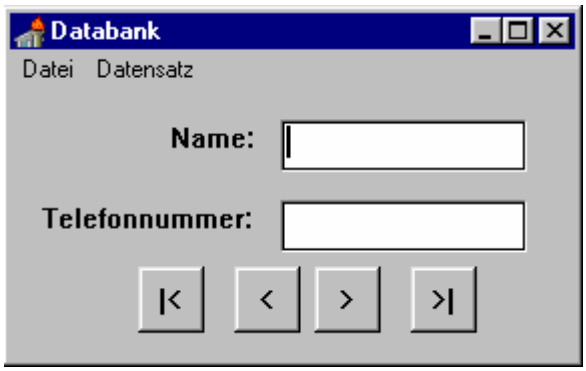
Projekt „Minidatenbank“

Sicherlich kennst du die kleinen elektronischen Notizbücher, in die man Namen und Telefonnummern von Bekannten eintragen kann. Diese „Minidatenbanken“ haben ein Display, welches dir jeweils einen Namen und dessen zugehörige Telefonnummer anzeigen kann – mehr nicht. Dennoch ist es möglich, mehrere Namen zu speichern, welche du mit Hilfe von Tasten durchblättern kannst. Im folgenden wollen wir eine solche Minidatenbank entwickeln, welches ungefähr so aussehen könnte:



Aufgabe 1: Entwerfe das Layout des Projekts. Welche Menüeinträge sollten unter dem Menüpunkt *Datei*, welche unter dem Menüpunkt *Datensatz* aufgeführt werden?

Einführung in die Projektidee

Bei der Entwicklung dieses Programms werden wir uns mit folgenden Fragen beschäftigen müssen:

- Welche Bedienfunktionen sollte unser Programm haben?
- Wie ist eine interne Speicherung der Daten möglich?
- Wie können die Daten auf längere Zeit gesichert werden (Datei speichern und öffnen)?
- Welcher Zusammenhang besteht zwischen den Daten im Hintergrund und dem Formular?

Fangen wir vielleicht am einfachsten mit der ersten Frage an. Welche Bedienfunktionen sind sinnvoll? Das Menü Datei steht im Normalfall für sämtliche Dateioperationen, also das Speichern und Laden einer ganzen Datei sowie der Einrichtung einer neuen und schließen einer vorhandenen Datei. Das Menü Datensatz ist für die einzelnen Einträge in der Datenbank zuständig. So sollte man neue Datensätze anlegen und bestehende Datensätze löschen können. Eventuell sollte man auch einen bereits vorhandenen Datensatz ändern oder nach einem vorhandenen Datensatz suchen können. Andere Funktionen wären noch denkbar, allerdings wollen wir uns auf diese vier beschränken. Damit wäre die erste Frage geklärt.

Die zweite Frage ist aus Informatikersicht zunächst interessanter. Du hast beim Lottoprojekt bereits eine Möglichkeit erfahren, mehrere Daten in einer Variablen zusammenzufassen – das ARRAY in DELPHI. Auch hier haben wir es mit mehreren Daten ein und des selben Typs zu tun, welche in eine Variable zusammengefasst werden sollen. Eine mögliche Variablendeklaration wäre also:

```
Name:    array[1..100] of string;
Telefon: array[1..100] of string;
```

Der Typ `string` bei der Variablen `Telefon` ist deshalb sinnvoll, da es sich bei einer Telefonnummer nicht um eine Zahl im mathematischen Sinne handelt (führende Nullen für Vorwahlen, evtl. Trennungszeichen zwischen Vor- und Durchwahl).
Schön ist diese Art der Variablendeklaration allerdings nicht. Eigentlich gehören doch auch Name und Telefon zusammen und sollten als eine Einheit aufgefasst werden. Wir könnten nun natürlich das ARRAY erweitern [1..200] und im ersten Teil nur die Namen und im zweiten Teil nur die Telefonnummern speichern. DELPHI bietet allerdings eine elegantere Möglichkeit – das sogenannte RECORD. Mit diesem Befehl lassen sich verschiedene Daten zu einer Variablen zusammenfassen:

```
var Eintrag: record
               Name, Telefon: string;
             end;
```

Will man einen solchen Eintrag z. B. auf dem Bildschirm ausgeben oder mit neuen Daten belegen, so funktioniert das ganz ähnlich wie bei Edit-Feldern:

```
Eintrag.Name:= ´Anton´;
E_Telefon.Text:= Eintrag.Telefon;
```

[Du kennst die Schreibweise mit dem Punkt (`.`) bereits von den Oberflächenobjekten. Ein Oberflächenobjekt (z. B. das Edit-Feld `E_Eingabe`) hat ebenfalls unterschiedliche Daten (z. B. `Text` vom Typ `string` oder `Width` vom Typ `integer`, etc.), auf welche mit dem Punkt als Trennung zugegriffen wird (z. B. `E_Eingabe.Width:= 100;`).]

Allerdings wollten wir ja nicht nur einen Eintrag speichern, sondern mehrere. Das würde zu einer solchen (unübersichtlichen) Datenstruktur führen:

```
Eintraege: array[1..100] of record
                              Name, Telefon: string;
                            end;
```

Beim Lotto hast du bereits gesehen, dass TYPE-Deklarationen sinnvoll sind, um die Lesbarkeit des Programms zu erhöhen. Außerdem erspart man sich eine Menge Tipparbeit, wenn an anderer Stelle eine neue Variable gleichen Typs deklariert werden muß. Die folgende Typdeklaration hilft uns hier weiter:

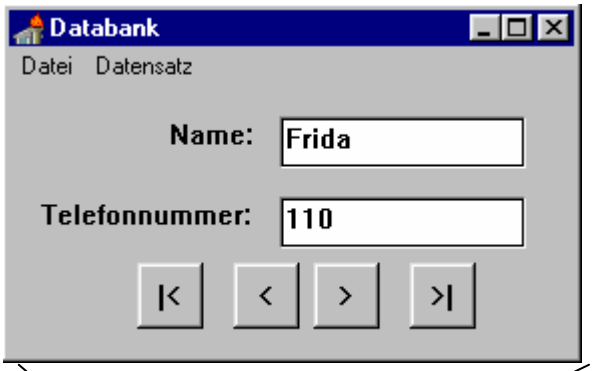
```
const maxAnzahl = 100;
type TEintrag=   record
                   Name, Telefon: string;
                 end;
     TEintraege= array[1..maxAnzahl] of TEintrag;
...
var Eintraege: TEintraege;
```

Aber was soll diese Konstantendeklaration in der ersten Zeile? Stelle dir vor, du stellst irgendwann fest, dass du mit 100 Speicherplätzen für deine Bekannten nicht mehr auskommst. Dann müsstest du überall die Zahl 100 durch eine größere Zahl ersetzen. Hast du vorher eine Konstante definiert, so brauchst du nur dort einmal die Zahl zu erhöhen, und dein ganzes Programm läuft mit mehr als 100 Speicherplätzen. Die Größe der Zahl ist allerdings begrenzt.

[Ein String benötigt 256 Bytes Speicherplatz. Die Datenstruktur besteht aus 200 Strings, also aus 200·256 = 51200 Bytes. Eine Datenstruktur in DELPHI darf maximal 64 Kilobytes bzw. 65536 Bytes groß sein – damit sind maximal 128 Einträge möglich. Will man mehr

Datensätze speichern, so muss man die Länge der Strings begrenzen, z. B. `name: string[10]`. Dieser String benötigt nur 11 (!!!ein Byte für die Länge!!!) Bytes, so dass mehr Datensätze in einer 64KB-Struktur Platz finden.]

Gut, Jetzt haben wir die Datenstruktur für die Speicherung klar:



`Eintraege:`

Anton 51189	Berta 4821	Carla 88112	Dieter 9448	Emil 3629	Frida 110	Moni 228	Paul 112	Uwe 59992			

Wie du erkennen kannst, wird auf dem Formular immer nur ein aktueller Datensatz angezeigt. die anderen Datensätze verbleiben im Hintergrund. Woher weiß man allerdings, welches der aktuelle Datensatz ist? Irgendwie muss man sich doch seine **`Position`** merken, damit man im ARRAY darauf zugreifen kann. Außerdem wäre es auch nicht schlecht zu wissen, welche **`Anzahl`** von Datensätzen insgesamt schon eingetragen wurden (die Konstante `maxAnzahl` gibt ja nur an, wie viele Datensätze höchstens gespeichert werden können). Weitere sinnvolle Informationen wären z. B. auch noch:

`DateiGeaendert`: Wurde an den Namen etwas geändert, so dass eine erneute Speicherung der Datei notwendig wird? Mit dieser Information kann der Benutzer beim Verlassen des Programms noch einmal gefragt werden, ob er die Datei speichern möchte.

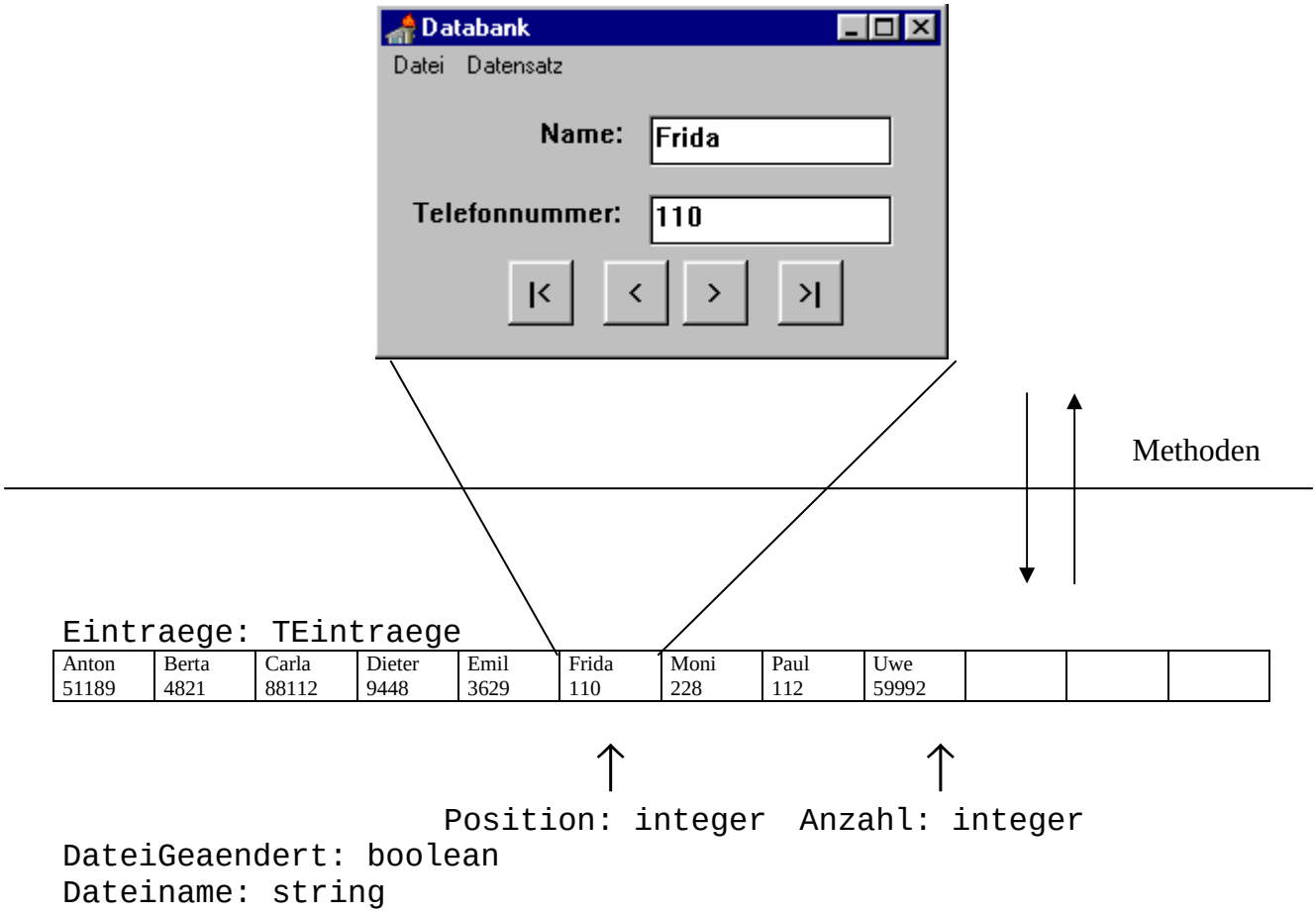
`Dateiname`: Es ist üblich, den Dateinamen mitzuverwalten.

Insgesamt hätten wir also neben der Variablen `Eintraege` noch vier weitere Variablen (`Position`, `Anzahl`, `DateiGeaendert` und `Dateiname`). Diese fünf Variablen gehören irgendwie zusammen, genauso wie die Daten einer Person (Name und Telefonnummer) zusammengehörten. Bevor wir nun aber auch hier einen RECORD definieren, sollten wir uns noch mehr Gedanken über die Programmierung des Projekts machen.

Aufgabe 2: Betrachte nochmals die fünf Variablen, die zur Datenbank gehören. Welchem Datentyp gehören diese an?

Die Idee der Modularisierung – Einführung in das Class-Konzept

Das Rad muss nicht immer wieder neu erfunden werden – dieses Sprichwort hat auch in der Programmierung seine Bedeutung. Einmal geschriebene Teilprogramme sollten so aufgebaut sein, dass sie in anderen Programmen wiederverwendet werden können. Eine solche Datenbank, die im Hintergrund des Formulars existiert, könnte auch für andere Zwecke (z. B. Drucken von Serienbriefen, Direktanwahl über Namensauswahl, etc.) sinnvoll sein. Sie sollte also wiederverwendbar sein. Eine Operation wie das Suchen eines Namens in der Datenbank soll ja nicht nur vom unserem Projekt aus funktionieren, sondern auch dann, wenn man unsere Datenbank in anderen Projekten nutzen will. Die „Operation“ oder „Methode“ Suchen gehört also logisch zur Datenbank und nicht zum Formular. Genauso ist es mit der „Methode“ Datei speichern. Der Benutzer klickt zwar im Menü auf Datei|Speichern..., allerdings ist es Sache der Datenbank, sich selbst zu speichern. Man müsste also der Datenbank den Befehl geben können: „Speichere dich unter dem Dateinamen xxx“. Ein anderer Befehl, den man der Datenbank geben können sollte, wäre z. B. auch: „Füge den Namen xxx mit Telefonnummer yyy in dich ein“, oder: „Wie lautet dein aktueller Datensatz, auf dem du gerade stehst?“. Unsere Datenbank hat also nicht nur Variablen (die fünf Stück von vorhin) sondern auch „Methoden“, welche die Kommunikation zwischen Formular und Datenbank herstellen. Bildlich kann man sich das wie folgt darstellen:



Aufgabe 3: Welche Operationen werden für die Kommunikation benötigt? Überlege dir bei jedem „Befehl“ an die Datenbank, welche Informationen du der Datenbank mitliefern musst (CbV-Variablen) bzw. welche Informationen du von der Datenbank bei einem „Befehl“ zurückerhalten möchtest (CbR-Variablen).

Folgende Methoden wären nötig (sie entsprechen im wesentlichen den in Kapitel 1 erwähnten Menüpunkten):

```
procedure DateiNeu;
procedure DateiSchliessen;
procedure DateiOeffnen(Dname: string);
procedure DateiSpeichern(Dname: string);
procedure DatensatzAnlegen(Date: TEintrag);
procedure DatensatzLoeschen;
procedure DatensatzSuchen(Name: string);
procedure DatensatzAendern(Date: TEintrag);
```

Dies waren alles Methoden, welche die Kommunikation vom Formular zur Datenbank herstellen. Allerdings reicht das nicht aus, denn das Formular soll ja von der Datenbank auch Informationen zurückerhalten. Wir brauchen also auch eine Methode, die den aktuellen Datensatz an das Formular zurückgibt:

```
procedure DatensatzLesen(var Date: TEintrag);
```

An der CbR-Variablen kannst du bereits erkennen, dass diese Methode dem Formular einen Wert zurückgeben soll.
Schließlich fehlen uns nur noch die Methoden, um uns in der Datenbank hin- und herzubewegen. Die Buttons vom Formular müssen also den Befehl geben können: „Gehe zum nächsten Datensatz“ oder „Springe an den Anfang der Datenbank“.

```
procedure NaechsterDatensatz;
procedure VorherigerDatensatz;
procedure ErsterDatensatz;
procedure LetzterDatensatz;
```

Damit haben wir alles das, was wir für unsere Datenbank benötigen. Eine solche Zusammenfassung von Daten (unsere fünf Variablen) und Operationen (unsere 13 Methoden) nennt man **Klasse**. Du kennst bereits eine Klasse aus DELPHI: Ein Formular, welches du erstellst, ist ein Klasse mit Daten (Buttons, Edit-Felder, etc.) und Methoden (z. B. Click-Routinen oder selbst geschriebene Prozeduren, welche im `private`-Teil deklariert wurden.). Deshalb wird unsere Klasse auch (fast) genauso aufgebaut, wie die Klasse des Formulars.

[Falls du dich nicht mehr daran erinnerst, so schaue dir im Skript zur 11.1 noch einmal den Aufbau eines DELPHI-Projekts an.]

Aufgabe 4: Versuche zuerst alleine sämtliche `const`-Definitionen und `type`-Definition zu schreiben. Orientiere dich dabei an die Typdefinition deines Formulars.

```
unit U_Datenb;

interface

const ...
type  ...
```

Falls du es nicht geschafft hast, so siehst du hier die Definitionen für die Datenbank.

```
unit U_Datenb;

interface

const maxAnzahl = 100;
type  TEintrag  = record
                    Name, Telefon: string;
                  end;
      TEintraege= array[1..maxAnzahl] of TEintrag;
      TDatenbank= class
                                    { DATEN der Klasse TDatenbank }
        Eintraege: TEintraege;
        Position: integer;
        Anzahl: integer;
        DateiGeaendert: boolean;
        Dateiname: string;
                                    { METHODEN der Klasse TDatenbank }
        procedure DateiNeu;
        procedure DateiSchliessen;
        procedure DateiOeffnen(Dname: string);
        procedure DateiSpeichern(Dname: string);

        procedure DatensatzAnlegen(Date: TEintrag);
        procedure DatensatzLoeschen;
        procedure DatensatzSuchen(Name: string);
        procedure DatensatzAendern(Date: TEintrag);
        procedure DatensatzLesen(var Date: TEintrag);

        procedure NaechsterDatensatz;
        procedure VorherigerDatensatz;
        procedure ErsterDatensatz;
        procedure LetzterDatensatz;
      end;

implementation

procedure TDatenbank.DateiNeu;
begin

end;

end.
```

Dies ist das Grundgerüst unserer Datenbank – das sogenannte Interface oder die Schnittstelle. Hier wird nun ganz genau beschrieben, welche Daten und Operationen anderen Programmen zur Verfügung stehen, die diese Datenbank benutzen wollen.

Aufgabe 5: Erstelle mit DELPHI eine neue Unit (*Datei*|*Neue Unit*) und gebe die obige Deklaration ein. Erstelle im `implementation`-Teil für jede Methode eine „Leer-Prozedur“ wie die oben angedeutete

`procedure TDatenbank.DateiNeu`,

damit es beim Compilieren deines Projekts keine Fehler gibt. Die jeweils nötigen Befehle für die Prozeduren werden wir nach und nach entwickeln. Speichere anschließend die Unit unter einem sinnvollen Namen (z. B. U_DATENB.PAS).

Aufgabe 6: Unser Formular möchte diese Datenbank benutzen können. Damit dies funktioniert muss die Unit des Formulars erweitert werden. Übernehme die unterstrichenen Änderungen in deine Formular-Unit:

```
uses SysUtils, WinTypes, ..., U_Datenb;
       ...
var Form1: TForm1;
    DB: TDatenbank;

procedure TForm1.FormCreate(Sender: TObject);
begin
  DB:= TDatenbank.Create;
end;
```

[Zur Erläuterung: Eine Variable, deren Typ eine Klasse ist, muss in DELPHI zu Anfang neu erzeugt werden. Dies geschieht mit dem Befehl <Variable>:= <Typname der Klasse>.Create, in unserem Beispiel also durch `DB:= TDatenbank.Create;`. Sinnvollerweise setzt man diesen Befehl in die FormCreate-Ereignisroutine, da die Variable global ist und immer benötigt wird. Würde sie nur lokal benutzt, so sollte dieser Befehl auch nur auf lokaler Ebene implementiert werden. Ach ja, die FormCreate-Ereignisroutine kennst du hoffentlich noch vom Lotto-Projekt. Dort hast du den Zufallsgenerator neu gestartet.]

Implementierung der Methoden

Jetzt geht es ans Eingemachte, der Programmierung der Methoden, schließlich wollen wir ja nicht nur eine leere Hülle der Datenbank haben.

Methode zum Anlegen einer neuen Datenbank – die Initialisierung

Als erstes wollen wir erreichen, dass eine neue Datenbank angelegt wird. Dies ist die Grundvoraussetzung dafür, dass man dein Programm überhaupt nutzen kann. Die Ereignisroutine zum Menüpunkt *Datei|Neu* des Formulars muss lediglich die entsprechende Methode der Datenbank aufrufen:

```
procedure TForm1.MI_NeuClick(Sender: TObject);
begin
  DB.DateiNeu;            { Methode der Datenbank }
  E_Name.Text:= ´´;
  E_Telefon.Text:= ´´;
end;
```

Die Methode der Datenbank dagegen muss nur dafür sorgen, dass alle Variablen sinnvoll vorbelegt werden. Dies heißt im einzelnen:

- Die Variable `Anzahl` muss auf 0 gesetzt werden (neue Datenbank ⇒ keine Datensätze).
- Die Variablen `Position` muss auf 0 gesetzt werden (dies ist sinnvoll, da man so anhand der Variablen Position direkt sehen kann, ob die Datenbank leer ist).
- Die Variable `DateiGeaendert` muss auf `false` gesetzt werden.
- Die Variable `Dateiname` sollte auf ´NONAME.DAT´ oder ähnliches gesetzt werden.
- Die Variable `Eintraege` sollte mit leeren Datensätzen (heißt Name und Telefon sind leere Zeichenketten) belegt werden.

Aufgabe 7: Implementiere die Methode DateiNeu der Datenbank.

```
procedure TDatenbank.DateiNeu;
var ...
begin
  Anzahl:= 0;
  ...
end;
```

Die Belegung der Variablen `Eintraege` kannst du am sinnvollsten mit einer FOR-Schleife realisieren (`for i:= 1 to maxAnzahl do ...`), in der du in jedem einzelnen Eintrag `Eintraege[i].Name:= ´´` und `Eintraege[i].Telefon:= ´´` setzt.

Methoden zur Erzeugung der Datensätze – das Einfügen und Löschen

Die erste Methode war noch nicht weiter schwer – kommen wir deswegen zu den Datensätzen. Die Datenbank soll über Methoden verfügen, Datensätze neu anzulegen, zu löschen, zu suchen oder zu ändern. Überlegen wir uns als erstes eine algorithmische Idee für das Anlegen von Datensätzen. Unsere Datenbank soll so angelegt sein, dass neue Namen gleich sortiert eingefügt werden.

Aufgabe 8: Entwickle einen Algorithmus, welcher in ein ARRAY von Namen einen neuen Namen sortiert einfügt. Orientiere dich an dem folgenden Datenbankbeispiel, in dem der Datensatz mit dem Namen *Carla* eingefügt werden soll. Beachte auch eventuelle Spezialfälle (es gibt noch gar keinen Namen bzw. es ist kein Platz mehr für einen neuen Namen).

`Eintraege: TEintraege`

1	2	3	4	5	6	7	8	9	10	11	...
Anton 51189	Berta 4821	Bruno 88112	Dieter 9448	Emil 3629	Frida 110	Moni 228	Paul 112	Uwe 59992			...

Ich hoffe, du hattest eine Idee, wie das Einfügen stattfinden könnte. Grundsätzlich gibt es natürliche mehrere Möglichkeiten, welche unterschiedlich kompliziert sind. Eine Möglichkeit ist die folgende, nur grob beschriebene:

- Wenn noch kein Name existiert, dann setzte den Namen *Carla* an die erste Position
- Sind schon Namen vorhanden, dann prüfe nach, ob überhaupt noch Platz ist.
- Ist noch Platz vorhanden, dann tue folgendes:
  - Durchlaufe das ganze Feld von vorne, bis man einen Namen gefunden hat, der größer als der Name *Carla* ist. (In unserem Beispiel würde man den Namen *Dieter* an Position 4 feststellen.)
  - Verschiebe alle Namen ab der gefundenen Position um eine Stelle nach rechts, d. h. *Uwe* muss an Position 10, *Paul* an Position 9, ... *Dieter* an Position 5.
  - Füge zum Abschluss den Namen *Carla* an der nun freien Position ein.

Diese Möglichkeit ist in Ordnung, aber es gibt eine geschicktere:
Das Feld wird nicht von vorne nach hinten, sondern von hinten nach vorne durchlaufen, bis man einen Namen gefunden hat, der kleiner ist als der Name *Carla*.

Aufgabe 9: Versuch zu dieser Strategie ebenfalls einen Algorithmus zu entwerfen. Welche Input-, Output- und Hilfsvariablen benötigt dein Algorithmus?

Ich hoffe, du bist auf eine der folgenden Lösung ähnliche Idee gekommen:

ALGORITHMUS ***DatensatzAnlegen***

Input: Datensatz: TEintrag
Hilfsobjekte: Index: integer
Global: Konstante: maxAnzahl
Variablen der Datenbank: Eintraege, Anzahl, Position, DateiGeaendert

- Falls Anzahl < maxAnzahl
  dann
  - Falls Anzahl = 0
    dann
    - Position ← 1

    sonst
    - Position ← Anzahl
    - Solange (Position >= 1) UND (Eintraege[Position] >= Datensatz)
      tue
      - Eintraege[Position + 1] ← Eintraege[Position]
      - Position ← Position – 1
    - Position ← Position + 1
  - Eintraege[Position] ← Datensatz
  - Anzahl ← Anzahl + 1
  - DateiGeaendert ← true

Aufgabe 10: Implementiere die entsprechende Methode in der Datenbank-Unit.

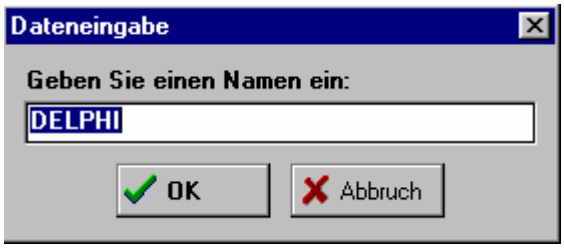
```
procedure TDatenbank.DatensatzAnlegen(Date: TEintrag);
begin
  ...
end;
```

Damit du deine eben erstellte Methode allerdings testen kannst, fehlt mal wieder die entsprechende Ereignisroutine des Formulars
(`procedure TForm1.MI_DatensatzNeuClick(Sender: TObject);`)

Wenn der Anwender auf *Datensatz|Anlegen* klickt, so sollte er die Möglichkeit bekommen, Name und Telefon der neuen Person einzugeben. DELPHI bietet dir für die Benutzereingabe die Funktion `InputBox(<Titel>,<Meldung>,<vordefinierter Text>)`.
Ruft man z. B. den folgenden Befehl auf: (x ist eine Variable vom Typ `string`)

```
x := InputBox(´Dateneingabe´,´Geben Sie einen Namen ein:´,´DELPHI´);
```

so erhält man das folgende Resultat:



Der Wert, den der Benutzer in das Eingabefeld eingibt, wird anschließend in der Variablen x gespeichert, d. h. x bekommt den Wert ´DELPHI´.

Mit Hilfe dieser DELPHI-Funktion kannst du nun nacheinander die Werte für den einzufügenden Datensatz einlesen lassen:

```
procedure TForm1.MI_DatensatzNeuClick(Sender: TObject);
var Date: TEintrag;
begin
  Date.Name:=    InputBox(´Name´,´Geben Sie den Namen ein:´,´´);
  Date.Telefon:= InputBox(´Telefon´,´und die Telefonnr:´,´´);
  DB.DatensatzAnlegen(Date);
  E_Name.Text:= Date.Name;
  E_Telefon.Text:= Date.Telefon;
end;
```

Aufgabe 11: Implementiere die Formular-Routine des Menüpunktes Datensatz|Anlegen. Dein Programm ist das erste mal wirklich testbereit, da jetzt Datensätze eingefügt werden können. Also, teste es aus.

Wenn man Datensätze neu anlegen kann, so sollte man auch Datensätze löschen können.

Aufgabe 12: Die Methode zum löschen des Datensatzes an der aktuellen Position ist etwas einfacher als das sortierte Einfügen eines neuen Datensatzes. Mache dir zuvor die algorithmische Idee klar und implementiere anschließend die entsprechende Methode in der Datenbank-Unit:

```
procedure TDatenbank.DatensatzLoeschen;
begin
  ...
end;
```

Implementiere anschließend auch die Ereignisroutine für den Menüpunkt im Formular.

Nach dem Löschen eines Datensatzes ist irgendwie nicht so ganz klar, welcher Datensatz denn danach angezeigt werden soll. Ist es nun sinnvoll, (a) den Datensatz nach dem gelöschten oder (b) den Datensatz vor dem gelöschten Datensatz anzeigen zu lassen? Prinzipiell stehen dir beide Möglichkeiten offen, allerdings solltest du die in beiden Fällen den jeweiligen Spezialfall beachten: Variante (a) hat den Sonderfall, dass der letzte Datensatz gelöscht wird, Variante (b) hat den Sonderfall, dass der erste Datensatz gelöscht wird. Wenn du keine Lösung gefunden hast, kannst du im Anschluss dieses Kapitels die Prozedur zum Löschen eines Datensatzes nachlesen. Der Algorithmus benutzt die Variante (a).

Methoden zur Navigation

Beim Test hast du zwar gesehen, dass neue Datensätze auf dem Bildschirm erschienen, allerdings konnte man sich einmal eingegebene Datensätze nicht mehr anschauen. Es wäre also nötig, sich durch die Datensätze „durchzuklicken“. Schauen wir uns deshalb die vier Methoden für die Navigation an, deren erste ich dir hier vorstelle:

```
procedure TDatenbank.NaechsterDatensatz;
begin
  if (Position < Anzahl)
    then inc(Position);
end;
```

[Die Abfrage `Position < Anzahl` ist deshalb nötig, da die Position nicht auf einen noch unbelegten Datensatz zeigen sollte.]

Aufgabe 13: Implementiere genauso die drei anderen Datenbank-Methoden für die Navigation. Denke daran, dass den Prozedur-Bezeichnern immer das `TDatenbank.` vorangestellt werden muss.

Diese Routinen waren ausschließlich für die Datenbank gedacht. Allerdings sollte auf dem Formular auch jedes Mal der aktuelle Datensatz angezeigt werden. Wir müssen also noch die Ereignisroutinen für die Navigations-Buttons schreiben. Der Button, mit dem man auf dem Formular den nächsten Datensatz anzeigen lässt, erhält folgende Ereignisroutine:

```
procedure TForm1.B_vorClick(Sender: TObject);
var Date: TEintrag;
begin
  Datenbank.ErhoehePosition;      { Datenbankposition erhöhen, }
  Datenbank.DatensatzLesen(Date); { aktuellen Datensatz holen }
  E_Name.Text:= Date.Name;        { und in die Edit-Felder ausgeben. }
  E_Telefon.Text:= Date.Telefon;
end;
```

Aufgabe 14: Implementiere für die drei anderen Navigationsbuttons die jeweiligen Ereignisroutinen in der Formular-Unit.

Bevor du jetzt allerdings einen Erfolg siehst, musst du natürlich noch die Methode `DatensatzLesen` der Datenbank-Unit implementieren. Diese sieht wie folgt aus:

```
procedure TDatenbank.DatensatzLesen(var date: TEintrag);
begin
  if (Position = 0)
    then begin
           date.Name:= ´´;
           date.Telefon:= ´´;
         end
    else date:= Eintraege[Position];
end;
```

Du hast wahrscheinlich gesehen, dass in allen Ereignisroutinen zur Navigation durch die Datensätze immer die gleiche Befehlesfolge auftritt:

```
Datenbank.DatensatzLesen(Date); { aktuellen Datensatz holen }
E_Name.Text:= Date.Name;        { und in die Edit-Felder ausgeben. }
E_Telefon.Text:= Date.Telefon;
```

Es ist sinnvoll, sich dafür eine eigene Prozedur zu schreiben, z. B. die Prozedur `Anzeigen`. Denke daran, dass die Prozedur Zugriff auf das Formular haben muss, da sie die Edit-Felder neu belegt. Du musst deshalb den Prozedurkopf im `private`-Bereich der Formulardefinition eintragen.

Aufgabe 15: Ändere dein Projekt wie oben angedeutet, indem du die zum Formular zugehörige Prozedur `Anzeigen` implementierst und in den Navigationsbuttons verwendest.

Wenn du jetzt schon eine Prozedur zum Anzeigen des Aktuellen Datensatzes auf dem Formular erstellt hast, so solltest du diese Prozedur auch grundsätzlich verwenden. In den Ereignisroutinen des Formulars für das Einfügen und Löschen eines Datensatzes wird ebenfalls eine Anzeige des aktuellen Datensatzes nötig. Verwende also auch hier die Prozedur `Anzeigen`. Weiterhin wird auch bei der Erstellung einer neuen Datenbank (Initialisierung) eine Ausgabe (von leeren Zeichenketten für Name und Telefonnummer) auf die Edit-Felder nötig. Verwende also auch dort die Prozedur `Anzeigen`.

Methoden zur Bearbeitung der Datensätze – das Ändern und Suchen

Im Zusammenhang mit der Verwaltung der Datensätze stehen jetzt noch die beiden Methoden für (a) das Suchen und (b) das Ändern eines Datensatzes aus.
Den Teil (b) überlasse ich dir alleine. Als Hinweis soll dir nur dienen, dass das Ändern eines Datensatzes damit zu bewerkstelligen ist, dass der betreffende Datensatz zuerst gelöscht wird und anschließend in geänderter Form wieder neu angelegt wird. Die Implementierung der Methode `procedure TDatenbank.DatensatzAendern(date: TEintrag)` läuft also auf die Verwendung der Methoden `DatensatzLoeschen` und `DatensatzAnlegen` heraus.

Die Frage nach dem Suchen eines Datensatzes ist da schon weitaus interessanter. Der Methodenaufruf DB.`DatensatzSuchen(´Hans´)` soll z. B. bewirken, dass die Datenbank alle Datensätze nach dem Namen „Hans“ durchsucht. Wird ein Datensatz gefunden, so sollte die aktuelle Position auf diesem Datensatz stehen bleiben. Ansonsten sollte die Position unverändert bleiben.

Die zugehörige Formular-Routine sollte folgendes machen:

```
procedure TForm1.B_MI_Suchen(Sender: TObject);
var name: string;
begin
```
- Namen eingeben lassen (DELPHI-Funktion InputBox)
- Aufruf der Datenbank-Methode DatensatzSuchen
- Anzeige des aktuellen Datensatzes

```
end;
```

Die Implementierung der drei Befehle dürfte dir nicht zu schwer fallen. Allerdings hat es die Datenbank-Routine schon etwas mehr in sich.

Aufgabe 16: Entwickle zuerst selbst eine Idee, wie das Suchen ablaufen könnte. Versuche auch, die zugehörige Methode selbst zu implementieren.

```
procedure TDatenbank.DatensatzSuchen(Name: string);
begin
  ...
end;
```

Wahrscheinlich durchläufst du in deinem Algorithmus die Datensätze von vorne nach hinten, bis du einen Datensatz gefunden hast, dessen Name mit dem Suchnamen übereinstimmt. Diese Art der Suche (man nennt sie die ***sequentielle Suche***) ist prinzipiell in Ordnung und würde wie folgt implementiert:

```
procedure TDatenbank.DatensatzSuchen(Name: string);
begin
  if (Anzahl > 0)     { nur dann gibt es auch was zu suchen... }
  then begin
         Position:= 0;
         repeat
           inc(Position);
         until (Eintraege[Position].Name = Name) or
               (Position = Anzahl)
       end;
end;
```

Dieser Algorithmus stoppt spätestens beim letzten Datensatz, falls kein entsprechender Eintrag gefunden wurde. Ansonsten bleibt die Position auf dem gefundenen Datensatz stehen.

Das Suchverfahren funktioniert, allerdings kann ein Suchvorgang nach dem Namen „Walter“ unter Umständen ziemlich lange dauern, da er erst sehr weit am Ende der Datenbank zu finden ist.
Wenn du z. B. den Namen „Müller“ im Telefonbuch suchst, so fängst du wahrscheinlich auch nicht mit der ersten Seite des Telefonbuchs an – du schlägst vielmehr die Mitte des Telefonbuchs auf. Das Suchverfahren funktioniert allerdings nur deswegen, weil du weißt, dass die Namen im Telefonbuch alphabetisch sortiert sind (zum Glück!).
Du wirst jetzt ein Suchverfahren kennen lernen, welches die Sortierung unserer Daten geschickt ausnutzen, um wesentlich schneller ans Ziel zu gelangen – die ***binäre Suche***.

ALGORITHMUS ***Binäre Suche***

Input: Suchelement: string
Hilfsobjekte: links, rechts, mitte: integer
Global: Konstante: maxAnzahl
Variablen der Datenbank: Eintraege, Anzahl, Position, DateiGeaendert

- links ← 1
- rechts ← Anzahl
- Wiederhole
  - mitte ← (links + rechts) DIV 2
  - Falls Suchelement > Eintraege[mitte].Name
    - dann • rechts ← mitte – 1
    - sonst • links ← mitte + 1

  bis (links > rechts) oder (Suchelement = Eintraege[mitte].Name)
- Position ← mitte

Aufgabe 17: Test das Suchverfahren mit dem Suchbegriff *Carla* an folgender Datenbankbelegung aus (Anzahl = 10). Was passiert, wenn z. B. der Name Hans gesucht wird?

1	2	3	4	5	6	7	8	9	10	11	...
Anton 51189	Berta 4821	Bruno 88112	Carla 12345	Dieter 9448	Emil 3629	Frida 110	Moni 228	Paul 112	Uwe 59992		...

Aufgabe 18: Implementiere das Suchverfahren in DELPHI.

So, nun noch wie versprochen die Routine zum Löschen eines Datensatzes:

```
procedure TDatenbank.DatensatzLoeschen;
var i: integer;
begin
  if Anzahl>0
  then begin
         for i:= Position to Anzahl-1
           do Eintraege[i]:= Eintraege[i+1];
         Dec(Anzahl);
         if Position>Anzahl then Dec(Position);
         DateiGeaendert:= true;
       end;
end;
```

Methoden zur Dateiverwaltung – Speichern und Laden

So weit so gut, allerdings müsste man bei dem aktuellen Stand des Projekts bei jedem Programmstart sämtliche Daten neu eingeben. Besser wäre es da, wenn man alle Daten gesammelt in eine Datei auf der Festplatte (oder andere Datenträger) speichern könnte. Und genau darum soll es in diesem Kapitel gehen.

Das **Speichern und Laden** einer Datei innerhalb eines eigenen Projektes erfolgt am besten über die beiden von DELPHI schon zur Verfügung gestellten Dialoge **OpenDialog** und **SaveDialog**. Du findest diese beiden Oberflächenobjekte in der Werkzeugleiste **Dialoge**.

Der „OpenDialog“

Verwende die Komponente TOpenDialog, um einen Windows-Standarddialog zum Öffnen von Dateien zu erzeugen, mit dessen Hilfe die Benutzer den Namen einer zu öffnenden Datei eingeben können. Mit der Methode ***Execute*** (eine boolesche Function) rufst du das Dialogfenster auf. Der vom Benutzer im Dialogfenster eingegebene Dateiname wird von DELPHI der Eigenschaft *FileName* als Wert zugewiesen. Verwende den Wert dieser Eigenschaft (*FileName*), um (aus dem Programm heraus) die Datei zu bestimmen, welche du öffnen willst. Beachte insbesondere die nachstehenden Eigenschaften:

- die Eigenschaft *DefaultExt* legt die Standardnamenserweiterung fest.
- die Eigenschaft *Filter* legt einen Filter bzw. eine Selektionsmaske fest, mit deren Hilfe für die Anzeige der Dateinamen eine Auswahl unter den Dateien getroffen werden kann.
- mit der Eigenschaft *FilterIndex* kann man den Filter mit Vorgabewerten initialisieren.
- die Eigenschaft *InitialDir* stellt das Arbeitsverzeichnis ein.
- mit der Eigenschaft *Title* änderst du die Titelzeile des Dialogs Öffnen.

Der „SaveDialog“

Analog zur Komponente TOpenDialog kannst du den SaveDialog verwenden. Mit der Methode ***Execute*** kannst du dieses Dialogfenster in deinem Programm zur Laufzeit erzeugen, also entsprechend dem OpenDialog-Fenster. Schau dir eventuell die Hilfe zu den Eigenschaften die Liste der *Options* an: ... *ofReadOnly* ... *ofOverwritePrompt* ...

Beispiel zum „SaveDialog“

Klickt der Benutzer in deinem Projekt z. B. auf den Menüpunkt Datei öffnen, so solltest du in der zugehörigen Ereignisroutine des Formulars folgende Programmzeilen haben:

```
procedure TForm1.MI_DateiOeffnen(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    SaveDialog1.Filename := OpenDialog1.FileName;
    DB.DateiSpeichern(OpenDialog1.FileName)
    Form1.Caption := ´Datenbank - ´+DB.Dateiname;
    { ... }
  end;
end;
```

Wie du siehst, ist die Methode ***Execute*** funktional realisiert! Bei Auswahl des OK-Buttons innerhalb des Dialogfensters liefert die Funktion *Execute* den Wert *true*. In dem Fall wird der THEN-Teil ausgeführt: die Eigenschaft *FileName* des Dialogfensters Speichern wird auf den gleichen Dateinamen gesetzt und der ausgewählte Dateiname wird zudem als Titel des Formulars angezeigt. Abschließend wird die Datenbank-Methode `TDatenbank.DateiSpeichern(...)` aufgerufen, um welche wir uns jetzt im Anschluss kümmern müssen.

Aufgabe 19: Erweitere dein Projekt um die Formular-Ereignisroutinen zum Speichern und Laden einer Datei.

Die Methoden der Datenbank

Die Verwendung der Dialoge war etwas knifflig, dafür wird das Speichern der Datenbank etwas einfacher, da die zugehörige Algorithmik im wesentlichen mit der der *sequentiellen Suche* übereinstimmt.
DELPHI stellt uns die Datenstruktur **`FILE Of <SatzTyp>`** zur Verfügung. Als *Satztyp* kannst du Standardtypen wählen oder – was wir hier benötigen – eigene, von dir selbst definierte Typen; z.B.:

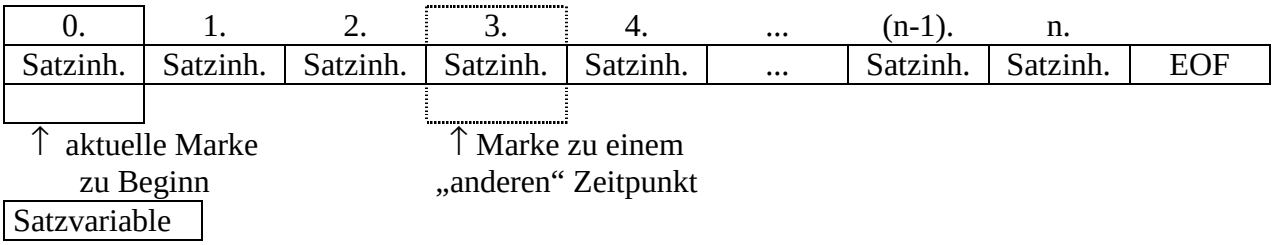
```
Beispiel 1: Type TEintraege = Array [1..maxAnzahl] of TEintrag;
            TDatei     = File of TEintraege;

Beispiel 2: Type TEintrag   = Record
                                { Komponenten des Datensatzes }
                              end;
            TDatei     = File of TEintrag;
```

Die beiden Beispiele weisen schon darauf hin, wie du deine Daten in der Datei organisieren kannst: einerseits als ganzes Array, welches komplett gespeichert und geladen wird, oder als Aneinanderreihung von beliebig vielen einzelnen Datensätzen vom Typ `TEintrag`. Welche Vorteile hat es, wenn du die zweite Variante wählst? Welche Vorteile gibt dir die erste Variante?

Am einfachsten ist die erste Möglichkeit, denn wie du mit dem ARRAY zu arbeiten hast, das weißt du bereits. Und dein FILE würde hier nur aus einer einzigen Komponente bestehen, nämlich dem ganzen ARRAY. Nehmen wir dennoch aufgrund der vielen Vorteile die zweite Möglichkeit. Der Befehl

```
var Datei: TDatei;
```

deklariert dir eine *Filevariable*. Mit dieser ist es nun möglich, eine Datei zu öffnen, zu speichern oder zu verändern. Allgemein kannst du dir ein File wie ein durchnummeriertes und an einer Stelle markiertes Band – z. B. wie einen Filmstreifen – vorstellen. Jedem einzelnen Bild des Films entspricht in der Datei ein einzelner Datensatz, welche hintereinander gereiht werden.



Im Prinzip ist die interne Speicherung unserer Datenbank ganz ähnlich aufgebaut. Der Variablen `Position` entspricht hier die aktuelle *Filemarke*, der Variablen `Anzahl` die Dateibegrenzung EOF (*End Of File*). Allerdings hast du im Gegensatz zum ARRAY nur den Zugriff auf einen einzigen Datensatz – und zwar genau auf den durch die *Filemarke* bezeichneten Datensatz! Zu Beginn, nach dem Öffnen des Files, steht diese Marke auf dem ersten, mit der Nummer 0 (!) bezeichneten Datensatz.

Eine Variable vom Typ `File Of ...` hat viele Informationen zu verwalten, wie z.B. den *Filenamen*, die *Datensatzgröße*, die *Filegröße*,... Außerdem ist beim Arbeiten mit einer *Filevariablen* die Reihenfolge zu beachten, in der man ihre Operationen verwendet. In der nachstehenden Liste dieser Operationen sei **`d`** die *Filevariable*:

erster Schritt,	zweiter Schritt,		dann...	... und Schluss.
`AssignFile(d,<extname>)`	**`Reset(d)`** **`Rewrite(d)`**	bzw.	**`Read(d,<SatzVar>)`** **`Write(d,<SatzVar>)`**	**`CloseFile(d)`**

Mit *Rewrite* legst du eine neue Datei an, mit *Reset* bearbeitest du eine schon existierende Datei. Die Parameterlisten der Operationen *Read* und *Write* zeigen, dass du für den jeweiligen Datensatz eine passende „Puffervariable“ brauchst – in der obigen File-Darstellung „Satzvariable“ genannt –, die den zu lesenden Datensatzinhalt aufnehmen kann oder einen zu speichernden Datensatz abgibt.
Die Operationen *Read* und *Write* verschieben im übrigen jedes mal die *Filmarke* um eine Position nach rechts.

Aufgabe 20: Analysiere das folgende Programmsegment und implementiere dann die Datenbank-Methoden `DateiOeffnen(Dname: string)` und `DateiSpeichern(Dname: string)`.

```
AssignFile(d, OpenDialog.Filename);
Reset(d);
While not EOF(d) Do
Begin
    Read(d, inhalt);
    zeige(inhalt)
End;
CloseFile(d);
```

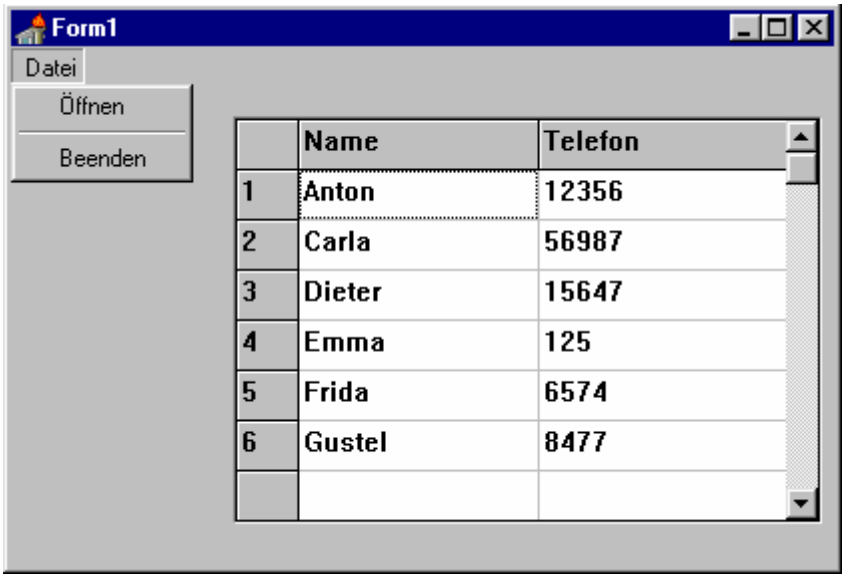
Wenn du es nicht geschafft hast, so findest du im Anschluss an dieses Kapitel die Lösung für das Speichern und Öffnen der Datenbank. Allerdings solltest du es schon alleine probiert haben.

So, jetzt müsste deine Datenbank eigentlich gut funktionieren. Man sollte nun in deinem Projekt eine neue Datenbank anlegen können, man sollte Datenbanken speichern und öffnen können. Die Bearbeitung der Datensätze umfasst das neue Anlegen, Löschen, Ändern und Suchen von Datensätzen. Perfekt.

Allerdings haben wir die ganze Zeit die zur Datenbank zugehörige Variable `DateiGeaendert` aktualisiert, ohne diese nur einmal sinnvoll zu gebrauchen. Das sollte sich ändern. Zum Beispiel sollte beim Verlassen des Programms ein Warnhinweis kommen, falls die Datei noch nicht gespeichert wurde. Genauso könnte eine Sicherheitsabfrage das versehentliche Öffnen einer Datenbank unterbinden.

Aufgabe 21: Mache dein Datenbank-Programm sicherer, indem du durch Sicherheitsabfragen verhinderst, dass ein Benutzer eine geänderte (`DateiGeaendert = true`) Datenbank ungespeichert verliert.

Aufgabe 22: Erstelle ein ganz neues Projekt, in dem der Benutzer eine bestehende Datenbank von der Festplatte öffnen kann und welches ihm daraufhin alle Datensätze in einem StringGrid (du kennst es vom Lottoprojekt) anzeigt. Verwende deine Datenbank-Unit.



Aufgabe 23: **a)** Die Datensätze sollen neben dem Namen und der Telefonnummer auch die Anschrift (Strasse, Hausnummer, Postleitzahl und Wohnort) sowie die email-Adresse der jeweiligen Person aufnehmen. Ändere die Datenbank entsprechend ab.

b) Schreibe einen Algorithmus, der erkennt, ob ein eingegebener `string` eine email-Adresse ist. [Es muss ein Klammeraffe (@) im `string` vorkommen.]

c) Erweitere die Datenbank um folgende Datenbankoperationen:

```
function  TDatenbank.DateiLeer: boolean;
function  TDatenbank.DateiVoll: boolean;
procedure TDatenbank.NummerSuchen(nummer: string);
procedure TDatenbank.NummernSortierung;
procedure TDatenbank.NamenSortierung;
```

Wie versprochen kommen jetzt noch die Datenbank-Methoden zum Speichern und Öffnen einer Datenbank:

```
procedure TDatenbank.DateiOeffnen(Dname: string);
var Datei: File of TEintrag;
    i: integer;
begin
  AssignFile(Datei,Dname);          { Der Filevariablen das zu     }
                                    {   öffnende File zuweisen     }
  Reset(Datei);                     { Datei zum Lesen öffnen.      }
    i:= 0;
    while not EOF(Datei)            { Solange noch keine File-Ende }
    do begin
         inc(i);
         Read(Datei, Eintraege[i]); { aktuellen Datensatz lesen    }
       end;
  CloseFile(Datei);                 { File schließen               }
  Anzahl:= i;
  if Anzahl>0 then Position:= 1     {Aktuelle Position anpassen    }
              else Position:= 0;
  Dateiname:= Dname;                { Der intern gespeicherte      }
                                    { Dateiname wird aktualisiert. }
  DateiGeaendert:= false;
end;
```

und

```
procedure TDatenbank.DateiSpeichern(Dname: string);
var Datei: File of TEintrag;
    i: integer;
begin
  AssignFile(Datei,Dname);          { Der Filevariablen das zu     }
                                    {   öffnende File zuweisen     }
  Rewrite(Datei);                   { Datei zum Lesen öffnen.      }
    for i:= 1 to Anzahl
      do Write(Datei, Eintraege[i]);{ aktuellen Datensatz speichern}
  CloseFile(Datei);                 { File schließen               }
  DateiGeaendert:= false;
end;
```

Du hast in den letzten beiden Aufgaben gesehen, wie schnell nun deine bestehende Datenbank erweitert werden kann und wie einfach sich diese in andere Programme einbinden lässt. Ein wesentlichen Nachteil hat die ganze Sache aber noch. Die maximale Anzahl der Datensätze ist von vorneherein festgelegt. Zwar könntest du die Konstante `maxAnzahl` erhöhen, aber irgendwann ist Schluss. Ärgerlich ist außerdem, dass man sich zu Anfang festlegen muss, wie viele Datensätze maximal zugelassen werden. Will man diese Maximalanzahl ändern, so muss man den Quellcode deines Programms ändern. Allerdings ist es nicht üblich, Programme als Quellcode zu verkaufen – oder hast du den Quellcode von Microsoft Word? – also muss es eine andere Möglichkeit geben, mit der die Anzahl der Datensätze nicht festgelegt werden muss.
Die Festplatte hat ein wesentlich größeres Speichervolumen als der RAM. Außerdem hast du beim Speichern der Datenbank gesehen, dass die angelegte Datei nur so viele Datensätze umfasste, wie wirklich benötigt wurden. Das ARRAY dagegen hatte eine Menge ungenutzter Datensätze. Also, warum sollten wir unsere Daten nicht sofort auf der Festplatte verwalten? Microsoft Word macht es genauso, denn das Programm weiß ja auch nicht von vorneherein, wie viele Seiten du mit dem Programm schreiben willst. Also legt sich Word eine Datei auf der Festplatte an und sämtliche Änderungen des Textes werden auf dieser Datei durchgeführt.

Projekt „Minidatenbank“ – externe Datenverwaltung

Wenn die Daten der Datenbank nicht mehr intern in einem ARRAY, sondern extern in einer Datei verwaltet werden, so muss sich zwangsläufig die Typdeklaration der Datenbank ändern:

```
unit U_Dat_ex;

interface

{ Eine Konstantendeklaration benötigen wir nicht mehr. }

type  TEintrag  = record
                    Name, Telefon: string;
                  end;
      TEintraege= File of TEintrag;
      TDatenbank= class
                                    { DATEN der Klasse TDatenbank }
        Eintraege: TEintraege;
        Position: integer;
        Anzahl: integer;
        DateiGeoeffnet: boolean;
        Dateiname: string;
                                    { METHODEN der Klasse TDatenbank }
... { wie vorher }
```

Es gibt in der Typ-Deklaration zwei wesentliche Unterschiede zur ersten Version:

- Es gibt keine Beschränkung der maximalen Anzahl an Datensätzen, da die Variable `Eintraege` eine *Filevariable* ist. Deshalb fällt die Konstantendeklaration weg.
- Das Flag DateiGeaendert ist überflüssig, da alle Veränderungen der Datenbank in jedem Fall auf der Festplatte vorgenommen werden. Ein versehentliches Verwerfen der Änderungen an der Datenbank ist damit ausgeschlossen. Dafür muss aber sichergestellt sein, dass eine Datei geöffnet vorliegt. Ansonsten sind Änderungen an der Datenbank nicht möglich. Deshalb wird ein Flag `DateiGeoeffnet` als boolesche Variable eingefügt.

Die Datenbankoperationen bleiben algorithmisch gesehen gleich. Allerdings kann man nun nicht mehr so einfach auf einen einzelnen Datensatz zugreifen, wie bisher, denn öffnet man eine Datei, so steht die *Filemarke* auf dem ersten Datensatz und nicht auf dem Datensatz, welcher durch die Variable `Position` markiert ist. DELPHI bietet dir allerdings einen Befehl, mit dem du die *Filemarke* auf einen beliebigen Datensatz setzen kannst:

`Seek(<Dateivariable>, <Positionsindex>)`

Die Verwendung dieses Befehls wird am deutlichsten, wenn du dir die beiden Datenbankoperationen DatensatzLesen sowohl in der ersten Fassung als auch in der Fassung mit externer Datenverwaltung anschaust:

Zur Erinnerung, dies war die alte Fassung:

```
procedure TDatenbank.DatensatzLesen(var date: TEintrag);
begin
  if (Position = 0)
    then begin
           date.Name:= ´´;
           date.Telefon:= ´´;
         end
    else date:= Eintraege[Position];
end;
```

In der Datenbank-Unit mit externer Speicherung müsste diese Routine wie folgt formuliert werden:

```
procedure TDatenbank.DatensatzLesen(var date: TEintrag);
begin
  if (Position = 0) or (not DateiGeoeffnet)
    then begin
           date.Name:= ´´;
           date.Telefon:= ´´;
         end
    else begin
           Seek(Eintraege, Position – 1)
           Read(Eintraege, date);
         end;
end;
```

Vielleicht wirst du jetzt stutzig, dass im Seek-Befehl auf Position – 1 zugegriffen wird? Die Datensätze in Dateien sind mit 0 beginnend durchnummeriert, nicht mit 1 beginnend wie es im ARRAY der Fall war.

Weitere nützliche Befehle im Zusammenhang mit Dateien sind die folgenden:

`FilePos(<Dateivariable>)`

ist eine Funktion, welche dir die Nummer des Datensatzes ausgibt, auf der die *Filemarke* aktuell steht. Der Befehl `Seek(Eintraege, FilePos(Eintraege)–1)` bewirkt also, dass die *Filemarke* um eine Stelle zurückgesetzt wird.

`FileSize(<Dateivariable>)`

ist eine Funktion, welche dir die Anzahl der gespeicherten Datensätze ausgibt. Der Befehl `Seek(Eintraege, FileSize(Eintraege))` bewirkt also, dass die *Filemarke* hinter den letzten Datensatz gesetzt wird. Dies kann sehr nützlich beim Anfügen eines neuen Datensatzes sein.

`Erase(<Dateivariable>)`

ist eine Prozedur, welche die momentane Datei von der Festplatte löscht.

`Truncate(<Dateivariable>)`

ist eine Prozedur, welche alle Datensätze ab dem durch die *Filemarke* markierten Datensatz abschneidet. Diese Prozedur benötigst du, wenn ein Datensatz gelöscht wird.

`Rename(<Dateivariable>,<Dateiname>)`

ist eine Prozedur, welche der aktuellen Datei einen neuen Dateiname verpasst.

Mit diesen Befehlen müsste es dir eigentlich gelingen, die Datenbank-Unit auf die externe Verwaltung von Datensätzen umzustellen. Am einfachsten ist es, wenn du die bereits bestehende Datenbank-Unit unter einem anderen Namen abspeicherst und dann nur noch die nötigen Änderungen vornimmst.

Aufgabe 24: Implementiere die Unit `U_Dat_ex.pas`, mit der eine Datenbank extern verwaltet wird. Da sich die Namen der Datenbankoperationen nicht geändert haben, sollte dein Projekt auch mit dieser Unit funktionieren. Probiere es aus.

```
Unit Unit1;                                  {Formularunit }
uses SysUtils, WinTypes, ..., U_Dat_ex;
       ...
var Form1: TForm1;
    DB: TDatenbank;
...
```

Aufgabe 25: Teste aus, ob auch das Programm aus Aufgabe 22 mit dieser Unit klar kommt.