Практическое занятие 8. Отладка приложения Блокнот - шифратор

Данная лабораторная работа закрепляет пройденный материал и показывает на практике реализацию текстового редактора - прототипа стандартного Блокнота Windows. В лекции рассматривается включение модуля стороннего разработчика, практическое применение четырех диалогов.

Цель Занятия

Закрепление работы с диалогами, модулями и компонентом `TMemo`.

Постановка задачи

Написать простой **Блокнот**, пусть даже со всеми возможностями стандартного Блокнота Windows, было бы скучновато. Вот если еще научить его шифровать/дешифровать текст, да еще и с переменным ключом-паролем - совсем другой разговор! Идея такова: делаем **Блокнот**, добавляем к нему модуль стороннего разработчика, в котором реализованы процедуры шифрации и дешифрации текста. Причем для шифрации/дешифрации требуется ввести пароль - слово или даже фразу. И если хоть один символ пароля будет неверен, текст не расшифруется! Затем вы сможете раздать такой **Блокнот** всем желающим, но прочитать ваш зашифрованный текст сможет лишь тот, кто знает пароль. Приступим?

Реализация

Подготовка формы очень проста, набор компонентов тоже не очень велик. Для начала откройте **Lazarus** с новым проектом. Скорее всего, вы эту программу будете передавать друзьям, поэтому имеет смысл сразу же отключить от проекта вставку отладочной информации. Если вы помните, нужно выбрать команду "**Проект -> Параметры проекта**", в разделе "**Параметры компилятора**" перейти на подраздел "**Компоновка**" и убрать флажок "**Генерировать отладочную информацию для GDB**".

Далее, как обычно, форму называем **fMain**, сохраняем проект в папку **18-01** под именем **CodeBook**, модулю главной формы присваиваем имя `Main`.

Теперь займемся самой формой. Блокнот Windows имеет возможность менять свой размер, минимизировать и максимизировать окно, оставим такую возможность и нашему **Блокноту**. Однако по умолчанию, Lazarus выводит слишком маленькое окно. Давайте сделаем окно среднего размера. Пусть будет `Height = 500`, а `Width = 750`, и расположите его примерно посреди окна. В свойстве `Caption` напишите "`Блокнот - шифровальщик`".

Далее, установим на форму компонент `TMemo`, имя оставим по умолчанию - `Memo1`, свойство `Align` установите в `alClient`. Текст в окне слишком мелкий, откройте кнопкой "**...**" редактор свойства `Font` и выберите шрифт Times New Roman с размером 12 (или любой другой, какой вам понравится больше, только не забывайте, что не у всех пользователей хорошее зрение!). Теперь кнопкой "…" зайдите в редактор свойства `Lines` и сотрите там весь текст - нам вовсе не нужно, чтобы при открытии программы отображался текст "`Memo1`", правда?

Займемся остальными компонентами. Нам потребуются диалоги: TOpenDialog, TSaveDialog, TFontDialog и TColorDialog. Поскольку нам придется в коде обращаться к ним по имени, измените их свойства Name соответственно, на OD, SD, FD и CD.

Кроме того, для TOpenDialog и TSaveDialog сделайте следующие настройки:

```
Filter=Текстовые файлы|*.txt|Все файлы|*.*
DefaultExt=.txt
```

Ну, и напоследок, нам потребуется главное меню TMainMenu. Имя самого компонента можно оставить без изменений - MainMenu1, а вот пункты этого меню будем переименовывать. Некоторым подпунктам меню будем присваивать "горячие клавиши". Первым в таблице идут параметры разделов (выделяются жирным шрифтом), затем параметры их подразделов. Подпункты-разделители, в которых будет отражаться разделительная линия, будем называть нейтрально: N1, N2, и т.д. Итак, создаем следующие пункты и подпункты:

Таблица 18.1. Параметры разделов и подразделов главного меню

Name	Caption	ShortCut
Раздел "Файл"		
FileMenu	**Файл**	
FileCreate	Создать	**Ctrl+N**
FileOpen	Открыть	**Ctrl+O**
FileSave	Сохранить	**Ctrl+S**
FileSaveAs	Сохранить как…	
N1	-	
FileExit	Выход	
Раздел "Правка"		
EditMenu	**Правка**	
EditCancel	Отменить	**Ctrl+Z**
N2	-	
EditCut	Вырезать	**Ctrl+X**
EditCopy	Копировать	**Ctrl+C**
EditPaste	Вставить	**Ctrl+V**
EditDelete	Удалить	
N3	-	
EditSelectAll	Выделить всё	**Ctrl+A**
Раздел "Формат"		
FormatMenu	**Формат**	
FormatFont	Шрифт	
FormatColor	Цвет	
N4	-	
FormatWordWrap	Перенос по словам	

Раздел "Кодирование"		
CoderMenu	**Кодирование**	
CoderCode	Шифровать	
CoderDecode	Дешифровать	
Раздел "Справка"		
HelpMenu	**Справка**	
HelpAbout	О программе	

В результате у вас должно получиться такое меню:

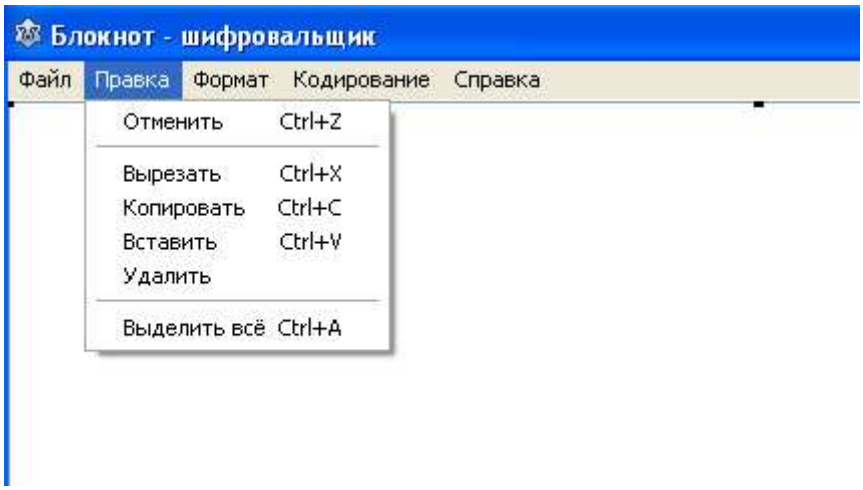

Рис. 18.1. Открытый раздел меню "Правка"

Нам нужно, чтобы по умолчанию, наш **Блокнот** делал перенос по словам. Для этого понадобится сделать три вещи:

1. Установить в `Memo1` вертикальную полосу прокрутки (`ScrollBars=ssVertical`).
2. Убедиться, что его свойство `WordWrap` (перенос по словам) установлено в `True`.
3. Поставить флажок на пункт меню "**Формат->Перенос по словам**". Для этого в **Редакторе меню** выделите данный пункт, и установите `True` в его свойстве `Checked`. Результат можно будет посмотреть, закрыв редактор меню, и открыв само главное меню:

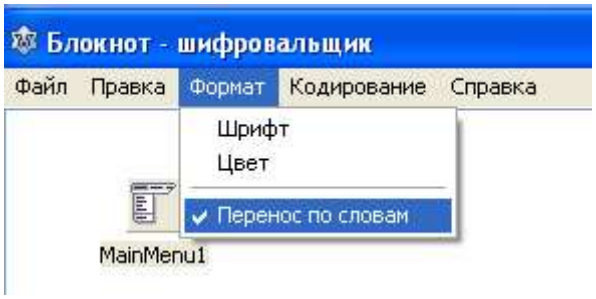

Рис. 18.2. Выделенный флажком пункт меню

Поскольку стандартный Блокнот Windows не имеет картинок в главном меню, мы их тоже привязывать не будем. Впрочем, если есть желание, вы можете самостоятельно добавить `TImageList` и украсить меню картинками.

Ну вот, все компоненты мы подготовили, и настроили, как нужно. Но прежде чем перейдем непосредственно к кодированию, подготовим модуль. Я обнаружил этот модуль в проекте DelphiWorld много лет назад, и несколько раз использовал, чтобы зашифровать ключи активации программ. Основная часть модуля написана на языке Ассемблер, однако для использования этого модуля знание Ассемблера нам не требуется. Поскольку автор выложил модуль в открытый доступ (спасибо ему большое за это), мы с вами можем использовать его в своих программах с чистой совестью.

Откройте любой редактор текстов, хотя бы тот же стандартный **Блокнот** Windows. Скопируйте в него нижеследующий листинг, строки

```
//------ Начало модуля ---------
//------- Конец модуля ---------
```

копировать в модуль не нужно они только обозначают границы листинга этого кода:

```
//------ Начало модуля ---------
{ **** UBPFD *********** by delphibase.endimus.com ****
>> Шифрование и дешифрование текстов по принципу S-Coder со скрытым ключом

После подключения модуля ключевое слово (фраза) будут в MyCript.MyPassword;
Шифровать примерно так:
  Memo1.Text:= MyCript.Write(MyCript.Encrypt(Memo1.Text));
Дешифровать примерно так:
  Memo1.Text:= MyCript.Decrypt(MyCript.Read(Memo1.Text));

Copyright:   EFD Systems http://www.mindspring.com/~efd
Дата:        23 мая 2002 г.
********************************************** }

unit MyCript;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

//Это просто интерфейс функций EnCipher и Crypt для шифрования строки текста:
function Encrypt(text: Ansistring): Ansistring;
//Это просто интерфейс функций EnCipher и Crypt для дешифрования строки
текста:
function Decrypt(text: Ansistring): Ansistring;

function Write(text: Ansistring): Ansistring;
function Read(text: Ansistring): Ansistring;

var
  MyPassword: AnsiString; //здесь будет пароль

implementation


procedure EnCipher(var Source: AnsiString);

{Low order, 7-bit ASCII (char. 32-127) encryption designed for database use.
 Control and high order (8 bit) characters are passed through unchanged.
```

```
 Uses a hybrid method...random table substitution with bit-mangled output.
 No passwords to worry with (the built-in table is the password). Not
industrial
 strength but enough to deter the casual hacker or snoop. Even repeating
char.
 sequences have little discernable pattern once encrypted.

 NOTE: When displaying encrypted strings, remember that some characters
       within the output range are interpreted by VCL components; for
       example, '&'.}
begin
{$asmmode intel}
asm
    Push ESI //Save the good stuff
    Push EDI

    Or EAX,EAX
    Jz @Done
    Push EAX
    Call UniqueString
    Pop EAX
    Mov ESI,[EAX] //String address into ESI
    Or ESI,ESI
    Jz @Done
    Mov ECX,[ESI-4] //String Length into ECX
    Jecxz @Done //Abort on null string
    Mov EDX,ECX //initialize EDX with length
    Lea EDI,@ECTbl //Table address into EDI
    Cld //make sure we go forward
@L1:
    Xor EAX,EAX
    Lodsb //Load a byte from string
    Sub AX,32 //Adjust to zero base
    Js @Next //Ignore if control char.
    Cmp AX,95
    Jg @Next //Ignore if high order char.
    Mov AL,[EDI+EAX] //get the table value
    Test CX,3 //screw it up some
    Jz @L2
    Rol EDX,3
@L2:
    And DL,31
    Xor AL,DL
    Add EDX,ECX
    Add EDX,EAX
    Add AL,32 //adjust to output range
    Mov [ESI-1],AL //write it back into string
@Next:
    Dec ECX
    Jnz @L1
// Loop @L1 //do it again if necessary

@Done:
    Pop EDI
    Pop ESI

    Jmp @Exit
// Ret //this does not work with Delphi 3 - EFD 971022

@ECTbl: //The encipher table
    DB 75,85,86,92,93,95,74,76,84,87,91,94
    DB 63,73,77,83,88,90,62,64,72,78,82,89
    DB 51,61,65,71,79,81,50,52,60,66,70,80
    DB 39,49,53,59,67,69,38,40,48,54,58,68
```

```
    DB 27,37,41,47,55,57,26,28,36,42,46,56
    DB 15,25,29,35,43,45,14,16,24,30,34,44
    DB 06,13,17,23,31,33,05,07,12,18,22,32
    DB 01,04,08,11,19,21,00,02,03,09,10,20
@Exit:

end;//asm
end;

procedure DeCipher(var Source: AnsiString);

{Decrypts a string previously encrypted with EnCipher.}
begin
{$asmmode intel}
asm
    Push ESI //Save the good stuff
    Push EDI
    Push EBX

    Or EAX,EAX
    Jz @Done
    Push EAX
    Call UniqueString
    Pop EAX
    Mov ESI,[EAX] //String address into ESI
    Or ESI,ESI
    Jz @Done
    Mov ECX,[ESI-4] //String Length into ECX
    Jecxz @Done //Abort on null string
    Mov EDX,ECX //Initialize EDX with length
    Lea EDI,@DCTbl //Table address into EDI
    Cld //make sure we go forward
@L1:
    Xor EAX,EAX
    Lodsb //Load a byte from string
    Sub AX,32 //Adjust to zero base
    Js @Next //Ignore if control char.
    Cmp AX,95
    Jg @Next //Ignore if high order char.
    Mov EBX,EAX //save to accumulate below
    Test CX,3 //unscrew it
    Jz @L2
    Rol EDX,3
@L2:
    And DL,31
    Xor AL,DL
    Add EDX,ECX
    Add EDX,EBX
    Mov AL,[EDI+EAX] //get the table value
    Add AL,32 //adjust to output range
    Mov [ESI-1],AL //store it back in string
@Next:
    Dec ECX
    Jnz @L1
// Loop @L1 //do it again if necessary

@Done:
    Pop EBX
    Pop EDI
    Pop ESI

    Jmp @Exit
// Ret Does not work with Delphi3 - EFD 971022
```

```
@DCTbl: //The decryption table
    DB 90,84,91,92,85,78,72,79,86,93,94,87
    DB 80,73,66,60,67,74,81,88,95,89,82,75
    DB 68,61,54,48,55,62,69,76,83,77,70,63
    DB 56,49,42,36,43,50,57,64,71,65,58,51
    DB 44,37,30,24,31,38,45,52,59,53,46,39
    DB 32,25,18,12,19,26,33,40,47,41,34,27
    DB 20,13,06,00,07,14,21,28,35,29,22,15
    DB 08,01,02,09,16,23,17,10,03,04,11,05
@Exit:
end;//asm
end;

procedure Crypt(var Source: Ansistring; const Key: AnsiString);

{Encrypt AND decrypt strings using an enhanced XOR technique similar to
 S-Coder (DDJ, Jan. 1990). To decrypt, simply re-apply the procedure
 using the same password key. This algorithm is reasonably secure on
 it's own; however,there are steps you can take to make it even more
 secure.

       1) Use a long key that is not easily guessed.
       2) Double or triple encrypt the string using different keys.
          To decrypt, re-apply the passwords in reverse order.
       3) Use EnCipher before using Crypt. To decrypt, re-apply Crypt
          first then use DeCipher.
       4) Some unique combination of the above

 NOTE: The resultant string may contain any character, 0..255.}
begin
{$asmmode intel}
asm
    Push ESI //Save the good stuff
    Push EDI
    Push EBX

    Or EAX,EAX
    Jz @Done
    Push EAX
    Push EDX
    Call UniqueString
    Pop EDX
    Pop EAX
    Mov EDI,[EAX] //String address into EDI
    Or EDI,EDI
    Jz @Done
    Mov ECX,[EDI-4] //String Length into ECX
    Jecxz @Done //Abort on null string
    Mov ESI,EDX //Key address into ESI
    Or ESI,ESI
    Jz @Done
    Mov EDX,[ESI-4] //Key Length into EDX
    Dec EDX //make zero based
    Js @Done //abort if zero key length
    Mov EBX,EDX //use EBX for rotation offset
    Mov AH,DL //seed with key length
    Cld //make sure we go forward
@L1:
    Test AH,8 //build stream char.
    Jnz @L3
    Xor AH,1
@L3:
    Not AH
    Ror AH,1
```

```
    Mov AL,[ESI+EBX] //Get next char. from Key
    Xor AL,AH //XOR key with stream to make pseudo-key
    Xor AL,[EDI] //XOR pseudo-key with Source
    Stosb //store it back
    Dec EBX //less than zero ?
    Jns @L2 //no, then skip
    Mov EBX,EDX //re-initialize Key offset
@L2:
    Dec ECX
    Jnz @L1
@Done:
    Pop EBX //restore the world
    Pop EDI
    Pop ESI
end;//asm
end;

function Encrypt(text: Ansistring): Ansistring;
//Это просто интерфейс функций EnCipher и Crypt для шифрования строки текста
begin
  {шифруем текст}
  EnCipher(Text);
  {зашифровываем ключом}
  Crypt(Text, MyPassword);
  Result := Text;
end;

function Decrypt(text: Ansistring): Ansistring;
//Это просто интерфейс функций EnCipher и Crypt для дешифрования строки
текста
begin
  {расшифровываем ключом}
  Crypt(Text, MyPassword);
  {расшифровываем результат}
  DeCipher(Text);
  Result := Text;
end;

function Write(text: Ansistring): Ansistring;
var
  i: integer;
begin
  Result := '';
  for i := 1 to Length(text) do
    {получаем hex код из текста}
    Result := Result + InttoHex(ord(text[i]), 2);
end;

function Read(text: Ansistring): Ansistring;
var
  i: integer;
begin
  Result := '';
  for I := 1 to Length(text) do
    if odd(i) then
      {получаем текст из hex кода}
      Result := Result + Chr(StrtoInt('$' + text[i] + text[i + 1]));
end;

end.
//------- Конец модуля ---------
```

Листинг . (html, txt)

Сохраните модуль в файл **MyCript.pas** туда, куда вы решили собирать свою коллекцию модулей. Не забудьте скопировать этот файл в папку с нашим текущим проектом.

Также в **Редакторе кода** нашей программы добавьте модуль `MyCript` в раздел `uses`, в конец списка подключаемых модулей.

Теперь можем приступить непосредственно к кодированию команд. Вы помните, что чтобы сгенерировать команду `OnClick` пункта меню, достаточно выбрать эту команду? Тогда начнем с более простых пунктов меню. Самым первым обработаем пункт "**Формат->Перенос по словам**":

```
procedure TfMain.FormatWordWrapClick(Sender: TObject);
begin
  //изменяем меню:
  FormatWordWrap.Checked:= not FormatWordWrap.Checked;
  //присваиваем настройку Memo1:
  Memo1.WordWrap:= FormatWordWrap.Checked;
  //если перенос по словам включен, нужна только вертикальная
  //линейка прокрутки, иначе нужны обе линейки:
  if Memo1.WordWrap then Memo1.ScrollBars:= ssVertical
  else Memo1.ScrollBars:= ssBoth;
end;
```

Мы с вами установили по умолчанию, что текст в `Memo1` переносится по словам, а данный пункт меню отмечен флажком. При выборе этой команды происходит обратное действие: состояние `Checked` пункта меню меняется на противоположное, состояние `WordWrap` компонента `Memo1` становится таким же. То есть, если флажок установлен, то и перенос будет работать, и наоборот. В заключение мы устанавливаем нужное состояние полос прокрутки. Если перенос работает, то горизонтальная полоса не нужна, только вертикальная. Если же переноса нет, то строка может получиться очень длинной, тут понадобятся обе полосы прокрутки.

Далее обработаем команды меню "**Правка**". Они очень простые, всего на команду придется ввести по одной строке кода, так что дополнительные комментарии не нужны. Меню "**Правка->Отменить**":

```
procedure TfMain.EditCancelClick(Sender: TObject);
begin
  Memo1.Undo;
end;

"Правка->Вырезать" не сложнее:
procedure TfMain.EditCutClick(Sender: TObject);
begin
  Memo1.CutToClipboard;
end;

"Правка->Копировать":
procedure TfMain.EditCopyClick(Sender: TObject);
begin
  Memo1.CopyToClipboard;
end;

"Правка->Вставить":
procedure TfMain.EditPasteClick(Sender: TObject);
begin
  Memo1.PasteFromClipboard;
end;
```

```
"Правка->Удалить":
procedure TfMain.EditDeleteClick(Sender: TObject);
begin
  Memo1.ClearSelection;
end;

"Правка->Выделить всё":
procedure TfMain.EditSelectAllClick(Sender: TObject);
begin
  Memo1.SelectAll;
end;
```

С пунктом "**Правка**" разобрались, вернемся к пункту "**Формат**". У нас остались необработанными два подпункта - "**Шрифт**" и "**Цвет**". Процедура `OnClick` для пункта "**Формат->Шрифт**":

```
procedure TfMain.FormatFontClick(Sender: TObject);
begin
  //сначала диалогу присваиваем шрифт как у Memo:
  FD.Font:= Memo1.Font;
  //если диалог прошел успешно, меняем шрифт у Memo:
  if FD.Execute then Memo1.Font:= FD.Font;
end;
```

Здесь мы сначала свойству `Font` диалога присвоили такой же шрифт, как и у `Memo1`. Зачем нужно было это делать? Если бы мы этого не сделали, диалог не знал бы, какой шрифт считается "текущим", там была бы пустая строка. И пользователю сложно было бы разобраться, какой шрифт, какого размера он сейчас использует. Теперь же, при вызове диалога, сразу будет выделен наш шрифт Times New Roman, 12. Если же пользователь изменит этот шрифт, то свойству `Font` компонента `Memo1` будет присвоен новый шрифт.

Примерно также обрабатываем и пункт "**Формат->Цвет**", только здесь мы обращаемся к свойству `Color` и диалогу `CD`:

```
procedure TfMain.FormatColorClick(Sender: TObject);
begin
  //сначала устанавливаем цвет диалога, как у Memo:
  CD.Color:= Memo1.Color;
  //если диалог прошел успешно, меняем цвет у Memo:
  if CD.Execute then Memo1.Color:= CD.Color;
end;
```

Теперь перейдем к более сложному пункту меню "**Файл**". Сначала самая простая команда, "**Файл->Выход**":

```
procedure TfMain.FileExitClick(Sender: TObject);
begin
  Close;
end;
```

Команда "**Файл->Сохранить как**" чуть сложнее:

```
procedure TfMain.FileSaveAsClick(Sender: TObject);
begin
  {Переписываем заголовок окна диалога, иначе он выйдет на английском.
   Если сохранение произошло, то свойство Modified у Memo переводим в false,
   так как все изменения уже сохранены}
  SD.Title:= 'Сохранить как';
  if SD.Execute then begin
    Memo1.Lines.SaveToFile(Utf8ToSys(SD.FileName));
    Memo1.Modified:= false;
  end; //if
end;
```

Здесь мы вначале изменили заголовок. Почему-то диалог, хоть и имеет по умолчанию текст в `Title` на русском языке, в работающей программе все равно выходит по-английски, хотя диалоги шрифта и цвета выводят нормальный русский заголовок. Может быть, в будущих версиях Lazarus эта недоработка будет устранена? Пока же будем менять заголовки внутри кода. Далее мы сохраняем текст из `Memo1` в файл, причем делаем это не так, как обычно:

```
Memo1.Lines.SaveToFile(SD.FileName);
```

а так:

```
Memo1.Lines.SaveToFile(Utf8ToSys(SD.FileName));
```

Зачем такие усложнения? Дело в том, что имя файла (свойство `FileName` диалога) сохраняется в формате UTF8, а процедура `SaveToFile` требует формата ANSI. Это не проблема, если вы будете давать файлу имя исключительно латинскими буквами или цифрами. Но если вы попробуйте сохранить файл с именем под русскими символами (или любыми неанглийскими), то получится то, что называют "кракозябрами" - непонятная абракадабра вместо имени. Такой файл и открыть потом не получится. Если же преобразовать имя файла в нужный формат, это гарантирует правильное сохранение (загрузку) этого файла, на каком бы языке вы не написали его имя.

Далее мы устанавливаем в `False` свойство `Modified` компонента `Memo1`. Это важно! Свойство `Modified` становится `True`, если в `Memo1` есть какие-то изменения. Если же мы эти изменения сохранили в файл, будем считать, что других изменений пока нет. Нам придется еще проверять состояние этого свойства.

Теперь обработаем команду "**Файл->Сохранить**":

```
procedure TfMain.FileSaveClick(Sender: TObject);
begin
  {если имя файла известно, то не нужно вызывать диалог SaveDialog,
    просто вызываем метод SaveToFile. }
  if SD.FileName <> '' then begin
    Memo1.Lines.SaveToFile(Utf8ToSys(SD.FileName));
    //устанавливаем Modified в false, так как изменения уже сохранили:
    Memo1.Modified:= false;
  end //if
  //иначе имя файла не известно, вызываем Сохранить как...:
  else FileSaveAsClick(Sender);
end;
```

Смотрите, что тут происходит. Мы смотрим, есть ли какой-нибудь текст в свойстве FileName диалога SD. Если текст есть (свойство FileName не равно пустой строке), значит, диалог SD уже вызывался, или же открывался существующий файл. В любом случае, мы знаем имя файла, куда нужно сохранять данный текст. Сохраняем, переводим в False свойство Modified нашего Memo1. Если же там текста нет, значит, текст новый, еще ни разу не сохранялся. В этом случае, нам нужно вызвать метод OnClick команды "**Файл->Сохранить как**". Обратите внимание, как мы это делаем:

```
FileSaveAsClick(Sender);
```

Дело в том, что в качестве параметра в метод нужно передать объект, откуда этот метод был вызван. Переменная Sender имеет тип TObject и обычно указывает на этот объект. Мы могли бы указать и какой то конкретный объект, например, так:

```
FileSaveAsClick(Memo1);
```

В данном случае не будет никакой разницы, команда все равно сработает одинаково.

Команда "**Файл->Создать**" чуть сложнее:

```
procedure TfMain.FileCreateClick(Sender: TObject);
begin
  {Если есть изменения текста, спросим пользователя, не хочет ли он сохранить
  их перед созданием нового текста}
  if Memo1.Modified then begin
    //если пользователь согласен сохранить изменения:
    if MessageDlg('Сохранение файла',
                  'Текущий файл был изменен. Сохранить изменения?',
          mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then
             FileSaveClick(Sender);
  end; //if
  //теперь очищаем Memo, если есть текст:
  if Memo1.Text <> '' then Memo1.Clear;
  //в SaveDialog убираем имя файла. это будет означать, что файл не сохранен:
  SD.FileName:= '';
end;
```

Когда пользователь выбирает команду "**Файл->Создать**", в Memo уже мог быть какой-то текст. Он нужен пользователю, или нам просто нужно очистить Memo, и подготовить его к новому тексту? Для этого, в случае, если у Memo есть изменения, мы выводим запрос-сообщение MessageDlg(). Если пользователь желает сохранить старый текст, мы вызываем команду "**Файл->Сохранить**". И в любом случае, затем мы очищаем Memo, если там что-то есть, и очищаем свойство FileName у диалога SD (TSaveDialog). Ведь текст новый, и мы еще не знаем, куда пользователь захочет его сохранить, верно?

Код команды "**Файл->Открыть**" самый длинный, хотя и в нем ничего сложного нет:

```
procedure TfMain.FileOpenClick(Sender: TObject);
begin
  //проверка необходимости сохранения файла, как в Файл->Создать:
```

```
  if Memo1.Modified then begin //изменения есть
    //если пользователь согласен сохранить изменения:
    if MessageDlg('Сохранение файла',
                  'Текущий файл был изменен. Сохранить изменения?',
          mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then
             FileSaveClick(Sender);
  end; //if

  //очищаем имя файла у диалога OpenDialog, изменяем заголовок, и
  //вызываем метод LoadFromFile, если диалог состоялся
  OD.FileName:= '';
  OD.Title:= 'Открыть существующий файл';
  if OD.Execute then begin
    //очищаем Memo, если есть текст:
    if Memo1.Text <> '' then Memo1.Clear;
    //читаем из файла
    Memo1.Lines.LoadFromFile(Utf8ToSys(OD.FileName));
    //копируем имя файла в диалог SaveDialog, чтобы потом знать,
    //куда сохранять:
    SD.FileName:= OD.FileName;
  end; //if
end;
```

В первой части процедуры мы смотрим, нет ли изменений в `Memo`, ведь пользователь мог редактировать один файл, а затем захотел открыть другой! Нужно ли сохранять эти изменения, если они есть? Как и в прошлом примере, для этого мы выводим запрос `MessageDlg()`. Сохраняем, если пользователь этого хочет.

Далее мы очищаем свойство `FileName` у диалога открытия файла. Зачем? Ладно, если этот диалог вызывается первый раз, тогда это свойство будет пусто. А если это не первый файл, который открывается пользователем? Если свойство не очистить, то при вызове диалога прежний файл будет там по умолчанию. Но пользователь то хочет открыть другой файл! Поэтому мы предварительно очищаем это свойство, а уж потом вызываем диалог. В коде мы изменяем заголовок и у этого диалога, иначе он тоже выйдет на английском языке.

Затем мы чистим `Memo`, если там есть старый текст, и загружаем новый текст из нового файла, после чего копируем имя этого файла в диалог сохранения, чтобы знать, куда сохранять изменения при выборе команды "**Файл->Сохранить**".

Теперь мы с вами должны предусмотреть еще вот что. Допустим, пользователь что-то записал в блокнот, а потом решил его закрыть. Изменения есть, а надо ли их сохранять? Нужно спросить об этом у пользователя. Это можно было бы сделать в команде "**Файл->Выход**", но мы этого не сделали. Почему? А если пользователь закроет программу не этой командой, а кнопкой с крестиком в верхней правой части окна? Или кнопками **<Alt+F4>**? Вот чтобы обработать закрытие программы, каким бы образом пользователь её не закрывал, воспользуемся событием `OnClose` формы главного окна, которое возникает ПЕРЕД закрытием программы. Проблема в том, что компонент `Memo1` закрывает всю форму, а нам нужно ее выделить. Выделить форму **fMain** можно либо в верхней части окна **Инспектора объектов**, где все объекты отображены в виде дерева - форма там расположена в самом верху, как главный, родительский компонент. Либо можно сделать проще - выделите `Memo1` и нажмите **<Esc>**, при этом выделение перейдет на внешний к `Memo1` компонент, то есть, к форме. Затем перейдите на вкладку **События Инспектора объектов**, и сгенерируйте событие `OnClose`. Его код вам уже знаком по предыдущему коду:

```
procedure TfMain.FormClose(Sender: TObject; var CloseAction: TCloseAction);
begin
```

```
  {Если есть изменения текста, спросим пользователя, не хочет ли он сохранить
  их перед созданием нового текста}
  if Memo1.Modified then begin
    //если пользователь согласен сохранить изменения:
    if MessageDlg('Сохранение файла',
                  'Текущий файл был изменен. Сохранить изменения?',
          mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then
             FileSaveClick(Sender);
  end; //if
end;
```

Листинг . (html, txt)

Теперь каким бы способом пользователь не закрывал программу, будет проверяться - нет ли изменений в `Memo`, и если есть, выйдет запрос - сохранять ли их. Практически всё, что умеет делать стандартный Блокнот, умеет теперь и наша программа. Самое время заняться шифрованием.

Код "**Кодирование->Шифровать**" очень простой:

```
procedure TfMain.CoderCodeClick(Sender: TObject);
begin
  //сначала очистим ключ:
  MyCript.MyPassword:= '';
  if InputQuery('Ввод ключа', 'Введите ключевое слово (фразу):',
    MyCript.MyPassword) then
      Memo1.Text:= MyCript.Write(MyCript.Encrypt(Memo1.Text));
end;
```

Вначале мы очищаем ключ. Делается это на всякий случай, вдруг это уже не первое шифрование, тогда в глобальной переменной `MyPassword` будет храниться предыдущий ключ. Или представьте такую ситуацию: пользователь вводил личный текст, ему понадобилось отлучиться, и он его зашифровал. Если ключ в переменной сохраняется, любой может подойти и расшифровать его. Элементарная безопасность требует, чтобы мы всегда очищали ключ.

Затем, с помощью `InputQuery()` мы выводим запрос, в котором пользователь может ввести пароль - ключевое слово или фразу. Этот ключ мы помещаем в переменную `MyPassword`, после чего вызываем шифрацию текста Memo так, как было указано в рекомендациях модуля `MyCript`. Дешифрация ("**Кодирование->Дешифровать**") происходит похожим образом:

```
procedure TfMain.CoderDecodeClick(Sender: TObject);
begin
  //сначала очистим ключ:
  MyCript.MyPassword:= '';
  if InputQuery('Ввод ключа', 'Введите ключевое слово (фразу):',
    MyCript.MyPassword) then
      Memo1.Text:= MyCript.Decrypt(MyCript.Read(Memo1.Text));
end;
```

Вот и весь шифратор. Сохраните его, скомпилируйте и опробуйте в работе. Имейте в виду, что если ввести неправильный пароль, то текст будет дешифрован неправильно и не прочитается. Обратного действия не предусмотрено в тех же целях безопасности.

Проведите нагрузочное и стресс - тестирование разработанного приложения. Проверьте, как влияет размер файла на производительность и работоспособность приложения. Установите, влияет ли многократное выполнение операций кодирования - декодирования на работоспособность приложения (работает ли освобождение памяти).

Оформите отчёт.