

11.19 Дерево элементов (TTreeView)	236
11.20 Список элементов (TListView).....	240
11.21 Простейший файловый менеджер.	242
11.22 Улучшенный файловый менеджер (С возможностью запуска файлов)	250
11.23 Подсказки для чайников (TStatusBar)	252
11.24 Панель инструментов (TToolBar и TControlBar).	254
11.25 Перемещаемые панели и меню в стиле MS (Docking).	258

11.19 Дерево элементов (TTreeView)

Сейчас нам предстоит познакомиться с достаточно сложным, но мощным компонентом – дерево элементов (*TreeView*). Любая более менее большая программа обязательно использует этот компонент потому что он очень удобен.



- **TreeView**

Давай сразу напишем пример и познакомимся с деревом на практике. Компонент *TreeView* достаточно сложный и с ним нужно знакомиться на практике, чтобы увидеть все прелести работы с ним.

Создай новый проект и брось на него два компонента: *TreeView* и *ImageList*. В список картинок *ImageList* засунь пару любых картинок, потом они пригодятся. А пока добавь ещё три кнопки (*TButton*):

1. Добавить (в свойстве Name укажи AddButton).
2. Добавить элемент (в свойстве Name укажи AddChildButton).
3. Удалить (в свойстве Name укажи DelButton).
4. Изменить заголовок (в свойстве Name укажи EditButton).

В результате у тебя должна быть форма приблизительно такого вида, как показано у меня на рисунке 11.19.1.

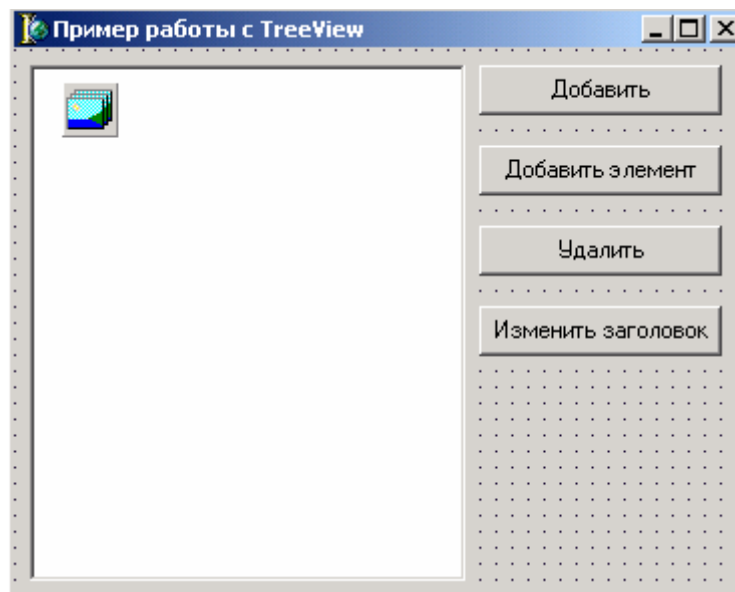


Рисунок 11.19.1. Форма будущей программы

Теперь нужно указать у нашего дерева в свойстве *Images* установленный набор картинок. После этого можно переходить к кодировке.

По нажатию кнопки «Добавить» мы должны добавлять в дерево новый элемент. Для этого напишем следующий код:

```
procedure TTreeViewForm.AddButtonClick(Sender: TObject);
var
  CaptionStr:String;
```

```
NewNode:TTreeNode;  
begin  
CaptionStr:="";  
if not InputQuery('Ввод имени', 'Введите заголовок элемента',CaptionStr) then exit;  
  
NewNode:=TreeView1.Items.Add(TreeView1.Selected, CaptionStr);  
if NewNode.Parent<>nil then  
NewNode.ImageIndex:=1;  
end;
```

Здесь я объявил две переменные: CaptionStr типа строка String и NewNode типа TTreeNode. Тип TTreeNode – это тип отдельного элемента дерева элементов.

В первой строчке кода я обнуляю строку CaptionStr. Эта строка в будущем будет использоваться для хранения имени будущего элемента дерева.

Вторая строка имеет следующий код:

```
if not InputQuery('Ввод имени', 'Введите заголовок элемента', CaptionStr) then  
exit;
```

Здесь выполняется функция *InputQuery*, которая используется для вывода на экран окна ввода. У этой функции есть три параметра:

1. Заголовок окна ввода.
2. Текст-пояснение, которое подсказывает пользователю, что ему надо вводить.
3. Строковая переменная, в которой мы передаём значение по умолчанию и получаем результат ввода. Если перед вызовом записать в эту переменную какое-нибудь значение, то оно будет использоваться в качестве значения по умолчанию. Но после вызова этой функции этот параметр всегда хранит реально введённое пользователем значение.

На рисунке 11.19.2 ты можешь увидеть это окно ввода.

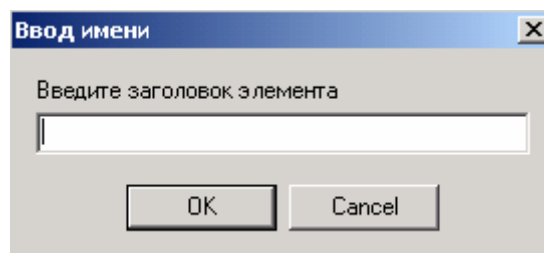


Рисунок 11.19.2. Окно ввода

Если окно было закрыто не кнопкой ОК, то происходит выход из процедуры. Об этом говорит наша конструкция:

```
if not InputQuery(...) then  
exit;
```

Следующая строка кода добавляет новый элемент в наше дерево:

```
NewNode:=TreeView1.Items.Add(TreeView1.Selected, CaptionStr);
```

У компонента *TreeView1* есть свойство *Items* в котором хранятся все элементы дерева. Это свойство имеет объектный тип **TTreeNode**. Чтобы добавить туда новый

элемент, нужно вызвать метод *Add* объекта *Items*. Получается, что в объекте *TreeView1* есть ещё один объект *Items* в котором хранятся все элементы. Мы уже сталкивались с такими случаями, когда внутри одного объекта был другой объект.

У метода *Add* есть два параметра:

1. Элемент, к которому надо добавить новый. Здесь я передаю выделенный элемент (*TreeView1.Selected*).
2. Заголовок нового элемента.

Результат выполнения этого метода – указатель на новый элемент. Этот результат мы сохраняем в переменной *TreeNode*. Теперь мы можем изменять и другие значения этого элемента. Например, как в следующем коде я буду изменять картинку:

```
if NewNode.Parent<>nil then  
    NewNode.ImageIndex:=1;
```

Здесь идёт проверка, если свойство *Parent* нашего дерева не равно нулю (т.е. компонент не является верхним в дереве), то изменить значение *ImageIndex* созданного нами элемента на 1 (по умолчанию это значение 0).

По нажатию кнопки «Добавить элемент» пишем следующий код:

```
var  
    CaptionStr:String;  
    NewNode:TTreeNode;  
begin  
    CaptionStr:='';  
    if not InputQuery('Ввод имени подэлемента',  
        'Введите заголовок подэлемента',CaptionStr) then exit;  
  
    NewNode:=TreeView1.Items.AddChild(TreeView1.Selected, CaptionStr);  
    if NewNode.Parent<>nil then  
        NewNode.ImageIndex:=1;
```

Здесь код практически тот же, что и для кнопки «Добавить». Единственная разница в том, что при добавлении нового элемента мы используем метод *AddChild*. Отличие этого метода от просто *Add* заключается в том, что он добавляет дочерний элемент. Например, если ты выделил в списке какой-то элемент и передал его в качестве первого параметра в *AddChild*, то новый элемент будет как бы подчиняться выделенному. При использовании метода *Add* новый элемент будет находиться на одном уровне дерева с переданным в качестве первого параметра.

Теперь напишем код для кнопки «Добавить»:

```
if TreeView1.Selected<>nil then  
    TreeView1.Items.Delete(TreeView1.Selected);
```

Здесь нужно удалить выделенный элемент, поэтому сначала я проверяю, есть ли вообще выделенный элемент в дереве:

```
if TreeView1.Selected<>nil then
```

Если такой элемент есть, то выполниться следующий код:

```
TreeView1.Items.Delete(TreeView1.Selected);
```

Здесь мы используем метод *Delete* объекта *Items*, чтобы удалить элемент дерева. В качестве параметра надо передать элемент, который мы хотим удалить (я передаю выделенный *TreeView1.Selected*).

Для кнопки «Изменить заголовок» мы напишем следующий код:

```
procedure TTreeViewForm.EditButtonClick(Sender: TObject);
var
  CaptionStr:String;
begin
  CaptionStr:="";
  if not InputQuery('Ввод имени',
    'Введите заголовок элемента',CaptionStr) then exit;

  TreeView1.Selected.Text:=CaptionStr;
end;
```

Здесь снова я вызываю окно *InputQuery*, чтобы пользователь смог ввести новое имя для выделенного элемента. Теперь, чтобы изменить имя надо изменить свойство *Text* для выделенного элемента: *TreeView1.Selected.Text*.

Давай теперь сделаем возможность сохранения и загрузки данных в наше дерево. Для этого создай обработчик события *OnClose* и напишем в нём следующее:

```
procedure TTreeViewForm.FormClose(Sender: TObject;
var Action: TCloseAction);
begin
  TreeView1.SaveToFile(ExtractFilePath(Application.ExeName)+'tree.dat');
end;
```

Для сохранения дерева нужно вызвать метод *SaveToFile* и в качестве единственного параметра указать имя файла. В качестве имени файла можно указывать что угодно, но я предпочитаю использовать полный путь, чтобы файл случайно не создавался в каком-нибудь другом месте. Для этого я использую следующую конструкцию:

```
ExtractFilePath(Application.ExeName)+'tree.dat'
```

Application.ExeName – указывает на имя запущенного файла.

ExtractFilePath – вытаскивает путь к файлу из указанного в качестве параметра пути к файлу. Получается, что вызов *ExtractFilePath(Application.ExeName)* вернёт путь к директории, откуда была запущена прога. Остаётся только к этому пути прибавить имя файла (у меня это *'tree.dat'*) и можно быть уверенным, что файл обязательно будет находиться там же, где и запускной файл.

Теперь нужно загрузить сохранённые данные. Для этого по событию *OnShow* напишем следующее:

```
procedure TTreeViewForm.FormShow(Sender: TObject);
```

```
begin
if FileExists(ExtractFilePath(Application.ExeName)+'tree.dat') then
TreeView1.LoadFromFile(ExtractFilePath(Application.ExeName)+'tree.dat');
end;
```

Здесь я сначала проверяю с помощью вызова функции *FileExists* существование файла. Этой функции нужно передать полное имя файла и если такой файл существует, то функция вернёт **true** иначе **false**.

Если файл существует, то можно его загрузить с помощью вызова метода *LoadFromFile*.

Обязательно проверяй файл на существование. Если его нет или кто-то его удалил, а ты попытаешься загрузить данные из несуществующего файла, то произойдёт ошибка.

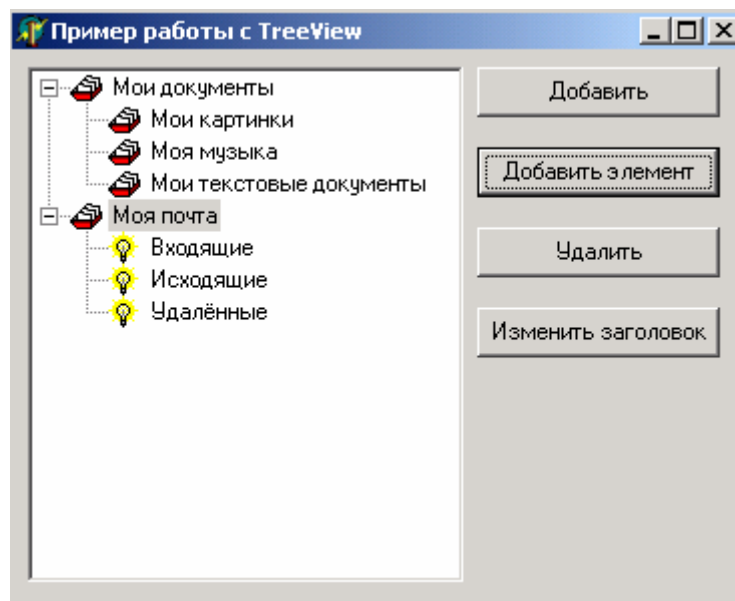



Рисунок 11.19.3. Результат работы программы

На рисунке 11.19.3 показано окно моего примера. Уже на рисунке ты можешь заметить, что дочерние элементы пункта «Моя почта» имеют другой рисунок, а дочерние элементы «Мои документы» нет. Почему так получилось? Ведь я же всегда меняю рисунок, если это дочерний элемент? Просто после закрытия программы дерево сохраняется в файле, а после загрузки оно загружается без учёта изображений. Состояние картинок не сохраняется и об этом нам нужно заботиться самостоятельно. Но это уже отдельная история, о которой мы поговорим в другой раз.

 На компакт диске, в директории \Примеры\Глава 11\TreeView ты можешь увидеть пример этой программы.

11.20 Список элементов (TListView)

Следующий компонент тоже достаточно сильно распространён. Попробуй запустить «Проводник» Windows и оглянись. Слева есть дерево каталогов. Как работает такое дерево мы уже разобрались. А вот справа находится список файлов в выделенной директории. Этот список как раз и хранится в компоненте *TListView*, с которым мы сейчас и познакомимся.



- TListView

С работой этого компонента мы так же познакомимся на практике. Для этого мы напишем простейший файловый менеджер и заодно закрепим большинство уже пройденного материала. Но всё это в следующей части этой главы, а сейчас я покажу тебе основные свойства компонента **TListView**, чтобы нам легче было двигаться дальше.

У списка элементов очень много свойств отвечающих за обрамление (внешний вид рамки) компонента. Я не буду их все перечислять, потому что разных вариантов очень много, я только советую тебе сейчас остановиться, создать новый проект, бросить на форму один компонент *ListView* и попробуй поиграть со следующими свойствами:

1. *BevelEdges* – здесь ты указываешь, с какой стороны должна быть оборка. По умолчанию со всех сторон стоит *true*.
2. *BevelInner* – вид внутренней оборки.
3. *BevelOuter* – вид внешней оборки.
4. *BevelKind* – тип оборки.
5. *BorderStyle* – стиль обрамления (плоский или трёхмерный).

Теперь остальные интересные свойства:

1. *Checkboxes* – если здесь **true**, то каждый элемент списка содержит ещё и компонент *CheckBox*.
2. *ColumnClick* – должны ли заголовки колонок выглядеть как кнопки и принимать сообщения от кнопок мыши.
3. *Columns* – если здесь щёлкнуть дважды мышкой, то появиться маленький редактор колонок списка.
4. *FlatScrollBar* – должны ли полосы прокрутки выглядеть в стиле *Flat* (плавающий).
5. *FullDrag* – полное перетаскивание.
6. *GridLines* – должны ли быть видна сетка, когда компонент выглядит в стиле *vsReport*.
7. *HotTrack* – включение режима *Hot*, когда при наведении на элемент списка могут происходить какие-то действия.
8. *HotTrackStyles* – Это группа свойств в которой описываются действия происходящие при включённом режиме *HotTrack*.
 - a. *htHandPoint* – если равно *true*, то при наведении на элемент курсором мыши, курсор принимает вид руки (как в IE при наведении на ссылку).
 - b. *htUnderlineCold* – если равно *true*, то надо подчёркивать надписи на элементах даже когда не наведена вышка.
 - c. *htUnderlineHot* – если равно *true*, то надо подчёркивать надписи на элементах только когда наведена вышка на элемент.
9. *IconOptions* – группа свойств отвечающих за иконки элементов.
 - a. *Arrangement* – расположение иконки сверху или слева.
 - b. *AutoArrange* – автоматическое выравнивание.
 - c. *WrapText* – переносить надпись под иконкой, когда она не помещается в одну строку.
10. *Items* – это объект, который хранит все элементы списка. Он очень похож на тот, что мы рассматривали в прошлой части при написании примера к дереву элементов.
11. *LargeImage* – здесь указывается компонент *TImageList*, в котором должны храниться большие иконки для элементов (32x32).
12. *MultiSelect* – есть ли возможность выделять сразу несколько элементов.

13. *RowSelect* – должна ли выделяться вся строка, когда компонент выглядит в стиле *vsReport*.
14. *ShowColumnHeader* – надо ли показывать заголовки, когда компонент выглядит в стиле *vsReport*.
15. *SmallImage* – здесь указывается компонент *TImageList*, в котором должны храниться маленькие иконки для элементов (16x16).
16. *ViewStyle* – стиль отображения списка. Здесь возможны варианты:
 - a. *vsIcon* – больше иконки.
 - b. *vsSmallIcon* – маленькие иконки.
 - c. *vsList* – список.
 - d. *vsReport* – отчёт.

На этом пока остановимся и перейдём к рассмотрению компонента в действии.

11.21 Простейший файловый менеджер.

Здесь я тебе хочу рассказать, как самому написать простейший файловый менеджер. Этим примером мы закрепим наши знания по работе с файлами и научимся работать со списком элементов.

Создай новый проект в Delphi и брось на него следующие компоненты: одну кнопку, одну строку вводу и один список элементов. Всё это расположи примерно так же, как на рисунке 11.21.1.

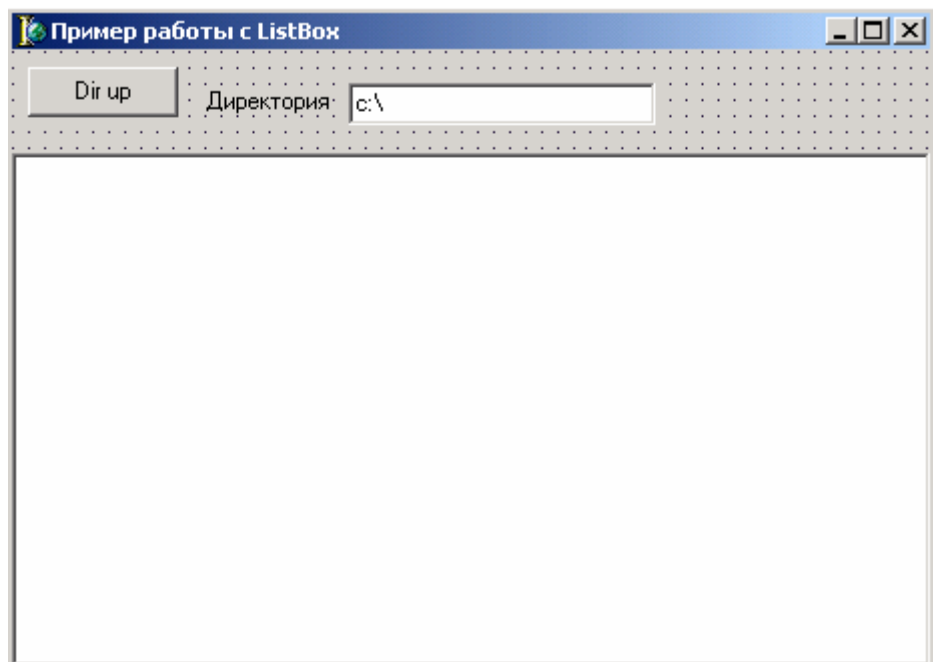


Рис 11.21.1

Для работы примера нужен модуль `shellapi`, поэтому давай сразу добавим его в раздел `uses`.

При рассмотрении примеров, обращай внимание на комментарии, они помогут тебе разобраться с происходящим в проге. А я буду расписывать только наиболее интересные моменты.

Эту программу мы начнём писать с процедуры FormCreate. Создай обработчик события OnCreate для формы и напиши в нём следующее:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  SysImageList: uint;
  SFI: TSHFileInfo;
begin
  //Создаю списки маленьких и больших иконок.
  ListView1.LargeImages:=TImageList.Create(self);
  ListView1.SmallImages:=TImageList.Create(self);

  //Запрашиваю большие иконки
  SysImageList := SHGetFileInfo("", 0, SFI,
    SizeOf(TSHFileInfo), SHGFI_SYSICONINDEX or SHGFI_LARGEICON);
  if SysImageList <> 0 then
  begin
    //Присваиваю системные иконки в ListView1
    ListView1.LargeImages.Handle := SysImageList;
    ListView1.LargeImages.ShareImages := TRUE;
  end;
  //Запрашиваю маленькие иконки
  SysImageList := SHGetFileInfo("", 0, SFI, SizeOf(TSHFileInfo),
    SHGFI_SYSICONINDEX or SHGFI_SMALLICON);
  if SysImageList <> 0 then
  begin
    ListView1.SmallImages.Handle := SysImageList;
    ListView1.SmallImages.ShareImages := TRUE;
  end;
end;
```

В первых двух строчках я создаю списки маленьких и больших иконок. Свойства *LargeImages* и *SmallImages* имеют тип *TImageList*, но сразу после создания компонента они равны *nil*. Поэтому я создаю их присваивая результат вызова *TImageList.Create(self)*. После этого они уже проинициализированы и необходимая память выделена.

Тут можно было поступить немного другим способом – поставить на форму два компонента *TImageList* и просто указать их в соответствующих свойствах. В этом случае не пришлось бы ничего инициализировать, потому что компоненты стоящие на форме инициализируются автоматически. Но я решил не делать этого, а показать тебе, что объектное свойство можно сразу заставить работать без использования дополнительных компонентов.

После этого я запрашиваю у системы список больших иконок. Для этого я использую функцию *SHGetFileInfo*, которая возвращает информацию о файле, директории или диске. Первый параметр - путь к файлу. Второй - атрибуты. Третий - указатель на *TSHFILEINFO*. Четвертый - размер *TSHFILEINFO*. Последний - флаги, указывающие на тип информации запрашиваемый у системы.

Теперь о том, как выглядит функция в моём примере. Первые два параметра пустые, это означает, что нам нужны глобальные данные, а не информация о конкретном файле. Если указать здесь реальные значения файла, то мы получим информацию о нём, а если указать нулевые значения, то мы получим системную информацию.

В качестве флагов я указываю *SHGFI_SYSICONINDEX* и *SHGFI_LARGEICON*. *SHGFI_SYSICONINDEX* означает, что я запрашиваю указатель на системный список иконок (*ImageList*). Второй флаг говорит, что мне нужны большие иконки.

При втором вызове этой функции (чуть ниже в коде) я запрашиваю маленькие иконки (*SHGFI_SMALLICON*). Функция возвращает мне указатель на соответствующий системе *SysImageList*, который я в последствии присваиваю в *ListView1.LargeImages.Handle*. После этого присваивания *ListView1.LargeImages* содержит все системные иконки размера 32x32.

Системный список иконок (*ImageList*) содержит все иконки, установленные в системе и ассоциированные с разными типами файлов. Эти иконки ты можешь видеть в Windows Explorer у файлов doc, txt, ini, zip и др.

Теперь создадим обработчик события *OnShow*. В ней я вызываю другую процедуру *AddFile*, которая считывает все файлы из текущей директории.

```
procedure TForm1.FormShow(Sender: TObject);
begin
  AddFile(Edit1.Text+'*.*',faAnyFile)
end;
```

Процедура *AddFile* объявлена в разделе **private** нашей формы **Form1** следующим образом:

```
private
{ Private declarations }
function AddFile(FileMask: string; FFileAttr:DWORD): Boolean;
```

Объяви эту процедуру так же и потом нажми клавиши Ctrl+Shift+C и *Delphi* создаст заготовку для будущей процедуры:

```
function TForm1.AddFile(FileMask: string; FFileAttr:DWORD): Boolean;
begin
end;
```

В эту заготовку напиши следующее:

```
function TForm1.AddFile(FileMask: string; FFileAttr:DWORD): Boolean;
var
  ShInfo: TSHFileInfo;
  attributes: string;
  FileName: string;
  hFindFile: THandle;
  SearchRec: TSearchRec;

function AttrStr(Attr: integer): string;
begin
  Result := '';
  if (FILE_ATTRIBUTE_DIRECTORY and Attr) > 0 then Result := Result + 'D';
  if (FILE_ATTRIBUTE_ARCHIVE and Attr) > 0 then Result := Result + 'A';
  if (FILE_ATTRIBUTE_READONLY and Attr) > 0 then Result := Result + 'R';
  if (FILE_ATTRIBUTE_HIDDEN and Attr) > 0 then Result := Result + 'H';
  if (FILE_ATTRIBUTE_SYSTEM and Attr) > 0 then Result := Result + 'S';
end;
```

```
begin
  ListView1.Items.BeginUpdate;
  ListView1.Items.Clear;

  Result := False;
  hFindFile := FindFirst(FileMask, FFileAttr, SearchRec);
  if hFindFile <> INVALID_HANDLE_VALUE then
  try
    repeat
      with SearchRec.FindData do
        begin

          if (SearchRec.Name = '.') or (SearchRec.Name = '..') or
            (SearchRec.Name = '') then continue;

          FileName := SlashSep(Edit1.Text, SearchRec.Name);
          SHGetFileInfo(PChar(FileName), 0, ShInfo, SizeOf(ShInfo),
            SHGFI_TYPENAME or SHGFI_SYSICONINDEX);
          Attributes := AttrStr(dwFileAttributes);
          //Добавляю новый элемент
          with ListView1.Items.Add do
            begin
              //Присваиваю его имя
              Caption := SearchRec.Name;
              //Присваиваю индекс из системного списка изображений
              ImageIndex := ShInfo.ilcon;
              //Присваиваю размер
              SubItems.Add(IntToStr(SearchRec.Size));
              SubItems.Add((ShInfo.szTypeName));
              SubItems.Add(FileTimeToDateTimeStr(ftLastWriteTime));
              SubItems.Add(attributes);
              SubItems.Add(Edit1.Text + cFileName);
              if (FILE_ATTRIBUTE_DIRECTORY and dwFileAttributes) > 0 then
                SubItems.Add('dir')
              else
                SubItems.Add('file');
            end;
          Result := True;
        end;
      until (FindNext(SearchRec) <> 0);
    finally
      FindClose(SearchRec);
    end;
  end;
  ListView1.Items.EndUpdate;
end;
```

Неплохая процедура получилась и надо бы подробно её описать. Я буду делать это по кусочкам, чтобы было легче воспринимать:

```
function TForm1.AddFile(FileMask: string; FFileAttr:DWORD): Boolean;
var
  ...
  ...

function AttrStr(Attr: integer): string;
begin
  Result := '';
  if (FILE_ATTRIBUTE_DIRECTORY and Attr) > 0 then Result := Result + ';
```

```
if (FILE_ATTRIBUTE_ARCHIVE and Attr) > 0 then Result := Result + 'A';  
if (FILE_ATTRIBUTE_READONLY and Attr) > 0 then Result := Result + 'R';  
if (FILE_ATTRIBUTE_HIDDEN and Attr) > 0 then Result := Result + 'H';  
if (FILE_ATTRIBUTE_SYSTEM and Attr) > 0 then Result := Result + 'S';  
end;
```

После имени процедуры идёт объявление локальных переменных. Это всё понятно и мы не раз уже такое делали. Но после объявления переменных, вместо начала процедуры (**begin**) у меня стоит объявление другой локальной процедуры - **function AttrStr(Attr: integer): string;**. Да, и такое в Delphi тоже возможно. Конечно же эту процедуру можно написать как полноценную, но я решил показать тебе, что такое локальная процедура.

Если одна процедура/функция (внутренняя) объявлена внутри другой (внешней), то внутренняя процедура может быть вызвана только из внешней. Весь остальной код программы не будет знать о существовании где-то внутренней процедуры.

Лично я такие вещи стараюсь не использовать, потому что пока не встречался с ситуацией, когда внутренняя процедура необходима. Я всегда прекрасно обхожусь и без неё. Но всё же рассказать тебе о ней необходимо, потому что ты можешь встретить такую конструкцию в других программах.

После объявления и описания внутренней процедуры идёт начало (**begin**) внешней процедуры. Вот тут уже начинается самое интересное. В самом начале я вызываю два метода компонента ListView1:

```
ListView1.Items.BeginUpdate;  
ListView1.Items.Clear;
```

Первый метод *BeginUpdate* говорит о том, что начинается обновление элементов списка. После этого вызова никакие изменения вносимые в элементы не будут отражаться на экране, пока не будет вызван *EndUpdate*.

Когда ты хочешь произвести незначительное изменение, то не надо вызывать эти методы, но когда ты чувствуешь, что здесь элементы списка будут изменяться очень сильно, то лучше все изменения заключить между вызовами *BeginUpdate* и *EndUpdate*. Это связано с тем, что когда ты вносишь хоть какое-то изменение, оно сразу отображается на экране. Логично? Я тоже так думаю. А что если тебе нужно удалить все элементы и потом в цикле добавить в список 1000 новых элементов. В этом случае после удаления и каждого добавления нового элемента будет происходить прорисовка компонента. Вот тут и возникает вопрос: «Зачем после каждого добавления рисовать?». В этом случае намного эффективнее будет добавить все элементы, а только потом их прорисовать все сразу. Вот именно для этого и существуют своеобразные скобки *BeginUpdate* и *EndUpdate*:

```
ListView1.Items.BeginUpdate; // Запрещаем прорисовку
```

```
// Делаем необходимые изменения
```

```
ListView1.Items.EndUpdate; // Прорисовываем все изменения сразу
```

С этим разобрались, можно ехать дальше. После вызова *BeginUpdate* я очищаю текущий список элементов с помощью вызова *ListView1.Items.Clear*.

Далее идёт цикл поиска файлов, с которым мы уже немного познакомились в 10-й главе моей книги. Здесь я только напомним тебе этот процесс:

FindFirst - открывает поиск. В качестве первого параметра выступает маска поиска. Если ты укажешь конкретный файл, то система найдёт его. Но это не серьёзно, лучше искать более серьёзные вещи. Например, ты можешь запустить поиск всех файлов в корне диска C. Для этого первый параметр должен быть 'C:*. *'. Для поиска только файлов EXE, в папке *Fold* ты должен указать 'C:\Fold*.exe'.

Второй параметр - атрибуты включаемых в поиск файлов. Я использую *faAnyFile*, чтобы искать любые файлы. Тебе доступны

faReadOnly - искать файлы с атрибутом *ReadOnly*.

faHidden - искать скрытые файлы.

faSysFile - искать системные файлы.

faArchive - искать архивные файлы.

faDirectory - искать директории.

Последний параметр - это структура в которой нам вернется информация о поиске, а именно имя найденного файла, размер, время создания и т.д. После вызова этой процедуры, я проверяю на корректность найденного файла. Если всё в норме, то запускается цикл **Repeat - Until**. Этот цикл выполняет операторы расположенные между **repeat** и **until**, пока условие расположенное после слова *until* является верным. Как только условие нарушается, цикл прерывается. Этот цикл похож на **while**, но с одним отличием. Если в цикле **while** условие заведомо не верно, то операторы внутри цикла не выполняются. А в **Repeat-Until** выполняются, потому что сначала происходит выполнение операторов, а лишь затем проверка **Until**. Рассмотрим пример:

```
index:=1;  
while index=0 do  
  Param:=0;
```

В этом примере оператор **Param:=0;** не будет выполнен, потому что **index=1** и условие заведомо не верно.

```
index:=1;  
repeat  
  Param:=0;  
until index=0;
```

В этом примере **Param:=0** выполнится, Потому что сначала выполняется этот оператор, а лишь потом проверка на равенство **index** нулю.

Хочу предупредить, что функция поиска, может возвращать в качестве найденного имени в структуре **SearchRec** (параметр **Name**) точку или две точки. Если ты посмотришь на директорию, то таких файлов не будет. Откуда берутся эти имена? Имя файла в виде точки указывает на текущую директорию, а имя файла из двух точек указывает на директорию верхнего уровня. Если я встречаю такие имена, то я их просто отбрасываю:

```
//Отбрасывание имён с точкой и двумя точками  
if (SearchRec.Name = '.') or (SearchRec.Name = '..') or
```

(SearchRec.Name = "") then continue;

Далее идёт вызов функции *SlashSep*:

FileName := SlashSep(Edit1.Text, SearchRec.Name);

Эта функция и *FileTimeToDateTimeStr* написаны мной и объявлены в разделе **var** после объявления объекта:

```
var  
  Form1: TForm1;  
  function SlashSep(Path, FName: string): string;  
  function FileTimeToDateTimeStr(FileTime: TFileTime): string;  
  
implementation
```

Я специально объявил их там, чтобы показать тебе, как можно пользоваться функциями не принадлежащими ни одному объекту. Здесь функция *SlashSep* объявлена не внутри объекта, значит она никому не принадлежит.

Вообще-то самостоятельные функции не обязательно где-либо объявлять. Ты можешь без проблем просто реализовать её и нигде не описывать. Но ты должен учитывать, что если ты где-то хочешь использовать эту функцию, то реализация обязательно должна быть раньше. Вот пример правильного использования самостоятельной процедуры/функции:

```
procedure Examp;  
begin  
end;  
  
procedure Form1.Examp2;  
begin  
  Examp;  
end;
```

В этом примере я создал самостоятельную процедуру *Examp* и метод объекта *Form1* – *Examp2*. Из метода *Examp2* я вызываю самостоятельную процедуру *Examp*. Этот код правильный, потому что процедура сначала реализовывается, а потом уже используется.

А теперь посмотри на неправильный код:

```
procedure Form1.Examp2;  
begin  
  Examp;  
end;  
  
procedure Examp;  
begin  
end;
```

В этом примере я пытаюсь вызвать процедуру, которая реализована после вызова и поэтому компилятор выдаст ошибку. Чтобы этого избежать, самостоятельные процедуры можно описывать в разделе `var`:

```
var
  procedure Examp;

  procedure Form1.Examp2;
  begin
    Examp;
  end;

  procedure Examp;
  begin
  end;
```

А теперь давай посмотрим, как же выглядит функция *SlashSep*:

```
function SlashSep(Path, FName: string): string;
begin
  if Path[Length(Path)] <> '\' then
    Result := Path + '\' + FName
  else Result := Path + FName;
end;
```

Как видишь, здесь я пишу просто «function *SlashSep*» не добавляя имени объекта перед именем функции, потому что эта функция самостоятельная.

Эта функция получает два параметра – путь к файлу и имя файла, которые она должна соединить в одну строку, чтобы получился полный путь к файлу. Но сначала мы должны проверить, заканчивается ли путь (первый полученный параметр – *Path*) знаком '\'. Именно это делается в первой строчке кода:

Path[Length(*Path*)] <> '\'

Переменная *Path* – это строка типа *String*, а значит мы можем к ней обращаться как к массиву символов. Это значит что чтобы получить доступ к первому символу мы должны написать *Path*[1]. Нам нужно проверить последний символ, поэтому в квадратных скобках я пишу *Length*(*Path*). Функция *Length* возвращает длину переданной ей строковой переменной, а это значит, что в квадратных скобках мы указываем длину строки, а это последний символ.

Если бы нам нужен был предпоследний символ, то мы бы написали *Path*[Length(*Path*)-1]. В этом случае я из длины строки вычитаю единицу и получаю предпоследний символ.

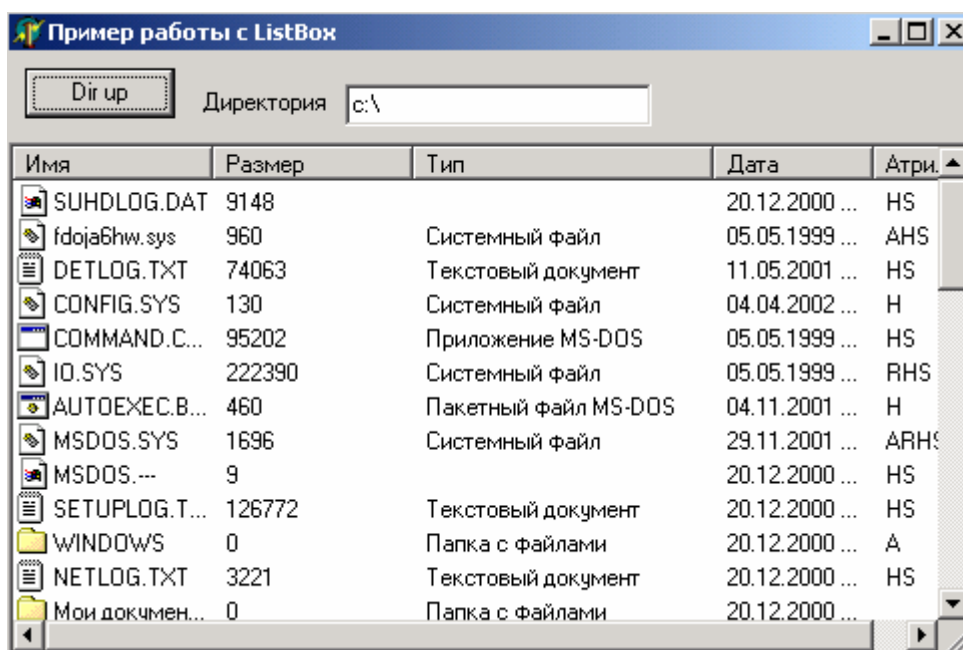
Если последний символ не равен '\', то я добавляю сначала его, а потом имя файла. Если равен, то нужно только добавить имя файла и записать в переменную *Result*, чтобы функция вернула полный путь к файлу.

С этой функцией покончено и пора вернуться к нашему перечислению файлов. Следующим идёт вызов системной функции *SHGetFileInfo*. Она возвращает информацию

о файле. Я не буду на ней сейчас останавливаться. В принципе она простая и ты наверно сможешь с ней разобраться, а если нет, то я немного позже вернусь к этому.

Сейчас нас больше интересует работа с компонентом `ListView`. Следующий код добавляет в список новый элемент: `ListView1.Items.Add`. Я это делаю внутри конструкции `with`, значит все последующие действия между `begin` и `end` будут выполняться с новым элементом. А именно, я изменяю заголовок нового элемента (`Caption := SearchRec.Name`) и картинку (`ImageIndex := ShInfo.iIcon`).

У каждого элемента есть свойство `SubItems`, которое хранит дополнительную информацию. Когда компонент находится в режиме отображения иконок, то дополнительная информация не видна. Но если выбрать в свойстве `ViewStyle` значение `vsReport`, то компонент будет выглядеть в виде таблицы, где каждый столбец отображает дополнительную информацию:



Чтобы добавить дополнительные колонки к новому элементу надо выполнить `SubItems.Add('значение');`.



Чтобы колонки отображались, нужно ещё в свойстве `Columns` указать имена колонок. Если имена колонок не указаны, то ничего отображаться не будет. И не забывай, что при описании колонок первая – это заголовок элементов, а остальные это дополнительные параметры в порядке их добавления с помощью `SubItems.Add`.

11.22 Улучшенный файловый менеджер (С возможностью запуска файлов)

Давай добавим к нашему файловому менеджеру возможность путешествия по директориям и запуска файлов. Для этого нужно создать обработчик события `OnDblClick` для компонента `ListView` и написать в нём следующее:

```
procedure TForm1.ListView1DblClick(Sender: TObject);
```



```
begin
//Это директория?
if (ListView1.Selected.SubItems[5] = 'dir') then
begin
//Если да, то прибавить имя выделенной директории к пути
//и перечитать файлы из неё.
Edit1.Text:=Edit1.Text+ListView1.Selected.Caption+'\\';
AddFile(Edit1.Text+'*.*',faAnyFile)
end
else
//Если нет, то это файл и я его запускаю.
ShellExecute(Application.MainForm.Handle, nil,
PChar(Edit1.Text+ListView1.Selected.Caption), "",
PChar(Edit1.Text), SW_SHOW);
end;
```

В этом коде я в самом начале проверяю по чём мы щёлкнули. Если это директория, то надо перейти в неё, а если файл, то надо его запустить. Для этого я проверяю 5-й дополнительный параметр выделенного элемента: *ListView1.Selected.SubItems[5]='dir'*. Когда я добавлял элементы и дополнительные параметры в *ListView*, то в качестве 5-го указывал для директорий значение 'dir', а для файлов 'file'. Теперь мне надо только проверить этот параметр.


Если выделенная строка – это директория, то я изменяю значение текущей директории в *Edit1.Text* и перечитываю её с помощью вызова *AddFile*, указав новое значение директории.

Если выделенная строка – это файл, то его надо запустить. Я люблю это делать с помощью вызова функции *ShellExecute*. У этой функции следующие параметры:

1. Программа, отвечающая за запуск приложения. Тут можно указать *nil*, но я указал главное окно моей программы (*Application.MainForm.Handle*).
 2. Строка, указывающая на операцию, которую надо выполнить. Укажем *nil* для запуска файла.
 3. Строка содержащая полный путь к файлу.
 4. Строка параметров передаваемых программе в командной строке.
 5. Директория по умолчанию.
 6. Команда показа. Здесь я указал *SW_SHOW* для нормального отображения окна. Можно указать и другие параметры (все ты найдёшь в файле помощи), но чаще всего используются *SW_SHOW* (нормальный режим), *SW_SHOWMAXIMIZED* (показать максимизировано) или *SW_SHOWMINIMIZED* (показать в свёрнутом состоянии).
-



Функция *ShellExecute* объявлена в модуле *Shellapi*, поэтому его необходимо добавить в раздел *uses*, иначе *Delphi* не сможет откомпилировать проект.

 На компакт диске, в директории \Примеры\Глава 11\ListView ты можешь увидеть пример этой программы.

11.23 Подсказки для чайников (TStatusBar)

Если ОС unix создавалась для профессионалов, то Windows создавалась для пользователей, чтобы им легче было работать. Потом она превратилась в ОС для чайников, ну а сейчас Windows превратили в ОС для полных кретин, которые с компьютером полностью несовместимы. Так что теперь для успеха любой программы нужно обязательно делать большое количество подсказок, потому что кетины не умеют читать мануалы и файлы помощи. Сейчас уже надо чтобы любой мог сесть за компьютер и сразу начинать работать.

Самым первым способом облегчения жизни бедным юзерам стали строки состояния. Они и сейчас широко используются, потому что просты в использовании и удобны в обращении. Именно с этим компонентом мы сейчас и познакомимся.



- TStatusBar

Поставить компонент на форму, это ещё не значит, что подсказки сразу же сами появятся на панели. Для полноценной работы надо выполнить следующее:

1. У компонента, при наведении на который должна отображаться подсказка, в свойстве *Hint* должен быть внесён текст подсказки.
2. Если ты хочешь, чтобы подсказка выскакивала не только в строке состояния, но и над компонентом, то у него или у родительского окна в свойстве *ShowHint* нужно установить *true*.
3. Мы должны создать обработчик события на подсказки.

Может это звучит сложно, но реально всё просто. Создай новое приложение и брось на него кнопку. Теперь в свойстве *Hint* напиши «*Это кнопка выхода*».

Попробуй запустить приложение и навести на кнопку. Никаких сообщений и подсказок пока не должно быть. Закрой программу и переходи опять в Delphi. Теперь попробуй установить в свойстве *ShowHint* у компонента или у главной формы значение *true*. Если ты установишь только у компонента, то подсказка будет выскакивать только у него. Если у формы, то подсказка будет появляться у всех компонентов на форме, у которых есть текст в свойстве *Hint* и *ParentShowHint* равно *true*.

Можешь запустить приложение и проверить появление подсказки.

Теперь мы добавим к нашему приложению возможность отображения такого же текста в строке состояния. Брось на форму компонент **TStatusBar**. Теперь перейди в редактор кода и найди раздел **private**. В нём добавь объявление процедуры *ShowHint*:

```
private
{ Private declarations }
procedure ShowHint(Sender: TObject);
```

Имя процедуры может быть и другим (например *MyShowHint*) но параметр должен быть именно такой.

Теперь нажми Ctrl+Shift+C чтобы Delphi создал заготовку для процедуры:

```
procedure TForm1.ShowHint(Sender: TObject);
begin
end;
```

Можешь и сам написать этот текст, но только после текста `{ TForm1 }` или ещё дальше после какой-нибудь процедуры:

implementation

{ \$R *.dfm }

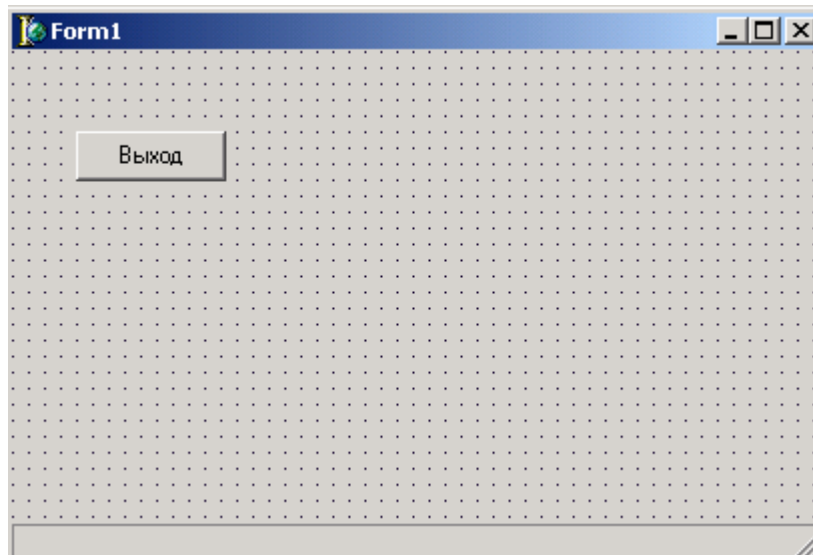
{ TForm1 }

```
procedure TForm1.ShowHint(Sender: TObject);  
begin  
end;
```

Внутри процедуры напиши следующее:

```
procedure TForm1.ShowHint(Sender: TObject);  
begin  
  StatusBar1.SimpleText := Application.Hint;  
end;
```

Итак, наша процедура должна будет вызываться каждый раз, когда надо вывести подсказку. Внутри процедуры мы присваиваем в свойство *SimpleText* строки состояния текст находящийся в *Application.Hint*. А в *Application.Hint* всегда находится подсказка, которую надо сейчас отобразить.



Теперь создай обработчик события *OnShow* для главной формы и в нём напиши:

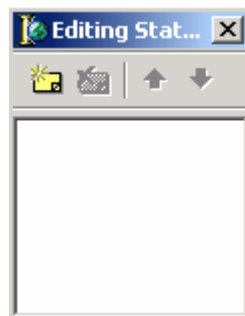
```
procedure TForm1.FormShow(Sender: TObject);  
begin  
  Application.OnHint := ShowHint;  
end;
```



Здесь мы программно назначаем нашу процедуру *ShowHint* в качестве обработчика события *OnHint*. Я люблю это делать программно, но можно было поступить и проще:

1. Поставить на форму компонент *TApplicationEvents* с закладки *Additional*.
2. У этого компонента на закладке *Events* создать обработчик события *OnHint* и там сразу же написать «*StatusBar1.SimpleText := Application.Hint*».

 На компакт диске, в директории \Примеры\Глава 11\Hint ты можешь увидеть пример этой программы.

Теперь попробуем создать строку состояния из нескольких панелей. Выдели строку состояния и дважды щёлкни по свойству *Panels*. Перед тобой должно открыться окно редактора панелей:




В этом окне первая кнопка создаёт новую панель  (также можно нажать клавишу *Ins*), а вторая  удаляет выделенную (также можно нажать *Del*).

Создай новую панель и в её свойстве *Width* (ширина) установи значение 200. Теперь создай ещё одну панель. Всё, можно закрывать окно.

Теперь перейди в процедуру обработчик события *OnHint* и измени её текст на:

```
procedure TForm1.ShowHint(Sender: TObject);
begin
  StatusBar1.Panels[1].Text := Application.Hint;
end;
```

Здесь я присваиваю текст сообщения (*Application.Hint*) в свойство *Text* первой панели строки состояния.

 На компакт диске, в директории \Примеры\Глава 11\HintPanels ты можешь увидеть пример этой программы.

11.24 Панель инструментов (TToolBar и TControlBar).

Панель инструментов уже давно въелась в нашу жизнь и уже трудно представить себе какой-нибудь хоть более менее значащий проект без неё. Некоторые считают, что меню достаточно, а некоторые наоборот обходятся одной панелью инструментов. Я же считаю, что любое оконное приложение должно иметь и то и другое.

Панель инструментов чаще всего располагается сразу же под меню, но это не обязательно. Иногда удобно расположить его вдоль какой-нибудь другой стороны окна

(левой, правой или нижней). В своей книге я буду располагать его в основном сверху (классический вариант), как это делается в большинстве программ, например MS Word.

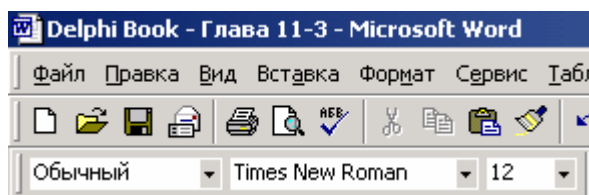




Рис 11.24.1 Панель инструментов MS Word. В ней ты видишь не только кнопки быстрого доступа к командам, но и выпадающие списки *ComboBox*.

Давай создадим маленькое приложение использующее панель инструментов. Брось на форму компонент *ControlBar*  с закладки **Additional** и измени его свойство *Align* на *alTop*, чтобы растянуть компонент вдоль верхней кромки окна. Сразу же желательно изменить и свойство *AutoSize* на *true*.

Компонент *ControlBar* я не рассматривал, потому что в нём нет ничего особенного, но он хорош тем, что на него удобно располагать панели инструментов. Они автоматически становятся перемещаемыми внутри *ControlBar*. Это значит, что панели можно будет двигать по своему усмотрению. Ну а если свойство *AutoSize* равно *true*, то компонент будет автоматически растягиваться и сужаться, когда ты будешь выстраивать все панели в одну строку или в столбик.

Давай теперь бросим внутрь компонента *ControlBar* одну панель *ToolBar*  с закладки **Win32**. Сразу же изменим одно его свойство – дважды щёлкни по свойству *EdgeBorders* и измени свойство *ebTop* на *false* (рисунок 11.24.2). Это заставит исчезнуть оборочку сверху панели.

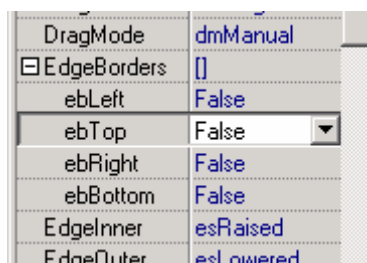


Рис 11.24.2 Убираем ненужную оборочку.

Ещё желательно сразу же изменить и здесь свойство *AutoSize* на *true*, чтобы панель принимала размеры соответствующие кнопкам.

Теперь создадим кнопки на панели. Для этого щёлкни по ней правой кнопкой крысы и выбери из появившегося меню пункт *New Button* (рисунок 11.24.3). Пункт *New Separator* этого же меню создаёт разделитель между кнопками. Если тебе нужно будет удалить кнопку или разделитель, то просто выделишь его и нажимаешь кнопку *Del* (на клавиатуре :)).

Таким образом создай две кнопки, потому разделитель и ещё одну кнопку. У тебя должно получиться нечто похожее на рисунок (рисунок 11.24.3).

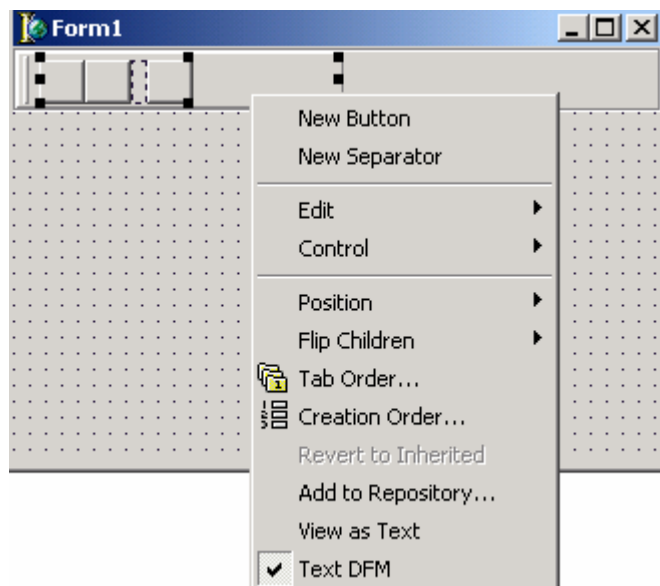


Рис 11.24.3 Создание новой кнопки.

В принципе, простая панель уже создана, но она простая. Теперь необходимо сделать так, чтобы кнопки что-то отображали. Но для начала я советую выделить саму панель и изменить свойство *Flat* на *true*, чтобы кнопки на панели выглядели более изящно (плоско).

Теперь бросим на форму компонент **TImageList** и добавим в него три картинки. Их изображение пока не имеет особого значения, поэтому можешь выбирать любые, главное, чтобы размер был 16x16.

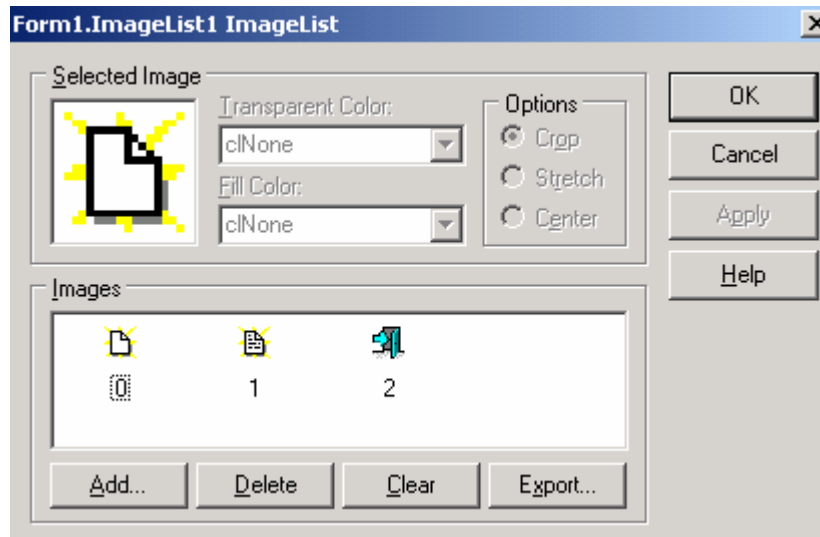


Рис 11.24.4 Набор изображений для панели инструментов

Теперь выдели панель и в свойстве *Images* укажи созданный набор картинок. На кнопках сразу же отобразятся картинки в той последовательности, в которой ты их добавил. Если ты хочешь изменить картинку на какой-нибудь кнопке, то надо выделить её и изменить свойство *ImageIndex*.

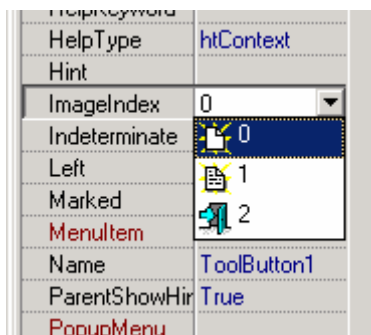


Рис 11.24.5 Изменение картинки для кнопки.

Теперь для каждой кнопки в свойстве *Caption* напиши осмысленный текст (по умолчанию там стоит текст *ToolButton* плюс порядковый номер кнопки). Желательно, чтобы текст соответствовал изображению на картинке. У нас хоть и пример, но всё же он должен быть приближён к боевым условиям.

Давай сделаем так, чтобы панель отображала на кнопках не только картинки, но и указанный в свойствах *Caption* текст. Для этого установи *true* в свойстве *ShowCaptions* у панели инструментов. Результат ты можешь увидеть на рисунке 11.24.6.

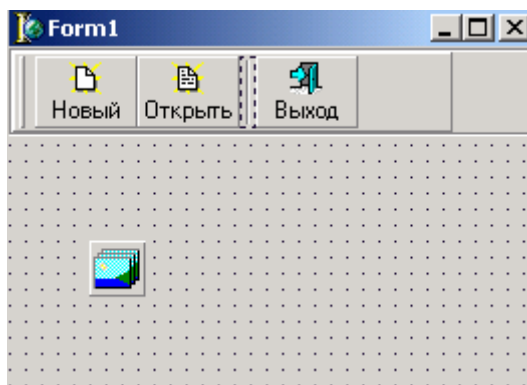


Рис 11.24.6 Панель отображающая картинки и текст.

Как видишь, в данном случае текст отображается под изображением. Если ещё установить свойство *List* у панели инструментов, то текст будет отображаться справа от картинки (рисунок 11.24.7).

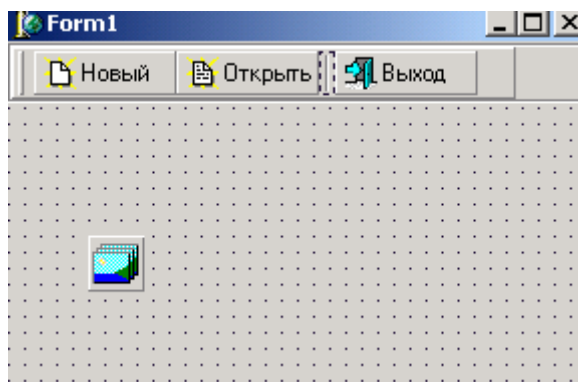



Рис 11.24.7 Панель отображающая текст слева от картинки.

Ну и для закрепления материала, давай создадим обработчик события для какой-нибудь кнопки. У меня последняя кнопка – *Выход*, вот для неё я и создам обработчик. Для этого я щёлкнул по ней дважды и в созданной процедуре обработчика написал **Close**;

 На компакт диске, в директории \Примеры\Глава 11\ToolBar ты можешь увидеть пример этой программы.

11.25 Перемещаемые панели и меню в стиле MS (Docking).

Почему-то меня очень часто просят рассказать, как можно добиться такого эффекта как у ToolBar-ов в MS Office. Для большей ясности - это когда палитру с кнопками можно оторвать от окна и прилепить в другое место или вообще превратить в отдельное окно.

Для того, чтобы TToolBar можно было перемещать, достаточно установить в нём свойство *DragKind* в *dkDock*. Вот и всё. Но главная проблема не в этом. Самое сложное здесь - это сохранить положение TToolBar после выхода из проги и восстановить его при запуске. Для примера я написал маленькую прогу, которую ты должен доделать до полноценной.

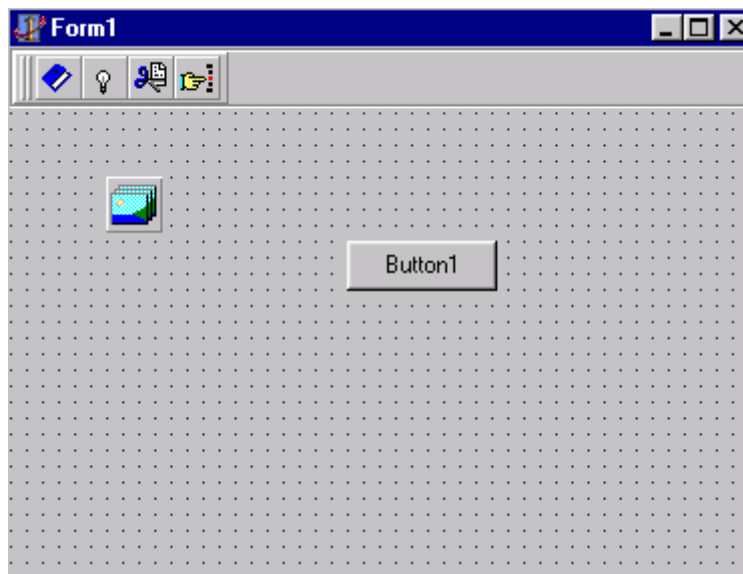


Рис 11.25.1 Форма примера

Для демонстрации мне понадобилась кнопка, по нажатию которой будет выводиться положение TToolBar. По нажатию кнопки пишем следующее:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  r:TRect;
begin
  if ToolBar1.HostDockSite<>ControlBar1 then
  begin
    GetWindowRect(ToolBar1.Handle, R);
    Application.MessageBox(PChar(IntToStr(r.Left)+'--'+IntToStr(r.Top)),
      'MM',IDOK);
  end;
end;
```

В первой строке я проверяю, лежит ли ToolBar1 на ControlBar1 с помощью (ToolBar1.HostDockSite<>ControlBar1). Если он лежит, то получить положение ToolBar1 очень просто. Для этого можно узнать всего лишь ToolBar1.Left и ToolBar1.Top.

Если ToolBar1 не лежит на ControlBar1 (ToolBar1 выглядит как отдельное окно), то задача усложняется. Тебе придётся вызывать GetWindowRect, чтобы получить реальное положение ToolBar1 на экране. В качестве первого параметра ты должен передать указатель на ToolBar1, а второй - это переменная типа TRect в которую запишется реальное положение окна. Для удобства я вывожу эти значения в окне сообщения Application.MessageBox.

Всё это я делаю для наглядности. Теперь ты можешь запустить прогу и переместить ToolBar1 по экрану. Каждый раз, когда ты будешь нажимать кнопку, программа будет выводить окно и показывать тебе реальное положение ToolBar1.

По событию OnShow я написал:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  ToolBar1.ManualDock(nil, nil, alNone);
  ToolBar1.ManualFloat(Bounds(100, 500, ToolBar1.UndockWidth,
    ToolBar1.UndockHeight));
end;
```

ToolBar1.ManualDock заставляет переместится ToolBar1 на новый компонент. В качестве первого параметра указывается указатель на компонент или окно, к которому мы хотим прилепить ToolBar1. Я хочу, чтобы после загрузки ToolBar1 превратился в отдельное окно, поэтому я указываю nil. Второй параметр можешь ставить nil. Он означает компонент внутри компонента указанного в качестве первого параметра, на который мы хотим поместить ToolBar1. Я указал nil. Третий параметр – выравнивание.

С помощью ToolBar1.ManualFloat я просто двигаю ToolBar1 внутри нового компонента. У меня новый компонент nil, т.е. окно, поэтому я двигаю ToolBar1 по окну. Может не совсем понятно? Попробуй запустить пример и поиграть с ним, тогда всё встанет на свои места.

И ещё ToolBar1.UndockWidth и ToolBar1.UndockHeight возвращают размер ToolBar1, когда он выглядит как окно, а не лежит на ControlBar1.

Когда ты будешь использовать это в своей проге для сохранения положения ToolBar1, тебе надо будет написать примерно следующее по событию OnClose:

```
var
  r: TRect;
begin
  if ToolBar1.HostDockSite<>ControlBar1 then
  begin
    GetWindowRect(ToolBar1.Handle, R);
    Здесь надо сохранить в реестре R.Left и R.Top.
    А также признак, что ToolBar1 не лежит на ControlBar1
  end
  else
  begin
    Здесь надо сохранить в реестре ToolBar1.Left и ToolBar1.Top.
    А также признак, что ToolBar1 лежит на ControlBar1
  end;
end;
```

На запуск программы ты должен написать примерно следующее:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  Прочитать положение ToolBar1. ControlBar1 to
  Begin
    ToolBar1.Left:=Сохранённая левая позиция
    ToolBar1.Top:=Сохранённая верхняя позиция
  End;
  Иначе
  begin
    ToolBar1.ManualDock(nil,nil,alNone);
    ToolBar1.ManualFloat(Bounds(Сохранённая левая позиция,
      Сохранённая правая позиция, ToolBar1.UndockWidth,
      ToolBar1.UndockHeight));
  End;
end;
```

Как видишь, подводные булыжники есть. Но всё же ничего сильно сложного нет.

Теперь мы сделаем менюшку в стиле M\$. Для этого нужно поставить ещё один `ToolBar` и установим его свойство `ShowCaption` в `true`. Создадим на нём две кнопки и назовём их *File* и *Edit*. Теперь установим компонент `MainMenu` и сделаем его таким как на рисунке 11.25.2. Меню *Not visible* сделаем невидимым (`Visible=false`), в этом случае всё меню будет подключено к форме но будет не видно. Для чего я это делаю, ведь можно было использовать `PopupMenu`? А потому что при использовании `PopupMenu` приходится мучиться с клавишами быстрого вызова, а в моём способе они подключаются автоматически вместе с главным меню.

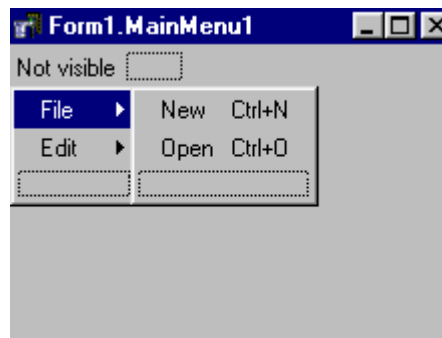


Рис 11.25.2 Меню




Чтобы создать подменю для меню *File*, нужно щёлкнуть по нём правой кнопкой и выбрать *Create Submenu* или нажать **CTRL+Стрелка в право**

Теперь кнопке *File* в свойстве `MenuItem` ставим `File1` (имя пункта меню), а кнопке *Edit* ставим `Edit1`. И напоследок обеим кнопкам нужно установить свойство `Grouped` в `true`.

Напоследок у каждой кнопки панели инструментов надо установить в свойстве *Grouped* – *true*.

Автор: Horrific. Home page: www.cydsoft.com/vr-online vr_online@cydsoft.com

Но это не единственный способ создания меню (это тот, что я чаще использую). Можно ещё просто создать полноценное главное меню. Потом просто убрать его из свойства главной формы *Меню*, а указать у созданной тобой пустой панели (без всяких кнопок) в таком же свойстве *Меню*.

 На компакт диске, в директории \Примеры\Глава 11\Dock ты можешь увидеть пример этой программы.