

16 Dateiarbeit in Delphi

16.1 Einführung

In unserer bisherigen Arbeit mit Delphi haben wir bereits an verschiedenen Stellen mit Dateien gearbeitet (möglicherweise, ohne dass uns dies bewusst war). Es war dies die Arbeit mit den Methoden *LoadFromFile* und *SaveToFile* verschiedener Komponenten und die Ansteuerung des Druckers im Textmodus. Diese Möglichkeiten waren jedoch jeweils auf spezielle Dateitypen beschränkt und nicht allgemein gültig. Die im Folgenden beschriebenen Vorgehensweisen lassen prinzipiell die Arbeit mit jeder Datei zu.

16.2 Dateitypen

In der Art des Ansprechens einer Datei unterscheidet Delphi verschiedene Dateitypen. Zum Bearbeiten der verschiedenen Dateitypen sind stellenweise auch unterschiedliche Befehle notwendig.

Textdateien

Textdateien sind Dateien, die druckbare ASCII-Zeichen enthalten, welche in Zeilen angeordnet sind. So sind z.B. die Dateien mit der Endung *PAS* oder *DPR* reine Textdateien. Auch die Methode *SaveToFile* der *List-Box*-Komponente erzeugt eine Textdatei aus dem Inhalt der Listbox.

Der Dateityp für Textdateien ist *Text* bzw. *TextFile*. Der Typ *Text* stammt noch aus der Turbo Pascal-Zeit. Um Verwechslungen mit der Eigenschaft *Text* des Edit-Feldes zu vermeiden (und auch aus anderen Gründen, die hier nicht näher behandelt werden sollen), muss man beim Typ *Text* speziell dazuschreiben, dass er in der Unit *System* deklariert ist (siehe Quelltextbeispiel).

Die Variablen Deklaration einer Dateivariablen zur Bearbeitung von Textdateien könnte also folgendermaßen aussehen:

```
var Datei1 : System.Text;           { oder besser }
    Datei2 : TextFile;
```

Wir werden bei unserer Arbeit immer die Schreibweise *TextFile* benutzen.

Typisierte Dateien

Besteht eine Datei nur aus einer Aufzählung von Daten des gleichen Typs, so spricht man auch von einem typisierten File. Bei der Deklaration der Variablen bzw. des Filetyps muss man angeben (nach dem Schlüsselwort *of*), von welchem Typ die einzelnen Daten sind. Beispiele dafür wären:

```
type TWetter = record
    Tag : Byte;
    Temperatur, Niederschlag : Real;
end;

var Wetterdatei      : File of TWetter;
    Lottodatei     : File of Byte;
    Testdatei       : File of Char;
    Mitarbeiterdatei : File of String[30];
    Zahlendatei    : File of Word;
```

Die Einzeldaten einer typisierten Datei können von allen bisher behandelten Variablenotypen sein, nur nicht vom Dateityp selbst.

Untypisierte Dateien

Eine Datei, die beliebige Werte beinhalten kann und auf die man auch am universellsten zugreifen kann, wird durch den Variablenotyp *File* gekennzeichnet. Im Gegensatz zu typisierten Dateien spricht man hier von einer untypisierten Datei. Ein Quelltextbeispiel für die Deklaration einer untypisierten Datei wäre:

```
var Datei : File;
```

Wir werden in unseren Projekten jedoch nicht mit untypisierten Dateien arbeiten.

16.3 Allgemeine Befehle zur Dateiarbeit

Die allgemeinste Arbeit mit einer Datei läuft immer in folgenden 4 Schritten ab:

1. Festlegung, mit welcher Datei gearbeitet werden soll,
2. Öffnen oder Erstellen der Datei,
3. Lesen oder Schreiben von Daten,
4. Schließen der Datei.

Zuerst muss man der Dateivariablen eine Datei auf dem Datenträger (Festplatte, Diskette, ...) zuweisen, mit welcher gearbeitet werden soll. Das geschieht mit der Prozedur *AssignFile*. Dabei erwartet *AssignFile* als ersten Parameter eine bereits deklarierte Dateivariable beliebigen Typs (siehe vorhergehender Abschnitt) und als zweiten Parameter den Namen einer Datei auf dem Datenträger, die entweder schon vorhanden ist oder die erstellt werden soll. Ein Aufruf von *AssignFile* könnte also folgendermaßen aussehen:

```
AssignFile(Dateivariable, Dateiname);
```

➤ Hinweis 1: Alle folgenden Befehle, die sich auf die angegebene Dateivariable beziehen, werden mit der durch *AssignFile* verknüpften Datei auf dem Datenträger durchgeführt.

Als Nächstes muss man die Datei entweder öffnen (dann muss die Datei schon vorhanden sein) oder neu erstellen. Zum Öffnen dient der Befehl *Reset*, zum Neuerstellen und anschließenden Öffnen dient der Befehl *Rewrite*. Beide Befehle erwarten als Parameter eine Dateivariable (nicht den Dateinamen).

Quelltextbeispiele für diese beiden Befehle wären:

```
Reset(Dateivariable1); { oder }
Rewrite(Dateivariable2);
```

✿ Achtung: Wenn man den Befehl *Rewrite* auf eine Datei anwendet, die schon existiert, so wird diese Datei ohne Nachfrage überschrieben.

Als nächster Schritt folgt die eigentliche Operation mit der Datei, also z.B. das Lesen oder Schreiben von Daten. Dabei geschieht das Lesen mit den Befehlen *ReadLn* (bei Textdateien) oder *Read* (bei allen anderen Dateien), das Schreiben wird mit den Befehlen *WriteLn* (bei Textdateien) oder *Write* (bei allen anderen Dateien) durchgeführt.

Hierbei müssen sowohl beim Lesen als auch beim Schreiben jeweils zwei Parameter übergeben werden. Im ersten Parameter wird angegeben, auf welche Dateivariable der Befehl zugreifen soll, der zweite Parameter gibt an, in welche Variable der Wert aus der Datei gelesen werden soll oder welcher Wert in die Datei geschrieben werden soll.

Mögliche Quelltextbeispiele für diese Befehle wären:

<i>ReadLn(TextDatei, Zeile);</i>	{ Beispiel für Textdateien }
<i>WriteLn(TextDatei, Zeile);</i>	{ Beispiel für Textdateien }

<i>Read(Dateivariable, Variable);</i>	{ Beispiel für typisierte Datei }
<i>Write(Dateivariable, Variable);</i>	{ Beispiel für typisierte Datei }

Den Abschluss der Dateioperation bildet der Befehl *CloseFile*. Dieser Befehl beendet die Arbeit mit der Datei und schreibt alle eventuell noch nicht gespeicherten Änderungen an der Datei auf den Datenträger. Anschließend wird die Datei geschlossen. Dieser Befehl erwartet nur einen Parameter, und zwar den Namen der entsprechenden Dateivariablen. Der Befehl *CloseFile* im folgenden Beispiel

```
AssignFile(Dateivariable, 'C:\test.txt') ;
...
CloseFile(Dateivariable) ;
```

schließt die Datei 'C:\test.txt' auf der Festplatte.

Mit Hilfe der Funktion *EOF* (Abkürzung für *end of file*) kann man ermitteln, ob beim Lesen das Ende einer Datei bereits erreicht wurde. Die Funktion gibt in diesem Fall den Wert *True* zurück, anderenfalls den Wert *False*. Die Funktion erwartet als Parameter wieder den Namen der entsprechenden Dateivariablen. *EOF* wird häufig in Verbindung mit einer Schleife zum kompletten Einlesen einer Datei benutzt, wie folgender Quelltextausschnitt zeigen soll:

```
repeat
  Read(Dateivariable, Variable) ;
  { Verarbeitung der eingelesenen Variablen }
until EOF(Dateivariable) ;
```

16.4 Beispiele zur Dateiarbeit

Arbeit mit Textdateien

Die Arbeit mit Textdateien soll an einigen kleinen Beispielen demonstriert werden.

Das erste Beispiel (siehe nebenstehendes Bild) liest eine Textdatei (in unserem Beispiel die Datei 'unit1.pas') ein, die sich im aktuellen Verzeichnis befindet. Dabei werden die Zeilen gezählt; die Anzahl wird anschließend ausgegeben.

Außerdem werden alle Zeilen der Textdatei in Großbuchstaben umgewandelt und anschließend in eine Listbox geschrieben.

Der zugehörige Quelltextausschnitt lautet:

```
var Anzahl      : Integer;
    Zeile       : String;
    TextDatei   : TextFile;
begin
  ListBox1.Items.Clear;
  Anzahl:=0;
  AssignFile(TextDatei, 'unit1.pas');
  Reset(TextDatei);
  repeat
    ReadLn(TextDatei,Zeile);
    Inc(Anzahl);
    ListBox1.Items.Add(AnsiUpperCase(Zeile));
  until EOF(TextDatei);
  CloseFile(TextDatei);
  AnzahlEdit.Text:=IntToStr(Anzahl);
end;
```



Bildschirmansicht des Beispiels

- ⇒ Aufgabe 1: Analysieren Sie den Quelltext und geben Sie an, was in den einzelnen Zeilen passiert. Programmieren Sie anschließend das kleine Beispielprogramm und testen Sie es.
- ⇒ Aufgabe 2: Ändern Sie das Programm so um, dass nur noch Zeilen in die Listbox übernommen werden, die nicht leer sind (im oben gezeigten Bild sollten also die zweite und die sechste sichtbare Zeile nicht übernommen werden).

Im zweiten Beispiel soll eine Textdatei erzeugt werden, die den Inhalt einer Listbox in umgekehrter Reihenfolge enthält.

Dazu lesen wir beim Programmstart den Inhalt einer beliebigen Textdatei (in unserem Beispiel die Datei 'obst.txt') in eine Listbox ein.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ListBox1.Items.LoadFromFile('obst.txt');
end;
```

Anschließend schreiben wir den Inhalt der Listbox (von hinten beginnend) Zeile für Zeile in eine Textdatei.

Zur Kontrolle wird der Inhalt der von uns erstellten Textdatei in einer zweiten Listbox angezeigt (siehe nebenstehendes Bild).

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var i : Integer;
  TextDatei : TextFile;
begin
  ListBox2.Items.Clear;
  AssignFile(TextDatei,'tausch.txt');
  Rewrite(TextDatei);
  for i:=ListBox1.Items.Count-1 downto 0 do
    WriteLn(TextDatei,ListBox1.Items[i]);
  CloseFile(TextDatei);
  ListBox2.Items.LoadFromFile('tausch.txt');
end;
```

- ⇒ Aufgabe 3: Analysieren Sie den Quelltext und geben Sie an, was in den einzelnen Zeilen passiert. Welche Dateibefehle sind anders als im ersten Programm? Warum mussten hier an einigen Stellen andere Befehle verwendet werden?
- ⇒ Aufgabe 4: Erstellen Sie eine Textdatei mit ca. 10 Zeilen Inhalt. Programmieren Sie anschließend das zweite Beispielprogramm und testen Sie es.
Sie ersparen sich das Sortieren der Textdatei in der ersten Listbox, wenn sie deren Eigenschaft *Sorted* auf *True* setzen.

Arbeit mit typisierten Dateien

Das nächste Beispiel soll eine Wetterstation simulieren, in der täglich die Werte für Tageshöchsttemperatur und Niederschlag gemessen werden. Diese Werte sollen (jeweils für einen Monat) gespeichert werden. Außerdem soll die Möglichkeit bestehen, die Werte auch wieder zu laden und auszuwerten. Zur Lösung dieses Problems definieren wir als Erstes einen neuen Variablentyp, der für diese Aufgabe geeignet ist.

```
type TWetter = record
  Tag : Byte;
  Temperatur, Niederschlag : Real;
end;
```

Eine typisierte Datei, welche die Variablen vom angegebenen Typ speichern kann, deklariert man wie folgt:

```
var Datei : File of TWetter;
```

In diese Datei können nun beliebig viele Einzeldaten vom Typ *TWetter* geschrieben werden.

Zur Vereinfachung der Aufgabe wollen wir annehmen, dass die Daten komplett aus einem Stringgitter ausgelenzen und dann gespeichert werden sollen. Später soll man diese Daten bei Bedarf wieder von der Datei einladen und in das Stringgitter ausgeben können.



Bildschirmansicht des Beispiels

Damit man in ein Stringgitter auch manuell schreiben kann (wir hatten bisher immer nur Ausgaben per Quelltext erzeugt), muss in der Eigenschaft *Options* des Stringgitters die Untereigenschaft *goEditing* auf *True* gesetzt werden.

Das fertige Programm könnte dann etwa das nebenstehende Aussehen haben.

Die Programmierung der grafischen Ausgabe wurde in den Kapiteln über Grafik behandelt und soll in unserem Beispiel nur als Demonstration dienen, wie Sie dieses Programm weiterentwickeln können.

Zur Vereinfachung des Problems gehen wir von einem Monat mit 31 Tagen aus. Bei der späteren Programmentwicklung können wir das Programm um eine Ermittlung der Tageszahlen für die einzelnen Monate ergänzen.

Die einzelnen Schritte zum Schreiben in die Datei funktionieren ähnlich, wie das bereits bei den Textdateien beschrieben wurde. Der Hauptunterschied besteht in der Verwendung von *Write* statt *WriteLn*.

Außerdem ist natürlich das Belegen einer Variablen vom Typ *TWetter* mit Werten etwas schwieriger, da diese Variable aus drei Einzelvariablen besteht und da die Daten außerdem von Stringwerten (aus dem Stringgitter) in Zahlen umgewandelt werden müssen. Der entsprechende Quelltext lautet:

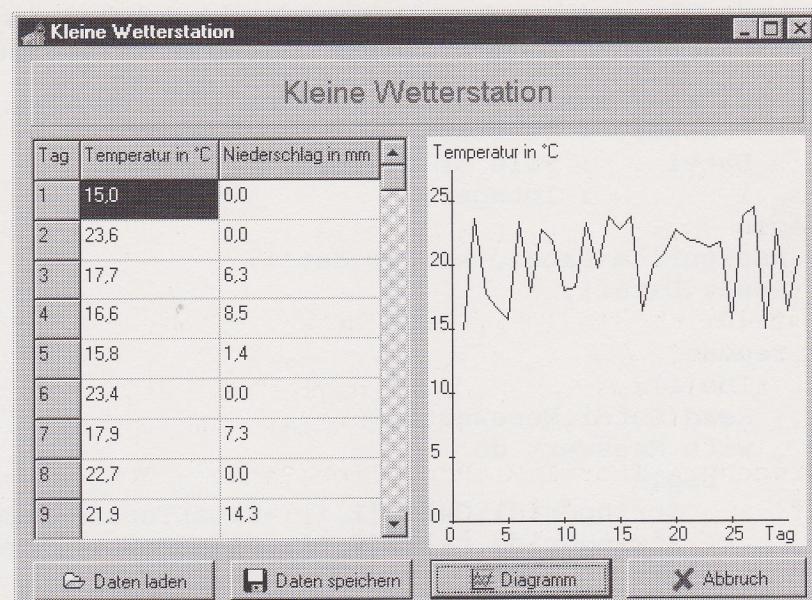
```
var Messwert : TWetter;
    Datei : File of TWetter;
    i : Integer;
begin
  AssignFile(Datei, 'wetter.dat');
  Rewrite(Datei);
  for i:=1 to 31 do
    with Messwert do
      begin
        Tag:=i;
        Temperatur:=StrToFloat(StringGrid1.Cells[1,i]);
        Niederschlag:=StrToFloat(StringGrid1.Cells[2,i]);
        Write(Datei, Messwert);
      end;
  CloseFile(Datei);
end;
```

➤ Hinweis 1: Um das Programm nicht unnötig kompliziert zu gestalten, wurde davon ausgegangen, dass alle 31 Tage komplett mit Werten bestückt sind.

➤ Hinweis 2: Die unterschiedliche Breite der Spalten wurde mit dem folgenden Quelltext erreicht. Konsultieren Sie bei Bedarf die Online-Hilfe zur Bedeutung der Eigenschaft *ColWidths*.

```
with StringGrid1 do
begin
  ColWidths[0]:=28;
  ColWidths[1]:=86;
  ColWidths[2]:=100;
end;
```

⇒ Aufgabe 5: Erzeugen Sie ein Formular mit dem oben gezeigten Aussehen (eventuell ohne die *Image*-Komponente). Geben Sie eigene Werte für Niederschlag und Temperatur ein und speichern Sie diese Werte wie angegeben in einer Datei.



Bildschirmsicht des erweiterten Beispiels

Auch das Einlesen der typisierten Datei gestaltet sich ähnlich wie das Einlesen der Textdatei. Der Hauptunterschied besteht darin, dass das Lesen einer Variablen aus der Datei mit *Read* statt mit *ReadLn* erfolgt.

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var Messwert : TWetter;
    Datei     : File of TWetter;
    i         : Integer;
begin
  AssignFile(Datei, 'wetter.dat');
  Reset(Datei);
  i:=0;
  repeat
    Inc(i);
    Read(Datei,Messwert);
    with Messwert do
      begin
        StringGrid1.Cells[1,i]:=FloatToStrF(Temperatur,ffFixed,10,1);
        StringGrid1.Cells[2,i]:=FloatToStrF(Niederschlag,ffFixed,10,1);
      end;
  until EOF(Datei);
  CloseFile(Datei);
end;
```

- Hinweis 3: Die Komponente *Tag* der Record-Variablen wurde in unserem Beispiel nicht benutzt. Sie könnte dazu dienen, das Programm, besonders aber die grafische Ausgabe auch dann funktionieren zu lassen, wenn nicht jeden Tag eine Temperatur eingetragen wird.
- ⇒ Aufgabe 6: Ergänzen Sie das Programm um die Möglichkeit, die gespeicherten Werte auch wieder laden zu können.
- ⇒ Aufgabe 7: Programmieren Sie eine grafische Ausgabe für die einzelnen Werte der Tabelle. Verwenden Sie für die Ausgabe von Temperatur und Niederschlag verschiedene Ausgabemöglichkeiten, z.B. Temperatur wie gezeigt, Niederschlag als Säulendiagramm.

Weitere Befehle für die Arbeit mit typisierten Dateien

Bei der Arbeit mit typisierten Dateien kann man noch eine Reihe weiterer Funktionen und Prozeduren verwenden, von denen noch 4 behandelt werden sollen.

Mit der Funktion *FileSize* kann man ermitteln, wie viele Elemente des in der Typdeklaration angegebenen Typs eine Datei enthält. Die Funktion *FileSize* kann allerdings erst benutzt werden, nachdem die entsprechende Datei mit *Reset* geöffnet wurde.

Öffnet man eine beliebige Datei als *File of Char* oder *File of Byte*, dann gibt *FileSize* ihre Größe in Byte an.

Den folgenden Quelltext öffnet die Datei aus unserem Wetterstations-Projekt einmal als *File of TWetter* und außerdem als *File of Byte*. Wie im nebenstehenden Bild zu erkennen ist, wird einmal die Anzahl der Eintragungen (31 Tage), zum anderen die Größe der Datei in Byte ermittelt (434 Byte).

```
procedure TForm1.AnzahlBtnClick(Sender: TObject);
var Datei     : File of TWetter;
begin
  AssignFile(Datei, 'wetter.dat');
  Reset(Datei);
  AnzahlEdit.Text:=IntToStr(FileSize(Datei));
  CloseFile(Datei);
end;
```



Beispielansicht zu *FileSize*

```

procedure TForm1.ByteBtnClick(Sender: TObject);
var Datei      : File of Char;
begin
  AssignFile(Datei, 'wetter.dat');
  Reset(Datei);
  ByteEdit.Text:=IntToStr(FileSize(Datei));
  CloseFile(Datei);
end;

```

⇒ Aufgabe 8: Erstellen Sie ein Testprogramm zur Funktion *FileSize*, das von einer vorhandenen typisierten Datei die Anzahl der gespeicherten Elemente und die Dateigröße in Byte ermittelt.

Die aktuelle Position innerhalb einer typisierten Datei kann mit der Funktion *FilePos* ermittelt werden. Dies ist besonders wichtig, wenn man wissen möchte, an welcher Position innerhalb der Datei die nächste Lese- oder Schreiboperation erfolgt. Als Parameter erwartet *FilePos* wieder die entsprechende Dateivariable.

Die Funktion *FilePos* kann natürlich auch erst benutzt werden, nachdem die entsprechende Datei mit *Reset* oder *Rewrite* geöffnet wurde.

In Anlehnung an Datenbanken, die häufig aus typisierten Dateien aufgebaut sind, spricht man beim Auslesen der aktuellen Position auch vom Datensatzzeiger, die einzelnen Elemente einer typisierten Datei bezeichnet man auch als Datensätze.

Die vorderste Position innerhalb einer typisierten Datei (vor dem ersten Datensatz) hat immer den Wert 0, die letzte Position (nach dem letzten Datensatz) hat demzufolge den Wert von *FileSize*.

⇒ Aufgabe 9: Überprüfen Sie den zweiten Teil der Aussage durch eine Skizze.

Die Befehle *Reset* und *Rewrite* positionieren den Datensatzzeiger immer an den Anfang einer Datei (also an die Position 0). Durch die Befehle *Read* oder *Write* wird der Datensatzzeiger immer um eine Position nach hinten geschoben.

Eine direkte Positionierung des Datensatzzeigers, also der aktuellen Position innerhalb der Datei, ist durch die Prozedur *Seek* möglich. Der erste Parameter ist wieder die Dateivariable, als zweiten Parameter erwartet *Seek* die Position, auf die der Datensatzzeiger positioniert werden soll. Dabei beginnt die Zählung wieder mit 0. Auch *Seek* kann erst benutzt werden, nachdem die Datei geöffnet wurde.

Die folgenden Quelltextzeilen sollen die Benutzung von *FilePos* und *Seek* demonstrieren.

p:=FilePos(Datei);	{ 1 }
Seek(Datei,0);	{ 2 }
Seek(Datei,FileSize(Datei));	{ 3 }
Seek(Datei,FilePos(Datei)-1);	{ 4 }

Die erste Zeile ermittelt die aktuelle Position des Datensatzzeigers und legt diesen Wert in einer *Integer*-Variablen ab. In der zweiten Zeile wird der Datensatzzeiger an den Anfang einer Datei (vor den ersten Datensatz) positioniert.

Die dritte Zeile ermittelt die Größe einer Datei und positioniert anschließend den Datensatzzeiger an das Ende der Datei (nach dem letzten Datensatz). Das kann notwendig sein, wenn man neue Daten an das Ende einer bestehenden Datei anhängen möchte.

Mit Hilfe der vierten Zeile kann der Datensatzzeiger um eine Position nach vorn geschoben werden. Dazu ermittelt man zuerst die aktuelle Position, verringert diesen Wert um 1 und positioniert anschließend den Datensatzzeiger an die so ermittelte Position. Diese Möglichkeit kann sinnvoll sein, wenn man sich in einer vorhandenen Datei rückwärts bewegen möchte.

Mit der Prozedur *Truncate* kann eine Datei ab einer bestimmten Position "abgeschnitten" werden. Das heißt, alle Datensätze der aktuellen Datei ab der aktuellen Position werden gelöscht.

Die Prozedur *Truncate* erwartet als einzigen Parameter die entsprechende Dateivariable der zu bearbeitenden Datei.

Vor der Anwendung von *Truncate* muss man die Datei wieder öffnen und den Datensatzzeiger an die gewünschte Stelle positionieren. Die folgenden Quelltextzeilen sollen die Anwendung der Prozedur *Truncate* demonstrieren.

```
Laenge:=FileSize(Datei);           { 1 }
Seek(Datei,Laenge-1);             { 2 }
Truncate(Datei);                 { 3 }
```

In der ersten Zeile wird die Größe der Datei (die Anzahl der Elemente) ermittelt. Dieser Wert wird um 1 verringert und der Datensatzzeiger wird mit dem so ermittelten Wert innerhalb der Datei positioniert (Zeile 2). Damit steht der Datensatzzeiger genau vor dem letzten Datensatz. Die Anwendung von *Truncate* in der dritten Zeile führt dazu, dass die Datei an dieser Position abgeschnitten wird. Im konkreten Beispiel wird ihre Größe also um genau ein Element verringert.

⇒ Aufgabe 10: Schreiben Sie ein kleines Programm, mit dem Sie die Größe einer typisierten Datei (sinnvollerweise die Datei aus Aufgabe 8) um einen Datensatz verringern. Überprüfen Sie dann das Ergebnis der Operation mit dem Programm aus Aufgabe 8.

16.5 Kontrollfragen

- ☛ Frage 1: Eine Textdatei mit Namen 'Aufgabe1.txt' soll erstellt werden. In diese Textdatei sollen 2 Zeilen mit den Inhalten 'Zeile 1' und 'Zeile 2' geschrieben werden. Geben Sie den entsprechenden Quelltext zur Lösung dieses Problems an.
- ☛ Frage 2: Die Textdatei aus Frage 1 soll geöffnet und anschließend um den Eintrag 'Zeile 3' erweitert werden. Wie lautet der entsprechende Quelltext?
- ☛ Frage 3: Wann wendet man den Befehl *Write* an, wann den Befehl *WriteLn*?
- ☛ Frage 4: Wie kann man den Inhalt einer Textdatei komplett in einer Listbox ausgeben? Geben Sie zwei verschiedene Möglichkeiten an.
- ☛ Frage 5: Eine vorhandene Textdatei soll zeilenweise durchsucht werden. Alle Zeilen, die einen deutschen Umlaut oder ein 'ß' enthalten, sollen in einer Listbox angezeigt werden. Geben Sie einen Quelltext an, der diese Aufgabe erfüllt.
- ☛ Frage 6: Wie muss man den Quelltext aus Frage 5 verändern, damit die entsprechenden Zeilen nicht in eine Listbox, sondern in eine andere Datei gespeichert werden?
- ☛ Frage 7: Welche 2 Sachverhalte muss man bei der Anwendung von *Rewrite* beachten?
- ☛ Frage 8: In einem Stringgitter sind in je einer Spalte der Name, der Vorname und die Telefonnummer von verschiedenen Personen gespeichert. Geben Sie eine Datenstruktur an, die geeignet ist, einen Datensatz mit den angegebenen Eigenschaften aufzunehmen. Achten Sie darauf, keinen Speicherplatz zu verschwenden!
- ☛ Frage 9: Wie lautet der Quelltext zum Speichern der Daten aus dem Stringgitter (Frage 8) in eine typisierte Datei? Von welchem Typ muss diese Datei sein?
- ☛ Frage 10: Die Datei aus Frage 9 soll in umgekehrter Reihenfolge wieder in das Stringgitter zurückgeschrieben werden. Wie muss man dabei vorgehen?
- ☛ Frage 11: Wie kann man die Größe einer typisierten Datei (in Byte) ermitteln?
- ☛ Frage 12: Wie kann man die Anzahl der Datensätze in einer typisierten Datei ermitteln?
- ☛ Frage 13: In einer typisierten Datei sollen der erste und der letzte Datensatz miteinander vertauscht werden. Wie muss man dabei vorgehen?
- ☛ Frage 14: Mit welchem Quelltext kann man die letzten 2 Datensätze einer typisierten Datei löschen?
- ☛ Frage 15: Wie kann man den ersten Datensatz einer typisierten Datei löschen? Beachten Sie, dass dabei auch die Größe der Datei verringert werden muss.
- ☛ Frage 16: Wie kann der Datensatzzeiger an das Ende einer typisierten Datei positioniert werden? Nennen Sie zwei verschiedene Möglichkeiten.
- ☛ Frage 17: Zwei typisierte Dateien vom gleichen Typ sollen miteinander zu einer dritten Datei verbunden werden. Wie muss man dabei vorgehen?