С. Бобровский

самоучитель программирования на языке C++

в системе Borland C++ Builder 5.0

ДЕСС
I-PRESS

Русский учебник

С. Бобровский

Самоучитель программирования на языке C++ в системе Borland C++Builder 5.0

«ДЕСС КОМ»
«I-Press»
Москва - 2001

С. Бобровский

Самоучитель программирования на языке C++ в системе Borland C++Builder 5.0

Книга подготовлена редакцией «I-Press» по заказу издательства «ДЕСС КОМ»

Группа подготовки издания;

Главный редактор:	Сергей Симонович
Научный релдактор:	Георгий Евсеев
Литературный редактор:	Ирина Симонович
Компьютерная верстка:	Ирина Симонович
Корректор:	Елена Сорокина

ISBN 5-93650-013-6 («ДЕСС КОМ»)

Лицензия ИД №00219 от 11.10.99. Подписано в печать 10.11.2000.
Формат 70x100/16. Печать офсетная. Печ. л. 17. Доп. тираж 1000. Заказ 1064

ООО «ДЕСС КОМ», 105484, г. Москва, ул. 16-я Парковая, д. 21, корп. 1.

Отпечатано с готовых диапозитивов
в Академической типографии «Наука» РАН
199034, С.-Петербург, 9 Линия, 12

Содержание

Введение

Что такое язык программирования?

Абсолютно все программы, будь то компьютерная игра, служебная программа для архивирования данных, обозреватель для работы в Интернете или операционная система Windows, написаны на одном или нескольких языках программирования. Сегодня в мире насчитывается около 400 таких языков, более-менее активно используемых для создания программ, а также еще несколько тысяч языков, давно забытых или не получивших широкой известности, иногда незаслуженно.

Язык программирования позволяет описывать алгоритмы с помощью набора определенных *(ключевых)* слов и различных вспомогательных символов. Используя ключевые слова (они называются *командами* или *операторами),* программист определяет конкретную последовательность действий, которую компьютер должен выполнить, чтобы решить требуемую задачу. Например, когда в компьютерной игре персонаж, управляемый программой, решает, зайти ли ему в замок или продолжить движение по дороге, он исполняет алгоритм, который программист мог записать примерно так:

```
проверить силу защитников замка;
сравнить силу своей армии и вражеской;
если своя армия сильнее противника в три и более раз,
  то
    атаковать замок;
иначе
    продолжить движение по дороге.
```

Исходные тексты программ очень напоминают обычные тексты, написанные на естественном языке. Это сделано специально, чтобы облегчить труд программиста.

Что такое компилятор?

После того как исходный текст набран (это можно сделать в любом текстовом редакторе, хотя для этого имеются и специальные приложения), его необходимо преобразовать в программу, которая будет исполняться на компьютере. Важно понять, что сам текст только формально описывает алгоритм вычислений — он *не является программой*.

Дело в том, что процессор может исполнять только двоичный код, представляющий собой очень простые *машинные команды*. В машинном коде написать более-менее сложную программу практически невозможно. Там, где в обычном языке программирования для сложения пары чисел достаточно одного оператора, могут потребоваться десятки машинных команд.

Когда мы говорим о том, что машинные команды очень просты, а написать приличную программу с их помощью невероятно трудно, то здесь нет противоречия. Все детали механической части современного автомобиля совершают очень простые вращательные или поступательные движения, но попробуйте создать автомобиль из куска металла с помощью молотка и напильника! Для создания автомобиля нужны специальные инструментальные средства.

Точно так же дело обстоит и в вычислительной технике. Здесь нам тоже нужны специальные инструментальные средства. Перевод текста в двоичный код осуществляется специальными программами, которые называют *трансляторами*. Они *транслируют,* то есть переводят тексты, написанные на языке программирования, в машинный код.

Как в реальной жизни существует два класса переводчиков: синхронные и литературные, — так и в вычислительной технике существует два класса трансляторов: интерпретаторы и компиляторы. *Интерпретатор* работает как синхронный переводчик. Он просматривает исходный текст строку за строкой, переводит каждую строку в промежуточный или сразу в машинный код и передает его на исполнение. Только если все в порядке, интерпретатор приступает к следующей строке. Компилятор (а именно к этому классу относится рассматриваемая нами система Borland C++) работает как литературный переводчик. Сначала он просмотрит весь текст, может быть и не один раз, найдет общие повторяющиеся места (их он не будет переводить дважды), тщательно подготовит стратегию перевода, подберет самые эффективные аналоги и только после этого переведет весь исходный текст целиком и полностью, создав при этом новый документ, который называется *объектным кодом*. Объектный код можно считать законченной программой, хотя и не вполне.

Поскольку компилятор все равно предварительно просматривает весь исходный текст, он может заодно проверить, не наделал ли разработчик программы ошибок. Ведь когда мы набираем на компьютере красиво оформленное поздравление своему другу с днем рождения, мы можем ошибиться, написав *приват* вместо *привет,* и не заметить этого. А транслятор сразу распознает незнакомое слово, как это делают программы проверки орфографии в текстовых редакторах, и сообщит о выявленной неточности. Конечно, законченного объектного кода при этом не получится. Пока компилятор не перестанет натыкаться на ошибки в исходном тексте, получить готовую программу мы не сможем. А исправлять эти ошибки надо самостоятельно: компилятор только указывает неверную строку и кратко описывает обнаруженную проблему.

Опечатки — достаточно распространенные, но не самые вредные ошибки. Бывают ошибки и похуже. Если в поздравлении написать все слова грамотно, но сделать *логическую ошибку,* например написать *Поздравляю с 30-летием* вместо *с 20-летием,* то никакая программа такую ошибку не выявит, а последствия от нее могут быть гораздо серьезнее. Наш друг скорее всего улыбнется при виде описки в слове *приват,* но может серьезно обидеться, если добавить к его возрасту десяток лет.

Точно также происходит и в программах. Если перепутать знаки сравнения в вышеприведенном примере, когда персонаж решает, атаковать ли ему противника, и случайно написать:

```
если своя армия меньше армии противника в три и более раз, то
    атаковать замок
```

(в тексте программы для этого достаточно вместо знака «>» набрать знак «<«), то компьютерный герой будет постоянно бросаться на превосходящие силы врага, а от слабых войск будет наоборот убегать.

Находить такие логические ошибки довольно трудно. Согласно статистике, 60% всего времени при создании программ уходит на поиск и выявление ошибок (этот процесс называется *отладкой).* Именно в этом и заключается мастерство настоящего программиста — писать свои программы как можно более качественно и надежно. Такое мастерство приходит только с опытом.

Почему C++?

Язык Си++ официально получил свое название в 1983 г. Он был создан на основе более старого языка Си и имел целью упростить процесс создания программ. Си++ позволил программистам составлять алгоритмы с помо-

щью привычных общечеловеческих понятий. Если программирование на Си напоминает скорее программирование в машинных кодах, то в Си++ можно настроить программу на конкретную предметную область и работать не с числами и переменными, а, например, с такими понятиями, как *армия, отряд, боевая единица,* что значительно легче и удобнее. При этом Си++ сохраняет преимущества Си и позволяет добиться весьма высокого быстродействия получаемых программ.

Сегодня язык Си++ очень широко распространен во всем мире. Большинство программ как в России, так и за рубежом создают именно на этом языке. В частности, операционная система Windows написана средствами языка Си++.

Что такое визуальное программирование?

Изучить команды языка Си++ и научиться писать на нем небольшие программы, например, вычисляющие сумму двух чисел, можно очень быстро. Однако сам язык не содержит никаких средств, позволяющих организовать ввод чисел в программу и их отображение на экране. В нем вообще нет никаких средств для организации взаимодействия с пользователем. Более того, в Си++ нет никаких средств для создания окон и элементов Windows. Да и придумывался Си++, когда Windows еще не было.

Средства для организации взаимодействия с пользователем, например окна, кнопки, меню и другие элементы управления, называют *интерфейсом пользователя.* Windows — графическая операционная система, поэтому говорят, что она обеспечивает *графический интерфейс пользователя.*

Графический интерфейс пользователя состоит из элементов оформления и элементов управления. Взгляните на экран любой программы Windows, и вы легко отличите элементы управления от элементов оформления. Элементами управления можно управлять с помощью мыши. Элементы оформления пассивны — их можно только смотреть и читать, а если речь идет о звуковом оформлении, то и слушать.

Все, что требуется для организации простого пользовательского интерфейса с помощью Си++, выделено в специальные библиотеки, содержащие множество самых разных дополнительных средств. Эти библиотеки имеются для большинства операционных систем и аппаратных платформ, благодаря чему одна и та же программа на Си++ может работать на разных типах платформ — надо только сменить библиотеку и заново выполнить компиляцию.

Так можно без существенных изменений перенести программу, например, из операционной системы Windows в операционную систему Linux.

Однако использование подобных библиотек для организации графического интерфейса весьма трудоемко. Если с их помощью попытаться сделать на Си++ самую простую программу для Windows, которая будет выводить на экран строку «Привет всем!», то потребуется написать сотни строк громоздкого и малопонятного исходного кода. А если мы захотим добавить в рабочее окно программы *элемент управления* (кнопку, меню и т. п.) или *элемент оформления* (например рисунок), то такая работа превратится в настоящее мучение.

К счастью, производители средств программирования пошли по пути, отличному от простого выпуска подключаемых библиотек. Сегодня они предлагают авторам программ так называемые *среды быстрой разработки* (*RAD-среды*), которые берут на себя всю рутинную работу, связанную с подготовкой программы к работе, автоматически генерируют соответствующий программный код и позволяют нам сосредоточиться не на оформлении интерфейса, а на логике работы будущей программы.

Различные элементы управления, такие, как кнопки, переключатели, значки и другие объекты Windows (которые в терминологии RAD-систем называются *компонентами),* можно перетаскивать в проектируемом окне с помощью мыши. Процесс создания интерфейса будущей программы напоминает забаву с игровым компьютерным конструктором. Поэтому RAD-среды еще называют *визуальными средами разработки:* какими мы видим рабочие и диалоговые окна программы при проектировании, такими они и будут, когда программа заработает.

В итоге программисту остается только определить, что должна делать программа при наступлении определенного события:

- при щелчке мышью на той или иной кнопке;
- при выборе определенного пункта меню;
- по прошествии определенного интервала времени;
- и вообще при наступлении какого-либо иного события, которое может произойти с программой или с операционной системой, под управлением которой она работает.

Поэтому программирование в RAD-средах называют *событийно-ориентированным.* Конечно, реагировать надо не на все события, а только на те, которые требуются для полноценной работы будущей программы.

С помощью всевозможных визуальных редакторов или Мастеров (специальных программ, которые серией последовательных запросов определяют, что мы хотим сделать) можно подчас создать программу, не написав вручную ни одной строчки кода! Более того, разработчику в большинстве случаев совершенно не надо знать внутреннее устройство Windows, а ведь это сотни специальных системных вызовов, разобраться в которых начинающему довольно сложно. Все нюансы работы операционной системы скрыты внутри готовых компонентов, которые можно использовать как строительные кубики для «складывания» своей программы.

Нестандартные компоненты (например, круглые кнопки или модули шифрования) распространяются самыми разными способами. Некоторые свободно доступны через Интернет, некоторые продаются как shareware-продукты. Вы и сами можете попробовать создать свой компонент и попытаться его распространить, если он окажется полезным.

Почему Borland C++Builder?

Пятая версия продукта Borland C++Builder, вышедшая в начале 2000 года, сегодня является наиболее совершенной визуальной средой быстрой разработки на Си++ для Windows. В ее состав входит около 200 самых разных компонентов, а создание законченной программы требует минимума усилий. Ближайший конкурент Borland C++Builder — это не система Microsoft Visual C++, которая построена по другой схеме и не является RAD-системой, а Microsoft Visual Basic, типичная среда разработки. Однако эффективность программ, создаваемых с помощью C++Builder, в десятки раз превосходит быстродействие программ, написанных на MS Visual Basic. Да и по числу свободных доступных компонентов равных среде C++Builder сегодня не найти.

У этой системы есть родной брат — RAD-среда Borland Delphi, технология работы с которой полностью совпадает с технологией, принятой в C++Builder. Только в Delphi программный код пишется не на языке C++, а на языке программирования Паскаль, точнее на его объектно-ориентированной версии ObjectPascal. Но самое интересное, что Borland C++Builder позволяет писать программу при желании одновременно и на Си++, и на Паскале!

Какой нам нужен компьютер?

Разработка программ, особенно в RAD-системах, требует хорошего компьютера. На жестком диске надо выделить около 500 Мбайт рабочего пространства, оперативной памяти требуется хотя бы 32 Мбайт — это нижний предел,

Конечно, всегда неплохо иметь быстрый процессор, но при создании программ он решающей роли не играет, потому что процесс компиляции (преобразование исходного текста Си++ в готовую программу) происходит очень быстро.

Отлаживать программу можно в Windows 95 или Windows 98, хотя желательно на этом этапе использовать операционную систему Windows NT. Она значительно более устойчива, чем ее более универсальные коллеги Windows 9x. Неотлаженные программы могут содержать множество ошибок, которые легко приводят операционную систему в стрессовое состояние, хотя для не слишком крупных проектов, создаваемых в одиночку, операционная платформа решающего значения не имеет.

Если вы решите заниматься отладкой программы в Windows NT, то ее окончательную версию надо собрать в C++Builder все-таки на той платформе, на которую программа в первую очередь ориентирована. То есть, если вы рассчитываете, что программа будет запускаться в Windows 98, то и скомпилировать ее в окончательном виде надо тоже в Windows 98, после чего обязательно проверить, как она работает. Впрочем, программы, созданные средствами C++Builder, почти всегда одинаково хорошо выполняются и в Windows 9x и в Windows NT.

Монитор желательно иметь чем больше, тем лучше. Программирование требует пристального внимания и тщательного рассматривания текстов С++ и проектируемых окон. Кроме того, процесс программирования становится проще, если удается охватывать взглядом большие части исходных текстов, поэтому неплохо установить на мониторе максимально допустимое разрешение.

Наша рекомендация:

монитор — 17 дюймов, разрешение экрана — 1024x768.

В крайнем случае:

монитор — 15 дюймов, разрешение экрана — 800x600.

Если глаза не слишком протестуют:

монитор — 15 дюймов, разрешение экрана — 1024x768.

1. Первое знакомство

Установка Borland C++Builder 5

Чтобы начать работу с системой программирования Borland C++Builder, ее надо установить *(инсталлировать)* на своем компьютере. Делается это следующим образом.

1. Закройте все активные программы. Вставьте дистрибутивный диск с Borland C++Builder в соответствующий дисковод. Программа установки запустится автоматически, но, если этого по каким-то причинам не произошло, вы можете запустить ее любым принятым в Windows способом. Установочная программа называется instalt.exe и расположена в корневом каталоге дистрибутивного диска.

2. После запуска устанавливающей программы на экране появится начальное окно выбора устанавливаемого пакета (на диске помимо C++Builder есть и другие приложения). Если установить указатель мыши на надписи C++Builder (см. рис. 1), появится всплывающая подсказка, информирующая нас о том, что для установки пакета необходимо 543 Мбайт свободного пространства на жестком диске (на самом деле это максимальное требование).

3. Щелкните левой кнопкой мыши на надписи C++Builder — произойдет запуск программы установки системы. Дальнейший ход установки протекает под управлением Мастера установки.

4. Ознакомившись с общей информацией о продукте в первом диалоговом окне Мастера, щелкните на кнопке Next (Далее).

5. Во втором диалоговом окне Мастера установки следует ввести серийный номер продукта и код авторизации — после их ввода кнопка Next (Далее) станет активной.

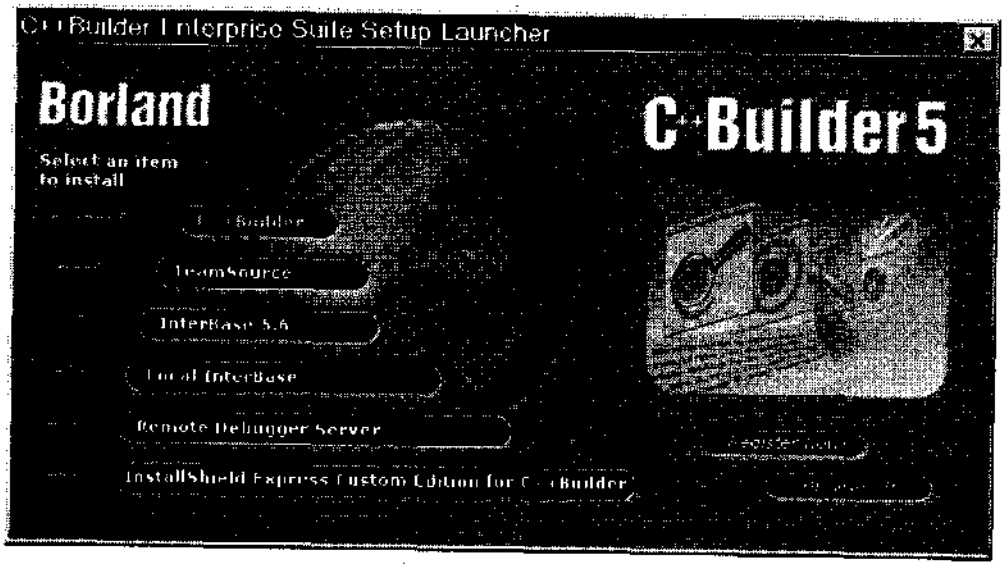


Рис. 1. Выборустанавливаемогопакета

6. Очередное диалоговое окно Мастера установки содержит текст лицензионного соглашения. Если вы думали, что приобрели систему C++Builder 5, то лицензионное соглашение напомнит о том, что это не совсем так. Подобные системы стоят не один десяток миллионов долларов, и вы приобрели только лицензию на право ее использования, а заодно на право владения всем тем, что вы с ее помощью создадите. Если вы согласны с таким положением вещей, щелкните на кнопке Yes и установка будет продолжена.

7. Ознакомившись с краткой информацией об устанавливаемом продукте в очередном диалоговом окне Мастера, щелкните на кнопке Next (Далее)...

8. ... а далее программа предлагает четыре варианта установки системы:
   - Стандартная установка (Typical);
   - Компактная установка (Compact);
   - Заказная установка (Custom);
   - Полная установка (Full).

По умолчанию принят стандартный вариант (Typical). Он достаточен для первого знакомства с системой и рекомендуется начинающим. В варианте компактной установки (Compact) система теряет большинство своих возможностей. Вариант заказной установки (Custom) позволяет самостоятельно

выбрать необходимые компоненты системы, но он рекомендуется для профессионалов, имеющих опыт работы с предыдущими версиями пакета. Вариант полной установки (Full) комментариев не требует, но зато требует около 550 Мбайт свободного пространства на жестком диске.

Если вы не выбрали вариант полной установки, то в дальнейшем при необходимости всегда сможете доустановить пропущенные компоненты.

9. Программа установки спросит, поддержка какой версии офисного пакета Microsoft Office вам желательна (MS Office 97 или MS Office 2000) — C++Builder содержит средства автоматического управления такими приложениями, как редактор Word, электронная таблица Excel и т. п.

10. Мастер предложит выполнить регистрацию новых типов файлов (которые будут создаваться в работе с системой) и присвоить им стандартные значки (см. рис. 2).

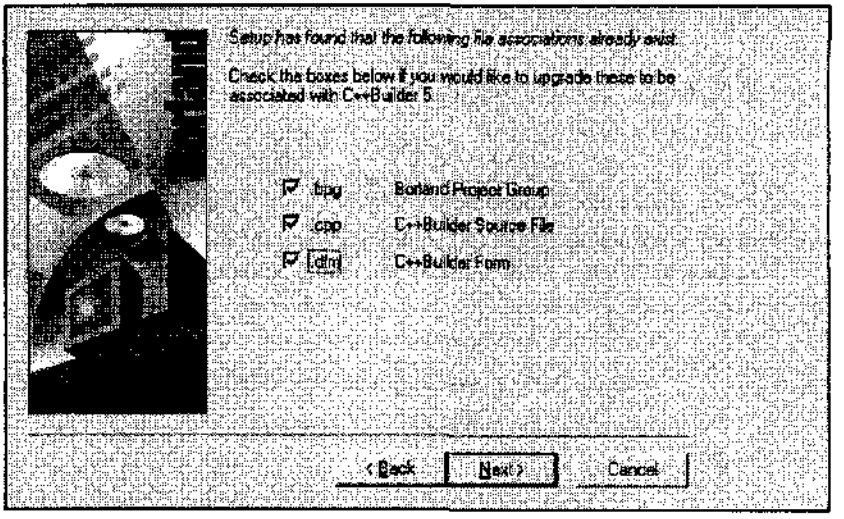


Рис. 2. Регистрация новых типов файлов

11. В следующем окне следует указать, требуется ли нам создавать распределенные приложения, компоненты которых будут работать на разных компьютерах. Это отдельная, очень большая и достаточно сложная область программирования, так что пока не устанавливайте флажок в строке Install VisiBroker For C++Builder.

Здесь же можно указать, требуется ли нам обращаться из создаваемых программ к системе управления базами данных (СУБД) InterBase. Наша книга предназначена для начинающих, и работа с базами данных

в ней не рассматривается, поэтому сбросьте до лучших времен соответствующий флажок Install InterBase Client.

Базы данных — это файлы, содержащие информацию, организованную по одному шаблону. Например, многие компании, торгующие компьютерами, ведут базы данных по своим товарам на основе примерно такого шаблона:

Процессор/ Частота, МГц/ОЗУ, Мбайт/Жесткий диск, Гбайт/Цена, $

Тогда в каждой строке файла базы данных можно очень компактно хранить описания всех моделей компьютеров, имеющихся в продаже, например так:

Pentium II/450/64/10/500

Вообще, любые данные, которые можно представить на основе одного и того же шаблона, удобнее всего хранить в базах данных? а обрабатывать с помощью специальных программ, которые называются *системами управления базами данных (СУБД).* Обрабатывать информацию баз данных можно также и с помощью некоторых систем программирования, например таких как Borland C++Builder.

12. Щелкните на кнопке Next (Далее), и вы получите еще одно лицензионное соглашение. На этот раз оно накладывает определенные ограничения на создание распределенных программ. Так как в ближайшее время нужды в разработке таких приложений у нас не возникнет, щелкните на кнопке Agree (Согласен).

13. В очередном диалоговом окне (см. рис. 3) надо указать каталог, в который будет устанавливаться C++Builder. Проще всего это сделать с помощью кнопки Browse (Обзор), которая позволит выбрать нужный каталог с помощью стандартного диалога. При этом пути во всех строках, определяющих каталоги для установки отдельных компонентов C++Builder, автоматически изменяются в соответствии с выбранным вариантом.

    Если на компьютере уже установлена другая система программирования компании Inprise (например, система Borland Delphi), то некоторые пути будут указывать на каталоги, общие для обеих систем. В любом случае в данном окне лучше ничего не менять, предварительно убедившись, что места на соответствующих дисках достаточно. Для каждой части C++Builder указывается, сколько места для установки на предложенном диске необходимо (Req:) и сколько имеется в наличии (Avail:).

    Закончив настройку, щелкните на кнопке Next (Далее).

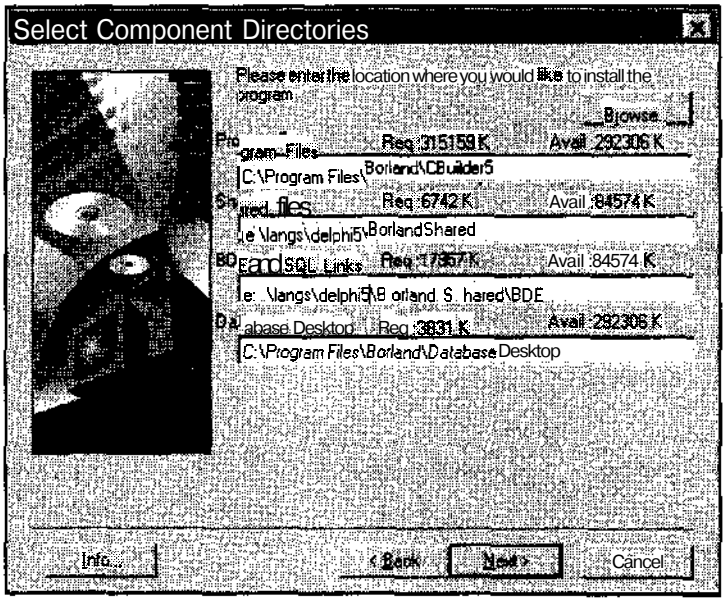


Рис. 3. Выбор папок для устанавливаемых компонентов

14. В следующем окне Мастера установки предлагается указать название папки и соответствующего раздела в Главном меню, где сохранятся значки всех элементов C++Builder после установки. Не меняя предлагаемого названия, щелкните на кнопке Next (Далее).

15. В последнем диалоговом окне Мастера установки проверяют окончательные настройки и щелчком на кнопке Install (Установить) запускают процесс.

    Процесс установки отображается на шкале хода работы. В среднем, стандартная установка требует около 10 минут. По окончании установки Мастер предложит просмотреть файл readme.txt с мелкими замечаниями и дополнениями к текущей версии C++Builder. Для удобства дальнейшей работы установите все флажки и щелкните на кнопке Next (Далее).

16. По окончании установки следует перезапустить компьютер. Щелкните на кнопке Finish (Готово) и дождитесь перезапуска Windows.

Запуск C++Builder

Итак, установка системы программирования нами успешно завершена. Доступ к программам группы Borland C++Builder осуществляется через Главное меню командой Пуск • Программы • Borland C++Builder 5. В этой группе

помимо самой системы C++ Builder содержатся некоторые вспомогательные программы, предназначенные в основном для работы с базами данных. Здесь же вы найдете группу Help, в которой находится полное справочное руководство по всем аспектам работы с системой.

Интегрированная среда разработки

После запуска программы на экране откроется *интегрированная среда разработки (IDE)* в начальном состоянии. Обратите внимание на этот термин — он достаточно часто встречается в программистской практике. В Borland C++Builder 5 эта среда состоит их четырех компонентов (см. рис. 4):

- панели управления C++Builder;
- панели Инспектора объектов (Object Inspector);
- визуального проектировщика рабочих окон;
- окна редактора программы.

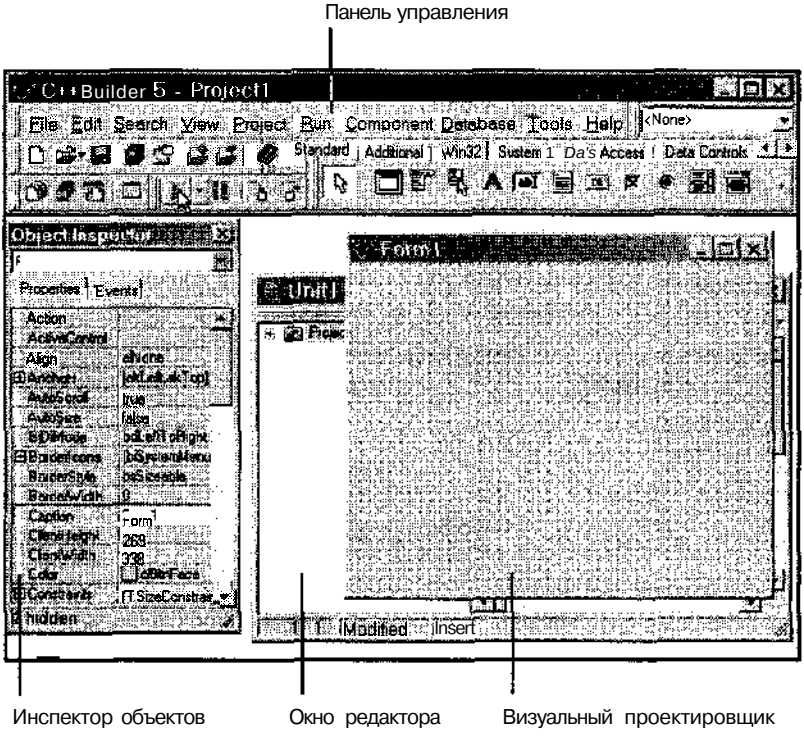


Рис. 4. Основные компоненты IDE

Окно редактора программы, в свою очередь, состоит из двух панелей:

- панели Просмотрщика классов (Class Explorer);
- панели редактора текста программы.

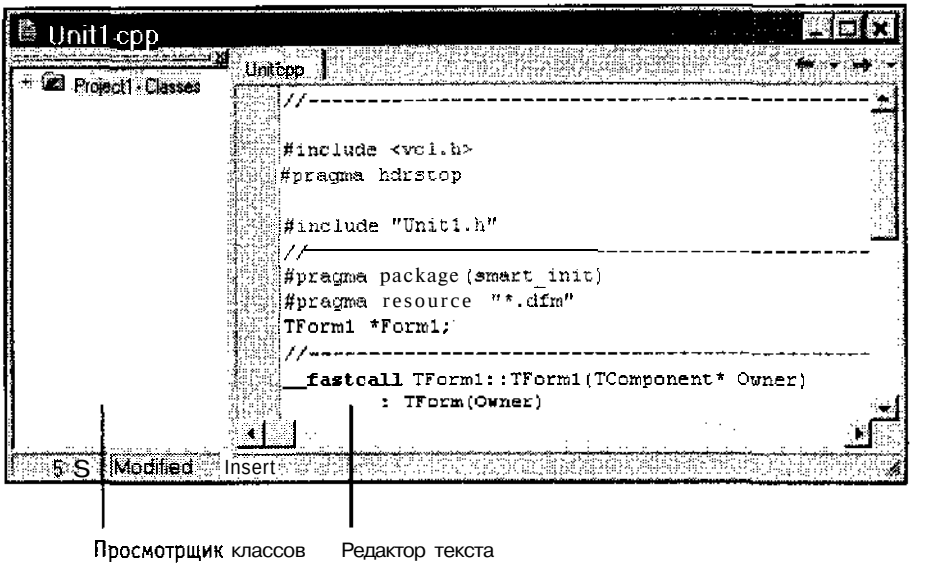


Рис. 5. Окно редактора программы

Традиционно интерфейс всех систем программирования англоязычен. Профессиональные программисты давно к этому привыкли, и попытки перевести ту или иную среду разработки на конкретный язык приводили к тому, что такой локализованный продукт никто не покупал. Поэтому сегодня все без исключения системы, предназначенные для создания программ, имеют интерфейс на английском языке. Прилагаемые к ним электронные и бумажные справочные руководства чаще всего тоже не переводятся. Так что, если вы решили стать настоящим программистом, обязательно научитесь читать технические тексты, написанные на английском языке.

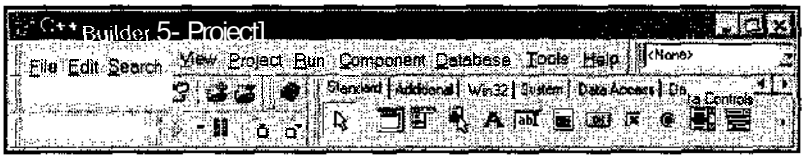
Если некоторое время просто смотреть на экран, ничего не делая, то C++Builder автоматически откроет окно C++Builder Direct. Это запускается специальный модуль системы, который предназначен для связи через Интернет с Web-узлом Inprise (www.inprise.com), быстрого получения информации о новостях компании, очередных версиях Borland C++Builder и доступа к часто задаваемым техническим вопросам по работе с системой. Закройте это окно — пока оно нам не понадобится.

Основные компоненты Borland C++Builder

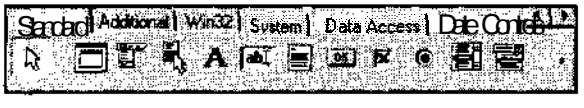
Главное окно C++Builder

Главное окно не разворачивается на весь экран, но его можно перетащить в любое удобное место. Традиционно его располагают в верхней части экрана. В состав главного окна входят:

- строка заголовка;
- строка меню;
- панель инструментов, на которой располагаются командные кнопки для выполнения наиболее часто требуемых действий;



- палитра компонентов, содержащая набор заготовок для элементов управления, из которых будет собираться интерфейс вашей программы. Каждый компонент представлен на палитре своим значком.



Палитра компонентов состоит из четырнадцати панелей, на которых компоненты сгруппированы по конкретным областям применения.

Кратко опишем области, охватываемые компонентами каждой панели:

- Standard — стандартные элементы управления Windows;
- Additional — дополнительные, нестандартные элементы управления Windows;
- Win32 — элементы управления Windows 9x;
- System — системные объекты (таймер, музыкальный проигрыватель и другие);
- Internet — все для приложений, работающих с Интернетом;
- Data Access — компоненты для организации связи с базами данных;
- Data Controls — управляющие элементы для работ с базами данных;

- ADO — компоненты для доступа к данным на основе одной из самых передовых на сегодняшний день Windows-технологии Microsoft ActiveX Data Objects (ADO);
- InterBase — компоненты для работы с СУБД InterBase производства корпорации Inprise;
- Midas — компоненты для создания приложений, способных работать на нескольких компьютерах;
- InternetExpress и Internet — средства быстрого создания приложений для Интернета;
- FastNet — компоненты, поддерживающие основные сетевые протоколы, ориентированные на Интернет;
- Decision Cube — компоненты системы анализа данных;
- QReport — компоненты создания различных отчетов;
- Dialogs — стандартные диалоговые окна Windows;
- Win 3.1 — элементы управления Windows 3.1;
- Samples — примеры компонентов, входящие в поставку системы;
- ActiveX — ActiveX-компоненты (ActiveX — формат активных компонентов, разработанный фирмой Microsoft. Borland C++Builder 4 также поддерживает этот формат);
- Servers — набор компонентов, с помощью которых можно управлять работой офисных программ Word, Excel, PowerPoint, Outlook и др.

Визуальный проектировщик рабочих форм

Что такое форма?

Форма — это окно Windows, в котором размещаются различные элементы управления (кнопки, меню, переключатели, списки, элементы ввода и т. д.). Когда создаваемая программа будет откомпилирована и запущена, форма превратится в обычное окно Windows и станет выполнять те действия, которые для нее определены. Таких окон в программе может быть сколько угодно.

Получить список всех созданных форм можно с помощью комбинации клавиш SHIFT+F12 или нажатием кнопки .

От компонентов формы к элементам управления программы

Элементы управления размещаются на форме путем выбора нужного компонента из палитры и его перетаскивания с помощью мыши. В дальнейшем их можно произвольно перемещать по полю формы и изменять их размеры (то же можно делать и с самими формами). В итоге получается так, что мы как бы составляем интерфейс будущей программы их готовых компонентов. Однако нам необходимо уточнить, что же такое компонент, чтобы при работе с C++ Builder не возникало путаницы.

Компонент — это не конкретный элемент управления на вашей форме, не конкретное окно и не конкретная кнопка. Компонент палитры содержит *обобщенный образ элемента управления*. Так, например, компонент с названием Button содержит обобщенный образ для всевозможных командных кнопок.

Когда мы перетаскиваем компоненты (например кнопки или переключатели) на проектируемую форму, они становятся *экземплярами* (или *объектами)* соответствующего компонента. Таких экземпляров может быть сколь угодно много, и все они будут отличаться друг от друга размерами, местоположением на поле формы и другими *индивидуальными свойствами,* сохраняя при этом *общие свойства* соответствующего компонента.

И только когда мы закончим разработку программы, откомпилируем ее и запустим, наши *экземпляры компонентов* станут *элементами управления* окна программы. Ими можно будет пользоваться в соответствии с теми свойствами, которые им назначил программист.

Компоненты C++Builder не обязательно описывают только элементы управления Windows. Они используются для самых разных целей — для создания соединений Интернета, для работы с базами данных, для воспроизведения музыки и видео. Размещая на форме экземпляр компонента, например NMPOP3, мы получаем возможность обращаться из программы к свойствам этого объекта и принимать электронную почту, то есть без особых усилий можем сделать свою собственную почтовую программу наподобие Microsoft Outlook Express. А используя другие компоненты, можем даже сделать оригинальный Web-обозреватель. При этом объем программирования будет минимальным, если структура будущего приложения тщательно продумана *(спроектирована).*

Главная и дополнительная формы

Пока на экране имеется только одна форма — главная. Она представляет собой главное окно нашей проектируемой программы. На каждой форме можно размещать любые объекты из палитры компонентов.

Число дополнительных форм в программе не ограничено. Чаще всего дополнительные формы используют для размещения элементов управления настройкой программы и для вывода вспомогательной информации. Когда в текстовом редакторе Word мы выбираем пункт меню Сервис • Параметры, на экране появляется диалоговое окно с несколькими панелями, множеством переключателей и различных кнопок. Создать точно такую же форму в C++Builder можно за полчаса, а вот запрограммировать все действия, которые будут происходить при работе с элементами управления этого окна, гораздо сложнее.

Инспектор объектов (Object Inspector)

Инспектор объектов — очень важная часть среды разработки. Он предназначен для задания свойств объектов и определения их реакции на *различные события*.

Свойства объектов

Свойство объекта — это одна из его характеристик, такая, как ширина для кнопки, название для окна, наличие полос прокрутки для списка, цвет и стиль для шрифта, имя файла для рисунка и т. д. Каждый объект имеет большое число свойств, свойства многих элементов управления схожи; свойства других объектов могут сильно различаться

Инспектор объектов позволяет быстро и удобно менять любые свойства текущего (выделенного на форме) объекта. При этом вносимые изменения немедленно сказываются на внешнем виде этого объекта. Например, если мы с помощью Инспектора изменим текст надписи на кнопке, это изменение мгновенно отобразится на самой кнопке в проектируемой форме.

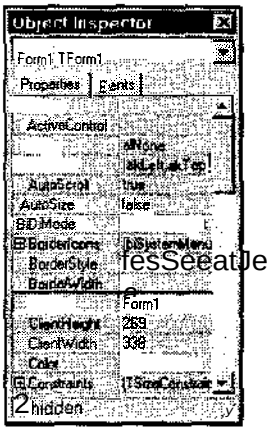
Для быстрого вызова Инспектора объектов можно воспользоваться клавишей F11.

События программные и системные

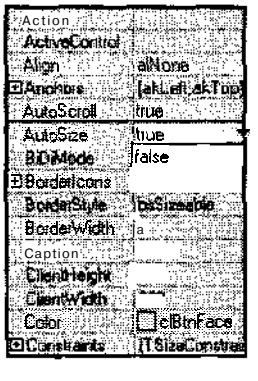
Событие позволяет программе реагировать на любые действия пользователя так, как это задано программистом. К событиям относятся щелчок мышью на кнопке, выбор пункта меню, изменение состояния переключателя и иные происшествия как внутри самой программы, так и в операционной системе Windows. Например, это может быть достижение таймером заданного порога времени, запуск другой программы, исчерпание системных ресурсов и т. п.

Задавая реакции на различные события, мы тем самым определяем всю внутреннюю логику работы программы. Если нужно, чтобы при щелчке на кнопке Button1 появлялось диалоговое окно с надписью Привет!, надо соответствующим образом запрограммировать действия, которые будут выполняться при наступлении события OnClick (ПриЩелчке) для кнопки Button1. Если нам надо воспроизвести музыкальную мелодию, когда в строке ввода **Edit 1** будет набрано слово музыка, необходимо отслеживать вводимую в эту строку информацию с помощью события OnChange (ПриИзменении). Данное событие вызывается всякий раз, когда содержимое поля Edit1 меняется. Для него необходимо определить алгоритм сравнения введенной строки со словом музыка и при совпадении запускать программу воспроизведения музыкального сопровождения.

Инспектор объектов состоит из нескольких частей. В раскрывающемся списке указывается, какой объект на форме выбран (является текущим) в данный момент. Сделать объект текущим можно, либо щелкнув на нем мышью, либо выбрав в этом списке.



В окне Инспектора объектов имеются две вкладки: вкладка свойств выбранного объекта (Properties) и вкладка событий (Events), на которые это объект может реагировать, Каждая из вкладок содержит панель, состоящую из двух колонок. В первой указываются названия свойств (их менять нельзя), во второй — текущие значения соответствующих свойств. Эти значения должны отвечать определенным ограничениям C++Builder. В частности, если, например, свойство — это название объекта, то в нем не должно быть пробелов. В некоторых случаях надо ввести число из небольшого диапазона или выбрать одно из фиксированных значений, заданных в раскрывающемся списке.



Например, мы можем изменить свойство Autosize главной формы, сменив значение false (выключено, по умолчанию) на true (включено). Для этого надо щелкнуть на строке AutoSize в правом столбце и в раскрывающемся списке выбрать значение true.

Если свойство Autosize включено, наша форма будет автоматически подстраиваться под размер размещаемых на ней объектов.

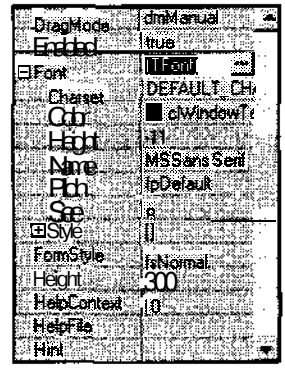
Некоторые свойства меняются простым вводом значения в соответствующей строке. Так, например, мы

можем изменить толщину рамки окна в свойстве BorderWidth, введя вместо нуля число 5.

Все свойства расположены в Инспекторе в алфавитном порядке, и, если какого-то свойства на панели не видно, воспользуйтесь полосой прокрутки.

Некоторые свойства могут иметь вложенную структуру, как, например, свойство Borderlcons. Это обозначается значком «+» слева от названия. Чтобы изменить вложенные значения, дважды щелкните мышкой на названии Borderlcons в левой колонке, и Инспектор объектов откроет доступ к скрытым свойствам. Таким же способом можно свернуть ранее раскрытое свойство BorderWidth — снова дважды щелкнуть на его названии левой кнопкой мыши.

Существует группа свойств, значения которых неудобно менять в Инспекторе объектов. К ним, в частности, относится свойство Font (Шрифт), определяющее параметры шрифта, которым выполняются надписи в текущей форме. Если раскрыть это свойство, как мы раньше поступали со свойством BorderWidth, то можно увидеть, что оно состоит из непонятных характеристик и значений. Так, например, параметр Height (Высота шрифта) равен отрицательному числу −11. Кроме того, в свойстве Font имеется еще одно свойство с вложенной структурой — Style (Стиль шрифта).



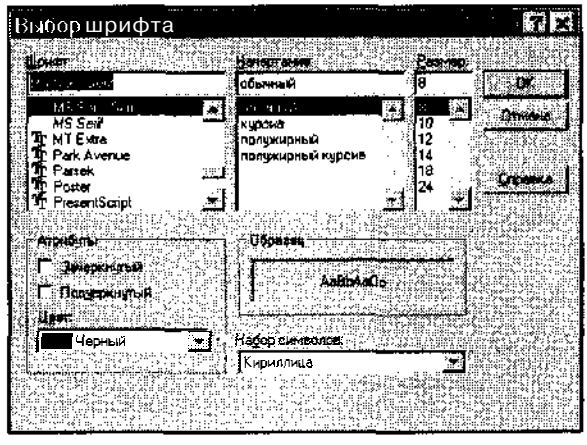
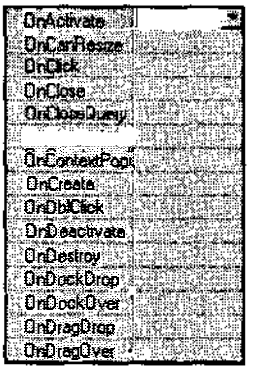


Рис. в. Работа Мастера-построителя для свойства Font (Шрифт)

Менять все это вручную довольно сложно. Легко что-то забыть, что-то не так указать и сделать ошибку. Поэтому для таких свойств, сложных в настройке, созданы специальные *Построители* (это аналоги Мастеров), которые позволяют задавать нужные значения удобным визуальным способом. Наличие такого Построителя для конкретного свойства легко обнаруживается по специальной кнопке на правом краю поля редактирования. При щелчке на этой кнопке вызывается соответствующий Мастер-построитель. Для свойства Font он выглядит как стандартное диалоговое окно Windows для выбора шрифта и его характеристик.



Если вы перейдете на вкладку событий (Events) текущего объекта в Инспекторе, то обнаружите, что никаких значений возможным событиям пока не задано. **Реакции на различные события нам придется задавать самостоятельно — в этом и заключается процесс программирования.**

Редактор программы

За главной формой скрыто окно редактора программы (см. рис. 4 и 5). Между текущей формой и редактором можно переключаться с помощью клавиши F12. Окно редактора, как уже говорилось, состоит из двух панелей: панели Просмотрщика классов и панели редактора исходного текста программы на Си++. Просмотрщик классов визуально отображает структуру связей между различными объектами нашей программы и позволяет быстро перемещаться по ее тексту. Пока он нам не понадобится. Закройте его щелчком на небольшой кнопке в правом верхнем углу панели.

Редактор исходного текста

Сейчас мы находимся в окне редактора исходного текста своей программы (см. рис. 7). Хотя мы еще ничего не знаем о языке программирования Си++, тем не менее в окне уже имеется довольно большой объем программного кода. Дело в том, что C++Builder как среда быстрой разработки приложений берет на себя львиную долю работы по ручному кодированию программы. Когда-то программистам приходилось набирать килобайты текстов только чтобы написать простое приложение, выводящее на экран строку «Привет всем!» А уж если требовалось расположить в окне различные элементы управления и организовать их эффективную работу, программирование превращалось в сущий кошмар — *95%* усилий разработчика уходило на кодирование второстепенных моментов работы готовящегося приложения.

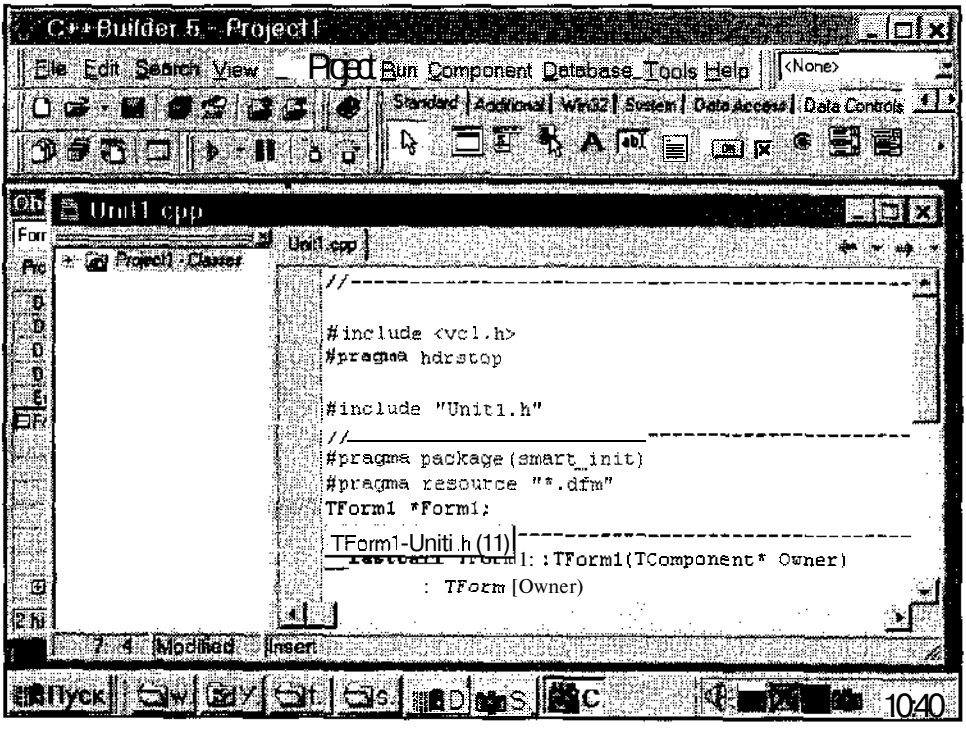


Рис. 7. Окно редактора программы

Программисты для Windows запоминали множество внутренних вызовов этой операционной системы и писали сотни строк кода, чтобы заставить программу выполнить подчас самое простое действие.

К счастью, современные средства разработки выполняют за программиста всю рутинную работу. Чтобы написать свое приложение, нам совершенно не требуется разбираться в коде, сгенерированном C++Builder. Только запомните — менять этот код ни в коем случае нельзя!

Никогда не меняйте исходные тексты Си++, которые были созданы автоматически. Это может привести к полной неработоспособности вашей программы.

Если поводить указателем мыши над готовым текстом, то над некоторыми словами (в частности, над TForm 1) может появиться всплывающая подсказка. В данном случае она указывает, в какой строке (11) какого файла (Unit1.h) описано понятие TForm1.

Пока подсказку лучше отключить. Она довольно медленно вызывается даже на очень мощных компьютерах. Нажмите правую кнопку мыши над текстом программы, выберите в контекстном меню пункт Properties (Свойства), в открывшемся диалоговом окне перейдите на вкладку Code Insight (Анализ кода), сбросьте флажок Tooltip symbol insight (Сведения об идентификаторе) и щелкните на кнопке OK.

Работает редактор исходных текстов примерно так же, как и большинство его собратьев. Если вы знакомы с программой Word или Блокнотом, то никаких проблем в наборе текстов на Си++ у вас не возникнет.

2. Быстрый старт

Учимся работать с визуальными компонентами

Если вам не терпится поскорее создать собственную программу, сейчас мы этим займемся. Ведь выше говорилось, что с помощью Borland C++Builder это совсем просто, и никакими особыми знаниями Windows обладать не надо. Это действительно так. Мы сложим свою первую программу из готовых кирпичиков примерно так, как складывают самые сложные машины из простых кубиков конструктора «Лего». В Borland C++Builder эти кирпичики называют компонентами.

Компонент — это строительный кирпичик вашей программы. Компоненты могут быть *визуальными* и *невизуальными.*

Визуальные компоненты

Визуальный компонент — это элемент управления (например кнопка или список), С которым пользователь нашей программы будет взаимодействовать во время ее работы. Визуальные компоненты, в свою очередь, делятся на объекты Windows (обычные элементы управления) и графические объекты-контейнеры, с помощью которых объекты Windows можно объединять в группы (всевозможные панели и вкладки). Такое объединение обычно приводит к экономии ресурсов операционной системы.

Невизуальные компоненты

С *невизуальными компонентами* во время работы программы пользователь взаимодействовать не может. Они предназначены для программирования вещей, не относящихся к видимому на экране интерфейсу, например для управления встроенными часами программы (обычно называемыми *таймером),* для подключения к Интернету и т. д. Кроме того, невизуальные

компоненты иногда служат для создания сложных элементов управления, например меню, структуру которого удобнее готовить с помощью специального Мастера, содержащегося в компоненте Меню.

Делаем валютный калькулятор

Тему нашей первой программы мы долго искать не будем. Поскольку этот проект чисто учебный, нам все равно, что создавать. Давайте попробуем написать программу, выполняющую роль валютного калькулятора. В него вводят сумму в долларах США и текущий курс ММВБ, а результат получают в рублях. Если эта задача выглядит не слишком актуальной, то подумайте о том, сколько замечательных программ вы создадите потом, когда освоите основы программирования.

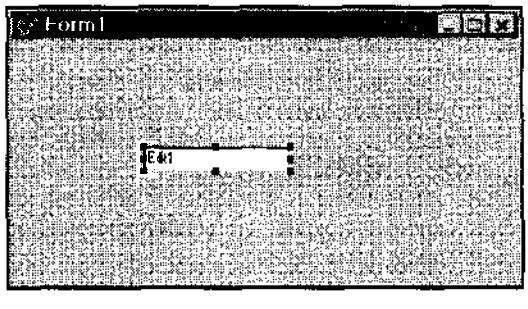
Не слишком ли трудно для начала, скажете вы? Совсем нет. Тем-то и хорош Borland C++Builder, что позволяет в самые короткие сроки создавать готовые программные продукты высокого качества, обходясь минимумом знаний и навыков.

Наш калькулятор будет устроен так: в рабочем окне программы присутствуют три поля и одна командная кнопка. В первое поле вводят денежную сумму, во второе — курс пересчета, а в третьем поле программа выдает результат вычислений после нажатия на командную кнопку.

Компонент Edit (Поле ввода)

Переключитесь из редактора текста в дизайнер форм (например с помощью клавиши F12). Теперь выберите с помощью мыши на палитре компонентов (вкладка Standard) компонент, который представляет элемент управления, называемый полем ввода. По-английски этот компонент называется Edit.

Переместите указатель мыши на форму и один раз щелкните левой кнопкой. На форме появится поле ввода — оно получит название Edit1.



Обратите внимание на то, что этот элемент выделен черной рамкой, на которой расположены восемь квадратных маркеров. Подведите указатель мыши к одному из этих маркеров, и указатель изменит свой вид.

Это означает, что теперь мы при желании можем изменить раз-

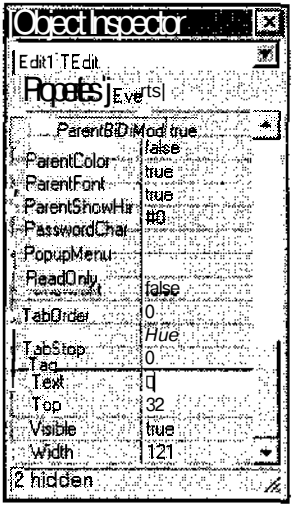
мер проектируемого элемента управления. Нажмите на левую кнопку мыши и, не отпуская ее, потяните рамку в сторону. Размер элемента сразу изменится. Этот прием называется *протягиванием.*

Сам элемент тоже можно перемещать по форме в любое место с помощью мыши. Этот прием называется *перетаскиванием.* Так, пользуясь протягиванием и перетаскиванием, мы можем менять местоположение и внешний вид любых визуальных компонентов на рабочей форме.

Форма, на которой мы размещаем компоненты, покрыта мелкой сеткой. При установке компонентов происходит автоматическое притягивание к ее узлам. Такой подход к проектированию внешнего вида очень удобен, а расстояние между линиями сетки нетрудно изменить, о чем мы расскажем чуть позже.

Придайте полю ввода желаемый размер и перетащите его в левую часть формы. Заметьте, что внутри поля уже имеется некий начальный текст. Он совпадает с автоматически сгенерированным названием элемента управления — Edit1. Если мы создадим в форме еще одно поле ввода, оно получит название Edit2 и так далее.

Чтобы стереть показываемый в поле ввода начальный текст и ввести новый, не спешите щелкать мышью по полю ввода. Ничего не получится, поскольку это еще не *элемент управления,* а только его *образ.* Поле ввода станет элементом управления, когда мы откомпилируем и запустим нашу программу, а пока редактировать его можно только изменением свойств в Инспекторе объектов.



1, Выделите на форме элемент **Edit 1.**

2. В Инспекторе объектов раскройте вкладку Properties (Свойства).

3. Разыщите свойство Text (Текст). Если надо, воспользуйтесь полосой прокрутки. Если это свойство найти не удается, проверьте, не забыли ли вы выделить элемент на форме. В Инспекторе всегда отображаются свойства только того объекта, который в данный момент выделен.

4, В поле свойства Text (Текст) сотрите значение **Edit 1** и введите число 0 (поскольку мы делаем калькулятор, то нам надо, чтобы в качестве начального значения там был не текст, а число).

Свойство Text — одно из главных свойств текстовых элементов управления. В нем хранится введенная в элемент информация.

Не путайте название элемента Editl (свойство Name в Инспекторе объектов) с его содержимым (свойство Text). Мы изменили не название элемента, а то, что в нем содержится в данный момент.

А что, если мы захотим изменить и название элемента управления? Это очень просто. Выберите в Инспекторе объектов пункт Name (Имя) и введите вместо Editl новое название — Dollars. При этом C++Builder автоматически внесет все изменения названий и начальных значений в исходный текст.

Свойство Name — неотъемлемое свойство каждого компонента C++Builder. В этом свойстве указывается конкретное имя, которое будет использоваться в вашей программе для обращения к данному объекту. Старайтесь давать объектам осмысленные имена, чтобы по их названиям сразу понимать, для чего они предназначены.

Теперь точно таким же способом создайте второе поле ввода, назовите его, например, Rate (Курс обмена), а в качестве начального значения опять-таки укажите 0. При этом размер поля можно уменьшить.

Компонент Label (Поле надписи)

А где же мы будем отображать конечный результат наших вычислений? Известное нам поле ввода по определению не подходит — оно служит для ввода данных. Нам нужно поле другого типа, которое можно использовать для пассивного отображения информации, взятой из программы, и которое не допускает ввода пользователем изменений в его содержимое. Для этих целей служит элемент управления типа *поле надписи* (Label) — его компонент мы возьмем в палитре компонентов.

У нового элемента Label1 свойства Text нет, так как вводить в надпись данные нельзя. А как же менять текст в этом объекте? Для этого служит свойство Caption (Заголовок). Этим свойством обладают многие визуальные компоненты C++Builder.

Выберите в Инспекторе объектов для поля Label1 свойство Caption и сотрите его значение. При этом само поле сожмется в узкую полоску. Это связано с тем, что у поля Label1 активизировано свойство AutoSize (автоматическая подстройка размера элемента под его содержимое). Чтобы отключить автоподстройку размера, выберите в Инспекторе объектов свойство AutoSize, щелкните на кнопке раскрывающегося списка и измените исходное значение True на False.

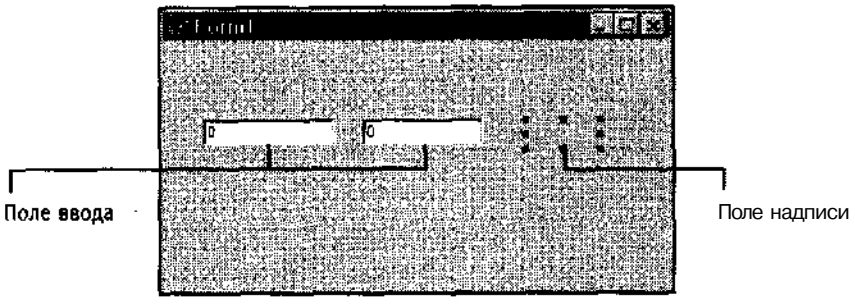


Рис. 8. Форма с первыми тремя элементами управления

Два английских слова True и False очень активно используются и в языке Си++, и в системе визуального программирования C++Builder. Значение True — по-английски «истина», «да», «включено». Значение False — соответственно «ложь», «нет», «выключено».

Используя квадратные маркеры, окружающие поле надписи (см, рис. 8), задайте полю желаемый размер методом протягивания. В это поле и будет выводиться итоговая информация, полученная в результате вычислений.

Компонент Button (Командная кнопка)

Теперь нам надо разместить на форме кнопку, при нажатии на которую и будет работать калькулятор. Выберите компонент Button на палитре компонентов, разместите его на форме перед полем надписи Label1, назовите кнопку, к примеру, TotalButton (в свойстве Name), а в качестве заголовка на кнопке укажите «=» (в свойстве Caption).

Итак, основные детали нашего калькулятора спроектированы. При этом мы еще не написали ни строчки кода, а программа почти готова к работе. В принципе, ее можно скомпилировать и запустить уже сейчас, только пока она ничего делать не будет — ведь мы не определили, какие действия должны выполняться при нажатии на кнопку TotalButton.

Сохраняем проект

Проект — это набор всех форм, файлов с текстами программы и всевозможных настроек. И чтобы наши труды не пропали даром, его надо сохранить в специальном файле.

Наверное, вам встречалась ситуация, когда из-за сбоев питания вся работа терялась. Особенно бывает обидно, если потерян текст сложной программы,

восстановить который довольно трудно. Кроме того, при разработке программ операционная система обычно функционирует в стрессовом режиме и может «зависать» чаще обычного, поэтому сохраняться лучше регулярно.

Есть «золотое» правило — сохранять работу через такие минимальные периоды времени, за которые ее жалко потерять. Жалко потерять часовой труд — сохраняйтесь не реже, чем раз в час. Жалко потерять десятиминутный труд — сохраняйтесь раз в десять минут.

Для сохранения всего проекта можно использовать кнопку панели инструментов Save All (Сохранить все). Система выдаст запрос, куда сохранить файл со сгенерированным текстом программы (все программы на Си++ имеют расширение .CPP). По умолчанию главный модуль программы получит имя Unit1.

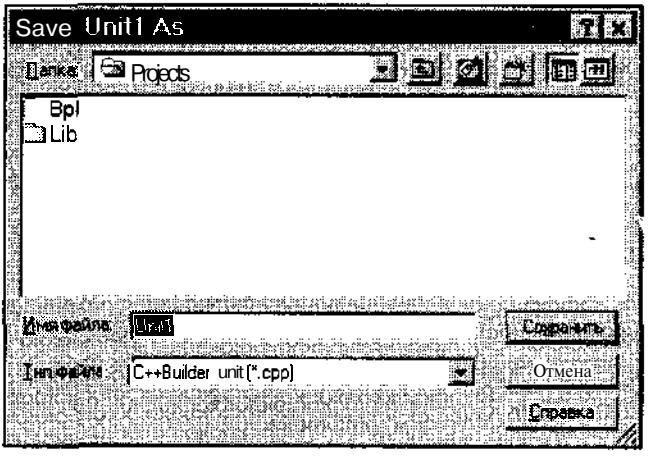


Рис. 9. Сохранение рабочего проекта

Измените это имя на Calc, выберите папку и сохраните файл. Затем система выдаст запрос о том, куда сохранить главный файл проекта (расширение .BPR), содержащий полную информацию о всех настройках, расположении окон на экране и о многих других вещах. Задайте проекту имя CalcProject и сохраните его в той же папке.

Начинаем программировать

Теперь все готово к заключительному этапу создания нашей первой программы — ручному вводу кода на языке Си++, который будет производить необходимые вычисления. В нашем случае это перемножение двух чисел: числа долларов и текущего курса.

От автора.

Очень прошу всех, кто пытается научиться программировать по моей книжке, — не забегайте вперед, не спешите, не набирайте тексты программ механически! Не понимая смысла каждого вводимого оператора Си++, научиться создавать даже самые простые программы не удастся — это все равно, что пытаться выучить иностранный язык, просто повторяя, например, английские слова, не зная их перевода.

Дважды щелкните мышью на командной кнопке TotalButton. Система автоматически сгенерирует программный код, который будет вызываться, когда пользователь нажмет на эту кнопку, а вы сразу попадете в редактор текста программы, причем именно в то место, куда надо вписать реакцию программы на нажатие кнопки TotalButton.

Не обращайте внимания на кажущуюся непонятность автоматически созданного текста Си++. Ниже мы убедимся, что ничего сложного в нем нет.

Текст надо вписывать между двумя фигурными скобками {...}, где исходно расположен текстовый курсор. Эти скобки определяют смысловые границы, внутри которых должен располагаться наш программный код. Ни в коем случае нельзя выходить за пределы этих границ, иначе компилятор сообщит об ошибке.

Итак, наша задача состоит в том, чтобы получить два числа из объектов Dollars и Rate, перемножить их и показать результат в поле надписи Label1. Для этого сначала возьмем число из поля ввода Dollars и где-нибудь временно его сохраним. Но где?!!

Переменные

Любая информация в программе хранится в *переменных*. Переменные могут хранить как простые данные: числа, символы, строки, — так и сложные структуры, например, домашний адрес с четырьмя составляющими (название города и улицы, номер дома и квартиры) или даже различные объекты.

Тип переменной

Каждая переменная должна иметь тин, определяющий, какого рода информация в ней хранится. Типом переменной может быть число, строка и всевозможные сложные структуры.

Как создаются переменные?

Перед тем как переменную использовать, ее следует *создать,* иначе говоря *определить* (иногда программисты еще говорят *объявить* или *декларировать).* То есть, переменную надо явно описать в тексте программы, указав при этом ее название и тип. Когда программа будет откомпилирована и запущена, для каждой переменной в памяти компьютера будет выделено специальное место. Для чисел это может быть четыре или восемь байтов, а для строк — заданная вами длина.

Если переменная не была описана до ее первого использования, например, для записи в нее конкретного числа, то компилятор при трансляции исходного текста выдаст сообщение об ошибке: Undefined symbol <имя переменной>. Это означает, что переменная с таким-то именем не определена.

Перед тем, как использовать переменную, ее необходимо описать — указать программе, что вы будете использовать переменную с выбранным вами именем и типом.

Определять переменные лучше всего в начале текущего логического блока внутри фигурных скобок {...} — сразу за первой скобкой. Хотя это и не обязательно, но желательно, поскольку позволяет быстро найти нужное определение в тексте и выяснить, какой тип имеет та или иная переменная (ведь их может быть много).

В языке Си++ все имена переменных могут состоять из произвольных комбинаций строчных и прописных английских букв, цифр и символа «_». Однако имена переменных не должны начинаться с цифры.

Допустимые имена переменных в Си++.

```
Cat
ListBox
Windows99
ge33iv5HsY8Ys
i100
```

Недопустимые имена переменных в Си++:

кошка (буквы не английского алфавита);

`football game` (между словами имеется пробел);

1c (имя переменной начинается с цифры).

Давайте определим свою первую переменную в программе. Определение переменных в Си++ записывается так:

```
тип  список-определяемых-переменных-через-запятую;
```

В Си++ в конце каждой логически законченной части текста обязательно ставится точка с запятой — «;».

Вводимая вами переменная должна хранить число. Назовем ее DollarsNum. В Си++ есть готовый тип «число», который обозначается словом int (сокращение от английского слова *integer* — целое число). Поэтому сразу за фигурной скобкой нам надо написать так:

```
{int  DollarsNum;
```

Зарезервированные слова Си++

Программа на языке программирования Си++ пишется с помощью набора английских слов и специальных символов. Каждое слово имеет определенный конкретный смысл. Некоторые из слов зарезервированы, то есть их нельзя применять для собственных надобностей, например, чтобы назвать таким же именем свою собственную переменную. Зарезервировано, в частности, уже использованное нами слово int.

В языке Си++ различаются строчные и заглавные буквы. То есть, слова int и Int не одно и то же, и если int как ключевое слово для своих целей использовать нельзя, то Int — можно без проблем.

В редакторе текста программы Borland C++Builder зарезервированные слова по ходу набора исходного текста выделяются специальным шрифтом или цветом, чтобы мы не спутали их с переменными и сразу могли обнаружить ошибку ввода. Например, если случайно ввести не `int  DollarsNum;` а `imt DollarsNum;`, то imt не выделится полужирным шрифтом (такая установка на выделение элементов текста программы принята в C++Builder по умолчанию, и при желании ее можно перенастроить), и мы сразу поймем, что ввели неправильное слово.

Смысл каждого зарезервированного слова исходно заложен в стандарте языка Си++ и изменять его не разрешается.

Помимо ключевых слов, в C++Builder используются библиотеки готовых компонентов. Для каждого компонента определен свой тип, например, все поля ввода (Edit) имеют тип TEdit. Кроме того, в автоматически созданной с помощью C++Builder программе обычно уже описано множество различ-

ных вспомогательных объетов и типов, например, тип главной формы TForm1. Такие названия, уже определенные программой или вами самими, повторно для своих целей использовать нельзя.

Поэтому в качестве названия переменной для хранения числа долларов было выбрано название DollarsNum, не совпадающее с названием уже используемого элемента управления Dollars. Конечно, вместо DollarsNum мы могли бы выбрать любое другое название, например number_of_dollars или просто dollars, написав это имя переменной со строчной буквы в отличие от имени элемента управления.

При наборе текстов программ разработчики обычно придерживаются множества негласных соглашений, своего рода стиля написания, чтобы потом в этом тексте можно было легко разобраться как самому ее создателю, так и постороннему человеку, если он захочет эту программу улучшить или изменить для своих задач. В частности, в C++Builder при выборе названий типов сложных объектов принято в качестве первой буквы использовать прописную букву «T» (от слова Type, Тип). В других системах программирования могут использоваться другие первые буквы. Например, в Microsoft Visual C++ название типа принято начинать с буквы «C» (Class, Класс).

Порядок определения переменных

Переменную DollarsNum мы определили. Теперь надо определить переменную RateValue, в которой во время работы программы будет храниться текущий курс обмена — сколько рублей приходится на один доллар. Это можно сделать либо описав ее на следующей строчке:

```
int DollarsNum;
int RateValue;
```

либо перечислив обе переменные через запятую:

```
int DollarsNum, RateValue;
```

При этом и DollarsNum, и RateValue будут иметь один и тот же тип int.

Порядок записи описаний переменных несуществен. Можно написать так:

```
int RateValue; int DollarsNum;
```

или так:

```
int RateValue, DollarsNum;
```

Самое главное — чтобы переменная была описана в тексте программы до ее первого использования.

Комментарии

Когда программа становится все больше и больше, запоминать, что делается в той или иной ее части, становится все сложнее и сложнее. Через месяц можно полностью забыть, что мы напрограммировали в каком-то проекте, а уж постороннему человеку разобраться в чужом тексте, даже аккуратно написанном, крайне сложно. Поэтому профессиональные программисты очень подробно комментируют свои тексты.

В комментариях можно писать все, что угодно, используя и русский язык, и любые символы. Компилятор не расценит это как ошибку — он поймет, что данный текст относится к комментарию и при анализе программы просто его пропустит.

Текст комментария заключается как в скобки в символьные пары /* — начало комментария, и */ — конец комментария. Между началом и концом текста такого комментария может быть сколько угодно пустых строк или строк с примечаниями и заметками.

Примеры:

```
/*
Эти строки
являются комментарием.
*/
/* Этот комментарий поместился на одной строке */
```

В последнем случае, когда весь остаток текущей строки отводится под комментарий, нет нужды в закрывающей парной «скобке». В этом случае вместо парных скобок можно один раз использовать пару символов //. При этом весь остаток строки за // будет рассматриваться как комментарий.

Попробуйте добавить к определению переменной DollarsNum комментарий:

```
int DollarsNum; // переменная для хранения числа долларов
```

Обратите внимание на то, что в редакторе комментарий выделяется синим цветом и наклонным шрифтом, чтобы разработчик мог сразу определить, что это не рабочий текст программы.

Комментируйте свои программы как можно более подробно. Только комментарии должны быть осмысленными. Не надо писать:

```
int DollarsNum; // здесь описана переменная DollarsNum
```

— это и так ясно. Указывайте, что реально делается в конкретных местах программы. Не пишите:

```
/* здесь перемножаются значения двух переменных DollarsNum
и RateValue */
```

Лучше отметьте:

```
/* доллары переводятся в рубли */
```

Теперь определите третью переменную — Result, в которой будет запоминаться результат умножения двух введенных в программу чисел:

```
int Result;
```

Определяемые имена переменных не должны совпадать с ключевыми словами Си++, с типами и названиями переменных и других объектов из стандартных библиотек, а также с ранее определенными переменными, чтобы в тексте программы не возникала путаница. Если переменную назвать TEdit. то компилятор не поймет, где надо использовать TEdit как переменную, а где — как название компонента.

Нельзя написать:

```
int DollarsNum;
int Result, DollarsNum, RateValue;
```

Компилятор выдаст сообщение об ошибке: Multiple declaration for <имя переменной> (Повторное определение переменной с таким-то именем), что напомнит о том, что переменная DollarsNum определена два раза.

Как получить строку из поля ввода?

Где хранить вводимые пользователем числа и их произведение, мы определили — в переменных. Теперь надо решить, как получить эти числа из полей ввода DollarsNum и RateValue.

Все элементы управления Windows и все компоненты C++Builder имеют свои типы. Имеют свой тип и поля DollarsNum и RateValue, и к ним можно обращаться, как к обычным переменным. Только их тип значительно более сложен, чем, например, int, — он состоит из нескольких объектов других типов.

В частности, в состав типа TEdit входят: числовые переменные, описывающие размер поля ввода; текстовые строки, хранящие название поля и его содержимое, и многое другое. Такие сложные типы в Си++ называются *классами*.

Класс — фундаментальный термин программирования и языка Си++. Любое понятие Windows и большинство понятий окружающего нас мира можно представить в виде класса Си++. Например, понятия «окно программы», «кнопка», «переключатель» и множество других описываются с помощью классов.

Каждый класс имеет свое название. Например, класс Кнопка имеет название TButton. Для большинства объектов Windows в C++Builder имеются стандартные классы со своими названиями. При решении конкретной задачи можно создавать собственные нестандартные классы, используя в качестве их названий любые допустимые сочетания символов.

Название класса — это тип, с помощью которого мы можем определять переменные в программе. Так, мы можем использовать готовые классы, например TButton, и описать новую кнопку:

```
TButton my_button;
```

(для большинства элементов управления C++Builder делает это автоматически). Можем мы и специальным образом определить новый класс (как это сделать, будет рассказано позже) и потом использовать его для собственных нужд.

Класс — это не конкретный объект программы, не конкретное поле или кнопка. Понятие (или класс) TEdit описывает не поле Dollars, которое мы создали в главной форме, а содержит в себе обобщенный образ редактируемого поля ввода — у него есть координаты на экране, размер, оно имеет название и может хранить введенную пользователем строку. Класс во многом похож на компонент палитры, только используется он не для проектирования формы, а для создания программы.

А что же тогда такое поле Dollars? Поле Dollars будет экземпляром (или объектом) класса TEdit в программе (и экземпляром компонента Edit на форме). Таких экземпляров может быть сколько угодно, и все они будут отличаться друг от друга размерами, местоположением на форме и другими свойствами, сохранив при этом общие черты класса TEdit.

Свойства и возможности каждого компонента C++Builder описываются соответствующим классом Си++.

Не путайте названия компонентов, принятые в визуальном проектировщике форм C++Builder (Edit Label Button и другие), с названиями классов. Названия компонентов к непосредственному программированию никакого отношения не имеют и используются только в процессе проек-

тирования форм. Эти названия вы можете свободно использовать в своих программах в качестве переменных, например так:

```
int Edit Label, Button;
```

А написать

```
int TEdit;
```

или

```
int TLabel;
```

нельзя.

Как уже говорилось, вводимая в окна с типом TEdit информация хранится в свойстве Text. Однако эта информация имеет текстовый тип, то есть, если в окно DollarsNum будет введено число 123, в свойстве Text оно будет храниться не как число, а как текстовая строка.

Свойства каждого класса — это обычные переменные, имеющие свой тип. У разных классов имеются свои наборы переменных, которые как бы скрыты, спрятаны внутри этих классов. К одним из них можно получить доступ, к другим — нет.

Как же так? — спросите вы. Ведь выше говорилось, что переменные не могут иметь одинаковые имена, а названия свойств компонентов (а эти свойства, оказывается, — переменные) постоянно повторяются. И в Dollars, и в Rate, и в TotalButtom, в частности, есть свойство Name (Имя).

Совершенно верно. Дело в том, что свойства — это переменные, принадлежащие конкретному классу. Они расположены внутри этого класса, из них данный класс *составлен,* и получить к ним доступ можно, только *явно* указав конкретный экземпляр класса, к которому данная переменная принадлежит. Например, вы можете определить в программе свою переменную Name и свободно ее использовать. Никакой ошибки при этом не возникнет, потому что доступ к свойству Name окна Dollars будет записываться специальным образом — сначала указывается имя объекта (не типа, а экземпляра класса!), обладающего данным свойством. В нашем случае это поле ввода Dollars. Затем записывается комбинация двух символов «—>», за которыми приводится конкретной свойство объекта — в нашем случае, Name. Получается так:

```
Dollars->Name
```

Чтобы обратиться к содержимому поля ввода Rate, надо соответственно написать

```
Rate->Text
```

Аналогично происходит обращение к любым свойствам других объектов. Например, доступ к свойству Caption объекта Label1 возможен с помощью конструкции

```
Label1->Caption
```

Когда вы научитесь создавать собственные классы (а это совсем несложно), то при желании тоже сможете использовать одинаковые названия для своих переменных в разных классах, хотя это не всегда удобно.

Свойство Name класса TEdit (или компонента Edit), как говорилось выше, текстовое. Чтобы преобразовать веденную пользователем и хранимую в этом свойстве строку, например, «100», в число 100, надо воспользоваться одной из возможностей стандартной библиотеки C++Builder.

Стандартные функции C++Builder

В стандартных библиотеках (а их насчитываются десятки) хранятся *стандартные функции*. Стандартная функция — это небольшая программа, которая используется для выполнения часто встречающегося действия. В нашем случае, это будет программа преобразования строки в число.

Когда мы записываем в тексте программы обращение к стандартной функции, никакого запуска программы для выполнения преобразования в этот момент, конечно, не происходит. До этого программу надо будет подготовить (на основе вашего текста) с помощью компилятора, который сам определит, где происходит вызов стандартных функций.

Функция преобразования строки в число называется StrToIntO *{StrToInt* - это сокращение от английских слов *String To Integer).* Обратите внимание на использование строчных и заглавных букв. Каждая стандартная функция имеет свой тип, точно также, как и переменная. Тип функции StrToInt(), как вы наверняка догадались, — int (целое число).

Тип функции

Что значит *тип функции?* С переменными ясно — они хранят конкретную информацию. А функция? Функция, конечно, ничего не хранит. Она выполняет определенные действия и получает конечный результат (в нашем случае она берет строку, анализирует ее содержимое и преобразовывает его в число). Тип функции — это тип ее результата. В таких случаях говорят, что функция *возвращает* значение конкретного типа. То есть StrToIntO возвращает значение типа int.

А как функция получит строку текста для анализа? В Си++ принято, что значения, передаваемые любой функции (их называют *параметры),* запи-

сываются сразу после названия функции в круглых скобках. Если этих значений несколько, они перечисляются через запятую. Каждый параметр имеет свой тип. Так, единственный параметр функции StrToInt должен быть текстовым. Если вы попробуете передать ей в качестве параметра число, то при работе компилятора вам будет выдано сообщение об ошибке.

В нашем примере функции StrToInt сначала надо передать функции StrToInt текстовую строку из поля ввода Dollars. Доступ к содержимому этого поля, как объяснялось выше, записывается конструкцией **Dollars->Text**, значит, вызов StrToInt будет выглядеть так:

```
StrToInt( Dollars->Text )
```

Сохраняем значение в переменной

Допустим, функция **StrToInt()** преобразовала строчку из поля Dollars в число и вернула это значение в программу. Как же им воспользоваться?

Для сохранения значения математического выражения или результата, возвращаемого функцией, используют *оператор присваивания*.

Все команды, которые мы отдаем компьютеру, записывая их на языке Си++, называются операторами. Каждая команда подразумевает выполнение определенного действия (поэтому, в частности, описание переменных нельзя называть оператором).

Действие оператора присваивания заключается в том, что выбранной нами переменной присваивается новое значение. Старое значение, хранившееся в этой переменной, пропадает.

Записывается это так:

```
переменная = значение;
```

Например,

```
I = 10;
```

Переменная I, конечно, должна быть предварительно описана:

```
int I;
```

В нашем случае в переменной **DollarsNum** надо сохранить значение, возвращаемое функцией StrToInt(). В тексте программы, сразу после объявления переменных, это надо записать так:

```
DollarsNum = StrToInt( Dollars->Text );
```

В переменной RateValue сохраним число из второго поля ввода:

```
RateValue = StrToInt( Rate->Text );
```

Теперь надо вычислить произведение полученных чисел. В этом опять поможет оператор присваивания. Для формирования присваиваемого переменной значения можно использовать не только функцию, а любое допустимое в Си++ математическое выражение, записываемое с помощью знаков четырех математических действий:

- «+» (сложить);
- «-» (вычесть);
- «*» (умножить);
- «/» (разделить);

а также круглых скобок ().

Эти действия могут выполняться не только над явно числами, но и над значениями, хранящимися в переменных на момент выполнения данного оператора. В нашем примере произведение значений переменных DollarsNum и RateValue будет записываться в переменную Result с помощью такого оператора:

```
Result = DollarsNum * RateValue;
```

Примеры допустимых операторов присваивания Си++:

```
x = (y + z) / 2 + 10;
H_123 = a*a + Y2000( d1,12 ) - 32000;
```

В последнем примере Y2000(d 1,12) — не переменная, а функция с двумя параметрами.

Правила записи операторов Си++

При записи операторов Си++ можно использовать пробелы любым удобным вам способом. Их применение не допускается только внутри различных названий и чисел. Например, нельзя написать Rate Value, Str To Intили 1 000 — компилятор воспримет такую запись как несколько отдельных слов или чисел. Нельзя также разделить «-» и «>» в выражении Dollars->Text, написав

```
Dollars-  >Text
```

потому что «->» — это используемая в выражении цельная конструкция *{операция)* языка Си++, такая же, как «=» (присваивание) или «–» (вычи-

тание). Она просто *обозначается* двумя символами, но в смысловом плане неделима. В Си++ есть еще несколько подобных операций, записываемых двумя или даже тремя символами, например, «>>=». **Вставлять пробелы внутрь их нельзя.**

Можно написать:

```
Dollars -> Text
```

или

```
Dollars-> Text
```

или

```
Dollars      ->Text
```

или даже

```
      Dollars
   ->
Text
```

с пустыми строчками. Здесь пробелы вставляются между логически законченными частями текста программы (переменной и операцией).

Пробелы во многих случаях можно и пропускать. Например, оператор

```
Result = DollarsNum * RateValue;
```

можно записать и таким способом:

```
Result=DollarsNum*RateValue;
```

Более того, можно в одной строке записать несколько операторов и описаний переменных, не разделяя их пробелами (только точками с запятой):

```
int  Result;DollarsNum=StrToInt(Dollars->Text);
```

Конечно, необходимо отделять пробелом названия переменных и ключевые слова. Нельзя написать:

```
intResult;
```

Компилятор не поймет, что вы имеете в виду, и не выделит ключевое слово int полужирным шрифтом.

Экономить на пробелах никогда не надо. Это очень затрудняет понимание текста программы и служит дополнительным источником ошибок.

Активно применяйте пробелы для повышения наглядности и удобочитаемости своих программ.

Вывод результата на экран

Итак, произведение двух чисел нами получено и успешно сохранено в переменной Result. Это произведение (очевидно, что это тоже какое-то число), надо показать в поле надписи Label!. Как говорилось выше, у такого поля нет свойства Text, но есть свойство Caption, тоже текстового типа. Когда наша профамма будет запущена, изменение значения этого свойства приведет к немедленному изменению текста, отображаемого в поле Label 1.

Как изменить значение Caption? С помощью того же оператора присваивания. Ведь свойство — это переменная, значит, ему можно передавать любое соответствующее его типу значение. Однако здесь возникает другая проблема. Б переменной Result хранится число, а записывать в Caption надо строку текста. Как преобразовать число в текст? Для этого есть стандартная функция IntToStr(), которая в качестве параметра получает число, а возвращает строку.

Тогда заключительный оператор нашей первой программы запишется так:

```
Label1->Caption = IntToStr( Result );
```

а вся программа займет шесть или семь (в зависимости от того, насколько компактно вы определяли переменные) строчек кода:

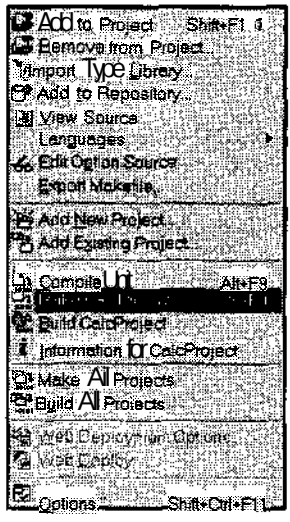
```
{
int DollarsNum, RateValue;
int Result;
DollarsNum = StrToInt( Dollars->Text );
RateValue = StrToInt( Rate->Text );
Result = DollarsNum + RateValue;
Label1->Caption = IntToStrf Result );
}
```

Си++ допускает запись в одной строке нескольких выражений, однако это сильно нарушает наглядность программы. Хорошим стилем считается размещение каждого оператора в отдельной строке.

Создаем свою первую работающую программу

Компиляция программы

Мы написали на языке программирования Си++ исходный текст своей первой программы — валютного калькулятора, переводящего доллары в рубли по текущему курсу. Теперь этот исходный текст можно преобразовать в готовую программу с помощью компилятора.



Предварительно сохраните свой проект с помощью упоминавшейся командной кнопки Save All (Сохранить все) и вызовите компилятор командой Project • MakeCalcProject(Проект • Сформировать CalcProject)или комбинацией клавиш CTRL+F9. Система C++Builder выведет на экран окно Background Compiling (Фоновая компиляция), представленное на рис. 10, в котором будет описываться процесс компиляции.

В системе C++Builder 5 введен новый режим компиляции — фоновый. Процесс сборки крупных проектов может занять большое время — до нескольких часов, и раньше программистам приходилось просто ждать окончания компиляции, так как она выполнялась в приоритетном режиме и не позволяла вносить никакие изменения в проект. Фоновая компиляция теперь представляет собой отдельный процесс Windows, и не мешает продолжать редактирование исходных текстов и форм. При этом компилятор поочередно транслирует каждый файл, входящий в проект, и если изменения в этот файл будут внесены уже после того, как он откомпилирован, эти изменения учтены не будут. Отключить фоновый режим можно, дав команду Tools • Environment (Сервис • Параметры среды) и сбросив на вкладке Preferences (Установки) флажок Background Compilation (Фоновая компиляция).

В разделе Project (Проект) указан файл проекта, в разделе Compiling (Компиляция) — текущий файл программы на Си++; в разделе Current Line (Текущая строка) — номер текущей компилируемой строки, по которому можно судить о том, сколько строк уже откомпилировано в текущем файле; в разделе Total Lines: (Всего строк) — суммарное число откомпилированное строк программы, которая может состоять из нескольких файлов с текстами на Си++.

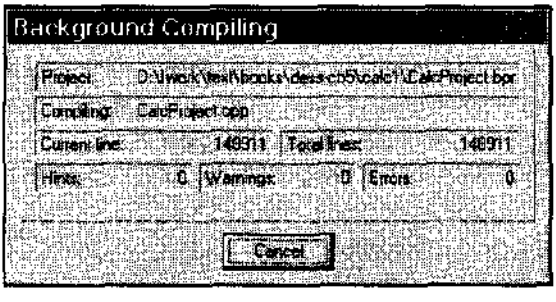


Рис. 10. Наблюдаем за компиляцией

Количество обнаруженных ошибок показывается в графе Errors: (Ошибок), число предупреждений — в графе Warnings: (Предупреждений), число подсказок — в графе Hints: (Подсказок).

Предупреждение — это сообщение C++Builder о том, что некоторое место в вашей программе хоть и не является ошибкой, однако потенциально может привести к ошибке. Например, если попытаться перемножить переменные DollarsNum и RateValue, но забыть предварительно записать в RateValue значение, то неверный код может быть таким:

```
int DollarsNum, RateValue;
int Result;
DollarsNum = StrToInt( Dollars->Text );
Result = DollarsNum * RateValue;
```

Тогда к моменту использования RateValue в ней будет храниться неизвестное случайное число, и результат умножения будет непредсказуемым. Выявить все сомнительные места в большой программе человеку не под силу, а компилятор определяет их без проблем.

Подсказки — более тонкая вещь. Они появляются, когда компилятор обнаруживает, что какие-то места в исходном тексте можно улучшить. Например, если написать

```
Result = 2;
Result = DollarsNum * RateValue;
```

то первый оператор присваивания

```
Result = 2;
```

будет выполняться впустую, поскольку в переменную Result сначала запишется число 2, которое тут же будет заменено произведением DollarsNum * RateValue.

Поэтому первый оператор присваивания можно смело убрать. Он не оказывает никакого влияния на логику работы программы, и только требует лишних тактов процессора для своего выполнения. Если подобные *пустые* (ненужные) операторы будут располагаться в частях программы, где происходят сложные и интенсивные вычисления, то потери времени могут стать заметными.

Работа по удалению ненужных и совершенствованию имеющихся операторов называется *оптимизацией* программы. Определенную оптимизацию C++Builder выполняет автоматически, а в спорных случаях, не рискует нарушить логику выполнения программы и обращается к человеку с советом

(подсказкой). Как правило, почти всегда эти советы оказываются полезными.

Старайтесь, чтобы компилятор не выдавал ни одного предупреждения или подсказки. Хотя программа, не содержащая ошибок, а только предупреждения, и будет работать, но гарантировать, что она будет работать правильно, нельзя.

Компилятор — умная и сложная программа, в нее заложены высокоэффективные алгоритмы анализа исходных текстов и их синтаксического разбора, существует даже большая математическая теория создания компиляторов, поэтому ко всем советам надо относиться очень внимательно.

Сборка программы

После компиляции исходных файлов формата .CPP будут созданы файлы формата .OBJ. Это так называемые *объектные файлы.* Например, на основе исходного кода Calc.cpp компилятором будет сформирован объектный код Calc.obj. Такой файл программой еще не является. Мы должны в конечном итоге получить исполнимый файл Calc.exe.

Объектный код — промежуточный. Для того чтобы сделать из объектного файла исполнимый файл, к объектному файлу надо подключить готовые функции типа StrTolnt(). Они хранятся в стандартных библиотеках, а библиотеки расположены в файлах с расширением LIB (сокращение от английского слова *Library — библиотека).* Если бы мы не использовали при написании исходного текста никаких стандартных функций, все равно функции из библиотек подключать надо, поскольку некоторый набор типовых системных функций был вставлен в текст нашей программы автоматически. Эти функции нужны для создания программы, отвечающей всем требованиям среды Windows. Нужно подключать также функции для использования готовых объектов, таких как поля, кнопки, окна и многие другие компоненты, которые хранятся в библиотеках в виде описаний типовых классов Си++.

Когда процесс компиляции закончится, надпись Compiling (Компиляция) автоматически сменится на надпись Linking (Сборка). Это означает, что компиляция прошла успешно, и система Borland C++Builder перешла к заключительному этапу *сборки* нашей программы. На этом этапе подключаются готовые компоненты из библиотек и все части программы собираются вместе в одно целое. Пока программа невелика, для хранения ее текста нам было достаточно одного файла, но, как только она значительно увеличится в размерах, логически независимые части удобно выделить в отдельные файлы. Эти

файлы с исходными текстами порознь компилируются в объектные файлы, которые затем на этапе сборки объединятся в законченную программу. Всю подобную работу по сборке программы выполняет не компилятор, а специальная программа-сборщик *(Linker).* Ее еще называют *редактором связей.*

Поскольку наша программа совсем небольшая, мы скорее всего не заметим всех этапов создания EXE-файла: компилятор и сборщик отработают очень быстро. Никаких сообщений при этом не появится. Начинающим программистам можно порекомендовать отключить режим фоновой компиляции, тогда C++Builder будет выдавать диалоговое окно с сообщением Done: Make, что означает Программа создана. Ни одной ошибки, предупреждения и подсказки быть не должно — это значит, текст программы набран правильно.

Если же появится надпись Done: There are errors — Найдены ошибки, значит, вы неправильно набрали текст или сделали какую-то ошибку во время проектирования формы или задания свойств компонентов. Следует внимательно проверить текст программы и попытаться повторить все действия заново. И так до тех пор, пока ошибок в программе не будет.

Запуск программы

Итак, программа готова к запуску. Закройте диалоговое окно компиляции Compiling (Компиляция), если режим фоновой компиляции отключен, и щелкните на командной кнопке Run (Запуск) или нажмите клавишу F9. По такой команде система Borland C++Builder автоматически запустит нашу первую программу. Выглядеть она будет примерно так, как показано на рис. 11.

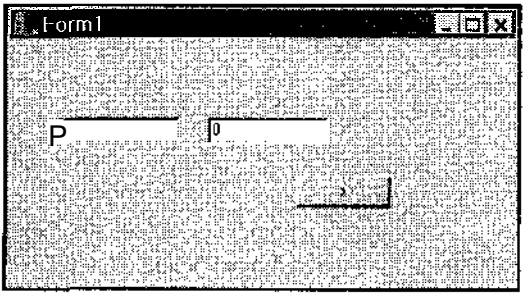


Рис. 11. Результат запуска программы

Введите в первое поле число 100 (100 долларов), а во второе — число 30 (курс 30 рублей за доллар), и нажмите кнопку >. В том месте, где мы расположили свободное поле, появится результат 3000 . Результат работы программы показан на рис. 12.

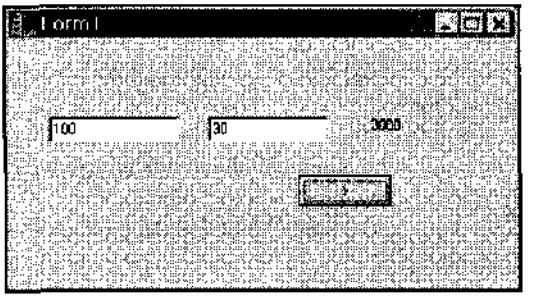


Рис. 12. Результат работы программы

Теперь мы можем себя поздравить. Наша первая программа успешно работает. Порадовавшись первому успеху, закройте ее любым принятым в Windows способом и вернитесь в систему Borland C++Builder, чтобы подумать, нельзя ли в ней что-либо улучшить.

Улучшаем калькулятор

Пересчет в доллары

Итак, наш калькулятор работает, но он очень односторонний: переводит доллары в рубли, а иногда надо и наоборот. Попробуем добавить в него обратный пересчет из рублей в доллары.

Для этого нам надо иметь возможность вводить в программу сумму в рублях. Пока для «рублей» используется только компонент Label (Поле записи), который годится лишь для просмотра информации и ввести в него ничего нельзя. Поэтому перейдите в режим проектирования формы, выделите щелчком левой кнопки мыши объект Label1 и нажмите на клавиатуре клавишу Delete (Удаление) — поле Label1 исчезнет с нашей формы. На его месте мы поставим еще один объект типа Edit (Поле ввода). Назовем его Roubles.

Мы изменили состав объектов на форме, но теперь это необходимо учесть и в тексте программы. Некоторые изменения система C++Builder сделает сама, однако те места, где мы явно обращались к измененным объектам, надо исправить. Б частности, в операторе, с помощью которого мы выводили результат на экран, присваивая вычисленное значение свойству Caption свободного поля Label1, надо изменить переменную, в которую будет заноситься конечный текст, на свойство Text объекта Roubles:

```
Roubles->Text = IntToStr( Result );
```

Всегда тщательно следите за использованием различных готовых элементов в своей программе. При удалении различных объектов или изменении их названия необходимо внести поправки и в текст программы, иначе в ней могут появиться обращения к несуществующим переменным. Такие ошибки будут выявлены компилятором, но лучше постараться все исправить заранее.

На поле формы между объектами Dollars и Rate вставьте новую кнопку, которая будет использоваться для обратного перевода. Дайте этому объекту название ConvertButton, а на кнопке сделайте надпись <.

Дважды щелкните на новой кнопке — откроется окно редактора программы. Создайте в нем программный код реакции на щелчок мышью на этой кнопке. Только теперь нам надо не умножать содержимое поля Dollars на содержимое поля Rate, а наоборот, делить содержимое поля Roubles на содержимое поля Rate. Измените для наглядности и название переменной для хранения числа рублей — назовите ее RoublesNum.

Весь текст, хоть и небольшой, заново набирать не обязательно. Просто скопируйте текст программы, описывающий реакцию на нажатие кнопки TotalButton, в буфер обмена Windows (командой CTRL+C), вставьте его в новое место между двумя фигурными скобками (командой CTRL + V) и измените нужные места:

```
{
int RoublesNum, RateValue;
int Result;
RoublesNum = StrToInt( Roubles->Text );
RateValue = StrToInt( Rate->Text );
Result = RoublesNum / RateValue;
Dollars->Text = IntToStr( Result );
}
```

Обратите внимание на последний оператор — в нем результат деления выводится в крайнем левом поле Dollars.

Подобный подход к подготовке исходных текстов, когда похожие части кода копируются и переносятся в новое место, таит в себе потенциальный источник ошибок. Если переносимый блок большой, легко можно забыть исправить какой-то оператор, что может привести к трудно выявляемым ошибкам. Например, если забыть в скопированном операторе

```
Result = DollarsNum * RateValue;
```

заменить DollarsNum на RoublesNum или знак умножения * на знак деления /, то никаких ошибок компилятор не найдет, поскольку текст с его точки зрения написан корректно, однако программа будет работать совершенно неправильно, и, возможно, потребуется немало времени, чтобы понять, где же таится ошибка.

Локальные переменные и область действия

А можно ли повторно определять переменную Result? Ведь она уже была описана, когда мы программировали реакцию на нажатие кнопки TotalButton, а выше неоднократно говорилось, что использовать одинаковые названия для переменных недопустимо.

Дело в том, что переменные в языке Си++ имеют свои *области действия,* которые определяют, где та или иная переменная используется. Применение одинаковых названий для переменных не разрешается, только если переменные с одинаковыми названиями попадают в одну и ту же область действия.

Формально переменная начинает свое существование с того места, где она была описана, а фактически — с начала логического блока, в котором она расположена. Этот блок, как уже отмечалось, задается фигурными скобками {...} и определяет область действия всех переменных, описанных внутри такого блока. Подобные переменные называются *локальными.* Они перестают существовать (исчезают, теряются) за закрывающей фигурной скобкой своего блока.

А так как переменную Result во второй раз мы определили внутри другого блока, то никакой ошибки не возникает.

Глобальные переменные и вложенные логические блоки

Есть еще один вид переменных — *глобальные.* Например, переменная Form 1, в которой условно хранится главная форма нашей программы — глобальная. Она определена вне всех логических блоков — просто в начале файла Calc.cpp. Областью действия глобальных переменных считается вся программа.

Настоятельно не рекомендуется использовать глобальные переменные (за исключением тех, которые C++Buitder создает автоматически). Применение глобальных переменных — очень плохой стиль программирования, приводящий ко множеству трудноуловимых логических ошибок, потому что очень сложно отследить, в каких частях программы и при каких условиях происходит изменение значений таких переменных.

— А можно ли использовать внутри логических блоков переменные с названиями, совпадающими с названиями глобальных переменных?

— Да, можно. Вообще, логические блоки разрешается неограниченно вкладывать друг в друга, и внутри каждого нового блока допускается описание переменных с именами, совпадающими с именами переменных из любых охватывающих блоков. Каждый раз при обращении к многократно определенной переменной берется значение, соответствующее значению переменной, описанной в текущем блоке.

Пример:

```
// здесь переменная T не определена
     {// начало блока A
          int T;
          T = 1;
               {// начало блока Б
                    float T;
                    T = 3.14;
               }// конец блока Б
     //здесь T снова содержит число «1»
               {// начало блока B
                    double T;
                    T = 3.3333;
               {// начало блока Г
                    int T;
                    T = 50;
               }// конец блока Г
               // T содержит 3.33 — блок B
          }// конец блока B
          // T содержит 1
     } // конец блока A
/* здесь T опять не определена и использовать ее ни в
  каких операторах нельзя */
```

Проверяем работу калькулятора

Теперь нажмите клавишу F9 (запуск программы). C++Builder автоматически определит, что исходный текст изменился, а в таком случае требуется заново перекомпилировать измененные файлы, что и будет проделано. Если все изменения сделаны правильно, то никаких ошибок не обнаружится и программа будет запущена.

Введем во второе окно число 30 (курс рубля), в третье — число 3000 (сумма в рублях), и щелкнем на кнопке обратного пересчета <. В первом поле мы увидим результат пересчета — 100. Все правильно! Так и должно быть.

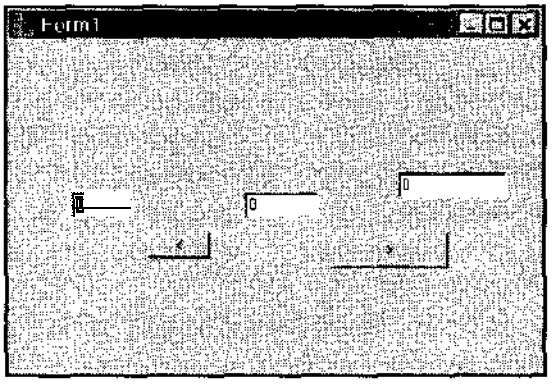


Рис. 13. Калькулятор модернизированный

А теперь попробуем ввести в последнем поле, где указывается число рублей, значение 1000, и снова попробуем пересчитать его в доллары, Получится 33 (доллара). Правильно ли это? Увы, нет! Даже на первый взгляд видно, что число 1000 на 33 нацело не делится, и результат должен быть дробным. Если прикинуть на калькуляторе, мы должны получить 33,33333...

Другие типы для целых чисел

Дело в том, что тип int, который мы использовали для описания переменных, в которых хранятся результаты вычислений, допускает работу только с *целыми* числами в диапазоне от -2 147 483 648 до 2 147 483 647, а в программе в результате получилось число с дробной частью. В Си++ принято, что *при делении целого числа на целое* ***всегда*** *получается тоже целое число* (дробная часть просто отбрасывается). То есть получившееся дробное значение 33,3333 было преобразовано в целое число 33.

— Почему значения целых чисел лежат именно в диапазоне от –2 147 483 648 ДО 2 147 483 647?

— Вы наверняка слышали, что процессор Pentium — 32-разрядный. Что это означает? Применительно к нашему случаю это означает, что он работает с числами, которые представлены в диапазоне от 0 до $2^{32}-1$, а именно от 0 до 4 294 967 295 (для этого в Си++ есть тип, называемый *unsigned long).* Чтобы охватить и отрицательные числа, надо этот диапазон поделить примерно пополам — отсюда и берутся границы от -2 147 483 648 до 2 147 483 647.

Для разных версий языка Си++ допустимые диапазоны значений различных типов могут различаться. В этой книге рассматриваются диапазоны, принятые в системе Borland C++Builder.

Два миллиарда с «хвостиком» — это, конечно, маловато. Даже бюджет нашей страны не подсчитать. Поэтому в C++Builder используются и другие типы чисел, в частности, тип__int64, который позволяет работать с числами в диапазоне от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807. Столь больших значений для решения обыденных задач вполне достаточно.

Только записываются столь длинные числа не совсем обычно. Чтобы компилятор понял, что это число типа__int64, к нему «в хвост» надо приписать обозначение i64, например:

```
3000000000i64
```

При записи целых чисел никаких пробелов или запятых не допускается!

Правильно:

3000000

3000000000i64

Неправильно:

3 000 000

3,000,000

— А что будет, если значение все-таки выйдет за разрешенные границы?

— Такие ситуации в Си++ ошибками не считаются, и никаких предупреждений по этому поводу компилятор C++Builder не выдает (хотя некоторые трансляторы это делают), даже если вы явно напишете (неверно!):

```
int i;
i = 3000000000;
```

— А что же произойдет во время работы программы?

— Число 3000000000 будет автоматически преобразовано в допустимое число из диапазона int — в число -1 294 967 296. Как вы видите, подобное преобразование может дать самый неожиданный, в нашем случае даже отрицательный, результат!

Если сложить абсолютное значение этого «странного», на первый взгляд, результата, и 3000000000, то мы получим 4 294 967 296 то есть 2^{32}.

Си++ — язык программирования очень мощный и эффективный. Именно благодаря тому, что во время работы программы не происходит всевозможных проверок, таких, например, как проверка на граничные значения, программа выполняется очень быстро. С другой стороны, по этим же причинам Си++ потенциально таит в себе множество подводных камней, различных источников ошибок, поэтому составлять программу на этом языке надо очень аккуратно и внимательно.

Типы для очень больших, очень малых и дробных чисел

— А что, если потребуется подсчитать число звезд на небе или определить, какую часть от общего числа молекул в солнечной системе составляют молекулы нашего компьютера?

— Здесь бухгалтерская точность не нужна, но требуется умение работать с очень большими или, наоборот, очень маленькими числами. Для этого в Си++ есть три специальных типа. Они позволяют использовать в программе дробные числа с длинными цепочками нулей. Записываются такие числа в натуральном виде например так;

б . 67e+24 или так 0 . 66E+25

- сначала записывается число, которое называется *мантиссой* (эта часть может быть дробной);
- за мантиссой ставится символ e или E;
- далее записывается *характеристика* — целое число со знаком, указывающее, на сколько разрядов влево или вправо сдвигается десятичная запятая; знак «+» разрешается не указывать.

Например:

```
3000000000 = 3e9
        10 = 1E1
0.03000000003 = 3e-12
```

При записи десятичных дробных чисел вместо привычной нам со школы запятой, отделяющей целую часть числа от дробной, в языке Си++, созданном американцем Бьорном Страуструпом, используется традиционно применяемая для этих целей в США точка.

Числа с дробной частью в Си++ называются числами с плавающей запятой (по-английски они, соответственно, называются числами с плавающей точкой). Термин «плавающая» появился потому, что одно и то же число можно записать по-разному:

• 123.45

1.2345e2

1234.5e-1

Получается, что десятичная запятая (точка) как бы «плавает» по числу, не меняя его значения.

Тип float

Все подобные числа с плавающей запятой имеют в Си++ тип float (действительное число с плавающей запятой). Этот тип определяет диапазон значений чисел от 1.18e-38 до 3 .4e38.

— Много это или мало?

— Конечно, число с 38 нолями очень велико. Однако в реальной жизни встречаются ситуации, когда и таких огромных диапазонов недостает. Например, количество всевозможных позиций в шахматной партии примерно определяется числом со 120 нолями.

Тип double

Для работы с подобными значениями используют тип double (действительное число двойной точности с плавающей запятой), охватывающий диапазон от 2.23e-308 до 1.79e3O8.

Тип long double

Выше мы без труда нашли вполне жизненный пример, когда может потребоваться число со 120 нолями. Однако не исключено, что кто-то захочет применить число с пятьюста нолями. Почему бы и нет? Желая подстраховаться, разработчики C++Builder ввели в язык программирования еще один тип чисел — long double (длинное действительное число двойной точности с плавающей запятой), который позволяет манипулировать значениями от 3 . 4e-4932 до 1.2e4932. Этот тип наверняка удовлетворит нужды всех программистов.

Операции с числами с плавающей запятой выполняются *приближенно.* Как правило, ошибка возникает в 7-10 разряде, то есть число 1,23456789 может храниться в компьютере как число 1,234567412 и т. п. Числа типов float и double ***нельзя*** применять в задачах, требующих высокой точности, например, нельзя выполнять бухгалтерские расчеты и подсчитывать деньги с помощью типа float. Для этого обычно используются обходные способы — например, суммы денег хранятся как целые числа, в которых два последних разряда условно считаются «копейками». При этом, правда, возникают другие проблемы, например, сточным делением таких «псевдодробных» чисел.

Язык Си++ исходно создавался для решения задач интенсивной обработки информации, и хотя сегодня его применяют с самыми разными целями, для применения в некоторых областях он подходит плохо. Однако он имеет огромную популярность, вызванную самыми разными причинами. В частности, Си++ позволяет создавать очень быстрые и компактные программы, что имело важное значение в 80-е годы, когда компьютеры были медленными и имели крохотную память.

Сегодня быстродействие и размер программ уже не так важны, но, хотя уже есть языки значительно лучше Си++, он традиционно остается самым распространенным языком программирования, поэтому мы не будем его критиковать, а постараемся научиться на нем хорошо работать, стараясь не забывать про всевозможные подвохи.

Исправляем типы переменных

Теперь наша ошибка ясна. Все упоминания типов int надо заменить на float. В нашем случае копеечные ошибки округления не важны, ведь главное — получить примерный конечный результат.

Измените текст программы. Это удобнее всего сделать с помощью команды Replace All (Заменить все). Нажмите комбинацию клавиш CTRL+R, и на экране появится диалоговое окно замены текста, наверное знакомое многим по работе с обычными текстовыми редакторами, представленное на рис. 14.

В строке Text to find: (Найти текст) введите `int` с пробелом в конце, чтобы не заменялись кусочки текста в названии функции IntToStr(), в строке Replace with: (Заменить на) введите: `float` (тоже с пробелом в конце), в разделе Origin (Область действия) включите переключатель Entire scope (Проверить весь текст) и щелкните на кнопке Replace All (Заменить все).

Проверьте, чтобы описания ваших переменных теперь выглядели так:

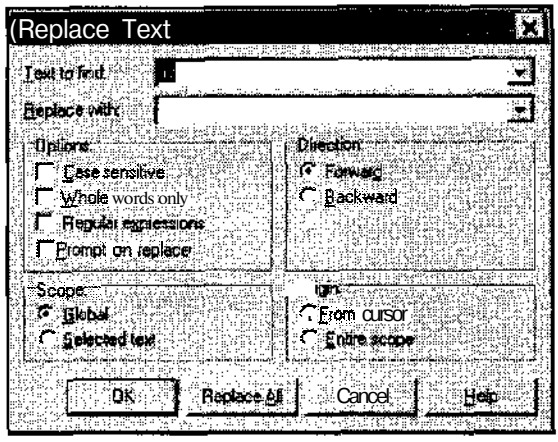


Рис. 14. Автоматическая правка текста программы

```
float  RoublesNum,  RateValue;
float  Result;
```

Если что-то сделано неправильно, например, не поставлен пробел после float, вернитесь к предыдущему варианту текста программы с помощью комбинации клавиш CTRL+Z (это так называемая команда Undo — «Откат», типичная для большинства приложений Windows) и попробуйте исправить текст заново.

Однако исправлению надо подвергнуть не только определения переменных, но и вызовы стандартных функций lntToStr() и StrToInt(), которые предназначены для операций с целыми числами. Если такие изменения не внести, компилятор выдаст сообщение о том, что мы попытались осуществить недопустимое преобразование типов — из дробного в целое.

— А почему компилятор так строг? То он не замечает странных преобразований чисел, а то не позволяет сделать из дробного 1,0 целое 1.

Это нужно как раз затем, чтобы программисты внимательно следили за типами данных в своей программе и не допускали «самопроизвольного» неправильного преобразования значений. Тем более, что 1,0 переделать в единицу не сложно, а вот если 3,14 округлить до целого числа 3 и ничего при этом программисту не сообщить, то скорее всего программа будет работать неправильно.

Если вы хотите преобразовать один тип в другой (это называется *приведением* типов), надо сделать это явно, указав тем самым компилятору, что вы понимаете, что творите, и тогда он успокоится.

Для преобразования одного типа в другой есть два пути. Первый — использовать стандартные функции C++Builder, как это было, например, в случае перевода числа в строку с помощью функции IntToStr(). Второй путь — указать в круглых скобках новый тип перед переменной или выражением, результатом которого будет значение определенного типа, указать в круглых скобках новый тип, например:

```
int m;
m = (int)3.14;
```

При этом никаких опшбок и предупреждений не появится, а переменная m получит значение 3.

Явно преобразовывать числа из целого типа в тип с плавающей запятой не требуется, так как потери данных здесь не происходит. К таким числам сразу за запятой просто автоматически приписывается ноль. Например:

```
float x;
x = 5; // это правильно
```

Вообще, в Си++ разрешены любые приведения типов от более простых к более сложным — там, где гарантированно не произойдет потери значений. Например, не считается ошибкой или просто некорректной записью, вызывающей предупреждение, приведение типа от float к double (но не наоборот).

Можно преобразовывать к другому типу весь результат выражения:

```
int N ;
N = (int)(5.5 + 4.4);
```

При этом результат выражения (5.5 + 4.4), равный «9,9», будет преобразован в тип int (получится целое число 9, а дробная часть отбросится).

Старшинство операций

Интересно, что вышеприведенный пример можно записать и так:

```
int K;
N = (int)5.5 + 4.4;
```

Может показаться, что к целому типу будет преобразовано только число «5.5», по это не так.

В Си++ каждая операция в выражении имеет свой *приоритет*, с помощью которого определяется порядок расчета этого выражения (это знакомый из школьного курса арифметики порядок действий).

Когда мы пишем

```
2*2 + 3*3
```

то подразумеваем, что сначала будут выполнены операции умножения и только потом — сложения. Так же и в Си++ — операции умножения и деления имеют более высокий приоритет, чем операции сложения и вычитания, и поэтому выполняются первыми.

Приведение типа в Си++ тоже считается операцией, только приоритет у нее очень низкий. Поэтому в выражении `(int)5.5 + 4.4` сначала будет выполнена операция сложения `5.5 + 4.4`, и только потом полученное число «9.9» будет преобразовано в тип int.

Лучше не полагаться на приоритеты, и в неочевидных случаях выделять порядок действий круглыми скобками. Если порядок вычисления выражения 2*2 + 3*3 понятен каждому, то в нашем примере порядок желательно указать явно — (int)(5.5 + 4.4).

Другие функции преобразования чисел в текст

Мы рассмотрели два способа преобразования типов — с использованием стандартных функций и с явным указанием типа. Лучше всегда использовать первый способ преобразования типов, то есть применять стандартные функции, поскольку они автоматически отслеживают возможные неверные значения параметров.

Как мы уже выяснили, стандартные функции IntToStr<) и StrToInt() в нашем случае использовать нельзя, так как они работают с целыми числами, а нам нужны дробные значения. Приведение типа к целому типу int тоже применять не надо, так как нам нужны дробные числа. Здесь нам помогут стандартные функции FloatToStr() и StrToFloat{) — исправьте повсюду в своей программе (в шести местах) IntToStr() на FloatToStr(), а StrToIntO на StrToFloat(). Откомпилируйте программу (ошибок и предупреждений быть не должно) и запустите.

Введите в первом окне число 100 (100 долларов), а во втором — курс, например 25,5 рублей за доллар.

Хотя, как уже говорилось, в Си++ для отделения целой части числа от дробной используется точка, но после запуска программы мы уже не находимся в системе программирования Borland C++Builder, а работаем под управлением операционной системы Windows, в которой правила отделения целой и дробной части могут быть иными.

Все правильно написанные прикладные программы Windows используют для этой цели символ, который определен в системном файле Windows WIN.INI, в разделе [intl] — строка sDecimal. Этот раздел специально предназначен для настройки Windows на требования конкретной страны. У тех, кто использует русифицированную версию Windows и ничего не менял в ее настройках, эта строка будет выглядеть так:

```
sDecimal=,
```

То есть во всех элементах пользовательского интерфейса при вводе дробных чисел надо использовать не точку, а запятую.

Не путайте десятичную запятую, считающуюся стандартной для всех программ Windows, с десятичной точкой Си++, которая используется *только* для записи дробных чисел в тексте программы.

Поэтому мы вводим число 25,5 (а не 25.5). После щелчка на командной кнопке > появится результат 2550 (рублей).

— А если курс изменится — резко поднимется до 29,9 рублей? Сколько долларов мы сможем купить тогда?

— Введите новый курс и щелкните на кнопке <. В первом поле появится результат 85,2842788696289. Конечно, не все цифры этого числа верны — ошибка допущена уже в седьмом знаке (правильное значение — 85,284280936454), но для нашего случая это не столь важно.

Проверьте и старый пример — пересчитайте 1000 рублей по курсу 30. Результат — 33,3333320617676 (с ошибкой в шестом знаке).

Оформляем результат

— Число 85,2842788696289 неудобно для восприятия. Хорошо бы показывать его только с двумя цифрами после запятой. Неужели для нормального представления длинных чисел в C++Builder есть специальная функция?

— Конечно, есть! Такая функция называется FloatToStrF(). Она преобразует число в строку в соответствии с нашими требованиями. У этой функции четыре параметра:

- первый параметр — дробное число;
- второй параметр — константа (фиксированное значение), определяющая способ представления данного числа; если мы хотим показать число в обычном виде, вторым параметром надо поставить константу ffFixed;
- третий параметр — количество знаков числа, при превышении которого в левой (от запятой) части, число преобразуется в нормализованную форму (в нашем примере можно указать гарантированно большое значение, например 10 — вряд ли нам понадобится пересчитывать суммы с десятью нулями);
- четвертый параметр — число, определяющее, сколько знаков после запятой мы хотим оставить.

Мы ввели новое понятие — *константа.* Константы используются в Си++ очень часто. Программисту нередко приходится применять значения, строго определенные создателями Windows и компилятора C++Builder, например, в качестве параметров функций стандартных библиотек или внутренних вызовов Windows. В случае с FloatToStrF() необходимость представлять число в различных форматах определяется разными значениями второго параметра, заучивать которые совсем не обязательно. Этим числам соответствуют определенные названия, которые запоминать гораздо легче и проще.

Теперь в тексте программы там, где определяется реакция на нажатие каждой кнопки, надо исправить последние операторы присваивания и представить значение с двумя цифрами после запятой. Закройте калькулятор, вернитесь в C++Builder и сделайте необходимые изменения.

Исправленные операторы будут выглядеть так:

```
Roubles->Text = FloatToStrF(Result,ffFixed,10,2);
Dollars->Text = FloatToStrF(Result,ffFixed,10,2);
```

Запустите программу, и теперь, наконец, пересчет 1000 рублей по курсу 30 даст нормальное значение 33,33.

Округление результата

Допустим, что нас не интересуют центы и копейки, и мы хотим получать только целочисленные результаты. Только желательно результат округлять, а не отбрасывать дробную часть. Согласитесь, что 9,9 рублей не то же самое, что 9 рублей.

Как ни странно, в C++Builder нет стандартной функции, округляющей значение дробного числа до ближайшего целого. Есть только функция floor(), которая просто убирает дробную часть, то есть работает как операция приведения (int).

Однако есть очень простая хитрость, чтобы добиться нужного результата. Для получения округленного значения переменной *x,* надо записать:

```
floor(x+0.5)
```

Этот старый способ работает безукоризненно.

Теперь вы можете исправить те операторы присваивания, где вычисляется результирующее значение: в переменную Result надо записывать округленные величины.

Исправленные операторы будут выглядеть так:

```
// зычисление рублевой суммы:
Result = floor( DollarsNum * RateValue + 0.5 );
// зычисление долларовой суммы:
Result = floor( RoublesNum / RateValue + 0.5 );
```

Однако не пытайтесь сейчас запустить программу. Компилятор сообщит вам об ошибке: Call to undefined function «floor» (Вызов неизвестной функции «floor»).

— Почему же она неизвестна? Ведь это же, как говорилось, стандартная функция?

Описание функций и подключение стандартных библиотек

— Да, действительно, функция floor() — стандартная. Существует очень много стандартных функций. Они объединены в группы и хранятся в отдельных библиотеках (LIB-файлах), сформированных по тематическому признаку (функции для математических расчетов, функции ввода/вывода и т. п.). Некоторые необходимые для работы C++Builder стандартные библиотеки подключаются к программе автоматически. Для других, более редких библиотек, это надо делать вручную,

— Как это сделать?

— Прежде всего, надо выяснить, в какой библиотеке хранится нужная нам функция, в частности, функция floor(). Для этого обратимся к подсказке из справочной системы. Справочная система вызывается командой Help • Contents (Справка ▸ Содержание). На вкладке Index (Указатель) следует ввести искомое слово: `floor`(см. рис. 15).

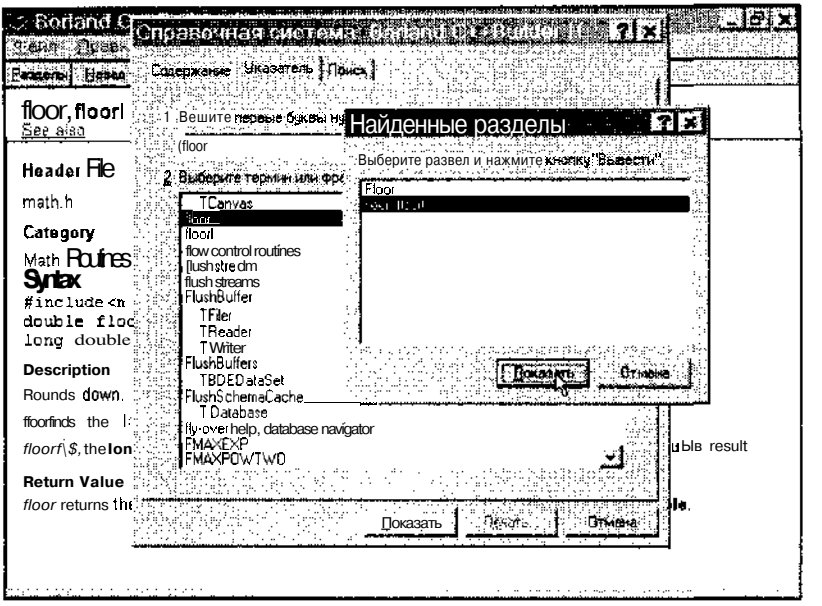


Рис. 15. Поиск информации в справочной системе

Щелкнем на кнопке Показать, выберем в списке функцию, точно совпадающую с названием floor с учетом регистра, и еще раз воспользуемся кнопкой Показать. На экране откроется окно со статьей справочной системы (см. рис. 16).

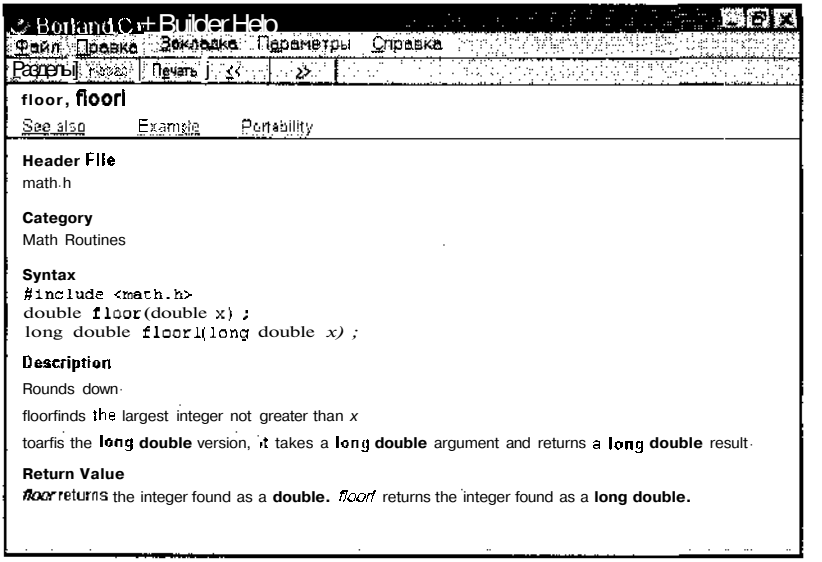


Рис. 16. Статья справочной системы

В самой первой строке написано: Header File — это название раздела. Здесь и указана нужная нам библиотека. В следующей строке — название файла, в котором содержится описание всех функций этой библиотеки (math.H). Такие файлы называются *заголовочными*. Они имеют расширение .H (иногда .HPP). В подобные файлы также нередко включают описание классов.

Чтобы компилятор понял, что мы хотим использовать конкретную функцию, ее описание надо включить в исходный текст примерно так, как мы это делали с переменными — до первого использования. Только описание функции надо вставлять не между фигурными скобками, а в начало файла.

Функция описывается следующим образом. Сначала указывается тип возвращаемого ею значения, затем название функции, далее в круглых скобках список параметров с типами. В конце ставится точка с запятой. В частности, функция floor() описывается так:

```
double  floor(double x) ;
```

В списке параметров указано имя переменной *x*. На самом деле при описании функции для каждого параметра название переменной выбирается совершенно произвольно и реально не имеет никакого смысла и значения. Более того, это имя можно вообще не указывать, то есть описание floor() может выглядеть следующим образом:

```
double  floor(double);
```

Если посмотреть содержимое файла math.h в текстовом редакторе, то там можно найти и описание функции floor(), и описание еще множества других функций. Заголовочные файлы, которые просто включаются в текст нашей программы, сделаны специально, чтобы не набирать вручную описание какой-либо функции, когда понадобится ее использовать.

— А как же включить заголовочный файл в текст?

— Сделать это можно с помощью *командной строки:*

```
#include КУПЛЯ-файла>
```

или

```
#include "имя-файла"
```

Имя включаемого файла берется в угловые скобки, если содержащиеся в нем функции входят в состав стандартных библиотек и при этом подразумевается, что он расположен в каталоге, где по умолчанию хранятся все заголовочные файлы стандартных библиотек (путь к этому каталогу можно изменить в настройках C++Builder). Если же во включаемом файле

содержатся описания пользовательских функций и классов, то его имя обычно берется в кавычки и этот файл ищется прежде всего в каталоге текущего проекта.

В нашем случае требуемая командная строка запишется так:

```
#include <math.h>
```

Фактически при компиляции она заменится на содержимое файла math.h.

Командные строки включения заголовочных файлов принято размещать в начале файла.

С помощью полосы прокрутки перейдите к началу файла Calc.cpp. Там можно найти автоматически сгенерированные команды включения файла vcl.h (стандартный файл библиотеки визуальных компонентов — *Visual Component Library)* и файла Calc.h, в котором описан класс нашей формы. Включите файл math.h сразу после vcl.h:

```
#include <vcl.h>
#include <math.h>
```

Теперь откомпилируйте и запустите программу (клавиша F9), и пересчет 1000 рублей по курсу 30 даст округленное значение 33,00. Это значение все равно будет выводиться с двумя нолями после запятой, как мы и указали в функции FloatToStrF() — ведь программе неважно, выводятся ноли или другие цифры, она просто в точности выполняет то, что ей указано.

Наводим красоту

Итак, наша первая программа работает. Однако чтобы сделать ее законченным продуктом, который было бы не стыдно как минимум показать друзьям, надо навести на нее «блеск»: аккуратно и ровно разместить все элементы управления в форме и дать рабочему окну нормальное название.

1. Начнем с дизайна формы. Сначала определим ее размер, потянув за правый нижний угол. Далее надо определить точность выравнивания элементов, то есть задать расстояние между линиями сетки — по умолчанию это значение равно 8 пикселам (экранным точкам). Если такая величина не устраивает, с помощью команды Tools • Environment Options (Сервис ▸ Параметры среды) можно открыть диалоговое окно для настройки параметров визуальной среды C++Builder (см. рис. 17). В этом диалоговом окне выберите вкладку Preferences (Установки), и в полях Grid size X: (Шаг по горизонтали) и Grid size Y: (Шаг по вертикали) задайте новые числа, например, 5. Щелкните на кнопке OK, и шаг сетки на форме станет мельче.

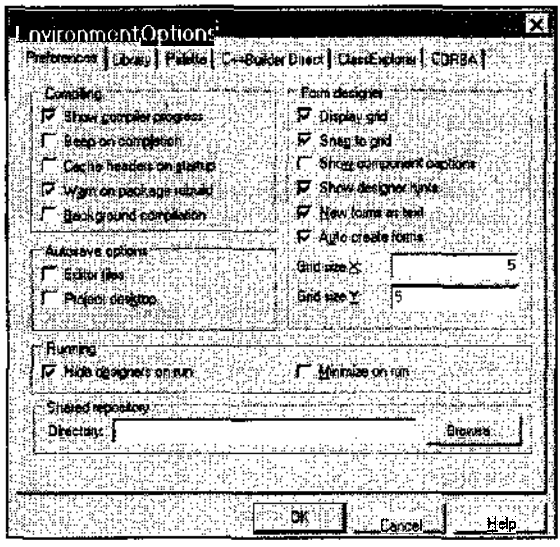
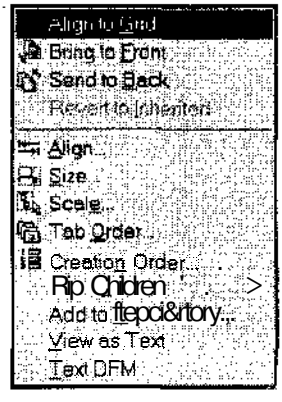


Рис. 17. Настройка параметров среды визуального программирования

2. Следующая задача — выровнять все элементы на форме по границам сетки. Для этого щелкните левой кнопкой мыши на левой верхней части экрана и, не отпуская кнопку, протяните указатель вправо и вниз. При этом на экране появится пунктирный прямоугольник, в который должны попасть все объекты формы. Это прием группового выделения объектов методом протягивания мыши. Отпустите кнопку, и все объекты в проектировщике формы останутся выделенными.

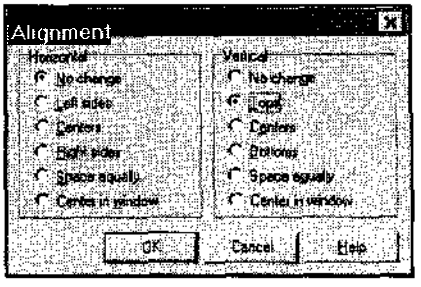


Теперь наведите указатель мыши на один из выделенных элементов и щелкните правой кнопкой — откроется контекстное меню, в котором имеется пункт Align to grid (Выровнять по сетке). Все объекты автоматически подравняются по ближайшим узлам настроечной сетки.

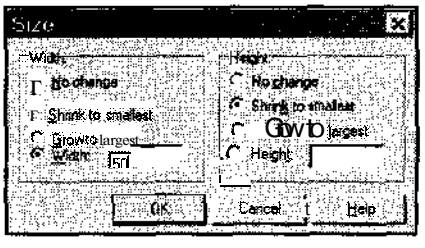
3. Теперь выстроим объекты по одной линии. Тем же способом вызывается контекстное меню, но на сей раз в нем выбирается пункт Align (Выравнивание). Выбор этого пункта влечет за собой открытие одноименного диалогового окна.

В данном диалоговом окне две панели. Одна содержит элементы управления выравниванием по горизонтали (Horizontal), а другая — по вертикали (Vertical). В силу горизонтальной компоновки элементов нашего калькулятора нам надо выбирать вертикальное выравнивание, напри-

мер по верхним краям элементов. Соответственно, включаем переключатель Tops (По верхним краям), а горизонтальное выравнивание оставляем без изменений с помощью переключателя No change (Без изменений). Задав конфигурацию переключателей, щелкните на кнопке OK.



4. Чтобы наши элементы управления выглядели аккуратно, им надо придать одинаковые размеры. Выделите элементы, щелкните правой кнопкой мыши и в контекстном меню выберите пункт Size (Размер) — откроется одноименное диалоговое окно выравнивания элементов по ширине (панель Width) и высоте (панель Height).



В системе Borland C++ Builder 5 предусмотрены несколько режимов выравнивания. Переключатель Shrink to smallest (Уменьшить до наименьшего) позволяет выровнять элементы по размеру самого малого выделенного объекта. Переключатель Grow to largest (Увеличить до наибольшего) обеспечивает выравнивание по размеру наибольшего элемента.

Переключатели Width (Ширина) или Height (Высота) служат для непосредственного ввода линейных размеров всех выделенных элементов управления. Размер задается в пикселах. Числа вводятся в соответствующие поля ввода. Если ничего изменять не надо, включите переключатель No change (Без изменений).

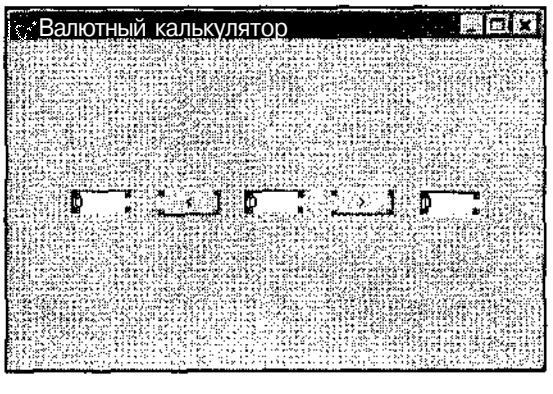


Рис. 18. Форма посла выравнивания элементов

Для нашего примера можно выровнять высоту всех элементов по наименьшему, а ширину сделать равной 50 пикселам. Установите соответствующие переключатели и щелкните на кнопке OK.

5. Теперь неплохо бы сделать между элементами формы одинаковые интервалы, но для этого сначала надо определить порядок перехода между элементами в форме.

Порядок перехода между элементами управления имеет важное значение при создании хорошего пользовательского интерфейса. Когда мы ввели в первое поле число, удобно перейти к следующему полю, не прибегая к мыши, а с помощью клавиатуры. В Windows для выполнения такой стандартной операции предназначена клавиша TAB. При ее нажатии управление передается другому элементу (в таких случаях говорят о том, что передается *фокус ввода).* Однако если объекты создавались нами в случайном порядке, а какие-то из них удалялись, то при нажатии клавиши TAB фокус ввода может скакать по элементам управления формы совершенно непредсказуемо. Это и неудобно, и неэргономично (эргономика — наука о способах повышения комфортности и эффективности работы человека).

Не снимая выделения с элементов управления, вызовите контекстное меню и выберите в нем пункт Tab Order (Порядок перехода). Появится диалоговое окно Edit Tab Order (Редактирование порядка перехода) — рис. 19. В этом окне представлен список всех наших объектов. Пункты этого списка можно перемещать с помощью кнопок Вверх и Вниз. Переход между этими элементами с помощью клавиши TAB в окне работающей программы будет происходить именно в том порядке, в каком эти элементы представлены в списке сверху вниз.

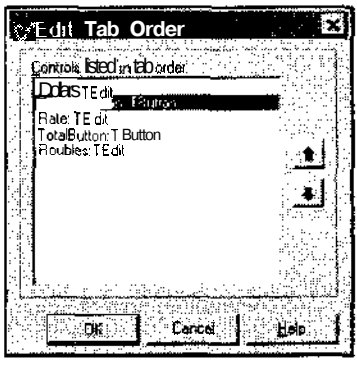


Рис. 19. Редактирование порядка перехода между элементами управления

Закончив настройку порядка перехода, щелкните на кнопке OK.

6. Возвращаясь к настройке одинаковых интервалов между элементами управления, вновь выделите *все* элементы и вызовите контекстное меню. В нем вновь, как ранее в п. 3, выберите пункт Align (Выравнивание) и в уже знакомом нам диалоговом окне включите переключатель Space equally (Распределить равномерно) на панели Horizontal (По горизонтали).

   После закрытия диалогового окна элементы управления распределятся по полю формы с равными интервалами. Теперь вновь вызовите это же самое диалоговое окно, и на обеих панелях включите переключатели Center in window (Выровнять по центру формы). Все наши объекты окажутся симметрично расположенными относительно центра будущего рабочего окна программы (см. рис. 20).

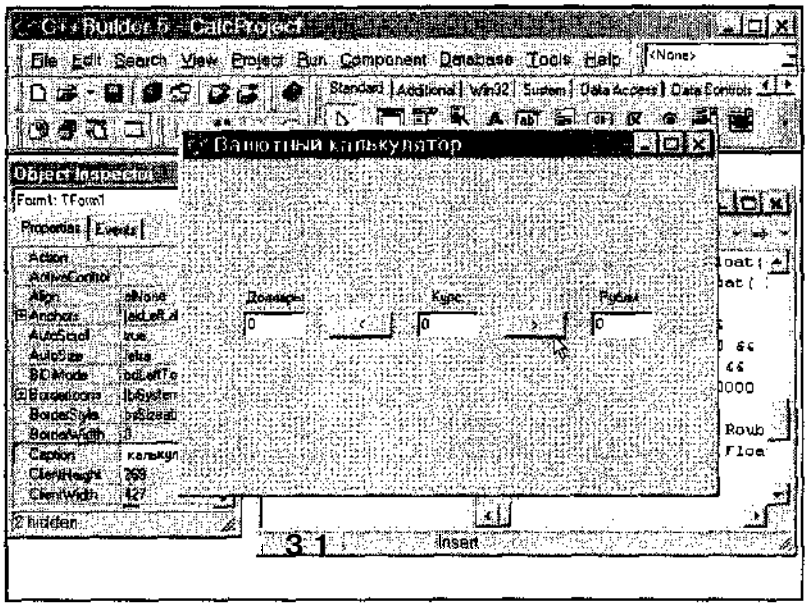


Рис. 20. Так выглядит форма будущей программы по завершении всех настроек

7. Кажется, что мы уже сделали все, что можно, однако надо подумать и о пользователе нашей программы — ведь интерфейс делается прежде всего для него. Постороннему человеку будет непонятно, как работает калькулятор, ведь у полей ввода нет ни одной надписи.

   Поэтому давайте разместим на форме три новых поля надписей (напомним, что эти компоненты называются Label). Дадим им соответствующие

названия и выровняем их так, как мы уже это делали для других объектов (см. рис. 20).

8. Теперь нам остался совсем пустяк — дать своей форме нормальный заголовок вместо Form 1, сгенерированного системой Borland C++Builder по умолчанию. Для этого щелкните на любом свободном месте формы, чтобы она стала в Инспекторе объектов текущим объектом, и в свойстве **Caption** укажите Валютный калькулятор.

На этом создание программы закончено. Для проверки откомпилируйте ее. Никаких ошибок быть не должно.

Программа готова

Теперь наш валютный калькулятор можно использовать как законченную программу. Перейдите, например, в Проводник, и найдите папку со своим проектом Calc. Там будет 14 файлов, и среди них и наша программа со стандартным значком C++Builder. Нам, конечно, этот значок не подходит, но его можно сменить.

Для смены значка щелкните на нем правой кнопкой мыши. В открывшемся контекстном меню выберите пункт Свойства — откроется диалоговое окно для настройки свойств значка. В этом диалоговом окне выберите вкладку Ярлык. Щелкните на кнопке Сменить значок — откроется диалоговое окно Смена значка. Если в окне представлена коллекция значков, выберите тот, который вас устроит, и щелкните на кнопке OK.

Операционная система Windows имеет несколько готовых библиотек значков. Вы можете использовать любую. Их адреса:

```
C:\WINDOWS\SYSTEM\shell32.dll
C:\WINDOWS\SYSTEM\pifmgr.dll
C:\WINDOWS\SYSTEM\moricons.dll
C:\WINDOWS\SYSTEM\progman.exe
```

Выбор нужной библиотеки выполняется после щелчка на кнопке Обзор.

Полученную программу можно скопировать в любой каталог, а в дальнейшем сделать из нее готовый продукт, переписать на дискету или распространить через Интернет и запускать на других компьютерах. Размер версии, способной работать на других компьютерах, где не установлены стандартные библиотеки C++Builder, составит всего 24 Кбайт!

3. Заглядываем внутрь работающей программы

Повторное открытие проекта

Возможно вы решили устроить паузу в работе, или намереваетесь сделать это в ближайшем будущем. Если закрыть C++Builder командой File ▸ Exit (файл • Выход) и через некоторое время запустить систему вновь, понадобится заново открывать проект CalcProject, так как исходно C++Builder создает новую пустую форму. Чтобы открыть существующий проект, есть два пути. Можно выбрать свой проект командой File • Open Project (Файл ▸ Открыть проект) (комбинация клавиш CTRL+F11), указав в стандартном диалоговом окне имя файла проекта CalcProject или командой File • Reopen (Файл ▸ Открыть повторно), где в верхнем разделе меню будет показан список проектов, с которыми недавно работали (см. рис. 21).

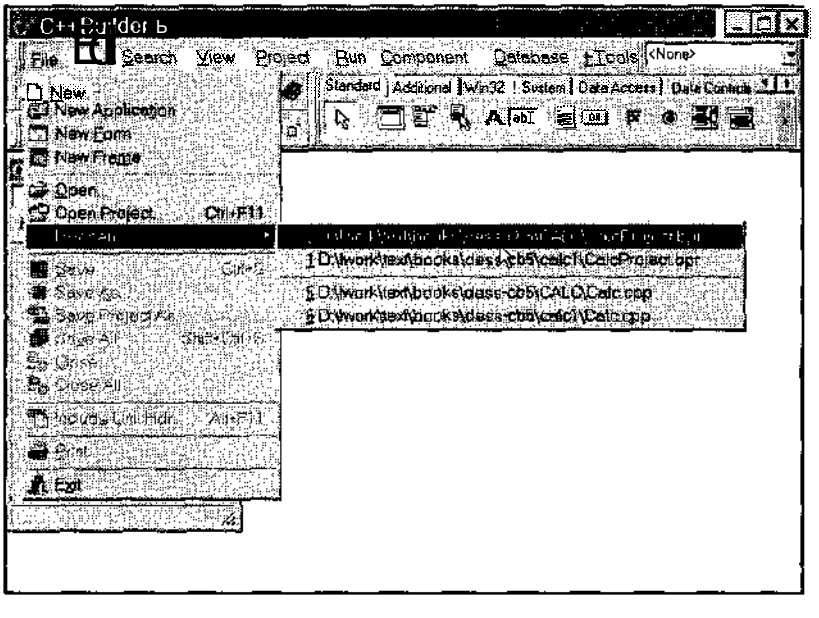


Рис. 21, Повторное открытие проекта

Это самый удобный способ. Выберите строку с номером 0, и все модули программы валютного калькулятора загрузятся автоматически с сохранением всех старых настроек и положений окон визуального проектировщика, редактора и Инспектора объектов.

Выполняем запрещенное действие

Запустите калькулятор (клавиша F9) и в окне, где указывается сумма в рублях, введите число 1000. Нажмите кнопку пересчета в доллары (курс не меняется и остается равным нулю, как задано по умолчанию). Работа калькулятора прервется, и на экране возникнет сообщение C++Builder об ошибке (см. рис. 22).

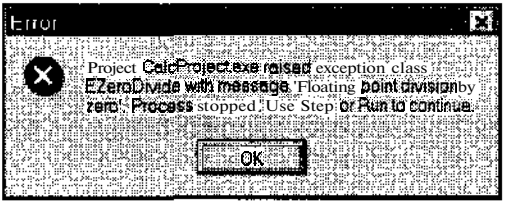


Рис. 22. Попытка разделить число на ноль неминуемо влечет за собой появление сообщения об ошибке

Почему так произошло? Да потому, что при пересчете рублей в доллары программа выполнила деление значения переменной RoubiesNum (в нашем случае — 1000) на значение переменной **RateValue** (в нашем случае — 0). Но как известно из школьного курса арифметики, на ноль делить нельзя! Поэтому в программе возникла так называемая исключительная ситуация, вызванная попыткой калькулятора выполнить действие, запрещенное в операционной системе. Вследствие этого работа программы была приостановлена, о чем и было сообщено.

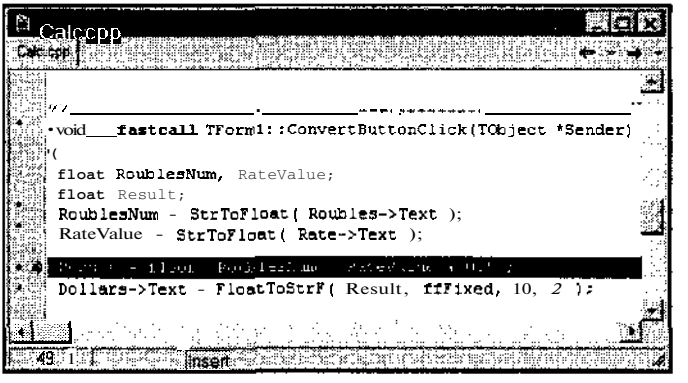


Рис. 23. Система подсказывает нам, где произошла запрещенная операция

Щелкните на кнопке OK, и C++Builder покажет исходный текст программы, выделив в нем строку, в которой располагается оператор, попытавшийся выполнить недопустимую операцию (см. рис. 23).

— С левой стороны окна редактора текста выводятся синие маркеры. Что они обозначают?

— Каждый маркер отмечает строку программы, в которой выполняется конкретное действие. Маркерами, например, помечены все операторы присваивания, а строки с описанием переменных не помечены, ведь никакой реальной работы в этих строках не происходит.

— Один из маркеров отмечает конец логического блока — закрывающую фигурную скобку. Не странно ли это?

— В этом месте действительно выполняются невидимые для программиста действия, связанные с удалением из памяти компьютера всех переменных, описанных в данном блоке, и еще с целым рядом системных операций Windows, завершающих обработку события (в нашем случае событием было нажатие на кнопку).

Работа калькулятора прервана, но не завершена. Нажмите клавишу F9, и на экране появится диалоговое окно, информирующее о возникшей исключительной ситуации — Floating point division by zero (Деление числа с плавающей запятой на ноль), представленное на рис. 24.

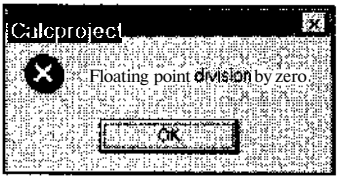


Рис. 24, Это сообщение об ошибке выдает уже не система программирования, а сама тестирующая программа

— А зачем нам еще одно сообщение о той же самой ошибке?

— Все очень просто. В первый раз сообщение выдала система C++Builder только для нас, как для разработчиков. Оно было подробным. А во второй раз сообщение с краткой информацией было выдано непосредственно самой программой-калькулятором. То есть, в нее, оказывается, уже автоматически встроен программный код, защищающий от большинства стандартных исключительных ситуаций. Это означает, что если пользователь программы будет запускать Calcexe не из среды C++Builder, а просто из Windows, то при попытке ввода неверного значения он увидит только второе, краткое сообщение об

ошибке. При этом оператор, попытавшийся выполнить недопустимое действие, просто будет пропущен.

Похожее сообщение можно получить и если ввести в поле текущего курса не чисто, а текст (например, слово двадцать). Тогда в сообщении об ошибке будет написано "двадцать" is not a valid floating point value ("двадцать" — неверное число с плавающей запятой).

Закройте диалоговое окно с сообщением, закройте калькулятор и вернитесь в C++Builder.

Проверяем значения переменных

На этапе отладки C++Builder позволяет быстро выяснить, где встретился ошибочный оператор. Однако хорошо бы еще посмотреть, какие значения в этот момент хранились в переменных, чтобы определить возможную причину ошибки.

Сделать это очень просто. Надо самому прервать работу калькулятора непосредственно перед тем, как будет выполнен ошибочный оператор, и выяснить, какие конкретно значения записаны в переменных. Для этого переместите курсор на нужную строку и нажмите клавишу F5. Строка подсветится красным цветом, а слева появится маркер в виде красного кружка.

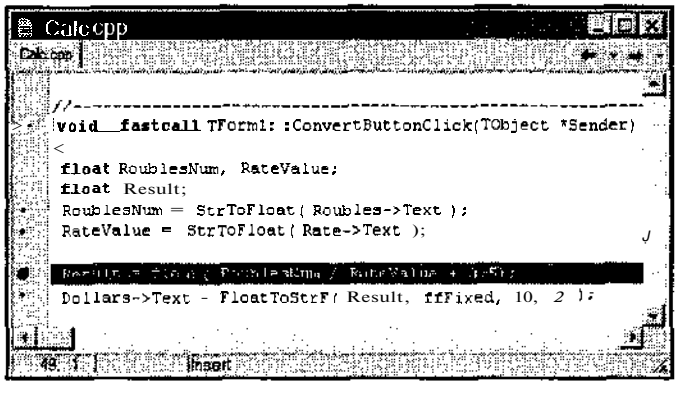


Рис. 25. Так выглядит точка остановки в тексте программы

Этим приемом мы задали своей программе точку остановки *(Breakpoint)*. Теперь, как только подойдет очередь выполнения оператора в помеченной строке, работа калькулятора временно прервется, а мы сможем изучить состояние переменных.

Точки остановки можно задавать только для тех строк, которые отмечены синими маркерами, показывающими, что в данной строке выполняется конкретное действие.

Точек остановки в программе может быть сколько угодно.

Убрать точку остановки можно повторным нажатием клавиши F5 над ранее выделенной строкой.

Запустите калькулятор, введите 1000 (рублей) и щелкните на кнопке пересчета в доллары. Работа программы будет прервана, откроется окно текстового редактора, и мы увидим оператор Си++, который должен быть выполнен в следующий момент. Он отмечается зеленой стрелкой.

В подсвеченном операторе наведите указатель мыши на переменную RateValue, и на экране сразу всплывает подсказка, показывающая текущее значение этой переменной (см. рис. 26).

Не правда ли, это удобно? Сразу видно, что в переменной RateValue хранится ноль, почему при попытке выполнить помеченный оператор и возникнет исключительная ситуация.

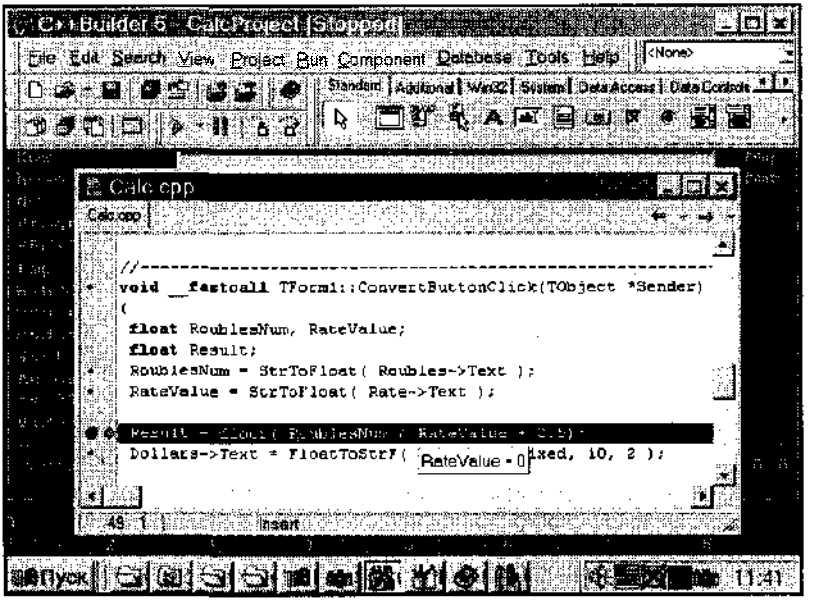


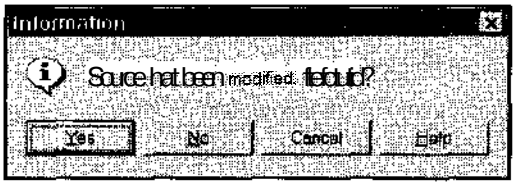
Рис. 26. Всплывающая подсказка — удивительно удобный способ узнать текущее значении любой программной переменной

Останавливаем программу

Продолжать выполнение программы далее не имеет смысла. Чтобы завершить ее работу, находясь в редакторе и выполняя отладку, достаточно нажать комбинацию клавиш CTRL+F2 (или выполнить команду меню Run • Program Reset), и C++Builder корректно закроет окно калькулятора, выполнив все необходимые действия по освобождению памяти и других системных ресурсов. Теперь вы можете внести в текст программы различные изменения.

— А можно ли менять исходный текст, пока калькулятор работает?

— Если в редакторе внести любые изменения в исходный текст программы до того, как она закончит работу, C++Builder предупредит о том, что исходные тексты были модифицированы, и спросит, надо ли выполнить их компиляцию заново.



Если ответить отказом — No (Нет), то выполнение программы будет отложено, однако, скорее всего, возникнет несоответствие между расположением операторов Си++ в редакторе и реальным порядком их выполнения. То есть, если вставить в текст программы новые операторы во время работы приложения или изменить текущие операторы, то C++Builder не поймет, что эти изменения были сделаны. Чтобы система восприняла модификацию кода, текст программы предварительно надо обязательно перекомпилировать, для чего работу приложения необходимо прервать.

Никогда не вносите изменения в текст работающей программы. Сначала прервите ее работу комбинацией клавиш CTRL+F2.

4. Оптимизация текста программы

Выбрасываем лишние операторы

Как уже говорилось, компилятор выполняет оптимизацию программы, удаляя из нее ненужные команды и выполняя множество других полезных действий. Но некоторые части кода, связанные с плохо продуманной логикой реализуемого алгоритма, он улучшить не способен.

Давайте посмотрим на код, который обрабатывает нажатие кнопки, переводящей доллары в рубли. В строке, где вычисляется значение результирующей переменной Result, вместо переменной DollarNum можно сразу подставить вызов функции StrToFloat (ее значение было предварительно сохранено в переменной DollarsNum). Так же можно поступить и с RateValue, заменив ее на нужный вызов стандартной функции. Переделанный оператор будет выглядеть так:

```
Result = floor( StrToFloat( Dollars->Text ) * StrToFloat
  ( Rate->Text ) + 0.5 );
```

Но тогда предварительные операторы вызова функций StrToFloat не нужны — их можно смело убрать! Итоговый текст примет такой вид:

```
float DollarsNum;
float RateValue;
float Result;
Result = floor( StrToFloat( Dollars->Text ) * StrToFloat
  ( Rate->Text ) + 0.5 );
Roubles->Text = FloatToStrF(Result, ffFixed, 10, 2);
```

Компилятор выдает предупреждение

Откомпилируйте программу (клавишу CTRL+F9). Ошибок быть не должно, но в строке Done: диалогового окна компиляции выведется сообщение There are warnings (Имеются предупреждения). Почему же они появились?

Чтобы C++Builder выдавал все предупреждения о возможных неточностях в программе, необходимо выполнить команду Project • Project Options (Проект • Параметры проекта), и на вкладке Compiler (Компилятор) в разделе Warnings (Предупреждение) установить переключатель в положение All (Все).

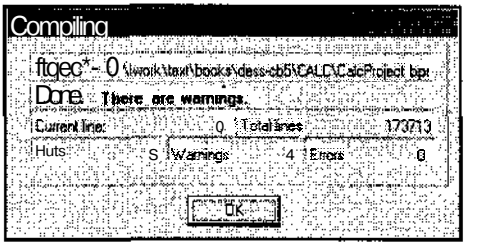


Рис. 27. Строка Warning свидетельствует о предупреждениях

Закройте диалоговое окно компиляции. В нижней части окна редактирования появилась новая панель, в которой выводится информация о найденных ошибках, предупреждениях и подсказках (см. рис. 28).

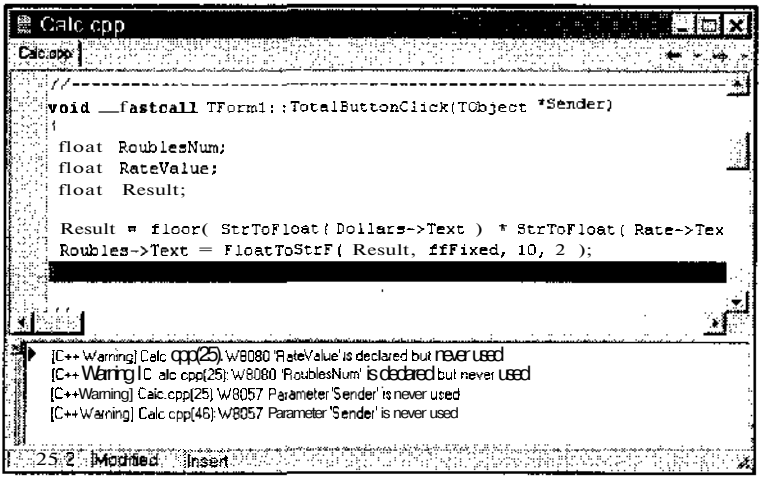


Рис. 28. Исследуем источники предупреждений

Дважды щелкните на первой строке предупреждения — курсор в текстовом редакторе переместится на строку, к которой данное предупреждение относится. Эта строка подсветится бордовым цветом.

Сообщение "RateValue" is declared but never used переводится как Переменная "RateValue" объявлена, но не используется. Действительно, это так, и строку с

определением RateValue (как и строку с определением DollarsNum) можно из текста удалить. Но почему компилятор указал вам именно на строчку с заключительной фигурной скобкой? Потому что здесь кончается логический блок, в котором переменная RateValue была определена. До конца блока она нигде ни разу не использовалась, а за фигурной скобкой такой переменной в программе уже не будет существовать.

В списке предупреждений есть еще два одинаковых сообщения «Parameter "Sender" is never used» (Параметр Sender не используется). Действительно, параметр Sender в каждой из функций TotalButtonClick и ConvertButtonClick внутри них нигде не применяется. Этот параметр описывает объект программы, от которого данное сообщение получено, в нашем случае кнопки, и является стандартным параметром большинства автоматически сгенерированных C++Builder функций. Эти сообщения можно игнорировать или даже отключить их вывод установив переключатель в настройках проекта на вкладке Compiler в разделе Warnings в положение Selected (Избранные) и пометив с помощью кнопки Warnings в большом списке только нужные предупреждения. Правда, при этом можно не заметить и собственную ошибку, например, когда полностью написанная программистом функция не обрабатывает все свои параметры.

Удалите строки описания, останется следующий текст:

```
float Result;
Result = floor( StrToFloat( Dollars->Text ) *
 StrToFloat( Rate->Text > + 0.5 );
Roubles->Text = FloatToStrF(Result, ffFixed, 10, 2);
```

Теперь вновь откомпилируйте программу — останутся только два предупреждения о параметре Sender.

А нельзя ли еще сократить число операторов? Ведь переменная Result используется только для промежуточного хранения результата. Можно попробовать заменить **Result** в последнем операторе на вызов функции floor():

```
Roubles->Text = FloatToStrF( floor( StrToFloat
  (Dollars->Text) * StrToFloat(Rate->Text) + 0.5),
  ffFixed, 10, 2);
```

Тогда переменная Result тоже становится не нужна, и ее описание вместе с оператором присваивания из текста программы надо выбросить.

Таким образом можно (но не нужно! почему — читайте в следующем разделе) укоротить код и другого обработчика события — когда пользователь жмет на кнопку преобразования рублей в доллары. Тогда вся программа

займет всего две (!) строчки на Си++, и при этом окажется в ряде случаев довольно полезной, будет симпатично выглядеть и сможет выполнять весьма сложный процесс преобразования данных.

Всегда ли надо экономить?

Действительно, выигрываете ли вы (и если выигрываете, то что), оптимизируя код программы подобным способом и «запихивая» всю логику работы в один-два оператора Си++? Многолетняя история и практика программирования показывает, что подобный подход к разработке приложений, позволяя в ряде случаев выиграть несколько процентов производительности, резко — в десятки раз — повышает время на создание готового продукта и значительно ухудшает его качество. Почему? Сравните новый код (одну строчку) со старым. Что делает единственный и очень длинный оператор присваивания, сразу понять трудно. Легко запутаться в параметрах функций, определить, где, что и в каком порядке вычисляется. А при просмотре прежнего кода, когда использовались наглядные названия переменных (RoublesNum, RateValue), можно быстро проследить логику программы.

Особенно важно писать ясный, понимаемый «на лету» код (что само по себе является признаком мастерства), когда над проектом трудится несколько разработчиков и часто возникает необходимость разобраться в чужих текстах. Кроме того, если бы вы с самого начала стали писать код реакции на нажатие кнопки в одном операторе, то непременно запутались бы в круглых скобках и наделали ошибок.

Никогда не усложняйте программу! Пишите код как можно нагляднее, не экономьте операторы и активно используйте названия переменных, несущие смысловую нагрузку.

Не тратьте слишком много времени на ручную оптимизацию текста на Си++. В лучшем случае вы повысите быстродействие работающего приложения на 5-10 процентов, а потратите на этот процесс довольно много времени. Время лучше расходовать не на ликвидацию нескольких на первый взгляд лишних операторов, а на совершенствование общего алгоритма вашего приложения. Есть такой золотой принцип разработки программ — **экономьте не пять операторов, а пятьдесят!** Улучшайте не код, а алгоритм — смысловую часть, на высоком уровне, на этапе проектирования, не привязанном к ручному кодированию. Удаление одного лишнего логического блока может дать значительно более существенную экономию, чем удаление десяти операторов Си++. При этом время разработки сокращается, а качество программы только повышается.

5. Обработка исключительных ситуаций

Охота за ошибками

Ошибок типа деления на ноль теперь можно не бояться. Осталось только добавить проверку текстовых строк, вводимых в поля, дабы убедиться, что в них записаны именно числа, а не произвольный набор символов, который невозможно преобразовать в тип float. Проверить это можно разными способами, например использовать подходящую стандартную функцию. Однако на все случаи жизни стандартных функций не напасешься, да и программисты часто забывают или ленятся вставлять соответствующие проверки. Поэтому для отслеживания и контроля за всевозможными непредвиденными и неприятными происшествиями в Си++ существует очень мощный механизм обработки исключительных ситуаций. Он позволяет указать программе, что надо сделать, когда в ней возникает та или иная ошибка — например, деление на ноль, неправильное приведение типов или попытка преобразовать набор букв в число.

Устанавливаем над программой контроль

Прежде всего необходимо определить, в каких операторах программы возможно возникновение исключительных ситуаций. В случае с калькулятором — это два оператора преобразования введенного в поля Roubles и Rate текста в числа с помощью функции StrToFloat(). Если в качестве ее параметра указать, например, «десять», то возникнет исключительная ситуация с типом EConvertError.

Для каждой исключительной ситуации определен свой класс, который начинается с буквы E (от английского Exception — прерывание). В системе C++Builder 5 насчитывается несколько десятков таких классов.

Узнать, возникает ли при выполнении различных стандартных функций какая-то исключительная ситуация, можно в справочной системе (Help). Найдите краткое описание StrToFloat() — там говорится, что если параметр этой функции некорректен, то возникает ситуация EConvertError.

Когда никакая реакция на те или иные исключительные ситуации в программе явно не определена, то при их возникновении вызывается стандартный обработчик исключений. Так было, в частности, когда калькулятор выдавал окно с сообщением о попытке деления на ноль.

Не для всех стандартных функций определены обработчики исключительных ситуаций. Некоторые функции, если им в качестве параметров переданы неверные величины, просто возвращают предопределенное значение, сигнализирующее об ошибке. Например, если функция FloatToStrF() по каким-то причинам не сможет преобразовать число в текст, то она может вернуть в качестве результирующей строки "NAN".

Теперь можно установить над процессом преобразования строки контроль со стороны программиста. Для этого надо выделить операторы в логический блок с помощью фигурных скобок, а перед этим блоком указать ключевое слово try, которое в данном случае можно приблизительно перевести как «попробовать выполнить»:

```
try
    {
    Roubles->Text = FloatToStrF(
      floor(StrToFloat(Dollars->Text) *
      StrToFloat(Rate->Text) + 0.5), ffFixed, 10, 2);
    }
```

Сразу за проверяемым блоком надо указать ключевое слово `catch` *(поймать),* вслед за которым в круглых скобках записывается такая конструкция:

```
catch (const название-ситуации &)
```

В рассматриваемом случае она будет выглядеть так:

```
catch (const EConvertError &)
```

Таким способом указывается, что в данном месте будет располагаться оригинальный (написанный программистом) обработчик ситуации **EConvertError.** Этот обработчик следует далее в отдельном логическом блоке:

```
try
    {
    Roubles->Text = FloatToStrF(
        floor( StrToFloat( Dollars->Text ) *
        StrToFloat( Rate->Text ) + 0.5 ),
        ffFixed, 10, 2 );
    }
catch (const EConvertError &)
    {
    /* Здесь надо вставить код обработки неверного
    преобразования*/
    }
```

А как же лучше обработать пойманную ошибку? В случае с калькулятором ничего хитроумного придумывать не надо. В принципе, было бы достаточно стандартного обработчика, но сообщения на английском не всем бывают понятны, поэтому можно просто вывести сообщение на русском языке о том, что в одно из полей введена строка, которую невозможно корректно преобразовать в число.

Вывести такое сообщение с помощью диалогового окна Windows можно, используя стандартную функцию ShowMessage() — ей в качестве единственного параметра передается показываемая строка.

Строки текста, которые должны выводиться на экран, в языке программирования Си++ заключаются в парные кавычки. Например:

```
"это допустимая строка Си++"
```

Строка может быть пустой, то есть не содержать на одного символа:

```
""  (не содержащая ни одного символа, пустая строка)
```

Иногда внутри текстовой строки надо использовать свои кавычки, например для вывода цитат. Чтобы внутренние кавычки отличались от внешних, перед ними надо ставить символ \ (обратная косая черта):

```
"внутри этой строки \"есть фрагмент текста,
  заключенный в кавычки\". "
```

Теперь, зная как в С++ происходит вывод текстовых сообщений, мы можем добавить в оригинальный обработчик один оператор — вызов функции ShowMessage():

```
catch (const EConvertError &)
    {
    ShowMessage("В одно из полей введены нечисловые
      данные");
    }
```

Порядок обработки исключительной ситуации

Откомпилируйте программу и попробуйте запустить ее, например, из Проводника — не из C++Builder (чтобы не возникало дополнительного прерывания работы программы — и так известно, где оно произойдет). Теперь, если ввести в поле Доллары текстовую строку (не число) и нажать на кнопку пересчета в рубли, то калькулятор выдаст сообщение, представленное на рис. 29.

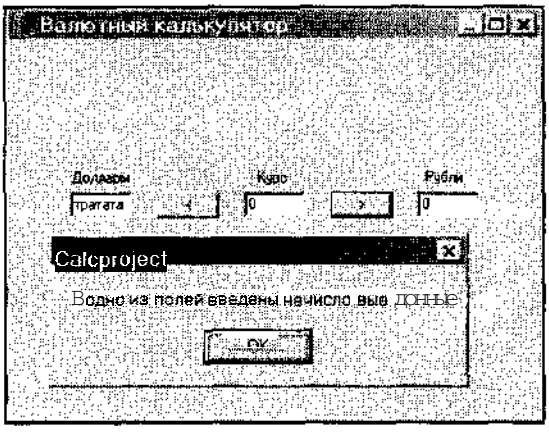


Рис. 29. Сообщение о том, что в числовое поле введено не числовое значение

Что при этом происходит внутри программы? Когда функция StrToFloat() в вызове StrToFloat(Roubles->Text) пытается преобразовать в число некорректную строку, возникает исключительная ситуация EConvertError. Так как функция StrToFloatO вызвана в блоке, который проверяется на наличие подобных ситуаций (это определено ключевым словом try), то программа проверяет, имеется ли в ней блок нестандартной обработки ситуации EConvertError (что определяется ключевым словом catch). Если такой блок найден, то управление сразу передается на него — все остальные операторы в блоке try *пропускаются.*

Если же никаких исключительных ситуаций в контролируемом с помощью оператора try блоке не появилось, то блок обработки этих ситуаций, записанный следом за catch, не выполняется. Он предназначен *только* для реагирования на ошибки. Управление при этом передается на первый оператор, следующий за catch-блоком.

Обрабатываем несколько исключительных ситуаций

— А можно ли с помощью собственного обработчика отловить попытку ввода слишком больших чисел?

— Да, можно. Си++ позволяет применять несколько catch-блоков, каждый из которых определяет способ обработки своей конкретной ситуации. Вслед за уже имеющимся catch-блоком можно добавить еще один (число таких *перехватчиков* не ограничено), только вместо названия класса EConvertError необходимо указать другой класс, например, EOverflow (контроль за использованием слишком больших чисел с плавающей запятой), и показывать сообщение с другим текстом. Весь код будет выглядеть так:

```
try
    {
    Roubles->Text = FloatToStrF(
      floor( StrToFloat( Dollars->Text ) *
      StrToFloat( Rate->Text ) + 0.5 ),
      ffFixed, 10, 2 );
    }

catch (const EConvertError &)
    {
    ShowMessage("В одно из полей ведены нечислозые
      данные");
    }

catch (const EOverflow &)
    {
    ShowMessage("В одно из полей введено слишком
      большое число");
    }
```

Если теперь запустить калькулятор и в поле Доллары ввести 1e1000 (число с тысячью нулей), а в поле Курс ввести число 1, то при щелчке на кнопке пересчета в рубли программа выдаст именно то сообщение, которое и требуется при обработке ситуации, связанной с использованием слишком больших чисел *(переполнением)*.

Вне зависимости от того, какой catch-блок был выполнен, управление будет передано первому оператору, следующему за последним catch-блоком, принадлежащем соответствующему оператору try.

6. Классы и их методы

Методы вместо функций

До сих пор мы преобразовывали числа в строки с помощью стандартной функции StrToFloat(). Однако такое преобразование можно осуществить и другими способами. Например, это способна сделать стандартная функция atoi(), перешедшая в Си++ в качестве наследства из Си, однако ее использование будет не совсем элегантно. Дело в том, что в Си не было типа данных «строка», а текстовые данные представлялись в виде простой последовательности символов, заканчивающейся нулевым значением. Для работы с такими последовательностями в Си лет двадцать назад была создана специальная библиотека (входящие в нее функции описаны в файле string.h), однако подобный подход к обработке строковых данных сегодня устарел, а программистам требуются новые, более мощные и более совершенные средства манипулирования текстовой информацией, позволяющие отвлечься от представления строки в виде последовательности символов и дающие возможность работать с ней как с законченным понятием.

В C++Builder текстовые строки имеют тип AnsiString. Это стандартный класс, который позволяет не только преобразовывать содержимое строк в вид, пригодный для обработки старыми стандартными функциями, но и выполнять над ними еще множество действий, используя более привычные человеку способы.

В C++Builder содержимое текстовой строки может иметь длину до 3 Гбайт, хотя, конечно, в каждом конкретном случае ей отводится не столько места, а сколько реально необходимо.

Для всевозможных преобразований и проверок содержимого классов в Си++ есть специальная и очень удобная возможность — это так называемые *методы* классов. Вы помните обращения к свойствам полей: заголовку (Caption), названию (Name), содержимому (Text)? Это все были свойства

соответствующих классов **TButton** или **TEdit**, а на самом деле эти свойства представлены скрытыми внутри данных классов переменными.

Так вот, помимо переменных, в классах скрыты и функции, позволяющие что-то сделать с этими переменными: изменить их, задать новые значения, а также *выполнить действия, связанные с логикой работы этих классов*. Такие функции, принадлежащие классу, и называются *методами*. Например, для класса Окно вполне естественно наличие методов Открыть и Закрыть. Класс Кнопка, очевидно, имеет метод Нажать. Для всех видимых элементов управления обязательно определен метод Нарисовать, который и задает, как конкретный элемент будет выглядеть на экране.

Вызов метода записывается в Си++ достаточно естественно. Команда Открыть окно А будет выглядеть как **A.open()**, где операция " . " связывает метод и переменную и указывает, что метод open вызывается конкретным экземпляром (переменной А) класса Окно. По другому (без указания их «хозяина», как обычные стандартные функции) методы вызывать нельзя.

Некоторые методы могут возвращать значения. Например, команда A. GetWidth () должна вернуть число, характеризующее ширину окна А.

Значение, возвращаемое функциями Си++, можно игнорировать. Многие стандартные функции помимо выполнения требуемых от них действий также возвращают некоторые величины, чаще всего это признак, характеризующий успешность действия. Так, команда A.open() помимо того, что попробует открыть окно А, вернет логическое значение true, если открыть А удалось, или значение false, если открыть А не получилось. Можно написать

```
ErrorCode = A.open();
```

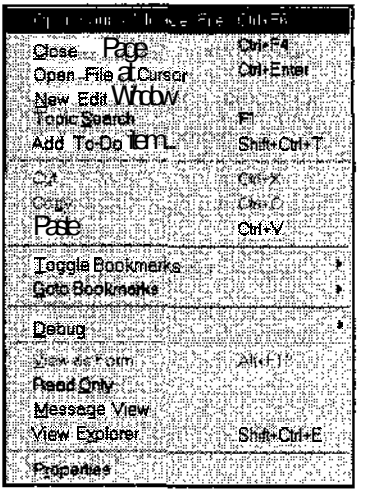
А можно и так:

```
A.open();
```

В этом случае разработчика не волнует, открылось ли окно А в действительности (хотя это и неправильно), и он не намерен сохранять результат открытия в переменной, чтобы потом этот результат проверить.

Как найти нужный метод

— А как найти нужный метод, преобразовывающий, в частности, строку в число, и как вообще определить, к какому классу принадлежит переменная, включенная в программу автоматически? Возьмем, например, переменную Roubles->Text. Как определить ее тип?



Все описания объектов, расположенных на проектируемой форме, находятся в заголовочном файле, название которого совпадает с названием файла с текстом программы, а в качестве расширения используется .H. В нашем случае описание переменной Roubles можно найти в файле Calc.h. Чтобы быстро перейти к нему в редакторе, достаточно вызвать контекстное меню и выбрать пункт Open Source/Header File (Открыть исходный/заголовочный файл) или нажать комбинацию клавиш CTRL+F6.

В файле Calc.h несложно обнаружить такую запись:

```
TEdit *Roubles;
```

То есть, переменная (объект) Roubles имеет тип TEdit (компонент Edit). На самом деле Roubles — это так называемый указатель на TEdit, что определяется символом «*». Об указателях будет рассказано позже.

Далее с помощью справочной системы C++Builder надо найти определение класса TEdit и в списке его свойств (Properties) выбрать Text. Там окажется следующее определение:

```
__property AnsiString Text = {read=GetText,
  write=SetText};
```

Ключевое слово__property переводится как «свойство», на последнюю часть описания Text прикладному разработчику можно не обращать внимания (она нужна создателям собственных компонентов). Основным здесь является тип — AnsiString. Перейдем к его описанию щелчком мыши на нем (на следующей странице справочной системы будет сказано, что AnsiString — это класс), после чего надо щелкнуть на разделе Methods (Методы), чтобы получить полный список всех методов класса AnsiString. Далее поиск нужного метода происходит экспериментальным способом — либо просмотром и ознакомлением со всеми методами подряд, либо поиском подходящего метода по названию — разработчики C++Builder стараются делать их достаточно наглядными и понятными.

По результатам просмотра нетрудно установить, что на роль нужного преобразователя претендует только метод ToDouble (его название можно перевести как ...в число с плавающей запятой). Действительно, в описании ToDouble сказано, что он предназначен для преобразования содержимого класса

AnsiString (строки) в число с плавающей запятой. Теперь можно вызов функции StrToFloat() заменить на метод **ToDouble()**:

```
RoublesNum = Roubles->Text.ToDouble();
```

Если преобразование выполнить не удастся, то возникнет уже знакомая исключительная ситуация EConvertError (о ней рассказано в описании метода **ToDouble()**), и для ее обработки потребуется организовать try- и catch-блоки.

Закрыть любой файл, открытый в текстовом редакторе, можно стандартной комбинацией клавиш CTRL+F4.

Дедовский способ

— А как бы программисты, использующие старую версию языка Си, выполнили проверку того, что введенная информация представляет собой корректную запись числа?

— Они бы сделали это проверенным дедовским способом — так же, как похожие задачи решались десятки лет назад. В классе AnsiString есть метод **c_str()**, который позволяет обращаться к содержимому переменной типа AnsiString как к простой последовательности символов с нулем в конце, в соответствии с требованиями языка Си, благодаря чему с этим содержимым можно работать с помощью стандартных функций тридцатилетней давности. Среди них есть функция atof(). Ее можно найти, если в разделе Указатель справочной системы C++Builder просмотреть список разделов, начинающихся со слова `string` (строка). Там можно быстро обнаружить рубрику string to floating-point conversions (преобразование строки в число с плавающей запятой) и сразу попасть на определение **atof()**. Эта функция в качестве своего значения получает адрес *первого символа* (физические координаты его расположения в оперативной памяти компьютера) строки AnsiString (такой адрес и возвращается методом c_str()) и преобразовывает строку в число типа double.

Используя эту функцию, преобразование строки теперь можно записать так (не вносите никаких изменений в программу — это только пример):

```
RoublesNum = atof( Roubles->Text.c_str() );
```

и калькулятор будет нормально работать.

Но как же отловить ошибку в случае ввода неверного значения? При работе функции **atof**() в случае ошибки никаких исключительных ситуаций не возникает — в языке Си их еще не было. Для контроля за результатом выполнения этой функции предусмотрена глобальная переменная **errno**. Только,

чтобы ее можно было использовать, потребуется добавить в текст программы командную строку включения соответствующего заголовочного файла errno.h, в котором хранится описание этой переменной.

В случае, если функция atof() обнаружит, что выполнить требуемое от нее преобразование она по каким-то причинам не может, в переменную errno занесется значение константы ERANGE. На всякий случай перед вызовом функции atof() в переменную errno надо записать другое значение (например, EZERO), не совпадающее с ERANGE, иначе определить, изменилось ли в действительности значение errno на ERANGE, не удастся.

Переменную errno нельзя использовать для контроля за правильностью работы большинства стандартных функций Си++. Случаи, когда применение переменной errno допускается, особо оговаривается в справочной системе C++Builder.

После того, как функция atof() отработает, переменную errno надо будет сравнить с ERANGE. Если их значения совпадут, надо вызвать диалоговое окно с информацией о том, что в конкретное (уже известное — Roubles) поле введены неверные данные, и пропустить все оставшиеся операторы.

Но тогда полная проверка корректного преобразования строк потребует уже десятков строк на Си++, и размер текста программы будет стремительно увеличиваться (а если таких полей ввода будет сто?), причем за счет второстепенных и вспомогательных действий, задумываться над программированием которых разработчик по большому счету не должен.

Всегда используйте систему обработки исключительных ситуаций Си++ вместо самостоятельного программирования действий по выявлению возможных ошибок при выполнении стандартных функций и методов классов.

7. Условные вычисления

Фильтры значений

Причину ошибки мы нашли — это деление на ноль. Конечно, хорошо, что C++Builder автоматически проверяет наличие исключительных ситуаций, однако на все случаи жизни таких ситуаций не напасешься. Хорошо бы самостоятельно проверять допустимые значения вводимых пользователем переменных — ведь можно, например, в качестве обменного курса указать -2, и калькулятор будет работать корректно, однако он станет выдавать отрицательные значения, что логически совершенно неправильно. Одно дело, когда сразу видно, что введено неверное в смысловом плане значение, и другое дело, когда вводимое значение может использоваться в программе для сложных вычислений. Например, если указать отрицательный курс доллара в бухгалтерском приложении, это может привести к полной остановке работы всей бухгалтерии и самым печальным последствиям.

Кстати, именно из-за отсутствия проверок в известных программах на допустимые диапазоны значений хакерам и удается их взламывать и находить «дыры» в защите. Например, в первых версиях Microsoft Internet Explorer 4.x была найдена довольно оригинальная ошибка: если длина строки с адресом интернетовского Web-узла превышает 255 символов (в Си++ есть тип unsigned char с диапазоном значений от 0 до 255; переменная этого типа, видимо, и использовалась для хранения длины строки), то остаток этого адреса может интерпретироваться Windows как программный код, который потенциально можно (хотя и очень трудно) сделать вредным.

Профессионально сделанные коммерческие программы не позволят вам ввести недопустимые по логике их работы значения (хотя, как вы только что могли убедиться на примере, так бывает не всегда). Для этого программисты применяют так называемые *фильтры,* проверяя вводимые величины

на соответствие допустимым диапазонам и показывают предупреждения в случае некорректного ввода.

Активно используйте в своих программах фильтры. Периодически проверяйте промежуточные значения важнейших переменных, в том числе и не обязательно вводимых пользователем. Делайте это, даже если кажется, что никакого отклонения от заданного диапазона в конкретном месте произойти не должно.

Фильтры помогают избежать множества самых разных и неожиданных ошибок. Использование фильтров называется *защитным* программированием.

Неплохо бы и нам проверять, что вводится в поля калькулятора. Это должны быть не отрицательные числа (а в случае курса и не равные нулю), а диапазоны их значений желательно сделать разумными — не очень большими. Надо также проверить, не вводит ли пользователь вместо чисел текстовые строки.

Условный оператор

Запись условного оператора

Для подобных проверок в Си++ имеется так называемый *условный оператор*. Это второй по важности оператор Си++ (после оператора присваивания), с которым мы сейчас познакомимся.

Записывается он так:

```
if ( условие ) выполняемый-оператор ;
```

Ключевое слово if переводится с английского как *если*. Оператор, следующий за скобками, в которые заключено условие, выполняется только в том случае, *если условие истинно* в соответствии с правилами вычисления логических выражений.

Значением логического выражения может быть либо истина (зарезервированное слово, константа true), либо *ложь* (зарезервированное слово, константа false). Значение константы true — 1, единица. Значение константы false — 0, ноль.

В Си++ есть специальный тип bool, который описывает переменные, способные принимать одно из двух значений true или false (только эти два и больше никакие другие).

— А если надо в случае истинности некоторого условия выполнить не один оператор, а несколько?

— Тогда все эти операторы надо объединить в логический блок с помощью фигурных скобок:

```
if( условие )
   {
   оператор-1 ;
   оператор-2 ;
   // и т. д.
   }
```

Все операторы внутри фигурных скобок будут исполнены, только если проверяемое условие истинно.

Обратите внимание на форму записи условного оператора — логический блок записан с небольшим, в два-три пробела, *отступом*. Это стиль хорошего программирования, позволяющий с первого взгляда понять, что данная группа операторов будет выполнена в зависимости от некоторого условия.

Логические выражения

В простейшем логическом выражении происходит сравнение двух величин (значений переменных или результатов вычисления математических выражений) с помощью следующих операций.

Операция	Обозначение
Равно	==
Не равно	!=
Больше	>
Меньше	<
Больше или равно	>=
Меньше или равно	<=

Обратите внимание на то, что почти все логические операции записываются двумя символами (как вы знаете, вставлять между ними пробел нельзя).

Примеры логических выражений:

```
x > 5
RateValue <= 29.5
2 != 3
(x*3) < (y+50)
```

В последнем случае сравниваются результаты вычислений выражений x*3 и y+50. Эти выражения взяты в скобки, но разрешается написать и так:

```
x*3 < y+50
```

Если переменная x хранит значение 10, а переменная y — 0, то значение выражения x*3 < y+50 (10*3 < 0+50) будет false или 0.

Приоритет всех логических операций *ниже* приоритета всех арифметических операций. Это значит, что сначала выполняются арифметические действия, и только потом — сравнения. Но и приоритеты самих операций сравнения тоже не одинаковы. В первую очередь выполняются проверки с помощью операций >, >=, <, <= и только потом сравнение == и !=.

Как уже говорилось выше, в достаточно сложных и длинных выражениях лучше всегда использовать скобки, наглядно задающие порядок вычислений.

Из простейших логических выражений (два значения и операция их сравнения) складываются более сложные выражения. Допустим, надо проверить, больше ли курс доллара нуля и не превышает ли он 100. В этом нам помогут логические операции И и ИЛИ, обозначаемые соответственно как && и ||. Результат применения операции && возвращает значение true, только если значения и левого, и правого выражений равны true (во всех остальных случаях получится false). Результат применения операции || возвращает значение true, если значение или левого, или правого выражения равны true.

Нужная нам проверка запишется так:

```
RateValue > 0 && RateValue <= 100
```

Приоритет операций && и || ниже приоритета операций сравнения, поэтому отдельные проверки значения RateValue в скобки брать не обязательно.

Если значение RateValue равно 101, то:

- значением выражения (RateValue > 0) станет true;
- а значением выражения (RateValue <= 100) — false.

В итоге результат выражения true && false будет равен false.

Порядок вычисления выражения

— А как будет вычисляться следующее выражение (никогда не используйте таких конструкций, хотя они и допустимы в Си++)?

```
x*3 < y+50 > z/3
```

— Так может написать начинающий программист, чтобы проверить, больше ли y+50, чем x*3, и больше ли y+50, чем z/3. Однако приведенная здесь запись неверна. Прежде всего, здесь не задан явно (с помощью круглых скобок) порядок вычисления, что само по себе является очень плохим стилем программирования. А в C++ принято, что в подобных ситуациях вычисление значения выражения всегда происходит *слева направо.*

Вычисление значения выражения Си++, где порядок выполнения операций с одинаковым приоритетом не указан явно, почти всегда выполняется слева направо. Случаи обратного порядка (справа налево) будут оговариваться в книге особо.

То есть, если переменная x хранит значение 10, переменная y — 0, а переменная z — 120, то сначала в соответствии с более высокими приоритетами будут рассчитаны значения арифметических выражений:

```
10*3 < 0+50 > 120/3
```

или

```
30 < 50 > 40
```

а далее возьмется самое первое слева сравнение (30 < 50) и вычислится его значение, равное true. В итоге получится ужасное и неестественное заключительное сравнение

```
true > 40
```

Однако оно не считается в Си++ ошибкой: как уже говорилось, константа `true` имеет значение 1, то есть выражение

```
true > 40
```

эквивалентно выражению

```
1 > 40
```

значение которого равно `false`(или 0).

Никогда не используйте константы true и false в арифметических операциях вместо 1 и 0! Это ненаглядно, а в ряде ситуаций это может привести к трудно обнаруживаемой ошибке.

Как мы видели, выражения наподобие x*3 < y+50 > z/3 потенциально таят в себе ошибку, да и компилятор выдаст предупреждение Sugest parentheses to clarify precedence (Рекомендуется использовать круглые скобки, чтобы уточнить порядок вычислений). Правильно записать нужное программисту условие можно, например, так:

```
(x*3 < y+50) && (y+50 > z/3)
```

Сложные логические выражения и побочные эффекты

Помимо проверки корректности введенного значения RateValue надо также проверить значения DollarsNum и RoublesNum — не отрицательные ли они.

Это можно сделать так:

```
RateValue > 0 && RateValue <= 100 && RoublesNum >= 0
  && RoublesNum < 100000
```

Если значение RateValue равно 30, а значение RoublesNum равно -100, то указанное выражение превратится в следующее:

```
30 > 0 && 30 <= 100 && -100 >= 0 && -100 < 100000
```

или

```
true && true && false && true
```

которое, при вычислении слева направо, будет последовательно превращаться в

```
(true && true) && false && true
(true && false) && true
false && true
```

и в итоге получится false.

Операции && и || имеют интересную особенность — в некоторый момент вычисления длинного логического выражения часто можно сразу сказать, каким будет результат, не доводя вычисления до логического конца. Как только в рассмотренном выше примере результатом очередной логической операции стало значение false, остальные операции уже роли не играют, поскольку false && false будет false и false && true будет false!

C++Builder — очень умная система. Она успешно распознает подобные ситуации и не будет выполнять лишних действий, экономя время работы. Но при этом возникает потенциальная опасность так называемых *побочных эффектов*.

Если программист напишет выражение:

```
RateValue > 0 && RateValue <= 100 &&
  ChangeRate(RateValue)
```

где ChangeRate() — некая условная функция, увеличивающая значение переменной RateValue на единицу, а значение RateValue к моменту вычисления данного выражения будет равно 110, то программа вычислит промежуточное значение

```
false && ChangeRate(RateValue)
```

и поймет, что дальше ничего делать не надо — все равно false && что-угодно будет равно false. В итоге программа *не вызовет* функцию ChangeRate(). А между тем разработчик рассчитывал, что в данном месте значение RateValue() увеличится на единицу. Однако ничего подобного не произойдет. Поэтому составлять логические выражения надо очень внимательно.

Избегайте использования техники побочных эффектов в своих программах. Эти эффекты — следствие невозможности однозначной и четкой трактовки логики работы различных конструкций Си++, что часто приводит к трудновыявляемым ошибкам.

Создаем фильтр

Теперь мы можем добавить в свою программу фильтр для проверки вводимых значений на их «нормальность». Перейдите к тексту, описывающему реакцию калькулятора на нажатие кнопки пересчета из рублей в доллары и перед вычислением значения Result вставьте условный оператор:

```
if( RateValue > 0 &&
    RateValue <= 100 &&
    RoublesNum >= 0 &&
    RoublesNum < 100000 )
```

(для наглядности его лучше записать в несколько строк), а операторы вычисления Result и вывода значения в поле Dollars надо выделить в логический блок:

```
{
Result = floor( RoublesNum / RateValue + 0.5);
Dollars->Text = FloatToStrF(Result, ffFixed, 10, 2);
}
```

Получится такой текст:

```
iff RateValue > 0 &&
    RateValue <= 100 &&
    RoublesNum >= 0 &&
    RoublesNum < 100000 )
    {
    Result = floor( RoublesNum / RateValue + 0.5);
    Dollars->Text = FloatToStrF(Result, ffFixed, 10, 2);
    }
```

Откомпилируйте программу, запустите ее и попробуйте вычислить сумму в долларах при нулевом или слишком большом (например, 2 00) курсе. Калькулятор никак не отреагирует на такие значения.

Хорошо бы подобную проверку добавить и в код обработки нажатия на кнопку пересчета в рубли, но там вся логика скомпонована в один-единственный оператор, и выполнить соответствующие проверки немного сложнее. Теперь вы понимаете, почему программу надо записывать как можно более просто и наглядно?

Конечно, условные операторы применяются не только для использования в фильтрах. На их основе строится вся логика работы любой маломальски сложной программы.

Сравнение чисел с плавающей запятой

Очень осторожно надо сравнивать числа с плавающей запятой. Как уже говорилось, в отличие от целых чисел в программе они представляются не точно, а приближенно, поэтому прямое их сравнение с помощью операции == может не дать желаемого результата. Например:

```
float x, y, z;
z = 0;
x = 3.1415927;
y = 3.1415927;
if( x == y ) 2 = 10;
```

Не исключено, что переменная *г* не получит значения 10, так как во время выполнения указанных операторов в x может быть занесено **3.1415926,** а **в** y — **3.1415929.**

В таких ситуациях надо сравнивать не сами числа, а проверять, насколько сильно они отличаются друг от друга. Для этого просто берется разность двух переменных и выясняется, не превышает ли она некоторого порога, меньше которого разницу в значениях можно не учитывать (в зависимости от задачи этот порог может колебаться). Только надо брать не простую разность переменных, а ее *модуль* (понятие из школьного курса, *абсолютное положительное значение;* модуль числа -2 равен +2, модуль числа -0,001 равен +0,001, модуль числа +3,14 равен +3,14). Ведь разница 4-5 равна -1, а −1 всегда меньше любого сколь угодно малого положительного числа.

Для вычисления модуля математического выражения Си++ применяется стандартная функция abs().

В нашем случае записать такую проверку с учетом вышесказанного можно, например, так:

```
if( abs( x-y ) < 0.0001 ) z = 10;
```

В этом случае переменная z получит значение, равное 10, если переменные x и y равны друг другу с точностью до 0,0001. Это стандартный прием сравнения действительных чисел. Он отнюдь не относится только к С++, а применяется и в большинстве прочих языков программирования.

8. Играем с компьютером. Более сложный пример

Во что будем играть?

Программа-калькулятор работает. Она не позволяет вводить неверные значения, отслеживает попытки деления на ноль и вполне может оказаться полезной. При этом пока были затронуты далеко не все возможности Си++. Для дальнейшего изучения этого языка попробуем написать программу, с которой можно в свободное время поиграть в довольно простую, но увлекательную игру в кости. Придумал эту игру француз Ж.-К. Бейиф, а правила ее таковы.

1. Играют двое (в нашем случае это будет человек и компьютер, а точнее говоря — программа).
2. Исходно у каждого игрока 0 очков.
3. Ходят по очереди.
4. На своем ходу игрок бросает кость (игральный кубик).
5. Если выпадает число от 2 до б, то оно записывается игроку в очки, накопленные на данном ходу. Далее игрок решает, будет ли он продолжагъ ход. Если он отказывается, то накопленные им на данном ходу очки прибавляются к его общему активу. Если он решает продолжить текущий ход, то снова бросает кубик.
6. Если выпадает число 1, то все накопленные игроком на данном ходу очки теряются, и очередь хода передается противнику.
7. Выигрывает тот, кто первым наберет как минимум 100 очков.

При разработке такой программы потребуется не только создать удобный интерфейс, но и написать на Си++ алгоритм работы компьютерного оппонента. Хотя эта игра на первый взгляд достаточно проста, ее оптимальная стратегия не так тривиальна, как может показаться.

Готовим новый проект

Проект, связанный с созданием валютного калькулятора, теперь можно закрыть — в прямом и переносном смысле. Впоследствии вы сами всегда сможете к нему вернуться и попробовать еще что-то усовершенствовать.

Закрывается текущий проект CalcProject (его название написано на заголовке главного окна C++Builder) командой File ▸ Close All (Файл • Закрыть все). При этом, если в проект вносились какие-то изменения, C++Builder спросит, надо ли их сохранить (см. рис. 30). В таких случаях лучше всегда отвечать утвердительно (Yes).

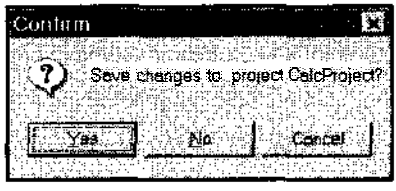


Рис. 30. Закрытие предыдущего проекта

Окна визуального проектировщика форм и редактора закроются. А чтобы создать новый проект, надо выполнить команду File • New Application (Файл ▸ Новое приложение). После этого визуальная оболочка примет вид, в точности совпадающий с тем, который возник при первом запуске C++Builder. Этот новый проект можно сразу сохранить, щелкнув на уже известной командной кнопке Save All (Сохранить все). Только для нового проекта надо организовать и новый каталог, чтобы файлы нового проекта не сваливать в одну кучу с файлами старых проектов.

Для наглядности файл программы на Си++ назовем DiceUnit.cpp, а файл проекта — DiceProject.bpr.

Постановка задачи

Не спешите сесть за компьютер!

Не надо сразу стараться поскорее начать программировать. Предварительно обязательно обдумайте внутреннюю структуру будущего приложения, логику его работы и пользовательский интерфейс. При создании больших программных комплексов применяют особые системы, позволяющие выполнять этап проектирования с помощью различных визуальных редакторов, но такие системы достаточно сложны в изучении, и при разработке программ малого и среднего объема удобнее всего обходиться простыми листами бумаги.

Согласно статистике трудоемкость устранения ошибок, допущенных на первом этапе проектирования, по мере приближения проекта к завершению возрастает в сотни раз! Например, в калькулятор всегда можно быстро вставить несколько операторов для проверки допустимых значений, а вот если захочется обрабатывать пересчет не только из рублей в доллары и наоборот, а поддерживать и другие валюты, то может оказаться, что переписывать придется всю программу полностью, с самого начала.

Проектирование пользовательского интерфейса

Во многих случаях наиболее удобно начинать создание программы с проектирования ее интерфейса — размещения элементов управления в будущем главном окне (или окнах, если их планируется несколько).

Для игры потребуются следующие элементы управления.

1. Строка меню с пунктами Игра (подпункты Новая игра и Выход) и Ход (подпункты Продолжить ход и Передать очередь хода).
2. Панель командных кнопок, соответствующих подпунктам строки меню.
3. Пять полей надписи, в которых будет отображаться следующая информация: сколько всего очков накоплено у человека и компьютера, сколько очков накоплено каждым на своем последнем ходу, сколько в последний раз выпало на кубике.
4. Список, в котором будут последовательно выводиться результаты сыгранных партий.

С полями надписей мы уже знакомы и можем разместить их произвольным образом, например так, как показано на рис. 31.

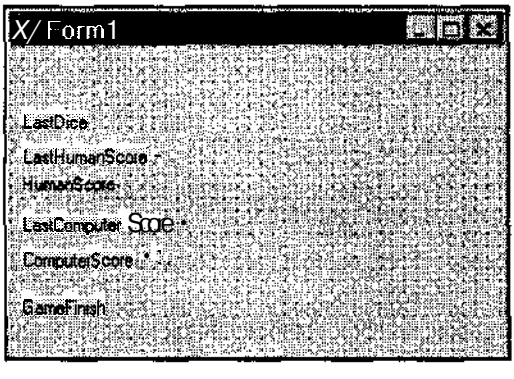


Рис. 31. Поля вывода текущей информации о ходе игры

Назвать эти поля можно заданием их свойства Name, например так:

- HumanScore **(Очки человека);**
- ComputerScore **(Очки компьютера);**
- **LastHumanScore (Очки человека на текущем ходу);**
- LastComputerScore **(Очки компьютера на текущем ходу);**
- **LastDice (Сколько выпало на кубике).**

Для ведения протокола матча надо добавить объект ListBox (Список) и назвать его TotalScores (Счет по партиям).

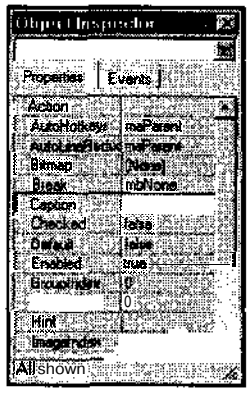
Создаем меню

Теперь надо разместить на главной форме строку меню. На палитре компонентов выбираем компонент MainMenu и размещаем его в любом месте формы.

— Почему в любом?

— Потому что создавать сложенные структуры меню достаточно сложно. Для этого в C++Builder имеется специальный визуальный редактор меню. С ним мы сейчас познакомимся, а пока объект, помещенный на форму, служит лишь сигналом о том, что в данном окне создаваемой программы есть меню. При запуске программы этот объект отображаться не будет.

Чтобы сформировать меню требуемого вида, надо дважды щелкнуть мышью на объекте меню. На экране появится визуальный редактор меню. Ввод пунктов и подпунктов меню выполняется следующим образом.



1. Нажмите клавишу ENTER — откроется Инспектор объектов, который предложит ввести название для текущего пункта в строке Caption.
2. Введите слово Игра и нажмите клавишу ENTER — C++Builder переключится на редактор меню. В нем фокус ввода находится на первом подпункте (см. рис. 32).
3. Нажмите клавишу ENTER и введите в Инспекторе объектов слова Новая игра. Еще раз нажмите клавишу ENTER и добавьте еще один подпункт — Выход (см. рис. 33).
4. С помощью курсорных клавиш переместите фокус ввода к новому пункту меню верхнего уровня, нажмите клавишу ENTER и способом, описанным в пунктах 2 и 3, добавьте подпункты Бросить кубик и Передать очередь хода (см. рис. 34).

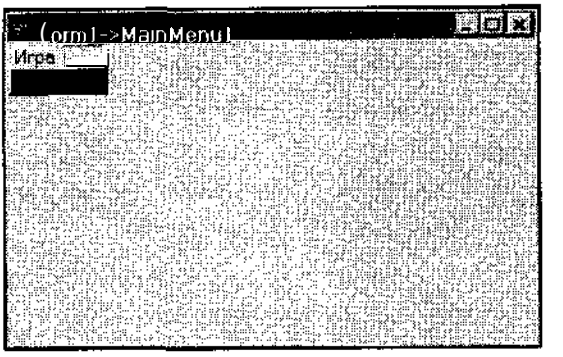


Рис. 32. Создание структуры меню

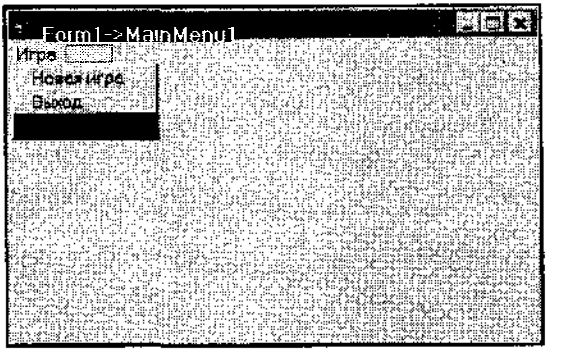


Рис. 33. Создание пунктов меню

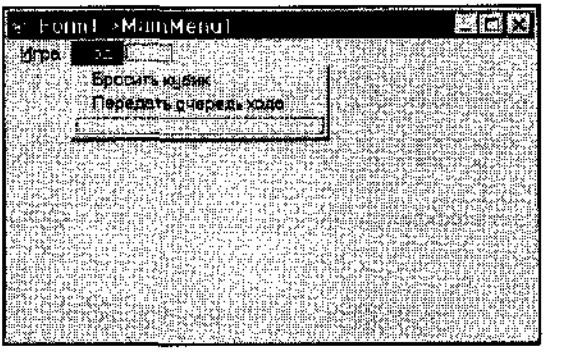


Рис. 34. Создание новых пунктов строки меню

Отредактировать ранее введенные названия можно, выбрав обычным способом в проектируемом меню нужный пункт и изменив в Инспекторе объектов свойство Caption.

Чтобы вставить в меню новый пункт, надо выделить пункт, за которым будет происходить добавление нового пункта и нажать клавишу INSERT. Например, чтобы вставить пункт-разделитель между пунктами Новая игра и Выход, надо выделить пункт Выход, нажать клавишу INSERT и ввести в свойстве Caption одно тире (условное обозначение разделителя).

— А как сделать «горячие» клавиши, чтобы быстро вызывать нужные пункты меню с помощью клавиатуры?

— Для этого при записи соответствующих названий надо выбрать «горячую» букву и поставить перед ней символ &. Например, чтобы сделать в пункте Игра «горячей» букву «И», надо название этого пункта записать так:

```
&Игра
```

Только надо следить, чтобы выбранные буквы в разных пунктах одного меню не совпадали друг с другом, иначе программа не поймет, какой же пункт надо выбрать при нажатии соответствующей «горячей» клавиши.

После того как меню спроектировано, редактор меню надо закрыть. На форме сразу появится строка меню, по которой можно «пробежаться».

«Быстрые» кнопки

Вы, конечно, знаете, что в абсолютном большинстве приложений Windows под строкой меню имеется панель управления с командными кнопками, которые дублируют действие наиболее важных пунктов строки меню. Мы тоже можем создать панель с «быстрыми» кнопками и каждому из подпунктов меню поставить в соответствие свою «горячую» кнопку. Делается это следующим образом.

1. Выберите на палитре компонентов панель Win32.
2. На панели выберите компонент ToolBar (он находится в невидимой части панели — прокрутите ее с помощью кнопки прокрутки).
3. Положите компонент на форму — он автоматически разместится под строкой меню (см. рис. 35).
4. Мы создали панель управления, а теперь должны «наполнить» ее кнопками. Перейдите к панели Additional, выберите компонент SpeedButton и установите его на панели управления (на компоненте ToolBar).

У нашей первой «быстрой» кнопки пока нет никакого изображения. Его можно нарисовать самостоятельно, а можно взять готовый рисунок из библиотеки, которая входит в поставку C++Builder.

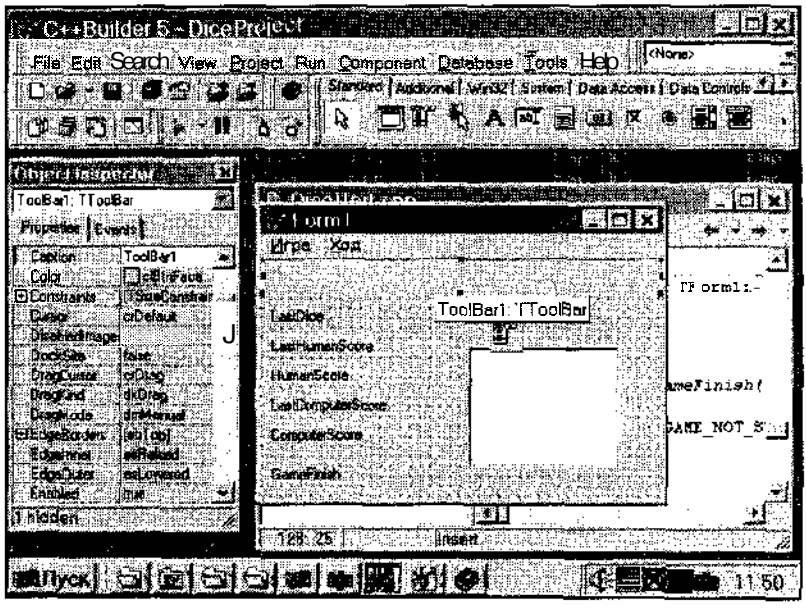


Рис. 35. Создание панели управления

Картинку для «быстрой» кнопки задают так.

1. В Инспекторе объектов выберите свойство Glyph и щелкните на созданной кнопке — откроется диалоговое окно редактора картинок, представленное на рис. 35. С его помощью можно выбрать картинку, подходящую для текущей кнопки.

2. Щелкните на кнопке Load (Загрузить), найдите в папке, где установлен C++Builder, вложенную папку \Borland Shared, в ней папку \Images, а в ней — папку \Buttons. На экране появится большой список готовых картинок. При выборе любой из них в правой части диалогового окна будет отображаться содержимое этой картинки.

Не обращайте внимания на то, что каждая картинка состоит из двух частей. Это сделано специально чтобы кнопка отображалась по-разному в зависимости от ее состояния (отпущена или нажата).

3. Подберите изображение, подходящее для кнопки Новая игра. Мы в нашем примере выбрали образ кнопки phonemg.bmp. Щелкните на кнопке Открыть, затем на кнопке OK, и на проектируемой кнопке появится соответствующая картинка.

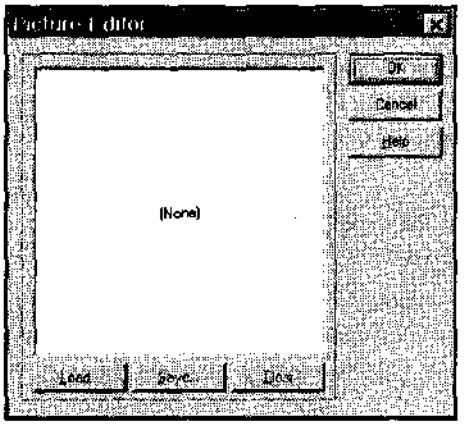


Рис. 36. Редактор изображений для кнопок

4. Таким же способом добавьте в форму еще три «быстрые» кнопки. В нашем примере для них последовательно выбирались картинки dooropen.bmp, arrow3r.bmp и comppc1.bmp.

Для каждой кнопки можно добавить всплывающую подсказку, которая будет появляться при наведении указателя мыши на эту кнопку. Такие подсказки записываются в свойство Hint.

На этом проектирование пользовательского интерфейса можно считать законченным.

Проектирование внутренней структуры программы

На самом глобальном уровне в рассматриваемой игре можно выделить три понятия: *игральный кубик, игрок* и *судья* (он занимается подсчетом очков). Каждое из них удобно описать отдельным классом. Переменных класса Игрок в программе, очевидно, будет две — Человек и Компьютер, а переменных классов Судья и Кубик — по одной.

Классы мы определили, и теперь подумаем об их содержании. Прежде всего начнем со свойств классов и рассмотрим класс Игрок.

— Чем характеризуется игрок?

— Игрок характеризуется суммарным количеством очков, очками, набранными на текущем ходу, а также числом выигранных партий. Это и будут *свойства* или, точнее говоря, *внутренние переменные* класса Игрок.

— А что игрок может делать?

— При своем ходе он может получить и обработать число, выпавшее на кубике, а также может принять решение, будет ли он продолжать броски далее или передаст ход противнику. Это и будут *методы* класса Игрок.

Похожим способом можно описать свойства и методы класса Судья, который определяет, закончилась ли текущая партия и помогает сделать ход компьютеру. Судья также бросает кубик и объявляет новую игру.

Класс Кубик будет совсем простым. Для него потребуется создать только один метод — Получить значение очередного броска и ввести всего одну переменную, хранящую значение последнего броска.

Теперь попробуем описать алгоритм работы программы в зависимости от действий пользователя и возникновения тех или иных событий.

Событие «Новая игра»

При наступлении этого события происходят следующие действия:

- все набранные очки каждого игрока обнуляются;
- в соответствующих полях главного окна выводится начальная информация;
- случайным образом выбирается игрок, ходящий первым;
- если это компьютер, то:

  выполняется ход компьютера.

Событие «Выход»

При наступлении этого события происходит только одно действие:

- программа закрывается.

Событие «Бросить кубик»

После этого события выполняются следующие операции:

- бросается кубик для человека;
- если выпала 1, то

  ход передается компьютеру;

  иначе:

  результат броска добавляется к текущим набранным очкам;

- если партия завершилась, то:

  выводятся соответствующие сообщения;

  начинается новая игра.

Событие «Передать очередь хода»

В результате этого события происходят следующие действия:

- выполняется ход компьютера;
- если партия завершилась, то:

  выводятся соответствующие сообщения;

  начинается новая игра.

Описание классов

Как включить в программу новый класс

Описание пользовательских (создаваемых разработчиком) классов, а также всю логику их работы принято выделять в отдельные файлы Си++. Чтобы добавить в проект новый файл, надо вызвать диалоговое окно создания нового объекта командой File • New (Файл ▸ Создать), и на вкладке New (Создание) этого диалогового окна выбрать значок Unit (Модуль) — см. рис. 37.

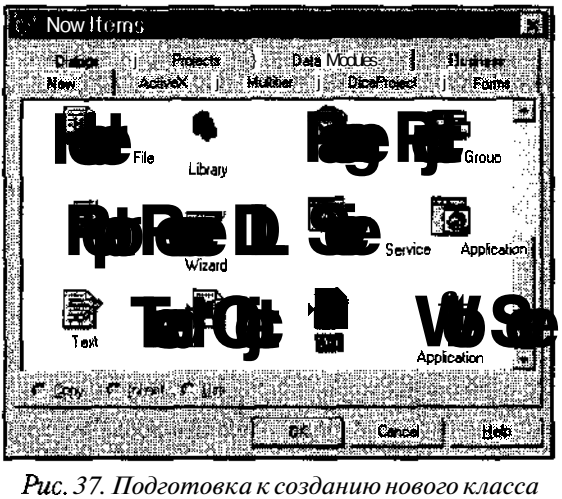


Рис. 37. Подготовка к созданию нового класса

После щелчка на кнопке OK в текстовом редакторе появится новый файл, который можно комбинацией клавиш CTRL+S сразу же сохранить в папке проекта DiceProject с именем Game.cpp.

Однако в файлах с расширением .CPP принято записывать логику работы методов классов (или, как принято говорить, *реализацию* классов), а вот само описание внутренней структуры классов, без раскрытия работы его методов, размещается в заголовочных файлах с таким же именем и расширением .H. Для файла Game.cpp это будет файл Game.h. Он уже создан автоматически и включен в тело Game.cpp командной строкой

```
#include "Game.h"
```

Чтобы быстро перейти к редактированию файла Game.h, надо в редакторе установить курсор на его названии и нажать комбинацию клавиш CTRL+F6.

Добавьте пустые строчки перед заключительными комментариями и командой завершения

```
#endif
```

Здесь и будет располагаться описание нового класса.

Формат (или *синтаксис)* записи нового класса, принятый в Си++, таков:

```
class  имя_класса
{
 // список свойств, переменных и методов
};
```

Свойства и переменные записываются как обычные определения, методы — как описания функций (все с символом «;» в конце).

Внутренние переменные класса (или, говоря иначе, данные, обрабатываемые различными методами класса) не всегда корректно называть свойствами этого класса. Свойства — это, как правило, те переменные, которые описывают определенные характеристики визуального компонента. Их можно редактировать в Инспекторе объектов. Стандартные классы содержат также довольно много вспомогательных переменных, о существовании которых обычный программист даже не подозревает.

В дальнейшем понятие «свойства класса» будет упоминаться исключительно для визуальных компонентов и только в отношении тех переменных, которые доступны в Инспекторе объектов.

Ограничение доступности содержимого класса

В Си++ специально введены ключевые слова public и private, с помощью которых можно сделать некоторые свойства и методы доступными для использования только методами этого же класса. Это придумано для того, чтобы

программист, использующий в своей работе готовые классы, не мог по собственному желанию произвольно менять значения некоторых важных переменных, а делал бы это под своеобразным контролем программы. Профессиональные программисты всегда рекомендуют для доступа к значениям переменных (если это, конечно, требуется в других частях программы) использовать специально написанные методы.

Вставляемые в список определения переменных и методов (в произвольном порядке) ключевые слова public и private со следующим за ними двоеточием информируют, что далее пойдет общедоступная (public) часть описания класса или личная, закрытая (private).

Определяем первый класс

Давайте попробуем добавить в программу самый простой класс — **Кубик**. В нем будет всего один метод, который можно назвать GetDiceValue(). Он будет возвращать значение типа int (целое число от 1 до 6), а вот параметров у него никаких нет — они не нужны. Надо также добавить одну переменную LastValue, в которую будет записываться последнее выброшенное на кубике значение.

```
class TDice
{
public:
int LastValue;
int GetDiceValue();
};
```

Теперь в файле Game.cpp надо указать, что же реально метод делает. Определение метода GetDiceValue() удобно добавить в самый конец файла Game.cpp.

Чтобы компилятор понял, метод какого класса будет определяться, перед названием метода приводится название класса и два двоеточия:

```
int TDice::GetDiceValue()
```

А далее, вместо заключительной точки с запятой, которая в файле Game.h завершала *описание* метода, приводятся фигурные скобки — знакомый нам логический блок, в котором размещаются операторы, определяющие логику работы данного метода.

```
int TDice::GetDiceValue()
{
}
```

Любая программа на Си++ строится подобным способом. Программист определяет нужные ему классы, а потом задает логику работы методов этих классов. Если посмотреть на код, автоматически сгенерированный C++Builder, то там везде будут встречаться методы различных классов — их можно сразу заметить по характерным парам двоеточий (::).

Все реакции на события, которые определялись при создании калькулятора (например, щелчок на кнопке), были автоматически создаваемыми методами главного окна.

Имитируем бросание кубика

В C++Builder есть удобная стандартная функция random(), которая в качестве своего параметра получает целое число *n*, а возвращает тоже целое число из диапазона от 0 до $n-1$. Тогда бросание кубика с шестью гранями будет моделироваться выражением `random(6)+1`. Только для использования функции random() в файл Game.cpp надо включить заголовочный файл stdlib.h:

```
#include <stdlib.h>
```

— А как внутри метода GetDiceValue() записать, что данное выражение будет возвращаться в качестве его значения?

— В этом поможет оператор return. Синтаксис его таков:

```
return возвращаемое_значение;
```

Как только этот оператор встречается по ходу выполнения некоторого метода, дальнейшая работа этого метода немедленно прерывается, и возвращается значение, указанное в качестве параметра return.

Не имеет смысла размещать операторы сразу после оператора return, так как они никогда не будут выполнены. Например, если написать

```
return 10;
x = 5;
```

то для оператора «x=5» компилятор выдаст предупреждение «Unreachable code» (недостижимый код).

Полностью метод, имитирующий бросание кубика, запишется так:

```
// бросаем кубик — получаем значение от 1 до 6
int TDice::GetDiceValue()
{
LastValue = random(6)+1;
return LastValue;
}
```

Перед определением метода принято записывать комментарий, объясняющий, что данный метод делает, какие у него параметры и что он возвращает.

Сейчас нами был создан класс «Игральный кубик», который фактически не привязан к создаваемой программе. Его реализацию можно легко выделить и в дальнейшем включать в другие приложения, уже не тратя время на повторное программирование. В этом заключается огромное преимущество языка Си++, позволяющего повторно использовать ранее разработанные классы, составляя программу из готовых компонентов. Такой подход наглядно реализован в C++Builder — в его библиотеке визуальных компонентов, представляющих собой классы Си++. Конечно, такие многократно используемые классы должны быть очень хорошо отлажены.

— А каков наиболее оптимальный размер метода (по числу операторов)?

— Мировая практика показывает, что реализовывать метод лучше всего не более, чем 50 операторами. Если же метод становится слишком большим, то разобраться в его работе становится сложно и его стоит разбить на несколько более мелких методов.

Описываем класс Игрок

Описание класса Игрок надо продолжить в файле Game.h, вслед за описанием класса TDice:

```
class TPlayer
{
public:
int Scores, CurrentScores, WinNum;
};
```

Переменные Scores, CurrentScores и WinNum будут хранить соответственно суммарное количество очков, очки, набранные на текущем ходу и число выигранных партий.

— А почему эти переменные помещены в общую (public) часть описания класса?

— Потому что размер создаваемой программы невелик, и специально ограничивать доступ к каким-то переменным нет необходимости.

Для *инициализации* игры (задания начальных значений переменных) можно подготовить метод fnit{), который будет вызываться при начале каждой новой партии. Правда, он не должен ничего возвращать, а просто выполнять исходную установку переменных. С другой стороны, мы ранее говорили о том,

что при записи методов обязательно надо указывать тип возвращаемого значения. Как же быть?

Для таких случаев в Си++ есть специальный тип void, который как раз и обозначает *отсутствие возвращаемого значения*. То есть при описании методов, которые используются не для вычисления значений, а для выполнения определенных действий, надо в качестве типа возвращаемого значения указывать ключевое слово void:

```
void  Init();
```

Метод Init(), как и другие методы, которые будут вызываться из других классов, надо разместить в общей части описания TPlayer:

```
class  TPlayer
{
public:
int Scores, CurrentScores, WinNum;
void  Init();
};
```

Определение метода Init() в файле Game.cpp запишется так:

```
// Инициализация внутренних переменных класса Игрок
void  TPlayer::Init()
{
}
```

Теперь надо каждой переменной класса TPlayer присвоить ноль. Это можно сделать несколькими операторами, но Си++ допускает более компактную запись:

```
переменная-1 = переменная-2 = ... = выражение ;
```

В результате все переменные получат одно и то же значение.

Тогда метод инициализации класса TPlayer запишется одним оператором присваивания:

```
Scores = CurrentScores = 0;
```

— А надо ли здесь обнулять переменную WinNum?

— Нет, не надо, потому что она должна накапливать результат о победных партиях на всем протяжении работы программы, и, значит, в нее надо записать начальное значение 0 только один раз.

— А когда это удобнее всего сделать?

Конструктор вызывается только один раз

Помимо методов, каждый класс может иметь *конструктор*. Конструктор — это особый метод, который вызывается автоматически и *только один раз, в момент создания экземпляра класса* (в том месте, где этот экземпляр описан), всегда *до его первого использования*. Например, конструктор окна будет вызываться, когда это окно создается (конструируется), до его первого показа.

Использовать конструктор очень удобно, когда требуется один раз на время работы программы задать начальные значения каким-то переменным.

Синтаксис конструктора:

```
имя-класса();
```

Описание конструктора должно быть включено в текст описания класса:

```
class TPlayer
{
public:
int Scores, CurrentScores, WinNum;
TPlayer();
void Init()/
};
```

Конструктор всегда размещается в общей (public) части описания класса, так как он по определению будет вызываться извне методов своего класса, в строке, где определяется переменная соответствующего класса.

В файле Game.cpp надо определить, что конструктор делает:

```
TPlayer::TPlayer()
{
WinNum = 0;
}
```

Подключаем судью

Теперь осталось описать третий класс — судью, TReferee. В него надо включить две переменные типа TPlayer — игрока-человека и игрока-компьютера, переменную типа TDice — кубик, метод NewGame (новая игра), ComputerMove (ход компьютера) и GameFinish (проверка на конец партии):

```
class TReferee
{
public:
TPlayer Human, Computer;
TDice  Dice;

void  ComputerMove();
bool  GameFinish();
void NewGame();
};
```

Метод GameFinishO будет возвращать логическое значение **true,** если один из игроков набрал больше 100 очков, и false в противном случае.

Вполне возможно, в дальнейшем понадобятся и другие, вспомогательные методы. Будем добавлять их по мере возникновения такой потребности.

Способность предусмотреть и заранее точно определить, какие методы каждого класса будут реально востребованы в работе, считается признаком высокого профессионального мастерства программиста.

Реализация метода **NewGame()** в файле Game.cpp будет немного посложнее, чем реализации предыдущих методов. Сначала надо проинициализировать внутренние переменные каждого игрока, а затем определить, чья очередь хода. Для этого можно бросить кубик, и если на нем выпадут числа 1, 2 или 3, то надо сделать ход за компьютер, в противном случае — считать, что очередь хода человека и ожидать его действий, завершив работу метода **NewGame()**.

```
// новая игра
void TReferee::NewGame()
{
Human.Init();
Computer.Init();

if( Dice.GetDiceValue() <= 3 }
     ComputerMove();
}
```

Доступ к внутренним переменным и свойствам класса

Метод ComputerMove() пока можно реализовать самым простым способом, не задумываясь над деталями эффективной стратегии игры за компьютер. Когда программа будет отлажена, появится возможность заняться совершенствованием этого алгоритма, пока же это не главное.

Для простоты предположим, что компьютер во время своего хода всегда бросает кубик только один раз. Во время проверки выброшенного значения потребуется обращаться к переменным Scores и CurrentScores из класса TPlayer (объект Computer). При создании калькулятора упоминалось, что обращение к свойствам стандартных классов осуществляется с помощью комбинации символов ->. В случае с классами, определяемыми программистом, это не так. Для обращения к их свойствам (точно так же, как и при обращении к методам) надо использовать точку. Например, чтобы получить доступ к переменной Scores, содержащейся в переменной Computer класса TPlayer, надо записать

```
Computer.Scores
```

— Но как же все-таки определить, когда использовать «.», а когда «—>»?

Здесь надо сделать небольшое историческое отступление. Язык Си изначально разрабатывался с целью получения очень компактных и очень быстро работающих программ. Такие программы удавалось создавать в основном при использовании языка ассемблера — низкоуровневого, машинного языка элементарных команд, который различен для разных моделей компьютеров. В ассемблере активно применялись средства работы с оперативной памятью, прямого обращения к конкретным ячейкам ОЗУ, и в Си были учтены общие особенности различных ассемблеров. Для этого в язык было введено понятие *указателей* — особых типов переменных, хранящих не данные, а физические адреса тех мест в памяти, где эти данные расположены. Для хранения адреса нужно обычно от двух (16-разрядные компьютеры) до четырех (32-разрядные компьютеры) байтов. Но реализация работы с указателями была сделана в Си независимой от марки компьютера, то есть в этом языке можно работать с адресами ОЗУ, не задумываясь над тем, как все это происходит на физическом уровне. Поэтому программы на Си, сохраняя эффективность, сравнимую с эффективностью программ на ассемблере, стали *машинно-независимыми* — тексты на Си можно легко переносить на любые компьютеры, где имеется соответствующий компилятор.

Однако с ростом производительности ЭВМ и повышением их быстродействия сложность создаваемых программ стала стремительно увеличиваться.

И сразу выявился недостаток указателей — из-за простоты их использования и полного отсутствия контроля за тем, на какую область памяти они указывают, при малейшей ошибке происходила порча посторонних (нередко системных) областей ОЗУ и серьезное нарушение работы и программы, и операционной системы. Возникла потребность в более мощном языке, который позволял бы быстро создавать крупные проекты и при этом обладал бы более высокой надежностью. Таким языком стал Си++. Поддержка указателей с целью совместимости с ранними версиями Си в нем сохранилась, однако появились и новые средства, уже более аккуратно работающие с памятью.

Поэтому для компиляторов Си++ различные фирмы впускают немало специализированных утилит, позволяющих, в частности, отслеживать некорректную работу с указателями. В C++Builder 5 для этих целей добавлена система CodeGuard (Страж Кода). О ней будет рассказано немного позже.

Кстати, в появившемся несколько лет назад языке Java, созданном на базе Си++, указатели вообще были ликвидированы.

Итак, подводя итоги нашего исторического экскурса, скажем, что сочетание «->» для доступа к свойствам объекта применяется, когда этот объект (точнее говоря, переменная соответствующего класса) является указателем. Символ «.» применяется, когда это обычная переменная (не указатель).

В примерах, приводимых в данной книге, указатели используются только по мере необходимости, и применять их во избежание трудноуловимых ошибок никогда не рекомендуется. Но в тех частях кода, которые были сгенерированы автоматически, по самым разным причинам большинство готовых объектов доступны только через указатели. Поэтому можно руководствоваться простым правилом.

При работе со стандартным или автоматически созданными объектами C++Builder (окнами и расположенными на них визуальными компонентами), представленными указателями, для доступа к их свойствам и методам надо использовать знак «->», а при работе с переменными собственных классов — символ «.».

— А можно по описанию переменной определить, указатель это или нет?

— Можно. В начале файла DiceUnit.cpp расположено описание переменной Form1 — главного окна создаваемой программы. Forml имеет тип TForm1 (класс формы), но после названия класса в описании Forml записана звездочка «*»:

```
TForml *Form1;
```

Именно эта звездочка и указывает на то, что Form1 — это не переменная типа TForml, а указатель на область памяти, где хранится некий объект типа TForml.

Например:

```
int i; // переменная типа int
int * i; // переменная-указатель на данные типа int
TPlayer Computer; // переменная типа TPlayer
TPlayer* Computer;//переменная-указатель на данные типа TPlayer
```

Функции тоже могут возвращать указатели и получать их в качестве параметров:

```
double * Compute( int * X );
```

В дальнейшем ситуации, когда надо будет использовать указатели, будут оговариваться особо.

С учетом вышесказанного запрограммируем работу компьютера по осуществлению своего хода:

```
// Ход компьютера
void  TReferee::ComputerMove()
{
 // Пока на данном ходу не набрано никаких очков:
 Computer.CurrentScores = 0;
 // Если на кубике выпала не единица, то...
 if( Dice.GetDiceValue() != 1 )
     // Запомнить очки текущего кода
     Computer.CurrentScores = Dice.LastValue;
     // Подсчитать очки:
     Computer.Scores = Computer.Scores +
      Computer.CurrentScores;
 // Так как теперь ход человека, надо обнулить
 // очки его текущего хода:
    Human.CurrentScores = 0;
}
```

Смысл предпоследнего оператора присваивания в том, что значение переменной Scores увеличивается на величину CurrentScores. Подобные действия в Си++ можно записывать компактнее:

```
Computer.Scores += Computer.CurrentScores;
```

Оператор присваивания «+=» увеличивает значение переменной, расположенной слева от него, на значение выражения, расположенного справа.

Помимо «+=», в Си++ имеются похожие операторы «–=», «*=» и «/=».

Например:

```
x -=5; // то же, что x = x - 5;
x *= 2; // то же, что x = x * 2;
x /= y+1; // то же, что x = x / (y+1);
```

Теперь осталось реализовать метод GameFinish(). Логический блок, который «скажет», окончена ли игра, должен проверить, достигнута или превышена сумма очков, набранная человеком или компьютером. В нашем примере предельное число очков равно 100. Такую проверку можно записать так:

```
Human.Scores >= 100 || Computer.Scores >= 100
```

Это выражение удобно использовать в операторе return, чтобы вернуть в программу логическое значение true или false:

```
// Проверка на окончание игры
bool TReferee::GameFinish()
{
return (Human.Scores >= 100 || Computer.Scores >=
   100);
}
```

Для наглядности вычисляемое выражение взято в круглые скобки.

Константы — вещь полезная

Допустим, что нам захочется немного изменить правила игры и повысить или понизить порог очков (100), после превышения которого партия считается оконченной. Число 100 упоминается несколько раз в разных местах файла, и постоянно менять все операторы, где оно используется, довольно сложно. Автоматической командой редактора Заменить (Search • Replace) замену выполнить нельзя, так как число 100 может встречаться в разных местах программного кода и обозначать не только предел набранных очков.

Поэтому лучше всего описать число 100 как *константу* и в дальнейшем обращаться к ней по имени. Тогда это число можно будет при необходимости заменить на любое другое число, внеся правку только в одно заранее известное место.

Константы обычно описываются в начале файла, сразу за строками включения заголовочных файлов. Синтаксис описания константы таков:

```
const имя_константы = значение ;
```

Если в программе некоторая числовая константа встречается два и более раз, ее лучше всего описать в виде константы, чтобы при необходимости быстро изменить.

Хорошим стилем считается использование в качестве имен констант слов, записываемых заглавными буквами. Для разделения нескольких слов в одном названии применяется символ подчеркивания «_».

Давайте опишем константу **GAME_FINISH_SCORE** в начале файла **Game.h**:

```
const GAME_FINISH_SCORE = 100;
```

и изменим текст метода ***GameFinish()***:

```
return (Human.Scores >= GAME_FINISH_SCORE ||
        Computer.Scores >= GAME_FINISH_SCORE);
```

Константа на то и константа, что она имеет жестко заданное значение. Нельзя пытаться изменить ее содержимое, например, с помощью оператора присваивания. Константа — просто символическое обозначение конкретного числа.

Теперь, закончив с определением новых классов, методов, переменных и констант, мы можем приступать к следующему этапу — программированию логики работы кнопок и меню.

9. Программирование пользовательского интерфейса

Добавляем переменную-судью

Реализация работы с интерфейсом будет описываться в файле DiceUnit.cpp. Но прежде всего в нем надо ввести в класс главной формы TForm1 (который содержит всю информацию о проектируемом окне и расположенных на нем компонентах) переменную Referee класса TReferee, которая и будет выполнять все действия, непосредственно связанные с логикой игры.

Как добавить новую переменную или новый метод в уже имеющийся, автоматически сгенерированный класс? Так же, как и в обычный. Надо открыть файл DiceUnit.h, найти в нем описание класса TForm 1 (оно начинается со слов «class TForm1») и перейти в конец его описания. Там в разделе public: с комментарием «// User declarations» (пользовательские описания) будет описан только конструктор этого класса. Вслед за конструктором и можно добавлять новые переменные, свойства и методы, в частности, Referee:

```
...
public:  // User declarations
         __fastcall TForm1(TComponent* Owner);

         TReferee Referee;
};
```

Не обращайте внимания на странные ключевые слова вроде __fastcall. Это специальные типы C++Builder, которые используются для внутренних нужд. Для повседневной работы они обычно не требуются.

— Но как в файле DiceUnit.h станет известно про тип TPlayer?

— Для этого и служат заголовочные файлы, в которых содержится информация о доступных в данном файле (программном модуле) классах, реализация которых расположена в других файлах.

В начале файла DiceUnit.h надо подключить заголовочный файл Game.h (в котором содержится описание класса TPlayer). Это делается командной строкой:

```
#include "Game.h"
```

Теперь при желании программу можно откомпилировать и даже запустить, только работать она пока не будет — ведь в ней не определены действия, которые будут происходить при выборе пунктов меню или нажатии на командные кнопки.

Выход из игры

Выход из игры и закрытие приложения — операция простая. Она будет выполняться, когда человек выберет в меню пункт Выход (или щелкнет на соответствующей командной кнопке). Закрыть программу надо в обработчике соответствующего события, которое будет сгенерировано C++Builder. Это выполняется следующим образом.

1. Выберите в меню формы пункт Выход из игры и один раз щелкните на нем мышкой (см. рис. 38).
2. С помощью Инспектора объектов задайте для выбранного таким способом пункта меню название (Name) GameExitItem.
3. Снова выберите в меню формы этот пункт и дважды на нем щелкните.

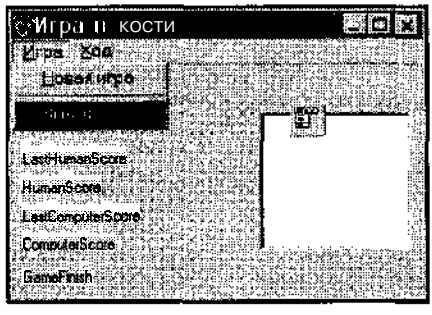


Рис. 38. Настройка действия пунктов меню Выход

C++Builder автоматически создаст новый метод GameExitItemClick() класса TForm1 главной формы. Этот метод будет вызываться, когда пользователь в работающей программе выберет пункт меню Выход из игры.

В методе достаточно привести один оператор — вызов метода Terminate() объекта Application (класс TApplication). Этот объект создается C++Builder автоматически. Он доступен через указатель и содержит в себе информацию обо всем создаваемом приложении — в частности, о названии EXE-файла, его значке и т. д.

```
Application->Terminate();
```

Так как Application — указатель, то вызывать метод этого объекта надо с помощью «->».

Новая игра

Соответствующий пункт меню в Инспекторе объектов можно назвать **NewGameItem**. Для него надо вышеописанным способом создать метод реакции (он автоматически получит название NewGameItemClick()) и прежде всего вызвать метод **NewGame()** объекта **Referee,** чтобы задать начальные значения внутренним переменным.

Далее требуется вывести в свободные поля информацию о состоянии текущей партии. Эти операции лучше всего вынести в отдельный метод, так как выполнять действия по отображению подобной информации придется и в других точках программы.

Всегда старайтесь выделить не зависящую от интерфейса логику работы программы в отдельные независимые модули (в нашем случае — это реализация трех классов в файле Game.cpp). Смешивание логики задачи и функций пользовательского интерфейса в одну кучу сильно затрудняет понимание программы и повышает вероятность внесения ошибок.

Новый метод — назовем его **ShowInfo()** — надо включить в класс главной формы **TForm1**. Реализация метода **ShowInfo()** помещается в конце файла DiceUnit.cpp:

```
// Показ текущей информации о ходе игры:
void  TForm1::ShowInfo()
{
}
```

а описание метода — в файле DiceUnit.h (в описании класса **TForm1**):

```
void  ShowInfo();
```

Используем текстовые константы

Информация будет выводиться в пять описанных полей подписи. Начнем с поля вывода последнего выпавшего на кубике значения LastDice. В него надо записать соответствующее число, преобразованное в текстовый вид с помощью стандартной функции IntToStr(). Только перед этим числом неплохо бы для наглядности выводить небольшой комментарий, например, Последний бросок был, для чего потребуется *сцепить* эту строку со строкой, возвращаемой IntToStr(). Готового метода сцепления в соответствующем классе AnsiString, описывающем текстовые строки, нет, однако в Си++ для удобства работы введена еще одна интересная возможность — в этом языке для любых классов разрешается определять *собственные операции,* такие, как «равно» (==), «сложить» (+) и другие, входящие в стандартный набор операций Си++. В частности, для AnsiString определены как операции «равно» (=) и «не равно» (!=), так и операция «+», которая выполняет сцепление двух строк. Например, значением выражения "123" + "45" будет строка "12345".

Как уже говорилось ранее, текстовые строки Си++ записываются в виде последовательности символов, заключенных в кавычки. Это так называемые текстовые константы, которые удобно описывать с помощью ключевого слова const и выносить в отдельный заголовочный файл. Ведь если понадобится перевести программу на другой язык, то тогда не надо будет искать в каждом файле текстовые константы и переводить их, стирая старое значение. Достаточно подготовить несколько заголовочных файлов и просто менять их по мере необходимости.

В C++Builder имеются гибкие возможности, позволяющие выполнять включение различных файлов и даже компиляцию отдельных частей кода в зависимости от некоторых условий — это так называемая *условная компиляция,* которая в данной книге не рассматривается.

Говоря более конкретно, выделение текстовых констант в отдельный файл делается так.

1. Выполните команду File ▸ New (Файл ▸ Создать), в открывшемся диалоговом окне на вкладке New (Создание) выберите значок Header File (Заголовочный файл) (см. рис. 39) и щелкните на кнопке OK. В Редакторе откроется новый файл File1.h.
2. Сохраните этот файл в каталоге проекта с помощью команды File • Save As (Файл ▸ Сохранить как), дав ему имя Text.h (произвольно выбранное название, обозначающее, что в этом файле будут располагаться текстовые константы).

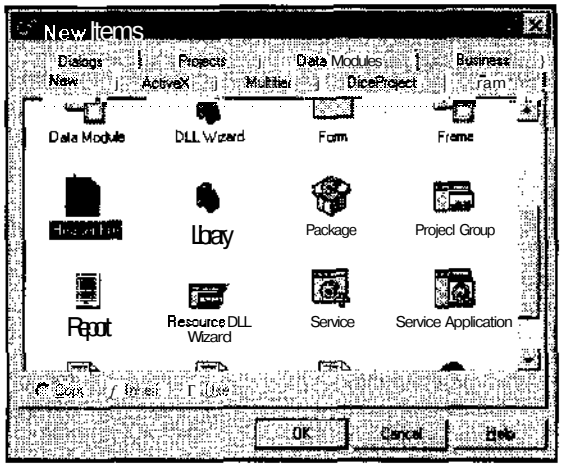


Рис. 39. Создание файла текстовых констант

3. Введите первую текстовую константу. При ее определении после ключевого слова const надо явно указать тип этой константы. По умолчанию считается, что тип константы — целое число (int) во всех остальных случаях этот тип требуется указывать:

```
const AnsiString LAST_DICE_STR = "Последний
  бросок был ";
```

Пробел в конце текстовой константы вставлен, чтобы слово «был» не сливалось с последующей цифрой.

Для того чтобы отличать текстовые константы от числовых, к названиям текстовых констант в конце обычно приписывается суффикс _STR.

4. Включите этот файл командной строкой

```
#include "Text.h"
```

в файл DiceUnit.cpp, перед командой включения DiceUnit.h.

Хотя используемые в программе текстовые константы и вынесены в отдельный файл, некоторые строки задаются в C++Builder с помощью Инспектора объектов (например, заголовки окон или всплывающие подсказки). Непосредственно в тексте приложения они не хранятся, а заменять их вручную при переходе к другому языку весьма неудобно. Для решения проблемы поддержки нескольких национальных языков на всех уровнях в C++Builder встроена специальная система создания многоязыковых приложений «Integrated Translation Environment».

Длинные выделения переменных вложенных классов

Текстовую константу LAST_DICE надо сцепить с переведенным в текстовый вид значением переменной LastValue, входящей в состав объекта Dice (такое обращение к LastValue запишется как Dice.LastValue). Однако сама переменная Dice не доступна напрямую в создаваемой программе, так как скрыта в классе TReferee. К ней можно обратиться так:

```
Referee.Dice
```

Тогда полное обращение к переменной LastValue запишется в виде:

```
Referee.Dice.LastValue
```

В этом выражении выделение элемента класса происходит два раза и выполняется, согласно правилам Си++, *слева направо*. Сначала в переменной Referee выделяется переменная Dice класса TDice:

```
Referee.Dice
```

а затем в переменой Dice выделяется переменная LastValue.

Тогда сцепление текстовой константы LAST_DICE_STR и результата вызова функции intToStr() будет выглядеть следующим образом:

```
LAST_DICE_STR + IntToStr( Referee.Dice.LastValue )
```

— А где сохранить итоговую строку?

— Это можно сделать в свойстве Caption объекта LastDice, который сразу покажет свое содержимое в соответствующем поле окна. Это свойство доступно через указатель.

```
LastDice->Caption = LAST_DICE_STR + IntToStr
  ( Referee.Dice.LastValue );
```

Теперь надо описать еще четыре текстовые константы:

```
const AnsiString HUMAN_CURRENT_STR = "Во время хода
  человека пока набрано ";
const AnsiString HUMAN_ALL_STR = "Всего человек
  набрал ";
const AnsiString COMPUTER_CURRENT_STR = "Во время
  хода компьютера пока набрано ";
const AnsiString COMPUTER_ALL_STR = "Всего компьютер
  набрал ";
```

Их использование в методе ShowInfo() будет выглядеть так:

```
LastDice->Caption = LAST_DICE_STR + IntToStr
  ( Referee.Dice.LastValue );
```

```
LastHumanScore->Caption = HUMAN_CURRENT_STR +
  IntToStr( Referee.Human.CurrentScores );
HumanScore->Caption = HUMAN_ALL_STR + IntToStr
  ( Referee.Human.Scores );
LastComputerScore->Caption = COMPUTER_CURRENT_STR +
  IntToStr( Referee.Computer.CurrentScores );
ComputerScore->Caption = COMPUTER_ALL_STR + IntToStr
  ( Referee.Computer.Scores );
```

Метод ShowInfoO будет вызываться в программе довольно часто, так как именно он отвечает за вывод информации пользователю.

Чтобы программа при запуске исходно сразу была готова к игре, неплохо также вызвать методы **Referee.NewGame()** и ShowInfoO один раз где-то в самом начале работы. Лучше всего это сделать в конструкторе класса **TForm1** — этот конструктор уже сгенерирован автоматически, и в его пока пустое содержимое надо только вписать два оператора:

```
__fastcall TForm1::TForm1(TComponent* Owner)
                    : TForm(Owner)
{
  Referee.NewGame();
  ShowInfo();
}
```

Проверка завершения текущей партии

Подобную проверку удобно выделить в отдельный метод — назовем его **EndGameTest()**. Описание этого метода класса **TForm** 1 в файле **DiceUnit.h** лучше разместить рядом с описанием ShowInfoO.

```
void  EndGameTest();
```

В методе EndGameTestO с помощью метода GameFinish() класса TReferee будет выполняться проверка, не завершена ли текущая партия, и если да, то будет вызвано диалоговое окно с сообщением о том, победил человек или проиграл. Метод должен также добавить в список TotatScores новую строку, информирующую об итоговом счете.

Проверку на окончание партии можно выполнить условным оператором:

```
if( Referee.GameFinish() )
```

Метод **GameFinish()** сразу возвращает логическое значение типа **bool** — именно то, что и требуется условному оператору.

За оператором if следует логический блок, заключенный в фигурные скобки. В начале этого блока надо определить временную переменную типа AnsiString:

```
AnsiString s;
```

которая потребуется для хранения строк, предназначенных для вывода на экран. Почему в данном случае вместо переменной не будут использоваться константы, станет понятно дальше.

Так как внутри нового блока известно, что один из игроков одержал победу, выясним, кто это был конкретно. В этом опять поможет условный оператор:

```
if( Referee.Human.Scores >= GAME_FINISH_SCORE )
```

Сначала проверяются набранные очки человека. Если они больше или равны 100, то в переменную s заносится строка о победе человека (ее надо описать в файле **Text.h**:

```
const AnsiString HUMAN_WON_STR = "Победил человек";
```

в противном случае — строка, сообщающая о поражении:

```
const AnsiString COMPUTER_WON_STR = "Победил
  компьютер";
```

Здесь же следует увеличить на единицу счетчик числа побед для соответствующего игрока (Human или Computer).

Расширенный условный оператор

Такую логику удобно реализовать условным оператором, учитывающим, что надо сделать, как в случае, если проверяемое условие истинно, так и в случае, если это условие ложно. В Си++ для этого есть специальная форма оператора if:

```
if( условие ) действие-1;
else действие-2;
```

Действия-2 выполняются, если условие ложно — его значение равно false.

Соответствующий оператор изменения числа побед с помощью расширенного if запишется так:

```
if( Referee.Human.Scores >= GAME_FINISH_SCORE )
        {
        Referee.Human.WinNum += 1;
        S = HUMAN_WON_STR;
        }
```

```
else
        {
        Referee.Computer.WinNum += 1;
        S = COMPUTER_WON_STR;
        }
```

Отметим, что здесь для доступа к переменной WinNum класса TPlayer, представленного переменными Human и Computer, используется уже знакомая запись последовательного обращения ко вложенным переменным с помощью соединяющей точки.

Увеличение значения переменной на единицу — часто выполняемое действие. Для его сокращения в Си++ есть специальная операция «++». Например, записать, что значение Referee.Human.WinNum увеличивается на единицу, можно таким образом:

```
Referee.Human.WinNum ++;
```

В отличие от операций сложения или вычитания, которым требуются соответственно два слагаемых или уменьшаемое с вычитаемым (они называются *операндами),* у операции ++ только один операнд.

Свое название язык Си++ получил именно от этой операции. Это как бы следующая, увеличенная на единичку, версия языка Си.

После выполнения расширенного условного оператора в переменной s будет храниться строка с сообщением о том, выиграл человек или проиграл. Теперь эту строку надо вывести на экран в диалоговом окне:

```
ShowMessage(s);
```

Далее информацию о победе и текущий счет необходимо добавить в список TotalScores. Для формирования такого сообщения к содержимому переменной s через пробел справа надо приписать соотношение побед, например, "8 : 5". Получить текстовые представления соответствующих переменных WinNum можно с помощью стандартной функции IntToStrO, а добавить пробел и двоеточие — используя уже упоминавшуюся операцию сцепления строк:

```
s = s + " " + IntToStr(Referee.Human.WinNum) + ":" +
   IntToStr  (Referee.Computer.WinNum);
```

Запись

```
s = s + выражение;
```

можно сократить. Для большинства типов Си++, для которых определена операция «+», определена также и операция «+=» (использовавшаяся ранее для числовых типов). То есть, упомянутую запись можно записать как:

```
s += выражение;
```

а конкретный оператор формирования текстовой строки как

```
s += " " + IntToStr(Referee.Human.WinNum) + ":" +
  IntToStr(Referee.Computer.WinNum);
```

Добавление строки в список

Элемент управления TotalScores состоит из строк. Исходно список пуст. Но в классе TListBox визуального компонента ListBox нет метода добавления новой строки. В нем содержатся лишь базовые методы, общие для всех компонентов, своим поведением в той или иной степени напоминающие работу списков.

Доступ к содержимому списка, представленному набором строк, осуществляется через свойство Items (Элементы) класса TListBox. Это свойство имеет тип TStrings (специальный класс, предназначенный для работы со списками строк). А уже в этом классе есть метод Add() (Добавить). С помощью метода Add() мы и включим новую текстовую строку (типа AnsiString) в конец списка.

Список TotalScores и свойство Items доступны, как и другие стандартные объекты C++Builder, через указатели:

```
TotalScores->Items->Add(s);
```

С помощью этого оператора в конец списка TotalScores будет добавлена новая строка s.

После того как все описанные действия будут выполнены, программа окажется в некотором промежуточном состоянии: прошлая игра уже закончилась, о чем имеется соответствующая запись в списке, а новая еще не началась. Такое состояние надо отслеживать особо, чтобы при щелчке на кнопке Бросить кубик никаких действий не происходило, а выдавалось сообщение о том, что новая партия не начата.

Для нормальной работы программы осталось реализовать такие действия пользователя, как выбор пунктов меню Новая игра, Бросить кубик и Передать очередь хода.

Когда выбран пункт меню «Новая игра»

С помощью Инспектора объектов задайте пункту меню Новая игра название NewGameItem. Далее создайте метод реакции на выбор этого пункта спосо-

бом, описанным ранее при формировании обработчика выбора пункта меню **Выход.**

Перед началом новой партии необходимо проверить, закончена ли текущая игра. Ведь если это не так, то человек получит определенные преимущества перед компьютером. Почувствовав, что проигрывает, он сможет начать новую партию, и отрицательный результат ему не зачтется.

```
// Проверяем, закончена ли партия:
if( Referee.GameFinish() )
    {
    // Если да, то начинаем новую игру:
    Referee.NewGame();
    // и выводим на экран начальную информацию:
    ShowInfo();
    }
```

— А что делать, если партия продолжается?

— Программа всегда должна в подобных случаях реагировать на действия человека, сообщая ему хотя бы минимальную информацию. Поэтому, используя расширенный вариант условного оператора, с помощью ключевого слова else добавим вызов диалогового окна с кратким напоминанием о том, что игра не завершена. Это напоминание надо определить как константу в файле **Text.h:**

```
const AnsiString GAME_NOT_STOP_STR = "Партия не
  закончена";
```

А альтернативная часть условного оператора запишется так:

```
else
    ShowMessage(GAME_NOT_STOP_STR);
```

Когда выбран пункт меню «Бросить кубик»

С помощью Инспектора объектов задайте пункту меню Бросить кубик название **UseDiceItem** и создайте метод реакции на выбор этого пункта. Он получит название UseDiceItemClick.

Что надо сделать в данном методе? Прежде всего проверить, закончена ли текущая партия. Если это так, то желательно сообщить пользователю, что сначала надо начать новую партию, а дальнейшие действия текущего метода не выполнять:

```
if( Referee.GameFinish() )
    ShowMessage(GAME_STOP_STR);
```

Константу GAME_STOP_STR надо предварительно описать в файле Text.h:

```
const AnsiString GAME_STOP_STR = "Начните новую партию";
```

— А как теперь завершить выполнение метода?

— В этом нам поможет оператор return. Ранее он использовался для завершения работы метода и формирования возвращаемого им значения, но метод UseDiceItemClick(), если посмотреть в его заголовок, возвращает значение типа void (или, иными словами, ничего не возвращает). Однако оператор return можно использовать все равно, только никакого выражения ему теперь приписывать не нужно. Указывается просто:

```
return;
```

Как только такой оператор встречается, выполнение текущего метода заканчивается.

В зависимости от того, возвращает метод значение, или нет (точнее говоря, имеет ли возвращаемое значение тип void), надо обязательно использовать правильную форму оператора return — с вычисляемым выражением или без него.

Полная проверка на завершение партии запишется так:

```
if( Referee.GameFinish() )
    {
    ShowMessage(GAME_STOP_STR);
    return;
    }
```

Если же игра продолжается, то сначала требуется проверить, выпала ли единичка. Если да, то с помощью ShowMessage() сообщить об этом человеку, для чего предварительно надо описать константу:

```
const AnsiString ONE_STR = "Выпала единица";
```

Не забывайте: все константы располагаются в файле Text.h.

Если выпала не единица, то необходимо увеличить набранные на текущем ходу очки на выпавшее число, по окончании всех проверок обновить содержимое экранных полей и выполнить проверку на завершение игры.

Выглядеть соответствующий текст будет так:

```
// если выпала единица...
if( Referee.Dice.GetDiceValue() == 1 )
        {
        // показать соответствующее сообщение
        ShowMessage(ONE_STR);
        // передать очередь хода компьютеру
        // (набранные очки человеку не засчитываются)
        Referee.ComputerMove();
        }
```

В противном случае надо просто увеличить значение переменной Referee.Human.CurrentScores на выпавшее число (оно хранится в переменной Referee.Dice.LastValue):

```
// иначе запомнить выброшенные очки:
else Referee.Human.CurrentScores +=
  Referee.Dice.LastValue;
```

В заключение вызываются два метода:

```
ShowInfo(); // показать новые данные
EndGameTest(); // проверка, не завершена ли партия
```

Когда выбран пункт меню «Передать очередь хода»

С помощью Инспектора объектов задайте пункту меню Передать очередь хода название NextItem и создайте метод реакции на выбор этого пункта.

Сначала, точно так же, как и в предыдущем методе обработки броска кубика, необходимо проверить, не завершена ли игра. Затем в переменной, где хранятся набранные человеком очки, надо запомнить число очков, накопленное на текущем ходу, после чего выполнить ход программы и, как и при ходе человека, обновить экран и проверить, не закончена ли партия:

```
Referee.Human.Scores = Referee.Human.CurrentScores;
Referee.ComputerMove();
ShowInfo();
EndGameTest()
```

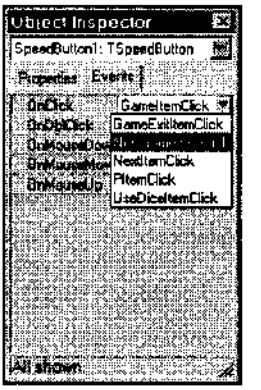
На этом создание игровой программы почти закончено. Осталось только присвоить командным кнопкам соответствующие действия, выполняемые при выборе пунктов меню.

Настраиваем командные кнопки

Каждая командная кнопка настраивается следующим образом.

1. Щелчком левой кнопки мыши выберите на форме нужную кнопку.

2. В Инспекторе объектов перейдите на вкладку Events (События).



3. В строке OnClick определяется метод, который будет вызван при щелчке на этой кнопке. В раскрывающемся списке приведен набор методов, вызываемых при выборе пунктов главного меню.

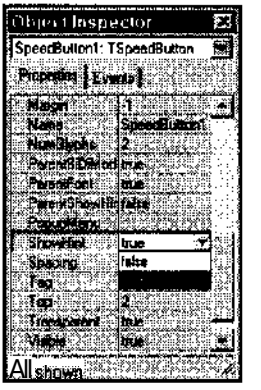
4. Задаем методы.

   Кнопка 1 — **NewGameItemClick** (Новая игра).

   Кнопка 2 — **GameExitItemClick** (Выход).

   Кнопка 3 — **UseDiceItemClick** (Бросить кубик).

   Кнопка 4 — **NextItemClick** (Передать ход компьютеру).



Попутно для каждой кнопки желательно задать значение свойства Hint — всплывающей подсказки. Подсказками, например, могут быть названия соответствующих пунктов меню. Для того, чтобы во время выполнения программы показывалась подсказка, надо также свойству ShowHint соответствующей кнопки задать значение true.

Наконец программа, на первый взгляд, готова. Просмотрите исходные тексты — не встретится ли где-нибудь явных опечаток и ошибок, откомпилируйте программу и запустите ее.

Проверяем, все ли в порядке

После запуска программы на экране отобразится такое окно (см. рис. 40).

Сейчас ход человека. То, что компьютер не ходил первым, понятно из того, что у него нет очков, а последней на кубике выпала единица.

Щелкните на кнопке Бросить кубик или выберите соответствующий пункт в меню. Должно выпасть 3 — это число запишется в текущие набранные человеком очки. Бросьте кубик еще раз (выпадет 5), и щелкните на кнопке Пере-

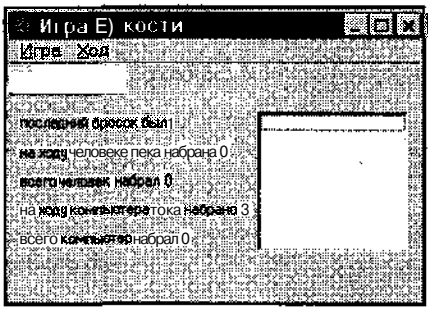


Рис. 40. Окно программы во время работы

дать очередь хода. В вашу общую копилку занесется 8, а компьютеру ничего не запишется — он выполнил свой ход мгновенно, но ничего не набрал, потому что у него выпала единица. Бросьте кубик (выпадет 4), потом еще раз и еще — и тут выпадет единица, о чем программа и сообщит в диалоговом окне.

Щелкните на кнопке OK, компьютер сделает ход и снова ничего не наберет. На вашем ходу выпадет 4. Попробуйте передать ход компьютеру, чтобы наверняка положить эту «четверку» в свою копилку, и тут выяснится, что число ваших очков стало не 12(8+4), а 4. Видимо, в программе допущена ошибка. Закройте приложение и вернитесь в редактор.

Пошаговая отладка

Выявить допущенную ошибку можно в встроенном отладчике C++Builder. С точками прерывания мы уже знакомы. С их помощью мы проверяли работу валютного калькулятора. Теперь попробуем проанализировать, как реально выполняется игровая программа.

Перезапустите программу, повторите все вышеописанные действия до момента, когда в копилке будет 8 очков, а на кубике выпадет четверка. Теперь перейдите в редактор C++Builder и найдите там метод NextItemClick(), который вызывается при щелчке на кнопке Передать очередь хода. Поставьте с помощью клавиши F5 точку остановки на первый оператор и вернитесь к запущенному приложению. Теперь при попытке передать ход компьютеру работа программы прервется, и зеленая стрелка укажет, какой оператор должен выполняться на следующем шагу (см. рис. 41). Можно посмотреть значения переменных Referee.Human.Score и Referee.Human.CurrentScore, наведя на них указатель мыши — получится, соответственно, 8 и 4.

Далее выполнение программы можно продолжать в пошаговом режиме — останавливаясь после выполнения очередного оператора, точнее, всех операторов, расположенных в очередной строке, имеющей пометку круглым маркером.

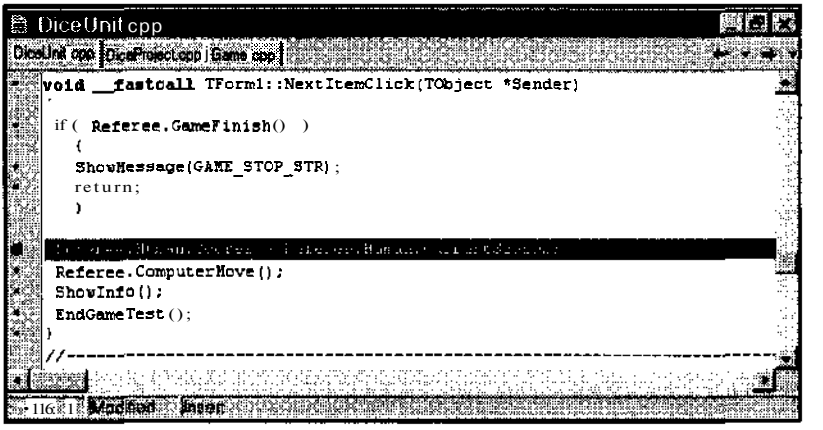


Рис. 41. Окно редактора в режиме пошаговой отладки

Пошаговое выполнение очень удобно, так как позволяет взглянуть на работу приложения изнутри, в динамике, непрерывно контролируя изменения любых переменных. Выполнить очередную строку, помеченную стрелкой, можно, нажав клавишу F8. Следующая строка, которая будет выполнена, выделится в редакторе синим цветом.

Если теперь проверить содержимое переменной **Referee.Human.Scores**, то выяснится, что в ней записано число 4 (хотя планировалось 12), которое методом ShowInfoO и будет выведено на экран.

Во время нахождения в отладчике и выполнения программы по шагам переключиться на запущенное приложение нельзя, так как его выполнение приостановлено. Продолжить его работу можно только нажатием клавиши F9.

Наверное, вы поняли ошибку. Дело в том, что значение переменной Referee.Human.CurrentScores не прибавлялось к переменной **Referee.Hu** man. Scores. Происходила просто перезапись старого значения. Для правильной реализации логики работы надо операцию = заменить на операцию +=:

```
Referee.Human.Scores += Referee.Human.CurrentScores;
```

Видите, как сильно может влиять на логику работы всего один пропущенный символ? А подобные описки не обнаружит никакой компилятор.

Теперь отлаживаемую программу надо завершить, нажав комбинацию клавиш CTRL+F2, затем убрать точку остановки, перекомпилировать текст и запустить приложение заново. Проверьте его работу — в нужный момент в накопленную человеком сумму запишется число 12. Так и должно быть.

По клавише F8 выполняются *сразу все операторы* в очередной строке. По клавише F7 происходит «вход внутрь» в очередной метод, при условии, что он, конечно, в данной строке имеется.

Например, если должна выполняться строка

```
Referee.ComputerMove();
```

то при нажатии на клавишу F7 отладчик перейдет к выполнению операторов метода ComputerMove(). По окончании выполнения этого метода отладчик вновь вернется обратно, в метод NextItemClick().

Исключаем повторяющиеся игры

Внимательный читатель, конечно же, заметил, что автор книги как будто заранее знает, что будет выпадать на кубике при пробном запуске игры. Интересно, откуда такая прозорливость? Разве результат броска кубика не определяется случайным числом? А может быть это число не случайно?

Нет, конечно же, генератор выдает случайное число, однако сама последовательность случайных чисел отнюдь не случайная, а стабильная, она всегда одна и та же.

— Но ведь скоро играть с программой будет неинтересно?

— Да, это так, и неслучайность последовательности надо исключить. Причина этого явления лежит в устройстве генератора случайных чисел random(). Наличие одной и той же стабильной последовательности очень удобно при отладке программы. Так, чтобы найти вышеупомянутую ошибку, нам потребовалось, чтобы проявление ошибочной ситуации было *повторяющимся*. В противном случае имитировать похожую ситуацию было бы значительно труднее.

Однако в законченном приложении повторение одних и тех же случайных чисел, конечно, недопустимо. Чтобы этого избежать, в C++Builder имеется специальная стандартная функция randomizeO, которую надо вызвать один раз при начале работы программы, и тогда в дальнейшем все обращения к функции random() не приведут к генерации уже известной последовательности. Вызывать функцию randomizeO удобнее всего в конструкторе главной формы, до вызова новой игры:

```
randomize();
Referee.NewGame();
ShowInfo();
```

Для работы этой функции в файле DiceUnit.cpp необходимо также выполнить подключение двух заголовочных файлов:

```
#include <stdlib.h>
#include <time.h>
```

Перезапустите программу. Теперь выбрасываемые значения никогда повторяться не будут.

Оператор цикла

Подходящий алгоритм

Если вы попробуете поиграть со своей программой, то вскоре убедитесь, что она будет, как правило, проигрывать. Это неудивительно, ведь логику работы компьютерного оппонента мы задали очень примитивно. Компьютер всегда отказывается от продолжения серии бросков и на каждом ходу довольствуется одним результатом.

Следующий этап в создании законченной программы — повышение силы игры компьютера. Чтобы его сила резко возрасла, необходимо в методе ComputerMove() класса TReferee реализовать более сильный алгоритм игры. Не вдаваясь в детали его создания, скажем, что близкая к самой лучшей стратегия выглядит так: кидать кубик, пока либо не наберется 20 и более очков, либо пока он не будет брошен пять раз (при дальнейшем бросании кубика риск получить «кол» и потерять накопленное становится слишком велик).

Цикл вместо ручного копирования

Чтобы смоделировать пятикратное бросание кубика, можно написать текст для одного броска, и затем в редакторе продублировать его четыре раза. Но если вдруг потребуется внести изменение в алгоритм обработки броска, то каждый раз надо будет стирать старые версии алгоритма и повторно копировать новые. Неудобно и очень громоздко. Нет ли лучшего способа?

Есть. В Си++ имеется специальный оператор (оператор *цикла),* который позволяет многократно (несколькими циклами) выполнять отдельный оператор или логический блок. Синтаксис его таков:

```
for( оператор-1; условное-выражение; оператор-2 )
    повторяемый-оператор_или_логический-блок;
```

Выполняется этот оператор по следующему алгоритму.

1. Выполнить **оператор–1**.
2. Если значение условного (логического) выражения ложно (false), то закончить выполнение оператора цикла.
3. Выполнить **повторяемый_оператор_или_логический_блок**.
4. Выполнить **оператор–2**.
5. Перейти к пункту 2.

Например, если надо пять раз выполнить блок, в котором выполняется бросание кубика, то это удобнее всего сделать, используя специальную переменную в качестве счетчика циклов. При этом **оператор–1** чаще всего используется для того, чтобы задать такой переменной начальное значение.

Оператор–1 выполняется **только** один раз. Он **как**бы предшествует выполнению цикла.

Условное выражение необходимо, чтобы определить, сколько раз цикл будет повторяться. В качестве такого выражения обычно используется проверка значения счетчика.

Оператор–2 выполняется в конце каждого цикла. Чаще всего он нужен для увеличения значения счетчика.

Как выполнить логический блок пять раз? Например, так:

```
int i; // описание переменной-счетчика
for( i = 1; // счетчику исходно присваивается
            // начальное значение 1
     i <= 5; // выполнение цикла будет продолжаться,
             // пока значение i меньше или равно 5
     i ++ ) //по окончании очередного цикла
            // увеличиваем значение на 1
 {
 // здесь расположен блок, повторяемый в цикле
 }
```

Внутри цикла

А как будет работать программа при обработке каждого броска? Примерно так же, как и раньше. Прежде всего в повторяемом логическом блоке (он еще называется *телом цикла)* надо проверить, не выпала ли на кубике единица. Если нет, то выпавшее значение надо прибавить к очкам, уже накопленным на данном ходу. Если выпала — накопленные очки надо обнулить. Также желательно проверить, набрано ли 20 (или более) очков, и при необходимости прекратить бросать кубик.

С накоплением очков все понятно:

```
// если на кубике выпала не единица, то...
if( Dice.GetDiceValue() != 1 )
    // запомнить очки
    Computer.CurrentScores += Dice.LastValue;
```

А вот в противном случае, когда выброшена единица, текущие очки надо обнулить:

```
else Computer.CurrentScores = 0;
```

Но верно ли это? Ведь после того, как в переменную Computer.CurrentScores записан ноль, выполнение цикла может продолжиться (если число, хранящееся в счетчике, меньшее пяти), и на следующем броске в эту переменную может записаться новое значение, что противоречит правилам. В случае выпадания единицы бросание кубиков надо немедленно прервать, вне зависимости от того, сколько раз их намечено бросить — один или пять.

Остановка цикла

Чтобы немедленно прервать работу оператора for, используется оператор break (он не имеет параметров и записывается так:

```
break;
```

Когда выполняется оператор break, управление в программе передается на оператор, следующий за for. Тогда обработку выпадения единицы надо переписать следующим образом:

```
else
      {
      Computer.CurrentScores = 0;
      break;
      }
```

Однако выполнены еще не все проверки. В теле цикла надо выяснить, не накоплено ли 20 и более очков, и если накоплено, то дальнейшее бросание кубика тоже надо прекратить, так же, как это было сделано только что при выпадении единицы:

```
// если набрано 20 и более очков, то
if( Computer.CurrentScores >= 20 )
    // прервать цикл бросания кубика
   break;
```

Итоговый текст реализации метода ComputerMove() примет следующий вид:

```
void  TReferee::ComputerMove()
{
 int i;
 Computer.CurrentScores = 0;
 for( i = 1; i <= 5; i ++ )
   {
   if ( Dice.GetDiceValue() != 1 )
       Computer.CurrentScores += Dice.LastValue;
   else
     {
     Computer.CurrentScores = 0;
     break;
     }
  if( Computer.CurrentScores >= 20 )
      break;
   }
 Computer.Scores += Computer.CurrentScores;
 Human.CurrentScores = 0;
}
```

Попробуйте откомпилировать и запустить программу. Поиграйте с ней — окажется, что выиграть у нее стало весьма трудно!

Неполные формы записи оператора цикла

Любую из трех составных частей оператора **for** можно опустить, только обязательно надо сохранять точки с запятой между ними. Допускается запись

```
for( ; i < 5; i ++ )
```

(перед первым циклом никаких предварительных действий не происходит) или

```
for( i = 1; i < 5; )
```

В последнем случае значение переменной i может увеличиваться внутри цикла, в его теле:

```
for( i = 1; i < 5; ) i ++;
```

Можно записать и так:

```
for( i = 1; ; i ++ )
```

Тогда считается, что значением неуказанного условного выражения всегда будет true, то есть подобный цикл будет выполняться бесконечное число раз, и может быть прерван только оператором break.

Всегда надо очень тщательно проверять условие прекращения цикла. Если оно опущено или всегда будет равняться true, то программа может *зациклиться* — безостановочно выполнять одно и то же действие. Прервать работу такой программы бывает довольно трудно — не исключено и зависание Windows.

Можно также опускать две или даже все части оператора for. Например, профессиональными программистами нередко применяется такая форма оператора цикла:

```
for(;;)
  {
  // действия, среди которых обязательно должен быть
       оператор break
  }
```

Такой цикл может повторяться бесконечное число раз. Он удобен, когда точно неизвестно, когда следует остановить работу оператора for, а в теле имеется несколько условных операторов, самостоятельно определяющих момент остановки и необходимость выполнения оператора break.

Создаем новое окно

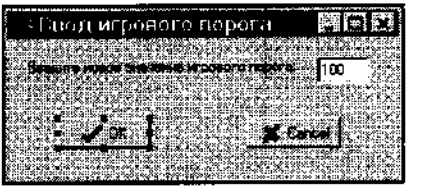
Создание новой формы

У вас может возникнуть желание играть с компьютером не до 100 очков, а до произвольного числа. Для этого целесообразно сделать данный параметр

настраиваемым, а настройку выполнять в отдельном диалоговом окне. Сейчас мы узнаем, как это можно сделать.

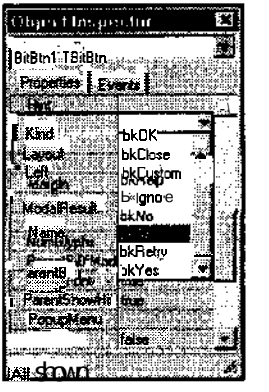
Добавить новое окно можно командой File • New Form (Файл • Новая форма). Сразу же в визуальном проектировщике появится новая форма под названием Form2.

Эту форму можно проектировать, размещать на ней различные элементы управления и работать с ней точно так же, как и с главной формой. Однако при запуске программы она не покажется на экране — ее надо вызывать специальным методом.



Новую форму можно назвать InputForm (свойство Name), а ее заголовком сделать строку Ввод игрового порога (свойство Caption). Щелчком на командной кнопке Save All файл с текстом реализации работы этой формы надо сохранить с именем IUnit.cpp.

На форме потребуется разместить одно поле ввода с названием Input и начальным отображаемым значением 100 (свойство Text), поле надписи-подсказки и две кнопки OK и Cancel (Отмена). Подобные кнопки стандартны для многих приложений, и в C++Builder на панели Additional (Дополнительным) можно найти удобный компонент BitBtn (кнопка с картинкой), которому соответствует класс TBitBtn.



Разместите две такие кнопки на форме. Для первой из них в свойстве Kind (Тип кнопки) выберите значение bkOk (стандартная кнопка OK), а для второй — bkCancel. При этом картинки на кнопках появятся автоматически.

Теперь следует научиться вызывать из главной формы это окно. Для этого в главном меню введите новый пункт &Порог с названием PItem (свойство Name в Инспекторе объектов). Для него не надо создавать вложенного подменю.

Дважды щелкните на этом пункте — C++Builder автоматически создаст метод реакции на его выбор, а уже из этого метода будет вызываться форма InputForm.

Логическое отрицание

Однако игровому порогу желательно задавать новое значение не в процессе партии, а только после ее окончания. Этот момент отслеживался в методах

обработки броска кубика и передачи хода (помните оператор return?) Новая проверка будет похожей, однако показывать вторую форму надо, только если игра закончена, в противном случае (партия продолжается) надо сообщить об этом игроку и завершить работу метода.

Проверку завершения игры мы выполняли методом Referee.GameFinish(). Но теперь блок условного оператора должен быть выполнен, только если игра *не* завершена. Как это записать?

В Си++ помимо логических операций «&&» и «||», для каждой из которых требуется два операнда, имеется также операция логического отрицания «!» (НЕ), которая записывается *всегда слева* от единственного операнда, то есть она работает как стандартная функция с единственным параметром, только не требует скобок. В рассматриваемом случае выражение, значением которого будет true, если игра продолжается, запишется так:

```
! Referee.GameFinish()
```

При вычислении данного выражения сначала будет получено значение Referee.GameFinish() (если игра продолжается, это будет false), а затем к нему будет применена операция !, и значение false изменится на противоположное значение true.

Операцию «!» корректно применять только к логическим значениям типа booL Результатом !true будет false, результатом !false — true.

Вызов новой формы

Чтобы вызвать вспомогательную (не главную — она вызывается автоматически) форму создаваемой программы, надо использовать метод этой формы Show() в обработчике выбора пункта меню Порог:

```
InputForm->Show();
```

Только чтобы переменная InputForm была доступна (иногда говорят — *видима)* в файле DiceUnit.cpp, в него надо включить заголовочный файл !Unit.h:

```
#include "IUnit.h"
```

Теперь программу можно запустить, и при выборе пункта меню Порог на экране появится новое окно. Однако работать с ним неудобно, потому что оно не диалоговое. Не закрыв это окно, можно переключиться на главное окно программы и продолжить с ним работу, что, конечно, неправильно. Необходимо дождаться, когда человек нажмет одну из кнопок во вспомогательной форме, не позволяя ему до этого вернуться к главному окну (такие окна, не допускающие перехода на другие окна своего приложения, называются *модальными.* Они больше известны как *диалоговые).*

Вернитесь в C++Builder и измените метод Show() на ShowModal() (показать как модальное окно):

```
InputForm->ShowModal();
```

Теперь форма InputForm будет работать так, как ей и полагается. Только при работе с подобными диалоговыми окнами обычно не разрешается менять их размеры. Чтобы сделать границы вспомогательного окна неизменяемыми, задайте свойству BorderStyle (Стиль границ) формы InputForm значение bsDialog (Стиль диалогового окна).

А как узнать, на какой кнопке щелкнул пользователь? Метод ShowModal() возвращает одно из предопределенных значений mrOk или mrCancel. С помощью условного оператора можно проверить, какое значение возвращено:

```
// если была нажата кнопка OK
if( InputForm->ShowModal() == mrOk )
```

В логическом блоке условного оператора надо перевести введенное в поле ввода Input текстовое значение в число и проверить его на «разумность величины» (допустим, от 50 до 1000). Это можно сделать с помощью другого условного оператора, как бы *вложенного* в предыдущий, а нужное число удобно сохранить во временной переменной input:

```
// если после показа InputForm нажата кнопка OK:
if( InputForm->ShowModal() == mrOk )
   {
   int input;
   // преобразовать введенный пользователем текст в
число
   input = StrToInt( InputForm->Input->Text );
   // проверить его на допустимый диапазон:
   if( input >= 50 &&   input <= 1000 )
   }
```

Из констант — в переменные

Но где сохранить новое значение? Ведь в качестве игрового порога используется константа GAME_FINISH_SCORE, а значения констант менять нельзя.

Действительно, это проблема, которая наглядно показывает, как плохо, когда какие-то требования к будущей программе не были сразу сформулированы и потому не были учтены при проектировании приложения. Если

бы сразу удалось додуматься до потребности в изменении игрового порога, то можно было исходно вместо константы использовать переменную.

Трудоемкость проекта можно оценить формально. Для этого есть специальные технологии *(например метод функциональных точек),* однако они сложны в использовании и применяются в основном при создании больших программных комплексов. Но далеко не всегда исходные требования удается сформулировать с самого начала. Часто бывает так, что при создании заказной программы у заказчика по ходу работы возникают новые пожелания, подчас требующие почти полной переделки всей внутренней структуры приложения. Люди, незнакомые с программированием, как правило, не способны оценить трудоемкость тех или иных модификаций готового продукта. Некоторые, на их взгляд, несущественные дополнения на самом деле требуют огромного труда и наоборот, самые сложные изменения пользовательского интерфейса могут потребовать всего лишь изменения нескольких свойств в Инспекторе объектов.

Поэтому все требования к будущей программе всегда желательно формулировать заранее, а в дальнейшем любые дополнения рассматривать как другой проект, например, по выпуску новой версии.

Наилучший в этой ситуации способ — сделать переменную из константы GAME_FINISH_SCORE. Для этого ее описание в файле Game.h надо убрать, а в класс TReferee добавить новую переменную:

```
int  GAME_FINISH_SCORE;
```

Ее название, выглядящее как название константы, менять не надо, потому что оно встречается в разных местах программы. Надо только проинициализировать ее начальное значение — создать конструктор TReferee и в его реализации записать один оператор:

```
GAME_FINISH_SCORE  =  100;
```

Но если теперь откомпилировать программу, то для оператора

```
if(  Referee.Human.Scores >=  GAME_FINISH_SCORE  )
```

появится сообщение об ошибке Undefined symbol "GAME_FINISH_SCORE" (Символ GAME_FINISH_SCORE не определен).

Действительно, ведь GAME_FINISH_SCORE превратилась из доступной во всех частях программы константы (как и должно быть) во внутреннюю переменную, и для обращения к ней надо указать содержащую ее переменную Referee:

```
Referee.GAME_FINISH_SCORE
```

а ошибочный оператор — изменить на

```
iff Referee.Human.Scores >= Referee.GAME_FINISH_SCORE )
```

Вывод нового значения в диалоговом окне

Если игровой порог менялся, но при последующих вызовах формы InputForm в поле ввода будет по-прежнему отображаться число 100, это значит, что интерфейс сделан неправильно. Так даже нельзя определить, до какого количества очков ведется игра. Чтобы этого избежать, в главную форму надо добавить новое поле надписи (назвать его можно GameFinish), а в реализацию метода **ShowInfo()** добавить вывод в это поле текущего порога:

```
GameFinish~>Caption = POROG_STR + IntToStr
      (Referee.GAME_FINISH_SCORE );
```

Константа **POROG_STR** описывается, как обычно, в файле Text.h:

```
const AnsiString POROG_STR = "Игра ведется до ";
```

Для того чтобы значение **Referee.GAME_FINISH_SCORE** исходно отображалось в поле ввода **Input** формы InputForm, надо в методе **PItemClick()** (выбор пункта меню Порог) перед вызовом формы присвоить свойству Text поля input соответствующее число, преобразованное в текстовый вид:

```
InputForm->Input->Text =
      IntToStr(Referee.GAME_FINISH_SCORE);
```

Теперь понятно, где надо сохранить введенное значение — в переменной **Referee.GAME_FINISH_SCORE**. И сразу же надо начать новую игру — ведь при проверке методом **Referee.GameFinish()** используется переменная **Referee.GAME_FINISH_SCORE**, и если ее увеличить в сравнении с текущим порогом, то в дальнейшем уже законченная партия может неожиданно продолжиться до нового числа очков, что не соответствует правилам.

```
if( input >= 50 && input <= 1000 )
        {
        Referee.GAME_FINISH_SCORE = input;
        Referee.NewGame();
        ShowInfo();
        }
```

В противном случае, когда значение input не укладывается в допустимые рамки, пользователь должен увидеть предупреждение о том, что введенное число не соответствует диапазону:

```
else ShowMessage(BAD_NUMBER_STR);
```

Избегайте глубокой вложенности условных операторов. Это значительно усложняет логику работы программы и процесс отладки. Оптимально использовать не более одного уровня такой *вложенности*. В принципе любой код *всегда* можно переделать так, чтобы он вообще не допускал вложенных операторов if. Например, кусочек данного метода, где обрабатывается нажатие кнопки OK, можно переписать следующим образом:

```
// если НЕ нажата кнопка OK, то завершить работу метода
if( InputForm->ShowModal() != mrOk ) return;
int input-
input = StrToInt( InputForm->Input->Text );
if( input >= 50 && input <= 1000 )
      {
      Referee.GAME_FINISH_SCORE - input;
      Referee.NewGame();
      ShowInfo();
      }
else ShowMessage(BAD_NUMBER_STR);
```

Теперь вложенных операторов if удалось избежать, а структура исходного текста стала более очевидной.

Подготовка законченного приложения

Позиционируем окна

Чтобы программа при запуске выглядела красиво, неплохо, чтобы главное окно (да и остальные тоже) располагалось в центре экрана (по умолчанию они будут располагаться в той точке, куда были помещены в визуальном проектировщике). Для такой настройки в Инспекторе объектов в раскрывающемся списке выбирается объект Form1 и его свойству Position (Положение) задается значение poScreenCenter из списка готовых значений.

В качестве заголовка окна (свойство Caption) можно указать строку Игра в кости.

Выбор подходящего значка

Пока в качестве значка созданной программы, отображаемого на Рабочем столе и на Панели задач, используется стандартный значок C++Builder.

Изменить его очень просто.

1. Вызовите диалоговое окно настройки параметров проекта командой Project • Options (Проект ▸ Параметры).
2. Выберите закладку Application (Приложение).
3. Нажмите кнопку Load Icon (Загрузить значок) — см. рис. 42.
4. С помощью стандартного диалогового окна открытия файла Windows выберите заранее подготовленный файл значка. Можно, например, взять готовый значок из стандартной библиотеки значков C++Builder (таким же способом, как это делалось ранее при выборе картинок для командных кнопок). Только во вложенной папке \Images надо выбрать папку с названием не \Buttons, а \Icons. В ней находится десять значков. Выберите для примера значок factory и щелкните на кнопке Открыть.

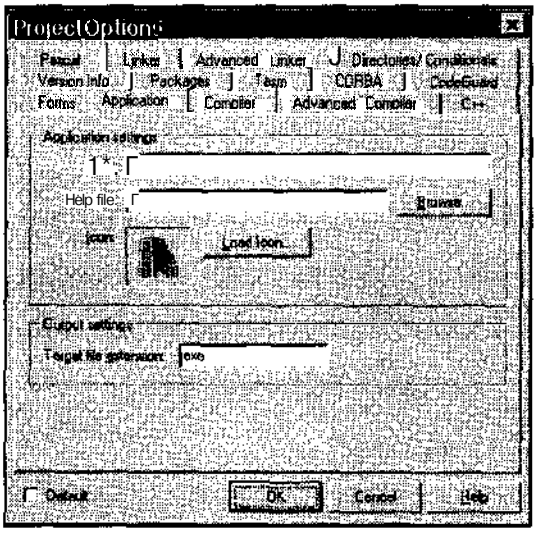


Рис. 42. Выбор значка для готового приложения

Значок в диалоговом окне проекта сменится на выбранный вами. В строке Title введите подпись Игра в кости, которая будет отображаться под значком. Щелкните на кнопке OK и запустите программу. Теперь, если взглянуть на Панель задач, то игровая программа будет представлена кнопкой с выбранным значком.

Создание готовой программы

Полноценное приложение должно работать, конечно, не только на том компьютере, на котором создавалось. Однако, скорее всего, наша Игра в кости на другом компьютере запускаться не будет. Это связано с тем, что при созда-

нии программы использовалось немало стандартных функций и классов, которые хранятся в специальных библиотеках. При отладке они подключались непосредственно в процессе работы программы, а на другом компьютере таких библиотек (они называются *Run-Time Libraries, RTL, библиотеки времени выполнения)* может не быть. Счастливое исключение составят только те компьютеры, на которых тоже установлена система C++Builder.

Библиотеки RTL можно включить внутрь программы, чтобы они все, что им надо, держали бы при себе и ее работа не зависела бы от наличия дополнительных системных файлов. Кроме того, в завершенном приложении желательно отключить всю отладочную информацию, которая несколько замедляет быстродействие и сказывается на размере программы (пока ее объем около 36 Кбайт).

Для этого в C++Builder (и в ряде других систем разработки) помимо отладочного режима работы Debug есть режим создания готового приложения Release. Эти режимы отличаются настройками компилятора и сборщика. Проще всего переключаться между ними так.

1. Вызвать диалоговое окно настройки параметров проекта командой Project ▸ Options (Проект ▸ Параметры).
2. Выбрать закладку Compiler (Компилятор).
3. Включить режим Full debug (Отладка) или Release (Публикация) в зависимости от того, что требуется в итоге получить.
4. Если создается законченный (Release) продукт, то для включения библиотек RTL в тело программы надо выбрать вкладку Linker (Редактор связей) и на ней сбросить флажок Use dynamic RTL (Использовать динамически подключаемые RTL).

Быстродействие программы сильно зависит от типа решаемой задачи. Если это числовые вычисления или обработка больших объемов информации, то существенный вклад в эффективность может внести настройка на конкретный тип процессора. Такая настройка выполняется на вкладке Advanced Compiler (Дополнительные возможности компилятора) в разделе Instruction set (Набор команд процессора). Тут доступны четыре типа процессора — 386, 486, Pentium и Pentium Pro. Надо учитывать, что ориентация на последние марки процессоров неприемлема при создании массовых продуктов, которые должны надежно выполняться на любых компьютерах. Поэтому использовать такие настройки лучше при решении своих конкретных задач. Или о поддерживаемых процессорах надо, как минимум, написать в сопроводительной инструкции.

5. Щелкнуть на кнопке OK и выполнить команду Project • Build (Проект ▸ Перестроить). Перестройку проекта в таких случаях надо выполнять обязательно, потому что некоторые части приложения при нажатии комбинации клавиш **Ctrl+F9** (компиляция) не будут меняться, если не менялись исходные тексты — это отслеживается автоматически, и в таком случае они не будут перекомпилированы при новых установках. А команда Build (Перестроить) выполняет компиляцию всех файлов проекта без исключения.

Режим Release в сравнении с Full debug создает более эффективную и компактную программу, но в этом режиме отлаживать программу (использовать точки остановки, выполнять ее по шагам и т. д.) *невозможно.*

Игра в кости с включенными библиотеками RTL займет теперь 91 Кбайт — почти в два раза больше, чем ранее, но зато сможет выполняться на любых компьютерах с Windows 9x.

Три кита Си++

Изученные в этой и предыдущих главах книги оператор присваивания, условный оператор и оператор цикла позволяют запрограммировать решение *любой,* сколь угодно сложной задачи. Это доказано математически. Используя мощный аппарат классов и богатую палитру визуальных компонентов C++Builder, вы теперь можете написать приложение практически любой направленности. Все будет зависеть лишь от вашей настойчивости и готовности постоянно практиковаться в создании всевозможных программ.

10. Обобщение — мать учения

Что дальше?

Если действительно с помощью классов и трех операторов можно запрограммировать любую задачу, то не правильнее было бы теперь рассмотреть наиболее часто встречающиеся компоненты C++Builder и способы работы с ними? Да, так и будет сделано, но немного попозже. Дело в том, что для решения задачи часто бывает недостаточно только одних операторов, даже самых мощных, и даже не трех, а тридцати трех. Любая более-менее интересная задача — это всегда обработка данных, нередко весьма сложно организованных.

Если потребуется написать программу, в которой будет вестись учет двадцати лучших достижений пользователей (что встречается во многих обучающих и игровых приложениях), то для хранения каждого достижения придется описать свою переменную, например, x_1, x_2, ..., x_{20}. А менять их значения будет еще сложнее — каждый раз, когда потребуется узнать N-й результат, придется писать двадцать условных операторов, проверяющих, соответствует ли iV номеру переменной.

А если надо обработать не двадцать однотипных значений, а тысячу? Каждый раз писать тысячи строк кода, чтобы выполнить одну, по сути, элементарную операцию, неразумно, а в реальных задачах постоянно возникает потребность в быстром и простом доступе к большим объемам однородной информации.

Для решения этой проблемы в Си++ введено понятие *массива*. Оно будет детально рассмотрено на примере создания программы для раскладывания пасьянса. На этом же примере будет выполнено закрепление и обобщение ранее пройденных вещей и изучение ряда понятий, которые необходимо знать при создании законченного коммерческого продукта.

Массив — основа для хранения однородных данных

Как описать массив

Массивы предназначены для хранения данных одного типа и для удобного доступа к содержимому этих данных. Синтаксис описания массива таков:

```
тип имя_переменной [ число_элементов ] ;
```

Например, вместо того, чтобы описывать двадцать переменных:

```
int x1, x2, /* ... */ x20;
```

можно записать совсем просто:

```
int x[20];
```

При этом в переменной *x* хранится не одно значение, а *последовательность* из 20 значений типа int. К каждому из них можно обращаться, указывая его номер *(индекс),* который может быть не только числовой константой, но и любым допустимым выражением Си++, возвращающим значение типа int.

В Си++ принято, что нумерация переменных начинается с нуля. То есть первый элемент имеет индекс 0, второй — индекс 1, двадцатый — индекс 19. Это очень важная особенность языка, немного непривычная начинающему программисту, со школы привыкшему вести отсчет от единицы.

Такая особенность связана с историей создания Си. Чтобы получить быстрый машинный код и сократить время компиляции (для программ размером в тысячи строк оно могло составлять часы) было решено пожертвовать удобством работы программистов ради получения высокоэффективных (для того времени) приложений. Дело в том, что элементы массива обычно располагаются в памяти строго последовательно, друг за другом, и при компиляции текста на Си (а теперь Си++) все упоминания переменной *x* заменяются на физический адрес первого элемента этого массива в оперативной памяти, а все обращения к элементам массива заменяются на машинные команды сложения адреса x с индексом — смещением соответствующего элемента относительно начата этой последовательности. Но такое смещение для первого элемента будет, очевидно, не 1 (в «штуках» предшествующих элементов), а 0. Для второго — смещение будет 1 и т. д. Поэтому, чтобы не тратить лишние такты процессора на выполнение дополнительных команд вычитания из смещения поправки на один элемент, и было решено приблизить технологию работы с массивами в Си к технологии, более естественной для реальной работе компьютера с памятью. Только для программистов она стала, к сожалению, менее естественной.

Но так или иначе, необходимо крепко запомнить, что индексация массивов в Си++ начинается с нуля.

Чтобы получить значение первого элемента массива x, надо записать:

```
x[0]
```

Обращение к десятому элементу:

```
x[9]
```

к двадцатому (последнему):

```
x[19]
```

Можно обращаться к элементам и так:

```
i = 5;
x[0] = x[i] + x[i+1];
```

(при этом в первый, если рассуждать общепринятым способом, элемент массива x запишется сумма шестого и седьмого элементов).

Размер массива

Размер массива в Си++ определяется заранее и указывается конкретным числом или константой. В отличие от индексов, в качестве которых может выступать любое выражение, размер массива на протяжении работы программы неизменен. Нельзя написать:

```
len = 20;
int x[len]; // неправильно!
```

Компилятор сразу сообщит об ошибке — «Constant expression required» (требуется константа).

Массивы, размер которых задается в программе заранее и в дальнейшем не изменяется, называются *статическими.* В Си++ можно также использовать и *динамические* массивы, размеры которых во время работы программы могут меняться, но для этого надо с помощью специальных операторов запрашивать у Windows память, потом освобождать ее и самому следить за корректностью ее использования. Это не очень простой процесс, и требуется он в основном для задач, где надо обрабатывать большие, но часто меняющиеся по размеру объемы информации.

Тип массива

В качестве типа элементов массива может выступать любой допустимый тип Си++, стандартный или введенный разработчиком, Это могут быть, в частности, классы. Например, если для решения какой-то статистической задачи требуется использовать сто игровых кубиков, то их можно описать так:

```
TDice dices[100];
```

Обращение к элементу массива классов

Ранее, когда простая *(простая* — в смысле не массив; еще говорят — *скалярная)* переменная Dice имела тип TDice, то обращение к ее методу GetDiceValue() записывалось в формате «переменная.метод»:

```
Dice.GetDiceValue()
```

А как записать, допустим, вызов метода пятидесятого элемента dices? Почти также, только вместо скалярной переменной надо подставить переменную с индексом:

```
dices[49] .GetDiceValue()
```

Б качестве индекса взято число 49, потому что пятидесятый элемент массива в Си++, как уже говорилось, будет иметь сорок девятый номер (нумерация начинается с нуля).

Некоторые стандартные классы C++Builder содержат в себе переменные-массивы. Например, с уже вам известным и активно используемым классом AnsiString (строка) можно работать как с массивом (последовательностью) символов, только не статическим, а динамическим, так как длина строки во время работы программы может меняться. Получить любой символ этой строки можно с помощью квадратных скобок индексации:

```
AnsiString s;
s = "12345";
```

s[0] будет равно "1", а s[4] = "5".

Некоторые классы C++Builder допускают работу с собой, как с массивами, однако реально они представляют собой не последовательности элементов, а обычные классы, для которых определена операция «[]» (операция так и называется — *«квадратные скобки»* или *«выделение элемента по индексу»).* Обращение s[0] по смыслу аналогично вызову метода класса AnsiString, который по номеру символа возвращает его значение. Просто использовать квадратные скобки значительно нагляднее и удобнее.

В дальнейшем речь будет вестись о способах работы только с обычными статическими массивами.

Многомерные массивы

Пока мы ограничивались упоминанием простых, *линейных* последовательностей элементов. Но более сложные структуры данных имеют, как правило, большую *размерность*. Например, при работе с электронной таблицей используются два измерения: строка и столбец. В большинстве стратегических компьютерных игр поле тоже двумерное — ширина и высота. Такие структуры было бы удобно и описывать соответствующим образом,

Как это сделать в Си++? Очень просто. Допустим, требуется подготовить массив, представляющий собой содержимое числовой электронной таблицы размером 1024 строки на 256 столбцов. Выглядеть нужное описание будет примерно так:

```
int  excel  [256][1024];
```

Переменная excel будет состоять из 256 последовательностей, по 1024 элемента типа int в каждой. Чтобы получить значение таблицы с координатами (100,100) (если счет идет с нуля), надо записать:

```
excel  [100][100]
```

Можно описать массив excel и по другому:

```
int  excel  [1024][256];
```

Смысл от этого не сильно изменится (1024 строки по 256 элементов).

В каком порядке указывать размерности массива, вопрос не праздный. Лучше всего придерживаться такого порядка, который позволит обращаться к элементам массива максимально наглядно. В частности, многие из нас со школы привыкли, что при записи координат на плоскости сначала указывается значение по оси X, а затем — по оси Y. Тогда область поверхности (например, карту) размером 64 единицы в ширину (ось X) и 128 единиц в высоту (ось Y) лучше описать так:

```
int Map [64] [128];
```

чтобы потом обращаться к нужным элементам в привычном порядке — сначала указывать координату X, а потом Y:

```
Map[x-4][y+1]
```

Только не надо забывать порядок выбора координат или измерений. Если обратиться к элементу

```
Map[y][x]
```

то программа не поймет, что это ошибка и вы перепутали индексы координат, и в итоге карта может оказаться перевернутой или повернутой на бок.

Можно описывать массивы с числом измерений, большим, чем два. Мы живем в трехмерном мире, и для его описания желательно использовать три координаты:

```
int World[20][30][10];
```

Вообще в Си++ разрешается использовать неограниченное число измерений массива. Однако на практике редко можно встретить задачи, где требовалось бы использовать четырех- и более мерные массивы (это связано, в первую очередь, с человеческой психологией). Известны, в частности, единичные случаи использования в некоторых программах семимерных массивов.

Контролируем границы

С проблемой корректного выбора смыслового содержания каждого измерения массива тесно связана проблема контроля за выходом индекса из допустимых границ. Если массив описан как

```
int x[20][2];
```

то можно обращаться к его содержимому и так:

```
x[10] [1]
```

и, случайно перепутав индексы, так:

```
x[1] [10]
```

Последняя запись логически, очевидно, неверна — ведь длина второго измерения *x* равна двум, а мы указываем в качестве номера элемента «10». Однако на физическом уровне произойдет корректное обращение к какому-то элементу массива *x* (ведь все его элементы занимают в памяти 2*20–40 ячеек, а обращение выполняется к 1*10 = 10-му элементу, который расположен в диапазоне этих ячеек). Этот элемент вполне может быть и нужным, но может быть и совершенно другим (каким конкретно, зависит от среды разработки — от того, как она располагает массивы в памяти).

Такие ошибки выявлять очень трудно, но и это еще полбеды. В Си++ допустима даже такая запись:

```
int x[10];
y = x[50];
```

Как ни удивительно, это не считается ошибкой и компилятор не выдаст никакого предупреждения. Во время работы программы при обращении к 50-му элементу *x* от начала последовательности, реально занимающей 10 ячеек (а точнее, 10*4 байтов — для типа int выделяется 4 байта или 32 бита), просто будет отсчитано 50 ячеек, и то, что хранится по соответствующему адресу, будет трактоваться программой как int.

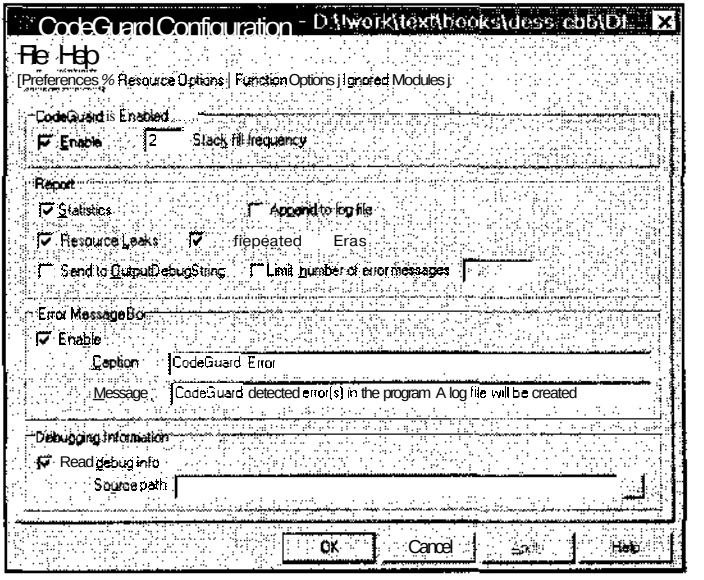
Самое страшное, когда допускается ошибка, обратная приведенной:

```
int x[10];
x[50] = y;
```

По адресу физической памяти, отстоящему от начала массива *x* на 50 элементов (50*4 байтов) будет произведена запись содержимого переменной *y*. А что реально записано по такому адресу, не известно. Может быть, это какие-то другие данные программы, которые затрутся значением переменной *y,* может быть, это программный код приложения, а может быть, и системная область, изменение которой приведет к краху Windows.

В языке Си++ нет *никаких* способов контроля за допустимостью значений индексов массивов. Эта работа полностью возлагается на программистов.

К счастью, в C++Builder 5 появилось прекрасное средство для выявления подобных логических ошибок. Оно один раз уже упоминалось в книге и называется CodeGuard (Страж Кода). Его лучше всего всегда применять и начинающим, и профессиональным разработчикам.



Чтобы использовать это средство, сначала в настройках проекта (команда Project • Options, вкладка CodeGuard) надо включить все флажки, определяющие, что отладка программы будет выполняться с помощью Стража, а потом полностью перекомпилировать проект. Чтобы дополнительно настроить CodeGuard, надо дать команду Tools • CodeGuard Configuration (Сервис > Конфигурация CodeGuard)— через несколько секунд Страж загрузит структуру текущей программы, и в диалоговом окне можно будет изменить настройки его работы. Конфигурацию по умолчанию можно не менять, достаточно щелкнуть на кнопке OK (главное, чтобы был помечен флажок Enable (Включено) в разделе "CodeGuard is Enabled").

Теперь Code Guard станет внимательно следить за возможными нарушениями в работе программы, и при попытке обратиться, например, за пределы массива x он сразу выдаст предупреждающее сообщение и укажет на ошибочный оператор в программе: `x[50] = y;`.

Массивы и циклы — родные братья

Чтобы эффективно обработать содержимое массива, удобнее всего применять циклы. Например, чтобы вычислить сумму всех элементов массива

```
int x[100];
```

можно использовать такой код на Си++:

```
int i, sum; // i - счетчик
sum = 0; // исходно сумма равна нулю
for( i = 0; i < 100; i ++ )
  sum += x[i];
```

Обратите внимание на то, что счетчик i принимает значение от 0 до 99, а условием выхода из цикла служит проверка счетчика на «больше или равно 100».

Если в теле цикла не происходит обращений к элементам массива, то лучше задавать начальное значение счетчика равным 1, чтобы контроль за числом повторений выглядел более естественно и не возникало путаницы из-за проблем с нулевой нумерацией.

Пасьянс «Колодец»

Почему пасьянс?

Пасьянс «Колодец» выбран в качестве примера по трем причинам. Во-первых, он очень хорошо подходит для знакомства с понятием массива.

Во-вторых, с его помощью удастся познакомиться с компонентами C++Builder, ответственными за работу с картинками и графикой, с важнейшим методом окна, связанным с его перерисовкой, а также со способом обработки действий пользователя (щелчком кнопкой мыши). И в-третьих, созданный пасьянс «Колодец» будет практически законченным продуктом и вполне может послужить хорошей основой для профессионального создания других пасьянсов, а карточных игры традиционно пользуются неплохим спросом на рынке условно-бесплатного (shareware) программного обеспечения.

Новый проект

Создание программы-пасьянса начинается, как обычно, с создания нового проекта C++Builder — сначала закрывается текущий проект (команда File • Close All), а затем открывается новый (команда File • New Application). Его лучше сразу сохранить в отдельной папке, назвав файл Си++ MainUnit.cpp, а сам проект — WellProject *(Well* по-английски — *колодец).*

Теперь надо разобраться с правилами.

Правила «Колодца»

1. Для «Колодца» требуются две колоды по 52 карты.
2. Карты будут раскладываться по пяти областям.
3. 4 стопки по 11 карт лицом вниз раскладываются на стены.
4. Верхние карты стен открываются и кладутся лицом вверх на соответствующие пристенки.
5. Верхние карты стен открываются.
6. Оставшиеся 60 карт (колода) кладутся в сторону лицом вниз.
7. Короли перекладываются в колонны (колодец) лицом вверх. На разных колоннах не может быть королей одинаковых мастей.
8. Если среди открытых карт есть тузы, то любой из них может быть положен лицом вверх в пустой склад.
9. Если карта перемещена с пристенка, а под ней больше ничего нет, то на пустой пристенок кладется карта с соседней стены.
10. Если под перемещенной со стены картой стопка еще не пуста, то ее верхняя карта переворачивается лицом вверх.
11. Цель игры — собрать все 104 карты на колонны с королями.

12. На колонны карты одинаковой масти кладутся в нисходящем порядке (на короля — дама, на тройку — двойка, на двойку — туз, на туза — король и т. д.). На колонны можно перемещать любую допустимую по правилам карту, открытую в пасьянсе лицом вверх. С самих колонн карты перемещать запрещено.

13. Открытые карты можно также перемещать на пристенок и в склад — на карты совпадающей масти в восходящем порядке (на даму — короля, на короля — туз, на туза — двойку и т. д.).

14. Когда все возможности перемещения карт закончатся, из отложенной колоды во вспомогательный ряд выкладываются 5 карт. Их можно попытаться переместить на стопки колон, пристенка и склада.

15. Далее процесс выкладывания 5 карт поверх ранее открытых продолжается — до исчерпания колоды.

16. Когда колода кончается, оставшиеся стопки со вспомогательного ряда собираются справа налево в колоду, которая после этого не тасуется, и снова выкладываются но 4 карты, опять до окончания колоды.

17. Процесс так же повторяется с выкладыванием линейки из 3, 2 и одной карты.

18. Если собрать все карты на колонны не удалось, значит, пасьянс не сошелся.

Надо отметить, что пасьянс этот довольно трудный. Основной его принцип — стараться поскорее открыть 36 карт, спрятанных в стенах.

Где взять рисунки карт?

Перед тем, как приступить к проектированию программы, надо подготовить 52 картинки карт, одну картинку рубашки карты и одну картинку «подкладки» для показа нужных положений карт на игровом столе.

Готовые картинки карт хранятся в стандартной библиотеке Microsoft, которая называется cards.dll. Ее можно найти в каталоге \Windows\System. Если файла cards.dll там нет, значит, на компьютере не установлены игры. Их можно быстро добавить, выбрав в панке Панель управления значок Установка и удаление программ и переключившись на вкладку Установка Windows. В разделе Стандартные находится пункт Игры, который надо пометить и щелкнуть на кнопке OK. Когда установка завершится, в Главном меню в разделе Программы • Стандартные появится подраздел Игры. Среди них будут три пасьянса, для которых и предназначена библиотека cards.dll.

После того как библиотека найдена, из нее нужно каким-то способом изъять подходящие картинки. В этом может помочь редактор ресурсов Workshop, дистрибутив которого находится на инсталляционном диске с C++Builder. После установки этой программы (вызовом программы setup.exe из папки \Workshop\disk1) надо ее запустить (значок редактора ресурсов — Resource Workshop), выполнить команду File • Open Project (Файл • Открыть проект), в качестве типа файла в раскрывающемся списке FileType (Тип файла) указать DLL, VBX, CPL Library, найти файл cards.dll и щелкнуть на кнопке OK.

В столбце BITMAP (растровый рисунок) окошка cards.dll появится список чисел. Числа от 1 до 52 соответствуют рисункам карт, от 53 до 58 — рисункам разнообразных рубашек, от 678 до 684 соответствуют нескольким вспомогательным рисункам.

При выборе любого из чисел справа будет показан соответствующий рисунок (см. рис. 43).

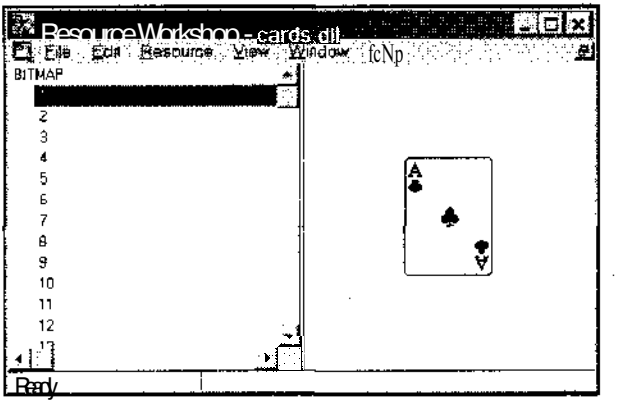


Рис. 43. Просмотр содержимого динамической библиотеки DLL

Теперь рисунки надо сохранить — каждую карту в отдельном файле. Делается это так.

1. Щелчком кнопкой мыши выбираем число.
2. Нажимаем правую кнопку мыши.
3. В контекстном меню выбираем пункт Save resource as (Сохранить ресурс как).
4. Картинка сохраняется с нужным номером и расширением .bmp.

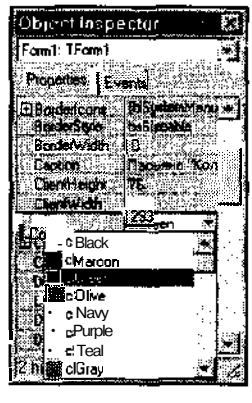
Таким образом надо сохранить по порядку все рисунки, начиная с первого по пятьдесят второй, в качестве фона можно выбрать рисунок номер 58,

сохранив его в файле 53.bmp, а в качестве подкладки — рисунок номер 53 (он сохранится в файле 54.bmp).

Все изображения, хранящиеся в явном виде или содержащиеся в программах, входящих в поставку Windows, являются исключительной собственностью корпорации Microsoft (как и любые другие данные Windows). Это означает, что картинки, которые с помощью Редактора ресурсов были извлечены из файла cards.dll, можно использовать только в качестве учебных пособий. Распространять их вместе с программой-пасьянсом не разрешается. Если в дальнейшем у вас возникнет желание сделать свое оригинальное карточное приложение, то надо либо нарисовать подходящие картинки самому, либо поискать бесплатные рисунки в Интернете — там они наверняка имеются.

В принципе, есть еще один юридически корректный способ доступа к ресурсам, записанным в cards.dll — это загрузка данного файла во время работы пасьянса и выборка нужных картинок из него динамически. Однако этот способ сложнее, он требует более глубокого знания внутреннего устройства Windows и DLL-библиотек. Кроме того, не у каждого пользователя установлены стандартные игры, а значит, файла cards.dll может и не быть. Плох этот способ еще и тем, что он по большому счету не является визуальным и не позволяет использовать удобные компоненты C++Builder. А ведь вполне может возникнуть потребность сменить вид карт или их рубашку на какую-то более оригинальную. В этом случае работа с файлом cards.dll напрямую не поможет.

Проектируем интерфейс



Сам интерфейс будет очень простым. Прежде всего, главное окно должно иметь зеленый фон — как в классических пасьянсах. Он устанавливается с помощью свойства Color (Цвет) формы Form1 — для него надо выбрать значение clGreen.

Форме Form 1 также надо дать подходящий заголовок — Пасьянс «Колодец» (свойство Caption).

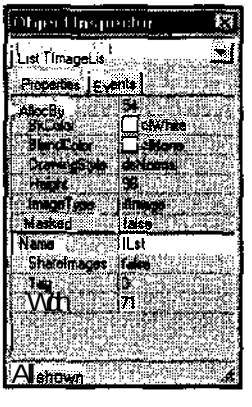
При запуске окно желательно сразу развернуть, так как на нем придется располагать много карт. Это можно сделать, установив в признаке WindowState (Статус окна) значение wsMaximized.

Стандартная карта из библиотеки cards.dll имеет размеры 71x96 точек. Чтобы расположить несколько стопок в окне в соответствии с правилами, надо переключить экран в разрешение как минимум 800x600 точек. В дальнейшем подразумевается, что вся работа с программой происходит именно в таком разрешении.

С помощью компонента MainMenu давайте создадим меню, которое будет состоять из трех подпунктов — Новая игра, разделитель и Выход.

Кроме того, в любом месте формы надо разместить компонент ImageList (Список картинок) с панели Win32. Он будет хранить в себе весь набор рисунков, требуемых для пасьянса.

Загружаем карты в компонент



С помощью ImageList очень удобно работать с наборами картинок одинакового размера. Фактически этот компонент представляет собой массив изображений, обращение к которым происходит по их индексу, начинающемуся, как вы догадались, с нуля.

Для загрузки колоды в массив ImageList надо выполнить следующие действия.

1. Выбрать объект ImageList на форме.
2. В свойстве AllocBy (Число картинок) указать число 54 (52 карты, фон и подложка).
3. В свойстве BkColor указать clWhite (основной цвет лица карты).
4. В свойстве DrawingStyle (Стиль рисования) указать dsNormal (Нормальный).
5. В свойстве Height (Высота рисунков) указать 96 (такую высоту в пикселах имеют все изображения карт из файла cards.dll).
6. В свойстве ImageType (Тип рисунка) указать itImage (Рисунок).
7. В свойстве Masked (производится ли отрисовка картинки с помощью специальной маски, шаблона рисунка) указать false (нет).

8. В свойстве Name (Название объекта) указать IList (имя соответствующей переменной).
9. В свойстве Width (Ширина рисунков) указать 71 (такова ширина рисунков в файле cards.dll, измеренная в пикселах).
10. Дважды щелкните левой кнопкой мыши на объекте ImageList. Откроется Редактор списка рисунков (см. рис. 44).

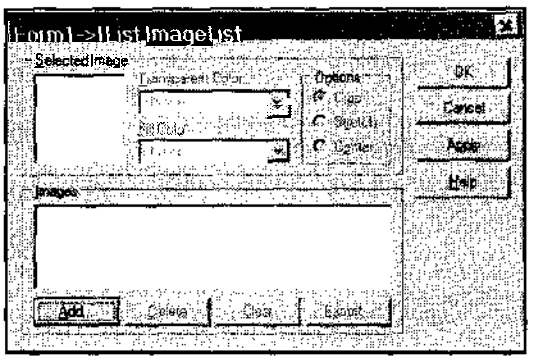


Рис. 44. Окно редактора списка рисунков

С помощью кнопки Add (Добавить) выбирается очередная картинка карты (ранее извлеченный из файла cards.dll файл с расширением .bmp) и добавляется в список. Добавленные карты показываются в списке Images с соответствующими индексами (см. рис. 45).

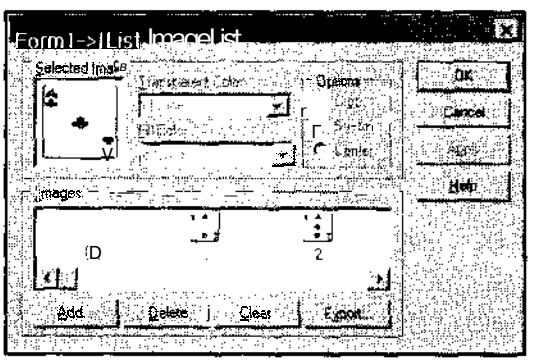


Рис. 45. Создание списка рисунков в редакторе

Карты надо добавлять точно в том порядке, в каком они были сохранены — первым должен идти файл 1.bmp, вторым — файл 2.bmp, 52-м — 52.bmp. Затем

добавляется файл с рубашкой, а последним — файл с подложкой. Проверить правильность сформированного списка можно, выделив в списке карту с индексом 0 (это должен быть туз пик), и затем с помощью клавиши «стрелка-вправо» просмотреть все следующие карты — они будут показываться в окне Selected Image (Текущая выбранная картинка).

Надо очень тщательно проследить за тем, чтобы карты шли в строгом порядке масти и возрастания значений. Если этот порядок будет нарушен, программа в дальнейшем может показывать карты, не соответствующие реальным значениям.

В раскрывающемся списке Transparent Color (Прозрачный цвет) Редактора списка необходимо установить значение clNone (Нет прозрачности рисунков), в списке Fill Color (Цвет заполнения вокруг картинки) — тоже clNone, и установить переключатель в разделе Options (Настройки) в положение Crop (показывать картинку, позиционируя ее по верхнему левому углу). Теперь закройте диалоговое окно щелчком на кнопке OK, и список картинок будет сформирован.

Как ни удивительно, но на этом проектирование интерфейса закончено. Карты, используемые в программе, будут рисоваться в окне с помощью специального метода.

Реализация необходимых классов

Проектирование колоды карт

Прежде чем начать проектирование класса Карточная_колода, надо определить, что будет представлять собой одна карта (класс TCard). Карта описывается мастью и значением, а также признаком того, расположена ли она в колоде, или находится на игровом поле.

Сама колода (класс TPack) будет содержать массив из нужного числа карт (например, 52), иметь конструктор, который проинициализирует массив карт начальными значениями, и методы тасования колоды, взятия карты из колоды (первую сверху или по индексу), возврата ее обратно в колоду и проверки, имеется ли в колоде хотя бы одна карта.

Создав соответствующие классы и их реализацию, в дальнейшем можно будет применять готовую «виртуальную» колоду для разработки любых карточных игр.

Класс «Карта»

Размещать определение классов «Карта» и «Колода» лучше всего в отдельном файле. Его потом можно будет без проблем включать в другие приложения, связанные с картами (командой Project > Add To Project).

Создается такой файл (новый модуль программы) командой File • New (Файл > Создать)и выбором значка Unit (Модуль). Новый модуль надо сразу сохранить под именем CardUnit.cpp.

В заголовочном файле CardUnit.h этого модуля будет размещаться описание двух классов. Первым станет класс «Карта»:

```
class TCard
{
private:
 int   Suit; // масть
 int   Value; // 0- туз, 1- 2 , 2- 3, ..., 10- валет,
  11- дама, 12-король

public:
 void  Init(int suit, int value);
 int   GetSuit();
 int   GetValue();

 bool Is;
};
```

Переменная Suit хранит масть (договоримся, что масти будут нумероваться так: 0 — трефы, 1 — бубны, 2 — червы, 3 — пики). Переменная Value — значение карты (0 — туз, 1 — двойка,..., 9 — десятка,..., 12 — король).

Значение каждой карты изменять во время работы программы некорректно. Правильный подход — исходно сформировать колоду из 52 переменных класса TCard, каждая из которых будет иметь оригинальное значение, которое в дальнейшем менять не потребуется. Поэтому переменные Suit и Value спрятаны в разделе protected — это значит, что извне методов класса TCard к ним обращаться нельзя. Однако надо разрешить считывать их установленные раз и навсегда значения с помощью методов GetSuit() и GetValue(). Задавать такие значения будет метод Init().

А вот переменная Is размещена в открытой (public) части класса и свободно доступна для изменения. Так сделано для удобства — ведь карты будут постоянно браться из колоды и возвращаться в нее обратно. Переменная Is равна true, если данная карта находится в колоде.

Класс «Колода»

Описание колоды тоже будет достаточно простым:

```
class TPack
{
private:
 TCard Cards[PACK_TOTAL];

public:
 TPack();
 void Shuffle();
 void AddCard(TCard card);
 TCard GetCard();
 TCard GetIndexCard(int i);
 bool IsCard();
};
```

В классе имеется массив из числа карт, определенного константой PACK_TOTAL (она будет описана чуть позже). Этот массив извне класса Колода недоступен. Конструктор TPack() создаст колоду, метод Shuffle() — перетасует ее, метод GetCard() вернет верхнюю карту, удалив ее из колоды (на самом деле у соответствующей карты просто изменится флажок Is), метод GetIndexCardO вытащит из колоды нужную карту, метод IsCard() скажет, пуста колода или нет.

Требуемые константы

В принципе, количество мастей (четыре) и число значений карт (от двойки до туза, тринадцать) можно использовать в программе не как символьные названия (константы), а просто как числа, но для некоторых специфических, но весьма распространенных игр мастей и значений карт может быть и значительно больше, и меньше (например, для преферанса нужны те же 4 масти, но 8 значений, начиная с семерки), поэтому лучше такие числа описать в виде констант:

```
const SUIT_NUM = 4; // число мастей
const VALUE_NUM = 13; // число значений карт
```

— А какой тогда будет размер колоды?

— Он равен числу мастей, умноженному на число значений.

```
const PACK_LEN = VALUE_NUM*SUIT_NUM;
```

При описании констант можно использовать произвольные выражения, в которых также используются числа или константы.

Такой подход очень удобен тем, что при изменении, например, числа мастей не надо вручную менять связанные с этим значением другие константы, тот же размер колоды — он просто будет пересчитан автоматически.

Колод для пасьянса часто требуется не одна, а две. Число колод и суммарное число всех карт опишутся так:

```
const PACK_NUM = 2; // число колод
const PACK_TOTAL = PACK_NUM*PACK_LEN; // общее число
  карт
```

Теперь можно определить еще две вспомогательные константы — значения короля и туза:

```
const KING = 12; // король
const ACE = 0; // туз
```

Часто бывает удобно обращаться к картам по их мнемоническим названиям, а конкретные значения от игры к игре могут меняться.

Описание констант надо разместить в начале файла CardUnit.h, непосредственно перед описанием класса TCard.

Реализация карты

Реализации всех методов записываются в файле CardUnit.cpp.

При инициализации карты внутренним переменным класса просто присваиваются значения параметров метода Init(). Признак наличия карты в колоде Is устанавливается равным true (да).

```
void TCard::Init(int suit, int value)
{
Suit = suit;
Value = value;
```

```
Is = true;
}
```

Методы, предназначенные для получения значений Suit и **Value,** можно записать одной строкой;

```
int TCard::GetSuit() { return Suit; }
int TCard::GetValue() { return Value; }
```

Конструируем колоду с помощью вложенных циклов

Конструктор должен присвоить оригинальные значения картам двух колод (первая колода от 0 до 51, вторая — от 52 до 103). Можно сформировать один цикл со счетчиком, который пробежится по всему массиву карт, но такой подход плох тем, что процесс определения, какой должны быть масть и значение очередной карты одной из двух колод, весьма сложен и неочевиден. Гораздо проще последовательно перебрать все масти (от 0 до 3), а для каждой из них — перебрать все значения карт (от 0 до 12). В итоге получится 4*13 – 52 карты, целая колода.

Опишем счетчики масти и значений:

```
int suit, value;
```

Цикл перебора четырех мастей запишется так:

```
for( suit = 0; suit < SUIT_NUM; suit ++ )
  {
  }
```

Внутри этого цикла надо организовать второй, *вложенный* цикл, который для текущего значения suit переберет весь диапазон значений карт и выполнит инициализацию очередной карты:

```
for( suit = 0; suit < SUIT_NUM; suit ++ )
  {
   for( value = 0; value < VALUE_NUM; value ++ )
     {
     // инициализация карты масти suit и значения
  value
     }
  }
```

Лишние фигурные скобки можно отбросить:

```
for( suit = 0; suit < SUIT_NUM; suit ++ )
```

```
for( value = 0; value < VALUE_NUM; value ++ )
  {
   // инициализация карты масти suit и значения
value
  }
```

Это классическая запись вложенных циклов.

Не надо забывать и про то, что инициализировать надо не одну колоду, а две, то есть вышеупомянутую конструкцию надо повторить два раза. В этом поможет еще один, третий цикл, только теперь его надо вынести «наружу», охватив им двойной цикл инициализации одной колоды (52 карты):

```
for( pack_num = 0; pack_num < PACK_NUM; pack_num ++ )
   for( suit = 0; suit < SUIT_NUM; suit ++ )
     for( value = 0; value < VALUE_NUM; value ++ )
        {
        // инициализация карты масти suit и
        //значения value
        }
```

Конечно, переменная-счетчик *pack_num* должна быть предварительно описана.

А как узнать, какую карту надо инициализировать? Так как процесс инициализации начнется с нулевой карты массива Cards, то надо добавить внутрь метода еще одну переменную **card_num**, присвоить ей ноль перед выполнением тройного цикла, и использовать как индекс при инициализации очередной карты, увеличивая его каждый раз после такой инициализации на единицу.

Итоговый вариант конструктора колоды запишется следующим образом:

```
TPack::TPack()
{
 int card_num, pack_num, suit, value;
 card_num = 0;
 for(pack_num = 0; pack_num < PACK_NUM; pack_num ++)
   for( suit = 0; suit < SUIT_NUM; suit ++ )
     for( value = 0; value < VALUE_NUM; value ++ )
```

```
        {
        // инициализация карты
        Cards[card_num].Init(suit, value);

        // переход к следующей карте
        card_num++;
        }
}
```

После выполнения конструктора в массиве Cards окажутся 104 упорядоченные по мастям и значениям карты.

Тасуем колоду

В исходном состоянии, после создания объекта класса TPack, колода является упорядоченной, то есть использовать ее в пасьянсе не имеет смысла — ведь карты в ней идут по порядку, и фактор случайности в игре отсутствует. Поэтому колоду сначала надо перетасовать.

Готовых алгоритмов тасовки карточных колод очень много, но в реальности быстро добиться действительно полностью случайного порядка последовательности объектов довольно сложно. В нашей программе мы реализуем не самый оптимальный, но зато очень простой и быстрый алгоритм суть которого состоит в следующем.

Сто четыре раза (размер двух колод) из общей пачки карт изымается случайная карта и меняется местами с последней (104-й) картой. Как ни покажется на первый взгляд странным, такой алгоритм выдает достаточно хорошо перемешанную колоду. В принципе, эффективность тасования можно еще повысить, взяв вместо числа перестановок 104, например, 1000. Тут вы можете поэкспериментировать сами.

```
void  TPack::Shuffle()
{
int n, rnd;
TCard card;

// каждой карте присваиваем признак "находится в колоде"
for! n = 0; n < PACK_TOTAL; n ++ )
      Cards[n].Is = true;
```

```
// PACK_TOTAL раз выполняем перестановку карт
for( n = 1; n <= PACK_TOTAL; n ++ )
    {
    // выбираем случайный номер карты в диапазоне от 0
    // до PACK_TOTAL-1
    rnd = random(PACK_TOTAL);

    // запоминаем последнюю карту
    card = Cards[PACK_TOTAL-1];

    // на ее место записываем карту с номером rnd
    Cards[PACK_TOTAL-1] = Cards[rnd];

    // на место карты с номером rnd записываем карту,
    // которая ранее была последней:
    Cards[rnd] = card;
    }
}
```

Стандартная функция random() описана в заголовочном файле <stdlib.h>, который надо включить в файл CardUnit.cpp командной строкой

```
#include <stdlib.h>
```

Обратите внимание на то, что в этом методе происходит присваивание значений переменных пользовательского класса (TCard). Операция присваивания «=» в отличие от многих других в Си++ определена и для классов. В реальности при выполнении оператора

```
card = Cards[PACK_TOTAL-1];
```

происходит простое копирование блока информации, отведенного для переменной Cards[PACK_TOTAL-1], в блок, отведенный для хранения переменной card. А вот такие операции, как «<» или «==», для пользовательских классов не определены, и их при необходимости надо описывать вручную. В принципе, можно переопределить и оператор «=», но в большинстве случаев это не требуется.

Изъятие карты из колоды

По ходу игры карты из колоды могут изыматься. В действительности они никуда из массива Cards, конечно, не исчезают, только признак Is меняется на false.

«Брать» карты из колоды надо аккуратно — сверху. Для удобства допустим, что верх колоды — это карта с индексом 0. Под ней лежит карта с индексом 1 и т. д. Тогда, чтобы взять из колоды верхнюю карту, надо пробежаться по массиву Cards, начиная с нулевой карты, и проверить признак Is каждой карты. Когда найдена карта с признаком Is, равным true (то есть карта в колоде есть), то его надо изменить на противоположный, и прервать цикл, использовав данную карту в качестве возвращаемого методом GetCard() значения.

А вот если цикл закончился нормально, значит, ни одна карта не имеет признака Is, равного true, то есть ни одной карты в колоде нет. Но это означает, что метод GetCard() вызван откуда-то ошибочно. Об этом желательно сообщить разработчику, показав диалоговое окно с информацией Запрошена карта из пустой колоды. Такую строку не обязательно выделять в отдельный файл с текстовыми константами — ведь это просто информация о внутренней ошибке программы, которая будет полезна только ее создателю.

Кроме того, в конец метода GetCard() (а он может быть достигнут только по ошибке) надо, тем не менее, добавить оператор return, возвращающий какую-нибудь карту, например, *нулевую*. Дело в том, что компилятор не может знать о возможных логических ошибках, но он понимает, что данный метод должен возвращать значение типа TCard. Поэтому при компиляции будет получено предупреждение Function should return a value (Функция или метод должны возвращать значение), а предупреждения — это всегда плохо.

```
TCard TPack::GetCard()
{
 int  i ;
 for( i = 0; i < PACK_TOTAL; i ++ )
   if( Cards[i].Is )
     {
     Cards[i] . Is = false;
     return Cards[i] ;
     }
```

```
ShowMessage("Запрошена карта из пустой колоды");
return Cards[0];
}
```

Иногда бывает необходимо выяснить, какая карта хранится в некоторой позиции внутри колоды. Этому поможет короткий метод

```
TCard TPack::GetIndexCard(int i) { return Cards[i]; }
```

который по индексу i возвращает соответствующий элемент массива Cards. По сути, он аналогичен простому обращению к массиву Cards, просто колода скрыта в разделе protected, поэтому напрямую к ней обращаться нельзя.

Добавление карты в колоду

С первого взгляда может показаться, что добавить карту в колоду можно, просто найдя первый элемент массива Cards со значением переменной Is, равным false (не в колоде), и сразу записав в нее переданную в качестве параметра карту. Однако это не так — ведь каждому элементу массива Cards соответствует *строго определенная карта* с конкретным значением и мастью (например, по индексу 0 располагается туз треф). Поэтому надо поступить по другому — найти в Cards карту, значение и масть которой совпадает с добавляемой картой, и изменить переменную Is соответствующего элемента на true, после чего прервать цикл (ведь дальше выполнять его не надо, это пустая трата времени) и покинуть метод. А вот если все элементы в Cards просмотрены, и нужной карты там нет, то это явно логическая ошибка, о чем и нужно выдать сообщение.

```
void TPack::AddCard(TCard card)
{
int i;
for( i = 0; i < PACK_TOTAL; i ++ )
   if( !Cards[i].Is &&
       Cards[i].GetValue() == card.GetValue() &&
       Cards[i].GetSuit() == card.GetSuit() )
     {
     Cards[i].Is = true;
     return;
     }
ShowMessage("Не удалось добавить карту");
}
```

Проверка на опустошение колоды

Последний метод TPack — проверка, есть ли в колоде хотя бы одна карта. Он реализуется совсем просто — последовательно перебираются элементы Cards, и если встречается карта, которая реально имеется в колоде (переменная Is равна true), то и метод сразу возвращает true, а оставшиеся карты можно не проверять. А вот если цикл закончился нормально, значит, все карты в колоде отсутствуют — возвращается значение false.

```
bool TPack::IsCard()
{
int i;
for( i = 0; i < PACK_TOTAL; i ++ )
  iff Cards[i].Is ) return true;
return false;
}
```

В итоге получена готовая реализация классов Карта и Колода, выделенная в файлах CardUnit.cpp и CardUnit.h, которые можно включать в любые другие программы, связанные с карточными играми.

Основная часть пасьянса

Проектирование логики работы главной формы

Прежде всего необходимо определить способ хранения игровых данных. От этого будет зависеть, удастся ли реализовать логику приложения легко и наглядно, или же придется изобретать хитроумные алгоритмы для решения ряда с первого взгляда простых задач.

При определении структуры данных будущей программы во многих случаях удобно исходить из принципа *информационной избыточности;* не надо пытаться придумывать слишком сложные и запутанные структуры и способы хранения информации. В различных книгах нередко можно встретить упоминания списков, «деревьев» и других структур, которые дают определенный выигрыш в памяти (например, вместо 100 Кбайт потребуется 15 Кбайт) или выигрыш в быстродействии (процентов на 20), но усложняют логику работы и часто приводят к трудно обнаруживаемым ошибкам. При современном развитии персональной вычислительной техники память вообще лучше не экономить (в разумных пределах, конечно). Как правило, данные, представленные с определенной избыточностью — в виде массива, некоторые элементы которого заведомо не будут использоваться, позволяют просто и наглядно обрабатывать себя с помощью циклов.

Давайте посмотрим на структуру игрового поля, где будет раскладываться пасьянс. На нем в смысловом плане выделяются пять групп, объединяющих несколько стопок карт: стена, колонны, пристенок, склад и вспомогательный ряд. Поэтому длина главного массива пасьянса (назовем его Well типа TCard), в котором будет храниться текущее расположение карт на столе, составит пять элементов:

```
TCard Well [5];
```

А какое максимальное число стопок может быть в каждой группе? В первых четырех — по четыре, а во вспомогательном ряду допускается выкладывать пять стопок. Значит, второе измерение массива Well — число стопок в соответствующей группе — тоже будет иметь длину 5:

```
TCard Well [5][5];
```

При этом пятые стопки колонн или стен реально использоваться не будут, и место для них в программе отведется напрасно, однако работать с массивом Well в описанной форме значительно проще, чем в какой-то более сложной и запутанной, позволяющей сэкономить ненужное место, но резко усложняющей структуру приложения.

Б каждой стопке может быть некоторое число карт — очевидно, что не более 104 (PACK_TOTAL). Поэтому размер третьего измерения примем равным PACK_TOTAL:

```
TCard Well [5] [5][PACK_TOTAL];
```

В данном случае это уже явное излишество — ведь реально на игровом столе никогда не будет более 104 карт, а мы резервируем место для 5*5*104 карт, то есть в 25 раз больше. Однако простота создания программы с такой высокой информационной избыточностью с лихвой окупает все затраты на память.

Да и так ли они велики, эти издержки? Попробуем их оценить. Всего имеется 5*5*104 = 2600 элементов. В каждом элементе класса TCard две переменные типа int — по 4 байта, и одна переменная типа bool (1 байт). В итоге получается 2600*(2*4 + 1) = 23400 байтов, то есть примерно 23 Кбайт. Это капля в море для современных 64 Мбайт ОЗУ обычного бытового компьютера! То есть экономить в данном случае смысла нет, а выигрыш от удобства программирования превзойдет все ожидания.

Переменную Well надо разместить в описании класса TFrom 1 главной формы Form1, в файле MainUnit.h, в последнем public-разделе.

Помимо переменной Well понадобится еще, очевидно, колода — добавьте переменную Pack:

```
TPack Pack;
```

Чтобы в классе TForm1 были понятны и доступны описания классов TCard и TPack, в начало файла MainUnit.cpp перед включением заголовочного файла MainUnit.h надо добавить строку включения файла CardUnit.h:

```
#include "CardUnit.h"
#include "MainUnit.h"
```

Наверняка потребуется переменная, учитывающая, сколько карт в данный момент может выкладываться во вспомогательном ряду (правила 14-17) — от пяти *до* одной. Назовем такую переменную Cycle:

```
int Cycle;
```

Какие методы будут требоваться при раскладывании пасьянса? Сразу определить их трудно, но можно точно сказать, что возникнет необходимость в методе начала новой игры, добавления/снятия карты с конкретной стопки определенной группы, перемещения карты между стопками, проверки допустимости перемещения карты и проверки конца игры. Кроме того, раскладку игрового поля надо как-то показывать.

Перевод карты в индекс картинки

Но прежде всего надо научиться вычислять позицию карты в объекте IList (список изображений карт, от 0 до 51) по ее масти и значению. Для этого определим два похожих метода:

```
int GetPackImageIndex();
```

который вернет номер картинки в IList для верхней карты колоды, и

```
int GetImageIndex(int line, int pos);
```

который вернет номер картинки для карты, лежащей в стопке pos группы line (как уже говорилось, группы нумеруются так: 0 — стена, 1 — колонна, 2 — пристенок, 3 — склад, 4 — вспомогательный ряд).

Предварительно в файле MainUnit.h надо определить две константы:

```
const BACK_NUM = PACK_LEN;
const NONE_NUM = PACK_LEN+1;
```

Константа BACK_NUM, фактически равная 52, — это индекс рубашки карты, а константа NONE_NUM — индекс подложки.

В методе GetPackImageIndex() происходит перебор всех карт колоды, начиная с нуля, и, как только встречается карта, которая в колоде действительно есть (переменная Is соответствующего элемента равна true), возвращается индекс элемента в списке IList, вычисляемый по очевидной формуле:

```
масть*13 + значение
```

Именно поэтому так важно, чтобы все карты в списке IList были упорядочены по мастям трефы — бубны — червы — пики и по возрастающему значению карт, от туза (самой младшей карты) до короля.

Если же колода пуста, то возвращается индекс подложки NONE_NUM, по внешнему виду которой будет понятно, что карт в колоде больше нет.

```
int  TForm1::GetPackImageIndex()
{
int  i;
for( i = 0; i < PACK_TOTAL; i ++ )
  if( Pack.GetIndexCard(i).Is )
       return Pack.GetIndexCard(i).GetSuit()*VALUE_NUM +
              Pack.GetIndexCard(i).GetValue();
return NONE_NUM;
}
```

Реализация метода GetImageIndex() во многом аналогична этому методу. Только перебор происходит не по колоде, а по настольному раскладу, хранящемуся в трехмерном массиве Well. И ведется он не с нуля, а с конца каждой стопки Well[line][pos] к началу — так удобнее для будущего показа содержимого стопок. Обратите внимание на этот цикл:

```
for( i = PACK_TOTAL-1; i >= 0; i — )
```

Здесь счетчик i получает значение 103 (последний элемент стопки из 104 карт, считая с нуля), и понижается до 0 включительно с помощью операции «--».

```
int TForm1::GetImageIndex(int line, int pos)
{
int i;
for( i = PACK_TOTAL-1; i >= 0; i — )
  if( Well [line] [pos] [i].Is)
       return Well[line][pos][i].GetSuit()*VALUE_NUM +
```

```
                Well[line][pos][i].GetValue();
return NONE_NUM;
}
```

Добавление и удаление карт из стопок

Для удаления карты из стопки pos группы line создадим метод

```
TCard GetCardfint line, int pos, bool view);
```

Он может использоваться не только для изъятия верхней карты соответствующей стопки, но и для простого просмотра ее значения (параметр view должен в этом случае иметь значение true). В любом случае методом будет возвращаться соответствующее значение карты.

```
TCard TForm1::GetCard(int line, int pos, bool view)
{
int i;
for( i = PACK_TOTAL-1; i >= 0; i — )
  if( Well[line][pos][i].Is )
    {
    // карта найдена:
    if( !view ) Well[line][pos][i].Is = false;
    return Well[line][pos][i];
    }

ShowMessage("Неверная попытка взять карту " );
return Well[0][0][0];
}
```

По аналогии с методом GetImageIndex() происходит просмотр всех карт стопки, начиная с последнего (PACK_TOTAL—1) элемента, считающегося верхним. Подразумевается, что метод GetCard() будет вызываться, только если стопка не пуста, в противном случае надо сообщить об ошибке.

Следующим будет метод добавления:

```
void AddCardfint line, int pos, bool pack, TCard card);
```

Карта card добавится на верх стопки pos группы line. Параметр pack характеризует, берется ли добавляемая карта из параметра card (тогда pack равен false) или с верха колоды.

Соответствующая стопка просматривается, начиная с ее верха (последнего элемента) в поисках карты, у которой значение переменной Is равно false (эта карта фактически в стопке отсутствует), и на ее место помещается (записывается) новая карта. Признак Is при этом должен, очевидно, получить значение true — карта в стопке имеется.

```
void TForm1::AddCard(int line, int pos, bool pack,
  TCard card)
{
if( pack && !Pack.IsCard() ) return;
int i;
for( i = 0; i < PACK_TOTAL; i ++ )
  if( ! Well[line][pos][i].Is )
    {
    if( pack ) Well[line][pos][i] = Pack.GetCard();
    else       Well[line][pos][i] = card;
    Well[line][pos][i].Is = true;
    return;
    }
}
```

Перемещение карты

Метод перемещения карты — основной в логике работы пасьянса. Ведь игра и построена на перекладывании карт из одной стопки в другую.

Этот метод описывается, как и все предыдущие, в классе TForm1 заголовочного файла MainUnit.h:

```
void MoveCard(int line1, int pos1, int line2, int
  pos2);
```

а его реализация — в файле MainUnit.cpp:

```
void TForm1::MoveCard(int line1, int pos1, int line2,
  int pos2)
{
TCard card;
card = GetCard(line1, pos1, false);
AddCard(line2, pos2, false, card);
}
```

Работает данный метод очень просто. Из стопки pos1 группы line1 методом GetCard() берется карта (параметр view — только просмотр — равен false), запоминается во временной переменной card, и затем с помощью метода AddCard() добавляется к стопке pos2 группы line2 (параметр взятия карты из колоды равен false).

Новая игра

Теперь, имея такой мощный метод, как MoveCard(), можно сгенерировать начальную раскладку пасьянса. Для этого понадобится метод Новая игра:

```
void NewGame () ;
```

Прежде всего колоду надо перетасовать (Pack.Shuffle()), а переменной Cycle (число выкладываемых во вспомогательном ряду карт) присвоить начальное значение 5. Затем надо очистить игровой стол — всем переменным Is массива карт Well присвоить false (никаких карты в стопке нет). Для этого потребуется тройной цикл — по каждому измерению массива.

Далее необходимо сформировать 4 стопки по 11 карт. Сделать это поможет метод AddCard(). Первым его параметром будет 0 (добавлять в группу 0 — стены), вторым — счетчик текущей колоды (4 колоды, от 0 до 3), третьим — true (карта берется из колоды), четвертым — временная переменная card, которая была описана только для применения в качестве неиспользуемого параметра.

Надо указывать все параметры метода, даже если некоторые из них реально и не будут использоваться — ведь компилятор об этом не знает.

После этого с помощью метода MoveCardO надо переложить по одной верхней карте из группы 0 стопок 0-3 в стопки 0-3 группы 2 (пристенок).

```
void TForm1::NewGame()
{
int i, j, p;
TCard card;
card.Init(0,0);

Pack.Shuffle();

Cycle = 5;
```

```
// инициализировать колодец:
for( i = 0; i < 5; i ++ )
  for( j = 0; j < 5; j ++ )
    for( p = 0; p < PACK_TOTAL; p ++ )
      Well[i][j][p].Is = false;

// 4 стопки по 11 карт:
for{ i = 0; i < 4; i ++ )
  for{ j = 0; j < 11; j ++ )
    AddCard(0, i, true, card);

// снятие по одной карте со стен на пристенки:
MoveCard(0, 0, 2, 0);
MoveCard(0, 1, 2, 1);
MoveCard(0, 2, 2, 2);
MoveCard(0, 3, 2, 3);
}
```

Самый главный метод

Все усилия по созданию программы будут тщетными, если не определить метод отрисовки текущего расклада — ведь на экране тогда ничего не появится. Поэтому давайте сделаем метод ShowAll().

```
void ShowAll();
```

В этом методе будет активно использоваться объект IList, содержащий рисунки карт. У этого объекта есть удобный метод **Draw(),** с помощью которого в любой точке формы можно показывать один из хранимых в списке IList рисунков.

Важный параметр Draw() — так называемая *канва* (класс TCanvas). Каждый элемент управления и каждая форма имеет свою канву — это та область окна, на которой допускается рисовать картинки, линии, выводить текст и т. д. Можно рисовать на канве конкретной кнопки, строки состояния, списка или целого окна. Для каждого элемента управления C++Builder есть собственные ограничения на допустимую область канвы — для окна это, например, область заголовка.

Пасьянс будет раскладываться на всем доступном пространстве главной формы — то есть рисовать мы будем только на ее канве. Переменная типа TCanvas любого окна (или элемента управления) в C++Builder всегда доступна внутри методов этого окна и называется Canvas.

Следующие два параметра метода Draw() — координаты (ширина и высота, *X* и У, в экранных точках или пикселах, считая от верхнего левого угла формы под заголовком или, если оно есть, под меню), определяющие, где будет отображаться рисунок. Он выводится в заданной точке, позиционируясь по своему верхнему левому углу.

Четвертый параметр — это индекс самого изображения из списка IList. Он будет получен с помощью метода GetImageIndex(). Последний параметр — true. Он означает, что картинка будет показана в нормальном, «включенном» режиме (если указать false, то картинка выведется в серых цветах и будет похожа на отключенный, недоступный элемент).

Для вывода шестнадцати стопок, а также колоды и склада их координаты на форме лучше всего описать в виде констант — если в дальнейшем потребуется эти координаты подкорректировать, то можно будет изменить только одно значение, вместо того чтобы исправлять десятки чисел в вызовах методов.

Константы надо разместить в начале файла MainUnit.cpp, сразу за строками включения заголовочных файлов. Этим константам лучше подобрать подходящие названия, допустим, такие: для координаты *X* — LEFT, для координаты *Y* — TOP. К этим именам будут приписываться две цифры: номер группы и номер колоды (нумерация будет начинаться с 1). Например, *X*-координата второй стопки колонн (королей) запишется как LEFT32, а У-координата третьей стопки стен — как TOP13.

```
// ширина и высота карты в пикселах
const CARD_WIDTH = 71;
const CARD_HEIGHT = 96;

// стены
const LEFT11 = 314;
const TOP11 = 232;
const LEFT12 = 402;
const TOP12 = 288;
```

```
const LEFT13 = LEFT11;
const TOP13 = 334;
const LEFT14 = 226;
const TOP14 = TOP12;

// подложки
const LEFT21 = LEFT11;
const TOP21 = 128;
const LEFT22 = 490;
const TOP22 = TOP12;
const LEFT23 = LEFT11;
const TOP23 = 438;
const LEFT24 = 138;
const T0P24 = TOP12;

// короли
const LEFT31 = LEFT14;
const TOP31 = 184;
const LEFT32 = LEFT12;
const TOP32 = TOP31;
const LEFT33 = LEFT32;
const TOP33 = 400;
const LEFT34 = LEFT31;
const TOP34 = TOP33;

// колода
const MAIN_LEFT = 24;
const MAIN_TOP = 10;

// склад
const INV_LEFT = MAIN_LEFT;
const INV_TOP = MAIN_TOP + CARD_HEIGHT + 20;
```

```
// вспомогательный ряд:
const SHIFT5 = CARD_WIDTH + 10;
const LEFT41 = MAIN_LEFT + SHIFT5;
const TOP41 = MAIN_TOP;
```

Некоторым константам присваиваются ранее определенные значения (например, LEFT23 = **LEFT11**), потому что многие стопки расположены на одинаковых вертикальных и горизонтальных уровнях.

Смещения на 20 и 10 пикселов нужны для того, чтобы отделять друг от друга смежные стопки.

Необходимо еще раз отметить, что приведенные координаты рассчитаны на работу в режиме 800x600 точек и выше.

Тогда вывод картинки группы 0 стопки 0 запишется таким оператором:

```
IList->Draw(Canvas, LEFT11,TOP11, GetImageIndex(0,0),
  true);
```

Весь метод будет выглядеть так:

```
void TForm1::ShowAll()
{
IList->Draw(Canvas, LEFT11,TOP11, GetImageIndex(0,0),
  true);
IList->Draw(Canvas, LEFT12,TOP12, GetImageIndex(0,1),
  true);
IList->Draw(Canvas, LEFT13,TOP13, GetImageIndex(0,2),
  true);
IList->Draw(Canvas, LEFT14,TOP14, GetImageIndex(0,3),
  true);

IList->Draw(Canvas, LEFT21,TOP21, GetImageIndex(2,0),
  true);
IList->Draw(Canvas, LEFT22,TOP22, GetImageIndex(2,1),
  true);
IList->Draw(Canvas, LEFT23,TOP23, GetImageIndex(2,2),
  true);
IList->Draw(Canvas, LEFT24,TOP24, GetImageIndex(2,3),
  true);
```

```
IList->Draw(Canvas, LEFT31,TOP31, GetImageIndex(1,0),
  true);
IList->Draw(Canvas, LEFT32 ,TOP32, GetImageIndex(1,1) ,
  true);
IList->Draw(Canvas, LEFT33,TOP33, GetImageIndex(1,2),
  true);
IList->Draw(Canvas, LEFT34 ,TOP34, GetImageIndexd,3) ,
  true);

IList->Draw(Canvas, LEFT41,TOP41, GetImageIndex(4,0),
  true);
IList->Draw(Canvas, LEFT41+SHIFT5,TOP41,
  GetImageIndex(4,1), true);
IList->Draw(Canvas, LEFT41+SHIFT5*2,TOP41,
  GetImageIndex(4,2), true);
IList->Draw(Canvas, LEFT41+SHIFT5*3,TOP41,
  GetImageIndex(4,3), true);
IList->Draw(Canvas, LEFT41+SHIFT5*4,TOP41,
  GetImageIndex(4,4), true);

// колода
if( Pack.IsCard() )
    IList->Draw(Canvas, MAIN_LEFT,MAIN_TOP, BACK_NUM,
  true);
else IList->Draw(Canvas, MAIN_LEFT,MAIN_TOP,
  NONE_NUM, true);

// склад
if( IsCards(3,0) )
     IList->Draw(Canvas, INV_LEFT,INV_TOP,
  GetImageIndex(3,0) , true);
else IList->Draw(Canvas, INV_LEFT,INV_TOP, NONE_NUM,
  true);
}
```

Два последних условных операторах служат для отображения:

- рубашки карты или подложки в зависимости от того, есть ли карты в колоде;
- подложки или верхней карты на складе в зависимости от того, есть в этой стопке карты.

Однако метод ShowAll(), очевидно, надо откуда-то вызвать и вызвать в подходящий момент — не тратить процессорное время на непрерывное перерисовывание всего изображения окна, а делать это только тогда, когда это действительно нужно: если окно открывается, загораживается другими окнами, а потом опять становится видимым, перетаскивается за границы экрана и обратно и т. д. Причем желательно все эти моменты не отслеживать вручную, а взять что-нибудь готовенькое. Возможно ли это? Конечно!

Самый главный метод-2

Для «умной» перерисовки окна именно тогда, когда это реально надо сделать, в C++Builder имеется специальное событие OnPaint(). Оно действует для большинства элементов управления и, конечно, для окон. Выберите в Инспекторе объектов главную форму **Form 1**, перейдите на закладку Events и дважды щелкните на правой части строки OnPaint.

C++Builder автоматически сгенерирует метод FormPaintO, который так же автоматически будет вызываться, именно тогда, когда окно требуется перерисовать. Все, что вам осталось сделать, — это вызвать в FormPaintO метод ShowAll().

```
void__fastcall TForm1::FormPaint(TObject *Sender)
{
ShowAll();
}
```

Теперь о правильном отображении содержимого окна можно не беспокоиться.

На событие OnPaint() требуется реагировать, только если в программе имеются операторы вывода различной графической информации на канву. Внешний вид элементов управления, спроектированный с помощью Инспектора объектов, будет поддерживаться и перерисовываться программой автоматически — то есть не надо пытаться самому выводить название на кнопке, для этого есть свойство Caption.

А есть ли карты?

При реализации метода ShowAll() использовался пока не упоминавшийся метод IsCarcls(). Он проверяет, есть ли карты в указанной стопке соответствующей группы:

```
bool IsCards(int line, int pos);
```

Этот метод возвращает значение true, если в стопке pos группы line карты есть. Выполняется такая проверка следующим образом:

```
bool TForm1::IsCards(int line, int pos)
{
int i;
for( i = 0; i < PACK_TOTAL; i ++ )
  iff Well[line][pos][i].Is )
      return true;

return false;
}
```

Логика работы интерфейса

Большинство методов пасьянса определены. Программа даже готова нарисовать начальный игровой расклад. Но как она будет взаимодействовать с пользователем?

Очень просто — как и положено Windows-приложению, с помощью мыши. Человек в первый раз щелкает на карте, которую хочет переместить, и второй раз — на стопке, куда он ее хочет переместить. Программа проверяет, допустимо ли такое перемещение, и если допустимо, то происходит вызов метода MoveCard() и новый расклад перерисовывается.

Ловим мышку

А как узнать, в каком месте главной формы был выполнен щелчок? В этом поможет событие OnMouseUp. Надо дважды щелкнуть на этом событии (как ранее — на событии OnPaint), и C++Builder сгенерирует реакцию программы на щелчок мыши:

```
void__fastcall TForm1::FormMouseUp(TObject *Sender,
  TMouseButton Button,
      TShiftState Shift, int X, int Y)
```

У этого стандартного метода формы, вызываемого при щелчке на форме (а еще точнее, на канве, но не на каком-нибудь из элементов управления!), несколько параметров. Самые важные из них следующие.

TMouseButton Button

Тип **TMouseButton** состоит из нескольких констант. Константа mbLeft означает, что щелчок был сделан левой кнопкой мыши, mbRight — щелчок правой кнопкой мыши, mbMiddle — щелчок средней кнопкой мыши. Соответствующее значение хранится в переменной Button.

intX,intY

Это, как вы, наверное, догадались, координаты точки, в которой был сделан щелчок мыши. Данные координаты отсчитываются от верхнего левого угла канвы.

Метод FormMouseUp() вызывается только при щелчке мыши на канве формы. Щелчки на заголовке, меню, области командных кнопок, границах окна этим методом не обрабатываются.

Весьма полезный параметр —

TShiftState Shift

Он характеризует, была ли при щелчке мыши нажата и удерживаема одна из системных клавиш CTRL, ALT или SHIFT. Это можно определить, сравнив переменную Shift с одним из значений ssAlt, ssShift или ssCtrl.

Дополнительное значение ssDouble определяет, был ли выполнен одинарный или двойной щелчок мышкой.

При этом возможно отслеживать нажатие сразу нескольких клавиш — для этого класс TShiftState имеет метод Contains(), который проверяет, содержит ли соответствующая переменная, способная хранить несколько значений (так устроен класс TShiftState), нужную константу. Например, чтобы выяснить, был ли сделан двойной щелчок мышкой и были ли при этом нажаты клавиши CTRL и SHIFT, надо записать следующее логическое выражение:

```
Shift.Contains(ssDouble) &&  Shift.Contains(ssCtrl) &&
  Shift.Contains(ssShift)
```

Более корректно говорить, что метод FormMouseUpO вызывается не просто при щелчке мышкой, а тогда, когда *отпущена* одна из ее клавиш. Ведь щелчок состоит из двух действий — нажатия клавиши мышки (для его обработки в C++Builder есть специальное событие OnMouseDown) и ее отпускания (уже известное событие OnMouseUp). Всегда лучше обрабатывать отпуска-

ние мышки, потому что, пока она нажата, Windows в силу своего внутреннего устройства будет непрерывно — множество раз — вызвать метод FormMouseDown(). А метод FormMouseUp() всегда вызывается только один раз и сигнализирует о том, что щелчок действительно окончен.

Метод FormMouseDown() подходит для обработки специфических действий пользователя по выделению, например, областей экрана или группы элементов с помощью нажатой мыши, но не для обработки одинарного или двойного щелчка.

Обрабатываем щелчок

Так как перемещение карты будет происходить в два этапа — сначала определяется исходная стопка, а потом конечная, то в обработчике мышиного щелчка надо проверять, какая стопка выбирается — исходная или конечная. Для этого в классе TForm1 создадим две переменные:

```
int Line, Pos;
```

Line определит группу стопок, Pos — конкретную стопку. Если в Line хранится значение -1, то будем считать, что никакой стопки пока не выбрано.

Это начальное значение надо задать переменной Line в самом начале работы программы в уже сгенерированном автоматически при создании нового проекта конструкторе формы Form1:

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
Line = -1;
}
```

Сразу после того, как был сделан щелчок мышкой, надо проверить, какая кнопка при этом использовалась. Если человек щелкнул правой кнопкой, то в Line запишем (-1) (как бы произошел сброс выбранной стопки и перемещение карты надо начинать заново — такой отказ от текущего выбора бывает полезен во многих приложениях) и прервем выполнение метода оператором return — в ожидании следующего нажатия мышки.

Далее надо выяснить, на какой конкретно стопке какой группы был сделан щелчок. В этом поможет условный оператор. Например, чтобы определить, выбрана ли стопка 0 группы 0, надо выяснить, попали ли координаты точки, где находился курсор мыши в момент щелчка, в прямоугольник, левой верхней координатой которого является точка (LEFT11, TOP11), а правой — точка

с теми же координатами плюс размеры карты (LEFT11+CARD_WIDTH, TOP11+CARD_HEIGHT):

```
if ( X > LEFT11 && X < LEFT11+CARD_WIDTH &&
     Y > TOPИ &&   Y < TOP11+CARD_HEIGHT )
     { line = 0; pos = 0; }
```

Найденные группу и стопку сохраним во временных переменных line и top.

Таких условных операторов потребуется 18. Их можно записывать поочередно — вслед за первым условным оператором выполнить второй:

```
if( X > LEFT12 && X < LEFT12+CARD_WIDTH &&
    Y > TOP12 && Y < TOP12+CARD_HEIGHT )
      { line = 0; pos =1 ; }
```

и т. д., но подобный подход не совсем эффективен, ведь если бы потребовалось выполнить сто проверок, то тогда код соответствующего метода неимоверно разросся.

Инициализация массивов

Правильно сделать так — выделить все координаты стопок и соответствующие им номера стопок и групп в отдельный массив и выполнить один оператор цикла, чтобы сразу определить нужные значения.

Подобный массив должен иметь два измерения, по первому — 18 элементов (18 стопок), по второму — 4 элемента (две координаты верхнего левого угла стопки и номер группы и стопки):

```
int  Coords [18] [4] ;
```

А как задать этому массиву начальные значения — ведь они все определяются до начала работы программы и представляют собой константы? К счастью, Си++ разрешает заранее, во время описания массивов, инициализировать их начальными значениями с помощью оператора присваивания. Эти значения берутся в фигурные скобки, а если измерений массива несколько, то они группируются тоже с помощью фигурных скобок, начиная с последнего измерения (в нашем случае — это четверки чисел), и разделяются запятыми:

```
int  Coords [18] [4]" =
{
{LEFT11,TOP11,0,0},  // первая четверка
{LEFT12,TOP12,0,1},{LEFT13,TOP13,0,2},{LEFT14,TOP14,
  0,3},
```

```
{LEFT21,TOP21,2,0},{LEFT22,TOP22,2,1},{LEFT23,TOP23,
  2,2},{LEFT24,TOP24,2,3},
{LEFT31,TOP31,1,0},{LEFT32,TOP32,1,1},{LEFT33,TOP33,
  1,2},{LEFT34,TOP34,1,3},

// вспомогательный ряд
{LEFT41,TOP41,4,0},{LEFT41+SHIFT5,TOP41,4,1},
  {LEFT41+SHIFT5*2,TOP41,4,2},
{LEFT41+SHIFT5*3,TOP41,4,3},{LEFT41+SHIFT5*4,
  TOP41,4,4},

{INV_LEFT,INV_TOP,3,0} // склад
};
```

Карты во вспомогательном ряду имеют одинаковую верхнюю координату TOP41, а по горизонтали сдвинуты друг относительно друга на фиксированное значение SHIFT5, равное ширине карты + 10 точек.

Однако оставлять Coords обычной переменной нехорошо. Желательно сделать ее константой:

```
const int Coords[18][4] = // ...
```

Разместить объявление Coords надо в файле MainUnit.cpp сразу за описаниями остальных констант.

Продолжаем проверку пользовательского выбора

Если координаты щелчка в массив Coords не попали, то выполнять метод их обработки смысла не имеет — просто щелчок был сделан где-то на незанятой части игрового поля. Тогда метод надо прервать с помощью оператора return.

Если же какая-то стопка выбрана в первый раз (то есть Line равно 1), то надо проверить, есть ли в этой стопке карты и не относится ли эта стопка к группе колонн (королей), из которой карты перемещать нельзя. Если это так, то надо запомнить группу и стопку в переменных Line и Pos и покинуть метод в ожидании дальнейших действий человека.

Наконец, когда в переменных Line и Pos уже хранятся координаты начальной стопки и они не совпадают с координатами конечной стопки (это может быть, если пользователь дважды щелкнет на одной и той же стопке), надо проверить, допустимо ли перемещение карты из стопки Pos группы Line в стопку pos группы line, и выполнить его.

Осталось еще отследить щелчок на колоде карт, лежащей рубашкой вверх, чтобы выполнить раскладку карт во вспомогательном ряду. Такая проверка выполнится с помощью условного оператора

```
if ( X > MAIN_LEFT && X < MAIN_LEFT+CARD_WIDTH &&
     Y > MAIN_TOP && Y < MAIN_TOP+CARD_HEIGHT )
```

Если колода пуста, то в нее надо собрать карты из вспомогательной колоды и уменьшить значение счетчика числа раскладываемых карт Cycle. Когда он станет равным нулю, других допустимых перекладываний в пасьянсе нет, о чем человеку надо сообщить — пасьянс не сошелся. В противном случае необходимо выполнить раскладку такого числа карт, какое значение хранится в Cycle. В заключение новую раскладку надо перерисовать.

```
void __fastcall TForm1::FormMouseUp(TObject *Sender,
   TMouseButton Button,
   TShiftState Shift, int X, int Y)
{
int i, line, pos;
TCard card;
card.Init(0,0);

// правая кнопка мыши — сброс
if( Button == mbRight )
   {
   Line = -1;
   return;
   }

// щелчок на главной колоде -
// разложить карты во вспомогательном ряду (4-я
// группа)
if( X > MAIN_LEFT && X < MAIN_LEFT+CARD_WIDTH &&
    Y > MAIN_TOP && Y < MAIN_TOP+CARD_HEIGHT )
     {
     int j, k;
```

```
if( !Pack.IsCard() ) // карты в колоде
    // закончились:
  {
// собрать карты со вспомогательного ряда:
for{ j = 4; j >= 0; j - )
    for{ k = 0; k < PACK_TOTAL; k ++ )
      if( Well[4][j][k].Is )
        {
       // убрать карту со стола:
        Well[4][j][k].Is = false;

        // вернуть карту обратно в колоду:
        Pack.AddCard(Well[4][j][k]);
        }
  Cycle -;
  }

iff Cycle <= 0 )
  {
  // пасьянс не удался
  ShowMessage(PACK_USED_STR);
  return;
  }

// раскладываем карты из колоды в "Cycle" стопок
    // во вспомогательном ряду (4-я группа)
for{ j = 0; j < Cycle; j ++ )
   AddCard(4, j , true, card);
ShowAll();
return;
}
```

```
// определение стопки, на которой был сделан щелчок:
line = -1;
for( i = 0; i < 18; i ++ )
  if{ X > Coords[i,0] &&  X < Coords[i,0]+CARD_WIDTH
  &&
       Y > Coords[i,1] && Y < Coords[i,1]+CARD_HEIGHT
  )
     {
    line = Coords[i,2];
    pos = Coords[i,3];
         // стопка найдена - прерываем цикл:
        break;
       }
// если щелчок сделан непонятно где, выходим из
   метода:
if( line == -1 ) return;

// выбрана верхняя карта с допустимой стопки
if( Line == -1 && IsCards(line,pos) && line i= 1 )
   {
   Line = line;
   Pos = pos;
   return;
   }

// проверка на допустимость и перемещение карты
if( Line != -1 && (Line != line || Pos != pos) &&
   CanMoveTo(line,pos) )
  {
  MoveCard{Line, Pos, line, pos);
  ShowAll();
  }
```

```
// готовимся к следующему действию —
// помечаем, что снова ничего не выбрано:
Line = -1;
}
```

В этом методе описана константа PACK_USED_STR — ее надо поместить в отдельный заголовочный файл текстовых констант Text.h, как это уже делалось раньше:

```
const AnsiString PACK_USED_STR = "Все попытки
 исчерпаны";
```

Этот файл надо включить командной строкой в начало файла MainUnit.cpp:

```
#include "Text.h"
#include "CardUnit.h"
#include "MainUnit.h"
```

Еще один пока не реализованный метод — это CanMoveToO, проверка на допустимость выбранного перемещения карты.

Если вам удалось осилить данный раздел, как следует разобраться в реализации метода FormMouseUp() и понять его работу — а ведь это фактически вся логика работы пользовательского интерфейса, то можно с уверенностью сказать, что вам по плечу любая задача прикладного программирования. Если же что-то осталось неясным, все-таки постарайтесь добиться понимания работы FormMouseUp() — вернитесь к предыдущим разделам или, может быть, даже к началу данной главы, и еще раз внимательно прочитайте описание всех аспектов функционирования создаваемого пасьянса.

Игра по правилам

Метод CanMoveToO остался последним довольно сложным методом проекта. Если в методе FormMouseUpO заключалась логика взаимодействия с человеком, то в методе CanMoveTo() надо реализовать правила перекладывания карт. Метод CanMoveToO опишется в класса TForm1 так:

```
bool CanMoveTo(int line, int pos);
```

В качестве параметров этот метод получает две переменные — line (номер группы) и pos (номер стопки), которые определяют координаты стопы назначения. Стопка, с верха которой карта будет браться, уже задана внутренними переменными Line и Pos класса TForm1.

Возвращаемое методом значение имеет тип **bool и** будет равно true, если карту в соответствии с правилами можно переместить на новое место.

О работе метода **CanMoveToO** удобнее всего рассказывать непосредственно в ого теле с помощью комментариев. Предварительно надо отметить, что метод состоит из трех условных операторов (с большими логическими блоками), каждый из которых проверяет, в какую группу будет перекладываться карта.

```
bool TForm1::CanMoveTo(int line, int pos)
{

// ---------------------------------------
// если карта кладется на королевскую колонну:
if; line == 1 )
   {
   // перемещение короля на пустую колонну:
   // если эта колонна пустая
   if( !IsCards(1,pos) &&

         // и на нее кладется король
    GetCard(Line,Pos,true).GetValue() == KING &&

         // то он берется не с другой колонны
     Line != 1 )
    {
    /* проверка на недопустимость совпадения масти
    нового короля с мастями других колонн: */
    int king_suit, i;
    // запоминаем масть исходного короля
    king_suit = GetCard(Line,Pos,true).GetSuit();

    /* проверяем, нет ли на пустых колоннах карт
       такой же масти: */
    for( i = 0; i < SUIT_NUM; i ++ )
```

```
            if( IsCards(1,i) &&
      GetCard(1,i,true).GetSuit() == king_suit )
              // если есть, то возвращается false
              return false;

          // карт такой же масти нет — можно перемещать:
          return true;
          }

        // перемещение масти на непустую колонну:
        // если на колонну берется карта не с другой колонны
         if( Line != 1 &&
            // и конечная колонна не пустая
             IsCards(1,pos) &&

             // и карты в ней такой же масти
             GetCard(Line,Pos,true).GetSuit() ==
      GetCard(1,pos,true).GetSuit() &&

            // и карта кладется на большую по значению
             OnBig(Line,Pos,1,pos,true) )

            // то перемещение допустимо
             return true;
      }

    // ---------------------------------
    // если карта кладется на пристенок:
    if( line == 2 )
      {
      // перемещение на пустой пристенок:

      // если карта снимется со стены
```

```
    if( Line == 0 &&

        // и эта стена — ближайшая
        pos == Pos &&

      // и на пристенке карт нет
        !IsCards(2,Pos) )

      // то перемещение допустимо
        return true;
  // обычное перемещение:
   // если карта берется со стены, пристенка
   // или из вспомогательного ряда:
    if( (Line == 0 || Line == 2 || Line == 4) &&

      // и кладется на непустой пристенок
        IsCards(2,pos) &&

      // на карту такой же масти
        GetCard(Line,Pos,true).GetSuit() ==
    GetCard(2,pos,true).GetSuit() &&

      // на 1 меньшую по значению
        OnBig(Line,Pos,2,pos,false) )

        // то перемещение допустимо
        return true;
    }

// ---------------------------------
// если карта кладется на склад:
iff iine == 3 )
```

```
    {
    // в пустой склад:
    // если склад пуст
    if( !IsCards(3,0) &&

      // и в него кладется туз
        GetCard(Line,Pos,true).GetValue() == ACE )

      // то перемещение допустимо
        return true;
   // обычное перемещение:
   // если карта берется со стены, пристенка
   // или из вспомогательного ряда:
    if( (Line == 0 || Line == 2 || Line == 4) &&

      // и кладется в непустой склад
        IsCards(3,pos) &&

        // и на карту такой же масти
        GetCard(Line,Pos,true).GetSuit() ==
   GetCard(3,pos,true).GetSuit() &&

        //и меньшую на 1 по значению
        OnBig(Line,Pos,3,pos,false) )

        // то перемещение допустимо
        return true,-
    }

// во всех остальных случаях перемещение недопустимо
return false,-
}
```

Нисходящее программирование

В методе CanMoveTo() не определен метод OnBig(), проверяющий, в строгом ли порядке возрастания или убывания значений карты кладутся друг на друга. Немногим ранее, когда писался метод реакции на щелчок мыши, также не был определен **CanMoveTo(),** он только упоминался, а реализован был позже.

Дело в том, что подобным способом разрабатывать программы гораздо проще. Сначала ее логика проектируется на самом высоком уровне абстракций, без привязки к конкретным конструкциям Си++, потом происходит детализация различных моментов, некоторые места, требующие особой реализации, выделяются в новые методы, которые программируются позже и т. д. То есть, создание приложения идет сверху вниз, а соответствующая технология называется *нисходящим программированием.* Она очень широко распространена среди профессиональных разработчиков и по праву пользуется заслуженным успехом, так как позволяет отвлечься от мелких деталей и быстро построить каркас программы.

Метод OnBigO описывается так:

```
bool OnBigfint 11, int p1, int 12, int p2, bool
  on_big);
```

Он имеет пять параметров, первые четыре из которых — координаты соответственно исходной и конечной группы и стоики для перемещаемой карты. Последний параметр on_big, должен иметь значение true, если исходная карта кладется на большую по значению.

Вот реализация этого метода.

```
bool TForm1::OnBig(int 11, int p1, int 12, int p2,
  bool on_big)
{
if( on_big &&
     GetCard(11,p1,true).GetValue() ==
    GetCard(12,p2,true).GetValue()-1 ||
     GetCard(11,p1,true).GetValue() == KING &&
     GetCard(12,p2,true).GetValue() == ACE )
     return true;

if( !on_big &&
```

```
        GetCard(l1,p1,true).GetValue() ==
        GetCard(l2,p2,true).GetValue()+1  ||
        GetCard(l1,p1,true).GetValue() == ACE &&
        GetCard(l2,p2,true).GetValue() == KING )
        return true;
    return false;
    }
```

В первом операторе проверяется, меньше ли значение исходной карты значения той карты, на которую она кладется. Дополнительно проверяется допустимость перемещения короля на следующую «большую» карту — туза.

Во втором операторе проверяется обратное соотношение, в частности, кладется ли туз на короля.

Проверка на конец игры

Такую проверку будет выполнять метод GameFin():

```
void GameFin() ;
```

Чтобы пасьянс был успешно закончен, надо, чтобы колода была пуста, а во всех группах, кроме 1-й (колонны) ни в одной стопке не было карт. Тогда можно показать сообщение GAME_WIN_STR: его надо описать в Text.h:

```
const AnsiString GAME_WIN_STR = "Пасьянс успешно
  сошелся";
```

и вызвать методы начала новой игры и отрисовки новой раскладки.

```
void TForm1::GameFin()
{
if( Pack.IsCard() ) return;
int i, j;
for( i = 0; i < 5; i ++ }
  if( i != 1 )
    for( j = 0; j < 5; j ++ )
      if( IsCards(i,j) ) return;
ShowMessage(GAME_WIN_STR);
NewGame();
ShowAll();
}
```

А где вызывать эту проверку на конец игры? Ее лучше всего вставить в следующие места: в начало логического блока, обрабатывающего щелчок мыши на главной колоде, и в начало блока, выполняющего перемещение карты (оба эти блока расположены в методе обработки щелчка мышью FormMouseUp()).

Последние мелочи

В конструкторе главного окна надо осуществить подготовку к новой игре, предварительно вызвав стандартную функцию randomize(), которая, как ранее говорилась, позволяет избежать повторения случайных последовательностей:

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
randomize();
NewGame();
Line = -1;
}
```

Чтобы функция randomize() могла вызываться, надо добавить заголовочные файлы:

```
#include <stdlib.h>
#include <time.h>
```

(первый из них уже был добавлен ранее).

Необходимо также определить методы реакции на выбор пунктов меню.

Реализация выбора пункта Выход будет состоять из одной уже знакомой вам строчки — завершения работы приложения:

```
Application->Terminate();
```

Реализация выбора пункта меню Новая игра тоже не потребует больших усилий — достаточно двух вызовов:

```
NewGame();
ShowAll();
```

Все! На этом создание пасьянса можно считать законченным. Откомпилируйте программу, если появятся предупреждения и ошибки, тщательно проверьте, правильно ли введен код — в корректно набранном примере их быть не должно.

Теперь пораскладывайте пасьянс — он наверняка вам понравится. А может быть, вы захотите что-то в нем улучшить или изменить.

Как уже говорилось, данная программа — это законченное приложение, близкое к коммерческому продукту. К нему еще нужно добавить справочное руководство по правилам пасьянса, реализовать незначительные нюансы типа пометки выбранной карты (например, небольшим красным кружочком), которые, несмотря на свою скромность, способны существенно повысить дружелюбность интерфейса. А это пользователи всегда очень ценят.

11. Вводивывод

Зачем это надо?

Создавать достаточно серьезные программы вы научились. Однако все коммерческие продукты рассчитаны не на однократные сеансы работы с пользователем, а, скорее, на долговременное сотрудничество. При таком сотрудничестве в промежутках между сеансами программа запоминает на диске самую разную информацию. C++Builder, например, помнит все настройки ваших проектов; текстовый редактор хранит популярные стили оформления, игра позволяет восстановить ранее отложенную партию и т. д. А реально эти программы сохраняют на жестком диске еще множество самых разных данных, о которых пользователь и не подозревает.

Все промежуточные настройки и прочая информация хранятся в файлах. Умение работать с файлами: открывать и закрывать их, записывать и считывать данные, копировать, переименовывать и удалять файлы — в обязательном порядке требуется при создании серьезных приложений. Обойтись без этого в современном программировании практически невозможно.

Как устроена работа с файлами в Си++

В самом языке C++ никаких средств для работы с файлами нет. Эти средства можно найти в стандартных библиотеках, причем их насчитывается довольно много, хотя все они предназначены только для одного — записи информации из переменных в файл и считывания информации из файла в переменные.

Разнообразие средств ввода/вывода связано с тем, что некоторые из них достались в наследство от Си, другие были созданы специально с ориентацией на возможности Си++, третьи связаны с особенностями системы Windows и т. д.

— Какие же средства лучше всего использовать в работе?

— Лучше всего, конечно, использовать те средства, которые проще в применении и в то же время обладают достаточно богатыми возможностями. Ведь для 99% задач какие-то уникальные особенности функций ввода/вывода не нужны, да и оставшийся 1% работ можно выполнить более простыми средствами, просто проявив немного выдумки.

В C++Builder есть очень хороший набор файловых функций, содержащихся в стандартном модуле SysUtils (который даже не требует добавления новых заголовочных файлов) и охватывающих все аспекты работы с файлами. Помимо простого ввода/вывода информации, эти функции позволяют удалять и переименовывать файлы, работать с каталогами, определять объем жесткого диска и размер свободного пространства на нем, а также выполнять еще множество прочих полезных вещей.

Работа с файлами организована по такому алгоритму:

1. Файл открывается для записи в него информации из переменных или для считывания из него информации в переменные программы.
2. Производятся соответствующие действия по вводу или выводу данных.
3. Файл закрывается.

Закрытие файла — обязательный этап. Если файл не закрыть, то хранимая в нем информация скорее всего пропадет.

Сохраняем текущий расклад

Создаем файл

Раскладывание некоторых вариантов в пасьянсе занимает нередко довольно длительное время, и если «плодотворный» рабочий день закончился, а бросить интересную партию жалко, то ее стоит сохранить на жестком диске, чтобы завтра с утра продолжить.

Рис. 46. Сохранение отложенной партии «до завтра»

Чтобы добавить в пасьянс новую возможность сохранения и восстановления партии, надо добавить в меню два новых пункта: Сохранить и Восстано-

вить. Соответствующие им переменные подпунктов меню можно для наглядности назвать SaveItem и LoadItem.

Добавьте в C++Builder реакцию на выбор пункта Сохранить. В реализации этого метода надо будет создать файл, в который будет сохранен текущий расклад, записать в него соответствующие данные и в заключение закрыть.

Файл создается с помощью стандартной функции

```
int FileCreate(AnsiString FileName);
```

В качестве единственного параметра указывается имя создаваемого файла вместе с полным путем доступа к нему. Если такой файл уже существует, то он предварительно *без предупреждения* уничтожается (точнее, стирается все его содержимое), а потом создается вновь, но уже пустой.

Эта функция возвращает значение -1, если создать файл по каким-то причинам не удалось (например, если указан неверный путь). Если же создание прошло успешно, то FileCreate() возвращает номер файла (типа int), представляющий собой некоторое внутреннее число Windows, характеризующее созданный файл. Это число (своеобразный идентификатор файла в программе) надо сохранить в переменной и в дальнейшем использовать ее в других функциях работы с файлами. Конечно, менять содержимое такой переменной ни в коем случае нельзя.

Допустим, что сохраняемый расклад будет записываться в пока несуществующий файл game.sol, который планируется расположить в каталоге c:\tmp. Название файла лучше оформить в виде текстовой константы и разместить в начале файла MainUnit.cpp:

```
const AnsiString FILE_NAME_STR = "c:\\tmp\\game.sol";
```

Обратите внимание на то, что внутри арок Си++ обратная наклонная черта «\» записывается обязательно двумя чертами — «\\».

Тогда создание такого файла потребует следующего кода на Си++:

```
int iFile;
iFile = FileCreate(FILE_NAME_STR);
```

В переменной iFile сохранен результат работы функции **FileCreate()**. Его значение теперь надо проверить:

```
if( iFile == -1 ) { ShowMessage(CANT_CREATE_STR +
  FILE_NAME_STR); return; }
```

Если создать файл не удалось, то пользователь увидит сообщение (его надо поместить в файл Text.h:

```
const AnsiString CANT_CREATE_STR = "Не могу открыть
  файл ";
```

В сообщении указывается имя файла, который не удалось создать, а работу метода придется прервать оператором return.

Перезаписывайте с осторожностью

Если файл уже существует, то не всегда правильно перезаписывать его содержимое, не предупредив об этом пользователя. Предварительно желательно проверить, не существует ли уже такой файл, и если существует — то запросить разрешение на его перезапись. Проверку на существование можно сделать с помощью стандартной функции

```
bool FileExists(AnsiString FileName);
```

Если файл, имя которого указано в качестве параметра, существует, то функция возвратит значение true.

Диалог с пользователем

А как спросить у человека, хочет ли он перезаписать существующий файл? В этом поможет метод MessageBox() глобальной переменной Application, описывающей все создаваемое приложение. Этот метод вызывает стандартное диалоговое окно Windows для несложного общения с человеком. В качестве первого параметра MessageBox() выступает строка с информационным сообщением, в качестве второго — строка-заголовок, а третий параметр может принимать одно из шести предопределенных значений:

MB_ABORTRETRYIGNORE	Стоп Повтор Пропустить
MB_OK	OK
MB_OKCANCEL	OK Отмена
MB_RETRYCANCEL	Повтор Отмена
MB_YESNO	Да Нет
MB_YESNOCANCEL	Да Нет Отмена

Каждому значению будет соответствовать указанный набор кнопок в диалоговом окне.

К кнопкам можно также добавить отображение в этом окне значка, символизирующего ситуацию. Для этого служит специальная операция Си++ «|» («побитовое сложение»). Таких стандартных значков может быть четыре:

MB_ICONWARNING	Предупреждение (знак «I»)
MB_ICONINFORMATION	Информация (буква «i»]

MB_ICONQUESTION	Вопрос (знак «?»)
MB_ICONSTOP	Остановка (стоп-знак)

Например, чтобы показать диалоговое окно с кнопками Да и Нет и значком Вопрос. в качестве третьего параметра надо использовать выражение

```
MB_YESNO | MB_ICONQUESTION
```

Функция MessageBoxO возвращает либо значение 0 (показать диалоговое окно не удалось), либо одно из семи значений, которые определяют, на какой из кнопок диалогового окна произошел щелчок.

IDOK	нажата OK
IDCANCEL	нажата «Отмена»
IDABORT	нажата «Стоп»
IDRETRY	нажата «Повтор»
IDIGNORE	нажата «Пропустить»
IDYES	нажата «Да»
IDNO	нажата «Нет»

К сожалению, нельзя в качестве первых двух параметров MessageBoxO использовать константы описанного в C++Builder типа AnsiString. Дело в том, что MessageBoxO — это стандартная функция Windows, а не C++Builder, и она требует параметров другого базового типа char.

Строка в Windows представляет собой последовательность символов, кончающихся символом с кодом 0. Каждый такой символ имеет тип char и занимает один байт. Чтобы передать в MessageBoxO значение типа AnsiString, приведенное к типу char (точнее, к указателю на последовательность символов в памяти — «char *»), надо воспользоваться методом класса AnsiString, который называется c_str().

Методы классов можно применять к любым данным соответствующих типов — не только к переменным, но и к константам.

Например, если строка «абвгд» описана как константа типа

```
const AnsiString STR = "абвгд";
```

то к STR можно примерять все методы класса AnsiString (за исключением присваивания — ведь значение константы менять нельзя):

```
STR.c_str()
```

и т. д.

В файле Text.h опишем новые текстовые константы:

```
const AnsiString HEADER_STR = "Внимание!";
const AnsiString FILE_EXIST_STR = "Перезаписать
  существующий файл ";
```

и после условного оператора проверки существования файла при необходимости вызовем функцию MessageBox():

```
if( FileExists(FILE_NAME_STR) )
  {
  AnsiString text;
  text = FILE_EXIST_STR + FILE_NAME_STR + " ?";
  if( Application->MessageBox(text.c_str(),
  HEADER_STR.c_str(),
          MB_YESNO | MB_ICONQUESTION) != IDYES )
      // Если не нажата кнопка «Да» — прекратить
  работу
    return;
  }
```

Так как результатом сложения констант FILE_EXIST_STR+FILE_NAME_STR будет строка типа AnsiString, то и к целому выражению тоже можно сразу применить метод c_str():

```
MessageBox((FILE_EXIST_STR + FILE_NAME_STR).c_str(), ..
```

Однако такой подход не очень нагляден и красив. Хотя экономится один оператор, вызов MessageBox() становится довольно громоздким, а проконтролировать его параметры в отладчике будет невозможно. Поэтому лучше избегать подобных «хитрых штучек» Си++ и писать код просто и наглядно.

Сохранение данных в файле

Запись информации в файл осуществляется с помощью стандартной функции FileWrite(). У нее три параметра. Первый — идентификатор ранее открытого файла (в программе-пасьянсе он сохранен в переменной iFile), второй — адрес сохраняемой в файле переменной, третий — размер этой переменной в байтах (сколько она занимает в памяти), чтобы знать, сколько байтов в файл записывать. Функция FileWrite() устроена именно так, чтобы в процессе работы программа сразу получала нужный адрес памяти и без дополнительных вычислений брала из него блок байтов соответствующей длины.

В файле надо сохранить пять переменных, ответственных за работу пасьянса: Cycle, Line, Pos, Pack и Well. Все методы класса хранятся непосредственно в коде программы в жестком, неизменном виде, а всю остальную специализированную информацию, имеющуюся в классе главной формы, сохранять не надо — вполне достаточно обойтись только пользовательскими данными.

Чтобы записать в файл содержимое переменной Cycle, надо указать ее адрес — таковы требования FileWrite(). Этот адрес можно получить с помощью операции Си++ «&», имеющий один операнд и располагаемой слева от него — в частности, адресом переменной Cycle будет выражение &Cycle.

В своих программах желательно избегать операции &, если того не требуют стандартные функции. Для работы с адресами надо применять технику указателей, которые, как уже неоднократно говорилось, нередко служат источником ошибок.

А как узнать размер переменной Cycle (сколько для нее отведено места в памяти) в байтах? Для этого служит операция Си++ под названием sizeof().

sizeof() — это не стандартная функция, а операция, такая же, как «=», «+» или «&». Просто sizeof() единственная в своем роде операция, которая записывается не специальным символом, а буквами. sizeof — зарезервированное слово Си++.

В качестве параметра операции sizeof() можно указывать как переменную, так и тип. Например, выражение sizeof(Cycle), которое будет иметь значение 4 (4 байта), равно значению sizeof(int) (размер переменной типа int), так как тип Cycle — int.

Тогда оператор записи содержимого переменной Cycle будет выглядеть так:

```
FileWrite(iFile, &Cycle, sizeof(Cycle));
```

Аналогично записываются и операторы сохранения других переменных, в том числе и пользовательского класса **Pack:**

```
FileWrite(iFile, &Pack, sizeof(Pack));
```

А вот с массивом Well будет немного посложнее. Дело в том, что массив Well — это не тип, а набор элементов типа **TCard.** Таких элементов в **Well,** очевидно, 5*5*PACK_TOTAL. Тогда размер всего массива в байтах составит

```
sizeof(TCard)*5*5*PACK_TOTAL
```

Адрес начала массива тоже определяется по другому. Можно выяснить адрес его самого первого элемента (с индексами (0, 0, 0)) — **&Well[0][0][0],** но это

немного громоздко. В Си++ принято, что адресом любого массива будет просто упоминание этого массива — Well. Тогда соответствующий оператор записи в файл будет выглядеть следующим образом:

```
FileWrite(iFile, Well, sizeof(TCard)*5*5*PACK_TOTAL);
```

В качестве второго параметра (адреса начала блока байтов) указан адрес массива — **Well**.

Чтобы получить адрес любого массива с произвольным числом измерений, достаточно использовать в программе его имя.

Контроль за ошибками

Каждую операцию вывода в файл, так же, как и его открытие, надо проверять — не возникла ли по каким-то причинам ошибка, например, если на диске нет свободного места. Функция FileWrite() в случае ошибки возвращает число (-1) — это и надо проконтролировать сразу за каждым вызовом соответствующей функции. Для этого в начале метода надо определить временную переменную:

```
int  err;
```

Контролировать корректность вывода будем так:

```
err = FileRead(iFile, Well,
  sizeof(TCard)*5*5*PACK_TOTAL);
if( err == -1 )
    {
    ShowMessage("Well" + BAD_OUTPUT_STR);
    FileClose(iFile);
    return;
    }
```

В файле **Text.h** надо описать текстовую константу:

```
const AnsiString BAD_OUTPUT_STR = " не выводится";
```

Закрываем файл и подводим итоги

В заключение процесса сохранения пасьянса файл надо закрыть. Делается это с помощью стандартной функции

```
void FileClose(int iFile) ;
```

При выборе пункта меню Сохранить должен выполняться следующий метод:

```
void__fastcall TForm1::SaveItemClick(TObject
  *Sender)
{
if( FileExists(FILE_NAME_STR) )
    {
    AnsiString text;
    text = FILE_EXIST_STR + FILE_NAME_STR + " ?";
    if( Application->MessageBox(text.c_str(),
        HEADER_STR.c_str(),
        MB_YESNO | MB_ICONQUESTION) != IDYES )
    return;
    }
int iFile, err;
iFile = FileCreate(FILE_NAME_STR);
iff iFile == -1 )
    { ShowMessage(CANT_CREATE_STR + FILE_NAME_STR);
    return;
    }
err = FileWrite(iFile, &Cycle, sizeof(Cycle));
iff err == -1 )
    { ShowMessage("Cycle"+BAD_OUTPUT_STR);
      FileClose(iFile); return; }
err = FileWrite(iFile, &Line, sizeof(Line));
if( err == -1 )
   { ShowMessage("Line"+BAD_OUTPUT_STR);
     FileClose(iFile);
   return;
   }
err = FileWrite(iFile, &Pos, sizeof(Pos));
if( err == -1 )
   { ShowMessage("Pos"+BAD_OUTPUT_STR);
     FileClose(iFile);
   return;
```

```
    }
  err = FileWrite(iFile, &Pack, sizeof(Pack));
  if( err == -1 )
    { ShowMessage("Pack"+BAD_OUTPUT_STR);
      FileClose(iFile);
    return;
    }
  err = FileWrite(iFile, Well,
   sizeof(TCard)*5*5*PACK_TOTAL);
  if( err == -1 )
    { ShowMessage("Well"+BAD_OUTPUT_STR);
      FileClosefiFile);
    return;
    }
  FileClose(iFile);
  }
```

Обратите внимание на то, что в любых ситуациях файл обязательно закрывается.

Считывание данных

Стандартная функция считывания данных практически ничем не отличается от своей коллеги, сохраняющей информацию в файле. Ее вид точно такой же, только название другое — FileRead().

Давайте подготовим метод реакции на выбор пункта меню Восстановить. В нем прежде всего надо выполнить открытие файла. В этом поможет стандартная функция

```
int FileOpen(AnsiString FileName, int Mode);
```

В качестве первого параметра указывается название открываемого файла (подразумевается, что он должен существовать, в случае неудачи данная функция возвращает -1), в качестве второго параметра задается режим открытия. Здесь лучше всего использовать константу fmOpenRead, которая разрешает открытие файла только для чтения и гарантирует, что его содержимое никакими действиями внутри программы изменено не будет.

Для считывания информации из файла в программу дополнительно понадобится текстовая константа (описываемая, как обычно, в файле Text.h):

```
const AnsiString BAD_INPUT_STR = " открыть нельзя";
```

Метод считывания сохраненного расклада похож на только что рассмотренный метод. Он выглядит так:

```
void__fastcall TForm1::LoadItemClick(TObject
  *Sender)
{
int iFile, err;
iFile = FileOpen(FILE_NAME_STR, fmOpenRead);
if; iFile == -1 )
    { ShowMessage(CANT_CREATE_STR + FILE_NAME_STR);
    return;
    }
err = FileRead(iFile, &Cycle, sizeof(Cycle));
if( err == -1 )
    { ShowMessage("Cycle"+BAD_INPUT_STR);
      FileClose(iFile);
    return;
    }
err = FileRead (iFile, &Line, sizeof(Line));
if( err == -1 )
   { ShowMessage("Line"+BAD_INPUT_STR);
     FileClose(iFile);
   return;
   }
err = FileRead(iFile, &Pos, sizeof(Pos));
if( err == -1 )
   { ShowMessage("Pos"+BAD_INPUT_STR);
      FileClose(iFile);
   return;
   }
err = FileRead(iFile, &Pack, sizeof(Pack));
if( err == -1 )
   { ShowMessage("Pack"+BAD_INPUT_STR);
```

```
      FileClose(iFile);
    return;
    }
  err = FileRead(iFile, Well,
      sizeof(TCard)*5*5*PACK_TOTAL);
  if( err == -1 )
    { ShowMessage("Well"+BAD_INPUT_STR);
      FileClose(iFile);
    return;
    }
  FileClose(iFile);
  ShowAll();
  }
```

Б конце метода вызывается метод ShowAII(), чтобы перерисовать весь расклад, что вполне естественно — ведь после считывания из файла он полностью изменился.

Обратите внимание на то, что данные из файла считываются строго в том порядке, в каком они записывались в файл. Если этот порядок будет нарушен, то в переменные программы из файла может быть считана неверная информация.

Переменных в пасьянсе пять. В крупных проектах переменных некоторых классов могут насчитываться сотни. При этом писать для каждой из них отдельный вызов функции ввода/вывода становится очень неудобно. Да и ошибиться легко — если вдруг в классе добавится новая переменная, ее надо будет специально добавлять в методы сохранения и восстановления, а при этом легко допустить ошибку, случайно изменив правильный порядок ввода/вывода переменных.

Эта проблема решается по-другому. Все пользовательские данные выделяются в программе в отдельный класс. Например, для пасьянса можно было описать специальный класс TSolData:

```
class  T6olData
{
int Cycle, Line, Pos;
TPack Pack;
```

```
TCard  Well[5][5][PACK_TOTAL];
};
```

а в классе TForm 1 ввести только одну переменную:

```
TSolData  SolData;
```

Тогда все обращения к внутренним переменным пришлось бы осуществлять через эту переменную, то есть писать не Pack, а SolData.Pack и т. д. Однако сохранение (и восстановление) всех данных потребовало бы только одного оператора:

```
FileWrite(iFile, &SolData, sizeof(TSolData));
```

При этом можно свободно добавлять новые переменные в класс TSolData, не утруждая себя лишней работой по написанию новых вызовов функций ввода/вывода и вычислению размеров массивов — новые данные будут сохраняться автоматически в составе класса TSolData.

Конечно, если информация предварительно была сохранена, а затем структура TSolData изменилась (переменные были добавлены или удалены), то считывать ее в переменную типа TSolData из-за несоответствия форматов хранения нельзя — возникнет ошибка ввода.

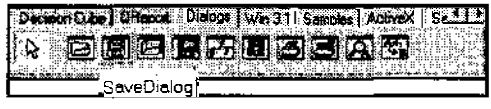
Выбор произвольных файлов

Сохранение и восстановление текущего расклада в файле реализовано. Но игра всегда будет записываться в один и тот же файл в одном и том же каталоге, что довольно неудобно — хотелось бы иметь возможность восстановления и сохранения пасьянса в произвольных файлах.

В этом помогут стандартные диалоговые окна Windows, предназначенные для выбора файлов. Эти окна применяются практически во всех известных приложениях — например, такой диалог вызывается в C++Builder командой File • Open (Файл • Открыть).

В C++Builder есть специальные компоненты — OpenDialog

и SaveDialog,



расположенные на панели Dialogs. Они отличаются только небольшими нюансами, связанными с особенностями действий по выбору файла для открытия или записи.

И тот и другой компонент надо разместить в главной форме (см. рис. 47).

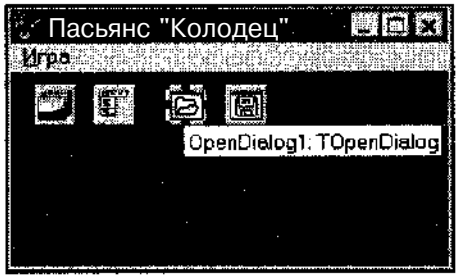


Рис. 47. Размещение элементов управления для сохранения и загрузки отложенной партии

Они, как и строка меню MainMenu и список картинок ImageList, не будут показываться в окне во время работы программы. Наличие их на форме просто говорит разработчику о том, что соответствующие объекты доступны внутри создаваемого приложения.

Каждый из двух помещенных на форму диалоговых объектов уже имеет автоматически сгенерированные названия OpenDialog1 и SaveDialog1 (они, как обычно, хранятся в свойстве Name). Каждому из объектов прежде всего надо дать заголовок, который указывается в свойстве Title. Для окна восстановления расклада это может быть «Загрузка пасьянса», для окна сохранения — «Сохранение пасьянса». Эти строки будут показываться в заголовках окон.

Файлы с раскладами будут иметь расширение .SOL (от английского *solitaire* — пасьянс). При этом было бы неплохо в списке доступных файлов сразу показывать только те файлы, которые имеют нужное расширение.

В этом поможет свойство Filter (Фильтр). В него записывается строка, которая содержит текст Сохраненные пасьянсы, выводимый в списке Тип файлов стандартных диалоговых окон, и маску соответствующих файлов *.sol, отделенную от него символом «|».

Кроме того, при открытии или сохранении пасьянсов было бы удобно указывать имя файла без расширения, чтобы это расширение подставлялось автоматически. Для этого предназначено свойство DefaultExt, в которое записывается данное расширение (без предварительной точки) — SOL.

Для открытия файла с игровым раскладом надо, чтобы этот файл существовал. Некоторые программы допускают «открытие» несуществующих файлов, при необходимости создавая их самостоятельно, но в нашем случае надо потребовать от человека, чтобы открываемый файл существовал. Такое требование задается установкой свойства ofFileMustExist (файл должен существовать), которое является подсвойством Options.

Установите в свойстве ofFileMustExist значение true.

Для открытия нужного стандартного диалогового окна применяется метод ExecuteO без параметров. Он возвращает значение true, если пользователь корректно выбрал имя файла и щелкнул на кнопке Открыть или Сохранить. В противном случае выдается значение false.

А как получить выбранное имя файла? Если метод ExecuteO вернул значение true, то нужное имя вместе с полным путем будет храниться в свойстве FileName.

Чтобы выполнить выбор файла для сохранения, надо предварительно удалить описание константы FILE_NAME_STR (жестко заданный путь больше не понадобится), а в начало методов сохранения и восстановления пасьянса добавить новую переменную FILE_NAME_STR (ее имя совпадет с именем константы — так сделано специально, чтобы не вносить много поправок в старый код).

```
AnsiString FILE_NAME_STR;
```

Для выбора файла с ранее сохраненным пасьянсом надо в начало метода LoadItemClick() внести следующий код:

```
// если файл не выбран, то покинуть метод:
if( !OpenDialog1->Execute() ) return,-
// сохранить выбранное имя файла в переменной
AnsiString FILE_NAME_STR;
FILE_NAME_STR = OpenDialog1->FileName;
```

Далее ничего менять не надо — FILE_NAME_STR будет теперь содержать не жестко заданное название, а то, которое выберет пользователь.

В начало метода считывания сохраненного пасьянса SaveItemClick() тоже надо добавить только три оператора:

```
// если файл не выбран, то покинуть метод:
if ( !SaveDialog1->Execute() ) return;
// сохранить выбранное имя файла в переменной:
AnsiString FILE_NAME_STR;
FILE_NAME_STR = SaveDialog1->FileName;
```

Теперь раскладывать пасьянсы станет значительно удобнее.

Другие возможности работы с файлами в C++Builder

Ниже приводится таблица, в которой кратко описаны возможности основных стандартных файловых функций в C++Builder. С их помощью можно решить большинство прикладных задач, связанных с обработкой сохраненных на жестком диске данных.

Примечания к таблице.

1. Часто используемый параметр FileName должен содержать полное имя файла вместе с полным путем доступа к нему. Если путь не указан, то файл ищется в текущей папке, которая исходно может быть в зависимости от настроек программы либо папкой, откуда она была запущена, либо одной из стандартных папок Windows.

   FileName применяется для работы с *закрытыми* файлами.

   Другой упоминаемый в таблице параметр функций — идентификатор *открытого* файла iFile.

2. Большинство функций возвращают значение -1, если их выполнение завершилось неудачно. Особые случаи оговариваются специально.

3. Для определения времени и даты создания («возраста») файла удобно использовать тип TDateTime, который содержит множество методов работы с датами и позволяет, в частности, получить текстовое представление даты (метод DateStringO) и времени (метод TimeString()), например:

```
TDateTime dt;
// Получить текущие дату и время:
dt.CurrentDateTime();
// В текстовую переменную Text
// записать строку, содержащую текущую дату:
Text = dt.DateString();
```

Переменная Text может получить, например, такое значение: «22/11/2000».

Стандартные функции для работы с файлами

int FileAge(AnsiString FileName);	Возвращает «возраст» файла FileName. Этот возраст можно преобразовать в тип TDateTime с помощью стандартной функции FileDateToDateTime():
	`int age;`
	`TDateTime dt;`
	`age = FileAge("c:\\command.com");`
	`dt = FileDateToDateTime(age);`

int FileGetDate(int iFile);
int FileSetDate(int iFile, int Age);
	Возвращает «возраст» файла iFile. Устанавливает новый «возраст» Age файла iFile. Age можно получить из переменной типа TDateTime с помощью стандартно функции DateTimeToFileDate():
	`int age;`
	`TDateTime dt;`
	`dt = FileDateToDateTime(age);`
	`age = DateTimeToFileDate(dt);`

AnsiString FileSearch(AnsiString Name, AnsiString DirList);	Поиск файла Name в списке папок DirList (папки в списке отделены точкой с запятой). Возвращает имя файла, если он в этом списке найден, и пустую строку "" в противном случае. `if ( FileSearch("referat.doc", "c:\\temp;c:\\doc;c:\\inet") != "" ) ..`
int FileSeek(int iFile, int Offset, int Origin);	Установка позиции в файле iFile, откуда будет происходить дальнейшее считывание или запись. Параметр Offset указывает на величину смещения в байтах, Origin — точку, от которой это смещение будет отсчитываться. Если Origin = 0, то отсчет будет вестись от начала файла; если Origin =1 — от текущей позиции; если Origin = 2 — от конца файла.
bool DeleteFile(AnsiString FileName);	Удаление файла FileName. При неудаче возвращает значение false.
bool RenameFile(AnsiString OldName, AnsiString NewName);	Переименование файла OldName в файл NewName, например: `RenameFile("start.doc", "start2.doc");` При неудаче возвращает false.

bool CreateDir(AnsiString Dir);	Создание новой папки Dir. При неудаче возвращает false. Все вышестоящие папки должны существовать, например, при вызове `CreateDir("c:\temp\web\koi");` Должна существовать папка c:\temp\web.
void ForceDirectories(AnsiString Dir);	Создание новой папки Dir. При этом можно создать не только конечную папку, но и все ей предшествующие, например, если на диске c: нет папки temp, то команда `ForceDirectories("c:\temp\1\2")` создаст три вложенные папки — \temp, \1 и \2.
boot DirectoryExists(AnsiString Name);	Проверка, существует ли папка Name.
AnsiString GetCurrentDir();	Возвращает путь доступа к текущей папке.
bool SetCurrentDir(AnsiString Dir);	Устанавливает папку Dir в качестве текущей: `SetCurrentDir("c:\\tmp");` При неудаче возвращает значение false.
bool RemoveDir(AnsiString Dir);	Уничтожает папку Dir. Эта папка обязательно должна быть пустой. При неудаче возвращает значение false.
__int64 DiskSize(Byte Drive);	Возвращает объем жесткого диска в байтах. Диск определяется по значению переменной Drive так: 0 — текущий диск 1 — диск A 2 — диск B 3 — диск C 4 — диск D и т. д.
__int64 DiskFree(Byte Drive);	Возвращает объем свободного места на диске Drive в байтах.

12. Компоненты, которые пригодятся

Сборка из кубиков

На трех предыдущих примерах можно было наглядно убедиться, насколько удобно использовать готовые компоненты C++Builder, как быстро можно с их помощью собрать работающее приложение, только настраивая нужные свойства в Инспекторе объектов или вызывая различные редакторы, встроенные в эти компоненты. Осталось только поближе познакомиться с набором компонентов, наиболее часто используемых при создании программ в системе C++Builder.

Панель Standard

Что уже известно

Ранее уже были рассмотрены в конкретных примерах такие базовые элементы управления, как строка меню, поле надписи и поле ввода, кнопка, обычные и раскрывающиеся списки.

Фреймы

Этот компонент появился только в C++Builder 5. Он напоминает форму, позволяя размещать на себе различные элементы управления, но конкретный экземпляр фрейма может использоваться сам по себе как *компонент* C++Builder. Это позволяет быстро создавать наборы пользовательских компонентов, отвечающих нуждам конкретной задачи, и повторно их использовать.

Фреймы также допускают неограниченное вложение друг в друга.

Контекстное меню

Практически в любом приложении Windows многие его возможности доступны по правой кнопке — через контекстное меню, которое открывается щелчком правой кнопки мыши. Такому меню в C++Builder соответствует компонент PopupMenu и класс TPopupMenu (класс всегда называется как компонент, только спереди добавляется буква T).

Это меню создается, как и главное меню формы, с помощью Редактора меню, а вызывается в любой точке программы методом Popup(), имеющим два параметра — координаты X и Y места, где это контекстное меню будет показано:

```
PopupMenu1->Popup(250,250);
```

Многострочное поле ввода

Некоторую информацию, например примечания в программе-записной книжке, удобно вводить в многострочные поля ввода. В этом поможет компонент Memo.

Для доступа к содержимому — массиву строк надо использовать свойство Lines. У этого свойства в свою очередь два полезных подсвойства: Count — число строк в поле и Strings — массив строк типа AnsiString, в котором таких строк столько, сколько указано в свойстве Count. Например, чтобы получить доступ к последней строке объекта Memo1, надо записать так:

```
s = Memo1->Lines->Strings[ Memo1->Lines->Count-1 ];
```

Единичка вычитается потому, что нумерация в массивах начинается с нуля.

Флажок

Флажки (CheckBox) — очень часто используемые элементы управления Windows.

Самое важное свойство флажка — Checked, которое имеет логический тип bool и принимает значение true, когда флажок установлен. Подпись у флажка содержится в свойстве Caption и может размещаться как справа от него (в свойстве Alignment указывается taRightJustify), так и слева (taLeftJustify).

На переключение объекта можно реагировать, обрабатывая событие OnCiick (щелчок левой кнопкой мыши).

Переключатель

Переключатели (RadioButton) не менее популярны, чем флажки. В одиночку их использовать, конечно, бессмысленно, так как они пред-

назначены для выбора одного из нескольких вариантов — на форме надо разместить несколько таких переключателей.

Выяснить, включен ли переключатель, можно с помощью свойства Checked. Подпись располагается справа или слева настройкой свойства Alignment.

Группа переключателей

Если планируется только один набор переключателей, то можно использовать компонент RadioButton. А если надо организовать несколько таких наборов (групп)? Для этого предназначен компонент RadioGroup.

Свойство Items (тип TStrings, список строк) содержит набор строк, каждая из которых соответствует подписи к одному из переключателей. Эти подписи вводятся в редакторе (см. рис. 48), вызываемом при нажатии на кнопку построителя в свойстве Items.

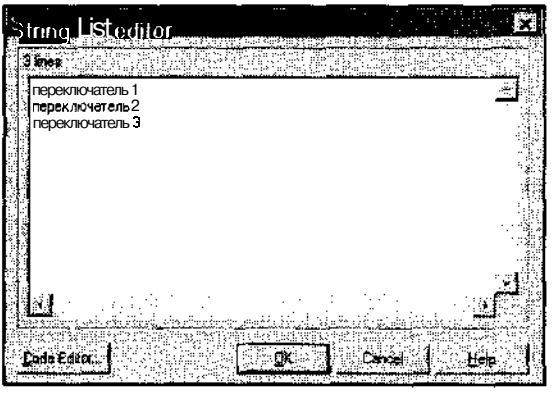


Рис. 48. Редактор списка переключателей в группе

Переключатели автоматически размещаются и выравниваются на поле формы в соответствии с числом колонок, указываемом в свойстве Columns.

Чтобы определить, какой переключатель включен в данный момент, надо воспользоваться свойством ItemIndex, которое укажет на номер переключателя (отсчитывается начиная с нуля). Номер каждого переключателя группы по порядку соответствует строке, введенной в поле Items. Если в свойстве ItemIndex хранится значение -1, значит, никакой переключатель не включен. С помощью свойства ItemIndex можно предварительно указать, какой переключатель должен быть включен по умолчанию, записав в поле ItemIndex номер соответствующего переключателя.

Полоса прокрутки

Полосу прокрутки (компонент ScrollBar) можно использовать не только для прокручивания содержимого окна, но и, например, в качестве приближенного регулятора значений.

Свойство Kind определяет, как полоса прокрутки будет расположена: горизонтально (sbHorizontal) или вертикально (sbVertical).

Этот объект характеризуется двумя главными свойствами: Min и Max, в которые записываются минимальное и максимальное числовые значения, принятые в качестве граничных — когда бегунок полосы расположен соответственно в крайнем левом (верхнем) положении или в правом (нижнем). Текущее положение бегунка определяется свойством Position, значение которого лежит, очевидно, в диапазоне от Min до Max. Когда пользователь нажимает клавиши PAGE UP или PAGE DOWN для прокрутки полосы, параметр Position изменяется на величину, значение которой указано в свойстве PageSize. Когда делается щелчок на самой полосе, чтобы прокрутить страницу, параметр Position изменяется на величину, указанную в свойстве LargeChange. При щелчке на кнопках со стрелками в конце полосы прокрутки параметр Position изменяется на величину, указанную в свойстве SmallChange.

В свойство Position можно записывать нужные значения в ходе работы программы — тогда бегунок полосы прокрутки сам установится в нужное положение.

Специальное событие OnScroll позволяет реагировать на прокрутку, дополнительно контролируя, что конкретно произошло — сдвиг на страницу, на один шаг, и где теперь находится бегунок.

Группа элементов

Группа элементов (GroupBox) позволяет объединить несколько элементов управления в одной группе с заголовком (свойство Caption). Никаких специфических задач этот компонент не выполняет.

Панель

Когда элементы управления хочется не просто сгруппировать, а объединить в некоторый наглядный блок, можно использовать панель (компонент Panel).

Она отличается от группы элементов тем, что позволяет гибко настраивать свой внешний вид. Панель состоит из двух каемок: внутренней (свойство BevelInner) и внешней (BevelOuter), которые могут принимать значения

«выпукло» (bvRaised), «вдавлено» (bvLowered) или «отсутствует» (bvNone). Толщина этих каемок задается в свойстве BevelWidth, расстояние между ними — в BorderWidth.

Панель Additional

Что уже известно

Из этой панели уже использовались кнопка с картинкой (BitBtn) и командная кнопка (SpeedButton).

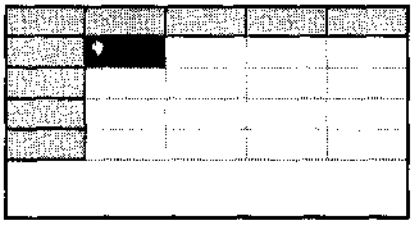
Поле ввода по маске

Компонент MaskEdit похож на обычное поле ввода, только он разрешает вводить в него не произвольную информацию, а соответствующую заданному шаблону (маске). Чтобы вызвать редактор маски, надо выделить объект на форме, нажать правую кнопку мышки и выбрать пункт меню Input Mask Editor.

Таблица строк

Компонент StringGrid — очень мощный. Он напоминает электронную таблицу, которую вполне можно сделать на его основе.

Этот компонент предназначен для отображения текстовой информации (и навигации по ней) в виде двумерной таблицы, размер которой определяется свойствами ColCount (число колонок) и RowCount (число строк). В этих значениях учитывается и граничное поле — каемка таблицы, на которой обычно пишется различная вспомогательная информация (название/номер колонки или ряда).



Для записи значений в ячейки таблицы или считывания этих значений используется свойство Cells — двумерный массив строк. Например, чтобы записать строку "123,45" в ячейку (2,3) таблицы StringGrid1, можно использовать такой оператор:

```
StringGrid1->Cells[2][33 = "123,45";
```

При этом в соответствующей ячейке в окне программы будет немедленно отображена строка 123,45.

Чтобы узнать, какая ячейка таблицы является текущей (выделена в данный момент), надо использовать свойства Col и Row (номера текущих колонки и строки).

Ширина и высота каждой ячейки задаются соответственно свойствами DefaultColWidth и DefaultColHeight.

Таблица чего угодно

В некоторых случаях от таблицы может потребоваться показывать в ячейках не только текстовую информацию, а, например, картинки, видеоклипы или еще что-то другое. Для этого предназначен компонент DrawGrid свойства которого совпадают со свойствами компонента StringGrid, только в нем нет массива строк Cells.

Вместо этого содержимое каждой ячейки требуется рисовать самостоятельно, основываясь на ее координатах. Такую перерисовку надо выполнять, определив метод DrawCell(), в котором указываются координаты ячейки, подлежащей перерисовке (более подробно этот метод описан в справочном руководстве по C++Builder). DrawCell() будет вызываться автоматически по мере необходимости.

Картинка

Для украшения программ, например, для размещения в окне логотипа компании, можно использовать компонент Image (Рисунок).

Основное его свойство — Picture, которое с помощью кнопки вызывает редактор картинок, где и выбирается подходящее изображение.

Геометрическая фигура

У некоторых пользователей возникает потребность в отображении на форме абстрактных геометрических фигур. Для этого в C++Builder имеется компонент Shape.

Он позволяет нарисовать в окне прямоугольник (stRectangle), квадрат (stSquare), прямоугольник со скругленными краями (stRoundRect), квадрат со скругленными краями (stRoundSquare), эллипс (stEllipse) или круг (stCircle). Соответствующее значение задается в свойстве Shape.

Фигура рисуется в окне с помощью виртуального карандаша (свойство Pen), которое имеет такие важные подсвойства, как Color (Цвет) и Width (Толщина

линии). Заполняется внутренность формы с помощью виртуальной кисти (свойство Brush), для которой надо определить подсвойство Color (Цвет заполнения).

Ширина и высота фигуры определяются свойствами Width и Height.

Рамка

Из двух экземпляров компонента Bevel (Рамка) состоит уже упоминавшийся выше компонент Panel. Рамка имеет стиль (свойство Style, принимающее значение bsRaised — «выпукло», или bsLowered — «вдавлено»). Также для нее можно задавать образ (свойство Shape), чтобы показать не всю рамку, а только одну из четырех сторон.

Прокручиваемая зона

C++Builder позволяет создавать отдельные части окна, содержимое которых (элементы управления) будет прокручиваться с помощью полос прокрутки. Для этого достаточно установить на форме компонент ScrollBox и потом размещать элементы управления внутри него. При этом возможность прокрутки проявится автоматически.

Список флажков

Расширение обычного списка строк, дополнительно сопровождающее каждый элемент флажком, который можно устанавливать в стандартные состояния «включено/выключено».

Заголовок

В целях дополнительного украшения программы различные подписи иногда желательно делать в красивых выпуклых или вдавленных рамочках. Для этого в C++Builder имеется компонент StaticText.

Помимо стандартного свойства Caption, в котором хранится показываемая текстовая строка, компонент StaticText обладает свойством BorderStyle, позволяющим немного изменять внешний вид каймы.

Панель элементов

В левой части главного окна C++Builder располагается несколько панелей с наборами командных кнопок. Эти панели можно перемещать в неких допустимых рамках, перетаскивая их за корешок, оформленный как двойная полоска.

Границы перемещения панелей кнопок можно определить с помощью компонента ControlBar.

Разместив его на форме, в дальнейшем внутри этого компонента можно располагать командные кнопки и панели таких кнопок. Поддержка их перетаскивания в рамках панели элементов — автоматическая.

Диаграмма

Компонент Chart очень интересен. Он содержит множество средств, даже поверхностное описание которых заняло бы слишком большой объем. Этот компонент предназначен для создания всевозможных трехмерных диаграмм.

Он позволяет настраивать самые разные детали внешнего вида этих диаграмм, но не содержит средств ручного ввода начальных значений — заполнение диаграммы надо производить с помощью операторов Си++.

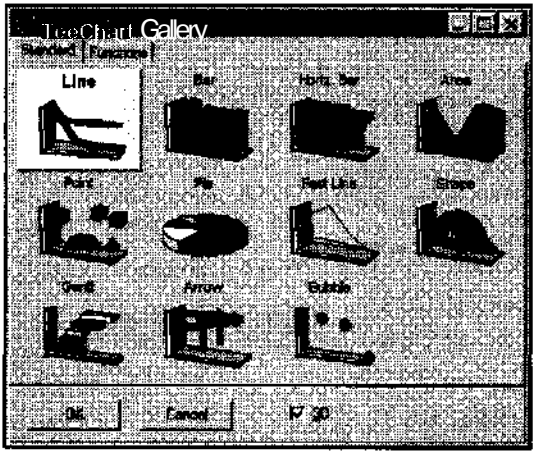


Рис. 49. Варианты оформления диаграмм

Панель Win32

Что уже известно

Уже известен такой компонент, как ImageList (Список рисунков), оказавшийся очень полезным при создании пасьянса, и компонент ToolBar — линейка инструментов, ранее использованная при разработке игры в кости.

Вкладки

Компонент PageControl (Вкладки) — весьма удобная и полезная вещь, часто используемая при разработке диалоговых окон.

В частности, на основе этого компонента создана вся палитра инструментов C++Builder — когда пользователь выбирает нужную вкладку, и на панели отображается оригинальный набор элементов управления. В подобном стиле сегодня создается множество приложений, и изучить данный компонент надо обязательно, тем более, что устроен он совсем просто.

После размещения компонента PageControl на поле формы надо задать нужное число вкладок. Страницы добавляются так.

1. Выбирается объект PageControl.
2. Выполняется щелчок правой кнопкой мыши.
3. Выбирается пункт контекстного меню New page — открывается новая страница вкладки.
4. Выполняется щелчок на поле новой страницы вкладки (не на корешке вкладки!), для ее выделения.
5. В свойстве Caption новой страницы указывается строка, которая будет служить для нее заголовком на корешке.

В дальнейшем между страницами объекта в режиме проектирования можно переключаться так же, как и во время работы программы — просто выбирая указателем мыши нужную закладку и размещая на соответствующей странице необходимые объекты.

Для удаления лишней страницы вкладки вместе со всем ее содержимым надо выделить эту страницу (но не весь объект со вкладками!) и нажать клавишу DEL.

На панели Win32 имеется еще один похожий компонент — TabControl. Он представляет собой просто набор вкладок и не обновляет содержимое страниц автоматически — программисту требуется делать это самостоятельно, поэтому компонент TabControl, используемый в специфических ситуациях, начинающим разработчикам лучше не применять.

Мощный редактор текста

Во ряде случаев возможностей простого многострочного редактора текста Memo недостаточно. Иногда желательно вручную форматировать различные части текста, изменять их цвет, способ выравнивания, шрифт и т. д. Для этого предназначен компонент RichEdit.

Его работа определяется с помощью различных свойств и методов непосредственно в программном коде. Компонент RichEdit позволяет, в частности,

сохранять свое содержимое в распространенном формате .RTF, поддерживаемом большинством текстовых процессоров.

Ползунок

Компонент TrackBar по своим возможностям и способу применения напоминает классическую полосу прокрутки — он имеет такие же свойства Min, Max и Position.

Только засечки на шкале определяются в свойстве Frequency (Частота засечек). Так, например, если Min = 0, Max = 10, а Frequency = 2, то засечек будет 6 (включая начальную и конечную). Свойство Orientation определяет, как ползунок будет расположен — горизонтально (trHorizontal) или вертикально (trVertical). Толщина полосы ползунка задается свойством ThumbLength, величина перемещения при использовании курсорных клавиш — свойством LineSize, а при использовании клавиш PAGEUP/PAGEDOWN — свойством PageSize.

Можно также указать «особый» диапазон, который задается свойствами SelStart и SelEnd. Например, значение SelStart может быть равно 3, а значение SelEnd — 8. Тогда в диапазоне от 3 до 8 на полосе ползунка появится зона, выделенная другим цветом, а ниже ее — две специальные засечки, дополнительно определяющие начало и конец этой зоны.

Индикатор выполнения операции

Компонент ProgressBar можно встретить, например, в программах установки, когда процент выполнения отображается в виде постепенно заполняемой шкалы.

Данный компонент характеризуется уже описанными свойствами Min, Max и Position, а также свойством Step, которое определяет, на сколько единиц будет увеличиваться полоса заполнения при выполнении метода Steplt() — основного метода, который требуется вызывать для удлинения полосы. Для увеличения значения Position надо использовать метод StepByO. В качестве параметра для него указывается величина приращения или уменьшения.

Кнопки счетчика

В некоторых случаях бывает необходимо организовать ввод числовой информации, располагающейся в жестком диапазоне, допустим, от 1 до 25. Для этого можно использовать обычное поле ввода и программно проверять, соответствует ли введенное число нужному диапазону, но это не совсем удобно и совсем неэлегантно. Гораздо проще применить для этих целей компонент TUpDown, напоминающий своим поведением ползунок.

Этот компонент состоит из двух стрелок, и свойство Position изменяется в большую или меньшую сторону при щелчке на одной из этих кнопок.

Для того, чтобы отображать текущее значение, установленное с помощью кнопки-счетчика, рядом на форме надо также расположить поле ввода, и в свойстве Associate (раскрывающийся список) выбрать имя этого поля. Тогда счетчик автоматически «приклеится» справа от поля ввода, и в этом поле будет выводиться текущее значение счетчика.

Помимо таких свойств, как Min, Max, Orientation и Position, кнопки счетчика характеризуются также свойством Increment (на сколько будет происходить изменение свойства Position при щелчке на кнопке), и свойством Thousands — признаком, надо ли разделять числовые классы (каждые три цифры числа) запятой, например, так: 1,000 или 9,876,543.

Находясь в поле ввода, менять значение числа можно, не только щелкая мышкой на стрелках кнопки, но и нажимая курсорные клавиши «вверх» и «вниз» (для этого в свойстве ArrowKeys должно быть задано значение true).

«Горячая» клавиша

«Горячие» клавиши используются во многих программах для выполнения основных операций. Так, в Windows, например, комбинация ALT+TAB служит для переключения между приложениями, в большинстве приложений комбинация CTRL+ S используется для сохранения текущего файла и т. д. Вводить такие комбинации удобнее всего с помощью компонента THotKey.

Когда объект «горячая клавиша» получает фокус ввода, то любая комбинация клавиш немедленно отражается в нем в виде строки: нажав комбинацию клавиш SHIFT+CTRL+A, пользователь увидит в поле этого элемента строку SHIFT+CTRL+A. Чтобы выяснить в самой программе, какая же комбинация клавиш введена, надо использовать свойство HotKey, которое содержит комбинацию различных констант, описанных в справочном руководстве C++Builder.

Анимация

Аналогично компоненту Image (Рисунок), компонент Animate (Анимация) позволяет добавлять на форму анимационные фрагменты (наподобие стандартных клипов, показываемых Windows при копировании файлов, поиске и т. д.), которые должны быть предварительно подготовлены и сохранены в отдельных файлах в формате .AVI.

Воспроизводимый файл указывается в свойстве FileName. Можно использовать также один из стандартных клипов:

- поиск папки (aviFindFolder);
- поиск файла (aviFindFile);
- поиск компьютера (aviFindComputer);
- копирование группы файлов (aviCopyFiles);
- копирование одного файла (aviCopyFile);
- выбрасывание файла в корзину (aviRecycle File);
- очистка корзины (aviEmptyRecycle);
- удаление фала (aviDeleteFile).

Соответствующие значения указываются в свойстве CommonAVI.

Перед тем, как начать в программе показ AVI-клипа, его надо подготовить — открыть, задав свойству Open значение true, а затем запустить анимацию, установив значение true в свойстве Active. Свойство Repetitions определяет число повторов полного клипа, свойство Transparent — будет ли использоваться при показе клипа его фоновый цвет, или же этот цвет будет трактоваться как «прозрачный». Диапазон кадров можно уточнить с помощью свойств StartFrame (Первый кадр) и StopFrame (Последний кадр). Нумерация кадров в клипе происходит с единицы.

Показать конкретный кадр можно, используя метод Seek(), который имеет единственный параметр — номер этого кадра. Остановить показ можно с помощью метода Stop(), продолжить — записав в Active значение true.

Поле ввода даты и времени

Вводить дату/время требуется во многих настольных приложениях, например, в планировщике работ. Для этого удобнее всего использовать компонент TDateTimePicker.

Он позволяет вводить вручную допустимые значения даты/времени. Что будет показываться — дата или время — определяется в свойстве Kindю Для даты надо установить значение dtkDate, для времени — dtkTime. Открыв раскрывающийся список, с помощью мыши можно выбрать нужный день в симпатичном графическом календарике.

Получить указанные пользователем дату или время можно, обращаясь к свойствам Date или Time (они имеют описанный ранее тип TDateTime). Дату можно вводить в двух форматах (свойство **DateFormat**) — в коротком dfShort (при этом на год отводится только две цифры, и могут возникнуть проблемы с поддержкой дат 2000 года и XXI века) или длинном dfLong, который лучше применять всегда. Разрешить ввод дат в жестком диапазоне можно, указав этот диапазон в свойствах **MinDate** (начальная дата) и MaxDate (конечная дата).

Месячный календарь

Компонент **MonthCalendar** представляет собой готовый месячный календарь. 

Он позволяет выбирать конкретную дату (она хранится в свойстве Date) или диапазон дат (для этого в свойстве MultiSelect надо задать значение true; конечная дата будет записана в свойстве EndDate).

Начинать отсчет дней допускается с любого дня недели, который задается в свойстве **FirstDayOfWeek**. Для каждой недели можно также показать ее номер в текущем году, записав в свойство WeekNumbers значение true.

Заголовок с разделами

Компонент HeaderControl позволяет размещать на форме заголовок с набором секций, каждая из которых представляет собой либо текстовую строку, либо фрагмент, содержимое которого надо вырисовывать вручную. 

Вызвать редактор разделов можно, выделив объект HeaderControl, нажав правую кнопку мыши и выбрав в контекстном меню пункт Sections Editor.

Строка состояния

Редкая программа обходится без строки состояния (компонент StatusBar). 

Ее содержимое может состоять из одной строки (что определяется установкой значения false в свойстве **SimplePanel**, а сама строка записывается в свойство **SimpleText**), или из нескольких отдельных панелей, которые формируются в редакторе панелей, вызываемом выбором в контекстном меню пункта Panels Editor.

Панель элементов

Компонент CoolBar во многом похож на компонент ToolBar. 

Он дополнительно позволяет для каждого размещенного на нем элемента управления создать свой небольшой блок, охватывающий этот элемент и позволяющий перетаскивать его или менять размеры. Такие блоки создаются с помощью специального редактора, вызываемого выбором пункта Bands Editor из контекстного меню.

Панель прокрутки

Если C++Builder запущен в разрешении 640x480 точек, то некоторые длинные панели компонентов могут не умещаться на экране, и их надо прокручивать с помощью специальных кнопок.

Автоматически поддерживать подобную прокрутку позволяет компонент PageScroller.

После размещения его на форме в свойстве Control (раскрывающийся список) надо указать, какой из имеющихся на форме элементов управления будет прокручиваться в окне данной панели. Если размер соответствующего элемента слишком велик, то PageScroller покажет с нужной стороны кнопку-стрелку. Данный компонент удобнее всего использовать вместе с длинными панелями инструментов или командных кнопок.

Панель System

Таймер

В некоторых приложениях сложно обойтись без контроля за временем. Например, если создается программа-будильник, то в ней просто необходимо использовать таймер. Однако устроена работа с таймером в Windows довольно хитро. Это связано с тем, что возможности этой операционной системы по поддержанию нескольких одновременно и активно работающих приложений довольно слабы. Если одна программа начинает длительные вычисления, не прерывая свою работу, то переключиться даже на текстовый редактор довольно затруднительно.

Поэтому корпорация Microsoft не рекомендует создавать программы, в которых ресурсы процессора захватываются надолго. Разработчикам предлагается постоянно передавать управление Windows, реагируя на генерируемые с заданной частотой сигналы от системного таймера.

Для поддержки пожеланий Microsoft в C++Builder реализован компонент Timer.

Этот компонент не отображается на экране и используется только внутри создаваемого приложения.

Чтобы запустить таймер, в его свойство Enabled надо записать значение true, чтобы остановить — значение false. Когда таймер запущен, он генерирует событие OnTimer, которое происходит через заданные интервалы в миллисекундах, указываемые в свойстве Interval. Реагируя на эти события в программе, можно подсчитывать, например, сколько таких вызовов произошло с определенного момента, чтобы узнать потраченное время.

Когда происходит такое *прерывание по таймеру,* Windows может временно отложить выполнение текущего приложения и переключиться на решение более важных задач — это вполне корректный прием при написании Windows-приложений, поэтому длинные вычисления лучше разбивать на небольшие фрагменты и постепенно выполнять их, обрабатывая событие OnTimer и не захватывая ресурсы компьютера монопольно.

Область рисования

В нашем пасьянсе вывод всей графической информации происходил непосредственно на канву главного окна. Более сложные графические приложения обычно имеют несколько областей, которые перерисовываются независимо друг от друга и с которыми удобнее работать как с отдельными объектами (каждый со своей системой координат и своей канвой). Чтобы определить на форме отдельную область рисования, надо использовать компонент PaintBox.

Никаких особенных свойств он не имеет. Выполнять его отрисовку надо в обработчике события OnPaint.

Универсальный проигрыватель

Показывать отдельные картинки и AVI-клипы мы уже научились, но в C++Builder имеется более мощный компонент MediaPlayer, который способен воспроизводить (и даже записывать!) не только клипы, но и цифровое видео, а также музыку в различных форматах и другую мультимедийную информацию, определяемую установленным оборудованием и доступными на компьютере соответствующими программами. Узнать о доступных в C++Builder форматах можно, заглянув в свойство DeviceType (Тип устройства) проигрывателя.

После размещения на форме компонента MediaPlayer надо также подготовить объект, в котором соответствующее изображение будет показываться (если это, конечно, видео, а не звук). Лучше всего в качестве такого объекта использовать компонент Panel. Объект, в котором будет воспроизводиться клип, привязывается к MediaPlayer через его свойство под названием Display.

Файл, содержащий анимацию или звук, указывается в свойстве FileName. Приступить к началу проигрывания можно, вызывав метод Play(), остановить — с помощью метода Stop(). Когда программа с объектом типа TMediaPlayer запустится, то на экране будет видна вся панель управляющих кнопок (Play, Stop, Next и т. д.).

Определить, какие из этих кнопок реально нужны, можно с помощью свойства VisibleButtons, которое содержит подсвойства, соответствующие каждой кнопке проигрывателя. Если установить значение подсвойства равным логическому значению false, то соответствующая ему кнопка на панели проигрывателя показана не будет. Например, можно оставить только одну кнопку Play.

Правда, если попробовать убрать все кнопки, то окажется, что для проигрывателя MediaPlayer на форме все равно отведен небольшой квадрат. Но во многих случаях использование возможностей проигрывателя может потребоваться только в ситуациях, определяемых программистом, например, когда надо исполнить короткую мелодию, и наличие кнопок проигрывателя в окне нежелательно. Совсем спрятать MediaPlayer от пользователя, сделать его недоступным, невидимым, можно с помощью свойства Visible (Видимость). Установив его значение равным false, можно добиться того, что проигрыватель в окне работающей программы не появится.

OLE-контейнер

Несмотря на несколько устрашающее название, компонент OleContainer во многих случаях может оказаться очень полезным.

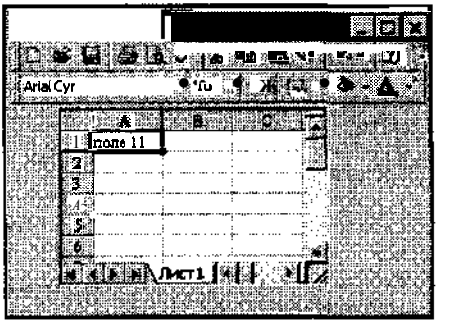
Технология OLE корпорации Microsoft была разработана и реализована в Windows довольно давно, еще в 16-разрядных версиях этой системы. OLE позволяет обращаться к другим приложениям Windows, как к готовым компонентам (это напоминает работу с компонентами в C++Builder), запускать их, управлять их работой и т. д. Для этого требуется, конечно, чтобы такое приложение поддерживало специальный OLE-протокол.

В Windows 9x большинство программ поддерживает технологию OLE. Это и Word, и Excel, и графический редактор Paint, и другие. Объекты, созданные этими программами, можно вызывать из своей программы с помощью компонента OleContainer, в котором будет показан обрабатываемый с помощью того или иного приложения документ.

После помещения компонента OleContainer на форму необходимо определить, какой конкретно это будет OLE-объект. В контекстном меню объекта выберите пункт Insert Object и в списке выберите тип документа, который требуется открыть.

При этом вызовется соответствующее приложение (к примеру, если выбран лист Excel, то запустится программа Microsoft Excel), и после выполнения действий над новым документом его можно будет сохранить в OLE-контейнере, выполнив в запустившемся приложении команду Файл • Закрыть. После этого внутри объекта на форме появится фрагмент электронной таблицы Excel.

Далее, когда программа уже будет откомпилирована и запущена, в контекстном меню данного элемента управления станут доступными два пункта: Изменить и Открыть. Если выбрать пункт Открыть, то для редактирования содержимого этого элемента запустится Excel, а если выбрать пункт Изменить, то сама форма примет вид, аналогичный Excel и позволит редактировать объект непосредственно внутри соответствующего элемента управления.



Технология OLE потенциально очень мощная и позволяет создавать различные управляющие программы и системы: планировщики работ и т. д.

Панель Dialogs

С двумя компонентами этой панели, содержащей стандартные диалоговые средства Windows, мы уже знакомы. Это диалоговые окна открытия и сохранения (OpenDialog и SaveDialog). Точно также можно работать и с диалоговыми окнами выбора и сохранения рисунка OpenPictureDialog и SavePictureDialog, которые отличаются только тем, что имеют заранее определенный набор типов изображений и позволяют предварительно просматривать изображения в правой части диалогового окна.

Компонент FontDialog позволяет выбрать шрифт (свойство Font) с заданными характеристиками размера, начертания и цвета.

Компонент ColorDialog предназначен для выбора цвета (он сохраняется в свойстве Color).

Для настройки принтера и подготовки к печати используется компонент PrinterSetupDialog.

Чтобы определить параметры печатаемого документа (выбор принтера, числа копий, диапазона страниц и т. п.), пригодится компонент PrintDialog.

Для вызова диалогового окна поиска нужной строки в файлах можно использовать компонент FindDialog. Введенная пользователем строка сохранится в свойстве FindText, а прочие характеристики — в многосоставном свойстве Options.

Аналогично устроен и компонент ReplaceOialog, который отображает стандартное диалоговое окно замены фрагмента текста. Искомая строка запишется в свойство FindText, а та, на которую ее надо заменить, — в свойство ReplaceText.

Все описываемые диалоговые компоненты реально не выполняют никаких действий — то есть диалоговое окно печати ничего не печатает, а диалоговое окно поиска ничего не ищет. Они предназначены исключительно для общения с пользователем — чтобы он выбрал принтер, указал число страниц, строку, которую надо найти, направление поиска и т. д. Конкретные действия, связанные с печатью или поиском, программист должен реализовывать сам.

Все диалоговые процессы запускаются с помощью метода Execute() без параметров и возвращают значение true, если пользователь закрыл их с помощью кнопки OK.

Маловато будет!

Хотя в стандартную поставку C++Builder входит около 200 компонентов, конечно, на все случаи жизни их не хватит. Огромное число компонентов, распространяемых бесплатно или условно-бесплатно, можно найти в Интернете. Самый популярный в этом плане Web-узел — www.torry.ru (правда, он сделан полностью на английском языке).

Итак, как же найти подходящий компонент и как его подключить к C++Builder?

На заглавной страничке www.torry.ru выберите раздел Components Page (страница компонентов). В левой части окна появится список разделов, в каждом из которых имеется около сотни всевозможных компонентов, доступных для скачивания. Они сопровождаются краткой информацией о том, что делает данный компонент, его названием и размером архива.

В разделе Buttons & Check Boxes # 2 (Кнопки и переключатели-2) найдите компонент Torry's Button, представляющий собой оригинальную овальную кнопку. Размер архива составляет 81 Кбайт. Скачайте его на свой компьютер и распакуйте во временный каталог.

Zip-архив содержит три файла: torrybtn.inf (справочная информация о компоненте), torrybtn.pas (исходный текст компонента на Паскале) и torrybtn.d32 (вспомогательный файл).

— Почему исходный текст компонента представлен паскалевским кодом, а не кодом Си++?

— Потому что визуально-компонентный подход был впервые реализован в Delphi — продукте, аналогичном C++Builder, только предназначенном для создания программ на Паскале. Полноценная поддержка этого подхода появилась в C++Builder только 4-й версии, вышедшей в феврале 1999 г. Хотя компонент представлен текстом, написанным на другом языке программирования, ничего страшного в этом нет. Как уже говорилось, C++Builder допускает создание приложений как на Си++, так и на Паскале, и имеет соответствующие компиляторы.

Для установки компонента Torry's Button закройте текущий проект и выполните команду Component • Install Component. На экране появится диалоговое окно установки нового компонента. Перейдите на страницу вкладки Into New Package (Установка в новый пакет), в строке Unit file name (Имя модуля) с помощью команды Browse (Обзор) выберите файл torrybtn.pas (при этом обратите внимание на расширение: оно должно быть .PAS, а не .CPP!), а в строке Package file name: введите torry.

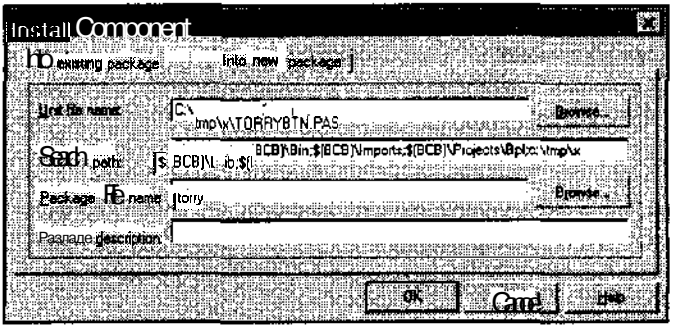


Рис. 50. Установка дополнительного компонента

Щелкните на кнопке OK, и начнется процесс установки нового компонента. C++Builder запросит подтверждение того, что требуется произвести установку пакета torry.bpl (он создастся на основе кода torry.pas автоматически).

Ответьте согласием — Yes. C++Builder выполнит компиляцию модуля и по окончании ее сообщит, что в систему добавлен (с указанием папки, куда он помещен) и зарегистрирован новый класс TTorryButton (ему будет соответствовать компонент TorryButton). Теперь он считается в C++Builder *стандартным,* наравне с такими классами, как TForm, TButton и другими.

Закройте диалоговое окно и перейдите на палитру компонентов. Где же искать установленный компонент? Как правило, он располагается на *новой* панели палитры. Прокрутите ее до конца — там действительно обнаружится новая панель Torry с одним компонентом.

Теперь закройте текущий проект, сохранив в нем все изменения, создайте новое приложение и разместите этот компонент на форме. Запустите программу — в окне появится оригинальная кнопка с подписью TorryButton (подпись меняется в свойстве Caption). При наведении указателя мыши на кнопку она принимает овальный вид, при нажатии на нее — меняет свой цвет, а в остальном ведет себя как обычная кнопка. Очень симпатичный компонент!

В некоторых случаях может потребоваться удаление ранее установленных компонентов. Чтобы, например, удалить компонент TorryButton, надо сделать так.

1. Выполнить команду Component ▸ Install Packages. (Компонент • Пакеты установки).
2. Найти в списке имя файла, соответствующее пакету с компонентом (оно указывалось ранее при установке — torry), и выделить его.

Не удаляйте пакеты, которые вы не устанавливали. Это может привести к неработоспособности C++Builder!

3. Щелкнуть на кнопке Remove (Удалить) и подтвердить удаление щелчком на кнопке Yes.
4. Щелкнуть на кнопке OK.

Класс TTorryButton и панель Torry удалятся из системы. При необходимости их можно будет установить снова.

Основные свойства компонентов C++Builder

Ниже приводится таблица, в которой описаны основные свойства наиболее часто используемых компонентов C++Builder. Эти свойства почти всегда имеют один и тот же смысл.

Примечания к таблицам

1. Все свойства объекта доступны не только на этапе проектирования, но и во время работы программы — их значения можно менять и в ходе ее выполнения, в программном коде.
2. Свойства, начинающиеся со слова Parent (Родитель), определяют различные аспекты общения объекта со своим «родителем» — объектом, на котором он расположен. Например, когда поле ввода помещается на форму, то для него «родителем» будет форма. Если командная кнопка размещена на панели инструментов, то для нее «родителем» будет панель и т. д. В большинстве прикладных задач эти свойства можно не учитывать.

Основные свойства компонентов

ActiveControl	Определяет, какой элемент управления на форме выделяется (имеет фокус) при запуске программы
Align	Выравнивание объекта внутри родителя. Принимает значения: alNone — нет выравнивания; alTop — по верхней границе; alBottom — по нижней границе; alLeft — по левой границе; alRight — по правой границе; alClient — по всему размеру родителя
Alignment	Горизонтальное выравнивание: по левому краю (taLeftJustify); по правому краю (taRightJustify); по центру (taCenter)
Anchors	Определяет, как объект будет перемещаться или сжиматься при изменении размеров «родителя». Например, если включить два подсвойства akLeft и akRight, то ширина объекта будет сжиматься при уменьшении его ширины. По умолчанию не используется
AutoHotkeys	Режим автоматического формирования «горячих» букв для пунктов меню, исключающий задание одинаковых букв

AutoScroll	Если равно true, то разрешается автоматическое появление полос прокрутки, когда содержимое объекта не умещается в его видимой части
AutoSize	Если равно true, то размер объекта будет автоматически подстраиваться под размер его содержимого
BiDiKeyboard	Кодовое название используемой раскладки клавиатуры при вводе текста справа налево
BiDiMode	Способ поведения объектов с учетом национальной специфики — ввод текста и прокручивание информации слева направо (bdLeftToRight, по умолчанию) или справа налево (bdRightToLeft)
BorderIcons	Набор значков в системном меню формы
BorderStyle	Вид рамки объекта. Принимает значения: bsNone (нет рамки); bsSingle (простая рамка); bsSizeable (рамка, позволяющая изменять размеры объекта мышью); bsDialog (рамка в стиле диалоговых окон); bsToolWindow (как bsSingle, но с небольшим заголовком); bsSizeToolWin (как bsSizeable, но с небольшим заголовком)
BorderWidth	Ширина рамки
Cancel	Определяет, будет ли происходить для данного объекта событие OnClick, когда пользователь нажмет клавишу ESC (для этого Cancel должно быть равно true).
Caption	Заголовок. Для одних объектов применяется, чтобы задать заголовок в окне или надпись на кнопке, для других — описывает их содержимое (например, у полей надписи)
Checked	Свойство, определяющее состояние флажков {true — включен)
Color	Цвет объекта
Columns	Число столбцов
Constraints	Содержит четыре подсвойства, определяющие минимально и максимально допустимые размеры объекта
Ctl3D	Вид объекта в объемном стиле (стандарт для Windows 95 и выше)
Cursor	Задает вид указателя (отображается при наведении на объект)

Default	Определяет, будет ли происходить для данного объекта событие **OnClick,** когда пользователь нажмет клавишу ENTER (для этого свойство Default должно иметь значение true)
DockSite	Применяется для различных панелей, объектов группирования и т. д. Когда включено **(true),** позволяет использовать этот объект как базу для стыковки других объектов на форме при перетаскивании. Обеспечивает режим «прилипания» (см. свойство **Drag**Kind)
Down	Состояние кнопки **(true** — нажата)
DragCursor	Задает вид указателя, который отображается при перетаскивании объекта мышью
DragKind	Определяет, можно ли объект произвольно перетаскивать по окну **(dkDrag)** или же его можно перемещать как стыкуемый объект **(dkDock),** который сам определяет свою форму при стыковке с другими объектами
DragMode	Определяет, можно ли объект перетаскивать (значение **dmAutomatic)** или нет **(dmManual,** по умолчанию) в соответствии с режимом, указанным в DragKind
Enabled	Включен ли объект. Когда Enabled имеет значение false, объект становится недоступным для пользователя (например, кнопка становится серой, и нажать на нее нельзя)
FileName	Имя файла. Используется для сохранения имени выбранного файла в стандартных диалоговых окнах, для указания файла с мультимедийным содержимым, для компонентов воспроизведения музыки или звука и т. д.
Flat	Вид границ объекта (true — плоские)
Font	Определяет шрифт, которым будут выполняться все надписи внутри объекта. Содержит множество **подсвойств**
FormStyle	Стиль формы. Может принимать значения: **fsNormal** (обычное окно); fsMDIChild (внутреннее окно многооконного приложения): **fsMDIForm** (родительское окно многооконного приложения); fsStayOnTop (всегда на поверхности экрана)
Glyph	Картинка, содержащая от одного до четырех (отпущено, недоступно, нажато, выделено) изображений графической **кнопки**
Height	Высота объекта

HelpContext	Номер экрана справочной системы, который будет вызываться нажатиемклавиши F1 при работе с объектом (требует знания устройства справочной системы Windows)
HelpFile	Имя файла справочной системы для данного приложения или окна
HorzScrollBar	Вид и стиль горизонтальной полосы прокрутки. Состоит из множества подсвойств
Highlighted	Задает выделение текущей панели окна особым цветом
Hint	Текст подсказки, которая всплывает при наведении указателя мыши на объект. Эта подсказка будет отображаться, если в свойстве ShowHint установлено значение true
HintFont	Шрифт, которым выводится всплывающая подсказка
Icon	Имя файла со значком для приложения
Images	Список картинок, которые будут использоваться для показа кнопок на панели инструментов, в пунктах меню и т. д.
ItemEnabled	Массив состояний флажков для компонента Список флажков
ItemIndex	Текущий выбранный элемент в списке Items. Нумерация начинается с нуля. Если ничего не выбрано, то значение ItemIndex равно —1
Items	Список объектов, хранящих основные данные. Используется во всевозможных списках, меню, наборах
Kind	Вид (тип) объекта. Для разных компонентов может принимать разные значения
Left	Левая координата объекта на родителе
Lines	Список строк. Используется в многострочных текстовых элементах. По структуре похож на свойство Items
Max	Максимальное значение диапазона, используемого во всевозможных компонентах прокрутки, ползунках и т. п.
MaxLength	Максимально допустимая длина вводимой строки символов
MenuAnimation	Режим анимированного появления меню, принятый в операционной системе Windows 2000

MenuFont	Шрифт, которым пишутся пункты меню
Min	Минимальное значение диапазона, используемого во всевозможных компонентах прокрутки, ползунках и т. п.
ModalResult	Значение, которое возвращает модальное диалоговое окно при его закрытии. Может принимать значения mrNone (по умолчанию), mrOk, mrCancel, mrAbort, mrRetry, mrIgnore, mrYes, mrNo и mrAll. Если во время работы диалога в это свойство записать значение, не равное mrNone, то диалоговое окно сразу закроется
Name	Название объекта (имя переменной, которая будет использоваться в программе для обращения к этому объекту)
Options	Опции настройки объекта. Различаются для разных компонентов
Orientation	Ориентация объекта. Может быть горизонтальной (udHorizontal) или вертикальной (udVertical)
OwnerDraw	Характеризует, будет ли объект рисовать свой образ на экране самостоятельно (true) или использовать стандартный вид
Picture	Картинка
PixelsPerInch	Число точек (пикселов) на дюйм экрана. Используется для масштабирования формы в зависимости от экранного разрешения. Будет учитываться, если свойство Scaled имеет значение true
PopupMenu	Контекстное меню, связанное с объектом и вызываемое при щелчке правой кнопки мыши над этим объектом. Выбирается в раскрывающемся списке доступных меню и должно быть подготовлено заранее
Position	Для формы — положение на экране. Принимает значения: poDesigned — положение окна во время работы программы соответствует положению формы на экране на этапе проектирования; poDefault — положение определяется Windows; poDefaultPosOnly — используются размеры, заданные разработчиком; poDefaultSizeOnly — используется положение, заданное разработчиком; poScreenCenter — положение в центре экрана для многомониторных систем; poDesktopCenter — положение в центре экрана. Для всевозможных компонентов прокрутки, ползунков и т. д. — текущее положение бегунка

ReadOnly	Если данное свойство имеет значение true, то объект будет доступен в режиме «только для чтения». Например, поле ввода с включенным свойством ReadOnly позволяет просматривать содержимое, но не изменять его
Scaled	Если имеет значение true, то учитывается свойство PixelsPerInch
ShowHint	Определяет, надо ли показывать всплывающую подсказку, хранящуюся в свойстве Hint
Sorted	Используется во всевозможных списках. Когда включено (true), содержимое списка будет автоматически сортироваться перед выводом на экран
Style	Стиль объекта. Специфичен для разных компонентов (кнопок, списков, шрифтов и т. д.)
TabOrder	Определяет номер объекта при передвижении по элементам управления в родительском окне с помощью клавиши TAB. Начинается с О
TabStop	Определяет, будет ли происходить выделение данного объекта и остановка на нем при передвижении по элементам управления в родительском окне с помощью клавиши TAB
Tag	Свойство, сделанное специально для разработчиков. Оно нигде не используется и предназначено только для хранения числа типа int. Tag можно рассматривать как пользовательское свойство и применять его для собственных нужд
Text	Содержимое различных текстовых элементов управления (полей ввода и т. п.)
Title	Титул (заголовок) приложения, колонки, сообщения
Top	Верхняя координата объекта на родителе
Transparent	Прозрачность фона объекта. Например, если свойство Transparent имеет значение true для поля надписи, помещенного на объект-изображение, то отображаться на этом изображении будет только текст, а затирания изображения фоном не произойдет
VertScrollBar	Вид и стиль вертикальной полосы прокрутки. Состоит из множества подсвойств

Visible	Определяет, будет ли видим объект во время работы программы (по умолчанию — true)
Width	Ширина объекта
WindowState	Статус окна при его открытии. Принимает значения: **wsNormal** — используются размеры и положение, заданные на этапе проектирования; **wsMinimized** — в свернутом виде; **wsMaximized** — в развернутом виде

Заключение

Что еще может Borland C++Builder

Мы познакомились с основными возможностями системы программирования Borland C++Builder производства корпорации Inprise. С ее помощью нам теперь вполне по силам создание разнообразных *настольных {desktop)* приложений, таких, как электронные таблицы, персональные менеджеры, записные книжки, программы составления расписания рабочего дня, различные файловые утилиты, не очень сложные мультимедийные приложения, обучающие программы, игры и многое другое. Однако осталось еще множество нерассмотренных возможностей C++Builder, которые выделены Inprise в стратегическое направление и очень активно развиваются в каждой новой версии C++Builder и Delphi. Даже для краткого их описания потребуется не одна книга.

Работа с базами данных

Компоненты для работы с базами данных — файлами, организованными по одному шаблону (панели Data Access, Data Contrlos, InterBase и ADO), легко позволяют обрабатывать большие массивы записанной на жесткий диск информации, тратя при этом минимальное время на программирование, а в некоторых случаях просто обходясь визуальной настройкой свойств. Например, не написав ни одной строчки кода, можно создать приложение, которое будет считывать данные из базы (не обязательно текст или числа — это могут быть и рисунки и что-то другое), показывать их на экране в виде таблицы, позволять просматривать записи, добавлять, изменять и удалять их, печатать отчеты по содержимому базы и делать еще множество полезных вещей. Причем работать можно с данными, хранящимися как на компьютере пользователя, так и на другом компьютере *{сервере)* — соединенным по сети. Системы управления базами данных, функционирующие на сервере, сами по себе являются сложными программами. Среди них такие

известные продукты, как Microsoft SQL Server, Oracle, IBM DB2, Inprise InterBase и другие.

Анализ данных

Когда информация хранится в базе данных, ее очень удобно обрабатывать с помощью программ автоматического анализа, способных представлять данные в виде всевозможных трехмерных графиков (например, показывать объемы продаж в зависимости от цены и марки товара), причем не просто отображать какие-то три измерения, а хранить сведения по множеству самых разных аспектов работы и самостоятельно выполнять различные подсчеты (панель Decision Cube). Сегодня направление аналитической обработки данных — одно из наиболее бурно развивающихся направлений современных информационных технологий.

Создание отчетов

С базами данных и анализом информации неразрывно связаны гибкие возможности C++Builder (панель QReport) по построению отчетов. Структура отчета может быть произвольной. Проектируется она в специальном визуальном редакторе (содержащем, в частности, встроенный калькулятор), способном, например, автоматически отображать данные из базы или трехмерные диаграммы. Созданный отчет в дальнейшем можно просмотреть и распечатать на любом принтере или сохранить в файле.

Интернет

На основе нескольких десятков компонентов с панелей Internet и FastNet можно реализовать практически любую программу для работы с Интернетом — будь то почтовое приложение, обозреватель собственной конструкции, программа публикации в Интернете автоматически обновляемой информации из базы данных, всевозможные служебные программы и т. д. А можно написать и целую многопользовательскую систему для ведения онлайновых разговоров, виртуального общения, торговли, развлечений и многого другого.

Распределенные вычисления

Имеется еще одна очень большая область программирования, которой в не очень далеком будущем отдает пальму первенства большинство ведущих специалистов и экспертов по информационным технологиям. Это — область создания распределенных приложений (панели Midas и Internet Express), когда

программа разбивается (распределяется) на набор модулей, каждый из которых выполняет свою часть общей задачи и работает при этом на *отдельном* компьютере в сети — она может быть локальной, а может быть, и глобальной (Интернетом). Надежность подобных приложений становится очень высокой — ведь если один компьютер выйдет из строя, то остальные продолжат работу, а временно отключившийся модуль автоматически перезапустится на другом доступном компьютере. Кроме того, благодаря программам управления загрузкой компьютеров удается очень эффективно регулировать эту загрузку и вместо дорогих суперЭВМ, стоящих миллионы долларов, можно обходиться сотней недорогих персоналок, получая при этом схожие по вычислительной мощности ресурсы.

Серверы

В последние годы офисные приложения Microsoft (редактор Word, электронная таблица Excel, средство создания презентаций PowerPoint, СУБД Access, почтовый клиент Outlook) фактически стали стандартом в своей области. Они построены на основе компонентной технологии COM, напоминающей компонентные решения C++Builder — программы собираются из готовых компонентов COM. Но самое важное, что такими программами можно управлять из других приложений: например, дать команду редактору загрузить документ, проверить правописание и сохранить исправленную версию или скомандовать электронной таблице выполнить определенные вычисления и сформировать и напечатать нужный отчет — и все это не вручную, а автоматически!

Программы, построенные на основе COM-технологии и допускающие управление своей работой из других приложений, называются COM-серверами. Компоненты для создания программ, управляющих работой COM-серверов (в основном офисных систем Microsoft), находятся на панели Servers.

Перспективы

Особый акцент на поддержке всех этих передовых технологий корпорация Inprise делает в своем новом продукте JBuilder 4. Это среда программирования на Java, языке, созданном на основе Си++ и позволяющем создавать приложения, независимые от операционной системы. Например, можно написать программу в Windows с красивым графическим интерфейсом, а потом без изменения исходного текста перенести ее на *любую* другую платформу, от Unix до систем для суперкомпьютеров. Важную роль в технологиях корпорации Inprise играет также проект Kylix. Это комбнация очередной версии C++Builder и среды программирования на Паскале Delphi. Она

позволяет на основе одного исходного текста формировать приложения для двух платформ: Windows и Linux. Не забыты, конечно, и создатели настольных приложений. Им предлагаются новые, еще более простые и удобные средства визуального программирования.

Кроме того, очень большую и регулярно обновляемую подборку разнообразной информации и ссылки на множество сайтов с бесплатными компонентами для C++Builder можно найти на Web-узле для разработчиков www.borland.com и www.borland.ru (подразделение Borland входит в состав Inprise).

Алфавитный указатель