Базовый файловый ввод-вывод в Delphi

Каждый программист должен уметь работать с текстовыми файлами, файла­ми, содержащими записи, и файлами, которые не имеют определенной структуры или обрабатываются так, как если бы они не имели структуры. Вначале мы рассмотрим текстовые файлы, поскольку этот тип файлов используется наибо­лее часто.

Для получения доступа к текстовым файлам служат переменные типа Text:

var myFile: Text;

Прежде чем можно будет приступить к работе с файлом, с помощью процедуры AssignFile его нужно присвоить переменной типа Text. Эта процедура принима­ет два параметра: переменную файла и имя файла.

procedure AssignFile(var F; FileName: string); AssignFile(myFile, 'с:file:///C:/data/data.txt);

После того как имя файла присвоено переменной файла, необходимо опреде­лить действия, которые нужно выполнять с файлом. Файл можно подготовить к чтению, записи или дополнению.

 Чтобы подготовить файл к записи, необходимо использовать процедуру Rewrite, которая всегда создаст новый пустой файл. Если файл с таким же именем файла уже существует, процедура Rewrite вначале удаляет существующий файл, а затем заме­няет его новым пустым файлом. Затем процедура открывает файл и устанавливает указатель позиции на начало файла.

При работе с текстовыми файлами процедуре Rewrite необходимо передавать только переменную типа Text:

procedure Rewrite(var F: File [; Recsize: Word ]); Rewrite(myFile);

Когда файл открыт и готов к записи, для записи текста в текстовый файл можно использовать стандартную процедуру WriteLn. При выполнении записи в тексто­вый файл первым параметром, переданным процедуре WriteLn, должна быть пе­ременная файла:

procedure WriteLn([ var F: Text; ] PI [, P2, ...,Pn ]); WriteLn(myFile, 'cave canem');

По завершении работы с файлом его всегда следует закрывать, чтобы обеспе­чить корректное сохранение файла на диске и освободить любую память, занятую в процессе записи. Для закрытия файла служит процедура CloseFile, прини­мающая единственный параметр — файл, который нужно закрыть:

procedure CloseFile(var F); CloseFile(myFile);

Пример программы записи строки текста в текстовый файл приведен в листинге 8.1.

*Листинг 8.1. Запись текста в текстовый файл*

program Project1; {$APPTYPE CONSOLE} uses SysUtils; var myFile: Text; begin AssignFile(myFile, 'c:\data.txt'); Rewrite(myFile); WriteLn(myFile, 'cave canem'); CloseFile(myFile); end.

 Для подготовки файла к чтению используется процедура Reset. Эта процедура, подобно процедуреRewrite, принимает только параметр типа файла. Ее можно считать безопасной в том смысле, что она успешно работает, если дисковод и/или каталог, указанный в имени файла, существует. В отличие отRewrite, выполне­ние процедуры Reset будет невозможным, если файл, присвоенный переменной файла, не существует.

Для выполнения чтения данных из текстового файла можно использовать про­цедуру ReadLn. При этом первым передаваемым процедуре параметром должна быть переменная файла, а вторым — строковая переменная, которая будет вре­менно хранить значение, считанное из файла.

Следующий пример демонстрирует считывание значений из текстового файла и их отображение на экране.

Листинг 8.2. Считывание текста из текстового файла

program Project1; {$APPTYPE CONSOLE} uses SysUtils; var myFile: Text; line: string; begin AssignFile(myFile, 'c:\data.txt'); Reset(myFile); ReadLn(myFile, line); WriteLn(line); CloseFile(myFile); ReadLn; end.

Этот код будет успешно работать до тех пор, пока существует файл data. txt. Если этот файл не существует, программа даст сбой. Во избежание остановки при­ложения при отсутствии файла необходимо выполнять проверку успешности от­крытия файла с помощью процедуры Reset.

 Для выяснения наличия ошибок ввода-вывода необходимо непосредственно после вызова процедуры ввода-вывода, такой как Rewrite или Reset, вызвать функцию IOResult. Функция IOResult возвращает результат последней выпол­ненной операции ввода-вывода. Если IOResult возвращает 0, это означает, что операция была выполнена успешно.

Для выполнения проверки ввода-вывода с помощью функции IOResult необхо­димо вначале отключить автоматическую проверку ввода-вывода. Для включения и отключения проверки ошибок ввода-вывода служит директива компилятора SI. Обычно автоматическую проверку ввода-вывода отключают перед вызовом проце­дуры ввода-вывода и снова включают сразу после выполнения этого вызова:

{$1-} Вызов процедуры ввода-вывода {$1 + }

Следующий пример иллюстрирует выполнение проверки ввода-вывода и счи­тывание текста из файла только в случае успешного его открытия.

*Листинг 8.3. Проверка ошибок ввода-вывода*

program Project1;

{$APPTYPE CONSOLE}

uses

SysUtils;

var

myFile: Text;

line: string;

fileName: string;

begin

fileName := 'c:\data.txt';

AssignFile(myFile, fileName);

{$I-}

Reset(myFile);

{$I+}

if IOResult = 0 then

begin

ReadLn(myFile, line);

WriteLn(line);

CloseFile(myFile);

end else

WriteLn('Cannot open file: ', fileName);

ReadLn;

end.

Помните, что после обращения к процедуре ввода-вывода функцию IOResult можно вызывать только один раз. Это обусловлено тем, что она сбрасывает ре­зультат выполнения последней операции ввода-вывода в 0. Поэтому, если вызвать функцию IOResult дважды подряд, первое обращение к ней правильно сообщит об ошибке, но второе обращение сообщит (ошибочно), что операция была выпол­нена успешно.

 Когда нужно выполнить считывание всего текстового файла, его необходимо счи­тывать последовательно до достижения конца файла. Для определения конца файла служит функция Eof. Эта функция принимает единственный параметр файла и воз­вращает булевское значение, указывающее, был ли достигнут конец файла:

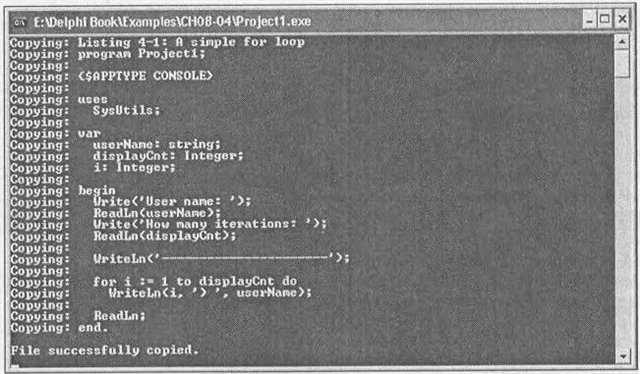
function Eof [ (var F: Text) ]: Boolean;

Как правило, наиболее рациональный способ считывания текстового файла предусматривает использование [цикл](http://delphicomponent.ru/)а while, продолжающего итерации вплоть до достижения конца файла. Следующий пример иллюстрирует копирование со­держимого одного текстового файла в другой с использованием[цикл](http://delphicomponent.ru/)а while not Eof (результаты можно видеть на рис. 8.1).

*Листинг 8.4. Копирование текстового файла*

program Project1; {$APPTYPE CONSOLE} uses SysUtils; var srcFile: Text; destFile: Text; line: string; begin { try to open the source file } AssignFile(srcFile, 'c:\data.txt'); {$I-} Reset(srcFile); {$I+} if IOResult = 0 then begin { try to open the destination file } AssignFile(destFile, 'c:\copy.txt'); {$I-} Rewrite(destFile); {$I+} if IOResult = 0 then begin { while loop that copies text from source to dest } while not Eof(srcFile) do begin ReadLn(srcFile, line); WriteLn(destFile, line); WriteLn('Copying: ', line); end; // while not Eof CloseFile(destFile); WriteLn; WriteLn('File successfully copied.'); end; // if destFile is OK CloseFile(srcFile); end; // if srcFile is OK ReadLn; end.

*Рис. 8.1. Копирование текстового файла*



Простейший способ считывания всего файла в память — его считывание в ди­намический массив строк. Для этого необходимо знать количество строк текста в файле. Поскольку никаких стандартных функций для выполнения этой задачи не существует, потребуется создать такую функцию. Для вычисления количества строк в текстовом файле необходимо в [цикл](http://delphicomponent.ru/)е while not Eof подсчитать количе­ство строк в файле, а затем вызвать процедуру Reset, чтобы вернуться к началь­ной позиции файла.

В следующем примере [цикл](http://delphicomponent.ru/) while not Eof в функции GetLineCount исполь­зуется для получения количества строк текстового файла. Полученный результат затем используется в вызове функцииSetLength для изменения размера динами­ческого массива.

*Листинг 8.5. Загрузка текстового файла в динамический массив*

program Project1;

{$APPTYPE CONSOLE}

uses

SysUtils;

var

myFile: Text;

lines: array of string;

cnt: Integer;

fileName: string;

function GetLineCount(var ATextFile: Text): Integer;

begin

Result := 0;

while not Eof(ATextFile) do

begin

ReadLn(ATextFile);

Inc(Result);

end;

Reset(ATextFile); { move position to the beginning }

end;

begin

fileName := 'c:\data.txt';

AssignFile(myFile, fileName);

{$I-}

Reset(myFile);

{$I+}

if IOResult = 0 then

begin

{ resize dynamic array and load the lines into the array}

SetLength(lines, GetLineCount(myFile));

for cnt := Low(lines) to High(lines) do

ReadLn(myFile, lines[cnt]);

{ close file }

CloseFile(myFile);

{ work with strings in memory }

for cnt := Low(lines) to High(lines) do

WriteLn(UpperCase(lines[cnt]));

end else

WriteLn('Cannot open file: ', fileName);

ReadLn;

end.

Как только приложение загрузило весь текстовый файл в динамический мас­сив, файл можно закрыть и продолжить работу со строками в памяти. Конечно, со строками в массиве можно выполнять любые действия, но этот код всего лишь с помощью функции Uppercase преобразует (временно) символы строк в прописные и отображает их на экране.

 Типизированные файлы представляют собой двоичные файлы, которые содер­жат элементы одинакового размера. Обычно типизированные файлы — это фай­лы, содержащие записи. Чтобы создать типизированный файл, вначале, исполь­зуя следующий синтаксис, необходимо создать новый файловый тип данных:

type NewFileType=file of DataType;

В следующем коде продемонстрировано создание нового файлового типа, кото­рый можно использовать для считывания и записи записей в типизированном файле:

type TPerson = record FirstName: string[20]; LastName: string[30]; Age: Integer; end; TPersonFile = file of TPerson;

Обратите внимание, что строковые поля в объявлении записи имеют явно оп­ределенную длину. Длина строки должна быть определена явно, поскольку размер всей записи должен оставаться постоянным. Если требуется сохранить запись в файле на диске, обычные строки использовать нельзя, поскольку их длина может изменяться в любое время и компилятор не может определить их длину во время компиляции.

Существует несколько различий между текстовыми и типизированными фай­лами:

* При сбросе типизированного файла в нем можно выполнять считывание и запись (при сбросе текстового файла в нем можно выполнять только считы­вание).
* При выполнении считывания или записи из типизированного файла необ­ходимо использовать процедуры Read и Write, а не ReadLn и WriteLn.

В листинге 8.6 демонстрируется работу с типизированными файлами.

*Листинг 8.6. Работа с типизированными файлами*

program Project1;

{$APPTYPE CONSOLE}

uses

SysUtils;

type

TPerson = record

FirstName: string[20];

LastName: string[30];

Age: Integer;

end;

TPersonFile = file of TPerson;

procedure ReadRecord(const AFileName: string; var Rec: TPerson);

var

F: TPersonFile;

begin

AssignFile(F, AFileName);

{$I-}

Reset(F);

{$I+}

if IOResult = 0 then

begin

Read(F, Rec);

CloseFile(F);

end; // if IOResult

end;

procedure WriteRecord(const AFileName: string; var Rec: TPerson);

var

F: TPersonFile;

begin

AssignFile(F, AFileName);

Rewrite(F);

Write(F, Rec);

CloseFile(F);

end;

procedure DisplayRecord(var Rec: TPerson);

begin

WriteLn('First Name: ', Rec.FirstName);

WriteLn('Last Name: ', Rec.LastName);

WriteLn('Age: ', Rec.Age);

WriteLn;

end;

var

TestRec: TPerson;

ReadRec: TPerson;

begin

TestRec.FirstName := 'Stephen';

TestRec.LastName := 'King';

TestRec.Age := 58;

WriteRecord('c:\info.dat', TestRec);

ReadRecord('c:\info.dat', ReadRec);

DisplayRecord(ReadRec);

ReadLn;

end.

Еще одно существенное различие между текстовыми и типизированными фай­лами состоит в том, что в то время как текстовые файлы допускают только после­довательный доступ, типизированные файлы допускают произвольный доступ к записям файла. Произвольный доступ возможен вследствие того, что все записи в файле имеют одинаковый размер, поэтому для считывания конкретной записи процедуре достаточно пропустить определенное, легко определяемое количество байт.

Для определения количества записей в файле можно использовать функцию FileSize. Для перехода к определенной записи в файле можно использовать про­цедуру Seek. Эта процедура принимает два параметра: переменную файла и цело­численное значение, указывающее номер записи (начиная с О), к которой необхо­димо выполнить переход.

var F: TPersonFile; RecCount: Integer; begin RecCount := FileSize(F); if RecCount = 0 then WriteLn(<sup>1</sup> Файл пуст') else Seek(F, FileSize(F)); end.

**Нетипизированные файлы** — это файлы без определенной структуры. В общем случае, нетипизированные файлы — это типизированные файлы, в которых вме­сто записей используются байты. Объявление переменной нетипизированного файла выглядит следующим образом:

var F: file;

При работе с нетипизированными файлами обращения к процедурам Reset и Rewrite выглядят несколько иначе. Обычно обе эти процедуры в качестве задан­ного по умолчанию размера записи используют 128байт. При работе с нетипизи­рованными файлами этот размер должен быть установлен равным 1 байту. Это можно выполнить, передавая 1 в качестве второго параметра в обоих вызовах:

Reset (F, 1); Rewrite(F,1);

Для считывания и записи данных в нетипизированные файлы применяются процедуры BlockRead иBlockWrite. Объявления этих процедур показаны ниже:

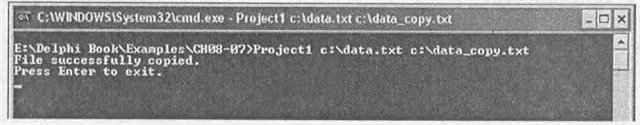
procedure BlockRead(var F: File; var Buf; Count: Integer [; var AmtTransferred : Integer]); procedure BlockWrite(var F: File; var Buf; Count: Integer [; var AmtTransferred: Integer]);

Первый параметр — переменная нетипизированного файла, используемая для доступа к файлу на диске. Второй параметр — буфер, который процедуры исполь­зуют для передачи данных в файл и из него.

Обычно этот буфер представляет собой статический массив байтов, но он мо­жет быть также и записью. Третий параметр указывает количество передаваемых байтов. Обычно это число равно размеру массива, который легко определить с по­мощью функции SizeOf. Необязательный параметр-переменнаяAmtTransferred можно использовать для отслеживания точного числа байтов, переданных в файл и из него.

Приложение, код которого представлен в листинге 8.7, использует процедуры BlockRead и BlockWrite для предоставления пользователю возможности копиро­вания файлов. Приложение также дает пользователю возможность указывать имена исходного и целевого файлов в командной строке, как показано на рис. 8.2.

*Рис. 8.2. Передача параметров приложению*



*Листинг 8.7. Копирование файлов с помощью процедур BlockRead и BlockWrite*

program Project1; {$APPTYPE CONSOLE} uses SysUtils; procedure BlockCopyFile(const SrcPath, DestPath: string); var Src: file; Dest: file; Buffer: array[1..1024] of Byte; BytesRead: Integer; begin if LowerCase(SrcPath) = LowerCase(DestPath) then Exit; AssignFile(Src, SrcPath); {$I-} Reset(Src, 1); {$I+} if IOResult = 0 then begin AssignFile(Dest, DestPath); {$I-} Rewrite(Dest, 1); {$I+} if IOResult = 0 then begin BytesRead := -1; while BytesRead <> 0 do begin BlockRead(Src, Buffer, SizeOf(Buffer), BytesRead); BlockWrite(Dest, Buffer, BytesRead); end; CloseFile(Dest); WriteLn('File successfully copied.'); end; // if Dest Rewrite CloseFile(Src); end; // if Source Reset end; var SourcePath: string; DestPath: string; begin { Accept parameters } if ParamCount = 2 then BlockCopyFile(ParamStr(1), ParamStr(2)) else begin { if there are no parameters in the command line, ask the user to enter filenames here } Write('Source path: '); ReadLn(SourcePath); Write('Destination path: '); ReadLn(DestPath); if (SourcePath <> '') and (SourcePath <> '') then BlockCopyFile(SourcePath, DestPath); end; WriteLn('Press Enter to exit.'); ReadLn; end.

Для выяснения количества параметров, которые пользователь передал прило­жению, применяется функция ParamCount. Эта функция не принимает никаких параметров и возвращает значение 0. если пользователь вызвал приложение без дополнительных параметров.

Для считывания параметров, переданных приложению, служит функция ParamStr, которая принимает единственный параметр типа Integer — индекс параметра. Если передать этой функции значение О, она возвратит путь и имя файла приложения. Индексы пользовательских параметров, если они переданы, начинаются с 1.

Вначале приложение проверяет, передал ли пользователь два параметра в ко­мандной строке. Если да, то ParamStr (1) содержит путь к исходному файлу, а ParamStr (2) —путь к файлу назначения.

Копирование выполняется в процедуре BlockCopyFile.

Ее первая строка:

if Lowercase(SrcPath) = Lowercase(DestPath) then Exit;

использует функцию LowerCase для временного преобразования имен обоих фай­лов в строчные буквы и проверяет, указывают ли оба имени файлов на один и тот же файл. Если файл назначения и файл-источник совпадают, никакое копирова­ние не требуется и оператор if-then вызывает процедуру Exit для выхода из про­цедуры.

Основная часть процедуры BlockCopyFile — [цикл](http://delphicomponent.ru/) while, который вызывает процедуры BlockRead иBlockWrite:

while BytesRead о 0 do begin BlockRead(Src, Buffer, SizeOf(Buffer), BytesRead); BlockWrite(Dest, Buffer, BytesRead); end;

Процедура BlockRead считывает из файла 1 Кбайт данных, записывает эти данные в массив Buffer и обновляет переменную BytesRead, которая всегда со­держит точное количество переданных байтов. Выполнение [цикл](http://delphicomponent.ru/)а будет повто­ряться до тех пор, пока процедура BlockRead продолжает считывать данные из файла. Когда эта процедура достигнет конца файла, значение переменной BytesRead будет сброшено в 0, условие [цикл](http://delphicomponent.ru/)а while станет ложным и копирова­ние файла завершится.

**Ввод и вывод**

|  |  |  |
| --- | --- | --- |
| **Автор:** | **Ерёмин Андрей** | *Начинать поиски надо с самого неподходящего места.* |
| **Номер урока:** | **13** |  |

Знаний, полученных в предыдущих уроках, достаточно, чтобы приступить к изучению механизмов ввода и вывода данных. Общая схема работы программы примерно такова: пользователь вводит данные, программа их считывает, производит с ними какие-то операции и выдаёт результат. Упрощённо: ввод, обработка, вывод. Что мы уже умеем? Для ввода данных нам понадобятся такие элементы как текстовое поле ввода (TEdit) или многострочное поле - TMemo, CheckBox или RadioButton для выбора параметров и др. Для вывода можно использовать текстовую метку (TLabel). Теперь разберём программную часть. Мы уже умеем описывать переменные, знаем, что такое тип данных и какие бывают типы (числа, строки и т.п.) Осталось лишь выяснить как связать визуальные элементы с программным кодом. В примерах, рассмотренных в предыдущих уроках, мы задавали значения переменным вручную (в коде), но это конечно же неприемлемо ни для одной программы. Программа должна быть универсальной и обрабатывать те данные, которые передал пользователь. Для ввода и вывода будут использоваться некоторые функции. Что такое функции, зачем они нужны, и как с ними работать, мы рассмотрели на прошлом уроке. Ну что, ближе к делу?

**Ввод на примере TEdit**

Текстовое поле ввода (TEdit) - простой и в то же время удобный элемент для ввода данных. В поле мы можем ввести и число и текст - никаких ограничений нет. Итак, наша первая задача - узнать, что введено в поле. Для этого мы должны обратиться к свойствам этого объекта. За текст, находящийся в поле, отвечает свойство **Text**. Именно им мы и воспользуемся. Попробуем считать число, которое будет введено... Расположим на форме само поле (TEdit) и кнопку (TButton). Создадим обработчик нажатия кнопки и сделаем следующее: объявим переменную целочисленного типа, а затем попробуем поместить в неё значение, которое введено в поле.

**procedure**TForm1.Button1Click(Sender: TObject);   
**var**A: Integer;   
**begin**   
  A:=Edit1.Text;   
**end**;

Казалоось бы, всё верно - переменную мы объявили, и просто хотим узнать, что введено в поле. Попробуем запустить (**F9**) - всё не так просто! Компилятор сообщает об ошибке:

[Error] Unit1.pas: Incompatible types: 'Integer' and 'TCaption'

Здесь сказано, что типы данных *Integer* и *TCaption* не соответствуют друг другу. *TCaption* - это тип данных свойства *Text* у TEdit. Этот тип "обозначает" текстовую строку, т.е. любые символы. Становится понятнее? В поле ввода мы можем ввести не только цифры, но и буквы и даже любые знаки, какие есть на клавиатуре (а на самом деле - вообще любые символы из кодовой таблицы). Однако наша переменная - число! Получается, что мы хотим в качестве числа записать символы. Непорядок! Да, вы скажете: но мы ведь будем вводить в поле цифры, а не буквы... Верно, но знает ли об этом компилятор? Конечно нет. А уговорить его не удастся :-)

Несоответствие типов данных - частая ошибка. Преобразовывать одни типы данных в другие требуется очень часто. Наш пример - яркое тому подтверждение. Чтобы работать с числом, мы должны введённый текст (именно текст) преобразовать в число.

Преобразования могут выполняться разными способами. Основной из них - применение специальных функций. Функции принимают один из несколько параметров (аргументов), одним из которых является переменная или значение, подлежащее преобразованию. Функция выполняет какие-то операции и возвращает значение нужного нам типа.

Для преобразования текста (строки) в целое число служит функция **StrToInt()**. Название абсолютно точно отражает назначение функции. У функции единственный параметр - строка, а на выходе получается число. Поэтому, чтобы наш пример работал, мы должны написать так:

**procedure**TForm1.Button1Click(Sender: TObject);   
**var**A: Integer;   
**begin**   
  A:=StrToInt(Edit1.Text);   
**end**;

Теперь программа запустится. Попробуем ввести число и нажать кнопку - никакой ответной реакции не наблюдается. Это верно, ведь мы кроме считывания введённого числа ничего не запрограммировали. А теперь попробуем ввести посторонние символы (т.е. не цифры) - программа "вылетит" с сообщением об ошибке:



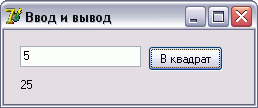
В сообщении указана и причина возникновения ошибки: "*'abc' - неверное целое число*". Как избегать подобных сообщений, а вместо этого выполнять свои действия, мы разберёмся позже.

**Вывод данных на примере TLabel**

Прежде чем делать вывод, давайте произведём с введённым числом какие-нибудь операции. К примеру, давайте возведём число в квадрат. Как это делается, вы уже знаете. Вывод осуществим в текстовую метку (TLabel). За текст метки отвечает свойство **Caption**. В данном случае наши действия должны быть обратными: при вводе мы заносили в переменную введённое в поле значение, а при выводе мы должны значение из переменной перенести в элемент. *Caption* у TLabel описывается тоже типом данных *TCaption*, т.е. это текст. А значит мы должны сделать обратное преобразование, т.е. преобразовать число в текстовую строку. Как называется функция для такого перевода, легко догадаться - **IntToStr()**. Финальный вариант обработчика кнопки:

**procedure**TForm1.Button1Click(Sender: TObject);   
**var**A: Integer;   
**begin**   
  A:=StrToInt(Edit1.Text);   
  A:=Sqr(A);   
  Label1.Caption:=IntToStr(A)   
**end**;

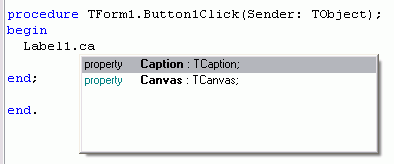
Запускаем программу и пробуем:



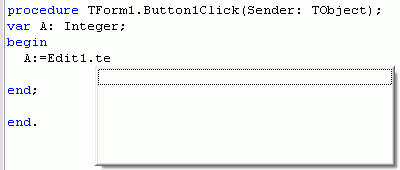
Простейшая программа ввода-вывода готова.

**Где узнать о типах данных свойств?**

Очень логичный вопрос - где узнать, какой тип данных имеет свойство элемента, кроме как смотреть сообщения об ошибках? Ответ прост - нам снова поможет редактор кода. После того, как мы ввели имя объекта и поставили точку, появляется список всех доступных команд. Напротив свойств через двоеточие указывается и тип данных:



Обратите внимание, что редактор автоматически отслеживает, какие команды содержатся в коде и в этом списке всегда находятся только те пункты, которые применимы в данной ситуации. Так, если написать "*A:=Edit1.*", то в списке будет отсутствовать свойство *Text*, ведь тип переменной *A* и тип этого свойства не совпадают:



При написании функции *StrToInt()* строка свойства *Text* конечно же появится.

**Обобщение сказанного**

Мы рассмотрели механизм ввода-вывода на примере обработки числа. Если обработке подлежит текст, то преобразовывать типы данных не придётся. Подобным образом осуществляется ввод и вывод практически любых данных, главное - выполнять преобразование одних типов в другие.

Для ввода целых чисел также есть удобный элемент - **TSpinEdit**. Расположен он на вкладке **Samples**. По внешнему виду похож на TEdit, только в поле ещё находятся две кнопки со стрелками - для изменения значения. Допустимые границы числа задаются свойствами *MinValue* и *MaxValue*, а само значение хранится в свойстве *Value*.

TSpinEdit

**Работа с вещественными числами**

Следует упомянуть и о дробных числах, ведь иногда без них не обойтись. В целом, их ввод и вывод практически такой же, как и целых чисел, только используются функции **StrToFloat()** и **FloatToStr()**. Всё просто. Однако с вещественными числами есть одна проблема - разделитель целой и дробной части. Дело в том, что нет определённости, какой символ считать разделителем - точку или запятую. Какой символ выбирается - зависит от настроек операционной системы (в Windows этот знак определяется в *Панели управления* в разделе *Язык и региональные стандарты*). Таким образом, если заточить программу под запятую, то на системах, где установлена точка, работать ничего не будет и будут возникать ошибки во время работы. В то же время, под точку тоже нельзя затачивать программу, ведь может стоять и запятая. К счастью, решение есть. Есть переменная, которую нигде не нужно объявлять, но значение которой можно менять - это переменная **DecimalSeparator**. Она объявлена в модуле *SysUtils* и хранит тот символ, который будет считаться разделителем. Установить нужное значение достаточно один раз - например при запуске программы. Вот так можно установить разделителем точку:

**procedure**TForm1.FormCreate(Sender: TObject);   
**begin**   
  DecimalSeparator:='.';   
**end**;

**Другие способы вывода**

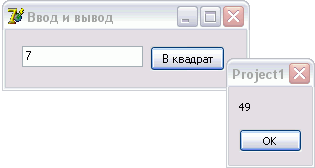
Помимо вывода в объекты (например, в TLabel) иногда удобно выводить данные в виде маленьких отдельных окон. Для этой цели существуют стандартные команды. В предыдущих уроках мы с ними уже встречались, но не разбирали их подробно.

**1. Простое окно с сообщением - ShowMessage()**

Самое примитивное окно содержит указанный текст и кнопку *OK* для закрытия окна. Вызвать такое окно можно процедурой **ShowMessage()**, параметром которой является текст - он и будет отображён в окне. Модифицируем нашу мини-программу, убрав TLabel и заменив строку вывода:

**procedure**TForm1.Button1Click(Sender: TObject);   
**var**A: Integer;   
**begin**   
  A:=StrToInt(Edit1.Text);   
  A:=Sqr(A);   
  **ShowMessage(IntToStr(A));**  
**end**;

В этом случае мы увидим следующее:



Следует обратить внимание на то, что пока окно сообщения находится на экране, интерфейс программы блокируется, а также прекращается выполнение дальнейших команд. Только после нажатия на кнопку программа продолжит свою работу.

**2. Диалоговое окно - MessageDlg()**

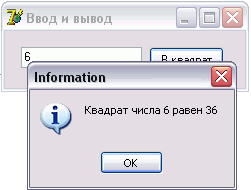
Диалоговые окна - более сложный тип окон. Диалоговые окна часто используются для "общения" с пользователем. Однажды такое окно нами уже использовалось - при закрытии окна программы мы спрашивали у пользователя, действительно ли окно нужно закрыть. Теперь разберём работу с такими окнами подробно.

Создаются диалоговые окна функцией **MessageDlg()**. Это именно функция, а не процедура. Возвращаемым значением является кнопка, которую нажал пользователь. У функции 4 входных параметра:

* **Текст сообщения** (тип данных - *String*);
* **Тип диалогового окна** (специальный тип данных - *TMsgDlgType*) - указывает на значок, расположенный в окне и на заголовок окна. Этот параметр задаётся одной из следующих констант:
  + **mtInformation** - информационное окно (значок в окне - буква "i");
  + **mtConfirmation** - окно с вопросом (значок - воспросительный знак);
  + **mtWarning** - предупреждающее окно (значок - восклицательный знак);
  + **mtError** - окно с сообщением об ошибке (значок - крест на красном фоне);
  + **mtCustom** - окно без значка, а в заголовке - название исполняемого файла программы (все остальные типы в заголовок помещают соответствующее типу диалога название - *Information*, *Warning* и т.д.)
* **Кнопки**, которые будут показаны в окне. Каждой кнопке также соответствует определённая константа. Кнопки перечисляются через запятую, а весь этот "комплект" обрамляется квадратными скобками (что означают квадратные скобки - узнаем позже). Вот константы всех доступных кнопок: *mbYes*, *mbNo*, *mbOK*, *mbCancel*, *mbAbort*, *mbRetry*, *mbIgnore*, *mbAll*, *mbNoToAll*, *mbYesToAll*, *mbHelp*. Название константы говорит о названии самой кнопки.
* Четвёртый параметр указывает на **индекс раздела в справочной системе**, соответствующий данному диалогу. Как правило, не используется и задаётся просто **0**.

В качестве возвращаемых функцией значений служат все те же константы кнопок, только с тем отличием, что вместо "*mb*" они начинаются на "*mr*" (сокращения от "*modal button*" и "*modal result*" соответственно).

Изменим нашу программу - пусть результат выдаётся в информационном окошке. Обратите внимание, насколько красивее выглядит такое окно:



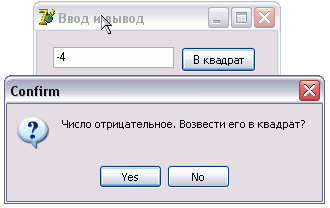
**procedure**TForm1.Button1Click(Sender: TObject);   
**var**A: Integer;   
**begin**   
  A:=StrToInt(Edit1.Text);   
  MessageDlg('Квадрат числа '+IntToStr(A)+' равен '+IntToStr(Sqr(A)),mtInformation,[mbOK],0);   
**end**;

**Примечание:** Текстовые строки можно объединять, просто складывая их - знак "+" (см. фрагмент кода выше). Сами строки записываются в одиночных кавычках.

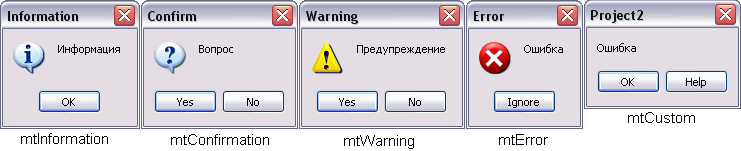
Ну и напоследок попробуем использовать диалог для уточнения у пользователя дополнительной информации. К примеру, если введено отрицательное число, будем спрашивать, нужно ли его возвести в квадрат... Код получится приблизительно таким:

**procedure**TForm1.Button1Click(Sender: TObject);   
**var**A: Integer;   
**begin**   
  A:=StrToInt(Edit1.Text);   
  **if** A < 0 **then**   
    **if** MessageDlg('Число отрицательное. Возвести его в квадрат?',mtConfirmation,[mbYes,mbNo],0) = mrYes **then**   
      MessageDlg('Квадрат числа '+IntToStr(A)+' равен '+IntToStr(Sqr(A)),mtInformation,[mbOK],0);   
**end**;

А результат будем таким:



Внешний вид окон всех типов:



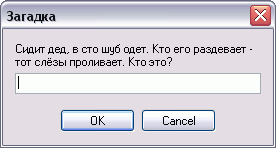
**3. Диалоговые окна MessageBox()**

C помощью фунции **MessageBox()** можно также выводить диалоговые окна. По внешнему виду они практически ничем не отличаются от окон *MessageDlg()*, однако есть возможность указать и заголовок окна. Для формирования строки кода с использованием *MessageBox()* есть программа [**Master MessageBox**](http://www.delphi.int.ru/download/file/107/). C помощью неё можно быстро создать нужное диалоговое окно. Проблема *MessageBox()* в том, что тип диалога и кнопки задаются немного необычным образом.

**Ввод с помощью диалогового окна**

Специальные окошки существуют не только для вывода на экран, но и для ввода. Примером такого окна является **InputBox()**. Функции передаётся 3 параметра: текст заголовка окна, текст пояснения и значение, находящееся в поле при показе окна на экран. Введённую в поле строку функция возвращает как значение-результат. Пример:

**procedure**TForm1.Button2Click(Sender: TObject);   
**begin**   
  **if** InputBox('Загадка','Сидит дед, в сто шуб одет. Кто его раздевает - тот слёзы проливает. Кто это?','') = 'лук' **then**   
    MessageDlg('Правильно!',mtInformation,[mbOk],0)   
  **else**   
    MessageDlg('Вы не угадали.',mtWarning,[mbOk],0)   
**end**;



**Заключение**

Мы рассмотрели способы ввода и вывода данных, а также познакомились с диалоговыми окнами. Приёмы ввода-вывода используются практически в любом приложении: в тех, которые имеют оконный интерфейс, это делается с использованием визуальных объектов, а для консольных приложений - в режиме командной строки.

**Домашнее задание**

# Borland Delphi. "Обработчик событий"

Сопровождается показом презентации [(Приложение1).](http://festival.1september.ru/articles/520215/pril1.ppt)

Основная идея ООП связана не с выполнением некоторой, заранее написанной последовательности инструкций, а с реагированием на те или иные события. Если не происходит ни одного события из тех, на которые предусмотрена реакция в приложении, оно ничего и не выполняет. Нажали клавишу - приложение отреагировало: его обработчик событий вывел что-либо на экран компьютера, что-то напечатал, то есть вызвал какую-либо процедуру.

**Обработчик событий**

Любой объект реальной жизни вынужден реагировать на те или иные события окружающего мира: зазвонил телефон - человек берёт трубку, пошёл дождь - мы открываем зонт.

С компьютером тоже связаны различные события: включение питания, нажатие клавиши на клавиатуре, щелчок кнопкой мыши и пр. Среда Delphi предоставляет возможность своим клиентам реагировать на них: она по указанию программиста устанавливает связь специальных процедур компонента, называемых обработчиками событий, с соответствующим механизмом операционной системы.

*Событие*- это реакция компонента на определённое воздействие.

Windows первой узнает о любом событии, и если связь с обработчиком такого события установлена, передаёт ему управление.

В общем случае это выглядит следующим образом:



На генерацию события программист обычными средствами повлиять не может. В его компетенции только обработать событие, то есть написать процедуру, которая будет выполняться при его возникновении.

Среда Delphi хранит для каждого компонента имена всех его обработчиков событий и выводит их на экран в окне "Инспектор объектов" (Object Inspector) во вкладке "События" (Events).

Для того, чтобы создать обработчик события, необходимо выполнить следующие действия:

* выберите компонент, для которого необходимо обработать событие.
* откройте вкладку "События" (Events) и два раза щелкните по правому столбцу выбранного события.

В результате этих действий Delphi автоматически сгенерирует процедуру

обработки события. Имя этой процедуры состоит из имени компонента и названия события. Оно может быть изменено пользователем непосредственно в окне "Инспектор объектов".

Для удаления события нужно только очистить содержимое процедуры (код между словами Begin и End), а при следующей компиляции, Delphi сам удалит ненужный код процедуры.

**Событие OnClick**

Самый простой способ сообщить приложению о событии - это щёлкнуть мышью на каком-либо компоненте. При щелчке по кнопке мышью в работающей программе возникает событие OnClick (по щелчку). Пока это событие не обрабатывается программой, поэтому нажатие кнопки не приведёт ни к каким последствиям. Чтобы заставить программу реагировать на нажатие кнопки, необходимо написать на языке Object Pascal фрагмент программы, который называется обработчиком события.

Этот фрагмент должен представлять собой последовательность текстовых строк, в которых программист указывает, что именно должна делать программа в ответ на нажатие кнопки. Фрагмент оформляется в виде процедуры на языке Pascal.

Чтобы заставить Delphi самостоятельно сделать заготовку для процедуры обработчика события OnClick, необходимо дважды щёлкнуть по вставленному компоненту. В ответ Delphi активизирует окно кода, в котором можно увидеть следующий фрагмент:

procedure TForm1.Button1Click(Sender: TObject);

begin

end;

Слово procedure извещает компилятор о начале процедуры. За ним следует имя метода TForm1.Button1Click. Это имя составное: оно состоит из имени класса TForm1 и собственно имени метода Button1Click.

Пока имеется только заготовка обработчика - тело метода пусто. Команды, которые будут написаны внутри обработчика, выполнятся при щелчке по компоненту.

Сделаем так, чтобы кнопка закрывала программу. Для этого в пустой строке между словами begin и end необходимо написать следующую команду:

Form1.Close;

Будет вызван метод Close (закрыть) для объекта Form1 (окно программы).

Примеры программ с обработчиком события OnClick можно посмотреть в:

[Приложении2](http://festival.1september.ru/articles/520215/pril2.exe) ("ЗАКРЫТЬ");

[Приложении3](http://festival.1september.ru/articles/520215/pril3.exe)("ЧИСЛО");

[Приложении4](http://festival.1september.ru/articles/520215/pril4.exe) ("АДРЕСА");

[Приложении5](http://festival.1september.ru/articles/520215/pril5.exe)("ТЕСТ");

[Приложении6](http://festival.1september.ru/articles/520215/pril6.exe)("ПРЫГАЮЩАЯ КНОПКА").

Фрагмент кода на языке Object Pascal приложения "Адреса"([Приложении 4](http://festival.1september.ru/articles/520215/pril4.exe)):

procedure TForm1.Button1Click(Sender: TObject);

begin

Label1.Caption:='ул. Базарова, дом 35, кв 35';

end;

procedure TForm1.Button2Click(Sender: TObject);

begin

Label2.Caption:='Проспект Программистов, корпус 3/1, кв. 275';

end;

procedure TForm1.Button3Click(Sender: TObject);

begin

Label3.Caption:='переулок Ковалевской, дом 27, кв 181';

end;

procedure TForm1.Button4Click(Sender: TObject);

begin

Label4.Caption:='ул. Беринга, дом 22, кв 209';

end;

end.

Фрагменты кода на языке Object Pascal приложения "Прыгающая кнопка" (Приложении 6):

procedure TForm1.Button1Click(Sender: TObject);

begin

Button1.Visible:=False;

Button2.Visible:=True;

end;

procedure TForm1.Button2Click(Sender: TObject);

begin

Button2.Visible:=False;

Button3.Visible:=True;

end;

...

procedure TForm1.Button7Click(Sender: TObject);

begin

Button7.Visible:=False;

Button8.Visible:=True;

end;

procedure TForm1.Button8Click(Sender: TObject);

begin

Button8.Visible:=False;

Button1.Visible:=True;

end; end.

**Другие события**

|  |  |
| --- | --- |
| **Название события** | **Когда возникает** |
| OnKeyPress | Наступает, когда пользователь нажимает какую-либо клавишу. |
| OnDBClick | Возникает при двойном щелчке на компоненте |
| OnMouseMove | Возникает при перемещении курсора мыши над компонентом |
| OnMouseDown | Возникает при нажатии на кнопку мыши |
| OnMouseUp | Возникает при отпускании кнопки мыши |
| OnEnter | Возникает при получении компонентом фокуса ввода |
| OnExit | Возникает при потере компонентом фокуса ввода |
| OnCreate | Возникает в момент создания формы. Существует только у формы |

* обработчик события - это процедура, которая выполняется при возникновении заданного события;
* для выбора события служит вкладка "События" (Events) "Инспектора объектов" (Object Inspector);
* научились создавать обработчик для самого популярного события OnClick.