Министерство образования Российской Федерации
Южно-Уральский государственный университет

Кафедра автоматизации механосборочного производства

681.3.06(07)
П305

Петрова Л.Н.

ПРОГРАММИРОВАНИЕ И ОСНОВЫ АЛГОРИТМИЗАЦИИ

Учебное пособие к лабораторным работам

Под редакцией В.А. Смирнова

Челябинск

Издательство ЮУрГУ

2003

УДК 681.3.06(075.8)

Петрова Л.Н. Программирование и основы алгоритмизации: Учебное пособие к лабораторным работам./ Под ред. В.А. Смирнова. — Челябинск: Изд-во ЮУрГУ, 2003. — 56 с.

Учебное пособие к лабораторным работам по курсу «Программирование и основы алгоритмизации» предназначено для студентов специальности 210200. В нем изложены основные примеры объектно-ориентированного визуального программирования, методика работы с интегрированной средой разработки C++Buider и дается справочный материал по ряду компонентов.

Может использоваться студентами других специальностей для создания прикладных программ в среде C++Buider.

Ил. 12, табл. 12.

Одобрено учебно-методической комиссией механико-технологического факультета.

Рецензенты: Кувшинов М.С., Юсупов Р.Х.

ВВЕДЕНИЕ

C++Builder — система объектно-ориентированного программирования для операционных систем Windows 95/98/2000 и NT. Интегрированная среда разработки (Integrated Development Environment, IDE) обеспечивает ускорение визуального проектирования, а также продуктивность повторно используемых компонентов в сочетании с усовершенствованными инструментами и разномасштабными средствами доступа к базам данных.

Основная цель множества введенных в C++Builder модификаций — радикально ускорить производственный цикл разработки как персональных, так и коллективных коммерческих проектов для различных отраслей применения.

В контексте C++Builder быстрая разработка приложений (Rapid Application Development, RAD) подразумевает не только реальное ускорение типичного цикла «редактирование→компиляция→компоновка→прогон→отладка», но и придает создаваемым объектам изящество компонентной модели.

Объектно-ориентированный подход уже давно стал естественным для любых прикладных программ (далее приложений) Windows пользователя. После получение сообщения происходят какие-то действия (проводятся вычисления, выполняется какая-то программа, появляется новое диалоговое окно и т.п.). После этого опять всякая активность затухает, пока тот или иной объект не получит нового сообщения от пользователя или от объекта. Таким образом, объектно-ориентированная программа не имеет жесткого алгоритма работы. Она представляет собой систему объектов, каждый из которых может выполнять какие-то функции в ответ на полученное сообщение.

Визуальное программирование позволило свести проектирование пользовательского интерфейса к простым и наглядным процедурам, которые дают возможность за минуты или часы сделать то, на что ранее уходили месяцы работы.

Интегрированная среда разработки C++Builder [1, 2] представляет пользователю формы (в приложениях их может быть несколько), на которых размещаются компоненты. С помощью простых манипуляций можно изменять размеры и расположение этих компонентов. При этом в процессе проектирования постоянно виден результат — изображение формы и расположение на ней компонентов.

Но самое главное заключается в том, что во время проектирования формы и размещения на ней компонентов C++Builder автоматически формирует код программы, включая в нее соответствующие фрагменты, описывающие данный компонент. То есть проектирование сводится, фактически, к размещению компонентов на форме, заданию некоторых их свойств и написанию, при необходимости, обработчиков событий.

ИНТЕГРИРОВАННАЯ СРЕДА РАЗРАБОТКИ (IDE) C++BUILDER

Общий вид окна IDE

После запуска C++Builder откроется основное окно IDE. Его вид представлен на рис.1.

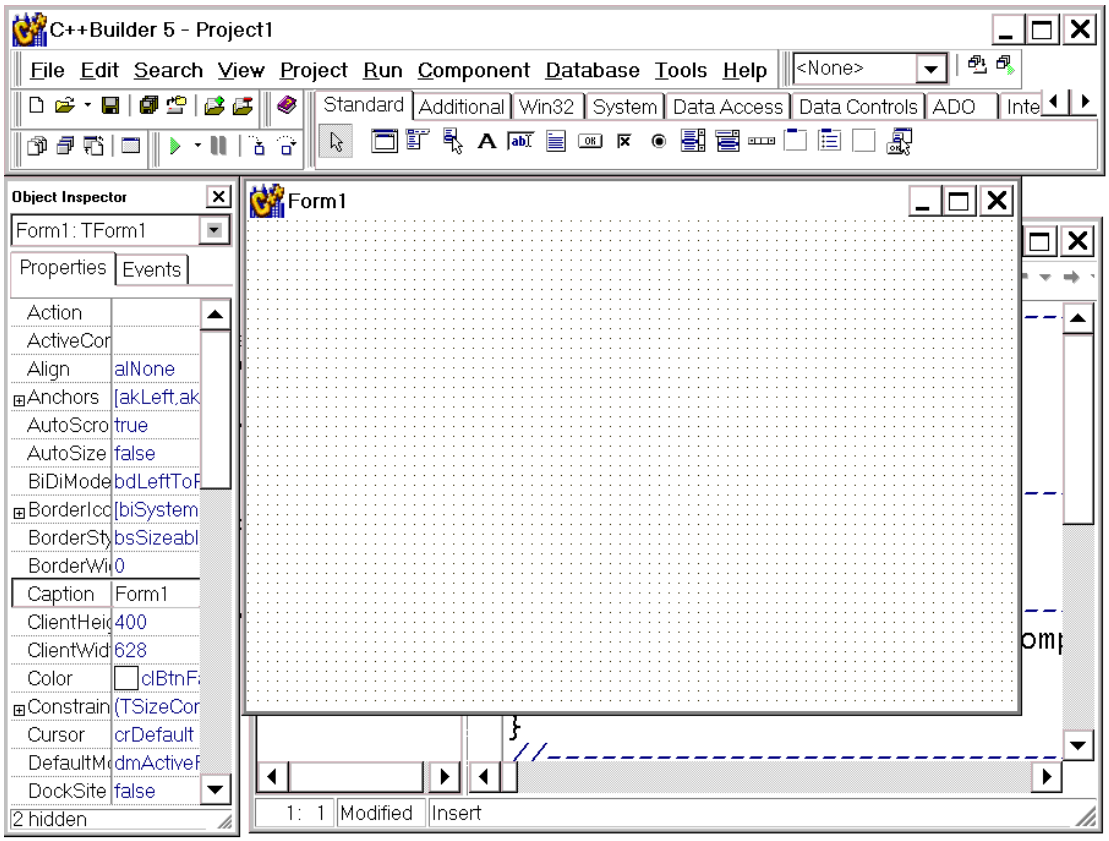


Рис.1.Основное окно IDE C++Builder

В верхней части окна расположена **полоса главного меню**, ниже — две **инструментальные панели**. Левая панель содержит два ряда **быстрых кнопок**, дублирующих некоторые наиболее часто используемые команды меню. Правая панель содержит **палитру компонентов** библиотеки визуальных компонентов (Visual Component Library — VCL). Палитра компонентов содержит ряд страниц, закладки которых видны в ее верхней части.

Правее полосы главного меню размещена еще одна небольшая инструментальная панель, содержащая выпадающий список и две быстрые кнопки. Это панель сохранения и выбора различных конфигураций окна IDE, которые можно создавать и запоминать.

В основном поле окна слева расположено окно Инспектора Объектов (Object Inspector), с помощью которого в дальнейшем можно задавать свойства компонентов и обработчики событий. Правее находится окно пустой формы, готовой для переноса на нее компонентов. Под ним расположено окно Редактора Кодов.

Полоса главного меню и всплывающие меню

Полоса главного меню для C++Builder представлены на рис.2.

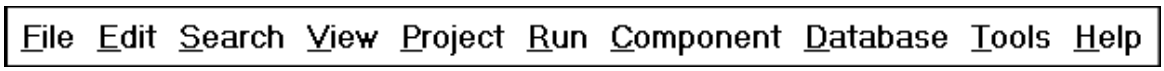


Рис.2.Полоса главного меню

Ниже приведен краткий обзор основных разделов:

- разделы меню **File** (файл) позволяют создать новый проект, новую форму, открыть ране созданный проект или форму, сохранить проекты или формы в файлах с заданными именами;
- разделы меню **Edit** (правка, редактирование) позволяют выполнять обычные для приложений Windows операции обмена с буфером Clipboard, а также дают возможность выравнивать группы размещенных на форме компонентов по размерам и местоположению;
- разделы меню **Search** (поиск) позволяют осуществить поиск и конкретные замены в коде приложения, которые свойственны большинству известных текстовых редакторов;
- разделы меню **View** (просмотр) позволяют вызывать на экране различного окна, необходимые для проектирования;
- разделы меню **Project** (проект) позволяют добавлять и убирать из проекта формы, задавать опции проекта, компилировать проект без его выполнения и делать много других полезных операций;
- меню **Run** (выполнение) дает возможность выполнять проект в нормальном или отладочном режимах (продвигаться по шагам, останавливаться в указанных точках кода, просматривать значения переменных и т.д.);
- меню **Component** (компонент) позволяет создавать и устанавливать новые компоненты, конфигурировать палитру компонентов, работать с пакетами; меню **Help** (справка) содержит разделы, помогающие работать со встроенной в C++Builder справочной системой;
- меню **Tools** (инструментарий) включает ряд разделов, позволяющих настраивать IDE и выполнять различные вспомогательные программы, например, вызывать редактор изображений (Image Editor), работать с программами, конфигурирующими базы данных и т.д. Кроме того, в это меню вы можете сами включить любые разделы, вызывающие те или иные приложения, и таким образом расширить возможности главного меню C++Builder, приспособив его для своих задач.

Кроме полосы главного меню в C++Builder имеется система контекстных всплывающих меню, которые появляются, если пользователь поместил курсор мыши в том или ином окне или на том или ином компоненте и щелкнул правой кнопкой мыши.

Быстрые кнопки

Инструментальные панели быстрых кнопок для C++Builder представлены на рис.3.

а)

б)

Рис.3.Инструментальные панели в C++Builder: основные (а) и панель выбора конфигурации окна (б)

В табл.1 приведены пиктограммы этих кнопок, соответствующие им команды меню и «горячие» клавиши, а также краткие пояснения.

Таблица 1

Быстрые кнопки

Пикто-грамма	Команда меню / «горячие» клавиши	Пояснение команды
	File | New	Открыть проект или модуль из Депозитария.
	File | Open File | Reopen	Открыть файл проекта, модуля, пакета. Кнопка со стрелкой справа соответствует команде Reopen, позволяющей открыть файл из списка недавно использовавшихся.
	File | Save (Ctrl – S)	Сохранить файл модуля, с которым в данный момент идет работа.
	File | Save All	Сохранить все (все файлы модулей и файл проекта).
	File | Open Project (Ctrl – F11)	Открыть файл проекта.
	Project | Add to Project (Shift – F11)	Добавить файл в проект.
	Project | Remove from Project	Удалить файл из проекта.
	Help | C++Builder Help	Вызов страницы «Содержание» встроенной справки.
	View | Units (Ctrl – F12)	Переключиться на просмотр текста файла модуля, выбираемого из списка.
	View | Form (Shift – F12)	Переключение на просмотр формы, выбираемой из списка.
	View | Toggle Form/ Unit (F12)	Переключение между формой и соответствующим ей файлом модуля.
	File | New Form	Включить в проект новую форму.

Окончание таблицы 1

	Run | Run	Выполнить приложение. Кнопка со стрелкой справа позволяет выбирать выполняемый файл, если вы работаете с группой приложений
	Run | Program Pause	Пауза выполнения приложения и просмотр информации CPU. Кнопка и соответствующий раздел меню доступны только во время выполнения приложения
	Run | Trace Into (F7)	Пошаговое выполнение программы с заходом в функции
	Run | Step Over (F8)	Пошаговое выполнение программы без захода в функции
	View | Desktops | Save Desktop	Сохранение текущей конфигурации окна
	View | Desktops | Set Debug Desktop	Установка конфигурации окна при отладке

На рис.3 и в табл.1 приведен стандартный состав инструментальных панелей быстрых кнопок. Однако в C++Builder есть возможность настраивать панели по своему усмотрению.

Палитра компонентов

Палитра компонентов (рис.4) — это витрина библиотеки визуальных компонентов (VCL). Она позволяет сгруппировать компоненты в соответствии с их смыслом и назначением. Эти группы или страницы снабжены закладками. Можно изменять комплектацию страниц, вводить новые, переставлять их, вносить на страницы, разработанные пользователем шаблоны и компоненты.

Рис.4.Палитра компонентов

По умолчанию в палитре C++Builder имеются следующие страницы.

Standard	Стандартная, содержащая наиболее часто используемые компоненты.
Additional	Дополнительная, являющаяся дополнением стандартной.
Win32	32-битные компоненты в стиле Windows 95/98 и NT.
System	Системная, содержащая такие компоненты, как таймеры, плееры и ряд других.
Data Access	Доступ к данным через Borland Database Engine (BDE).
Data Controls	Управление данными.
ADO	Связь с базами данных через Active Data Object (ADO) —

	множество компонентов ActiveX, использующих для доступа к информации баз данных Microsoft OLE DB.
Inter Base	Прямая связь с InterBase, минуя Borland Database Engine (BDE) и Active Data Object (ADO).
Midas	Построение приложений баз данных с параллельными потоками.
Internet Express	Построение приложений Internet Express — одновременно приложений сервера Web и клиента баз данных с параллельными потоками.
Internet	Компоненты приложений, работающих и Internet.
Fast Net	Различные протоколы доступа к Internet.
Decision Cube	Многомерный анализ данных.
QReport	Быстрая подготовка отчетов.
Dialogs	Стандартные системные диалоги Windows.
Win 3.1	Компоненты в стиле Windows 3.x (для обратной совместимости).
Samples	Образцы, различные интересные, но не до конца документированные компоненты.
ActiveX	Активные элементы ActiveX.
Serves	Оболочки VCL для распространенных сервером COM.

Чтобы перенести компонент на форму, надо открыть соответствующую страницу библиотеки и указать курсором мыши необходимый компонент. При этом кнопка-указатель  приобретет вид нажатой кнопки. Далее вы можете выбрать компонент и указать место на форме, куда необходимо его поместить. Есть и другой способ поместить компонент на форму — достаточно сделать двойной щелчок на пиктограмме компонента в палитре, и он автоматически разместится в центре вашей формы.

Окно формы и окно редактора кода

Основой почти всех приложений C++Builder является форма, на которую размещаются другие компоненты. Форма имеет те же свойства, что присущи другим окнам Window 95/98.

Во время проектирования форма покрыта сеткой из точек, в узлах которой могут размещаться компоненты. Во время выполнения сетка не видна.

Окно редактора кода расположено под окном форм и показано на рис.5. Редактор кода является полноценным текстовым редактором, который можно настраивать на различные стили работы. В редакторе применяется выделение цветом и шрифтом синтаксических элементов.

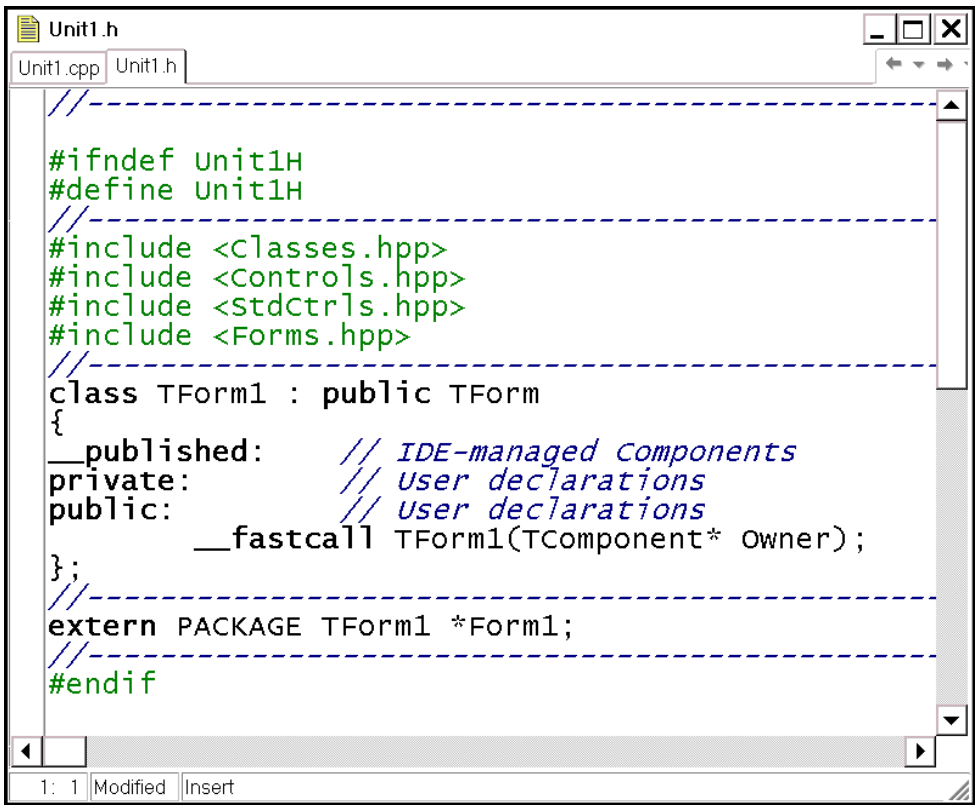


Рис.5.Окно редактора кода

Жирным цветом выделяются ключевые слова C++ (на рис.5 это **class, __published, public**), зеленым цветом выделяются директивы препроцессора (на рис.5 это директива **#include**), синим курсивом выделяются комментарии (на рис.5 это текст «// User declarations»).

Инспектор объектов

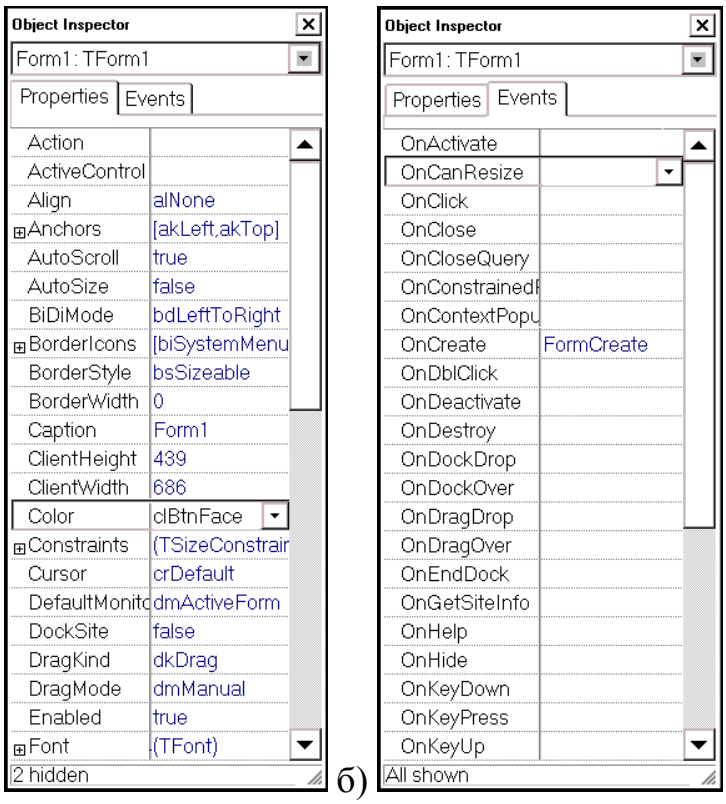
Инспектор объектов (Object Inspector) обеспечивает простой и удобный интерфейс для изменения свойств объектов C++Builder и управления событиями, на которые реагирует объект.

Окно инспектора объектов (рис.6) имеет две страницы. Выше их имеется выпадающий список компонентов, размещенных на форме. В нем пользователь имеет возможность выбрать тот компонент, свойства и события (**Events**) которого его интересуют.

Страница свойства (**Properties**) инспектора объектов (рис.6а), показывает свойства того объекта, который в данный момент выделен. Есть возможность изменять эти свойства.

Рядом с некоторыми свойствами можно увидеть знак плюс. Это означает, что данное свойство является объектом, который в свою очередь имеет ряд свойств.

Страница событий (**Events**) составляет вторую часть инспектора объектов (рис.6б). На ней указаны все события, на которые может реагировать выбранный объект. Например, если необходимо выполнить какие-то действия в момент создания формы, то нужно выбрать событие **OnCreate**.



а) б)

Рис.6.Страница свойств (а) и страница событий (б) инспектора объектов

Рядом с именем этого события откроется окно с выпадающим списком. Двойной щелчок в окне списка откроет нужный обработчик события.

В окне редактора кода, появится текст:

```
void__fastcall TForm1 :: FormCreate
(TObject*Sender)
{

}
```

Приведенный выше код — это заготовка обработчика событий, которую автоматически создан C++Builder. Остается только в промежутках между скобками «{» и «}», написать необходимые операторы.

Проекты C++Builder

Проект C++Builder состоит из форм [3, 4], модулей с их заголовочными файлами и файлами реализации, установок параметров проектов, ресурсов и т.д. Вся эта информация размещается в файлах. Многие из этих файлов автоматически создаются C++Builder, когда создается приложение. Из всех файлов важнейшими являются файлы .**cpp (**головной проект или файл реализации модуля**)**, .**h (**заголовочный файл модуля**)**, .**dfm (**файл формы**)**, .**bpr (**файл опций проекта**)**, .**res (**файл ресурсов проекта**)**. Это те файлы, которые необходимы для переноса ранее созданного проекта на другой компьютер.

ЛАБОРАТОРНЫЕ РАБОТЫ

Лабораторная работа №1

Тема: «Знакомство с Borland C++Builder»

Цель работы: познакомиться с оболочкой Borland C++ Builder и получить первые навыки по созданию Windows-приложения «Светофор».

Порядок работы: прежде чем приступить к созданию первого приложения необходимо самостоятельно познакомиться с интегрированной средой разработки (IDE) C++Builder, краткая информация, о которой приведена во введении данного приложения.

Для создания нового приложения запустите Borland C++ Builder и выберите в главном меню пункт File|New Application.

В инспекторе объектов выберите свойство **Name** и задайте новое имени формы, например **MainForm**.

Для дальнейшей работы необходимо выбрать компоненты, поместить их на форму, выровнять при необходимости.

Компонентом называется функционально законченный участок двоичного кода, выполняющий некоторую предопределенную функцию (например, отображение текста, реализация линейки прокрутки и т.п.). Использование компонента сводится к помещению графического значка, также называемого компонентом, в форму создаваемого приложения.

Для **выбора** компонента необходимо выбрать соответствующую вкладку палитры компонентов и щелкнуть по требуемому компоненту. Щелчок внутри формы приведет к помещению, выбранного компонента в форму — верхний левый угол компонента будет совпадать с положением курсора в момент щелчка.

Для **помещения** нескольких одинаковых компонентов необходимо щелкнуть по компоненту в палитре компонентов при нажатой клавише **Shift**. Каждый последующий щелчок внутри формы позволят помещать в нее новый компонент. Прекращения вставки осуществляется щелчком по стрелке в левой части палитры компонентов.

Для **выделения** одного компонента в форме нужно по нему щелкнуть мышью или несколько компонентов выделить рамкой с черными узлами. Несколько компонентов можно также выделить последовательными щелчками при нажатой клавише **Shift**. Выделенные компоненты имеют обрамление в виде рамки с серыми узлами. Повторный щелчок по компоненту при нажатой клавише **Shift** снимает выделение с компонента. Выделение компонентов можно осуществлять путем охвата пунктирной рамкой, появляющейся при нажатии на левую кнопку мыши.

Выделенный компонент (компоненты) можно перемещать по форме при помощи мыши. При выходе компонента за границы формы он будет усекаться. В ряде случаев смещение компонента за границы вызовет появление линеек прокрутки. Используя мышь можно изменять размеры одиночного выделенного компонента (но не у всех компонентов).

Для **выравнивания** компонента внутри формы наиболее часто используется палитра выравнивания **Align** (команда основного меню View/Alignments Palette).

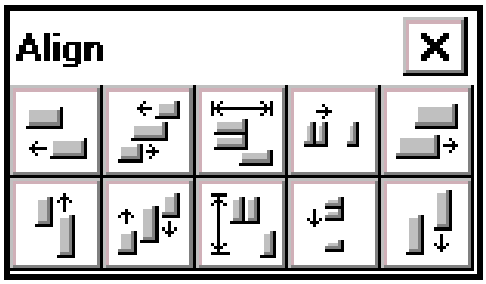


Рис.7.Окно выравнивания

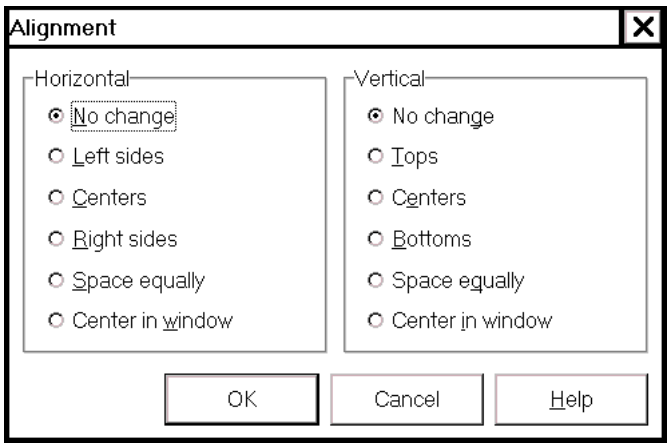


Рис.8.Окно палитры выравнивания

Функции кнопок данного окна приведены ниже.

1. Выровнять левые края.
2. Выровнять центры по вертикали.
3. Отцентрировать в окне по горизонтали.
4. Установить равный шаг по горизонтали.
5. Выровнять правые края.
6. Выровнять верхние края.
7. Выровнять центры по горизонтали.
8. Отцентрировать в окне по вертикали.
9. Установить равный шаг по вертикали.
10. Выровнять нижние края.

Выравнивание может быть применено как к одному выделенному компоненту, так и к группе выделенных компонентов.

Задание 1

Создайте новое приложение. Заполните форму нового приложения пятью компонентами **Button** (кнопка), находящимися на вкладке **Standard**. Месторасположение данных компонентов произвольно.

Используя полученные знания, закрепите навыки выделения компонентов, изменения размеров отдельных компонентов, перемещения компонента (компонентов), а также выравнивания компонентов с использованием палитры выравнивания.

В **инспекторе объектов** отображаются свойства и события компонентов. Инспектор отображает свойства выделенного компонента или общие свойства для группы выделенных компонентов.

Для знакомства с некоторыми свойствами компонентов выполните следующие действия.

1. Очистите форму от всех имеющихся компонентов. Для этого выделите все компоненты, например командой Edit/Select All главного меню и нажмите клавишу **Delete**.

2. Поместите в форму компонент **Shape**, находящийся на вкладке **Additional** палитры компонентов. Инспектор объектов отражает свойства или события этого

компонента, о чем свидетельствует надпись Shape1 : TShape в ниспадающем меню в верхней части инспектора объектов.

3. Выберите, если необходимо, вкладку "Свойства" (**Properties**) инспектора объектов.

4. Найдите свойство **Name** и измените имя, предложенное по умолчанию с **Shape1** на **Circle**. Обратите внимание на изменения в ниспадающем меню.

5. Найдите свойство **Brush**. Плюс перед названием свойства означает наличие раскрывающегося меню, расширяющего данное свойство. Дважды щелкните по названию свойства **Brush**. Обратите внимание на появление меню со свойствами **Color** и **Style**, а также на изменение знака "+" на "–" перед свойством **Brush**.

6. Щелкните один раз по свойству **Color**. Справа от названия свойства появится список возможных значений этого свойства. Раскройте этот список и выберите **clRed**. Обратите внимание на изменение цвета компонента в форме.

7. Найдите свойство **Shape** и выберите **stCircle**.

Выполнив данные действия, из белого квадрата получим красный круг.

Изменение и контроль свойств компонентов, а также привязка их изменений к событиям составляют существенную часть программирования в Borland C++ Builder.

Задание 2

Создайте при помощи нескольких компонентов, изменяя, при необходимости их свойства, модель светофора в виде прямоугольника синего цвета. Задайте компоненту, изображающему красный фонарь, имя (свойство **Name**) **Red**, желтый фонарь – имя **Yellow**, зеленый фонарь – имя **Green**.

> ***Внимание!*** Выполните следующие действия точно в соответствие с указанным порядком.

1. Добавьте в форму компонент **Timer** (вкладка **System**).
2. Измените свойство **Name** этого компонента на **Timer**.
3. Дважды щелкните по компоненту **Timer**, который уже расположен на форме. Откроется окно редактора кода. Курсор расположен между открывающей и закрывающей тело функции скобками.
4. Далее приводится текст обработчика события для компонента таймер. Введите следующую программу.

```
if (Red→Brush→Color = = clRed)
    {
    Red→Brush→Color = clGray;
    Yellow→Brush→Color = clYellow;
    }
    else
    {
    if (Yellow→Brush→Color = = clYellow)
       {
      Yellow→Brush→Color = clGray;
```

```
        Green→Brush→Color = clGreen;
        }
        else
        {
        Green→Brush→Color = clGray;
        Red→Brush→Color = clRed;
        }
}
```

5. Сохраните созданное приложение.

Сохранение лучше всего выполнять с использованием пункта File/Save All. Далее вам будет предложено сохранить форму (формы) с именем по умолчанию **Unit1** (**Unit2**, и т.д.), а затем проект с именем **Project1**. Имена, предлагаемые по умолчанию можно изменять по своему усмотрению, но никогда форма и проект не должны быть названы одинаково.

6. Нажмите клавишу **F9**, для запуска программы на выполнение.

После компиляции программы (выявленные компилятором ошибки в программе устраните самостоятельно) откроется окно с созданным Вами приложением. Проанализируйте работу данного приложения в соответствие с записанным ранее программным модулем.

Контрольные вопросы и задание к лабораторной работе №1

1. Как запустить Borland C++ Builder?
2. Как открыть новый проект (Windows-приложение)?
3. Как выбрать компонент и поместить его на форму?
4. Как выделить компонент (компоненты), переместить и изменить его (их) размеры?
5. Как выровнять компонент (компоненты) внутри формы?
6. Как запустить приложение на компиляцию?
7. Написать приложение, которое показывает работу светофора по следующему циклическому алгоритму:

«красный→красный + желтый→зеленый→мигающий зеленый»

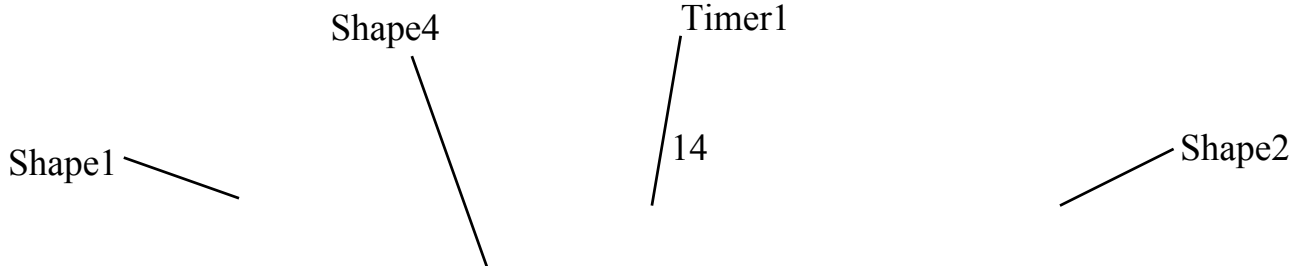
Лабораторная работа №2

Тема: «Первая программа на движение»

Цель работы: научиться создавать приложения, в которых компоненты перемещаются относительно друг друга.

Порядок работы: создание приложения «Поршень» начнем с того, что поместим на форму необходимые компоненты, так же как это показано на рис.9

1. Компоненты находятся во вкладках: **Shape** (Additional), **Button** (Standard), **Timer** (System).



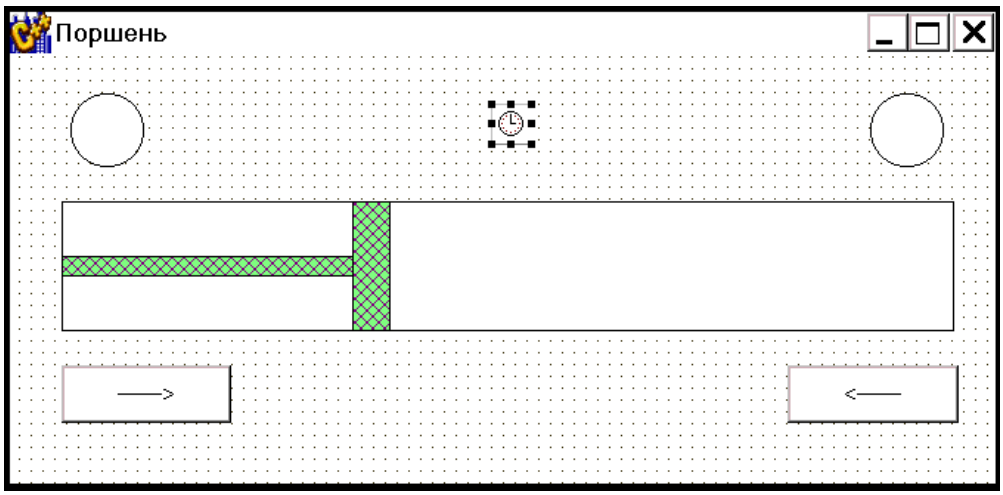


Рис.9.Пример заполнения формы приложение «Поршень»

2. Далее, при нажатии на кнопку → «поршень» (Shape4) должен двигаться вправо, а при нажатии на кнопку ← влево. Если он достигает крайне левого (или крайне правого) положения, загорается соответствующая сигнальная лампа и «поршень» останавливается.

3. Таймер в приложении будет использоваться как элемент повторения. Поэтому зададим для **Timer1** свойства **Interval** (интервал) равный 15 и **Enabled** (разрешение доступа к компоненту) как **false**.

4. Теперь щелкните на **Button1**, откроется редактор кода на событии **Button1Click**. Нужно записать следующее:

```
Timer1→Enabled=true;                    //включаем таймер
Timer1→OnTimer=Button1Click;            //задаем режим повторений
if (Shape1→Brush→Color= =clRed)
    Shape1→Brush→Color=clWhite;
if (Shape4→Left+Shape4→Width= =Shape3→Left+Shape3→Width)
    {
    Shape4→Left=Shape3→Left+Shape3→Width–Shape4→Width;
                                        //останавливаем поршень
    Shape2→Brush→Color=clGreen;         //включаем лампу
    Timer1→Enabled=false;               //выключаем таймер
    }
    else
    {
    Shape4→Left+=1;                     //перемещаем поршень на 1
    Shape5→Width+=1;}                   //вправо
```

5. Теперь щелкните на **Button2**, откроется редактор кода на событии **Button2Click**. Нужно записать следующее:

```
Timer1→Enabled=true;
Timer1→OnTimer=Button2Click;
```

```
if (Shape2→Brush→Color= =clGreen)
    Shape2→Brush→Color=clWhite;
if (Shape4→Left= =Shape3→Left)
    {
    Shape4→Left=Shape3→Left;
    Shape1→Brush→Color=clRed;
    Timer1→Enabled=false;
    }
    else
    {
    Shape4→Left–=1;                    //перемещаем поршень на 1 влево
    Shape5→Width–=1;
    }
```

6. Написаны два обработчика событий, которые будут выполняться всякий раз, когда выбирается соответствующая кнопка (**Button1** или **Button2**).

7. Запустите приложение на выполнение.

Контрольное задание к лабораторной работе №2.

Создать приложение «Автопогрузчик», примерный вид формы показан на рис.10. Для кнопок описать соответствующее движение:

«Вперед»	автопогрузчика с грузом до опоры
«Поднять»	груз с подъемной платформой поднять до верха опоры
«Переместить»	груз переместить на опору
«Опустить»	опустить подъемную платформу
«Назад»	автопогрузчик без груза откатывается на исходную позицию

В процессе разработки приложения предусмотреть следующие условия работы: груз не может быть перемещен, если автопогрузчик не подведен к опоре и подъемная платформа не находится на уровне верха опоры.

Для удобства работы, возможно, осуществлять совместные действия для группы компонентов. На этапе проектирования назначают дополнительное свойство **Tag** отличное от нуля, но одинаковое для каждого компонента группы. Например, если необходимо перемещать группу компонентов со свойством **Tag** = 1 влево, то запишем следующий код:

```
for (int i=0; i < ComponentCount; i++)
  {
  if (Components[i] → Tag == 1)
    ((TControl *)Components[i]) →Left +=1;
  }
```

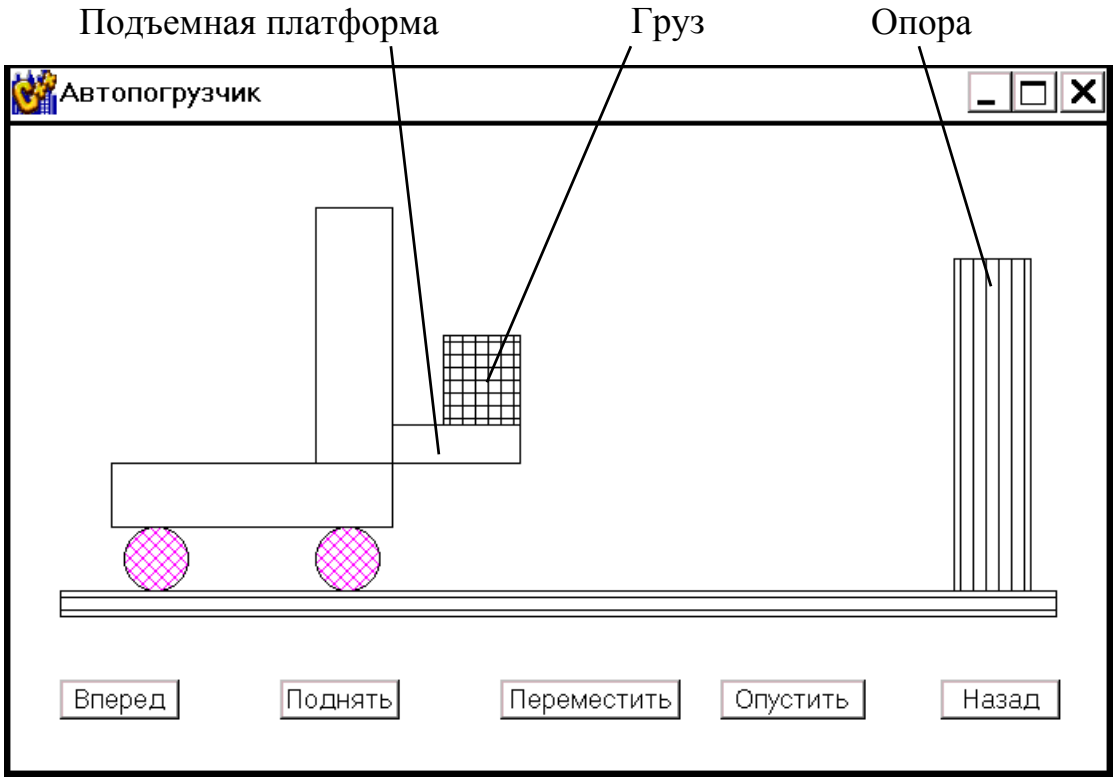


Рис.10.Пример заполнения формы приложение «Автопогрузчик»

Лабораторная работа №3

Тема: «Работа с диалоговыми окнами»

Цель работы: познакомиться с назначением форм в Borland C++ Builder и исследовать свойства диалоговых окон.

Порядок работы: при создании приложений используют, по крайней мере, одну форму, которая является главным окном приложения. При создании главного окна существует множество вариантов, определяемых требованиями к проекту. В частности, главное окно может иметь меню, панели с кнопками, растровые изображения и т.п.

Помимо формы главного окна, в приложении могут присутствовать формы диалоговых окон. Диалоговым окнам характерны следующие свойства:

- фиксированный размер;
- наличие кнопки **OK** для закрытия окна (некоторые диалоговые окна имеют кнопку **Close**, выполняющую аналогичную функцию);
- наличие кнопок **Cancel** и **Help**;
- наличие в строке заголовка только кнопки закрытия окна;
- наличие вкладок;

- использование клавиши **Tab** для перемещения между элементами управления диалогового окна.

Некоторые диалоговые окна могут удовлетворять рассмотренным свойствам не в полной мере.

Работу с диалоговыми окнами рассмотрим на примере создания приложения, имеющего диалоговое окно **About**, выводящее информацию об этом приложении. Выполним следующие действия.

1. Создайте новое приложение. Форма **Form1** будет являться главным окном создаваемого приложения.
2. Измените свойство **Caption** данной формы на "BMP Viewer". Обратите внимание на изменения в строке заголовка формы.
3. Поместите в форму компонент **Button**. Для поиска компонентов не прибегая к палитре компонентов можно использовать лист компонентов (Component List), вызываемый через команду главного меню **View**.
4. Расположите кнопку в правой верхней части формы. Измените свойство **Name** кнопки на **AboutButton**. В свойстве **Caption** запишите **About**. Данная кнопка будет служить для вызова диалогового окна **About**, выводящего информацию о приложении.
5. Создадим новую форму, которая будет служить основой для диалогового окна. Новая форма создается командой основного меню File/New Form. Измените имя новой формы на **AboutBox**. Выполните действия, позволяющие получить в строке заголовка данной формы надпись "About This Program" (разумеется, без кавычек).
6. Каждой форме ставится в соответствие своя вкладка редактора кода. Переключитесь в редактор кода. Обратите внимание, что редактор кода имеет две вкладки **Unit1.cpp** и **Unit2.cpp**, соответствующие двум открытым формам.
7. Сохраним наработанные данные. Выполним File/Save Project As…. В открывшимся окне сохранения, выберем требуемый каталог (каталог вашей группы в папке **Student**). Предлагается сохранить программу с именем **Unit1.cpp** изменим имя на **prog1.cpp** и сохраним. Следующее имя **Unit2.cpp** изменим на **about.cpp** и сохраним. Далее запрашивается разрешения на сохранение проекта с именем **Project1.bpr**. Изменим, название проекта на **Viewer.bpr** и сохраним.
8. Возвращаемся к редактору кода. Обратите внимание на изменения названий вкладок в этом редакторе. Попытайтесь объяснить изменения.
9. Сформируем диалоговое окно из формы **AboutBox**.

Изменим размеры окна. Высота окна задается свойством **Height** (задаем 150), ширина задается свойством **Width** (200). Размеры можно изменять и при помощи мыши, как рассматривалось ранее применительно к компонентам.

> ***Внимание!*** Уменьшенная в размерах форма легко может спрятаться под другую форму, создавая иллюзию исчезновения с экрана.

Для перехода между формами используйте кнопку **View Form** из панели инструментов главного окна Borland C++ Builder. Потренируйтесь!

10.Чтобы зафиксировать размер создаваемого окна, требуется изменить свойство **BorderStyle** формы на **bsSingle** или **bsDialog**. Последнее предпочтительнее, так как в строке заголовка диалогового окна будет находиться только одна кнопка закрытия окна.

11.Поместите в форму **AboutBox** один под другим четыре компонента **Label** из вкладки **Standard**. Данные компоненты обычно используются для вывода текстовой информации. Поместите в свойства **Caption** этих компонентов слова (без кавычек) "Программа", "просмотра", "растровых", "изображений". Обратите внимание на свойство **AutoSize** — оно отвечает за автоматическое изменение размера компонента и должно быть включено (**true**). Поместите компоненты с текстовой информацией желаемым образом, например, друг под другом в середине формы.

12.Поместите в форму компонент **BitBtn**. Данный компонент позволяет сформировать стандартную кнопку с рисунком и надписью. Поместите этот компонент по середине формы ближе к нижнему краю. Измените свойство **Kind** на **bkOk**.

13.Поместите в форму компонент **Bevel** из вкладки **Additional**. Для этого щелкните по соответствующему значку палитры компонентов. Значок "западет". Затем поместите курсор несколько выше и левее компонента **Label**, содержащего слово "Программа", нажмите (и держите) левую кнопку мыши и ведите курсор вниз и вправо, стремясь охватить появившейся рамкой весь текст. В нужном месте отпустите кнопку мыши. Размер компонента **Bevel** можно изменять стандартным образом.

14.Установите свойство **Shape** компонента **Bevel** в **bsFrame**.

15.Перейдите к форме **Form1**. Дважды щелкните по кнопке с надписью "About...". В раскрывшемся окне редактора кода введите следующий фрагмент программы:

```
AboutBox → ShowModal();
```

Данный код открывает форму **AboutBox** как модальное окно.

Модальное окно прерывает работу вызвавшего его приложения. Работа приложения может быть продолжена только после закрытия модального окна. **Диалоговое окно** может быть открыто как немодальное посредством программного кода:

```
AboutBox → Show();
```

16.Остается связать воедино два имеющихся программных модуля **prog1.cpp** и **about.cpp.** Перейдем в редактор кода на вкладку *prog1.cpp* и выполним следующие действия:

- команда меню File/Include Unit Hdr...;
- в раскрывшемся диалоговом окне **Include Unit** выбрать **About**.

17.Откомпилируйте и запустите приложение.

Контрольные вопросы и задание к лабораторной работе №3

1. Какое окно называется модальным (немодальным)?
2. Как открыть форму в виде модального (немодального) окна?

3. Как включить несколько форм в проект?

4. Создайте приложение, состоящее из двух форм. Причем, вторая форма должна открываться как немодальное окно при нажатии на кнопку. Сохранить проект в своем рабочем каталоге.

5. Исследуйте доступные свойства компонентов, используемых в приложении.

6. Создайте приложение, основная форма которого имеет кнопку, которая открывает новую форму. Новая форма содержит две кнопки, одна из которых закрывает эту форму, а другая открывает новую форму. Новая форма содержит две кнопки, одна из которых закрывает эту форму, а другая открывает новую форму и т.д. до четвертой формы, которая имеет только закрывающую кнопку. Всего получается четыре уровня.

Лабораторная работа №4

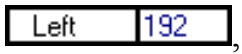
Тема: «Свойство — как элемент триады "свойство — метод — событие"».

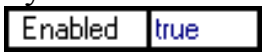
Цель работы: изучить различные свойства компонентов и форм. Создать приложение «Пятнашки».

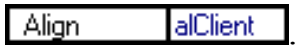
Порядок работы: рассмотрим основные свойства компонентов и варианты работы с ними.

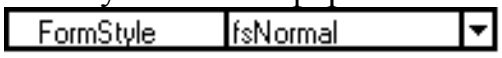
Быстрая разработка приложений (RAD) подразумевает поддержку свойств, методов и событий компонентов в рамках объектно-ориентированного программирования.

Свойства компонентов определяют их работу. Изменение свойств обычно приводит к выполнению некоторого двоичного кода. Некоторые свойства одинаковы для всех компонентов. Например, свойства **Left** и **Top** определяют положение компонента в форме, или формы на экране. Однако компоненты могут иметь свои индивидуальные свойства.

Свойства могут быть численными значениями 

, логическими переменными 

или строками символов 

. Значения некоторых свойств могут быть оформлены в виде перечислений (раскрывающегося списка) 

. Значения свойств могут меняться во время разработки программы и во время ее выполнения. Не все свойства способны производить видимые действия на этапе разработки программы, но когда это возможно, результат тут же отобразится в форме.

Свойства имеют два спецификатора доступа, которые используются при чтении и модификации их свойств. Это спецификатор чтения и спецификатор записи. Когда имеет место ссылка на свойство, то есть свойство используется справа от оператора присваивания, например

```
int a = Form1→Top;
```

то идет обращение к спецификатору чтения. Во многих случаях спецификатор чтения только возвращает текущее значение свойства.

Свойство может быть доступно только для чтения (read-only). В этом случае изменения значения свойства невозможно. Иногда свойство может быть доступно

только для записи (write-only). Некоторые свойства могут быть доступны только во время выполнения (runtime-only). Доступ к такому свойству невозможен во время разработки, и оно не отображается во вкладке "Свойства" инспектора объектов. Свойства свойств задаются разработчиком компонента.

Рассмотрим некоторые наиболее **важные** свойства.

1. Свойство **Name.**

При помещении компонента в форму C++Builder создает указатель на компонент и использует свойство **Name** в качестве имени переменной. Очевидно, что данное свойство широко используется при работе с компонентом. Например, при помещении в форму компонента **Edit**, свойство **Name** которого изменено на **MyEdit**, то C++ Builder поместит в заголовочный файл формы следующую запись

```
TEdit* MyEdit;
```

В дальнейшем указатель **MyEdit** может использоваться в программах для обращения к компоненту

```
MyEdit→Text ="Переменная равна =";
```

Для компонентов, помещаемых в форму, свойство **Name** устанавливается по умолчанию и может быть изменено с использованием инспектора объектов.

2. Свойство **Align.** Данное свойство определяет способ выравнивания компонента в окне. Возможные значения свойства приведены ниже.

alBottom	Компонент располагается вдоль нижнего края окна.
alClient	Компонент заполняет всю рабочую область окна.
alLeft	Компонент располагается вдоль левого края окна.
alNone	Компонент располагается произвольно.
alRight	Компонент располагается вдоль правого края окна.
alTop	Компонент располагается вдоль верхнего края окна.

Задание 1

Поместите в форму компонент **Panel**.

Найдите свойство **Align** для этого компонента и отметьте, какое значение присвоено по умолчанию.

Измените, значение данного свойства последовательно на **alTop**, **alLeft**, **alBottom**, **alRight** и отметьте результаты, обратив внимание на возможность изменения размеров компонента **Panel**.

Измените значение свойства на **alClient**. Попытайтесь изменить размер компонента.

---

3. Свойство **Color.** Свойство **Color** определяет цвет фона компонента. Значения свойства представлены в виде раскрывающегося списка. Первая группа в списке — от **clBlack** до **clWhite** — это предопределенные цвета C++Builder. Вторая группа цветов — системные цвета Windows. Рекомендуется в приложениях для стандартных элементов управления использовать цвета из второй группы.

Двукратный щелчок по свойству **Color** в столбце **Value** вызовет диалоговое окно выбора заказных цветов.

4. Свойство **Cursor.** Данное свойство определяет вид курсора мыши во время его перемещения через компонент. Данное свойство может меняться программно.

Задание 2

Поместите в форму двадцать один компонент **Button**. Расположите в виде матрицы 7×3, с одинаковым шагом по вертикали и по горизонтали. Для каждой кнопки измените свойство **Cursor** так, чтобы охватить все возможные значения этого свойства. Запустите приложение. Составьте таблицу, содержащую название значения свойства **Cursor**, краткой характеристики и рисунка курсора.

---

5. Свойство **Enabled.** Это свойство определяет возможность доступа к компоненту. Обычно разрешение/запрещение доступа к компоненту выполняется во время работы программы.

6. Свойство **Hint.** Свойство **Hint** (Подсказка) задает текст подсказки для компонента. Текст подсказки состоит из двух частей.

Первая часть называется короткой (ей соответствует функция **GetShortHint**) и выводится, когда пользователь задерживает курсор на компоненте. Чтобы выводить краткие подсказки, необходимо установить свойство **ShowHint** в "**true**".

Вторая часть, называемая длинной (**GetLongHint**), содержит текст, отображаемый в какой-то выделенной для этого части окна, например, в строке состояния. Тексты короткой и длинной подсказок отделяются друг от друга символом «|».

Для того, чтобы вторая часть сообщения, записанного в **Hint**, отображалась в строке состояния (или в каком-либо другом компоненте) в момент, когда курсор мыши проходит над компонентом, надо использовать обработку события **OnHint**.

Пример, создадим две подсказки для компонента **Button**:

- на форму поместите компоненты: **Button** (кнопка) и **StatusBar** (статусная строка);
- добавить в заголовочный файл формы (*.h) в раздел **public:** строку;

```
void __fastcall DisplayHint(TObject *Sender);
```

- в свойстве **Hint** для **Button** запишите;

```
Открыть файл | Открыть файл для редактирования.
```

- установите для кнопки **Button** свойство **ShowHint** в **true**;
- в обработчике события **FormCreate** запишите;

```
Application → OnHint = DisplayHint;
```

- во вкладке редактора кода создайте обработчик события **DisplayHint**;

```
void __fastcall TForm1::DisplayHint(TObject *Sender)
    {
    StatusBar1→SimpleText = GetLongHint(Application→Hint);
```

}

- запустите приложение на компиляцию.

Форма в приложении выступает как специфический компонент, который обладает специфическими свойствами, методами, событиями.

При желании в приложении 1 можно посмотреть основные свойства форм, доступные как на этапе разработки программы, так и на этапе ее выполнения.

Контрольные вопросы и задание к лабораторной работе №4

1. Изучите основные свойства компонентов вкладок **Standard**, **Additional**. Найдите компоненты близкие по назначению и разбейте их на группы. Какие свойства являются общими для группы? Какие уникальными?
2. Можно ли изменять свойство **Name** компонента во время выполнения программы?
3. Какое свойство используется для того, чтобы разрешить или запретить доступ к компоненту?
4. Как можно во время выполнения программы сообщить пользователю, что доступ к кнопке запрещен?
5. Как используется свойство **ModalResult** в компонентах кнопок?
6. Какой компонент обычно используется в качестве контейнера для других компонентов?
7. Написать приложение «Пятнашки». На основной форме расположите 16 компонентов, выровненных в виде матрицы 4×4. Компоненты должны быть различны по цвету и стилю заливки. Для каждого компонента написать два вида подсказок (например, «1|Кнопка 1»). Длинную подсказку выводить в заголовок формы.

Лабораторная работа №5

Тема: «Методы — как элемент триады “свойство – метод – событие”»

Цель работы: изучить различные методы компонентов и форм. Как пример создать приложение «Калькулятор».

Порядок работы: рассмотрим основные методы компонентов и варианты работы с ними.

Методы в компонентах VCL — это функции, которые могут быть вызваны для выполнения компонентом определенных действий. Методы могут принимать аргументы и возвращать значения. Но могут не принимать аргументов и не возвращать значений. Методы могут быть закрытыми (private), защищенными (protected) и открытыми (public). Пользователь компонента может вызвать только открытые методы. В отличие от свойств и событий, методы не имеют соответствующей вкладки в инспекторе объектов (Object Inspector). Методы вызываются при помощи оператора косвенного доступа (→).

Насчитывается более 20 методов, которые используются большинством компонентов. Компоненты оконного типа имеют более 40 общих методов. Однако не все из них используются широко. Многие функциональные возможности компонентов реализуются при помощи свойств. Например, чтобы скрыть

компонент, вы можете вызывать метод **Hide()** или установить для свойства **Visible** значение **false**. Кроме того, компоненты обычно имеют свои специальные методы, и именно эти методы для данного компонента используются чаще всего. Ниже в табл.2 приведены методы, часто используемые для создания приложений.

Таблица 2

Общие методы компонентов

Методы	**Описание**
Hide	Делает компонент невидимым.
Show	Делает видимым невидимый компонент.
Clear	Удаляет текст, изображение или очищает список.
SelectAll	Выделяет весь текст.
Undo	Отменяет результат последней операции редактирования.
FloatToStr	Преобразует вещественное число в строковую переменную.
StrToFloat	Преобразует строковую переменную в вещественное число.
Broadcast	Используется для отправки сообщений всем порожденным компонентам оконного типа.
ClientToScreen	Преобразует локальные координаты окна в экранные координаты.
ContainsControl	Возвращает **true**, если указанный компонент порожден компонентом или формой.
Invalidate	Запрашивает перерисовку компонента. Компонент будет перерисован Windows при первой же возможности.
Perform	Позволяет компоненту послать сообщение самому себе напрямую, а не через систему Windows.
Refresh	Запрашивает немедленную перерисовку компонента, стирая перед этим изображение компонента.
Repaint	Запрашивает немедленную перерисовку компонента. Стирание фона компонента перед этим не производится.

SetBounds	Позволяет вам задать свойства **Top**, **Left**, **Width** и **Height** одновременно. Это удобнее, чем определять их по отдельности.
SetFocus	Устанавливает фокус ввода в компоненте и делает этот компонент активным. Применяется только к компонентам оконного типа.
Update	Вызывает немедленную принудительную перерисовку компонента. Обычно для этого следует использовать методы **Refresh** и **Repaint**.

При построении приложения «Калькулятор» необходимо выполнить следующее;

- на первую форму поместить следующие компоненты: три компонента **Label**, два компонента **Edit**, компонент **Panel**, два компонента **Button**;
- жирным шрифтом сделать надписи в метках, так же как показано на рис.11а;

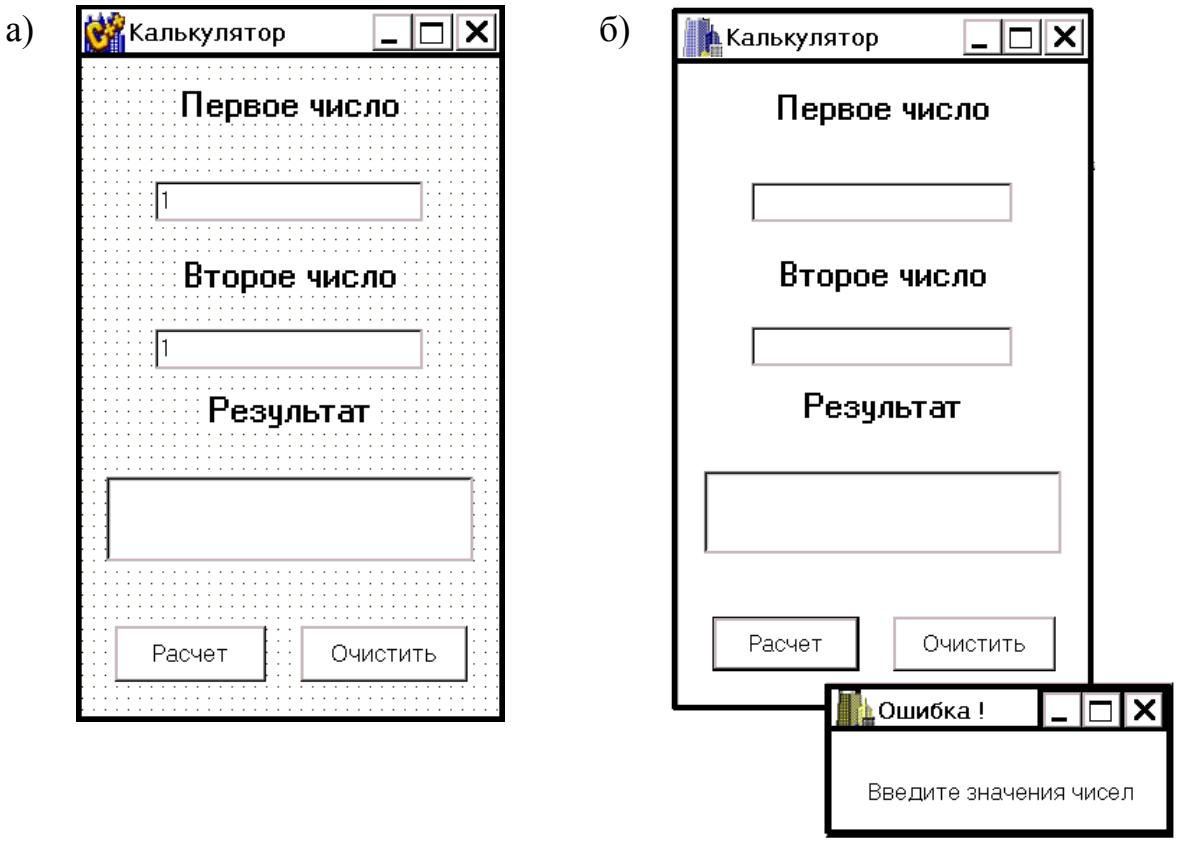


Рис.11.Окончательный вид формы (а) и окно во время выполнения при обнаружении ошибки ввода (б)

- для компонентов **Edit** и **Button** выполнить аналогичные действия самостоятельно;
- написать обработчик щелчка кнопки **Button1**;

При этом будем использовать два метода **FloatToStr()** и **StrToFloat()**.

```
if (Edit1→Text != “” || Edit2→Text != “”)
```

```
Panel1→Caption = Edit1→Text + “*” + Edit2→Text + “=”
+FloatToStr(StrToFloat(Edit1→Text)*StrToFloat(Edit2→Text));
else Form2→ShowModal();
```

- написать обработчик щелчка кнопки **Button2**;

```
Edit1→Clear();
Edit1→SetFocus();
Edit2→Clear();
Panel1→Caption = “”;
```

- создать вторую форму, в которой будет один компонент **Label**. Форма будет вызываться в случае обнаружения ошибки ввода данных. Для этого откройте редактор кода, а между { } вставьте следующую строку:

```
Label1→Caption = «Введите значение чисел»;
```

- подключите вторую форму к первой;
- приложение запустите на выполнение. Убедитесь, что оно работает нормально, иначе откроется окно ошибки (рис.11б).

Формы по своей сути также являются компонентами. А это значит, что они имеют много общих с компонентами методов. К этим методам, в частности, относятся **Show()**, **ShowModal()**, **Invalidate()**. Однако существуют методы, используемые только в формах. В приложении 2 приведены часто используемее методы форм, с которыми вы можете познакомиться самостоятельно.

Контрольные вопросы и задание к лабораторной работе №5

1. Изучите основные методы компонентов.
2. Назовите три метода, которые могут использоваться для принудительной перерисовки элементов управления?
3. Внести изменения в приложение «Калькулятор», чтобы было возможным производить выбор арифметической операции (+ , –, *, /).

Лабораторная работа №6

Тема: «События — как элемент триады “свойство – метод – событие”»

Цель работы: изучить различные события компонентов и форм. Создание приложения «Квадрат».

Порядок работы: прежде чем рассматривать основные свойства компонентов (или форм) нужно знать, что Windows является средой, управляемой событиями. К событиям относятся, например, перемещение мыши, щелчки кнопками мыши, нажатия кнопок на клавиатуре. Прикладная программа непрерывно опрашивает Windows на предмет возникновения событий. Windows уведомляет программу о событии, посылая соответствующее сообщение. Общее количество сообщений превышает 170. На практике используется несколько десятков.

Событие в Borland C++ Builder — это все происходящее в компоненте, о чем пользователь может захотеть узнать. Каждый компонент спроектирован так, чтобы реагировать на определенные события. Реакция на событие, относящееся к

данному компоненту, называется обработкой (handle) события. События обрабатываются функциями, которые называются **обработчиками событий**.

Обработка событий крайне проста. Список событий, на который реагирует компонент, приведен во вкладке «События» (**Event**) инспектора объектов. Имя события служит одновременно его описанием. Например, щелчок мышью называется **OnClick**.

Нет необходимости обрабатывать все события. Если какие-либо события не прописываются, то они либо игнорируются, либо обрабатываются предопределенным способом. Ниже в табл.3 приведены основные события компонентов, используемые при создании приложений.

Таблица 3

Основные события компонентов

Событие	**Описание**
OnChange	Событие возникает, когда в элементе управления происходят какие-либо изменения. Конкретная реализация зависит от компонента.
OnClick	Происходит при щелчке на компоненте любой кнопкой мыши.
OnDblClick	Происходит при двойном щелчке на компоненте.
OnEnter	Происходит, когда компонент оконного типа получает фокус.
OnExit	Происходит, когда компонент оконного типа теряет фокус при переключении на другой элемент управления. Событие не возникает, когда пользователь переключается между формами или приложениями
OnKeyDown	Происходит при нажатии пользователем клавиши, когда компонент находится в фокусе.
OnKeyPress	Происходит при нажатии одной из алфавитно-цифровых клавиш, клавиши **Tab**, **BackSpace**, **Enter** или **Esc**.
OnKeyUp	Происходит при отпускании клавиши.
OnMouseDown	Событие возникает при нажатии кнопки мыши на компоненте. Параметры, передаваемые обработчику события, содержат информацию о том, какой кнопкой мыши щелкнули и какие функциональные клавиши при этом были задействованы (**Alt**, **Ctrl** или **Shift**), а также координаты **X** и **Y** курсора мыши в момент щелчка.
OnMoveMouse	Происходит при перемещении курсора через объект управления
OnMouseUp	Событие возникает при отпускании кнопки мыши на компоненте, если перед этим кнопка была нажата на этом же компоненте.

Обратите внимание, что при выполнении некоторых действий может возникать несколько событий.

Перечисленные выше события характерны не для всех компонентов. Например, компонент **Shape** имеет (помимо прочих) только три последних события, из перечисленных выше.

Обработка события программистом состоит в написании программного кода, отражающего реакцию разрабатываемого приложения на возникшее событие.

Для перехода в редактор кода с целью написания кода обработки события необходимо:

- сделать активным тот компонент, помещенный в форму, событие для которого предполагается обработать;
- перейти на вкладку **Events** (События) инспектора объектов;
- найти требуемое событие;
- дважды щелкнуть в правой колонке напротив названия выбранного события;
- записать код обработки события в открывшемся окне редактора кода.

При выполнении этих действий в редакторе кода создается функция, которая будет вызываться при обработке события. Имя этой функции появится в правой колонке вкладки **Events** инспектора объектов напротив названия события.

Функцию, ранее созданную для обработки одного события, можно использовать для обработки другого события. В правой колонке вкладки **Events** инспектора объектов напротив названия события может находиться раскрывающийся список имеющихся обработчиков событий, доступных для данного события.

Формы имеют специфические **события форм,** некоторые из которых описаны в приложении 3.

Задание

Имеется белое квадратное поле в нижней части окна, и желтое поле такого же размера с цифрой 1 в верхней части окна. Требуется при помощи мыши перетаскивать желтое поле в белое. Если при отпускании левой кнопки мыши желтое поле точно попадает в белое, оно там фиксируется. В противном случае оно возвращается к своему исходному месту.

Листинг программы, реализующей решение данной задачу, приведен ниже.

```
……..
int x0, y0, xn, yn;
bool flag1 = false;
Tform1 *Form1;
__fastcall TformExample1::TformExample1(Tcomponent* Owner) : Tform(Owner)
{
   xn = Label1→Left;
   yn = Label1→Top;
}
void __fastcall TformExample1::Label1MouseDown (Tobject*Sender, TmouseButton
Button, TshiftState Shift, int X, int Y)
{
   flag1 = true;
   x0 = X;
   y0 = Y;
```

```
}
void __fastcall TformExample1::Label1MouseMove (Tobject*Sender, TshiftState
Shift, int X, int Y)
{
  if (flag1)
  {
    Label1→Left += X–x0;
    Label1→Top += Y–y0;
  }
}
void __fastcall TformExample1::Label1MouseUp (Tobject*Sender, TmouseButton
Button, TshiftState Shift, int X, int Y)
{
  if (abs(Label1→Left – Shape1→Left) < 10 &&
    abs(Label1→Top – Shape1→Top) < 10)
  {
    Label1→Left=Shape1→Left;
    Label1→Top=Shape1→Top;
  }
    else
    {
      Label1→Left=xn;
      Label1→Top=yn;
    }
    flag1 = false;
}
```

Контрольные вопросы и задание к лабораторной работе №6

1. Проанализируйте программу. Постарайтесь ответить на следующие вопросы:

- какое имя (свойство **Name**) имеет главная форма программы;
- какие компоненты использовались для создания приложения;
- обработчики, каких событий и для, каких компонентов используются в программе;
- что обозначает строка:

void __fastcall*TformExample1::Label1MouseDown(Tobject*Sender, TmouseButton Button, TshiftState Shift, int X, int Y);

- для чего используется переменная **flag1**;
- что делает следующий фрагмент программы:

---

* Модификатор __fastcall используется при задании функции, параметры которой размещаются в регистрах процессора. Данный модификатор при ответе не комментировать.

```
if (flag1)
{
   Label1→Left += X–x0;
   Label1→Top += Y–y0;
}
```

2. После ответа на вопросы создайте приложение, решающее рассматриваемую задачу, и проверьте его работоспособность.
3. Перечислите события, возникающие при однократном и двойном щелчке кнопкой мыши по компоненту.
4. Перечислите события клавиатуры и события перемещения фокуса ввода.

Лабораторная работа №7

Тема: «Стандартные диалоговые окна на примере окон работы с файлами»

Цель работы: изучить компоненты системного диалога. Создание приложения «Текстовый редактор».

Порядок работы: рассмотрим, как в Borland C++ Builder реализованы стандартные режимы диалога. Windows содержит целый ряд стандартных диалоговых окон, которые могут быть доступны любой прикладной программе.

В табл.4 перечислены стандартные окна Windows и функции, выполняемые окнами и компоненты C++ Builder, обеспечивающие работу с этими окнами. Стандартные диалоговые окна находятся на вкладке **Dialog** панели компонентов. Данные компоненты относятся к невизуальным.

Таблица 4

Основные события компонентов

Пиктог-рамма	**Компонент**	**Описание**	**Окна Windows**
	OperDialog	Открытие файла	File Open
	SaveDialog	Сохранение файла	File Save
	OperPictureDialog	Открытие файла с графикой	

Продолжение таблицы 4

	SavePictureDialog	Сохранение файла с графикой	
	FontDialog	Настройка шрифта	Font
	ColorDialog	Выбор цвета	Color
	PrintDialog	Настройка печати	Print
	PrinterSetup	Установки принтера	Print Setup
	FindDialog	Поиск	Find
	ReplaceDialog	Замена	Replace

Для создания и отображения всех стандартных окон используется метод **Execute**(). Диалоговое окно отображается в модальном режиме, за исключением немодальных окон **Find** и **Replace**. Метод **Execute**() возвращает **true**, если пользователь нажал кнопку «**OK**», дважды щелкнул по имени файла или нажал **Enter** на клавиатуре. Если пользователь нажал кнопку «**Cancel**», нажал клавишу **Esc** или закрыл кнопкой закрытия окна, то метод возвращает **false**. Стандартное диалоговое окно может быть реализовано в программе, например, следующим образом.

```
if (OpenDialog→Execute())
   {
   Memo→Lines→LoadFromFile(OpenDialog→FileName);
   }
```

Данный код выводит диалоговое окно **FileOpen**, в котором пользователь выбирает нужный файл. Если файл выбран, то будет выполнен код внутри **if**, в результате чего содержимое файла будет загружено в компонент **Memo**.

Диалоговые окна для работы с файлами достаточно просты, но имеют свои специфические свойства, которые приведены в табл.5.

Таблица 5

Некоторые свойства диалоговых компонентов

Свойство	**Описание**
DefaultExt	Свойство определяет расширение, которое будет присвоено файлу по умолчанию.
FileName	Содержит имя файла, которое выбирает пользователь. Если установить это свойство заранее, то при вызове диалогового окна оно будет содержать имя файла.
Files	Содержит список выбранных файлов, если разрешено групповое выделение. Доступно только для чтения.

Окончание таблицы 5

Filter	Содержит список типов файлов, которые пользователь может выбирать. При разработке программы фильтры создаются при помощи окна **Filter Editor**. Для вызова этого окна необходимо дважды щелкнуть в колонке **Value** рядом со свойством **Filter**. Столбец **Filter Name** содержит текстовое описание типа файла, например, **Text files**. В столбце **Filter** указывается маска, которая будет использоваться при отборе файлов, например *.txt.
InitialDir	Определяет каталог, который будет использоваться в качестве начального при открытии окна. Если свойство не определено, то в окне будет открыт текущий каталог.
Title	Используется для задания и чтения заголовка диалогового окна.

С диалоговыми окнами **File** не связаны никакие события.

> ***Внимание!*** Компоненты **OpenDialog** и **SaveDialog** оперируют только с именем файла. Реализация каких-либо действий в отношении этого файла целиком зависит от программиста.

Задание

Создать приложение — простой текстовый редактор, способный открывать существующий текстовый файл, создавать новый файл, сохранять файл.

Создание данного приложения опирается на знаниях, полученных ранее. Выполнение описанных ранее операции приводиться не будет. Одновременно рассмотрим неизученные ранее компоненты и приемы работы с ними.

1. Создайте новое приложение.
2. Переименуйте форму в **MyNotepad**.
3. Сделайте так, чтобы строка заголовка формы содержала надпись **MyNotepad v1.0**.
4. Сохраните приложение в личном каталоге под именем **MyNotepad**. Имя файла формы программы задайте **Notepad.cpp**. В процессе работы над приложением периодически сохраняйте результаты при помощи команды File/Save All.
5. В последующем (при создании приложений) может потребоваться разделитель (—), линия, отделяющая одни элементы приложения от других. Создадим, для примера, горизонтальный разделитель. Поместите в форму компонент **Bevel**, задайте его высоту равной 2 и сделайте так, чтобы он располагался по верхней границе формы на всю ее ширину.
6. В приложения кнопки обычно находятся на панелях. Поэтому создадим панель с требуемыми нам кнопками "**New**", "**Open**", "**Save**". Поместите в форму компонент **Panel** из палитры компонентов. Переименуйте ее в **ToolBar**.
7. Сделайте высоту панели равной 32. Свойство **BevelOuter** установите в **bvNone**. Обратите внимание на изменения, происшедшие с панелью.

8. Сотрите значение свойства **Caption**.

9. Измените свойство **Align** на **alTop**. Обратите внимание на взаимное расположение панели и разделителя.

10. В качестве кнопок используем компонент **SpeedButton**, специально предназначенный для установки в панель. Поместите компонент **SpeedButton** в панель (не в форму!). Не беспокойтесь о точном положении кнопки, выравнивание осуществим позднее.

11. Переименуйте кнопку в **FileOpenBtn**.

12. Установите свойство **Left** кнопки равным 5.

13. С помощью палитры выравнивания отцентрируйте кнопку в панели по вертикали.

14. На кнопку, реализованную через компонент **SpeedButton** (или через **BitBtn**) может быть помещен графический объект — значок, характеризующий функцию кнопки. Дважды щелкните в колонке **Value** напротив свойства **Glyph** компонента **SpeedButton**. Откроется окно редактора изображений. При помощи соответствующих кнопок перейдите в каталог …\Builder\Borland Shares\Images\Buttons и выберите файл Fileopen.bmp **(**указать точный путь невозможно, так как он различен для разных версий Borland C++ Builder**)**. Если все действия были выполнены правильно, на кнопке появится графическое изображение.

15. Аналогичным образом добавьте в панель кнопки **FileSaveBtn** и **FileNewBtn**. Разместите их "без зазора" по горизонтали.

16. На все кнопки создайте короткие всплывающие подсказки, аналогичные тем которые используются в Windows.

17. Добавьте в форму компонент **Memo**. Данный компонент предназначен для отображения текстовой информации, а также для ее набора. Информация определяется свойством **Lines** компонента.

18. Добейтесь того, чтобы компонент **Memo** занимал все оставшееся поле формы.

19. Дважды щелкните в колонке **Value** инспектора объектов напротив свойства **Lines**. В открывшемся окне удалите надпись **Memo1**.

20. Разрешите использование в компоненте **Memo** линеек прокрутки (**ScrollBar**).

21. Добавьте в форму, место не имеет значения, компоненты **OpenDialog** и **SaveDialog**. У данных компонентов настройте фильтр на работу с текстовыми (*.txt) и всеми файлами (*.*).

22. Форма должна иметь вид такой же как показано на рис.12.

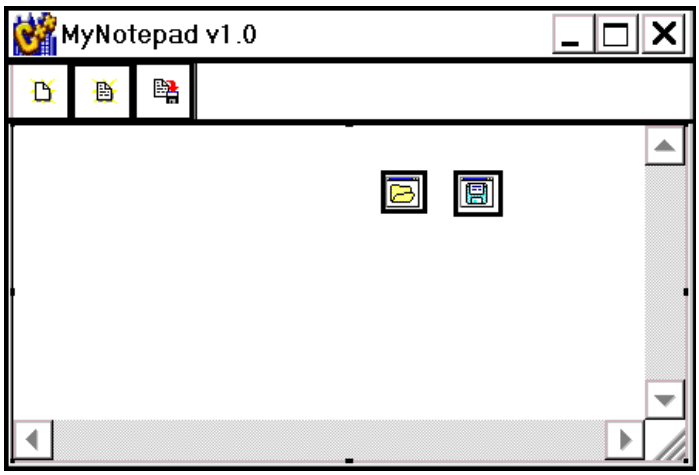


Рис.12.Вид формы приложения
«Текстовый редактор»

23.Ниже приведен листинг программы, реализующей создание нового файла, считывание и сохранение файла. Найдите функции — обработчики события щелчка по соответствующей кнопке панели Вашего приложения. Реализуйте эти функции у себя.

```
TMyNotepad *MyNotepad;
//_____________________________________________________________
__fastcall TMyNotepad :: TMyNotepad(TComponent*Owner)
      : TForm(Owner)
{
}
//_____________________________________________________________
void __fastcall TmyNotepad :: FileOpenBtnClick(TObject*Sender)
{
  if (OpenDialog1→Execute())
  {
    if (Memo1→Modified)
    {
    int res=Application→MessageBox("The current file has changed. Save  changes?",
    "MyNotepad Message", MB_YESNOCANCEL | MB_ICONWARNING);
    if (res == IDYES) FileSaveBtnClick(Sender);
    if (res == IDCANCEL) return;
    }
    Memo1→Lines→LoadFromFile(OpenDialog1→FileName);
  }
}
//_____________________________________________________________
void __fastcall TMyNotepad :: FileNewBtnClick(TObject*Sender)
{
  if (Memo1→Modified)
  {
```

```
      int res=Application→MessageBox("The current file has changed. Save  changes?",
      "MyNotepad Message", MB_YESNOCANCEL | MB_ICONWARNING);
      if (res == IDYES) FileSaveBtnClick(Sender);
      if (res == IDCANCEL) return;
   }
   if (Memo1→Lines→Count > 0) Memo1→Clear();
   SaveDialog1→FileName = " ";
}
//_____________________________________________________________________
void __fastcall TmyNotepad :: FileSaveBtnClick(TObject *Sender)
{
if (SaveDialog1→FileName != " ")
   {
   Memo1→Lines→SaveToFile(SaveDialog1→FileName);
   Memo1→Modified = false;
   }
   else
      {
      SaveDialog1→Title="Save As";
      if (SaveDialog1→Execute())
         {
         Memo1→Lines→SaveToFile(SaveDialog1→FileName);
         Memo1→Modified=false;
         }
      }
}
```

- Запустите приложение.

Контрольные вопросы и задание к лабораторной работе №7

1. Внесите необходимые изменения в созданную программу, чтобы все предупреждения выдавались на русском языке.
2. Создайте приложение для просмотра и сохранения графических файлов.

Лабораторная работа №8

Тема: «Создание меню приложения»

Цель работы: научиться строить падающее меню. Добавить меню в созданное ранее приложение «Текстовый редактор».

Порядок работы: рассмотрим, как в C++ Builder реализует возможность создания падающего меню.

Меню являются важной частью большинства приложений Windows. В некоторых программах меню отсутствует, но таких программ очень мало.

В C++ Builder **для создания меню** используется специальный редактор меню, что сильно облегчает работу. Редактор меню обладает следующими возможностями:

- создание, как главного, так и контекстных меню;
- обеспечение оперативного доступа к редактору кода для обработки событий **OnClick**, соответствующих пунктам меню;
- вставка меню из шаблонов или файлов ресурсов;
- сохранение пользовательских меню в виде шаблонов.

Все команды редактора меню доступны через его контекстное меню или из инспектора объектов. Редактор меню позволяет быстро разрабатывать меню любой структуры. Главное меню представлено компонентом **MainMenu**, которому соответствует класс TMainMenu. Каждый пункт меню, в свою очередь, является компонентом **MenuItem**, инкапсулированным в классе TMenuItem. Пользователю не требуется знать, как функционируют эти классы, поскольку редактор меню берет на себя всю работу по созданию меню. Рассмотрим все этапы создания падающего меню.

1. Создайте новое приложение и поместите в него компонент **MainMenu**. Измените имя этого компонента на **MainMenu**.
2. В верхней части формы поместите горизонтальный разделитель.
3. Дважды щелкните по значку главного меню. Появится редактор меню.

Хотя существует более легкий способ создания меню **File**, первый раз проделаем это вручную. Редактор меню всегда содержит пустой пункт меню, который обозначает место для вновь создаваемого пункта. При первом запуске редактора меню пустой пункт будет выделен.

4. Измените значение свойства **Name** на **FileMenu**.
5. Щелкните на свойстве **Caption** в окне инспектора объектов, наберите **&File** и нажмите **Enter**.

Амперсант (&) используется для ввода подчеркнутого символа в название пункта меню. Подчеркнутый символ служит для ускоренного выбора пункта меню с клавиатуры путем нажатия соответствующей клавиши в сочетании с **Alt**. Амперсант можно размещать в любом месте текста.

После проведенных операций в редакторе меню появится пункт **File** (он появился и в главной форме, закрытой редактором меню), под только что созданным пунктом **File** появился новый пустой пункт, и такой же пустой пункт

возник справа. Инспектор объектов содержит пустой компонент **MenuItem**, ожидая ввода значений для свойств **Caption** и **Name**.

Продолжим создание меню.

6. Смените значение свойства **Name** нового пункта на **FileNew**.

7. Измените, значение свойства **Caption** на **&New** и нажмите **Enter**. Редактор меню снова создаст пустой пункт.

8. Повторите шаги 1 и 2 для создания пунктов меню **Open**..., **Save** и **Save As**...

9. Теперь нам нужен разделитель меню. **Разделитель** (separator) — это горизонтальная линия в меню, разделяющая группы пунктов. Добавление разделителя с помощью редактора меню C++Builder выглядит очень просто. Нужно только установить для свойства **Caption** значение «–» (дефис). Перейдите к пустому пункту меню под **Save As**, введите дефис в поле **Caption** и нажмите **Enter**.

10. Добавьте в меню пункты **Exit**.

11. Для создания меню **Edit** рассмотрим более легкий путь. Сначала щелкните на пустом пункте справа от пункта **File**. Затем щелкните правой кнопкой мыши и выберите в открывшемся контекстном меню пункт **Insert From Template**. На экране появится диалоговое окно **Insert Template**. В диалоговом окне приведен список доступных шаблонов. Есть возможность использовать предопределенные шаблоны или создавать свои собственные. Сейчас нам нужно загрузить меню **Edit**, поэтому выберите из списка **Edit Menu** и нажмите кнопку «**ОК**». В редактор меню сразу же будет вставлено полное меню **Edit**. На самом деле оно содержит больше пунктов, чем нужно. Удалением лишнего займемся чуть позже.

12. Давайте заодно вставим в меню **Help**. Щелкните на пустом пункте справа от **Edit**. Снова выберите в контекстном меню пункт **Insert From Template** и вставьте шаблон **Help Menu**. (Не выберите по ошибке **Expanded Help Menu**.). Обратите внимание, что вновь вставленные пункты уже отображены в главной форме.

13. Меню **Edit**, вставленное ранее, содержит слишком много пунктов. Некоторые из них никогда не будут использоваться в программе. Для того чтобы удалить пункт меню нужно:

- выбрать из меню **Edit** пункт **Repeat**<command>;
- нажать клавишу **Delete** или выбрать пункт **Delete** в контекстном меню редактора, выделенный пункт исчезнет, а оставшиеся пункты сдвинутся вверх;
- аналогичным образом удалить пункты **Paste Special**, **Go To**, **Links**, **Objects** и разделитель после **Replace**;
- в меню **Help** оставьте только **Contents** и **About...**

Вставка пунктов меню выполняется очень просто. Щелкнуть мышью на том пункте, над которым нужно поместить новый пункт, и нажать клавишу **Insert** (или выбрать **Insert** в контекстном меню редактора). После этого можно установить свойства **Caption** и **Name** для нового пункта.

14. Давайте вставим еще один пункт в меню **Edit**:

- вызовите из меню **Edit** пункт **Find**;

- нажмите клавишу **Insert** (в меню возникнет новый пункт, а все лежащие под ним пункты сдвинутся вниз);
- измените значение свойства **Name** на **EditSelectAll**, а свойства **Caption** — на **Select&All**;
- в пустом пункте в нижней части меню **Edit** поставьте разделитель (для этого достаточно ввести в поле **Caption** дефис);
- снова щелкните на пустом пункте и установите для свойства **Name** значение **EditWordWrap**, а для свойства **Caption — &WordWrap**.

Перемещение пунктов меню можете проделать очень легко. При этом, возможно, перемещать пункты как в пределах одного всплывающего меню, так и между различными меню. Для перемещения пунктов существуют два способа. Первый заключается в использовании буфера обмена. Операции **Cut** и **Paste** выполняются как обычно, поэтому не будем описывать их. Другой способ заключается в использовании техники drag-and-drop. При необходимости размещения пункта **SelectAll** под пунктом **Undo** проделайте следующее:

- откройте пункт **Edit** для вызова соответствующего меню;
- щелкните на пункте **SelectAll** и перетащите его, пока разделитель под **Undo** не окажется выделен;
- отпустите кнопку мыши, и пункт займет новое место.

Создание подменю не имеет каких-либо особенностей. **Подменю** (submenu) — это меню, которое вызывается при выборе пункта вышестоящего меню. Наличие подменю обозначается стрелкой вправо, расположенной за названием пункта. Вы можете создать подменю, выбрав в контекстном меню редактора пункт **Create Submenu**, или нажав одновременно **Ctrl** и стрелку вправо на клавиатуре. При создании подменю, справа от соответствующего пункта основного меню размещается пустой пункт. Вы можете добавлять в подменю пункты точно так же, как и при создании главного меню. Создать подменю можно также вставкой шаблона.

Добавить сочетания клавиш, соответствующие пунктам меню, можно изменив значение свойства **ShortCut** в инспекторе объектов. Для меню **Edit**, которое было вставлено из шаблона, сочетания клавиш уже определены. Например, пункту **Cut** соответствует сочетание **Ctrl+X**. Если вы посмотрите на меню **Edit**, то увидите, что за названием пункта стоит **Ctrl+X**. Щелкнув на пункте **Cut**, вы увидите, что свойство **ShortCut** в окне инспектора объектов также имеет значение **Ctrl+X**. Щелкните в столбце **Value** рядом со свойством **ShortCut**. Справа вы увидите кнопку со стрелкой. Щелкните на ней для вызова списка возможных значений. Этот список содержит практически все сочетания, которые могут когда-либо понадобиться. Чтобы установить новое сочетание клавиш, просто выберите его из списка.

Стандартным сочетанием клавиш для **SelectAll** является **Ctrl+A**, поэтому давайте установим его для этого пункта меню:

- выберите в созданном меню пункт **Edit | SelectAll**;
- щелкните на свойстве **ShortCut** в окне инспектора объектов;

- выберите **Ctrl+A** из списка доступных сочетаний, и теперь за названием пункта **SelectAll** появится **Ctrl+A**.

Это все, что вам нужно сделать — остальное берет на себя C++Builder. Для использования сочетаний клавиш не требуется писать какой-либо код.

Давайте сделаем еще несколько действий, направленных на "облагораживание" созданного меню.

Во-первых, по умолчанию включим пункт **WordWrap**. Этот пункт предназначен для включения и выключения режима заворачивания слов. Когда режим заворачивания включен, пункт **WordWrap** отмечен. Щелкните на пункте **WordWrap** и установите для свойства **Checked** значение **true**. При этом рядом с названием пункта появится отметка, сообщающая, о его включении.

Во-вторых, необходимо установить значения свойств **Name** для всех пунктов, вставленных из шаблона. Они имеют имена по умолчанию, которые желательно заменить на более информативные. Выберите пункт **Edit | Undo**. Измените значение свойства **Name** с **Undo1** на **EditUndo**. Можно использовать любое соглашение об именах, но будьте при этом последовательны. Повторите процедуру для пунктов **Cut**, **Copy**, **Paste**, **Find** и **Replace**. Потом перейдите к меню **Help** и измените значения свойств **Name** для пунктов **Contents** и **About** на **HelpContents** и **HelpAbout**.

На этом создание меню завершено. Просмотрите его еще раз и устраните замеченные ошибки. Убедившись, что меню построено правильно, закройте окно редактора меню.

15. Переименуйте главную форму приложения в **TextViewer**. Дополните созданное приложение панелью, содержащей кнопки **New**, **OpenFile**, **SaveFile**, **Cut**, **Copy**, **Paste**. Кнопки должны содержать соответствующие графические значки и имена. Обработчики событий для этих кнопок не писать!

16. Написать обработчик события для команды меню **Help | About...**, выводящий модальное окно **AboutBox** с информацией о приложении.

Далее приводится текст программы, который содержит все необходимые обработчики событий.

```
//———————————————————————————————————————
#include <vcl.h>
#pragma hdrstop
#include "SPMain.h"
//———————————————————————————————————————
#pragma resource "*.dfm"
TScratchPad *ScratchPad;
//———————————————————————————————————————
__fastcall TScratchPad :: TScratchPad(TComponent*Owner)
      : TForm(Owner)
{
}
//———————————————————————————————————————
void__fastcall TScratchPad :: FileNewClick(TObject*Sender)
```

```
{
//Открытие нового файла. Перед этим
//проверка необходимости сохранения текущего файла
if(Memo→Modified)
{
//Отображение окна сообщения
int result = Application→MessageBox(
"The current file has changed. Save changes?",
"Scratchpad Message", MB_YESNOCANCEL | MB_ICONWARNING);
//Если нажата кнопка Yes, сохранение текущего файла
if(result == IDYES) FileSaveClick(Sender);
//Если нажата кнопка No, текущий файл не сохраняется
if(result == IDCANCEL) return;
}
//Стирание строк в компоненте Memo, если они присутствуют
if(Memo→Lines→Count > 0) Memo→Clear();
//Установка для свойства FileName диалога SaveDialog
//значения "пустая строка".
//Это будет означать, что файл еще не был сохранен.
SaveDialog→FileName = " ";
}
//---------------------------------------------------------
void__fastcall TScratchPad :: FileOpenClick(TObject*Sender)
{
//Открытие файла. Проверка необходимости сохранения
//текущего файла (логика работы аналогична FileNewClick())
if(Memo→Modified)
{int result = Application→MessageBox(
"The current file has changed. Save changes?",
"Scratchpad Message", MB_YESNOCANCEL | MB_ICONWARNING);
if(result == IDYES) FileSaveClick();
if(result == IDCANCEL) return;
}
//Вызов диалога File Open. Если нажата кнопка OK,
//открытие файла с помощью метода LoadFromFile().
//Перед этим очистка свойства FileName
OpenDialog→FileName = " ";
if (OpenDialog→Execute())
{
if(Memo→Lines→Count > 0) Memo→Clear();
Memo→Lines→LoadFromFile(OpenDialog→FileName);
SaveDialog→FileName = OpenDialog→FileName;
}
```

```
}
//---------------------------------------------------------------------
void __fastcall TScratchPad :: FileSaveClick(TObject*Sender)
{
//Если имя файла уже было введено, нет необходимости
//вызывать диалог FileSave. Достаточно сохранить файл
//с помощью SaveToFile()
if(SaveDiaiog→FileName != " ")
{
Memo→Lines→SaveToFile(SaveDialog→FileName);
//Установка Modified в false, т.к. файл уже сохранен
Memo→Modified = false;
}
//Если имя файла не было указано, выполнение SaveAs()
else FileSaveAsClick(Sender) ;
}
//---------------------------------------------------------------------
void__fastcall TScratchPad :: FileSaveAsClick(TObject*Sender)
{
//Вызов диалога FileSave для сохранения файла.
//Установка Modified в false, т.к. файл уже сохранен
SaveDiaiog→Title = "SaveAs";
if(SaveDialog→Execute())
{
Memo→Lines→SaveToFile(SaveDialog→FileName);
Memo→Modified = false;
}
}
//---------------------------------------------------------------------
void__fastcall TScratchPad :: FileExitClick(TObject*Sender)
{
//Работа завершена. Закрытие формы.
Close();
}
//---------------------------------------------------------------------
void__fastcall TScratchPad :: EditUndoClick (TObject*Sender)
{
//TMemo не содержит метода Undo, поэтому нужно послать
//сообщение Windows WM_UNDO для компонента Memo
SendMessage(Memo→Handle, WM_UNDO, 0, 0);
}
//---------------------------------------------------------------------
void__fastcall TScratchPad :: EditSelectAllClick(TObject*Sender)
{
```

```
//Вызов TMemo::SelectAll()
Memo→SelectAll();
}
//———————————————————————————————————————
void_fastcall TScratchPad :: EditCutClick(TObject*Sender)
{
//Вызов TMemo::CutToClipboard()
Memo→CutToClipboard() ;
}
//———————————————————————————————————————
void __fastcall TScratchPad :: EditCopyClick(TObject*Sender)
{
//Вызов TMemo::CopyToClipboard()
Memo→CopyToClipboard();
}
//———————————————————————————————————————
void_fastcall TScratchPad :: EditPasteClick(TObject*Sender)
{
//Вызов TMemo::PasteFromClipboard().
Memo→PasteFromClipboard();
}
//———————————————————————————————————————
void_fastcall TScratchPad :: EdltWordWrapClick(TObject*Sender)
{
//Переключение свойства TMemo::Wordwrap. Установка такого же
//значения для свойства Checked пункта меню Word Wrap
Memo→WordWrap = !Memo→WordWrap;
EditWordWrap→Checked = Memo→WordWrap;
//Если режим Wordwrap включен, достаточно только вертикальной
//линейки прокрутки; в противном случае требуются обе линейки
if (Memo→WordWrap) Memo→ScrollBars = ssVertical;
else Memo→ScrollBars = ssBoth;
}
//———————————————————————————————————————
```

Контрольные вопросы и задание к лабораторной работе №8

1. Что означает многоточие, следующее за названием пункта меню?
2. Какими двумя способами можно переместить пункт меню?
3. Как сопоставить пункту меню определенное сочетание клавиш?
4. Как изначально запретить пункт меню?
5. Свяжите с кнопками **New**, **OpenFile**, **SaveFile**, **Cut**, **Copy**, **Paste** обработчики событий для соответствующих пунктов меню.
6. Добавьте сочетание **Ctrl+S** для пункта меню **File | Save**.

Лабораторная работа №9

Тема: «Графика в Borland C++ Builder»

Цель работы: познакомиться с графическими возможностями C++ Builder.

Порядок работы: в Windows используется термин контекст устройства (device context) — это что-то вроде грифельной доски, на которой могут рисовать программы (иначе холст — **Canvas**). Контекст устройства может использоваться для рисования на различных поверхностях:

- на рабочей области или рамке окна;
- на рабочем столе;
- в памяти;
- на принтере или других устройствах вывода.

C++Builder предоставляет нам класс **TCanvas**, чтобы облегчить работу с контекстами устройств. Класс **TCanvas** имеет много свойств и методов (см. табл.6 и табл.7). Рассмотрим некоторые из них по мере работы над материалом сегодняшнего дня.

Таблица 6

Основные свойства **TCanvas**

Свойства	**Описание**
Brush	Цвет кисти или шаблона, используемого для заполнения фигур.
ClipRect	Текущий ограничивающий прямоугольник холста. Любое изображение будет обрезано на границе этого прямоугольника. Свойство предназначено только для чтения.
CopyMode	Определяет способ наложения изображения (нормальный, с инверсией, и т.д.)
Font	Шрифт, который будет использоваться для рисования текста.
Handle	Дескриптор (HOC) холста. Используется при прямых вызовах API Windows.
Pen	Определяет стиль и цвет линий, рисуемых на холсте.
PenPos	Текущая позиция рисования в координатах X и Y.
Pixels	Массив пикселов холста.

Таблица 7

Основные методы **TCanvas**

Метод	**Описание**
Arc	Рисует дугу на холсте, используя текущее перо.
BrushCopy	Отображает растр с прозрачным фоном.
CopyRect	Копирует часть изображения на холст.
Draw	Копирует изображение из памяти на холст.
Ellipse	Используя текущее перо, рисует на холсте эллипс, закрашенный текущей кистью.
FloodFill	Закрашивает область холста, используя текущую кисть.
LineTo	Проводит линию от текущей позиции до точки, заданной параметрами **X** и **Y**.
MoveTo	Устанавливает текущую позицию рисования.
Pie	Рисует на холсте сектор.
Polygon	Рисует на холсте многоугольник по массиву точек и закрашивает его текущей кистью.
Polyline	Используя текущее перо, рисует на холсте ломаную линию по массиву точек. Линия не замыкается автоматически.
Rectangle	Рисует на холсте прямоугольник, обведенный текущим пером и закрашенный текущей кистью.
RoundRect	Рисует закрашенный прямоугольник со скругленными углами.
StretchDraw	Копирует растровое изображение из памяти на холст. Изображение будет растянуто или сжато в зависимости от размеров области назначения.
TextExtent	Возвращает ширину и высоту (в пикселях) строки, переданной через параметр **Text**. При вычислении ширины используется текущий шрифт холста.
TextHeight	Возвращает высоту (в пикселях) строки, переданной через параметр **Text**. При вычислении ширины используется текущий шрифт холста.
TextOut	Рисует в указанном месте холста текст, используя текущий шрифт.
TextRect	Рисует текст в пределах ограничивающего прямоугольника.

Эти свойства и методы представляют лишь малую часть функциональных возможностей контекста устройства Windows. Тем не менее, они используются в 80 процентах задач, которые возникают при работе с графикой.

Следующий код написанный для компонента **Image1** прочертит диагональную линию из верхнего левого в правый нижний угол, рисует эллипс и закрашивает его синим цветом.

```
Image1 → Canvas → Pen → Color = clRed;
Image1 → Canvas → Brush → Color = clBlue;
Image1 → Canvas → Ellipse(20,20,200,100);
Image1 → Canvas → MoveTo(0,0);
Image1 → Canvas → LineTo(ClientRect.Right, ClientRect.Bottom);
```

Перед тем, как более подробно рассматривать класс **TCanvas**, поговорим о графических объектах, используемых в программировании для Windows.

Интерфейс графических устройств (GDI) Windows содержит много типов объектов, определяющих функционирование холста. Наиболее широко используемыми объектами являются перья, кисти и шрифты.

Перо представляет объект, который используется для рисования линий. Линия может быть как отдельной (от одной точки до другой), так и контуром прямоугольника, эллипса или многоугольника. Перо доступно через свойство **Pen** класса **TCanvas**. В табл.8 перечислены свойства **TPen**.

Таблица 8

Свойства **TPen**

Свойства	**Описание**
Color	Устанавливает цвет линии.
Handle	Дескриптор пера (HPEN) Используется при прямых вызовах GDI.
Mode	Определяет способ рисования линии (нормальный с инверсией и т.д.)
Style	Стиль линии. Линия может быть сплошной, пунктирной, штрихпунктирной, нулевой и т.д.
Width	Ширина пера в пикселях.

В большинстве случаев эти свойства используются так, как можно ожидать. В следующем примере рисуется красная пунктирная линия:

```
Image1 → Canvas → Pen → Color = clRed;
Image1 → Canvas → Pen → Style = psDash;
Image1 → Canvas → MoveTo(20, 20);
Image1 → Canvas → LineTo(120, 200);
```

Пунктирные линии, как штриховые (dashed), так и точечные (dotted), могут рисоваться только пером с толщиной 1. Стиль **psClear** используется в том случае, когда контур объекта (прямоугольника, эллипса и закрашенного многоугольника) должен быть невидимым.

Кисть представляет закрашенную область графической фигуры. При изображении замкнутой области (эллипс, прямоугольник или многоугольник) она будет закрашена текущей кистью. Обычно под кистью подразумевается сплошной цвет, хотя это не всегда так. Кисть может включать шаблон или растр. Класс

TCanvas имеет свойство с названием **Brush**, которое может быть использовано для управления кистью. Свойства **TBrush** перечислены в табл.9.

Таблица 9

Свойства **TBrush**

Свойства	Описание
Bitmap	Растровое изображение, которое будет использоваться в качестве фона кисти. Для Windows95 размер растра не должен превышать 8×8 пикселов.
Color	Устанавливает цвет линии.
Handle	Дескриптор кисти (HBRUSH). Используется при прямых вызовах GDI.
Style	Стиль кисти. Кисть может быть сплошной, прозрачной или одним из шаблонов.

По умолчанию для свойства **Style** установлено значение **bsSolid**. Если использовать структурную заливку, то установите для свойства **Style** один из стилей шаблонов (bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross или bsDiagCross).

В следующем примере рисуется круг, заштрихованный под углом 45 градусов.

```
Canvas → Brush → Color = clBlue;
Canvas → Brush → Style = bsDiagCross;
Canvas → Ellipse(20, 20, 220, 220);
```

При использовании шаблонов свойство **Color** определяет цвет линий, которые составляют шаблон. При этом автоматически устанавливается прозрачный фон. Это означает, что цвет фона кисти будет таким же, как и у окна, содержащего нарисованную фигуру. Чтобы задать цвет фона, придется обойти VCL и обратиться напрямую к API. Вот как будет выглядеть код, если захотите заштриховать эллипс, вписанный в компонент **Image1** (красная штриховку на белом фоне):

```
TCanvas *pCanvas = Image1 → Canvas;
pCanvas → Brush → Color = clRed;
pCanvas → Brush → Style = bsDiagCross;
pCanvas → Ellipse(0, 0, Imagel → Width, Imagel → Height);
```

Теперь цвет фона кисти будет белым.

Другое интересное свойство кистей — возможность использования растровых изображений в качестве фона.

```
Imagel → Canvas → Brush → Bitmap = new Graphics :: TBitmap;
Imagel → Canvas → Brush → Bitmap → LoadFromFile("abort.bmp");
Imagel → Canvas → Ellipse(20, 20, 220, 220);
delete Imagel → Canvas → Brush → Bitmap;
```

В первой строке этого фрагмента кода создается объект **TBitmap**, который затем присваивается свойству **Bitmap** кисти. Свойство **Bitmap** не устанавливается

по умолчанию, поэтому нужно специально создать объект **TBitmap** и присвоить его свойству **Bitmap**. Во второй строке из файла загружается растр, который должен иметь размер 8x8 пикселов. Можно использовать изображения большего размера, но они будут усечены до размера 8x8. В третьей строке рисуется круг. После того, как круг нарисован, свойство **Brush** стирается. Очистка свойства **Brush** требуется потому, что Builder в данном случае не будет делать это автоматически. Если не очистить свойство **Brush**, программа вызовет утечку памяти.

Иногда потребуется пустая кисть. Через пустую (или прозрачную) кисть виден фон. Чтобы создать пустую кисть, установите для свойства **Style** значение **bsClear**. Давайте вернемся к предыдущему примеру и нарисуем второй круг внутри первого, используя, пустую кисть.

```
Canvas → Pen → Width = 1;
Canvas → Brush → Bitmap = new Graphics :: TBitmap;
Canvas → Brush → Bitmap → LoadFromFile("about.bmp");
Canvas → Ellipse (20, 20, 220, 220);
Canvas → Brush → Style = bsClear;
Canvas → Pen → Width = 5;
Canvas → Ellipse (70, 70, 170, 170);
delete Canvas → Brush → Bitmap;
```

Растровые изображения и палитры в большинстве случаев используются совместно. Растровые объекты в C++Builder инкапсулированы в классе **TBitmap**. С его помощью загрузка и вывод изображений выполняются очень просто.

Палитры — это один из тех аспектов программирования в Windows, которые вызывают наибольшую путаницу. Чаще всего управление палитрой берет на себя объект **TBitmap**.

Начните новое приложение и введите следующий код для обработчика события **OnPaint**. Обратите внимание на правильность введенного пути к файлу HANDSHAK.BMP (он должен находиться в каталоге Common Files/Borland Shared/Images/Splash/256Color).

```
Graphics :: TBitmap* bitmap = new Graphics :: TBitmap;
//bitmap → IgnorePalette = true;
bitmap → LoadFromFile("handshak.bmp");
Canvas → Draw(0, 0, bitmap);
delete bitmap;
```

Данный фрагмент кода выводит в форму растровое изображение, находящиеся в файле handshak.bmp. Если включить в программу закомментированную строку, которая содержит указание игнорировать информацию о палитре при выводе изображения, то все цвета на рисунке станут неправильными. Это потому, что палитра не применялась. Использование палитры гарантирует, что все цвета изображения будут корректно отображены на системную палитру.

Класс **TCanvas** имеет несколько методов для отображения растров. Из них наиболее часто используется метод **Draw**(), который был рассмотрен ранее. Он просто выводит растровое изображение на холст в указанной позиции.

Метод **StretchDraw()** используется при необходимости изменения размера изображения. Укажите прямоугольник для размещения изображения и графический файл. Если прямоугольник больше, чем исходный размер изображения, оно будет растянуто. Если прямоугольник меньше, изображение будет сжато.

StretchDraw() не принимает никаких попыток сохранить пропорции изображения. Обеспечение исходного отношения ширины к высоте полностью зависит от вас.

```
Graphics :: TBitmap *bitmap = new Graphics :: TBitmap;
bitmap → LoadFromFile("handshak.bmp");
TRect rect = Rect(0, 0, 100, 100):
Canvas → StretchDraw(rect, bitmap);
delete bitmap;
```

Ограничивающие области — это участки экрана, которые используются для указания фрагментов холста, на которых будет выполняться рисование. Класс **TCanvas** имеет свойство **ClipRect**, но оно доступно только для чтения. Для изменения ограничивающей области необходимо использовать API Windows. Давайте возьмем предыдущий пример и немного изменим его, чтобы проиллюстрировать работу ограничивающих областей.

```
Graphics :: TBitmap *bitmap = new Graphics :: TBitmap;
bitmap → LoadFromFile("handshak.bmp");
HRGN hRgn = CreateRectRgn(50, 50, 250, 250);
SelectClipRgn (Imagel → Canvas → Handle, hRgn);
Imagel → Canvas → Draw(0, 0, bitmap);
delete bitmap;
```

На этот раз при запуске программы видно, что отображается только часть картинки. Функция **SelectClipRgn()** устанавливает в качестве ограничивающей области прямоугольник с координатами 50, 50, 250, 250. Растровое изображение выводится в том же месте, что и раньше, но теперь видна только его часть. Все, что находится вне ограничивающей области, игнорируется.

Шрифты не являются для нас чем-то новым. Шрифты, используемые классом **TCanvas**, не отличаются от шрифтов, используемых формами или другими компонентами. Свойство **Font** класса **TCanvas** аналогично свойству **Font** любого другого компонента. Чтобы изменить шрифт для холста, достаточно сделать следующее:

```
Canvas →Font → Name = "Courier New";
Canvas → Font → Size = 14;
Canvas → Font → Style = Canvas → Font → Style << fsBold;
Imagel → Canvas → TextOut(20. 20, "Testing");
```

Для вывода текста на холст используют два метода **TextOut()** и **TextRet()**. **TextOut()** является основным способом. Методу передаются координаты **X** и **Y**, а также текст, который должен быть отображен.

```
Canvas → TextOut(20. 20; "1234567");
```

Указанная строка выводится в позиции 20, 20. Координаты **X** и **Y** указывают верхний левый угол текста, а не базовую линию. Метод **TextOut()** используйте при отображении текста, для которого не требуется точное позиционирование.

Метод **TextRet()** позволяет в дополнение к самому тексту указать ограничивающий прямоугольник. Этот метод следует использовать, если текст должен выводится внутри некоторой области. Текст, который выходит за границу указанной области, будет отсечен.

Следующий фрагмент кода гарантирует отображение не более 100 пикселов текста:

```
Imagel → Canvas → TextRect( Rect(20, 50, 120, 70),
20, 50,"This is a very long line that might get clipped.");
```

Как **TextOut**(), так и **TextRect()** могут отображать только одну строку текста. Никакого заворачивания не выполняется.

Фон текста определяется текущей кистью (белой по умолчанию). Чтобы убрать этот нежелательный эффект, нужно сделать одно из двух: либо изменить цвет кисти холста, либо сделать фон текста прозрачным. Какой цвет использовать для этого?

В данном случае цвет фона может совпадать с цветом формы, поэтому сделаем следующее:

```
Imagel → Canvas → Brush → Color = Color;
```

Это годится для большинства ситуаций, но иногда нужен больший контроль. Было бы проще сделать фон текста прозрачным.

```
TBrushStyle oldStyle:
oldStyle = Canvas → Brush → Style;
Canvas → Brush → Style = bsClear;
Canvas → TextOut(20. 20, "This is a test.");
Canvas → Brush → Style = oldStyle;
```

Сначала сохраняем текущий стиль кисти. После этого устанавливаем прозрачную кисть (**bsClear**). Закончив вывод текста, возвращаем кисти тот стиль, который она имела раньше.

Использование прозрачного фона имеет и другое преимущество. Допустим, нужно отобразить некоторый текст поверх растрового фона.

Контрольные вопросы и задание к лабораторной работе №9

1. Какое свойство **TCanvas** определяет цвет закрашивания холста?
2. Какое свойство **TCanvas** определяет цвет и вид линий, нарисованных на холсте?
3. Что делает ограничивающая область?
4. Могут ли ограничивающие области иметь произвольную форму, или они обязательно должны быть прямоугольными?
5. Как сохранить скрытое изображение в файле?
6. Создайте приложение, которое помещает на форму растровое изображение. На него выведите несколько строк текста, изменяя варианты шрифтов, фона, цвета, и нарисуйте различные геометрические фигуры, изменяя их цвет и заливку.
7. Для ранее сформулированного задания измените программу так, чтобы она сохраняла изображение из памяти на диске.

ПРИЛОЖЕНИЯ

Приложение 1

Таблица 10

Свойства форм

Свойства	**Описание**
Свойства, доступные во время разработки и выполнения программы	
ActiveControl	Используется для выбора элемента управления, который будет активным при обращении к форме. Во время выполнения столбец значений этого свойства содержит список всех компонентов формы. Выбранный компонент будет активным при первом вызове формы.
AutoScroll, HorzScrollBar, VertScrollBar	Управляют линейками прокрутки для формы.
BorderStyle	Определяет тип рамки вокруг формы. Значением по умолчанию является **bsSizeable**, что соответствует окну переменного размера. Для фиксации размера окна используют значения **bsDialog** и **bsNone**.
ClientWidth, ClientHeight	Указывает ширину и высоту рабочей области формы. Рабочая область — это область ниже строки заголовка и меню. Изменение значений этих свойств автоматически изменяет значения свойств **Width** и **Height**.
Font	Задает шрифт, который по умолчанию будет использоваться всеми компонентами формы.
Icon	Определяет значок, который будет отображаться в строке заголовка формы при выполнении программы, а также для свернутой формы. В некоторых случаях, например для диалоговых окон, значение свойства игнорируется.
Position	Определяет размер и положение формы на экране при первом отображении. Свойство имеет три варианта: • **poDesigned** — форма отображается в том же месте, где находилась при разработке; • **poDefault** — размер и положение формы устанавливает Windows; • **poScreenCenter** — форма помещается в центр экрана.
Visible	Определяет, будет ли видима форма, изначальна.
WindowState	Свойство может быть считано для определения текущего состояния формы (развернута, свернута, имеет нормальный размер), а также может использоваться для начальной установки размера формы.

Окончание таблицы 10

Свойства, доступные только во время выполнения программы	
Canvas	Представляет поверхность формы, доступную для рисования. Используя это свойство, можно отображать в форме растровые рисунки, линии и т.п.
ClientRect	Содержит координаты рабочей области формы. Удобно использовать, если необходимо узнать ширину и высоту рабочей области формы для размещения в ее центре растрового изображения.
ModalResult	Используется для закрытия модального окна. Если у вас есть диалоговое окно с кнопками «**ОК**» и «**Cancel**», вы можете присвоить **ModalResult** значение **mrOK** при нажатии кнопки «**ОК**» или значение **mrCancel** при нажатии кнопки «**Cancel**». Вызвавшая окно форма затем может считать **ModalResult** для определения того, какая кнопка была нажата. К другим возможным значениям относятся **mrYes**, **mrNo** и **mrAbort**.
Owner	Представляет собой указатель на владельца формы. Владелец формы — это объект, отвечающий за ее уничтожение после использования. С другой стороны, объектом, порождающим компонент, является окно (форма или другой компонент), которое содержит данный компонент. В случае главной формы приложение является одновременно и владельцем, и порождающим объектом формы. Для остальных компонентов владельцем может быть форма, но порождающим объектом должен быть другой компонент, например, панель.
Parent	Представляет собой указатель на порождающий объект формы. Различие между **Owner** и **Parent** объяснено выше, в разделе “Owner”.

Приложение 2

Таблица 11

Методы форм

Методы	**Описание**
BringToFront	Располагает данную форму поверх всех остальных форм приложения.
Close	Закрывает форму.
Print	Выводит на печать содержимое формы. Печатается только рабочая область формы, без подписи, строки заголовка или рамок. Метод удобен для быстрой распечатки формы.
ScrollInView	Прокручивает форму, пока указанный компонент не окажется в поле зрения.
SetFocus	Активизирует форму и перемещает ее поверх других окон.
Show ShowModal	Отображает форму как немодальную (**Show**) или модальную (**ShowModal**).

Приложение 3

Таблица 12

События форм

События	**Описание**
OnActivate	Активизирует форму. Форма может быть активизирована в результате ее первоначального создания, при переключении от одной формы к другой или при переключении между приложениями.
OnClose	Возникает при закрытии приложения.
OnCreate	Происходит при первоначальном создании формы. Для каждого экземпляра отдельной формы будет вызвано только одно событие **OnCreate**. Обработчик **OnCreate** можно использовать для выполнения инициализирующих действий.
OnDestroy	Это событие противоположно **OnCreate**, используется для освобождения динамически выделенной памяти или других завершающих действий.
OnDragDrop	Происходит при помещении объекта в форму. Его можно использовать, если ваша форма поддерживает технику drag-and-drop.
OnMouseDown, OnMouseMove, OnMouseUp	Используются для реакции на щелчки мышью на форме.
OnPaint	Происходит тогда, когда форма по какой-либо причине требует перерисовки. Используйте это событие для вывода изображений, которые должны постоянно присутствовать на экране. В большинстве случаев отдельные компоненты самостоятельно обновляют изображения, но иногда вам может потребоваться рисовать на самой форме.
OnResize	Возникает при каждом изменении размера формы. Его можно использовать для выравнивания объектов в форме или для перерисовки формы.
OnShow	Происходит перед тем, как форма станет видимой. Вы можете использовать это событие для выполнения операций, которые требуются непосредственно перед отображением формы.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Кент Рейсдорф. Borland C++ Builder 3. Освой самостоятельно / Пер. с англ. — М.: ЗАО Издательство БИНОМ, 1999. — 736 с.

2. А.Я. Архангельский. Программирование в C++ Builder 5. — М.: ЗАО Издательство БИНОМ, 2001. — 975 с.

3. Шамис В.А. C++ Builder 5. Техника визуального программирования. — М.: Издательство Нолидж, 1998. — 507 с.

4. Теллес М. Borland C++Builder: Библиотека программиста. — СПб.: ПитерКом, 1998. — 512 с.

ОГЛАВЛЕНИЕ