3 Vielfältige Speichermöglichkeiten – strukturierte Datentypen

Delphi 6 lernen
ISBN 3-8273-1776-2

Was Sie in diesem Kapitel lernen:

- die Verwendung von Unterbereichstypen
- die Nutzung von Array-Typen
- die Anwendung von Record-Typen
- Grundlagen der Mengentypen
- die Verwendung von Dateitypen und Streams zur Datenspeicherung
- die zwei Arten von Strings

Sie haben bereits einige der in Delphi verwendeten Typen kennen gelernt. Sie kennen die einfachen Typen, zu denen ordinale Typen wie Integer-Typen und Aufzählungstypen sowie die Real-Typen zählen. Auch über Arrays und Strings haben Sie bereits etwas gehört.

Strukturierte Typen

Die einfachen Typen sind recht schnell zu erklären und zu handhaben. Oft sind diese Typen aber zu unflexibel oder einfach ungeeignet, um mit komplexeren oder größeren Datenstrukturen umzugehen. Deshalb gibt es in Delphi eine Reihe von strukturierten Typen, die aus den einfachen Typen zusammengesetzt sind und einfache Mittel zur Handhabung komplexer Datenstrukturen bereitstellen. Die Erklärung dieser Typen ist ebenfalls nicht schwierig; allerdings brauchen Sie etwas mehr Übung, um diese Typen auch so verwenden und kombinieren zu können, dass sie ein Maximum von Effizienz und Übersichtlichkeit in Ihren Programmen gewährleisten.

Sie werden daher hier auch keine Referenz der verschiedenen Typen finden – dazu ist die Online-Hilfe da. Stattdessen werden Sie die Verwendung der Typen an einfachen Beispielen üben. Sie finden in diesem Kapitel fast alle Arten von strukturierten Typen – lediglich die Klassentypen werden erst in Kapitel 5 behandelt.

Die meisten Beispielprogramme in diesem Kapitel schreiben auf die Festplatte, deshalb sollten Sie diese von der CD auf Ihre Festplatte kopieren, falls Sie sie ausprobieren möchten.

3.1 Die MaskEdit-Komponente

Sie beginnen diesmal gleich mit einer Übung, da Sie die *MaskEdit*-Komponente in diesem und im nächsten Kapitel benutzen werden. Diese Komponente stellt eine Erweiterung der *Edit*-Komponente dar. Mit der *MaskEdit*-Komponente kann die Eingabe unerwünschter Zeichen unterbunden werden; außerdem können Sie die Maske zur Formatierung der Datenanzeige verwenden. So wie eine Maske Teile des Gesichtes verdeckt, so „verdeckt" in unserem Fall die *MaskEdit*-Komponente Teile der Eingabemöglichkeiten – durch diese Komponente wird festgelegt, was in das Editierfeld eingegeben werden darf.

Erzeugen Sie jetzt eine neue Anwendung, damit Sie die folgenden Erläuterungen nachvollziehen können.

Platzieren Sie eine **MaskEdit**-Komponente auf dem Formular. Sie finden diese Komponente im Register ZUSÄTZLICH an dritter Stelle von links. Die Platzierung einer Komponente haben Sie schon so oft vorgenommen, dass es dazu keiner Beschreibung mehr bedarf. Lesen Sie erst die nachstehende Erläuterung, bevor Sie Veränderungen vornehmen!

Die Komponente anpassen

Eigenschaft	Voreingestellter Wert	Neuer Wert
EditMask		999;0;_
Text		0

Tabelle 3.1: Die wichtigsten Einstellungen von MaskEdit

EditMask

Die Eigenschaft *EditMask* muss näher erklärt werden. Nachdem Sie auf die Zusatzschaltfläche in der Wertespalte der Eigenschaft *EditMask* geklickt haben, öffnet sich der in Abbildung 3.1 gezeigte Dialog. In diesem Dialog wird die Eingabemaske definiert; es wird also festgelegt, welche Zeichen zur Laufzeit in das Editierfeld eingegeben werden können.

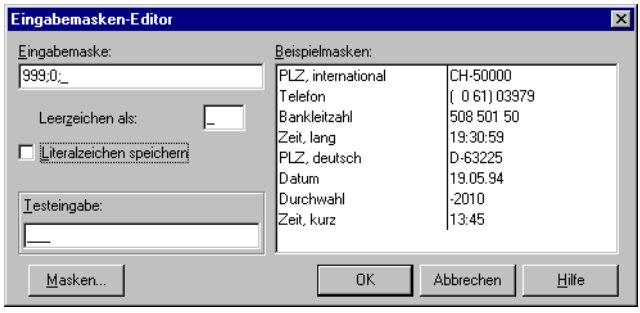

Abbildung 3.1: Der Dialog der Eigenschaft „EditMask"

Zunächst werden Sie der Komponente *EditMask* eine Eingabemaske zuweisen, sodass nur Zahlen eingegeben werden können. Führen Sie die nachfolgenden Schritte aus:

1. Geben Sie in das Feld EINGABEMASKE drei Neunen ein. Die Neun steht für eine beliebige ganze Zahl. Damit erzwingen Sie, dass nur Ziffern eingegeben werden können, und zwar maximal drei.
2. Klicken Sie auf das Kästchen mit der Bezeichnung *Literalzeichen speichern*, damit der Haken verschwindet.
3. Wenn Ihr Dialog jetzt so aussieht wie in Abbildung 3.1, dann klicken Sie auf OK.
4. Nun tragen Sie den Anfangswert der Eigenschaft *Text* ein, wie aus ersichtlich.

Literalzeichen

Unter einem *Literalzeichen* versteht man ein Zeichen, das als Teil des Zeichensatzes aufgefasst wird. Wenn Sie eine Zahl in das Editierfeld eingeben, kann diese auf zwei Arten behandelt werden – als Zahlenwert oder als Zeichenkette. Ist das Kästchen angekreuzt, dann wird nicht die Zahl selbst, sondern ihre Zeichendarstellung gespeichert. Die Deaktivierung des Kästchens bedeutet dagegen, dass die Eingabe tatsächlich auch als nummerischer Wert (als Zahl) verwaltet und gespeichert wird.

In der Darstellung der Eingabemaske entspricht der zweite Teil (die Null) der abgeschalteten Literaldarstellung. Der dritte Teil (der Unterstrich) zeigt an, wie ein Leerzeichen in der Eingabe dargestellt wird.

Das Kästchen ist, nebenbei bemerkt, eine Komponente mit dem Namen *CheckBox* (Kontrollkästchen). Sie finden sie im Standardregister der Komponentenpalette. Die *CheckBox* schaltet Zustände an und aus.

Eine Prozedur schreiben

Ein *OnChange*-Ereignis reagiert, wie schon erwähnt, auf jede Änderung, die der Anwender im Editierfeld vornimmt. Sie werden jetzt ein solches Ereignis programmieren, das dafür sorgt, dass das Editierfeld nicht leer bleibt – in ein leeres Feld wird automatisch eine Null geschrieben. Wir gehen davon aus, dass Sie gerade den *EditMask*-Dialog geschlossen haben.

Dann gehen Sie wie folgt vor:

1. Wechseln Sie auf die Ereignisseite der *MaskEdit*-Komponente.
2. Doppelklicken Sie in die Wertespalte des *OnChange*-Ereignisses.
3. Tragen Sie dann die aufgeführten Zeilen ein.

```
procedure TForm1.MaskEdit1Change(Sender: TObject);
begin
  if MaskEdit1.Text = '' then // wenn leer, dann auf 0 setzen
    MaskEdit1.Text := '0';
end;
```

Das Gleichheitszeichen vergleicht den aktuell eingegebenen Text (*MaskEdit1.Text*) mit einem Leerstring. Ist der String leer, dann wird ihm das Zeichen `'0'` zugewiesen.

Wiederholen Sie die notwendigen Schritte für eine zweite *MaskEdit*-Komponente mit dem Namen *MaskEdit2*, und setzen Sie die Eigenschaft *Text* auf 100.

In der *Edit*- und der *MaskEdit*-Komponente arbeiten Sie immer mit Zeichen. Auch wenn Sie Ziffern eingeben, werden diese zunächst als Zeichen behandelt. Möchten Sie mit den Eingaben rechnen, müssen Sie diese mit der Funktion *StrToInt* umwandeln.

Speichern und starten Sie das Programm, und versuchen Sie, Buchstaben einzugeben oder alles zu löschen – es wird Ihnen nicht gelingen!

3.2 Unterbereichstypen

Bevor die Array-Typen behandelt werden, wollen wir zunächst noch die Unterbereichstypen (oder Teilaufzählungstypen) besprechen. Sie nutzen Unterbereichstypen, wenn Sie die Anzahl der Werte, die eine Variable annehmen kann, beschränken wollen – genauso wie bei den Aufzählungstypen, die Sie bereits kennen. Ein Unterbereichstyp kann immer dann verwendet werden, wenn alle enthaltenen Werte (alle Werte, die eine Variable dieses Typs annehmen kann) direkt aufeinander folgen. In diesem Fall kann der Bereich eindeutig durch die Angabe des jeweils kleinsten und größten Wertes bestimmt werden.

Unterbereichstypen werden, wie Aufzählungstypen, zur Erhöhung der Typsicherheit verwendet, also zur Vermeidung von Fehlern, die durch die Verwendung falscher Typen oder falscher Bereiche entstehen. Außerdem ermöglicht die Definition eigener Unterbereichstypen (die üblicherweise am Anfang einer Unit stehen) auch eine einfache Änderung von Bereichen.

Es folgen einige Beispiele, welche die Anwendung verdeutlichen:

```
type
  TMessbereich = -100..100;
  TToleranzbereich = 0..50;
  TStunden = 0..23; // 24 Stunden
  TMinuten = 0..59; // 60 Minuten
```

Die Unterbereichstypen werden durch einen Minimalwert, gefolgt von zwei Punkten und einem Maximalwert dargestellt; dadurch wird ein Bereich präsentiert.

Sie können Unterbereichstypen (Teilaufzählungstypen) auch aus vorhandenen Aufzählungstypen bilden. Dann wirken sich Änderungen im Aufzählungstyp auch auf den Unterbereichstyp aus:

```
type
  TWochentage = (wtMontag, wtDienstag, wtMittwoch, wtDonnerstag, wtFreitag,
    wtSamstag, wtSonntag);
  TWerktage  = wtMontag..wtFreitag;
var
  werktage: TWerktage; // kann nur Werte zwischen Montag und Freitag anne-
hmen
```

Wie Sie sehen, erspart Ihnen ein sinnvoll eingesetzter Unterbereichstyp Schreibarbeit.

Den folgenden Aufzählungstyp kennen Sie aus der Abschlussübung in Kapitel 2 – es ist ein in Delphi definierter Aufzählungstyp, der die Ausrichtung eines Textes bestimmt. Der Aufzählungstyp sieht folgendermaßen aus:

```
  // links, rechts oder zentriert ausrichten
  TAlignment = (taLeftJustify, taRightJustify, taCenter);
```

Der folgende Unterbereichstyp wird zum Beispiel für die Komponente *CheckBox* in Delphi genutzt. Schauen Sie sich die Eigenschaft *Alignment* im Objektinspektor an. Sie können nur zwischen links ausrichten (*taLeftJustify*) und rechts ausrichten (*taRightJustify*) wählen.

```
  // Text nur links oder rechts
  TLeftRight = taLeftJustify..taRightJustify;
```

3.2.1 Ein Unterbereichstyp

Sie sehen, dass es an einem Unterbereichstyp nicht viel Neues gibt. Ein kleines Beispiel soll die Theorie veranschaulichen. Sie können im folgenden Beispiel nur das kleine Einmaleins durchführen. Wieso und warum, wird nach der Fertigstellung der Anwendung erklärt.

Zunächst sollten Sie die nachstehend aufgeführten Schritte durchführen. Auf die Darstellung des Formulars wird hier verzichtet, denn es gibt nichts zu sehen, was Sie nicht schon kennen.

1. Erstellen Sie eine neue Anwendung mit zwei **MaskEdit**-Komponenten und einer **BitBtn**-Komponente.
2. Ändern Sie die Eigenschaft *EditMask* der beiden Eingabefelder, wie Sie es in der Übung zuvor getan haben.
3. Erzeugen Sie für die beiden *MaskEdit*-Komponenten die entsprechenden *OnChange*-Ereignisse, die ein leeres Eingabefeld auf Null setzen.
4. Für die Schaltfläche genügt die Erzeugung des Prozedurrumpfes; die Vergabe des Namens ist Ihnen überlassen. Sie wissen, worauf es bei Bezeichnern ankommt. Hier wurde der Bezeichner *IstGleichBB* verwendet.
5. Fügen Sie die entsprechenden Zeilen ein. Die Erklärung erfolgt nach dem Abdruck des Quelltextes.

Schreiben Sie den Quelltext nicht mechanisch ab, sondern versuchen Sie mit Hilfe der Kommentare, den Zweck der Zeilen zu erfassen. Achten Sie auch auf die Bezeichner, die Sie verwendeten, falls Sie sich für andere Bezeichner entschieden haben sollten.

```
{ 43:} {$R+} // Bereichsprüfung anschalten
{ 44:}
{ 45:} procedure TForm1.IstGleichBBClick(Sender: TObject);
{ 46:} type
{ 47:}   TKleinesEinmaleins = 0..100; // Unterbereichstyp
{ 48:} var
{ 49:}   produkt: TKleinesEinmaleins;
{ 50:} begin
{ 51:}   // Multiplikation wird durchgeführt
{ 52:}   produkt := StrToInt(Fakt1ME.Text) * StrToInt(Fakt2ME.Text);
{ 53:}   // Bei einem Bereichsfehler wird hier abgebrochen
{ 54:}   Label2.Caption := IntToStr(produkt);
{ 55:} end;
```

Bereichsüberprüfung

Der Ausdruck in Zeile 43 ist, wie Sie wissen, kein Kommentar, sondern eine Compiler-Direktive; sie schaltet die Bereichsüberprüfung ein. Sie können die Bereichsüberprüfung zwar auch über die Compi-

ler-Optionen (Menüoption PROJEKT | OPTIONEN) einstellen; mit der Compiler-Direktive haben Sie aber die Sicherheit, dass diese Einstellung auf jeden Fall aktiv ist.

Es ist, insbesondere bei größeren Projekten, immer günstig, wichtige Compiler-Einstellungen auch als Compiler-Direktive in den Quelltext zu schreiben. Wenn Sie den Cursor in den Quelltext setzen und die Tastenkombination Strg+O und danach O drücken, dann werden oben in die Datei alle Compiler-Direktiven geschrieben, die den zurzeit gültigen Einstellungen entsprechen. Wenn Sie sicher sein wollen, dass Ihre Dateien auch auf einem anderen PC mit anderen Delphi-Einstellungen genauso übersetzt werden wie mit Ihren aktuellen Einstellungen, dann können Sie diese Möglichkeit benutzen.

Ist die Bereichsüberprüfung eingeschaltet, dann wird zur Laufzeit ein Fehler erzeugt, wenn das Ergebnis größer als 100 ist. Sie erhalten dann eine Fehlermeldung wie in Abbildung 3.2.

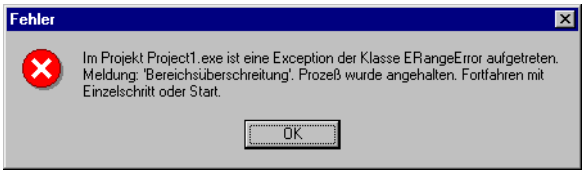

Abbildung 3.2: Die Fehlermeldung während der Arbeit mit Delphi

In diesem Fall reagiert Delphi auf den Fehler und stoppt die Ausführung Ihres Programms. Diese Meldung erscheint nur, wenn Sie das Programm aus der Entwicklungsumgebung starten. Delphi gibt Ihnen damit die Möglichkeit, mit Hilfe des Debuggers den aufgetretenen Fehler zu analysieren.

Um fortzufahren, klicken Sie einfach auf **Start**. Jetzt bekommen Sie eine weitere Fehlermeldung (siehe Abbildung 3.3). Diese Meldung erhalten Sie auch, wenn Sie das Programm in der Windows-Umgebung (also nicht in der Delphi-IDE) starten.

Abbildung 3.3: Die Fehlermeldung zur Laufzeit außerhalb der IDE

Würden Sie die Zeile 43 wirklich zum Kommentar machen (// {$R+}), dann erhalten Sie alle Ergebnisse ohne Einschränkung und Fehlermeldungen (vorausgesetzt, Sie ändern nicht die Standardeinstellung

der Compiler-Optionen und schalten die Bereichsüberprüfung dort an).

Weitere Informationen zu Compiler-Direktiven liefert Ihnen, wie bereits erwähnt, die Online-Hilfe unter dem Stichwort „Compiler-Direktiven verwenden“.

Sie finden dieses Programm auf der CD-ROM im Verzeichnis *Kapitel 3\Unterbereich*.

Eine weitere Möglichkeit, den gültigen Bereich zu überprüfen, werden Sie in Kapitel 4 kennen lernen, wenn es um die Ausnahmebehandlung geht. Diese Art der Fehlerprüfung bietet Ihnen die Gelegenheit zur individuellen Fehlerbehandlung, wie Sie dann sehen werden.

3.3 Laufwerke, Dateien und Verzeichnisse

Sie kennen bereits die Standarddialoge zum Laden und Speichern von Dateien. Oft braucht man aber eine ähnliche Funktionalität oder einen Teil dieser Funktionalität in eigenen Dialogen. Diese Übung vermittelt Ihnen die grundlegenden Kenntnisse, um eigene Dialoge oder Programme zu erstellen, mit denen Sie Laufwerke und Verzeichnisse wechseln sowie Dateien verarbeiten können. Die Übung ist dank Delphi sehr kurz, denn die eigentliche Arbeit übernimmt hier die Programmiersprache für Sie. Die Aufgabe reduziert sich darauf, die richtigen Einstellungen im Objektinspektor vorzunehmen – Sie müssen nicht eine Zeile Programmtext schreiben. Das Zusammenspiel der Laufwerks-, Verzeichnis- und Dateiliste wird gleich zur Entwicklungszeit eingestellt.

Wir verwenden hier die Elemente für die Dateidialoge im Windows 3.1-Stil. Für die Dateiauswahl im Explorer-Stil kann die *TListView*-Komponente verwendet werden.

Sie finden die verwendeten Komponenten im Register WIN 3.1 der Komponentenpalette (siehe Abbildung 3.4).

Abbildung 3.4: Die Position der Komponenten auf der Palette

Alle vier Komponenten sind visuelle Komponenten, das heißt sie zeigen schon zur Entwicklungszeit ihr späteres Erscheinungsbild. Die Abbildung 3.5 zeigt das Programm, mit dem Sie Ihre Festplatte zur Laufzeit inspizieren können.

Abbildung 3.5: Das Beispielformular zur Laufzeit

Die erforderlichen Einstellungen sind in aufgeführt. Setzen Sie erst alle Komponenten auf das Formular, bevor Sie die Einstellungen vornehmen.

Komponente	Eigenschaft	Neuer Wert
DriveComboBox	DirList	DirectoryListBox
DirectoryListBox	FileList	FileListBox
DirectoryListBox	DirLabel	Label1
FileListBox	-	-
FilterListBox	FileList	FileListBox
FilterListBox	Filter	Siehe

Tabelle 3.2: Die Einstellungen der Komponenten

Es ist sicherlich nicht erforderlich, näher auf die Komponenten einzugehen. Jede Komponente stellt sich so dar, dass Sie sofort ihren Zweck erkennen können. Dadurch, dass Sie die logische Verbindung zwischen *DriveComboBox*, *DirectoryListBox*, *FileListBox* und *FilterListBox* über die Eigenschaften *DirList* und *FileList* schon zur Entwicklungszeit hergestellt haben, brauchen Sie keine einzige Zeile Quelltext zu schreiben. Es funktioniert wie von Geisterhand geführt – visuelle Programmierung in Reinkultur!

Auf der CD-ROM im Verzeichnis *Aha-Programme\Sonstiges\Viewer* finden Sie ein Programm, das diese Komponenten benutzt.

3.4 Ein Korb voller Äpfel – die Array-Typen

Sie erinnern sich sicherlich noch an das Beispiel vom CD-Rack in Kapitel 1, das Ihnen eine Vorstellung von Arrays (Datenfeldern) vermitteln sollte. Die Arrays sammeln *mehrere* Variablen *desselben* Typs. Stellen Sie sich vor, Sie brauchen aus irgendeinem Grund für jeden Buchstaben des Alphabets eine eigene Variable (beispielsweise zur einfachen Kodierung von Text). Anstatt jetzt 26 einzelne Variablen vom Typ *Char* anzulegen, werden diese in einem Array zusammengefasst. Abbildung 3.6 zeigt ein solches Array, in dem jedem Index ein Buchstabe zugeordnet ist. Der Wert mit dem Index 1 enthält den Buchstaben A, der Wert mit dem Index 2 den Buchstaben B usw.

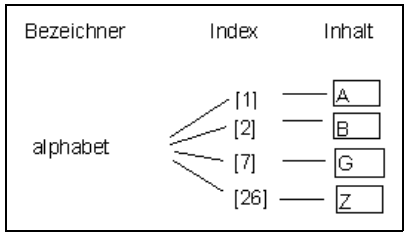

Abbildung 3.6: Ein Array und sein Inhalt

So sieht die allgemeine Darstellung eines Array-Typs aus:

```
type
  TBezeichner = array[Unterbereichstyp] of Datentyp;
```

Das könnte dann für das Alphabet-Beispiel so aussehen:

```
type
  // 26 Zeichen sollen aufgenommen werden
  TAlphabet = array[1..26] of Char;
```

Hier wird ein Array-Typ vom Typ *Char* mit einer Liste von 26 Elementen deklariert. Der Zugriff auf die einzelnen Elemente erfolgt über ihre Nummer, den Index.

Achten Sie auf den Bereichsbeginn! In diesem Beispiel beginnt der Bereich bei eins – das hat sich in Pascal so eingebürgert. Andererseits beginnen Arrays in Windows normalerweise mit null (der Standard für die Programmiersprache C). Sie werden daher beide Schreibweisen antreffen – achten Sie immer darauf, wenn Sie fremde Programme lesen, und benutzen Sie möglichst einen einheitlichen Stil, sonst können sich schwer lokalisierbare Fehler in Ihre Programme einschleichen!

Sie finden ein Beispiel für einen solchen Fehler im Verzeichnis *Kapitel 3\Arrays\Arraygrenzen*. In den beiden Unterverzeichnissen finden Sie das gleiche Programm, bei dem aber der Array-Bereich einmal mit null und einmal mit eins beginnt. Da dieser Unterschied nicht beachtet wurde, ist eines der Programme fehlerhaft, und zwar so, dass sehr schwer zu erkennen ist, woher der Fehler eigentlich kommt.

Alle Arrays werden also im Wesentlichen durch zwei Eigenschaften charakterisiert:

- einen bestimmten Datentyp (beispielsweise *Integer*)
- die Größe (Anzahl der Elemente)

Größe des Arrays

Beim Anlegen von Arrays müssen Sie also bereits entscheiden, wie viele Einträge das Array maximal aufnehmen kann. Das ist notwendig, damit der Compiler den Speicherplatz bereitstellen kann.

Es ist auch möglich, die Größe der Arrays erst zur Laufzeit zu bestimmen und sie in der Größe zu verändern (so genannte *dynamische Arrays*); dazu müssen Sie sich aber zunächst mit Pointern (Zeigern, Adressvariablen) beschäftigen; Sie finden weitere Informationen dazu in Kapitel 5.

Um den Typ (*TAlphabet*) zu verwenden, müssen Sie eine Variable dieses Typs erstellen:

```
var
  alphabet: TAlphabet;
```

Index

Der Zugriff auf die Variablen des Arrays erfolgt dann über die Angabe des Array-Namens (Array-Variable) und des Zahlenwertes (Index), der die Position beschreibt:

```
  alphabet[4] := 'D'; // Initialisieren
  Label1.Caption := alphabet[4]; // Den Buchstaben D anzeigen
```

Es ist natürlich nicht praktisch, jedes einzelne Element mit der entsprechenden Nummer aufzurufen – weder bei der Initialisierung noch beim Zugriff. Deshalb wird an Stelle der Ziffer eine Variable, eine so genannte Indexvariable, verwendet, die die entsprechenden Zahlenwerte annimmt:

```
var
  alphabet: TAlphabet;
  index: Integer; // Laufvariable
```

Initialisierung von Arrays

Die Initialisierung eines Arrays erfolgt normalerweise in einer *for*-Schleife. Es ist eine gute Idee, Arrays und andere Variablen mit einem