка вершин

Метод `PrintTops` имеет один параметр `Box`, который указывает поле, в котором следует расположить маршрут, полученный в результате решения задачи. В нем всего один цикл по числу узлов, представляющих систему дорог, в котором информация из узлов собирается в строку – вспомогательную переменную `Ptr`, которая добавляется к полю `Box`. Описание метода представлено в листинге 12.21.

Листинг 12.21. Описание метода класса, предназначенного для вывода кратчайшего маршрута, найденного в представленной системе дорог

```
void WayNet::PrintTops(TMemo* Box)
{
   for( int i = 0; i<NumberTops; i++)
    {
     AnsiString Ptr ="";
     Ptr = " Из  "+IntToStr(Tops[i].Previous)+ " ";
```

```
      Ptr = Ptr+ "к вершине " +IntToStr(i) + " ";
      Ptr = Ptr + " за время "+
                        FloatToStr(Tops[i].BestTime );
      Box->Lines->Add(Ptr);
     }
    return;
}
```

Связывание вершин с ребрами

Метод `Do_stek` имеет один параметр `NameFile`. Этот параметр задает имя файла, который содержит исходные данные для ребер.

ОБЩАЯ структура ФАЙЛА

Общая структура файла с информацией для описания графа дорог предлагается следующая:

- количество строк в файле совпадает с количеством городов, от которых есть связывающий его путь с другим городом;
- первым элементом в строке стоит целое число – номер города, от которого ведет путь;
- вторым элементом в строке стоит целое число – номер города, к которому ведет путь;
- третьим элементом в строке стоит число – время, за которое можно пройти путь между связывающими городами.

Эти данные описывают систему дорог, а именно: номера городов, которые они связывают; время, за которое можно пройти расстояние между связываемыми городами. Формирование для каждого узла реализуется в виде стека так, как это показано на рис. 12.4. Основу алгоритма составляет цикл по числу ребер. В цикле выполняются следующие операции:

- отводится место под узел, представляющий ребро в схеме дорог;
- из файла читается номер начального города (этот номер будет определять номер узла/города в схеме дорог);
- из файла читается конечный город и заносится в структуру ребра в схеме дорог;
- из файла читается время и заносится в структуру ребра в схеме дорог;
- формируется адресное поле в структуру ребра;
- переопределяется адрес стека.

Описание метода представлено в листинге 12.22.

Листинг 12.22. Описание метода класса, предназначенного сформировать для каждой вершины стек, состоящий из ребер, связанных с ней

```
void WayNet::Do_stek (char* NameFile)
{
    FReb* Work;
    int j; // номер ребра
    ifstream fin (NameFile, ios::in);
    for (int i = 0; i<NumberReb; i++)  // для каждого ребра
     {
       Work = new FReb; // место под узел
       fin>>j;          // нач. узел
       fin>>Work->NTop;   // конеч. узел
```

```
        fin>>Work->TimeReb;   // время
        Work->next = Tops[j].Rebs;  // инициализация указателя
        Tops[j].Rebs = Work;
      };
}
```

Поиск наименьшего пути

Метод `DoMethod` не имеет параметров. В программе используется внутренняя переменная `fout`, которая позволяет открыть файл с именем «z.txt», предназначенный для записи маршрута, определенного в результате проведенного поиска. В цикле пересчитываются характеристики структуры узлов. Для каждого из узлов определяются минимальные расстояния и вершина, его предоставляющая. Для представленной задачи состояние узлов после пересчета примет вид, как это показано на рис. 12.5. Описание метода представлено в листинге 12.23.

Листинг 12.23. Описание метода класса, предназначенного для поиска оптимального маршрута

```
void WayNet::DoMethod()
{
      // выходной файл
   ofstream fout("z.txt",ios::out);
   FReb* Rebro; // вспомогательная переменная
   float Sum; // для суммы
// т.к. изменение параметров цикла в самом начале
// оператора while, то начальные значения – от -1
   int i = Start-1;
   int j = -1;
   while( j < NumberTops-1) // цикл по вершинам
     {
        j++;
        // чтобы не было выхода за кол-во вершин
        i = i % (NumberTops-1) + 1;
        if (Tops[i].Go) // если вершина не обработана
          {
                 // проверяются ребра на мин-ое время
            j = -1;
            Rebro = Tops[i].Rebs; // на начальное ребро
            while ( Rebro != NULL)
              {
                Sum = Tops[i].BestTime + Rebro->TimeReb;
                if(Sum < Tops[Rebro->NTop].BestTime)
                  {
                       // изменяются параметры мин. узла
                    Tops[Rebro->NTop].BestTime = Sum;
                    Tops[Rebro->NTop].Previous = i;
                    Tops[Rebro->NTop].Go = True;
                  }
                 Rebro = Rebro->next; // к след. ребру
              }
             Tops[i].Go = False; // узел обработан
          }
     }
          // определение пути
    if(Tops[Finish].BestTime != 10000 &&
                                   Tops[Finish].BestTime !=0)
       {
            fout<<"Путь из "<< Finish <<" до " <<Start<<endl;
```

```
            i = Finish; // финиш
            while( i != -1) // пока есть предшественники
               {
                    fout<<" "<<i<<" "; // печать
                    i = Tops[i].Previous; // к предшеств.
                    if (i!=-1)
                         fout<<".........";
                }
             fout<<endl<< "Требуемое время  "
                                <<Tops[Finish].BestTime<<endl;
          }
     else fout<<"Пути нет "<<endl;
     fout.close();
}
```



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. По системе односторонних дорог определите, есть ли в ней город, из которого можно добраться до заданного города, проезжая не более 100 км.
2. Задана система односторонних дорог. Найдите путь, соединяющий города *A* и *B* и не проходящий через заданное множество городов.
3. Задана система двусторонних дорог, причем для каждой пары городов можно указать соединяющий их путь. Найдите такой город, для которого сумма расстояний до остальных городов минимальна.

Рисование простейших графических объектов 13

Приложение, разработанное для работы в среде *Win32*, имеет графический интерфейс, то есть окно программы и все элементы на нем представляют собой рисунки, отображаемые на экране монитора.

Легкое и непринужденное манипулирование окнами на рабочем столе *Windows* заставляет систему выполнять огромную работу по обновлению изображений этих окон. Или, по крайней мере, по обновлению тех участков, которые стали видимыми после того, как закрывающее окно отодвинулось в сторону или просто само окно среди других окон переместилось на передний план, то есть окно полностью стало видимым.

Для более быстрого проектирования и разработки приложения в среде *C++ Builder* каждый элемент окна, как и само окно, оформлены в класс (в смысле ООП), который реализует возможности функционирования этого элемента и его отображение на экране монитора.

Для выполнения операций рисования в *C++ Builder* разработан *класс* `TCanvas`.

Canvas (холст)

Объект класса `TCanvas` можно рассматривать как «холст», на котором «пишется» изображение. Он имеет довольно сложную структуру и позволяет разработчику приложения рассматривать область отображения компонента как матрицу *пикселей*.

Пиксель

Пиксель представляет собой мельчайшую точку изображения, каждой из которых присвоено значение цвета. С каждым пикселем `TCanvas` сопоставлена пара координат *X* и *Y*. Система координат `TCanvas`, как это принято в компьютерной графике, имеет началом левый верхний угол изображения. Координата *X* возрастает при перемещении слева направо, а координата *Y* – при перемещении сверху вниз.

На холсте можно рисовать, просто присваивая пикселям определенные значения цвета, используя его свойство `Pixels`, которое можно интерпретировать как прямоугольную матрицу с элементами типа `TColor`. Помимо попиксельного формирования изображений, объект `Canvas` имеет набор методов для построения изображения с использованием ГЕОМЕТРИЧЕСКИХ ПРИМИТИВОВ: отрезков прямой линии, прямоугольников, эллипсов, ломаных линий, кусочно-гладких кривых в виде кривых Безье, а также средства заливки областей заданным цветом.

Для единообразия программирования визуализации различных компонентов на экране в их состав входит холст, то есть объект `Canvas`. С помощью свойств и методов класса `TCanvas` можно создавать рисунки на поверхности видимых объектов, которые включают этот класс и, соответственно, имеют свойство `Canvas`. Инструментами для создания рисунка на холсте `Canvas`, используемыми, в свою очередь, различными его методами, являются объекты: *перо* – типа `TPen`, *кисть* – типа `TBrush` и *шрифт* – типа `TFont`.

При конструировании формы в *C++ Builder* можно также использовать *специализированные* компоненты, предназначенные для создания визуальных эффектов

и изображений. Действия, связанные с построением изображения в таких случаях, напоминают работу в среде графического редактора, но с помощью программного кода. Наиболее часто используются такие графические компоненты, как графический образ `Image`, геометрическая фигура `Shape`.

Примечание

Компонент `TImage` предназначен для показа на форме изображений: битовой карты (`TBitmap`), метафайла (`TMetafile`), значка (`TIcon`). Также он позволяет формировать изображение программным путем. При этом компонент `TImage` обеспечивает хранение и корректное отображение построенного изображения в ходе работы с окнами системы *Windows*.

Центральным элементом компонента `TImage`, конечно же, является объект `Canvas`. Помимо этого компонент `TImage` обладает рядом других полезных методов и свойств, таких как:

`Picture` – свойство типа `TPicture`, содержит отображаемый графический объект типа битовой матрицы, пиктограммы, метафайла или определенного пользователем типа;

`AutoSize` – свойство типа `bool`. Если установить свойству значение `true`, то размер компонента `Image` будет автоматически подгоняться под размер помещенной в него картинки. Если свойство `AutoSize` установлено в `false`, то изображение может не поместиться в компонент или наоборот, так как площадь компонента может оказаться много больше площади изображения;

`Stretch` – свойство типа `bool`. Если установить значение `true` в это свойство, то размер картинки будет автоматически подгоняться под размер компонента `Image`. Необходимо помнить, что пропорции компонента и картинки, помещенной в него, не обязательно совпадают, а следовательно, изображение исказится;

`Center` – свойство типа `bool`. Если установить значение `true` в это свойство, то картинка будет автоматически центрироваться, если размер компонента `Image` превышает размер картинки;

`Transparent` – свойство типа `bool`. Если установить значение `true` в это свойство, то цвет фона изображения будет прозрачным. Он используется, чтобы сквозь рисунок было видно нижележащее изображение.

`LoadromFile`, `SaveToFile` – методы свойства `Picture`, позволяющие загружать изображения из файла в компонент или сохранять их в файле.

Помимо матрицы пикселей, пера и кисти, компонент `TCanvas` имеет большое количество методов, позволяющих строить изображение целыми готовыми примитивами – прямоугольниками, эллипсами, отрезками и ломаными, и тому подобное. При этом легко можно настраивать параметры рисования этих примитивов, используя свойства холста.

Задача 13.1. Требуется предусмотреть такую окантовку сторон окна `Image`, чтобы каждая сторона отличалась стилем, толщиной линии и цветом, как показано на рис. 13.1.

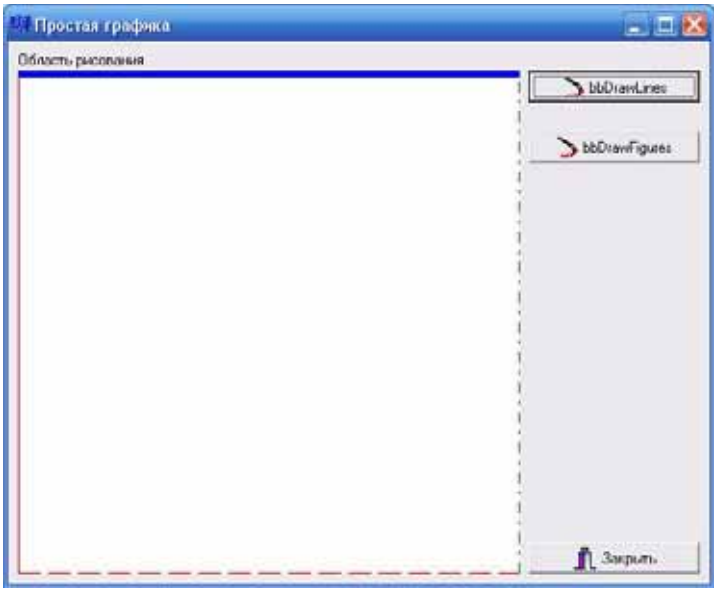


Рис. 13.1. Интерфейс приложения для окантовки сторон окна `Image`

Решение

Согласно рис. 13.1 компонент `Image` оформлен как квадрат и прижат к левому краю формы. Левая и нижняя сторона квадрата окантована красным цветом, правая и верхняя отличаются толщиной и стилем линии.

Поскольку компонент `Image` включает класс `TCanvas`, для рисования окантовки целесообразно воспользоваться методами и свойствами этого класса, а именно: воспользуемся *пером* `Pen`, которое можно *перемещать* при изображении линий, и которое *имеет свойства*, необходимые для решения задачи:

Color – цвет пера; список стандартных цветов можно увидеть в *Help*, в теме *TControl/Color*);
Width – ширина или толщина линии; задается целым числом;
Mode – режим рисования линии; определяет взаимодействие нового изображения с ранее нанесенным рисунком;
PenPos – текущие координаты графического курсора;
Style – стиль вычерчиваемой линии (сплошная, пунктирная и тому подобное):

- *psSolid* – сплошная линия;
- *psDash* – штриховая линия;
- *psDot* – пунктирная линия;
- *psDashDot* – штрихпунктирная линия;
- *psDashDotDot* – штрихпунктирная линия (две точки).

Внимание

Штриховая и пунктирная линии вычерчиваются только при *ширине*, равной *единице*.

Простейшее рисование выполняется вычерчиванием набора отрезков. При этом можно перемещать перо по прямой «не оставляя следа», или «оставлять след» – чертить отрезки прямой.

MoveTo, LineTo

Перемещать перо можно посредством процедуры холста `MoveTo (int X, int Y)`. Это означает, что перо изменяет положение, но не оставляет следа. А *прочерчивать отрезок прямой линии* из текущей точки до указанной может метод `LineTo(int X, int Y)`. После работы обеих процедур текущей точкой является точка с координатами (*X*, *Y*), значения которых указаны в качестве входных параметров.

Алгоритм решения задачи прост и понятен: настраиваются параметры пера, в зависимости от того, какой отрезок необходимо прочертить, и выполняется само построение отрезка. При этом отрезок строится от текущей позиции графического курсора до точки, заданной в параметрах метода построения. Откуда следует, что решение задачи окантовки окна `Image` будет сводиться к выполнению последовательности операций:

1) задать цвет пера;
2) поставить перо в начальную точку (эта точка становится текущей);
3) провести линию от текущей точки до заданной (эта заданная точка становится текущей);
4) задать стиль пера;
5) провести следующую линию;
6) повторить пункты 4–5 для проведения оставшихся линий.

Описанный алгоритм целесообразно реализовать на событие `OnClick` под кнопку, например с именем `bbDrawLines`. Код обработчика приведен в листинге 13.1.

Листинг 13.1. Рисование серии отрезков прямой в окне `Image`

```
void __fastcall
             TfoSimpleGraph::bbDrawLinesClick (TObject *Sender)
{
  // Рисование отрезков разными стилями, толщиной и цветом
  Image->Canvas->Pen->Color = clRed;   // задать цвет пера
                 // стиль используется текущий; здесь начальный
  Image->Canvas->MoveTo (0, 0);  // переместить перо
                   // в начальную точку; она становится текущей
  Image->Canvas->LineTo (0, Image->Height - 1);
                                          // прочертить отрезок
                              // от текущей в указанную; теперь
                              // эта точка становится текущей
  Image->Canvas->Pen->Style = psDash;
                           // задать стиль пера: штриховая линия
  Image->Canvas->LineTo (Image->Width - 1, Image->Height - 1);

  Image->Canvas->Pen->Color = clGreen;  // изменить цвет пера
  Image->Canvas->Pen->Style = psDashDot;  // изменить стиль пера
  Image->Canvas->LineTo (Image->Width - 1, 2);
                                           // прочертить отрезок

  Image->Canvas->Pen->Color = clBlue;  // изменить цвет пера
  Image->Canvas->Pen->Width = 5;  // изменить толщину пера
```

```
                                  //! стиль не будет использоваться
  Image->Canvas->LineTo (0, 2);  // прочертить отрезок
}
```

Напоминание

При использовании хорошо проработанных приложений, таких как *C++ Builder*, следует помнить, что значения всех переменных инициализируются начальными значениями. Этого правила необходимо придерживаться и при разработке собственных приложений.

Так как параметры пера определены при создании холста, то присваивание значений в тексте подпрограммы лишь меняет значения, заданные по умолчанию, на новые. Поэтому термины «задать» и «изменить» здесь фактически выступают как синонимы. Также следует помнить, что параметры пера остаются без изменения, пока не заданы новые значения. Это позволяет «экономить» операции, то есть задавать новые параметры только в случае необходимости.

Примечание

Также следует помнить, что *при построении отрезка толщиной более единицы*, он имеет толщину в несколько пикселей. Координаты, задаваемые в методе построения, являются координатами центрального отрезка, поэтому пиксели отрезка располагаются около его центральной линии. Пока изображение отрезка размещается внутри полотна, он отображается полностью. Но если отрезок слишком близок к границе полотна, то он будет отображаться не полностью. Поэтому последний отрезок, построенный в процедуре, представленной в листинге 13.1, чертится не по самому краю, а с небольшим отступом, равным половине ширины отрезка.

Задача 13.2. В поле контейнера `TImage` требуется нарисовать две геометрические фигуры: эллипс и прямоугольник. Предусмотреть такую окантовку сторон окна `Image`, чтобы каждая сторона отличалась стилем, толщиной линии и цветом, как это показано на рис. 13.2.

Решение

Для решения задачи воспользуемся теми методами объекта `Canvas`, которые предоставляют для рисования набор графических примитивов. При построении примитивов помимо пера используются также параметры кисти, которые определяют, как должна заполняться внутренняя область примитива. Кисть `Brush` *имеет свойства*, необходимые для решения задач графики:

Color – цвет кисти; задается подобно цвету пера;
Style – стиль заполнения области (сплошное, крест и тому подобное):
- *bsSolid* – сплошная область;
- *bsClear* – прозрачная область;
- *bsCross* – пересекающимися горизонтальными и вертикальными линиями;
- *bsDiagCross* – линиями, пересекающимися по диагонали;
- *bsBDiagonal* – параллельными линиями с наклоном вправо;
- *bsFDiagonal* – параллельными линиями с наклоном вправо;
- *bsHorizontal* – параллельными горизонтальными линиями;
- *bsVertical* – параллельными вертикальными линиями.

Построение изображения посредством примитивов осуществляется так же, как изображение отрезками: задаются параметры пера и кисти, а затем вызывается соответствующий метод.

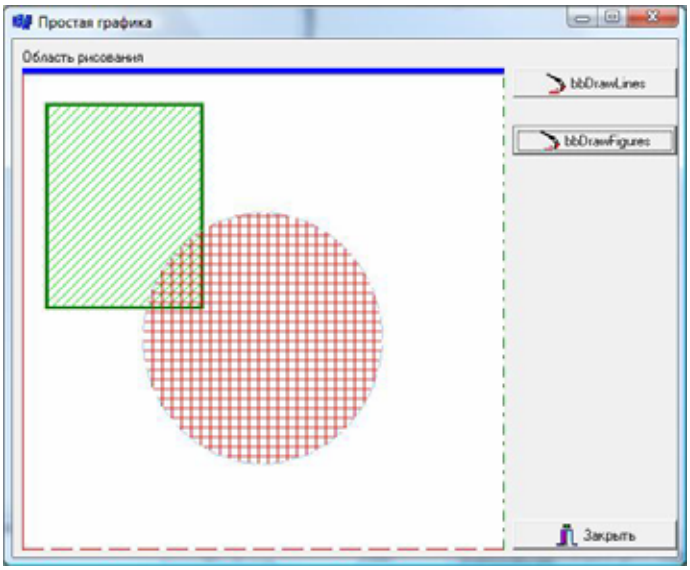


Рис. 13.2. Интерфейс приложения для простого рисования

Алгоритм решения задачи построения изображения, в соответствии с рис. 13.2, целесообразно реализовать в процедуре-обработчике события `OnClick` кнопки, например с именем `bbDrawFigures`. Код обработчика приведен в листинге 13.2.

Листинг 13.2. Рисование примитивов в окне `Image`

```
void __fastcall
             TfoSimpleGraph::bbDrawFiguresClick(TObject *Sender)
{
   // Рисование примитивов: прямоугольника и эллипса
   // определяем цвет, толщину или стиль контура
  Image->Canvas->Pen->Color = clGreen;   // изменить цвет пера
  Image->Canvas->Pen->Width = 3;  // изменить толщину пера
                // стиль нет смысла менять, так как толщина > 1
    // определяем цвет и стиль заливки
  Image->Canvas->Brush->Color = clLime;  // изменить цвет
  Image->Canvas->Brush->Style = bsBDiagonal;  // изменить стиль

    // чертим прямоугольник
  Image->Canvas->Rectangle (20, 30, 150, 200);

  Image->Canvas->Pen->Color = clSkyBlue;  // изменить цвет пера
  Image->Canvas->Pen->Width = 1;  // изменить толщину пера
  Image->Canvas->Pen->Style = psDash;  // изменить стиль пера
    // определяем цвет и стиль заливки
  Image->Canvas->Brush->Color = clRed;  // изменить цвет
  Image->Canvas->Brush->Style = bsCross;  // изменить стиль
    // чертим эллипс
  Image->Canvas->Ellipse (100, 120, 300, 330);
}
```

Напоминание

Конечно, координаты левой верхней или правой нижней точки примитива могут быть и отрицательными, и превышать ширину или высоту холста. Функции построения примитивов и отрезков строят эти изображения, но видна будет только та часть примитива, которая попадает в область холста.

Задача 13.3. Создать примитивный графический редактор, который позволял бы на холсте рисовать одну из двух геометрических фигур – эллипс или прямоугольник – с заданными размерами и положением, а также делать надписи на рисунке. Редактор должен позволять задавать толщину контура фигур и цвет как контура, так и заливки его внутренней области. Предусмотреть контроль на ошибку поля с числовыми данными.

Решение

Выполняя жесткую последовательность построения фигур с заданными координатами и параметрами, как показано при решении задач 13.1 и 13.2, программа может синтезировать лишь единственное изображение.

Задача, заявленная в условии, более интересна, поскольку в ней требуется создать приложение, представляющее собой графический редактор, хотя бы простейший.

Пример интерфейса такого приложения приведен на рис. 13.3. Отличительной особенностью такого редактора является наличие четырех окон однострочного редактора для задания координат *левой верхней* точки и *правой нижней* точки прямоугольника или охватывающего прямоугольника для эллипса. Пятое окно позволяет пользователю задать толщину линии контура объекта. Для задания цвета контура и заливки элемента, который будет построен, в данном приложении используются два компонента типа `TColorDialog` – стандартное диалоговое окно задания цвета. Для вызова этих окон используются две кнопки.

Используя эти семь параметров (четыре координаты, одну толщину и два цвета) по командам кнопок «Эллипс» и «Прямоугольник», программа будет строить соответствующую фигуру. Описание процедур-обработчиков для вызова одного из диалоговых окон цвета и построения фигуры прямоугольника приведены в листинге 13.3.

Листинг 13.3. Задание цвета и построение примитива «прямоугольник» в окне `Image`

```
void __fastcall
           TfoSimpleGraphEdit::btPenColorClick (TObject *Sender)
{   // вызов диалогового окна цвета пера
    // само окно и хранит цвет
  PenColorDialog->Execute();
}

void __fastcall
         TfoSimpleGraphEdit::btBrushColorClick (TObject *Sender)
{   // вызов диалогового окна цвета кисти
    // само окно и хранит цвет
  BrushColorDialog->Execute();
}

void __fastcall
```

```
          TfoSimpleGraphEdit::btRectangleClick (TObject *Sender)
{   // настройка параметров пера
  Image->Canvas->Pen->Color = PenColorDialog->Color;
  Image->Canvas->Pen->Width = StrToInt (edPenWidth->Text);
  Image->Canvas->Pen->Style = psSolid;
    // настройка параметров кисти
  Image->Canvas->Brush->Color = BrushColorDialog->Color;
  Image->Canvas->Brush->Style = bsSolid;

  Image->Canvas->Rectangle (StrToInt (edLeft->Text),
                            StrToInt (edTop->Text),
                            StrToInt (edRight->Text),
                            StrToInt (edBottom->Text));
}
```

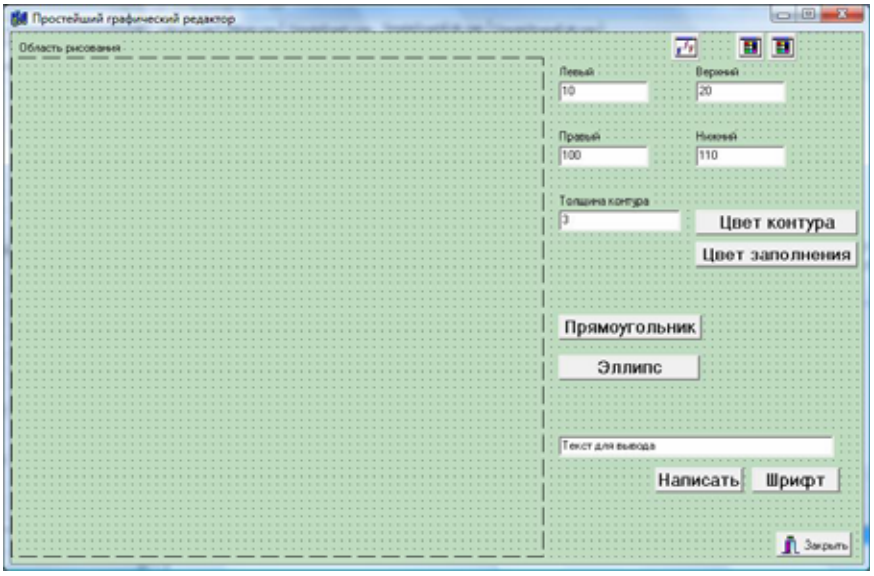


Рис. 13.3. Интерфейс простейшего графического редактора

Построение изображения надписи на холсте так же просто выполнить, как и изобразить примитив.

TextOut (X, Y, Text)

Эта процедура выводит строку текста, заданную параметром `Text`, а координаты `X` и `Y` определяют левый верхний угол области вывода. Параметры шрифта определяются текущим значением свойства `Font` объекта, на поверхности которого отображается текст.

Для задания нужного текста в данном приложении используется однострочный редактор, а сам текст выводится по нажатию кнопки. Чтобы приложение было более универсальным, то есть выводило текст различными шрифтами и цветом, здесь используется компонент `TFontDialog` – стандартное диалоговое окно выбора шрифта. Вызов этого окна осуществляется по отдельной кнопке. Вывод текста также выводится по нажатию кнопки. Обработчик этого события устанавливает выбранные параметры шрифта в холсте `TCanvas` объекта `TImage` и выводит надпись. Местоположение надписи определяется левой верхней точкой прямоугольника ох-

ватывающей надпись. Координаты этой точки в данном приложении берутся из двух соответствующих однострочных редакторов, используемых для этих же целей при построении примитивов. Описание процедуры вызова диалогового окна выбора шрифта и процедуры вывода текста приведены в листинге 13.4.

Листинг 13.4. Задание шрифта и текста в окне `Image`

```
void __fastcall TfoSimpleGraphEdit::btFontClick(TObject *Sender)
{   // вызов диалогового окна настройки шрифта
    // настройка шрифта в холсте
  FontDialog->Font = Image->Canvas->Font;
  if (FontDialog->Execute())
    Image->Canvas->Font = FontDialog->Font;
}

void __fastcall TfoSimpleGraphEdit::btWriteClick(TObject *Sender)
{   // написание заданного текста в заданной позиции,
    // шрифтом установленным на холсте
  Image->Canvas->TextOut (StrToInt (edLeft->Text),
                          StrToInt (edTop->Text),
                          edText->Text);
}
```



Обратите внимание, что вызовы диалоговых окон задания цвета и шрифта выполнены по-разному:

- диалоговые окна выбора цвета вызываются и только. После работы с ними пользователь закрывает это окно с помощью кнопки <*Ok*> или <*Cancel*>. И свойство этого компонента `Color` хранит выбранное пользователем новое значение или старое, установленное ранее. И это значение используется позже уже при выполнении операции рисования;
- для задания шрифта выбрана другая схема. Вызывается диалоговое окно. И если пользователь закрыл окно по кнопке <*Ok*>, то программа настраивает шрифт холста компонента `Image`.

Какая схема предпочтительнее – решать разработчику приложения. Здесь лишь показаны возможные варианты использования диалоговых окон. Тем не менее, мы получили законченное приложение простейшего редактора.



Остается одна сложность – правильное задание числовых значений, определяющих координаты. Для того чтобы текст в окне редактора был правильным числом и значение его было в допустимых пределах, необходимо иметь довольно-таки сложный компонент. Стандартная библиотека *C++ Builder* не содержит такого компонента. Другой путь – тщательно проверять введенный текст – является ли этот текст числом и находится ли это число в допустимом диапазоне. Такая проверка всех окон ввода координат и толщины линии контура должна осуществляться перед выполнением команды вычерчивания, которая использует эти значения.

Проверять все числовые окна в каждой процедуре построения – громоздкий процесс. С другой стороны, все окна ввода числовых значений данного приложения одинаковы.

Это наталкивает на идею ОСУЩЕСТВЛЯТЬ ПРОВЕРКУ ТЕКСТА *единой* процедурой и выполнять ее непосредственно *при выходе* из окна, то есть в обработчике события `OnExit`.

Примечание

В предлагаемой подпрограмме проверка на правильность будет осуществляться вызовом процедуры преобразования `StrToInt`. Поскольку вероятность ошибки при наборе текста числа пользователем очень высока, то данный оператор заключен в блок `try…catch`. Так как интересует, собственно, не значение, а лишь отсутствие ошибки, то значение никакой переменной не присваивается. Обработчику через параметр `Sender` передается адрес окна, вызвавшего его. Поэтому при наличии ошибки легко вернуть фокус именно этому окну редактирования, используя метод `SetFocus`. Такой принудительный возврат в окно редактирования накладывает определенную жесткость на поведение программы: пока не будет набрано число, пользователь *не сможет переместить курсор в другое окно*.

Следовательно, если пользователь переходит в другое окно редактирования или нажимает на кнопку, то он полагает, что в текущем окне уже все правильно написано. Текст такого обработчика события `OnExit` окон редактирования `Edit` приведен в листинге 13.5.

Проверка на диапазон значений

Сложнее обстоит ситуация с проверкой на диапазон значений. Чтобы правильно проверить диапазон, необходимо точно знать, что эти числа обозначают – значения по оси *0x* или по оси *0y*. Для этого достаточно проверить адрес окна редактирования: если выполняется проверка окон `edLeft` или `edRight`, то диапазон должен быть сравним с шириной, иначе – с высотой. Толщина контура примитива на диапазон здесь не проверяется, хотя такая проверка тоже должна бы быть. При проверке диапазона значений в данном проекте не применяется жесткий возврат фокуса окну редактирования, а лишь выдается окно сообщения. То есть пользователь получает предупреждение, что введенное значение выходит за пределы компонента `Image`, а из этого следует, что примитив будет видно частично или, вообще, будет отображен вне холста, то есть не виден.

Листинг 13.5. Обработчик события `OnExit` окон редактирования `TEdit`

```
void __fastcall TfoSimpleGraphEdit::edEditExit(TObject *Sender)
{
  TEdit *ptr = (TEdit *)Sender;
                          // чтобы каждый раз не переобозначать
  char tmp [100] = "Значение должно быть положительное и менее";
  int aux;
  try {
        aux = StrToInt (ptr->Text);  // попытка конвертировать
      }
  catch (EConvertError &E) // ошибка конвертирования
  {
     ShowMessage ("Введите целое число ...");
     ptr->SetFocus ();  // установка фокуса на окно с ошибкой
     return;
  };

  if (Sender == edLeft || Sender == edRight)
```

```
    if (aux < 0 || aux > Image->Width)
      ShowMessage (StrCat (tmp,
                           IntToStr (Image->Width).c_str ()));
  if (Sender == edTop || Sender == edBottom)
    if (aux < 0 || aux > Image->Height)
      ShowMessage (StrCat (tmp,
                           IntToStr (Image->Height).c_str ()));
}
```



Подобное простое рисование можно выполнять не только с помощью методов холста `TCanvas`, но и используя соответствующие готовые компоненты, например `TShape`. Такое рисование можно выполнять по форме или по панели. Рисование заключается в динамическом создании объекта типа и задании ему соответствующих значений: координат положения, размера и параметров отображения.

Задача 13.4. Создать примитивный графический редактор, который позволял бы в требуемом месте *указанной панели* рисовать одну из двух геометрических фигур – эллипс или прямоугольник – с заданными свойствами. Предусмотреть контроль на ошибку поля с числовыми данными.

Решение

Реализация данного приложения во многом схоже с решением предыдущей задачи, но имеет принципиальные отличия:

- рисование не по холсту, а по панели;
- фактически выполняется не рисование, а создание объекта;
- не выполняется вывод текста.

Интерфейс приложения на стадии проектирования и пример работы приложения приведены на рис. 13.4.

Как отмечалось выше для рисования фигур можно воспользоваться компонентом `TShape` с готовыми графическими формами. У компонента `TShape` имеется свойство `Shape`, которое может принимать одно из значений:

stRectangle	– прямоугольник;
stSquare	– квадрат;
stRoundRect	– округлый прямоугольник;
stRoundSquare	– округлый квадрат;
stEllipse	– эллипс;
stCircle	– круг.

Установить соответствующие цвета и стили выбранной фигуры можно с помощью таких свойств компонента `TShape`, как `Pen` и `Brush`. Размеры фигуры соответствуют размерам компонента: ширине и высоте. Если заданы формы «квадрат» или «круг», то размеры квадрата и круга равны меньшей стороне компонента. Алгоритм построения геометрической фигуры посредством объекта можно свести к выполнению следующих действий:

1) создать объект класса `TShape`;
2) задать в качестве собственника форму;

3) определить родителя для созданного объекта – панель;
4) установить вид геометрической фигуры;
5) установить координаты и размер, взяв для этого соответствующие данные из окон редакторов;
6) задать цвет контура и заливки объекта.

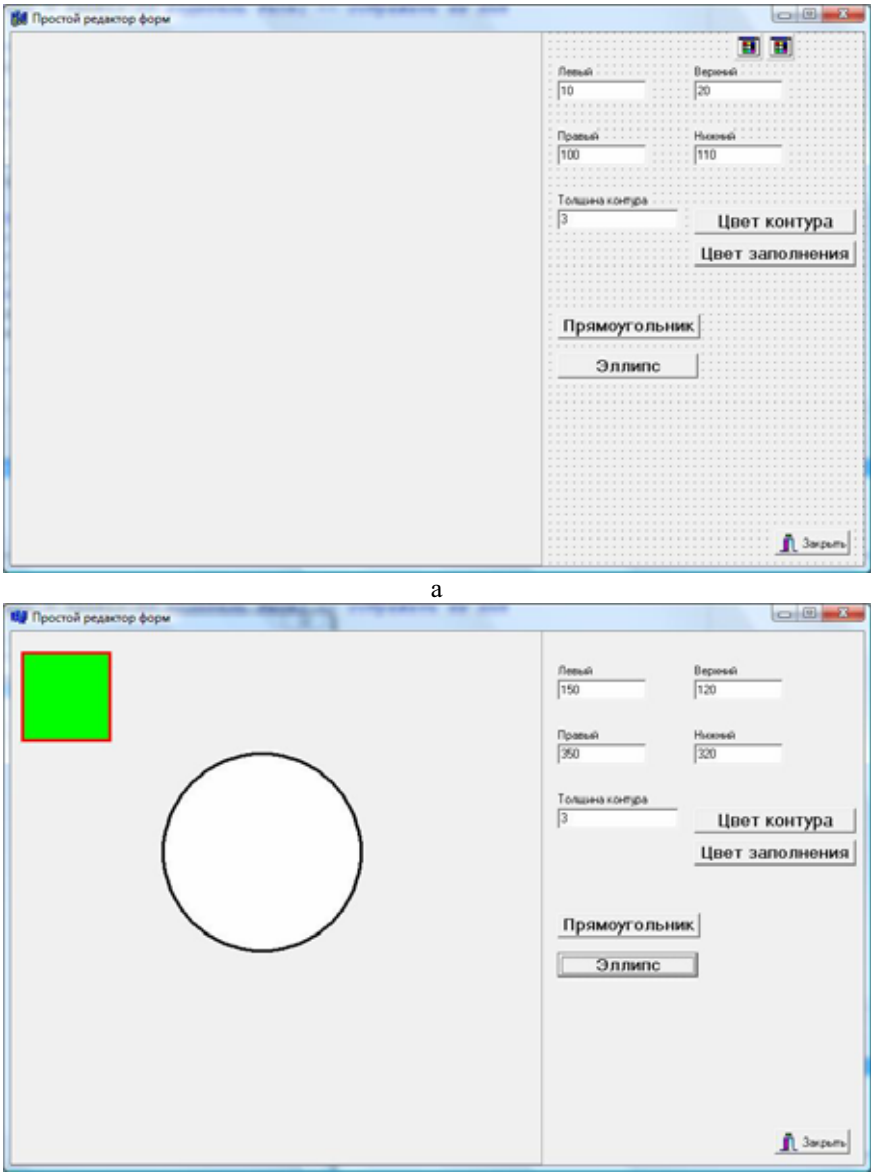


Рис. 13.4. Интерфейс простейшего графического редактора из объектов TShape: а) проект формы; б) пример работы приложения

Решение задачи – создание соответствующего компонента, аналогично предыдущей задачи, привязаны к событию `OnClick` соответствующей кнопки, создающей объект и формирующей форму и окантовку изображения компонента, изменяя свойства `Pen` и `Brush`. Объект привязывается к панели и становится полноправным элементом формы. Проверка значений в окнах редактирования полностью аналогична решению предыдущей задачи. Отличие лишь в проверке диапазона – теперь сравнение выполняется с размерами панели. Решения задачи порождения объекта и кон-

троля значений в окнах однострочных редакторов можно записать с помощью следующих операторов, как показано в листинге 13.6.

Листинг 13.6. Рисование прямоугольника объектом TShape по панели

```
void __fastcall
           TfoSimpleShapeEdit::btRectangleClick (TObject *Sender)
{ TShape *ptr = new TShape (foSimpleShapeEdit);
                                  // собственник элемента форма
  ptr->Parent = Panel; // родитель Panel -- отображать на ней

    // настройка формы
  ptr->Shape = stRectangle;

    // настройка положения и размеров
  ptr->Left = StrToInt (edLeft->Text);
  ptr->Top = StrToInt (edTop->Text);
  ptr->Width = StrToInt (edRight->Text) -
               StrToInt (edLeft->Text);
  ptr->Height = StrToInt (edBottom->Text) -
                StrToInt (edTop->Text);

    // настройка параметров пера
  ptr->Pen->Color = PenColorDialog->Color;
  ptr->Pen->Width = StrToInt (edPenWidth->Text);
  ptr->Pen->Style = psSolid;
    // настройка параметров кисти
  ptr->Brush->Color = BrushColorDialog->Color;
  ptr->Brush->Style = bsSolid;
}
```



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. Разработать программу, позволяющую рисовать, сохранять нарисованное изображение в виде файла, читать файлы с изображениями и корректировать.
2. Доработать примитивный редактор, созданный для *задачи 13.4*, дополнив этот проект элементом `TComboBox`, в котором пользователь смог бы выбирать тип заливки.
3. Дополнить интерфейс примитивного редактора, созданный для *задачи 13.4*, двумя компонентом `TShape`, на котором бы отражались цвет контура и заливки, выбранные пользователем.
4. Доработать проект решения *задачи 13.3* четырьмя кнопками. Реализовать перемещение последнего созданного компонента по панели. Для запоминания этого компонента можно использовать глобальную переменную или поле формы.
5. Доработать проект решения *задачи 13.3* четырьмя кнопками. Реализовать изменение ширины и высоты последнего созданного компонента.
6. Доработать проект решения *задачи 13.4*. Дополнить проект кнопками, создающими на панели объекты `TShape` других видов.

Применение технологии Drag&Drop для работы пользователя с интерфейсом программы 14

Напомним, что интерфейс операционной системы *Windows* имеет графический формат. Отсюда следует, что все элементы от рабочего стола до кнопок на форме – это *графические изображения* специального вида. Но при разработке программы в среде *C++ Builder* о кнопке и других элементах интерфейса приходится думать как о законченном объекте, который еще и способен отображаться на рабочей области формы. В главе 13 показано как можно организовать интерфейс приложения, чтобы предоставить пользователю возможность редактировать графические изображение, используя, в основном, только манипулятор «мышь».

Но использование «мыши» при работе с текстовыми компонентами окна программы может сильно увеличить удобство работы с программой. Разработчик в разрабатываемом приложении может предоставить пользователю возможность «перетаскивать» информацию между элементами формы, например отдельные строки из списка в окно редактирования или в другой компонент, заменяя этим единственным движением серию команд на копирование в «карман» и обратно. Такие возможности разработчику приложения предоставляет технология *Drag&Drop*.

Суть технологии *Drag&Drop*

Суть процесса *Drag&Drop* заключается в следующем:

1. Отправление сообщений конкретному компоненту-источнику о том, что пользователь начинает «перетаскивать» его мышью.
2. Отправление сообщений каждому компоненту, над которым пользователь «тащит» компонент-источник о том, что он это делает.
3. Отправление сообщений компоненту-приемнику, что на него «бросают» компонент-источник.
4. Отправление сообщений компоненту-источнику о том, что его «бросают» на компонент-приемник.

Внимание

При «перетаскивании» компонентов пользователем визуального «перемещения» компонента-источника по форме *не происходит*. Просто операционная система и приложение *работают в особом режиме*. Внешне эти ситуации, как правило, проявляются лишь в изменении вида курсора «мыши».

Элементы технологии *Drag&Drop* изнутри

При использовании готовых программ, в которых применена технология *Drag&Drop*, легко понять, что внутри приложения реализованы процедуры, *обеспечивающие взаимодействие двух элементов интерфейса*. Причем перемещение «мыши» от одного компонента к другому программа понимает и выполняет однозначно. Для правильного проектирования и разработки программного кода, реализующего технологию *Drag&Drop*, необходимо понимать, какие события происходят с каждым «участником» процесса, и как он должен реагировать, если такое событие произошло.

Напоминание

Для полноты изложения напомним, что *у формы и почти у каждого компонента формы* в среде *C++ Builder* имеются свойства и обработчики события, связанные с технологией *Drag&Drop*.

- Свойства:
  - `DragCursor` – определяет вид курсора при перемещении курсора над объектом;
  - `DragKind` – определяет тип перетаскивания *Drag&Drop* или *Drag&Dock*;
  - `DragMode` – определяет тип начала процесса перетаскивания: автоматически или программно.
- Обработчики событий:
  - `OnDragDrop` – отпускание над данным компонентом другого;
  - `OnDragOver` – перетаскивание над данным компонентом другого;
  - `OnStartDrag` – начало перетаскивания компонента;
  - `OnEndDrag` – окончание перетаскивания компонента.

Полный перечень названий элементов, конечно, не дает понимания того, как они связаны между собой и как они «работают в команде» для решения проблемы. Необходимо понимать, что простота работы пользователя – это лишь «верхушка айсберга», и обеспечивается эта простота сложными процессами и вычислениями, которые выполняет приложение.

Технология *Drag&Drop* со стороны пользователя

Со стороны пользователя применение технологии *Drag&Drop* выглядит следующим образом:

а) пользователь нажимает кнопку «мыши» на одном элементе;
б) удерживая кнопку в нажатом положении, пользователь «тащит» элемент на другой элемент;
в) когда курсор «мыши» оказывается над вторым элементом, пользователь отпускает кнопку «мыши» – элемент на новом месте.

Технология *Drag&Drop* со стороны программы

Со стороны программы при «перетаскивании» информации или компонента на другой компонент также взаимодействуют два элемента. При этом активное участие принимает *операционная система как посредник*, которая формирует целый ряд событий, на которые реагирует наша программа. По-другому выражаясь, за время одного движения руки пользователя операционная система передает программе несколько сообщений, которые программа и обрабатывает. Для полноты можно также упомянуть и о множестве других элементов формы, которые задействуются, когда над ними «протаскивают» курсор в режиме *Drag&Drop*. Просто обработчики этих элементов посылают информацию, что «бросать» на них нельзя, они «проигнорируют» это действие.

Порядок событий

Если свойство `DragMode` «источника» и «приемника» установлено в значение `dmAutomatic`, то все события, связанные с технологией, формируются системой автоматически. В порядке возникновения такие события можно перечислить так:

1) пользователь нажимает кнопку «мыши» и начинает «тащить» объект «источник»;
2) для «источника» возникает событие `StartDrag` – курсор «мыши» видоизменяется;
3) для «приемника» возникает событие `DragOver`, если курсор «мыши» перемещается над поверхностью «приемника»;
4) когда пользователь отпускает кнопку «мыши», у «источника» возникает событие `EndDrag`, а у «приемника» – `DragDrop`.

ТРУДНОСТИ

Если свойство `DragMode` имеет значение `dmManual`, то обработка команд «мыши» несколько усложняется. В этом случае разработчику приходится «запускать процесс вручную», то есть вызывать метод `BeginDrag`. Такой вариант реализации технологии предлагается опробовать читателю *самостоятельно*.

Задача 14.1. Требуется разработать фрагмент приложения. В этом приложении следует реализовать возможность перетаскивания строки текста из списка строк `TListBox` в окно однострочного редактора `TEdit`.

Стратегия решения задачи

Для правильного решения задачи необходимо лишь правильно настроить параметры компонентов и подключить обработчики соответствующих методов. В данном случае для компонента `ListBox`, объекта «источника», необходимо установить в свойстве `DragMode` значение `dmAutomatic`. Для компонента `Edit`, объекта «приемника», необходимо установить:

1) в свойстве `DragMode` значение `dmAutomatic`;
2) к указателям событий `OnDragOver` и `OnDragDrop` – соответствующие обработчики.

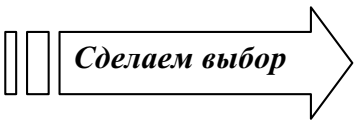
Как и многие другие, эта задача имеет несколько вариантов решения. Каждый вариант имеет свои достоинства и недостатки. Разные приложения могут по-иному решать задачу переноса информации в ответ на одно движение руки пользователя между компонентами формы. Стратегии решения здесь могут быть следующие:

1) распределение необходимых действий между несколькими обработчиками:
   а) обработчик «источника» `OnListBoxStartDrag` должен решить задачу, какую именно строку начинает перетаскивать пользователь и сохранить такую информацию необходимо до окончания операции «перетаскивания»;
   б) обработчик «приемника» `OnEditDragOver` должен регулировать вид курсора, чтобы он ясно указывал пользователю, можно сюда «бросать информацию» или нет;

в) обработчик же `OnEditDragDrop` должен убедиться, что «перетаскивание» выполнено корректно, и произвести действия, необходимые для завершения операции копирования.

К недостатку такого решения можно отнести то, что для хранения информации необходимо привлекать дополнительные ресурсы, например глобальные переменные. Но в других приложениях может возникнуть ситуация, что задача вычленения переносимой информации разрешима только на этапе старта процесса.

2) Сосредоточение необходимых действий в одном обработчике:
   а) обработчик «приемника» `OnEditDragOver` регулирует вид курсора;
   б) обработчик же `OnEditDragDrop` должен убедиться, что «перетаскивание» выполнено корректно, и произвести все действия, необходимые в случае операции копирования.



Остановим выбор на второй стратегии.

Помимо этого, реализации могут иметь различные варианты завершения процесса перетаскивания:

- ✓ все завершающие действия выполняются только в обработчике «приемника» `OnEditDragDrop`;
- ✓ все завершающие действия выполняются только в обработчике «источника» `OnListBoxEndDrag`;
- ✓ завершающие действия выполняются частично в обработчике «приемника» `OnEditDragDrop`, а частично – в обработчике «источника» `OnListBoxEndDrag`. При этом необходимо помнить, что вызов обработчиков выполняется последовательно: а) `OnEditDragDrop`; б) `OnListBoxEndDrag`.

Решение

Предположим, что интерфейс приложения содержит следующие элементы (рис. 14.1):

1) список строк `TListBox`;
2) однострочный редактор `TEdit`;
3) кнопка `TBitButton`, со значением `bkClose` в поле `Kind`;
4) две метки `TLabel` с соответствующими текстами.

Как уже оговаривалось выше, решение данной задачи заключается:

1) в присваивании значения `dmAutomatic` полям `DragMode` компонентов `ListBox` и `Edit`;
2) в разработке соответствующих обработчиков событий для компонента `Edit`.

Рассмотрим два варианта решения задачи.

Вариант 1. Решение задачи посредством обработчика `OnDragDrop` компонента «приемника».

Поскольку при выборе стратегии нами сделан выбор в пользу 2-й стратегии, решение задачи по «перетаскиванию» текста должно быть сосредоточено в двух обработчиках событий: `OnDragOver` и `OnDragDrop` компонента «приемника» `Edit`.

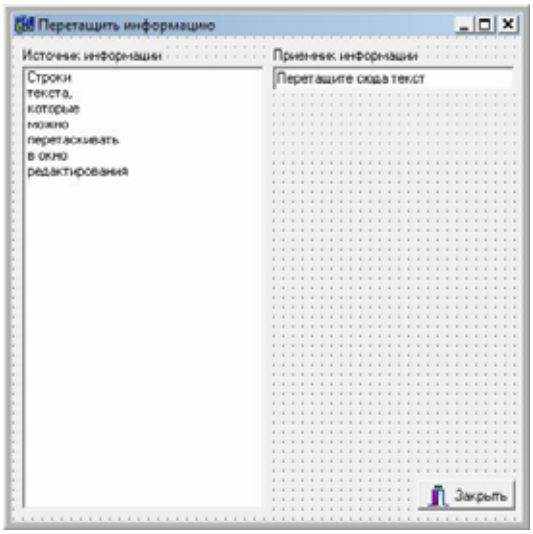


Рис. 14.1. Интерфейс программы «перетаскивания» информации

СОБЫТИЕ OnDragOver

Событие `OnDragOver` возникает для объекта «приемника», если курсор мыши перемещается над поверхностью «приемника». Эта процедура обработки события имеет 6 параметров [2]:

- `Sender` – передается указатель на объект, который сгенерировал событие, над которым «перемещают» другой объект;
- `Source` – передает указатель на объект, который «перемещают»;
- `X`, `Y` – координаты «мыши» в отсчетах объекта «приемника»;
- `State` – состояние перетаскиваемого объекта по отношению к «приемнику»: объект вошел в зону «приемника», перемещается над ним или выходит из зоны;
- `Accept` – возвращаемый параметр; должен иметь значение `true`, если «приемник» готов принять другой объект, иначе – `false`.

СОБЫТИЕ OnDragDrop

Событие `OnDragDrop` возникает для объекта «приемника», когда пользователь отпускает кнопку мыши. Процедура обработки события имеет 4 параметра:

- `Sender` – передается указатель на объект, на который «бросили» другой объект, и он сгенерировал событие;
- `Source` – передает указатель на объект, который был «брошен» на данный, указанный в `Sender`;
- `X`, `Y` – координаты «мыши» в отсчетах объекта «приемника».

Обработчик события `OnDragOver` для компонента `Edit`

Обработчик `OnDragOver` здесь ограничивается лишь проверкой типа источника.

Текст реализации этого обработчика события `OnDragOver` для компонента `Edit` приведен в листинге 14.1.

Листинг 14.1. Обработчик события `OnDragOver` для компонента `Edit`

```
void __fastcall TfoDragInfo::EditDragOver (TObject *Sender,
```

```
                              TObject *Source, int X, int Y,
                              TDragState State, bool &Accept)
{
   Accept = true;
}
```

Обработчик события `OnDragDrop` для компонента `Edit`

Обработчик `OnDragDrop` выполняет операцию присваивания.

Текст реализации этого обработчика события `OnDragDrop` для компонента `Edit` приведен в листинге 14.2.

Листинг 14.2. Обработчик события `OnDragDrop` для компонента `Edit`

```
void __fastcall TfoDragInfo::EditDragDrop (TObject *Sender,
                               TObject *Source, int X, int Y)
{
   Edit->Text =
            ListBox->Items->Strings [ListBox->ItemIndex];
}
```



Серьезная проблема состоит в том, что созданное приложение делает еще и те операции, которые при проектировании НЕ ЗАКЛАДЫВАЛИСЬ. И порой эти операции завершаются сообщениями об ошибках. Основные ошибочные действия данного приложения:

1) обработчик `EditDragDrop` срабатывает, даже если просто щелкнуть «мышью» по строке редактирования `TEdit`. Во время работы приложения это доставляет «забавную» неприятность: выделенная в текущий момент строка списка переносится в окно редактора. Если щелчок «мышью» выполнить сразу после старта, то приложение выдаст сообщение об ошибке «индекс *в списке не определен*», что соответствует истине.
2) Если информацию из строки редактирования «потаскать» над формой, а затем «бросить обратно», то будет такая же реакция приложения, как и в пункте 1).



Все это лишний раз напоминает, что после реализации приложение должно тестироваться и на действия пользователя, не предполагавшиеся разработчиком. И это необходимо учитывать еще при разработке: блокировать реакцию программы на недопустимые действия.

Конечно, в данном приложении есть несколько вариантов для исправления неправильной реакции программы.

1. Напомним, что `Edit` должен принимать информацию из списка `ListBox`, но не из других мест, в том числе и самого себя. Поэтому обработчик `EditDragDrop` должен анализировать параметр `Source` и предпринимать такие действия, чтобы не возникало ошибки. Типовой проверкой в таком обработчике является проверка на *тип объекта*, который присылает информацию.
2. Более того, также возможно идентифицировать конкретный объект формы, например по его *имени* в проекте. Хотя можно использовать и другие параметры.

3. Также можно более тщательно анализировать параметр `Source` в обработчике `EditDragOver`, чтобы просто не допускать вызова обработчика `EditDragDrop`.
4. Такой анализ выполняется на каждое движение «мыши» над «приемником», и поэтому он должен быть достаточно простым и коротким.
5. Такой анализ при «перетаскивании» и «бросании», может потребоваться для того, чтобы точно идентифицировать, что инициатором процесса стал объект, которому разрешено «сбрасывать» информацию в этот «приемник».
6. Более того, «сбрасывать» информацию в этот «приемник» может быть позволено нескольким объектам. А выполнять «вытаскивание» информации из источников разного типа необходимо различными процедурами.

Пример такого анализа приведен в листингах 14.3 и 14.4.

Обработчик события `OnDragOver` для компонента `Edit` с элементом дополнительного анализа

Обработчик события `OnDragOver`, представленный в листинге 14.3, в отличие от обработчика, представленного в листинге 14.1, не ограничивается лишь проверкой типа источника – в нем проверяется, является ли источник компонентом `ListBox`.

Листинг 14.3. Вариант обработчика события `OnDragOver` для компонента `Edit` с анализом объекта источника информации

```
void __fastcall TfoDragInfoWAnalize::EditDragOver
                      (TObject *Sender, TObject *Source,
             int X, int Y, TDragState State, bool &Accept)
{
   Accept = Source->ClassNameIs ("TListBox");
}
```

Обработчик события `OnDragDrop` для компонента `Edit` с элементом дополнительного анализа

Обработчик события `OnDragDrop`, представленный в листинге 14.4, в отличие от обработчика, представленного в листинге 14.2, проверяет тип и имя источника, а также тип «приемника» и лишь потом выполняет операцию.

Листинг 14.4. Вариант обработчика события `OnDragDrop` для компонента `Edit` с анализом объекта источника информации

```
void __fastcall TfoDragInfoWAnalize::EditDragDrop
                      (TObject *Sender, TObject *Source,
                       int X, int Y)
{
  if (Source->ClassNameIs ("TListBox"))
                          // идентификация типа источника
    if (((TListBox *)Source)->Name == "ListBox")
                                        // иден-ция имени
      if (Sender->ClassNameIs ("TEdit"))
                                // иден-ция типа приемника
        Edit->Text = ((TListBox *)Source)->Items->
                 Strings [((TListBox *)Source)->ItemIndex];
}
```



Идентифицировать «приемник» требуется обязательно, если один и тот же обработчик `DragDrop` подключается к нескольким объектам-«приемникам».

Вариант 2. Решение задачи посредством обработчика «приемника» `EndDrag`.

Для полноты изложения приведем пример решения этой же задачи посредством обработчика-«приемника» `EndDrag`. Как и в первом варианте, для решения задачи необходимо подготовить два обработчика событий:

1. Обработчик события `OnDragOver` для компонента `Edit`.
2. Обработчика `ListBoxEndDrag` для компонента `ListBox`.

В действия обработчика события `OnDragOver` для компонента `Edit` можно включить элементы анализа объекта – источника информации. Описание обработчика события приведено в листинге 14.5.

Обработчик события `OnDragOver` для компонента `Edit`

Обработчик события `OnDragOver` не ограничивается проверкой типа источника – в нем проверяется, является ли источник компонентом `ListBox`. Текст процедуры обработчика приведен в листинге 14.5.

Листинг 14.5. Вариант обработчика события `OnDragOver` для компонента `Edit` с анализом объектов источника информации

```
void __fastcall TfoDragInfoByEndDrag::EditDragOver
                          (TObject *Sender, TObject *Source,
               int X, int Y, TDragState State, bool &Accept)
{
   Accept = Source->ClassNameIs ("TListBox");
}
```

А текст обработчика `ListBoxEndDrag` для данного случая будет почти идентичен обработчику `EditDragDrop`. Отличие состоит в том, что теперь адрес «источника» передается через параметр `Sender`, а «приемника» – через параметр `Target`.

СОБЫТИЕ OnEndDrag

Событие `OnEndDrag` возникает для объекта, который перетаскивают, когда пользователь отпускает кнопку «мыши», то есть по завершении процесса. Процедура обработчик события имеет 4 параметра:

- `Sender` – передается указатель на объект, который подвергался перетаскиванию и сгенерировал событие;
- `Target` – передается указатель на «приемник»;
- `X`, `Y` – координаты мыши в координатах «приемника»; или (0, 0), если в параметре `Target` значение `NULL`.

Второе, и довольно таки существенное отличие, – в первом операторе обработчика `ListBoxEndDrag`. Чтобы графический курсор «мыши» правильно отражал запрет на право «сбрасывания» информации, достаточно проверять, можно ли это делать, и если нет, то устанавливать значение параметра `Accept` в `false`.

Если OnDragOver возвращает параметр Accept со значением, равным false, то событие OnDragDrop «приемника» *не* возникает, а событие OnEndDrag «источника» возникает. Вот здесь-то и возникает конфликт: обработчик вызывается, а был запрет. В этом случае обработчик OnEndDrag получает значение NULL через параметр Target, то есть «приемник» информации неизвестен. Поэтому, чтобы не возникало ошибок, предварительно необходимо проверить значение параметра Target на значение NULL. И если значение указателя действительно есть NULL, то выполнять другие действия нельзя, и просто выполняется выход из подпрограммы.

Обработчик события OnEndDrag для компонента ListBox

Обработчик события OnDragDrop проверяет, был ли запрет на право «сбросить» информацию, проверяет тип и имя «источника», а также тип «приемника» и лишь потом выполняет операцию. Описание обработчика представлено в листинге 14.6.

Листинг 14.6. Вариант обработчика события OnEndDrag для компонента ListBox с анализом объектов «источника» и «приемника» информации

```
void __fastcall TfoDragInfoByEndDrag::ListBoxEndDrag
           (TObject *Sender, TObject *Target, int X, int Y)
{ if (Target == NULL) return;
  if (Sender->ClassNameIs ("TListBox"))
                              // идентификация типа источника
    if (((TListBox *)Sender)->Name == "ListBox")
                                          // иден-ция имени
      if (Target->ClassNameIs ("TEdit"))
                                // иден-ция типа приемника
        Edit->Text = ((TListBox *)Sender)->Items->
                 Strings [((TListBox *)Sender)->ItemIndex];
}
```

Обмен информацией между окнами

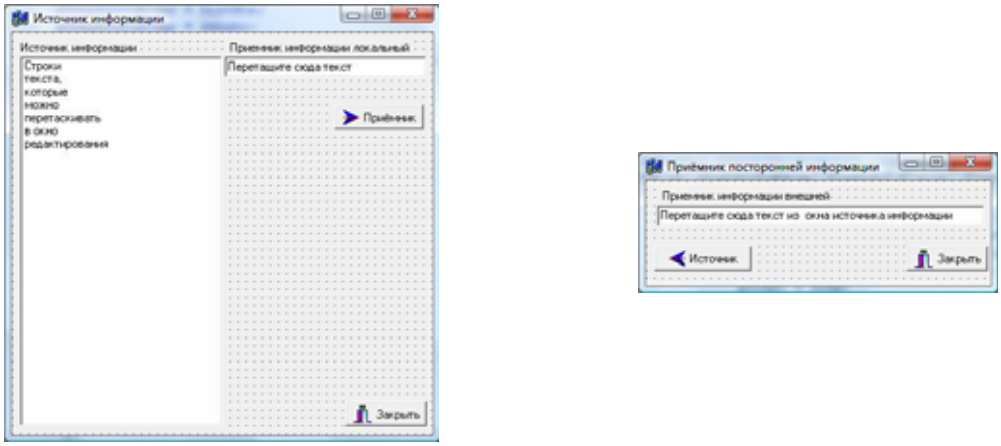
Возможности современных компьютерных систем быстро возрастают и уже часто встречаются не только двухъядерные, но и системы с двумя мониторами. Крупное приложение, как правило, имеет несколько окон, которые пользователь для удобства работы может располагать на разных дисплеях. Отдельное окно может отображать картографическую информацию, а другие окна – палитру инструментов и параметры. В таком приложении пользователю может потребоваться передавать информацию из одного окна в другое. Поэтому было бы логично, помимо других возможностей, предоставлять пользователю возможность «перетаскивать» информацию между окнами посредством «мыши».

Среда программирования *C++ Builder* позволяет организовать такой обмен информацией между окнами приложения посредством «мыши» так же легко, как и между компонентами одного окна.

Задача 14.2. Требуется разработать фрагмент приложения, состоящего из двух окон. В этом приложении следует реализовать возможность перетаскивания строки текста из списка строк TListBox, располагающегося на одной форме, в окно однострочного редактора TEdit, находящегося на другой форме.

Решение

Интерфейс окон такого приложения может выглядеть, как на рис. 14.2. Для быстрого открытия окна-приемника или окна-источника на каждую форму добавлена соответствующая кнопка.



а б

Рис. 14.2. Интерфейс программы «перетаскивания» информации между окнами приложения: а) окно с источником информации – списком строк; б) окно с приемником информации – однострочным редактором

Рассуждения, проведенные для технологии *Drag&Drop*, выполняемой между компонентами одной формы, справедливы и здесь. Так как приложение имеет единое адресное пространство, то адреса объектов, передаваемые через параметры, правильно идентифицируют компоненты «источник» и «приемник» и не зависят от того, «перетаскивается» ли информация внутри одного или между двумя окнами.

Усложнение контроля может потребоваться лишь по той причине, что «перетаскивание» может выполняться и с другой формы. Поэтому обработчикам `OnDragOver` и `OnDragDrop` «приемника» и обработчику `OnEndDrag` «источника» для полной идентификации *приходится анализировать не только компонент, но и форму*, которой он принадлежит.

Текст реализации обработчиков для однострочного редактора `TEdit` формы «приемника» информации приведен в листинге 14.7. Реализация обработчиков событий для всех компонентов формы «источника» информации не приведено по той причине, что они такие же, как в листинге 14.2. И обмен информацией между компонентами формы «источника» также выполняется. Но помимо этого информацию можно переносить и на форму «приемник».

Обработчик события `OnClick` для кнопки `bbSource`

Листинг 14.7. Вариант обработчиков событий `OnClick` для кнопки `bbSource` формы, принимающей информацию

```
void __fastcall TfoDragInfoTarget::bbSourceClick (TObject *Sender)
{
   foDragInfoSource->Show();    // показать форму
}
```

Обработчик события `OnDragOver` для компонента `Edit`

Листинг 14.8. Вариант обработчиков событий `OnDragOver` для компонента `Edit` формы, принимающей информацию

```
void __fastcall TfoDragInfoTarget::EditDragOver
                          (TObject *Sender, TObject *Source,
               int X, int Y, TDragState State, bool &Accept)
{
  Accept = false;
  if (Source->ClassNameIs ("TListBox"))
                             // идентификация типа источника
    if (((TListBox *)Source)->Name == "ListBox")
                                                 // иден-ция имени
      if (((TListBox *)Source)->Parent->
                         ClassNameIs ("TfoDragInfoSource"))
                        // иден-ция типа родителя источника
        if (Sender->ClassNameIs ("TEdit"))
                                 // иден-ция типа приемника
          Accept = true;
}
```

Обработчик события `OnDragDrop` для компонента `Edit`

Листинг 14.9. Вариант обработчиков событий `OnDragDrop` для компонента `Edit` формы, принимающей информацию

```
void __fastcall TfoDragInfoTarget::EditDragDrop
                          (TObject *Sender, TObject *Source,
                           int X, int Y)
{ if (Source->ClassNameIs ("TListBox"))
                             // идентификация типа источника
    if (((TListBox *)Source)->Name == "ListBox")
                                                 // иден-ция имени
      if (Sender->ClassNameIs ("TEdit"))
                                 // иден-ция типа приемника
        Edit->Text = ((TListBox *)Source)->Items->
                  Strings [((TListBox *)Source)->ItemIndex];
}
```

Примечание

В тексте обработчика `EditDragOver` приведен пример контроля на тип объекта-родителя компонента `TListBox`. В реальном проекте, конечно, проверки в этих обработчиках могут быть и другими. Критерием правильности контроля является ответ на вопрос: передается ли информация с требуемых компонентов и форм и не передается с других?

Перемещение компонентов по форме

Посредством технологии *Drag&Drop* можно организовывать не только обмен информацией между компонентами формы, но и выполнять действия непосредственно над компонентами, например удалять с формы, делать невидимым, перемещать по форме и тому подобное. Возможность включать и выключать какие-то параметры компонента при перетаскивании его на другой компонент реализуется аналогично приведенным выше примерам. Просто оператор присваивания текстовой информации в этих процедурах необходимо заменить на

оператор воздействия на соответствующее свойство компонента. Например, можно отключить видимость объекта, присвоив свойству `Visible` значение `false`.

Аналогичным образом можно организовать и перемещение компонентов по форме или по отдельной панельке на форме. Наличие такой возможности в приложении позволит пользователю настраивать местоположение компонентов на форме на свой вкус, дабы повысить эффективность работы с приложением. Естественно, что такое приложение разработчик должен дополнить возможностью сохранения нового положения компонентов, а также процедурой чтения и восстановления такого измененного положения компонентов.

Задача 14.3. Требуется разработать приложение, состоящее из формы на которой расположены компонент `TShape`, два компонента `TImage` и две кнопки <*Восстановить*> и <*Закрыть*>. В этом приложении реализовать следующие возможности:

1) перетаскивания компонента `TShape` по форме;
2) исчезновения его при попадании на один из компонентов `TImage`;
3) запрета на выполнение операции *Drag&Drop* при попадании на другой компонент `TImage`;
4) восстановления видимости и местоположения компонента `TShape` по кнопке <*Восстановить*>.

Решение

Примерный вид интерфейса такого приложения и план его реализации указаны в условиях задачи (см. рис. 14.3). Основные отличия такого проекта:

1) рабочей областью является вся клиентская часть формы и поверхности других компонентов;
2) требуется правильно вычислять координаты нового положения компонента на форме.

Первый пункт не привносит больших новшеств в решение. Просто «приемником» здесь становится объект класса `TForm`, поэтому обработчики событий `OnDragOver` и `OnDragDrop` разрабатываются для него. А вот второй пункт требует пополнения кода процедур несложной арифметикой и операторами опроса и задания положения компонента. Тесты таких обработчиков приведены в листинге 14.10.

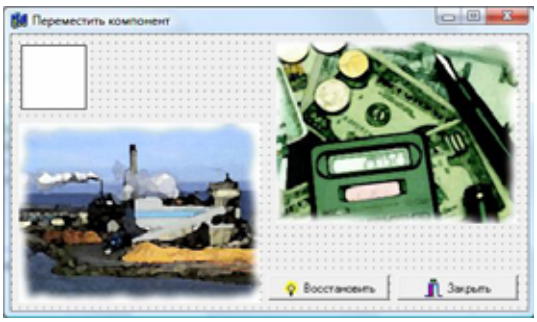


Рис. 14.3. Интерфейс программы перемещения компонента по форме

Обработчик события `OnClick` для кнопки `bbRetry`

Листинг 14.10. Вариант обработчиков событий для манипулирования компонентами на форме

```
void __fastcall TfoDragComponent::bbRetryClick (TObject *Sender)
{ Shape->Show();
```

```
  Shape->Top = 10;
  Shape->Left = 10;
}
```

Обработчик события `OnDragOver` для компонента `ImageOff`

Листинг 14.11. Вариант обработчиков событий `OnDragOver` для компонента `ImageOff` для манипулирования компонентами на форме

```
void __fastcall TfoDragComponent::ImageOffDragOver
                          (TObject *Sender, TObject *Source,
              int X, int Y, TDragState State, bool &Accept)
{ Accept = Source->ClassNameIs ("TShape");
}
```

Обработчик события `OnDragDrop` для компонента `ImageOff`

Листинг 14.12. Вариант обработчиков событий `OnDragDrop` для компонента `ImageOff` для манипулирования компонентами на форме

```
void __fastcall TfoDragComponent::ImageOffDragDrop
            (TObject *Sender,TObject *Source, int X, int Y)
{ if (Source->ClassNameIs ("TShape"))
                              // идентификация типа источника
    if (((TShape *)Source)->Name == "Shape")
                                                 // иден-ция имени
      if (Sender->ClassNameIs ("TImage"))
                                  // иден-ция типа приемника
        ((TShape *)Source)->Hide();
}
```

Обработчик события `OnDragOver` для компонента формы

Листинг 14.13. Вариант обработчиков событий `OnDragOver` для компонента формы для манипулирования компонентами на форме

```
void __fastcall TfoDragComponent::FormDragOver
                           (TObject *Sender, TObject *Source,
               int X, int Y, TDragState State, bool &Accept)
{ Accept = Source->ClassNameIs ("TShape");
}
```

Обработчик события `OnDragDrop` для компонента `Form`

Листинг 14.14. Вариант обработчиков событий для манипулирования компонентами на форме

```
void __fastcall TfoDragComponent::FormDragDrop
            (TObject *Sender, TObject *Source, int X, int Y)
{ if (Source->ClassNameIs ("TShape"))
                              // идентификация типа источника
    if (((TShape *)Source)->Name == "Shape")
                                                 // иден-ция имени
      if (Sender ->ClassNameIs ("TfoDragComponent"))
                                  // иден-ция типа приемника
       {((TShape *)Source)->Left = X;
        ((TShape *)Source)->Top = Y;
       }
}
```

РАБОТА ПРОГРАММЫ. При «перетаскивании» квадрата `TShape` на изображение с финансами `TImage` он исчезает, а по кнопке *Восстановить* появляется вновь. При «бросании» на изображение фабрики `TImage` ничего не происходит. При «перетас-

кивании» квадрата TShape он перемещается по форме. Но перемещение это выполняется «как-то нехорошо»: пока курсор «мыши» движется в режиме *Drag&Drop*, квадрат остается на месте; когда пользователь отпускает кнопку «мыши» квадрат появляется своим левым верхним углом в точке, указанной курсором «мыши».

Было бы более правильно, чтобы перемещение компонента происходило также как при разработке форм в среде *C++ Builder*: компонент должен плавно следовать за курсором «мыши». Реализовать такое решение несложно, так как для TShape установлено значение dmAutomatic в свойстве DragMode, то для этого компонента сразу и возникают события OnStartDrag и OnDragOver при малейшем движении «мыши». Поэтому необходимо и достаточно реализовать обработчики этих двух событий, решив две задачи:

1) вычислить и запомнить положение курсора «мыши» относительно квадрата;
2) перемещать квадрат вслед за курсором «мыши», при этом сохраняя взаимное относительное расположение курсора и квадрата.

Тексты всех процедур нового варианта решения задачи приведены в листингах 14.15 – 14.19.

Исправленный вариант обработчика события OnDragDrop для компонента ImageOff

Листинг 14.15. Исправленный вариант обработчиков событий OnDragDrop для компонента ImageOff

```
void __fastcall TfoDragCompSmooth::ImageOffDragDrop
                      (TObject *Sender, TObject *Source,
                       int X, int Y)
{ if (Source->ClassNameIs ("TShape"))
                            // идентификация типа источника
    if (((TShape *)Source)->Name == "Shape")
                                              // иден-ция имени
      if (Sender->ClassNameIs ("TImage"))
                                    // иден-ция типа приемника
        ((TShape *)Source)->Hide();
}
```

Исправленный вариант обработчика события OnDragOver для компонента ImageOff

Листинг 14.16. Исправленный вариант обработчиков событий OnDragOver для компонента ImageOff

```
void __fastcall TfoDragCompSmooth::ImageOffDragOver
              (TObject *Sender, TObject *Source,
               int X, int Y, TDragState State, bool &Accept)
{ Accept = Source->ClassNameIs ("TShape");
}
```

Исправленный вариант обработчика события OnClick для кнопки bbRetry

Листинг 14.17. Исправленный вариант обработчиков событий OnClick для кнопки bbRetry

```
void __fastcall TfoDragCompSmooth::bbRetryClick (TObject *Sender)
{
  Shape->Show();
  Shape->Top = 10;
```

```
  Shape->Left = 10;
}

TPoint  gShapePnt;
```

Исправленный вариант обработчика события OnStartDrag для компонента Shape

Листинг 14.18. Исправленный вариант обработчика события OnStartDrag компонента Shape для плавного перемещения компонента по форме приложения

```
void __fastcall TfoDragCompSmooth::ShapeStartDrag
                (TObject *Sender, TDragObject *&DragObject)
{ gShapePnt = Shape->ScreenToClient (Mouse->CursorPos);
                           // координаты 'мыши'
                           // переводим в координаты TShape
}
```

Исправленный вариант обработчика события OnDragOver для компонента Shape

Листинг 14.19. Исправленный вариант обработчика события OnDragOver компонента Shape для плавного перемещения компонента по форме приложения

```
void __fastcall TfoDragCompSmooth::ShapeDragOver
             (TObject *Sender, TObject *Source,
              int X, int Y, TDragState State, bool &Accept)
{ TPoint aux;

  aux = Shape->ClientToParent (Point (X, Y));
                                       // пересчет координат
  Shape->Left = aux.x - gShapePnt.x;
                         // учет стартового смещения 'мыши'
  Shape->Top = aux.y - gShapePnt.y;
}
```

Обработчики событий для картинки ImageOff и кнопки *Восстановить* остались без изменения. А реализация обработчиков для формы и компонента TShape изменились. Точнее, для формы вовсе исчезли, а для квадрата были разработаны. Действительно ведь перемещение квадрата осуществляется, когда курсор «мыши» находится над квадратом, а не над формой. Поэтому события у формы не возникают и обрабатывать их нет необходимости.

Обработчик StartDrag квадрата TShape должен выполнить несколько действий:

1) *запомнить* насколько курсор был смещен относительно левого верхнего угла, когда началась операция *Drag&Drop*. Так как параметры этого обработчика не содержат координат курсора «мыши» их приходится опрашивать непосредственно у глобального объекта Mouse. Но эти координаты глобальные, поэтому их приходится пересчитывать в систему координат квадрата TShape. Для этого достаточно вызвать соответствующий метод ClientToParent компонента TShape. Вычисленные координаты и есть смещение курсора относительно левого верхнего угла;
2) *передать* это значение обработчику DragOver. Здесь используется глобальная переменная gShapePnt. Другой вариант решения – это поле формы.

Обработчик `DragOver` переводит координаты мыши, переданные через параметры, из системы координат квадрата в систему координат формы и вносит поправку, запомненную на старте. Полученные координаты и есть координаты левого верхнего угла квадрата `TShape` на форме.

Теперь программа работает правильно. Почти. Правильно она работает, только если пользователь «перетаскивает» квадрат очень медленно. Чуть более резкое движение – и курсор «слетает» с квадрата, и движение останавливается. Для исправления этого дефекта достаточно дополнить последний проект обработчиком события `OnDragOver` формы (см. листинг 14.20).

Обработчик события `OnDragOver` для компонента формы

Листинг 14.20. Обработчик события `OnDragOver` формы

```
void __fastcall TfoDragCompSmooth::FormDragOver
               (TObject *Sender, TObject *Source,
                int X, int Y, TDragState State, bool &Accept)
{ Shape->Left = X - gShapePnt.x;
  Shape->Top = Y - gShapePnt.y;
}
```

Наличие такого обработчика «исправляет дефект» программы. Как только курсор «мыши» попадает на форму, для нее возникает событие `DragOver` и обработчик «ставит на место» квадрат `TShape`. Теперь все манипуляции пользователя «мышью» отрабатываются программой корректно.



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. Доработать приложение 14.1 перетаскивания строки текста из списка строк в окно редактирования. Реализовать «перетаскивание» информации из окна редактора обратно в список.
2. Доработать приложение 14.1. Реализовать возможность дополнения списка или замены текущей его строки при «перетаскивании» текста из окна редактора.
3. Дополнить приложение возможностью «перетаскивать» информацию из окна редактирования в другой список. При этом в список строка должна попадать лишь в том случае, если такой строки в списке еще нет, то есть в списке не должно возникать дублирующихся строк.

Построение графиков 15

Различные приложения, выполняющие математические вычисления, отображают результаты в текстовом формате и в виде графиков. Текстовый формат подразумевает вывод отдельных чисел и комментариев к ним или вывод большого количества данных, организованных в таблицы.

Но для пользователей программы большое значение имеет возможность увидеть данные в виде графиков, а не только в виде колонок цифр. Общий вид графика функции значительно упрощает первоначальный анализ данных. На графике легко можно оценить динамику изменения данных – возрастание или убывание функции, монотонность или унимодальность.

При построении кода вывода графика произвольной функции необходимо учитывать следующие факторы:

- функция задана на интервале *целых чисел* и принимает только *целочисленные* значения или аргумент, и значения являются *вещественными* данными;
- интервал отображения графика функции, то есть пределы изменения аргумента, может быть достаточно *малым*, например доли единицы, или достаточно *большим*, например миллионы единиц. Аналогично, независимо от величины интервала отображения графика, *размах значений* функций может быть очень малым или очень большим;
- помимо самой функции график должен содержать *изображение осей координат* и различные *надписи*.

Если диапазон значений аргумента и функции сравним с размерами экрана, а сами значения целочисленные, то для вывода графика достаточно простого цикла, в котором для набора значений абсцисс x_i вычисляются значения функции y_i, и точки экрана с координатами (x_i, y_i) соединяются ломаной линией. Если же значения абсциссы или функции вещественные, то процесс построения линии графика дополняется операцией преобразования из вещественного в целочисленный тип.

Чтобы учесть второй из этих факторов, необходимо выполнять *масштабирование* значений абсцисс и ординат. Масштабирование, в свою очередь, приводит к задаче отображения областей – из математической в экранную и обратно.

Третий фактор связан с задачей построения осей координат. Эта задача значительно сложнее, чем процесс построения графика функций. Сами оси координат, как некий объект, являются сложным комплексом, состоящим из нескольких элементов: собственно оси, маркеры, подписи к маркерам, подписи к осям. Поэтому для решения задачи целесообразно *создать отдельный класс* «Оси координат».

15.1. Построение графиков функций

Как уже оговаривалось в тех случаях, когда функция $y = F(x)$ имеет ограниченный *размах значений* функций в области определения, построение графика не вызывает затруднений. График такой функции можно получить в виде ломаной линии. В рассмотренном примере, для общности подхода, вычисление значений функции выполняется в отдельной подпрограмме. Ее описание представлено в листинге 15.2.

Задача 15.1. Требуется построить график функции $y = F(x) = 200 \cdot \cos^2(0.01 \cdot x)$ в интервале [0; 500] с шагом $h = 20$.

Решение

Реализовать процедуру отображения такого графика достаточно просто – в виде одного цикла: перебирать значения абсциссы с заданным шагом, вычислять значения функции в этих точках и соединять эти пары точек (x_i, y_i) отрезками прямой линии.

Построение графика заданной функции с равномерным шагом

Процедура построения графика имеет один параметр типа `TImage`. Этот параметр определяет компонент, предназначенный для отображения графика. Алгоритм сводится к выполнению следующих шагов:

1) вычисляем значение функции в точке $x = 0$ – точке начала холста; следовательно, получаем координаты точки на плоскости отображения $p_0 = (0, y(0))$;
2) посредством команды `MoveTo` холста, перемещаем курсор в начальную точку p_0;
3) вычисляем новое значение $x_{i+1} = x_i + h$;
4) вычисляем значение функции в точке $x_{i+1} - y_{i+1} = F(x_{i+1})$;
5) методом `LineTo` холста чертим отрезок из точки p_i в точку p_{i+1};
6) если очередная абсцисса x_{i+1} меньше ширины холста, то переходим к шагу 3, иначе завершаем процесс.

Простейшее приложение для построения графика может содержать компонент `Image` для отображения графика и кнопку для старта операции построения (см. рис. 15.1).

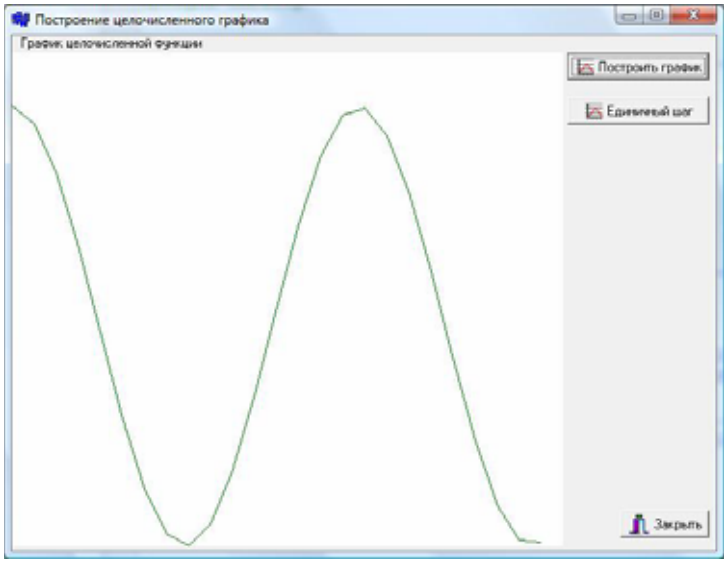


Рис. 15.1. Интерфейс приложения для построения простого графика

Листинг 15.1. Процедуры вычисления значения функции в заданной точке

```
int __fastcall func (int aX)
```

```
{   // подпрограмма вычисления функции
  return int (400 * cos (0.01 * aX) * cos (0.01 * aX));
}
```

Листинг 15.2. Процедуры построения графика функции

```
void __fastcall
              TfoMain::bbCreateGraphicClick(TObject *Sender)
{ int h = 20;  // шаг движения по абсциссе
  Image->Canvas->Pen->Color = clGreen;  // цвет графика
  Image->Canvas->MoveTo (0, Image->Height - func (0) - 1);
               // переместить перо в начальную точку графика
  int i = h;  // вторая точка
  while (i < Image->Width)
   {Image->Canvas->LineTo (i, Image->Height -
                                              func (i) - 1);
                                    // чертить отрезок ломаной
    i = i + h;  // переход к следущей абсциссе
   }
}
```

Данная процедура на языке программирования *C++* выполняет вычисления, в соответствии с описанным алгоритмом. Из-за того, что по вертикали значение координаты растет вниз, вместо значения функции `func(i)` используется значение выражения:

```
Image->Height - func (i) - 1.
```

Это обеспечивает такое формирование изображения графика, что точка $(x_i, 0)$ отображается в нижней строке холста, а максимальная ордината – в верхней.

Построение графика функции с минимальным шагом

Одним из недостатков предыдущего графика является то, что он отображается в виде ломаной линии, что особенно заметно на участках с большой кривизной. Улучшить вид графика можно, задав шаг табулирования функции, равный единице. Это минимальный шаг табулирования. Более мелкий шаг не имеет смысла, так как минимальный шаг движения по горизонтали (да и по вертикали) всего один пиксель. Поэтому логичнее было бы решать следующую задачу.

Задача 15.2. Требуется построить график функции $y = F(x) = 200 \cdot \cos^2(0.01 \cdot x)$ в интервале [0; 500] с шагом $h = 1$.

Решение

Решение такой задачи эквивалентно решению предыдущей задачи. Отличие состоит лишь в шаге движения по оси абсцисс. Здесь он – единичный ($h = 1$). Поэтому и процедура построения графика с единичным шагом такая же, как прцедура построения графика с произвольным шагом. Текст такой процедуры приведен в листинге 15.3.

Листинг 15.3. Процедура построения графика функции с единичным шагом

```
void __fastcall TfoMain::
                  bbOnePixelStepGraphClick(TObject *Sender)
```

```
{ int h = 1;  // шаг движения по абсциссе
  Image->Canvas->Pen->Color = clBlue;  // цвет графика
  Image->Canvas->MoveTo (0, Image->Height - func (0) - 1);
              // переместить перо в начальную точку графика
  for (int i = h; i < Image->Width; i += h)
                                          // перебор абсцисс
    Image->Canvas->LineTo (i, Image->Height - func (i) - 1);
                                  // чертить отрезок ломаной
}
```

Основной цикл здесь записан в другой форме – через оператор `for`, но по сути – это такой же цикл.

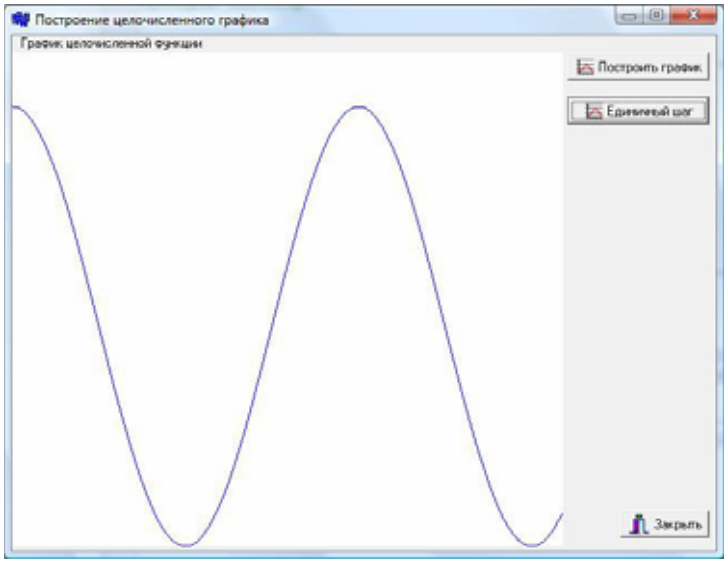


Рис. 15.2. Интерфейс приложения для построения простого графика с единичным шагом

Отметим

Конечно, вид и этого графика (см. рис. 15.2) также может вызывать нарекания. Но дальнейшее улучшение вида линии графика требует применения сложных технологий компьютерной графики, в частности *метода антиалиазинга*, а это резко увеличивает сложность такой процедуры.

15.2. Использование элементов масштабирования при построении графиков

Вывод графика произвольной функции – процесс более сложный, чем продемонстрировано в предыдущих примерах.

При решении различных задач с участием явно заданных функций вида $y = f(x)$ важно помнить об *области определения* этих функций. Большинство функций определено на всей числовой оси $x \in [-\infty, \infty]$. Но нередко функции имеют разрывы, в том числе и бесконечные, или просто не определены на некотором интервале. Помимо этого есть интервал значений, которые принимает функция – *область значений функции*.

При построении графика функции, как правило, рассматривают определенный отрезок числовой оси $x \in [x_1, x_2]$, на котором этот график строится, а также область значений $y \in [y_1, y_2]$, которые принимает функция на этом интервале. То есть рассматривается некоторая прямоугольная область на плоскости $x0y$.

Как правило, оси координат, отображаемые на графике, проходят через нулевые значения и пересекаются в точке (0, 0). Но если рассматриваемая область лежит на значительном расстоянии от центра системы координат, то оси проводят через нижнюю левую точку области (или близлежащую точку) и указывают значения по осям *OX* и *OY* в точке их пересечения. Кроме того, необходимо помнить, что экран представляет собой сетку из дискретных точек, то есть отобразить можно график только приближенно, в дискретно заданных значениях оси ординат *OX*.

Следовательно, при построении изображения графика возникает задача *отображения* прямоугольной области (x_1, y_1; x_2, y_2) *в область экрана* (0, 0, *Width* – 1, *Height* – 1), где *Width*, *Height* – *свойства компонента* `TImage`.

Координата *x*, соответствующая *i*-столбцу на экране, может быть рассчитана по формуле:

$$x = (x_2 - x_1) \cdot i / Image1.Width + x_1. \qquad (15.1)$$

Аналогично можно поставить в соответствие значение по оси ординат 0*y* строке экрана *j*. По подобной формуле делается пересчет от координат на области в координаты экрана. Так, по значению ординат *y* можно посчитать строку экрана, в которой необходимо строить эту точку:

$$j = (y - y_1) \cdot Image1.Height / (y_2 - y_1). \qquad (15.2)$$

Такой пересчет данных называют задачей масштабирования.

МАСШТАБИРОВАНИЕ ДАННЫХ

Масштабирование данных представляет собой процесс отображения *математической* области данных в *экранную* область построения графика. При этом все элементы графика на экране должны размещаться точно в соответствии с их математическим местоположением.

Для хранения данных о математической и экранной областях отображения достаточно хранить четыре значения – минимальное и максимальное значения каждой координаты. Эти данные можно описать структурой, как показано в листингах 15.4 и 15.5.

Листинг 15.4. Интерфейс структур для описания математической области

```
struct TRegion
{
  long double Xmin, Ymin, Xmax, Ymax; // координаты области
};
```

Листинг 15.5. Интерфейс структур для описания экранной области

```
struct TScreenRect
{
  int SXmin, SYmin, SXmax, SYmax; // координаты области
};
```

Предположим, что значения конкретных данных об областях хранятся в ГЛОБАЛЬНЫХ переменных `gMathReg` – для математической области и `gScrRect` – для области экрана:

```
TRegion     gMathReg;
```

```
TScreenRect gScrRect;
```

Для того чтобы установить значения глобальных переменных целесообразно создать специальную процедуру.

Установка координатных значений отображаемых областей

Процедура `SetRegions` имеет восемь входных параметров:

- `aSXmin`, `aSYmin`, `aSXmax`, `aSYmax` – для передачи максимальных и минимальных координатных значений экранной области;
- `aXmin`, `aYmin`, `aXmax`, `aYmax` – для передачи максимальных и минимальных координатных значений математической области.

Описание процедуры `SetRegions` преставлено в листинге 15.6.

Листинг 15.6. Инициализация координат отображаемых областей

```
void SetRegions (int aSXmin, int aSYmin,
                 int aSXmax, int aSYmax,
                 long double aXmin, long double aYmin,
                 long double aXmax, long double aYmax)
{
  gScrRect.SXmin = aSXmin; // инициализация
  gScrRect.SYmin = aSYmin;
  gScrRect.SXmax = aSXmax;
  gScrRect.SYmax = aSYmax;

  gMathReg.Xmin = aXmin;
  gMathReg.Ymin = aYmin;
  gMathReg.Xmax = aXmax;
  gMathReg.Ymax = aYmax;
};
```

Тогда пересчет координат точек для отображения их из одной области в другую можно описать с помощью процедур, которые представлены в листингах 15.7 и 15.8.

Пересчет координат точек математической области в точки экрана

Процедура `MathToScr` имеет два входных параметра `aX` и `aY` – координаты математической точки. И два выходных параметра `aSX` и `aSY` – координаты экранной точки. Пересчет координат реализуется по формуле 15.2 с учетом введенных обозначений и использования значений глобальных переменных `gMathReg` и `gScrRect`. Код процедуры представлен в листинге 15.7.

Листинг 15.7. Пересчет координатных значений в экранную область

```
void MathToScr (long double aX, long double aY, int &aSX, int &aSY)
{
  aSX = (aX - gMathReg.Xmin) * (gScrRect.SXmax - gScrRect.SXmin) /
(gMathReg.Xmax - gMathReg.Xmin) + gScrRect.SXmin;
  aSY = (aY - gMathReg.Ymin) * (gScrRect.SYmax - gScrRect.SYmin) /
(gMathReg.Ymax - gMathReg.Ymin) + gScrRect.SYmin;
}
```

Пересчет координат точек экранной области в точки математические

Процедура `ScrToMath` имеет два входных параметра: `aSX` и `aSY` – координаты экранной точки. И два выходных параметра `aX` и `aY` – координаты математической экранной точки. Пересчет координат реализуется по формуле 15.1 с учетом введенных обозначений и использования глобальных переменных `gMathReg` и `gScrRect`. Код процедуры представлен в листинге 15.8.

Листинг 15.8. Пересчет координатных значений в математическую область

```
void ScrToMath (int aSX, int aSY, long double &aX, long double &aY)
{
  aX = (aSX - gScrRect.SXmin) *
         (gMathReg.Xmax - gMathReg.Xmin) /
         (gScrRect.SXmax - gScrRect.SXmin) + gMathReg.Xmin;
  aY = (aSY - gScrRect.SYmin) *
         (gMathReg.Ymax - gMathReg.Ymin) /
         (gScrRect.SYmax - gScrRect.SYmin) + gMathReg.Ymin;
}
```

Примечание

Описанные структуры хранения параметров областей и процедуры пересчета лучше описать в отдельном модуле, на который бы могли ссылаться все модули, входящие в проект приложения, предназначенного для построения графиков функций.

Задача 15.3. Требуется построить график функции $y = F(x) = 10 \cdot \cos^2(x)$ на интервале [–1; 10]. По оси $0y$ выбрать такой масштаб, чтобы график отображался на всю высоту области отображения.

Решение

Решение этой задачи подобно решению задачи 15.2. Отличительной чертой является процесс масштабирования. Это позволяет график функции с любыми областями определения и значений отображать на всей площади холста. То есть любой интервал, составляющий доли единицы и имеющий длину в миллионы единиц, может быть отображен на интервале, сравнимым с размерами окна. В силу этого процесс построения графика функции немного модифицируется и принимает следующий вид:

- производится вычисление начальной точки вывода графика: подпрограммой `ScrToMath` рассчитывается значение x_H, соответствующее левой границе интервала отображения; определяется значение функции y_H в точке x_H; и пересчитывается эта математическая точка (x_H, y_H) в точку изображения $(x_Э, y_Э)$;
- графический курсор устанавливается в эту точку;
- осуществляется перебор всех пикселей по горизонтали, начиная с индекса $i = 1$. Для каждого значения индекса, как значения абсциссы в области отображения, делается следующее:
  - значение индекса принимается за значение абсциссы $x_Э$ в области изображения;
  - производится вычисление соответствующего ей значения абсциссы в математическом пространстве x;
  - выполняется вычисление значения функции в математическом пространстве y;

  - производится пересчет полученной математической ординаты в ординату области изображения $y_Э$;
  - вычерчивается отрезок прямой линии из предыдущей экранной точки во вновь вычисленную точку изображения;
- процесс построения графика функции завершается на последней колонке пикселей, входящей в область отображения.

Такой порядок построения определяется тем, что целочисленный перебор колонок пикселей является точным, и погрешность вычислений определяется только погрешностью пересчета в математическое пространство и обратно, а также погрешностью вычисления функции.

Пример реализации простейшего класса `TFuncGraphic`, предназначенного для построения графиков функции приведен в листингах 15.9–15.12, а общий вид графика функции, заявленной в условии задачи 15.3, представлен на рис. 15.3.

Листинг 15.9. Интерфейс класса `TFuncGraphic` для построения графика функции в произвольной математической области

```
class TFuncGraphic
  {
  private:
    int    FWidth; // толщина линии
    TColor FColor; // цвет линии
  protected:
    long double Func1 (long double); // вычисление функции
  public:
    TFuncGraphic (); // конструктор
    void __fastcall DrawGraphic(TImage *); // построить график
  };
```

Листинг 15.10. Описание конструктора класса `TFuncGraphic`

```
TFuncGraphic::TFuncGraphic ()  // конструктор
{  // инициализировать поля
 FWidth = 1;
 FColor = clBlack;
}
```

Листинг 15.11. Вычисление значения функции в точке `aX`

```
long double TFuncGraphic::Func1 (long double aX)
{ return 10.0 * cos (aX) * cos (aX);
}
```

Листинг 15.12. Описание метода построения графика функции в произвольной математической области

```
void __fastcall TFuncGraphic::DrawGraphic (TImage *aIm)
{ aIm->Canvas->Pen->Color = FColor;
  aIm->Canvas->Pen->Width = FWidth;

    // расчет начальной точки графика на левом краю
  int iX, iY;
  long double begX, begY;

  ScrToMath (0, -1, begX, begY);
     // координата Y несущественна (не будет использоваться)
  MathToScr (begX, Func1 (begX), iX, iY);
```

```
  aIm->Canvas->MoveTo (iX, aIm->Height - 1 - iY);
                          // установка графического курсора

  for (int i = 1; i < aIm->Width; i++)
   {long double currX, currY;

    ScrToMath (i, -1, currX, currY);
     // координата Y несущественна (не будет использоваться)
     // координата X соответствует очередному пикселю экрана
    MathToScr (currX, Func1 (currX), iX, iY);
                                  // расчет координаты Y; и
                                  // расчет координат экрана
    aIm->Canvas->LineTo (iX, aIm->Height - 1 - iY);
                               // рисование очередного отрезка
   };
}
```

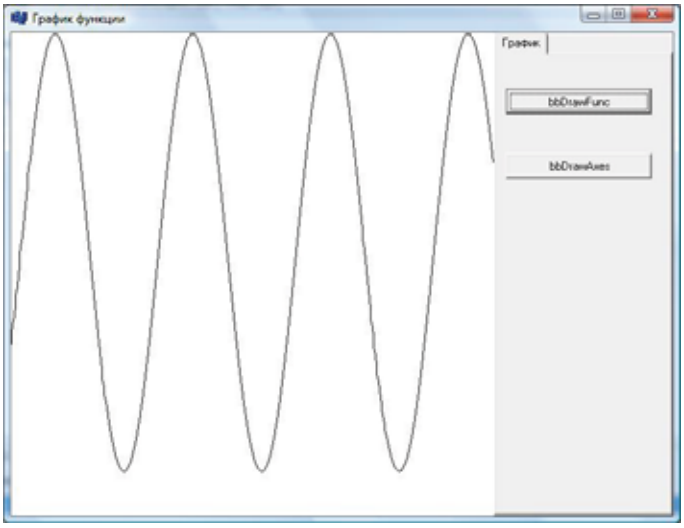


Рис. 15.3. Построение графика функции $y = F(x) = 10 \cdot \cos^2(x)$ *с необходимым масштабом*

Замечание

Реализация процесса построения графика в виде класса здесь приведена с целью создания ЗАГОТОВКИ такого класса. Ее доработка в дальнейшем может привести к разработке вполне полновесного класса. Но это не затеняет центрального вопроса, так как весь процесс построения графика сосредоточен в методе `DrawGraphic`.

Если значения переменных хранятся с большим количеством разрядов

Если значения вещественных переменных, используемых для вычислений в математическом пространстве, хранятся с большим количеством разрядов, то есть погрешности вычислений очень малы, то процесс вычисления точек графика функции может быть несколько сокращен. А именно: весь процесс вычисления значений функции осуществляется *исключительно в математической* области и только в последний момент производится пересчет математической ординаты в экранную область. Это значит, что просто в математической области производится табулирование значений функции с шагом, соответствующим перемещению на один пиксель, и производится пересчет матема-

тических точек (x, y) в точки изображения ($x_Э$, $y_Э$). Проблемы погрешности вычислений абсцисс такие же, как и при обычном табулировании функций, которые рассматривались в главе 3. Описание такой процедуры представлено в листинге 15.13.

Построение графика функции в произвольной математической области

В этом случае процедура построения графика может быть описана следующими шагами:

- производится вычисление начальной абсциссы x_H и конечной абсциссы x_K, математического интервала табулирования функции; вычисляется шаг табулирования h, исходя из длины интервала отображения в математической и экранной области;
- производится вычисление начальной точки вывода графика: вычисляется значение функции y_H в точке x_H и пересчитывается эта математическая точка (x_H, y_H) в точку изображения ($x_Э$, $y_Э$);
- устанавливается в эту точку графический курсор;
- осуществляется табулирование абсцисс, начиная с точки $x = x_H + h$ с шагом h. Для каждого значения абсциссы выполняется следующее:
  - выполняется вычисление значения функции в математическом пространстве y;
  - производится пересчет полученной математической ординаты в ординату области изображения $y_Э$;
  - вычерчивается отрезок прямой линии из предыдущей экранной точки во вновь вычисленную точку изображения;
- процесс построения графика функции завершается, когда очередная точка выходит за границу интервала отображения функции, то есть когда наступает случай $x > x_K$.

Описание процедуры построения графика функции в произвольной математической области представлено в листинге 15.13.

Листинг 15.13. Подпрограмма построения графика функции в произвольной математической области

```
void __fastcall TFuncGraphic::DrawGraphicF (TImage *aIm)
{ aIm->Canvas->Pen->Color = FColor;
  aIm->Canvas->Pen->Width = FWidth;

    // расчет начальной точки графика на левом краю,
    // конечной точки графика на правом краю
    // и шага
  int iX, iY;
  long double begX, begY, endX, endY, h;

  ScrToMath (0, -1, begX, begY);
     // координата Y несущественна (не будет использоваться)
  ScrToMath (aIm->Width - 1, -1, endX, endY);
     // координата Y несущественна (не будет использоваться)
  h = (endX - begX) / (aIm->Width - 1);

  MathToScr (begX, Func1 (begX), iX, iY);

  aIm->Canvas->MoveTo (iX, aIm->Height - 1 - iY);
```

```
                              // установка графического курсора

  long double currX, currY;
  currX = begX + h;
  while (currX <= endX + 0.5 * h)
   {MathToScr (currX, Func1 (currX), iX, iY);
                                    // расчет координаты Y; и
                                    // расчет координат экрана
    aIm->Canvas->LineTo (iX, aIm->Height - 1 - iY);
                                   // рисование очередного отрезка
    currX += h;
   };
}
```

Примечание

График той же функции на том же интервале будет выглядеть так же, как на рис. 15.3.

15.3. Разработка класса «Оси координат»

Построение изображения графика функции не может обойтись без изображения осей координат. Ведь для полного анализа значений функции человеку необходимо опираться на специальные метки на изображении, которые бы указывали на положение точек функции в математическом пространстве.

При построении осей координат, как и при построении графиков, приходится учитывать, какое математическое пространство отображается в данную область изображения. Поэтому процесс построения осей координат сопровождается большим количеством расчетов местоположения различных элементов на изображении.

При разработке класса (назовем его `TAxes`), необходимо определиться с его параметрами: полями и методами. Выбор полей производится на основании следующих рассуждений.

Изображение осей координат графиков функций хорошо известно из школьной программы. Пример осей координат приведен на рис. 15.4.

Как следует из анализа изображения осей координат, в нем можно выделить следующие элементы:

- ось абсцисс (горизонтальная) и ось ординат (вертикальная);
- каждая ось имеет следующие элементы:
  - линия оси – основная линия, обозначающая ось координат, часто завершается стрелкой;
  - основные отметки – штрихи, пересекающие линию оси, обозначающие местоположение основных отсчетов;
  - подписи к основным отметкам – значения оси в точках, указанных отметками;
  - дополнительные отметки – мелкие штрихи, указывающие отсчеты оси с более мелким шагом; как правило, не подписываются;
  - подписи к осям – текст, указывающий, какая величина откладывается по данной оси, и размерность этой величины; например, ось частот обычно подписывают как *f*, Гц (размещается на конце оси);

– название оси – текст с названием оси, например «Частота»; обычно размещается в середине оси.

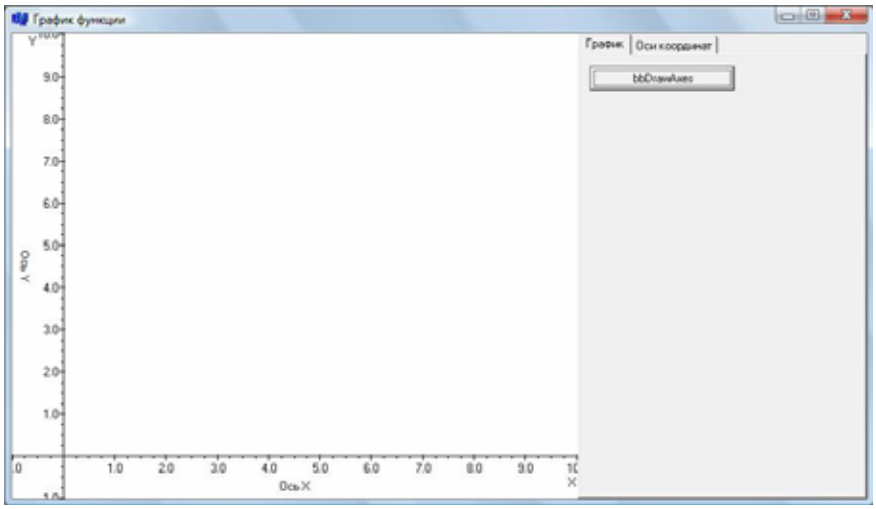


Рис. 15.4. Изображение осей координат

При этом необходимо учитывать, что нанесение штрихов и надписей на горизонтальной и вертикальной осях принципиально отличаются, по крайней мере, углом вывода.

Представленные элементы, характеризующие оси, можно описать в виде структуры, интерфейс которой представлен в листинге 15.14.

Листинг 15.14. Интерфейс структуры, предназначенной для описания осей координат

```
struct TAxisParam  // структура для индивидуальных параметров
 {
   int PTicks,  // длина штриха в положительную сторону
       NTicks,  // длина штриха в отрицательную сторону
      PmTicks,  // длина малого штриха положительную сторону
      NmTicks;  // длина малого штриха отрицательную сторону
   long double Step;  // шаг штриха по оси
   int mCount;  // кол-во малых штрихов
   int Digits;  // количество знаков после запятой
   AnsiString Script;  // подпись оси
   AnsiString Title;  // название оси
 };
```

Размер математической области по вертикали и горизонтали при выводе графиков, как правило, имеет разные диапазоны. Поэтому *структурные параметры для горизонтальной и вертикальной осей* целесообразно хранить в виде отдельных полей класса (`FX` и `FY`). В поля класса также необходимо включить *две точки пересечения осей*, одну `FPointCross` – математическую (из математической области отображения), а другую `FCrossPnt` – экранную. Для описания *экранной* точки используется стандартный тип `TPoint`, определенный в библиотеке *Windows*. Этот тип также определяет экранные координаты *X* и *Y* левого верхнего угла прямоугольной области, но измеряет в пикселах. Имеет смысл в поля класса также поместить переменные для задания *цвета линий* (`FColor`) и *цвета шрифта* подписей (`FFontColor`).

Интерфейс структуры, описывающей *математическую* точку, представлен в листинге 15.15.

Листинг 15.15. Интерфейс структуры, предназначенной для описания координат точки пересечения осей в математической области

```
struct TMathPoint
{
  long double X, Y;  // координаты точки
};
```

Задача построения осей координат включает в себя решение значительного количества частных задач: *рисование линий осей, вывод штрихов и подписей по осям, вывод общего заголовка.* Эти задачи целесообразно оформить методами класса. Как уже отмечалось ранее, вывод элементов горизонтальной и вертикальной осей координат имеют большие отличия. Поэтому многие функции являются дубликатами по назначению, но для разных осей. Для увеличения универсальности класса *Axes* при выводе осей координат в различных приложениях, компонент *TImage* передается как параметр, а не является полем класса *Axes*. И, конечно, в данном случае было бы разумным определить свойства, назначение которых логически следует из полей класса. Перечисленные методы могут быть размещены в разделе **protected** класса *TAxes*.

Примечание

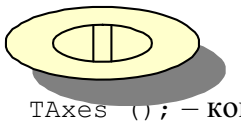
Для построения полного изображения осей можно разработать *открытый* метод, который будет вызывать описанные выше методы.

Интерфейс класса представлен в листинге 15.16.

Листинг 15.16. Интерфейс класса «Координатные оси»

```
class TAxes
 {
   private:
   protected:
    TAxisParam FX, FY;  // параметры осей Ox и Oy
    TMathPoint FPointCross;
                    // математическая точка пересечения осей
    TPoint FCrossPnt;   // точка пересечения осей на экране
    TColor FColor;      // цвет осей и штрихов
    TColor FFontColor;  // цвет шрифтов для подписей
    void DrawAxesLine (TImage *);
    void XDrawTicks (TImage *);
    void YDrawTicks (TImage *);
    void XDrawMinors (TImage *);
    void YDrawMinors (TImage *);
    void XValueOut (TImage *);
    void YValueOut (TImage *);
    void XScriptOut (TImage *);
    void YScriptOut (TImage *);
    void XTitleOut (TImage *);
    void YTitleOut (TImage *);
   public:
      TAxes ();
      void DrawAxes (TImage *);
       /* свойства */
    __property int XPTicks = {read = FX.PTicks,
                              write = FX.PTicks};
    __property int XNTicks = {read = FX.NTicks,
                              write = FX.NTicks};
```

```
  __property int YPTicks = {read = FY.PTicks,
                            write = FY.PTicks};
  __property int YNTicks = {read = FY.NTicks,
                            write = FY.NTicks};
  __property int MXPTicks = {read = FX.PmTicks,
                             write = FX.PmTicks};
  __property int MXNTicks = {read = FX.NmTicks,
                             write = FX.NmTicks};
  __property int MYPTicks = {read = FY.PmTicks,
                             write = FY.PmTicks};
  __property int MYNTicks = {read = FY.NmTicks,
                             write = FY.NmTicks};
  __property long double XStep = {read = FX.Step,
                                  write = FX.Step};
  __property long double YStep = {read = FY.Step,
                                  write = FY.Step};
  __property int XmCount = {read = FX.mCount,
                            write = FX.mCount};
  __property int YmCount = {read = FY.mCount,
                            write = FY.mCount};
  __property int XDigits = {read = FX.Digits,
                            write = FX.Digits};
  __property int YDigits = {read = FY.Digits,
                            write = FY.Digits};
  __property AnsiString XScript = {read = FX.Script,
                                   write = FX.Script};
  __property AnsiString YScript = {read = FY.Script,
                                   write = FY.Script};
  __property AnsiString XTitle = {read = FX.Title,
                                  write = FX.Title};
  __property AnsiString YTitle = {read = FY.Title,
                                  write = FY.Title};
  __property TColor Color = {read = FColor,
                             write = FColor};
  __property TColor FontColor = {read = FFontColor,
                                 write = FFontColor};
  __property TPoint CrossPnt  = {read = FCrossPnt,
                                 write = FCrossPnt};
                    // точка пересечения осей на экране
};
```



МЕТОДЫ КЛАССА `TAxes`:

- `TAxes ();` – конструктор не имеет параметров. Он задает стандартные значения полей;
- `void DrawAxesLine (TImage* aIm);` – метод класса для прорисовки осевых линий в области `aIm`;
- `void XDrawTicks (TImage* aIm);` – вывод основных штрихов по оси *OX* в области `aIm`;
- `void YDrawTicks (TImage* aIm);` – вывод основных штрихов по оси *OY* в области `aIm`;
- `void XDrawMinors (TImage* aIm);` – вывод дополнительных штрихов по оси *OX* в области `aIm`;

- `void YDrawMinors (TImage* aIm);` – вывод дополнительных штрихов по оси *OY* в области `aIm`;
- `void XValueOut (TImage* aIm);` – вывод подписей основных штрихов по оси *OX* в области `aIm`;
- `void YValueOut (TImage* aIm);` – вывод подписей основных штрихов по оси *OY* в области `aIm`;
- `void XScriptOut (TImage* aIm);` – вывод подписи оси *OX* в области `aIm`;
- `void YScriptOut (TImage* aIm);` – вывод подписи оси *OY* в области `aIm`;
- `void XTitleOut (TImage* aIm);` – вывод названия оси *OX* в области `aIm`;
- `void YTitleOut (TImage* aIm);` – вывод названия оси *OY* в области `aIm`.

Конструктор класса «Оси координат»

Поскольку в описанном классе нет динамических переменных, в задачу конструктора входит только инициализация полей. Те значения, которые заданы полям, ввиду их большого количества, не передаются через список параметров. В принципе, значения рационально подобраны, но их можно изменять (в коде конструктора) согласно требованиям пользователя. На данном этапе они находятся в соответствии с изображением, представленным на рис. 15.4. Код конструктора представлен в листинге 15.17.

Листинг 15.17. Описание конструктора класса `TAxes`

```
TAxes::TAxes ()
{
  FX.PTicks = 1; FX.NTicks = 4;
  FY.PTicks = 1; FY.NTicks = 4;
  FX.PmTicks = 0; FX.NmTicks = 2;
  FY.PmTicks = 0; FY.NmTicks = 2;
  FX.Step = 1.0; FX.mCount = 3; FX.Digits = 1;
  FY.Step = 1.0; FY.mCount = 3; FY.Digits = 1;
  FX.Script = "X"; FX.Title = "Ось X";
  FY.Script = "Y"; FY.Title = "Ось Y";
  FPointCross.X = 0.0; FPointCross.Y = 0.0;
  FColor = clBlack;
  FFontColor = clBlack;
  FCrossPnt.x = 0; FCrossPnt.y = 0;
}
```

Построение осей координат

Процедура `DrawAxes`, предназначенная для построения осей координат, имеет один параметр типа `TImage` – в его поле строится график. Метод `DrawAxes` вызывает следующие процедуры: рисование линий осей, вывода штрихов и подписей по осям. Действия этой процедуры можно описать следующими шагами.

1. Устаналиваются параметры: стиль линии; ширина; цвет.
2. Проводятся координатные линии.
3. Выполняется проверка, требуется ли рисовать основные штрихи по оси *OX* (если длина основного штриха вверх или вниз больше нуля). Если так, то необходимо проделать следующее:
    - наносятся штрихи по оси *OX*;

- выполняется проверка, требуется ли рисовать малые штрихи (если длина малого штриха вверх или вниз больше нуля). Если так, то рисуются малые штрихи по оси *OX*.

4. Подписываются штрихи по оси *OX*.
5. Выполняется проверка, требуется ли рисовать основные штрихи по оси *OY* (если длина основного штриха влево или вправо больше нуля). Если так, то
   - наносятся штрихи по оси *OY*;
   - выполняется проверка, требуется ли рисовать малые штрихи (если длина малого штриха влево или вправо больше нуля). Если так, то рисуются малые штрихи по оси *OY*.
6. Подписываются штрихи по оси *OY*.
7. Подписываются оси *OX* и *OY*.
8. Подписываются названия осей *OX* и *OY*.

Код процедуры `DrawAxes` построения координатных осей представлен в листинге 15.18.

Листинг 15.18. Процедура построения координатных осей

```
void TAxes::DrawAxes (TImage *aIm)
{   // построение осей координат
  aIm->Canvas->Pen->Style = psSolid;  // параметры линии
  aIm->Canvas->Pen->Width = 1;
  aIm->Canvas->Pen->Color = FColor;

  DrawAxesLine (aIm);  // вычерчивание осей

  if (FX.NTicks > 0 || FX.PTicks > 0)
                              // нанесение штрихов по оси Ox
   {XDrawTicks (aIm);
    if (FX.NmTicks > 0 || FX.PmTicks > 0)
      XDrawMinors (aIm);
   }
  XValueOut (aIm);  // подпись штрихов оси Ox

  if (FY.NTicks > 0 || FY.PTicks > 0)
                              // нанесение штрихов по оси Oy
   {YDrawTicks (aIm);
    if (FY.NmTicks > 0 || FY.PmTicks > 0)
      YDrawMinors (aIm);
   }
  YValueOut (aIm);  // подпись штрихов оси Oy

  XScriptOut (aIm);  // подпись оси Ox
  YScriptOut (aIm);  // подпись оси Oy

  XTitleOut (aIm);  // название оси Ox
  YTitleOut (aIm);  // название оси Oy
}
```

Изображение линий осей координат

Процедура `DrawAxesLine`, предназначенная для построения линий осей координат, имеет один параметр типа `TImage` – в его поле строится график. Прорисовка каждой из двух координатных линий сводится к двум операциям: поместить перо в

точку – начало линии, и прочертить линию до заданной точки. Здесь необходимо учитывать, что пиксель с координатами (0, 0) располагается в левом верхнем углу окна `Image`, а сами линии отодвигаются от границ на один пиксел. Описание метода представлено в листинге 15.19.

Листинг 15.19. Описание метода для прорисовки линий осей координат

```
void TAxes::DrawAxesLine (TImage *aIm)
{      // горизонтальная ось
  aIm->Canvas->MoveTo (0, FCrossPnt.y);
  aIm->Canvas->LineTo (aIm->Width - 1, FCrossPnt.y);
      // вертикальная ось
  aIm->Canvas->MoveTo (FCrossPnt.x, 0);
  aIm->Canvas->LineTo (FCrossPnt.x, aIm->Height - 1);
}
```

Вывод основных штрихов по оси *OX*

Процедура `XDrawTicks`, предназначенная для вывода штриховых линий осей координат, имеет один параметр типа TImage. Для упрощения выполнения расчетов и разработки процедур вывода осей координат будем исходить из предположения, что математическая область, которая выводится на экран, содержит нулевые отсчеты по обеим осям. Это несколько упрощает построение осей. Тогда построение отсчетов и подписей к ним для одной оси можно выполнить в два этапа:

- построить отсчеты на отрицательной полуоси, начиная с нулевого отсчета;
- построить отсчеты на положительной полуоси, начиная с нулевого отсчета.

Оси координат отображаются отрезками прямых линий – вертикальной и горизонтальной, – которые пересекаются в пикселе отображения точки (0, 0). Реализация методов вывода основных штрихов, по-существу, сводится к задаче табулирования с заданным шагом с целью определения местоположения штриха на оси. Алгоритм построен с помощью двух циклов. Один цикл выставляет штрихи в отрицательном, а другой – в положительном направлениях. Поперечные размеры, то есть длина штриха, заданы в пикселях в сторону положительной и отрицательной полуоси (поля `PTicks` и `NTicks`). Следует отметить, что *Y*-координаты штрихов в этом случае остаются неизменными, их значения устанавливаются в начале процедуры (переменные `sY0` и `sY1`). А *X*-координаты пересчитываются в циклах. Текущее значение *X*-координаты хранится в переменной `tX`. В цикле значение этой переменной изменяется на величину шага.

Внутри функции вызывается процедура `MathToScr`, которая преобразует математические координаты точки пересечения осей в экранные. Текст реализации такой процедуры приведен в листинге 15.20.

Листинг 15.20. Процедура вывода основных штрихов по оси *OX*

```
void TAxes::XDrawTicks (TImage *aIm)
{
       // инициализация начальных значений
  int sX,
      sY0 = FCrossPnt.y - FX.PTicks,
       sY1 = FCrossPnt.y + FX.NTicks + 1,
      sY;
    // вывод штрихов в отрицательной стороне оси
  int iX = -1; // начальный номер штриха
  do  // делать цикл
```

```
   {
     // вычислить математическую X - координату
    long double tX = FPointCross.X + iX * FX.Step;
      // перевести в экранную X - координату
    MathToScr (tX, 0.0, sX, sY);
    if (sX < 0) break; // проверить на окончание
      // нарисовать штрих
    aIm->Canvas->MoveTo (sX, sY0);
    aIm->Canvas->LineTo (sX, sY1);
    iX--; // перейти к следующему номеру
   } while (sX >= 0);  // пока не подошли к 0

      // вывод штрихов в положительной стороне оси
  iX = 1; // начальный номер штриха
  do  // делать цикл
   {  // вычислить математическую X - координату
    long double tX = FPointCross.X + iX * FX.Step;
      // перевести в экранную X - координату
    MathToScr (tX, 0.0, sX, sY);
    if (sX >= aIm->Width) break;  //пока не правая граница
            // нарисовать штрих
    aIm->Canvas->MoveTo (sX, sY0);
    aIm->Canvas->LineTo (sX, sY1);
    iX++; // перейти к следующему номеру
   } while (sX < aIm->Width); //пока не правая граница
}
```

Вывод основных штрихов по оси *OY*

Процедура `YDrawTicks`, предназначенная для вывода штриховых линий осей координат, имеет один параметр типа `TImage`. Рассуждения о построении этой оси совершенно аналогичны предыдущему разделу. Принципиальное отличие от предыдущего метода в том, что для оси *OY* приходится учитывать, что на экране эта ось направлена вниз, и сохраняются неизменными *X*-координаты, а *Y*-координаты пересчитываются (см. листинг 15.21).

Листинг 15.21. Процедура вывода основных штрихов по оси *OY*

```
void TAxes::YDrawTicks (TImage *aIm)
{  // вывод основных штрихов по оси Oy
  int sY,
      sX0 = FCrossPnt.x - FY.NTicks,
      sX1 = FCrossPnt.x + FY.PTicks + 1,
      sX;
  int iY = -1;
  do  // вывод отсчетов в сторону уменьшения
   {long double tY = FPointCross.Y + iY * FY.Step;
    MathToScr (0.0, tY, sX, sY);
    if (sY < 0) break;
    aIm->Canvas->MoveTo (sX0, aIm->Height - 1 - sY);
    aIm->Canvas->LineTo (sX1, aIm->Height - 1 - sY);
    iY--;
   } while (sY >= 0);
  iY = 1;
  do  // вывод отсчетов в сторону увеличения
   {long double tY = FPointCross.Y + iY * FY.Step;
    MathToScr (0.0, tY, sX, sY);
    if (sY >= aIm->Height) break;
```

```
    aIm->Canvas->MoveTo (sX0, aIm->Height - 1 - sY);
    aIm->Canvas->LineTo (sX1, aIm->Height - 1 - sY);
    iY++;
   } while (sY < aIm->Height);
}
```

Вывод дополнительных штрихов по осям *OX* и *OY*

Процедуры `XDrawMinors` и `YDrawMinors` имеют один параметр типа TImage. Вывод малых штрихов выполняется аналогично выводу основных штрихов. Предложенная реализация процедур вывода малых штрихов базируется на алгоритме вывода основных штрихов. Доработка заключается в выполнении цикла заполнения малыми штрихами пространства между основными штрихами. Текст реализации этих процедур приведен в листинге 15.22 и 15.23.

Листинг 15.22. Процедуры вывода дополнительных штрихов по осям *OX*

```
void TAxes::XDrawMinors (TImage *aIm)
{
            // вывод дополнительных штрихов
  int sX,
      sY0 = FCrossPnt.y - FX.PmTicks,
      sY1 = FCrossPnt.y + FX.NmTicks + 1,
      sY;
  long double mStep = FX.Step / (FX.mCount + 1);
  int iX = 0;
  do  // вывод отсчетов в сторону уменьшения
   {long double tX = FPointCross.X + iX * FX.Step;
    for (int imX = 0; imX < FX.mCount; imX++)
                                              // малые отсчеты
     {tX -= mStep;
      MathToScr (tX, 0.0, sX, sY);
      if (sX < 0) break;  // не надо уже выводить малый штрих
      aIm->Canvas->MoveTo (sX, sY0);
      aIm->Canvas->LineTo (sX, sY1);
     }
    iX--;
   } while (sX >= 0);
  iX = 0;
  do  // вывод отсчетов в сторону увеличения
   {long double tX = FPointCross.X + iX * FX.Step;
    for (int imX = 0; imX < FX.mCount; imX++)
                                              // малые отсчеты
     {tX += mStep;
      MathToScr (tX, 0.0, sX, sY);
      if (sX >= aIm->Width) break;
                          // не надо уже выводить малый штрих
      aIm->Canvas->MoveTo (sX, sY0);
      aIm->Canvas->LineTo (sX, sY1);
     }
    iX++;
   } while (sX < aIm->Width);
}
```

Листинг 15.23. Процедуры вывода дополнительных штрихов по осям *OY*

```
void TAxes::YDrawMinors (TImage *aIm)
{   // вывод дополнительных штрихов
  int sY,
```

```
      sX0 = FCrossPnt.x - FY.NmTicks,
      sX1 = FCrossPnt.x + FY.PmTicks + 1,
      sX;
  long double mStep = FY.Step / (FY.mCount + 1);
  int iY = 0;
  do  // вывод отсчетов в сторону уменьшения
   {long double tY = FPointCross.Y + iY * FY.Step;
    for (int imY = 0; imY < FY.mCount; imY++)
                                              // малые отсчеты
     {tY -= mStep;
      MathToScr (0.0, tY, sX, sY);
      if (sY < 0) break;
      aIm->Canvas->MoveTo (sX0, aIm->Height - 1 - sY);
      aIm->Canvas->LineTo (sX1, aIm->Height - 1 - sY);
     }
    iY--;
   } while (sY >= 0);
  iY = 0;
  do  // вывод отсчетов в сторону увеличения
   {long double tY = FPointCross.Y + iY * FY.Step;
    for (int imY = 0; imY < FY.mCount; imY++)
                                              // малые отсчеты
     {tY -= mStep;
      MathToScr (0.0, tY, sX, sY);
      if (sY >= aIm->Height) break;
      aIm->Canvas->MoveTo (sX0, aIm->Height - 1 - sY);
      aIm->Canvas->LineTo (sX1, aIm->Height - 1 - sY);
     }
    iY++;
   } while (sY < aIm->Height);
}
```

Вывод подписей основных штрихов по оси *OX*

Процедура `XValueOut` имеет один параметр типа `TImage`. Вывод подписей штрихов повторяет процедуру вывода штрихов за той разницей, что в соответствующие координаты/позиции выставляется текст/подпись с помощью стандартного метода `TextOut`. Подпись соответствует числовому значению математической *X*-координаты.

Вывод значений под основными штрихами осей координат можно реализовать подобно алгоритму вывода самих штрихов. Просто операторы формирования изображения штриха заменяются операторами вывода текста, соответствующего данному штриху. При этом необходимо учитывать оформление. Подписи штрихов по оси *OX* желательно размещать так, чтобы штрих оказывался посередине текста. Отступ от штриха до текста также можно было задавать в соответствующем поле класса. Здесь же это расстояние задано минимальным значением и непосредственно в процедуре. Описание приведено в листинге 15.24.

Листинг 15.24. Процедура вывода подписей основных штрихов по оси *OX*

```
void TAxes::XValueOut (TImage *aIm)
{   // вывод подписей оси Ox
  int sX, sY,
      sYtop = FCrossPnt.y + FX.NTicks + 1 + 1;
  int iX = -1;
  do  // вывод подписей в сторону уменьшения
   {long double tX = FPointCross.X + iX * FX.Step;
```

```
    AnsiString _text = FloatToStrF (tX, ffFixed,
                                    10, FX.Digits);
    MathToScr (tX, 0.0, sX, sY);
    if (sX < 0) break;
    aIm->Canvas->TextOut (sX -
         aIm->Canvas->TextWidth (_text) / 2, sYtop, _text);
    iX--;
   } while (sX >= 0);
  iX = 1;
  do  // вывод подписей в сторону увеличения
   {long double tX = FPointCross.X + iX * FX.Step;
    AnsiString _text = FloatToStrF (tX,
                                    ffFixed, 10, FX.Digits);
    MathToScr (tX, 0.0, sX, sY);
    if (sX >= aIm->Width) break;
    aIm->Canvas->TextOut (sX -
         aIm->Canvas->TextWidth (_text) / 2, sYtop, _text);
    iX++;
   } while (sX < aIm->Width);
}
```

Вывод подписей основных штрихов по оси *OY*

Процедура `YValueOut` имеет один параметр типа `TImage`. Подписи штрихов по оси *OY* формируются аналогично оси *OX*, но обращены к штрихам другой стороной. Поэтому текст подписи приходится центрировать по высоте, а левый край текста задавать с учетом длины конкретной подписи так, чтобы правый край располагался от штриха на заданном расстоянии. Поэтому тексты этих процедур очень схожи, хотя имеют различия в определении координат верхней левой точки прямоугольной области для вывода текста подписи. Описание метода представлено в листинге 15.25.

Листинг 15.25. Вывод подписей основных штрихов по оси *OY*

```
void TAxes::YValueOut (TImage *aIm)
{  // вывод подписей оси Oy
  int sX, sY,
      sXleft = FCrossPnt.x - FY.NTicks - 1;
  int iY = -1;
  do  // вывод подписей в сторону уменьшения
   {long double tY = FPointCross.Y + iY * FY.Step;
    AnsiString _text = FloatToStrF (tY,
                                    ffFixed, 10, FX.Digits);
    MathToScr (0.0, tY, sX, sY);
    if (sY < 0) break;
    aIm->Canvas->
           TextOut (sXleft - aIm->Canvas->TextWidth(_text),
           aIm->Height - 1 - sY -
                      aIm->Canvas->TextHeight (_text) / 2,
           _text);
    iY--;
   } while (sY >= 0);
  iY = 1;
  do  // вывод подписей в сторону увеличения
   {long double tY = FPointCross.Y + iY * FY.Step;
    AnsiString _text = FloatToStrF (tY,
                                    ffFixed, 10, FX.Digits);
    MathToScr (0.0, tY, sX, sY);
```

```
    if (sY >= aIm->Height) break;
    aIm->Canvas->
           TextOut (sXleft – aIm->Canvas->TextWidth(_text),
           aIm->Height - 1 - sY –
                       aIm->Canvas->TextHeight (_text) / 2,
           _text);
    iY++;
   } while (sY < aIm->Height);
}
```

Вывод подписей оси *OX*

Процедура `XScriptOut` имеет один параметр типа `TImage`. Вывод подписей осей и их названий выполняются довольно просто. Подписи осей наносятся на их концы, а названия – на середину. Поэтому при выводе подписи оси *OX* необходимо отступить от оси на достаточное расстояние, например ниже подписей штрихов и чуть левее правого края области изображения. Вывод названия оси можно выполнить еще чуть ниже, но в середине области изображения. Описание метода представлено в листинге 15.26.

Листинг 15.26. Процедура вывода подписи оси *OX*

```
void TAxes::XScriptOut (TImage *aIm)
{   // вывод подписи оси 0x
  int sXleft = aIm->Width –
                     aIm->Canvas->TextWidth (FX.Script) - 2,
      sYtop = FCrossPnt.y + FX.NTicks + 1 + 1 + 1 +
                              aIm->Canvas->TextHeight ("W");
  aIm->Canvas->TextOut (sXleft, sYtop, FX.Script);
}
```

Вывод подписи оси *OY*

Процедура `YScriptOut` имеет один параметр типа TImage. Все что говорилось о нанесении подписей к оси *OX*, относится и к оси *OY*. Но есть принципиальное отличие. Текст подписей к штрихам на оси *OX* вытянут вдоль оси, а на оси *OY* они располагаются поперек. Это принципиальное отличие, так как крайняя нижняя кромка подписей к штрихам определяется легко; для этого достаточно выполнить простые вычисления. Другая ситуация с осью *OY*. Чтобы узнать левую крайнюю кромку подписей к штрихам, требуется перебрать все эти подписи и вычислить для них левый край, и найти самый левый. Поэтому процедура нанесения подписи к оси *OY* несколько сложнее и содержит фрагменты кода нанесения подписей к штрихам. Описание метода представлено в листинге 15.27.

Листинг 15.27. Процедура вывода подписи оси *OY*

```
void TAxes::YScriptOut (TImage *aIm)
{   // вывод подписи оси 0y
  int sXleft = FCrossPnt.x - FY.NTicks - 1,
      sYtop = 1;
  int sXmin = aIm->Width;
  int sX, sY;

  int iY = -1;
  do  // перебор подписей в сторону уменьшения
   {long double tY = FPointCross.Y + iY * FY.Step;
    AnsiString _text = FloatToStrF (tY, ffFixed,
                                    10, FX.Digits);
```

```
    MathToScr (0.0, tY, sX, sY);
    if (sY < 0) break;
    if (sXmin > sXleft - aIm->Canvas->TextWidth(_text))
      sXmin = sXleft - aIm->Canvas->TextWidth(_text);
    iY--;
   } while (sY >= 0);
  iY = 1;
  do  // перебор подписей в сторону увеличения
   {long double tY = FPointCross.Y + iY * FY.Step;
    AnsiString _text = FloatToStrF (tY, ffFixed,
                                    10, FX.Digits);
    MathToScr (0.0, tY, sX, sY);
    if (sY >= aIm->Height) break;
    if (sXmin > sXleft - aIm->Canvas->TextWidth(_text))
      sXmin = sXleft - aIm->Canvas->TextWidth(_text);
    iY++;
   } while (sY < aIm->Height);

  aIm->Canvas->TextOut (sXmin -
                    aIm->Canvas->TextWidth (FY.Script) - 1,
                    sYtop, FY.Script);
}
```

Вывод названия оси *OX*

Процедура `XTitleOut` имеет один параметр типа `TImage`. При расчете положения названия оси *OX* применимы те же рассуждения, что и при выводе подписи к оси. Отличие состоит в том, что текст названия должен располагаться от оси дальше и посередине. А в остальном реализации этих процедур схожи. Описание метода представлено в листинге 15.28.

Листинг 15.28. Процедура вывода названия оси *OX*

```
void TAxes::XTitleOut (TImage *aIm)
{
             // вывод название оси 0x
  int sXleft = (aIm->Width -
                    aIm->Canvas->TextWidth (FX.Title)) / 2,
      sYtop = FCrossPnt.y + FX.NTicks + 1 + 1 + 5 +
                            aIm->Canvas->TextHeight ("W");
  aIm->Canvas->TextOut (sXleft, sYtop, FX.Title);
}
```

Вывод названия оси *OY*

Процедура `YTitleOut` имеет один параметр типа `TImage`. Так же как и в случае подписи к оси *OY* здесь ситуация несколько ухудшается. Текст подписей штрихов выступает на различное расстояние от оси в зависимости от формата вывода и шрифта, поэтому предварительно требуется перебрать выводимые подписи к штрихам, но не выводить их, а оценить насколько будет выступать влево самая длинная подпись.

Другая трудность – в длине названия. Если текст подписи к оси *OY*, как правило, короток (5–8 символов), то название оси может быть достаточно длинным. Поэтому такую надпись следует наносить не горизонтально, а вертикально. Это обстоятельство еще несколько усложняет процедуру вывода названия оси *OY*.

Как уже отмечалось раньше, библиотека классов *VCL* позволяет быстро строить приложения и управлять различными элементами на высоком уровне. При этом оставлен доступ к этим элементам средствами процедур *WinAPI*, то есть управление этими элементами на низком уровне. И даже использование единичных функций *WinAPI* может значительно расширить возможности приложения, разработанного в *C++ Builder*, незначительно увеличив сложность разработки приложения.

Так как, к сожалению, класс *TCanvas* не предоставляет простой возможности вывода текста под произвольным углом, или хотя бы в четырех основных направлениях – горизонтальном и вертикальном. Но использование нескольких функций *WinAPI* позволяет устранить этот недостаток. Тогда функция вывода названия оси *OY*, подобно функции вывода подписи, на начальном этапе должна определить самый левый выступ подписи штрихов для определения точки вывода по горизонтали. По вертикали местоположение рассчитывается так, чтобы название размещалось в середине. Порядок вывода названия может быть следующим:

- посредством функции *GetObject* получить параметры шрифта, используемого *Canvas* в текущий момент, в структуру типа `LOGFONT`;
- в этой структуре изменить значение поля `lfEscapement`, отвечающего за угол вывода текста. Задать значение `lfEscapement = 2700`, соответствующее углу 270°, то есть выводу текста сверху вниз;
- по данным структуры создать новый объект-шрифт;
- в *Canvas* заменить текущий шрифт на вновь созданный. Текущий шрифт запомнить в отдельной переменной;
- вывести текст в указанной позиции;
- вернуть прежний шрифт в *Canvas*;
- уничтожить новый шрифт за ненадобностью в дальнейшем использовании.

Реализация процедуры вывода названия вдоль вертикальной оси приведена в листинге 15.29.

Листинг 15.29. Процедура вывода названия оси Oy

```
void TAxes::YTitleOut (TImage *aIm)
{ // вывод названия оси Oy
  int sXleft = FCrossPnt.x - FY.NTicks - 1,
      sYtop = (aIm->Height -
                    aIm->Canvas->TextWidth (FY.Title)) / 2;
  int sXmin = aIm->Width;
  int sX, sY;

  int iY = -1;
  do  // перебор подписей в сторону уменьшения
   {long double tY = FPointCross.Y + iY * FY.Step;
    AnsiString _text = FloatToStrF (tY, ffFixed,
                                    10, FX.Digits);
    MathToScr (0.0, tY, sX, sY);
    if (sY < 0) break;
    if (sXmin > sXleft - aIm->Canvas->TextWidth(_text))
      sXmin = sXleft - aIm->Canvas->TextWidth(_text);
    iY--;
   } while (sY >= 0);
  iY = 1;
  do  // перебор подписей в сторону увеличения
   {long double tY = FPointCross.Y + iY * FY.Step;
```

```
    AnsiString _text = FloatToStrF (tY, ffFixed,
                                    10, FX.Digits);
    MathToScr (0.0, tY, sX, sY);
    if (sY >= aIm->Height) break;
    if (sXmin > sXleft - aIm->Canvas->TextWidth(_text))
      sXmin = sXleft - aIm->Canvas->TextWidth(_text);
    iY++;
   } while (sY < aIm->Height);

  HFONT newFont = 0, oldFont = 0;
  LOGFONT lfont;
        // создаем LOGFONT из текущего шрифта
  GetObject (aIm->Canvas->Font->Handle, sizeof(LOGFONT),
             (LPVOID)&lfont);
        // вывод текста под 270 градусов
  lfont.lfEscapement = 2700;
        // выбрать шрифт Arial (True Type)
  if ((newFont = CreateFontIndirect (&lfont)) != NULL )
   {oldFont = SelectObject (aIm->Canvas->Handle, newFont);
                                  // заменить шрифт на новый
        // вывести текст
    aIm->Canvas->TextOut (sXmin - 5, sYtop, FY.Title);
        // вернуть старый шрифт
    SelectObject (aIm->Canvas->Handle, oldFont);
        // новый уничтожить за ненадобностью
    DeleteObject (newFont);
   }
}
```

Примечание

Для разработки приложения, обеспечивающего вывод графика функции и осей координат, достаточно объединить рассуждения и разработки, проведенные выше, и реализовать дополнительные методы. Вывод графика функции и осей координат будет выполняться на единый объект `Image`.

Задача 15.4. Требуется построить график эпициклоиды, уравнение которой задано в параметрическом виде:

$$\begin{cases} x = (a+b)\cdot\cos(t) - a\cdot\cos((a+b)\cdot t/a), \\ y = (a+b)\cdot\sin(t) - a\cdot\sin((a+b)\cdot t/a) \end{cases}.$$

Отдельно рассмотреть случай, когда b/a есть целое положительное число, и $t \in [0, 2\pi]$.

Решение

Подобные функции, когда и абсцисса, и ордината являются функциями от некоторого третьего параметра *t*, называются параметрическими. График такой функции представляет собой кривую на плоскости *XOY*, которая не имеет однозначного отображения ни на ось *OX*, ни на ось *OY*. Более того, такие кривые часто имеют точки самопересечения. Алгоритм решения данной задачи, то есть построение графика параметрической функции, подобен построению графика явно заданной функции с табулированием в математической области (см. листинг 14.7). При этом, конечно, есть несколько особенностей:

- пределы изменения параметра *t* задаются в зависимости от того, какой фрагмент кривой необходимо воспроизвести;
- шаг табулирования параметра *t* должен быть достаточно малым. Действительно, так как производится вычисление отдельно отстоящих точек кривой, то при соединении их отрезками получается график ломаной линии. Если шаг параметра достаточно мал, то и отрезки тоже малы, и ломаная достаточно качественно аппроксимирует кривую.

Для решения этой задачи также лучше разработать простейший класс, например, `TParamGraphic`. Для передачи пределов и шага табулирования можно использовать поля класса. Интерфейс класса, конструктор класса, метод построения графика и подпрограмма вычисления значения функции приведены в листинге 15.30–15.33. Пример работы такого приложения показан на рис. 15.5.

Листинг 15.30. Интерфейс класса `TParamGraphic` для построения графика функции, заданной в параметрическом виде

```
class TParamGraphic
{
private:
  int    FWidth;
  TColor FColor;
  long double FStart, FFinish;
  long double FStep;
protected:
  TMathPoint FuncPar (long double);
public:
  TParamGraphic();
  void __fastcall DrawGraphic (TImage *);
  __property long double Start = {write = FStart};
  __property long double Finish = {write = FFinish};
  __property long double Step = {write = FStep};
};
```

Листинг 15.31. Описание конструктора класса `TParamGraphic`

```
TParamGraphic::TParamGraphic ()
{
            // инициализация полей класса
 FWidth = 1;
 FColor = clBlack;

 FStart = 0.0;
 FFinish = 1.0;
 FStep = 0.1;
}
```

Листинг 15.32. Описание функции, предназначенной для вычисления значений

```
TMathPoint TParamGraphic::FuncPar (long double aT)
{ TMathPoint pnt;
  long double a = 1.5, b = 1.5;
  pnt.X = (a + b)*cos(aT) - a*cos((a + b)*aT / a);
  pnt.Y = (a + b)*sin(aT) - a*sin((a + b)*aT / a);
  return pnt;
}
```

Листинг 15.33. Построение графика параметрической функции

```
void __fastcall TParamGraphic::DrawGraphic (TImage *aIm)
```

```
{ aIm->Canvas->Pen->Color = FColor;
  aIm->Canvas->Pen->Width = FWidth;

    // расчет начальной точки графика
  long double currX, currY;
  long double t = FStart;

  TMathPoint pnt = FuncPar (t);

  int iX, iY;
  MathToScr (pnt.X, pnt.Y, iX, iY);
  aIm->Canvas->MoveTo (iX, aIm->Height - 1 - iY);
                        // установка графического курсора

  t = FStart + FStep;
  while (t <= FFinish + 0.5 * FStep)
   {pnt = FuncPar (t);  // расчет координат X и Y;
    MathToScr (pnt.X, pnt.Y, iX, iY);
                                  // и расчет координат экрана
    aIm->Canvas->LineTo (iX, aIm->Height - 1 - iY);
                               // рисование очередного отрезка
    t += FStep;
   };
}
```

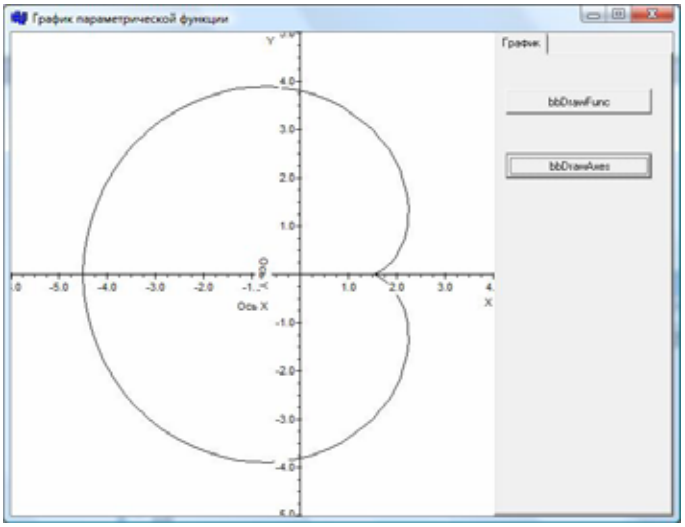


Рис. 15.5. Пример построения графика параметрической функции

15.4. Построение изолиний функции двух переменных

Математические функции и другая информация, представляющая собой функциональную зависимость, как правило, требует отображения в виде графиков. Но если многомерные функции не могут быть отображены непосредственно, то для функций одной и двух переменных разработано множество способов визуализации. И если функция одной переменной чаще всего отображается ломаной или гладкой линией, то для отображения функции двух переменных $z = F(x, y)$ используют или перспективное изображение поверхности, или *изолинии*. Но если формирование трехмерного изображения поверхности, да еще имеющего искажения с учетом пер-

спективы, является достаточно трудоемким процессом, то для формирования изолиний можно предложить простейший алгоритм.

ИЗОЛИНИЯ

Под *изолинией* понимается линия, лежащая на поверхности двумерной функции, точки которой удовлетворяют условию $z = const$.

Состав графика функции двух переменных в виде изолиний такой же, как и для графика одной переменной: линии графика и оси координат. Отличие только в процедуре формирования линий графика.

Функция двух переменных в общем виде записывается как $z = F(x, y)$. *Математическое решение* задачи построения изолиний заключается в формировании множества точек (x, y), удовлетворяющих равенству $F(x, y) = const$. *Графическое решение* этой задачи заключается в определении линии пересечения поверхности функции и горизонтальной плоскости, пересекающей ось *OZ* на заданном уровне *const*.

Замечание

В общем случае линия пересечения может состоять из нескольких замкнутых и незамкнутых кривых, так как функция может иметь несколько «холмов», которые «протыкают» горизонтальную плоскость.

Построение изолиний для функции двух переменных является непростой задачей. Но для формирования изображения изолиний на растровом устройстве можно реализовать достаточно простой алгоритм.

Алгоритм формирования изображения изолиний функции двух переменных

При работе с графикой удобно использовать объекты типа `TBitMap` (битовый образ).

БИТОВЫЙ ОБРАЗ

Битовый образ представляет собой находящуюся в памяти компьютера (и следовательно, невидимую) графическую поверхность, на которой программа может сформировать изображение. Содержимое битового образа (картинка) легко и, что особенно важно, быстро может быть выведено на поверхность формы или области вывода иллюстрации (`Image`). Поэтому в программах битовые образы обычно используются для хранения небольших изображений, например картинок командных кнопок. Иногда битовые изображения называют *растровыми* изображениями, подчеркивая тот факт, что его можно рассматривать как совокупность строк растра (горизонтальных линий развертки).

Напоминание

Напомним, что любая печатная картинка, начиная с появления цинкографического способа воспроизведения изображений, состоит из отдельных точек, то есть представляет собой *растр*.

Качество печатных изображений напрямую зависит от подробности растра. Считается, что разрешение в 300 точек на дюйм (*dpi*) достаточно для самой высококачественной полиграфической печати.

РАСТР ЭКРАННЫЙ

Растр экранный представляет собой матрицу пикселей, расположенных в узлах квадратной решетки экрана дисплея.

Как известно, любая кривая линия, начерченная на листе бумаги карандашом или мелом на доске, – это лишь образ некоторого математического объекта. Начерченная линия обязательно имеет толщину. А вот математическая линия нет. Потому, как бы тонко не затачивать карандаш, начерченная линия будет грубой и толстой, сердцевину которой составляет «бесконечно тонкая» математическая линия. Такая же ситуация и с *растровым* изображением линии.

Отличие

Начерченная карандашом линия является непрерывной гладкой. А растровое изображение представляет собой множество отдельных пятен.

Конечно, растущее качество современных мониторов и принтеров позволяет забывать об этом «недостатке». Но при разработке алгоритмов отображения этот факт помогает ограничиваться более примитивными алгоритмами, так как усложнение алгоритмов не всегда дает улучшение изображения. В качестве примера можно привести общеизвестный алгоритм Брезенхема, который позволяет определить достаточное множество пикселей для формирования изображения отрезка прямой линии. Основную идею этого алгоритма продемонстрируем на следующем примере.

На рис. 15.6 представлен фрагмент растра, изображающего отрезок линии. Белой линией обозначена математическая линия, которую отображает растровое устройство. Черные круги – это условное обозначение точек-пикселей, которые формируют изображение линии. А сетка, на фоне которой нанесены пиксели, обозначает центры пикселей. Как это видно из рисунка 15.6, в любом случае при вычерчивании линии на растровом устройстве приходится ставить ей в соответствие набор близлежащих пикселей. Вполне понятно, что таким же набором пикселей отображается некоторое множество отрезков, близких к данному отрезку.

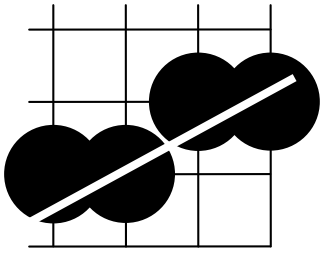


Рис. 15.6. Фрагмент растра, изображающего отрезок линии

При построении графиков функций приходится производить пересчет математических координат в координаты экрана и обратно. Пиксель занимает целую область на экране. Но в компьютерной графике часто *пикселю ставят в соответствие его геометрический центр*.

РАСТР МАТЕМАТИЧЕСКИЙ

В математике считается, что растр – это матрица точек, которые располагаются в узлах прямоугольной решетки.

Тогда узлам этой экранной решетки в математической области можно поставить в соответствие узлы некоторой прямоугольной решетки экрана.

Задача 15.5. Разработать алгоритм построения изображения графика функции $z = x^2 - y^2$ в виде цветовых областей, в соответствии с заданной полутоновой или цветной палитрой.

Решение

Опираясь на рассуждения, приведенные выше, и алгоритмы, разработанные для графика функции одной переменной, предлагается стратегия, логика которой может быть описана следующими шагами.

- Посредством двойного цикла перебрать пиксели области отображения.
- Координаты каждого пикселя пересчитать в координаты узлов прямоугольной решетки математической области.
- В каждом узле математической решетки вычислить значения заданной функции.
- Поставить в соответствие вычисленному значению цвет.
- Задать текущему пикселю полученный цвет.

Такой алгоритм очень прост. Некоторую сложность представляет *процесс создания палитры*.

Примечание

Для задания палитры необходимо создать массив из переменных типа `TColor` и заполнить его плавно изменяющимися цветовыми значениями. Причем для разных интервалов значений функции можно задавать разные зоны плавно изменяющихся цветов, например для положительных значений функции и для отрицательных.

Для установки соответствия цвета и значения функции можно переводить текущую величину из пространства значений функции в пространство индексов элементов палитры, и цвет соответствующей ячейки палитры выбирать в качестве цвета пикселя.

Для решения этой задачи также можно создать простейший класс (`TFuncXYRegionGraphic`). Такой класс будет содержать конструктор, метод отображения графика функции, а также метод пересчета из значений функции в цвет. Для простоты разработки класса длина массива с цветами отображения графика и некоторые другие значения заданы в виде констант. Описание интерфейсной части и реализация процедур этого класса представлено в листингах 15.34–15.38, а пример работы программы – на рис. 15.7.

Листинг 15.34. Интерфейсная часть класса отображения графика двумерной функции в виде цветовых областей

```
const coNumColors = 10;  // количество цветов
const long double coMinValue = -100, coMaxValue = 100;
                              // пределы изменения палитры
```

```
const TColor coLowColor = clMaroon, coHigh = clLime;
                                // цвет ниже и выше пределов

class TFuncXYRegionGraphic
  {
  private:
    TColor FColors [10];
  protected:
    long double Func1 (long double, long double);
  public:
    TFuncXYRegionGraphic();
    TColor __fastcall GetColor (long double);
    void __fastcall DrawGraphic (TImage *);
  };
```

Листинг 15.35. Описание конструктора класса `TFuncXYRegionGraphic`

```
TFuncXYRegionGraphic::TFuncXYRegionGraphic ()
{FColors [0] = (TColor) 0xFF0000; // инициализация цветов
 FColors [1] = (TColor) 0xFF3030;
 FColors [2] = (TColor) 0xFF6060;
 FColors [3] = (TColor) 0xFF9090;
 FColors [4] = (TColor) 0xFFB0B0;

 FColors [5] = (TColor) 0xB0FFB0;
 FColors [6] = (TColor) 0x90FF90;
 FColors [7] = (TColor) 0x60FF60;
 FColors [8] = (TColor) 0x30FF30;
 FColors [9] = (TColor) 0x00FF00;
}
```

Листинг 15.36. Описание метода класса `TFuncXYRegionGraphic`, предназначенного для вычисления значения функции в заданной точке с координатами `aX` и `aY`

```
long double TFuncXYRegionGraphic::Func1 (long double aX,
                                         long double aY)
{ return aX * aX - aY * aY;
}
```

Листинг 15.37. Описание метода класса `TFuncXYRegionGraphic`, предназначенного для вычисления цвета растра по значению функции `aZ`

```
TColor __fastcall TFuncXYRegionGraphic::GetColor
                                        (long double aZ)
{
      // вычисление цвета по значению функции aZ
  int aux = coNumColors * (aZ - coMinValue) /
                              (coMaxValue - coMinValue);
  if (aux >= coNumColors) return coLowColor;
  if (aux < 0) return coHigh;
  else return FColors [aux];
}
```

Листинг 15.38. Описание метода класса `TFuncXYRegionGraphic`, предназначенного для построения графика функции в виде цветовых областей

```
void __fastcall TFuncXYRegionGraphic::DrawGraphic (TImage *aIm)
{ long double x0, y0, f0;
  aIm->Canvas->Lock();
  for (int iY = 0; iY < aIm->Height; iY++)
   {
```

```
    for (int iX = 0; iX < aIm->Width; iX++)
     {ScrToMath(iX, iY, x0, y0);
      f0 = Func1 (x0, y0);
      aIm->Canvas->Pixels [iX][iY] = GetColor (f0);
     }
    Application->ProcessMessages();
   }
  aIm->Canvas->Unlock();
}
```

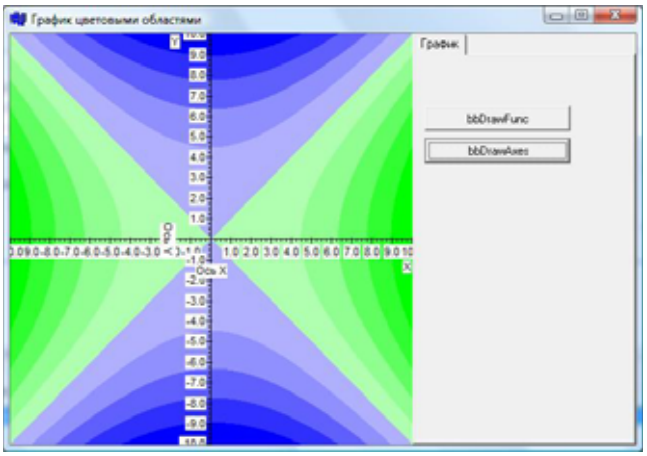


Рис. 15.7. Пример построения графика двумерной функции

Задача 15.6. Разработать программу построения изображения графика функции $z = x^2 - y^2$ в виде изолиний.

Решение

При построении полутонового изображения графика функции приходится вычислять значения в узлах решетки, поэтому истинная математическая кривая лавирует между этими узлами и, возможно, не проходит ни через один из узлов. Полученные значения функции тоже представляют собой набор отдельных дискретных значений, причем ни одно из значений может не равняться заданному уровню $z = const$. То есть все значения функции, вычисленные в узлах решетки, делятся на два множества: те, которые больше заданного $z = const$, и те, которые меньше. И только отдельные отсчеты, возможно, точно равны заданному уровню. То же относится и к любым четырем соседним узлам, расположенным в вершинах прямоугольника. Большое количество прямоугольников находится или внутри, или вне изолинии.

На рис. 15.8 представлен фрагмент растра, изображающего отрезок изолинии. Отрезком прямой линии обозначена математическая линия, которую необходимо отобразить на растровом устройстве. Черный круг – это условное обозначение основного пикселя, который подвергается анализу, а белые круги – соседние пиксели, участвующие в расчетах цвета основного. Сетка обозначает центры всех пикселей растра. Согласно рисунку 15.8, часть прямоугольников, образованных центрами пикселей, пересекаются изолинией. То есть математическая кривая, соответствующая изолинии, проходит рядом с вершинами, пересекая стороны прямоугольника. Это означает, что значения функции в некоторых вершинах этого прямоугольника больше заданного уровня, а в остальных – меньше. Поэтому одну из вершин прямоугольника, например верхнюю левую, можно принять за представителя этой облас-

ти. И для обозначения пути прохождения изолинии закрашивать верхний левый пиксель, если изолиния проходит между четверкой соседей.

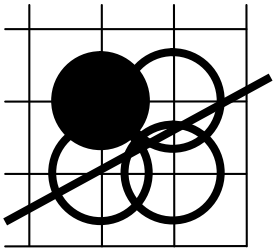


Рис. 15.8. Фрагмент растра и отрезка изолинии

По результатам проведенного анализа можно построить следующий алгоритм отображения изолинии:

- перебрать двойным циклом все пиксели холста;
- выполнить пересчет экранных координат в математические для каждого писеля и его трех соседей;
- вычислить значения функции в полученных четырех точках математической области;
- закрасить текущий пиксель, если значение функции в точке, соответствующей ему, и хотя бы одно значение функции, соответствующее соседним пикселям, лежат по разные стороны от заданной величины *const*. Если же все значения лежат по одну сторону от величины *const*, то пиксель не закрашивается;
- перейти к следующему пикселю.

Для решения задачи построения изолиний функции также удобнее разработать класс или дополнить ранее разработанный класс методом отображения функции изолиниями. Для построения нескольких изолиний дополнить интерфейс списком, например `TListBox` (с именем `lbLevels`), в котором хранить значения требуемых уровней.

Для построения изолиний на форме приложения можно подготовить кнопку, например, с именем `bbDrawFunc` (построить график функции). А в обработчике события `OnClick` этой кнопки вызвать соответствующие методы класса. И, конечно, для работы с классовой переменной, эту классовую переменную (например, с именем `FFunc`) нужно *объявить в списке глобальных* переменных.

Описание обработчика события `OnClick` этой кнопки `bbDrawFunc` формы будет содержать один цикл (по числу элементов `lbLevels`), в котором вызывается метод класса `DrawGraphic`, предназначенный для построения изображения соответствующей изолинии (см. листинг 15.40). Код обработчика события представлен в листинге 15.39, а пример работы такого приложения – на рис. 15.9.

Листинг 15.39. Обработчик события `OnClick` кнопки `bbDrawFunc` формы для формирования изолиний функции двух переменных

```
void __fastcall TfoMain::bbDrawFuncClick(TObject *Sender)
{// построение изолиний для уровней, заданных в lbLevels
  for (int i = 0; i < lbLevels->Count; i++)
   { FFunc.DrawGraphic (imGraph,
               StrToFloat (lbLevels->Items->Strings [i]));
    Application->ProcessMessages();
   }
}
```

Листинг 15.40. Описание метода класса `TFuncXYRegionGraphic`, предназначенного для формирования изолиний функции двух переменных

```
void __fastcall TFuncGraphic::DrawGraphic
                          (TImage *aIm, long double aLevel)
{ long double x0, y0, x1, y1;
  long double f0, f1, f2, f3;
  aIm->Canvas->Lock();
  for (int iY = 0; iY < aIm->Height; iY++)
   {ScrToMath(0, iY, x0, y0); ScrToMath(1, iY+1, x1, y1);
    f0 = Func1 (x0, y0); f2 = Func1 (x0, y1);
    for (int iX = 0; iX < aIm->Width; iX++)
     {ScrToMath(iX+1, iY+1, x1, y1);
      f1 = Func1 (x1, y0); f3 = Func1 (x1, y1);
      if (f0 == aLevel ||
         ((f0 > aLevel) &&
          (f1 < aLevel || f2 < aLevel || f3 < aLevel) ||
          (f0 < aLevel) &&
          (f1 > aLevel || f2 > aLevel || f3 > aLevel)))
        aIm->Canvas->Pixels[iX][aIm->Height - iY - 1] = FColor;
      f0 = f1; f2 = f3;
     }
   }
  aIm->Canvas->Unlock();
}
```

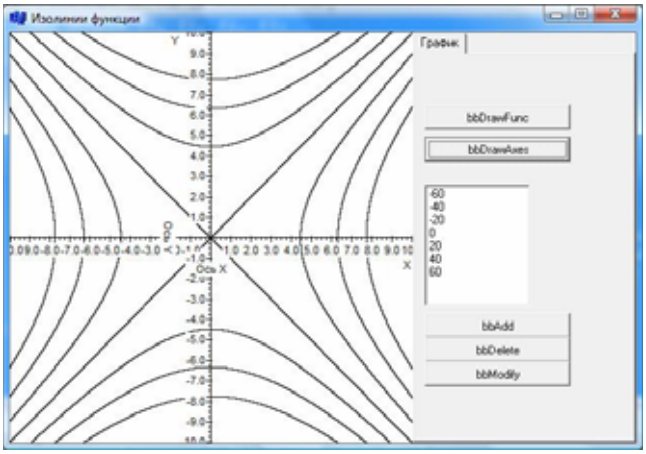


Рис. 15.9. Пример работы программы построения изолиний

Конечно, такое формирование изображений изолиний не самое эффективное, но его недостатки несколько компенсируются простотой реализации.



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. Доработать приложение построения графика функции $y = F(x) = e^{-x} \cdot Sin(2 \cdot x)$ на интервале [–3,2; 3,5]. На форме программы разместить четыре окна однострочных редакторов с метками, в которых пользователь будет задавать математическую область отображения графика. В процедурах вывода графика и осей задавать математичекую область, считывая данные из однострочных редакторов.

2. Требуется построить график функции $y = F(x) = e^{-x} \cdot Sin(2 \cdot x)$ на интервале [–3,2; 3,5] с использованием класса «Оси координат».

Рекомендация: вывод графика функции и осей координат выполнять на единый объект `Image`.

3. Доработать приложение для возможности построения нескольких графиков функций одновременно на интервале, указанном пользователем.

Рекомендация: исправить подпрограмму вычисления значения функции так, чтобы для заданного аргумента подпрограмма возвращала массив значений, соответствующих различным функциям. Исправить процедуру вывода линии графика так, чтобы одновременно выполнялось построение нескольких кривых.

4. Доработать приложение для возможности построения графика произвольной функции на интервале, указанном пользователем.

Рекомендация: использовать динамическую библиотеку типа `DLL`. Создать отдельный проект по разработке динамической библиотеки. Библиотека может содержать единственную подпрограмму, вычисляющую значение функции. Само приложение исправить так, чтобы график строился для функции, вызываемой из динамической библиотеки.

5. Доработать приложение для возможности построения нескольких произвольных графиков функций на интервале, указанном пользователем.

Рекомендация: использовать динамическую библиотеку типа `DLL`. Создать отдельный проект по разработке динамической библиотеки. Библиотека может содержать несколько подпрограмм, вычисляющих значения соответствующих функций. Помимо этих подпрограмм, библиотека должна содержать подпрограмму, возвращающую список с названиями функций или уравнениями, записанными в текстовом виде. Также список должен содержать указатели на сами процедуры вычисления значений функций.

Отображение картографической информации 16

Построения графиков функций одной и двух переменных, по сути, является процессом формирования изображения, состоящего из линий, областей и текстов, и адекватно отображающего исходную математическую информацию. Аналогичным образом можно формировать и другие изображения, например *картографическую* информацию.

Картографические системы составляют основу географических информационных систем (ГИС), в которых главными источниками информации служат различные геоизображения, содержащие пространственные данные. Главным преимуществом ГИС является их *способность определять и учитывать пространственные связи* между объектами. Геоинформационные технологии находят в настоящее время широкое применение и активно развиваются.

Примечание

Особенностью картографического изображения является то, что эти объекты имеют точные географические координаты. Поэтому вид таких объектов и их местоположение жестко определены, и при любом масштабе изображения оно должно адекватно отражать исходную информацию.

Вполне понятно, что карту можно рассматривать как набор изображений разной сложности. Разнотипные элементы карты отображаются определенными графическими изображениями.

Замечание

Здесь явно просматривается та особенность прикладной математики, когда разнообразные объекты воспринимаются как однородные и обрабатываются единообразно и даже, возможно, в цикле.

Для создания изображения карт достаточно поставить в соответствие каждому *географическому* объекту *графический* объект. То есть на карту нанести изображение, которое определяет местоположение связанного с этим изображением объекта относительно других объектов в некоторой системе координат.

Можно составить примерный перечень проблем, которые приходится решать при отображении картографической информации:

- адаптация изображения к изменениям размера окна или масштаба;
- предоставление различных вариантов отображения объектов:
  - отображение некоторых объектов единым размером независимо от масштаба изображения;
  - отображение некоторых объектов размером, соответствующим масштабу изображения;
  - изменение детализации изображения отдельных объектов в зависимости от масштаба изображения;
- построение осей координат вне изображения. Нанесение сетки координат на изображение.

Карта может отображать достаточно большой географический регион и содержать большое количество графических объектов. Но, независимо от насыщенности картографического изображения, процесс его формирования всегда одинаков.

Формирование изображения карт

Процесс формирования картографического изображения состоит из следующих шагов:

а) загрузка информации о географических объектах, в том числе об их местоположении;
б) отображение всех объектов на изображении карты.

При создании программы всегда следует думать, нельзя ли ее каким-либо образом обобщить, то есть сделать более универсальной с возможностями расширения спектра действий. И потому разработку приложения целесообразно вести в рамках технологии ООП, то есть начинать с разработки класса.

Задача 16.1. Требуется разработать класс, позволяющий отображать картографическую информацию о системах дорог, связывающих между собой географические объекты двух видов: населенные пункты (города/поселки) и архитектурные сооружения (здания/дома).

Решение

Разнообразие географических объектов приводит к задачам выполнения однотипной работы над разнотипными данными. Поэтому изначально следует проектировать иерархическое семейство классов. При этом поля будут создаваться в каждом дочернем классе индивидуально. И работа с этими полями будет скрыта в методах дочернего класса.

Поскольку карту также можно считать географическим объектом, для отображения карты необходимо создать класс. Предлагается следующая стратегия решения задачи:

1) разработать базовый класс (назовем его `TBaseElem`), который должен быть независим от вида географического объекта;
2) определить характеристики, необходимые для описания картографических данных: изображения математических объектов: «система дорог», «архитектурные сооружения»;
3) разработать структуру файлов с картографическими данными;
4) разработать классы, предназначенные для:
   а) отображения карты (с именем `TMap`);
   б) системы дорог (с именем `TRoad`);
   в) архитектурных сооружений (с именем `TBuilding`);
5) разработать интерфейс приложения.

Описание базового класса `TBaseElem`

Базовый класс должен быть независим от вида географического объекта, следовательно, в нем *будут отсутствовать* поля, характеризующие объект, так как эти поля индивидуальны.

Как показано выше, для формирования картографического изображения необходимо выполнять два действия. Естественно, базовый класс и должен содержать два метода, назовем их: `LoadElem` и `ShowElem`. Так как сложно найти общие действия в процессе визуализации разнородных объектов, то следует эти методы не реализовывать в базовом классе, а сделать их *абстрактными* и, следовательно, *виртуальными*.

Помимо этих методов, конечно, необходимо реализовать конструктор и деструктор для описания правильного порядка создания и уничтожения объектов. Так как инициализация полей класса, возможно, потребуется в разных методах, то было бы логично создать отдельный абстрактный метод `InitFields`.

Примечание

Методы виртуальные, то есть предполагается, что в дальнейшем эти методы будут заменяться вызовами конкретных методов, уже привязанных к соответствующему географическому объекту.

Интерфейс абстрактного базового класса представлен в листинге 16.1.

Листинг 16.1. Интерфейс абстрактного базового класса `TBaseElem` с обозначением действий, необходимых для формирования картографического изображения

```
class TBaseElem
 {private:
  protected:
  public:
     __fastcall TBaseElem ();  // конструктор
     __fastcall ~TBaseElem ();  // деструктор
     virtual void __fastcall InitFields () = 0;
     virtual void __fastcall LoadElem (ifstream &) = 0;
     virtual void __fastcall ShowElem (TImage *) = 0;
 };
```

МЕТОДЫ КЛАССА `TBaseElem`:

- `__fastcall TBaseElem ();` – конструктор класса;
- `__fastcall ~TBaseElem ();` – деструктор класса;
- `virtual void __fastcall InitFields () = 0;` – метод класса для инициализации полей класса;
- `virtual void __fastcall LoadElem (ifstream &) = 0;` – метод загрузки, который имеет один входной параметр с типом файловой переменной. Этот параметр будет связываться с файлом, содержащим данные об объекте;
- `virtual void __fastcall ShowElem (TImage *) = 0;` – метод отображения, который имеет входной параметр с типом `Image`. Этот параметр будет определять место для размещения картографического изображения.

Абстрактные методы

Абстрактные методы не имеют реализации. Но задание им параметра «абстрактный» вынудит разработчика наследника ПЕРЕКРЫТЬ все эти методы, чтобы можно было создать объекты такого класса-наследника.

Для любой программы по обработке и отображению информации большого объема необходимо тщательно прорабатывать структуру хранения информации на диске и в оперативной памяти. Это важно для достижения компромисса:

- универсальность – программа способна отображать на карте широкий спектр объектов;
- простота реализации – код не является очень громоздким.

Одним из элементов, часто отображаемым на картах являются *дороги*, так как они помогают ориентироваться при перемещении с помощью транспортных средств и характеризуют возможные каналы связи. Также зачастую параллельно с дорогами строятся и другие протяженные элементы цивилизации, например линии электропередачи. Такие мелочи, как ухабы и колдобины, не заносятся на карту. И поэтому независимо от класса и качества покрытия все дороги на картах выглядят плавно изгибающимися линиями, изредка имеющими четко обозначенный угол в том месте, где дорога имеет крутой поворот. Поэтому в предлагаемом проекте ДОРОГИ БУДУТ ОТОБРАЖАТЬСЯ КРИВЫМИ БЕЗЬЕ.

Кривая Безье

Кривая Безье – это кусочно-полиномиальная кривая или *сплайн*. Звенья такого сплайна чаще всего являются полиномами *третьей степени*. Форма каждого звена определяется характеристической ломаной из четырех точек: две задают концевые точки звена, а две другие определяют форму этого кубического звена. Подробнее познакомиться с кривыми Безье можно в главе 17.

Как и отдельные звенья, так и вся кривая Безье будет определяться положением узлов характеристической ломаной. Проще говоря, такая кривая будет описываться вершинами характеристической ломаной. Для кодирования дороги предлагаются следующие показатели:

- код объекта типа дорога;
- толщина линии, отображающей траекторию дороги (задается в пикселях);
- цвет линии (целое число в шестнадцатеричном виде);
- количество вершин, описывающих характеристическую ломаную линию (вершин должно быть $M = 3N + 1$);
- $2M$ вещественных чисел, где каждая пара чисел определяет вершину ломаной линии.

Архитектурные сооружения, или ЗДАНИЯ, БУДУТ ОТОБРАЖАТЬСЯ МНОГОУГОЛЬНИКАМИ, ОТРАЖАЮЩИМИ ПЛАН СТРОЕНИЯ. Для кодирования зданий предложим следующие показатели:

- код объекта типа здания;
- толщина линии контура многоугольника;
- цвет лини контура (целое число в шестнадцатеричном виде);
- цвет лини заливки многоугольника (целое число в шестнадцатеричном виде);
- количество вершин M, описывающих многоугольник;
- $2M$ вещественных чисел, где каждая пара чисел определяет вершину многоугольника.

ОБЩАЯ структура ФАЙЛА

Общая структура файла с закодированной информацией предлагается следующая:

- первая строка – четыре вещественных числа, определяющие регион, который должен отображаться после загрузки карты; координаты задаются в порядке x_{min}, y_{min}, x_{max}, y_{max};
- вторая строка – одно целое число, определяющее количество географических объектов, закодированных в файле;
- последующие строки – в каждой строке закодированная информация об одном объекте:
  - первым элементом в строке стоит целое число – код объекта;
  - структура каждой строки в остальной части произвольная и имеет разный формат для различных объектов.

Примечание

Построчное деление здесь условно. Чтение будет выполняться процедурами потокового чтения из текстового файла. Поэтому важен лишь порядок следования данных, а не их расположение по строкам. Также важно, чтобы все данные, описывающие единственный объект, были считаны соответствующей процедурой полностью. То есть, когда программа перейдет к чтению информации о другом объекте, на ее вход поступал *код* этого нового объекта.

Для правильного соответствия размеров объектов на карте и первоначального масштаба отображения, необходимо оценить координаты и размеры объектов. Такую оценку пределов изменения считываемых данных можно выполнить на этапе загрузки или в отдельной процедуре. Здесь предлагается более простой вариант:

- *пределы* первоначального отображения данной карты хранятся в файле карты и задаются в виде *первых четырех чисел* файла;
- затем задается целое число, указывающее *количество объектов* на карте;
- потом располагается список данных обо всех объектах.

Пример такого файла приведен в рисунке 16.1. Здесь объект типа *дорога* имеет код 1, а *здание* – 2.

```
-1000 -1000 1000 1000

2

1 3 0xFF0000 7 -1000 -1000
               -1000  -900
               -100    000
                000    000
                100    000
               1000    900
               1000   1000

2 2 0x00FF00 0x80FF80 6 10 10
                       20 10
                       20 15
                       15 15
                       15 20
                       10 20
```

Рис. 16.1. Пример содержания текстового файла с закодированной картой

Описание класса `TMap`, предназначенного для отображения карты

Класс (`TMap/карта`) имеет простую структуру. Поля:

- массив указателей на базовый класс (`FElems`); при создании объектов дочерних классов указатели на эти объекты будут сохраняться в этом массиве;

- количество элементов в массиве (`FCount`);
- переменная (`FRegion`) структурного типа для хранения координат исходного региона отображения; тип структуры представлен в листинге 16.2.

Примечание

В файле с исходными данными хранятся координаты выбранного региона. Программа первоначально отображает карту именно этого региона, и во время работы в любой момент можно вернуться для отображения карты этого региона.

Листинг 16.2. Интерфейс структуры для описания координат исходного региона

```
struct TRegion
{
  long double Xmin, Ymin, Xmax, Ymax;
};
```

Помимо конструкторов и деструктора, в классе включены два метода: загрузка (`LoadMap`) и отображение (`ShowMap`) карты; а также два свойства: количество `Count` и `Region`, которые обеспечивают доступ к соответствующей информации объекта. Интерфейс `TMap` может иметь следующий вид, представленный в листинге 16.3.

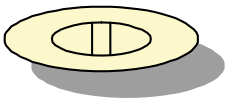
Листинг 16.3. Интерфейс класса `TMap`

```
class TMap
 {
  private:
    TBaseElem **FElems; // массив указателей на элементы карты
    int       FCount;  // количество элементов на карте
    TRegion   FRegion; // регион начального отображения карты
  protected:

  public:
    __fastcall TMap ();
    __fastcall TMap (AnsiString aFileName);
    __fastcall ~TMap ();

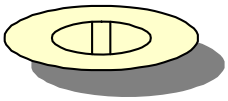
    void __fastcall LoadMap (AnsiString aFileName);
    void __fastcall ShowMap (TImage *);

  __property int Count = {read = FCount};
  __property TRegion Region = {read = FRegion};
 };
```



МЕТОДЫ КЛАССА `TMap`:

- `__fastcall TMap ();` – пустой конструктор;
- `__fastcall TMap (AnsiString aFileName);` – конструктор с параметром `aFileName` (имя файла);
- `__fastcall ~TMap ();` – деструктор класса;
- `void __fastcall LoadMap (AnsiString aFileName);` – метод класса для загрузки карты из файла `aFileName`;
- `void __fastcall ShowMap (TImage *)` – метод класса для отображения карты.



СВОЙСТВА КЛАССА `TMap`:

- `__property int Count = {read = FCount};` – количество элементов на карте;
- `__property TRegion Region = {read = FRegion};` – регион отображения.

Конструктор класса «Карта»

Данный конструктор лишь вызывает метод загрузки данных, так как инициализация полей осуществляется загруженными данными. Конструктор имеет в качестве входного параметра имя файла с картой. Описание реализации конструктора приведено в листинге 16.4.

Листинг 16.4. Описание конструктора класса `TMap`

```
__fastcall TMap::TMap (AnsiString aFileName)
{
    LoadMap (aFileName);
}
```

Деструктор класса «Карта»

Деструктор уничтожает объекты, отображающие элементы карты, и освобождает память, занятую массивом указателей на эти объекты. Описание реализации деструктора представлено в листинге 16.5.

Листинг 16.5. Описание деструктора класса `TMap`

```
__fastcall TMap::~TMap ()
{
 for (int iMap = 0; iMap < FCount; iMap++)
   delete FElems [iMap];
 delete[] FElems;
}
```

Загрузка карты из файла

Метод загрузки карты `LoadMap`, по сути, только лишь считывает данные о регионе, о количестве объектов, выделяет память под массив объектов. Далее в цикле считывается код объекта и создается соответствующий объект, конструктор которого автоматически загружает информацию об объекте. Описание метода приведено в листинге 16.6.

Листинг 16.6. Описание метода `LoadMap` – загрузчика файла карт класса `TMap`

```
void __fastcall TMap::LoadMap (AnsiString aFileName)
{
  ifstream fin (aFileName.c_str(), ios_base::in);
  long double xmin, ymin, xmax, ymax;
  fin >> xmin >> ymin >> xmax >> ymax;

  FRegion.Xmin = xmin; FRegion.Ymin = ymin;
  FRegion.Xmax = xmax; FRegion.Ymax = ymax;

  SetMathReg (xmin, ymin, xmax, ymax);

  fin >> FCount; // количество элементов на карте
  FElems = new TBaseElem* [FCount];
  int code;
```

```
  for (int iEl = 0; iEl < FCount; iEl++)
   {fin >> code;
    switch (code)  // создание объектов и загрузка данных
     {case 1 : FElems [iEl] = new TRoad (fin); break;
      case 2 : FElems [iEl] = new TBuilding (fin); break;
     }
   }
}
```

Формирование изображения карты

Метод отображения `ShowMap` обеспечивает основное действие объекта-карты – визуализация объектов карты, поэтому суть его работы сводится к перебору всех объектов и вызову их метода отображения. Реализация метода приведена в листинге 16.7.

Листинг 16.7. Отображение карты

```
void __fastcall TMap::ShowMap (TImage *aIm)
{
   for (int iElem = 0; iElem < FCount; iElem++)
      FElems [iElem]->ShowElem(aIm);
}
```

Описание класса `TRoad`, предназначенного для отображения дорог

Класс является потомком базового класса `TBaseElem`. Поля класса (назовем его `TRoad`) должны соответствовать данным, хранящимся в файле, предназначенном для отображения дорог. Поскольку методы базового класса абстрактные, в классе-потомке они должны быть переопределены. Интерфейс класса `TRoad` приведен в листинге 16.8.

Листинг 16.8. Интерфейс класса `TRoad`

```
class TRoad : public TBaseElem
 {
  private:
  protected:
    TMathPoint *FPnts;   // узлы Безье кривой
    int         FCount;  // количество узлов (3N+1)
    int         FWidth;  // толщина линии в пикселях
    TColor      FColor;  // цвет линии
  public:
    __fastcall TRoad ();// пустой конструктор
    __fastcall TRoad (ifstream &);// конструктор – из файла
    __fastcall ~TRoad ();
    virtual void __fastcall InitFields ();
    virtual void __fastcall LoadElem (ifstream &);
    virtual void __fastcall ShowElem (TImage *);
 };ы
```

Конструктор класса `TRoad` – «Дорога»

Конструктор класса имеет один параметр `aStream` – имя файловой потоковой переменной, необходимой для связи с файлом данных. Вызывает функции инициализации полей и загрузки данных о картографическом объекте типа *дорога*. Описание конструктора приведено в листинге 16.9.

Листинг 16.9. Конструктор класса `TRoad`

```
__fastcall TRoad::TRoad (ifstream &aStream) : TBaseElem ()
{ InitFields ();
  LoadElem (aStream);
}
```

Деструктор класса `TRoad` – «Дорога»

Деструктор только освобождает память, занятую массивом координат узлов линии, составляющей характеристическую ломаную кривой Безье. Описание реализации процедуры деструктора приведено в листинге 16.10.

Листинг 16.10. Деструктор класса `TRoad`

```
__fastcall TRoad::~TRoad ()
{
  delete[] FPnts;
}
```

Загрузка «дорог» из файла

Метод `LoadElem` имеет один параметр `aStream` – имя файловой переменной, необходимой для связи с файлом данных. Он осуществляет чтение данных о дороге: толщину и цвет линии; количество узлов характеристической ломаной и сами координаты узлов, предварительно, естественно, выделив достаточно памяти.

Порядок работы этой процедуры достаточно прозрачен. Из файлового потока считываются отдельные параметры кривой, а после прочтения количества узлов характеристической ломаной в цикле читаются их координаты. Здесь хотелось бы обратить внимание на задание в файле и чтение в процедуре значения *цвета* кривой. Так как цвет удобнее задавать в шестнадцатеричной форме, то в файле он и задается в таком виде. Поэтому при чтении цвета приходится указать, что данные необходимо читать в шестнадцатеричном виде. Код реализации метода `LoadElem` приведен в листинге 16.11.

Листинг 16.11. Описание метода загрузки данных «дорога»

```
void __fastcall TRoad::LoadElem (ifstream &aStream)
{ if (FPnts != NULL) delete[] FPnts;
  aStream >> FWidth;
  int aux;
  aStream >> hex >> aux; // чтение цвета
  FColor = StringToColor (IntToStr (aux));
  aStream >> FCount; ResizeGPNTS (FCount);
  FPnts = new TMathPoint [FCount];
  for (int iElem = 0; iElem < FCount; iElem++)
   {aStream >> FPnts [iElem].X;
    aStream >> FPnts [iElem].Y;
   }
}
```

Формирование изображения «дорог»

Метод `ShowElem` настраивает параметры отображения кривой и рисует ее. Работа этой подпрограммы заключается в пересчете математических координат кривой в экранные координаты и в укладывании этих значений в целочисленный массив. Построение кривой выполняется при использовании соответствующего метода холста. Код реализации метода представлен в листинге 16.12.

Листинг 16.12. Описание метода формирования изображения «дорога»

```
void __fastcall TRoad::ShowElem (TImage *aIm)
{ int auxX, auxY;
  aIm->Canvas->Pen->Width = FWidth;
  aIm->Canvas->Pen->Color = FColor;
  for (int i = 0; i < FCount; i++) // пересчет всех координат
   {MathToScr (FPnts[i].X, FPnts[i].Y, auxX, auxY);
    gPnts[i].x = auxX;
    gPnts[i].y = auxY;
   }
  aIm->Canvas->PolyBezier (gPnts, FCount - 1);
}
```

Инициализация полей класса «Дорога»

Метод `InitFields` инициализирует поля класса начальными значениями. Код реализации метода представлен в листинге 16.13.

Листинг 16.13. Описание метода формирования изображения «дорога»

```
void __fastcall TRoad::InitFields ()
{ FPnts = NULL;
  FCount = 0;
  FWidth = 1;
  FColor = clBlack;
}
```

Описание класса `TBuilding`, предназначенного для отображения зданий

Класс `TBuilding` строится по образу и подобию класса `TRoad` – «Дорога». Описание интерфейсной части класса `TBuilding` и реализация методов приведена в листингах 16.14–16.17. Описание конструкторов и деструктора здесь не приведено, так как их реализация идентична классу `TRoad`. Да и приведенные в листинге методы не имеют особых отличий: инициализируются и загружаются такие же поля, как и ранее, и еще цвет заливки внутренней области многоугольника. А при выводе изображения используется другой метод холста.

Листинг 16.14. Интерфейс класса `TBuilding` – «Здание»

```
class TBuilding : public TBaseElem
 {private:
  protected:
    TMathPoint *FPnts; // узлы ломаной, контура здания
    int     FCount;    // количество узлов
    int     FWidth;    // толщина линии в пикселях
    TColor  FBrdrClr;  // цвет контура здания
    TColor  FColor;    // цвет изображения здания
  public:
    __fastcall TBuilding ();
    __fastcall TBuilding (ifstream &);
    __fastcall ~TBuilding ();
    virtual void __fastcall InitFields ();
    virtual void __fastcall LoadElem (ifstream &);
    virtual void __fastcall ShowElem (TImage *);
 };
```

Листинг 16.15. Описание метода `InitFields` класса `TBuilding` – «Здание»

```
void __fastcall TBuilding::InitFields ()
{ FPnts = NULL;
  FCount = 0;
  FWidth = 1;
  FBrdrClr = clBlack;
  FColor = clWhite;
}
```

Листинг 16.16. Описание метода `LoadElem` класса `TBuilding` – «Здание»

```
void __fastcall TBuilding::LoadElem (ifstream &aStream)
{ if (FPnts != NULL) delete[] FPnts;
  aStream >> FWidth;
  int aux;
  aStream >> hex >> aux; FBrdrClr = StringToColor (IntToStr (aux));
  aStream >> hex >> aux; FColor = StringToColor (IntToStr (aux));
  aStream >> FCount; ResizeGPNTS (FCount);
  FPnts = new TMathPoint [FCount];
  for (int iElem = 0; iElem < FCount; iElem++)
   {aStream >> FPnts [iElem].X;
    aStream >> FPnts [iElem].Y;
   }
}
```

Листинг 16.17. Описание метода `ShowElem` класса `TBuilding` – «Здание»

```
void __fastcall TBuilding::ShowElem (TImage *aIm)
{ int auxX, auxY;
  aIm->Canvas->Pen->Width = FWidth;
  aIm->Canvas->Pen->Color = FBrdrClr;
  aIm->Canvas->Brush->Color = FColor;
  for (int i = 0; i < FCount; i++)  //  пересчет всех координат
   {TPoint aux;
    MathToScr (FPnts[i].X, FPnts[i].Y, auxX, auxY);
    gPnts[i].x = auxX; gPnts[i].y = auxY;
   }
  aIm->Canvas->Polygon (gPnts, FCount - 1);
}
```

Примечание

В условиях задачи требуется отображать дороги, строения и населенные пункты. Населенный пункт – это набор отдельных строений, расположенных компактно. Поэтому в простейших случаях достаточно отображать только дороги и строения. Но при крупном масштабе карты изображение отдельных строения населенных пунктов сливаются в одно пятно. При таких масштабах проще отображать некий многоугольник, очерчивающий границы населенного пункта. Для этого достаточно усложнить программу, чтобы она принимала решение, когда отображать отдельные строения, а когда – лишь контур населенного пункта.

Задача 16.2. К картографическому изображению системы дорог заданного региона требуется добавить оси координат с предоставлением возможности изменения масштаба изображения.

Решение

Для построения осей координат можно воспользоваться классом `TAxes`, разработанным для приложения построения графиков функций. Но есть некоторые важ-

ные отличия как в реализации методов классов построения изображения карт, так и при построении осей для карт:

а) при построении графиков функций, как правило, изображения осей координат и графика совмещены. При этом оси координат часто пересекаются в точке (0, 0);
б) иная ситуация с картографическими изображениями. Они насыщены мелкими элементами и чтобы оси координат не закрывали детали, то оси наносят вне поля карты. А для лучшей ориентации на карту наносят тонкие линии сетки;
в) при построении графиков функций математические координаты точек вычисляются и конвертируются поочередно. То есть в один момент времени известна *только одна точка* и ее математические и экранные координаты;
г) в случае картографического изображения приходится хранить *все отсчеты*, а вывод кривой осуществляется стандартной процедурой, которая в качестве входного параметра имеет массив координат экрана.

Следовательно, при построении изображения карты необходимо решить задачу, *где должны быть описаны массивы*, в которых будут храниться математические координаты и, хотя бы кратковременно, координаты точек экрана. То есть и массив пар вещественных чисел, и массив пар целых чисел. Для массива математических координат ответ практически однозначен – внутри каждого класса. Что же касается массива экранных координат, то здесь не все так однозначно, и есть, как всегда, несколько схем:

1) можно краткосрочно создавать такой массив внутри процедуры вывода изображения кривой;
2) можно сделать такой массив полем класса;
3) можно сделать такой массив общим глобальным ресурсом.

Отдадим предпочтение третьему варианту. При этом хотелось обратить внимание на два существенных момента.

Во-первых

Глобальная переменная должна быть объявлена где-то в модуле, на который ссылаются все модули проекта, то есть сделать ее внешней с типом `extern`. Так как в этих проектах в основе лежит модуль преобразований координат `UMath2Scr.h`, а массив экранных координат логически с этими преобразованиями связан, то описание помещено в этот модуль в виде строк:

```
extern PACKAGE TPoint *gPnts;      // массив точек экрана
extern PACKAGE int     gSizePnts;  // размер этого массива
```

И здесь одна тонкость. Это только *объявление имен* этих переменных. Память ни под указатель, ни, тем более, под массив целых переменных не выделяется. В описании и указывается, что эти переменные где-то вне этого модуля описаны (ключевое слово `extern`). Поэтому в каком-либо модуле `.CPP` проекта необходимо описать такие переменные, например, следующим образом:

```
TPoint *gPnts;      // массив точек экрана
int     gSizePnts;  // размер этого массива
```

То есть при окончательной сборке приложения именно эти переменные будут использоваться всеми модулями, которые даже и не ссылаются на тот самый `.CPP`, но ссылаются на переменные `gPnts` и `gSizePnts` из модуля `UMath2Scr.h`.

Во-вторых

Переменная `gPnts` есть лишь указатель на массив, поэтому необходимо динамически выделять память. Но здесь особенность в том, что необходимо *выделить память под самый большой* массив, который потребуется какому-нибудь из объектов. Такое решение может принять только сам объект при загрузке данных.

Для разрешения таких противоречий можно разработать подпрограмму (например, с именем `ResizeGPNTS`), которая будет оценивать необходимость увеличения объема массива и корректно выполнять эту операцию: *выделять новую память* большего объема и, если необходимо, копировать прежние данные. Прототип и текст такой процедуры приведен в листинге 16.18.

Листинг 16.18. Описание прототипа и реализации процедуры выделения новой памяти для вспомогательного массива

```
void ResizeGPNTS (int, bool = false);
                    // переопределение размера массива gPnts

void ResizeGPNTS (int aSize, bool aCopy)
{   // переопределение размера массива gPnts
  if (gSizePnts >= aSize || aSize <= 0) return;
  TPoint *aux = new TPoint [aSize];
  if (aCopy)
    for (int i = 0; i < gSizePnts; i++)
      aux [i] = gPnts [i];
  delete[] gPnts;

  gPnts = aux;
  gSizePnts = aSize;
}
```

Примечание

Вариант, когда все объекты используют единый массив для экранных точек, имеет право на существование. При перерисовывании изображения требуется строго следить, изменился ли регион отображения или масштаб. Если да, то приходится содержимое массива экранных координат обновлять, иначе текущие значения остаются корректными. Хотя могут возникнуть и другие поводы пересчитать экранные координаты. Другой способ – это каждый раз обновлять значения в массиве координат перед отображением, поэтому для всех объектов можно использовать единый массив экранных координат. В него каждый объект насчитывает экранные координаты и строит изображение.

Интерфейс приложения может быть реализован в виде, представленном на рис. 16.2. Элементы управления размещены на панели `paPanel`.

Замечание

Для того чтобы отображать оси вне карты требуется три компонента `Image`: `imMap` – для вывода карты, `imLeft` и `imBottom` – для вывода вертикальной и горизонтальной осей координат, соответственно. Дело осложняется тем, что размеры формы могут меняться произвольным образом. И при этом необходимо, чтобы изменялись размеры компонентов `Image`, но оставались согласованными, а именно: `imLeft` всегда совпадал по высоте с `imMap`, а `imBottom` – по ширине.

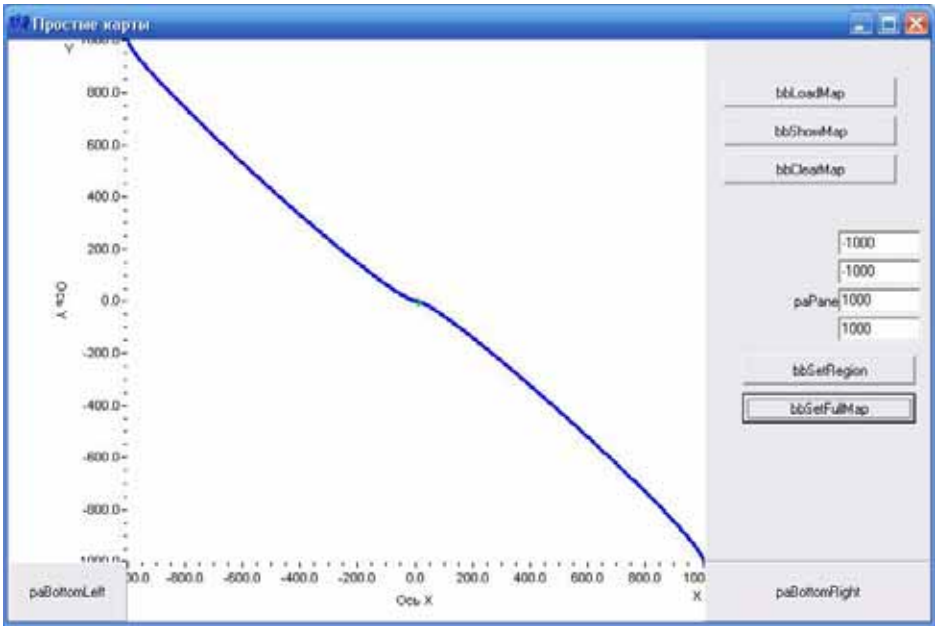


Рис. 16.2. Интерфейс программы построения карт

Так как по условию задачи карту следует рассматривать на разных участках с различным увеличением, поэтому интерфейс программы должен содержать элементы выбора региона и масштаба изображения. Для этого, в простейшем случае, достаточно на интерфейсе разместить четыре окна для ввода координат региона отображения. Программа выбирает эти значения и масштабирует элементы отображения в соответствии с этими значениями.

Реализация такого проекта заключается в разработке обработчиков событий на создание и уничтожение главной формы программы, изменения ее размеров, а также обработчиков события `OnClick` кнопок. Процедура обработки события создания формы задает точку как символ разделения дробной части и целой в виде вещественных чисел, обнуления некоторых полей и глобальных переменных. Помимо этого инициализируется объект оси `FAxes`, а также устанавливаются параметры области отображения карты и точка пересечения осей координат. Текст обработчика приведен в листинге 16.19.

Листинг 16.19. Описание обработчика события создания главной формы программы

```
void __fastcall TfoMain::FormCreate (TObject *Sender)
{ DecimalSeparator = '.';  // настройка разделителя дробной части

  gPnts = NULL;    // настройка глобальной переменной буфер
  gSizePnts = 0;   // экранных точек

  FMap = NULL;
  SetScrRect (0, 0, imMap->Width-1, imMap->Height-1);

  FAxes = new TAxes;
  FAxes->CrossPnt = Point (imLeft->Width - 1, 0);

  imLeft->Canvas->Font->Name = "Arial";  // установка шрифта True Type
  imBottom->Canvas->Font->Name = "Arial";
}
```

При закрытии программы достаточно освободить память, занятую экранными точками объекта и объектом «карта». Текст обработчика приведен в листинге 16.20.

Листинг 16.20. Описание обработчика события закрытия главной формы программы

```
void __fastcall TfoMain::FormClose (TObject *Sender,
                                    TCloseAction &Action)
{ if (gPnts != NULL)
    delete[] gPnts;
  if (FMap != NULL)
    delete FMap;
}
```

При изменении размеров формы программы необходимо изменить размеры областей отображений осей координат синхронно с областью отображения карты, задать новые размеры региона, очистить области изображения карты и осей координат и нарисовать их заново. Текст обработчика приведен в листинге 16.21.

Листинг 16.21. Описание обработчика события изменения размеров главной формы программы

```
void __fastcall TfoMain::FormResize(TObject *Sender)
{   // корректировка размеров полей отображения
  imMap->Picture->Graphic->Width = imMap->Width;
  imMap->Picture->Graphic->Height = imMap->Height;
  imLeft->Picture->Graphic->Height = imLeft->Height;
  imBottom->Picture->Graphic->Width = imBottom->Width;
    // задание нового экранного поля
  SetScrRect (0, 0, imMap->Width-1, imMap->Height-1);
    // очистка и рисование
  bbClearMapClick (NULL);
  bbShowMapClick (NULL);
}
```

При загрузке карты по кнопке «Загрузить карту» обработчик ее события `OnClick` порождает объект из выбранного пользователем файла, устанавливает регион отображения карты, считанный из файла, и отображает его границы в окнах редактирования, а затем подключает обработчик события изменения размера. Текст обработчика приведен в листинге 16.22.

Листинг 16.22. Описание обработчика события нажатия кнопки «Загрузить карту»

```
void __fastcall TfoMain::bbLoadMapClick(TObject *Sender)
{ if (dlgOpenMap->Execute())
   {
    if (FMap == NULL) delete FMap;
    FMap = new TMap (dlgOpenMap->FileName);
    FAxes->XStep = (gMathReg.Xmax - gMathReg.Xmin) / 10;
    FAxes->YStep = (gMathReg.Ymax - gMathReg.Ymin) / 10;

    edXmin->Text = FloatToStr (gMathReg.Xmin);
    edXmax->Text = FloatToStr (gMathReg.Xmax);
    edYmin->Text = FloatToStr (gMathReg.Ymin);
    edYmax->Text = FloatToStr (gMathReg.Ymax);

    OnResize = FormResize;  // подключение обработчика события
   }
}
```

Для отображения карты по кнопке «Показать карту» обработчик ее события `OnClick` просто обращается к соответствующим объектам для отображения карты и осей координат на соответствующих компонентах `TImage`. Текст обработчика приведен в листинге 16.23.

Листинг 16.23. Описание обработчика события нажатия кнопки «Показать карту»

```
void __fastcall TfoMain::bbShowMapClick (TObject *Sender)
{ FMap->ShowMap (imMap);
  FAxes->DrawHorzAxis(imBottom, true);
  FAxes->DrawVertAxis(imLeft, true);
}
```

Обработчик события `OnClick` кнопки «Стереть» выполняет очистку изображений на всех компонентах `TImage`. Текст обработчика приведен в листинге 16.24.

Листинг 16.24. Описание обработчика события нажатия кнопки «Стереть»

```
void __fastcall TfoMain::bbShowMapClick (TObject *Sender)
{ FMap->ShowMap (imMap);
  FAxes->DrawHorzAxis(imBottom, true);
  FAxes->DrawVertAxis(imLeft, true);
}
```

Обработчик события `OnClick` кнопки «Установить регион» выполняет чтение значений из однострочных редакторов, настраивает математическую область отображения и шаг по осям координат, и, конечно, очищает, и строит новое изображение карты и осей. При наличии ошибок в написании чисел пользователем, процедура выдает сообщение пользователю и предоставляет возможность исправить их. Текст обработчика приведен в листинге 16.25.

Листинг 16.25. Описание обработчика события нажатия кнопки «Установить регион»

```
void __fastcall TfoMain::bbSetRegionClick(TObject *Sender)
{ long double xmin, xmax, ymin, ymax;
  if (TryStrToFloat (edXmin->Text, xmin) && // попытка конвертирования
      TryStrToFloat (edXmax->Text, xmax) &&
      TryStrToFloat (edYmin->Text, ymin) &&
      TryStrToFloat (edYmax->Text, ymax))
   {gMathReg.Xmin = xmin; gMathReg.Xmax = xmax;
    gMathReg.Ymin = ymin; gMathReg.Ymax = ymax;

    FAxes->XStep = (gMathReg.Xmax - gMathReg.Xmin) / 10;
    FAxes->YStep = (gMathReg.Ymax - gMathReg.Ymin) / 10;

    bbClearMapClick (NULL);
    bbShowMapClick (NULL);
   }
  else MessageDlg
           ("В одном из полей неверно написано вещественное число!",
            mtError, TMsgDlgButtons() << mbOK, 0);
}
```

Обработчик события `OnClick` кнопки «Полный размер» устанавливает значения отображения карты, считанные из файла, в структуру хранения математической области, поднастраивает оси координат и записывает их в однострочные редакторы. После этого очищает и строит новое изображение карты и осей. Текст обработчика приведен в листинге 16.26.

Листинг 16.26. Описание обработчика события нажатия кнопки «Полный размер»

```
void __fastcall TfoMain::bbSetFullMapClick(TObject *Sender)
{  // установка исходного района отображения
  gMathReg = FMap->Region;

  FAxes->XStep = (gMathReg.Xmax - gMathReg.Xmin) / 10;
  FAxes->YStep = (gMathReg.Ymax - gMathReg.Ymin) / 10;

  edXmin->Text = FloatToStr (gMathReg.Xmin);
  edXmax->Text = FloatToStr (gMathReg.Xmax);
  edYmin->Text = FloatToStr (gMathReg.Ymin);
  edYmax->Text = FloatToStr (gMathReg.Ymax);

    // очистка и рисование
  bbClearMapClick (NULL);
  bbShowMapClick (NULL);
}
```

Конечно, для простейшего проекта отображения карт таких элементов и обработчиков достаточно. Но надо учитывать, что в хорошем приложении отображения карт не может быть сжатия-растяжения изображения. Поэтому в более сложном случае программа должна не только воспринимать заданные координаты, но и корректировать их, чтобы не происходило искажения изображения.



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. Доработать проект задачи 16.1, чтобы он содержал и описание строений и описание контуров населенных пунктов.
2. Доработать проект задачи 16.2 так, чтобы выполнялось сохранение соотношения масштаба по координатам X и Y.
3. Доработать предыдущий проект, чтобы он также содержал изображения водных ресурсов региона, например реки и озера.
4. Доработать проект с отображением карты местности так, чтобы он отображал и особенности ландшафта местности, например, контурными линиями или областями с разным уровнем интенсивности цвета (как на картах).

Разработка элементов графического редактора 17

Современные программы, даже простейшие, имеют элементы графического интерфейса. Например, это может быть только лишь панель инструментов, несущая кнопки, и скроллеры – для перемещения изображения окна. А чуть более сложные приложения могут содержать элементы редактирования информации посредством манипулятора типа «мышь» или, вообще, содержать элементы управления и редактирования, максимально схожие отдельными частями с графическими редакторами.

При разработке подобных приложений можно использовать какие-либо подходящие библиотеки процедур. Но и сама система *C++ Builder* предоставляет достаточно средств для графического управления и редактирования при разработке собственных приложений.

Основу растровой графики, как известно, составляет растровое изображение. Для формирования такого изображения достаточно задать серию команд, которые бы окрасили пиксели экрана соответствующими цветами. Для смены изображения достаточно окрасить пиксели в другие цвета. В момент редактирования графического изображения ситуация несколько меняется. Изображение условно делится на две составляющие:

1) *неизменная часть* – часть изображения, которая не изменяется в ходе выполнения данной команды редактирования;
2) *переменная часть* – часть изображения, которая, собственно, и подвергается корректировке данной командой. Для элементов, составляющих эту часть изображения, характерно динамичное изменение местоположения, да и не редко и формы.

При этом, как правило, *эти части пересекаются*: изображение статичных элементов частично перекрывается изображениями динамичных. Поэтому при реализации команд редактирования приходится тщательно прорабатывать процесс формирования изображения в момент выполнения операции: необходимо воспроизвести ситуацию, так хорошо известную нам по работе с мощными общеизвестными графическими редакторами.

При разработке набора процедур редактирования графического изображения необходимо выделять две составляющие:

- четкая фиксация собственно команды управления, подаваемой пользователем;
- корректное формирование изображения.

Чего хочет пользователь Пользователь программы при редактировании графики должен чувствовать, что она его «однозначно понимает». А на экране четко видеть, что часть изображения, которую он и не собирается менять, остается неизменной, а другая часть чутко меняется под его руководством.

Возможны два подхода к решению задачи редактирования графического изображения:

а) полностью перерисовывать изображение с учетом нового положения и формы редактируемого элемента. Это простейший вариант. Для изображений с малым количеством элементов это срабатывает. Но для изображения с огромным количеством элементов полное перерисовывание вы-

зывает визуальные проблемы: изображение начинает мерцать при каждой прорисовке и тому подобное;

б) перерисовывать не все изображение, а только ту его часть, которая действительно нуждается в обновлении, потому что в единый момент времени редактированию подвергается единственный элемент, а его изображение затрагивает изображения всего нескольких элементов статической части.

Для более быстрого рисования следует воспользоваться вторым подходом. При этом можно выделить несколько принципиально различных идей, лежащих в основе такого рисования:

- изображение делится на составляющие элементы. Бо́льшая часть изображения остается неизменной, а перерисовывается минимальное количество элементов, но достаточное, чтобы изображение оставалось корректным каждое мгновение;
- изображение делится на области. Перерисовке подвергается только изменяющаяся область. Остальная часть изображения не перерисовывается;
- редактируемые элементы рисуются специальными приемами, суть которых в следующем:
    - рисуется статическая часть изображения;
    - при прорисовке динамичного элемента *только искажается* изображение статичной части изображения. В этом искажении однозначно видна геометрия редактируемого элемента;
    - по окончании команды редактирования этот элемент корректно отображается в правильной цветовой палитре.

Использование такого приема хорошо известно, например, по командам рисования в редакторе *Word for Windows*. Следует также отметить, что задача наложения изображений тесно связана с задачей наложения цветов этих изображений.

Примечание

Если на холсте изображаются разноцветные фигуры, то в случае *наложения/пересечения фигур* возникает также задача *наложения цветов* друг на друга. Требуется, чтобы заданный цвет не просто заносился, а преобразовывался с учетом *текущего* цвета *пикселя* или даже цвета *фона*.

ФОН

Фоном называется область изображения, свободная от основных элементов изображения, формирующих рисунок.

Прежде чем приступить к разработке алгоритма рисования/исправления той или иной фигуры необходимо решить, как должны взаимодействовать между собой цвет пера *Pen* и цвет холста (текущий цвет и новый цвет). Для установки нужного режима для взаимодействия цветов можно воспользоваться свойством пера `Mode`[1].

Примечание

По умолчанию свойство пера `Mode` имеет значение `pmCopy`, которое обеспечивает простое копирование.

[1] Может принимать одно из 16 значений.

Задача 17.1. Подготовить рисунок, содержащий три геометрические фигуры: круг, квадрат и прямоугольник. Причем требуется эти три фигуры разместить в окне `Image` таким образом, как это показано на рис. 17.1. Предусмотреть такую закраску фигур, чтобы каждая имела свой цвет и свою окантовку. Опробовать разные режимы рисования фигур на фоне имеющегося рисунка.

Решение

Анализ изображения фигур, представленных на рис.17.1, допускает выделение в этом рисунке двух составляющих: круг – это будет неизменная часть; прямоугольник и квадрат – это изменяющаяся часть. Такое распределение ролей позволит посмотреть возможности рисования в различных режимах.

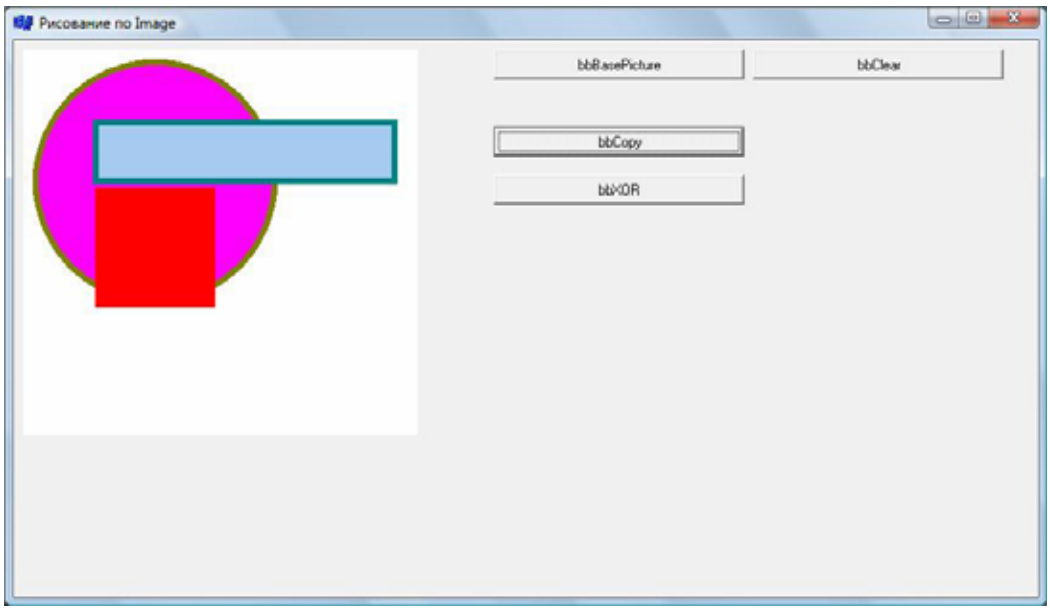


Рис. 17.1. Интерфейс приложения формирования простейшего изображения

Предлагается следующий порядок действий для построения изображения:
1) построить статическую часть – окружность;
2) построить динамическую часть – прямоугольник и квадрат.

Рассмотрим *два варианта* решения. В первом варианте будет использоваться режим `Mode` – `pmCopy`, который обеспечивает простое копирование. Во втором варианте будет использоваться режим `Mode` – `pmXOR`, который обеспечивает комбинацию цветов пера (текущего и нового).

Вариант 1.

Для каждого действия целесообразно предусмотреть отдельную процедуру (можно коды построения включить в обработчики событий под кнопки, как это показано в интерфейсе приложения на рис. 17.1):

- `bbBasePicture` (строит статическую часть изображения; здесь – только окружность);
- `bbCopy` (строит переменную часть изображения, имитирующую редактируемый элемент изображения).

Изображение окружности

Процедура (действия по построению) встроена в код обработчика события `OnClick` кнопки `Button` (с именем `bbBasePicture`). В теле процедуры устанавливается режим рисования, настраивается размер и положение элемента, цвета для отображения границы и области и вызывается подпрограмма `Ellipse`, необходимая для построения круга.

Предупреждение

На первый взгляд, необходимости в первой строке, устанавливающей режим рисования как копирование, у этой процедуры нет, так как этот режим установлен по умолчанию. Но предполагается, что процедура будет работать совместно с другими процедурами, которые будут изменять значение свойства `Mode`. Поэтому в будущем эта строка потребуется, чтобы установить именно режим `pmCopy` независимо от того, какое значение `Mode` было ранее.

Код процедуры обработчика события `OnClick` кнопки `bbBasePicture` представлен в листинге 17.1.

Листинг 17.1. Процедуры формирования изображения «Круг»

```
void __fastcall TfoMain::bbBasePictureClick(TObject *Sender)
{
  Image->Canvas->Pen->Mode = pmCopy; // режим рисования
            // параметры эллипса/круга
  int xLT_E = 10, yLT_E = 10, xRB_E = 210, yRB_E = 210;
            // цвет и ширина пера
  Image->Canvas->Pen->Color = clOlive;
  Image->Canvas->Pen->Width = 5;
            // цвет кисти – заливки круга
  Image->Canvas->Brush->Color = clFuchsia;
  Image->Canvas->Ellipse(xLT_E, yLT_E, xRB_E, yRB_E);
}
```

Изображение квадрата и прямоугольника

Процедура, обеспечивающая действия по построению квадрата и прямоугольника, встроена в код обработчика события `OnClick` кнопки `Button` с именем `bbCopy`. В теле процедуры

- устанавливается режим рисования;
- настраивается размер и положение прямоугольника;
- настраивается цвет отображения границы и области;
- вызывается подпрограмма `Rectangle`, необходимая для построения прямоугольника;
- строится квадрат.

Квадрат же рисуется просто: в цикле закрашиваются соответствующие пиксели, заданным цветом. В процедуре прямоугольник и квадрат строятся разными способами: первый – *посредством функции*; второй – *попиксельно*. Описание процедуры-обработчика представлено в листинге 17.2.

Листинг 17.2. Процедуры формирования изображения «Прямоугольник и квадрат»

```
void __fastcall TfoMain::bbCopyClick(TObject *Sender)
```

```
{ Image->Canvas->Pen->Mode = pmCopy; // режим рисования
      // размер, положение
  int xLT_R = 60, yLT_R = 60, xRB_R = 310, yRB_R = 110;
      // цвета
  Image->Canvas->Pen->Color = clTeal;
  Image->Canvas->Brush->Color = clSkyBlue;
      // построение прямоугольника
  Image->Canvas->Rectangle(xLT_R, yLT_R, xRB_R, yRB_R);
      // построение квадрата
  int xLT_S = 60, yLT_S = 115, xRB_S = 160, yRB_S = 215;
  for (int iy = yLT_S; iy < yRB_S; iy++)
    for (int ix = xLT_S; ix < xRB_S; ix++)
      Image->Canvas->Pixels [ix][iy] = clRed;
}
```

Вариант 2.

Оставим построение статической части (процедуру `BasePicture`) без изменения. А для отображения динамической части построим новую процедуру – обработчик события `OnClick` кнопки `Button` с именем `bbXOR`. Описание этой процедуры представлено в листинге 17.3. Отличие этой процедуры от текста процедуры `bbCopy`, представленной в листинге 17.2, лишь в значении свойства `Mode`. Здесь оно равно `pmXOR`. В таком режиме `Canvas` не просто копирует в пиксель заданный цвет, а выполняет для каждого пикселя операцию `XOR` над заданным цветом и текущим цветом пикселя.

Примечание

Важно отметить, что для того, чтобы добиться требуемого цвета изображения, процедуру `bbXOR` *нужно выполнить дважды*. Первый раз получится неправильная цветовая гамма. Второй раз – соответствует заданному цвету.

Листинг 17.3. Процедура формирования переменной части изображения

```
void __fastcall TfoMain::bbXORClick (TObject *Sender)
{ Image->Canvas->Pen->Mode = pmXor;  // режим рисования
      // размер, положение
  int xLT_R = 60, yLT_R = 60, xRB_R = 310, yRB_R = 110;
      // цвета
  Image->Canvas->Pen->Color = clTeal;
  Image->Canvas->Brush->Color = clSkyBlue;
      // построение прямоугольника
  Image->Canvas->Rectangle(xLT_R, yLT_R, xRB_R, yRB_R);
           // построение квадрата
  int xLT_S = 60, yLT_S = 115, xRB_S = 160, yRB_S = 215;
  for (int iy = yLT_S; iy < yRB_S; iy++)
    for (int ix = xLT_S; ix < xRB_S; ix++)
      Image->Canvas->Pixels [ix][iy] = clRed;
}
```

Результат работы функций `bbBasePicture` и первого выполнения `bbXOR` приведен на рис. 17.2. Хорошо видно, что цвет изображения переменной части нигде не соответствует заданным цветам, но контуры переменной части изображения точно соответствуют заданным.

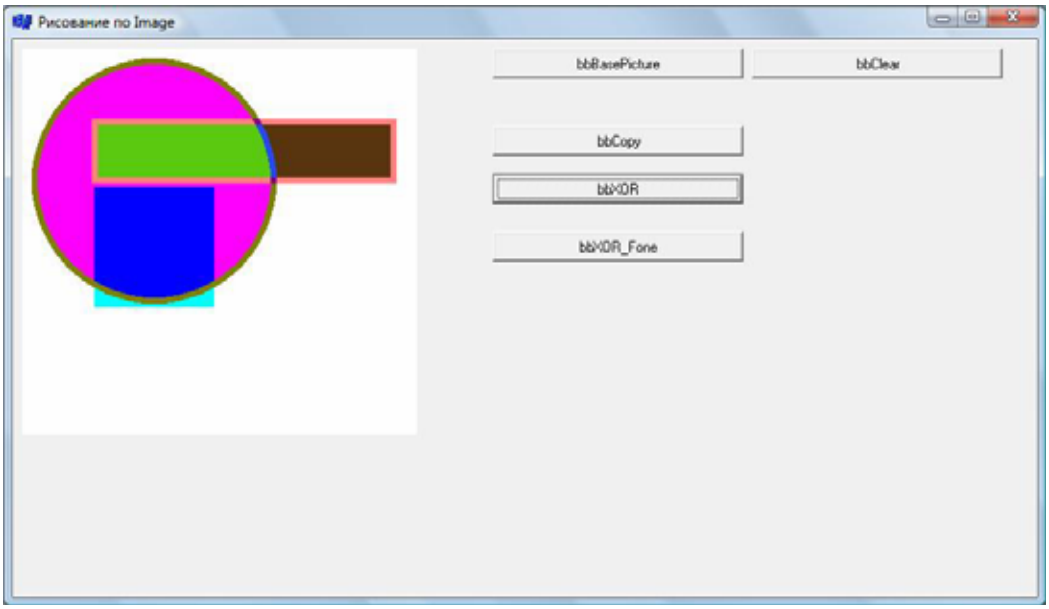


Рис. 17.2. Изображение, сформированное с использованием режима XOR с неправильным отображением цвета

Если теперь повторно выполнить процедуру `bbXOR`, то получим изображение базовой статичной части в «чистом виде», как это требуется.

ПОВЫСИМ ИНТЕЛЛЕКТ

Можно немного повысить «интеллект» процедуры рисования переменной части, например, заставив ее на фоновых участках изображения отображать цвета переменной части корректно. Для этого необходимо выполнять рисование *не заданным цветом*, а цветом, который *в сочетании* с фоновым даст *заданный цвет*.

Операция ИСКЛЮЧАЮЩЕГО ИЛИ

Операция «исключающего или» `XOR` переводит биты в новое состояние в соответствии с используемой маской. А повторное применение операции с той же маской приводит к «восстановлению» битовых значений.

Можно сделать «опережающий шаг» – сразу задать требуемый цвет. Для этого следует преобразовать требуемый цвет в некую переменную `aux`:

```
TColor aux = clRed ^ clWhite;
```

Здесь заданный цвет `clRed` (красный) выполняет маскирование цвета фона `clWhite` (белый). Теперь при построении изображения, если задавать не цвет `clRed` а `aux`, то этот цвет после операции `XOR` с фоном отобразится на экране как `clRed`, а после второго рисования цвет фона будет восстановлен как `clWhite`. Функция-обработчик события `OnClick` кнопки `Button` с именем `bbXOR_Fone`, выполняющая такое рисование, приведена в листинге 17.4, а результат ее работы – на рис. 17.3.

Листинг 17.4. Процедура формирования переменной части изображения с «правильным» отображением цвета на фоновой части

```
void __fastcall TfoMain::bbXOR_FoneClick(TObject *Sender)
{ Image->Canvas->Pen->Mode = pmXor; // режим рисования
```

```
  TColor auxTeal = clWhite ^ clTeal; // предварительная
  TColor auxSky = clWhite ^ clSkyBlue; // подготовка цвета
      // размер, положение
  int xLT_R = 60, yLT_R = 60, xRB_R = 310, yRB_R = 110;
  Image->Canvas->Pen->Color = auxTeal;
  Image->Canvas->Brush->Color = auxSky;
      // построение прямоугольника
  Image->Canvas->Rectangle(xLT_R, yLT_R, xRB_R, yRB_R);
      // подготовка требуемого цвета
  TColor aux = clRed ^ clWhite;
      // построение квадрата
  int xLT_S = 60, yLT_S = 115, xRB_S = 160, yRB_S = 215;
  for (int iy = yLT_S; iy < yRB_S; iy++)
    for (int ix = xLT_S; ix < xRB_S; ix++)
      Image->Canvas->Pixels [ix][iy] = aux;
}
```

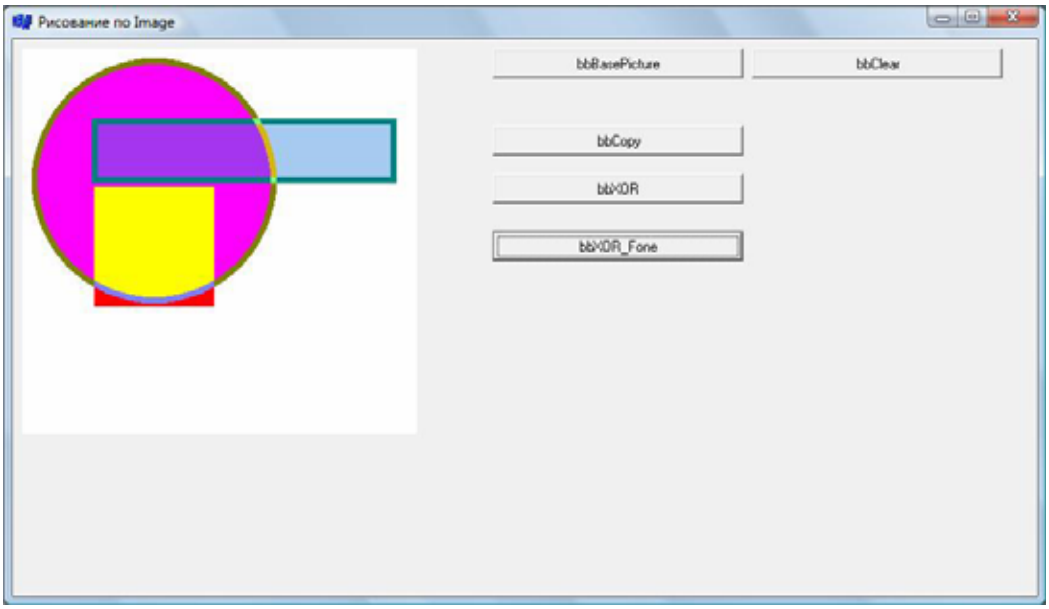


Рис. 17.3. Изображение с «правильным» отображением цвета на фоновой части

ВЫВОД

Нет необходимости рисовать статичную часть, достаточно рисовать только переменную часть дважды в режиме `pmXOR`:

- первое рисование отображает переменную часть в неправильной цветовой гамме;
- второе рисование «стирает» переменную часть, оставляя неискаженное изображение базовой части.

Задача 17.2. В окне `Image` помещено изображение некоторых графических объектов. Требуется стереть/очистить это изображение.

Решение

Стирание изображения или его фрагмента является важной операцией графического редактора. Единственный простой и универсальный метод решения – это нарисовать на всем поле изображения прямоугольник, цвет границы и области которого совпадает с цветом фона.

Если же цвет фона белый, что бывает довольно часто, то цвета пера `Pen` и кисти `Brush` можно не перенастраивать, а воспользоваться опять же свойством `Mode` пера, но на этот раз установить значение `pmWhite`, что обозначает *рисовать только белым цветом, несмотря на заданный цвет*. Заметим, что функция очистки используется редко, поэтому, вероятно, было бы правильно, чтобы после своей работы функция очистки `bbClear` приводила бы все параметры в прежнее состояние, здесь это только режим рисования – свойство `Mode`. Код процедуры обработки события `OnClick` кнопки `Button` с именем `bbClear`, выполняющую очистку изображения, представлен в листинге 17.5.

Листинг 17.5. Процедура очистки изображения белым фоном

```
void __fastcall TfoMain::bbClearClick(TObject *Sender)
{
            // запомнить прежний режим
  TPenMode tmp = Image->Canvas->Pen->Mode;
  Image->Canvas->Pen->Mode = pmWhite; // установить новый
            // установить границы области стирания
  int xLT_R = 0, yLT_R = 0, xRB_R = Image->Width, yRB_R = Image->Height;
            // нарисовать «стирающий» прямоугольник
  Image->Canvas->Rectangle(xLT_R, yLT_R, xRB_R, yRB_R);

  Image->Canvas->Pen->Mode = tmp; // восстановить режим
}
```

Задача 17.3. Имеется некая картинка, состоящая из изображения неизменной части и изображения отрезка прямой линии. Требуется отредактировать изображение этого отрезка – изменить его положение.

Решение

Пусть задача содержит простое изображение с отрезком прямой линии (рис. 17.4). Для общности случая полагаем, что часть отрезка прямой рисуется *над некоторыми* элементами изображения (круг), а другие элементы (прямоугольник) *закрывают* часть отрезка.

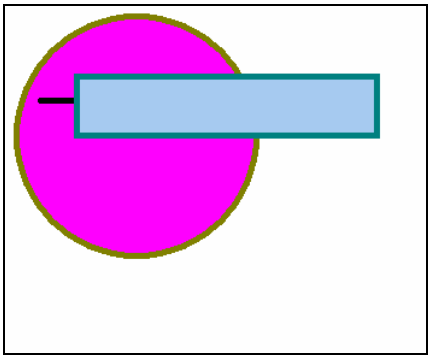


Рис. 17.4. Изображение, содержащее отрезок прямой

Предлагается следующая стратегия решения:

1. Определяется, какие элементы/фигуры изображения будут подвергаться изменению.
2. На основании такой классификации элементов определяются статическая и динамическая области изображения.

3. Определяется порядок действий для пользователя, которому будет предоставлена возможность корректировать основное изображение.
4. Подготавливаются специальные процедуры для обеспечения действий пользователя, а также для редактирования каждого элемента из динамической области.

Подготовка решения задачи в соответствии с описанной стратегией.

1. На основании условия задачи можно сделать вывод, что изменению подвергается один элемент – отрезок прямой.
2. Область изображения условно разбивается на две составные части: статичное изображение (круг и прямоугольник) и динамическое изображение (прямая, которая должна подвергаться редактированию).
3. Предлагается обеспечить следующий порядок работы с приложением:
   а) на старте программа отображает полное изображение;
   б) при необходимости пользователь с помощью кнопок (назовем их `bbBasePicture` и `bbClear`) может вновь строить изображение или удалять его;
   в) посредством соответствующей кнопки (назовем ее `sbEditLine`) пользователь может включать или выключать режим редактирования отрезка;
   г) в режиме редактирования программа поверх картинки строит изображение редактируемого отрезка с квадратиками на концах. В этом режиме пользователь *может перемещать концевые квадратики* отрезка и тем самым выполнять редактирование положения этого отрезка.

Интерфейс приложения может иметь вид, как на рис. 17.5.

Подготовка процедур

Самой важной составляющей при разработке процедур является решение задачи редактирования элемента (в данном случае отрезка). При редактировании какого-либо элемента можно воспользоваться одним из двух способов:

1) рисовать неизменную часть изображения, а редактируемый элемент рисовать только в текущем состоянии, то есть в том положении, какое он занимает в данный момент процесса редактирования;
2) рисовать все изображение полностью, а поверх этого изображения – редактируемый элемент повторно. При таком способе пользователь может видеть и прежнее, и текущее положение объекта.

В данном решении будем реализовывать второй вариант. В любом случае РЕДАКТИРУЕМЫЙ ОБЪЕКТ ОТРАЖАЕТСЯ С ДОПОЛНИТЕЛЬНЫМИ ГРАФИЧЕСКИМИ ЭЛЕМЕНТАМИ, которые можно перемещать манипулятором «мышь».

Дополнительные графические элементы

Дополнительные графические элементы – это небольшие квадратики с небольшой площадью, размещенные по контуру редактируемого изображения.

Поскольку в данной задаче редактируется только отрезок прямой, то достаточно иметь два квадратика на концах отрезка. При этом надо понимать, что в момент редактирования пользователь может выполнить щелчок по любому квадратику и начать его перемещение. Так как каждый квадратик имеет небольшую площадь,

то щелчок обычно выполняется не в центр этого квадратика, а рядом с ним. Этот факт необходимо учитывать при реализации процедур редактирования.

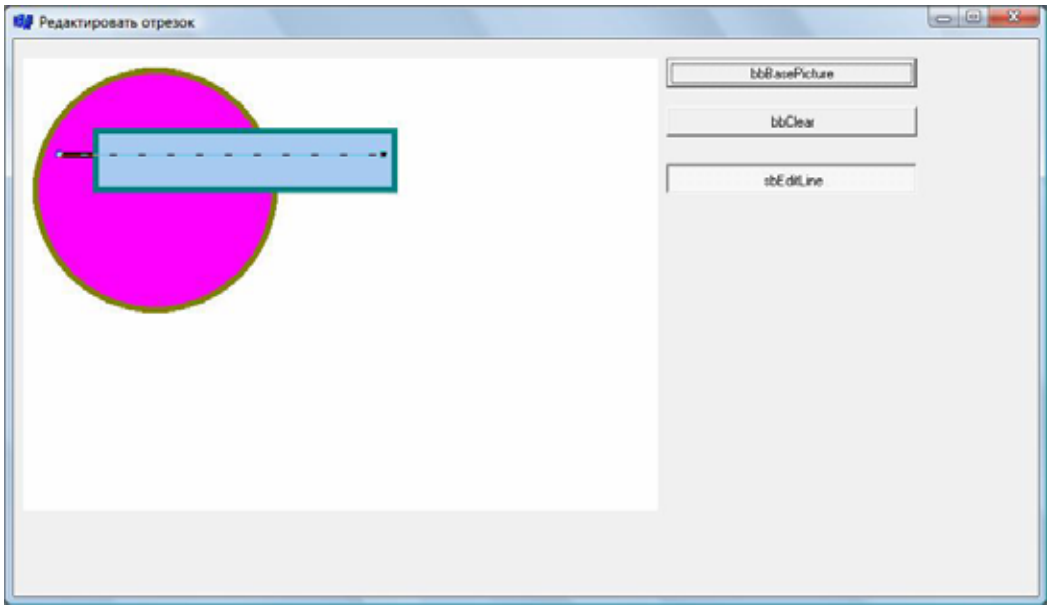


Рис. 17.5. Интерфейс программы в режиме редактирования отрезка

Для того чтобы предоставить пользователю возможность выполнения действий в соответствии с описанным порядком и особенностями решения задачи редактирования, целесообразно реализовать следующие подпрограммы:

- `DrawFullPict();` – подпрограмма отображения полного изображения;
- `DrawEditLine();` – подпрограмма отображения редактируемого отрезка в режиме «исключающего или»;
- `bbBasePictureClick(TObject *Sender);` – обработчик события `OnClick` кнопки `bbBasePicture`, предназначенной для построения полного изображения;
- `bbClearClick(TObject *Sender);` – обработчик события `OnClick` кнопки `bbClear`, предназначенной для очистки изображения;
- `FormCreate(TObject *Sender);` – обработчик события «на создание формы»;
- `sbEditLineClick(TObject *Sender);` – обработчик события `OnClick` кнопки `SpeedButton` с именем `sbEditLine`, предназначенной для переключения приложения в режим редактирования отрезка и обратно;
- `ImageMouseDown(TObject *Sender, TMouseButton Button, TShiftState Shift, int X, int Y); ImageMouseMove(TObject *Sender, TShiftState Shift, int X, int Y); ImageMouseUp(TObject *Sender, TMouseButton Button, TShiftState Shift, int X, int Y);` – обработчики событий «мыши», предназначенные для управления изображением во время редактирования отрезка:
  - нажатие левой кнопки «мыши» над компонентом `Image`;
  - отпускание левой кнопки «мыши» над компонентом `Image`;
  - перемещение нажатой «мыши» над компонентом `Image`.

Кнопка `sbEditLine` играет роль и обычной кнопки, и переключателя. Свои действия она определяет по своему состоянию – нажата или нет. Напомним, чтобы

заставить эту кнопку «залипать», необходимо ее свойству `AllowAllUp` присвоить значение `true`, а свойству `GroupIndex` значение, отличное от нуля.

Примечание

В силу простоты данного примера для хранения информации о местоположения отрезка введем глобальные целочисленные переменные: `FxLT_L`, `FyLT_L`, `FxRB_L`, `FyRB_L`, которые определим в приватной области класса формы. Для оперативных координат конечных точек отрезка введем переменные: `gxLT_L`, `gyLT_L`, `gxRB_L`, `gyRB_L` – описанные в заголовочном файле. То есть эти координаты будут использоваться во время редактирования. Там же описана переменная `grectSize`, хранящая размеры элементов, отображающих точки редактирования. Значение ей можно задать там же – в заголовочном файле.

Отображение полного изображения

Процедура `DrawFullPict` не имеет параметров. В процедуре в режиме копирования отображаются круг, отрезок прямой линии и прямоугольник. Каждая из фигур закрашивается своим цветом. В начале программы производится очистка области рисунка. Код процедуры представлен в листинге 17.6.

Листинг 17.6. Процедура построения полного изображения

```
void __fastcall TfoMain::DrawFullPict()
{
  bbClearClick (NULL); // очистить область рисунка
  Image->Canvas->Pen->Mode = pmCopy;  // режим копирования
      // установить координаты для эллипса
  int xLT_E = 10, yLT_E = 10, xRB_E = 210, yRB_E = 210;
  Image->Canvas->Pen->Color = clOlive; // цвет пера
  Image->Canvas->Pen->Width = 5; // ширина пера
  Image->Canvas->Brush->Color = clFuchsia; // цвет кисти
            // нарисовать круг
  Image->Canvas->Ellipse(xLT_E, yLT_E, xRB_E, yRB_E);
            // построить отрезок
  Image->Canvas->Pen->Color = clBlack; // цвет пера
  Image->Canvas->MoveTo (FxLT_L, FyLT_L); // начало
  Image->Canvas->LineTo (FxRB_L, FyRB_L); // отрезок
            // построить прямоугольник
  int xLT_R = 60, yLT_R = 60, xRB_R = 310, yRB_R = 110;
  Image->Canvas->Pen->Color = clTeal;
  Image->Canvas->Brush->Color = clSkyBlue;
  Image->Canvas->Rectangle(xLT_R, yLT_R, xRB_R, yRB_R);
}
```

Отображения редактируемого отрезка в режиме «исключающего или»

Процедура `DrawEditLine` не имеет параметров. Редактируемый элемент – отрезок прямой – отображается в прямоугольную область (прямоугольник), который «охватывает» отрезок. Координаты этой области определяются на основании значения оперативных координат редактируемого отрезка и размеров дополнительных графических элементов, необходимых для редактирования.

Для оперативного управления положением редактируемого отрезка необходимо иметь четыре переменные, которые будут хранить оперативные координаты отрезка во время редактирования: в данном случае `gxLT_L`, `gyLT_L`, `gxRB_L`, `gyRB_L`. Значения этих переменных определяются из текущего положения

конечных точек отрезка при старте режима редактирования, то есть при нажатии кнопки `sbEditLine`.

Как было отмечено выше, по контуру редактируемого изображения располагаются графические элементы – небольшие квадратики. Каждый из таких квадратиков имеет фиксированный размер стороны, значение которого хранится в переменной `grectSize`, определенной и инициализированной в заголовочном файле. Для выделения области редактирования необходимо выполнить щелчок «мышью» по этой области. Этот щелчок в случае рисования линии может оказаться где-то близко от нее, но при этом будет считаться, что линия «захвачена». Для учета этого факта необходимо несколько расширить редактируемую область, нежели просто ограничиться областью отрезка. Для этого следует воспользоваться значением переменной `grectSize`.

Алгоритм процедуры отображения прямоугольной области можно описать следующими действиями:

1) устанавливаются параметры пера;
2) рисуется отрезок прямой;
3) пересчитываются координаты прямоугольника отображения на основании оперативных координат и величины `grectSize`;
4) отображается прямоугольник с изображением.

Описание процедуры `DrawEditLine` приведено в листинге 17.7.

Листинг 17.7. Процедура отображения редактируемого отрезка изображения

```
void __fastcall TfoMain::DrawEditLine()
{
      // устанавливаются параметры пера
  Image->Canvas->Pen->Mode = pmXor;
  Image->Canvas->Pen->Color = clRed;
  Image->Canvas->Pen->Style = psDash;
  Image->Canvas->Pen->Width = 1;
      // рисуется отрезок прямой
  Image->Canvas->MoveTo (gxLT_L, gyLT_L);
  Image->Canvas->LineTo (gxRB_L, gyRB_L);
      // пересчитываются координаты
  int xLT_R = gxLT_L - grectSize, yLT_R = gyLT_L - grectSize,
      xRB_R = gxLT_L + grectSize, yRB_R = gyLT_L + grectSize;
  Image->Canvas->Rectangle(xLT_R, yLT_R, xRB_R, yRB_R);

      xLT_R = gxRB_L - grectSize, yLT_R = gyRB_L - grectSize,
      xRB_R = gxRB_L + grectSize, yRB_R = gyRB_L + grectSize;
      // построение прямоугольника
  Image->Canvas->Rectangle(xLT_R, yLT_R, xRB_R, yRB_R);
}
```

Обработчик события кнопки, предназначенной для построения полного изображения

В действия обработчика события `OnClick` кнопки `bbBasePicture` входит только одно – вызов процедуры полного построения изображения. Описание обработчика события `OnClick` кнопки `bbBasePicture` приведено в листинге 17.8.

Листинг 17.8. Текст обработчика события «Построить полное изображение»

```
void __fastcall TfoMain::bbBasePictureClick(TObject *Sender)
{
```

```
  DrawFullPict();
}
```

Обработчик события OnClick кнопки bbClear, предназначенной для очистки изображения

Алгоритм работы этого обработчика сводится к следующим действиям:

1) запоминается текущий режим пера;
2) устанавливается режим «Рисовать белым цветом»;
3) устанавливаются границы для прямоугольной области стирания;
4) отображается прямоугольник на область стирания;
5) восстанавливается сохраненный режим.

Описание обработчика события OnClick кнопки bbClear приведено в листинге 17.9.

Листинг 17.9. Процедура очистки области изображения

```
void __fastcall TfoMain::bbClearClick(TObject *Sender)
{
      // запомнить текущий режим
  TPenMode tmp = Image->Canvas->Pen->Mode;
      // режим «рисовать белым цветом»
  Image->Canvas->Pen->Mode = pmWhite;
      // установить границы
  int xLT_R = 0, yLT_R = 0, xRB_R = Image->Width, yRB_R = Image->Height;
      // отобразить прямоугольник
  Image->Canvas->Rectangle(xLT_R, yLT_R, xRB_R, yRB_R);
      // восстановить сохраненный режим
  Image->Canvas->Pen->Mode = tmp;
}
```

Переключение приложения в режим редактирования отрезка и обратно

Действия обработчика события OnClick кнопки sbEditLine, предназначенной для переключения режима редактирования, основаны на оценке того факта, произошло или нет событие «Включен режим редактирования».

1. Если режим включен, то выполняется следующее:
   а) формируются оперативные координаты редактируемого отрезка;
   б) вызывается процедура отображения области редактирования отрезка.
2. Если режим не включен, то выполняется следующее:
   а) формируются параметры области полного изображения;
   б) рисуется полное изображение.

Описание текста процедуры обработчика приведено в листинге 17.10.

Листинг 17.10. Процедура отображения редактируемого отрезка изображения

```
void __fastcall TfoMain::sbEditLineClick(TObject *Sender)
{
  if (sbEditLine->Down) // если кнопка нажата
    {
            // сформировать оперативные координаты
       gxLT_L = FxLT_L, gyLT_L = FyLT_L,
       gxRB_L = FxRB_L, gyRB_L = FyRB_L;
     DrawEditLine(); // отобразить область редактирования
    }
  else // если кнопка не нажата
   {
```

```
            // сформировать параметры области отображения
      FxLT_L = gxLT_L, FyLT_L = gyLT_L,
       FxRB_L = gxRB_L, FyRB_L = gyRB_L;
     DrawFullPict(); // полное изображение
    }
}
```

Обработчик события «Нажатие левой кнопки ‘мыши’ над компонентом `Image`»

Событие OnMouseDown

Обработчик события `OnMouseDown` содержит два параметра `X` и `Y`. Эти параметры определяют координаты указателя «мыши». Параметр `Button` передает код нажатой кнопки «мыши». Два оставшихся параметра этой процедуры в данном приложении не используются.

В задачу обработчика события входит следующая проверка: включен или не включен режим редактирования. Когда режим редактирования включен, то необходимо выполнить:

1. Если нажата левая кнопка «мыши» и ее указатель находится в области квадратика редактирования левой точки отрезка, то запомнить величину смещения точки нажатия «мыши», относительно левого конца отрезка.
2. Если нажата левая кнопка «мыши» и ее указатель находится в области квадратика редактирования правой точки отрезка, то запомнить величину смещения точки нажатия «мыши», относительно правого конца отрезка.

Чтобы запомнить, на сколько пикселей пользователь «промахнулся» мимо конца отрезка, используются переменные `gshX`, `gshY`. Описание обработчика события `OnMouseDown` приведено в листинге 17.11.

Листинг 17.11. Процедура отображения редактируемого отрезка изображения

```
void __fastcall TfoMain::ImageMouseDown (TObject *Sender,
       TMouseButton Button, TShiftState Shift, int X, int Y)
{
  if (sbEditLine->Down) // включен режим редактирования
   {
      if (Button == mbLeft) // нажата левая кнопка
         if (abs (X - gxLT_L) <= grectSize &&
                          abs (Y - gyLT_L) <= grectSize)
      {
           // щелкнули по начальной точке
       gNumPnt = 0;  // указываем – первая точка
       gshX = X - gxLT_L;  // запоминаем смещение щелчка
       gshY = Y - gyLT_L;
      }
     if (abs (X - gxRB_L) <= grectSize &&
                           abs (Y - gyRB_L) <= grectSize)
      {
           // щелкнули по конечной точке
       gNumPnt = 1;  // указываем – вторая точка
       gshX = X - gxRB_L;  // запоминаем смещение щелчка
       gshY = Y - gyRB_L;
      }
   }
}
```

Обработчик события «Отпускание левой кнопки ‘мыши’ над компонентом Image»



Обработчик события OnMouseUp содержит также пять параметров. Здесь тоже используются X и Y, которые определяют координаты указателя «мыши». Другие параметры аналогичны обработчику OnMouseDown.

Переменные gxLT_L, gyLT_L, gxRB_L, gyRB_L хранят оперативные координаты отрезка, которые при отпускании кнопки «мыши» требуется зафиксировать. Следовательно, алгоритм процедуры сводится к запоминанию новых значений в областях глобальных переменных FxLT_L, FyLT_L, FxRB_L, FyRB_L, которые должны теперь совпадать с gxLT_L, gyLT_L, gxRB_L, gyRB_L. И изображения самого отрезка, и его редактируемого двойника будут совпадать только после отпускания кнопки «мыши». Поэтому заново отображаются полный рисунок и отрезок прямой линии. Описание обработчика события OnMouseUp приведено в листинге 17.12.

Листинг 17.12. Описание действий на событие «Отпускание левой кнопки ‘мыши’ над компонентом Image»

```
void __fastcall TfoMain::ImageMouseUp(TObject *Sender,
                 TMouseButton Button, TShiftState Shift, int X, int Y)
{
  if (sbEditLine->Down) // включен режим редактирования
    if (Button == mbLeft) // нажата левая кнопка
      {
         gNumPnt = -1;  // отменяем указание точки
         FxLT_L = gxLT_L, FyLT_L = gyLT_L, FxRB_L = gxRB_L,
           FyRB_L = gyRB_L;
     DrawFullPict(); // полное изображение
     DrawEditLine(); // редактируемая линия
    }
}
```

Обработчик события «Передвижение нажатой ‘мыши’ над компонентом Image»



Обработчик события OnMouseMove содержит четыре параметра, среди которых X и Y определяют координаты указателя «мыши». А код кнопки «мыши» приходится узнавать из параметра Shift.

В процедуре используются координаты указателя «мыши» X и Y, на основании значений которых пересчитываются оперативные координаты отрезка – первой и второй точек. Глобальные переменные gshX и gshY, описанные в заголовочном файле и проинициализированные в обработчике нажатия кнопки «мыши», помогают правильно корректировать координаты элементов редактирования, чтобы складывалось впечатление, что эти элементы перемещаются за ту же точку, как и при захвате. Описание обработчика события OnMouseMove приведено в листинге 17.13.

Листинг 17.13. Описание действий на событие «Перемещение ‘мыши’ над компонентом `Image`»

```
void __fastcall TfoMain::ImageMouseMove(TObject *Sender,
  TShiftState Shift, int X, int Y)
{ if (Shift.Contains(ssLeft))  // нажата левая кнопка
   if (sbEditLine->Down)
    {
     DrawEditLine();
     switch (gNumPnt) // номер точки
      {
       case 0 : gxLT_L = X + gshX; gyLT_L = Y + gshY; break;
       case 1 : gxRB_L = X + gshX; gyRB_L = Y + gshY;
      }
     DrawEditLine();
    }
}
```

Обработчик события «На создание формы»

Обработчик события создания формы инициализирует поля класса и обращается к процедуре рисования всего изображения. Текст обработчика приведен в листинге 17.14.

Листинг 17.14. Описание действий на событие «Создать форму»

```
void __fastcall TfoMain::FormCreate(TObject *Sender)
{
  FxLT_L = 30, FyLT_L = 80, FxRB_L = 300, FyRB_L = 80;
  DrawFullPict();
}
```

Задача 17.4. Имеется некая картинка, состоящая из изображения неизменной части и ломаной линии. Требуется отредактировать изображение ломаной линии – изменить положение ее узлов.

Решение

Задача сходна с предыдущей задачей. Структуру проекта можно сохранить. Принципиальное отличие состоит в структуре данных: редактированию будут подвергаться не две точки, а набор точек с произвольным количеством. Поэтому отличие реализации данного проекта заключается в том, что координаты вершин ломаной линии *необходимо хранить в массиве*, а процедуры отображения полного изображения и отображения редактируемой ломаной линии должны *включать циклы построения* отображения набора отрезков или вызов соответствующих методов холста. На рис. 17.6 приведен вид интерфейса программы, включенной в режим редактирования ломаной линии.

Для хранения координат вершин ломаной линии вместо четырех переменных будем использовать массив `FPnts`, состоящий из элементов типа `TPoint`. Переменная `FCount` хранит количество элементов в этом массиве. А для временного хранения координат ломаной линии, точнее ее дубликата, в момент редактирования будет использоваться массив `gPnts`. Этих изменений в интерфейсной части достаточно для перехода от редактирования отрезка к редактированию ломаной линии.

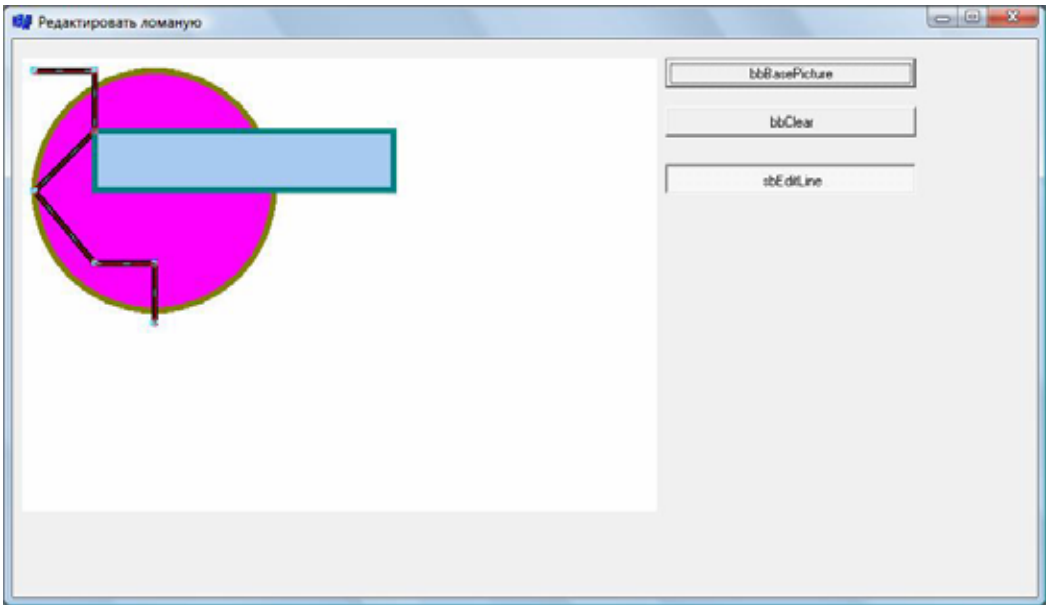


Рис. 17.6. Интерфейс программы в режиме редактирования ломаной линии

Изменения реализации методов проекта касается почти всех подпрограмм. Но эти изменения почти однотипные:

- присваивание значений отдельным переменным заменятся на циклы;
- отображения единственного отрезка – на циклы отображения набора отрезков, составляющих ломаную линию.

Тексты обновленных подпрограмм приведены в листингах 17.15–17.21.

Отображение полного изображения рисунка

Листинг 17.15. Построение полного изображения

```
void __fastcall TfoMain::DrawFullPict()
{
  bbClearClick (NULL);
  Image->Canvas->Pen->Mode = pmCopy;  // режим копирования

  int xLT_E = 10, yLT_E = 10, xRB_E = 210, yRB_E = 210;
  Image->Canvas->Pen->Color = clOlive;
  Image->Canvas->Pen->Width = 5;
  Image->Canvas->Brush->Color = clFuchsia;
  Image->Canvas->Ellipse(xLT_E, yLT_E, xRB_E, yRB_E);

       // построение ломаной линии
  Image->Canvas->Pen->Color = clBlack;
  Image->Canvas->MoveTo (FPnts[0].x, FPnts[0].y);
  for (int i = 1; i < FCount; i++)
    Image->Canvas->LineTo (FPnts[i].x, FPnts[i].y);

  int xLT_R = 60, yLT_R = 60, xRB_R = 310, yRB_R = 110;
  Image->Canvas->Pen->Color = clTeal;
  Image->Canvas->Brush->Color = clSkyBlue;
  Image->Canvas->Rectangle(xLT_R, yLT_R, xRB_R, yRB_R);
}
```

Описание обработчика события «На создание формы»

Листинг 17.16. Реализация процедуры инициализации ломаной линии

```
void __fastcall TfoMain::FormCreate(TObject *Sender)
{ FCount = cCount;
  FPnts[0].x = 10; FPnts[0].y = 10;
  FPnts[1].x = 60; FPnts[1].y = 10;
  FPnts[2].x = 60; FPnts[2].y = 60;
  FPnts[3].x = 10; FPnts[3].y = 110;
  FPnts[4].x = 60; FPnts[4].y = 170;
  FPnts[5].x = 110; FPnts[5].y = 170;
  FPnts[6].x = 110; FPnts[6].y = 220;
  DrawFullPict();
}
```

Отображения редактируемой ломаной линии в режиме «исключающего или»

Листинг 17.17. Реализация процедуры рисования редактируемой ломаной линии

```
void __fastcall TfoMain::DrawEditLine()
{ Image->Canvas->Pen->Mode = pmXor;
  Image->Canvas->Pen->Color = clRed;
  Image->Canvas->Pen->Style = psDash; // psInsideFrame;
  Image->Canvas->Pen->Width = 1;

       // построение ломаной линии
  Image->Canvas->MoveTo (gPnts[0].x, gPnts[0].y);
  for (int i = 1; i < FCount; i++)
    Image->Canvas->LineTo (gPnts[i].x, gPnts[i].y);

       // построение квадратиков на вершинах линии
  for (int i = 0; i < FCount; i++)
   {
    int xLT_R = gPnts[i].x - grectSize, yLT_R = gPnts[i].y - grectSize,
        xRB_R = gPnts[i].x + grectSize, yRB_R = gPnts[i].y + grectSize;
    Image->Canvas->Rectangle(xLT_R, yLT_R, xRB_R, yRB_R);
   }
}
```

Обработчик события кнопки, предназначенной для переключения приложения в режим редактирования отрезка и обратно

Листинг 17.18. Реализация процедуры переключения режима редактирования

```
void __fastcall TfoMain::sbEditLineClick(TObject *Sender)
{ if (sbEditLine->Down)
   {for (int i = 0; i < FCount; i++)
     {gPnts[i].x = FPnts[i].x; gPnts[i].y = FPnts[i].y;}
    DrawEditLine();
   }
  else
   {for (int i = 0; i < FCount; i++)
     {FPnts[i].x = gPnts[i].x; FPnts[i].y = gPnts[i].y;}
    DrawFullPict();
   }
}
```

Обработчик события «Нажатие левой кнопки ‘мыши’ над компонентом `Image`»

Листинг 17.19. Реализация процедур проекта редактирования ломаной линии

```
void __fastcall TfoMain::ImageMouseDown (TObject *Sender,
       TMouseButton Button, TShiftState Shift, int X, int Y)
{ if (sbEditLine->Down) // включен режим редактирования
   {if (Button == mbLeft) // нажата левая кнопка
     for (int i = 0; i < FCount; i++)
       if (abs (X - gPnts[i].x) <= grectSize &&
                                   abs (Y - gPnts[i].y) <= grectSize)
        { // щелкнуть по точке редактирования
         gNumPnt = i;  // указать номер точки
         gshX = X - gPnts[i].x; // запомнить смещение щелчка
         gshY = Y - gPnts[i].y;
        }
   }
}
```

Обработчик события «Перемещение ‘мыши’ с нажатой кнопкой над компонентом `Image`»

Листинг 17.20. Реализация процедуры перемещения «мыши»

```
void __fastcall TfoMain::ImageMouseMove(TObject *Sender,
                                    TShiftState Shift, int X, int Y)
{ if (Shift.Contains(ssLeft))  // нажата левая кнопка
   if (sbEditLine->Down)
    {
     DrawEditLine();
     gPnts[gNumPnt].x = X + gshX; gPnts[gNumPnt].y = Y + gshY;
     DrawEditLine();
    }
}
```

Обработчик события «Отпускание левой кнопки ‘мыши’ над компонентом `Image`»

Листинг 17.21. Реализация процедур проекта редактирования ломаной линии

```
void __fastcall TfoMain::ImageMouseUp(TObject *Sender,
              TMouseButton Button, TShiftState Shift, int X, int Y)
{ if (sbEditLine->Down) // включен режим редактирования
   if (Button == mbLeft) // нажата левая кнопка
    {gNumPnt = -1;  // отменяем указание точки
     for (int i = 0; i < FCount; i++)
      {FPnts[i].x = gPnts[i].x, FPnts[i].y = gPnts[i].y;}
     DrawFullPict();
     DrawEditLine();
    }
}
```

Задача 17.5. Имеется некий рисунок, состоящий из изображения неизменной части и изображения кривой Безье. Требуется отредактировать изображение кривой Безье – изменить положение ее характеристических узлов, то есть вид и положении кривой.

Решение

В ходе развития математических аспектов и алгоритмов компьютерной графики оказалось, что полиномы третьей степени и сплайны, построенные из таких полиномов, являются очень гибкими и простыми инструментами при построении произвольных кривых.

СПЛАЙН

Сплайн – это функция, определенная на заданном отрезке, совпадающая на частичных отрезках с некоторыми алгебраическими многочленами степени, не выше заданной (например, m) и имеющая непрерывную (m – 1) производную.

Одной из форм описания сплайнов третьей степени является описание их в форме кривых Безье. Не вдаваясь в математические аспекты построения кривой Безье, укажем только, что форма дуги каждого звена, то есть кубического полинома, определяется вершинами характеристической ломаной линии, состоящей из трех звеньев.

ХАРАКТЕРИСТИЧЕСКАЯ ЛОМАНАЯ

Характеристическая ломаная – это кусочно-линейная линия, которая формирует некий каркас для кривой Безье. Каждые четыре вершины ломаной определяют параметры одного звена кривой Безье.

Пример кривой Безье и ее характеристической ломаной приведен на рис. 17.7. Первые четыре вершины ломаной определяют форму первого кубического звена. Следующая четверка вершин – форму второго кубического звена кривой Безье. При этом надо помнить, что *последняя* вершина первой четверки является *первой* вершиной второй четверки вершин. То есть в данном примере ломаная, имеющая *семь* вершин, управляет формой кривой Безье, составленной из *двух* кубических звеньев.

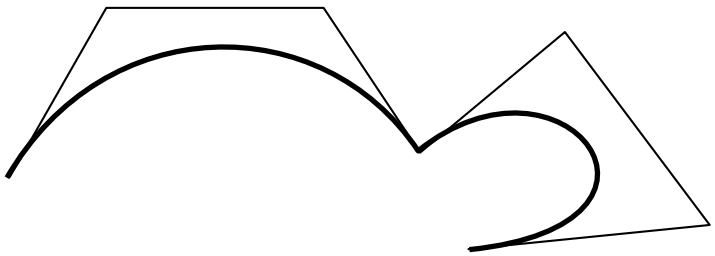


Рис. 17.7. Пример кривой Безье и ее характеристической ломаной

В общем случае для управления формой кривой Безье, составленной из N кубических звеньев, необходимо иметь ломаную линию с ($3N + 1$) вершинами. И этот факт необходимо точно соблюдать при построении кривой Безье с использованием методов холста.

Для построения кривой Безье в системе *C++ Builder* достаточно вызвать соответствующий метод *Canvas* – `PolyBezier`.

Функция PolyBezier

Метод *Canvas* – `PolyBezier` имеет прототип:

```
void __fastcall PolyBezier (const TPoint* Points,
                            const int Points_Size);
```

где первый параметр – указатель на массив из элементов типа `TPoint`, а второй – индекс последнего элемента. Обратите внимание, что второй параметр задает *не количество* элементов в массиве, а *индекс* последнего элемента.

Кроме этого метода, класс *Canvas* имеет метод `PolyBezierTo`. Единственное отличие этого метода состоит в том, что графический курсор системы после вывода кривой устанавливается на конечную точку, в то время как в методе `PolyBezier` курсор не меняет положения.

Во время редактирования кривой приложение должно отображать характеристическую ломаную линию и квадратики, подчеркивающие вершины редактирования. Это позволит пользователю управлять формой кривой с высокой точностью. При этом саму ломаную можно выводить не посредством оператора цикла, а с помощью метода `Polyline` холста. Этот метод имеет такие же параметры, как и метод `PolyBezier`, но строит ломаную линию с вершинами, координаты которых хранятся в массиве `Points`. Поэтому, несмотря на принципиально другой объект, подвергаемый редактированию, данная задача необычайно сходна с предыдущей. Структура проекта практически совпадает с предыдущим проектом. Изменению подвергаются лишь реализация методов `DrawFullPict` и `DrawEditLine`, да и то – только в части отображения ломаной. Точнее, в подпрограмме `DrawFullPict` вместо ломаной линии строится кривая Безье, а в подпрограмме `DrawEditLine` строится и кривая Безье, и ломаная линия с квадратиками в вершинах.

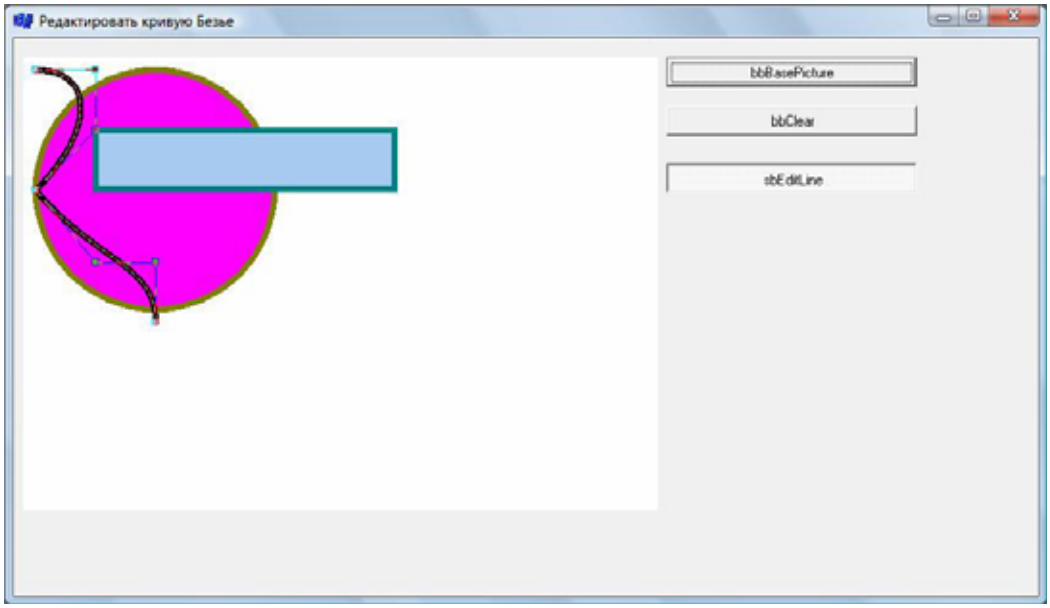


Рис. 17.8. Интерфейс программы в режиме редактирования кривой Безье

Вообще, решение данной задачи заключается лишь в корректировке текста процедур `DrawFullPict` и `DrawEditLine` с целью вывода кривой Безье и ее характеристической линии. Текст этих измененных процедур приведен в листингах 17.22 –

17.23, а пример интерфейса программы в режиме редактирования кривой – на рис. 17.8.

Отображение полного изображения рисунка

Листинг 17.22. Описание процедуры изображения рисунка с кривой Безье

```
void __fastcall TfoMain::DrawFullPict()
{ bbClearClick (NULL);
  Image->Canvas->Pen->Mode = pmCopy;  // режим копирования

  int xLT_E = 10, yLT_E = 10, xRB_E = 210, yRB_E = 210;
  Image->Canvas->Pen->Color = clOlive;
  Image->Canvas->Pen->Width = 5;
  Image->Canvas->Brush->Color = clFuchsia;
  Image->Canvas->Ellipse(xLT_E, yLT_E, xRB_E, yRB_E);

  Image->Canvas->Pen->Color = clBlack;

  Image->Canvas->PolyBezier(&FPnts[0], FCount-1);
               // вычерчивание кривой Безье

  int xLT_R = 60, yLT_R = 60, xRB_R = 310, yRB_R = 110;
  Image->Canvas->Pen->Color = clTeal;
  Image->Canvas->Brush->Color = clSkyBlue;
  Image->Canvas->Rectangle(xLT_R, yLT_R, xRB_R, yRB_R);
}
```

Отображения редактируемой кривой Безье в режиме «исключающего или»

Листинг 17.23. Реализация процедур проекта редактирования кривой Безье

```
void __fastcall TfoMain::DrawEditLine()
{ Image->Canvas->Pen->Mode = pmXor;
  Image->Canvas->Pen->Color = clRed;
  Image->Canvas->Pen->Width = 1;

  Image->Canvas->Pen->Style = psDot; // psInsideFrame;
  Image->Canvas->PolyBezier(&gPnts[0], FCount-1);
               // вычерчивание кривой Безье

  Image->Canvas->Pen->Style = psDash;
  Image->Canvas->MoveTo (gPnts[0].x, gPnts[0].y);
  for (int i = 1; i < FCount; i++)
    Image->Canvas->LineTo (gPnts[i].x, gPnts[i].y);

  Image->Canvas->Pen->Style = psSolid;
  for (int i = 0; i < FCount; i++)
   {
    int xLT_R = gPnts[i].x - grectSize, yLT_R = gPnts[i].y - grectSize,
        xRB_R = gPnts[i].x + grectSize, yRB_R = gPnts[i].y + grectSize;
    Image->Canvas->Rectangle(xLT_R, yLT_R, xRB_R, yRB_R);
   }
}
```

Задача 17.6. Заданное изображение требуется дополнить новыми ломаными. То есть обеспечить возможность построения новой ломаной линии на основании узлов, отмеченных «мышью». Предусмотреть возможность создания замкнутой/не замкнутой линии.

Решение

Предлагается следующая стратегия решения.

1. Определяется последовательность действий, которые необходимо выполнить пользователю для построения изображения ломаной линии.
2. Определяется интерфейс приложения, предназначенного для решения задачи.
3. Подготавливаются специальные процедуры, чтобы обеспечить функциональность приложения в соответствии с предложенным порядком действий.

Подготовка решения в соответствии с описанной стратегией.

1. Для задания ломаной линии необходимо обозначить узлы, ее образующие, а затем соединить эти узлы отрезками прямых линий. Определим правила построения:
   - пользователь находит на корректируемом изображении точку – *начало ломаной* – и фиксирует ее положение нажатием левой кнопки «мыши»;
   - затем находит положение следующего узла и также фиксирует его нажатием левой кнопки «мыши», и так далее до тех пор, пока не будет поставлена последняя точка в изображении;
   - нарисованное «закрепляется», например, с помощью правой кнопки «мыши».
2. В соответствии с поставленной задачей пользователю должна быть предоставлена возможность:
   - загрузить базовое изображение;
   - сообщить программе о своей готовности приступить к рисованию ломаной;
   - нарисовать эту ломаную;
   - очистить или убрать нарисованное изображение при необходимости.

Интерфейс приложения может быть таким, как на рис. 17.9.

3. Для того чтобы обеспечить требуемую функциональность приложения, целесообразно подготовить следующие процедуры и обработчики событий:
   - `bbBasePictureClick (TObject *Sender);` – обработчик события `OnClick` кнопки `bbBasePicture`, предназначенной для построения базового изображения;
   - `bbClearClick (TObject *Sender);` – обработчик события `OnClick` кнопки `bbClear`, предназначенной для очистки изображения;
   - `DrawFullPict();` – подпрограмма построения базового рисунка;
   - `DrawEditLine();` – подпрограмма отображения редактируемого отрезка прямой линии в режиме «исключающего или»;
   - `ImageMouseDown (TObject *Sender, TMouseButton Button, TShiftState Shift, int X, int Y);` – обработчик события «нажать кнопку ‘мыши’» в области изображения;
   - `ImageMouseMove(TObject *Sender, TShiftState Shift, int X, int Y);` – обработчик события «передвинуть ‘мышь’» в области изображения;
   - `sbCreateLineClick(TObject *Sender);` – обработчик события `OnClick` кнопки `sbCreateLine` – «создать линию» в области изображения;
   - `FormCreate (TObject *Sender);` – обработчик события `OnCreate` «на создание формы»;
   - `FormClose (TObject *Sender, TCloseAction &Action);` – обработчик события `OnClose` «на закрытие формы»;

- `sbEditLineClick(TObject *Sender);` – обработчик события `OnClick` кнопки `sbEditLine`, предназначенной для старта режима редактирования ломаной линии.

Примечание

Для реализации описанных процедур и обработчиков событий в программу включены следующие *глобальные* константы и переменные:

```
const cCount = 100; // максимальная длина ломаной – число узлов
const cSizeCurve = 100; // максимальное количество создаваемых
                        // линий за сеанс
TPoint *FPnts [cSizeCurve];  // массив указателей на кривые
int    FCounts [cSizeCurve];  // массив длин каждой из кривых
int    FCount;  // текущее количество кривых
bool   FClosed [cSizeCurve];  // признак замкнутости кривой
```

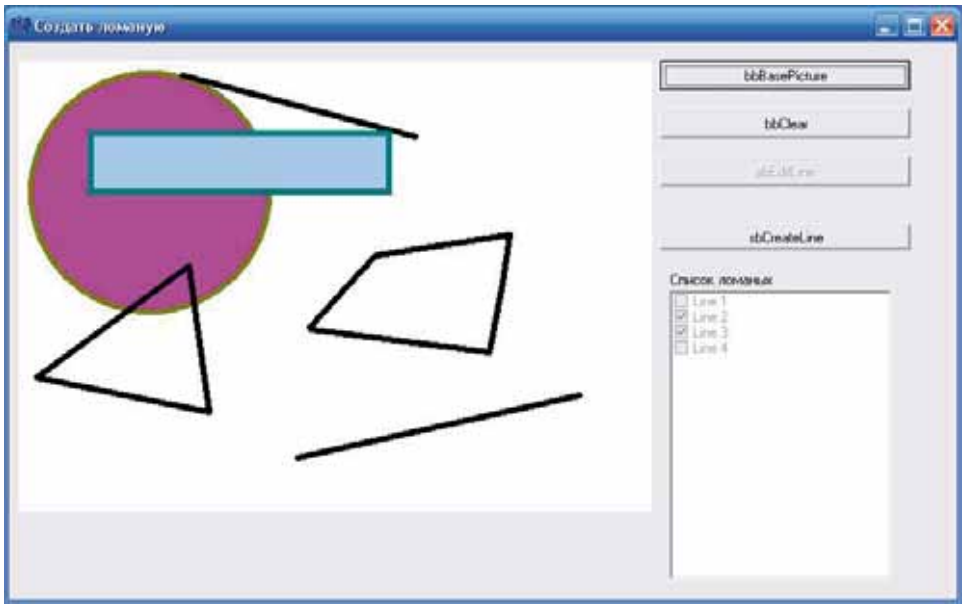


Рис.17.9. Интерфейс приложения, предназначенного для построения ломаных линий в соответствии с условием задачи 17.6



Рис.17.10. Диалоговое окно для подтверждения создания замкнутой кривой

Обработчик события `OnClick` кнопки «Создать базовый рисунок»

Алгоритм обработки этого события сводится к одному действию: вызову метода построения базового изображения. Код обработчика представлен в листинге 17.24.

Листинг 17.24. Действия под кнопку «Базовый рисунок»

```
void __fastcall TfoMain::bbBasePictureClick (TObject *Sender)
{
```

```
   DrawFullPict();
}
```

Обработчик события `OnClick` кнопки, предназначенной для очистки изображения

Алгоритм обработки этого события сводится к выполнению серии действий для очистки области изображения. Код обработчика представлен в листинге 17.25.

Листинг 17.25. Реализация процедуры очистки полотна

```
void __fastcall TfoMain::bbClearClick (TObject *Sender)
{
  TPenMode tmp = Image->Canvas->Pen->Mode;
  Image->Canvas->Pen->Mode = pmWhite;

  int xLT_R = 0, yLT_R = 0,
      xRB_R = Image->Width, yRB_R = Image->Height;
  Image->Canvas->Rectangle(xLT_R, yLT_R, xRB_R, yRB_R);
  Image->Canvas->Pen->Mode = tmp;
}
```

Построение базового рисунка

Листинг 17.26. Описание процедуры построения базового изображения

```
void __fastcall TfoMain::DrawFullPict()
{
  bbClearClick (NULL);
  Image->Canvas->Pen->Mode = pmCopy;  // режим рисования
  int xLT_E = 10, yLT_E = 10, xRB_E = 210, yRB_E = 210;
  Image->Canvas->Pen->Color = clOlive;
  Image->Canvas->Pen->Width = 5;
  Image->Canvas->Brush->Color = clFuchsia;
  Image->Canvas->Ellipse(xLT_E, yLT_E, xRB_E, yRB_E);
  Image->Canvas->Pen->Color = clBlack;
  for (int ind = 0; ind < FCount; ind++)
   {
    Image->Canvas->MoveTo (FPnts[ind][0].x, FPnts[ind][0].y);
    for (int i = 1; i < FCounts [ind]; i++)
      Image->Canvas->LineTo (FPnts[ind][i].x, FPnts[ind][i].y);
    if (FClosed [ind]) Image->Canvas->
                  LineTo (FPnts[ind][0].x, FPnts[ind][0].y);
   }
  int xLT_R = 60, yLT_R = 60, xRB_R = 310, yRB_R = 110;
  Image->Canvas->Pen->Color = clTeal;
  Image->Canvas->Brush->Color = clSkyBlue;
  Image->Canvas->Rectangle(xLT_R, yLT_R, xRB_R, yRB_R);
}
```

Отображение редактируемого отрезка прямой линии в режиме «исключающего или»

Листинг 17.27. Реализация процедуры отображения линии редактирования

```
void __fastcall TfoMain::DrawEditLine()
{ Image->Canvas->Pen->Mode = pmXor;
  Image->Canvas->Pen->Color = clRed;
  Image->Canvas->Pen->Style = psDash;
  Image->Canvas->Pen->Width = 1;
```

```
  Image->Canvas->MoveTo (gPnts[0].x, gPnts[0].y);
  for (int i = 1; i < gCount; i++)
    Image->Canvas->LineTo (gPnts[i].x, gPnts[i].y);

  for (int i = 0; i < gCount; i++)
   {int xLT_R = gPnts[i].x - grectSize, yLT_R = gPnts[i].y - grectSize,
        xRB_R = gPnts[i].x + grectSize, yRB_R = gPnts[i].y + grectSize;
    Image->Canvas->Rectangle (xLT_R, yLT_R, xRB_R, yRB_R);
   }
}
```

Обработчик события «Нажать кнопку ‘мыши’» в области изображения

Листинг 17.28. Реализация процедуры нажатия кнопки «мыши»

```
void __fastcall TfoMain::ImageMouseDown (TObject *Sender,
          TMouseButton Button, TShiftState Shift, int X, int Y)
{ if (sbCreateLine->Down) // включен режим создания
    if (Button == mbLeft) // нажата левая кнопка
     {if (gFirstPnt)  // нажатие первое, только порождаем
       {gPnts[0].x = X; gPnts[0].y = Y;
        gPnts[1].x = X; gPnts[1].y = Y;
        gCount = 2;
        gFirstPnt = false;
       }
      else
       {DrawEditLine ();  // стираем прежнее изображение
        gPnts[gCount].x = X; gPnts[gCount].y = Y;
        gCount++;
       }
      DrawEditLine();
     }
   else  // нажато, но не левая
     if (Button == mbRight && !gFirstPnt) // нажата правая кнопка
      {bool isClosed = false;
       if (abs (X - gPnts [0].x) <= grectSize &&
           abs (Y - gPnts [0].y) <= grectSize) // щелчок близок к первой
         isClosed = mrYes == MessageDlg
                              ("Замкнуть кривую?", mtConfirmation,
                                TMsgDlgButtons()<<mbYes<<mbNo, 0);
       if (isClosed) {gCount--; FClosed [FCount] = true;}
       FPnts [FCount] = new TPoint [gCount];
                                  // заведение нового массива точек
       for (int ind = 0; ind < gCount; ind++)
        {FPnts[FCount][ind].x = gPnts[ind].x;
         FPnts[FCount][ind].y = gPnts[ind].y;
        }
       FCounts [FCount] = gCount;
                          // фиксируем количество точек в этой ломаной
       FCount++;  // количество ломаных увеличилось
       DrawFullPict();
       sbCreateLine->Down = false;  // отключить режим создания
         // отобразить в списке
       clbLineList->Items->Add ("Line " + IntToStr (FCount));
       clbLineList->Checked [clbLineList->Count - 1] = isClosed;
       clbLineList->ItemEnabled [clbLineList->Count - 1] = false;
      }
}
```

Обработчик события «Передвинуть ‘мышь’» в области изображения

Листинг 17.29. Реализация процедуры перемещения «мыши»

```
void __fastcall TfoMain::ImageMouseMove(TObject *Sender,
                                        TShiftState Shift, int X, int Y)
{ if (sbCreateLine->Down)  // если включен режим редактирования
    if (!gFirstPnt)
     {DrawEditLine();
      gPnts[gCount - 1].x = X + gshX;
      gPnts[gCount - 1].y = Y + gshY;
      DrawEditLine();
     }
}
```

Обработчик события «Создать ломаную линию»

Листинг 17.30. Реализация процедуры включения режима создания ломаной

```
void __fastcall TfoMain::sbCreateLineClick(TObject *Sender)
{ if (sbCreateLine->Down)
   {gCount = 0;
    gFirstPnt = true;
   }
}
```

Обработчик события «На создание формы»

Листинг 17.31. Реализация процедуры старта программы

```
void __fastcall TfoMain::FormCreate (TObject *Sender)
{ FCount = 0;  // еще нет кривых
  for (int ind = 0; ind < cSizeCurve; ind++)
   {FPnts [ind] = NULL;
    FCounts [ind] = 0;
    FClosed [ind] = false;
   }
}
```

Обработчик события «На закрытие формы»

Листинг 17.32. Реализация процедуры завершения программы

```
void __fastcall TfoMain::FormClose (TObject *Sender,
                                    TCloseAction &Action)
{ for (int ind = 0; ind < FCount; ind++)
    delete[] FPnts[ind];
}
```

Следует обратить внимание на то, что обработчик события нажатия кнопки «мыши» анализирует кнопку «мыши», которая нажата. При нажатии левой кнопки «мыши» создается очередная вершина ломаной и фиксируется ее положение, а при нажатии правой кнопки фиксируется конечная вершина и пополняется список линий. При этом если линия является замкнутой, то строка с названием линии помечается галочкой перед именем. Так, после создания набора линий всегда можно посмотреть какая линия является замкнутой, а какая нет.



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. Дополнить реализацию проекта к задаче 17.6 кнопками и обработчиками, для реализации возможности сохранения и чтения набора построенных ломаных.
2. Дополнить реализацию проекта возможностью редактирования созданных ломаных линий.
3. Дополнить реализацию проекта возможностью создания и редактирования кривых Безье.
4. Написать программу редактирования простейшего изображения, состоящего из прямоугольников, эллипсов, отрезков прямых, ломаных линий и кривых Безье.

Рекомендации

Поскольку рисунок состоит простейших графических примитивов, то для хранения информации о них достаточно разработать простейшую структуру, где будут запоминаться местоположение и размеры объектов, и их цвет. Рисование можно реализовать единственной функцией, которая будет выполнять рисование каждого объекта в соответствие с его кодом. Такой рисунок также несложно закодировать в текстовом файле.

Для редактирования прямоугольников и эллипсов достаточно разместить четыре квадратика по углам охватывающего прямоугольника. Или их можно разместить посередине от каждой стороны прямоугольника. Этого достаточно, чтобы редактировать параметры прямоугольника или вписанного эллипса. В отличие от редактирования ломаной линии, здесь придется корректировать не только координаты квадратика, который перетаскивается, но и других, так как здесь появляется взаимосвязь координат квадратиков редактирования.

Конечно, можно реализовать и более мощное средство редактирования – восемь точек редактирования. Это увеличит объем средств редактирования, предоставляемых пользователю. Если же добавить девятый квадратик (поместить его в центре), то его перемещение можно обрабатывать как перемещение всего прямоугольника. Варианты размещения квадратиков по сторонам прямоугольника приведены на рис. 17.11.

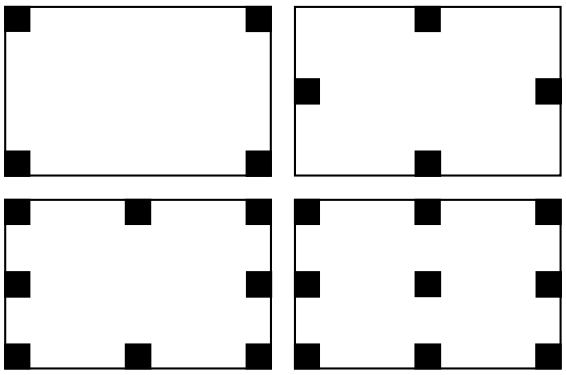


Рис. 17.11. Примеры размещения квадратиков редактирования

Связь приложения с MS Excel и Word по технологии OLE 18

Программирование приложений для работы под *ОС Windows* активно опирается на библиотеку функций *Win API*. Можно сократить затраты на разработку таких приложений, воспользовавшись такой системой визуального программирования, как *C++ Builder*, которая «запечатывает» эту сложную систему процедур в объектно-ориентированную оболочку, оставляя разработчику квинтэссенцию самого необходимого. Результатом этого является мощный инструмент, с использованием которого процесс разработки приложений средней сложности превращается в создание простых приложений. Приложения, исполняемые в операционной системе, можно рассматривать как изделие, состоящее из двух частей – внутренней и внешней.

Внутренняя часть программы – это набор процедур и других элементов, которые способны решать определенный набор задач. Но пользователю необходимо, чтобы программа решала задачи над его данными. Передача данных программе и возврат результатов осуществляется через *внешнюю* часть программы, интерфейс.

Пользователям операционной системы *Windows* хорошо знакомы интерфейсы программы, выполненные в виде окон. Но понятие интерфейса куда шире, чем графические окна. Например, интерфейсная часть классов описывает правила взаимодействия с объектами этого класса.

Возможности воздействия на программу могут быть увеличены. Так, помимо оконного графического интерфейса, программе может быть придан ПРОГРАММНЫЙ интерфейс, аналогичный классам. Тогда подобная программа может быть изготовлена из *двух частей*. Одна часть может быть выполнена в виде приложения, не имеющего графического интерфейса в виде привычного окна и просто работающего в недрах *ОС Windows*. Но при этом она должна иметь интерфейс, через который другие программы могли бы управлять ею: задавать данные и вызывать ее методы, также забирать результаты. Вторая часть, наоборот, должна иметь оконный графический интерфейс для общения с пользователем, с одной стороны, и содержать код, умеющий управлять первой частью, то есть другой программой. Тогда первая программа, по сути, является ядром, «горнилом», способным переплавлять исходные «данные», а вторая – набором удобных рычагов управления первой.

Сервер и контроллер

Программа, которая может управляться другой программой, называется *сервером автоматизации*. А программа, выполняющая управление, – *контроллером автоматизации*. Здесь приложения *Word* или *Excel* являются серверами, а программа пользователя – контроллер.

В общем случае нет необходимости обязательно создавать систему взаимодействующих программ. Просто одна и та же программа может иметь и графический оконный интерфейс, и программный. Такой интерфейс строится по определенным технологиям. Современными подобными технологиями являются COM-технология, OLE и автоматизация OLE.

OLE –
Object Linking and Embedding

Object Linking and Embedding переводится как «связывание и внедрение объекта». Пользуясь технологией OLE, можно, разрабатывая собственное приложение, часть задач решать средствами других приложений.

Не вдаваясь в подробности и дебри соотношений и связей между этими технологиями, в данной главе мы будем говорить лишь о взаимодействии двух программ, и называть это OLE-технологией.

Так, например, разрабатывая приложение в *C++ Builder*, требующее выполнения достаточно сложных математических вычислений, можно не разрабатывать и отлаживать собственные математические процедуры, а обратиться за этими вычислениями к стандартному приложению *Excel*.

OLE-технология реализует концепции объектно-ориентированного подхода и повторного использования кода не на уровне наследования реализации классов внутри одного приложения, как при традиционном использовании *C++*, а на уровне разных приложений и операционной системы. Это позволяет создавать *«классы»*, предоставляющие различные *сервисы другим приложениям* и самой операционной системе с помощью довольно широкого спектра средств разработки.

Технология OLE определяется правилами (протоколом) взаимодействия *Windows*-приложений.

18.1. Связь с MS Excel по технологии OLE

Использование технологий объектно-ориентированного программирования при создании приложений изменило восприятие самого приложения разработчиками. Программы стали восприниматься как изделия, состоящие из блоков, работающих согласованно для достижения общего результата. Это резко упростило разработку сложных приложений. При этом описание коллекции однотипных объектов в программе, то есть описание класса, состоит из двух частей – интерфейсная часть и описание реализаций методов. Интерфейсная часть описывает правила использования методов: как называется функция, входные и выходные параметры.

Технология OLE позволяет надстроить интерфейс и к приложению. В этом случае само приложение превращается в объект, доступ к полям и методам которого определяется его интерфейсом.

В рамках этого учебника не будет рассматриваться порядок и технология разработки интерфейса серверов. А вот примеры обращения к приложениям *Excel* и *Word* и использования их ресурсов будут приведены.

Структура приложения *Excel* с точки зрения технологии OLE

Программа *Excel*, с точки зрения OLE, представляет собой сложный составной объект. По-другому его же можно воспринимать как набор взаимосвязанных объектов, некоторые из которых являются элементами других. Для лучшего понимания взаимодействия с *Excel* средствами OLE необходимо вспомнить основные моменты работы в этом приложении. Рассмотрим ключевые объекты *Excel* и основные операции с ними.

Основные объекты программы *Excel* следующие:

- в первую очередь это само *приложение*. Программа *Excel* уже может быть запущена на момент его использования нашим приложением, а возможно, и нет. Следовательно, приложение должно проверять текущее состояние *Excel* и устанавливать с ним связь. Но простое приложение может и просто запустить новый экземпляр *Excel*, не проверяя состояния. При необходимости приложение может дать команду на завершение работы *Excel*. Как правило, это происходит на этапе завершения программы;
- запущенная программа *Excel* может содержать одну или несколько открытых *книг*, то есть файлов;
- каждая книга состоит из нескольких *страниц*;
- страница книги состоит из *ячеек*, которым можно присваивать какие-либо данные или даже формулы и считывать значения из этих ячеек;
- другие дополнительные элементы, например, *диаграммы*.

Основные операции над этими объектами:

- запускать и останавливать работу программы *Excel*;
- открывать и закрывать книги, сохранять их содержимое на диске;
- добавлять в книге новые страницы, удалять их или переименовывать;
- заполнять ячейки значениями, считывать значения из них;
- задавать в ячейки формулы и считывать результаты вычислений;
- управлять форматами и другими параметрами ячеек;
- управлять построением диаграмм и регулировать их параметры.

Попросту говоря, средствами автоматизации OLE можно выполнять в программе *Excel* те же действия, что и с помощью клавиатуры и «мыши».

Помимо объектов, которые представлены видимыми элементами в *Excel*, при работе приложения средствами автоматизации OLE приходится создавать временные указатели на различные дополнительные элементы, как то активная страница или выделенный регион ячеек.

Отметим Необходимо понимать, что непосредственная работа с *Excel* посредством клавиатуры и «мыши», несколько отличается от работы с ней средствами технологии OLE-интерфейса. В последнем случае приходится иметь дело с б*о*льшим количеством объектов. Но при проектировании и реализации подпрограмм работы с *Excel*, конечно, следует руководствоваться навыками работы в программе *Excel*, ведь общая логика выполнения действий такая же.

Управление серверами OLE из программы *C++ Builder*

Помимо общеизвестных базовых и производных типов переменных, в языке *C++* определен и тип *Variant*.

Variant

Переменная типа `Variant` может хранить любой другой тип данных. Это зависит от фактически сохраняемых данных. Удобством этого типа является его универсальность. Ему можно присваивать любые значения.

Но в данной главе интересно то, что переменной такого типа можно также присваивать и значение указателя на OLE-объект, а затем с этим OLE-объектом работать как с объектом класса: переменная типа *Variant* в таком случае будет представителем класса, который состоит из множества *методов* и *свойств*, подобно классу.

Напомним, что в ООП методы класса являются подпрограммами. При этом различают подпрограммы двух типов:

- процедура – подпрограмма, которая выполняет некие действия;
- функция – подпрограмма, которая выполняет действия, а результат возвращает через имя функции.

Также в состав интерфейса классов в ООП входят и СВОЙСТВА. То есть, это как бы поля, но обладающие, куда большими возможностями. Обращение к таким полям выполняется посредством процедур, и при этом со свойствами выполняют два действия:

- *присвоить* значение свойству;
- *считать* значение из свойства.

В общем случае с некоторыми свойствами можно выполнять обе операции, а с некоторыми – только одну.

При создании приложения автоматизации OLE нас интересуют только четыре метода, которые, собственно, и соответствуют четырем указанным операциям:

- `OleFunction` – вызов функции объекта OLE;
- `OleProcedure` – вызов процедуры объекта OLE;
- `OlePropertyGet` – чтение значения свойства объекта OLE;
- `OlePropertySet` – запись значения в свойство объекта OLE.

Все разнообразие наименований методов и свойств различных OLE-объектов указывается в параметрах этих четырех методов переменной типа `Variant`.

Следует отметить также, что методы переменной типа `Variant`, которые потребуются в этой главе для работы с OLE-объектами – это `CreateObject`, `GetActiveObject` и `IsEmpty`:

- метод `CreateObject` предназначен для старта OLE-приложения и установления связи с переменной типа `Variant`;
- метод `CreateObject` – для установления связи переменной типа `Variant` с уже запущенным приложением;
- метод `IsEmpty` можно использовать для проверки: присвоено ли этой переменной значение или еще нет. Такую проверку можно выполнять и функцией `VarIsEmpty`.

В объектно-ориентированной программе объект, то есть экземпляр класса, порождается динамически. Связь с ним программа осуществляет через переменную типа указатель, специально для этого выделенную. В программе же автоматизации OLE после старта объекта автоматизации, например *Excel*, связь с ним осуществляет

через переменную типа *Variant*. Общий порядок работы с объектами OLE, как бы экземплярами класса, такой же, как и с обычными объектами:

- создать объект автоматизации – соединить переменную типа с новым экземпляром приложения *Excel* или ранее запущенным;
- выполнить работу – активизировать методы приложения, используя методы переменной типа *Variant*;
- уничтожить объект – отсоединить переменную типа *Variant* и, возможно, выполнить команду завершения работы приложения *Excel*.

Порядок работы программы с сервером автоматизации OLE во многом схож с порядком работы с классическими объектами (экземплярами класса) в объектно-ориентированной программе. Отличие лишь в том, что этим объектом взаимодействия является целое приложение и иногда с невероятно большими возможностями, как, например, *Excel.*

Примечание

Еще одно отличие, которое следует отметить – это то, что приложение *Excel* во время работы приложения автоматизации может быть невидимым или видимым. Во втором случае пользователь может наблюдать, как выполняются действия в *Excel*. Но есть опасность, что пользователь может вмешаться и нарушить работу программы, даже по неосторожности.

Задача 18.1. Разработать приложение, предназначенное для открытия книг программой *Excel* и чтения данных со страниц этих книг.

Решение

Предлагается следующая стратегия решения задачи.

1. Определяется, какую информацию следует предоставить пользователю.
2. Определяется последовательность действий, которые должен выполнить пользователь для того, чтобы обратиться к *Excel*, выбрать необходимую книгу *Excel* и просмотреть в ней нужные страницы и данные.
3. Разрабатывается интерфейс приложения.
4. Подготавливаются специальные процедуры для обеспечения действий пользователя.

Подготовка решения в соответствии с описанной стратегией.

1) Пользователю нужно предоставить информацию следующего вида:
   а) перечень книг *Excel*, с возможностью выбора книги;
   б) наименование выбранной книги;
   в) перечень страниц выбранной книги *Excel* с возможностью выбрать страницу;
   г) окно, содержащее прочитанные данные с выбранной страницы.
2) Предлагается обеспечить следующую последовательность действий с приложением:
   а) запустить *Excel* с помощью кнопки;
   б) открыть книгу *Excel* – пользователь нажимает кнопку, после чего производит выбор в стандартном диалоге. На экране отображается наименование выбранной книги и список наименований страниц;

в) выбрать страницу из списка и выполнить команду «Прочитать страницу». На экране отражается содержимое страницы.

3) Для решения задачи может быть предложен интерфейс программы, как на рис. 18.1. Интерфейс данного приложения будет содержать следующие элементы, представленные в таблице 18.1.

Таблица 18.1

Основные компоненты приложения чтения документов Excel

№	Тип	Наименование	Назначение
1	TListBox	lbExcelBooks	Список открытых книг
2	TComboBox	cbExcelSheets	Список страниц текущей книги
3	TListBox	lbContent	Отображение содержимого текущей страницы
4	TBitBtn	bbInitExcel	Инициализация *Excel*
5	TBitBtn	bbOpenBook	Открытие книги
6	TCheckBox	cbVisible	Управляет видимостью *Excel*
7	TActionList	ActionList	Набор подпрограмм работы
8	TOpenDialog	dlgOpenExcelBook	Диалог открытия книги

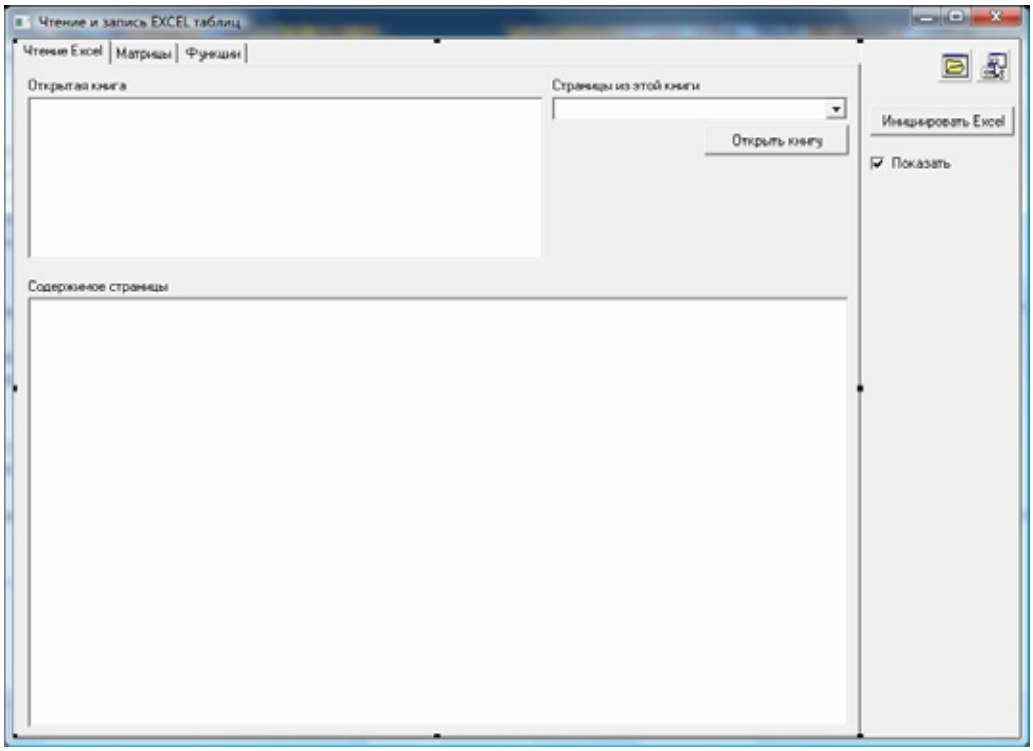


Рис. 18.1. Интерфейс программы чтения данных из документов Excel

4) Для того чтобы пользователь имел возможность выполнить предусмотренные действия и получить требуемую информацию, целесообразно разработать следующие подпрограммы (так как в этом приложении использован компонент `TActionList`, то основные методы обработки реализованы в нем):

- `actCreateExcelExecute(TObject *Sender);` – подпрограмма для запуска *Excel*;

- `cbVisibleClick(TObject *Sender);` – подпрограмма переключения видимости программы *Excel*;
- `actOpenWorksheetsExecute(TObject *Sender);` – подпрограмма для открытия книги;
- `lbExcelBooksClick(TObject *Sender);` – обработчик события выбора книги и страницы;
- `cbExcelSheetsChange(TObject *Sender);` – обработчик события «Выбор страницы *Excel* документа»;
- `FormCloseQuery(TObject *Sender, bool &CanClose);` – обработчик события «Запрос на закрытие главной формы программы».

Для связи программы на различных этапах с одним и тем же экземпляром программы *Excel* и некоторой выбранной страницей в текущей книге используются следующие поля типа `Variant`:

- `FExcel` – хранит указатель на запущенный экземпляр приложения *Excel*;
- `FSheet` – хранит указатель на текущую страницу.

Эти переменные являются полями главной формы приложения. По сути это могут быть и глобальные переменные.

Запуск программы *Excel*

Работа этой подпрограммы довольно прозрачна: если к переменной `FExcel` еще не привязан объект автоматизации OLE, то пытаемся установить связь с активным экземпляром программы *Excel*. Если нет активной сессии *Excel*, то стартуем новый экземпляр. После старта устанавливаем видимость окна *Excel* в соответствии с состоянием флага «Показать», а затем запретить работу этой подпрограммы, чтобы во время работы программы пользователь не имел возможности стартовать другой экземпляр *Excel*. Код программы представлен в листинге 18.1.

Листинг 18.1. Подпрограмма старта программы *Excel*

```
void __fastcall TfoMainForm::actCreateExcelExecute(TObject *Sender)
{
  if (FExcel.IsEmpty())  // если Excel еще не стартовали
   {
try {
    FExcel = Variant::GetActiveObject ("Excel.Application");
 }
catch (...) {
    FExcel = Variant::CreateObject ("Excel.Application");
 }
    FExcel.OlePropertySet ("Visible", cbVisible->Checked);
    actCreateExcel->Enabled = false;
   }
}
```

Обработчик запроса на закрытие формы программы

Завершение программы достаточно сложный процесс. По щелчку «мыши» по кнопке в заголовочной строке формы или пункту главного меню программа должна проанализировать свое состояние, проделать соответствующие действия и принять решение выполнить ли приказ пользователя или проигнорировать. Поэтому после

щелчка пользователя приложение получает от *Windows* несколько событий, в том числе и запрос `CloseQuery`. Эта процедура – суть обработчик этого запроса и имеет два параметра `Sender` и `CanClose`. О первом из них речь уже шла раньше (см. предыдущие главы). Второй параметр `CanClose` – булевского типа. Он является выходным и передается по ссылке. Такой запрос позволяет системе спросить у формы приложения: а можно ли ее закрывать? В данной подпрограмме выполняется проверка: есть ли активная сессия программы той *Excel*, которая привязана к полю `FExcel`. Выдается предупреждение пользователю и в зависимости от ответа выполняется завершение сессии *Excel* и приложения, или происходит возврат к работе. В последнем случае параметру `CanClose` присваивается значение `false`. Это извещает систему, что процесс закрытия формы не надо выполнять. А так как это главная форма, то не завершается и приложение.

СЕССИЯ

Сессией называется период активного состояния приложения *Excel*, при котором программа пользователя находится во взаимодействии с программой *Excel*.

В программе вызывается стандартная функция `MessageDlg`, которая выдает сообщение пользователю и предоставляет ему возможность выбора Да/Нет/Отменить (Ok/No/Cancel) и в качестве результата своей работы возвращает выбранное значение. В отличие от подпрограммы старта, здесь проверка связи с *Excel* выполняется внешней функцией `VarIsEmpty`, а не методом переменной. Код подпрограммы представлен в листинге 18.2.

Листинг 18.2. Обработчик запроса на закрытие формы программы

```
void __fastcall TfoMainForm::FormCloseQuery
                          (TObject *Sender, bool &CanClose)
{
  CanClose = true;  // установка начального значения

  if (!VarIsEmpty (FExcel)) // если есть сессия
   {
            // сообщить об этом и получить ответ
    int answ = MessageDlg
                ("Excel не закрыт! Завершить его работу?",
                 mtConfirmation,
         TMsgDlgButtons() << mbYes << mbNo << mbCancel, 0);
    switch (answ)  // действия в зависимости от выбора
     {case mrYes : FExcel.OleProcedure ("Quit");
      case mrNo : FExcel = NULL; break;
      case mrCancel : CanClose = false;
     }
   }
}
```

Открытие книги

Для демонстрации методов OLE-объектов, в частности при работе с *Excel*, предлагаемый алгоритм реализован далеко не оптимально. Выбор в пользу такого решения обеспечен простотой алгоритма. Порядок ее работы следующий.

- Проверяется, открыта ли программа *Excel*. Если нет, то выдается сообщение и процедура завершает работу.

- Открывается стандартное диалоговое окно для выбора файла.
- Визуализируется программа *Excel* в зависимости от наличия галочки в переключателе `cbVisible`.
- Опрашивается *Excel* и запоминается объект: список книг `workBooks`.
- Через объект-список книг открывается выбранный файл.
- Очищается список имен файлов книг в компоненте `lbExcelBooks`. Опрашивается количество книг, открытых на текущий момент в программе *Excel*.
- В цикле перебираются открытые книги, опрашиваются их имена и количество страниц. Заносятся эти данные в список книг `lbExcelBooks`.
- Для каждой книги опрашиваются имена всех страниц и загружают их в выпадающий список `cbExcelSheets`, предварительно его очистив.

Код программы представлен в листинге 18.3.

Листинг 18.3. Процедура открытия книги

```
void __fastcall
   TfoMainForm::actOpenWorksheetsExecute(TObject *Sender)
{
      // если Excel подключен к программе
  if (!VarIsEmpty(FExcel))
   {      // открыть диалог для выбора файла
    if (dlgOpenExcelSheet->Execute ())
     {
           // сделать Excel видимый, если так указано
      FExcel.OlePropertySet ("Visible", cbVisible->Checked);
         // получить указатель на список книг
      Variant workBooks = FExcel.OlePropertyGet ("WorkBooks");
         // открываем выбранный файл-книгу
      workBooks.OleProcedure
             ("Open", dlgOpenExcelSheet->FileName.c_str());

      lbExcelBooks->Clear ();  // чистим список книг

           // узнаем количество открытых книг
      int couWB = workBooks.OlePropertyGet ("Count");
      for (int indW = 1; indW <= couWB; indW++)
                // цикл по книгам
       {  // выбираем очередную книгу из Excel
        Variant wb = workBooks.OlePropertyGet ("Item", indW);
          // спрашиваем ее имя
        AnsiString name = wb.OlePropertyGet ("Name");
          // указатель на список страниц книги
        Variant sheets = wb.OlePropertyGet ("Sheets");
          // узнаем количество страниц в книге
        int couSh = sheets.OlePropertyGet ("Count");
          // пополняем список книг
        lbExcelBooks->Items->Add ("В книге \"" + name + "\" "
                                  + IntToStr (couSh) + " страниц.");
          // чистим список страниц
        cbExcelSheets->Items->Clear();
        for (int indS = 1; indS <= couSh; indS++)
                  // по всем страницам
         {
          Variant sheet = wb.OlePropertyGet
                                        ("Sheets", indS);
```

```
          AnsiString name = sheet.OlePropertyGet ("Name");
                                  // имя очередной страницы
            // добавляем имена страниц в ComboBox
          cbExcelSheets->Items->Add (name);
         }  // for indS
        cbExcelSheets->ItemIndex = 0;
       }
     }  // if dlgOpenExcelSheet
   }  // if !VarIsEmpty
  else MessageDlg ("Приложение MS Excel не активизировано!",
                   mtError, TMsgDlgButtons() << mbOK, 0);
}
```

Выбор книги

Для просмотра содержимого книг и их страниц средствами данного приложения в компонент `lbExcelBooks` загружается список книг по мере их открытия. Для просмотра же списка страниц в текущей книге и содержимого конкретной страницы к компонентам типа список `lbExcelBooks` и `cbExcelSheets` достаточно присоединить обработчики событий по выбору в них строк. Текст реализации обработчика по выбору книги в списке `lbExcelBooks` приведены в листинге 18.4.

Листинг 18.4. Обработчик события – выбор книги *Excel*

```
void __fastcall TfoMainForm::lbExcelBooksClick (TObject *Sender)
{
  if(lbExcelBooks->Count>0 && lbExcelBooks->ItemIndex>=0)
   {
    Variant workBooks = FExcel.OlePropertyGet ("WorkBooks");
      // выбираем очередную книгу из Excel
    Variant wb = workBooks.OlePropertyGet
                     ("Item", lbExcelBooks->ItemIndex + 1);
      // спрашиваем ее имя
    AnsiString name = wb.OlePropertyGet ("Name");
      // указатель на страницы книги
    Variant sheets = wb.OlePropertyGet ("Sheets");
      // количество страниц
    int couSh = sheets.OlePropertyGet ("Count");

    cbExcelSheets->Items->Clear();
      // проходим по всем страницам
    for (int indS = 1; indS <= couSh; indS++)
     {  // выбираем очередную страницу из книги
      Variant sheet = wb.OlePropertyGet ("Sheets", indS);
        // опрашиваем имя очередной страницы
      AnsiString name = sheet.OlePropertyGet ("Name");
        // заносим это имя в ComboBox
      cbExcelSheets->Items->Add (name);
     }  // for indS
    cbExcelSheets->ItemIndex = 0;
   }
}
```

Логика работы подпрограммы проста: по индексу выбранной строки выбираем книгу в *Excel* с тем же индексом и заполняем выпадающий список `cbExcelSheets` названиями страниц в этой книге.

Выбор страницы

Структура и порядок работы этой процедуры аналогичен предыдущим и даже частично совпадает. Дополнительного пояснения, вероятно, требует один фрагмент из обработчика выбора страниц. Строка –

```
      // определение занятых ячеек
     Variant aux = sheet.OlePropertyGet ("Cells").
              OlePropertyGet ("SpecialCells", xlCellTypeLastCell);
```

– несколько отличается от предыдущих операторов обращения к OLE-объекту. Но только на первый взгляд. Здесь, как и при работе с привычными объектами, происходит обращение к свойству `Cells` объекта `sheet` и к свойству `SpecialCells` этого свойства `Cells`. Как и в сложных классах, свойство может быть ссылкой на объект, являющийся составной частью сложного.

Другой момент, который здесь следует оговорить, это именно обращение к свойству `SpecialCells` с параметром `xlCellTypeLastCell`. Эта специфичная команда позволяет создать ссылку на внутренний объект `SpecialCells`. Значения свойств `Row` и `Column` этого объекта `SpecialCells` соответствуют нижнему правому углу заполнения данных. Напомним, что страница *Excel* имеет огромные размеры – 65536×256 ячеек. Но в практической работе используется только малая часть из этого. При работе в программе *Excel* пользователь может просмотреть страницу и определить, в какой части имеются данные. При эксплуатации же программы как средства автоматизации OLE-приложения *Excel*, границы данных можно определить, как показано в листинге 18.5, если это требуется, конечно.

Завершающий цикл в листинге 18.5 построчно перебирает ячейки активной страницы в пределах области заполнения данных и отображает их в виде текстовых строк в компоненте `lbContent` приложения. Индекс ячейки страницы книги *Excel* указывается привычным в программах образом – номер строки и номер колонки, задаваемые целыми константами и переменными.

Листинг 18.5. Обработчики события выбора страницы

```
void __fastcall TfoMainForm::cbExcelSheetsChange(TObject *Sender)
{  // Выбор страницы Excel документа
  if (lbExcelBooks->Count>0 && lbExcelBooks->ItemIndex>=0)
   {
    if(cbExcelSheets->Items->Count > 0 &&
       cbExcelSheets->ItemIndex >= 0)
     {
      Variant workBooks = FExcel.OlePropertyGet ("WorkBooks");
        // выбираем очередную книгу из Excel
      Variant wb = workBooks.OlePropertyGet
                     ("Item", lbExcelBooks->ItemIndex + 1);
        // указатель на страницы книги
      Variant sheets = wb.OlePropertyGet ("Sheets");
      Variant sheet = wb.OlePropertyGet
                  ("Sheets", cbExcelSheets->ItemIndex + 1);
        // активизировать выбранную страницу
      sheet.OleProcedure ("Activate");
        // определение занятых ячеек
      Variant aux = sheet.OlePropertyGet ("Cells").
                OlePropertyGet ("SpecialCells", xlCellTypeLastCell);
      int rowLast = aux.OlePropertyGet ("Row");
      int colLast = aux.OlePropertyGet ("Column");
```

```
      lbContent->Clear();
      for (int iR = 1; iR <= rowLast; iR++)
       {AnsiString auxStr = "";
        for (int iC = 1; iC <= colLast; iC++)
         {
          auxStr += sheet.OlePropertyGet ("Cells", iR, iC);
          auxStr += " | ";
         }
        lbContent->Items->Add (auxStr);
       }
     }
   }
}
```

Задача 18.2. Разработать приложение табулирования функций с использованием программы *Excel*.

Решение задачи

Решение задачи 18.1 предыдущего раздела предполагает чтение данных из различных книг и страниц *Excel*. Конечно, писать в ячейки страницы *Excel* так же легко. Пример оператора заполнения ячейки может выглядеть так:

```
FSheet.OlePropertySet ("Cells", row, col, val);
```

где индексированному свойству `Cells` объекта `FSheet` присваивается значение переменной `val`. Индекс тоже указывается целочисленными номерами строки и колонки. При этом переменная `val` может быть различного типа – или строковая, или числовая вещественного и целого типа.

Как известно, основное назначение программы *Excel* – это создание электронных таблиц. Естественно, что все эти возможности программы *Excel* доступны и при обращении к ней по технологии автоматизации OLE. Так, например, ячейкам можно присваивать формулы, которые будут вычисляться.

Предлагается следующая стратегия решения задачи.

1. Определяется, какую информацию следует предоставить пользователю и получить от пользователя.
2. Определяется список и последовательность действий, которые должен выполнить пользователь для того, чтобы передать необходимую информацию и получить требуемые результаты.
3. Разрабатывается интерфейс приложения.
4. Подготавливаются специальные процедуры для обеспечения действий пользователя.

Подготовка решения в соответствии с описанной стратегией.

1. Требуется получить информацию (входные данные) от пользователя следующего вида:
   а) границы интервала и шаг табулирования;
   б) формулу для функции, таблицу значений которой требуется получить.

   Для пользователя следует передать таблицу значений функции в виде графика функции, для лучшего восприятия ее поведения.
2. Предлагается обеспечить следующую последовательность действий с приложением:

а) запустить *Excel* с помощью кнопки;
б) задать значения входных данных. Для этого нужно предоставить соответствующие поля;
в) выполнить команду ввода данных;
г) выполнить команду «Произвести табулирование».

3. Может быть предложен интерфейс программы для решения задачи, как на рис. 18.2. Тогда интерфейс данного приложения будет содержать следующие элементы, представленные в таблице 18.2.

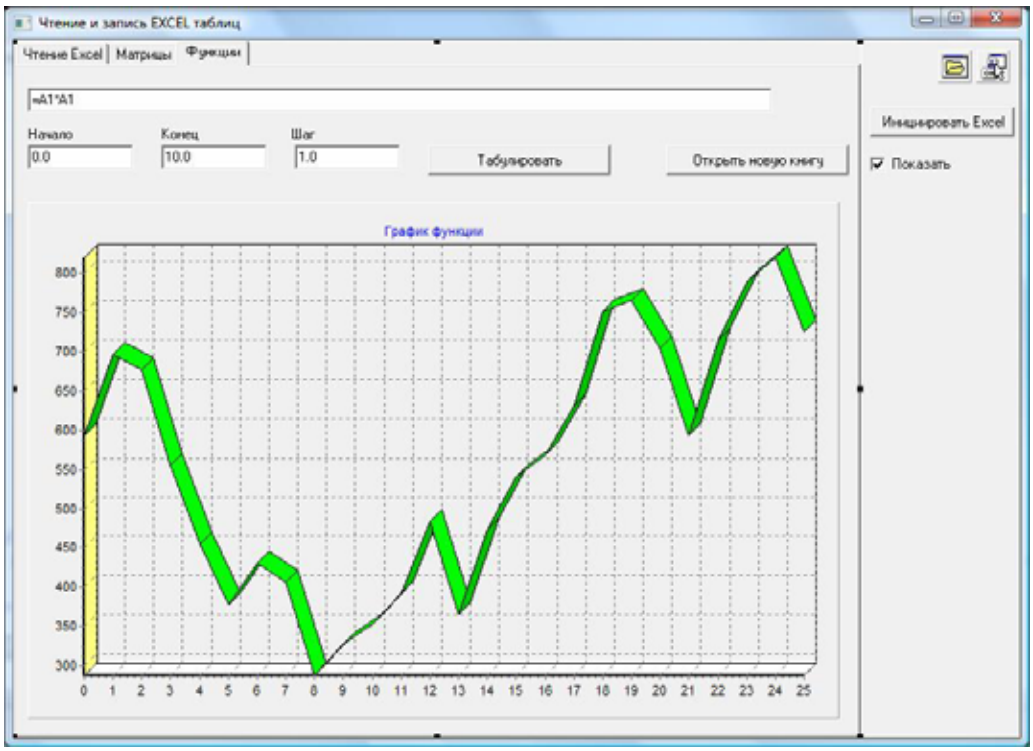


Рис. 18.2. Интерфейс программы табулирования функции

Таблица 18.2

Основные компоненты приложения табулирования функции

№	Тип	Наименование	Назначение
1	TEdit	edFormula	Строка для ввода формулы
2	TEdit	edStart	Строка для ввода начала интервала табулирования
3	TEdit	edFinish	Строка для ввода конца интервала табулирования
4	TEdit	edStep	Строка для ввода шага табулирования
5	TBitBtn	bbInitExcel	Инициализация *Excel*
6	TBitBtn	bbInitBookFunc	Открытие книги
7	TBitBtn	bbTabulate	Запуск процесса табулирования
8	TChart	chGraphic	Отображение графика функции
9	TCheckBox	cbVisible	Управление видимостью *Excel*
10	TActionList	ActionList	Набор подпрограмм работы

4. Целесообразно разработать следующие подпрограммы для того, чтобы пользователь имел возможность выполнить предусмотренные действия, ввести и получить требуемую информацию:

- `actCreateExcelExecute(TObject *Sender);` – подпрограмма для запуска *Excel*;
- `FormCloseQuery(TObject *Sender, bool &CanClose);` – обработчик события «На закрытие формы»;
- `actOpenFuncBookExecute(TObject *Sender);` – создание нового документа в программе *Excel*;
- `actTabulateExecute(TObject *Sender);` – табулирование произвольной функции;

Примечание

В работе этого приложения используются те же процедуры инициализации работы программы *Excel* и обработки запроса на завершения работы, что и в предыдущей задаче. Поэтому описание их не приводится.

Создание новой книги

Алгоритм создания новой книги можно описать следующими шагами.

1. Выполняется проверка, выполнено ли уже подключение к *Excel*. Если подключение есть, то необходимо:
   1.1. Прочитать свойство «Рабочая книга».
   1.2. Выполнить процедуру «Добавить книгу».
   1.3. Установить свойство «Видимый» в соответствии с отметкой «Показать».
   1.4. Прочитать в книге свойство «количество страниц».
   1.5. Активизировать первую станицу.
2. Если подключения нет, то необходимо:
   2.1. Сообщить об этом.
   2.2. Завершить работу функции.

Описание функции приведено в листинге 18.6.

Листинг 18.6. Подпрограмма создания нового документа в программе *Excel*

```
void __fastcall TfoMainForm::actOpenFuncBookExecute
                                              (TObject *Sender)
{// открытие файла Excel для работы с функциями и матрицами
     // есть Excel, подключенный к программе,
  if (!VarIsEmpty(FExcel))
   {  // добавление нового документа
    FExcel.OlePropertyGet ("WorkBooks")
          .OleProcedure ("Add");
    FExcel.OlePropertySet ("Visible", cbVisible->Checked);
      // установка на первую страницу
      // только что загруженной книги
    int couWB = FExcel.OlePropertyGet ("WorkBooks")
                      .OlePropertyGet ("Count");
    FSheet = FExcel.OlePropertyGet ("WorkBooks")
                   .OlePropertyGet ("Item", couWB)
                   .OlePropertyGet ("Sheets", 1);
   }
  else MessageDlg ("Приложение MS Excel не активизировано!",
                   mtError, TMsgDlgButtons() << mbOK, 0);
}
```

Табулирование функции

Процедура табулирования функции с использование программы *Excel* почти такая же, как и стандартная процедура, приведенная в главе 3. Отличия лишь в том, что данная процедура загружает выражение функции в ячейку страницы *Excel*, и задает абсциссу и считывает ординату из соответствующих ячеек *Excel*.

Примечание

Функцию необходимо задавать по правилам *Excel*, так как текст функции передается в ячейку страницы *Excel* без искажения. Так, например, для построения графика функции $F(x) = x^2$ можно задать строку `=A1*A1` или `=A1^2`.

В реализации данного алгоритма используются локальные переменные: `start`, `finish`, `step` – для описания границ и шага табулирования; `curr` – текущее значение аргумента функции. Тип `double` для переменных принят из-за того, что в программе используется функция проверки на ошибки `TryStrToFloat`, выполняющая преобразование именно в этот тип.

Алгоритм табулирования можно описать следующими шагами.

1. Читаются входные значения из окон строковых редакторов и выполняется проверка на правильность написания чисел. Если нет ошибок при чтении данных, то
   - очищается область отображения графика;
   - заносится формула в ячейку страницы *Excel*;
   - инициализируется начальное значение аргумента;
   - выполняется цикл до тех пор, пока значение аргумента в области. В теле этого цикла:
     - считывается значение функции;
     - это значение добавляется к графику;
     - вычисляется новое значение аргумента.
2. Выдается сообщение пользователю, если есть ошибки при написании. Описание функции приведено в листинге 18.7.

Листинг 18.7. Процедура табулирования функции

```
void __fastcall TfoMainForm::actTabulateExecute (TObject *Sender)
{  // процесс табулирования произвольной функции
   // посредством Excel
   // тип double из-за того,
   // что TryStrToFloat преобразует в него
  double start, finish, step;
    // есть ли ошибки в написании чисел
  if (TryStrToFloat (edStart->Text, start) &&
      TryStrToFloat (edFinish->Text, finish) &&
      TryStrToFloat (edStep->Text, step))
   {serGraphicOfFunction->Clear();  // чистим график

    FSheet.OlePropertySet ("Cells", 2, 1,
               edFormula->Text.c_str()); // заносим формулу

    double aux;
    double curr = start;
    while (curr <= finish)
     {FSheet.OlePropertySet ("Cells", 1, 1, curr);
                                // заносим текущее значение
```

```
      aux = FSheet.OlePropertyGet ("Cells", 2, 1);
                                // считываем значение функции
      serGraphicOfFunction->AddXY (curr, aux);
                                        // значение в график

      curr += step;  // и т.д. и т.п.
     }
   }
  else MessageDlg ("Одно из чисел не правильно написано!",
                   mtError, TMsgDlgButtons() << mbOK, 0);
}
```

Такое решение задачи, то есть применение автоматизации OLE-программы *Excel*, позволяет строить простое приложение, в котором используются произвольные формулы.

Задача 18.3. Разработать приложение по выполнению основных операций программой *Excel* с матрицами. Реализовать операции: вычисление определителя, нахождением суммы элементов матрицы, вычисление обратной матрицы и их перемножение.

Решение

Как известно, возможности программы *Excel* не ограничиваются вычислениями простейших формул. Она позволяет решать довольно широкий круг стандартных задач. Например, выполнять операции над матрицами. И тогда, естественно, имея шаблоны обращения к программе *Excel*, можно быстро разработать приложение, которое будет решать такие задачи посредством *Excel*. В последующем, конечно, можно разрабатывать и тщательно тестировать свои процедуры выполнения таких операций.

Прежде чем приводить примеры выполнения операции с матрицами, отметим такой существенный аспект. В предыдущем примере формула задавалась как текстовое значение ячейки страницы *Excel*. Но не всегда такая простая конструкция позволяет задавать формулы. Другой способ задавать формулы в ячейку страницы программы *Excel* – это воспользоваться свойством `FormulaR1C1` ячейки. В качестве значения этому свойству передается такая же строка формулы. Отличительной особенностью строки-формулы является то, что адреса ячеек задаются не в привычном для *Excel* формате, например `"A1"`, а в виде `"R1C1"`. То есть адрес строки задается в виде: номер строки – номер столбца, предваряемые литерами `R` (row) и `C` (column).

Предлагается следующая стратегия решения задачи.

1. Определяется, какую информацию следует предоставить пользователю и получить от него.
2. Определяется список и последовательность действий, которые должен выполнить пользователь для того, чтобы передать необходимую информацию и получить требуемые результаты.
3. Разрабатывается интерфейс приложения.
4. Подготавливаются специальные процедуры для обеспечения действий пользователя.

Подготовка решения в соответствии с описанной стратегией.

1. Требуется от пользователя получить информацию (входные данные) следующего вида:

а) размеры матрицы;
б) значения элементов матрицы.

Пользователю следует передать результаты вычислений в виде числовых значений.

2. Предлагается обеспечить следующую последовательность действий с приложением:
   - запустить *Excel* с помощью кнопки;
   - задать значения входных данных. Для этого нужно предоставить соответствующие поля;
   - выполнить команду ввода данных;
   - выполнить команду «Вычислить/произвести операцию»;
   - для решения задачи может быть предложен интерфейс программы, как на рис. 18.3. Интерфейс данного приложения будет содержать следующие элементы, представленные в табл. 18.3.

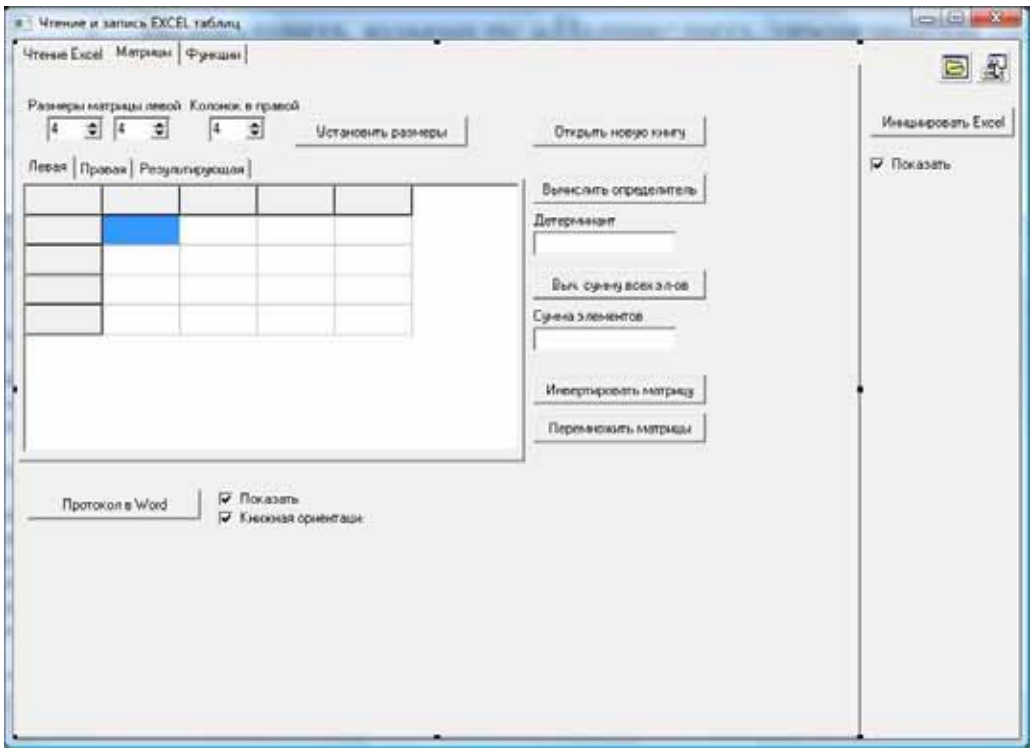


Рис. 18.3. Интерфейс программы работы с матрицами

3. Для того чтобы пользователь имел возможность выполнить предусмотренные действия, ввести и получить требуемую информацию, целесообразно разработать следующие подпрограммы:

- `actCreateExcelExecute(TObject *Sender);` – подпрограмма для запуска *Excel*;
- `FormCloseQuery(TObject *Sender, bool &CanClose);` – обработчик события «На закрытие формы»;
- `actOpenFuncBookExecute(TObject *Sender);` – создание нового документа в программе *Excel*;
- `actSetSizeExecute(TObject *Sender);` – ввод размеров матрицы и значений элементов матрицы в отведенные для них области;

- `actDeterminateExecute(TObject *Sender);` – процедура вычисления определителя матрицы;
- `actSummOfElemExecute(TObject *Sender);` – процедура суммирования элементов матрицы;
- `actMatrixInverse(TObject *Sender);` – процедура вычисления обратной матрицы;
- `actMatrixMultiple(TObject *Sender);` – процедура вычисления произведения матриц.

Таблица 18.3

Основные компоненты приложения работы с матрицами

№	Тип	Наименование	Назначение
1	TCSpinEdit	cseRowCount	Количество строк в матрице
2	TCSpinEdit	cseColCount	Количество столбцов в матрице
3	TCSpinEdit	cseColCountR	Количество столбцов в правой матрице
4	TStringGrid	sgLeftMatr	Таблица для задания и отображения значений исходной матрицы, левого сомножителя
5	TStringGrid	sgRightMatr	Таблица для задания и отображения значений исходной матрицы, правого сомножителя
6	TStringGrid	sgProductMatr	Таблица для отображения значений матрицы результатов
7	TBitBtn	bbInitExcel	Инициализация *Excel*
8	TBitBtn	bbInitBookMatrix	Открытие книги
9	TBitBtn	bbSetSize	Установка размеров и заполнение матриц случайными числами
10	TBitBtn	bbDeterminate	Определение детерминанта
11	TBitBtn	bbSumm	Определение суммы элементов
12	TBitBtn	bbInverse	Вычисление обратной матрицы
13	TBitBtn	bbProd	Вычисление произведения матриц
14	TEdit	edDeterminate	Окно отображения результата вычисления детерминанта
15	TEdit	edSummOfElem	Окно отображения результата вычисления суммы элементов
16	TCheckBox	cbVisible	Управляет видимостью *Excel*
17	TActionList	ActionList	Набор подпрограмм работы

Кнопки `bbInitExcel` и `bbInitBookMatrix` аналогичны по своему назначению подобным кнопкам из предыдущих примеров: открытие сессии связи с программой *Excel* и открытие нового документа.

Ввод размеров матрицы и значений элементов матрицы

Процедура задания размеров компонента `StringGrid` и заполнение его случайными данными довольно проста:

- из окон редакторов `TCSpinEdit` читаются значения, определяющие размеры матрицы, и на основании этих размеров определяется число строк и число столбцов в сетках `StringGrid` для левой, правой и результирующей матриц;
- выполняются двойные циклы (по строкам и столбцам сетки), в теле которых генерируются случайные числа; и из них формируются значения ячеек сетки. Для результирующей сетки ячейки очищаются.

Текст реализации процедуры приведен в листинге 18.8.

Листинг 18.8. Процедура заполнения таблицы значениями матрицы

```
void __fastcall TfoMainForm::actSetSizeExecute (TObject *Sender)
{   // установить размеры матрицы
  sgLeftMatr->RowCount = cseRowCount->Value + 1;
  sgLeftMatr->ColCount = cseColCount->Value + 1;

  for (int iRow = 1; iRow < sgLeftMatr->RowCount; iRow++)
    for (int iCol = 1; iCol < sgLeftMatr->ColCount; iCol++)
      sgLeftMatr->Cells[iCol][iRow] =
                            RandomRange (-100, 100) * 0.1;

  sgRightMatr->RowCount = cseColCount->Value + 1;
  sgRightMatr->ColCount = cseColCountR->Value + 1;

  for (int iRow = 1; iRow < sgRightMatr->RowCount; iRow++)
    for (int iCol = 1; iCol < sgRightMatr->ColCount; iCol++)
      sgRightMatr->Cells[iCol][iRow] =
                             RandomRange (-100, 100) * 0.1;

  sgProductMatr->RowCount = cseRowCount->Value + 1;
  sgProductMatr->ColCount = cseColCountR->Value + 1;

  for (int iRow = 1; iRow < sgProductMatr->RowCount; iRow++)
    for (int iCol = 1; iCol <
                           sgProductMatr->ColCount; iCol++)
      sgProductMatr->Cells[iCol][iRow] = "";
}
```

Примечание

Результаты работы следующих двух подпрограмм (значения определителя и суммы) заносятся в текстовую строку редактирования. Поэтому из ячейки *Excel* результат переносится как текст. Если необходимо получить результат в числовом формате, то нужно считывать его непосредственно в вещественную переменную или потом правильно доставать и преобразовать из компонента `edSummOfElem`.

Вычисление определителя матрицы

Поскольку задача решается из предположения, что используется чистая страница *Excel*, то значения элементов матрицы заносятся в страницу, начиная с ячейки `A1`. Тогда при формировании строки формулы основная трудность – это написать завершающую часть, в частности `RnCm`, где `n` – номер последней строки, а `m` – номер последнего столбца.

Так как для формирования строки формулы (переменная `aux`) в процедуре используется вспомогательная переменная типа `AnsiString`, то удалось обойтись без

явного вызова функций преобразований из численного формата в текстовый, а использовать только операторы, определенные для этого класса. В качестве параметра задается не вся переменная, а ее содержимое в виде стандартного массива типа `char`.

Логика работы этой процедуры заключается в следующем:

- переписать значения из таблицы `StringGrid` в ячейки страницы *Excel*;
- сформировать строку команды вычисления определителя;
- записать формулу в свободную ячейку страницы *Excel*;
- считать результаты вычисления – значение определителя данной матрицы.

Свойство `FormulaR1C1` предназначено для присвоения ячейке страницы *Excel* текста формулы. При этом адреса ячеек в формуле должны быть записаны в формате `R1C1`. Здесь текст формулы формируется в локальной переменной `aux` типа строка. В параметрах функции МОПРЕД формируется интервал размещения коэффициентов матрицы в ячейках страницы. Например, для матрицы 4×4 текст формулы будет сформирован в виде `"МОПРЕД(R1C1:R4C4)"`. Код процедуры вычисления детерминанта приведен в листинге 18.9.

Листинг 18.9. Процедура вычисления определителя матрицы

```
void __fastcall TfoMainForm::actDeterminateExecute (TObject *Sender)
{  // процедура вычисления определителя
  for (int iRow = 1; iRow < sgMatrix->RowCount; iRow++)
    for (int iCol = 1; iCol < sgMatrix->ColCount; iCol++)
     {  // заносим значения из StringGrid в Excel
      FSheet.OlePropertySet ("Cells", iRow, iCol,
                        sgMatrix->Cells[iCol][iRow].c_str());
     }
    // формирование строки с формулой
  AnsiString aux = "=МОПРЕД(R1C1:R";
  aux += sgMatrix->RowCount - 1; aux += "C";
  aux += sgMatrix->ColCount - 1; aux += ")";
    // выполнение команды
  FExcel.OlePropertyGet ("Cells", sgMatrix->RowCount + 1, 1)
                      .OlePropertySet ("FormulaR1C1", aux.c_str());
     // чтение результата
  edDeterminate->Text = FSheet.OlePropertyGet ("Cells",
                                 sgMatrix->RowCount + 1, 1);
}
```

Вычисление суммы элементов матрицы

Текст этой подпрограммы почти полностью совпадает с подпрограммой вычисления определителя. Отличие, естественно, в имени вызываемой функции. Также выбрана другая ячейка страницы для занесения формулы и считывания результата. Помимо этого результат сохраняется в другой компонент приложения – в `edSummOfElem`. Код процедуры вычисления суммы элементов приведен в листинге 18.10.

Листинг 18.10. Процедура вычисления суммы элементов матрицы

```
void __fastcall TfoMainForm::actSummOfElemExecute (TObject *Sender)
{  // процедура суммирования элементов матрицы
  for (int iRow = 1; iRow < sgMatrix->RowCount; iRow++)
    for (int iCol = 1; iCol < sgMatrix->ColCount; iCol++)
     {  // заносим значения из StringGrid в Excel
```

```
      FSheet.OlePropertySet ("Cells", iRow, iCol,
                        sgMatrix->Cells[iCol][iRow].c_str());
     }
    // формирование строки с формулой
  AnsiString aux = "=СУММ(R1C1:R";
  aux += sgMatrix->RowCount - 1; aux += "C";
  aux += sgMatrix->ColCount - 1; aux += ")";
    // выполнение команды
  FExcel.OlePropertyGet ("Cells", sgMatrix->RowCount + 1, 3)
              .OlePropertySet ("FormulaR1C1", aux.c_str());
     // чтение результата
  edSummOfElem->Text = FSheet.OlePropertyGet
                        ("Cells", sgMatrix->RowCount + 1, 3);
}
```

Примечание

Все процедуры, приведенные в этой главе, сильно упрощены. Точнее приведены только необходимые действия. Вполне понятно, что здесь нет, например, проверки на допустимость операции – некоторые операции могут выполняться только над квадратными матрицами, нет проверки правильности результата. Помимо этого, конечно, эти подпрограммы должны быть дополнены проверками: установлена ли связь с программой *Excel*, со страницей *Excel* и тому подобное.

Вычисление обратной матрицы

Задача решается аналогично предыдущим. Последовательность выполнения действий следующая:

- заносятся значения из ячеек `StringGrid` в ячейки страницы *Excel*;
- формируется текст формулы и заносится в ячейку *Excel*;
- вычисляется формула с заполнением результатов по соответствующему интервалу ячеек страницы *Excel*;
- результаты переписываются в сетку `StringGrid`, предназначенную для этого.

Код процедуры вычисления суммы элементов приведен в листинге 18.11.

Листинг 18.11. Процедура вычисления обратной матрицы

```
void __fastcall TfoMainForm::actMatrixInverseExecute
                                              (TObject *Sender)
{ /* Вычисление инвертированной матрицы */
    // вычисление адресов региона ниже будущей матрицы
  AnsiString rgnLT = "=АДРЕС(";
  rgnLT += sgLeftMatr->RowCount + 1; rgnLT += ";1";
  FSheet.OlePropertySet ("Cells", 1, 1, rgnLT.c_str());
                                          // заносим формулу
  rgnLT = FSheet.OlePropertyGet ("Cells", 1, 1);
                          // читаем адрес "$A$6", например

  AnsiString rgnRB = "=АДРЕС(";
  rgnRB += 2 * sgLeftMatr->RowCount - 1; rgnRB += ";";
  rgnRB += sgLeftMatr->ColCount - 1; rgnRB += ")";
  FSheet.OlePropertySet ("Cells", 1, 1, rgnRB.c_str());
                                          // заносим формулу
  rgnRB = FSheet.OlePropertyGet ("Cells", 1, 1);
                         // читаем адрес "$G$15", например
```

```
    // процесс вычисления обратной матрицы
  for (int iRow = 1; iRow < sgLeftMatr->RowCount; iRow++)
    for (int iCol = 1; iCol < sgLeftMatr->ColCount; iCol++)
     {  // заносим значения из StringGrid в Excel
      FSheet.OlePropertySet ("Cells", iRow, iCol,
                     sgLeftMatr->Cells[iCol][iRow].c_str());
     }

    // формирование строки с формулой
  AnsiString aux = "=МОБР(R1C1:R";
  aux += sgLeftMatr->RowCount - 1; aux += "C";
  aux += sgLeftMatr->ColCount - 1; aux += ")";
    // выполнение команды
  FExcel.OlePropertyGet ("Cells",
                              sgLeftMatr->RowCount + 1, 1).
               OlePropertySet ("FormulaR1C1", aux.c_str());

  FSheet.OlePropertyGet ("Range", rgnLT.c_str(),
                       rgnRB.c_str()).OleFunction ("Select");
  FExcel.OlePropertyGet ("Selection").OlePropertySet
                             ("FormulaArray", aux.c_str());

    // копирование результатов
  for (int iRow = 1; iRow < sgProductMatr->RowCount; iRow++)
    for (int iCol = 1; iCol <
                           sgProductMatr->ColCount; iCol++)
     {  // заносим значения из Excel
        // в StringGrid матрица результатов
      sgProductMatr->Cells[iCol][iRow] =
                  FSheet.OlePropertyGet ("Cells",
                         iRow + sgLeftMatr->RowCount, iCol);
     }
}
```

Как видно из листинга, общий ход решения стандартный, но есть две особенности. Первая особенность заключается в том, что решение необходимо «распространить» на область. Для этого используется свойство `FormulaArray` выделенной области `Selection`. Для выделения достаточно обратиться к функции `Select` свойства `Range`. Параметрами этой функции являются адреса левой верхней и правой нижней ячеек области, требующей выделения.

Вторая особенность заключается в том, что адреса ячеек в функции `Select` задаются в стандартном виде, принятом в электронных таблицах *Excel*: например, `"A1"` для самой левой верхней ячейки таблицы, поэтому номер столбца необходимо выразить в буквенном виде. Есть два пути – или разработать свою функцию для перевода чисел в соответствующие буквы или буквосочетания, или воспользоваться функциями того же *Excel*. В данной процедуре реализован второй вариант. Так как страница *Excel* перед выполнением операции обращения матрицы считается полностью свободной, то ячейка `A1` используется для пересчета необходимых адресов из формата `R1C1` в формат `A1`. В начале процедуры записываются формулы с функцией `АДРЕС`, например, `"АДРЕС(6, 1)"` и считывается результат `"A6"` в локальную переменную `rgnLT`. Аналогично вычисляется адрес правой нижней ячейки и сохраняется в переменной `rgnRB`. Позже эти переменные используются при указании области требующей выделения.

Вычисление произведения матриц

Решение этой задачи вполне аналогично решению предыдущей. Отличие лишь в том, что на страницу *Excel* приходится копировать две исходные матрицы и пользоваться функцией МУМНОЖ с указанием двух регионов расположения коэффициентов матриц. Код процедуры вычисления суммы элементов приведен в листинге 18.12.

Листинг 18.12. Процедура вычисления произведения матриц

```
void __fastcall TfoMainForm::actMatrixMultipleExecute(TObject *Sender)
{ /* Умножение матриц */
    // вычисление адресов региона НИЖЕ будущей матрицы
  AnsiString rgnLT = "=АДРЕС(";  // левая верхняя ячейка
  rgnLT += sgLeftMatr->RowCount + 1; rgnLT += ";1";
  FSheet.OlePropertySet ("Cells", 1, 1, rgnLT.c_str());
                                        // заносим формулу
  rgnLT = FSheet.OlePropertyGet ("Cells", 1, 1);
                          // читаем адрес "$A$6", например

  AnsiString rgnRB = "=АДРЕС(";  // правая нижняя ячейка
  rgnRB += 2 * sgLeftMatr->RowCount - 1; rgnRB += ";";
  rgnRB += sgLeftMatr->ColCount - 1; rgnRB += ")";
  FSheet.OlePropertySet ("Cells", 1, 1, rgnRB.c_str());
                                       // заносим формулу
  rgnRB = FSheet.OlePropertyGet ("Cells", 1, 1);
                         // читаем адрес "$G$15", например

    // процесс вычисления произведения матриц
  for (int iRow = 1; iRow < sgLeftMatr->RowCount; iRow++)
    for (int iCol = 1; iCol < sgLeftMatr->ColCount; iCol++)
     {  // заносим значения из StringGrid
        // в Excel левая матрица
      FSheet.OlePropertySet ("Cells", iRow, iCol,
                             sgLeftMatr->Cells[iCol][iRow].c_str());
     }

  for (int iRow = 1; iRow < sgRightMatr->RowCount; iRow++)
    for (int iCol = 1; iCol < sgRightMatr->ColCount; iCol++)
     {  // заносим значения из StringGrid
        // в Excel правая матрица
      FSheet.OlePropertySet ("Cells", iRow,
                    iCol + sgRightMatr->ColCount,
                    sgRightMatr->Cells[iCol][iRow].c_str());
     }

    // формирование строки с формулой
  AnsiString aux = "=МУМНОЖ(R1C1:R";
  aux += sgLeftMatr->RowCount - 1;
  aux += "C"; aux += sgLeftMatr->ColCount - 1;
  aux += ";R1C";
  aux += sgLeftMatr->ColCount + 1; aux += ":R";
  aux += sgRightMatr->RowCount - 1; aux += "C";
  aux += sgLeftMatr->ColCount +
                    sgRightMatr->ColCount - 1; aux += ")";

    // выполнение команды
  FExcel.OlePropertyGet ("Cells",
              sgLeftMatr->RowCount + 1, 1).
```

```
                OlePropertySet ("FormulaR1C1", aux.c_str());

  FSheet.OlePropertyGet ("Range", rgnLT.c_str(),
                      rgnRB.c_str()).OleFunction ("Select");
  FExcel.OlePropertyGet ("Selection").
              OlePropertySet ("FormulaArray", aux.c_str());

    // копирование результатов
  for (int iRow = 1; iRow < sgProductMatr->RowCount; iRow++)
    for (int iCol = 1; iCol < sgProductMatr->ColCount; iCol++)
     {  // заносим значения из Excel в StringGrid
        // матрица результатов
      sgProductMatr->Cells[iCol][iRow] =
                              FSheet.OlePropertyGet ("Cells",
                         iRow + sgLeftMatr->RowCount, iCol);
     }
}
```

18.2. Связь с MS Word for Windows по технологии OLE

Эксплуатация приложений, выполняющих обработку данных и проводящих анализ, как правило, сопровождается составлением отчетов по результатам обработки. Обычно такие отчеты содержат таблицы и графики. И если текстовая часть отчета создается составителем, то таблицы и отдельные фрагменты текста могут быть стандартизованы и их составление можно поручить, например, тому же приложению, которое проводит обработку. Так как сегодня фактически стандартным средством составления текстовых документов является текстовый редактор *Word for Windows*, то здесь будут показаны основные приемы формирования фрагментов текста средствами OLE.

Для примера работы программой *Word* средствами OLE будет решена следующая задача.

Задача 18.4. Разработать процедуру формирования отчета в текстовом редакторе *Word* по результатам обработки матриц.

Решение задачи

Логика и структура процедуры формирования отчета о работе должна быть проста – переносить различные данные из приложения в виде текстов в документ в формате *Word*. Несмотря на то, что текст данной процедуры достаточно длинен, тем не менее, его структура проста и понятна по комментариям. Последовательность работы этой подпрограммы следующая:

- открывается новая сессия программы *Word*, если она еще не запущена;
- создается новый документ. Если указано в интерфейсе, то устанавливается ориентация листа как альбомная;
- вставляется произвольный текст и, как пример, дата и время создания документа;
- вставляется текст, соответствующий номеру и заголовку таблицы, с необходимыми параметрами выравнивания;
- определяется требуемое количество строк и колонок таблицы;
- создается таблица требуемого размера;
- прописываются названия колонок и строк;

- переписываются значения из интерфейса программы (компонент `StringGrid`) в созданную таблицу;
- выделяется таблица и задается требование отображать границы таблицы и ячеек и параметры толщины внешней линии и линий, разделяющих ячейки;
- добавляется текст из строк редактирования и размещается после таблицы;
- задается вопрос пользователю и, если требуется, производится завершение сессии программы *Word*.

Код, содержащий описание процедуры, приведен в листинге 18.13.

Листинг 18.13. Процедура формирования документа *Word*

```
void __fastcall TfoMainForm::actResultToWordExecute (TObject *Sender)
{ // сохранение результатов в документе Word for Windows
    // открытие программы Word
  if (FWord.IsEmpty())  // если Word еще не стартовали
   {
    FWord = Variant::CreateObject ("Word.Application");
    FWord.OlePropertySet ("Visible",
                          cbVisibleWord->Checked);
   }
    // создание нового документа
  FWord.OlePropertyGet ("Documents").OleProcedure ("Add");

    // установка на только что созданный документ
  Variant newDoc = FWord.OlePropertyGet ("ActiveDocument");

    // установка ориентации страниц
  if (chOrientation->Checked)
    newDoc.OlePropertyGet ("PageSetup")
                         .OlePropertySet ("Orientation", 0);
  else
    newDoc.OlePropertyGet ("PageSetup")
                         .OlePropertySet ("Orientation", 1);
    // занесение текста
  FWord.OlePropertyGet ("Selection")
  .OleProcedure ("TypeText", "Данные текст был создан ...");
    // перевод строки -- параграф
  FWord.OlePropertyGet ("Selection")
             .OleProcedure ("TypeText", WideString('\012'));
    // вывод даты ...
  AnsiString datetime = "в день " + DateToStr(Date ())
                        + " в " + TimeToStr(Time ()) + ";";
  FWord.OlePropertyGet ("Selection")
              .OleProcedure ("TypeText", datetime.c_str());
    // перевод строки -- параграф
  FWord.OlePropertyGet ("Selection")
             .OleProcedure ("TypeText", WideString('\012'));

    // создание таблицы
  Variant parL = newDoc.OlePropertyGet
                     ("Paragraphs").OlePropertyGet ("Last");
  parL.OlePropertyGet ("Format")
                           .OlePropertySet ("Alignment", 2);

  FWord.OlePropertyGet ("Selection")
                 .OleProcedure ("TypeText", "Таблица\012");
```

```
parL.OlePropertyGet ("Format")
                          .OlePropertySet ("Alignment", 1);
FWord.OlePropertyGet ("Selection")
       .OleProcedure ("TypeText", "Исходная матрица\012");
  // настройка региона на последнюю позицию
Variant range = newDoc.OlePropertyGet ("Content");
range.OlePropertySet ("Start", range.OlePropertyGet ("End"));

int nRow = sgMatrix->RowCount; // сколько строк
int nCol = sgMatrix->ColCount; // сколько колонок
  // добавляем таблицу
Variant wordTable = newDoc.OlePropertyGet ("Tables")
                          .OleFunction ("Add", range, nRow, nCol);
  // подписываем колонки и строки
for (int iCol = 1; iCol < nCol; iCol++)
 {AnsiString aux = "K"; aux += iCol;
  wordTable.OleFunction ("Cell", 1, iCol + 1)
           .OlePropertyGet ("Range")
           .OlePropertySet ("Text", aux.c_str());
 }
for (int iRow = 1; iRow < nRow; iRow++)
 {AnsiString aux = "C"; aux += iRow;
  wordTable.OleFunction ("Cell", iRow + 1, 1)
           .OlePropertyGet ("Range")
           .OlePropertySet ("Text", aux.c_str());
 }
  // заносим данные
for (int iRow = 1; iRow < nRow; iRow++)
  for (int iCol = 1; iCol < nCol; iCol++)
    wordTable.OleFunction ("Cell", iRow + 1, iCol + 1)
           .OlePropertyGet ("Range")
           .OlePropertySet ("Text",
                    sgMatrix->Cells [iCol][iRow].c_str());
  // очерчиваем границы
wordTable.OleProcedure ("Select"); // выделить таблицу
wordTable.OlePropertyGet ("Borders")
                          .OlePropertySet ("Enable", true);
wordTable.OlePropertyGet ("Borders")
                   .OlePropertySet ("OutsideLineWidth", 8);
wordTable.OlePropertyGet ("Borders")
                    .OlePropertySet ("InsideLineWidth", 4);

parL.OlePropertyGet ("Range").OleProcedure ("Select");
  // пустая строка
FWord.OlePropertyGet ("Selection")
           .OleProcedure ("TypeText", WideString('\012'));

  // занесение текста -- значение определителя
AnsiString aux = "Определитель = "
                            + edDeterminate->Text + "\012";
parL.OlePropertyGet ("Format")
                          .OlePropertySet ("Alignment", 0);
FWord.OlePropertyGet ("Selection")
                   .OleProcedure ("TypeText", aux.c_str());

  // занесение текста -- сумма элементов
aux = "Сумма элементов = " + edSummOfElem->Text + "\012";
```

```
  FWord.OlePropertyGet ("Selection")
                    .OleProcedure ("TypeText", aux.c_str());

   // закрытие документа Word
  if (!FWord.IsEmpty())
   {
    int answ = MessageDlg ("Завершить его работу Word?",
     mtConfirmation, TMsgDlgButtons() << mbYes << mbNo, 0);
    switch (answ)
     {case mrYes : FWord.OleProcedure ("Quit");
     }
   }
}
```

Примечание

Конечно, данные могут переноситься не только из переменных и компонентов приложения, но и из ячеек таблиц *Excel* напрямую. В данном примере все операции по работе с *Word for Windows* помещены в единую процедуру.

Для реализации более сложного приложения, необходимо тщательно планировать какие действия должны выполняться в одной процедуре, а какие разделены по разным процедурам.



ВЫПОЛНИТЬ САМОСТОЯТЕЛЬНО

1. Исправить приложение к задаче 18.3 так, чтобы оно работало только для вещественных данных, записанных через точку.

Рекомендации: учесть, что приложение *Excel* отображает вещественные числа или через точку, или через запятую в зависимости от настроек *Windows*.

2. В процедуре перемножения матриц через *Excel* в таблице матрицы правильно размещаются только для квадратных матриц. Исправить процедуру, правильно расположив прямоугольные матрицы в *Excel* странице.

Рекомендации: желательно выполнить расчет местоположения матриц так, чтобы исходные матрицы-сомножители располагались в верхних ячейках страницы, а матрица-произведение ниже их без пересечения областей.

3. Дополнить процедуру формирования отчета в *Word* результатами вычисления обратной матрицы и умножения.

Рекомендации: дополнить процедуры вычисления *флагами*, а при составлении отчетов формировать их в зависимости от того, что было вычислено, то есть от состояния флагов.

4. Синтаксис текста программ проверяет компилятор, в отличие от текстов комментариев в коде программы. Разработать приложение, которое будет считывать файлы .cpp и .h или .hpp программы, написанной на *C++ Builder*, и искать тексты комментариев, и сообщать об ошибках.

Рекомендации: дополнить код поиском русского текста и проверить эти слова на правильность написания. Можно усложнить код поиском знаков начала комментариев, но учесть наличие однострочного комментария и многострочного. Более сложное решение – учесть возможность вложенных комментариев.

СПИСОК ЛИТЕРАТУРЫ

1. Абрамов С.А. Задачи по программированию [Текст] / С.А. Абрамов [и др.]. – М.: Наука, 1988. – 224 с.
2. Архангельский А.Я. C++ Builder 6 [Текст]: справочное пособие. Книга 2: Классы и компоненты. – М.: Бином-Пресс, 2002. – 528 с.: ил.
3. Бобровский С.И.. Delphi 7. Учебный курс [Текст] / С.И. Бобровский. – СПб.: Питер, 2004. – 736 с.: ил.
4. Боровский А.Н. C++ и Borland C++ Builder. Самоучитель [Текст] / А.Н. Боровский. – Спб.: Питер, 2005. – 256 с.
5. Васильков Ю.В. Компьютерные технологии в математическом моделировании [Текст]: учебное пособие / Ю.В. Васильков, Н.Н. Василькова. – М.: Финансы и статистика, 2002. – 256 с.
6. Вирт Н. Алгоритмы + структуры данных = программы: пер. с англ / Н. Вирт. – М.: Мир, 1985. – 406 с.
7. Воеводин В.В. Матрицы и вычисления [Текст] / В.В. Воеводин, Ю.А. Кузнецов. – М.: Наука, 1984. – 320 с.
8. Головач В.В. Дизайн пользовательского интерфейса^2. Искусство мыть слона [Электронный ресурс] / В.В. Головач. – Электрон. дан. – Б.м., 2006. – URL: http://uibook2.usethics.ru (дата обращения: 15.09.2008).
9. Грис Д. Конструирование компиляторов для цифровых вычислительных машин [Текст] / Д. Грис. – М.: Мир, 1975. – 544 с.
10. Гудман С. Введение в разработку и анализ алгоритмов [Текст] / С. Гудман, С. Хидетниеми. – М.: Мир, 1981. – 368 с.
11. Елманова Н. Delphi и технология COM (+CD). Мастер-класс [Текст] / Н. Елманова, С. Трепалин, А. Тенцер – СПб.: Питер, 2003. – 698 с.: ил.
12. Касьянов В.Н. Сборник заданий по практикуму на ЭВМ: учебное пособие для вузов [Текст] / В.Н. Касьянов, В.К. Сабельфельд. – М.: Наука., 1986. – 272 с.
13. Костин В.А. Генерация перестановок [Текст] // Компьютерные инструменты в образовании. – 2003. – № 2. – С. 84–91.
14. Лисков Б. Использование абстракций и спецификаций при разработке программ [Текст] / Б. Лисков, Дж. Гатэг. – М.: Мир, 1989. – 424 с.
15. Математический энциклопедический словарь [Текст] / гл. ред. Ю.В. Прохоров. – М.: Советская энциклопедия, 1988. – 847 с.
16. Мелюхин И.С. Информационное общество: истоки, проблемы, тенденции развитии [Текст] / И.С. Мелюхин. – М.: Изд-во Московского университета, 1999. – 208 с.
17. Морозов В.П. Основы алгоритмизации, алгоритмические языки и системное программирование. Задачник [Текст] / В.П. Морозов, В.В. Шураков. – М.: Финансы и статистика, 1994. – 224 с.
18. Крут Д. Основные алгоритмы [Текст]: в 3 т. / пер. с англ. Г.П. Бабенко, Ю.М. Баянковский; под ред. К.И. Бабенко, В.С. Штаркмана. – М.: Мир, 1976. – Т. 1: Искусство программирования для ЭВМ. — 735 с.
19. Пахомов Б.И. C/C++ и Borland C++ Builder для начинающих [Текст] / Б.И. Пахомов. – СПб.: БХВ-Петербург, 2006. – 640 с.
20. Ревич Ю.В. Нестандартные приемы программирования на Delphi [Текст] / Ю.В. Ревич. – СПб.: БХВ-Петербург, 2005. – 560 с.: ил.

21. Светозарова Г.И. Практикум по программированию на БЕЙСИК [Текст] / Г.И. Светозарова, А.А. Мельников, А.В. Козловский. – М.: Наука, 1986. – 368 с.
22. Седжвик Р. Фундаментальные алгоритмы на C++ [Текст] / Р. Седжвик. – Киев.: Издательство «ДиаСофт», 2001. – 688 с.
23. Уэзерелл Ч. Этюды для программистов: пер. с англ [Текст] / Ч. Уэзерелл. – М.: Мир, 1982. – 288 с.
24. Флёнов М.Е. Библия Delphi [Текст] / М.Е. Флёнов. – СПб.: БХВ-Петербург, 2004. – 880 с.: ил.
25. Флёнов М.Е. Delphi 2005. Секреты программирования (+ CD) [Текст] / М.Е. Флёнов. – СПб.: Питер, 2006. – 266 с.: ил.

Список алгоритмов, реализованных в задачах и методах классов

Глава 1. Формулы

Вычислить значение. Использование sscanf. Отображение формулы. Сборка результата для вывода. Функция задана на всем интервале и не зависит от местоположения аргумента (1.1–1.2). Чтение исходных данных из отдельных строк.

Отображение формулы при вычислении (1.3–1.4). Функция задана разными формулами в зависимости от местоположения аргумента с элементами создания динамических компонентов при выполнении программы (1.5). Выделение чисел из строки символов (1.6). Калькулятор.

Глава 2. Обработка строковых данных методами компонентов

Расположить строки в алфавитном порядке. Из заданного набора слов удалить слова, начинающиеся с заданной буквы. Исключение избыточной (повторяющейся) информации из разных текстов. Создание таблицы, ячейки которой заполнены случайным текстом. Сообщение, имеются ли в выделенном столбце сетки ячейки с совпадающими значениями. Подбор ширины столбцов сетки под ширину текста, расположенного в ячейках сетки. Закрашивание отдельных ячеек сетки.

Глава 3. Табулирование

Таблица значений на заданном интервале с заданным шагом (2.1). Вычисление значения функции в заданной точке. Построение графиков функции и ее производной (2.2), заданных массивами значений. Поиск максимума функции методом сканирования (2.3). Вычисление значения функции, заданной бесконечным рядом. График зависимости числа членов ряда от точности, если функция задана бесконечным рядом.

Глава 4. Обработка векторов/одномерных массивов

Поиск номера максимального элемента. Суммирование элементов, удовлетворяющих условию. Циклический сдвиг элементов массива. Перестановка элементов в массиве. Инвертирование массива. Проверка массива на упорядоченность. Упорядочивание элементов массива (сортировка). Слияние упорядоченных массивов в один с сохранением упорядоченности. Сумма векторов. Разность векторов. Смена знака вектора.

Чтение вектора из файла. Создание вектора ДСЧ. Установка границ векторных значений. Определение максимального и минимального значений вектора.

Чтение вектора из StringList. Запись вектора в StringList.

Генерация множества неповторяющихся случайных целых чисел из заданного диапазона значений. Сортировка массива методом плавающего пузырька. Сортировка массива методом быстрой сортировки. Формирование нового массива без дублирующих элементов. Циклический сдвиг элементов массива. Проверка числа на повторяемость. Выбор неповторяющихся элементов в новый массив. Поиск наибольшей по длине подпоследовательности из одинаковых элементов, расположенных подряд. Объединение двух упорядоченных массивов в один упорядоченный. Получение сдачи наименьшим количеством монет. Расположение элементов в обратном порядке.

Глава 5. Битовые строки

Построение маски для выделения заданной последовательности битов. Рекомбинация (кроссинговер двух хромосом) битов, расположенных в двух разрядных сетках. Инвертирование заданной подпоследовательности битов внутри одной разрядной сетки. Инвертирование битов в массиве данных. Получение значения бита по его номеру. Получение содержимого разрядной сетки числа. Подсчет количества единичных битов в разрядной сетке числа. Вычисление степени числа на основании разрядной сетки показателя степени.

Глава 6. Строка

Оператор сложения строк. Оператор сложения строки с массивом символов. Ввод строковых данных из файла. Вывод строки в файл. Взять из строки слово по его порядковому номеру. Определение количества слов в строке. Сортировка слов по алфавиту. Поиск самого длинного слова в строке. Удаление слов, начинающихся с заданной буквы. Выделение из текста слова по его номеру. Сборка слов в одно предложение. Нахождение одинаковых слов в двух предложениях. Расположение слов по алфавиту. Разделение текста на строки.

Чтение числовых значений из строковых данных.

Глава 7. Двумерные массивы

Операция умножения матрицы на число. Умножение матрицы на вектор. Умножение матриц. Сложение матриц. Присваивание матриц. Транспонирование матрицы «на месте». Получение транспонированной матрицы. Вычисление определителя матрицы. Вычисление обратной матрицы. Чтение матрицы из файла. Запись матрицы в сетку StringGrid. Получение псевдообратной матрицы. Решение СЛАУ. Создание программы «Калькулятор» для выполнения матричных операций. Копирование матриц. Чтение матрицы из файла.

Матричные операторные функции: +, *, = , (int), [].

Операторная функция для присваивания целого числа. Оператор приведения к типу int. Оператор приведения к типу char*.

Контроль правильности задания матриц.

Транспонирование матрицы.

Глава 8. Использование генератора перестановок

Создание итерационного генератора перестановок некоторого множества из *n* элементов в лексикографическом порядке. Создание итеративного генератора перестановок некоторого множества из *n* элементов в антилексикографическом порядке.

Глава 9. Список

Создание списка. Поиск узла с заданным числовым значением. Получение значения узла по заданному адресу. Удаление узла по указанному адресу. Поиск узла, расположенного до заданного указателя. Поиск узла, расположенного после заданного указателя. Удаление узла, расположенного в конце списка. Добавление к списку нового узла по заданному адресу. Перестановка элементов списка с заданными адресами. Проверка значений на совпадение. Сортировка данных в списке.

Удаление из списка дублирующих элементов. Моделирование работы магазина самообслуживания с одним кассовым аппаратом.

Создание очереди. Удаление элемента из очереди. Добавление элемента к очереди.

Создание дека. Добавление элемента к началу/концу дека. Удаление элемента из начала/конца дека. Определение значения элемента по его порядковому номеру.

Глава 10. Стек для обратной польской нотации

Добавление элемента к стеку. Извлечение элемента из стека. Установление ранга для операции. Добавление элемента к стеку. Извлечение элемента из стека. Запись обратной нотации в строку.

Глава 11. Бинарное, или двоичное, дерево

Построение сбалансированного дерева. Построение дерева поиска. Добавление к дереву поиска нового узла. Поиск узла с заданным значением информационного поля. Поиск узла с заданным значением информационного поля с использованием барьерного узла. Удаление из дерева узла с заданным значением информационного поля. Удаление из дерева узла, имеющего двух потомков. Процедуры полного обхода дерева. Создание выходного файла из значений дерева (формирование файла значений из узлов дерева).

Глава 12. Алгоритмы поиска кратчайшего пути

Алгоритм «с возвратом» для поиска всех возможных путей (маршрутов) из одного заданного города в другой заданный город при условии, что в один и тот же город нельзя заходить дважды. Поиск всех маршрутов от одного города до другого методом Дейкстры.

Глава 13. Рисование простейших объектов

Рисование геометрических фигур с окантовкой рамки окна. Построение рисунка, содержащего три геометрические фигуры: круг, квадрат и прямоугольник с заливкой фигур, чтобы каждая имела свой цвет и свою окантовку.

Создание примитивного графического редактора. Помещение текстовой информации в графическое окно. Сохранение нарисованного изображения в виде файлов. Чтение файлов с изображениями с целью коррекции.

Глава 14. Применение технологии `Drag&Drop`

Перекомпоновка графических объектов с помощью технологии «перетащить и оставить». Перетаскивание строки текста из списка строк `TListBox` в окно однострочного редактора `TEdit` внутри одной формы. Перетаскивание строки текста из списка строк `TListBox`, располагающегося на одной форме, в окно однострочного редактора `TEdit`, располагающегося на другой форме.

Глава 15. Построение графиков

Построение графика заданной функции с равномерным шагом. Построение графика функции с минимальным шагом. Построение графика функции на всю высоту области отображения. Установка координатных значений отображаемых областей. Пересчет координат точек математической области в точки экрана. Пересчет координат точек экранной области в точки математические. Построение графика функции с использованием элементов масштабирования. Построение осей координат. Изображение линий осей координат. Вывод основных штрихов по оси OX. Вы-

вод основных штрихов по оси OY. Вывод дополнительных штрихов по оси OX. Вывод дополнительных штрихов по оси OY. Вывод подписей основных штрихов по оси OX. Вывод подписей основных штрихов по оси OY. Вывод подписи оси OX. Вывод подписи оси OY. Вывод названия оси OX. Вывод названия оси OY. Построение графика эпициклоиды, уравнение которой задано в параметрическом виде. Построение изображения графика функции двух переменных в виде цветовых областей. Построение изображения графика функции двух переменных в виде изолиний.

Глава 16. Отображение картографической информации

Отображение картографической информации о системах дорог, связывающих между собой географические объекты двух видов: населенные пункты (города и поселки) и архитектурные сооружения (здания/жилые дома). Добавление к картографическому изображению системы дорог заданного региона, осей координат с предоставлением возможности изменения масштаба изображения. Загрузка карты из файла. Формирование изображения карты. Загрузка «дорог» из файла. Формирование изображения «дорог». Добавление к карте осей координат с предоставлением возможности изменения масштаба изображения.

Глава 17. Графический редактор

Построение рисунка, содержащего три геометрические фигуры: круг, квадрат и прямоугольник с заливкой фигур, чтобы каждая имела свой цвет и свою окантовку. Очистка изображения. Редактирование изображения отрезка. Отображение полного изображения. Отображения редактируемого отрезка в режиме «исключающего или». Редактирование изображения ломаной линии. Отображение редактируемой ломаной линии в режиме «исключающего или». Редактирование изображения кривой Безье. Рисование ломаной замкнутой/незамкнутой линии на основании узлов, отмеченных мышью.

Глава 18. Связь с OLE

Запуск программы Excel. Открытие документов программой Excel и чтения данных с его страниц. Открытие книги. Выбор книги и страницы. Табулирование функций с использованием программы Excel. Создание новой книги и установка указателей на ее первую страницу. Выполнение основных операций с матрицами программой Excel: вычисление определителя матрицы, суммы элементов матрицы, обратной матрицы, перемножение матриц. Формирование отчета в текстовом редакторе Word по результатам обработки матриц.

Программные решения

Список классов

Класс для обработки одномерного массива.
Класс для обработки двумерного массива.
Класс для обработки битов.
Класс для обработки текстов.
Класс «Список». Класс «Стек». Класс «Очередь». Класс «Двусвязный список».
Класс «Координатные оси».
Класс «Стек для польской нотации».
Класс для построения графиков функции.

Класс, позволяющий отображать картографическую информацию о системах дорог.
Класс для отображения карты (с именем TMap).
Класс для отображения системы дорог (с именем TRoad).
Класс для отображения списка дорог (с именем WayNet).
Класс для отображения архитектурных сооружений (с именем TBuilding).

Список решений визуального программирования

Подсоединение обработчика события.
Обмен информацией между окнами одной формы.
Перемещение компонентов по форме.
Показать/активизировать форму, выделенную в списке форм.
Синхронизация списков.
Загрузка векторов в компонент ListBox.
Контроль правильности ввода векторных данных.
Опрос ресурсов программы и формирование списка форм в окне списка.
Контроль правильности задания матриц.
Создание компонента во время работы приложения.
Интерфейс программы для работы с матрицами.
Интерфейс программы, разработанный для «перетаскивания» информации между окнами одного приложения.
Интерфейс программы перемещения компонента по форме.
Интерфейс программы чтения данных из документов Excel.
Интерфейс приложения формирования простейшего изображения.
Интерфейс программы в режиме редактирования ломаной линии.
Интерфейс программы в режиме редактирования кривой Безье.
Интерфейс простейшего графического редактора.
Обработчик OnDragOver здесь ограничивается лишь проверкой типа источника.
Обработчик события OnDragDrop для захвата текста в компоненте Edit.
Обработчик события OnDragOver для компонента Edit с элементом дополнительного анализа (типа источника): проверяется, является ли источник компонентом.
Обработчик события OnDragDrop для компонента Edit с элементом дополнительного анализа: проверяет тип и имя источника, а также тип приемника и лишь потом выполняет операцию.
Обработчик события OnDragOver: проверяется, является ли источник компонентом ListBox.
Обработчик события OnDragDrop проверяет, был ли запрет на право «сбросить» информацию, проверяет тип и имя источника, а также тип приемника и лишь потом выполняет операцию.
Использование встроенных редакторов других визуальных компонентов для редактирования таблицы.

Предметный указатель

A

абстрактные методы 394
алгоритм:
- быстрая сортировка 133
- выбор из массива неповторяющихся элементов 136
- выделение чисел из строки символов 19
- выполнение основных операций с матрицами программой *Excel* 452
- вычисление значения функции 9, 13, 14, 15, 16, 18, 74, 76, 79, 86, 358
- вычисление значения функции, представленной в виде суммы бесконечного ряда 89, 96
- вычисление псевдообратной матрицы 208
- вычисление V^K, используя битовое представление показателя степени K 153
- генерация неповторяющихся случайных чисел из заданного диапазона 131
- Дейкстры 316
- добавление к изображению ломаной линии 430
- добавление к картографическому изображению осей координат 402
- заполнение таблицы случайным текстом с выделением совпадений 42
- изменить положение изображения 416
- имитирование калькулятора 22, 198
- инвертирование последовательности битов в массиве 151
- исключение повторяющихся строк из двух файлов 40
- моделирование работы магазина самообслуживания 266
- объединение двух упорядоченных массивов в один упорядоченный 138
- окантовка сторон окна Image разным стилем 329
- открытие книг программой *Excel* и чтение данных со страниц 441
- перемножение матриц 205
- перетаскивание строки текста 343, 349
- перетаскивание компонента 351-352
- подключение *Word for Windows* для проверки слов 225
- подсчет количества сочетаний 219
- поиск всех возможных путей из одного города в другой 304, 312
- поиск кратчайшего пути между двумя заданными городами 316, 326
- поиск максимального значения методом сканирования 82
- поиск подпоследовательности из одинаковых элементов, расположенных подряд 137
- поиск слов, составленных из заданного множества букв 226
- поиск самого длинного слова 171
- получение всех перестановок всех возможных сочетаний 230
- получение сочетаний 220
- получение суммы векторов 130
- получение таблицы значения 79, 86
- получение таблицы значений функций с использованием *Excel* 448, 451
- построение графика зависимости количества членов бесконечного ряда от точности 89
- построение графика функции 75, 80, 358
- построение графика функции, уравнение которой задано в параметрическом виде 381
- построение изображений 411
- построение изолиний функции двух переменных 384, 388
- построение кроссинговера двух хромосом 148
- проверка символа на цифру 20
- проверка числа на повторяемость в сформированной части массива 132
- расположение строк в алфавитном порядке 35
- расположение элементов массива в обратном порядке 134
- редактирование изображения 424
- редактирование табличных данных 52, 61-62
- решение системы линейных уравнений 209
- рисование геометрических фигур 332, 334
- создание генератора перестановок данных в антилексикографическом порядке 218
- создание латинского квадрата 231
- создание обратной польской нотации 281
- создание перестановки, следующей за данной перестановкой в лексикографическом порядке 212
- создание перестановки, следующей за данной перестановкой в антилексикографическом порядке 215

создание примитивного графического редактора 338
создание списка всех возможных слов из заданного набора букв с заданными длинами 223
удаление слов, начинающихся с заданной буквы 38
формирование нового массива из неповторяющихся элементов 136
формирование отчета в текстовом редакторе *Word* по результатам обработки матриц 460
циклический сдвиг массива 135

Б

базовый класс 103
битовый образ 384

В

вычисления по формулам 9, 13, 14, 15, 16, 18
вычисление суммы бесконечного ряда 90

Г

генератор перестановок 212
граф 303
графическое изображение формулы 15
групповой обработчик событий 53

Д

действия, реализованные в обработчике события:
выбор книги *Excel* 446
выбор страницы книги *Excel* 447
вычисление обратной матрицы средствами *Excel* 457
вычисление определителя матрицы средствами *Excel* 455
вычисление произведения матриц средствами *Excel* 459
вычисление суммы элементов матрицы средствами *Excel* 456
генерация случайного символа из алфавита 45
генерация случайного текста 45
завершение сессии *Excel* 444
загрузка векторных значений 126
загрузка карты 406
загрузка шаблона таблицы 64
задание шрифта и текста в окне Image 336
закрашивание отдельной ячейки сетки 49
закрытие главной формы программы 444
запуск программы *Excel* 443
изменение размеров главной формы программы 406
исправление векторных значений 126, 127
контроль значений 128
коррекция данных в сетке 57
нажатие кнопки мыши на изображении 426, 427, 434
открытие книги *Excel* 444
отображение редактируемого отрезка изображения 419
отображение кривой Безье 430
очистка изображений 415, 421, 433, 407
перемещение «мыши» над компонентом Image 423, 435
перемножение матриц 205
переход к вторичной форме 197
подгон ширины столбцов сетки под ширину текста 46
подключение *Word for Windows* 225
построение графика функции 82, 363
построение полного изображения 420
проверка корректности векторных данных 128
расположить элементы массива в обратном порядке 134
реакция калькулятора на ввод операции 30
реакция калькулятора на ввод констант “π”, “e” 30
реакция калькулятора на ввод очередной цифры числа 28
реакция калькулятора на знак «=» 29
реакция калькулятора на ввод символа операции 30
реакция калькулятора на смену знака 30
синхронизация списков 129
создание главной формы программы 405
создание новой книги 450
создание списка покупателей 272
сортировка строк 37
создание экземпляров классов 121
сохранение строк в файле 38, 39
сформировать новый массив без дубликатов 134
очистка изображения 421
очистка строки ввода 27
удаление из дерева всех файлов, последнее обращение к которым происходило до определенной даты 299
удаление последнего введенного символа 27
установка региона 407

формирование документа *Word* 461
формирование списка вторичных форм 196
формирование списков 59
деструктор 107, 115, 185
деструктор потомка 109, 119, 119-120
дерево бинарное 280
дерево поиска 280
дополнительные графические элементы 417

З

задание значений свойства *Options* 47
задачи конструктора 104

И

идеально сбалансированное дерево 281
изолиния 384

К

классы:
Calc 274
Poisk 308
PoiskEx 310
PoiskTask 313
Stroka 158
StrokaEx 166
StrokaTask 169
TArray 103
TAxes 369, 402
TBuilding 401
TDataDeck 258
TDataList 237
TDataListEx 246
TDataListTask 250
TDataStack 254
TDataQueue 255
TBaseElem 393
TFuncXYRegionGraphic 386
TMap 396
TMatrix 175-176
TMatrixEx 187
TMatrixTask 193
TParamGraphic 382
Tree 282
TreeEx 283
TreeTask 294
TRoad 399
TStrings 34
TStringList 34
TVector 107
TVectorEx 113
TVectorInBit 145
TVectorTask 119
WayNet 322
компонент Chart 80
конструктор:
без параметров 104
задачи конструктора 104
класса-потомка 109, 115, 119, 123, 192, 198
копирования 106, 120, 160, 181, 188
с параметрами – размерами массива 104
с параметром «Имя файла» 105, 117
кривая Безье 395

Л

латинский квадрат 231

М

маска 140
масштабирование данных 361
матрица инцидентности 303
матрица смежности 303
машинное слово 140
метод проб и ошибок 305
методы класса:
AddXY 80
LineTo 331
MoveTo 331
TextOut 335
OleFunction 440
OleProcedure 440
OlePropertyGet 440
OlePropertySet 440
CreateObject 440
GetActiveObject 440
IsEmpty 440
взять из строки слово по заданному номеру 165
вставить значение в начало списка 241
вставить значение по заданному адресу 240
выделить слово из строки 168
вывести дополнительные штрихи по осям 375
вывести в список матрицу инцидентности 314
вывести маршрут 315
вывести название оси 379
вывести основные штрихи по оси 373, 374
вывести подписи основных штрихов по осям 376-378
вывести слова в окно списка 170
вывести список в сетку 278
вывести строки в окно списка 170
вывести строку в окно редактора 170
вычисление обратной матрицы 191

вычисление определителя матрицы 190
вычисление цвета растра 399
добавление к дереву поиска нового узла 288
добавление элемента к деку 260, 261
добавление элемента к очереди 257
добавление элемента к стеку 284, 275, 276
загрузка карты из файла 398
загрузка «дорог» из файла 400
задание левой границы векторных значений 118
задание правой границы векторных значений 118
запись векторных значений в список 121
запись матрицы в сетку 194
запись обратной польской нотации в строку 277
извлечение элемента из списка 275
изменение значения элемента массива с заданным номером 112
изображение квадрата и прямоугольника 412
изображение линий осей координат 372
изображение окружности 412
инвертирование заданной подпоследовательности битов внутри одной ячейки 146
инициализация вершин системы дорог 324
определение количества слов и инициализация полей класса 167
отображение списка 252
переопределение длины вектора 112
перестановка значений элементов списка с заданными адресами 248
перестановка элементов списка с заданными адресами 249
подсчет количества единичных битов в разрядной сетке числа 146
поиск всех маршрутов от одного города до другого 312
поиск ненулевого диагонального элемента в матрице 190
поиск узла по указателю 244
поиск узла с заданным числовым значением 241, 290
получение адреса узла 290
получение значения бита с заданным номером разрядной сетки 143
получение значения узла по заданному адресу 242
получение/распечатка содержимого разрядной сетки числа 145
построение дерева поиска 287
построение маски 142
построение осей координат 371
построение сбалансированного дерева 286
распечатка вершин системы дорог 325
связывание вершин системы дорог с ребрами 325
сделать список пустым 245, 262
создание вектора с помощью датчика случайных чисел 117
создание дека 259
создание очереди 256
создание списка 239
сортировка методом плавающего пузырька 132
сортировка данных в списке 249
сортировка слов по алфавиту 168
транспонирование матрицы 185,186
удаление из списка совпадающих элементов 247
удаление узла из дерева 291, 294
удаление узла по указанному адресу 243
удаление узла с вершины списка 246
удаление узла со дна списка 246
удаление элемента из дека 261
удаление элемента из очереди 257
установка левой границы векторных значений 117
установка правой границы векторных значений 118
установление ранга для операции 276
формирование изображения «дорог» 400
формирование изображения карты 399
чтение вектора из файла 117
чтение векторных значений из списка 120
чтение из файла матрицы инцидентности 314
чтение матрицы из сетки 194
модальное окно 123

О

обмен информацией между окнами 349
оператор:
[ ] 109, 112
= 110, 161, 182, 189, 268
+ 111, 166, 162, 269, 309
* 111, 183, 184
< 269, 277, 310
➢ 310
== 309
>> 163, 310
<< 164, 310
! 183, 269

!= 269
int 162
char* 162
операция «исключающего или» 414
описание переменной типа «класс» 121
определение размеров ячеек сетки 63

П

параметр:
ACol 48
Accept 48
ARow 48
CanSelect 51
Rect 48
Sender 12
Source 345
State 48
Target 348
X 48
Y 48
перемещение компонентов по форме 351
пиксель 328
подсоединение обработчика события 12
порядок подготовки кодов программ 72
правила использования сторонних редакторов для исправления данных в сетке StringGrid 62

Р

размеры ячеек редактирования 63
разработка классов:
«Биты» 140
«Двоичное дерево» 281
«Деки» 257
для алгоритма Дейкстры 316
для отображения графика двумерной функции в виде цветовых областей 386
для отображения дорог 399
для отображения зданий 401
для отображения картографической информации о системах дорог, связывающих между собой географические объекты 393
для построения графика функции, заданной в параметрическом виде 382
для «Системы дорог» 308
для отображения карты 396
«Линейный односвязный список» 234
«Матрица» 173
«Одномерный массив» 100
«Оси координат» 367
«Очередь» 254
«Стек» 253
«Строка» 157
разработка интерфейса:
для решения задач с классом «Одномерный массив» 122
для решения задач с классом «Матрицы» 195
для перемножения матриц 198
для решения задач с классом «Список» 262
растр математический 386
растр экранный 385

С

свойства:
Canvas (холст) 328
Cells 35
DragMode 354
KeyPreview 26
property 176
сервер и контроллер 437
сессия 444
события:
OnClick 10
OnDragOver 345
OnDragDrop 345
OnDrawCell 48
OnEndDrag 348
OnKeyUp 31
OnMouseDown 422
OnMouseUp 423
OnMouseMove 423
OnSelectCell 51
OnStartDrag 352
создание компонента во время работы приложения 17
сортировка:
быстрая сортировка 133
методом плавающего пузырька 132
сплайн 428

Т

технология:
Drag&Drop 341
связи с *Excel* по OLE 438
связи с *Word for Windows* по OLE 460
тип:
AnsiString 10
Rect 48
Variant 440
__closure 298

У

указатель this 106
унимодальная функция 83
управление классом через элемент интерфейса 124

Ф

фон 410
формирование изображения карт 393
функция:
- EncodeTime 268
- friend – друг 111
- LineTo 341
- MoveTo 341
- PolyBezier 429
- scanf 13
- sscanf 14
- strtok 165
- TextRect 49
- TextOut 335
- запись элементов целочисленного массива в строку AnsiString 218
- отображение полного изображения 419, 420, 425, 430
- отображение редактируемого отрезка 419, 422, 433
- отображение редактируемой ломаной 426
- пересчет координат точек математической области в точки экрана 362
- пересчет координат точек экранной области в точки математические 363
- полный обход дерева 296
- построение базового рисунка 433
- формирование таблицы значений с заданным шагом 74
- установка координатных значений отображаемых областей 362

Х

характеристическая ломаная 428

Оглавление

Введение ... 3
Глава 1. Вычисление значений функций по заданным формулам ... 9
Глава 2. Алгоритмы обработки строковых данных методами компонентов C++ Builder, содержащих списки строк ... 34
Глава 3. Табулирование функций. Построение графиков таблично заданных функций ... 70
3.1. Общая постановка задачи и основные положения ... 70
3.2. Табулирование функций, заданных аналитическим выражением ... 72
3.3. Табулирование функций, заданных в виде суммы бесконечного ряда ... 89
Глава 4. Класс «Одномерный массив» ... 100
4.1. Разработка классов одномерных массивов ... 100
4.2. Управление классом через элементы интерфейса ... 121
4.3. Разработка интерфейса приложения для решения задач с классом «Одномерный массив» ... 122
4.4. Решение задач векторной алгебры с использованием класса «Одномерный массив» ... 130
Глава 5. Разработка алгоритмов, предназначенных для обработки битовых строк ... 140
Глава 6. Класс «Строка» ... 157
6.1. Разработка классов «Строка» ... 157
6.2. Решение задач с использованием классов «Строка» ... 171
Глава 7. Класс «Матрица» ... 173
7.1. Разработка классов «Матрица» ... 173
7.2. Разработка интерфейса приложения для решения задач с классом «Матрица» ... 195
7.3. Решение задач линейной алгебры с использованием типа «Матрица» ... 198
Глава 8. Использование элементов комбинаторной математики в решении задач ... 212
Глава 9. Класс «Список» ... 234
9.1. Разработка классов «Линейный односвязный список» ... 234
9.2. Разработка классов «Стеки» ... 253
9.3. Разработка классов «Очередь» ... 254
9.4. Разработка классов «Деки» ... 257
9.5. Разработка интерфейса приложения для решения задач с классом «Список» ... 262
9.6. Решение задач с использованием классов «Список» ... 266
Глава 10. Запись алгебраического выражения в виде обратной польской нотации ... 273
Глава 11. Класс «Бинарное дерево» ... 280
11.1. Разработка классов «Бинарное или двоичное дерево» ... 281
11.2. Решение задач с использованием классов «Бинарное или двоичное дерево» ... 299
Глава 12. Алгоритмы поиска кратчайшего пути ... 303
Глава 13. Рисование простейших графических объектов ... 328

Глава 14. Применение технологии Drag&Drop для работы пользователя с интерфейсом программы 341
Глава 15. Графики и карты 357
15.1. Построение графиков функций 357
15.2. Использование элементов масштабирования при построении графиков 360
15.3. Разработка класса «Оси координат» 367
15.4. Построение изолиний функции двух переменных 383
Глава 16. Отображение картографической информации 392
Глава 17. Разработка элементов графического редактора 409
Глава 18. Связь приложения с MS Excel и Word по технологии OLE 437
18.1. Связь с MS Excel по технологии OLE 438
18.2. Связь с MS Word for Windows по технологии OLE 460
Список литературы 464
Список алгоритмов 466
Предметный указатель 471

Учебное издание

РЫБАЛКА Сергей Анатольевич
ШКАТОВА Галина Ивановна

C++ BUILDER
ЗАДАЧИ И РЕШЕНИЯ

Учебное пособие

Научный редактор *доктор физико-математических наук, профессор В.П. Григорьев*

Выпускающий редактор *Т.С. Савенкова*
Редактор *А.В. Высоцкая*
Компьютерная верстка *С.А. Рыбалка*
Дизайн обложки *Т.А. Фатеева*

Подписано к печати 14.05.10. Формат 60x84/8. Бумага «Снегурочка».
Печать XEROX. Усл.печ.л. 27,86. Уч.-изд.л. 25,20.
Заказ 856-10. Тираж 100 экз.

Национальный исследовательский Томский политехнический университет
Система менеджмента качества
Томского политехнического университета сертифицирована
NATIONAL QUALITY ASSURANCE по стандарту ISO 9001:2008

ИЗДАТЕЛЬСТВО ТПУ. 634050, г. Томск, пр. Ленина, 30.
Тел./факс: 8(3822)56-35-35, www.tpu.ru