Object Pascal Sprachreferenz

Borland®
Delphi™ 5
für Windows 95, Windows 98 & Windows NT

Inprise GmbH, Robert-Bosch-Straße 11, D-63225 Langen

Deutsche Ausgabe © 1999 Inprise GmbH, Robert-Bosch-Straße 11, D-63225 Langen, Telefon 06103/979-0, Fax 06103/979-290

Update/Übertragung ins Deutsche: Krieger, Zander & Partner GmbH, München

Satz: Krieger, Zander & Partner GmbH, München

Hauptsitz: 100 Enterprise Way, P.O. Box 660001, Scotts Valley, CA 95067-0001, +1-(408)431-1000

Niederlassungen in: Australien, Deutschland, Frankreich, Großbritannien, Hong Kong, Japan, Kanada, Lateinamerika, Mexiko, den Niederlanden und Taiwan

HDA1350GE21002

Inhalt

Kapitel 5
Datentypen, Variablen und Konstanten 5-1

Kapitel 6
Prozeduren und Funktionen 6-1

Kapitel 7
Klassen und Objekte 7-1

Tabellen

Kapitel

1

Einführung

Dieses Handbuch enthält Informationen zu der in Delphi verwendeten Programmiersprache Object Pascal.

Inhalt dieses Handbuchs

Die ersten sieben Kapitel beschreiben die meisten der in der normalen Programmierung verwendeten Sprachelemente. Kapitel 8 enthält einen Überblick über die Standardroutinen für E/A-Operationen und String-Bearbeitung.

Die nächsten beiden Kapitel enthalten Informationen über Spracherweiterungen, DLLs und Delphi-Packages (Kapitel 9) und über Objektschnittstellen und COM (Kapitel 10). In den letzten drei Kapiteln werden Themen für fortgeschrittene Programmierer behandelt: Speicherverwaltung (Kapitel 11), Programmsteuerung (Kapitel 12) und Verwendung von Assembler-Routinen in Pascal-Programmen (Kapitel 13).

Delphi und Object Pascal

Die meisten Delphi-Entwickler schreiben und compilieren ihre Programme in der integrierten Entwicklungsumgebung (IDE). Hier kümmert sich Delphi um alle Details, die beim Einrichten von Projekten und Quelltextdateien von Bedeutung sind (z.B. Verwalten der Abhängigkeiten zwischen den Units). Delphi stellt dabei spezielle Regeln bezüglich der Programmorganisation auf, die aber nicht zur Sprachdefinition von Object Pascal gehören. So werden beispielsweise bestimmte Namenskonventionen für Dateien und Programme verwendet, an die Sie aber nicht gebunden sind, wenn Sie Programme außerhalb der IDE schreiben und in der Befehlszeile compilieren.

Im allgemeinen geht dieses Handbuch davon aus, daß Sie in der Delphi-IDE arbeiten und Anwendungen mit Hilfe der VCL (Visual Component Library) erstellen. In man-

chen Abschnitten wird jedoch zwischen speziellen Delphi-Regeln und der normalen Programmierung mit Object Pascal unterschieden.

Typografische Konventionen

Bezeichner (Konstanten, Variablen, Typen, Eigenschaften, Prozeduren, Funktionen, Programme, Units, Bibliotheken und Packages) sind im Text *kursiv* gesetzt. Operatoren, reservierte Wörter und Direktiven der Sprache Object Pascal werden **fett** geschrieben. Quelltextbeispiele und Text, der literal eingegeben werden muß (in eine Datei oder in der Befehlszeile), erkennen Sie an einer `Schrift mit fester Zeichenbreite`.

Reservierte Wörter und Direktiven werden auch in Programmlistings **fett** geschrieben:

```
function Calculate(X, Y: Integer): Integer;
begin
   ⋮
end;
```

Auch im Quelltexteditor werden Direktiven und reservierte Wörter fett angezeigt, wenn die Syntaxhervorhebung aktiviert ist.

In manchen Quelltextbeispielen (siehe oben) werden Fortsetzungszeichen (... oder ⋮) verwendet. Sie stehen für weitere Programmzeilen, die in einer richtigen Quelltextdatei an dieser Stelle vorhanden wären. Diese Zeichen dürfen nicht in den Quelltext übernommen werden.

Kursive Angaben in Syntaxbeschreibungen sind Platzhalter, die im echten Quelltext durch syntaktisch gültige Konstrukte ersetzt werden müssen. Die Syntax der Funktion im vorhergehenden Beispiel könnte beispielsweise folgendermaßen angegeben werden:

`function` *Funktionsname(Argumente)*: *Rückgabetyp;*

In Syntaxangaben werden auch Fortsetzungszeichen (...) und Subskripte verwendet:

`function` *Funktionsname(Arg1, ..., Argn)*: *Rückgabetyp;*

Weitere Informationsquellen

Die Online-Hilfe von Delphi enthält Informationen über die IDE und die Benutzeroberfläche. Außerdem finden Sie hier das aktuelle VCL-Referenzmaterial. Ausführliche Informationen zu vielen Programmierthemen (z.B. Entwickeln von Datenbankanwendungen) finden Sie im *Entwicklerhandbuch*. Einen Überblick über die Delphi-Dokumentation finden Sie in der *Einführung*.

Software-Registrierung und technische Unterstützung

Borland bietet einzelnen Entwicklern, EDV-Fachleuten und Firmen vielfältige Unterstützung an. Damit Sie diese Möglichkeiten nutzen können, senden Sie das Registrierungsformular mit den für Sie zutreffenden Angaben zurück. Weitere Informationen zur technischen Unterstützung und zu weiteren Borland-Diensten erhalten Sie bei Ihrem Händler oder auf der Borland-Homepage unter http://www.inprise.com/.

Teil

I

Beschreibung der Sprachen

Die Kapitel in Teil I beschreiben die grundlegenden Sprachelemente, die für die meisten Programmieraufgaben erforderlich sind.

- Kapitel 2, »Übersicht«
- Kapitel 3, »Programme und Units«
- Kapitel 4, »Syntaktische Elemente«
- Kapitel 5, »Datentypen, Variablen und Konstanten«
- Kapitel 6, »Prozeduren und Funktionen«
- Kapitel 7, »Klassen und Objekte«
- Kapitel 8, »Standardroutinen und E/A«

Kapitel

2

Übersicht

Object Pascal ist eine höhere Compiler-Sprache mit strenger Typisierung, die eine strukturierte und objektorientierte Programmierung ermöglicht. Zu ihren Vorzügen gehören einfache Lesbarkeit, schnelle Compilierung und modulare Programmierung durch Verteilen des Quelltextes auf mehrere Unit-Dateien.

Object Pascal verfügt über spezielle Leistungsmerkmale, die das Komponentenmodell und die visuelle Entwicklungsumgebung von Delphi unterstützen. Die Beschreibungen und Beispiele dieser Sprachreferenz gehen davon aus, daß Sie Delphi-Anwendungen mit Object Pascal erstellen.

Programmorganisation

Programme werden normalerweise auf mehrere Quelltextmodule (sogenannte Units) verteilt. Jedes Programm beginnt mit einer Kopfzeile, in der sein Name angegeben wird. Darauf folgt eine optionale **uses**-Klausel und ein Block von Deklarationen und Anweisungen. Mit der **uses**-Klausel geben Sie die Units an, die in das Programm eingebunden werden. Units können von mehreren Programmen gemeinsam benutzt werden und verfügen häufig über eigene ***uses***-Klauseln.

Die **uses**-Klausel liefert dem Compiler Informationen über die Modulabhängigkeiten. Da diese Abhängigkeiten in den Modulen selbst gespeichert werden, benötigen Object-Pascal-Programme keine Make-Dateien, Header-Dateien oder *#include*-Präprozessoranweisungen (die Projektverwaltung von Delphi generiert zwar eine Make-Datei, wenn Sie ein Projekt in die IDE laden, speichert diese aber nur für Projektgruppen mit mehreren Projekten).

Weitere Informationen über Programmstrukturen und Modulabhängigkeiten finden Sie in Kapitel 3, »Programme und Units«.

Pascal-Quelltextdateien

Der Compiler erwartet Pascal-Quelltexte in einem der folgenden Dateitypen:

- *Unit*-Quelltextdateien mit der Namenserweiterung .PAS
- *Projektdateien* mit der Namenserweiterung .DPR
- *Package*-Quelltextdateien mit der Namenserweiterung .DPK

In den Unit-Quelltextdateien befindet sich der größte Teil des Programmcodes. Jede Delphi-Anwendung besteht aus einer Projektdatei und mehreren Unit-Dateien. Die Projektdatei entspricht dem Hauptprogramm im herkömmlichen Pascal und organisiert die Units der Anwendung. Delphi verwaltet für jede Anwendung automatisch eine Projektdatei.

Wenn Sie ein Programm in der Befehlszeile compilieren, können Sie den gesamten Quelltext in Unit-Dateien (PAS) aufnehmen. Erstellen Sie die Anwendung jedoch mit Hilfe der Delphi-IDE, wird eine Projektdatei (DPR) benötigt.

Package-Quelltextdateien ähneln Projektdateien, werden aber für spezielle DLLs (*Packages*) verwendet. Weitere Informationen zu Packages finden Sie in Kapitel 9, »DLLs und Packages«.

Weitere Anwendungsdateien

Neben den Quelltextmodulen werden in Delphi-Anwendungen auch die folgenden Dateitypen verwendet, die keinen Pascal-Code enthalten und ebenfalls automatisch verwaltet werden:

- *Formulardateien* mit der Namenserweiterung .DFM
- *Ressourcendateien* mit der Namenserweiterung .RES
- *Projektoptionsdateien* mit der Namenserweiterung .DOF
- *Desktop-Konfigurationsdateien* mit der Namenserweiterung .DSK

Eine Formulardatei (DFM) ist eine Windows-Ressourcendatei, die Bitmaps, Strings usw. enthält. Sie ist die binäre Entsprechung eines Delphi-Formulars, das normalerweise einem Fenster oder Dialogfeld in einer Windows-Anwendung entspricht. Formulardateien können in der IDE auch als Text angezeigt und bearbeitet werden. Normalerweise werden dazu jedoch die visuellen Tools verwendet. Jedes Delphi-Projekt hat mindestens ein Formular. Zu jedem Formular gehört eine Unit-Datei (PAS), die standardmäßig denselben Namen wie die Formulardatei erhält.

Zusätzlich zu den Formulardateien verwendet jedes Delphi-Projekt eine Standard-Ressourcendatei (RES), in der die Bitmap-Grafik des Anwendungssymbols gespeichert wird. Diese Datei erhält automatisch denselben Namen wie die Projektdatei (DPR). Im Dialogfeld *Projektoptionen* können Sie ein anderes Symbol festlegen.

Die Projektoptionsdatei (DOF) enthält Compiler- und Linker-Einstellungen, Suchverzeichnisse, Versionsinformationen usw. Diese Datei erhält automatisch denselben

Namen wie die Projektdatei (DPR). Die Einstellungen in dieser Datei können im Dialogfeld *Projektoptionen* festgelegt werden.

Für jedes Projekt gibt es eine Desktop-Konfigurationsdatei (DSK) mit Informationen über die Anordnung der Fenster und andere Einstellungen der Delphi-IDE. Diese Datei erhält automatisch denselben Namen wie die Projektdatei (DPR). Die meisten Einstellungen in dieser Datei können im Dialogfeld *Umgebungsoptionen* festgelegt werden.

Vom Compiler generierte Dateien

Wenn Sie eine Anwendung oder eine DLL zum ersten Mal erstellen, generiert der Compiler für jede im Projekt verwendete Unit eine Objektdatei (DCU = Delphi Compiled Unit). Die Objektdateien werden dann zur ausführbaren Datei (EXE) oder zur Bibliothek (DLL) gelinkt. Bei einem Package wird für jede Unit eine DCU-Datei und anschließend eine DCP- und BPL-Datei erstellt (weitere Informationen zu DLLs und Packages finden Sie in Kapitel 9, »DLLs und Packages«). Wenn die Befehlszeilenoption **–GD** angegeben wird, generiert der Linker eine Map- und eine DRC-Datei. Die DRC-Datei enthält String-Ressourcen und kann in die Ressourcendatei compiliert werden.

Beim erneuten Erstellen eines Projekts wird eine Unit nur dann compiliert, wenn ihre Quelltextdatei (PAS) seit dem letzten Compilieren geändert wurde, die zugehörige DCU-Datei nicht gefunden werden kann oder der Compiler explizit dazu angewiesen wird. Wenn der Compiler auf die entsprechende Objektdatei zugreifen kann, braucht für eine Unit keine Quelltextdatei vorhanden zu sein.

Beispielprogramme

Die folgenden Beispielprogramme zeigen die Grundzüge der Programmierung mit Object Pascal und Delphi. Die ersten beiden Beispiele sind keine Delphi-Anwendungen, können aber in der Befehlszeile compiliert werden.

Eine einfache Konsolenanwendung

Das folgende Beispiel zeigt eine einfache Konsolenanwendung, die Sie in der Befehlszeile compilieren und ausführen können.

```
program Greeting;

{$APPTYPE CONSOLE}

var MyMessage: string;

begin
   MyMessage := 'Hallo Welt!';
   Writeln(MyMessage);
end.
```

Durch die erste Anweisung wird das Programm *Greeting* deklariert. Die Direktive **{$APPTYPE CONSOLE}** teilt dem Compiler mit, daß es sich um eine Konsolenanwendung handelt, die in der Befehlszeile ausgeführt wird. In der nächsten Zeile wird die String-Variable *MyMessage* deklariert (in Object Pascal gibt es echte String-Datentypen). Im Programmblock wird der Variablen *MyMessage* der String `'Hallo Welt!'` zugewiesen. Anschließend wird die Variable mit der Standardprozedur *Writeln* an die Standardausgabe gesendet (*Writeln* ist implizit in der Unit *System* definiert, die automatisch in jede Anwendung eingebunden wird).

Wenn Sie Delphi in Ihrem System installiert haben und der Suchpfad das Verzeichnis DELPHI\BIN enthält (in dem sich DCC32.EXE und DCC32.CFG befinden), können Sie nun das Programm in eine Datei mit dem Namen GREETING.PAS oder GREETING.DPR eingeben und mit folgender Anweisung in der Befehlszeile compilieren:

```
DCC32 GREETING
```

Die ausführbare Datei (GREETING.EXE) gibt die Meldung `'Hallo Welt!'` auf dem Bildschirm aus.

Neben seiner Einfachheit unterscheidet sich dieses Beispiel noch in anderen wichtigen Punkten von den Anwendungen, die Sie normalerweise mit Delphi erstellen. Zum einen handelt es sich bei dem Beispiel um eine Konsolenanwendung. Mit Delphi werden hauptsächlich Windows-Anwendungen mit grafischen Benutzeroberflächen entwickelt, in denen natürlich keine *Writeln*-Aufrufe benötigt werden. Außerdem befindet sich das gesamte Beispielprogramm in einer einzigen Datei. Bei einer Delphi-Anwendung steht der Programmkopf (im Beispiel die erste Zeile) in einer separaten Projektdatei, die mit Ausnahme einiger Methodenaufrufe keinerlei Programmlogik enthält.

Ein komplexeres Beispiel

Das Programm im nächsten Beispiel ist auf zwei Dateien verteilt, eine *Projektdatei* und eine *Unit-Datei*. Die Projektdatei, die Sie unter dem Namen GREETING.DPR speichern können, hat folgenden Inhalt:

```
program Greeting;

{$APPTYPE CONSOLE}

uses Unit1;

begin
   PrintMessage('Hallo Welt!');
end.
```

In der ersten Zeile wird das Programm *Greeting* deklariert, das wiederum eine Konsolenanwendung ist. Danach wird *Unit1* durch die Klausel `uses Unit1` in das Programm eingebunden. Zum Schluß wird die Prozedur *PrintMessage* mit dem auszugebenden String aufgerufen. Woher kommt aber die Prozedur *PrintMessage*? Sie ist in *Unit1* definiert. Diese Unit hat folgenden Inhalt (speichern Sie sie unter dem Namen UNIT1.PAS):

```
unit Unit1;

interface

procedure PrintMessage(msg: string);

implementation

procedure PrintMessage(msg: string);
begin
   Writeln(msg);
end;

end.
```

Die Prozedur *PrintMessage* sendet den als Parameter übergebenen String an die Standardausgabe. In Pascal heißen Routinen ohne Rückgabewert *Prozeduren*. Routinen, die einen Wert zurückgeben, heißen *Funktionen*. Beachten Sie, daß *PrintMessage* zweimal deklariert ist. Die erste Deklaration im **interface**-Abschnitt macht *PrintMessage* für andere Module verfügbar (z.B. für das Programm *Greeting*), die *Unit1* einbinden. Durch die zweite Deklaration (im **implementation**-Abschnitt) erfolgt die eigentliche Definition der Prozedur.

Sie können *Greeting* nun in der Befehlszeile compilieren:

```
DCC32 GREETING
```

Unit1 braucht in der Befehlszeile nicht angegeben zu werden. Der Compiler liest beim Bearbeiten von GREETING.DPR automatisch die Unit-Dateien ein, von denen *Greeting* abhängig ist. Die neue ausführbare Datei (GREETING.EXE) führt dieselbe Operation durch wie unser erstes Beispiel, d.h. sie gibt die Meldung "Hallo Welt!" aus.

Eine Windows-Anwendung

Als nächstes erstellen wir mit Hilfe der VCL (Visual Component Library) eine richtige Windows-Anwendung. Im Programm werden die automatisch generierten Formular- und Ressourcendateien verwendet. Es kann daher nicht allein aus dem Quelltext compiliert werden. Das Beispiel zeigt einige wichtige Merkmale von Object Pascal. Neben mehreren Units werden auch Klassen und Objekte verwendet (siehe Kapitel 7, »Klassen und Objekte«).

Die Anwendung besteht aus einer Projektdatei und zwei Unit-Dateien. Betrachten wir zuerst die Projektdatei:

```
program Greeting;  { Kommentare stehen in geschweiften Klammern }

uses
   Forms,
   Unit1  { Die Unit für Form1 },
   Unit2  { Die Unit für Form2 };

{$R *.RES}  { Diese Anweisung bindet die Ressourcendatei des Projekts ein. }

begin
```

```
   { Aufrufe von Methoden des Application-Objekts }
   Application.Initialize;
   Application.CreateForm(TForm1, Form1);
   Application.CreateForm(TForm2, Form2);
   Application.Run;
end.
```

Unser Programm heißt auch diesmal *Greeting*. In der **uses**-Klausel werden drei Units eingebunden. *Forms* ist eine Unit der VCL, *Unit1* gehört zum Hauptformular der Anwendung (*Form1*) und *Unit2* zu einem weiteren Formular (*Form2*).

Innerhalb des **begin...end**-Blocks finden mehrere Aufrufe des Objekts *Application* statt, das eine Instanz der in *Forms* deklarierten Klasse *TApplication* ist (für jedes Delphi-Projekt wird automatisch ein *Application*-Objekt erstellt). Die Methode *CreateForm* wird zweimal aufgerufen. Durch den ersten Aufruf wird *Form1* erstellt, eine Instanz der in *Unit1* deklarierten Klasse *TForm1*. Der zweite Aufruf erstellt *Form2*, eine Instanz der in *Unit2* deklarierten Klasse *TForm2*.

Unit1 hat folgenden Inhalt:

```
unit Unit1;

interface

uses  { Die folgenden Units gehören zur Komponentenbibliothek (VCL) von Delphi. }
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

type
   TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
   end;

var
   Form1: TForm1;

implementation

uses Unit2;  { Hier ist Form2 definiert. }

{$R *.DFM}  { Diese Direktive bindet die Formulardatei von Unit1 ein. }

procedure TForm1.Button1Click(Sender: TObject);
begin
   Form1.Hide;
   Form2.Show;
end;

end.
```

Nach dem Einbinden der Units wird der Typ *TForm1* (abgeleitet von *TForm*) und eine Instanz dieser Klasse (*Form1*) erstellt. *TForm1* enthält eine Schaltfläche (die *TButton*-Instanz *Button1*) und die Prozedur *TForm1.Button1Click*, die aufgerufen wird, wenn der Benutzer zur Laufzeit auf *Button1* klickt. In *TForm1.Button1Click* werden zwei

Operationen durchgeführt. Die erste Anweisung verbirgt *Form1*, die zweite Anweisung zeigt *Form2* an. Das Objekt *Form2* ist in *Unit2* definiert:

```
unit Unit2;

interface

uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
   StdCtrls;

type
   TForm2 = class(TForm)
    Label1: TLabel;
    CancelButton: TButton;
    procedure CancelButtonClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
   end;

var
   Form2: TForm2;

implementation

uses Unit1;

{$R *.DFM}

procedure TForm2.CancelButtonClick(Sender: TObject);
begin
   Form2.Close;
end;

procedure TForm2.FormClose(Sender: TObject; var Action: TCloseAction);
begin
   Form1.Show;
end;

end.
```

Nach dem Einbinden der Units wird der Typ *TForm2* und eine Instanz dieser Klasse (*Form2*) erstellt. *TForm2* enthält eine Schaltfläche (die *TButton*-Instanz *CancelButton*) und eine Beschriftung (die *TLabel*-Instanz *Label1*). Obwohl es aus dem Quelltext nicht ersichtlich ist, zeigt *Label1* unseren Lieblingsgruß `'Hallo Welt!'` an. Die Komponente ist in der Formulardatei von *Form2*, UNIT2.DFM, definiert.

In *Unit2* sind auch zwei Prozeduren definiert. In *TForm2.CancelButtonClick* wird das Formular *Form2* geschlossen. Diese Prozedur wird aufgerufen, wenn der Benutzer zur Laufzeit auf die Schaltfläche *CancelButton* klickt. *TForm2.FormClose* wird beim Schließen von *Form2* aufgerufen und öffnet wieder *Form1*. Solche Prozeduren (wie auch *TForm1.Button1Click* in *Unit1*) nennt man *Ereignisbehandlungsroutinen*, weil sie auf Ereignisse reagieren, die zur Laufzeit der Anwendung eintreten. Sie werden durch die Formulardateien (DFM) für *Form1* und *Form2* bestimmten Ereignissen zugewiesen.

Nach dem Starten der Anwendung *Greeting* wird lediglich *Form1* angezeigt (standardmäßig ist nur das erste in der Projektdatei erstellte Formular, das *Hauptformular*, zur Laufzeit sichtbar). Klickt der Benutzer auf die Schaltfläche in *Form1*, wird dieses Formular vom Bildschirm entfernt und durch *Form2* ersetzt, wo die Meldung `'Hallo Welt!'` angezeigt wird. Wenn der Benutzer *Form2* schließt (durch Klicken auf *Cancel-Button* oder auf das Schließfeld), wird wieder *Form1* angezeigt.

Kapitel

3

Programme und Units

Ein Programm besteht aus einzelnen Quelltextmodulen, den sogenannten *Units*. Jede Unit wird in einer eigenen Datei gespeichert und separat compiliert. Die compilierten Units (DCU-Dateien) werden dann zu einer Anwendung gelinkt. *Units* bieten folgende Möglichkeiten:

- Aufteilung großer Programme in Module, die separat bearbeitet werden können.
- Erstellung von Bibliotheken, die von mehreren Programmen genutzt werden können.
- Weitergabe von Bibliotheken an andere Entwickler, ohne den Quelltext verfügbar zu machen.

Bei der herkömmlichen Pascal-Programmierung wird der gesamte Quelltext - einschließlich des Hauptprogramms - in PAS-Dateien gespeichert. Delphi verwendet eine *Projektdatei* (DPR), in der das Hauptprogramm gespeichert wird. Der weitere Quelltext befindet sich zum größten Teil in sogenannten *Unit*-Dateien (PAS). Jede Anwendung bzw. jedes *Projekt* besteht aus einer Projektdatei und einer oder mehreren Unit-Dateien. Es müssen keine Units in ein Projekt aufgenommen werden, alle Programme verwenden aber automatisch die Unit *System*. Damit ein Projekt vollständig compiliert werden kann, benötigt der Compiler entweder die entsprechenden Quelltextdateien oder eine bereits compilierte DCU-Datei für jede Unit.

Programmstruktur und -syntax

Ein Programm enthält folgende Komponenten:

- Programmkopf
- **uses**-Klausel (optional)
- Deklarations- und Anweisungsblock

Der Programmkopf enthält den Namen des Programms. Die **uses**-Klausel gibt die vom Programm verwendeten Units an. Der nächste Block enthält die Deklarationen und Anweisungen, die beim Starten des Programms ausgeführt werden. In der Delphi-IDE wird vorausgesetzt, daß sich diese Elemente in einer einzelnen Projektdatei (DPR) befinden.

Das folgende Beispiel zeigt die Projektdatei für ein Programm namens *Editor:*

```
1     program Editor;
2
3     uses
4       Forms,
5       REAbout in 'REABOUT.PAS' {AboutBox},
6       REMain in 'REMain.pas' {MainForm};
7
8     {$R *.RES}
9
10    begin
11      Application.Title := 'Text Editor';
12      Application.CreateForm(TMainForm, MainForm);
13      Application.Run;
14    end.
```

Zeile 1 enthält den Programmkopf. Die **uses**-Klausel erstreckt sich über die Zeilen 3 bis 6. Zeile 8 enthält eine Compiler-Direktive, die bewirkt, daß die Ressourcendatei des Projekts zum Programm gelinkt wird. In den Zeilen 10 bis 14 befindet sich der Anweisungsblock, der beim Starten des Programms ausgeführt wird. Die Projektdatei wird wie alle Quelltextdateien mit einem Punkt beendet.

Das Beispiel ist durchaus realistisch. Projektdateien sind normalerweise kurz, da die Programmlogik üblicherweise in Unit-Dateien erstellt wird. Projektdateien werden von Delphi generiert und verwaltet und nur in seltenen Fällen manuell bearbeitet.

Der Programmkopf

Der Programmkopf gibt den Namen des Programms an. Er besteht aus dem reservierten Wort **program**, einem nachgestellten gültigen Bezeichner und einem abschließenden Semikolon. In Delphi-Anwendungen muß der Bezeichner dem Namen der Projektdatei entsprechen. Im obigen Beispiel müßten Sie die Projektdatei also EDITOR.DPR nennen, da der Name des Programms *Editor lautet.*

In Standard-Pascal kann der Programmkopf hinter dem Programmnamen auch Parameter angeben:

```
program Calc(input, output);
```

Der Delphi-Compiler ignoriert diese Parameter.

Die uses-Klausel

Die **uses**-Klausel gibt alle Units an, die in das Programm aufgenommen werden. Diese Units können eigene **uses**-Klauseln enthalten. Weitere Informationen zur **uses**-Klausel finden Sie im Abschnitt »Unit-Referenzen und die uses-Klausel« auf Seite 3-6.

Der Block

Ein Block enthält eine einfache oder strukturierte Anweisung, die beim Starten des Programms ausgeführt wird. In den meisten Delphi-Programmen besteht ein Block aus einer zusammengesetzten Anweisung zwischen den reservierten Wörtern **begin** und **end**. Die einzelnen Anweisungen sind Aufrufe von Methoden des Anwendungsobjekts des Projekts. (Jedes Delphi-Projekt verfügt über eine *Application*-Variable, die eine Instanz von *TApplication*, *TWebApplication* oder *TServiceApplication* ist.) Ein Block kann außerdem Deklarationen von Konstanten, Typen, Variablen, Prozeduren und Funktionen enthalten. Die Deklarationen müssen im Block vor den Anweisungen stehen.

Unit-Struktur und -Syntax

Eine Unit besteht aus Typen (einschließlich Klassen), Konstanten, Variablen und Routinen (Prozeduren und Funktionen). Jede Unit wird in einer separaten Unit-Datei (PAS) definiert.

Eine Unit-Datei beginnt mit dem Unit-Kopf und enthält dann die Abschnitte **interface, implementation, initialization und finalization.** Die Abschnitte **initialization** und **finalization** sind optional. Die Struktur einer Unit-Datei sieht also folgendermaßen aus:

```
unit Unit1;

interface

uses  { Liste der verwendeten Units }

   { interface-Abschnitt }

implementation

uses  { Liste der verwendeten Units }

   { implementation-Abschnitte }

initialization
   { initialization-Abschnitt }
```

```
finalization
   { finalization-Abschnitt }

end.
```

Eine Unit muß mit dem Wort **end** und einem Punkt abgeschlossen werden.

Der Unit-Kopf

Der Unit-Kopf gibt den Namen der Unit an. Er besteht aus dem reservierten Wort **unit**, einem gültigen Bezeichner und einem abschließenden Semikolon. In Delphi-Anwendungen muß der Bezeichner dem Namen der Unit-Datei entsprechen. Ein Beispiel:

```
unit MainForm;
```

Dieser Unit-Kopf kann in einer Quelltextdatei namens MAINFORM.PAS verwendet werden. Die Datei mit der compilierten Unit trägt dann den Namen MAINFORM.DCU.

Unit-Namen müssen in einem Projekt eindeutig sein. Auch wenn die Unit-Dateien in unterschiedlichen Verzeichnissen gespeichert werden, dürfen in einem Programm keine Units mit identischen Namen verwendet werden.

Der interface-Abschnitt

Der **interface-**Abschnitt einer Unit beginnt mit dem reservierten Wort **interface**. Er endet mit dem Beginn des **implementation**-Abschnitts. Der **interface**-Abschnitt deklariert Konstanten, Typen, Variablen, Prozeduren und Funktionen, die für *Clients* verfügbar sind. Clients sind andere Units oder Programme, die diese Unit über die **uses**-Klausel einbinden. Solche Entitäten werden als *öffentlich* bezeichnet, da der Client auf sie wie auf Entitäten zugreifen kann, die im Client selbst deklariert sind.

Die **interface**-Deklaration einer Prozedur oder Funktion enthält nur den Kopf der Routine. Der Block der Prozedur bzw. Funktion wird dagegen im **implementation**-Abschnitt definiert. Prozedur- und Funktionsdeklarationen im **interface**-Abschnitt entsprechen also **forward**-Deklarationen, obwohl die Direktive **forward** nicht verwendet wird.

Die **interface**-Deklaration einer Klasse muß die Deklarationen aller Klassenelemente enthalten.

Der **interface**-Abschnitt kann eine eigene **uses**-Klausel enthalten, die unmittelbar auf das Wort **interface** folgen muß. Informationen zur **uses-**Klausel finden Sie im Abschnitt »Unit-Referenzen und die uses-Klausel« auf Seite 3-6.

Der implementation-Abschnitt

Der **implementation**-Abschnitt einer Unit beginnt mit dem reservierten Wort **implementation** und endet mit dem Beginn des **initialization**-Abschnitts oder, wenn kein **initialization**-Abschnitt vorhanden ist, mit dem Ende der Unit. Der **implementation**-

Abschnitt definiert Prozeduren und Funktionen, die im **interface**-Abschnitt deklariert wurden. Im **implementation**-Abschnitt können diese Prozeduren und Funktionen in beliebiger Reihenfolge definiert und aufgerufen werden. Sie brauchen in den Prozedur- und Funktionsköpfen keine Parameterlisten anzugeben, wenn diese im **implementation**-Abschnitt definiert werden. Geben Sie jedoch eine Parameterliste an, muß diese der Deklaration im **interface**-Abschnitt exakt entsprechen.

Außer den Definitionen der öffentlichen Prozeduren und Funktionen kann der **implementation**-Abschnitt Deklarationen von Konstanten, Typen (einschließlich Klassen), Variablen, Prozeduren und Funktionen enthalten, die für die Unit *privat* sind, auf die also Clients nicht zugreifen können.

Der **implementation**-Abschnitt kann eine eigene **uses**-Klausel enthalten, die unmittelbar auf das Wort **implementation** folgen muß. Weitere Informationen zur **uses**-Klausel finden Sie im Abschnitt »Unit-Referenzen und die uses-Klausel« auf Seite 3-6.

Der initialization-Abschnitt

Der **initialization**-Abschnitt ist optional. Er beginnt mit dem reservierten Wort **initialization** und endet mit dem Beginn des **finalization**-Abschnitts oder - wenn kein **finalization**-Abschnitt vorhanden ist - mit dem Ende der Unit. Der **initialization**-Abschnitt enthält Anweisungen, die beim Programmstart in der angegebenen Reihenfolge ausgeführt werden. Arbeiten Sie beispielsweise mit definierten Datenstrukturen, können Sie diese im **initialization**-Abschnitt initialisieren.

Die **initialization**-Abschnitte von Units, die von Clients eingebunden werden, werden in der Reihenfolge ausgeführt, in der die Units in der **uses**-Klausel des Clients angegeben sind.

Der finalization-Abschnitt

Der **finalization**-Abschnitt ist optional und kann nur in Units verwendet werden, die auch einen **initialization**-Abschnitt enthalten. Der **finalization**-Abschnitt beginnt mit dem reservierten Wort **finalization** und endet mit dem Ende der Unit. Er enthält Anweisungen, die beim Beenden des Hauptprogramms ausgeführt werden. Im **finalization**-Abschnitt sollten Sie die Ressourcen freigeben, die im **initialization**-Abschnitt zugewiesen wurden.

finalization-Abschnitte werden in der umgekehrten Reihenfolge der **initialization**-Abschnitte ausgeführt. Initialisiert eine Anwendung beispielsweise die Units *A*, *B* und *C* in dieser Reihenfolge, werden die **finalization**-Abschnitte dieser Units in der Reihenfolge *C*, *B* und *A* ausgeführt.

Mit dem Beginn der Ausführung des Initialisierungscodes einer Unit ist sichergestellt, daß der zugehörige **finalization**-Abschnitt beim Beenden der Anwendung ausgeführt wird. Der **finalization**-Abschnitt muß deshalb auch unvollständig initialisierte Daten verarbeiten können, da der Initialisierungscode beim Auftreten eines Laufzeitfehlers möglicherweise nicht vollständig ausgeführt wird.

Unit-Referenzen und die uses-Klausel

Eine **uses**-Klausel in einem Programm, einer Bibliothek oder einer Unit gibt die von diesem Modul verwendeten Units an. Weitere Informationen zu Bibliotheken finden Sie in Kapitel 9, »Dynamische Link-Bibliotheken und Packages«. Eine **uses**-Klausel kann an folgenden Stellen im Quelltext verwendet werden:

- Projektdatei eines Programms oder einer Bibliothek
- **interface**-Abschnitt einer Unit
- **implementation**-Abschnitt einer Unit

Die meisten Projektdateien enthalten wie die **interface**-Abschnitte der meisten Units eine **uses**-Klausel. Der **implementation**-Abschnitt einer Unit kann eine eigene **uses**-Klausel enthalten.

Die Unit *System* wird automatisch von jeder Delphi-Anwendung verwendet und darf nicht in der **uses**-Klausel angegeben werden. (*System* implementiert Routinen für die Datei-E/A, String-Verarbeitung, Gleitkommaoperationen, dynamische Speicherzuweisung usw.) Andere Standard-Units (Bibliotheken) wie *SysUtils* müssen dagegen in der **uses**-Klausel angegeben werden. In den meisten Fällen fügt Delphi alle erforderlichen Units in die **uses**-Klausel ein, wenn eine Quelltextdatei generiert und gewartet wird.

Informationen zur Position und zum Inhalt der **uses**-Klausel finden Sie in den Abschnitten »Mehrere und indirekte Unit-Referenzen« auf Seite 3-7 und »Zirkuläre Unit-Referenzen« auf Seite 3-8.

Die Syntax der uses-Klausel

Eine **uses**-Klausel besteht aus dem reservierten Wort **uses**, einem oder mehreren durch Kommas voneinander getrennten Unit-Namen und einem abschließenden Semikolon. Zwei Beispiele:

```
uses Forms, Main;
uses Windows, Messages, SysUtils, Strings, Classes, Unit2, MyUnit;
```

In der **uses**-Klausel eines Programms bzw. einer Bibliothek kann auf den Namen jeder Unit das reservierte Wort **in** mit dem Namen einer Quelltextdatei folgen. Der Name wird mit oder ohne Pfad in Hochkommas angegeben. Die Pfadangabe kann absolut oder relativ sein. Zwei Beispiele:

```
uses Windows, Messages, SysUtils, Strings in 'C:\Classes\Strings.pas', Classes;
uses
   Forms,
   Main,
   Extra in '..\EXTRA\EXTRA.PAS';
```

Geben Sie **in** ... nach dem Unit-Namen ein, wenn Sie die Quelltextdatei einer Unit angeben müssen. Da in der Delphi-IDE vorausgesetzt wird, daß die Unit-Namen den Namen der Quelltextdateien entsprechen, in denen sie gespeichert sind, ist die Angabe des Namens der Quelltextdatei normalerweise nicht erforderlich. Das reservierte

Wort **in** wird nur benötigt, wenn die Position der Quelltextdatei aus folgenden Gründen nicht eindeutig ist:

- Die Quelltextdatei befindet sich in einem anderen Verzeichnis als die Projektdatei, und dieses Verzeichnis ist weder im Suchpfad des Compilers noch im Bibliothekspfad von Delphi aufgeführt.
- Mehrere Verzeichnisse im Suchpfad des Compilers enthalten Units mit identischen Namen.
- Sie compilieren in der Befehlszeile eine Konsolenanwendung und haben einer Unit einen Bezeichner zugeordnet, der nicht dem Namen der Quelltextdatei entspricht.

In der **uses**-Klausel einer Unit können Sie **in** nicht verwenden, um für den Compiler die Position einer Quelltextdatei anzugeben. Jede Unit muß sich im Suchpfad des Compilers, im Bibliothekspfad von Delphi oder in demselben Verzeichnis wie die Unit befinden, die auf diese Unit zugreift. Außerdem müssen die Namen der Units mit den Namen der Quelltextdateien identisch sein.

Mehrere und indirekte Unit-Referenzen

Die Reihenfolge der Units in der **uses**-Klausel bestimmt die Reihenfolge der Initialisierung dieser Units (siehe »Der initialization-Abschnitt« auf Seite 3-5) und wirkt sich auf die Suche des Compilers nach den Bezeichnern aus. Wenn zwei Units eine Variable, eine Konstante, einen Typ, eine Prozedur oder eine Funktion mit identischem Namen deklarieren, verwendet der Compiler die Deklaration der in der **uses**-Klausel zuletzt angegeben Unit. Wollen Sie auf den Bezeichner in einer anderen Unit zugreifen, müssen Sie den vollständigen Bezeichnernamen angeben: *Unitname.Bezeichner*.

Eine **uses**-Klausel muß nur die Units enthalten, die direkt vom Programm bzw. von der Unit verwendet werden, in dem bzw. der die **uses**-Klausel steht. Referenziert beispielsweise Unit *A* Konstanten, Typen, Variablen, Prozeduren oder Funktionen, die in Unit *B* deklariert sind, muß die Unit *B* explizit in der **uses**-Klausel von Unit *A* angegeben werden. Referenziert *B* wiederum Bezeichner aus Unit *C*, ist Unit *A* indirekt von Unit *C* abhängig. In diesem Fall muß Unit *C* nicht in einer **uses**-Klausel in Unit *A* angegeben werden. Der Compiler benötigt jedoch Zugriff auf die Units *B* und *C*, während Unit *A* verarbeitet wird.

Das folgende Beispiel illustriert diese indirekte Abhängigkeit:

```
program Prog;
uses Unit2;
const a = b;
⋮
unit Unit2;
interface
uses Unit1;
const b = c;
⋮
```

```
unit Unit1;
interface
const c = 1;
⋮
```

Hier hängt das Programm *Prog* direkt von *Unit2* ab, die wiederum direkt von *Unit1* abhängig ist. *Prog* ist also indirekt von *Unit1* abhängig. Da *Unit1* nicht in der **uses**-Klausel von *Prog* angeben ist, sind die in *Unit1* deklarierten Bezeichner für *Prog* nicht verfügbar.

Damit ein Client-Modul compiliert werden kann, muß der Compiler Zugriff auf alle Units haben, von denen der Client direkt oder indirekt abhängt. Sofern der Quelltext dieser Units nicht geändert wurde, benötigt der Compiler nur die DCU-Dateien, nicht jedoch die Quelltextdateien (PAS).

Werden im **interface**-Abschnitt einer Unit Änderungen vorgenommen, müssen die von dieser Unit abhängigen Units neu compiliert werden. Werden die Änderungen dagegen nur im **implementation**- oder einem anderen Abschnitt einer Unit vorgenommen, müssen die abhängigen Units nicht neu compiliert werden. Der Compiler überwacht diese Abhängigkeiten und nimmt Neucompilierungen nur vor, wenn dies erforderlich ist.

Zirkuläre Unit-Referenzen

Wenn sich Units direkt oder indirekt gegenseitig referenzieren, werden sie als gegenseitig voneinander abhängig bezeichnet. Gegenseitige Abhängigkeiten sind zulässig, wenn keine zirkulären Pfade auftreten, die eine **uses**-Klausel im **interface**-Abschnitt einer Unit mit der **uses**-Klausel im **interface**-Abschnitt einer anderen Unit verbinden. Ausgehend vom **interface**-Abschnitt einer Unit darf es also nicht möglich sein, über die Referenzen in den **uses**-Klauseln in **interface**-Abschnitten anderer Units wieder zu dieser Ausgangsklausel zu gelangen. Damit eine Gruppe gegenseitiger Abhängigkeiten gültig ist, muß der Pfad jeder zirkulären Referenz über die **uses**-Klausel mindestens eines **implementation**-Abschnitts führen.

Im einfachsten Fall mit zwei gegenseitig voneinander abhängigen Units bedeutet dies, daß die Units sich nicht gegenseitig in den **uses**-Klauseln der jeweiligen **interface**-Abschnitte referenzieren dürfen. Das folgende Beispiel führt also bei der Compilierung zu einem Fehler:

```
unit Unit1;
interface
uses Unit2;
⋮
unit Unit2;
interface
uses Unit1;
⋮
```

Eine legale gegenseitige Referenzierung ist jedoch möglich, indem eine der Referenzen in den **implementation**-Abschnitt verschoben wird:

```
unit Unit1;
interface
```

```
uses Unit2;
⋮
unit Unit2;
interface
⋮
implementation
uses Unit1;
⋮
```

Um unzulässige zirkuläre Referenzen zu vermeiden, sollten Sie die Units möglichst in der **uses-**Klausel des **implementation**-Abschnitts angeben. Werden jedoch Bezeichner einer anderen Unit im **interface**-Abschnitt verwendet, muß die betreffende Unit in der **uses-**Klausel im **interface**-Abschnitt angegeben werden.

Kapitel

4

Syntaktische Elemente

Object Pascal verwendet den ASCII-Zeichensatz mit den Buchstaben *A* bis *Z* und *a* bis *z*, den Ziffern *0* bis *9* und weiteren Standardzeichen. Die Sprache unterscheidet *nicht* zwischen Groß- und Kleinschreibung. Das Leerzeichen (ASCII 32) und die Steuerzeichen (ASCII 0 bis 31 einschließlich ASCII 13 für Zeilenvorschub) werden als *Blanks* bezeichnet.

Kombinationen, die sich aus den grundlegenden syntaktischen Elementen (den sogenannten *Token*) zusammensetzen, ergeben Ausdrücke, Deklarationen und Anweisungen. Eine *Anweisung* beschreibt eine algorithmische Aktion, die innerhalb eines Programms ausgeführt werden kann. Ein *Ausdruck* ist eine syntaktische Einheit, die in einer Anweisung enthalten ist und einen Wert beschreibt. Eine *Deklaration* definiert einen Bezeichner (z.B. den Namen einer Funktion oder Variablen), der in Ausdrükken und Anweisungen verwendet wird. Sie weist dem Bezeichner bei Bedarf auch Speicherplatz zu.

Grundlegende syntaktische Elemente

Im Prinzip ist ein Programm eine Folge von *Token*, die durch Trennzeichen voneinander getrennt sind. *Token* sind die kleinsten Texteinheiten in einem Programm. Ein *Trennzeichen* kann entweder ein Leerzeichen oder ein Kommentar sein. Prinzipiell ist die Trennung zweier Token durch ein Trennzeichen nicht in jedem Fall erforderlich. Das folgende Quelltextfragment ist beispielsweise korrekt:

```
Size:=20;Price:=10;
```

Aufgrund gängiger Konventionen und zugunsten der besseren Lesbarkeit verdient aber die folgende Schreibweise den Vorzug:

```
Size := 20;
Price := 10;
```

Ein Token ist ein *Symbol*, ein *Bezeichner*, ein *reserviertes Wort*, eine *Direktive*, eine *Ziffer*, ein *Label* oder eine *String-Konstante*. Außer bei String-Konstanten kann ein Trennzei-

chen nicht Teil eines Token sein. Zwei aufeinanderfolgende Bezeichner, reservierte Wörter, Ziffern oder Label müssen durch eines oder mehrere Trennzeichen voneinander getrennt werden.

Symbole

Symbole sind nichtalphanumerische Zeichen bzw. Zeichenpaare, die eine feste Bedeutung haben. Als Symbole gelten folgende Einzelzeichen:

 # $ & ' ( ) * + , – . / : ; < = > @ [ ] ^ { }

Die folgenden Zeichenpaare sind ebenfalls Symbole:

(* (. *) .) .. // := <= >= <>

Die öffnende eckige Klammer (**[**) ist gleichbedeutend mit dem Zeichenpaar, das aus einer öffnenden runden Klammer und einem Punkt besteht (**(.**). Die schließende eckige Klammer (**]**) entspricht dem Zeichenpaar, das aus einem Punkt und einer schließenden runden Klammer besteht (**.)**). Die Folge öffnende runde Klammer-Sternchen-Sternchen-schließende runde Klammer (**(* *)**) entspricht einem Paar geschweifter Klammer ({ }).

Beachten Sie, daß **!**, " (doppeltes Anführungszeichen), **%**, **?**, \, **_** (Unterstrich), **|** und **~** keine Symbole sind.

Bezeichner

Bezeichner werden für Konstanten, Variablen, Felder, Typen, Eigenschaften, Prozeduren, Funktionen, Programme, Units, Bibliotheken und Packages verwendet. Obwohl ein Bezeichner beliebig lang sein, sind nur die ersten 255 Zeichen signifikant. Ein Bezeichner muß mit einem Buchstaben oder einem Unterstrich (_) beginnen und darf keine Leerzeichen enthalten. Auf das erste Zeichen können Buchstaben, Ziffern und Unterstriche folgen. Reservierte Wörter dürfen nicht als Bezeichner verwendet werden.

Da die Groß-/Kleinschreibung in Object Pascal nicht berücksichtigt wird, sind beispielsweise für den Bezeichner *CalculateValue* folgende Schreibweisen zulässig:

```
CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE
```

Qualifizierte Bezeichner

Wenn Sie einen Bezeichner verwenden, der an mehreren Stellen deklariert wurde, muß dieser unter Umständen *qualifiziert* werden. Die Syntax für einen qualifizierten Bezeichner lautet

Bezeichner$_1$.*Bezeichner*$_2$

Dabei qualifiziert *Bezeichner*$_1$ den *Bezeichner*$_2$. Angenommen, zwei Units deklarieren eine Variable namens *CurrentValue*. Mit der folgenden Anweisung legen Sie fest, daß auf *CurrentValue* in *Unit2* zugegriffen werden soll:

```
Unit2.CurrentValue
```

Bezeichner können über mehrere Ebenen qualifiziert werden. So wird z.B. mit der folgenden Zeile die Methode *Click* in *Button1* von *Form1* aufgerufen.

```
Form1.Button1.Click
```

Wenn Sie einen Bezeichner nicht qualifizieren, wird seine Interpretation von den Regeln für den Gültigkeitsbereich festgelegt, die im Abschnitt »Blöcke und Gültigkeitsbereich« auf Seite 4-29 beschrieben werden.

Reservierte Wörter

Die folgenden reservierten Wörter können weder neu definiert noch als Bezeichner verwendet werden.

Tabelle 4.1 Resrevierte Wörter

and	**downto**	**in**	**or**	**string**
array	**else**	**inherited**	**out**	**then**
as	**end**	**initialization**	**packed**	**threadvar**
asm	**except**	**inline**	**procedure**	**to**
begin	**exports**	**interface**	**program**	**try**
case	**file**	**is**	**property**	**type**
class	**finalization**	**label**	**raise**	**unit**
const	**finally**	**library**	**record**	**until**
constructor	**for**	**mod**	**repeat**	**uses**
destructor	**function**	**nil**	**resourcestring**	**var**
dispinterface	**goto**	**not**	**set**	**while**
div	**if**	**object**	**shl**	**with**
do	**implementation**	**of**	**shr**	**xor**

Neben den in Tabelle 4.1 aufgeführten Wörtern gelten in Objekttypdeklarationen auch **private**, **protected**, **public**, **published** und **automated** als reservierte Wörter. In allen anderen Fällen werden sie als Direktiven behandelt. Die Wörter **at** und **on** haben ebenfalls eine besondere Bedeutung.

Direktiven

Direktiven haben in Object Pascal spezielle Bedeutungen. Sie werden aber im Gegensatz zu reservierten Wörtern nur in Umgebungen verwendet, in denen benutzerdefinierte Bezeichner nicht auftreten können. Aus diesem Grund lassen sich Bezeichner definieren, die wie Direktiven lauten. (Dieses Vorgehen ist allerdings nicht empfehlenswert.)

Tabelle 4.2 Direktiven

absolute	**dynamic**	**name**	**protected**	**resident**
abstract	**export**	**near**	**public**	**safecall**
assembler	**external**	**nodefault**	**published**	**stdcall**
automated	**far**	**overload**	**read**	**stored**
cdecl	**forward**	**override**	**readonly**	**virtual**
contains	**implements**	**package**	**register**	**write**
default	**index**	**pascal**	**reintroduce**	**writeonly**
dispid	**message**	**private**	**requires**	

Ziffern

Integer- und Real-Konstanten lassen sich in dezimaler Notation als Folge von Ziffern ohne Kommas oder Leerzeichen darstellen. Das Vorzeichen wird durch den vorangestellten Operator + bzw. - angegeben. Werte sind per Vorgabe positiv (67258 ist also identisch mit +67258) und müssen im Bereich des größten vordefinierten Real- bzw. Integer-Typs liegen.

Zahlen mit Nachkommastellen oder Exponenten bezeichnen Real-Konstanten. Andere Dezimalzahlen sind Integer-Konstanten. Bei Real-Typen wird die wissenschaftliche Notation (E oder e gefolgt von einem Exponenten) als »mal 10 hoch« gelesen. So bedeutet 7E-2 beispielsweise 7×10^{-2}, 12.25e+6 oder 12.25e6 bedeutet 12.25×10^{6}.

Das Dollarzeichen als Vorzeichen kennzeichnet eine hexadezimale Zahl, beispielsweise $8F. Hexadezimalzahlen müssen im Bereich von $00000000 bis $FFFFFFFF liegen. Das Vorzeichen einer Hexadezimalzahl ist durch das am weitesten links stehende (das höchstwertige) Bit der binären Entsprechung vorgegeben.

Ausführliche Informationen über Real- und Integer-Typen finden Sie in Kapitel 5, »Datentypen, Variablen und Konstanten«.

Label

Ein Label ist eine Folge von maximal vier Ziffern, d.h. eine Zahl zwischen 0 und 9999. Führende Nullen sind nicht signifikant. Auch Bezeichner können als Label fungieren.

Label werden in **goto**-Anweisungen verwendet. Ausführliche Informationen über **goto**-Anweisungen und Label finden Sie im Abschnitt »Goto-Anweisungen« auf Seite 4-19.

Zeichen-Strings

Ein Zeichen-String (auch *String-Literal* oder *String-Konstante* genannt) kann aus einem String in Anführungszeichen, einem Steuerzeichen-String oder einer Kombination aus beiden bestehen. Trennzeichen treten nur bei Strings in Anführungszeichen auf.

Ein String in Anführungszeichen setzt sich aus einer Folge von maximal 255 Zeichen des erweiterten ASCII-Zeichensatzes zusammen, muß in einer Zeile stehen und in halbe Anführungszeichen eingeschlossen sein. Ein String in Anführungszeichen, der zwischen den halben Anführungszeichen kein Zeichen enthält, ist ein *Null-String*. Zwei in einem String in Anführungszeichen unmittelbar aufeinanderfolgende halbe Anführungszeichen stehen für ein einzelnes Anführungszeichen. Einige Beispiele:

```
'BORLAND'           { BORLAND }
'Müller''s Büro'    { Müller's Büro }
''''                { ' }
''                  { Ein Null-String }
' '                 { Ein Leerzeichen }
```

Ein Steuerzeichen-String ist eine Folge von einem oder mehreren Steuerzeichen. Jedes dieser Steuerzeichen besteht aus einem **#**-Symbol und einer vorzeichenlosen Integer-Konstante zwischen 0 und 255 (dezimal oder hexadezimal), die das entsprechende ASCII-Zeichen bezeichnet. Der Steuerzeichen-String

```
#89#111#117
```

entspricht dem folgenden String in Anführungszeichen:

```
'You'
```

Sie können Strings in Anführungszeichen mit Steuerzeichen-Strings kombinieren und damit längere Strings bilden. Beispielsweise wird mit dem folgenden String das Zeichen für ein Wagenrücklauf / Zeilenvorschub zwischen Zeile 1 und Zeile 2 eingefügt:

```
'Zeile 1'#13#10'Zeile 2'
```

Strings in Anführungszeichen lassen sich allerdings nicht auf diese Weise miteinander verbinden, weil ein Paar aufeinanderfolgender halber Anführungszeichen als ein einzelnes Zeichen interpretiert wird. Für das Zusammenfügen von Strings in Anführungszeichen steht der Operator + zur Verfügung, der im Abschnitt »String-Operatoren« auf Seite 4-9 beschrieben wird. Sie können die betreffenden Strings aber auch einfach zu einem einzigen String in Anführungszeichen kombinieren.

Die *Länge* eines Zeichen-Strings entspricht der Anzahl der Zeichen im String. Zeichen-Strings mit beliebiger Länge sind kompatibel zu allen String-Typen bzw. zum Typ PChar, wenn die erweiterte Syntax **({$X+})** aktiviert ist. Ein Zeichen-String der Länge 1 ist kompatibel zu jedem Zeichentyp. Ein nichtleerer String der Länge *n* ist kompatibel zu gepackten Arrays mit *n* Zeichen. Ausführliche Informationen zu String-Typen finden Sie in Kapitel 5, »Datentypen, Variablen und Konstanten«.

Kommentare und Compiler-Direktiven

Kommentare werden vom Compiler ignoriert. Dies gilt jedoch nicht für Kommentare, die als Trennzeichen (zur Trennung aufeinanderfolgender Token) oder als Compiler-Direktiven fungieren.

Kommentare lassen sich auf verschiedene Arten kennzeichnen:

```
{ Text zwischen einer öffnenden und einer schließenden geschweiften Klammer. }
(* Text zwischen einer öffnenden runden Klammer plus einem Sternchen und
```

```
   einem Sternchen plus einer schließenden runden Klammer. *)
// Beliebiger Text zwischen einem doppelten Schrägstrich und dem Zeilenende.
```

Ein Kommentar, in dem unmittelbar nach { oder (* ein Dollarzeichen ($) steht, ist eine Compiler-Direktive. Die folgende Direktive weist beispielsweise den Compiler an, Warnmeldungen zu generieren:

```
{$WARNINGS OFF}
```

Ausdrücke

Ein *Ausdruck* ist eine Konstruktion, die einen Wert zurückliefert. Hier einige Beispiele für Ausdrücke:

```
X                  { Variable }
@X                 { Adresse einer Variable }
15                 { Integer-Konstante }
InterestRate       { Variable }
Calc(X,Y)          { Funktionsaufruf }
X * Y              { Produkt von X und Y }
Z / (1 - Z)        { Quotient von Z und (1 - Z) }
X = 1.5            { Boolescher Ausdruck }
C in Range1        { Boolescher Ausdruck }
not Done           { Negation eines Booleschen Ausdrucks }
['a','b','c']      { Menge }
Char(48)           { Wert einer Typumwandlung }
```

Die einfachsten Ausdrücke sind Variablen und Konstanten (siehe Kapitel 5, »Datentypen, Variablen und Konstanten«). Komplexere Ausdrücke werden mit Hilfe von *Operatoren*, *Funktionsaufrufen*, *Mengenkonstruktoren*, *Indizes* und *Typumwandlungen* aus einfachen Ausdrücken gebildet.

Operatoren

Operatoren verhalten sich wie vordefinierte Funktionen, die Bestandteil der Sprache Object Pascal sind. So setzt sich beispielsweise der Ausdruck *(X + Y)* aus den Variablen *X* und *Y* (den sogenannten *Operanden*) und dem Operator + zusammen. Wenn *X* und *Y* den Typ Integer oder Real haben, liefert der Ausdruck *(X + Y)* die Summe der beiden Werte zurück. Operatoren sind **@**, **not**, **^**, *****, **/**, **div**, **mod**, **and**, **shl**, **shr**, **as**, **+**, –, **or**, **xor**, **=**, **>**, **<**, **<>**, **<=**, **>=**, **in** und **is**.

Die Operatoren **@**, **not** und **^** sind *unäre* Operatoren und haben nur einen Operanden. Alle anderen Operatoren sind *binär* und haben zwei Operanden. Eine Ausnahme bilden die Operatoren **+** und **-**, die entweder unär oder binär sein können. Ein unärer Operator steht immer vor seinem Operanden (z.B. *-B*). Eine Ausnahme von dieser Regel bildet der Operator **^**, der auf seinen Operanden folgt (z.B. *P^*). Binäre Operatoren stehen immer zwischen ihren Operanden (z.B. *A = 7*).

Das Verhalten einiger Operatoren hängt von dem Datentyp ab, der an sie übergeben wird. Beispielsweise führt der Operator **not** eine bitweise Negation eines Integer-Operanden und eine logischen Negation eines Booleschen Operanden aus. Solche Operatoren sind deshalb auch mehreren Kategorien zugeordnet.

Mit Ausnahme von **^, is** und **in** können alle Operatoren Operanden vom Typ *Variant* akzeptieren. Einzelheiten hierzu finden Sie im Abschnitt »Variante Typen« auf Seite 5-33.

Die Erläuterungen in den folgenden Abschnitten gehen davon aus, daß Sie mit den Datentypen von Object Pascal vertraut sind. Weitere Informationen über Datentypen finden Sie in Kapitel 5, »Datentypen, Variablen und Konstanten«.

Informationen zur Rangfolge der Operatoren in komplexen Ausdrücken finden Sie im Abschnitt »Regeln für die Rangfolge von Operatoren« auf Seite 4-13.

Arithmetische Operatoren

Zu den arithmetische Operatoren für Real- oder Integer-Operanden gehören die Operatoren +, –, *, /, **div** und **mod**.

Tabelle 4.3 Binäre arithmetische Operatoren

Operator	Operation	Operandtyp	Ergebnistyp	Beispiel
+	Addition	Integer, Real	Integer, Real	`X + Y`
	Subtraktion	Integer, Real	Integer, Real	`Ergebnis - 1`
*	Multiplikation	Integer, Real	Integer, Real	`P * InterestRate`
/	Gleitkommadivision	Integer, Real	Real	`X / 2`
div	Ganzzahlige Division	Integer	Integer	`Total div UnitSize`
mod	Rest	Integer	Integer	`Y mod 6`

Tabelle 4.4 Unäre arithmetische Operatoren

Operator	Operation	Operandtyp	Ergebnistyp	Beispiel
+	Positives Vorzeichen	Integer, Real	Integer, Real	`+7`
-	Negatives Vorzeichen	Integer, Real	Integer, Real	`-X`

Für arithmetische Operatoren gelten die folgenden Regeln:

- Der Wert von *x/y* entspricht dem Typ Extended, unabhängig vom Typ von *x* und *y*. Bei allen anderen Operatoren ist das Ergebnis vom Typ Extended, wenn mindestens ein Operand den Typ Real hat. Ist das nicht der Fall, ist das Ergebnis vom Typ Int64, wenn mindestens ein Operand den Typ Int64 hat, ansonsten ist das Ergebnis vom Typ Integer. Wenn der Typ eines Operanden ein Unterbereich eines Integer-Typs ist, wird er wie ein Operand vom Typ Integer behandelt.
- Der Wert von *x* **div** *y* entspricht dem Wert von *x/y*, abgerundet in Richtung Null bis zum nächsten Integer-Wert.
- Der Operator **mod** liefert den Rest, der sich bei der Division seiner Operanden ergibt. Das bedeutet: *x mod y = x – (x div y) * y*.
- Wenn y in einem Ausdruck der Form *x/y*, *x div y* oder *x mod y* den Wert Null hat, tritt ein Laufzeitfehler ein.

Boolesche Operatoren

Die Operanden der Booleschen Operatoren **not**, **and**, **or** und **xor** können einen beliebigen Booleschen Typ haben. Die Operatoren liefern einen Wert vom Typ Boolean zurück.

Tabelle 4.5 Boolesche Operatoren

Operator	Operation	Operandtyp	Ergebnistyp	Beispiel
not	Negation	Boolescher Typ	*Boolean*	`not (C in MySet)`
and	Konjunktion	Boolescher Typ	*Boolean*	`Done and (Total > 0)`
or	Disjunktion	Boolescher Typ	*Boolean*	`A or B`
xor	Exklusive Disjunktion	Boolescher Typ	*Boolean*	`A xor B`

Diese Operatoren folgen den Standardregeln der Booleschen Logik. Beispielsweise liefert ein Ausdruck der Form *x and y* nur dann den Wert *True*, wenn beide Operanden den Wert *True* haben.

Vollständige Auswertung und Kurzschlußverfahren im Vergleich

Der Delphi-Compiler unterstützt zwei Auswertungsmodi für die Operatoren **and** und **or**: die vollständige Auswertung und das Kurzschlußverfahren. Bei der *vollständigen Auswertung* werden alle Operanden eines **and-** oder **or**-Ausdrucks auch dann ausgewertet, wenn das Resultat des gesamten Ausdrucks bereits feststeht. Das *Kurzschlußverfahren* geht streng von links nach rechts vor und wird beendet, sobald das Ergebnis des gesamten Ausdrucks feststeht. Wenn beispielsweise der Ausdruck *A and B* im Kurzschlußverfahren ausgewertet wird und *A* den Wert *False* hat, wertet der Compiler *B* nicht mehr aus, da bereits feststeht, daß nach der Auswertung von *A* auch der gesamte Ausdruck den Wert *False* haben wird.

Das Kurzschlußverfahren ist normalerweise vorzuziehen, da es schneller ausgeführt wird und (in den meisten Fällen) einen geringeren Quelltextumfang erfordert. Die vollständige Auswertung ist von Nutzen, wenn ein Operand eine Funktion mit Nebeneffekten ist, die Änderungen in der Ausführung des Programms bewirken.

Das Kurzschlußverfahren ermöglicht auch Konstruktionen, die sonst zu unzulässigen Laufzeitoperationen führen würden. Die folgende Anweisung iteriert bis zum ersten Komma durch den String *S*:

```
while (I <= Length(S)) and (S[I] <> ',') do
begin
   ⋮
   Inc(I);
end;
```

Wenn *S* keine Kommas enthält, wird *I* bei der letzten Iteration auf einen Wert erhöht, der größer ist als *Length(S)*. Beim nachfolgenden Test der **while**-Bedingung wird bei einer vollständigen Auswertung versucht, *S[I]* zu lesen, was zu einem Laufzeitfehler führen kann. Beim Kurzschlußverfahren wird dagegen der zweite Teil der **while**-Bedingung (*S[I]* <> *','*) nicht mehr ausgewertet, nachdem der erste Teil *False* ergibt.

Zur Steuerung des Auswertungsmodus dient die Compiler-Direktive **$B**. Voreingestellt ist der Status **{$B–}** für das Kurzschlußverfahren. Um die vollständige Auswer-

tung lokal zu aktivieren, fügen Sie die Direktive **{$B+}** in den Quelltext ein. Sie können auch auf Projektebene in diesen Status wechseln, indem Sie in der Registerkarte *Compiler* des Dialogfeldes *Projektoptionen* die Option *Boolesche Ausdrücke* vollständig aktivieren.

Logische (bitweise) Operatoren

Die folgenden logischen Operatoren bearbeiten Integer-Operanden bitweise. Wenn z.B. der in *X* (binär) gespeicherte Wert 001101 und der in *Y* gespeicherte Wert 100001 lautet, weist die folgende Operation an *Z* den Wert 101101 zu:

```
Z := X or Y;
```

Tabelle 4.6 Logische (bitweise) Operatoren

Operator	Operation	Operandtyp	Ergebnistyp	Beispiele
not	Bitweise Negation	Integer	Integer	`not X`
and	Bitweises and	Integer	Integer	`X and Y`
or	Bitweises or	Integer	Integer	`X or Y`
xor	Bitweises xor	Integer	Integer	`X xor Y`
shl	Bitverschiebung nach links	Integer	Integer	`X shl 2`
shr	Bitverschiebung nach rechts	Integer	Integer	`Y shl I`

Für bitweise Operatoren gelten die folgenden Regeln:

- Das Ergebnis einer **not**-Operation hat denselben Typ wie der Operand.
- Wenn beide Operanden einer **and**-, **or-** oder **xor**-Operation vom Typ Integer sind, hat das Ergebnis den vordefinierten Integer-Typ mit dem kleinsten Bereich, im dem alle für beide Typen möglichen Werte enthalten sind.
- Die Operationen *x* **shl** *y* und *x* **shr** *y* verschieben den Wert von *x* um *y* Bits nach links oder rechts. Dies entspricht der Multiplikation oder Division von *x* durch 2^{y}. Das Ergebnis hat denselben Typ wie *x*. Wenn beispielsweise in *N* der Wert 01101 (dezimal 13) gespeichert ist, liefert *N shl 1* den Wert 11010 (dezimal 26) zurück.

String-Operatoren

Die relationalen Operatoren **=**, **<>**, **<**, **>**, **<=** und **>=** funktionieren auch mit String-Operanden (siehe den Abschnitt »Relationale Operatoren« auf Seite 4-11). Der Operator **+** verkettet zwei Strings.

Tabelle 4.7 String-Operatoren

Operator	Operation	Operandtyp	Ergebnistyp	Beispiel
+	Verkettung	String, gepackter String, Char	String	`S + '. '`

Für die Verkettung von Strings gelten folgende Regeln:

- Die Operanden für **+** können Strings, gepackte Strings (gepackte Arrays vom Typ Char) oder Zeichen sein. Wenn jedoch ein Operand vom Typ WideChar ist, muß der andere Operand ein langer String sein.
- Das Ergebnis einer **+**-Operation ist mit allen String-Typen kompatibel. Wenn aber beide Operanden kurze Strings oder Zeichen sind und ihre gemeinsame Länge größer als 255 ist, wird das Ergebnis nach dem 255. Zeichen abgeschnitten.

Zeiger-Operatoren

Die relationalen Operatoren **<**, **>**, **<=** und **>=** können Operanden vom Typ PChar haben (siehe den Abschnitt »Relationale Operatoren« auf Seite 4-11). Bei den folgenden Operatoren können die Operanden auch Zeiger sein. Ausführlichere Informationen über Zeiger finden Sie im Abschnitt »Zeiger und Zeigertypen« auf Seite 5-27.

Tabelle 4.8 Zeichenzeiger-Operatoren

Operator	Operation	Operandtyp	Ergebnistyp	Beispiel
+	Zeiger-Addition	Zeichenzeiger, Integer	Zeichenzeiger	`P + I`
-	Zeiger-Subtraktion	Zeichenzeiger, Integer	Zeichenzeiger, Integer	`P - Q`
^	Zeiger-Dereferenzierung	Zeiger	Basistyp von Zeiger	`P^`
=	Gleichheit	Zeiger	Boolean	`P = Q`
<>	Ungleichheit	Zeiger	Boolean	`P <> Q`

Der Operator **^** dereferenziert einen Zeiger. Sein Operand kann ein Zeiger auf einen beliebigen Typ mit Ausnahme des generischen Typs Pointer sein, der vor der Dereferenzierung umgewandelt werden muß.

P = *Q* ist nur dann *True*, wenn *P* und *Q* auf dieselbe Adresse zeigen.

Mit den Operatoren **+** und **–** kann der Offset eines Zeichenzeigers erhöht oder erniedrigt werden. Mit dem Operator **–** können Sie außerdem den Unterschied zwischen den Offsets zweier Zeichenzeiger berechnen. Für Zeiger-Operatoren gelten die folgenden Regeln:

- Wenn *I* vom Typ Integer und *P* ein Zeichenzeiger ist, addiert *P* + *I* den Wert *I* zu der von *P* angegebenen Adresse. Es wird also ein Zeiger auf die Adresse zurückgegeben, die *I* Zeichen hinter *P* liegt. (Der Ausdruck *I* + *P* entspricht *P* + *I*.) *P* – *I* subtrahiert *I* von der Adresse, die von *P* angegeben wird. Das Ergebnis ist ein Zeiger auf die Adresse, die *I* Zeichen vor *P* liegt.
- Wenn *P* und *Q* Zeichenzeiger sind, berechnet *P* – *Q* die Differenz zwischen der von *P* angegebenen (höheren) und der von *Q* angegebenen (niedrigeren) Adresse. Es wird also ein Integer-Wert für die Anzahl der Zeichen zwischen *P* und *Q* zurückgegeben. Das Ergebnis von *P* + *Q* ist nicht definiert.

Mengenoperatoren

Die folgenden Operatoren haben eine Menge als Operanden.

Tabelle 4.9 Mengenoperatoren

Operator	Operation	Operandtyp	Ergebnistyp	Beispiel
+	Vereinigung	Set	Set	`Set1 + Set2`
–	Differenz	Set	Set	`S - T`
*	Schnittmenge	Set	Set	`S * T`
<=	Teilmenge	Set	Boolean	`Q <= MySet`
>=	Obermenge	Set	Boolean	`S1 >= S2`
=	Gleich	Set	Boolean	`S2 = MySet`
<>	Ungleich	Set	Boolean	`MySet <> S1`
in	Element	Ordinal, Set	Boolean	`A in Set1`

Für **+**, – und ***** gelten die folgenden Regeln:

- Der Ordinalwert *O* ist nur in *X* **+** *Y* enthalten, wenn *O* in *X* oder *Y* (oder beiden) enthalten ist. *O* ist nur in *X* **–** *Y* enthalten, wenn *O* in *X* und nicht in *Y* enthalten ist. *O* ist nur in *X* ***** *Y* enthalten, wenn *O* sowohl in *X* als auch in *Y* enthalten ist.
- Das Ergebnis einer Operation mit **+, –** oder ***** ist vom Typ **set** of *A..B*, wobei *A* der kleinste und *B* der größte Ordinalwert in der Ergebnismenge ist.

Für **<=**, **>=**, **=**, **<>** und **in** gelten die folgenden Regeln:

- *X* **<=** *Y* ist nur dann *True*, wenn jedes Element von *X* ein Element von *Y* ist; *Z* **>=** *W* ist gleichbedeutend mit *W* **<=** *Z*. *U* **=** *V* ist nur dann *True*, wenn *U* und *V* genau dieselben Elemente enthalten.
- Für einen Ordinalwert *O* und eine Menge *S* ist *O* **in** *S* nur dann *True*, wenn *O* ein Element von *S* ist.

Relationale Operatoren

Relationale Operatoren dienen dem Vergleich zweier Operanden. Die Operatoren =, <>, <= und >= lassen sich auch auf Mengen anwenden (siehe »Mengenoperatoren« auf Seite 4-11). Die Operatoren = und <> können auch auf Zeiger angewendet werden (siehe »Zeiger-Operatoren« auf Seite 4-10).

Tabelle 4.10 Relationale Operatoren

Operator	Operation	Operandtyp	Ergebnistyp	Beispiel
=	gleich	Einfacher Typ, Klassen-, Klassenreferenz-, Schnittstellen-, String- und gepackter String-Typ	Boolean	`I = Max`
<>	ungleich	Einfacher Typ, Klassen-, Klassenreferenz-, Schnittstellen-, String- und gepackter String-Typ	Boolean	`X <> Y`

Tabelle 4.10 Relationale Operatoren (Fortsetzung)

Operator	Operation	Operandtyp	Ergebnistyp	Beispiel
<	kleiner als	Einfacher Typ, String-, gepackter String und PChar-Typ	Boolean	`X < Y`
>	größer als	Einfacher Typ, String-, gepackter String und PChar-Typ	Boolean	`Len > 0`
<=	kleiner oder gleich	Einfacher Typ, String-, gepackter String und PChar-Typ	Boolean	`Cnt <= I`
>=	größer oder gleich	Einfacher Typ, String-, gepackter String und PChar-Typ	Boolean	`I >= 1`

Bei den meisten einfachen Typen ist der Vergleich unkompliziert. *I* = *J* ist beispielsweise nur dann *True*, wenn *I* und *J* denselben Wert haben. Für relationale Operatoren gelten die folgenden Regeln:

- Operanden müssen kompatible Typen haben, mit folgender Ausnahme: Reelle und Integer-Typen können miteinander verglichen werden.
- Strings werden basierend auf der Reihenfolge des erweiterten ASCII-Zeichensatzes verglichen. Zeichen-Typen werden als Strings der Länge 1 behandelt.
- Zwei gepackte Strings müssen beim Vergleich dieselbe Anzahl von Komponenten aufweisen. Wird ein gepackter String mit *n* Komponenten mit einem String verglichen, wird der gepackte String als String der Länge *n* behandelt.
- Die Operatoren <, >, <= und >= werden nur dann auf PChar-Operanden angewendet, wenn die beiden Zeiger auf Elemente in demselben Zeichen-Array zeigen.
- Die Operatoren = und <> akzeptieren Operanden mit einem Klassen- oder Klassenreferenztyp. Mit Operanden eines Klassentyps werden = und <> entsprechend den Regeln für Zeiger ausgewertet: *C* = *D* ist nur dann *True*, wenn *C* und *D* auf dasselbe Instanzobjekt zeigen. Mit Operanden eines Klassenreferenztyps ist *C* = *D* nur dann *True*, wenn *C* und *D* dieselbe Klasse bezeichnen. Ausführliche Informationen zu Klassen finden Sie in Kapitel 7, »Klassen und Objekte«.

Klassen-Operatoren

Die Operatoren **as** und **is** übernehmen Klassen und Instanzobjekte als Operanden; **as** kann auch auf Schnittstellen angewendet werden. Ausführliche Informationen hierzu finden Sie in Kapitel 7, »Klassen und Objekte«, und in Kapitel 10, »Objektschnittstellen«.

Die relationalen Operatoren **=** und **<>** können auch auf Klassen angewendet werden. Informationen hierzu finden Sie im Abschnitt »Relationale Operatoren« weiter oben.

Der Operator @

Der Operator **@** liefert die Adresse einer Variable, Funktion, Prozedur oder Methode, d.h. er erzeugt einen Zeiger auf seinen Operanden. Ausführlichere Informationen über Zeiger finden Sie im Abschnitt »Zeiger und Zeigertypen« auf Seite 5-27. Für den Operator **@** gelten folgende Regeln:

- Wenn *X* eine Variable ist, liefert *@X* die Adresse von *X* zurück. (Wenn *X* eine prozedurale Variable ist, gelten besondere Regeln; siehe »Prozedurale Typen in Anweisungen und Ausdrücken« auf Seite 5-31.) *@X* ist vom Typ Pointer, wenn die Standard-Compilerdirektive **{$T–}** aktiviert ist. Im Status **{$T+}** ist *@**X*** vom Typ **^T**, wobei *T* denselben Typ wie *X* hat.
- Wenn *F* eine Routine (eine Funktion oder Prozedur) ist, liefert *@F* den Eintrittspunkt von *F*. *@F* ist immer vom Typ Pointer.
- Wenn **@** auf eine in einer Klasse definierte Methode angewendet wird, muß der Methodenbezeichner mit dem Klassennamen qualifiziert werden. So zeigt die folgende Anweisung auf die Methode *DoSomething von TMyClass:*

```
@TMyClass.DoSomething
```

Ausführliche Informationen zu Klassen und Methoden finden Sie in Kapitel 7, »Klassen und Objekte«.

Regeln für die Rangfolge von Operatoren

In komplexen Ausdrücken wird die Reihenfolge, in der Operationen ausgeführt werden, durch die Rangfolge der Operatoren festgelegt.

Tabelle 4.11 Wertigkeit der Operatoren

Operator	Rangfolge
@, **not**	Erste (höchste)
*****, **/**, **div**, **mod**, **and**, **shl**, **shr**, **as**	Zweite
+, , **or**, **xor**	Dritte
=, <>, <, >, <=, >=, **in**, **is**	Vierte (niedrigste)

Ein Operator mit einer höheren Rangfolge wird vor einem Operator mit einer niedrigeren Rangfolge ausgewertet. Operatoren mit gleicher Rangfolge werden von links nach rechts ausgewertet. Aus diesem Grund multipliziert der folgende Ausdruck zunächst *Y* mit *Z* und addiert dann *X* zum Resultat der Multiplikation:

```
X + Y * Z
```

Die Operation ***** (Multiplikation) wird zuerst ausgeführt, weil dieser Operator eine höhere Rangfolge als der Operator **+** hat. Im folgenden Ausdruck wird aber zunächst *Y* von *X* subtrahiert und danach *Z* zum Resultat addiert:

```
X - Y + Z
```

Die Operatoren **–** und **+** weisen dieselbe Rangfolge auf. Aus diesem Grund wird die Operation auf der linken Seite zuerst ausgeführt.

Mit Hilfe von runden Klammern lassen sich die Rangfolgeregeln außer Kraft setzen. Ein Ausdruck in runden Klammern wird immer zuerst ausgewertet und danach wie ein einzelner Operand behandelt. Der folgende Ausdruck multipliziert *Z* mit der Summe von *X* und *Y*:

```
(X + Y) * Z
```

Manchmal sind Klammern in Situationen erforderlich, die dies auf den ersten Blick nicht vermuten lassen. Sehen Sie sich den folgenden Ausdruck an:

```
X = Y or X = Z
```

Die beabsichtigte Interpretation lautet offensichtlich

```
(X = Y) or (X = Z)
```

Wenn keine Klammern gesetzt werden, hält sich der Compiler an die Regeln für die Rangfolge der Operatoren und interpretiert den Ausdruck folgendermaßen:

```
(X = (Y or X)) = Z
```

Wenn *Z* nicht vom Typ Boolean ist, führt dies zu einem Compilierungsfehler.

Runde Klammern erleichtern zwar häufig das Schreiben und Lesen des Quelltextes, sind aber strenggenommen überflüssig. Das erste der obigen Beispiele könnte auch folgendermaßen formuliert werden:

```
X + (Y * Z)
```

Hier sind die Klammern (für den Compiler) unnötig, erleichtern aber dem Programmierer und dem Leser die Interpretation, weil die Regeln für die Operatorrangfolge nicht berücksichtigt werden müssen.

Funktionsaufrufe

Funktionsaufrufe sind Ausdrücke, da sie einen Wert zurückliefern. Wenn Sie beispielsweise eine Funktion mit dem Namen *Calc* definiert haben, die zwei Integer-Argumente übernimmt und einen Integer-Wert zurückgibt, stellt der Funktionsaufruf `Calc(24,47)` einen Integer-Ausdruck dar. Sind *I* und *J* Integer-Variablen, ist *I + Calc(J, 8)* ebenfalls ein Integer-Ausdruck. Hier einige Beispiele für Funktionsaufrufe:

```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volumen(Radius, Höhe)
GetValue
TSomeObject.SomeMethod(I,J);
```

Ausführliche Informationen über Funktionen finden Sie in Kapitel 6, »Prozeduren und Funktionen«.

Mengenkonstruktoren

Ein Mengenkonstruktor bezeichnet einen Wert eines Mengentyps, z.B.

```
[5, 6, 7, 8]
```

Dieser Mengenkonstruktur bezeichnet eine Menge mit den Ziffern 5, 6, 7 und 8. Dieselbe Menge könnte auch mit dem folgenden Mengenkonstruktor bezeichnet werden:

```
[ 5..8 ]
```

Die Syntax für einen Mengenkonstruktor lautet:

[$Element_1$, ..., $Element_n$]

Hierbei kann *Element* ein Ausdruck sein, der einen Ordinalwert des Basistyps der Menge bezeichnet, oder ein Paar solcher Ausdrücke, die durch zwei Punkte (**..**) miteinander verbunden sind. Hat ein *Element* die Form *x***..***y*, bezeichnet es alle Ordinalwerte von *x* bis *y*. Ist *x* jedoch größer als *y*, repräsentiert *x***..***y* keine Elemente, und [*x***..***y*] steht für eine leere Menge. Der Mengenkonstruktor [ ] bezeichnet die leere Menge, während [*x*] eine Menge repräsentiert, deren einiges Element der Wert *x* ist.

Hier einige Beispiele für Mengenkonstruktoren:

```
[rot, grün, Farbe]
[1, 5, 10..K mod 12, 23]
['A'..'Z', 'a'..'z', Chr(Ziffer + 48)]
```

Ausführliche Informationen über Mengen finden Sie unter »Mengentypen« auf Seite 5-17.

Indizes

Strings, Arrays, Array-Eigenschaften und Zeiger auf Strings oder Arrays können indiziert werden. Beispielswiese liefert der Ausdruck *Dateiname[3]* für die String-Variable *Dateiname* den dritten Buchstaben in dem durch *Dateiname* bezeichneten String. Dagegen gibt *Dateiname[I + 1]* das Zeichen zurück, das unmittelbar auf das mit *I* indizierte Zeichen folgt. Ausführliche Informationen über Strings finden Sie unter »String-Typen« auf Seite 5-10. Einzelheiten zu Arrays und Array-Eigenschaften finden Sie unter »Array-Typen« auf Seite 5-18 und unter »Array-Eigenschaften« auf Seite 7-19.

Typumwandlungen

In bestimmten Situationen ist es erforderlich, den Typ eines Ausdrucks zu ändern. Durch eine Typumwandlung kann einem Ausdruck vorübergehend ein anderer Typ zugeordnet werden. Beispielsweise konvertiert die Anweisung `Integer ('A')` den Buchstaben *A* in einen Integer.

Die Syntax für eine Typumwandlung lautet:

Typbezeichner(*Ausdruck*)

Handelt es sich bei dem Ausdruck um eine Variable, spricht man von einer *Variablenumwandlung*, andernfalls von einer *Wertumwandlung*. Obwohl die Syntax einer Wertumwandlung mit derjenigen einer Variablenumwandlung identisch ist, gelten für die beiden Umwandlungsarten unterschiedliche Regeln.

Wertumwandlungen

Bei einer Wertumwandlung müssen sowohl der Typbezeichner als auch der umzuwandelnde Ausdruck entweder ein ordinaler Typ oder ein Zeigertyp sein. Hier einige Beispiele für Wertumwandlungen:

```
Integer('A')
Char(48)
Boolean(0)
Color(2)
Longint(@Buffer)
```

Der resultierende Wert ergibt sich aus der Umwandlung des Ausdrucks in Klammern. Dabei wird der ursprüngliche Wert möglicherweise abgeschnitten oder erweitert, wenn die Größe des neuen Typs sich vom Typ des Ausdrucks unterscheidet. Das Vorzeichen des Ausdrucks bleibt aber in jedem Fall erhalten.

Die folgende Anweisung weist der Variablen *I* den Wert von `Integer ('A')` zu (also 65):

```
I := Integer('A');
```

Auf eine Wertumwandlung dürfen keine Qualifizierer folgen. Außerdem sind Wertumwandlungen nur auf der rechten Seite einer Zuweisung erlaubt.

Variablenumwandlungen

Variablen können in jeden beliebigen Typ umgewandelt werden. Dabei muß allerdings die Größe gleich bleiben, und Integer-Typen dürfen nicht mit Real-Typen vermischt werden (verwenden Sie zur Umwandlung numerischer Typen Standardfunktionen wie *Int* und *Trunc*). Hier einige Beispiele für Variablenumwandlungen:

```
Char(I)
Boolean(Count)
TSomeDefinedType(MyVariable)
```

Variablenumwandlungen sind auf beiden Seiten einer Zuweisung erlaubt:

```
var MyChar: char;
⋮
Shortint(MyChar) := 122;
```

Bei dieser Umwandlung wird *MyChar* das Zeichen z (ASCII 122) zugewiesen.

Variablen können in prozedurale Typen umgewandelt werden. Mit den Deklarationen

```
type Func = function(X: Integer): Integer;
var
   F: Func;
   P: Pointer;
   N: Integer;
```

sind z.B. folgende Zuweisungen möglich:

```
F := Func(P);           { Prozeduralen Wert in P an F zuweisen }
Func(P) := F;           { Prozeduralen Wert in F an P zuweisen }
@F := P;                { Zeigerwert in P an F zuweisen }
P := @F;                { Zeigerwert in F an P zuweisen }
N := F(N);              { Funktion über F aufrufen }
N := Func(P)(N);        { Funktion über P aufrufen}
```

Wie das folgende Beispiel zeigt, dürfen auf Variablenumwandlungen auch Qualifizierer folgen:

```
type
   TByteRec = record
    Lo, Hi: Byte;
   end;
   TWordRec = record
    Low, High: Word;
   end;
   PByte = ^Byte;
var
   B: Byte;
   W: Word;
   L: Longint;
   P: Pointer;
begin
   W := $1234;
   B := TByteRec(W).Lo;
   TByteRec(W).Hi := 0;
   L := $01234567;
   W := TWordRec(L).Low;
   B := TByteRec(TWordRec(L).Low).Hi;
   B := PByte(L)^;
end;
```

In diesem Beispiel wird mit *TByteRec* auf das niedrigst- und höchstwertige Byte eines Word zugegriffen. Über *TWordRec* erfolgt ein Zugriff auf das niedrigst- und höchstwertige Word eines Longint. Zu diesem Zweck könnten auch die vordefinierten Funktionen *Lo* und *Hi* verwendet werden. Die Variablenumwandlung bietet aber den Vorteil, daß sie auf der linken Seite einer Zuweisung stehen kann.

Informationen über die Typumwandlung von Zeigern finden Sie unter »Zeiger und Zeigertypen« auf Seite 5-27. Ausführliche Informationen zur Umwandlung von Klassen- und Schnittstellentypen finden Sie unter »Der Operator as« auf Seite 7-25 und unter »Schnittstellenumwandlungen« auf Seite 10.

Deklarationen und Anweisungen

Neben der **uses**-Klausel (und reservierten Wörtern zur Abgrenzung von Bereichen in Units, wie z.B. **implementation**) besteht ein Programm ausschließlich aus *Deklarationen* und *Anweisungen*, die in sogenannten *Blöcken* organisiert sind.

Deklarationen

Die Namen von Variablen, Konstanten, Typen, Feldern, Eigenschaften, Prozeduren, Funktionen, Programmen, Units, Bibliotheken und Packages werden *Bezeichner* genannt (numerische Konstanten wie 26057 sind keine Bezeichner). Damit ein Bezeichner verwendet werden kann, muß er zuerst *deklariert* werden. Ausnahmen bilden einige vordefinierte Typen, Routinen und Konstanten, die vom Compiler auch ohne

Deklaration interpretiert werden können, die Variable *Result* innerhalb eines Funktionsblocks und die Variable *Self* in einer Methodenimplementation.

Eine Deklarationen definiert den Bezeichner und sorgt dafür, daß bei Bedarf Speicher für ihn reserviert wird. Die folgende Anweisung deklariert eine Variable namens *Size*, die einen Extended-Wert (Typ Real) enthält:

```
var Size: Extended;
```

Die folgende Anweisung deklariert eine Funktion mit dem Namen *DoThis*, die zwei Strings als Argumente übernimmt und einen Integer zurückgibt:

```
function DoThis(X, Y: string): Integer;
```

Jede Deklaration wird durch einen Strichpunkt abgeschlossen. Werden mehrere Variablen, Konstanten, Typen oder Label gleichzeitig deklariert, genügt es, das entsprechende reservierte Wort nur einmal anzugeben:

```
var
   x, y: Extended;
   a, b, c: Integer;
   name, vorname: string ;
```

Die Syntax und der Ort einer Deklaration ist vom Typ des Bezeichners abhängig, der definiert werden soll. In der Regel erfolgen Deklarationen nur am Anfang eines Blocks bzw. am Anfang des **interface**- oder **implementation**-Abschnitts einer Unit (nach der **uses**-Klausel). Die Konventionen für die Deklaration von Variablen, Konstanten, Typen, Funktionen usw. werden in den jeweiligen Kapiteln der Dokumentation zu diesen Themen erläutert.

Anweisungen

Anweisungen definieren algorithmische Aktionen in einem Programm. Einfache Anweisungen (z.B. Zuweisungen und Prozeduraufrufe) können kombiniert werden. Auf diese Weise lassen sich Schleifen, bedingte Anweisungen und andere strukturierte Anweisungen erzeugen.

Mehrere Anweisungen in einem Block sowie im **initialization**- und **finalization**-Abschnitt einer Unit werden durch Strichpunkte voneinander getrennt.

Einfache Anweisungen

Einfache Anweisungen enthalten keine anderen Anweisungen. Zu ihnen gehören Zuweisungen, Aufrufe von Prozeduren und Funktionen sowie **goto**-Anweisungen.

Zuweisungen

Eine Zuweisung hat folgendes Format:

Variable := *Ausdruck*

Hierbei ist *Variable* eine beliebige Variablenreferenz, d.h. eine Variable, eine Variablenumwandlung, ein dereferenzierter Zeiger oder eine Komponente einer strukturierten Variable. *Ausdruck* kann jeder zuweisungskompatible Ausdruck sein. In ei-

nem Funktionsblock kann *Variable* durch den Namen der zu definierenden Funktion ersetzt werden. Einzelheiten finden Sie in Kapitel 6, »Prozeduren und Funktionen«. Das Symbol **:=** wird als *Zuweisungsoperator* bezeichnet.

Eine Zuweisung ersetzt den aktuellen Wert von *Variable* durch den Wert von *Ausdruck*. Die folgende Zuweisung ersetzt beispielsweise den aktuellen Wert der Variable *I* durch den Wert 3:

```
I := 3;
```

Die Variablenreferenz auf der linken Seite der Zuweisung kann auch im Ausdruck auf der rechten Seite enthalten sein:

```
I := I + 1;
```

Dadurch erhöht sich der Wert von *I* um 1. Hier noch einige weitere Beispiele für Zuweisungen:

```
X := Y + Z;
Done := (I >= 1) and  (I < 100);
Hue1 := [Blue, Succ(C)];
I := Sqr(J) - I  * K;
Shortint(MyChar) := 122;
TByteRec(W).Hi := 0;
MyString[I] := 'A';
SomeArray[I + 1] := P^;
TMyObject.SomeProperty := True;
```

Prozedur- und Funktionsaufrufe

Ein Prozeduraufruf besteht aus dem Namen einer Prozedur (mit oder ohne Qualifizierer) und (falls erforderlich) einer Parameterliste. Hier einige Beispiele für Prozeduraufrufe:

```
PrintHeading;
Transpose(A, N, M);
Find(Smith, William);
Writeln('Hallo, Welt!');
DoSomething();
Unit1.SomeProcedure;
TMyObject.SomeMethod(X,Y);
```

Wie Prozeduraufrufe können auch Funktionsaufrufe selbst als Anweisungen behandelt werden:

```
MyFunction(X);
```

Bei dieser Verwendung eines Funktionsaufrufs wird der Rückgabewert verworfen.

Ausführliche Informationen über Prozeduren und Funktionen finden Sie in Kapitel 6, »Prozeduren und Funktionen«.

Goto-Anweisungen

Die Syntax für eine **goto**-Anweisung lautet

```
goto Label
```

Eine **goto**-Anweisung setzt die Ausführung des Programms mit der Anweisung fort, die mit dem angegebenen Label markiert ist. Zur Markierung einer Anweisung deklarieren Sie zunächst das Label. Dann fügen Sie vor der zu markierenden Anweisung das Label und einen Doppelpunkt ein:

Label: *Anweisung*

Label werden folgendermaßen deklariert:

`label` *Label*;

Es können auch mehrere Label gleichzeitig deklariert werden:

`label` $Label_1, ..., Label_n$;

Als Label kann jeder gültige Bezeichner und jede Zahl zwischen 0 und 9999 verwendet werden.

Da Label keine Sprünge aus einer Prozedur bzw. Funktion in eine andere erlauben, müssen sich die Label-Deklaration, die markierte Anweisung und die **goto**-Anweisung innerhalb eines Blocks befinden (siehe »Blöcke und Gültigkeitsbereich« auf Seite 4-29). In einem Block darf jedes Label nur einmal zur Markierung einer Anweisung verwendet werden. Das folgende Beispiel führt zu einer Endlosschleife, in der immer wieder die Prozedur *Beep* aufgerufen wird:

```
label StartHere;
⋮
StartHere: Beep;
goto StartHere;
```

Im Sinne eines guten Programmierstils sollten **goto**-Anweisungen so sparsam wie möglich eingesetzt werden. In bestimmten Fällen bieten sie sich aber an, um eine verschachtelte Schleife zu verlassen. Dazu ein Beispiel:

```
procedure FindFirstAnswer;
var X, Y, Z, Count: Integer;
label FoundAnAnswer;
begin
   Count := SomeConstant;
   for X := 1 to Count do
    for Y := 1 to Count do
      for Z := 1 to Count do
        if ... { Eine Bedingung, die X, Y und Z abfragt } then
          goto FoundAnAnswer;

   ⋮ { Quelltext, der ausgeführt wird, wenn keine Antwort gefunden wird }
   Exit;

   FoundAnAnswer:
    ⋮ { Quelltext, der ausgeführt wird, wenn eine Antwort gefunden wird }
end;
```

Die **goto**-Anweisung wird hier eingesetzt, um eine verschachtelte Schleife zu *verlassen*. Das Label einer **goto**-Anweisung darf sich *nicht* innerhalb einer anderen Schleife oder strukturierten Anweisung befinden, da dies zu unerwünschten Ergebnissen führen könnte.

Strukturierte Anweisungen

Strukturierte Anweisungen sind aus anderen Anweisungen aufgebaut. Sie werden eingesetzt, wenn andere Anweisungen sequentiell, wiederholt oder in Abhängigkeit von einer Bedingung ausgeführt werden sollen.

- Eine Verbundanweisung (auch als **with**-Anweisung bezeichnet) führt eine Folge von Teilanweisungen aus.
- Eine bedingte Anweisung (also eine **if**- oder **case**-Anweisung) führt nach der Auswertung einer Bedingung mindestens eine ihrer Teilanweisungen aus.
- Schleifenanweisungen (**repeat**-, **while**- und **for**-Schleifen) führen eine Folge von Teilanweisungen mehrmals hintereinander aus.
- Ein weitere Gruppe spezieller strukturierter Anweisungen (Konstruktionen mit **raise, try...except** und **try...finally**) dienen zur Generierung und Behandlung von *Exceptions*. Einzelheiten über Exceptions finden Sie unter »Exceptions« auf Seite 7-27.

Verbundanweisungen

Eine Verbundanweisung setzt sich aus einer Folge von anderen (einfachen oder strukturierten) Anweisungen zusammen, die in der genannten Reihenfolge ausgeführt werden. Die in einer Verbundanweisung enthaltenen Teilanweisungen sind zwischen den reservierten Wörtern **begin** und **end** eingeschlossen und durch Strichpunkte voneinander getrennt. Ein Beispiel:

```
begin
   Z := X;
   X := Y;
   Y := Z;
end;
```

Der letzte Strichpunkt vor **end** ist optional. Die Verbundanweisung könnte demnach auch folgendermaßen lauten:

```
begin
   Z := X;
   X := Y;
   Y := Z
end;
```

Verbundanweisungen sind wichtig, wenn die Syntax von Object Pascal genau eine Anweisung verlangt. Sie können in Programm-, Funktions- und Prozedurblöcke und in andere strukturierte Anweisungen (z.B. bedingte Anweisungen oder Schleifen) integriert werden:

```
begin
   I := SomeConstant;
   while I > 0 do
   begin
    ⋮
    I := I - 1;
   end;
```

```
end;
```

Es gibt auch Verbundanweisungen, die nur eine einzelne Teilanweisung enthalten. Wie runde Klammen in einem komplexen Ausdruck tragen auch die Wörter **begin** und **end** zur Vermeidung von Mehrdeutigkeiten und zur Verbesserung der Lesbarkeit bei. Es ist auch möglich, mit einer leeren Verbundanweisung einen Block zu erzeugen, der keine Aktion ausführt:

```
begin
end;
```

with-Anweisungen

Eine **with**-Anweisung stellt eine einfache und bequeme Möglichkeit dar, die Felder eines Records oder die Felder, Eigenschaften und Methoden eines Objekts zu referenzieren. Die Syntax für eine **with**-Anweisung lautet

with *Objekt* do *Anweisung*

oder

with $Objekt_1$, ..., $Objekt_n$ do *Anweisung*

Hierbei ist *Objekt* eine Variablenreferenz, die ein Objekt oder einen Record bezeichnet. *Anweisung* ist eine einfache oder eine strukturierte Anweisung. Innerhalb von *Anweisung* brauchen Felder, Eigenschaften und Methoden von *Objekt* nur über ihren Bezeichner referenziert zu werden. Die Angabe von Qualifizierern ist nicht erforderlich.

Für das folgende Beispiel wird zunächst ein Typ und eine Variable deklariert:

```
type TDate = record
   Day: Integer;
   Month: Integer;
   Year: Integer;
end;
var OrderDate: TDate;
```

Eine **with**-Anweisung könnte dann folgendermaßen lauten:

```
with OrderDate do
   if Month = 12 then
   begin
    Month := 1;
    Year := Year + 1;
   end
   else
    Month := Month + 1;
```

Diese Anweisung ist gleichbedeutend mit der folgenden:

```
if OrderDate.Month = 12 then
begin
   OrderDate.Month := 1;
   OrderDate.Year := OrderDate.Year + 1;
end
else
   OrderDate.Month := OrderDate.Month + 1;
```

Wenn die Interpretation von Objekt die Indizierung von Arrays oder die Dereferenzierung von Zeigern verlangt, werden diese Aktionen einmal durchgeführt, bevor *Anweisung* ausgeführt wird. Dies macht **with**-Anweisungen sowohl effizient als auch kurz. Außerdem können sich während der aktuellen Ausführung der **with**-Anweisung Zuweisungen an eine Variable innerhalb von *Anweisung* nicht auf die Interpretation von *Objekt* auswirken.

Jede Variablenreferenz und jeder Methodenname in einer **with**-Anweisung wird, wenn möglich, als Element des angegebenen Objekts bzw. Records interpretiert. Wenn in der **with**-Anweisung auf eine andere Variable oder Methode mit demselben Namen zugegriffen werden soll, ist ein Qualifizierer erforderlich:

```
with OrderDate do
   begin
    Year := Unit1.Year
    ⋮
   end;
```

Wenn auf **with** mehrere Objekte oder Records folgen, wird die gesamte Anweisung als Folge von verschachtelten **with**-Anweisungen behandelt. Somit ist

with $Objekt_1$, $Objekt_2$, ..., $Objekt_n$ do *Anweisung*

gleichbedeutend mit

```
with Objekt1 do
   with Objekt2 do
     ⋮
     with Objektn do
       Anweisung
```

In diesem Fall wird jede Variablenreferenz und jeder Methodenname in Anweisung als Element von $Objekt_n$ behandelt, wenn dies möglich ist. Andernfalls wird die Referenz bzw. der Name als Element von $Objekt_{n-1}$ interpretiert usw. Dieselbe Regel gilt für die Interpretation der Objekte selbst. Ist beispielsweise $Objekt_n$ sowohl ein Element von $Objekt_1$ als auch von $Objekt_2$, wird es als $Objekt_2.Objekt_n$ interpretiert.

if-Anweisungen

Es gibt zwei Formen der **if**-Anweisung: **if...then** und **if...then...else**. Die Syntax einer **if...then**-Anweisung lautet folgendermaßen:

if *Ausdruck* then *Anweisung*

Das Ergebnis von *Ausdruck* ist ein Wert vom Typ Boolean. *Anweisung* wird nur ausgeführt, wenn der Ausdruck *True* ergibt. Ein Beispiel:

```
if J <> 0 then Result := I/J;
```

Die Syntax einer **if...then...else**-Anweisung lautet

if *Ausdruck* then $Anweisung_1$ else $Anweisung_2$

Das Ergebnis von *Ausdruck* ist ein Wert vom Typ Boolean. Wenn der Ausdruck *True* ergibt, wird $Anweisung_1$ ausgeführt, andernfalls $Anweisung_2$. Ein Beispiel:

```
if J = 0 then
   Exit
else
```

```
   Result := I/J;
```

Die **then-** und **else-**Klauseln enthalten jeweils nur eine Anweisung. Es kann sich hierbei auch um eine strukturierte Anweisung handeln:

```
if J <> 0 then
begin
   Result := I/J;
   Count := Count + 1;
end
else if Count = Last then
   Done := True
else
   Exit;
```

Zwischen der **then**-Klausel und dem Wort **else** darf kein Strichpunkt stehen. Nach einer vollständigen **if**-Anweisung können Sie einen Strichpunkt verwenden, um sie von der nächsten Anweisung innerhalb des Blocks zu trennen. Zwischen der **then**- und der **else**-Klausel wird nur ein Leerzeichen oder ein Wagenrücklaufzeichen benötigt. Das Einfügen eines Strichpunkts vor dem Wort **else** (in einer **if**-Anweisung) stellt einen der häufigsten Programmierfehler dar.

Auch verschachtelte **if**-Anweisungen bergen die Gefahr von Programmierfehlern. Das Problem besteht darin, daß für bestimmte **if**-Anweisungen **else**-Klauseln existieren, für andere jedoch nicht. Ansonsten ist die Syntax für die beiden **if**-Konstrukte identisch. Wenn in einer Folge von Bedingungen weniger **else**-Klauseln als **if**-Anweisungen vorhanden sind, ist möglicherweise die Zuordnung der **else**-Klauseln zu den **if**-Anweisungen nicht eindeutig erkennbar. Die Anweisung

if *Ausdruck*$_1$ then if *Ausdruck*$_2$ then *Anweisung*$_1$ else *Anweisung*$_2$;

könnte demnach auf zwei Arten interpretiert werden:

if *Ausdruck*$_1$ then [if *Ausdruck*$_2$ then *Anweisung*$_1$ else *Anweisung*$_2$];
if *Ausdruck*$_1$ then [if *Ausdruck*$_2$ then *Anweisung*$_1$] else *Anweisung*$_2$;

Der Compiler verwendet immer die erste Interpretationsart. Dies bedeutet, daß die Anweisung

```
if ... { Ausdruck1 } then
   if ... { Ausdruck2 } then
    ... { Anweisung1 }
   else
    ... { Anweisung2 } ;
```

mit diesem Konstrukt identisch ist:

```
if ... { Ausdruck1 } then
begin
   if ... { Ausdruck2 } then
    ... { Anweisung1 }
   else
    ... { Anweisung2 }
end;
```

Die Interpretation von verschachtelten Bedingungen beginnt immer bei der innersten Bedingung. Dabei wird jedes **else** dem letzten vorhergehenden **if** zugeordnet. Wenn

der Compiler die zweite Interpretationsart verwenden soll, muß der Quelltext folgendermaßen aussehen:

```
if ... { Ausdruck1 } then
begin
   if ... { Ausdruck2 } then
    ... { Anweisung1 }
end
else
   ... { Anweisung2 } ;
```

case-Anweisungen

Die **case**-Anweisung ist eine Alternative zur **if**-Anweisung, die aufgrund der besseren Lesbarkeit bei komplexen Verschachtelungen eingesetzt werden sollte. Die Syntax einer **case**-Anweisung lautet

```
case Selektor of
   CaseListe1: Anweisung1;
   ⋮
   CaseListen: Anweisungn;
end
```

Hierbei ist *Selektor* ein beliebiger Ausdruck eines ordinalen Typs (String-Typen sind nicht zulässig). Für *CaseListe* kann folgendes angegeben werden:

- Eine Zahl, eine deklarierte Konstante oder ein anderer Ausdruck, den der Compiler auswerten kann, ohne dazu das Programm selbst auszuführen. *CaseListe* muß von einem ordinalen Typ sein, der zum Typ von *Selektor* kompatibel ist. *7, True, 4 + 5 * 3, 'A'* und *Integer('A')* sind demnach als *CaseListe* zulässig, Variablen und die meisten Funktionsaufrufe dagegen nicht. Einige integrierte Funktionen wie *Hi* und *Lo* können in einer *CaseListe* enthalten sein. Weitere Einzelheiten finden Sie unter »Konstante Ausdrücke« auf Seite 5-44.
- Ein Teilbereich der Form *Erster..Letzter*, wobei *Erster* und *Letzter* den obigen Kriterien entsprechen müssen und *Erster* kleiner oder gleich *Letzter* sein muß.
- Eine Liste der Form $Element_1, ..., Element_n$, wobei jedes *Element* den obigen Kriterien entsprechen muß.

Jeder in einer *CaseListe* angegebene Wert muß innerhalb der **case**-Anweisung eindeutig sein. Teilbereiche und Listen dürfen sich nicht überschneiden. Eine **case**-Anweisung kann über eine abschließende **else**-Klausel verfügen:

```
case Selektor of
   CaseListe1: Anweisung1;
   f
   CaseListen: Anweisungn;
else
   Anweisung;
end
```

Bei der Ausführung einer **case**-Anweisung wird mindestens eine ihrer Teilanweisungen ausgeführt, und zwar diejenige, deren *CaseListe* mit dem Wert von *Selektor* identisch ist. Ist keine entsprechende *CaseListe* vorhanden, wird die Anweisung in der **else**-Klausel (falls vorhanden) ausgeführt.

Die **case**-Anweisung

```
case I of
   1..5: Caption := 'Low';
   6..9: Caption := 'High';
   0, 10..99: Caption := 'Out of range';
else
   Caption := '';
end;
```

ist mit der folgenden verschachtelten **if...then...else**-Anweisung identisch:

```
if I in [1..5] then
   Caption := 'Low'
   else if I in [6..10] then
    Caption := 'High'
    else if (I = 0) or (I in [10..99]) then
      Caption := 'Out of range'
      else
        Caption := '';
```

Hier einige weitere Beispiele für **case**-Anweisungen:

```
case MyColor of
   Red: X := 1;
   Green: X := 2;
   Blue: X := 3;
   Yellow, Orange, Black: X := 0;
end;
case Selection of
   Done: Form1.Close;
   Compute: CalculateTotal(UnitCost, Quantity);
else
   Beep;
end;
```

Schleifen

Schleifen ermöglichen die wiederholte Ausführung einer Anweisungsfolge. Eine Bedingung oder eine Variable bestimmt, wann die Ausführung angehalten wird. Object Pascal unterstützt drei Arten von Schleifen: **repeat**-, **while**- und **for**-Schleifen.

Mit den Standardprozeduren *Break* und *Continue* können Sie in den Ablauf einer **repeat-, while-** oder **for-**Schleife eingreifen. *Break* beendet die Schleife, *Continue* fährt mit der nächsten Iteration fort. Ausführliche Informationen dazu finden Sie in der Online-Hilfe.

repeat-Anweisungen

Die Syntax für eine **repeat**-Anweisung lautet

`repeat` *Anweisung*$_1$`;` `...;` *Anweisung*$_n$`;` `until` *Ausdruck*

Ausdruck gibt einen Booleschen Wert zurück. Die Angabe des letzten Strichpunkts vor **until** ist optional. Alle Anweisungen zwischen **repeat** und **until** werden der Reihe nach ausgeführt. Nach jedem Durchlauf wird der angegebene *Ausdruck* ausgewertet. Liefert *Ausdruck* den Wert *True*, wird die **repeat**-Anweisung beendet. Da *Ausdruck*

erst am Ende der ersten Iteration ausgewertet wird, wird die Anweisungsfolge mindestens einmal durchlaufen.

Hier einige Bespiele für **repeat**-Anweisungen:

```
repeat
   K := I mod J;
   I := J;
   J := K;
until J = 0;
repeat
   Write('Geben Sie einen Wert ein (0..9): ');
   Readln(I);
until (I >= 0) and (I <= 9);
```

while-Anweisungen

Eine **while**-Anweisung ähnelt in vieler Hinsicht einer **repeat**-Anweisung. Die Bedingung wird aber bereits vor der ersten Ausführung der Anweisungsfolge ausgewertet. Wenn das Ergebnis der Bedingung *False* ist, wird die Anweisungsfolge nicht ausgeführt.

Die Syntax einer **while**-Anweisung lautet

```
while Ausdruck do Anweisung
```

Ausdruck liefert einen Booleschen Wert zurück. Bei *Anweisung* kann es sich auch um eine Verbundanweisung handeln. Die **while**-Anweisung führt *Anweisung* wiederholt aus und wertet vor jedem neuen Durchlauf den angegebenen *Ausdruck* aus. Solange *Ausdruck* den Wert *True* ergibt, wird die Ausführung fortgesetzt.

Hier einige Bespiele für **while**-Anweisungen:

```
while Data[I] <> X do I := I + 1;
while I > 0 do
begin
   if Odd(I) then Z := Z * X;
   I := I div 2;
   X := Sqr(X);
end;
while not Eof(InputFile) do
begin
   Readln(InputFile, Line);
   Process(Line);
end;
```

for-Anweisungen

Im Gegensatz zur **repeat**- und **while**-Anweisung wird bei einer **for**-Anweisung angegeben, wie oft die Schleife durchlaufen werden soll. Die Syntax einer **for**-Anweisung lautet

```
for Zähler := Anfangswert to Endwert do Anweisung
```

oder

```
for Zähler := Anfangswert downto Endwert do Anweisung
```

Die Variablen haben folgende Bedeutung:

- *Zähler* ist eine lokale Variable ordinalen Typs ohne Qualifizierer. Sie wird in dem Block deklariert, in dem die **for**-Anweisung enthalten ist.
- *Anfangswert* und *Endwert* sind Ausdrücke, die zu *Zähler* zuweisungskompatibel sind.
- *Anweisung* ist eine einfache oder eine strukturierte Anweisung, die den Wert von *Zähler* nicht ändert.

Die **for**-Anweisung weist *Zähler* den angegebenen *Anfangswert* zu und führt anschließend wiederholt *Anweisung* aus. Abhängig von der verwendeten Syntax wird der Wert von *Zähler* bei jedem Durchlauf erhöht (**for...to**) oder erniedrigt (**for...downto**). Sobald der Wert von *Zähler* mit dem Wert von *Endwert* identisch ist, wird *Anweisung* ein letztes Mal ausgeführt und anschließend die **for**-Anweisung beendet. *Anweisung* wird also einmal für jeden Wert ausgeführt, der im Bereich von *Anfangswert* bis *Endwert* liegt. Ist *Anfangswert* mit *Endwert* identisch, wird *Anweisung* genau einmal ausgeführt. Wenn *Anfangswert* in einer **for...to**-Anweisung größer als *Endwert* ist, wird *Anweisung* kein einziges Mal ausgeführt. Dasselbe gilt für eine **for...downto**-Anweisung, in der *Anfangswert* kleiner als *Endwert* ist. Nach der Beendigung der **for**-Anweisung ist der Wert von *Zähler* nicht definiert.

Die Ausdrücke *Anfangswert* und *Endwert* werden zur Steuerung der Schleifenausführung nur einmal ausgewertet, und zwar vor Beginn der Schleife. Die **for...to**-Anweisung hat starke Ähnlichkeit mit dem folgenden **while**-Konstrukt:

```
begin
   Zähler := Anfangswert;
   while Zähler <= Endwert do
   begin
    Anweisung;
    Zähler := Succ(Zähler);
   end;
end
```

Im Unterschied zur **for...to-**Anweisung wird *Endwert* aber in der **while**-Schleife vor jedem Durchlauf erneut ausgewertet. Wenn *Endwert* ein komplexer Ausdruck ist, kann dies eine Verlangsamung der Ausführungsgeschwindigkeit zur Folge haben. Außerdem können sich von *Anweisung* verursachte Änderungen an *Endwert* auf die Ausführung der Schleife auswirken.

Hier einige Beispiele für **for**-Anweisungen:

```
for I := 2 to 63 do
   if Data[I] > Max then
    Max := Data[I];
for I := ListBox1.Items.Count - 1 downto 0 do
    ListBox1.Items[I] := UpperCase(ListBox1.Items[I]);
for I := 1 to 10 do
   for J := 1 to 10 do
   begin
    X := 0;
    for K := 1 to 10 do
      X := X + Mat1[I, K] * Mat2[K, J];
    Mat[I, J] := X;
```

```
  end;
for C := Red to Blue do Check(C);
```

Blöcke und Gültigkeitsbereich

Deklarationen und Anweisungen werden in *Blöcken* zusammengefaßt. Alle in einem Block deklarierten Label und Bezeichner sind für diesen Block lokal, d.h. der Block stellt den *Gültigkeitsbereich* für diese Label und Bezeichner dar. Blöcke ermöglichen es, einem Bezeichner (z.B. einem Variablennamen) in den einzelnen Programmabschnitten eine unterschiedliche Bedeutung zu geben. Ein Block ist immer Teil der Deklaration eines Programms, einer Funktion oder einer Prozedur. Umgekehrt verfügt jede Programm-, Funktions- oder Prozedurdeklaration über mindestens einen Block.

Blöcke

Ein Block besteht aus Deklarationen und einer Verbundanweisung. Alle Deklarationen müssen zusammen am Anfang des Blocks stehen. Ein Block ist folgendermaßen aufgebaut:

```
Deklarationen
begin
  Anweisungen
end
```

Der Abschnitt *Deklarationen* besteht aus Deklarationen von Variablen, Konstanten (einschließlich Ressourcen-Strings), Typen, Prozeduren, Funktionen und Labeln. Die Reihenfolge der Deklarationen ist beliebig. In einem Programmblock können im Deklarationen-Abschnitt auch eine oder mehrere **exports**-Klauseln enthalten sein (siehe Kapitel 9, »Dynamische Link-Bibliotheken und Packages«).

Hier ein Beispiel für eine Funktionsdeklaration:

```
function UpperCase(const S: string): string;
var
  Ch: Char;
  L: Integer;
  Source, Dest: PChar;
begin
  :
end;
```

Die erste Zeile bildet den Funktionskopf, die restliche Zeilen stellen den Block dar. *Ch*, *L*, *Source* und *Dest* sind lokale Variablen. Ihre Deklarationen sind nur im Funktionsblock von *UpperCase* gültig. Sie setzen in diesem Block (und nur hier) die Deklarationen aller Bezeichner gleichen Namens außer Kraft, die im Programmblock bzw. im **interface-** oder **implementation**-Abschnitt einer Unit enthalten sind.

Gültigkeitsbereich

Ein Bezeichner, z.B. ein Variablen- oder Funktionsname, kann nur innerhalb des *Gültigkeitsbereichs* seiner Deklaration verwendet werden. Der Gültigkeitsbereich einer Deklaration hängt davon ab, wo die Deklaration stattfindet. Der Gültigkeitsbereich eines Bezeichners, der in der Deklaration eines Programms, einer Funktion oder einer Prozedur deklariert ist, beschränkt sich auf den Block, der die Deklaration enthält. Der Gültigkeitsbereich der im **interface**-Abschnitt einer Unit deklarierten Bezeichner erstreckt sich über alle Units und Programme, die diese Unit einbinden. Bezeichner mit eingeschränktem Gültigkeitsbereich (besonders solche, die in Funktionen und Prozeduren deklariert sind) werden *lokale* Bezeichner genannt. Bei Bezeichnern mit großem Gültigkeitsbereich spricht man von *globalen* Bezeichnern.

Die folgende Tabelle faßt die Regeln für den Gültigkeitsbereich von Bezeichnern zusammen:

Deklarationsort	Gültigkeitsbereich
Deklaration eines Programms, einer Funktion oder einer Prozedur	Von der Deklaration bis zum Ende des aktuellen Blocks, einschließlich aller verschachtelten Blökke.
interface-Abschnitt einer Unit	Von der Deklaration bis zum Ende der Unit sowie in allen anderen Units und Programmen, die diese Unit einbinden (siehe Kapitel 3, »Programme und Units«).
implementation-Abschnitt einer Unit, aber nicht innerhalb des Blocks einer Funktion oder Prozedur	Von der Deklaration bis zum Ende des **implementation**-Abschnitts. Der Bezeichner ist für alle Funktionen und Prozeduren innerhalb des **implementation**-Abschnitts verfügbar.
Definition eines Record-Typs (d.h. der Bezeichner ist der Name eines Record-Feldes)	Von der Deklaration bis zum Ende der Feldtypdefinition (siehe »Record-Typen« auf Seite 5-23).
Definition einer Klasse (d.h. der Bezeichner ist der Name einer Eigenschaft oder Methode der Klasse)	Von der Deklaration bis zum Ende der Klassentypdefinition sowie in allen Nachkommen der Klasse und den Blöcken aller in der Klasse definierten Methoden und deren Nachkommen (siehe Kapitel 7, »Klassen und Objekte«).

Namenskonflikte

Ein Block, der einen anderen umgibt, wird als *äußerer Block* bezeichnet, der eingeschlossene Block als *innerer Block*. Ein im äußeren Block deklarierter Bezeichner kann jederzeit in einem inneren Block redeklariert werden. Die innere Deklaration setzt jedoch die äußere außer Kraft und bestimmt die Bedeutung, die der Bezeichner im inneren Block hat. Wenn beispielsweise im **interface**-Abschnitt einer Unit eine Variable mit dem Namen MaxValue deklariert ist und Sie innerhalb dieser Unit eine weitere Variable gleichen Namens in einem Funktionsblock deklarieren, beziehen sich alle nichtqualifizierten Vorkommen von MaxValue im Funktionsblock auf die lokale Deklaration. Ähnlich erzeugt eine Funktion, die innerhalb einer anderen Funktion deklariert wird, einen neuen, abgegrenzten Gültigkeitsbereich, in dem die Bezeichner der äußeren Funktion lokal redeklariert werden können.

Die Verwendung mehrerer Units kompliziert die Abgrenzung der einzelnen Gültigkeitsbereiche. Jede Unit in einer **uses**-Klausel ergibt einen neuen Gültigkeitsbereich,

der die anderen verwendeten Units und das Programm oder die Unit einschließt , die die Klausel enthält. Die erste Unit in einer **uses**-Klausel entspricht dem äußersten, die letzte Unit dem innersten Gültigkeitsbereich. Ist ein Bezeichner in den **interface**-Abschnitten mehrerer Units deklariert, bezieht sich eine nichtqualifizierte Referenz auf den Bezeichner auf die Deklaration im innersten Gültigkeitsbereich, d.h. auf die Deklaration in der Unit, in der die Referenz selbst auftritt. Ist der Bezeichner nicht in dieser Unit deklariert, bezieht sich die Referenz auf die in der letzten Unit der **uses**-Klausel deklarierte Instanz.

Die Unit *System* wird automatisch von jedem Programm und von jeder Unit verwendet. Für die in *System* enthaltenen Deklarationen sowie für die vom Compiler automatisch erkannten vordefinierten Typen, Routinen und Konstanten gilt immer der äußerste Gültigkeitsbereich.

Mit qualifizierten Bezeichnern oder mit **with**-Anweisungen können diese Regeln für den Gültigkeitsbereich außer Kraft gesetzt und lokale Deklarationen umgangen werden (siehe »Qualifizierte Bezeichner« auf Seite 4-2 und »with-Anweisungen« auf Seite 4-22).

Kapitel

5

Datentypen, Variablen und Konstanten

Ein *Typ* ist im wesentlichen ein Name für eine bestimmte Art von Daten. Wenn Sie eine Variable deklarieren, müssen Sie ihren Typ festlegen. Der Typ gibt an, welche Werte die Variable aufnehmen kann und welche Operationen mit ihr ausgeführt werden können. Alle Ausdrücke und Funktionen geben Daten eines bestimmten Typs zurück, wobei für die meisten Funktionen und Prozeduren auch Parameter eines ganz bestimmten Typs erforderlich sind.

Object Pascal ist eine streng typisierte Sprache. Sie unterscheidet zwischen einer Vielzahl unterschiedlicher Datentypen, die nicht immer durch andere Typen ersetzbar sind. Diese Einschränkung ist normalerweise von Vorteil, da sie dem Compiler eine »intelligente« Datenbehandlung und eine gründliche Überprüfung des Quelltextes erlaubt, wodurch sich die Gefahr schwer diagnostizierbarer Laufzeitfehler verringert. Wenn Sie in bestimmten Fällen mehr Flexibilität benötigen, läßt sich diese strenge Typisierung mit Hilfe besonderer Techniken umgehen. Dazu gehören die sogenannte *Typumwandlung* (siehe »Typumwandlungen« auf Seite 4-15), *Zeiger* (siehe »Zeiger und Zeigertypen« auf Seite 5-27), *Varianten* (siehe »Variante Typen« auf Seite 5-33), *variante Teile* in Record-Typen (siehe »Variante Teile in Record-Typen« auf Seite 5-24) und die *absolute Adressierung* von Variablen (siehe »Absolute Adressen« auf Seite 5-42).

Typen

Die Datentypen von Object Pascal können verschiedenen Kategorien zugeordnet werden:

- Einige Typen sind *vordefiniert* (oder *integriert*). Der Compiler erkennt diese Typen automatisch, so daß für sie keine Deklaration erforderlich ist. Die meisten der in dieser Sprachreferenz dokumentierten Typen sind vordefiniert. Andere Typen

werden über eine Deklaration erzeugt. Dazu gehören benutzerdefinierte und in Delphi-Bibliotheken definierte Typen.

- Typen sind entweder *fundamental* oder *generisch*. Der Wertebereich und das Format fundamentaler Typen ist in allen Implementationen von Object Pascal identisch, unabhängig von der zugrundeliegenden CPU und dem Betriebssystem. Der Wertebereich und das Format eines generischen Typs hingegen ist plattformspezifisch und von Implementation zu Implementation verschieden. Die meisten vordefinierten Typen sind fundamentale Typen, einige der Integer-, Zeichen-, String- und Zeigertypen sind jedoch generisch. Generell verdienen die generischen Typen den Vorzug, da sie eine optimale Ausführungsgeschwindigkeit und Portabilität gewährleisten. Wenn sich das Speicherformat eines generischen Typs jedoch in einer neuen Implementation ändert, können Kompatibilitätsprobleme (beispielsweise beim Streamen von Daten in eine Datei) auftreten.
- Typen lassen sich in *einfache*, *String-*, *strukturierte*, *Zeiger-*, *prozedurale* oder *variante* Typen einteilen. Auch Typbezeichner selbst gehören zu einem speziellen Typ, da sie als Parameter an bestimmte Funktionen (z.B. *High*, *Low* und *SizeOf*) übergeben werden können.

Die folgende Aufstellung zeigt die Struktur der Datentypen in Object Pascal.

Einfache Typen
- Ordinal
  - Integer
  - Zeichen
  - Boole
  - Aufzählung
  - Teilbereich
- Reell

String-Typen

Strukturierte Typen
- Menge
- Array
- Record
- Datei
- Klasse
- Klassenreferenz
- Schnittstelle

Zeigertypen

Prozedurale Typen

Variante Typen

Typbezeichner

Die Standardfunktion *SizeOf* kann alle Variablen und Typbezeichner verarbeiten. Sie liefert einen Integer-Wert zurück, der angibt, wieviele Bytes zum Speichern von Daten eines bestimmten Typs verwendet werden. Beispielsweise liefert `SizeOf(Longint)` den Wert 4, weil eine *Longint*-Variable im Speicher vier Byte belegt.

Typdeklarationen werden in den folgenden Abschnitten erläutert. Allgemeine Informationen zu Typdeklarationen finden Sie im Abschnitt »Typdeklaration« auf Seite 5-40.

Einfache Typen

Einfache Typen, zu denen die ordinalen und reellen Typen gehören, definieren eine Menge von Werten mit eindeutiger Reihenfolge.

Ordinale Typen

Zu den ordinalen Typen gehören Integer-, Zeichen-, Aufzählungs-, Teilbereichs- und Boolesche Typen. Ein ordinaler Typ definiert eine Menge von Werten mit eindeutiger Reihenfolge, in der jeder Wert mit Ausnahme des ersten einen eindeutigen *Vorgänger* und mit Ausnahme des letzten einen eindeutigen *Nachfolger* hat. Jeder Wert hat eine *Ordinalposition*, die seine Position in der Reihenfolge festlegt. Bei Integer-Typen ist die Ordinalposition mit dem Wert selbst identisch. Bei allen anderen ordinalen Typen, mit Ausnahme der Teilbereichstypen, hat der erste Wert die ordinale Position 0, der nächste 1 usw. Ein Wert mit der Ordinalposition *n* hat einen Vorgänger mit der Ordinalposition *n*–1 und einen Nachfolger mit der Ordinalposition *n*+1.

Einige vordefinierte Funktionen operieren mit ordinalen Werten und Typbezeichnern. Die wichtigsten dieser Funktionen sind in der folgenden Tabelle zusammengefaßt.

Funktion	Parameter	Ergebniswert	Hinweis
Ord	Ordinaler Ausdruck	Ordinalposition des Ausdruckswertes	Akzeptiert keine *Int64*-Argumente.
Pred	Ordinaler Ausdruck	Vorgänger des Ausdruckswertes	Kann nicht für Eigenschaften eingesetzt werden, die eine **write**-Prozedur verwenden.
Succ	Ordinaler Ausdruck	Nachfolger des Ausdruckswertes	Kann nicht für Eigenschaften eingesetzt werden, die eine **write**-Prozedur verwenden.
High	Ordinaler Typbezeichner oder Variable mit ordinalem Typ	Höchster Wert des Typs	Verarbeitet auch kurze String-Typen und Arrays.
Low	Ordinaler Typbezeichner oder Variable mit ordinalem Typ	Niedrigster Wert des Typs	Verarbeitet auch kurze String-Typen und Arrays.

Beispielsweise liefert `High(Byte)` den Wert 255, weil 255 der höchste Wert des Typs Byte ist. `Succ(2)` liefert 3, weil 3 der Nachfolger von 2 ist.

Die Standardprozeduren *Inc* und *Dec* erhöhen bzw. erniedrigen den Wert der ordinalen Variable. Beispielsweise ist `Inc(I)` identisch mit `I := Succ(I)` oder mit `I := I + 1`, wenn *I* eine Integer-Variable ist.

Integer-Typen

Ein Integer-Typ repräsentiert eine Untermenge der ganzen Zahlen. Es gibt zwei generische Integer-Typen: *Integer* und *Cardinal.* Diese Typen sollten, wenn möglich, immer verwendet werden, da sie die optimale Ausführungsgeschwindigkeit für die zugrundeliegende CPU und das Betriebssystem gewährleisten. Die nachfolgende Tabelle enthält die Bereiche und Speicherformate der generischen Integer-Typen für den aktuellen 32-Bit-Compiler von Object Pascal.

Tabelle 5.1 Generische Integer-Typen für 32-Bit-Implementationen von Object Pascal

Typ	Bereich	Format
Integer	-2147483648..2147483647	32 Bit mit Vorzeichen
Cardinal	0..4294967295	32 Bit ohne Vorzeichen

Zu den fundamentalen Integer-Typen gehören *Shortint, Smallint, Longint, Int64, Byte, Word* und *Longword.*

Tabelle 5.2 Fundamentale Integer-Typen

Typ	Bereich	Format
Shortint	-128..127	8 Bit, mit Vorzeichen
Smallint	-32768..32767	16 Bit, mit Vorzeichen
Longint	-2147483648..2147483647	32 Bit, mit Vorzeichen
Int64	$-2^{63}..2^{63}-1$	64 Bit, mit Vorzeichen
Byte	0..255	8 Bit, ohne Vorzeichen
Word	0..65535	16 Bit, ohne Vorzeichen
Longword	0..4294967295	32 Bit, ohne Vorzeichen

Generell gilt, daß arithmetische Operationen mit Integer-Werten einen Wert des Typs Integer zurückliefern, der in der aktuellen Implementation mit dem 32-Bit-Longint identisch ist. Operationen liefern nur dann einen Wert vom Typ Int64, wenn sie für einen Int64-Operanden ausgeführt werden. Aus diesem Grund ergibt der folgende Quelltext kein korrektes Resultat:

```
var
   I: Integer;
   J: Int64;
   ⋮
I := High(Integer);
J := I + 1;
```

Um in dieser Umgebung einen Rückgabewert vom Typ Int64 zu erhalten, muß für *I* eine Typumwandlung in Int64 ausgeführt werden:

```
   ⋮
J := Int64(I) + 1;
```

Ausführliche Informationen finden Sie im Abschnitt »Arithmetische Operatoren« auf Seite 4-7.

Hinweis Die meisten Standardroutinen mit Integer-Argumenten verkürzen Int64-Werte auf 32 Bits. Die Routinen *High*, *Low*, *Succ*, *Pred*, *Inc*, *Dec*, *IntToStr* und *IntToHex* unterstützen Int64-Argumente jedoch vollständig. Auch die Funktionen *Round*, *Trunc*, *StrToInt64* und *StrToInt64Def* liefern Int64-Werte zurück. Einige wenige Routinen (z.B. *Ord*) können keine Int64-Werte verarbeiten.

Wenn Sie den letzten Wert eines Integer-Typs erhöhen oder den ersten erniedrigen, erhalten Sie als Ergebnis den niedrigsten bzw. den höchsten Wert des Bereichs. Der Typ Shortint umfaßt beispielsweise den Bereich –128..127. Nach der Ausführung des folgenden Quelltextes hat *I* den Wert –128:

```
var I: Shortint;
   ⋮
I := High(Shortint);
I := I + 1;
```

Wenn die Bereichsprüfung des Compilers eingeschaltet ist, führt dieser Code jedoch zu einem Laufzeitfehler.

Zeichentypen

Die fundamentalen Zeichentypen sind AnsiChar und WideChar. AnsiChar-Werte stellen Zeichen mit einer Breite von einem Byte (8 Bit) dar. Ihre Reihenfolge wird durch den erweiterten ANSI-Zeichensatz festgelegt. Werte des Typs WideChar repräsentieren Zeichen mit der Breite eines Word (16 Bit). Ihre Reihenfolge ist durch den Unicode-Zeichensatz definiert. Die ersten 256 Zeichen des Unicode-Zeichensatzes entsprechen den ANSI-Zeichen.

Der generische Zeichentyp ist Char, eine Entsprechung zu AnsiChar. Die Implementation des Typs Char kann sich in zukünftigen Versionen ändern. Wenn Sie Programme schreiben, in denen Zeichen unterschiedlicher Länge verarbeitet werden, sollten Sie deshalb anstelle hart codierter Konstanten die Standardfunktion *SizeOf* verwenden.

Eine String-Konstante der Länge 1 (z.B. `'A'`) kann einen Zeichenwert darstellen. Die vordefinierte Funktion *Chr* liefert den Zeichenwert aller Integer-Werte im Bereich von AnsiChar oder WideChar. So gibt beispielsweise `Chr(65)` den Buchstaben *A* zurück.

Wie Integer-Werte liefern auch Zeichenwerte den ersten bzw. den letzten Wert im Bereich, wenn der höchste Wert erhöht oder der niedrigste erniedrigt wird. Beispielsweise hat *Letter* nach der Ausführung des folgenden Quelltextes den Wert *A* (ASCII 65):

```
var
   Letter: Char;
   I: Integer;
begin
   Letter := High(Letter);
   for I := 1 to 66 do
    Inc(Letter);
end;
```

Ausführliche Informationen über Unicode-Zeichen finden Sie in den Abschnitten »Erweiterte Zeichensätze« auf Seite 5-13 und »Nullterminierte Strings« auf Seite 5-14.

Boolesche Typen

Es gibt vier vordefinierte Boolesche Typen: Boolean, ByteBool, WordBool und LongBool. In der Praxis wird in erster Linie der Typ Boolean verwendet. Die anderen Typen dienen der Kompatibilität zu verschiedenen Sprachen und zur Windows-Umgebung.

Eine Boolean-Variable belegt ebenso wie eine ByteBool-Variable ein Byte Speicherplatz. Eine WordBool-Variable belegt zwei (ein Word) und eine LongBool-Variable vier Bytes (zwei Word).

Boolesche Werte werden mit den vordefinierten Konstanten *True* und *False* dargestellt. Dabei gelten folgende Beziehungen:

Boolean	ByteBool, WordBool, LongBool
False < *True*	*False* <> *True*
Ord(*False*) = 0	*Ord*(*False*) = 0
Ord(*True*) = 1	*Ord*(*True*) <> 0
Succ(*False*) = *True*	*Succ*(*False*) = *True*
Pred(*True*) = *False*	*Pred*(*False*) = *True*

Ein Wert vom Typ ByteBool, LongBool oder WordBool hat den Wert *True*, wenn seine ordinale Position ungleich Null ist. Tritt ein derartiger Wert in einem Kontext auf, in dem ein Wert vom Typ Boolean erwartet wird, wandelt der Compiler automatisch einen Wert mit einer Ordinalposition ungleich Null in den Wert *True* um.

Die obigen Erläuterungen beziehen sich auf die Ordinalposition von Booleschen Werten, nicht jedoch auf die Werte selbst. In Object Pascal können Boolesche Ausdrücke nicht mit Integer- oder reellen Typen verglichen werden. Wenn beispielsweise *X* eine Integer-Variable ist, führt die folgende Anweisung zu einem Compilierungsfehler:

```
if X then ...;
```

Die Umwandlung der Variable in einen Booleschen Typ ist nicht empfehlenswert. Verwenden Sie statt dessen eine der folgenden Alternativen:

```
if X <> 0 then ...;     { Längeren Ausdruck verwenden, der einen Booleschen Wert
                          liefert. }
var OK: Boolean    { Boolesche Variable verwenden. }
   ⋮
if X <> 0 then OK := True;
if OK then ...;
```

Aufzählungstypen

Aufzählungstypen definieren eine Menge von Werten mit eindeutiger Reihenfolge, indem einfach die einzelnen Bezeichner dieser Werte aneinandergereiht werden. Die Werte haben keine eigene Bedeutung. Die ordinale Struktur der Sequenz ergibt sich aus der Reihenfolge, in der die Bezeichner angeordnet sind.

Die Syntax für die Deklaration eines Aufzählungstyps lautet

```
type Typname = (Wert1, ..., Wertn)
```

Typname und *Wert* sind zulässige Bezeichner. Die folgende Deklaration definiert beispielsweise einen Aufzählungstyp namens *Suit* mit den Werten *Club, Diamond, Heart* und *Spade*:

```
type Suit = (Club, Diamond, Heart, Spade);
```

Jeder *Wert* des Aufzählungstyps wird als Konstante des Typs *Typname* deklariert. Wenn die *Wert*-Bezeichner innerhalb desselben Gültigkeitsbereichs auch für einen anderen Zweck eingesetzt werden, können Namenskonflikte auftreten. Angenommen, Sie deklarieren folgenden Typ:

```
type TSound = (Click, Clack, Clock);
```

Click ist gleichzeitig der Name einer Methode, die für die Klasse *TControl* und für alle von ihr abgeleiteten Objekte in der Delphi-VCL definiert ist. Wenn Sie eine Delphi-Anwendung entwickeln und die folgende Ereignisbehandlungsroutine erstellen, tritt ein Compilierungsfehler auf:

```
procedure TForm1.DBGrid1Enter(Sender: TObject);
var Thing: TSound;
begin
  ⋮
  Thing := Click;
  ⋮
end;
```

Der Compiler interpretiert *Click* innerhalb des Gültigkeitsbereichs der Prozedur als Referenz auf die Methode *Click* von *TForm*. Sie können dieses Problem umgehen, indem Sie den Bezeichner qualifizieren. Wenn *TSound* beispielsweise in *MyUnit* deklariert ist, lautet die korrekte Anweisung

```
Thing := MyUnit.Click;
```

Die bessere Lösung besteht aber in der Verwendung von Konstantennamen, die nicht mit anderen Bezeichnern in Konflikt stehen:

```
type
  TSound = (tsClick, tsClack, tsClock);
  TMyColor = (mcRed, mcBlue, mcGreen, mcYellow, mcOrange);
  Answer = (ansYes, ansNo, ansMaybe);
```

Sie können die Konstruktion *(Wert*$_1$*, ..., Wert*$_n$*)* wie einen Typnamen direkt in einer Variablendeklaration angeben:

```
var MyCard: (Club, Diamond, Heart, Spade);
```

Nach dieser Deklaration von *MyCard* ist es aber nicht mehr möglich, im selben Gültigkeitsbereich eine weitere Variable mit diesen Konstantenbezeichnern zu deklarieren. Die folgenden Zeilen ergeben demnach einen Compilierungsfehler:

```
var Card1: (Club, Diamond, Heart, Spade);
var Card2: (Club, Diamond, Heart, Spade);
```

Dagegen werden die folgenden Zeilen fehlerfrei compiliert:

```
var Card1, Card2: (Club, Diamond, Heart, Spade);
type Suit = (Club, Diamond, Heart, Spade);
var
   Card1: Suit;
   Card2: Suit;
```

Teilbereichstypen

Ein Teilbereich ist eine Untermenge der Werte eines anderen ordinalen Typs (des sogenannten *Basistyps*). Alle Konstruktionen der Form *Erster..Letzter*, in denen *Erster* und *Letzter* konstante Ausdrücke desselben ordinalen Typs sind und *Erster* kleiner ist als *Letzter*, bezeichnen einen Teilbereichstyp, der alle Werte von *Erster* bis *Letzter* enthält. Beispielsweise können Sie für den deklarierten Aufzählungstyp

```
type TColors = (Red, Blue, Green, Yellow, Orange, Purple, White, Black);
```

einen Teilbereichstyp der folgenden Form definieren:

```
type TMyColors = Green..White;
```

In diesem Fall umfaßt *TMyColors* die Werte *Green*, *Yellow*, *Orange*, *Purple* und *White*.

Zur Definition von Teilbereichstypen können auch numerische Konstanten und Zeichen (String-Konstanten der Länge 1) verwendet werden:

```
type
   SomeNumbers = -128..127;
   Caps = 'A'..'Z';
```

Bei der Verwendung von numerischen oder Zeichenkonstanten ist der Basistyp der kleinste Integer- oder Zeichentyp, der den angegebenen Bereich enthält.

Die Konstruktion *Erster..Letzter* funktioniert wie ein Typname, weshalb sie auch direkt in der Variablendeklaration verwendet werden kann. In der folgenden Zeile wird beispielsweise eine Integer-Variable deklariert, deren Wert im Bereich zwischen 1 und 500 liegt:

```
var SomeNum: 1..500;
```

Die Ordinalposition der Werte eines Teilbereichs wird vom Basistyp bestimmt. (Wenn im ersten der obigen Beispiele *Color* eine Variable mit dem Wert *Green* ist, liefert `Ord(Color)` den Wert 2 zurück, unabhängig davon, ob *Color* vom Typ *TColors* oder *TMyColors* ist.) Wenn Sie den letzten Wert eines Teilbereichs erhöhen oder den ersten erniedrigen, erhalten Sie als Ergebnis *nicht* den niedrigsten bzw. den höchsten Wert im Teilbereich. Dies gilt auch dann, wenn der Basistyp ein Integer- oder Zeichentyp ist. Eine Erhöhung oder Erniedrigung über die Grenzen eines Teilbereichs hinaus führt nur dazu, daß der Wert in den Basistyp umgewandelt wird. Die folgende Anweisung führt zu einem Fehler:

```
type Percentile = 0..99;
var I: Percentile;
⋮
I := 100;
```

Dagegen weist die folgende Anweisung der Variablen *I* den Wert 100 zu (außer wenn die Bereichsprüfung des Compilers eingeschaltet ist):

```
⋮
I := 99;
Inc(I);
```

Die Verwendung von konstanten Ausdrücken in Teilbereichsdefinitionen bringt ein syntaktisches Problem mit sich. Wenn in Typdeklarationen das erste bedeutungstragende Zeichen nach einem Gleichheitszeichen (=) eine öffnende Klammer ist, geht der Compiler davon aus, daß ein Aufzählungstyp definiert wird. Aus diesem Grund erzeugt der folgende Quelltext einen Fehler:

```
const
   X = 50;
   Y = 10;
type
   Scale = (X - Y) * 2..(X + Y) * 2;
```

Sie können dieses Problem umgehen, indem Sie bei der Typdeklaration die führende Klammer vermeiden:

```
type
   Scale = 2 * (X - Y)..(X + Y) * 2;
```

Reelle Typen

Ein reeller Typ definiert eine Menge von Zahlen, die in Gleitkommanotation dargestellt werden können. Die folgende Tabelle enthält die Bereiche und Speicherformate der fundamentalen reellen Typen.

Tabelle 5.3 Fundamentale reelle Typen

Typ	Bereich	Signifikante Stellen	Größe in Byte
Real48	2.9×10^{39} .. 1.7×10^{38}	11-12	6
Single	1.5×10^{45} .. 3.4×10^{38}	7-8	4
Double	5.0×10^{324} .. 1.7×10^{308}	15-16	8
Extended	3.6×10^{4951} .. 1.1×10^{4932}	19-20	10
Comp	$2^{63}+1$ .. 2^{63} 1	19-20	8
Currency	922337203685477.5808.. 922337203685477.5807	19-20	8

Der generische Typ Real ist in der aktuellen Implementation mit dem Typ Double identisch.

Tabelle 5.4 Generische reelle Typen

Typ	Bereich	Signifikante Stellen	Größe in Byte
Real	$5.0 \times 10^{-324} .. 1.7 \times 10^{308}$	15-16	8

Hinweis Der Typ Real48 (6 Byte) hatte in früheren Object-Pascal-Versionen den Namen *Real*. Wenn Sie Quelltext neu compilieren, der den alten Typ Real (6 Byte) enthält, ändern Sie diesen Typ in Real48. Die Compiler-Direktive **{$REALCOMPATIBILITY ON}** wandelt den Typ Real wieder in den alten 6-Byte-Typ um.

Die folgenden Erläuterungen beziehen sich auf die fundamentalen reellen Typen.

- *Real48* wird nur aus Gründen der Abwärtskompatibilität verwendet. Da das Speicherformat dieses Typs kein natives Format der Intel-CPU ist, laufen die entsprechenden Operationen langsamer ab als mit anderen Gleitkommatypen.
- Der Typ *Extended* bietet eine höhere Genauigkeit, ist aber nicht so einfach portierbar wie die anderen reellen Typen. Verwenden Sie Extended mit Bedacht, wenn Sie Datendateien anlegen, die gemeinsam und plattformübergreifend genutzt werden sollen.
- Der Typ *Comp* (für »computational«) ist ein natives Format der Intel-CPU und stellt einen 64-Bit-Integer dar. Er ist dennoch als reeller Typ klassifiziert, weil sein Verhalten nicht dem eines ordinalen Typs entspricht (beispielsweise läßt sich ein *Comp*-Wert weder erhöhen noch erniedrigen). *Comp* ist nur aus Gründen der Abwärtskompatibilität vorhanden. Eine höhere Ausführungsgeschwindigkeit erhalten Sie mit dem Typ *Int64*.
- Der Typ *Currency* ist ein Festkomma-Datentyp, der Rundungsfehler in finanzmathematischen Berechnungen minimiert. Er wird als skalierter 64-Bit-Integer gespeichert, bei dem die vier niedrigstwertigen Stellen implizit vier Nachkommastellen repräsentieren. Bei einer Kombination mit anderen reellen Typen in Zuweisungen und Ausdrücken werden *Currency*-Werte automatisch mit 10000 multipliziert.

String-Typen

Ein String-Typ stellt eine Folge von Zeichen dar. Object Pascal unterstützt die folgenden vordefinierten String-Typen:

Tabelle 5.5 String-Typen

Typ	Maximale Länge	Erforderlicher Speicherplatz	Verwendungszweck
ShortString	255 Zeichen	2 bis 256 Byte	Abwärtskompatibilität
AnsiString	~2^31 Zeichen	4 Byte bis 2 GB	8-Bit-Zeichen (ANSI)
WideString	~2^30 Zeichen	4 Byte bis 2 GB	Unicode-Zeichen; COM-Server und Schnittstellen

Der am häufigsten verwendete Typ ist AnsiString (auch als *langer String* bezeichnet).

String-Typen können in Zuweisungen und Ausdrücken miteinander kombiniert werden. Der Compiler führt die erforderlichen Umwandlungen automatisch durch. Strings, die als Referenz an eine Funktion oder Prozedur übergeben werden (z.B. als **var**- und **out**-Parameter), müssen jedoch den korrekten Typ aufweisen. Strings können explizit in einen anderen String-Typ umgewandelt werden (siehe »Typumwandlungen« auf Seite 4-15).

Das reservierte Wort **string** funktioniert wie ein generischer Typbezeichner. Die folgende Zeile deklariert beispielsweise eine Variable namens S, in der ein String gespeichert wird:

```
var S: string;
```

Im voreingestellten Status **{$H+**} interpretiert der Compiler **string** als AnsiString (wenn auf das reservierte Wort keine Zahl in eckigen Klammern folgt). Bei Verwendung der Direktive **{$H–}** wird **string** als ShortString interpretiert.

Die Standardfunktion *Length* gibt die Anzahl der Zeichen in einem String zurück. Mit der Prozedur *SetLength* wird die Länge eines Strings festgelegt. Ausführliche Informationen zu diesem Thema finden Sie in der Online-Hilfe.

Der Vergleich von Strings wird durch die Wertigkeit der Zeichen an den entsprechenden Positionen definiert. Bei Vergleichen zwischen Strings von unterschiedlicher Länge wird jedes Zeichen im längeren String, dem kein Zeichen im kürzeren String entspricht, als »größer« angesehen. So ist beispielsweise `'AB'` größer als `'A'`. Dies bedeutet, daß `'AB' > 'A'` den Wert *True* hat. Strings mit der Länge Null enthalten die niedrigsten Werte.

Sie können eine String-Variable wie ein Array indizieren. Wenn *S* eine String-Variable und *i* ein Integer-Ausdruck ist, stellt *S[i]* das *i*-te Zeichen in *S* dar. Bei einer ShortString- oder AnsiString-Variable ist *S[i]* vom Typ AnsiChar, bei einer WideString-Variable vom Typ *WideChar*. Die Anweisung `MyString[2] := 'A'` weist dem zweiten Zeichen von *MyString* den Wert *A* zu. Im folgenden Quelltext wird *MyString* mit der Standardfunktion *UpCase* in Großbuchstaben *umgewandelt:*

```
var I: Integer;
begin
   I := Length(MyString);
   while I > 0 do
   begin
    MyString[I] := UpCase(MyString[I]);
    I := I - 1;
   end;
end;
```

Wenn Sie Strings auf diese Weise indizieren, müssen Sie darauf achten, daß Sie nicht über das Ende des Strings hinausschreiben, da dies zu einem Indexfehler führen würde. Außerdem sollten Sie Indizes für lange Strings nicht als **var**-Parameter übergeben, da dies ineffizienten Code ergibt.

Sie können einer String-Variablen den Wert einer String-Konstante oder eines anderen Ausdrucks zuweisen, der einen String zurückliefert. Die Länge des Strings ändert sich bei der Zuweisung dynamisch. Hier einige Beispiele:

```
MyString := 'Hello world!';
MyString := 'Hello ' + 'world';
MyString := MyString + '!';
MyString := ' ';              { Leerzeichen }
MyString := '';               { Leerstring }
```

Ausführliche Informationen hierzu finden Sie in den Abschnitten »Zeichen-Strings« auf Seite 4-4 und »String-Operatoren« auf Seite 4-9.

Kurze String-Typen

Ein Wert vom Typ ShortString hat eine Länge von 0 bis 255 Zeichen. Obwohl sich die Länge eines ShortString dynamisch ändern kann, beträgt die statische Speicherplatzzuweisung immer 256 Bytes. Im ersten Byte wird die Länge des Strings gespeichert, die restlichen 255 Bytes stehen für die Zeichen zur Verfügung. Wenn *S* eine ShortString-Variable ist, liefert `Ord(S[0])` die Länge von *S* zurück (dasselbe Ergebnis erzielen Sie mit `Length(S)`). Durch Zuweisung eines Wertes an `S[0]` können Sie (wie durch einen Aufruf von *SetLength*) die Länge von *S* ändern. ShortString verwendet 8-Bit-ANSI-Zeichen und dient lediglich der Abwärtskompatibilität.

Object Pascal unterstützt kurze String-Typen (Untertypen von ShortString), deren maximale Länge zwischen 0 und 255 Zeichen liegen kann. Diese Typen werden mit einer Zahl in eckigen Klammern dargestellt, die auf das reservierte Wort **string** folgt. Die folgende Zeile deklariert beispielsweise eine Variable namens *MyString*, deren Länge maximal 100 Zeichen beträgt:

```
var MyString: string[100];
```

Die folgenden Deklarationen sind mit der obigen Zeile identisch:

```
type CString = string[100];
var MyString: CString;
```

Bei Variablen, die auf diese Weise deklariert werden, wird dem Typ nur soviel Speicherplatz zugewiesen, wie für die angegeben Länge plus ein Byte erforderlich ist. Im obigen Beispiel belegt *MyString* 101 Bytes. Für eine Variable des vordefinierten Typs ShortString wären dagegen 256 Bytes erforderlich.

Bei einer Wertzuweisung an eine kurze String-Variable wird der String abgeschnitten, wenn die maximale Länge für den Typ überschritten wird.

Die Standardfunktionen *High* und *Low* bearbeiten Variablen und Typbezeichner für kurze Strings. *High* liefert die maximale Länge des kurzen String-Typs, während *Low* Null zurückgibt.

Lange String-Typen

Der Typ AnsiString (auch als langer String bezeichnet) stellt einen dynamisch zugewiesenen String dar, dessen maximale Länge nur durch den verfügbaren Speicherplatz begrenzt wird. Der Typ verwendet 8-Bit-ANSI-Zeichen.

Eine AnsiString-Variable ist ein Zeiger, der vier Bytes Speicherplatz belegt. Wenn die Variable leer ist (d.h. wenn sie einen String der Länge Null enthält), hat der Zeiger

den Wert **nil**, und der String belegt keinen Speicherplatz. Ist die Variable nicht leer, zeigt sie auf einen dynamisch zugewiesenen Speicherblock, der neben dem String-Wert eine Längenangabe und einen Referenzzähler von je 32 Bit enthält. Da dieser Speicherplatz auf dem Heap reserviert und vollkommen automatisch verwaltet wird, erfordert er keinerlei Benutzercode.

Da es sich bei AnsiString-Variablen um Zeiger handelt, können zwei oder mehrere dieser Variablen auf denselben Wert zeigen, ohne zusätzlichen Speicherplatz zu belegen. Der Compiler nützt dies zur Einsparung von Ressourcen. Auch Zuweisungen werden schneller ausgeführt. Sobald eine AnsiString-Variable freigegeben oder mit einem neuen Wert belegt wird, wird der Referenzzähler des alten Strings (d.h. des vorhergehenden Wertes der Variable) erniedrigt und der Referenzzähler des neuen Wertes (falls ein solcher zugewiesen wurde) erhöht. Erreicht der Referenzzähler eines Strings den Wert Null, wird der belegte Speicherplatz freigegeben. Dieser Vorgang wird als *Referenzzählung* bezeichnet. Wenn der Wert eines einzelnen Zeichens im String über einen Index geändert werden soll, wird eine Kopie des Strings angelegt. Dies ist aber nur möglich, wenn der betreffende Referenzzähler größer als 1 ist. Diesen Vorgang nennt man *Copy-on-Write*-Semantik.

WideString-Typen

Der Typ WideString repräsentiert einen dynamisch zugewiesenen String mit 16-Bit-Unicode-Zeichen. Dieser Typ ähnelt in vielerlei Hinsicht dem Typ AnsiString. Er ist jedoch weniger effizient, weil er die Referenzzählung und die Copy-on-Write-Semantik nicht unterstützt.

WideString ist mit dem COM-Typ BSTR kompatibel. Delphi verfügt über Eigenschaften zur COM-Unterstützung, die AnsiString-Werte in WideString-Werte umwandeln. Wenn Sie Funktionen der COM-API aufrufen, müssen Sie vorhandene Strings explizit in den Typ WideString umwandeln.

Erweiterte Zeichensätze

Windows unterstützt *Einzelbyte*- und *Multibyte*-Zeichensätze sowie den *Unicode*-Zeichensatz. Bei einem Einzelbyte-Zeichensatz (SBCS = Single-Byte Character Set) repräsentiert jedes Byte eines Strings ein Zeichen. Ein Beispiel für einen Einzelbyte-Zeichensatz ist der ANSI-Zeichensatz, der in den meisten Windows-Versionen der westlichen Welt verwendet wird.

In einem Multibyte-Zeichensatz (MBCS = Multi-Byte Character Set) werden einige Zeichen mit einem einzelnen Byte und andere mit mehreren Bytes dargestellt. Das erste Byte eines Multibyte-Zeichens wird als *führendes Byte* bezeichnet. Im allgemeinen stimmen die ersten 128 Zeichen eines Multibyte-Zeichensatzes mit den 7-Bit-ASCII-Zeichen überein. Jedes Byte, dessen Ordinalwert größer ist als 127, fungiert als führendes Byte eines Multibyte-Zeichens. Multibyte-Zeichen können im Gegensatz zu Einzelbyte-Zeichen nicht den Wert Null (#0) enthalten. Multibyte-Zeichensätze, insbesondere Doppelbyte-Zeichensätze (DBCS = Double-Byte Character Set), werden in erster Linie für asiatische Sprachen verwendet.

Im Unicode-Zeichensatz wird jedes Zeichen mit zwei Bytes dargestellt. Ein Unicode-String ist demnach eine Folge von Words und nicht von einzelnen Bytes. Unicode-Zeichen und -Strings werden auch als *Wide-Zeichen* bzw. *Wide-Strings* bezeichnet. Die ersten 256 Unicode-Zeichen stimmen mit dem *ANSI-Zeichensatz* überein.

Über die Typen Char, PChar, AnsiChar, PAnsiChar und AnsiString unterstützt Object Pascal sowohl Einzelbyte- als auch Multibyte-Zeichen und -Strings. Für alle Delphi-Standardfunktionen zur Stringverarbeitung existieren multibytefähige Entsprechungen, die auch die Besonderheiten länderspezifischer Zeichensätze berücksichtigen. Die Namen von Multibyte-Funktionen beginnen normalerweise mit dem Wort *Ansi*. Beispielsweise trägt die Multibyte-Version von *StrPos* den Namen *AnsiStrPos*. Die Unterstützung von Multibyte-Zeichen hängt vom Betriebssystem ab und basiert auf dem aktuell installierten Sprachtreiber.

Object Pascal unterstützt Unicode-Zeichen und -Strings über die Typen WideChar, PWideChar und WideString.

Nullterminierte Strings

In vielen Programmiersprachen, z.B. in C und C++, fehlt ein spezieller String-Datentyp. Diese Sprachen und die mit ihnen programmierten Umgebungen (z.B. Windows) verwenden sogenannte *nullterminierte Strings*. Ein nullterminierter String ist ein Zeichen-Array, dessen Index bei 0 beginnt und das mit einer NULL (#0) endet. Da das Array keine Längenangabe hat, wird das Ende des Strings durch das erste NULL-Zeichen markiert. Die Verarbeitung nullterminierter Strings erfolgt in Object Pascal mit Hilfe von Routinen, die sich in der Unit *SysUtils* befinden (siehe Kapitel 8, »Standardroutinen und E/A«). Dies ist beispielsweise erforderlich, wenn Sie Daten gemeinsam mit Systemen benutzen, die nullterminierte Strings verwenden.

Hier einige Beispiele für die Deklaration von Typen, die nullterminierte Strings speichern können:

```
type
  TIdentifier = array[0..15] of Char;
  TFileName = array[0..259] of Char;
  TMemoText = array[0..1023] of WideChar;
```

Sie können einem statisch zugewiesenen Zeichen-Array, dessen Index bei 0 beginnt, eine String-Konstante zuweisen (mit einem dynamischen Array ist dies nicht möglich). Wenn Sie eine Array-Konstante mit einem String initialisieren, der kürzer als die deklarierte Länge des Arrays ist, werden die verbleibenden Zeichen auf #0 gesetzt. Ausführliche Informationen über Arrays finden Sie im Abschnitt »Array-Typen« auf Seite 5-18.

Zeiger, Arrays und String-Konstanten

Bei der Bearbeitung von nullterminierten Strings ist häufig der Einsatz von Zeigern erforderlich (siehe »Zeiger und Zeigertypen« auf Seite 5-27). String-Konstanten sind zuweisungskompatibel zu den Typen PChar und PWideChar, die Zeiger auf nullterminierte Arrays mit Char- und WideChar-Werten darstellen. Im folgenden Beispiel

zeigt *P* auf einen Speicherbereich, der eine nullterminierte Kopie von `Hello world!` enthält:

```
var P: PChar;
   ⋮
P := 'Hello world!';
```

Die folgenden Programmzeilen sind mit den obigen Zeilen identisch:

```
const TempString: array[0..12] of Char = 'Hello world!'#0;
var P: PChar;
   ⋮
P := @TempString;
```

Sie können String-Konstanten auch an Funktionen übergeben, die Wert- oder **const**-Parameter des Typs PChar oder PWideChar akzeptieren, z.B. `StrUpper('Hello world!')`. Der Compiler erzeugt (wie bei Zuweisungen an PChar) eine nullterminierte Kopie des Strings und übergibt der Funktion einen Zeiger auf diese Kopie. Außerdem können Sie PChar- und PWideChar-Konstanten einzeln oder in strukturierten Typen mit String-Literalen initialisieren. Hier einige Beispiele:

```
const
Message: PChar = 'Program terminated';
   Prompt: PChar = 'Enter values: ';
   Digits: array[0..9] of PChar = (
    'Zero', 'One', 'Two', 'Three', 'Four',
    'Five', 'Six', 'Seven', 'Eight', 'Nine');
```

Zeichen-Arrays, deren Index bei 0 beginnt, sind mit PChar und PWideChar kompatibel. Wenn Sie anstelle eines Zeigerwertes ein Zeichen-Array verwenden, wandelt der Compiler das Array in eine Zeiger-Konstante um, deren Wert der Adresse des ersten Elements im Array entspricht. Ein Beispiel:

```
var
   MyArray: array[0..32] of Char;
   MyPointer: PChar;
begin
   MyArray := 'Hello';
   MyPointer := MyArray;
   SomeProcedure(MyArray);
   SomeProcedure(MyPointer);
end;
```

Dieser Programmcode ruft *SomeProcedure* zweimal mit demselben Wert auf.

Ein Zeichenzeiger kann wie ein Array indiziert werden. Im obigen Beispiel liefert `MyPointer[0]` den Wert *H*. Der Index legt einen Offset fest, der dem Zeiger vor der Dereferenzierung hinzuaddiert wird (bei PWideChar-Variablen wird der Index automatisch mit zwei multipliziert). Wenn es sich bei *P* um einen Zeichenzeiger handelt, ist *P*`[0]` identisch mit *P*^ und bezeichnet das erste Zeichen im Array, *P*`[1]` das zweite Zeichen usw. *P*`[-1]` bezeichnet das »Zeichen«, das unmittelbar links neben *P*`[0]` steht. Der Compiler führt für diese Indizes aber *keine* Bereichsprüfung durch.

Das folgende Beispiel zeigt anhand der Funktion *StrUpper*, wie unter Verwendung der Zeigerindizierung ein nullterminierter String durchlaufen wird:

```
function StrUpper(Dest, Source: PChar; MaxLen: Integer): PChar;
var
   I: Integer;
begin
   I := 0;
   while (I < MaxLen) and (Source[I] <> #0) do
   begin
    Dest[I] := UpCase(Source[I]);
    Inc(I);
   end;
   Dest[I] := #0;
   Result := Dest;
end;
```

Kombination von Pascal-Strings und nullterminierten Strings

Lange Strings (AnsiString-Werte) und nullterminierte Strings (PChar-Werte) lassen sich in Ausdrücken und Zuweisungen kombinieren. Außerdem können PChar-Werte an Funktionen und Prozeduren übergeben werden, die AnsiString-Parameter akzeptieren. Die Zuweisung *S := P*, in der *S* eine String-Variable und *P* ein PChar-Ausdruck ist, kopiert einen nullterminierten String in einen langen String.

Wenn in einer binären Operation der eine Operand ein langer String und der andere ein PChar-Ausdruck ist, wird der PChar-Operand in einen langen String umgewandelt.

Sie können einen PChar-Wert als langen String verwenden. Dies kann beispielsweise erforderlich sein, wenn eine String-Operation für zwei PChar-Werte durchgeführt werden soll:

```
S := string(P1) + string(P2);
```

Es ist auch möglich, einen langen String in einen nullterminierten String umzuwandeln. In diesem Fall gelten folgende Regeln:

- Wenn *S* ein AnsiString-Ausdruck ist, wird *S* mit `PChar(S)` in einen nullterminierten String umgewandelt. Das Ergebnis ist ein Zeiger auf das erste Zeichen in *S*. Wenn beispielsweise *Str1* und *Str2* lange Strings sind, kann die Funktion *MessageBox* der Win32-API folgendermaßen aufgerufen werden:

  ```
  MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);
  ```

  MessageBox ist in der Unit *Windows* deklariert.

- Außerdem haben Sie die Möglichkeit, mit `Pointer(S)` einen langen String in einen untypisierten Zeiger umzuwandeln. Wenn *S* allerdings leer ist, ergibt die Umwandlung **nil**.

- Wenn Sie eine AnsiString-Variable in einen Zeiger konvertieren, bleibt dieser gültig, bis die Variable den Gültigkeitsbereich verläßt oder ihr ein neuer Wert zugewiesen wird. Wenn Sie einen beliebigen anderen AnsiString-Ausdruck in einen Zeiger umwandeln, ist der Zeiger nur innerhalb der Anweisung gültig, in der die Typumwandlung durchgeführt wird.

- Nach der Konvertierung eines AnsiString-Ausdrucks in einen Zeiger sollten Sie den Zeiger als schreibgeschützt ansehen. Der String kann über den Zeiger nur dann gefahrlos geändert werden, wenn folgende Bedingungen erfüllt sind:
  - Der Ausdruck, der umgewandelt werden soll, ist eine AnsiString-*Variable*.
  - Der String ist nicht leer.
  - Der String ist eindeutig, d.h. der Referenzzähler hat den Wert 1. Um sicherzustellen, daß der String eindeutig ist, rufen Sie eine der Prozeduren *SetLength*, *SetString* oder *UniqueString* auf.
  - Der String wurde seit der letzten Typumwandlung nicht geändert.
  - Die zu ändernden Zeichen befinden sich alle innerhalb des Strings. Sie dürfen für den Zeiger auf keinen Fall einen Index verwenden, der außerhalb des Bereichs liegt.

Diese Regeln gelten auch, wenn Sie WideString- mit PWideChar-Werten kombinieren.

Strukturierte Typen

Die Instanzen eines strukturierten Typs enthalten mehrere Werte. Zu den strukturierten Typen gehören Mengen-, Array-, Record- und Datei-, Klassen-, Klassenreferenz- und Schnittstellentypen. Informationen über Klassen- und Klassenreferenztypen finden Sie in Kapitel 7, »Klassen und Objekte«. Informationen zu Schnittstellen enthält Kapitel 10, »Objektschnittstellen«. Mit Ausnahme von Mengen, die nur ordinale Werte enthalten, können strukturierte Typen auch andere strukturierte Typen beinhalten. Ein Typ kann beliebig viele strukturelle Ebenen umfassen.

Per Voreinstellung sind die Werte in einem strukturierten Typ in einem Word- oder Double-Word-Raster ausgerichtet, um den Zugriff zu beschleunigen. Wenn Sie einen strukturierten Typ deklarieren, können Sie das reservierte Wort **packed** einfügen, um die Daten in komprimierter Form zu speichern:

```
type TNumbers = packed array[1..100] of Real;
```

Die Verwendung **packed** verlangsamt den Zugriff auf die Daten. Im Falle eines Zeichen-Arrays beeinflußt **packed** auch die Kompatibilität der Typen. Ausführliche Informationen hierzu finden sie in Kapitel 11, »Speicherverwaltung«.

Mengentypen

Eine Menge setzt sich aus mehreren Werten desselben ordinalen Typs zusammen. Die Werte haben keine feste Reihenfolge. Wenn ein Wert in einer Menge doppelt vorkommt, hat jedes Vorkommen dieselbe Bedeutung.

Der Bereich eines Mengentyps ist die Potenzmenge eines bestimmten Ordinaltyps, der als *Basistyp* bezeichnet wird. Die möglichen Werte eines Mengentyps sind Teilmengen des Basistyps, einschließlich der leeren Menge. Der Basistyp darf aus maxi-

mal 256 Werten bestehen. Die Ordinalpositionen der Werte müssen zwischen 0 und 255 liegen. Alle Konstruktionen der Form

```
set of Basistyp
```

bezeichnen einen Mengentyp. Dabei ist *Basistyp* ein entsprechender ordinaler Typ.

Aufrund der Größenbeschränkung von Basistypen werden Mengentypen normalerweise mit Teilmengen definiert. Mit den folgenden Deklarationen wird beispielsweise ein Mengentyp namens *TIntSet* angelegt, dessen Werte Integer-Zahlen im Bereich zwischen 1 und 250 sind:

```
type
   TSomeInts = 1..250;
   TIntSet = set of TSomeInts;
```

Dasselbe Ergebnis erreichen Sie mit der folgenden Zeile:

```
type TIntSet = set of 1..250;
```

Mit dieser Deklaration können Sie beispielsweise folgende Mengen erstellen:

```
var Set1, Set2: TIntSet;
   ⋮
Set1 := [1, 3, 5, 7, 9];
Set2 := [2, 4, 6, 8, 10]
```

Die Konstruktion **set of ...** kann direkt in Variablendeklarationen verwendet werden:

```
var MySet: set of 'a'..'z';
   ⋮
MySet := ['a','b','c'];
```

Hier einige weitere Beispiele für Mengentypen:

```
set of Byte
set of (Club, Diamond, Heart, Spade)
set of Char;
```

Der Operator **in** überprüft, ob ein Element zu einer Menge gehört:

```
if 'a' in MySet then ... { Aktionen } ;
```

Jeder Mengentyp kann die leere Menge enthalten, die mit `[]` gekennzeichnet wird. Ausführliche Informationen über Mengen finden Sie in den Abschnitten »Mengenkonstruktoren« auf Seite 4-14 und »Mengenoperatoren« auf Seite 4-11.

Array-Typen

Ein Array ist eine indizierte Menge von Elementen desselben Typs (des sogenannten Basistyps). Da jedes Element einen eindeutigen Index hat, kann ein Array (im Gegensatz zu einer Menge) denselben Wert mehrmals und mit unterschiedlicher Bedeutung enthalten. Arrays können *statisch* oder *dynamisch* zugewiesen werden.

Statische Arrays

Statische Array-Typen werden mit der folgenden Konstruktion definiert:

```
array[Indextyp1, ..., Indextypn] of Basistyp
```

Indextyp ist immer ein ordinaler Typ, dessen Bereich 2 GB nicht überschreitet. Da das Array über den *Indextyp* indiziert wird, ist die Anzahl der Elemente durch den angegebenen *Indextyp* beschränkt. In der Praxis sind die *Indextyp*en normalerweise Integer-Teilbereiche.

Im einfachsten Fall eines eindimensionalen Arrays ist nur ein einziger *Indextyp* vorhanden. Beispielsweise wird in der folgenden Zeile eine Variable namens *MyArray* deklariert, die ein Array mit 100 Zeichenwerten umfaßt:

```
var MyArray: array[1..100] of Char;
```

Aufgrund dieser Deklaration bezeichnet `MyArray[3]` das dritte Zeichen in *MyArray*. Wenn Sie ein statisches Array anlegen, in dem nicht allen Elementen einen Wert zugewiesen ist, wird auch für die nicht verwendeten Elemente Speicherplatz reserviert. Diese Elemente enthalten zu Beginn zufällige Daten und sind mit nicht initialisierten Variablen vergleichbar.

Ein mehrdimensionales Array ist ein Array, das andere Arrays enthält. Die Anweisung

```
type TMatrix = array[1..10] of array[1..50] of Real;
```

ist gleichbedeutend mit

```
type TMatrix = array[1..10, 1..50] of Real;
```

Unabhängig von der Art der Deklaration repräsentiert *TMatrix* immer ein Array mit 500 reellen Werten. Eine Variable namens *MyMatrix* vom Typ *TMatrix* kann auf zwei Arten indiziert werden: `MyMatrix[2,45]` oder `MyMatrix[2][45]`. Die Anweisung

```
packed array[Boolean,1..10,TShoeSize] of Integer;
```

ist also gleichbedeutend mit

```
packed array[Boolean] of packed array[1..10] of packed array[TShoeSize] of Integer;
```

Die Standardfunktionen *Low* und *High* können auf Array-Typbezeichner und Variablen angewendet werden. Sie liefern die untere und obere Grenze des ersten Indextyps im Array. Die Standardfunktion *Length* liefert die Anzahl der Elemente in der ersten Dimension des Arrays zurück.

Ein eindimensionales, gepacktes, statisches Array mit Char-Werten wird als *gepackter* String bezeichnet. Gepackte String-Typen sind mit String-Typen und anderen gepackten String-Typen kompatibel, die dieselbe Anzahl von Elementen haben. Informationen hierzu finden Sie im Abschnitt »Kompatibilität und Identität von Typen« auf Seite 5-37.

Ein Array der Form `array[0..x] of Char` wird als *nullbasiertes Zeichen-Array* bezeichnet, da sein Index bei 0 beginnt. Nullbasierte Zeichen-Arrays werden zum Speichern von nullterminierten Strings verwendet und sind kompatibel mit PChar-Werten. Informationen hierzu finden Sie unter »Nullterminierte Strings« auf Seite 5-14.

Dynamische Arrays

Dynamische Arrays haben keine feste Größe oder Länge. Der Speicher für ein dynamisches Array wird reserviert, sobald Sie dem Array ein Wert zuweisen oder es an die Prozedur *SetLength* übergeben. Dynamische Array-Typen werden folgendermaßen deklariert:

```
array of Basistyp
```

Das folgende Beispiel deklariert ein eindimensionales Array mit Elementen vom Typ Real:

```
var MyFlexibleArray: array of Real;
```

Diese Deklaration weist *MyFlexibleArray* keinen Speicherplatz zu. Um ein Array im Speicher anzulegen, rufen Sie *SetLength* auf. Ausgehend von der obigen Deklaration weist die folgende Zeile einem Array mit 20 reellen Zahlen und einem Index von 0 bis 19 Speicherplatz zu:

```
SetLength(MyFlexibleArray, 20);
```

Dynamische Arrays haben immer einen Integer-Index, der bei 0 beginnt.

Dynamische Array-Variablen sind implizit Zeiger und werden mit derselben Referenzzählung verwaltet wie lange Strings. Um ein dynamisches Array freizugeben, weisen Sie einer Variablen, die das Array referenziert, den Wert **nil** zu, oder Sie übergeben die Variable an *Finalize*. Beide Methoden geben das Array unter der Voraussetzung frei, daß keine weiteren Referenzen darauf vorhanden sind. Wenden Sie auf dynamische Array-Variablen nicht den Derefenzierungsoperator (^) an, und übergeben Sie sie auch nicht an die Prozeduren *New* oder *Dispose*.

Wenn *X* und *Y* Variablen desselben dynamischen Array-Typs sind, setzt die Anweisung *X* := *Y* die Variable *X* auf die Größe von *Y*, und *X* zeigt anschließend auf dasselbe Array wie *Y*. Im Gegensatz zu Strings und statischen Arrays werden dynamische Arrays nämlich nicht automatisch kopiert, bevor einem ihrer Elemente ein Wert zugewiesen wird. Beispielsweise hat `A[0]` nach der Ausführung des folgenden Quelltextes den Wert *2*:

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := A;
  B[0] := 2;
end;
```

Wenn *A* und *B* statische Arrays wären, hätte `A[0]` immer noch den Wert 1.

Durch Zuweisungen an ein dynamisches Array über den Index (wie beispielsweise `MyFlexibleArray[2] := 7`) wird für das Array kein neuer Speicherplatz reserviert. Der Compiler akzeptiert auch Indizes, die außerhalb des angegebenen Bereichs liegen.

Bei einem Vergleich von dynamischen Array-Variablen werden nicht die Array-Werte, sondern die Referenzen verglichen. Deshalb liefert `A = B` nach Ausführung des folgenden Quelltextes den Wert *False*, während `A[0] = B[0]` *True* zurückgibt:

```
var
   A, B: array of Integer;
begin
   SetLength(A, 1);
   SetLength(B, 1);
   A[0] := 2;
   B[0] := 2;
end;
```

Um ein dynamisches Array abzuschneiden, übergeben Sie es an die Funktion *Copy* und weisen das Ergebnis wieder der Array-Variablen zu. Wenn beispielsweise *A* ein dynamisches Array ist, können Sie mit der Anweisung `A := Copy(A, 0, 20)` die ersten 20 Elemente von *A* beibehalten und den Rest abschneiden.

Sobald einem dynamischen Array Speicherplatz zugewiesen wurde, kann es an die Standardfunktionen *Length*, *High* und *Low* übergeben werden. *Length* liefert die Anzahl der Elemente im Array, *High* den höchsten Index des Arrays (*Length*–1) und *Low* den Wert 0. Bei einem Array mit der Länge Null liefert *High* das Ergebnis –1 (mit der unsinnigen Folge, daß *High* kleiner als *Low* ist).

Hinweis In einigen Funktions- und Prozedurdeklarationen werden Array-Parameter in der Form **array of** *Basistyp* ohne festgelegten *Indextyp* angegeben:

```
function CheckStrings(A: array of string): Boolean;
```

In diesem Fall kann die Funktion auf alle Arrays des angegebenen Basistyps angewendet werden, unabhängig von der Größe der Arrays und der Art ihrer Indizierung. Es spielt auch keine Rolle, ob den Arrays der Speicherplatz statisch oder dynamisch zugewiesen wird. Weitere Informationen hierzu finden Sie im Abschnitt »Offene Array-Parameter« auf Seite 6-14.

Mehrdimensionale dynamische Arrays

Zur Deklaration von mehrdimensionalen dynamischen Arrays verwenden Sie aufeinanderfolgende **array of ...**-Konstruktionen. Die beiden folgenden Zeilen deklarieren ein zweidimensionales String-Array:

```
type TMessageGrid = array of array of string;
var Msgs: TMessageGrid;
```

Um dieses Array zu instantiieren, rufen Sie *SetLength* mit zwei Integer-Argumenten auf. Wenn beispielsweise *I* und *J* Integer-Variablen sind, wird Speicherplatz für ein *I* mal *J* großes Array zugewiesen. `Msgs[0,0]` bezeichnet dann ein Element dieses Arrays:

```
SetLength(Msgs,I,J);
```

Sie können auch mehrdimensionale dynamische Arrays anlegen, die nicht gleichförmig sind. Rufen Sie dazu als erstes die Funktion *SetLength* auf, und übergeben Sie ihr Parameter für die ersten *n* Dimensionen des Arrays:

```
var Ints: array of array of Integer;
SetLength(Ints,10);
```

Mit dieser Anweisung weisen Sie dem Array *Ints* Speicherplatz für zehn Zeilen zu. Den Speicher für die Spalten können Sie später einzeln zuweisen (und dabei unterschiedliche Längen angeben):

```
SetLength(Ints[2], 5);
```

Die dritte Spalte von *Ints* kann damit fünf Integer-Werte aufnehmen, und Sie können ihr von nun an Werte zuweisen, z.B. `Ints[2,4] := 6`. Dies ist auch dann möglich, wenn den anderen Spalten noch kein Speicherplatz zugewiesen wurde.

Im folgenden Beispiel wird mit Hilfe von dynamischen Arrays (und der in der Unit *SysUtils* deklarierten Funktion *IntToStr*) eine ungleichförmige String-Matrix erstellt:

```
var
   A : array of array of string;
   I, J : Integer;
begin
   SetLength(A, 10);
   for I := Low(A) to High(A) do
   begin
    SetLength(A[I], I);
    for J := Low(A[I]) to High(A[I]) do
      A[I,J] := IntToStr(I) + ',' + IntToStr(J) + ' ';
   end;
end;
```

Array-Typen und Zuweisungen

Arrays sind nur dann zuweisungskompatibel, wenn sie denselben Typ haben. Da Pascal Namensäquivalente für Typen verwendet, wird folgender Quelltext nicht compiliert:

```
var
   Int1: array[1..10] of Integer;
   Int2: array[1..10] of Integer;
   ⋮
Int1 := Int2;
```

Damit die Zuweisung korrekt bearbeitet werden kann, deklarieren Sie die Variablen folgendermaßen:

```
var Int1, Int2: array[1..10] of Integer;
```

oder

```
type IntArray = array[1..10] of Integer;
var
   Int1: IntArray;
   Int2: IntArray;
```

Record-Typen

Ein Record (in einigen Programmiersprachen auch als *Struktur* bezeichnet) stellt eine heterogene Menge von Elementen dar. Die Elemente werden *Felder* genannt. In der Deklaration eines Record-Typs wird für jedes Feld ein Name und ein Typ festgelegt. Die Syntax für die Deklaration eines Record-Typs lautet

```
type Recordtypname = record
   Feldliste1: Typ1;
   ⋮
   Feldlisten: Typn;
end
```

Recordtypname ist ein gültiger Bezeichner, *Typ* gibt einen Typ an, und *Feldliste* ist ein gültiger Bezeichner oder eine Liste von Bezeichnern, die durch Kommas voneinander getrennt sind. Der letzte Strichpunkt ist optional.

Die folgende Deklaration legt einen Record-Typ namens *TDateRec* an:

```
type
   TDateRec = record
    Year: Integer;
    Month: (Jan, Feb, Mar, Apr, May, Jun,
            Jul, Aug, Sep, Oct, Nov, Dec);
    Day: 1..31;
   end;
```

TDateRec enthält immer drei Felder: einen Integer-Wert namens *Year*, einen Aufzählungswert namens *Month* und einen weiteren Integer-Wert zwischen 1 und 31 namens *Day*. Die Bezeichner *Year*, *Month* und *Day* sind *Feldbezeichner* für *TDateRec* und verhalten sich wie Variablen. Die Typdeklaration für *TDateRec* weist den Feldern *Year*, *Month* und *Day* aber keinen Speicherplatz zu. Die Reservierung des Speichers erfolgt erst, wenn der Record instantiiert wird:

```
var Record1, Record2: TDateRec;
```

Diese Variablendeklaration erzeugt zwei Instanzen von *TDateRec* namens *Record1* und *Record2*.

Sie können auf die Felder eines Records zugreifen, indem Sie die Feldbezeichner mit dem Record-Namen qualifizieren:

```
Record1.Year := 1904;
Record1.Month := Jun;
Record1.Day := 16;
```

Alternativ dazu ist auch die Verwendung einer **with**-Anweisung möglich:

```
with Record1 do
begin
   Year := 1904;
   Month := Jun;
   Day := 16;
end;
```

Nun können die Werte der Felder von *Record1* nach *Record2* kopiert werden:

```
Record2 := Record1;
```

Da der Gültigkeitsbereich eines Feldbezeichners auf den Record beschränkt ist, in dem er sich befindet, brauchen Sie nicht auf eventuelle Namenskonflikte zwischen Feldbezeichnern und anderen Variablen zu achten.

Die Konstruktion **record ...** kann auch direkt in Variablendeklarationen verwendet werden:

```
var S: record
   Name: string;
   Age: Integer;
end;
```

Eine solche Deklaration macht aber wenig Sinn, da der eigentliche Zweck eines Records darin besteht, die wiederholte Deklaration ähnlicher Variablengruppen zu vermeiden. Außerdem sind separat deklarierte Records auch dann nicht zuweisungskompatibel, wenn ihre Strukturen identisch sind.

Variante Teile in Record-Typen

Ein Record-Typ kann einen *varianten* Teil enthalten, der einer **case**-Anweisung ähnelt. Dieser variante Teil muß in der Typdeklaration nach den Feldern angegeben werden.

Mit der folgenden Syntax deklarieren Sie einen Record-Typ mit einem varianten Teil:

```
type Record-TypName = record
   Feldliste1: Typ1;
   ⋮
   Feldliste1: Typn;
case Tag: Ordinaltyp of
   Konstantenliste1: (Variante1);
   ⋮
   Konstantenlistenn: (Variantenn);
end;
```

Der erste Teil der Deklaration (bis zum reservierten Wort **case**) ist identisch mit der Deklaration eines Standard-Records. Der Rest der Deklaration (von **case** bis zum abschließenden optionalen Strichpunkt) stellt den varianten Teil mit folgenden Komponenten dar:

Tag ist optional und kann ein beliebiger, gültiger Bezeichner sein. Wenn Sie *Tag* weglassen, entfällt auch der Doppelpunkt (:).

- *Ordinaltyp* bezeichnet einen ordinalen Typ.
- Jede *Konstantenliste* ist eine Konstante (oder eine Liste von Konstanten, die durch Kommas voneinander getrennt sind), die einen Wert des Typs *Ordinaltyp* bezeichnet. In allen *Konstantenlisten* darf jeder Wert nur einmal vorkommen.
- *Variante* ist eine Liste mit Deklarationen, die durch Kommas voneinander getrennt sind. Die Liste ähnelt in etwa den *Feldliste:Typ*-Konstruktionen im Hauptteil des Record-Typs. *Variante* hat demnach folgende Form:

  Feldliste$_1$: *Typ*$_1$;

  ⋮

  Feldliste$_n$: *Typ*$_n$;

Dabei ist *Feldliste* ein gültiger Bezeichner oder eine Liste von Bezeichnern, die durch Kommas voneinander getrennt sind. *Typ* ist ein Typ. Der letzte Strichpunkt ist optional. Bei *Typ* darf es sich nicht um einen langen String, ein dynamisches Array, eine Variante (Typ Variant) oder eine Schnittstelle handeln. Strukturierte Typen, die lange Strings, dynamische Arrays, Varianten oder Schnittstellen enthalten, sind ebenfalls nicht zulässig. Zeiger auf diese Typen dürfen jedoch verwendet werden.

Die Syntax für Records mit varianten Teilen ist kompliziert, ihre Semantik ist jedoch einfach. Der variante Teil eines Records enthält mehrere *Varianten*, die sich denselben Speicherplatz teilen. Auf die Felder einer Variante kann jederzeit ein Lese- oder Schreibzugriff ausgeführt werden. Wenn Sie allerdings zunächst in ein Feld einer Variante schreiben und anschließend in ein Feld einer anderen Variante, kann das zum Überschreiben der eigenen Daten führen. Falls vorhanden, agiert *Tag* als gesondertes Feld (des Typs *Ordinaltyp*) im nichtvarianten Teil des Records.

Variante Teile erfüllen zwei Funktionen, die sich am besten anhand eines Beispiels verdeutlichen lassen. Angenommen, Sie möchten einen Record-Typ erstellen, der Felder für unterschiedliche Daten enthält. Sie wissen, daß Sie nie alle Felder einer einzelnen Record-Instanz benötigen werden. Daraufhin deklarieren Sie folgenden Record-Typ:

```
type
   TEmployee = record
   FirstName, LastName: string[40];
   BirthDate: TDate;
   case Salaried: Boolean of
    True: (AnnualSalary: Currency);
    False: (HourlyWage: Currency);
end;
```

Diesem Beispiel liegt die Überlegung zugrunde, daß ein Angestellter entweder ein jährliches Festgehalt (*AnnualSalary*) oder einen Stundenlohn (*HourlyWage*) erhält, und daß für einen Angestellten immer nur eine der Zahlungsarten in Frage kommt. Wenn Sie eine Instanz von *TEmployee* anlegen, braucht also nicht für beide Felder Speicherplatz reserviert zu werden. In diesem Beispiel unterscheiden sich die Varianten nur durch die Feldnamen. Die Felder könnten aber auch unterschiedliche Typen haben. Hier einige komplexere Beispiele:

```
type
   TPerson = record
   FirstName, LastName: string[40];
   BirthDate: TDate;
   case Citizen: Boolean of
    True: (Birthplace: string[40]);
    False: (Country: string[20];
            EntryPort: string[20];
            EntryDate, ExitDate: TDate);
   end;
type
   TShapeList = (Rectangle, Triangle, Circle, Ellipse, Other);
   TFigure = record
    case TShapeList of
      Rectangle: (Height, Width: Real);
```

```
    Triangle: (Side1, Side2, Angle: Real);
    Circle: (Radius: Real);
    Ellipse, Other: ();
  end;
```

Der Compiler weist jeder Record-Instanz so viel Speicherplatz zu, wie die Felder in der größten Variante benötigen. Die optionalen Komponenten *Tag* und *Konstantenliste* (wie *Rectangle*, *Triangle* usw. im letzten Beispiel) spielen für die Art und Weise, wie der Compiler die Felder verwaltet, keine Rolle. Sie erleichtern lediglich die Arbeit des Programmierers.

Wie bereits erwähnt, erfüllen variante Teile noch eine zweite Aufgabe. Sie können dieselben Daten so behandeln, als würden sie zu unterschiedlichen Typen gehören. Dies gilt auch in den Fällen, in denen der Compiler eine Typumwandlung nicht zuläßt. Wenn beispielsweise das erste Feld einer Variante einen 64-Bit-Real-Typ und das erste Feld einer anderen Variante einen 32-Bit-Integer-Wert enthält, können Sie dem Real-Feld einen Wert zuweisen und anschließend die ersten 32 Bits als Integer-Wert verwenden (indem Sie sie beispielsweise an eine Funktion übergeben, die einen Integer-Parameter erwartet).

Dateitypen

Eine Datei besteht aus einer geordneten Menge von Elementen desselben Typs. Für Standard-E/A-Routinen wird der vordefinierte Typ TextFile oder Text verwendet. Dieser Typ repräsentiert eine Datei, die in Zeilen angeordnete Zeichen enthält. Ausführliche Informationen über die Datei-E/A finden Sie in Kapitel 8, »Standardroutinen und E/A«.

Die Deklaration eines Dateityps erfordert folgende Syntax:

```
type Dateitypname = file of Typ
```

Dateitypname ist ein gültiger *Bezeichner. Typ* ist ein Typ fester Länge. Zeiger sind weder als implizite noch als explizite Typen erlaubt. Dynamische Arrays, lange Strings, Klassen, Objekte, Zeiger, Varianten, andere Dateien oder strukturierte Typen, die einen dieser Typen beinhalten, können deshalb nicht in Dateien enthalten sein.

Im folgenden Beispiel wird ein Dateityp für die Aufzeichnung von Namen und Telefonnummern deklariert:

```
type
  PhoneEntry = record
   FirstName, LastName: string[20];
   PhoneNumber: string[15];
   Listed: Boolean;
  end;
  PhoneList = file of PhoneEntry;
```

Sie können die Konstruktion **file of ...** auch direkt in einer Variablendeklaration verwenden:

```
var List1: file of PhoneEntry;
```

Das Wort **file** selbst gibt eine untypisierte Datei an:

```
var DataFile: file;
```

Weitere Informationen hierzu finden Sie im Abschnitt »Untypisierte Dateien« auf Seite 8-4.

In Arrays und Records sind Dateitypen nicht erlaubt.

Zeiger und Zeigertypen

Ein Zeiger ist eine Variable, die eine Speicheradresse angibt. Wenn ein Zeiger die Adresse einer anderen Variable enthält, *zeigt* er auf die Position dieser Variable im Speicher oder auf die Daten, die an dieser Position gespeichert sind. Bei einem Array oder einem anderen strukturierten Typ enthält der Zeiger die Adresse des ersten Elements der Struktur.

Zeiger sind *typisiert*. Sie geben also die Art der Daten an, auf die sie zeigen. Der Allzwecktyp Pointer repräsentiert einen Zeiger auf beliebige Daten, während spezialisierte Zeigertypen nur auf bestimmte Datentypen zeigen. Zeiger belegen immer vier Byte Speicherplatz.

Zeiger im Überblick

Das folgende Beispiel zeigt, wie Zeiger funktionieren:

```
1   var
2     X, Y: Integer;   // X und Y sind Integer-Variablen
3     P: ^Integer;     // P zeigt auf einen Integer
4   begin
5     X := 17;         // Einen Wert an X zuweisen
6     P := @X;         // Adresse von X an P zuweisen
7     Y := P^;         // P dereferenzieren; Ergebnis an Y zuweisen
8   end;
```

In Zeile 2 werden *X* und *Y* als Integer-Variablen deklariert. Zeile 3 deklariert *P* als Zeiger auf einen Integer-Wert. *P* kann also auf die Position von *X* oder *Y* zeigen. In Zeile 5 wird *X* ein Wert zugewiesen. Zeile 6 weist *P* die Adresse von *X* (angegeben durch `@X`) zu. Schließlich wird in Zeile 7 der Wert an der Adresse ermittelt, auf die *P* zeigt (angegeben durch `^P`), und *Y* zugewiesen. Nach der Ausführung dieses Programms haben *X* und *Y* denselben Wert (17).

Der Operator **@** wird hier verwendet, um die Adresse einer Variable zu ermitteln. Sie können diesen Operator aber auch für Funktionen und Prozeduren einsetzen. Weitere Informationen finden Sie unter»Der Operator @« auf Seite 4-12 und unter »Prozedurale Typen in Anweisungen und Ausdrücken« auf Seite 5-31.

Wie das obige Beispiel zeigt, erfüllt das Symbol ^ zwei Funktionen. Es kann vor einem Typbezeichner stehen, z.B.:

^Typname

In diesem Fall bezeichnet das Symbol einen Typ, der Zeiger auf Variablen des Typs *Typname* darstellt. Das Symbol ^ kann aber auch auf eine Zeigervariable folgen:

Zeiger^

In diesem Fall *dereferenziert* das Symbol den Zeiger, d.h. es liefert den Wert an der Speicheradresse, die der Zeiger angibt.

Das obige Beispiel sieht auf den ersten Blick wie eine etwas umständliche Möglichkeit aus, den Wert einer Variablen in eine andere zu kopieren. Dies ließe sich viel einfacher durch eine entsprechende Zuweisung erreichen. Die Verwendung von Zeigern ist aber aus mehreren Gründen sinnvoll. Sie sind für das Verständnis der Sprache Object Pascal wichtig, da sie in einem Programm oft hinter den Kulissen agieren und nicht explizit auftreten. Zeiger werden von allen Datentypen verwendet, die große, dynamisch zugewiesene Speicherblöcke benötigen. Beispielsweise sind lange String-Variablen ebenso wie Klassenvariablen implizite Zeiger. In vielen komplexen Programmierkonstrukten ist die Verwendung von Zeigern unverzichtbar.

In vielen Situationen sind Zeiger die einzige Möglichkeit, die strikte Typisierung der Daten durch Object Pascal zu umgehen. Sie können z.B. die in einer Variablen gespeicherten Daten ohne Berücksichtigung ihres Typs verarbeiten, indem Sie sie über den Allzweckzeiger Pointer referenzieren, diesen in den gewünschten Typ umwandeln und ihn anschließend wieder dereferenzieren. Hier ein Beispiel, in dem die in einer reellen Variable gespeicherten Daten an eine Integer-Variable zugewiesen werden:

```
type
  PInteger = ^Integer;
var
   R: Single;
   I: Integer;
   P: Pointer;
   PI: PInteger;
begin
   ⋮
  P := @R;
  PI := PInteger(P);
  I := PI^;
end;
```

Reelle und Integer-Werte werden natürlich in unterschiedlichen Formaten gespeichert. Durch die Zuweisung werden lediglich die binären Rohdaten von *R* nach *I* kopiert. Eine Konvertierung findet dabei nicht statt.

Neben der Zuweisung des Ergebnisses einer **@**-Operation können aber auch diverse Standardroutinen eingesetzt werden, um einem Zeiger einen Wert zuzuweisen. Mit den Prozeduren *New* und *GetMem* läßt sich z.B. einem vorhandenen Zeiger eine Speicheradresse zuweisen. Die Funktionen *Addr* und *Ptr* geben einen Zeiger auf eine bestimmte Adresse oder Variable zurück.

Dereferenzierte Zeiger können qualifiziert werden und als Qualifizierer agieren. Ein Beispiel hierfür ist der Ausdruck `P1^.Data^`.

Das reservierte Wort **nil** ist eine spezielle Konstante, die jedem Zeiger zugewiesen werden kann. Ein solcher Zeiger verweist dann auf kein bestimmtes Ziel.

Zeigertypen

Mit der folgenden Syntax kann ein Zeiger auf jeden beliebigen Typ deklariert werden:

```
type Zeigertypname = ^Typ
```

Es ist gängige Programmierpraxis, bei der Definition eines Record-Typs oder eines anderen Datentyps auch einen Zeiger auf diesen Typ zu deklarieren. Dies erleichtert die Verarbeitung von Instanzen des Typs, da das Kopieren größerer Speicherblöcke entfällt.

Es gibt Standardzeigertypen für unterschiedliche Verwendungszwecke. Mit dem Allzwecktyp Pointer kann jeder Datentyp referenziert werden. Eine Dereferenzierung von Variablen des Typs Pointer ist nicht möglich. Den Versuch, einer Pointer-Variablen das Symbol ^ nachzustellen, weist der Compiler zurück. Wenn Sie auf Daten zugreifen wollen, auf die eine Pointer-Variable zeigt, müssen Sie diese Variable in einen anderen Zeigertyp umwandeln, bevor Sie sie dereferenzieren.

Zeiger auf Zeichen

Die beiden fundamentalen Zeigertypen PAnsiChar und PWideChar stellen Zeiger auf AnsiChar- bzw. WideChar-Werte dar. Der generische Typ PChar repräsentiert einen Zeiger auf einen Char-Wert (in der aktuellen Implementation auf einen AnsiChar-Wert). Diese Zeigertypen auf Zeichen werden zur Verarbeitung von nullterminierten Strings eingesetzt. Weitere Einzelheiten finden Sie unter »Nullterminierte Strings« auf Seite 5-14.

Weitere Standardzeigertypen

In den Units *System* und *SysUtils* sind viele Standardzeigertypen deklariert. Obwohl es sich dabei nicht um integrierte Typen handelt, werden sie in der Delphi-Programmierung oft verwendet.

Tabelle 5.6 Eine Auswahl der in System und SysUtils deklarierten Zeigertypen

Zeigertyp	Beschreibung
PAnsiString, PString	Zeigt auf Variablen des Typs *AnsiString*.
PByteArray	Zeigt auf Variablen des Typs *ByteArray* (deklariert in *SysUtils*) und wird für die Typumwandlung dynamisch verwalteter Speicherblöcke verwendet, um Array-Zugriffe zu ermöglichen.
PCurrency	Zeigt auf Variablen des Typs *Currency*.
PExtended	Zeigt auf Variablen des Typs *Extended*.
POleVariant	Zeigt auf Variablen des Typs *OleVariant*.
PShortString	Zeigt auf Variablen des Typs *ShortString* und ist bei der Portierung von Legacy-Code hilfreich, in dem der Typ *PString* verwendet wird.
PTextBuf	Zeigt auf Variablen des Typs *TextBuf* (deklariert in *SysUtils*). TextBuf ist der interne Puffer für den Datei-Record TTextRec.
PVarRec	Zeigt auf Variablen des Typs *TVarRec* (deklariert in *System*).
PVariant	Zeigt auf Variablen des Typs *Variant*.

Tabelle 5.6 Eine Auswahl der in System und SysUtils deklarierten Zeigertypen (Fortsetzung)

Zeigertyp	Beschreibung
PWideString	Zeigt auf Variablen des Typs *WideString*.
PWordArray	Zeigt auf Variablen des Typs *TWordArray* (deklariert in *SysUtils*) und wird für die Typumwandlung dynamisch verwalteter Speicherblöcke verwendet, wenn diese als Array mit 2-Byte-Werten bearbeitet werden sollen.

Prozedurale Typen

Prozeduren und Funktionen können als Wert betrachtet und einer Variablen zugewiesen werden, so daß sie sich als Parameter übergeben lassen. Dieses Verfahren wird durch sogenannte prozedurale Typen ermöglicht. Nehmen wir an, Sie definieren eine Funktion mit dem Namen *Calc*, die zwei Integer-Parameter übernimmt und einen Integer-Wert zurückgibt:

```
function Calc(X,Y: Integer): Integer;
```

Sie können die *Calc*-Funktion nun der Variablen *F* zuweisen:

```
var F: function(X,Y: Integer): Integer;
F := Calc;
```

Wenn Sie im Kopf einer Prozedur oder Funktion den Bezeichner nach dem Wort **procedure** bzw. **function** entfernen, erhalten Sie den Namen eines prozeduralen Typs. Dieser Typname kann direkt in einer Variablendeklaration (siehe oben) verwendet oder zur Deklaration neuer Typen benutzt werden:

```
type
   TIntegerFunction = function: Integer;
   TProcedure = procedure;
   TStrProc = procedure(const S: string);
   TMathFunc = function(X: Double): Double;
var
   F: TIntegerFunction;             { F ist eine parameterlose Funktion, die einen
                                      Integer zurückgibt }
   Proc: TProcedure;                { Proc ist eine parameterlose Prozedur }
   SP: TStrProc;                    { SP ist eine Prozedur, die einen String-
                                      Parameter übernimmt }
   M: TMathFunc;                    { M ist eine Funktion, die einen Double-
                                      Parameter (reeller Wert)übernimmt und einen
                                      Double-Wert zurückgibt }
procedure FuncProc(P: TIntegerFunction);        { FuncProc ist eine Prozedur, deren
                                                  einziger Parameter eine
                                                  parameterlose Funktion ist, die
                                                  einen Integer zurückgibt }
```

Alle oben aufgeführten Variablen sind sogenannte *Prozedurzeiger*, also Zeiger auf die Adresse einer Prozedur oder Funktion. Um eine Methode eines Instanzobjekts zur referenzieren (siehe Kapitel 7, »Klassen und Objekte«), muß dem Namen des prozeduralen Typs die Klausel **of object** hinzugefügt werden:

```
type
   TMethod = procedure of object;
   TNotifyEvent = procedure(Sender: TObject) of object;
```

Diese Typen stellen *Methodenzeiger* dar. Ein Methodenzeiger wird in Form zweier Zeiger codiert, von denen der erste die Adresse der Methode speichert. Der zweite enthält eine Referenz auf das Objekt, zu dem die Methode gehört. Ein Beispiel:

```
type
   TNotifyEvent = procedure(Sender: TObject) of object;
   TMainForm = class(TForm)
    procedure ButtonClick(Sender: TObject);
    ⋮
   end;
var
   MainForm: TMainForm;
   OnClick: TNotifyEvent
```

Nach diesen Deklarationen ist die folgende Zuweisung korrekt:

```
OnClick := MainForm.ButtonClick;
```

Zwei prozedurale Typen sind kompatibel, wenn sie folgende Bedingungen erfüllen:

- Sie verwenden dieselbe Aufrufkonvention.
- Sie haben denselben (oder keinen) Rückgabetyp.
- Sie verwenden eine identische Anzahl von Parametern. Die Parameter an einer bestimmten Position müssen identische Typen haben (die Parameternamen spielen keine Rolle).

Zeiger auf Prozeduren sind niemals kompatibel zu Zeigern auf Methoden. Der Wert **nil** kann jedem prozeduralen Typ zugewiesen werden.

Verschachtelte Prozeduren und Funktionen (Routinen, die in anderen Routinen deklariert sind), können nicht als prozedurale Werte verwendet werden. Dasselbe gilt für vordefinierte Prozeduren und Funktionen (Standardroutinen). Wenn Sie eine Standardroutine wie *Length* als prozeduralen Wert verwenden wollen, müssen Sie sie gewissermaßen »verpacken«:

```
function FLength(S: string): Integer;
begin
   Result := Length(S);
end;
```

Prozedurale Typen in Anweisungen und Ausdrücken

Wenn links in einer Zuweisung eine prozedurale Variable steht, erwartet der Compiler auf der rechten Seite einen prozeduralen Wert. Durch die Zuweisung wird aus der Variablen auf der linken Seite ein Zeiger auf die Funktion oder Prozedur auf der rechten Seite. In einem anderen Kontext führt die Verwendung einer prozeduralen Variable zu einem Aufruf der referenzierten Prozedur oder Funktion. Prozedurale Variablen können auch als Parameter übergeben werden:

```
var
   F: function(X: Integer): Integer;
   I: Integer;
function SomeFunction(X: Integer): Integer;
   ⋮
F := SomeFunction;  // SomeFunction an F zuweisen
I := F(4);          // Funktion aufrufen; Ergebnis an I zuweisen
```

In Zuweisungen bestimmt der Typ der Variablen auf der linken Seite, wie die Prozedur oder Methode auf der rechten Seite interpretiert wird. Ein Beispiel:

```
var
   F, G: function: Integer;
   I: Integer;
function SomeFunction: Integer;
   ⋮
F := SomeFunction;  // SomeFunction an F zuweisen
G := F;             // F nach G kopieren
I := G;             // Funktion aufrufen; Ergebnis I zuweisen
```

Die erste Anweisung weist *F* einen prozeduralen Wert zu. Die zweite Anweisung kopiert diesen Wert in eine andere Variable. In der dritten Anweisung wird die referenzierte Funktion aufgerufen und das Ergebnis *I* zugewiesen. Da *I* keine prozedurale, sondern eine Integer-Variable ist, führt die letzte Zuweisung zum Aufruf der Funktion (die als Ergebnis einen Integer zurückgibt).

Es gibt Situationen, in denen nicht offensichtlich ist, wie eine prozedurale Variable interpretiert werden muß. Ein Beispiel:

```
if F = MyFunction then ...;
```

In diesem Fall führt *F* zum Aufruf einer Funktion. Der Compiler ruft zuerst die Funktion auf, die von *F* referenziert wird, und dann die Funktion *MyFunction*. Anschließend werden die Ergebnisse verglichen. Hier gilt folgende Regel: Eine prozedurale Variable innerhalb eines Ausdrucks stellt einen Aufruf der referenzierten Prozedur oder Funktion dar. Referenziert *F* eine Prozedur (die keinen Wert als Ergebnis liefert) oder eine Funktion, an die Parameter übergeben werden müssen, führt die obige Anweisung zu einem Compiler-Fehler. Um den prozeduralen Wert von *F* mit *MyFunction* zu vergleichen, verwenden Sie folgende Anweisung:

```
if @F = @MyFunction then ...;
```

`@F` konvertiert *F* in eine untypisierte Zeigervariable, die eine Adresse enthält, und `@MyFunction` liefert die Adresse von *MyFunction*.

Um anstelle der in einer prozeduralen Variablen gespeicherten Adresse ihre Speicheradresse zu erhalten, verwenden Sie **@@**. So liefert beispielsweise **`@@F`** die Adresse *von F.*

Der Operator **@** kann auch verwendet werden, um einer prozeduralen Variablen einen untypisierten Zeigerwert zuzuweisen:

```
var StrComp: function(Str1, Str2: PChar): Integer;
   ⋮
@StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');
```

Diese Anweisung ruft die Windows-Funktion *GetProcAddress* auf und weist *StrComp* das Ergebnis zu.

Jede prozedurale Variable kann den Wert **nil** enthalten, was bedeutet, daß sie auf keine bestimmte Adresse zeigt. Der Aufruf einer prozeduralen Variablen, die den Wert **nil** enthält, führt zu einem Fehler. Mit der Standardfunktion *Assigned* können Sie feststellen, ob einer prozeduralen Variablen ein Wert zugewiesen ist:

```
if Assigned(OnClick) then OnClick(X);
```

Variante Typen

Es gibt Situationen, in denen Daten verarbeitet werden müssen, deren Typ sich zur Laufzeit dynamisch ändert oder zur Compilierzeit noch nicht bekannt ist. In derartigen Fällen werden Variablen und Parameter des Typs Variant eingesetzt. Die sogenannten Varianten bieten zwar eine größere Flexibilität, belegen aber auch mehr Speicher als normale Variablen. Operationen mit Varianten verlaufen deshalb deutlich langsamer als solche mit statischen Typen. Außerdem führen unzulässige Operationen, die bei regulären Variablen während des Compilierens bereinigt werden, bei Varianten oft zu Laufzeitfehlern.

Eine Variante kann zur Laufzeit die verschiedensten Typen annehmen. Records, Mengen, statische Arrays, Dateien, Klassen, Klassenreferenzen, Zeiger und der Typ Int64 sind jedoch nicht erlaubt. Varianten können also Werte beliebigen Typs mit Ausnahme von strukturierten Typen, Zeigern und Int64 enthalten. Wenn eine Variante ein COM- oder CORBA-Objekt enthält, können dessen Eigenschaften mit der Variante verwaltet und Objektmethoden aufgerufen werden. Einzelheiten hierzu finden Sie in Kapitel 10, »Objektschnittstellen«. Varianten können auch dynamische Arrays und *variante Arrays* (ein spezieller Typ von statischen Arrays) aufnehmen. Weitere Informationen finden Sie unter »Variante Arrays« auf Seite 5-36. Varianten können in Ausdrücken und Zuweisungen mit anderen Varianten, aber auch mit Integer-Werten, reellen Werten, Strings und Booleschen Werten kombiniert werden. Die erforderlichen Typumwandlungen nimmt der Compiler automatisch vor.

Varianten, die Strings enthalten, können nicht indiziert werden. Wenn die Variante *V* beispielsweise einen String-Wert enthält, ist die Konstruktion `V[1]` nicht zulässig.

Eine Variante belegt 16 Bytes Speicher und besteht aus einem Typencode und einem Wert (bzw. einem Zeiger auf einen Wert), dessen Typ durch den Code festgelegt ist. Alle Varianten werden bei der Erstellung mit dem speziellen Wert *Unassigned* initialisiert. Der Wert Null steht für unbekannte oder fehlende Daten.

Der Typencode einer Variante kann mit der Standardfunktion *VarType* ermittelt werden. Die Konstante *varTypeMask* ist eine Bit-Maske, mit der der Typencode aus dem Rückgabewert der Funktion *VarType* isoliert werden kann. Der folgende Ausdruck ergibt beispielsweise *True*, wenn *V* einen Double-Wert oder ein Array solcher Werte enthält:

```
VarType(V) and varTypeMask = varDouble
```

Die Maske verbirgt einfach das erste Bit, das anzeigt, ob die Variante ein Array enthält. Der in der Unit *System* definierte Record-Typ TVarData ermöglicht die Typum-

wandlung einer Variante, um Zugriff auf deren interne Darstellung zu erhalten. Eine Liste der aktuellen Typencodes finden Sie in der Online-Hilfe zu *VarType*. In zukünftigen Versionen von Object Pascal können weitere Typencodes hinzukommen.

Typkonvertierung bei Varianten

Alle Integer-Typen, reellen Typen, Strings, Zeichentypen und Booleschen Typen (außer Int64) sind zum Typ Variant zuweisungskompatibel. Ausdrücke können explizit in eine Variante konvertiert werden. Zusätzlich kann die interne Darstellung einer Variante mit den Standardroutinen *VarAsType* und *VarCast* verändert werden. Das folgende Beispiel zeigt die Verwendung von Varianten und die automatische Konvertierung, die bei einer Kombination von Varianten mit anderen Typen durchgeführt wird:

```
var
   V1, V2, V3, V4, V5: Variant;
   I: Integer;
   D: Double;
   S: string;
begin
   V1 := 1; { Integer-Wert }
   V2 := 1234.5678; { Reeller Wert }
   V3 := 'Hello world!'; { String-Wert }
   V4 := '1000'; { String-Wert }
   V5 := V1 + V2 + V4; { Reeller Wert 2235.5678}
   I := V1; { I = 1 (Integer-Wert) }
   D := V2; { D = 1234.5678 (Reeller Wert) }
   S := V3; { S = 'Hello world!' (String-Wert) }
   I := V4; { I = 1000 (Integer-Wert) }
   S := V5; { S = '2235.5678' (String-Wert) }
end;
```

Die folgende Tabelle enthält die Regeln, die bei der Konvertierung von Varianten in andere Typen gelten.

Tabelle 5.7 Regeln für die Typkonvertierung bei Varianten

Ausgangstyp \ Zieltyp	Integer	Real	String	Zeichen	Boolean
Integer	Konvertiert Integer-Formate	Konvertiert in reelle Werte	Konvertiert in String-Darstellung	Identisch mit String (links)	Ergibt *False*, wenn 0, andernfalls *True*
Real	Rundet zum nächsten Integer	Konvertiert reelle Formate	Konvertiert in String-Darstellung (länderspezifische Windows-Einstellungen werden verwendet)	Identisch mit String (links)	*Ergibt False*, wenn 0, andernfalls *True*
String	Konvertiert in Integer (Wert wird evtl. abgeschnitten); wenn der String nicht numerisch ist, wird eine Exception ausgelöst	Konvertiert in reelle Werte (länderspezifische Windows-Einstellungen werden verwendet); wenn der String nicht numerisch ist, wird eine Exception ausgelöst	Konvertiert String-/ Zeichenformate	Identisch mit String (links)	Ergibt *False*, wenn der String "false" ist (ohne Berücksichtigung der Groß-/ Kleinschreibung) oder ein numerischer String, der zu 0 ausgewertet wird; ergibt *True*, wenn der String "true" oder ein numerischer String ungleich 0 ist; andernfalls wird eine Exception ausgelöst
Zeichen	Identisch mit String (oben)	Identisch mit String (oben)	Identisch mit String (oben)	Identisch mit String-zu-String	Identisch mit String (oben)
Boolean	*False* = 0, *True* = 1 (255 wenn Byte)	*False* = 0, *True* = 1	*False* = "0", *True* = "1"	Identisch mit String (links)	*False* = *False*, *True* = *True*
Unassigned	Ergibt 0	Ergibt 0	Ergibt einen leeren String	Identisch mit String (links)	Ergibt *False*
Null	Löst Exception aus	Löst Exception aus	Löst Exception aus	Identisch mit String (links)	Löst Exception aus

Wenn Zuweisungen außerhalb des zulässigen Bereichs liegen, wird der Zielvariablen meist der höchste Wert im Bereich zugewiesen. Ungültige Zuweisungen oder Konvertierungen führen zu einer *EVariantError*-Exception.

Für den in der Unit *System* deklarierten reellen Typ TDateTime gelten gesonderte Konvertierungsregeln. TDateTime-Variablen werden bei einer Typumwandlung als normaler Double-Wert behandelt. Wenn ein Integer-Wert, ein reeller Wert oder ein Boolescher Wert in den Typ TDateTime konvertiert wird, erfolgt zunächst eine Umwandlung in den Typ Double. Anschließend wird der Wert als Datums-/Zeitwert ge-

lesen. Bei der Umwandlung eines Strings in den Typ TDateTime wird der String unter Verwendung der länderspezifischen Windows-Einstellungen als Datums-/Zeitwert interpretiert. Ein in den Typ TDateTime konvertierter *Unassigned*-Wert wird als reeller oder als Integer-Wert 0 behandelt. Bei der Konvertierung eines Null-Wertes in den Typ TDateTime wird eine Exception ausgelöst.

Für eine Variante, die ein COM-Objekt referenziert, wird beim Versuch einer Konvertierung die Standardeigenschaft des Objekts gelesen und der betreffende Wert in den geforderten Typ umgewandelt. Besitzt das Objekt keine Standardeigenschaft, wird eine Exception ausgelöst.

Varianten in Ausdrücken

Alle Operatoren außer ^, **is** und **in** unterstützen variante Operanden. Operationen mit Varianten liefern Werte des Typs Variant. Wenn einer oder beide Operanden Null sind, ist der Ergebniswert ebenfalls Null. Hat einer oder beide Operanden den Status *Unassigned*, wird eine Exception ausgelöst. Ist in einer binären Operation nur ein Operand eine Variante, wird der andere in eine Variante konvertiert.

Der Rückgabetyp einer Operation ist von den verwendeten Operanden abhängig. Im allgemeinen gelten für Varianten dieselben Regeln wie für Operanden mit statisch gebundenen Typen. Wenn z.B. die Varianten *V1* und *V2* einen Integer und einen reellen Wert enthalten, ist das Ergebnis von `V1 + V2` eine Variante vom Typ Real (siehe »Operatoren« auf Seite 4-6). Mit Varianten können binäre Operationen mit Wertkombinationen ausgeführt werden, die bei Ausdrücken mit statischem Typ nicht möglich sind. Der Compiler versucht, Varianten unterschiedlichen Typs zu konvertieren. Diese Konvertierung erfolgt nach den Regeln in Tabelle 5.7. Enthalten beispielsweise die Varianten *V3* und *V4* einen numerischen String und einen Integer, liefert der Ausdruck `V3 + V4` eine Variante vom Typ Integer. Die Umwandlung des numerischen Strings in einen Integer erfolgt vor der Ausführung der Operation.

Variante Arrays

Einer Variante kann kein normales statisches Array zugewiesen werden. Es wird ein variantes Array benötigt, das durch einen Aufruf der Standardfunktionen *VarArrayCreate* oder *VarArrayOf* erzeugt werden kann. Die folgende Anweisung erzeugt ein variantes Array mit 10 Integer-Werten und weist es der Variante *V* zu:

```
V: Variant;
   ⋮
V := VarArrayCreate([0,9], varInteger);
```

Das Array kann mit `V[0], V[1]` usw. indiziert werden. Es ist aber nicht möglich, Elemente eines varianten Arrays als **var**-Parameter zu übergeben. Variante Arrays werden immer mit Integer-Werten indiziert.

Der zweite Parameter in einem Aufruf von *VarArrayCreate* ist der Typencode für den Basistyp des Arrays. Eine Liste dieser Codes finden Sie in der Online-Hilfe zu *VarType*. Die Übergabe des Codes *varString* an *VarArrayCreate* ist nicht zulässig. Für variante String-Arrays muß der Typencode *varOleStr* verwendet werden.

Varianten können variante Arrays mit unterschiedlichen Größen, Dimensionen und Basistypen enthalten. Die Elemente eines varianten Arrays können jeden in Varianten zulässigen Typ mit Ausnahme von ShortString und AnsiString haben. Wenn das Array den Basistyp Variant hat, können die Elemente sogar heterogen sein. Die Größe eines varianten Arrays kann mit der Funktion *VarArrayRedim* verändert werden. Weitere Routinen zur Bearbeitung varianter Arrays sind *VarArrayDimCount*, *VarArrayLowBound*, *VarArrayHighBound*, *VarArrayRef*, *VarArrayLock* und *VarArrayUnlock*.

Wenn eine Variante, die ein variantes Array enthält, einer anderen Variante zugewiesen oder als Wertparameter übergeben wird, entsteht eine Kopie des gesamten Arrays. Dieser Vorgang ist jedoch sehr speicherintensiv und sollte möglichst vermieden werden.

OleVariant

Der Typ OleVariant bezeichnet eine Variante, die ausschließlich COM-kompatible Typen enthält. Wenn ein Variant-Wert einem OleVariant-Wert zugewiesen wird, werden alle nichtkompatiblen Typen in ihre kompatiblen Entsprechungen umgewandelt. Wenn z.B. eine Variante, die einen AnsiString-Wert enthält, einem OleVariant-Wert zugewiesen wird, wird AnsiString in den Typ WideString konvertiert.

Kompatibilität und Identität von Typen

Um zu verstehen, welche Operationen mit unterschiedlichen Ausdrücken durchgeführt werden können, müssen Ihnen die verschiedenen Arten der Kompatibilität zwischen Typen und Werten bekannt sein. Dazu gehört die *Typidentität*, die *Typkompatibilität* und die *Zuweisungskompatibilität*.

Typidentität

Für die Typidentität gilt folgende Regel: Wenn ein Typbezeichner durch Verwendung eines anderen Typbezeichners deklariert wird (ohne Qualifizierer), bezeichnen beide denselben Typ. Betrachten Sie die folgenden Typdeklarationen:

```
type
   T1 = Integer;
   T2 = T1;
   T3 = Integer;
   T4 = T2;
```

Hieraus folgt, daß *T1*, *T2*, *T3*, *T4* und Integer identische Typen sind. Um unterschiedliche Typen zu definieren, wiederholen Sie das Wort **type** in der Deklaration. Die folgende Anweisung erzeugt z.B. einen neuen Typ mit dem Namen TMyInteger, der nicht mit Integer identisch ist:

```
type TMyInteger = type Integer;
```

Sprachkonstrukte, die als Typnamen fungieren, erzeugen bei jedem Auftreten einen anderen Typ. Die folgenden Deklarationen definieren beispielsweise die beiden unterschiedlichen Typen TS1 und TS2:

```
type
   TS1 = set of Char;
   TS2 = set of Char;
```

Mit den Variablendeklarationen

```
var
   S1: string[10];
   S2: string[10];
```

werden zwei Variablen unterschiedlichen Typs erzeugt. Um Variablen mit identischem Typ zu deklarieren, gehen Sie folgendermaßen vor:

```
var S1, S2: string[10];
```

oder

```
type MyString = string[10];
var
   S1: MyString;
   S2: MyString;
```

Typkompatibilität

Jeder Typ ist mit sich selbst kompatibel. Zwei verschiedene Typen sind kompatibel, wenn sie mindestens eine der folgenden Bedingungen erfüllen:

- Beide Typen sind reelle Typen.
- Beide Typen sind Integer-Typen.
- Ein Typ ist ein Teilbereich des anderen.
- Beide Typen sind Teilbereiche desselben Typs.
- Beide Typen sind Mengentypen mit kompatiblen Basistypen.
- Beide Typen sind gepackte String-Typen mit identischer Anzahl von Komponenten.
- Ein Typ ist ein String-Typ, der andere ist ein String-Typ, ein gepackter String-Typ oder der Typ Char.
- Ein Typ ist eine Variante, der andere ein Integer-Typ, ein reeller Typ, ein String-Typ, ein Zeichentyp oder ein Boolescher Typ.
- Beide Typen sind Klassentypen, Klassenreferenztypen oder Schnittstellentypen, wobei der eine Typ vom anderen abgeleitet ist.
- Ein Typ ist PChar oder PWideChar, der andere ein nullbasiertes Zeichen-Array der Form `array[0..`*n*`] of Char`.
- Ein Typ ist vom Typ Pointer (untypisierter Zeiger), der andere ist ein beliebiger Zeigertyp.
- Beide Typen sind (typisierte) Zeiger auf denselben Typ, und die Compiler-Direktive {$T+} ist aktiviert.

- Beide Typen sind prozedurale Typen mit identischem Ergebnistyp und identischer Parameteranzahl, wobei Parameter an derselben Position identische Typen haben müssen.

Zuweisungskompatibilität

Bei der Zuweisungskompatibilität handelt es sich nicht um ein symmetrisches Verhältnis. Ein Ausdruck des Typs *T2* kann einer Variablen des Typs *T1* zugewiesen werden, wenn der Ausdruck im Bereich von *T1* liegt und mindestens eine der folgenden Bedingungen erfüllt ist:

- *T1* und *T2* sind identische Typen, und beide sind weder ein Dateityp noch ein strukturierter Typ, der Dateikomponenten enthält.
- *T1* und *T2* sind kompatible ordinale Typen.
- *T1* und *T2* sind reelle Typen.
- *T1* ist ein reeller Typ, und *T2* ist ein Integer-Typ.
- *T1* ist ein PChar oder ein String-Typ, und der Ausdruck ist eine String-Konstante.
- *T1* und *T2* sind String-Typen.
- *T1* ist ein String-Typ, und *T2* ist ein gepackter String-Typ oder ein PChar.
- *T1* ist ein langer String, und *T2* ist ein PChar.
- *T1* und *T2* sind kompatible gepackte String-Typen.
- *T1* und *T2* sind kompatible Mengentypen.
- *T1* und *T2* sind kompatible Zeigertypen.
- *T1* und *T2* sind Klassentypen, Klassenreferenztypen oder Schnittstellentypen, wobei *T2* von *T1* abgeleitet ist.
- *T1* ist ein Schnittstellentyp, und *T2* ist ein Klassentyp, der *T1* implementiert.
- *T1* ist vom Typ PChar oder PWideChar, und *T2* ist ein nullbasiertes Zeichen-Array der Form *array[0..n] of Char*.
- *T1* und *T2* sind kompatible prozedurale Typen. (Ein Funktions- oder Prozedurbezeichner wird in bestimmten Zuweisungsanweisungen als Ausdruck eines prozeduralen Typs behandelt. Einzelheiten finden Sie unter »Prozedurale Typen in Anweisungen und Ausdrücken« auf Seite 5-31.)
- *T1* ist eine Variante, und *T2* ist ein Integer-Typ, ein reeller Typ, ein String-Typ, ein Zeichentyp, ein Boolescher Typ oder ein Schnittstellentyp.
- *T1* ist ein Integer-Typ, ein reeller Typ, ein String-Typ, ein Zeichentyp oder ein Boolescher Typ, und *T2* ist eine Variante.
- *T1* ist der Schnittstellentyp IUnknown oder IDispatch, und *T2* ist ein Variante. (Die Variante muß den Typencode *varEmpty*, *varUnknown* oder *varDispatch* haben, wenn *T1* IUnknown ist, bzw. *varEmpty* oder *varDispatch*, wenn *T1* IDispatch ist.)

Typdeklaration

Bei einer Typdeklaration wird ein Bezeichner angegeben, der einen Typ festlegt. Die Syntax für eine Typdeklaration lautet

```
type neuerTyp = Typ
```

Hierbei ist *neuerTyp* ein gültiger Bezeichner. Nach der Typdeklaration

```
type TMyString = string;
```

ist beispielsweise die folgende Variablendeklaration möglich:

```
var S: TMyString;
```

Die Typdeklaration selbst liegt nicht im Gültigkeitsbereich des Typbezeichners (dies gilt allerdings nicht für den Typ Pointer). Aus diesem Grund kann beispielsweise ein Record-Typ definiert werden, der sich selbst rekursiv verwendet.

Wenn Sie einen Typ deklarieren, der mit einem vorhandenen Typ identisch ist, behandelt der Compiler den neuen Typbezeichner als Alias für den alten. Bei der folgenden Deklaration haben z.B. *X* und *Y* denselben Typ:

```
type TValue = Real;
var
   X: Real;
   Y: TValue;
```

Hier ist zur Laufzeit keine Unterscheidung zwischen *TValue* und *Real* möglich (was normalerweise auch nicht nötig ist). Wenn der neue Typ aber Laufzeit-Typinformationen bereitstellen soll (z.B. zur Bearbeitung von Eigenschaften eines bestimmten Typs in einem Delphi-Eigenschaftseditor), muß eine Unterscheidung zwischen den Typen möglich sein. Verwenden Sie in solchen Fällen die folgende Syntax:

```
type neuerTyp = Typ Typ
```

Die folgende Anweisung zwingt den Compiler beispielsweise, einen neuen, eigenen Typ namens *TValue* zu erzeugen:.

```
type TValue = type Real;
```

Variablen

Variablen sind Bezeichner für Werte, die sich zur Laufzeit ändern können. Eine Variable ist sozusagen ein Name für eine Speicheradresse. Sie kann für Lese- und Schreibzugriffe auf diese Speicheradresse verwendet werden. Variablen dienen als Container für Daten eines bestimmten Typs. Die Typisierung liefert dem Compiler Informationen darüber, wie die Daten zu interpretieren sind.

Variablendeklarationen

Die grundlegende Syntax für eine Variablendeklaration lautet

```
var Bezeichnerliste: Typ;
```

Bezeichnerliste ist eine Liste mit gültigen Bezeichnern, die durch Kommas voneinander getrennt sind. *Typ* ist ein gültiger Typ. Die folgende Anweisung deklariert eine Variable mit dem Namen *I* und dem Typ Integer:

```
var I: Integer;
```

Die folgende Anweisung deklariert die Variablen *X* und *Y* vom Typ Real:

```
var X, Y: Real;
```

Bei aufeinanderfolgenden Deklarationen braucht das reservierte Wort **var** nicht wiederholt zu werden:

```
var
  X, Y, Z: Double;
  I, J, K: Integer;
  Digit: 0..9;
  Okay: Boolean;
```

Variablen, die innerhalb von Prozeduren und Funktionen deklariert werden, werden als *lokale* Variablen bezeichnet. Alle anderen Variablen sind *globale* Variablen. Einer globalen Variable kann bei der Deklaration mit folgender Syntax ein Anfangswert zugewiesen werden:

```
var Bezeichner: Typ = konstanterAusdruck;
```

Dabei ist *konstanterAusdruck* ein beliebiger konstanter Ausdruck, der einen Wert des Typs *Typ* darstellt. (Ausführliche Informationen über konstante Ausdrücke finden Sie unter »Konstante Ausdrücke« auf Seite 5-44.) Die Deklaration

```
var I: Integer = 7;
```

entspricht damit der folgenden Deklaration und Anweisung:

```
var I: Integer;
  ⋮
I := 7;
```

Aufeinanderfolgende Variablendeklarationen (wie `var X, Y, Z: Real;`) dürfen weder Initialisierungen enthalten noch Deklarationen von Varianten und Variablen des Typs Datei.

Wenn Sie eine globale Variable nicht explizit initialisieren, wird sie vom Compiler zunächst auf 0 gesetzt. Lokale Variablen können bei der Deklaration nicht initialisiert werden. Sie sind nicht definiert, solange ihnen kein Wert zugewiesen wird.

Der bei der Deklaration einer Variablen zugewiesene Speicher wird automatisch freigegeben, wenn die Variable nicht mehr benötigt wird. Lokale Variablen werden freigegeben, wenn die Prozedur oder Funktion beendet wird, in der sie deklariert sind. Weitere Informationen über Variablen und die Verwaltung des Speichers finden Sie in Kapitel 11, »Speicherverwaltung«.

Absolute Adressen

Zur Deklaration einer Variablen an einer bestimmten Speicheradresse geben Sie nach dem Namen des Typs das Wort **absolute** und die Adresse an:

```
var CrtMode: Byte absolute $0040;
```

Dieses Vorgehen ist nur bei der Low-Level-Programmierung sinnvoll, z.B. bei der Entwicklung von Gerätetreibern.

Um eine neue Variable an der Adresse zu erstellen, an der bereits eine Variable existiert, geben Sie nach dem Wort **absolute** den Namen der vorhandenen Variablen an (anstelle der Adresse):

```
var
   Str: string[32];
   StrLen: Byte absolute Str;
```

Diese Anweisung legt fest, daß die Variable *StrLen* an derselben Adresse wie die Variable *Str* beginnt. Da das erste Byte eines kurzen Strings dessen Länge angibt, ist der Wert von *StrLen* die Länge von *Str.*

Eine Variable in einer **absolute**-Deklaration kann nicht initialisiert werden.

Dynamische Variablen

Durch einen Aufruf der Prozedur *GetMem* oder *New* können Sie dynamische Variablen erzeugen. Diese Variablen werden auf dem Heap zugewiesen und nicht automatisch verwaltet. Die Freigabe des für eine dynamische Variable reservierten Speichers liegt in der Verantwortung des Programmierers. Variablen, die mit *GetMem* erzeugt wurden, müssen mit *FreeMem* freigegeben werden. Variablen, die mit *New* erzeugt wurden, werden mit *Dispose* freigegeben. Weitere Routinen zur Verwaltung von dynamischen Variablen sind *ReallocMem*, *Initialize*, *StrAlloc* und *StrDispose.*

Lange Strings und dynamische Arrays sind ebenfalls dynamische Variablen, die auf dem Heap zugewiesen werden. Ihr Speicher wird allerdings automatisch verwaltet.

Thread-Variablen

Thread-Variablen (oder Thread-lokale Variablen) finden in Multithread-Anwendungen Verwendung. Eine Thread-Variable entspricht in etwa einer globalen Variable, die einzelnen Ausführungs-Threads erhalten jedoch eine eigene, private Kopie der Variable, auf die andere Threads keinen Zugriff haben. Thread-Variablen werden nicht mit **var**, sondern mit **threadvar** deklariert:

```
threadvar X: Integer;
```

Die Deklaration einer Thread-Variablen

- darf nicht innerhalb einer Prozedur oder Funktion erfolgen;
- darf keine Initialisierungen enthalten;
- darf nicht die Direktive **absolute** spezifizieren.

Referenzgezählte Variablen (wie lange Strings, dynamische Arrays oder auch Schnittstellen) sind selbst dann nicht thread-sicher, wenn sie mit **threadvar** deklariert wurden. Die Verwendung von dynamischen Thread-Variablen ist nicht möglich, da es in der Regel keine Möglichkeit gibt, den auf dem Heap zugewiesenen Speicher wieder freizugeben (der von den einzelnen Ausführungs-Threads reserviert wird). Die Generierung von Thread-Variablen mit Zeiger- oder prozeduralem Typ ist nicht zulässig.

Deklarierte Konstanten

Unter dem Begriff *Konstanten* werden verschiedene Sprachkonstrukte zusammengefaßt. Es gibt numerische Konstanten wie die Zahl `17` und String-Konstanten wie `'Hello World!'`. Numerische Konstanten werden auch einfach als *Zahlen* bezeichnet, String-Konstanten als *Zeichen-Strings* oder *String-Literale.* Ausführliche Informationen über numerische und String-Konstanten finden Sie in Kapitel 4, »Syntaktische Elemente«. Jeder Aufzählungstyp definiert Konstanten, die Werte dieses Typs darstellen. Neben vordefinierten Konstanten wie *True, False* und **nil** gibt es auch Konstanten, die durch eine Deklaration erzeugt werden (z.B. Variablen).

Deklarierte Konstanten sind entweder *echte Konstanten* oder *typisierte Konstanten*. Oberflächlich betrachtet besteht zwischen diesen beiden Konstantentypen kein großer Unterschied. Für ihre Deklaration gelten aber verschiedene Regeln, und sie werden für unterschiedliche Zwecke eingesetzt.

Echte Konstanten

Unter einer echten Konstante versteht man einen deklarierten Bezeichner, dessen Wert sich nicht ändern kann. Die folgende Anweisung deklariert z.B. eine Konstante namens *MaxValue* mit dem Integer-Wert 237:

```
const MaxValue = 237;
```

Die Syntax für die Deklaration echter Konstanten lautet

```
const Bezeichner = konstanterAusdruck
```

Dabei ist *Bezeichner* ein gültiger Bezeichner, und *konstanterAusdruck* ein Ausdruck, der vom Compiler ohne Ausführung des Programms ausgewertet werden kann. Weitere Einzelheiten finden Sie unter »Konstante Ausdrücke« auf Seite 5-44.

Wenn *konstanter Ausdruck* einen ordinalen Wert zurückgibt, kann der Typ der deklarierten Konstante über eine Wertumwandlung festgelegt werden. Die folgende Anweisung deklariert eine Konstante mit dem Namen *MyNumber* und dem Typ Int64, die den Integer-Wert 17 zurückgibt:

```
const MyNumber = Int64(17);
```

Wenn der Typ nicht auf diese Weise festgelegt wird, erhält die deklarierte Konstante den Typ von *konstanterAusdruck*.

- Wenn es sich bei *konstanterAusdruck* um einen Zeichen-String handelt, ist die deklarierte Konstante mit jedem String-Typ kompatibel. Hat der Zeichen-String die Länge 1, ist er zusätzlich mit jedem Zeichentyp kompatibel.
- Wenn *konstanterAusdruck* vom Typ Real ist, hat die deklarierte Konstante den Typ Extended. Ist *konstanterAusdruck* ein Integer, hat die deklarierte Konstante den Typ Longint. Dies gilt aber nur, wenn der Wert von *konstanterAusdruck* nicht im Bereich von Longint liegt (in diesem Fall hat die Konstante den Typ Int64).

Hier einige Beispiele für Konstantendeklarationen:

```
const
   Min = 0;
   Max = 100;
   Center = (Max - Min) div 2;
   Beta = Chr(225);
   NumChars = Ord('Z') - Ord('A') + 1;
   Message = 'Out of memory';
   ErrStr = ' Error: ' + Message + '. ';
   ErrPos = 80 - Length(ErrStr) div 2;
   Ln10 = 2.302585092994045684;
   Ln10R = 1 / Ln10;
   Numeric = ['0'..'9'];
   Alpha = ['A'..'Z', 'a'..'z'];
   AlphaNum = Alpha + Numeric;
```

Konstante Ausdrücke

Ein *konstanter Ausdruck* kann vom Compiler ohne Ausführung des Programms ausgewertet werden. Als konstante Ausdrücke können Zahlen, Zeichen-Strings, echte Konstanten, Werte von Aufzählungstypen und die speziellen Konstanten *True*, *False* und **nil** verwendet werden. Zulässig sind auch Ausdrücke, die sich aus diesen Elementen zusammensetzen und Operatoren, Typumwandlungen und Mengenkonstruktoren enthalten. In konstanten Ausdrücken können keine Variablen, Zeiger und Funktionsaufrufe verwendet werden. Eine Ausnahme bilden Aufrufe der folgenden, vordefinierten Funktionen:

Abs	*Exp*	*Length*	*Ord*	*Sqr*
Addr	*Frac*	*Ln*	*Pred*	*Sqrt*
ArcTan	*Hi*	*Lo*	*Round*	*Succ*
Chr	*High*	*Low*	*Sin*	*Swap*
Cos	*Int*	*Odd*	*SizeOf*	*Trunc*

Die Definition eines *konstanten Ausdrucks* wird in verschiedenen Bereichen der Syntaxdefinition von Object Pascal verwendet. Konstante Ausdrücke werden zur Initialisierung globaler Variablen, zur Definition von Teilbereichstypen, zur Festlegung von Standardparameterwerten, zur Erstellung von **case**-Anweisungen und zur Deklaration von echten und typisierten Konstanten eingesetzt.

Hier einige Beispiele für konstante Ausdrücke:

```
100
'A'
```

```
256 - 1
(2.5 + 1) / (2.5 - 1)
'Borland' + ' ' + 'Delphi'
Chr(32)
Ord('Z') - Ord('A') + 1
```

Ressourcen-Strings

Ressourcen-Strings werden als Ressource gespeichert und zu einer ausführbaren Datei oder Bibliothek gelinkt. Deshalb ist keine erneute Compilierung des Programms erforderlich, wenn ein Ressourcen-String geändert wird. Weitere Informationen finden Sie in den Hilfethemen zur Lokalisierung von Anwendungen.

Ressourcen-Strings werden wie echte Konstanten deklariert, wobei das Wort **const** durch **resourcestring** ersetzt wird. Der Ausdruck rechts neben dem Symbol **=** muß ein konstanter Ausdruck sein, der als Ergebnis einen String-Wert liefert:

```
resourcestring
   CreateError = 'Cannot create file %s';       {  Einzelheiten zu Formatbezeichnern
                                                   finden Sie unter 'Format-Strings'}
   OpenError = 'Cannot open file %s';              in der Online-Hilfe. }
   LineTooLong = 'Line too long';
   ProductName = 'Borland Delphi\000\000';
   SomeResourceString = SomeTrueConstant;
```

Der Compiler löst Konflikte, die aufgrund doppelter Ressourcen-Strings in verschiedenen Bibliotheken auftreten, automatisch auf.

Typisierte Konstanten

Typisierte Konstanten können im Gegensatz zu echten Konstanten auch Werte mit Array-, Record-, Zeiger- und prozeduralem Typ enthalten. Konstante Ausdrücke dürfen keine typisierten Konstanten enthalten.

Im Standardstatus des Compilers **({$J+})** können typisierten Konstanten neue Werte zugewiesen werden. Sie entsprechen damit initialisierten Variablen. Ist dagegen die Compiler-Direktive **{$J–}** aktiviert, können die Werte typisierter Konstanten zur Laufzeit nicht geändert werden. Sie verhalten sich dann wie schreibgeschützte Variablen.

Typisierte Konstanten werden folgendermaßen deklariert:

```
const Bezeichner: Typ = Wert
```

Dabei ist *Bezeichner* ein gültiger Bezeichner, *Typ* ist jeder beliebige Typ mit Ausnahme von Datei und Variante, und *Wert* ist ein Ausdruck des Typs *Typ*. Beispiel:

```
const Max: Integer = 100;
```

In den meisten Fällen muß *Wert* ein konstanter Ausdruck sein. Wenn für *Typ* ein Array-, Record-, Zeiger- oder prozeduraler Typ angegeben wird, gelten spezielle Regeln.

Array-Konstanten

Bei der Deklaration einer Array-Konstante werden die Werte der Array-Elemente in Klammern und durch Kommas getrennt angegeben. Zur Angabe der Werte müssen konstante Ausdrücke verwendet werden. Die folgende Anweisung deklariert eine typisierte Konstante mit dem Namen *Digits*, die ein Zeichen-Array enthält:

```
const Digits: array[0..9] of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

Da nullbasierte Zeichen-Arrays oft nullterminierte Strings darstellen, können zur Initialisierung von Zeichen-Arrays Konstanten verwendet werden. Die obige Deklaration läßt sich folgendermaßen vereinfachen:

```
const Digits: array[0..9] of Char = '0123456789';
```

Konstanten, die mehrdimensionale Arrays darstellen, werden deklariert, indem die Konstanten jeder einzelnen Dimension in Klammern gesetzt und durch Kommas voneinander getrennt werden. Die Deklaration

```
type TCube = array[0..1, 0..1, 0..1] of Integer;
const Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6,7)));
```

erstellt ein Array namens *Maze* mit den folgenden Werten:

Maze[0,0,0] = 0

Maze[0,0,1] = 1

Maze[0,1,0] = 2

Maze[0,1,1] = 3

Maze[1,0,0] = 4

Maze[1,0,1] = 5

Maze[1,1,0] = 6

Maze[1,1,1] = 7

Array-Konstanten dürfen auf keiner Ebene Dateitypen enthalten.

Record-Konstanten

Bei der Deklaration einer Record-Konstante geben Sie am Ende der Deklaration den Wert jedes Feldes in Klammern an. Die Angabe erfolgt in der Form *Feldname*: *Wert*, wobei die einzelnen Feldzuweisungen durch Strichpunkte voneinander getrennt werden. Die Werte müssen durch konstante Ausdrücke dargestellt werden. Die Reihenfolge der Felder muß der Reihenfolge bei der Deklaration des Record-Typs entsprechen. Wenn ein Tag-Feld vorhanden ist, muß dessen Wert angegeben werden. Enthält der Record einen varianten Teil, können Werte nur an die Variante zugewiesen werden, die durch das Tag-Feld angegeben ist.

Einige Beispiele für Record-Konstanten:

```
type
   TPoint = record
    X, Y: Single;
   end;
```

```
    TVector = array[0..1] of TPoint;
    TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
    TDate = record
     D: 1..31;
     M: TMonth;
     Y: 1900..1999;
    end;
const
    Origin: TPoint = (X: 0.0; Y: 0.0);
    Line: TVector = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
    SomeDay: TDate = (D: 2; M: Dec; Y: 1960);
```

Record-Konstanten dürfen auf keiner Ebene Dateitypen enthalten.

Prozedurale Konstanten

Zur Deklaration einer prozeduralen Konstanten geben Sie den Namen einer Funktion oder Prozedur an, die mit dem deklarierten Typ der Konstanten kompatibel ist:

```
function Calc(X, Y: Integer): Integer;
begin
   ⋮
end;
type TFunction = function(X, Y: Integer): Integer;
const MyFunction: TFunction = Calc;
```

Die prozedurale Konstante *MyFunction* kann dann in einem Funktionsaufruf eingesetzt werden:

```
I := MyFunction(5, 7)
```

Prozeduralen Konstanten kann auch der Wert **nil** zugewiesen werden.

Zeigerkonstanten

Bei der Deklaration einer Zeigerkonstanten muß diese mit einem Wert initialisiert werden, der vom Compiler (zumindest als relative Adresse) aufgelöst werden kann. Die Initialisierung der Konstanten kann mit dem Operator **@**, mit **nil** oder (bei PChar-Konstanten) mit einem String-Literal erfolgen. Ist z.B. I eine globale Integer-Variable, können Sie folgende Konstante deklarieren:

```
const PI: ^Integer = @I;
```

Die Konstante kann vom Compiler aufgelöst werden, da globale Variablen Bestandteil des Codesegments sind. Dasselbe gilt für Funktionen und globale Konstanten:

```
const PF: Pointer = @MyFunction;
```

Da String-Literale als globale Konstanten zugewiesen werden, können Sie eine PChar-Konstante mit einem String-Literal initialisieren:

```
const WarningStr: PChar = 'Warning!';
```

Adressen von lokalen (auf dem Stack zugewiesenen) und dynamischen (auf dem Heap zugewiesenen) Variablen können nicht an Zeigerkonstanten zugewiesen werden.

Kapitel

6

Prozeduren und Funktionen

Prozeduren und Funktionen, die zusammenfassend auch als *Routinen* bezeichnet werden, sind abgeschlossene Anweisungsblöcke, die von unterschiedlichen Positionen in einem Programm aufgerufen werden können. Eine *Funktion* ist eine Routine, die nach Ausführung einen Wert zurückgibt. Eine *Prozedur* ist eine Routine, die keinen Wert zurückgibt.

Funktionsaufrufe können auch in Zuweisungen und Operationen eingesetzt werden, da sie einen Wert zurückgeben. Ein Beispiel:

```
I := SomeFunction(X);
```

Diese Anweisung ruft die Funktion *SomeFunction* auf und weist das Ergebnis der Variablen *I* zu. Funktionsaufrufe dürfen nicht auf der linken Seite einer Zuweisungsanweisung stehen.

Funktions- und Prozeduraufrufe können als eigenständige Anweisungen verwendet werden:

```
DoSomething;
```

Diese Anweisung ruft die Routine *DoSomething* auf. Ist *DoSomething* eine Funktion, wird der Rückgabewert verworfen.

Prozeduren und Funktionen können auch rekursiv aufgerufen werden.

Prozeduren und Funktionen deklarieren

Beim Deklarieren einer Prozedur oder Funktion geben Sie den Namen, die Anzahl und den Typ der Parameter und - wenn es sich um eine Funktion handelt - den Typ des Rückgabewertes an. Dieser Teil der Deklaration wird auch *Prototyp, Einleitung* oder *Kopf* genannt. Anschließend schreiben Sie den Code, der beim Aufrufen der Prozedur oder Funktion ausgeführt werden soll. Dieser Teil wird auch als Rumpf oder Block der Routine bezeichnet.

Die Standardprozedur *Exit* kann jederzeit im Rumpf einer Funktion oder Prozedur aufgerufen werden. *Exit* beendet die Ausführung der betreffenden Routine und gibt die Steuerung wieder an die Routine zurück, die diese Funktion oder Prozedur aufgerufen hat.

Prozedurdeklarationen

Eine Prozedurdeklaration hat folgende Form:

```
procedure Prozedurname(Parameterliste); Direktiven;
lokale Deklarationen;
begin
   Anweisungen
end;
```

Prozedurname ist ein beliebiger gültiger Bezeichner, *Anweisungen* eine Folge von Anweisungen, die beim Aufruf der Prozedur ausgeführt werden. *Parameterliste, Direktiven* und *lokale Deklarationen* sind optional.

- Informationen zur *Parameterliste* finden Sie im Abschnitt »Parameter« auf Seite 6-10.
- Informationen zu *Direktiven* finden Sie in den Abschnitten »Aufrufkonventionen« auf Seite 6-5, »forward- und interface-Deklarationen« auf Seite 6-6, »external-Deklarationen« auf Seite 6-7 und »Prozeduren und Funktionen überladen« auf Seite 6-8. Mehrere Direktiven werden durch Semikolons voneinander getrennt.
- Informationen zu lokale *Deklarationen*, die der Deklaration lokaler Bezeichner dienen, finden Sie im Abschnitt »Lokale Deklarationen« auf Seite 6-9.

Ein Beispiel für eine Prozedurdeklaration:

```
procedure NumString(N: Integer; var S: string);
var
   V: Integer;
begin
   V := Abs(N);
   S := '';
   repeat
    S := Chr(V mod 10 + Ord('0')) + S;
    V := V div 10;
  until V = 0;
  if N < 0 then S := '-' + S;
end;
```

Nach dieser Deklaration können Sie die Prozedur *NumString* folgendermaßen aufrufen:

```
NumString(17, MyString);
```

Dieser Prozeduraufruf weist der Variablen *MyString* (es muß sich um eine Variable vom Typ **String** handeln) den Wert '17' zu.

Im Anweisungsblock einer Prozedur können Sie Variablen und Bezeichner verwenden, die im Abschnitt *lokale Deklarationen* der Prozedur deklariert wurden. Sie können außerdem die Parameternamen aus der Parameterliste (N und S in diesem Bei-

spiel) verwenden. Die Parameterliste definiert eine Gruppe lokaler Variablen. Aus diesem Grund dürfen im Abschnitt *lokale Deklarationen* keine gleichlautenden Parameternamen auftauchen. Natürlich können Sie auch alle Bezeichner verwenden, in deren Gültigkeitsbereich die Prozedur deklariert wurde.

Funktionsdeklarationen

Eine Funktionsdeklaration entspricht einer Prozedurdeklaration, definiert aber zusätzlich einen Rückgabetyp und einen Rückgabewert. Funktionsdeklarationen haben folgende Form:

```
function Funktionsname(Parameterliste): Rückgabetyp; Direktiven;
lokale Deklarationen;
begin
   Anweisungen
end;
```

Funktionsname ist ein beliebiger gültiger Bezeichner, *Rückgabetyp* ein beliebiger Typ, *Anweisungen* enthält die Folge von Anweisungen, die beim Aufruf der Funktion ausgeführt werden sollen. *Parameterliste*, *Direktiven* und *lokale Deklarationen* sind optional.

- Information zur *Parameterliste* finden Sie im Abschnitt »Parameter« auf Seite 6-10.
- Informationen zu *Direktiven* finden Sie in den Abschnitten »Aufrufkonventionen« auf Seite 6-5, »forward- und interface-Deklarationen« auf Seite 6-6, »external-Deklarationen« auf Seite 6-7 und »Prozeduren und Funktionen überladen« auf Seite 6-8. Mehrere Direktiven werden durch Semikolons voneinander getrennt.
- Informationen über *lokale Deklarationen*, die lokale Bezeichner deklarieren, finden Sie im Abschnitt »Lokale Deklarationen« auf Seite 6-9.

Für den Anweisungsblock der Funktion gelten die Regeln, die bereits für Prozeduren erläutert wurden. Sie können Variablen und Bezeichner verwenden, die im Abschnitt *lokale Deklarationen* der Funktion deklariert wurden. Sie können außerdem die Parameternamen aus der Parameterliste verwenden. Natürlich können Sie auch alle Bezeichner verwenden, in deren Gültigkeitsbereich die Funktion deklariert wurde. Der Funktionsname ist eine spezielle Variable, die wie die vordefinierte Variable *Result* den Rückgabewert der Funktion enthält.

Ein Beispiel:

```
function WF: Integer;
begin
  WF := 17;
end;
```

Diese Deklaration definiert eine Konstantenfunktion namens *WF*, die keine Parameter entgegennimmt und immer den Integerwert 17 zurückgibt. Diese Deklaration ist zur folgenden äquivalent:

```
function WF: Integer;
begin
  Result := 17;
end;
```

Das nächste Beispiel enthält eine etwas komplexere Funktionsdeklaration:

```
function Max(A: array of Real; N: Integer): Real;
var
  X: Real;
  I: Integer;
begin
  X := A[0];
  for I := 1 to N - 1 do
    if X < A[I] then X := A[I];
  Max := X;
end;
```

Sie können der Variablen *Result* oder dem Funktionsnamen im Anweisungsblock mehrmals einen Wert zuweisen. Die zugewiesenen Werte müssen jedoch dem deklarierten Rückgabetyp entsprechen. Sobald die Ausführung der Funktion beendet wird, bildet der Wert, der zuletzt der Variablen *Result* oder dem Funktionsnamen zugewiesen wurde, den Rückgabewert der Funktion. Ein Beispiel:

```
function Power(X: Real; Y: Integer): Real;
var
   I: Integer;
begin
   Result := 1.0;
   I := Y;
  while I > 0 do
  begin
    if Odd(I) then Result := Result * X;
    I := I div 2;
    X := Sqr(X);
  end;
end;
```

Result und der Funktionsname repräsentieren immer denselben Wert:

```
function MyFunction: Integer;
begin
   MyFunction := 5;
   Result := Result * 2;
   MyFunction := Result + 1;
end;
```

Diese Funktion gibt den Wert 11 zurück. *Result* ist jedoch nicht mit dem Funktionsnamen identisch. Wenn der Funktionsname auf der linken Seite einer Zuweisungsanweisung verwendet wird, verwaltet der Compiler den Funktionsnamen als Variable (wie *Result*) zur Aufzeichnung des Rückgabewerts. Taucht der Funktionsname dagegen an einer anderen Stelle im Anweisungsblock auf, wird der Funktionsname vom Compiler als rekursiver Aufruf der Funktion interpretiert. *Result* kann dagegen als Variable in Operationen eingesetzt werden, beispielsweise bei Typkonvertierungen, in Konstruktoren, Indizes und in Aufrufen anderer Routinen.

Result wird implizit in jeder Funktion deklariert. Sie dürfen diese Variable deshalb nicht manuell deklarieren.

Wenn die Ausführung einer Funktion beendet wird, bevor *Result* oder dem Funktionsnamen ein Wert zugewiesen wurde, ist der Rückgabewert der Funktion nicht definiert.

Aufrufkonventionen

Wenn Sie eine Prozedur oder Funktion deklarieren, können Sie eine *Aufrufkonvention* mit einer der Direktiven **register, pascal**, **cdecl**, **stdcall** und **safecall** angeben. Ein Beispiel:

```
function MyFunction(X, Y: Real): Real; cdecl;
  ⋮
```

Aufrufkonventionen bestimmen die Reihenfolge, in der Parameter an die Routine übergeben werden. Sie beeinflussen außerdem das Entfernen der Parameter vom Stack, den Einsatz der Register zur Übergabe von Parametern sowie die Fehler- und Exception-Verarbeitung. Die Standard-Aufrufkonvention ist **register**.

- Die Konventionen **register** und **pascal** übergeben Parameter von links nach rechts. Der links stehende Parameter wird also zuerst ausgewertet und übergeben, der rechts stehende Parameter zuletzt. Die Konventionen **cdecl**, **stdcall** und **safecall** übergeben die Parameter von rechts nach links.
- Bei allen Konventionen außer **cdecl** entfernt die Prozedur bzw. Funktion die Parameter vom Stack, sobald die Steuerung zurückgegeben wird. Wird die Konvention cdecl verwendet, entfernt die aufrufende Routine die Parameter vom Stack, sobald sie wieder die Steuerung erhält.
- Die Konvention **register** verwendet bis zu drei CPU-Register zur Übergabe der Parameter, alle anderen Konventionen übergeben die Parameter an den Stack.
- Die Konvention **safecall** implementiert die Verarbeitung von COM-Fehlern und Exceptions.

Die folgende Tabelle beschreibt die Aufrufkonventionen.

Tabelle 6.1 Aufrufkonventionen

Direktive	Parameterreihenfolge	Bereinigung	Parameterübergabe in Registern?
register	Von links nach rechts	Routine	Ja
pascal	Von links nach rechts	Routine	Nein
cdecl	Von rechts nach links	Aufrufer	Nein
stdcall	Von rechts nach links	Routine	Nein
safecall	Von rechts nach links	Routine	Nein

Die weitaus effizienteste Aufrufkonvention ist **register**, da hier meistens kein Stack-Rahmen für die Parameter angelegt werden muß. Die Konvention **cdecl** ist hilfreich, wenn Funktionen in DLLs aufgerufen werden, die in C oder C++ geschrieben wurden. **stdcall** und **safecall** werden für Aufrufe der Windows-API verwendet. Die Konvention **safecall** muß für die Deklaration dualer Schnittstellenmethoden verwendet werden (siehe Kapitel 10, »Objektschnittstellen«). Die Konvention **pascal** gewährlei-

stet die Abwärtskompatibilität. Weitere Informationen zu Aufrufkonventionen finden Sie in Kapitel 12, »Ablaufsteuerung«.

Die Direktiven **near** und **far** und **export** beziehen sich auf die Aufrufkonventionen bei der Programmierung für 16-Bit-Windows-Umgebungen. Sie dienen ausschließlich der Abwärtskompatibilität und wirken sich nicht auf 32-Bit-Anwendungen aus.

forward- und interface-Deklarationen

Die Direktive **forward** in einer Prozedur- oder Funktionsdeklaration wird durch einen Rumpf mit Anweisungen und lokalen Variablendeklarationen ersetzt. Ein Beispiel:

```
function Calculate(X, Y: Integer): Real; forward;
```

Die hier deklarierte Funktion *Calculate* enthält die Direktive **forward**. An einer anderen Position muß der Rumpf der Routine deklariert werden. Diese definierende Deklaration kann beispielsweise folgendermaßen aussehen:

```
function Calculate;
   ⋮   { Deklarationen }
begin
   ⋮   { Anweisungsblock }
end;
```

Grundsätzlich braucht eine definierende Deklaration die Parameterliste oder den Rückgabetyp der Routine nicht zu wiederholen. Werden diese Angaben jedoch wiederholt, müssen Sie denen in der **forward**-Deklaration exakt entsprechen. Standardparameter in der definierenden Deklaration werden allerdings ignoriert. Wenn die **forward**-Deklaration eine überladene Prozedur oder Funktion definiert (siehe »Prozeduren und Funktionen überladen« auf Seite 6-8), muß die Parameterliste in der definierenden Deklaration wiederholt werden.

Zwischen einer **forward**-Deklaration und der zugehörigen definierenden Deklaration dürfen sich außer anderen Deklarationen keine weiteren Quelltextkomponenten befinden. Die definierende Deklaration kann vom Typ **external** oder **assembler** sein, es darf sich jedoch nicht um eine weitere **forward**-Deklaration handeln.

Mit einer **forward**-Deklaration kann der Gültigkeitsbereich eines Prozedur- oder Funktionsbezeichners auf einen früheren Punkt im Quelltext ausgedehnt werden. Andere Prozeduren und Funktionen können also die mit **forward** deklarierte Routine aufrufen, bevor sie tatsächlich definiert wurde. **forward**-Deklarationen bieten also nicht nur mehr Flexibilität bei der Programmierung, sie werden auch gelegentlich für Rekursionen benötigt.

Die Direktive **forward** darf nicht im **interface**-Abschnitt einer Unit verwendet werden. Prozedur- und Funktions-Header im **interface**-Abschnitt verhalten sich wie **forward**-Deklarationen. Die zugehörigen definierenden Deklarationen müssen in den **implementation**-Abschnitt eingefügt werden. Eine im **interface**-Abschnitt deklarierte Routine ist von jeder Position in der Unit und für jede Unit und jedes Programm verfügbar, die bzw. das die Unit mit der Deklaration einbindet.

external-Deklarationen

Die Direktive **external** ersetzt den Anweisungsblock in einer Prozedur- oder Funktionsdeklaration. Sie ermöglicht das Aufrufen von Prozeduren und Funktionen, die unabhängig vom aktuellen Programm compiliert wurden.

OBJ-Dateien linken

Um Routinen aus einer separat compilierten OBJ-Datei aufrufen zu können, müssen Sie die OBJ-Datei mit der Compiler-Direktive **$L** (oder $**LINK**) zur gewünschten Anwendung linken:

```
{$L BLOCK.OBJ}
```

Diese Anweisung linkt die Datei BLOCK.OBJ zu dem Programm bzw. der Unit, in der die Anweisung verwendet wird. Anschließend müssen Sie die aufzurufenden Funktionen und Prozeduren deklarieren:

```
procedure MoveWord(var Source, Dest; Count: Integer); external;
procedure FillWord(var Dest; Data: Integer; Count: Integer); external;
```

Jetzt können Sie die Routinen *MoveWord* und *FillWord* in BLOCKOBJ aufrufen.

Deklarationen wie die oben gezeigten werden häufig verwendet, um auf externe Assembler-Routinen zuzugreifen. Informationen zum Einfügen von Assembler-Routinen in Object-Pascal-Quelltext finden Sie in Kapitel 13, »Der integrierte Assembler«.

Funktionen aus DLLs importieren

Mit einer Direktive der Form

```
external Stringkonstante;
```

können Sie Routinen aus einer DLL importieren. Die Anweisung muß an das Ende des Prozedur- bzw. Funktions-Header angefügt werden. *Stringkonstante* gibt den Namen der DLL-Datei in Hochkommas an. Ein Beispiel:

```
function SomeFunction(S: string): string; external 'strlib.dll';
```

Mit dieser Anweisung wird die Funktion *SomeFunction* aus der Datei STRLIB.DLL importiert.

Sie können Routinen unter einem anderen als dem in der DLL verwendeten Namen importieren, indem Sie den Originalnamen in der **external**-Direktive angeben:

```
external Stringkonstante1 name Stringkonstante2;
```

Die erste *Stringkonstante* gibt den Namen der DLL-Datei, die zweite den Originalnamen der Routine an. Die folgende Deklaration importiert beispielsweise eine Funktion aus USER32.DLL (Teil der Windows-API):

```
function MessageBox(HWnd: Integer; Text, Caption: PChar; Flags: Integer): Integer;
  stdcall; external 'user32.dll' name 'MessageBoxA';
```

Der Orginalname der Funktion lautet *MessageBoxA*, sie wird jedoch unter dem Namen *MessageBox* importiert.

Sie können die zu importierende Routine auch über eine Nummer angeben:

```
external Stringkonstante index Integerkonstante;
```

Integerkonstante ist der Index der Routine in der Exporttabelle der DLL.

In der Importdeklaration müssen Sie die genaue Schreibweise des Routinennamens verwenden (einschließlich Groß-/Kleinschreibung). Beim Aufruf der importierten Routine wird die Groß-/Kleinschreibung nicht mehr berücksichtigt.

Weitere Informationen zu DLLs finden Sie in Kapitel 9, »Dynamische Link-Bibliotheken und Packages«.

Prozeduren und Funktionen überladen

Sie können mehrere Routinen mit identischen Namen in demselben Gültigkeitsbereich deklarieren. Dieses Verfahren wird *Überladen* genannt. Überladene Routinen müssen mit der Direktive **overload** deklariert werden und unterschiedliche Parameterlisten haben:

```
function Divide(X, Y: Real): Real; overload;
begin
  Result := X/Y;
end;

function Divide(X, Y: Integer): Integer; overload;
begin
  Result := X div Y;
end;
```

Diese Deklarationen definieren zwei Funktionen namens *Divide*, die Parameter unterschiedlicher Typen entgegennehmen. Wenn Sie *Divide* aufrufen, ermittelt der Compiler die zu verwendende Funktion durch Prüfung des übergebenen Parametertyps. `Divide(6.0, 3.0)` ruft beispielsweise die erste Divide-Funktion auf, da es sich bei den Argumenten um reelle Zahlen handelt.

Überladene Routinen müssen hinsichtlich der Anzahl der entgegengenommenen Parameter oder der Typen dieser Parameter eindeutig sein. Die folgenden beiden Deklarationen führen deshalb zu einem Fehler bei der Compilierung:

```
function Cap(S: string): string; overload;
   ⋮
procedure Cap(var Str: string); overload;
   ⋮
```

Dagegen sind diese Deklarationen zulässig:

```
function Func(X: Real; Y: Integer): Real; overload;
   ⋮
function Func(X: Integer; Y: Real): Real; overload;
   ⋮
```

Wird eine überladene Routine mit **forward** oder **interface** deklariert, muß die Parameterliste in der definierenden Deklaration der Routine wiederholt werden.

Bei Verwendung von Standardparametern in überladenen Routinen müssen Sie mehrdeutige Parametersignaturen vermeiden. Weitere Informationen finden Sie im Abschnitt »Standardparameter und überladene Routinen« auf Seite 6-17.

Die möglichen Auswirkungen des Überladens lassen sich beschränken, indem Sie beim Aufruf den qualifizierten Routinennamen verwenden. Mit `Unit1.MyProcedure(X, Y)` können nur die in *Unit1* deklarierte Routinen aufgerufen werden. Entsprechen der Name und die Parameterliste keiner Routine in *Unit1*, gibt der Compiler einen Fehler aus.

Weitere Informationen zu überladenen Methoden in einer Klassenhierarchie finden Sie im Abschnitt »Methoden überladen« auf Seite 7-13.

Lokale Deklarationen

Der Rumpf einer Funktion oder Prozedur beginnt in vielen Fällen mit der Deklaration lokaler Variablen, die im Anweisungsblock der Routine verwendet werden. Sie können beispielsweise Konstanten, Typen und andere Routinen deklarieren. Der Gültigkeitsbereich lokaler Bezeichner ist auf die Routine beschränkt, in der sich die Deklaration befindet.

Verschachtelte Routinen

Funktionen und Prozeduren können im Abschnitt mit den lokalen Deklarationen ihrerseits Funktionen und Prozeduren enthalten. Die folgende Deklaration der Prozedur *DoSomething* enthält beispielsweise eine verschachtelte Prozedur:

```
procedure DoSomething(S: string);
var
  X, Y: Integer;

  procedure NestedProc(S: string);
  begin
    ⋮
  end;

begin
   ⋮
  NestedProc(S);
   ⋮
end;
```

Der Gültigkeitsbereich einer verschachtelten Routine ist auf die Funktion bzw. Prozedur beschränkt, in der sie deklariert ist. In obigem Beispiel kann *NestedProc* nur in *DoSomething* aufgerufen werden.

Echte Beispiele verschachtelter Routinen sind die Prozedur *DateTimeToString*, die Funktion *ScanDate* und andere Routinen in der Unit *SysUtils*.

Parameter

Die meisten Prozedur- und Funktions-Header enthalten eine *Parameterliste*. Im Header

```
function Power(X: Real; Y: Integer): Real;
```

lautet die Parameterliste beispielsweise `(X: Real; Y: Integer)`.

Eine Parameterliste ist eine Folge von Parameterdeklarationen, die durch Semikolons voneinander getrennt und in Klammern eingeschlossen werden. Jede Deklaration besteht aus einer Reihe von Parameternamen, die durch Kommas voneinander getrennt werden. Hinter den Parameternamen steht in den meisten Fällen ein Doppelpunkt und ein Typbezeichner, in Einzelfällen außerdem das Gleichheitszeichen (=) und ein Standardwert. Parameternamen müssen gültige Bezeichner sein. Jeder Deklaration kann eines der reservierten Wörter **var**, **const** und **out** vorangestellt werden.

Einige Beispiele:

```
(X, Y: Real)
(var S: string; X: Integer)
(HWnd: Integer; Text, Caption: PChar; Flags: Integer)
(const P; I: Integer)
```

Die Parameterliste gibt Anzahl, Reihenfolge und Typ der Parameter an, die beim Aufruf an die Routine übergeben werden müssen. Wenn eine Routine keine Parameter entgegennimmt, geben Sie in der Deklaration weder die Bezeichnerliste noch die Klammern an:

```
procedure UpdateRecords;
begin
   ⋮
end;
```

Im Rumpf der Prozedur oder Funktion können die Parameternamen (*X* und *Y* im ersten Beispiel) als lokale Variablen eingesetzt werden. Sie dürfen die Parameternamen im Abschnitt mit den lokalen Deklarationen nicht erneut deklarieren.

Parametersemantik

Parameter können auf verschiedene Weise kategorisiert werden:

- Jeder Parameter ist als *Wert-*, *Variablen-*, *Konstanten-* oder *Ausgabeparameter* klassifiziert. Wertparameter werden verwendet, sofern nicht mit den reservierten Wörtern **var, const** oder **out** einer der anderen Typen angegeben wird.
- Wertparameter sind immer *typisiert*, während Variablen-, Konstanten und Ausgabeparameter auch *untypisiert* sein können .
- Für Array-Parameter gelten spezielle Regeln. Weitere Informationen finden Sie unter »Array-Parameter« auf Seite 6-14.

Dateien und Instanzen strukturierter Typen, die Dateien enthalten, können nur als Variablenparameter (**var**) übergeben werden.

Wert- und Variablenparameter

Die meisten Parameter sind Wert- (Standard) oder Variablenparameter (**var**). Wertparameter werden als *Wert* übergeben, Variablenparameter als *Referenz*. Der Unterschied wird anhand der folgenden beiden Funktionen deutlich:

```
function DoubleByValue(X: Integer): Integer;     // X ist ein Wertparameter
begin
   X := X * 2;
   Result := X;
end;
function DoubleByRef(var X: Integer): Integer;              // X ist ein
                                                            // Variablenparameter
begin
   X := X * 2;
   Result := X;
end;
```

Diese Funktionen liefern das gleiche Ergebnis. Nur die zweite Funktion (*DoubleByRef*) kann jedoch den Wert der an sie übergebenen Variablen ändern. Die Funktionen können beispielsweise folgendermaßen aufgerufen werden:

```
var
   I, J, V, W: Integer;
begin
   I := 4;
   V := 4;
   J := DoubleByValue(I);   // J = 8, I = 4
   W := DoubleByRef(V);     // W = 8, V = 8
end;
```

Nachdem diese Anweisungen ausgeführt wurden, enthält die an *DoubleByValue* übergebene Variable *I* immer noch den ursprünglichen Wert. Die an *DoubleByRef* übergebene Variable *V* enthält dagegen einen neuen Wert.

Ein Wertparameter verhält sich wie eine lokale Variable, die durch den im Aufruf der Funktion oder Prozedur übergebenen Wert initialisiert wird. Wenn Sie eine Variable als Wertparameter übergeben, erstellt die Prozedur oder Funktion eine Kopie dieser Variablen. Änderungen des Wertes dieser Variablen wirken sich nicht auf die ursprüngliche Variable aus. Sie werden verworfen, sobald die Steuerung wieder an den Aufrufer zurückgegeben wird.

Ein Variablenparameter ist dagegen mit einem Zeiger vergleichbar. Änderungen des Parameters im Rumpf einer Prozedur oder Funktion bleiben deshalb erhalten, wenn die Programmausführung wieder dem Aufrufer übergeben wird und der Parametername nicht mehr im Gültigkeitsbereich liegt.

Auch wenn dieselbe Variable in mehreren **var**-Parametern übergeben wird, werden keine Kopien erstellt. Dieses Merkmal wird im folgenden Beispiel illustriert:

```
procedure AddOne(var X, Y: Integer);
begin
   X := X + 1;
   Y := Y + 1;
end;
var I: Integer;
```

```
begin
   I := 1;
   AddOne(I, I);
end;
```

Nachdem dieser Code ausgeführt wurde, enthält die Variable *I* den Wert 3.

Enthält die Deklaration einer Routine einen **var**-Parameter, müssen Sie im Aufruf der Routine einen Ausdruck übergeben, dem ein Wert zugewiesen werden kann, also eine Variable, einen dereferenzierten Zeiger, ein Feld oder eine indizierte Variable. Im Beispiel weiter oben würde `DoubleByRef(7)` einen Fehler generieren, während der Aufruf `DoubleByValue(7)` zulässig wäre.

In **var-P**arametern (beispielsweise `DoubleByRef(MyArray[I])`) übergebene Indizes und dereferenzierte Zeiger werden vor der Ausführung der Routine einmal ausgewertet.

Konstantenparameter

Ein Konstantenparameter (**const**) entspricht einer lokalen bzw. schreibgeschützten Variablen. Konstantenparameter entsprechen weitgehend Wertparametern. Sie können ihnen jedoch im Rumpf einer Prozedur oder Funktion keinen Wert zuweisen und sie nicht als **var**-Parameter an eine andere Routine übergeben. Übergeben Sie eine Objektreferenz als Konstantenparameter, können Sie weiterhin auf die Objekteigenschaften zugreifen und diese ändern.

Die Verwendung von **const** ermöglicht dem Compiler die Optimierung des Codes für strukturierte und String-Parameter. Gleichzeitig wird die versehentliche Übergabe eines Parameters als Referenz an eine andere Routine verhindert.

Das folgende Beispiel ist der Header der Funktion *CompareStr* in der Unit *SysUtils*:

```
function CompareStr(const S1, S2: string): Integer;
```

Da *S1* und *S2* im Rumpf von *CompareStr* nicht geändert werden, können sie als Konstantenparameter deklariert werden.

Ausgabeparameter

Ausgabeparameter werden mit dem Schlüsselwort **out** deklariert und wie Variablenparameter als Referenz übergeben. Der ursprüngliche Wert der referenzierten Variablen wird jedoch verworfen, wenn diese als Ausgabeparameter an eine Routine übergeben wird. Der Ausgabeparameter dient nur der Ausgabe. Er weist die Funktion oder Prozedur an, wo die Ausgabe zu speichern ist, stellt aber keinerlei Eingaben bereit.

Ein Beispiel ist der folgende Prozedur-Header:

```
procedure GetInfo(out Info: SomeRecordType);
```

Wenn *Sie GetInfo* aufrufen, müssen Sie eine Variable des Typs *SomeRecordType* übergeben.

```
var MyRecord: SomeRecordType;
   ⋮
GetInfo(MyRecord);
```

In *MyRecord* werden jedoch keine Daten an die Prozedur *GetInfo* übergeben. *MyRecord* dient nur als Container für die von *GetInfo* generierten Informationen. Beim Aufruf von *GetInfo* wird der von *MyRecord* belegte Speicher sofort freigegeben, noch bevor die Steuerung an die Prozedur übergeben wird.

Ausgabeparameter werden häufig in verteilten Objektmodellen wie COM und CORBA eingesetzt. Sie sollten diesen Parametertyp auch verwenden, wenn Sie nicht initialisierte Variablen an Funktionen oder Prozeduren übergeben.

Untypisierte Parameter

Im Unterschied zu Wertparametern brauchen **Variablen**-, **Konstanten**- und **Ausgabeparameter** nicht typisiert zu werden. Ein Beispiel:

```
procedure TakeAnything(const C);
```

Diese Anweisung deklariert eine Prozedur namens *TakeAnything*, die einen Parameter beliebigen Typs entgegennimmt.

Im Rumpf einer Prozedur oder Funktion sind untypisierte Parameter mit keinem Typ kompatibel. Bevor der untypisierte Parameter bearbeitet werden kann, muß eine Typkonvertierung erfolgen. Der Compiler kann nicht feststellen, ob Operationen, die mit untypisierten Parametern durchgeführt werden, zulässig sind.

Die folgende Beispielfunktion *Equal* vergleicht die angegebene Anzahl Bytes in zwei beliebigen Variablen, die als untypisierte Parameter übergeben werden:

```
function Equal(var Source, Dest; Size: Integer): Boolean;
type
   TBytes = array[0..MaxInt - 1] of Byte;
var
   N: Integer;
begin
   N := 0;
   while (N < Size) and (TBytes(Dest)[N] = TBytes(Source)[N]) do
    Inc(N);
   Equal := N = Size;
end;
```

Nach den Deklarationen

```
type
   TVector = array[1..10] of Integer;
   TPoint = record
    X, Y: Integer;
   end;
var
   Vec1, Vec2: TVector;
   N: Integer;
   P: TPoint;
```

kann die Funktion *Equal* folgendermaßen aufgerufen werden:

```
Equal(Vec1, Vec2, SizeOf(TVector))          // Vec1 mit Vec2 vergleichen
Equal(Vec1, Vec2, SizeOf(Integer) * N)      // Die ersten N Elemente von Vec1 und
                                            // Vec2 vergleichen
```

```
Equal(Vec1[1], Vec1[6], SizeOf(Integer) * 5)  // Die ersten fünf mit den letzten
                                              // fünf Elementen von Vec1 vergleichen
Equal(Vec1[1], P, 4)                          // Vec1[1] mit P.X und Vec1[2] mit P.Y
                                              // vergleichen
```

Array-Parameter

Bei der Deklaration von Routinen, die Array-Parameter entgegennehmen, können Sie in den Parameterdeklarationen keinen Indextyp angeben. Ein Beispiel:

```
procedure Sort(A: array[1..10] of Integer);   // Syntaxfehler
```

Diese Deklaration führt zu einem Compilierungsfehler.

```
type TDigits = array[1..10] of Integer;
procedure Sort(A: TDigits);
```

Diese Deklaration ist dagegen gültig. In den meisten Fällen ist der Einsatz *offener Array-Parameter* jedoch die bessere Lösung.

Offene Array-Parameter

Offene Array-Parameter ermöglichen die Übergabe von Arrays unterschiedlicher Größe an dieselbe Funktion oder Prozedur. Die Definition eines offenen Array-Parameters erfolgt mit der Syntax `array of` *Typ* anstelle der Syntax `array[`*X*`..`*Y*`] of` *Typ* in der Parameterdeklaration. Ein Beispiel:

```
function Find(A: array of Char): Integer;
```

Hier wird eine Funktion namens *Find* deklariert, die ein Zeichen-Array beliebiger Größe entgegennimmt und einen Integerwert zurückgibt.

Hinweis Die Syntax offener Array-Parameter erinnert an dynamische Arrays, obwohl diese beiden Array-Typen nicht identisch sind. Das obige Beispiel deklariert eine Funktion, die ein beliebiges Zeichen-Array entgegennimmt, also auch ein dynamisches Array. Die Deklaration eines Parameters, bei dem es sich um ein dynamisches Array handeln soll, muß mit Angabe eines Typbezeichners erfolgen:

```
type TDynamicCharArray = array of Char;
function Find(A: TDynamicCharArray): Integer;
```

Informationen zu dynamischen Arrays finden Sie im Abschnitt »Dynamische Arrays« auf Seite 5-20.

Im Rumpf einer Routine gelten die folgenden Regeln für offene Array-Parameter.

- Das erste Element trägt immer die Indexnummer 0, das zweite die 1 usw. Die Standardfunktionen *Low* und *High* geben *0* bzw. *Length*–1 zurück. Die Funktion *SizeOf* gibt die Größe des Arrays zurück, das an die Routine übergeben wird.
- Der Zugriff kann nur auf die einzelnen Elemente erfolgen. Zuweisungen an einen offenen Array-Parameter insgesamt sind dagegen nicht zulässig.
- Sie können nur als offene Array-Parameter oder als untypisierte Variablenparameter an andere Prozeduren und Funktionen übergeben werden.

- Anstelle eines Arrays können Sie eine Variable mit dem Basistyp des offenen Array-Parameters übergeben, die dann wie ein Array der Länge 1 verarbeitet wird.

Wenn Sie ein Array als offenen Array-Wertparameter übergeben, erstellt der Compiler im Stack-Bereich der Routine eine lokale Kopie des Arrays. Die Übergabe großer Parameter kann also zu einem Stack-Überlauf führen.

Die folgenden Beispiele verwenden offene Array-Parameter, um eine Prozedur namens *Clear* zu definieren, die jedem Element eines Arrays mit reellen Zahlen den Wert Null zuweist. Außerdem wird die Funktion Sum definiert, mit der die Summe der Elemente in einem Array mit reellen Zahlen ermittelt werden kann.

```
procedure Clear(var A: array of Real);
var
   I: Integer;
begin
   for I := 0 to High(A) do A[I] := 0;
end;
function Sum(const A: array of Real): Real;
var
   I: Integer;
   S: Real;
begin
   S := 0;
   for I := 0 to High(A) do S := S + A[I];
   Sum := S;
end;
```

Wenn Sie Routinen aufrufen, die offene Array-Parameter verarbeiten, können Sie *offene Array-Konstruktoren* übergeben. Weitere Informationen dazu finden Sie im Abschnitt »Offene Array-Konstruktoren« auf Seite 6-19. Ist der Basistyp eines offenen Array-Parameters *Char*, können Sie auch eine String-Konstante an die Routine übergeben, z.B. `SomeFunction('Hello world!')`.

Variante offene Array-Parameter

Variante offene Array-Parameter ermöglichen die Übergabe eines Arrays mit Ausdrücken unterschiedlicher Typen an eine Prozedur oder Funktion. In der Definition einer Routine mit einem varianten offenen Array-Parameter geben Sie als Typ des Parameters **array of const** an:

```
procedure DoSomething(A: array of const);
```

Diese Zeile deklariert eine Prozedur namens *DoSomething*, die auch heterogene Arrays verarbeiten kann.

array of const ist zur Konstruktion **array** of *TVarRec* äquivalent. *TVarRec* ist in der Unit *System* deklariert und repräsentiert einen Record mit variantem Bestandteil, der Werte der Typen Integer, Boolean, Zeichen, Real, String, Zeiger, Klasse, Klassenreferenz, Schnittstelle und Variant aufnehmen kann. Das Feld *VType* im Record *TVarRec* gibt den Typ der einzelnen Elemente im Array an. Einige Typen werden nicht als Wert, sondern als Referenz übergeben. Insbesondere lange Strings werden als *Pointer* übergeben und müssen in den Typ **string** umgewandelt werden. Weitere Informationen finden Sie in der Online-Hilfe zu *TVarRec*.

Das folgende Beispiel verwendet einen varianten offenen Array-Parameter in einer Funktion, die aus jedem Element im Array einen String erzeugt. Die einzelnen Strings werden dann verkettet. Die in dieser Funktion aufgerufenen Routinen zur String-Verarbeitung sind in *SysUtils* definiert.

```
function MakeStr(const Args: array of const): string;
const
   BoolChars: array[Boolean] of Char = ('F', 'T');
var
   I: Integer;
begin
   Result := '';
   for I := 0 to High(Args) do
    with Args[I] do
      case VType of
        vtInteger:    Result := Result + IntToStr(VInteger);
        vtBoolean:    Result := Result + BoolChars[VBoolean];
        vtChar:       Result := Result + VChar;
        vtExtended:   Result := Result + FloatToStr(VExtended^);
        vtString:     Result := Result + VString^;
        vtPChar:      Result := Result + VPChar;
        vtObject:     Result := Result + VObject.ClassName;
        vtClass:      Result := Result + VClass.ClassName;
        vtAnsiString: Result := Result + string(VAnsiString);
        vtCurrency:   Result := Result + CurrToStr(VCurrency^);
        vtVariant:    Result := Result + string(VVariant^);
        vtInt64:      Result := Result + IntToStr(VInt64^);
    end;
end;
```

Der Aufruf dieser Funktion kann mit einem offenen Array-Konstruktor erfolgen (siehe »Offene Array-Konstruktoren« auf Seite 6-19):

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

Dieser Aufruf gibt den String 'test100 T3.14159TForm' zurück.

Standardparameter

Sie können im Header einer Prozedur oder Funktion Standardparameterwerte angeben. Standardwerte sind nur für typisierte Konstanten- und für Wertparameter zulässig. Die Angabe des Standardwertes erfolgt mit dem Gleichheitszeichen (=) hinter der Parameterdeklaration und einem Konstantenausdruck, der zum Typ des Parameters zuweisungskompatibel ist.

Ein Beispiel:

```
procedure FillArray(A: array of Integer; Value: Integer = 0);
```

Nach dieser Deklaration sind die folgenden Prozeduraufrufe äquivalent:

```
FillArray(MyArray);
FillArray(MyArray, 0);
```

In einer Deklaration mehrerer Parameter kann kein Standardwert angegeben werden. Während die Deklaration

```
function MyFunction(X: Real = 3.5: Y: Real = 3.5): Real;
```

zulässig ist, führt die folgende Deklaration zu einem Fehler:

```
function MyFunction(X, Y: Real = 3.5): Real;   // Syntaxfehler
```

Parameter mit Standardwerten müssen am Ende der Parameterliste angegeben werden. Sobald einem Parameter ein Standardwert zugewiesen wurde, müssen Sie auch allen folgenden Parametern Standardwerte zuweisen. Die folgende Deklaration ist aus diesem Grund nicht zulässig:

```
procedure MyProcedure(I: Integer = 1; S: string);   // Syntaxfehler
```

In einem prozeduralen Typ angegebene Werte überladen die in einer Routine angegeben Werte. Auf Grundlage der Deklarationen

```
type TResizer = function(X: Real; Y: Real = 1.0): Real;
function Resizer(X: Real; Y: Real = 2.0): Real;
var
   F: TResizer;
   N: Real;
```

führen die Anweisungen

```
F := Resizer;
F(N);
```

zur Übergabe der Werte (*N*, 1.0) an *Resizer*.

Für Standardparameter dürfen nur Werte verwendet werden, die in Form eines Konstantenausdrucks angegeben werden können. Weitere Informationen dazu finden Sie im Abschnitt »Konstante Ausdrücke« auf Seite 5-44. Für prozedurale Parameter oder Parameter vom Typ dynamisches Array, Klasse, Klassenreferenz oder Schnittstelle kann deshalb nur der Standardwert **nil** verwendet werden. Für Parameter vom Typ Record, Variant, Datei, statisches Array oder Objekt sind keine Standardwerte zulässig.

Informationen zum Aufrufen von Routinen mit Standardparameterwerten finden Sie im Abschnitt »Prozeduren und Funktionen aufrufen« auf Seite 6-18.

Standardparameter und überladene Routinen

Wenn Sie Standardparameterwerte in überladenen Routinen einsetzen, müssen Sie mehrdeutige Parametersignaturen vermeiden. Ein Beispiel:

```
procedure Confused(I: Integer); overload;
   ⋮
procedure Confused(I: Integer; J: Integer = 0); overload;
   ⋮
Confused(X);    //  Welche Prozedur wird aufgerufen?
```

Tatsächlich wird keine der beiden Prozeduren aufgerufen. Diese Zeilen führen zu einem Compilierungsfehler.

Standardparameter in forward- und interface-Deklarationen

Wenn eine Routine eine **forward**-Deklaration enthält oder im **interface**-Abschnitt einer Unit definiert ist, können Sie die Standardparameterwerte nur in der **forward**- bzw. **interface**-Deklaration angeben. Standardwerte in der definierenden Deklaration werden ignoriert. Liegt für eine Routine keine **forward**- oder **interface**-Deklaration vor, kann die definierende Deklaration Standardparameterwerte angeben.

Prozeduren und Funktionen aufrufen

Wenn Sie eine Prozedur oder Funktion aufrufen, wird die Steuerung vom Punkt des Aufrufs an den Rumpf der Routine übergeben. Der Aufruf kann mit dem deklarierten Namen der Routine (mit oder ohne Qualifizierer) oder mit einer prozeduralen Variablen erfolgen, die auf die Routine zeigt. In beiden Fällen müssen im Aufruf Parameter übergeben werden, die in der Reihenfolge und im Typ den in der Parameterliste der Routine angegeben Parametern entsprechen. Die an eine Routine übergebenen Parameter werden auch als *tatsächliche Parameter* bezeichnet - im Gegensatz zu den *formalen Parametern* in der Deklaration der Routine.

Beachten Sie beim Aufrufen einer Routine folgendes:

- Ausdrücke zur Übergabe typisierter Konstanten- und Wertparameter müssen zu den entsprechenden formalen Parametern zuweisungskompatibel sein.
- Ausdrücke zur Übergabe von Variablen- und Ausgabeparametern müssen gegebenenfalls wie die entsprechenden formalen Parameter typisiert sein.
- Variablen- und Ausgabeparameter können nur in zuweisungsfähigen Ausdrükken übergeben werden.
- Wenn die formalen Parameter einer Routine nicht typisiert sind, dürfen Zahlen und echte Konstanten mit numerischen Werten nicht als tatsächliche Parameter verwendet werden.

Wenn Sie eine Routine mit Standardparameterwerten aufrufen, müssen für alle Parameter nach dem ersten akzeptierten Standardwert ebenfalls Standardwerte existieren. Aufrufe der Form `SomeFunction(,,X)` sind nicht zulässig.

Nimmt eine Routine Parameter entgegen, müssen Sie im Aufruf die Klammern angeben, auch wenn alle tatsächlichen Parameter Standardwerte sind. Um beim Aufruf der Prozedur

```
procedure DoSomething(X: Real = 1.0; I: Integer = 0; S: string = '');
```

alle Standardwerte zu übernehmen, geben Sie folgendes ein:

```
DoSomething();
```

Offene Array-Konstruktoren

Offene Array-Konstruktoren ermöglichen die Bildung von Arrays im Aufruf einer Funktion oder Prozedur. Sie müssen als offene Array-Parameter oder variante offene Array-Parameter übergeben werden.

Ein offener Array-Konstruktor ist wie ein Mengenkonstruktor eine Folge von Ausdrücken, die durch Kommas voneinander getrennt und in eckigen Klammern angegeben werden. Ein Beispiel:

```
var I, J: Integer;
procedure Add(A: array of Integer);
```

Nach dieser Deklaration können Sie die Prozedur *Add* mit folgender Anweisung aufrufen:

```
Add([5, 7, I, I + J]);
```

Dieses Verfahren ist zum folgenden äquivalent:

```
var Temp: array[0..3] of Integer;
   ⋮
Temp[0] := 5;
Temp[1] := 7;
Temp[2] := I;
Temp[3] := I + J;
Add(Temp);
```

Offene Array-Konstruktoren können nur als Wert- oder Konstantenparameter übergeben werden. Die Ausdrücke im Konstruktor müssen zum Basistyp des Array-Parameters zuweisungskompatibel sein. Wird ein varianter offener Array-Parameter verwendet, können die Ausdrücke unterschiedliche Typen aufweisen.

Kapitel

7

Klassen und Objekte

Eine *Klasse* (oder ein *Klassentyp*) definiert eine Struktur von *Feldern*, *Methoden* und *Eigenschaften*. Die Instanzen eines Klassentyps heißen *Objekte*. Die Felder, Methoden und Eigenschaften einer Klasse nennt man ihre *Komponenten* oder *Elemente*.

- Felder sind im wesentlichen Variablen, die zu einem Objekt gehören. Sie definieren Datenelemente, die in jeder Instanz der Klasse vorhanden sind.
- Eine Methode ist eine Prozedur oder Funktion, die zu einer bestimmten Klasse gehört. Die meisten Methoden führen Operationen mit Objekten (Instanzen) durch. Manche Methoden arbeiten jedoch mit den Klassentypen selbst.
- Eine Eigenschaft ist eine Schnittstelle zu den Daten eines Objekts (die oftmals in einem Feld gespeichert sind). Eigenschaften verfügen über *Zugriffsangaben*, die bestimmen, wie ihre Daten gelesen und geändert werden. Sie erscheinen für die anderen Bestandteile eines Programms (außerhalb des Objekts) in vielerlei Hinsicht wie ein Feld.

Objekte sind dynamisch zugewiesene Speicherblöcke, deren Struktur durch ihren Klassentyp festgelegt wird. Jedes Objekt verfügt über eine eigene Kopie der in der Klasse definierten Felder. Die Methoden werden jedoch von allen Instanzen gemeinsam genutzt. Das Erstellen und Freigeben von Objekten erfolgt mit Hilfe spezieller Methoden, *den Konstruktoren* und *Destruktoren*.

Eine Klassentypvariable ist eigentlich ein Zeiger auf ein Objekt. Aus diesem Grund können auch mehrere Variablen auf dasselbe Objekt verweisen. Klassentypvariablen können wie andere Zeiger den Wert **nil** annehmen. Sie müssen aber nicht explizit dereferenziert werden, um auf das betreffende Objekt zuzugreifen. So wird beispielsweise durch die Anweisung `MeinObjekt.Size := 100` der Eigenschaft *Size* des Objekts, auf das *MeinObjekt* zeigt, der Wert 100 zugewiesen. Sie brauchen in diesem Fall nicht `MeinObjekt^.Size := 100` anzugeben.

Klassentypen deklarieren

Ein Klassentyp muß deklariert und benannt werden, bevor er instantiiert werden kann (er kann also nicht in einer Variablendeklaration definiert werden). Deklarieren Sie Klassen nur im äußersten Gültigkeitsbereich eines Programms oder einer Unit, nicht in einer Prozedur oder Funktion.

Eine Klassentyp wird folgendermaßen deklariert:

```
type Klassenname = class (Vorfahrklasse)
  Elementliste
  end;
```

Klassenname ist ein beliebiger, gültiger Bezeichner, (*Vorfahrklasse*) ist optional, und *Elementliste* definiert die Felder, Methoden und Eigenschaften der Klasse. Wird keine *Vorfahrklasse* angegeben, erbt die neue Klasse direkt vom vordefinierten Basistyp *TObject*. Wenn Sie eine *Vorfahrklasse* angeben *und die Elementliste* leer ist, brauchen Sie end nicht anzugeben. Eine Klassentypdeklaration kann auch eine Liste von *Schnittstellen* enthalten, die von der Klasse implementiert werden (siehe »Schnittstellen implementieren« auf Seite 10-4).

Methoden werden in einer Klassendeklaration als Funktions- oder Prozedurköpfe ohne Rumpf angegeben. Die definierenden Deklarationen folgen dann an einer anderen Stelle im Programm.

Das folgende Beispiel zeigt die Deklaration der Klasse *TListColumns* in der VCL-Unit *ComCtrls*.

```
type
  TListColumns = class(TCollection)
  private
   FOwner: TCustomListView;
   function GetItem(Index: Integer): TListColumn;
   procedure SetItem(Index: Integer; Value: TListColumn);
  protected
   function GetOwner: TPersistent; override;
   procedure Update(Item: TCollectionItem); override;
  public
   constructor Create(AOwner: TCustomListView);
   function Add: TListColumn;
   property Owner: TCustomListView read FOwner;
   property Items[Index: Integer]: TListColumn read GetItem write SetItem; default;
  end;
```

TListColumns ist von *TCollection* (in der Unit *Classes*) abgeleitet und erbt die meisten Elemente dieser Klasse. Zusätzlich werden jedoch mehrere Eigenschaften und Methoden einschließlich des Konstruktors *Create* definiert. Der Destruktor *Destroy* wird ohne Änderung von *TCollection* übernommen und daher nicht erneut deklariert. Die verschiedenen Elemente sind als **private, protected** oder **public** deklariert (die Klasse verfügt über keine **published**-Elemente). Informationen zu diesen Sichtbarkeitsangaben finden Sie im Abschnitt »Sichtbarkeit von Klassenelementen« auf Seite 7-4.

Ausgehend von dieser Deklaration kann ein *TListColumns*-Objekt folgendermaßen erstellt werden:

```
var ListColumns: TListColumns;
ListColumns := TListColumns.Create(SomeListView);
```

SomeListView ist eine Variable, die ein *TCustomListView*-Objekt enthält.

Vererbung und Gültigkeitsbereich

Beim Deklarieren einer Klasse kann wie im folgenden Beispiel der direkte Vorfahr angegeben werden:

```
type TSomeControl = class(TWinControl);
```

Hier wird die Klasse *TSomeControl* von *TWinControl* abgeleitet. Ein Klassentyp erbt automatisch alle Elemente seines direkten Vorfahren. Außerdem können jederzeit neue Elemente erstellt oder geerbte Elemente neu definiert werden. Es ist aber nicht möglich, Elemente zu entfernen, die in einer Vorfahrklasse definiert wurden. Aus diesem Grund enthält *TSomeControl* alle in der Klasse *TWinControl* und deren Vorfahren definierten Elemente.

Der Gültigkeitsbereich eines Elementbezeichners reicht von seiner Deklaration bis zum Ende der Klassendeklaration und erstreckt sich über alle Nachkommen des Klassentyps und alle in der Klasse und ihren Nachfahren definierten Methoden.

TObject und TClass

Die in der Unit *System* deklarierte Klasse *TObject* ist der absolute Vorfahr aller anderen Klassentypen. Sie definiert nur einige wenige Methoden einschließlich eines Grundkonstruktors und -destruktors. In *System* ist außer *TObject* auch noch der Klassenreferenztyp *TClass* deklariert.

```
TClass = class of TObject;
```

Weitere Informationen zu *TObject* finden Sie in der Online-VCL-Referenz. Hinweise zu Klassenreferenztypen finden Sie im Abschnitt »Klassenreferenzen« auf Seite 7-24.

Wenn Sie in der Deklaration eines Klassentyps keinen Vorfahren angegeben, erbt die Klasse direkt von *TObject*. Aus diesem Grund ist die Deklaration

```
type TMyClass = class
   ⋮
end;
```

identisch mit

```
type TMyClass = class(TObject)
   ⋮
end;
```

Die zweite Variante verdient jedoch aus Gründen der Lesbarkeit den Vorzug.

Kompatibilitätsregeln für Klassentypen

Ein Klassentyp ist mit jedem seiner Vorfahren zuweisungskompatibel. Eine Klassentypvariable kann daher auf eine Instanz einer beliebigen übergeordneten Klasse verweisen. Ein Beispiel:

```
type
   TFigure = class(TObject);
   TRectangle = class(TFigure);
   TSquare = class(TRectangle);
var
   Fig: TFigure;
```

Der Variablen *Fig* können Instanzen von *TFigure*, *TRectangle* und *TSquare* zugewiesen werden.

Objekttypen

Als Alternative zu Klassentypen können mit der folgenden Syntax auch *Objekttypen* deklariert werden:

```
type Objekttypname = object (VorfahrObjekttyp)
   Elementliste
end;
```

Objekttypname ist ein beliebiger, gültiger Bezeichner, (*VorfahrObjekttyp*) ist optional, und *Elementliste* definiert die Felder, Methoden und Eigenschaften der Klasse. Wird kein *VorfahrObjekttyp* angegeben, hat der neue Typ keinen Vorfahren. Bei Objekttypen können Elemente nicht als **published** deklariert werden.

Da Objekttypen nicht von *TObject* abgeleitet sind, verfügen sie über keine integrierten Konstruktoren, Destruktoren oder andere Methoden. Instanzen können mit der Prozedur *New* erstellt und mit *Dispose* freigegeben werden. Sie können Variablen eines Objekttyps aber auch einfach wie bei einem Record-Typ deklarieren.

Objekttypen werden nur aus Gründen der Abwärtskompatibilität unterstützt und sollten in neuen Anwendungen nicht mehr verwendet werden.

Sichtbarkeit von Klassenelementen

In einer Klasse hat jedes Element ein Sichtbarkeitsattribut, das durch die reservierten Wörter **private**, **protected**, **public**, **published** und **automated** angegeben wird. Im folgenden Beispiel wird die Eigenschaft *Color* als **published** deklariert:

```
published property Color: TColor read GetColor write SetColor;
```

Die Sichtbarkeit bestimmt, wo und wie auf ein Element zugegriffen werden kann. **private** entspricht der geringsten, **protected** einer mittleren und **public, published** und **automated** der größten Sichtbarkeit.

Ein Element ohne Attribut erhält automatisch die Sichtbarkeit des vorhergehenden Elements in der Deklaration. Die Elemente am Anfang einer Klassendeklaration ohne explizite Sichtbarkeitsangabe werden standardmäßig als **published** deklariert, wenn die Klasse im Status **{$M+}** compiliert oder von einer mit **{$M+}** compilierten Klasse abgeleitet wurde. Andernfalls erhalten sie das Attribut **public**.

Aus Gründen der Lesbarkeit sollten Sie die Elemente einer Klassendeklaration nach ihrer Sichtbarkeit gruppieren. Nehmen Sie daher zuerst alle **private**-Elemente, anschließend alle **protected**-Elemente usw. in die Deklaration auf. Bei dieser Vorgehensweise braucht das Sichtbarkeitsattribut nur einmal angegeben zu werden, und es

markiert immer den Anfang eines neuen Deklarationsabschnitts. Eine typische Klassendeklaration sieht somit folgendermaßen aus:

```
type
  TMyClass = class(TControl)
  private
  { private-Deklarationen }
  protected
  { protected-Deklarationen }
  public
  { public-Deklarationen }
  published
  { published-Deklarationen }
  end;
```

Sie können die Sichtbarkeit eines Elements in einer untergeordneten Klasse durch Redeklarieren erhöhen, jedoch nicht verringern. So kann beispielsweise eine **protected**-Eigenschaft in einer abgeleiteten Klasse als **public** deklariert werden, nicht aber als **private**. Außerdem können **published**-Elemente nicht zu **public**-Elementen gemacht werden. Weitere Informationen zu diesem Thema finden Sie im Abschnitt »Eigenschaften überschreiben und neu deklarieren« auf Seite 7-22.

private-, protected- und public-Elemente

Auf ein **private**-Element kann nur innerhalb des Moduls (Unit oder Programm) zugegriffen werden, in dem die Klasse deklariert ist. Mit anderen Worten: eine **private**-Methode kann nicht von anderen Modulen aufgerufen werden, und als **private** deklarierte Felder oder Eigenschaften können nicht von anderen Modulen gelesen oder geschrieben werden. Indem Sie verwandte Klassendeklarationen im selben Modul zusammenfassen, können Sie diesen Klassen also den Zugriff auf alle **private**-Elemente ermöglichen, ohne die Elemente anderen Modulen bekanntzumachen.

Ein **protected**-Element ist innerhalb des Moduls mit der Klassendeklaration und in allen abgeleiteten Klassen (unabhängig davon, in welchem Modul sie deklariert sind) sichtbar. Mit anderen Worten: auf ein **protected**-Element können alle Methoden einer Klasse zugreifen, die von der Klasse mit der Elementdeklaration abgeleitet ist. Mit diesem Sichtbarkeitsattribut werden also Elemente deklariert, die nur in den Implementierungen abgeleiteter Klassen verwendet werden sollen.

Ein **public**-Element unterliegt keinerlei Zugriffsbeschränkungen. Es ist überall dort sichtbar, wo auf seine Klasse verwiesen werden kann.

published-Elemente

published-Elemente haben dieselbe Sichtbarkeit wie **public**-Elemente. Im Unterschied zu diesen werden jedoch für **published**-Elemente *Laufzeit-Typinformationen* generiert. Sie ermöglichen einer Anwendung, die Felder und Eigenschaften eines Objekts dynamisch abzufragen und seine Methoden zu lokalisieren. Delphi verwendet diese Informationen, um beim Speichern und Laden von Formulardateien (DFM) auf die Werte von Eigenschaften zuzugreifen, Eigenschaften im Objektinspektor anzuzeigen und spezielle Methoden (sogenannte *Ereignisbehandlungsroutinen*) bestimmten Eigenschaften (den *Ereignissen*) zuzuordnen.

published-Eigenschaften sind auf bestimmte Datentypen beschränkt. Ordinal-, String-, Klassen-, Schnittstellen- und Methodenzeigertypen können mit dieser Sichtbarkeit deklariert werden. Bei Mengentypen ist dies nur möglich, wenn die Ober- und Untergrenze des Basistyps einen Ordinalwert zwischen 0 und 31 hat (die Menge muß also in ein Byte, Wort oder Doppelwort passen). Auch alle Real-Typen außer Real48 können als **published** deklariert werden. Für Array-Eigenschaften kann dieser Gültigkeitsbereich nicht angegeben werden.

Obwohl alle Methoden als **published** deklariert werden können, sind in einer Klasse nicht zwei oder mehr überladene **published**-Methoden mit demselben Namen erlaubt. Felder können nur mit dieser Sichtbarkeit angegeben werden, wenn sie einen Klassen- oder Schnittstellentyp haben.

Eine Klasse kann nur **published**-Elemente haben, wenn sie mit **{$M+}** compiliert wird oder von einer Klasse abgeleitet ist, die mit **{$M+}** compiliert wurde. Die meisten Klassen mit **published**-Elementen sind von der Klasse *TPersistent* abgeleitet, die im Status **{$M+}** compiliert ist. Die Anweisung **{$M}** wird daher nur äußerst selten benötigt.

automated-Elemente

automated-Elemente haben dieselbe Sichtbarkeit wie **public**-Elemente. Im Unterschied zu diesen werden aber für **automated**-Elemente *Automatisierungs-Typinformationen* (für Automatisierungs-Server) generiert. Elemente mit dieser Sichtbarkeit werden normalerweise in Klassen verwendet, die von *TAutoObject* (in der Unit OleAuto) abgeleitet sind. Diese Unit und das reservierte Wort **automated** sind nur aus Gründen der Abwärtskompatibilität vorhanden. Für die Klasse *TAutoObject* (in der Unit *ComObj*) wird **automated** nicht verwendet.

Für **automated**-Methoden und -Eigenschaften gelten folgende Einschränkungen:

- Die Typen aller Eigenschaften, Array-Eigenschaftsparameter, Methodenparameter und Funktionsergebnisse müssen automatisierbar sein. Solche Typen sind *Byte*, *Currency*, *Real*, *Double*, *Longint*, *Integer*, *Single*, *Smallint*, *AnsiString*, *WideString*, *TDateTime*, *Variant*, *OleVariant*, *WordBool* und alle Schnittstellentypen.
- Methodendeklarationen müssen die Standardaufrufkonvention **register** verwenden. Sie können virtuell sein, nicht aber dynamisch.
- Eigenschaftsdeklarationen dürfen nur die Zugriffsangaben **read** und **write**, aber keine anderen Bezeichner (**index**, **stored**, **default** und **nodefault**) enthalten. Die Zugriffsangaben müssen einen Methodenbezeichner angeben, der die Standardaufrufkonvention **register** verwendet. Feldbezeichner sind nicht zulässig.
- Eigenschaftsdeklarationen müssen einen Typ angeben. Überladen ist hier nicht erlaubt.

Bei der Deklaration einer Methode oder Eigenschaft im **automated**-Abschnitt kann optional die Anweisung **dispid** mit einer nachfolgenden Integer-Konstante (die Automatisierungs-Dispatch-ID des Elements) angegeben werden. Fehlt die Direktive, weist der Compiler dem Element automatisch eine Dispatch-ID zu, die um eins größer ist als die höchste ID, die von den Methoden und Eigenschaften der Klasse und

ihrer Vorfahren verwendet wird. Die Angabe einer bereits vergebenen Nummer in einer **dispid**-Anweisung führt zu einem Fehler.

Weitere Informationen zur Automatisierung finden Sie im Abschnitt »Automatisierungsobjekte« auf Seite 10-11.

Vorwärtsdeklarationen und voneinander abhängige Klassen

Wenn die Deklaration eines Klassentyps wie im folgenden Beispiel mit dem Wort **class** und einem Semikolon endet und auf **class** kein Vorfahr und keine Elemente folgen, handelt es sich um eine *Vorwärtsdeklaration.*

```
type Klassenname = class;
```

Eine Vorwärtsdeklaration muß durch eine *definierende Deklaration* dieser Klasse im selben Typdeklarationsabschnitt aufgelöst werden. Zwischen einer Vorwärtsdeklaration und der zugehörigen definierenden Deklaration dürfen also mit Ausnahme anderer Typdeklarationen keine weiteren Anweisungen stehen.

Vorwärtsdeklarationen ermöglichen wie im folgenden Beispiel die Deklaration voneinander abhängiger Klassen:

```
type
   TFigure = class;  // Vorwärtsdeklaration
   TDrawing = class
    Figure: TFigure;
     ⋮
   end;
   TFigure = class  // Definierende Deklaration
    Drawing: TDrawing;
     ⋮
   end;
```

Verwechseln Sie Vorwärtsdeklarationen nicht mit vollständigen Deklarationen von Typen, die ohne Angabe von Elementen von *TObject* abgeleitet werden.

```
type
   TFirstClass = class;           // Dies ist eine Vorwärtsdeklaration.
   TSecondClass = class           // Dies ist eine vollständige Klassendeklaration.
    end;
   TThirdClass = class(TObject);  // Dies ist eine vollständige Klassendeklaration.
```

Felder

Ein Feld ist eine Variable, die zu einem bestimmten Objekt gehört. Felder können jeden Typ annehmen, auch Klassentypen (sie können also Objektreferenzen aufnehmen). Sie werden normalerweise als **private** deklariert.

Um ein Feld als Element einer Klasse zu definieren, deklarieren Sie es einfach wie eine normale Variable. In einer Klassendeklaration müssen die Felder vor den Eigenschaften und Methoden angegeben werden. Im folgenden Beispiel wird die Klasse

TNumber deklariert. Ihr einziges Element ist das Integer-Feld *Int* (außerdem erbt sie natürlich die Methoden von *TObject*)..

```
type TNumber = class
   Int: Integer;
end;
```

Felder werden statisch gebunden (die Feldreferenzen werden also beim Compilieren aufgelöst). Was dies in der Praxis bedeutet, zeigt das folgende Beispiel:

```
type
   TAncestor = class
    Value: Integer;
   end;
   TDescendant = class(TAncestor)
    Value: string;  // Verdeckt das geerbte Feld Value.
   end;
var
   MyObject: TAncestor;
begin
   MyObject := TDescendant.Create;
   MyObject.Value := 'Hallo!';  // Fehler
   TDescendant(MyObject).Value := 'Hallo!';  // Funktioniert
end;
```

MyObject enthält zwar eine Instanz von *TDescendant*, ist aber als *TAncestor* deklariert. Der Compiler interpretiert daher *MyObject.Value* als Verweis auf das in *TAncestor* deklarierte Integer-Feld. Dennoch sind im *TDescendant*-Objekt beide Felder vorhanden. Das geerbte Feld ist lediglich durch das neue verdeckt. Der Zugriff ist mit Hilfe einer Typumwandlung weiterhin möglich.

Methoden

Eine Methode ist eine Prozedur oder Funktion, die zu einer bestimmten Klasse gehört. Daher wird beim Aufruf einer Methode das Objekt (bzw. bei einer Klassenmethode die Klasse) angegeben, mit dem die Operation durchgeführt werden soll. Im folgenden Beispiel wird die Methode *Free* in *SomeObject* aufgerufen:

```
SomeObject.Free;
```

Methodenimplementierungen

Methoden werden in der Deklaration einer Klasse als Prozedur- und Funktionsköpfe angegeben (entsprechend einer Vorwärtsdeklaration). Die verschiedenen Methoden müssen dann nach der Klassendefinition im selben Modul durch eine definierende Deklaration implementiert werden. Die Deklaration der Klasse *TMyClass* im folgenden Beispiel enthält eine Methode mit dem Namen *DoSomething*:

```
type
   TMyClass = class(TObject)
     ⋮
    procedure DoSomething;
```

```
    ⋮
end;
```

Weiter unten im selben Modul wird die definierende Deklaration für *DoSomething* implementiert:

```
procedure TMyClass.DoSomething;
begin
   ⋮
end;
```

Während eine Klasse im **interface**- oder **implementation**-Abschnitt einer Unit deklariert werden kann, *müssen* die definierenden Deklarationen ihrer Methoden im **implementation**-Abschnitt stehen.

In der Kopfzeile einer definierenden Deklaration wird der Name der Methode immer mit der Klasse qualifiziert, zu der die Methode gehört. Optional kann auch die Parameterliste aus der Klassendeklaration wiederholt werden. In diesem Fall müssen aber Reihenfolge, Typ und Name der Parameter genau übereinstimmen. Bei einer Funktion muß auch der Rückgabewert identisch sein.

inherited

Das reservierte Wort **inherited** ist für die Polymorphie von großer Bedeutung. Es kann in Methodenimplementierungen (mit oder ohne nachfolgenden Bezeichner) angegeben werden.

Folgt auf **inherited** ein Methodenbezeichner, entspricht dies einem normalen Methodenaufruf. Der einzige Unterschied besteht darin, daß die Suche nach der Methode bei dem direkten Vorfahren der Klasse beginnt, zu der die Methode gehört. Durch die folgende Anweisung wird beispielsweise die geerbte Methode *Create aufgerufen:*

```
inherited Create(...);
```

Die Anweisung **inherited** ohne Bezeichner verweist auf die geerbte Methode mit demselben Namen wie die aufrufende Methode. In diesem Fall kann **inherited** mit oder ohne Parameter angegeben werden. Ohne Parameterangabe werden der geerbten Methode einfach die Parameter der aufrufenden Methode übergeben. So wird beispielsweise

```
inherited;
```

häufig in der Implementierung von Konstruktoren verwendet. Der geerbte Konstruktor wird also mit den Parametern aufgerufen, die an die abgeleitete Klasse übergeben wurden.

Self

Der Bezeichner *Self* verweist in der Implementierung einer Methode auf das Objekt, in dem die Methode aufgerufen wird. Das folgende Beispiel zeigt die Methode *Add* der Klasse *TCollection* (in der Unit *Classes*).

```
function TCollection.Add: TCollectionItem;
begin
   Result := FItemClass.Create(Self);
```

```
end;
```

Add ruft die Methode *Create* der Klasse auf, auf die das Feld *FItemClass* verweist (immer ein Nachkomme von *TCollectionItem*). Da an *TCollectionItemCreate* ein Parameter des Typs *TCollection* übergeben wird, übergibt *Add* die Instanz von *TCollection*, in der die Methode aufgerufen wird. Der folgende Code veranschaulicht dies:

```
var MyCollection: TCollection;
   ⋮
MyCollection.Add  // MyCollection wird an die Methode TCollectionItem.Create übergeben.
```

Self ist in vielen Situationen hilfreich. Wenn beispielsweise ein Element in einer Methode seiner Klasse erneut deklariert wird, kann mit ***Self.****Bezeichner* auf das Originalelement zugegriffen werden.

Informationen zu ***Self*** in Klassenmethoden finden Sie im Abschnitt »Klassenmethoden« auf Seite 7-26.

Methodenbindung

Methoden können *statisch* (Standard), *virtuell* oder *dynamisch* sein. Virtuelle und dynamische Methoden können *überschrieben* werden, und sie können *abstrakt* sein. Diese Angaben spielen eine Rolle, wenn eine Variable eines bestimmten Klassentyps eine Instanz einer abgeleiteten Klasse enthält. Sie bestimmen dann, welche Implementierung beim Aufruf der Methode aktiviert wird.

Statische Methoden

Methoden sind standardmäßig statisch. Beim Aufruf bestimmt der deklarierte Typ (also der Typ zur Compilierzeit) der im Aufruf verwendeten Klassen- bzw. Objektvariable, welche Implementierung aktiviert wird. Die *Draw*-Methoden im folgenden Beispiel sind statisch:

```
type
   TFigure = class
    procedure Draw;
   end;
   TRectangle = class(TFigure)
    procedure Draw;
   end;
```

Ausgehend von diesen Deklarationen zeigt das folgende Beispiel, wie sich Aufrufe statischer Methoden auswirken. Im zweiten Aufruf von *Figure.Draw* referenziert die Variable *Figure* ein Objekt der Klasse *TRectangle*. Es wird jedoch die *Draw*-Implementierung in *TFigure* aufgerufen, weil *Figure* als *TFigure* deklariert ist.

```
var
   Figure: TFigure;
   Rectangle: TRectangle;
begin
   Figure := TFigure.Create;
   Figure.Draw;  // Ruft TFigure.Draw auf.
   Figure.Destroy;
   Figure := TRectangle.Create;
```

```
  Figure.Draw;  // Ruft TFigure.Draw auf
  TRectangle(Figure).Draw;  // Ruft TRectangle.Draw auf.
  Figure.Destroy;
  Rectangle := TRectangle.Create;
  Rectangle.Draw;  // Ruft TRectangle.Draw auf.
  Rectangle.Destroy;
end;
```

Virtuelle und dynamische Methoden

Mit Hilfe der Direktiven **virtual** und **dynamic** können Methoden als virtuell oder dynamisch deklariert werden. Virtuelle und dynamische Methoden können im Gegensatz zu statischen Methoden in abgeleiteten Klassen *überschrieben* werden. Beim Aufrufen einer überschriebenen Methode bestimmt nicht der deklarierte, sondern der aktuelle Typ (also der Typ zur Laufzeit) der im Aufruf verwendeten Klassen- bzw. Objektvariable, welche Implementierung aktiviert wird.

Um eine Methode zu überschreiben, braucht sie nur mit der Direktiven **override** erneut deklariert zu werden. Dabei müssen Reihenfolge und Typ der Parameter sowie der Typ des Rückgabewertes (falls vorhanden) mit der Deklaration in der Vorfahrklasse übereinstimmen.

Im folgenden Beispiel wird die in der Klasse *TFigure* deklarierte Methode *Draw* in zwei abgeleiteten Klassen überschrieben:

```
type
  TFigure = class
   procedure Draw; virtual;
  end;
  TRectangle = class(TFigure)
   procedure Draw; override;
  end;
  TEllipse = class(TFigure)
   procedure Draw; override;
  end;
```

Ausgehend von diesen Deklarationen zeigt der folgende Programmcode, wie sich der Aufruf einer virtuellen Methode durch eine Variable auswirkt, deren aktueller Typ zur Laufzeit geändert wird.

```
var
  Figure: TFigure;
begin
  Figure := TRectangle.Create;
  Figure.Draw;  // Ruft TRectangle.Draw auf.
  Figure.Destroy;
  Figure := TEllipse.Create;
  Figure.Draw;  // Ruft TEllipse.Draw auf.
  Figure.Destroy;
end;
```

Nur virtuelle und dynamische Methoden können überschrieben werden. Alle Methoden können jedoch *überladen* werden (siehe »Methoden überladen« auf Seite 7-13.)

Unterschiede zwischen virtuellen und dynamischen Methoden

Virtuelle und dynamische Methoden sind von der Semantik her identisch. Sie unterscheiden sich nur bei der Implementierung der Aufrufverteilung zur Laufzeit. Virtuelle Methoden werden auf Geschwindigkeit, dynamische Methoden auf Code-Größe optimiert.

Im allgemeinen kann mit virtuellen Methoden polymorphes Verhalten am effizientesten implementiert werden. Dynamische Methoden sind hilfreich, wenn in einer Basisklasse eine große Anzahl überschreibbarer Methoden deklariert ist, die von vielen abgeleiteten Klassen geerbt, aber nur selten überschrieben werden.

Unterschiede zwischen Überschreiben und Verdecken

Wenn in einer Methodendeklaration dieselben Bezeichner- und Parameterangaben wie bei einer geerbten Methode ohne die Anweisung **override** angegeben werden, wird die geerbte Methode durch die neue Deklaration *verdeckt*. Beide Methoden sind jedoch in der abgeleiteten Klasse vorhanden, in der die Methode statisch gebunden wird. Ein Beispiel:

```
type
   T1 = class(TObject)
    procedure Act; virtual;
   end;
   T2 = class(T1)
    procedure Act;  // Act wird neu deklariert, aber nicht überschrieben.
   end;
var
   SomeObject: T1;
begin
   SomeObject := T2.Create;
   SomeObject.Act;  // Ruft T1.Act auf.
end;
```

reintroduce

Mit Hilfe der Anweisung **reintroduce** kann verhindert werden, daß der Compiler Warnungen ausgibt, wenn eine zuvor deklarierte virtuelle Methode verdeckt wird. Ein Beispiel:

```
procedure DoSomething; reintroduce;  // In der Vorfahrklasse ist auch eine
                                     // DoSomething-Methode deklariert.
```

Verwenden Sie **reintroduce**, wenn eine geerbte virtuelle Methode durch eine neue Deklaration verdeckt werden soll.

Abstrakte Methoden

Eine abstrakte Methode ist eine virtuelle oder dynamische Methode, die nicht in der Klasse implementiert wird, in der sie deklariert ist. Die Implementierung wird erst später in einer abgeleiteten Klasse durchgeführt. Bei der Deklaration abstrakter Methoden muß wie im folgenden Beispiel die Anweisung **abstract** nach **virtual** oder **dynamic** angegeben werden:

```
procedure DoSomething; virtual; abstract;
```

Eine abstrakte Methode kann nur in einer Klasse (bzw. Instanz einer Klasse) aufgerufen werden, in der sie überschrieben wurde.

Methoden überladen

Eine Methode kann auch mit der Anweisung **overload** neu deklariert werden. Wenn sich die Parameterangaben von denen ihres Vorfahren unterscheiden, wird die geerbte Methode überladen, ohne daß sie dadurch verdeckt wird. Bei einem Aufruf der Methode in einer abgeleiteten Klasse wird dann diejenige Implementierung aktiviert, bei der die Parameter übereinstimmen.

Verwenden Sie beim Überladen einer virtuellen Methode die Anweisung **reintroduce**, wenn die Methode in einer abgeleiteten Klasse neu deklariert wird. Beispiel:

```
type
   T1 = class(TObject)
    procedure Test(I: Integer); overload; virtual;
   end;
   T2 = class(T1)
    procedure Test(S: string); reintroduce; overload;
   end;
   ⋮
   SomeObject := T2.Create;
   SomeObject.Test('Hello!');  // Ruft T2.Test auf.
   SomeObject.Test(7);         // Ruft T1.Test auf.
```

Innerhalb einer Klasse dürfen nicht mehrere überladene Methoden mit demselben Namen als published deklariert werden. Die Pflege von Typinformationen zur Laufzeit bedarf für jedes als published deklariertes Element eines eindeutigen Namens.

```
type
  TSomeClass = class
  published
    function Func(P: Integer): Integer;
    function Func(P: Boolean): Integer  // error
     ⋮
```

Bei der Implementierung einer überladenen Methode muß die Parameterliste aus der Klassendeklaration wiederholt werden. Weitere Informationen zum Überladen finden Sie im Abschnitt »Prozeduren und Funktionen überladen« auf Seite 6-8.

Konstruktoren

Ein Konstruktor ist eine spezielle Methode, mit der Instanzobjekte erstellt und initialisiert werden können. Die Deklaration gleicht einer normalen Prozedurdeklaration, beginnt aber mit dem Wort **constructor**.

```
constructor Create;
constructor Create(AOwner: TComponent);
```

Für Konstruktoren muß die standardmäßige **register**-Aufrufkonvention verwendet werden. Obwohl die Deklaration keinen Rückgabewert enthält, gibt ein Konstruktor immer einen Verweis auf das erstellte Objekte zurück, wenn er mit einer Klassenreferenz aufgerufen wird.

Eine Klasse kann auch mehrere Konstruktoren haben. Im Normalfall ist jedoch nur einer vorhanden. Konstruktoren heißen normalerweise immer *Create*.

Das folgende Beispiel zeigt, wie Sie ein Objekt durch einen Aufruf des Konstruktors eines Klassentyps erstellen können:

```
MyObject := TMyClass.Create;
```

Diese Anweisung reserviert zuerst Speicher für das neue Objekt auf dem Heap. Anschließend wird allen Ordinalfeldern der Wert Null, allen Zeigern und Klassentypfeldern der Wert **nil** und allen String-Feldern ein leerer String zugewiesen. Anschließend werden die weiteren Aktionen in der Implementierung des Konstruktors ausgeführt (z.B. Initialisieren der Objekte mit den als Parameter übergebenen Werten). Am Ende gibt der Konstruktor eine Referenz auf das neu erstellte und initialisierte Objekt zurück. Der Typ entspricht dem im Aufruf angegebenen Klassentyp.

Tritt in einem mit einer Klassenreferenz aufgerufenen Konstruktor eine Exception auf, wird das unvollständige Objekt automatisch durch einen Aufruf des Destruktors *Destroy* freigegeben.

Wenn Sie einen Konstruktor anstelle einer Klassenreferenz mit einer Objektreferenz aufrufen, wird weder ein Objekt erstellt, noch ein Wert zurückgegeben. Statt dessen werden wie bei einer normalen Routine die angegebenen Anweisungen mit dem Objekt ausgeführt. Beim Aufruf mit einer Objektreferenz wird normalerweise der geerbte Konstruktor mit **inherited** ausgeführt.

Das folgende Beispiel zeigt einen Klassentyp und den zugehörigen Konstruktor:

```
type
   TShape = class(TGraphicControl)
   private
    FPen: TPen;
    FBrush: TBrush;
    procedure PenChanged(Sender: TObject);
    procedure BrushChanged(Sender: TObject);
   public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
    ⋮
   end;
constructor TShape.Create(Owner: TComponent);
```

```
begin
  inherited Create(Owner);  // Geerbten Konstruktor aufrufen.
  Width := 65;  // Geerbte Eigenschaften ändern.
  Height := 65;
  FPen := TPen.Create;  // Neue Felder initialisieren.
  FPen.OnChange := PenChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := BrushChanged;
end;
```

Als erste Anweisung wird normalerweise immer der geerbte Konstruktor aufgerufen, um die geerbten Felder zu initialisieren. Danach werden den in der abgeleiteten Klasse deklarierten Feldern Werte zugewiesen. Da der Konstruktor grundsätzlich den Speicherbereich bereinigt, der dem neuen Objekt zugewiesen wird, erhalten alle Felder automatisch den Anfangswert Null (Ordinaltypen), **nil** (Zeiger und Klassentypen), Leerstring (String-Typen) oder *Unassigned* (Varianten). Aus diesem Grund brauchen nur solche Felder explizit initialisiert zu werden, denen ein Anfangswert ungleich Null (bzw. kein Leerstring) zugewiesen werden soll.

Ein als **virtual** deklarierter Konstruktor, der mit einem Klassentypbezeichner aufgerufen wird, entspricht einem statischen Konstruktor. In Verbindung mit Klassenreferenztypen können jedoch durch virtuelle Konstruktoren Objekte polymorph erstellt werden (d.h. der Objekttyp ist beim Compilieren noch nicht bekannt). Informationen hierzu finden Sie im Abschnitt »Klassenreferenzen« auf Seite 7-24.

Destruktoren

Ein Destruktor ist eine spezielle Methode, die ein Objekt im Speicher freigibt. Die Deklaration gleicht einer normalen Prozedurdeklaration, beginnt aber mit dem Wort **destructor**.

```
destructor Destroy;
destructor Destroy; override;
```

Für Destruktoren muß die standardmäßige **register**-Aufrufkonvention verwendet werden. Obwohl in einer Klasse mehrere Destruktoren implementiert werden können, ist es ratsam, nur die geerbte *Destroy*-Methode zu überschreiben und keine weiteren Destruktoren zu deklarieren.

Ein Destruktor kann nur über ein Instanzobjekt aufgerufen werden.

```
MyObject.Destroy;
```

Beim Aufruf eines Destruktors werden zuerst die in der Implementierung angegebenen Aktionen ausgeführt. Normalerweise werden hier untergeordnete Objekte und zugewiesene Ressourcen freigegeben. Danach wird das Objekt im Speicher freigegeben.

Das folgende Beispiel zeigt eine typische Destruktorimplementierung:

```
destructor TShape.Destroy;
begin
  FBrush.Free;
  FPen.Free;
  inherited Destroy;
```

```
end;
```

Die letzte Anweisung ruft den geerbten Destruktor auf, der die geerbten Felder freigibt.

Wenn beim Erstellen eines Objekts eine Exception auftritt, wird das unvollständige Objekt automatisch durch einen Aufruf von **Destroy** freigegeben. Der Destruktor muß daher auch in der Lage sein, Objekte freizugeben, die nur teilweise erstellt wurden. Da im Konstruktor alle Felder eines neuen Objekts mit Null initialisiert werden, haben Klassenreferenz- und Zeigerfelder in einer unvollständigen Instanz immer den Wert **nil**. Testen Sie solche Felder im Destruktor immer auf den Wert **nil**, bevor Sie Operationen mit ihnen durchführen. Wenn Sie Objekte nicht mit *Destroy*, sondern mit der Methode *Free* (von *TObject)* freigeben, wird diese Prüfung automatisch durchgeführt.

Botschaftsbehandlungsroutinen

Botschaftsbehandlungsroutinen sind Methoden, in denen Reaktionen auf dynamisch gesendete Botschaften implementiert werden können. Sie werden beispielsweise in der VCL für Windows-Botschaften verwendet.

Sie erstellen eine Botschaftsbehandlungsroutine, indem Sie die Anweisung **message** und eine Integer-Konstante zwischen 1 und 49151 (die sogenannte Botschafts-ID) in eine Methodendeklaration aufnehmen. Bei Routinen für VCL-Steuerelemente muß die Konstante einer der in der Unit *Messages* definierten Windows-Botschaften entsprechen. Ein Beispiel:

```
type
  TTextBox = class(TCustomControl)
   private
   procedure WMChar(var Message: TWMChar); message WM_CHAR;
   .
   .
  end;
```

Eine Botschaftsbehandlungsroutine muß immer eine Prozedur mit einem **var**-Parameter sein.

In einer Botschaftsbehandlungsroutine braucht die Anweisung **override** nicht angegeben zu werden, um eine geerbte Routine zu überschreiben. Es muß nicht einmal derselbe Methodenname oder Parametertyp wie bei der zu überschreibenden Methode verwendet werden. Allein die Botschafts-ID bestimmt, auf welche Botschaft die Methode reagiert, und ob es sich um eine überschriebene Methode handelt.

Botschaftsbehandlungsroutinen implementieren

In der Implementierung einer Botschaftsbehandlungsroutine kann die geerbte Routine wie im folgenden Beispiel aufgerufen werden:

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
  if Chr(Message.CharCode) = #13 then
   ProcessEnter
  else
```

```
    inherited;
end;
```

Die Anweisung **inherited** durchsucht die Klassenhierarchie nach oben und ruft die erste Routine mit derselben ID wie die aktuelle Methode auf. Dabei wird automatisch der Botschafts-Record übergeben. Ist in keiner Vorfahrklasse eine Routine mit dieser ID implementiert, wird die in *TObject* definierte Originalmethode *DefaultHandler* aufgerufen.

Die Implementierung von *DefaultHandler* gibt einfach die Steuerung zurück, ohne eine Aktion auszuführen. Durch Überschreiben von *DefaultHandler* kann in einer Klasse eine eigene Standardbehandlung implementiert werden. In der Methode *DefaultHandler* von VCL-Steuerelementen wird die Windows-Funktion *DefWindowProc* aufgerufen.

Botschaftsweiterleitung

Botschaftsbehandlungsroutinen werden normalerweise nicht direkt aufgerufen. Statt dessen werden die Botschaften mit Hilfe der von *TObject* geerbten Methode *Dispatch* an ein Objekt weitergeleitet:

```
procedure Dispatch(var Message);
```

Der Parameter *Message* muß ein Record sein und als erstes Element ein *Cardinal*-Feld mit einer Botschafts-ID enthalten. Beispiele dafür finden Sie in der Unit *Messages*.

Dispatch durchsucht die Klassenhierarchie nach oben (beginnend bei der Klasse des Objekts, in dem sie aufgerufen wird) und ruft die erste für die übergebene ID gefundene Botschaftsbehandlungsroutine auf. Wird keine solche Routine gefunden, ruft *Dispatch* die Methode *DefaultHandler* auf.

Eigenschaften

Eine Eigenschaft definiert wie ein Feld ein Attribut eines Objekts. Felder sind jedoch nur Speicherbereiche, die überprüft und geändert werden können. Eigenschaften können hingegen mit Hilfe bestimmter Aktionen gelesen und geschrieben werden. Sie erlauben eine größere Kontrolle über den Zugriff auf die Attribute eines Objekts und ermöglichen das Berechnen von Attributen.

Die Deklaration einer Eigenschaft muß einen Namen, einen Typ und mindestens eine Zugriffsangabe enthalten. Die Syntax lautet folgendermaßen:

```
property Eigenschaftsname[Indizes]: Typ index Integer-Konstante Bezeichner;
```

- *Eigenschaftsname* ist ein beliebiger gültiger Bezeichner.
- [*Indizes*] ist optional und besteht aus einer Folge von durch Semikolons getrennten Parameterdeklarationen. Jede Deklaration hat die Form $Bezeichner_1$ ... $Bezeichner_n$: *Typ*. Weitere Informationen hierzu finden Sie im Abschnitt »Array-Eigenschaften« auf Seite 7-19.
- Die Klausel `index` *Integer-Konstante* ist optional (siehe »Indexangaben« auf Seite 7-21).

- *Bezeichner* ist eine Folge von **read-**, **write-**, **stored-**, **default-** (oder **nodefault-**) und **implements**-Angaben. Jede Eigenschaftsdeklaration muß zumindest einen **read-** oder **write-**Bezeichner enthalten. Informationen zu **implements** finden Sie im Abschnitt **»Schnittstellen delegieren« auf Seite 10-6**.

Eigenschaften werden durch ihre Zugriffsangaben definiert. Sie können im Gegensatz zu Feldern nicht als **var**-Parameter übergeben oder mit dem Adreßoperator **@** versehen werden. Der Grund dafür liegt darin, daß eine Eigenschaft nicht notwendigerweise im Speicher vorhanden sein muß. Sie kann beispielsweise eine **read**-Methode haben, die einen Wert aus einer Datenbank abruft oder einen Zufallswert generiert.

Auf Eigenschaften zugreifen

Jede Eigenschaft verfügt über eine **read-** oder eine **write**-Angabe (oder über beide). Diese Zugriffsbezeichner werden in folgender Form angegeben:

```
read FeldOderMethode
write FeldOderMethode
```

FeldOderMethode ist der Name eines Feldes oder einer Methode, das bzw. die in derselben Klasse oder in einer Vorfahrklasse der Eigenschaft deklariert ist.

- Wenn *FeldOderMethode* in derselben Klasse deklariert wird, muß dies vor der Eigenschaft geschehen. Ist das Element in einer Vorfahrklasse deklariert, muß es in der abgeleiteten Klasse sichtbar sein. Es kann also kein Element einer abgeleiteten Klasse angegeben werden, das in einer anderen Unit als **private** deklariert ist.
- Handelt es sich bei *FeldOderMethode* um ein Feld, muß dieses Feld denselben Typ wie die Eigenschaft haben.
- Wird in einem **read**-Bezeichner für *FeldOderMethode* eine Methode angegeben, muß eine Funktion ohne Parameter verwendet werden, die einen Wert mit dem Typ der Eigenschaft zurückgibt.
- Wird in einem **write**-Bezeichner für *FeldOderMethode* eine Methode angegeben, muß eine Prozedur verwendet werden, die einen Wert- oder **const**-Parameter mit dem Datentyp der Eigenschaft entgegennimmt.

Betrachten Sie beispielsweise die folgende Deklaration:

```
property Color: TColor read GetColor write SetColor;
```

Die Methode *GetColor* muß hier folgendermaßen deklariert werden:

```
function GetColor: TColor;
```

Die Deklaration der *Methode SetColor* muß eine der folgenden Formen haben:

```
procedure SetColor(Value: TColor);
procedure SetColor(const Value: TColor);
```

Der Parameter muß natürlich nicht unbedingt *Value* heißen.

Wenn eine Eigenschaft in einem Ausdruck verwendet wird, erfolgt der Zugriff auf ihren Wert mit Hilfe des mit **read** angegebenen Elements (Feld oder Methode). Bei

Zuweisungen wird das mit **write** angegebene Element für das Schreiben der Eigenschaft verwendet.

Im folgenden Beispiel wird die Klasse *TCompass* mit der **published**-Eigenschaft *Heading* deklariert. Zum Lesen der Eigenschaft wird das Feld *FHeading*, zum Schreiben die Prozedur *SetHeading* verwendet.

```
type
   THeading = 0..359;
   TCompass = class (TControl)
   private
    FHeading: THeading;
    procedure SetHeading(Value: THeading);
   published
    property Heading: THeading read FHeading write SetHeading;
    ⋮
   end;
```

Ausgehend von dieser Deklaration ist die Anweisung

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

mit der folgenden Anweisung identisch:

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);
```

Zum Lesen der Eigenschaft *Heading* wird keine bestimmte Aktion verwendet. Die **read**-Operation besteht lediglich aus dem Abrufen des im Feld *FHeading* gespeicherten Wertes. Wertzuweisungen an die Eigenschaft werden in Aufrufe der Methode *SetHeading* umgesetzt. Diese Methode speichert den neuen Wert im Feld *FHeading* und führt optional weitere Operationen durch. *SetHeading* könnte beispielsweise folgendermaßen implementiert werden:

```
procedure TCompass.SetHeading(Value: THeading);
begin
   if FHeading <> Value then
   begin
    FHeading := Value;
    Repaint;  // Benutzeroberfläche aktualisieren, damit der neue Wert angezeigt wird.
   end;
end;
```

Eine Eigenschaft, die nur mit einer **read**-Angabe deklariert ist, nennt man Nur-Lesen-Eigenschaft. Ist nur ein **write**-Bezeichner vorhanden, handelt es sich um eine *Nur-Schreiben-Eigenschaft*. Wenn Sie einer *Nur-Lesen-Eigenschaft* einen Wert zuweisen oder eine *Nur-Schreiben-Eigenschaft* in einem Ausdruck verwenden, tritt ein Fehler auf.

Array-Eigenschaften

Array-Eigenschaften sind indizierte Eigenschaften. Sie werden beispielsweise für die Einträge eines Listenfeldes, die untergeordneten Objekte eines Steuerelements oder die Pixel einer Bitmap-Grafik verwendet.

Die Deklaration einer Array-Eigenschaft enthält eine Parameterliste mit den Namen und Typen der Indizes. Hier einige Beispiele:

```
property Objects[Index: Integer]: TObject read GetObject write SetObject;
property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
property Values[const Name: string]: string read GetValue write SetValue;
```

Das Format der Liste ist mit dem Format der Parameterliste einer Prozedur oder Funktion identisch. Der einzige Unterschied besteht darin, daß die Parameterdeklarationen nicht in runden, sondern in eckigen Klammern angegeben werden. Im Gegensatz zu Arrays, bei denen nur ordinale Indizes erlaubt sind, können die Indizes von Array-Eigenschaften einen beliebigen Typ haben.

Bei Array-Eigenschaften müssen die Zugriffsbezeichner keine Felder, sondern Methoden angeben. Die Methode in einer **read**-Angabe muß eine Funktion sein, bei der Anzahl, Reihenfolge und Typ der Parameter mit der Indexparameterliste der Eigenschaft identisch sind und der Ergebnistyp mit dem Typ der Eigenschaft übereinstimmt. Die Methode in einer **write**-Angabe muß eine Prozedur sein, bei der Anzahl, Reihenfolge und Typ der Parameter mit der Indexparameterliste der Eigenschaft identisch sind. Außerdem muß ein zusätzlicher Wert- oder **const**-Parameter mit dem Typ der Eigenschaft vorhanden sein.

Die Zugriffsmethoden für die obigen Array-Eigenschaften können beispielsweise folgendermaßen deklariert werden:

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
function GetValue(const Name: string): string;
procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);
```

Auf eine Array-Eigenschaft kann durch Indizieren ihres Bezeichners zugegriffen werden. So sind beispielsweise die Anweisungen

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\DELPHI\BIN';
```

mit den folgenden Anweisungen identisch:

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\DELPHI\BIN');
```

Wenn Sie die Direktive **default** nach der Definition einer Array-Eigenschaft angeben, wird diese als Standardeigenschaft der betreffenden Klasse verwendet. Ein Beispiel:

```
type
   TStringArray = class
   public
    property Strings[Index: Integer]: string ...; default;
    ⋮
   end;
```

Auf die Array-Standardeigenschaft einer Klasse kann mit der Kurzform *Objekt*[*Index*] zugegriffen werden. Diese Anweisung ist mit *Objekt*.*Eigenschaft*[*Index*] identisch.

Ausgehend von der vorhergehenden Deklaration kann z.B. `StringArray.Strings[7]` zu `StringArray[7]` verkürzt werden. Jede Klasse kann nur eine Standardeigenschaft haben. Die Standardeigenschaft kann in abgeleiteten Klassen nicht gewechselt oder verdeckt werden.

Indexangaben

Mit Hilfe von Indexangaben können mehrere Eigenschaften dieselbe Zugriffsmethode verwenden, auch wenn sie unterschiedliche Werte repräsentieren. Dazu muß die Direktive **index** zusammen mit einer Integer-Konstanten zwischen –2147483647 und 2147483647 angegeben werden. Bei Eigenschaften mit Indexangaben muß auf die Direktiven **read** und **write** eine Methode (kein Feld) folgen. Ein Beispiel:

```
type
   TRectangle = class
   private
    FCoordinates: array[0..3] of Longint;
    function GetCoordinate(Index: Integer): Longint;
    procedure SetCoordinate(Index: Integer; Value: Longint);
   public
    property Left: Longint index 0 read GetCoordinate write SetCoordinate;
    property Top: Longint index 1 read GetCoordinate write SetCoordinate;
    property Right: Longint index 2 read GetCoordinate write SetCoordinate;
    property Bottom: Longint index 3 read GetCoordinate write SetCoordinate;
    property Coordinates[Index: Integer]: Longint read GetCoordinate write SetCoordinate;
    ⋮
   end;
```

Eine Zugriffsmethode für eine Eigenschaft mit einer Indexangabe benötigt einen zusätzlichen Wert-Parameter vom Typ **Integer**. Bei einer **read**-Funktion muß dies der letzte, bei einer **write**-Prozedur der vorletzte Parameter sein. Diese Konstante (der Index) wird beim Zugriff auf die Eigenschaft automatisch an die Zugriffsmethode übergeben.

Wenn *Rectangle* ein Objekt der zuvor deklarierten Klasse *TRectangle* ist, dann ist die Anweisung

```
Rectangle.Right := Rectangle.Left + 100;
```

mit der folgenden Anweisung identisch:

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

Speicherangaben

Die optionalen Direktiven **stored**, **default** und **nodefault** sind *Speicherangaben*. Sie haben keinerlei Auswirkungen auf die Funktionsweise des Programms, sondern steuern lediglich die Verwaltung der Laufzeit-Typinformationen. Genauer gesagt bestimmen sie, ob die Werte der **published**-Eigenschaften in der Formulardatei (DFM) gespeichert werden.

Nach der Angabe **stored** muß der Wert *True* oder *False*, der Name eines Booleschen Feldes oder der Name einer parameterlosen Methode folgen, die einen Booleschen Wert zurückgibt. Ein Beispiel:

```
property Name: TComponentName read FName write SetName stored False;
```

Wird eine Eigenschaft ohne die Angabe **stored** deklariert, entspricht dies der Definition **stored** True.

Nach **default** muß eine Konstante angegeben werden, die denselben Datentyp wie die Eigenschaft hat:

```
property Tag: Longint read FTag write FTag default 0;
```

Mit Hilfe des Bezeichners **nodefault** kann ein geerbter **default**-Wert ohne Angabe eines neues Wertes außer Kraft gesetzt werden. **default** und **nodefault** werden nur für Ordinal- und Mengentypen unterstützt, bei denen die Ober- und Untergrenze des Basistyps einen Ordinalwert zwischen 0 und 31 hat. Enthält eine Eigenschaftsdeklaration weder **default** noch **nodefault**, gilt sie als mit **nodefault** definiert.

Beim Speichern einer Komponente werden die Speicherbezeichner ihrer published-Eigenschaften überprüft. Wenn sich der aktuelle Wert einer Eigenschaft von ihrem default-Wert unterscheidet (oder kein default-Wert vorhanden ist) und **stored** True ist, wird der Wert gespeichert. Treffen diese Bedingungen nicht zu, wird der Wert nicht gespeichert.

Hinweis Bei Array-Eigenschaften werden Speicherangaben nicht unterstützt. **default** hat bei diesem Eigenschaftstyp eine andere Bedeutung (siehe »Array-Eigenschaften« auf Seite 7-19).

Eigenschaften überschreiben und neu deklarieren

Das Deklarieren einer Eigenschaft ohne Angabe eines Typs nennt man *Überschreiben*. Diese Vorgehensweise ermöglicht das Ändern der geerbten Sichtbarkeit bzw. des geerbten Bezeichners einer Eigenschaft. In der einfachsten Form braucht nur das reservierte Wort **property** zusammen mit einem geerbten Eigenschaftsbezeichner angegeben zu werden. Auf diese Weise kann die Sichtbarkeit der betreffenden Eigenschaft geändert werden. So kann beispielsweise eine in einer Vorfahrklasse als **protected** deklarierte Eigenschaft im **public**- oder **published**-Abschnitt einer abgeleiteten Klasse neu deklariert werden. Eigenschaftsüberschreibungen können die Angaben **read**, **write**, **stored**, **default** und **nodefault** enthalten, durch die die entsprechende geerbte Direktive außer Kraft gesetzt wird. Mit Hilfe einer Überschreibung können Sie geerbte Zugriffsangaben ersetzen, fehlende Angaben hinzufügen oder den Gültigkeitsbereich einer Eigenschaft erweitern, jedoch keine Zugriffsangaben entfernen oder die Sichtbarkeit verringern. Optional kann auch mit der Direktive **implements** die Liste der implementierten Schnittstellen ergänzt werden, ohne die geerbten Schnittstellen zu entfernen.

Die folgenden Deklarationen zeigen, wie Eigenschaften überschrieben werden können:

```
type
  TAncestor = class
```

```
    ⋮
    protected
     property Size: Integer read FSize;
     property Text: string read GetText write SetText;
     property Color: TColor read FColor write SetColor stored False;
     ⋮
    end;
type
    TDerived = class(TAncestor)
    ⋮
    protected
     property Size write SetSize;
    published
     property Text;
     property Color stored True default clBlue;
     ⋮
    end;
```

Beim Überschreiben von *Size* wird die Angabe **write** hinzugefügt, damit der Wert der Eigenschaft geändert werden kann. Die Sichtbarkeit der Eigenschaften *Text* und *Color* wird von **protected** in **published** geändert. Für die Eigenschaft *Color* wird außerdem festgelegt, daß sie in der Formulardatei gespeichert wird, wenn sie einen anderen Wert als *clBlue* hat.

Wenn Sie beim Redeklarieren einer Eigenschaft einen Typbezeichner angeben, wird die geerbte Eigenschaft nicht überschrieben, sondern lediglich verdeckt. Dadurch wird eine neue Eigenschaft mit demselben Namen wie die geerbte erstellt. Diese Art der Deklaration muß immer vollständig vorgenommen werden. Es muß immer zumindest eine Zugriffsangabe vorhanden sein.

Unabhängig davon, ob eine Eigenschaft in einer abgeleiteten Klasse verdeckt oder überschrieben wird, erfolgt die Suche nach der Eigenschaft immer *statisch*. Der deklarierte (also zur Compilierzeit bekannte) Typ der Variablen bestimmt die Interpretation des Eigenschaftsbezeichners. Aus diesem Grund wird nach dem Ausführen des folgenden Codes durch das Lesen oder Schreiben von *MyObject.Value* die Methode *Method1* bzw. *Method2* aufgerufen, obwohl *MyObject* eine Instanz von *TDescendant* enthält. Sie können aber durch eine Typumwandlung in *TDescendant* auf die Eigenschaften und Zugriffsangaben der abgeleiteten Klasse zugreifen.

```
type
    TAncestor = class
      ⋮
     property Value: Integer read Method1 write Method2;
    end;
    TDescendant = class(TAncestor)
      ⋮
     property Value: Integer read Method3 write Method4;
    end;
var MyObject: TAncestor;
    ⋮
MyObject := TDescendant.Create;
```

Klassenreferenzen

In manchen Situationen werden Operationen mit einer Klasse selbst und nicht mit ihren Instanzen (Objekten) durchgeführt. Dies geschieht beispielsweise, wenn Sie einen Konstruktor mit einer Klassenreferenz aufrufen. Sie können auf eine bestimmte Klasse immer über ihren Namen zugreifen. Machmal müssen aber Variablen oder Parameter deklariert werden, die Klassen als Werte aufnehmen. Für diese Fälle benötigen Sie *Klassenreferenztypen*.

Klassenreferenztypen

Klassenreferenztypen werden auch als *Metaklassen* bezeichnet. Die Definition erfolgt folgendermaßen:

```
class of Typ
```

Typ ist ein beliebiger Klassentyp. Der Bezeichner *Typ* gibt einen Wert des Typs **class of** Typ an. Ist Typ_1 ein Vorfahr von Typ_2, dann ist **class of** Typ_2 zuweisungskompatibel zu **class of** Typ_1. Ein Beispiel:

```
type TClass = class of TObject;
var AnyObj: TClass;
```

Die Variable *AnyObj* kann eine beliebige Klassenreferenz aufnehmen (ein Klassenreferenztyp darf nicht direkt in einer Variablendeklaration oder Parameterliste definiert werden). Klassenreferenzvariablen kann auch der Wert **nil** zugewiesen werden.

Das folgende Beispiel (Konstruktor der Klasse *TCollection* in der Unit *Classes*) zeigt, wie Klassenreferenztypen verwendet werden:

```
type TCollectionItemClass = class of TCollectionItem;
   ⋮
constructor Create(ItemClass: TCollectionItemClass);
```

Diese Deklaration besagt, daß beim Erstellen eines *TCollection*-Instanzobjekts der Name einer von *TCollectionItem* abgeleiteten Klasse an den Konstruktor übergeben werden muß.

Klassenreferenztypen sind hilfreich, wenn Sie eine Klassenmethode oder einen virtuellen Konstruktor in einer Klasse oder einem Objekt aufrufen wollen, dessen aktueller Typ zur Compilierzeit nicht bekannt ist.

Konstruktoren und Klassenreferenzen

Ein Konstruktor kann mit einer Variablen eines Klassenreferenztyps aufgerufen werden. Auf diese Weise können Objekte erstellt werden, deren Typ zur Compilierzeit nicht bekannt ist. Ein Beispiel:

```
type TControlClass = class of TControl;
function CreateControl(ControlClass: TControlClass;
    const ControlName: string; X, Y, W, H: Integer): TControl;
begin
   Result := ControlClass.Create(MainForm);
```

```
    with Result do
    begin
     Parent := MainForm;
     Name := ControlName;
     SetBounds(X, Y, W, H);
     Visible := True;
    end;
end;
```

Der Funktion *CreateControl* wird eine Klassenreferenz als Parameter übergeben. Er bestimmt, welche Art von Steuerelement erstellt wird. Der Parameter wird anschließend beim Aufruf des Konstruktors verwendet. Da Klassentypbezeichner Klassenreferenzwerte enthalten, kann im Aufruf von *CreateControl* der Bezeichner der Klasse angegeben werden, um eine Instanz vor ihr zu erstellen:

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
```

Konstruktoren, die mit Klassenreferenzen aufgerufen werden, sind normalerweise virtuell. Die entsprechende Implementierung wird anhand des beim Aufruf angegebenen Laufzeittyps aktiviert.

Klassenoperatoren

Jede Klasse erbt von *TObject* die Methoden *ClassType* und *ClassParent*, mit denen die Klasse eines Objekts und seines direkten Vorfahren ermittelt werden kann. Beide Methoden geben einen Wert des Typs *TClass* (*TClass* = ***class of*** *TObject*) zurück, der in einen spezielleren Typ umgewandelt werden kann. Alle Klassen erben außerdem die Methode *InheritsFrom*, mit der Sie ermitteln können, ob ein Objekt von einer bestimmten Klasse abgeleitet ist. Diese Methoden werden von den Operatoren **is** und **as** verwendet und normalerweise nicht direkt aufgerufen.

Der Operator is

Der Operator **is** führt eine dynamische Typprüfung durch. Mit ihm können Sie den aktuellen Laufzeittyp eines Objekts ermitteln. Der Ausdruck

Objekt `is` *Klasse*

gibt *True* zurück, wenn *Objekt* eine Instanz der angegebenen *Klasse* oder eines ihrer Nachkommen ist. Trifft dies nicht zu, wird *False* zurückgegeben (hat *Objekt* den Wert **nil**, ist der Rückgabewert ebenfalls *False*). Wenn der deklarierte Typ von *Objekt* nicht mit *Klasse* verwandt ist (wenn die Typen also unterschiedlich und nicht voneinander abgeleitet sind), gibt der Compiler eine Fehlermeldung aus. Ein Beispiel:

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

Diese Anweisung prüft zuerst, ob die Variable eine Instanz von *TEdit* oder einem ihrer Nachkommen ist, und führt anschließend eine Typumwandlung in *TEdit* durch.

Der Operator as

Der Operator **as** führt eine Typumwandlung mit Laufzeitprüfung durch. Der Ausdruck

Objekt as *Klasse*

gibt eine Referenz auf dasselbe Objekt wie *Objekt*, aber mit dem von *Klasse* angegebenen Typ zurück. Zur Laufzeit muß *Objekt* eine Instanz von Klasse oder einem ihrer Nachkommen bzw. **nil** sein. Andernfalls wird eine Exception ausgelöst. Wenn der deklarierte Typ von Objekt nicht mit Klasse verwandt ist (wenn die Typen also unterschiedlich und nicht voneinander abgeleitet sind), gibt der Compiler eine Fehlermeldung aus. Ein Beispiel:

```
with Sender as TButton do
begin
  Caption := '&Ok';
  OnClick := OkClick;
end;
```

Die Regeln der Auswertungsreihenfolge machen es häufig erforderlich, **as**-Typumwandlungen in Klammern zu setzen:

```
(Sender as TButton).Caption := '&Ok';
```

Klassenmethoden

Eine Klassenmethode ist eine Methode, die nicht mit Objekten, sondern mit Klassen arbeitet. Die Definition muß wie im folgenden Beispiel mit dem reservierten Wort **class** beginnen:

```
type
   TFigure = class
   public
    class function Supports(Operation: string): Boolean; virtual;
    class procedure GetInfo(var Info: TFigureInfo); virtual;
    ⋮
   end;
```

Auch die definierende Deklaration einer Klassenmethode muß mit **class** eingeleitet werden:

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
   ⋮
end;
```

In der definierenden Deklaration einer Klassenmethode kann mit dem Bezeichner *Self* auf die Klasse zugegriffen werden, in der die Methode aufgerufen wird (dies kann auch ein Nachkomme der Klasse sein, in der sie definiert ist). Wird die Methode beispielsweise in der Klasse *K* aufgerufen, hat *Self* den Typ **class of** *K*. Daher können Sie ***Self*** nicht für den Zugriff auf Felder, Eigenschaften und normale (Objekt-) Methoden, sondern nur für Aufrufe von Konstruktoren und anderen Klassenmethoden verwenden.

Eine Klassenmethode kann über eine Klassenreferenz oder eine Objektreferenz aufgerufen werden. Bei einer Objektreferenz erhält *Self* als Wert die Klasse des betreffenden Objekts.

Exceptions

Eine *Exception* wird ausgelöst, wenn die normale Programmausführung durch einen Fehler oder ein anderes Ereignis unterbrochen wird. Die Steuerung wird dadurch an eine *Exception-Behandlungsroutine* übergeben. Mit Hilfe dieser Routinen kann die normale Programmlogik von der Fehlerbehandlung getrennt werden. Da Exceptions Objekte sind, können sie durch Vererbung in einer Hierarchie organisiert werden. Sie bringen bestimmte Informationen (z.B. eine Fehlermeldung) von der Stelle im Programm, an der sie ausgelöst wurden, zu dem Punkt, an dem sie behandelt werden.

Wenn die Unit *SysUtils* in einer Anwendung verwendet wird, werden alle Laufzeitfehler automatisch in Exceptions konvertiert. Fehler, die andernfalls zum Beenden der Anwendung führen (z.B. Speichermangel, Division durch Null oder allgemeine Schutzverletzungen), können so abgefangen und behandelt werden.

Exception-Typen deklarieren

Exception-Typen werden genau wie andere Klassen deklariert. Eigentlich können Sie eine Instanz einer beliebigen Klasse als Exception verwenden. Es ist aber zu empfehlen, Exceptions immer von der Klasse *Exception* (Unit *SysUtils*) abzuleiten.

Mit Hilfe der Vererbung können Exceptions in Familien organisiert werden. Die folgenden Deklarationen in *SysUtils* definieren beispielsweise eine Familie von Exception-Typen für mathematische Fehler:

```
type
  EMathError = class(Exception);
  EInvalidOp = class(EMathError);
  EZeroDivide = class(EMathError);
  EOverflow = class(EMathError);
  EUnderflow = class(EMathError);
```

Aufgrund dieser Deklarationen können Sie eine Behandlungsmethode für *EMathError* bereitstellen und in dieser auch *EInvalidOp*, *EZeroDivide*, *EOverflow* und *EUnderflow* behandeln.

In Exception-Klassen sind manchmal auch Felder, Methoden oder Eigenschaften definiert, die zusätzliche Informationen über den Fehler liefern. Ein Beispiel:

```
type EInOutError = class(Exception)
    ErrorCode: Integer;
   end;
```

Exceptions auslösen und behandeln

Um ein Exception-Objekt zu erstellen, rufen Sie den Konstruktor der Exception-Klasse in einer **raise**-Anweisung auf:

```
raise EMathError.Create;
```

Im allgemeinen hat diese Anweisung folgende Form:

```
raise Objekt at Adresse
```

Objekt und at *Adresse* sind optional. Ohne Angabe von *Objekt* wird die aktuelle Exception erneut ausgelöst (siehe»Exceptions erneut auslösen« auf Seite 7-31). *Adresse* ist normalerweise ein Zeiger auf eine Prozedur oder Funktion. Mit Hilfe dieser Option kann die Exception an einem früheren Punkt im Stack ausgelöst werden.

Wenn eine Exception *ausgelöst* (d.h. in einer **raise**-Anweisung angegeben) wird, unterliegt sie einer speziellen Behandlungslogik. Die Programmsteuerung wird durch eine **raise**-Anweisung nicht auf normale Weise zurückgegeben. Sie wird statt dessen an die innerste Behandlungsroutine übergeben, die Exceptions der jeweiligen Klasse verarbeiten kann (die innerste Behandlungsroutine ist diejenige, deren **try...except**-Block zuletzt ausgeführt, aber noch nicht beendet wurde).

In der folgenden Funktion wird ein String in einen Integer-Wert konvertiert. Wenn dieser Wert nicht innerhalb eines bestimmten Bereichs liegt, wird eine *ERangeError*-Exception ausgelöst.

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
   Result := StrToInt(S);  // StrToInt ist in SysUtils deklariert.
   if (Result < Min) or (Result > Max) then
    raise ERangeError.CreateFmt(
      '%d liegt nicht im gültigen Bereich zwischen %d..%d',
      [Result, Min, Max]);
end;
```

Beachten Sie die Methode *CreateFmt*, die in der **raise**-Anweisung aufgerufen wird. Die Klasse *Exception* und ihre Nachkommen verfügen über spezielle Konstruktoren, um Fehlermeldungen und Kontext-IDs zu erstellen. Weitere Informationen hierzu finden Sie in der Online-Hilfe.

Eine ausgelöste Exception wird nach ihrer Behandlung automatisch wieder freigegeben. Versuchen Sie daher niemals, Exceptions manuell freizugeben.

Hinweis Das Auslösen einer Exception im **initialization**-Abschnitt einer Unit führt nicht zum gewünschten Ergebnis. Die normale Exception-Unterstützung wird durch die Unit *SysUtils* eingebunden, die daher zuerst initialisiert werden muß. Wenn während der Initialisierung eine Exception ausgelöst wird, werden alle initialisierten Units (einschließlich *SysUtils*) finalisiert, und die Exception wird erneut ausgelöst. Anschließend wird sie von der Unit *System* abgefangen und behandelt. Dabei wird das Programm normalerweise unterbrochen.

Die Anweisung try...except

Exceptions werden mit Hilfe von **try...except**-Anweisungen behandelt.

```
try
   X := Y/Z;
except
   on EZeroDivide do HandleZeroDivide;
end;
```

Zuerst wird im **try**-Block die Division *Y / Z* durchgeführt. Tritt dabei eine *EZero-Divide*-Exception (Division durch Null) auf, wird die Behandlungsroutine *HandleZeroDivide* aufgerufen.

Die Syntax einer **try...except**-Anweisung lautet folgendermaßen:

```
try Anweisungsliste except ExceptionBlock end
```

Anweisungsliste ist eine Folge beliebiger Anweisungen. *ExceptionBlock* ist entweder

- eine weitere Anweisungsfolge oder
- eine Folge von Exception-Behandlungsroutinen, optional mit nachfolgendem

  `else` *Anweisungsliste*

Eine Exception-Behandlungsroutine hat folgende Form:

`on` *Bezeichner*`:` *Typ* `do` *Anweisung*

Bezeichner: ist optional und kann ein beliebiger Bezeichner sein. *Typ* ist ein für die Exception verwendeter Typ, und *Anweisung* ist eine beliebige Anweisung.

In einer **try...except**-Anweisung werden zuerst die Programmzeilen in *Anweisungsliste* ausgeführt. Werden dabei keine Exceptions ausgelöst, wird *ExceptionBlock* ignoriert und die Steuerung an den nächsten Programmteil übergeben.

Tritt bei der Ausführung von *Anweisungsliste* eine Exception auf (entweder durch eine **raise**-Anweisung oder eine aufgerufene Prozedur bzw. Funktion), versucht das Programm, diese zu behandeln:

- Stimmt eine der Behandlungsroutinen im Exception-Block mit der betreffenden Exception überein, wird die Steuerung an diese Routine übergeben. Eine Übereinstimmung liegt vor, wenn der *Typ* in der Behandlungsroutine der Klasse der Exception oder eines ihrer Nachkommen entspricht.
- Wenn keine Behandlungsroutine existiert, wird die Steuerung an den Block *Anweisungsliste* in der ***else***-Klausel übergeben (falls vorhanden).
- Besteht der Exception-Block lediglich aus einer Folge von Anweisungen (ohne Exception-Behandlungsroutinen), wird die Steuerung an die erste Anweisung in der Liste übergeben.

Trifft keine dieser Bedingungen zu, wird die Suche im Exception-Block der zuletzt ausgeführten und noch nicht beendeten **try...except**-Anweisung fortgesetzt. Kann dort keine entsprechende Behandlungsroutine, **else**-Klausel oder Anweisungsliste gefunden werden, wird die nächste **try...except**-Anweisung durchsucht usw. Ist die Exception bei Erreichen der äußersten **try...except**-Anweisung immer noch nicht behandelt worden, wird das Programm beendet.

Beim Behandeln einer Exception wird der Aufruf-Stack nach oben bis zu der Prozedur oder Funktion durchlaufen, in der sich die **try...except**-Anweisung befindet, in der die Behandlung durchgeführt wird. Die Steuerung wird dann an die entsprechende Exception-Behandlungsroutine, **else**-Klausel oder Anweisungsliste übergeben. Bei diesem Vorgang werden alle Prozedur- und Funktionsaufrufe verworfen, die nach dem Eintritt in den **try...except**-Block stattgefunden haben. Anschließend wird das Exception-Objekt durch einen Aufruf seines Destruktors *Destroy* automatisch freigegeben, und die Programmausführung wird mit der nächsten Anweisung nach

dem **try...except**-Block fortgesetzt (das Objekt wird auch automatisch freigegeben, wenn die Behandlungsroutine durch einen Aufruf der Standardprozedur *Exit*, *Break* oder *Continue* verlassen wird).

Im folgenden Beispiel sind drei Behandlungsroutinen definiert. Die erste behandelt Divisionen durch Null, die zweite Überläufe, und die dritte alle anderen mathematischen Exceptions. Der Typ *EMathError* ist zuletzt aufgeführt, da er der Vorfahr der anderen beiden Exception-Klassen ist. Würde er an erster Stelle genannt, käme es nie zu einem Aufruf der beiden anderen Routinen.

```
try
   ⋮
except
   on EZeroDivide do HandleZeroDivide;
   on EOverflow do HandleOverflow;
   on EMathError do HandleMathError;
end;
```

Vor dem Namen der Exception-Klasse kann optional ein Bezeichner angegeben werden. Dieser Bezeichner dient in der auf **on...do** folgenden Anweisung zum Zugriff auf das Exception-Objekt. Der Gültigkeitsbereich des Bezeichners ist auf die Anweisung beschränkt. Ein Beispiel:

```
try
   ⋮
except
   on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

Im Exception-Block kann auch eine **else**-Klausel angegeben werden. Dort werden alle Exceptions behandelt, die nicht von den **on...do**-Behandlungsroutinen abgedeckt werden. Ein Beispiel:

```
try
   ⋮
except
   on EZeroDivide do HandleZeroDivide;
   on EOverflow do HandleOverflow;
   on EMathError do HandleMathError;
else
   HandleAllOthers;
end;
```

In diesem Fall werden in der **else**-Klausel alle Exceptions außer *EMathError* behandelt.

Ein Exception-Block, der nur eine Liste von Anweisungen, jedoch keine Behandlungsroutinen enthält, behandelt alle Exceptions. Ein Beispiel:

```
try
   ⋮
except
   HandleException;
end;
```

Hier behandelt die Routine *HandleException* alle Exceptions, die bei der Ausführung der Anweisungen zwischen **try** und **except** ausgelöst werden.

Exceptions erneut auslösen

Wenn Sie das reservierte Wort **raise** ohne nachfolgende Objektreferenz in einem Exception-Block angeben, wird die aktuell behandelte Exception nochmals ausgelöst. Auf diese Weise kann in einer Behandlungsroutine begrenzt auf einen Fehler reagiert und anschließend die Exception erneut ausgelöst werden. Diese Möglichkeit ist hilfreich, wenn in einer Prozedur oder Funktion nach Auftreten einer Exception Aufräumarbeiten durchgeführt werden sollen (z.B. Objekte oder Ressourcen freigeben).

Im folgenden Beispiel wird ein *TStringList*-Objekt erstellt und mit den Namen der Dateien im übergebenen Pfad gefüllt:

```
function GetFileList(const Path: string): TStringList;
var
   I: Integer;
   SearchRec: TSearchRec;
begin
   Result := TStringList.Create;
   try
    I := FindFirst(Path, 0, SearchRec);
    while I = 0 do
    begin
      Result.Add(SearchRec.Name);
      I := FindNext(SearchRec);
    end;
   except
    Result.Free;
    raise;
   end;
end;
```

In dieser Funktion wird ein *TStringList*-Objekt erstellt und mit Hilfe der Funktionen *FindFirst* und *FindNext* (Unit *SysUtils*) mit Werten gefüllt. Tritt dabei ein Fehler auf (z.B. aufgrund eines ungültigen Pfades oder wegen Speichermangel), muß das neue Objekt freigegeben werden, da es der aufrufenden Routine noch nicht bekannt ist. Aus diesem Grund muß die Initialisierung der String-Liste in einer **try...except**-Anweisung durchgeführt werden. Bei einer Exception wird das Objekt im Exception-Block freigegeben und anschließend die Exception erneut ausgelöst.

Verschachtelte Exceptions

In einer Exception-Behandlungsroutine können wiederum Exceptions ausgelöst und behandelt werden. Solange dieser Vorgang ebenfalls innerhalb der Routine stattfindet, hat er keinen Einfluß auf die ursprüngliche Exception. Wenn die zweite Exception jedoch die Routine verläßt, geht die Original-Exception verloren. Beispiel:

```
type
   ETrigError = class(EMathError);
function Tan(X: Extended): Extended;
begin
   try
    Result := Sin(X) / Cos(X);
   except
    on EMathError do
```

```
      raise ETrigError.Create('Ungültiges Argument für Tan');
    end;
end;
```

Wenn während der Ausführung von Tan eine *EMathError*-Exception auftritt, wird in der Behandlungsroutine eine *ETrigError*-Exception ausgelöst. Da in *Tan* keine Routine für *ETrigError* definiert ist, verläßt die Exception die Behandlungsroutine, und die ursprüngliche *EMathError*-Exception wird freigegeben. Für die aufrufende Routine stellt sich der Vorgang so dar, als ob die Funktion *Tan* eine *ETrigError*-Exception ausgelöst hat.

Die Anweisung try...finally

In manchen Situationen muß sichergestellt sein, daß bestimmte Operationen auch bei Auftreten einer Exception vollständig abgeschlossen werden. Wenn beispielsweise in einer Routine eine Ressource zugewiesen wird, ist es sehr wichtig, daß sie unabhängig von der Beendigung der Routine wieder freigegeben wird. In diesen Fällen können **try...finally**-Anweisungen verwendet werden.

Das folgende Beispiel zeigt, wie eine Datei auch dann wieder geschlossen werden kann, wenn beim Öffnen oder Bearbeiten eine Exception auftritt:

```
Reset(F);
try
  ⋮ // Datei F öffnen und bearbeiten.
finally
  CloseFile(F);
end;
```

Eine **try...finally**-Anweisung hat folgende Syntax:

try *Anweisungsliste*$_1$ finally *Anweisungsliste*$_2$ end

Jede Anweisungsliste setzt sich aus einer Folge von Anweisungen zusammen. In einem **try...finally**-Block werden zuerst die Programmzeilen in *Anweisungsliste*$_1$ (**try**-Klausel) ausgeführt. Wenn dabei keine Exceptions auftreten, wird anschließend *Anweisungsliste*$_2$ (**finally**-Klausel) ausgeführt. Bei einer Exception wird die Steuerung an *Anweisungsliste*$_2$ übergeben und danach die Exception erneut ausgelöst. Befindet sich ein Aufruf der Standardprozedur *Exit*, *Break* oder *Continue* in *Anweisungsliste*$_1$, wird dadurch automatisch *Anweisungsliste*$_2$ aufgerufen. Daher wird die **finally**-Klausel unabhängig davon, wie der **try**-Block beendet wird, immer ausgeführt.

Wenn eine Exception ausgelöst, aber in der **finally**-Klausel nicht behandelt wird, führt sie aus der **try...finally**-Anweisung hinaus, und jede zuvor in der **try**-Klausel ausgelöste Exception geht verloren. In der **finally**-Klausel sollten daher alle lokal ausgelösten Exceptions behandelt werden, damit die Behandlung anderer Exceptions nicht gestört wird.

Exception-Standardklassen und -Standardroutinen

In der Unit *SysUtils* sind verschiedene Standardroutinen für die Exception-Behandlung deklariert (z.B. *ExceptObject*, *ExceptAddr* und *ShowException*). Diese und andere

VCL-Units enthalten auch zahlreiche Exception-Klassen (außer *OutlineError*), die von *Exception* abgeleitet sind.

Die Klasse *Exception* verfügt über die Eigenschaften *Message* und *HelpContext*, durch die eine Fehlerbeschreibung und eine Kontext-ID für die kontextbezogene Online-Dokumentation übergeben werden kann. Außerdem definiert sie verschiedene Konstruktor-Methoden, mit denen Fehlerbeschreibungen und Kontext-IDs auf unterschiedliche Arten angegeben werden können. Detaillierte Informationen hierzu finden Sie in der Online-Hilfe.

Kapitel

8

Standardroutinen und E/A

Dieses Kapitel behandelt die Text- und Datei-E/A und gibt einen Überblick über die Standardbibliotheksroutinen. Viele der hier aufgeführten Prozeduren und Funktionen sind in der Unit *System* definiert, die implizit mit jeder Anwendung compiliert wird. Andere Routinen sind im Compiler integriert, werden jedoch so behandelt, als wären sie in der Unit *System* enthalten.

Einige Standardroutinen befinden sich in Units wie *SysUtils*, die in einer **uses**-Klausel aufgeführt werden müssen, wenn sie in ein Programm eingebunden werden sollen. *System* darf jedoch nicht in einer **uses**-Klausel angegeben werden. Außerdem sollten Sie die Unit *System* nicht bearbeiten oder explizit neu compilieren.

Weitere Informationen zu den hier aufgeführten Routinen finden Sie in der Online-Hilfe.

Dateiein und -ausgabe

Die folgende Tabelle enthält die Ein- und Ausgaberoutinen.

Tabelle 8.1 Ein- und Ausgaberoutinen

Routine	Beschreibung
Append	Öffnet eine vorhandene Textdatei zum Anhängen von Daten.
AssignFile	Weist einer Dateivariablen den Namen einer externen Datei zu.
BlockRead	Liest einen oder mehrere Blöcke aus einer untypisierten Datei.
BlockWrite	Schreibt einen oder mehrere Blöcke in eine untypisierte Datei.
ChDir	Wechselt das aktuelle Verzeichnis.
CloseFile	Schließt eine geöffnete Datei.
Eof	Gibt den EOF-Status einer Datei zurück.
Eoln	Gibt den Zeilenende-Status einer Textdatei zurück.
Erase	Löscht eine externe Datei.

Tabelle 8.1 Ein- und Ausgaberoutinen (Fortsetzung)

Routine	Beschreibung
FilePos	Gibt die aktuelle Position in einer typisierten oder untypisierten Datei zurück.
FileSize	Gibt die aktuelle Größe einer Datei zurück (nicht für Textdateien).
Flush	Leert den Puffer einer Ausgabe-Textdatei.
GetDir	Gibt das aktuelle Verzeichnis eines bestimmten Laufwerks zurück.
IOResult	Gibt einen Integer-Wert zurück, der den Status der zuletzt durchgeführten E/A-Operation angibt.
MkDir	Legt ein Unterverzeichnis an.
Read	Liest einen oder mehrere Werte aus einer Datei in eine oder mehrere Variablen.
Readln	Wie *Read*, positioniert aber anschließend den Dateizeiger auf den Anfang der nächsten Zeile der Textdatei.
Rename	Benennt eine externe Datei um.
Reset	Öffnet eine vorhandene Datei.
Rewrite	Erzeugt und öffnet eine neue Datei.
RmDir	Entfernt ein leeres Unterverzeichnis.
Seek	Setzt den Dateizeiger in einer typisierten oder untypisierten Datei auf die angegebene Position (nicht bei Textdateien).
SeekEof	Gibt den EOF-Status einer Textdatei zurück.
SeekEoln	Gibt den Zeilenende-Status einer Textdatei zurück.
SetTextBuf	Weist einer Textdatei einen E/A-Puffer zu.
Truncate	Schneidet eine typisierte oder untypisierte Datei an der aktuellen Position ab.
Write	Schreibt einen oder mehrere Werte in eine Datei.
Writeln	*Wie Write*, schreibt aber anschließend ein Zeilenendezeichen in die Textdatei.

Jede Variable vom Typ File ist eine Dateivariable. Es gibt drei Klassen von Dateien: *typisierte Dateien*, *untypisierte Dateien* und *Textdateien*. Die Syntax für Dateitypen finden Sie unter »Dateitypen« auf Seite 5-26.

Bevor eine Dateivariable verwendet werden kann, muß sie durch einen Aufruf der Prozedur *AssignFile* einer externen Datei zugeordnet werden. Eine externe Datei ist entweder eine Datei auf einem Laufwerk oder ein Gerät (beispielsweise die Tastatur oder der Bildschirm). Die externe Datei speichert Informationen oder stellt sie zur Verfügung.

Nachdem die Zuordnung zu einer externen Datei hergestellt wurde, muß die Dateivariable geöffnet werden, um Ein- oder Ausgaben zu ermöglichen. Eine vorhandene Datei kann mit der Prozedur *Reset* geöffnet werden. Mit der Prozedur *Rewrite* wird eine neue Datei erzeugt und geöffnet. Textdateien, die mit der Prozedur *Reset* geöffnet wurden, erlauben nur Lesezugriffe. Textdateien, die mit *Rewrite* oder *Append* geöffnet wurden, erlauben nur Schreibzugriffe. Typisierte und untypisierte Dateien erlauben Lese- und Schreibzugriffe, unabhängig davon, ob sie mit *Reset* oder *Rewrite* geöffnet wurden.

Jede Datei enthält eine lineare Folge von Komponenten, die alle dem Komponententyp (oder dem Record-Typ) der Datei entsprechen. Jeder Komponente ist eine Nummer zugeordnet. Die erste Komponente einer Datei hat die Nummer 0.

Normalerweise erfolgt der Dateizugriff sequentiell, d.h. beim Lesen einer Komponente mit Hilfe der Standardprozedur *Read* oder beim Schreiben mit Hilfe der Standardprozedur *Write* wird der Dateizeiger auf die nächste Komponente positioniert. Auf typisierte und untypisierte Dateien können Sie jedoch auch wahlfrei zugreifen, indem Sie die Standardprozedur *Seek* verwenden, die den Dateizeiger von der aktuellen zur angegebenen Position verschiebt. Mit den Standardfunktionen *FilePos* und *FileSize* können Sie die aktuelle Position des Dateizeigers und die aktuelle Größe der Datei ermitteln.

Nach der Bearbeitung durch das Programm muß die Datei mit der Standardprozedur *CloseFile* geschlossen werden. Nach dem Schließen wird die entsprechende externe Datei aktualisiert. Die Dateivariable kann dann einer anderen externen Datei zugeordnet werden.

In der Voreinstellung werden sämtliche Aufrufe von Standard-E/A-Routinen auf Fehler überprüft. Beim Auftreten eines Fehlers wird eine Exception ausgelöst (wenn keine Exception-Behandlung aktiviert ist, wird das Programm abgebrochen). Die automatische Überprüfung kann mit Hilfe der Compiler-Direktiven **{$I+}** und **{$I–}** ein- und ausgeschaltet werden. Wenn die E/A-Prüfung ausgeschaltet ist (d.h. wenn eine Routine im Status **{$I–}** compiliert wird), löst ein E/A-Fehler keine Exception aus. In diesem Fall muß zur Überprüfung des Ergebnisses einer E/A-Operation die Standardfunktion *IOResult* aufgerufen werden.

IOResult sollte auch dann aufgerufen werden, wenn der aufgetretene Fehler für das Programm nicht weiter von Bedeutung ist. Wenn Sie den Aufruf unterlassen, schlägt der nächste Aufruf einer E/A-Funktion im Status **{$I+}** fehl**.**

Textdateien

Dieser Abschnitt behandelt E/A-Operationen mit Dateivariablen vom Standardtyp *Text*.

Beim Öffnen einer Datei vom Typ Text wird die externe Datei als Folge von Zeichen interpretiert, die in Form von Zeilen vorliegen. Am Ende jeder Zeile steht ein Zeilenendezeichen (das Zeichen für Wagenrücklauf, eventuell gefolgt vom Zeichen für einen Zeilenvorschub). Der Typ Text unterscheidet sich vom Typ `File of Char`.

Für Textdateien gibt es besondere Formen der Prozeduren *Read* und *Write*, mit denen Sie Werte lesen und schreiben können, die nicht vom Typ Char sind. Die Werte werden automatisch in ihre Zeichendarstellung übersetzt. Die Prozedur `Read(F, I)` liest beispielsweise eine Folge von Ziffern, interpretiert sie als dezimale Integer und speichert sie in *I* (wobei *I* vom Typ Integer ist).

Als Standard-Textdateivariablen sind *Input* und *Output* definiert. *Input* ermöglicht nur Lesezugriffe und ist der Standard-Eingabedatei des Betriebssystems (normalerweise die Tastatur) zugeordnet. *Output* ermöglicht nur Schreibzugriffe und ist der Standard-Ausgabedatei des Betriebssystems (normalerweise der Bildschirm) zuge-

ordnet. *Input* und *Output* werden vor der Ausführung eines Programms automatisch geöffnet. Dies entspricht der Ausführung folgender Anweisungen:

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

Hinweis Textorientierte E/A-Operationen sind nur in Konsolenanwendungen verfügbar, also in Anwendungen, die entweder mit der Option *Textbildschirm-Anwendung* (Registerkarte *Linker* im Dialogfeld *Projektoptionen*) oder mit der Befehlszeilenoption **-cc** compiliert wurden. In einer GUI-Anwendung verursacht jede Lese- oder Schreiboperation mit *Input* oder *Output* einen E/A-Fehler.

Nicht allen Standard-E/A-Routinen, die mit Textdateien arbeiten, muß explizit eine Dateivariable als Parameter übergeben werden. Wenn der Parameter fehlt, wird standardmäßig *Input* oder *Output* verwendet, je nachdem, ob die Prozedur oder Funktion eingabe- oder ausgabeorientiert ist. `Read(X)` entspricht beispielsweise `Read(Input, X)`, und `Write(X)` entspricht `Write(Output, X)`.

Eine Datei, die einer Ein- oder Ausgaberoutine für Textdateien übergeben wird, muß zuvor mit *AssignFile* einer externen Datei zugeordnet und mit *Reset*, *Rewrite* oder *Append* geöffnet werden. Wenn Sie eine mit *Reset* geöffnete Datei an eine ausgabeorientierte Prozedur oder Funktion übergeben, wird eine Exception ausgelöst. Gleiches gilt, wenn Sie eine mit *Rewrite* oder *Append* geöffnete Datei an eine eingabeorientierte Routine übergeben.

Untypisierte Dateien

Untypisierte Dateien kann man sich als Low-Level-E/A-Kanäle vorstellen, die vorrangig für den direkten Zugriff auf Dateien verwendet werden, und zwar unabhängig von deren Typ und Struktur. Eine untypisierte Datei wird nur mit dem Wort **file** deklariert:

```
var DataFile: file;
```

Bei untypisierten Dateien kann den Prozeduren *Reset* und *Rewrite* ein Parameter übergeben werden, der die Größe der Blöcke bei Lese- und Schreiboperationen festlegt. Die Standardgröße beträgt aus historischen Gründen 128 Bytes. Nur der Wert 1 führt zuverlässig bei jeder beliebigen Datei zur korrekten Größe (ein Block der Größe 1 kann nicht weiter unterteilt werden).

Mit Ausnahme von *Read* und *Write* können alle Standardroutinen für typisierte Dateien auch für untypisierte Dateien verwendet werden. Anstelle von *Read* und *Write* stehen mit *BlockRead* und *BlockWrite* zwei Prozeduren für besonders schnelle Lese- und Schreiboperationen zur Verfügung.

Gerätetreiber für Textdateien

Sie können für Windows-Programme Ihre eigenen Gerätetreiber für Textdateien definieren. Ein Gerätetreiber für Textdateien besteht aus vier Funktionen, die eine Schnittstelle zwischen einem Gerät und dem Dateisystem von Object Pascal implementieren.

Jeder Gerätetreiber wird durch die Funktionen Open, InOut, Flush und *Close* definiert. Der Kopf einer jeden Funktion wird folgendermaßen deklariert:

```
function DeviceFunc(var F: TTextRec): Integer;
```

Dabei ist *DeviceFunc* der Name der entsprechenden Funktion (d.h. *Open, InOut, Flush* oder *Close*). Weitere Informationen über den Typ *TTextRec* finden Sie in der Online-Hilfe. Der Rückgabewert einer Gerätetreiberfunktion ist immer der von *IOResult* zurückgegebene Wert. Wenn *IOResult* 0 zurückgibt, war die Operation erfolgreich.

Um die Gerätetreiberfunktionen einer bestimmten Datei zuzuordnen, müssen Sie selbst eine Zuordnungsprozedur implementieren. Diese Prozedur muß den vier Funktionszeigern der Textdateivariablen die Adressen der vier Gerätetreiberfunktionen zuweisen. Zusätzlich sollte sie dem Feld *Mode* die Konstante *fmClosed*, dem Feld *BufSize* die Größe des Textdateipuffers, dem Feld *BufPtr* einen Zeiger auf den Textdateipuffer und dem Feld *Name* einen Leerstring zuweisen.

Mit den Gerätetreiberfunktionen *DevOpen, DevInOut, DevFlush* und *DevClose* könnte die Prozedur *Assign* beispielsweise wie folgt aussehen:

```
procedure AssignDev(var F: Text);
begin
   with TextRec(F) do
   begin
    Mode := fmClosed;
    BufSize := SizeOf(Buffer);
    BufPtr := @Buffer;
    OpenFunc := @DevOpen;
    InOutFunc := @DevInOut;
    FlushFunc := @DevFlush;
    CloseFunc := @DevClose;
    Name[0] := #0;
   end;
end;
```

Die Gerätetreiberfunktionen können das Feld *UserData* im Datei-Record zum Speichern interner Informationen verwenden, da dieses Feld vom Delphi-Dateisystem nie geändert wird.

Gerätetreiberfunktionen

Nachfolgend werden die Funktionen beschrieben, aus denen Gerätetreiber für Textdateien bestehen.

Die Funktion Open

Die *Funktion Open* wird von den Standardprozeduren *Reset, Rewrite* und *Append* zum Öffnen von Textdateien verwendet, die einem Gerät zugeordnet sind. Das Feld *Mode* enthält beim Aufruf einen der Werte *fmInput, fmOutput* oder *fmInOut*, der angibt, ob *Open* von *Reset, Rewrite* oder *Append* aufgerufen wurde.

Open bereitet die Datei entsprechend dem Wert von *Mode* für die Ein- oder Ausgabe vor. Wenn *Mode* den Wert *fmInOut* hat (was bedeutet, daß *Open* von *Append* aufgerufen wurde), muß dieser in *fmOutput* geändert werden, bevor *Open* beendet wird.

Die Funktion *Open* wird immer vor allen anderen Gerätetreiberfunktionen aufgerufen. Aus diesem Grund initialisiert *AssignDev* nur das Feld *OpenFunc* und überläßt die Initialisierung der restlichen Felder der Funktion *Open*. Je nach dem Wert von *Mode* kann *Open* dann die Zeiger für die ein- oder ausgabeorientierten Funktionen installieren. Die Funktionen *InOut* und *Flush* und die Prozedur *CloseFile* brauchen also den aktuellen Modus nicht zu ermitteln.

Die Funktion InOut

Die *Funktion InOut* wird von den Standardroutinen *Read, Readln, Write, Writeln, Eof, Eoln, SeekEof, SeekEoln* und *CloseFile* immer dann aufgerufen, wenn eine Ein- oder Ausgabeoperation ansteht.

Wenn *Mode* den Wert *fmInput* hat, liest die Funktion *InOut* maximal *BufSize* Zeichen in `BufPtr^` und gibt in *BufEnd* die Anzahl der gelesenen Zeichen zurück. Außerdem wird *BufPos* der Wert 0 zugewiesen. Wenn die Funktion *InOut* als Ergebnis einer Eingabeaufforderung in *BufEnd* den Wert 0 zurückgibt, ist *Eof* für diese Datei *True*.

Wenn *Mode* auf *fmOutput* gesetzt ist, schreibt die Funktion *InOut BufPos* Zeichen aus `BufPtr^` und gibt in *BufPos* den Wert 0 zurück.

Die Funktion Flush

Die Funktion *Flush* wird nach jedem *Read, Readln, Write* und *Writeln* aufgerufen. Sie leert optional den Textdateipuffer.

Wenn *Mode* auf *fmInput* gesetzt ist, kann *Flush* in *BufPos* und *BufEnd* den Wert 0 speichern, um die verbleibenden (nicht gelesenen) Zeichen im Puffer zu löschen. Diese Möglichkeit wird aber nur selten verwendet.

Wenn *Mode* auf *fmOutput* gesetzt ist, kann *Flush* wie die Funktion *InOut* den Inhalt des Puffers schreiben. Dadurch wird sichergestellt, daß der geschriebene Text sofort angezeigt wird. Wenn *Flush* keine Aktion durchführt, wird der Text erst dann auf dem Gerät angezeigt, wenn der Puffer voll ist oder die Datei geschlossen wird.

Die Funktion Close

Die Funktion *Close* wird von der Standardprozedur *CloseFile* aufgerufen. Sie schließt eine Textdatei, die einem Gerät zugeordnet ist. Die Prozeduren *Reset, Rewrite* und *Append* rufen *Close* ebenfalls auf, wenn die zu öffnende Datei bereits geöffnet ist. Wenn *Mode* auf *fmOutput* gesetzt ist, ruft das Dateisystem vor dem Aufruf von *Close*

die Funktion *InOut* auf, um sicherzustellen, daß alle Zeichen auf das Gerät geschrieben wurden.

Nullterminierte Strings

Aufgrund der erweiterten Syntax von Object Pascal können die Standardprozeduren *Read*, *Readln*, *Str* und *Val* mit nullbasierten Zeichen-Arrays und die Standardprozeduren *Write*, *Writeln*, *Val*, *AssignFile* und *Rename* sowohl mit nullbasierten Zeichen-Arrays als auch mit Zeichenzeigern umgehen. Zusätzlich stehen die folgenden Funktionen zur Bearbeitung nullterminierter Strings zur Verfügung. Weitere Informationen zu nullterminierten Strings finden Sie unter »Nullterminierte Strings« auf Seite 5-14.

Tabelle 8.2 Funktionen für nullterminierte Strings

Funktion	Beschreibung
StrAlloc	Reserviert auf dem Heap einen Zeichenpuffer der angegebenen Größe.
StrBufSize	Gibt die Größe eines Zeichenpuffers zurück, der mit *StrAlloc* oder *StrNew* angelegt wurde.
StrCat	Verkettet zwei Strings.
StrComp	Vergleicht zwei Strings.
StrCopy	Kopiert einen String.
StrDispose	Gibt einen Zeichenpuffer frei, der mit *StrAlloc* oder *StrNew* angelegt wurde.
StrECopy	Kopiert einen String und gibt einen Zeiger auf das Ende des Strings zurück.
StrEnd	Gibt einen Zeiger auf das Ende eines Strings zurück.
StrFmt	Formatiert einen oder mehrere Werte in einem String.
StrIComp	Vergleicht zwei Strings ohne Beachtung der Groß-/Kleinschreibung.
StrLCat	Verkettet zwei Strings bei vorgegebener Maximallänge des resultierenden Strings.
StrLComp	Vergleicht die vorgegebene Maximallänge zweier Strings.
StrLCopy	Kopiert einen String bis zu einer vorgegebenen Maximallänge.
StrLen	Gibt die Länge eines Strings zurück.
StrLFmt	Formatiert einen oder mehrere Werte zur Positionierung in einem String mit vorgegebener Maximallänge.
StrLIComp	Vergleicht die Länge von zwei Strings ohne Beachtung der Groß-/Kleinschreibung.
StrLower	Konvertiert einen String in Kleinbuchstaben.
StrMove	Verschiebt eine Gruppe von Zeichen aus einem String in einen anderen.
StrNew	Legt einen String auf dem Heap an.
StrPCopy	Kopiert einen Pascal-String in einen nullterminierten String.
StrPLCopy	Kopiert einen Pascal-String in einen nullterminierten String mit vorgegebener Länge.
StrPos	Gibt einen Zeiger auf das erste Vorkommen eines bestimmten Teilstrings innerhalb eines Strings zurück.

Tabelle 8.2 Funktionen für nullterminierte Strings (Fortsetzung)

Funktion	Beschreibung
StrRScan	Gibt einen Zeiger auf das letzte Vorkommen eines bestimmten Teilstrings innerhalb eines Strings zurück.
StrScan	Gibt einen Zeiger auf das erste Vorkommen eines bestimmten Zeichens innerhalb eines Strings zurück.
StrUpper	Konvertiert einen String in Großbuchstaben.

Für die Standardfunktionen zur Stringbearbeitung gibt es jeweils Gegenstücke, die Multibyte-Zeichensätze unterstützen und die sprachspezifische Sortierfolge für Zeichen implementieren. Die Namen der Multibyte-Funktionen beginnen mit *Ansi-*. So ist *AnsiStrPos* beispielsweise die Multibyte-Version von *StrPos*. Die Unterstützung von Multibyte-Zeichen ist betriebssystemabhängig und basiert auf dem aktuellen Windows-Sprachtreiber.

Wide-Strings

Die Unit System stellt drei Funktionen zur Verfügung, die zur Umwandlung von nullterminierten Wide-Strings in lange Einzel- oder Doppelbyte-Strings verwendet werden können: *WideCharToString*, *WideCharLenToString* und *StringToWideChar*.

Weitere Informationen zu Wide-Strings finden Sie unter »Erweiterte Zeichensätze« auf Seite 5-13.

Weitere Standardroutinen

In der folgenden Tabelle finden Sie einige (aber nicht alle) der gebräuchlichsten Prozeduren und Funktionen aus den Delphi-Bibliotheken. Weitere Informationen zu diesen und weiteren Funktionen finden Sie in der Online-Hilfe.

Tabelle 8.3 Weitere Standardroutinen

Routine	Beschreibung
Abort	Beendet einen Prozeß ohne Fehlermeldung.
Addr	Gibt einen Zeiger auf das angegebene Objekt zurück.
AllocMem	Weist einen Speicherblock zu und initialisiert jedes Byte mit Null.
ArcTan	Berechnet den Arcustangens der angegebenen Zahl.
Assert	Überprüft, ob ein Boolescher Ausdruck *True* ist.
Assigned	Überprüft einen Zeiger oder eine prozedurale Variable auf **nil**.
Beep	Generiert einen Signalton, der über den Lautsprecher des Computers ausgegeben wird.
Break	Beendet eine **for-**, **while-** oder **repeat**-Anweisung vorzeitig.
ByteToCharIndex	Gibt die Position eines Zeichens in einem String zurück, das ein bestimmtes Byte enthält.
Chr	Gibt das Zeichen mit dem angegebenen ASCII-Wert zurück.

Tabelle 8.3 Weitere Standardroutinen (Fortsetzung)

Routine	Beschreibung
Close	Beendet die Zuordnung einer Dateivariablen zu einer externen Datei.
CompareMem	Führt einen binären Vergleich zweier Speicherbereiche durch.
CompareStr	Vergleicht Strings unter Berücksichtigung der Groß-/Kleinschreibung.
CompareText	Vergleicht Strings auf der Grundlage ihrer Ordinalwerte ohne Berücksichtigung der Groß-/Kleinschreibung.
Continue	Führt die nächste Iteration einer **for-**, **while-** oder **repeat**-Anweisung aus.
Copy	Gibt einen Teil eines Strings oder ein Segment eines dynamischen Arrays zurück.
Cos	Berechnet den Cosinus des angegebenen Winkels.
CurrToStr	Konvertiert einen Währungswert in einen String.
Date	Gibt das aktuelle Datum zurück.
DateTimeToStr	Konvertiert eine Variable des Typs *TDateTime* in einen String.
DateToStr	Konvertiert eine Variable des Typs *TDateTime* in einen String.
Dec	Erniedrigt eine ordinale Variable.
Dispose	Gibt den Speicherplatz frei, der einer dynamischen Variablen zugewiesen war.
ExceptAddr	Gibt die Adresse zurück, an der die aktuelle Exception ausgelöst wurde.
Exit	Beendet die aktuelle Prozedur.
Exp	Berechnet den Exponenten von X.
FillChar	Weist aufeinanderfolgenden Bytes einen bestimmten Wert zu.
Finalize	Finalisiert eine dynamisch zugewiesene Variable.
FloatToStr	Konvertiert einen Gleitkommawert in einen String.
FloatToStrF	Konvertiert einen Gleitkommawert in einen String mit einem bestimmten Format.
FmtLoadStr	Formatiert einen String anhand eines Format-Strings, der in einer Ressource hinterlegt ist.
FmtStr	Gibt einen formatierten AnsiString zurück, der aus einer Reihe von Array-Argumenten gebildet wird.
Format	Gibt einen formatierten AnsiString zurück, der aus einem Format-String und einer Reihe von Array-Argumenten gebildet wird.
FormatDateTime	Formatiert einen Datums-/Zeitwert.
FormatFloat	Formatiert einen Gleitkommawert.
FreeMem	Gibt eine dynamische Variable frei.
GetMem	Erzeugt eine dynamische Variable und einen Zeiger auf die Adresse des Blocks.
GetParentForm	Gibt das Formular oder die Eigenschaftsseite zurück, in dem bzw. der sich das angegebene Steuerelement befindet.
Halt	Bricht das Programm ab.
Hi	Gibt das höherwertige Byte eines Ausdrucks als vorzeichenlosen Wert zurück.
High	Gibt den höchsten Wert im Bereich eines Typs, Arrays oder Strings zurück.

Tabelle 8.3 Weitere Standardroutinen (Fortsetzung)

Routine	Beschreibung
Inc	Erhöht eine ordinale Variable.
Initialize	Initialisiert eine dynamisch zugewiesene Variable.
Insert	Fügt einen Teilstring an der angegebenen Position in einen String ein.
Int	Gibt den ganzzahligen Anteil einer reellen Zahl zurück.
IntToStr	Konvertiert einen Integer-Wert in einen AnsiString.
Length	Gibt die Länge eines Strings oder die Größe eines Arrays zurück.
Lo	Gibt das niederwertige Byte eines Ausdrucks als vorzeichenlosen Wert zurück.
Low	Gibt den niedrigsten Wert im Bereich eines Typs, Arrays oder Strings zurück.
LowerCase	Konvertiert einen ASCII-String in Kleinbuchstaben.
MaxIntValue	Gibt den größten vorzeichenbehafteten Wert in einem Integer-Array zurück.
MaxValue	Gibt den größten vorzeichenbehafteten Wert in einem Array zurück.
MinIntValue	Gibt den kleinsten vorzeichenbehafteten Wert in einem Integer-Array zurück.
MinValue	Gibt den kleinsten vorzeichenbehafteten Wert in einem Array zurück.
New	Erzeugt eine dynamische Variable und referenziert sie über den angegebenen Zeiger.
Now	Gibt das aktuelle Datum und die aktuelle Uhrzeit zurück.
Ord	Gibt den ordinalen Wert eines Ausdrucks mit ordinalem Typ zurück.
Pos	Gibt den Index des ersten Zeichens eines angegebenen Teilstrings innerhalb eines Strings zurück.
Pred	Gibt den Vorgänger eines ordinalen Wertes zurück.
Ptr	Konvertiert die angegebene Adresse in einen Zeiger.
Random	Generiert Zufallszahlen innerhalb eines angegebenen Bereichs.
ReallocMem	Weist eine dynamische Variable neu zu.
Round	Rundet eine reelle Zahl auf die nächste ganze Zahl.
SetLength	Legt die dynamische Länge einer String-Variablen oder eines Arrays fest.
SetString	Legt den Inhalt und die Länge eines bestimmten Strings fest.
ShowException	Zeigt eine Exception-Meldung und ihre physikalische Adresse an.
ShowMessage	Zeigt ein Meldungsfenster mit einem unformatierten String und der Schaltfläche *OK* an.
ShowMessageFmt	Zeigt ein Meldungsfenster mit einem formatierten String und der Schaltfläche *OK* an.
Sin	Gibt den Sinus eines Winkels zurück.
SizeOd	Gibt die Anzahl der Bytes zurück, die von einer Variablen oder einem Typ belegt werden.
Sqr	Gibt das Quadrat einer Zahl zurück.
Sqrt	Gibt die Quadratwurzel einer Zahl zurück.
Str	Formatiert einen String und gibt ihn an eine Variable zurück.
StrToCurr	Konvertiert einen String in einen Währungswert.

Tabelle 8.3 Weitere Standardroutinen (Fortsetzung)

Routine	Beschreibung
StrToDate	Konvertiert einen String in einen Datumswert (*TDateTime*-Objekt).
StrToDateTime	Konvertiert einen String in einen *TDateTime*-Wert.
StrToFloat	Konvertiert einen String in einen Gleitkommawert.
StrToInt	Konvertiert einen String in einen Integer-Wert.
StrToTime	Konvertiert einen String in ein *TDateTime*-Objekt.
StrUpper	Gibt einen String in Großbuchstaben zurück.
Succ	Gibt den Nachfolger einer Ordinalzahl zurück.
Sum	Berechnet die Summe aller Elemente eines Arrays.
Time	Gibt die aktuelle Uhrzeit zurück.
TimeToStr	Konvertiert eine Variable des Typs *TDateTime* in einen AnsiString.
Trunc	Konvertiert eine reelle Zahl in einen Integer-Wert.
UniqueString	Macht einen String eindeutig.
UpCase	Konvertiert ein Zeichen in einen Großbuchstaben.
UpperCase	Gibt einen String in Großbuchstaben zurück.
VarArrayCreate	Erzeugt ein variantes Array.
VarArrayDimCount	Gibt die Anzahl der Dimensionen in einem varianten Array zurück.
VarARrayHighBound	Gibt die Obergrenze einer bestimmten Dimension eines varianten Arrays zurück.
VarArrayLock	Sperrt ein variantes Array und gibt einen Zeiger auf die Daten zurück.
VarArrayLowBound	Gibt die Untergrenze einer bestimmten Dimension eines varianten Arrays zurück.
VarArrayOf	Erzeugt und füllt ein eindimensionales variantes Array.
VarArrayRedim	Ändert die Größe eines varianten Arrays.
VarArrayRef	Gibt eine Referenz auf das übergebene variante Array zurück.
VarArrayUnlock	Hebt die Sperrung eines varianten Arrays auf.
VarAsType	Konvertiert eine Variante in den angegebenen Datentyp.
VarCast	Konvertiert eine Variante in den angegebenen Datentyp und speichert das Ergebnis in einer Variablen.
VarClear	Löscht eine Variante.
CarCopy	Kopiert eine Variante.
VarToStr	Konvertiert eine Variante in einen String.
VarType	Gibt den Typencode der angegebenen Variante zurück.

Weitere Informationen zu Format-Strings finden Sie unter »Format-Strings« in der Online-Hilfe.

Teil

II

Spezielle Themen

Die Kapitel in Teil II behandeln spezielle Sprachelemente und weiterführende Themen.

- Kapitel 9, »Dynamische Link-Bibliotheken und Packages«
- Kapitel 10, »Objektschnittstellen«
- Kapitel 11, »Speicherverwaltung«
- Kapitel 12, »Ablaufsteuerung«
- Kapitel 13, »Der integrierte Assembler«

Kapitel

9

Dynamische Link-Bibliotheken und Packages

Eine dynamische Link-Bibliothek (DLL) ist eine Sammlung von Routinen, die von anderen Anwendungen oder DLLs aufgerufen werden können. DLLs enthalten wie Units gemeinsam genutzten Code und Ressourcen. Eine DLL ist jedoch eine separat compilierte, ausführbare Datei, die zur Laufzeit zu den Programmen gelinkt wird, die sie verwenden.

Um sie von eigenständigen ausführbaren Dateien unterscheiden zu können, haben Dateien, die compilierte DLLs enthalten, die Namenserweiterung DLL. Object-Pascal-Programme können DLLs aufrufen, die in anderen Sprachen geschrieben wurden. Windows-Anwendungen, die in anderen Sprachen programmiert sind, können DLLs aufrufen, die mit Object Pascal erstellt wurden.

DLLs aufrufen

Bevor Sie Routinen aufrufen können, die in einer DLL definiert sind, müssen Sie diese Routinen *importieren*. Der Import kann auf zwei Arten durchgeführt werden: Sie deklarieren eine Prozedur oder Funktion als **external**, oder Sie rufen die Windows-API direkt auf. Bei beiden Methoden werden die Routinen erst zur Laufzeit zur Anwendung gelinkt. Das bedeutet, daß die DLL zur Compilierzeit nicht benötigt wird. Es bedeutet aber auch, daß der Compiler nicht überprüfen kann, ob eine Routine korrekt importiert wird.

Variablen aus DLLs können in Object Pascal nicht importiert werden.

Statisches Laden

Die einfachste Art, eine Prozedur oder Funktion zu importieren, besteht darin, sie mit der Direktive **external** zu deklarieren:

```
procedure DoSomething; external 'MYLIB.DLL';
```

Durch diese Deklaration wird beim Programmstart die Datei MYLIB.DLL geladen. Während der Ausführung des Programms bezieht sich der Bezeichner *DoSomething* immer auf denselben Eintrittspunkt in derselben DLL.

Sie können die Deklaration einer importierten Routine direkt in das Programm oder die Unit einfügen, in dem bzw. der sie aufgerufen wird. Um Ihre Programme leichter pflegen zu können, sollten Sie aber alle **external**-Deklarationen in einer separaten »Import-Unit« zusammenfassen. Diese Unit enthält dann auch die Konstanten und Typen, die für die Schnittstelle zur DLL erforderlich sind (die Unit *Windows* von Delphi ist dafür ein gutes Beispiel). Andere Module, die auf die Import-Unit zugreifen, können alle darin deklarierten Routinen aufrufen.

Ausführliche Informationen über **external**-Deklarationen finden Sie im Abschnitt »Funktionen aus DLLs importieren« auf Seite 6-7.

Dynamisches Laden

Sie können auf die Routinen einer DLL über direkte Aufrufe von API-Funktionen zugreifen (z.B. *LoadLibrary*, *FreeLibrary* und *GetProcAddress*, die in der Delphi-Unit *Windows* deklariert sind). In diesem Fall werden die importierten Routinen über prozedurale Variablen referenziert:

```
uses Windows, ...;
type
   TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
   end;
   TGetTime = procedure(var Time: TTimeRec);
   THandle = Integer;
var
   Time: TTimeRec;
   Handle: THandle;
   GetTime: TGetTime;
   ⋮
begin
   Handle := LoadLibrary('DATETIME.DLL');
   if Handle <> 0 then
   begin
    @GetTime := GetProcAddress(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('Es ist ', Hour, ':', Minute, ':', Second);
    end;
    FreeLibrary(Handle);
   end;
end;
```

Wenn Sie Routinen auf diese Weise importieren, wird die DLL erst bei der Ausführung des Quelltextes geladen, der den Aufruf von *LoadLibrary* enthält. Später wird die DLL mit einem Aufruf von *FreeLibrary* wieder aus dem Speicher entfernt. Auf diese Weise wird Speicherplatz gespart und das Programm auch dann ausgeführt, wenn einige der aufgerufenen DLLs nicht zur Verfügung stehen.

DLLs schreiben

Die Struktur einer DLL ist identisch mit der Struktur eines Programms. Der einzige Unterschied besteht darin, daß eine DLL nicht mit **program**, sondern mit dem reservierten Wort **library** beginnt.

Das folgende Beispiel zeigt eine DLL mit den exportierten Funktionen *Min* und *Max:*

```
library MinMax;
function Min(X, Y: Integer): Integer; stdcall;
begin
   if X < Y then Min := X else Min := Y;
end;
function Max(X, Y: Integer): Integer; stdcall;
begin
   if X > Y then Max := X else Max := Y;
end;
exports
   Min index 1,
   Max index 2;
begin
end.
```

Sie können einer DLL auch Anwendungen zur Verfügung stellen, die in anderen Sprachen geschrieben wurden. Für diesen Zweck ist es am sichersten, wenn Sie in den Deklarationen exportierter Funktionen die Direktive **stdcall** angeben. Die standardmäßig verwendete Object-Pascal-Aufrufkonvention **register** wird nicht von allen anderen Sprachen unterstützt.

DLLs können aus mehreren Units bestehen. In diesem Fall enthält die Quelltextdatei der Bibliothek nur eine **uses**-Klausel, eine **exports**-Klausel und den Initialisierungscode der DLL:

```
library Editors;
uses EdInit, EdInOut, EdFormat, EdPrint;
exports
   InitEditors index 1,
   DoneEditors index 2,
   InsertText index 3,
   DeleteSelection index 4,
   FormatSelection index 5,
   PrintSelection index 6,
   .
   .
   .
   SetErrorHandler index  53;
begin
   InitLibrary;
end.
```

Andere Bibliotheken und Programme können nur auf die Routinen zugreifen, die eine Bibliothek explizit exportiert.

Die exports-Klausel

Eine Routine wird exportiert, wenn sie in einer **exports**-Klausel wie der folgenden angegeben wird:

```
exports Eintritt1, ..., Eintrittn;
```

Eintritt steht für den Namen einer Prozedur oder Funktion (die zuvor mit der **exports**-Klausel deklariert werden muß). Auf den Namen kann optional ein **index**- und ein **name**-Bezeichner folgen. Der Name der Prozedur oder Funktion kann optional mit dem Namen einer Unit qualifiziert werden.

(Eintrittspunkte können außerdem die Direktive **resident** enthalten, die der Abwärtskompatibilität dient und vom Compiler ignoriert wird.)

Ein **index**-Bezeichner besteht aus der Direktive **index** und einer numerischen Konstanten von 1 bis 2.147.483.647 (effizientere Programme erhalten Sie mit niedrigen Index-Werten). Ist kein **index**-Bezeichner angegeben, wird der Routine in der Exporttabelle der DLL automatisch eine Nummer zugewiesen.

Ein **name**-Bezeichner besteht aus der Direktive **name** und einer nachfolgenden String-Konstante. Verfügt ein Eintrittspunkt über keinen **name**-Bezeichner, wird die Routine unter ihrem ursprünglich deklarierten Namen (in derselben Schreibweise) exportiert. Verwenden Sie die **name**-Klausel, wenn Sie eine Routine unter einem anderen Namen exportieren wollen:

```
exports
  DoSomethingABC index 1 name 'DoSomething';
```

Die **exports**-Klausel kann im Deklarationsteil des Programms oder der Bibliothek an beliebigen Stellen und beliebig oft angegeben werden. Programme enthalten nur selten eine **exports**-Klausel, weil Anwendungen unter Windows keine Routinen exportieren dürfen.

Code für die Initialisierung der Bibliothek

Die Anweisungen im Block einer Bibliothek bilden den *Initialisierungscode* der Bibliothek. Diese Anweisungen werden nur einmal beim Laden der Bibliothek ausgeführt. Mit diesen Anweisungen werden beispielsweise Fensterklassen registriert und Variablen initialisiert. Außerdem kann der Initialisierungscode mit Hilfe der Variablen *ExitProc* eine Exit-Prozedur installieren (Informationen dazu finden Sie im Abschnitt »Exit-Prozeduren« auf Seite 12-4). Die Exit-Prozedur wird ausgeführt, wenn die DLL aus dem Speicher entfernt wird.

Der Initialisierungscode einer Bibliothek kann einen Fehler signalisieren. Zu diesem Zweck wird der Variablen *ExitCode* ein Wert zugewiesen, der ungleich Null ist. *ExitCode* ist in der Unit *System* deklariert und hat den Standardwert Null. Dieser Wert gibt an, daß die Initialisierung erfolgreich war. Wenn der Initialisierungscode der Bibliothek der Variablen *ExitCode* einen anderen Wert zuweist, wird die DLL aus dem

Speicher entfernt, und die aufrufende Anwendung wird über den Fehler benachrichtigt. Tritt während der Ausführung des Initialisierungscodes eine nicht verarbeitete Exception auf, wird die aufrufende Anwendung über einen Fehler beim Laden der DLL benachrichtigt.

Hier ein Beispiel für eine Bibliothek mit Initialisierungscode und Exit-Prozedur:

```
library Test;
var
   SaveExit: Pointer;
procedure LibExit;
begin
   ⋮ // Exit-Code der Bibliothek
   ExitProc := SaveExit;  // Kette der Exit-Prozeduren wiederherstellen
end;
begin
   ⋮ // Initialisierungscode der Bibliothek
   SaveExit := ExitProc;  // Kette der Exit-Prozeduren speichern
   ExitProc := @LibExit;  // Exit-Prozedur LibExit installieren
end.
```

Sobald eine DLL aus dem Speicher entfernt wird, werden die Exit-Prozeduren der Bibliothek ausgeführt. Dazu wird die in *ExitProc* gespeicherte Adresse immer wieder aufgerufen, bis *ExitProc* den Wert **nil** hat. Die Initialisierungsteile aller von einer Bibliothek verwendeten Units werden vor dem Initialisierungscode der Bibliothek, die finalization-Abschnitte nach der Exit-Prozedur der Bibliothek ausgeführt.

Globale Variablen in einer DLL

In einer DLL deklarierte globale Variablen können von Object-Pascal-Anwendungen nicht importiert werden.

Eine DLL kann von mehreren Anwendungen gleichzeitig verwendet werden. Jede Anwendung verfügt aber in ihrem Verarbeitungsbereich über eine Kopie der DLL mit einem eigenen Satz globaler Variablen. Damit mehrere DLLs (oder mehrere Instanzen einer DLL) den Speicher gemeinsam nutzen können, müssen die DLLs Speicherzuordnungstabellen verwenden. Weitere Informationen zu diesem Thema finden Sie in der Dokumentation zur Windows-API.

DLLs und Systemvariablen

Einige der in der Unit *System* deklarierten Variablen sind für Programmierer von besonderem Interesse. Mit *IsLibrary* können Sie feststellen, ob der Code in einer Anwendung oder in einer DLL ausgeführt wird. IsLibrary ist in einer Anwendung immer *False*, in einer DLL dagegen immer *True*. Während der Lebensdauer einer DLL enthält die Variable *HInstance* das **Instanzen**-Handle der DLL. Die Variable *CmdLine* ist in einer DLL immer **nil**.

Die Variable *DLLProc* ermöglicht einer DLL die Überwachung der Betriebssystemaufrufe an ihrem Eintrittspunkt. Sie wird normalerweise nur von DLLs verwendet,

die Multithreading unterstützen. Für die Überwachung von Betriebssystemaufrufen erstellen Sie eine Callback-Prozedur mit einem Integer-Parameter.

```
procedure DLLHandler(Reason: Integer);
```

Außerdem müssen Sie der Variable *DLLProc* die Adresse der Prozedur zuweisen. Wenn Windows die Prozedur aufruft, wird ihr einer der folgenden, in der Unit *Windows* definierten Werte übergeben:

DLL_PROCESS_DETACH	Gibt an, daß die DLL als Ergebnis einer Beendigungsprozedur oder eines Aufrufs von *FreeLibrary* vom Adreßraum des aufrufenden Prozesses getrennt wird.
DLL_THREAD_ATTACH	Gibt an, daß der aktuelle Prozeß einen neuen Thread erstellt.
DLL_THREAD_DETACH	Gibt an, daß ein Thread ohne Probleme beendet wurde.

Sie können die Aktionen im Rumpf der Prozedur davon abhängig machen, welcher Parameter an die Prozedur übergeben wird.

Exceptions und Laufzeitfehler in DLLs

Wenn in einer DLL eine Exception erzeugt, aber nicht behandelt wird, wird sie nach außen an den Aufrufer weitergegeben. Wenn die aufrufende Anwendung oder DLL in Object Pascal geschrieben wurde, kann die Exception in einer normalen **try...except**-Anweisung behandelt werden. Wenn die aufrufende Anwendung oder DLL in einer anderen Sprache entwickelt wurde, kann die Exception wie eine Betriebssystem-Exception mit dem Code $0EEDFACE behandelt werden. Der erste Eintrag im *Array ExceptionInformation* des Records mit der Betriebssystem-Exception enthält die Adresse, der zweite Eintrag eine Referenz auf das Exception-Objekt von Object Pascal.

Wenn in einer DLL die Unit *SysUtils* nicht verwendet wird, ist die Unterstützung von Exceptions in Delphi deaktiviert. Tritt in diesem Fall in einer DLL ein Laufzeitfehler auf, wird die aufrufende Anwendung beendet. Da die DLL nicht feststellen kann, ob sie von einem Object-Pascal-Programm aufgerufen wurde, kann sie auch nicht die Exit-Prozeduren der Anwendung aufrufen. Die Anwendung wird einfach beendet und aus dem Speicher entfernt.

Der Shared-Memory-Manager

Wenn eine DLL Routinen exportiert, die lange Strings oder dynamische Arrays als Parameter oder Funktionsergebnisse übergeben (entweder direkt oder in Records bzw. Objekten), müssen die DLL und ihre Client-Anwendungen (oder DLLs) die Unit *ShareMem* verwenden. Dasselbe gilt, wenn eine Anwendung oder DLL mit *New* oder *GetMem* Speicherplatz reserviert, der in einem anderen Modul durch einen Aufruf von *Dispose oder FreeMem* wieder freigegeben wird. *ShareMem* sollte in der **uses**-Klausel von Programmen oder Bibliotheken, von denen sie eingebunden wird, immer als erste *Unit* genannt werden.

ShareMem ist die Schnittstellen-Unit für den Speicher-Manager BORLANDMM.DLL, der es Modulen ermöglicht, dynamisch zugewiesenen Speicherplatz gemeinsam zu nutzen. BORLANDMM.DLL muß mit Anwendungen und DLLs weitergegeben werden, die *ShareMem* einbinden. Wenn eine Anwendung oder DLL *ShareMem* einsetzt, wird der Speicher-Manager der Anwendung oder DLL durch den Speicher-Manager BORLANDMM.DLL ersetzt.

Packages

Ein Package ist eine auf spezielle Weise compilierte dynamische Link-Bibliothek, die von Delphi-Anwendungen und/oder von der Delphi-IDE verwendet wird. *Laufzeit-Packages* stellen die Funktionalität bereit, die dem Benutzer die Ausführung einer Anwendung ermöglicht. *Entwurfszeit-Packages* dienen der Installation von Komponenten in der Delphi-IDE und der Erstellung von speziellen Eigenschaftseditoren für benutzerdefinierte Komponenten. Jedes Package kann sowohl zur Entwurfszeit als auch zur Laufzeit verwendet werden. *Entwurfszeit-Packages* referenzieren häufig *Laufzeit-Packages* in ihren **requires**-Klauseln.

Zur Unterscheidung von anderen DLLs werden Package-Bibliotheken in Dateien mit der Namenserweiterung BPL (Borland Package Library) gespeichert.

Normalerweise werden Packages beim Start einer Anwendung statisch geladen. Mit Hilfe der Routinen *LoadPackage* und *UnloadPackage* (in der Unit *SysUtils*) können Sie Packages aber auch dynamisch laden.

Hinweis Wenn eine Anwendung Packages verwendet, muß der Name jeder dort eingebundenen Unit weiterhin in der **uses**-Klausel jeder Quelltextdatei angegeben werden, von der die Unit referenziert wird. Ausführliche Informationen zu Packages finden Sie in der Online-Hilfe.

Package-Deklarationen und Quelltextdateien

Um eine Verwechslung mit anderen Dateien zu vermeiden, die Object-Pascal-Code enthalten, wird jedes Package in einer separaten Quelltextdatei mit der Namenserweiterung DPK (Delphi Package) gespeichert. Eine Package-Quelltextdatei enthält keinerlei Typ-, Daten-, Prozedur- oder Funktionsdeklarationen, sondern nur die folgenden Elemente:

- Einen *Namen* für das Package.
- Eine Liste weiterer Packages, die vom neuen Package benötigt werden. Das neue Package wird zu diesen Packages gelinkt.
- Eine Liste der Unit-Dateien, die das compilierte Package enthält. Das Package stellt im Grund eine Hülle für diese Quelltext-Units dar, die die Funktionalität der compilierten BPL bereitstellen.

Eine Package-Deklaration hat folgende Form:

```
package PackageName;
   requiresKlausel;
```

```
   containsKlausel;
end.
```

PackageName ist ein gültiger Bezeichner. Die Angabe von *requiresKlausel* und *containsKlausel* ist optional. Im folgenden Beispiel wird das Package *VCLDB40* deklariert:

```
package VCLDB40;
   requires VCL40;
   contains Db, Dbcgrids, Dbctrls, Dbgrids, ... ;
end.
```

Die **requires**-Klausel enthält weitere externe Packages, die vom deklarierten Package verwendet werden. Die Klausel setzt sich aus der Direktive **requires**, einer Liste mit Package-Namen, die durch Kommas voneinander getrennt sind, und einem Strichpunkt zusammen. Wenn ein Package keine weiteren Packages referenziert, ist keine **requires**-Klausel erforderlich.

Die **contains**-Klausel gibt die Unit-Dateien an, die compiliert und in das Package eingebunden werden sollen. Die Klausel setzt sich aus der Direktive **contains**, einer Liste mit Unit-Namen, die durch Kommas voneinander getrennt sind, und einem Strichpunkt zusammen. Auf einen Unit-Namen kann das reservierte Wort **in** und der Name einer Quelltextdatei mit oder ohne Pfadangabe in halben Anführungszeichen folgen. Verzeichnispfade können absolut oder relativ sein. Ein Beispiel:

```
contains MyUnit in 'C:\MyProject\MyUnit.pas';
```

Hinweis Auf Thread-Variablen (Variablen, die mit **threadvar** deklariert wurden) in einer Package-Unit können Clients, die dieses Package verwenden, nicht zugreifen.

Packages benennen

Beim Compilieren eines Package werden mehrere Dateien erzeugt. Die Quelldatei für das Package *VCL40* ist beispielsweise VCL40.DPK, aus der der Compiler eine ausführbare Datei und ein binäres Abbild (VCL40.BPL und VCL40.DCP) erzeugt. Über den Namen VCL40 wird das Package in der **requires**-Klausel anderer Packages referenziert oder in einer Anwendung verwendet. Package-Namen müssen innerhalb eines Projekts eindeutig sein.

Die requires-Klausel

Die **requires**-Klausel enthält andere externe Packages, die vom aktuellen Package verwendet werden. Diese Klausel funktioniert wie die **uses**-Klausel in einer Unit. Ein externes Package, das in der **requires**-Klausel enthalten ist, wird beim Compilieren automatisch zu jeder Anwendung gelinkt, die sowohl das neue Package als auch eine der Units verwendet, die im externen Package enthalten sind.

Wenn die Unit-Dateien eines Package andere Units referenzieren, die ebenfalls in Packages enthalten sind, sollten diese anderen Packages im **requires**-Abschnitt des ersten Package angegeben werden. Andernfalls lädt der Compiler die referenzierten Units aus den entsprechenden DCU-Dateien.

Zirkuläre Bezüge bei Packages vermeiden

Da Packages in ihren **requires**-Klauseln keine zirkulären Bezüge herstellen dürfen, müssen sie folgende Bedingungen erfüllen:

- Sie dürfen sich im **requires**-Abschnitt nicht selbst referenzieren.
- Eine Folge von Referenzen darf keine rückwärtsgerichtete Referenz enthalten. Wenn Package *A* in seinem **requires**-Abschnitt Package *B* referenziert, kann Package *B* nicht Package *A* referenzieren. Wenn Package *A* Package *B* und Package *B* Package *C* referenziert, kann *C* nicht *A* referenzieren.

Doppelte Package-Referenzen

Der Compiler ignoriert doppelte Referenzen in der **requires**-Klausel eines Package. Wegen der besseren Lesbarkeit Ihrer Programme sollten Sie derartige Referenzen aber entfernen.

Die contains-Klausel

Die **contains**-Klausel enthält die Unit-Dateien (ohne Dateinamenserweiterungen), die in das Package eingebunden werden sollen.

Redundante Verwendung von Quelltext vermeiden

Ein Package kann nicht in der **contains**-Klausel eines anderen Package oder in der **uses**-Klausel einer Unit enthalten sein.

Alle Units, die direkt in der **contains**-Klausel eines Package oder indirekt in **uses**-Klauseln der betreffenden Units enthalten sind, werden zur Compilierzeit in das Package eingebunden. Die Units, die (direkt oder indirekt) in einem Package vorhanden sind, dürfen in keinem der anderen Packages enthalten sein, die in der **requires**-Klausel dieses Package referenziert werden.

Eine Unit kann weder direkt noch indirekt in mehr als einem Package einer Anwendung enthalten sein.

Packages compilieren

Packages werden normalerweise in der Delphi-IDE unter Verwendung von DPK-Dateien erstellt, die mit dem Package-Editor generiert werden. Sie können DPK-Dateien auch direkt in der Befehlszeile anlegen. Wenn Sie ein Projekt neu compilieren, das ein Package enthält, wird das Package implizit neu compiliert, wenn dies erforderlich ist.

Generierte Dateien

Die folgende Tabelle enthält die Dateien, die beim erfolgreichen Compilieren eines Package generiert werden.

Tabelle 9.1 Dateien eines compilierten Package

Namenserweiterung	Inhalt
DCP	Ein binäres Abbild, das einen Package-Header und die Verkettung aller DCU-Dateien des Package enthält. Für jedes Package wird eine DCP-Datei erzeugt. Der Basisname der DCP-Datei entspricht dem Basisnamen der DPK-Quelltextdatei.
DCU	Ein binäres Abbild einer Unit, die in einem Package enthalten ist. Für jede Unit wird bei Bedarf eine DCU-Datei generiert.
BPL	Das Laufzeit-Package. Diese Datei ist eine Windows-DLL mit Delphi-spezifischen Eigenschaften. Der Basisname der BPL-Datei entspricht dem Basisnamen der DPK-Quelltextdatei.

Zur Unterstützung der Package-Compilierung stehen mehrere Compiler-Direktiven und Befehlszeilenoptionen zur Verfügung.

Package-spezifische Compiler-Direktiven

Die folgende Tabelle enthält die Package-spezifischen Compiler-Direktiven, die in den Quelltext eingefügt werden können. Ausführliche Informationen dazu finden Sie in der Online-Hilfe.

Tabelle 9.2 Compiler-Direktiven für Packages

Direktive	Beschreibung
{$IMPLICITBUILD OFF}	Verhindert, daß ein Package in Zukunft implizit neu compiliert wird. Diese Direktive wird in DPK-Dateien verwendet, wenn Packages mit Low-Level-Funktionen compiliert werden, die sich nur selten ändern, oder wenn der Quelltext des Package nicht weitergegeben wird.
{$G–} oder {IMPORTEDDATA OFF}	Deaktiviert die Erstellung von Referenzen auf importierte Daten. Diese Direktive führt zu schnelleren Speicherzugriffen, verhindert aber, daß die Unit, in der sie verwendet wird, Variablen in anderen Packages referenzieren kann.
{$WEAKPACKAGEUNIT ON}	Units werden »weich« eingebunden. Informationen hierzu finden Sie in der Online-Hilfe.
{$DENYPACKAGEUNIT ON}	Verhindert, daß die Unit in ein Package eingebunden wird.
{$DESIGNONLY ON}	Das Package wird für die Installation in der Delphi-IDE compiliert (nur in DPK-Dateien).
{$RUNONLY ON}	Das Package wird nur als Laufzeit-Package compiliert (nur in DPK-Dateien).

Die Verwendung von **{$DENYPACKAGEUNIT ON}** im Quelltext verhindert, daß die Unit in ein Package eingebunden wird. **{$G–}** oder {**IMPORTEDDATA OFF}** kön-

nen unter Umständen verhindern, daß ein Package in derselben Anwendung zusammen mit anderen Packages eingesetzt werden kann.

Bei Bedarf können auch andere Compiler-Direktiven in den Quelltext eines Package aufgenommen werden.

Package-spezifische Befehlszeilenoptionen

Die folgenden Package-spezifischen Optionen stehen für den Befehlszeilen-Compiler zur Verfügung. Ausführlichere Informationen dazu finden Sie in der Online-Hilfe.

Tabelle 9.3 Befehlszeilenoptionen für Packages

Option	Beschreibung
–$G–	Deaktiviert die Erstellung von Referenzen auf importierte Daten. Diese Option führt zu beschleunigten Speicherzugriffen, verhindert aber, daß mit dieser Option compilierte Packages Variablen in anderen Packages referenzieren.
–LE Pfad	Gibt das Verzeichnis an, in dem die BPL-Datei des Package abgelegt wird.
–LN Pfad	Gibt das Verzeichnis an, in dem die DCP-Datei des Package abgelegt wird.
–LuPackageName [;PackageName2;...]	Gibt zusätzliche Laufzeit-Packages für die Verwendung in einer Anwendung an. Wird bei der Compilierung eines Projekts verwendet.
–Z	Verhindert, daß ein Package in Zukunft implizit neu compiliert wird. Diese Option wird verwendet, wenn Packages mit Low-Level-Funktionen compiliert werden, die sich nur selten ändern, oder wenn der Quelltext des Package nicht weitergegeben wird.

Die Verwendung der Option **–$G–** führt möglicherweise dazu, daß ein Package nicht mit anderen Packages in derselben Anwendung eingesetzt werden kann.

Bei Bedarf können für die Compilierung von Packages auch andere Befehlszeilenoptionen angegeben werden.

Kapitel

10

Objektschnittstellen

Eine *Objektschnittstelle* (oder einfach nur *Schnittstelle*) definiert Methoden, die von einer Klasse implementiert werden können. Schnittstellen werden wie Klassen deklariert. Sie können aber nicht direkt instantiiert werden und verfügen auch nicht über eigene Methodendefinitionen. Es liegt vielmehr in der Verantwortung der Klasse, von der eine Schnittstelle unterstützt wird, für die Implementierung von Schnittstellenmethoden zu sorgen. Eine Variable vom Typ der Schnittstelle kann ein Objekt referenzieren, dessen Klasse die betreffende Schnittstelle implementiert. Über diese Variable können aber nur die Methoden aufgerufen werden, die in der Schnittstelle deklariert sind.

Schnittstellen bieten einige Vorteile der Mehrfachvererbung, umgehen aber deren semantische Probleme. Außerdem sind Sie bei der Verwendung von verteilten Objektmodellen wie COM (Component Object Model) und CORBA (Common Object Request Broker Architecture) von größter Bedeutung. Mit Delphi erstellte Objekte, die Schnittstellen unterstützen, können mit COM-Objekten interagieren, die in C++, Java oder anderen Sprachen geschrieben wurden.

Schnittstellentypen deklarieren

Schnittstellen können wie Klassen nur im äußersten Gültigkeitsbereich eines Programms oder einer Unit, nicht aber in einer Prozedur oder Funktion deklariert werden. Die Deklaration eines Schnittstellentyps sieht folgendermaßen aus:

```
type Schnittstellenname = interface (Vorfahrschnittstelle)
   ['{GUID}']
   Elementliste
end;
```

Vorfahrschnittstelle und *GUID* sind optional. Eine Schnittstellendeklaration ähnelt in weiten Teilen einer Klassendeklaration. Es gelten jedoch folgende Einschränkungen:

- Die *Elementliste* darf nur Methoden und Eigenschaften enthalten. Felder sind in Schnittstellen nicht erlaubt.

- Da für Schnittstellen keine Felder verfügbar sind, müssen die Zugriffsattribute für Eigenschaften (**read** und **write**) Methoden sein.
- Alle Elemente einer Schnittstelle sind als **public** deklariert. Sichtbarkeitsattribute und Speicherattribute sind nicht erlaubt. Es kann aber eine Array-Eigenschaft mit der Direktive **default** als Standardeigenschaft deklariert werden.
- Schnittstellen haben keine Konstruktoren oder Destruktoren. Sie können nicht instantiiert werden, ausgenommen durch Klassen, über die die Methoden der Schnittstelle implementiert werden.
- Methoden können nicht als **virtual**, **dynamic**, **abstract** oder **override** deklariert werden. Da Schnittstellen keine eigenen Methoden implementieren, haben diese Bezeichnungen keine Bedeutung.

Hier ein Beispiel für eine Schnittstellendeklaration:

```
type
   IMalloc = interface(IUnknown)
    ['{00000002-0000-0000-C000-000000000046}']
    function Alloc(Size: Integer): Pointer; stdcall;
    function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
    procedure Free(P: Pointer); stdcall;
    function GetSize(P: Pointer): Integer; stdcall;
    function DidAlloc(P: Pointer): Integer; stdcall;
    procedure HeapMinimize; stdcall;
   end;
```

IUnknown und Vererbung

Eine Schnittstelle erbt wie eine Klasse alle Methoden ihres Vorfahren. Schnittstellen *implementieren* aber im Gegensatz zu Klassen keine Methoden. Eine Schnittstelle erbt die *Verpflichtung* zur Implementation von Methoden. Diese Verpflichtung geht auf alle Klassen über, die die Schnittstelle unterstützen.

In der Deklaration einer Schnittstelle kann eine Vorfahrschnittstelle angegeben werden. Wird kein Vorfahr festgelegt, ist die Schnittstelle ein direkter Nachkomme der Schnittstelle *IUnknown*, die in der Unit *System* definiert ist und den absoluten Vorfahr aller Schnittstellen darstellt. *IUnknown* deklariert drei Methoden: *QueryInterface*, *_AddRef* und *_Release*. *QueryInterface* stellt die Mittel bereit, mit deren Hilfe die verschiedenen Schnittstellen, die ein Objekt unterstützen, angesprochen werden können. *_AddRef* und *_Release* sorgen während der Lebensdauer einer Schnittstelle für die Verwaltung der Schnittstellenreferenzen. Die einfachste Art, diese Methoden zu implementieren, besteht darin, die implementierende Klasse von *TInterfacedObject* in der Unit *System* abzuleiten.

Identifikation einer Schnittstelle

Die Deklaration einer Schnittstelle kann einen global eindeutigen Bezeichner (GUID) enthalten, der als String-Literal in eckigen Klammern unmittelbar vor der Elementliste steht. Der GUID-Abschnitt der Deklaration hat folgende Form:

```
['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']
```

Dabei steht jedes *x* für eine hexadezimale Ziffer (0 bis 9 und A bis F).

Eine GUID (Schnittstellen-ID) ist ein binärer 16-Byte-Wert, der eine Schnittstelle eindeutig bezeichnet. Wenn eine Schnittstelle eine GUID hat, können Sie über eine Schnittstellenabfrage Referenzen auf ihre Implementationen abrufen (siehe »Schnittstellenabfragen« auf Seite 10-13.)

Die Typen TGUID und PGUID, die in der Unit *System* deklariert sind, werden zur Bearbeitung von GUIDs eingesetzt:

```
type
  PGUID = ^TGUID;
  TGUID = record
   D1: Integer;
   D2: Word;
   D3: Word;
   D4: array[0..7] of Byte;
  end;
```

Wenn Sie eine typisierte Konstante vom Typ TGUID deklarieren, können Sie ihren Wert als String-Literal angeben. Ein Beispiel:

```
const IID_IMalloc: TGUID = '{00000002-0000-0000-C000-000000000046}';
```

In Prozedur- und Funktionsaufrufen kann entweder eine GUID oder ein Schnittstellenbezeichner als Wert- oder Konstantenparameter vom Typ TGUID fungieren. Betrachten Sie dazu die folgende Deklaration:

```
function Supports(Unknown: IUnknown; const IID: TGUID): Boolean;
```

Supports kann auf zwei Arten aufgerufen werden:

```
if Supports(Allocator, IMalloc) then ...
if Supports(Allocator, IID_IMalloc) then ...
```

Aufrufkonventionen

Die Standard-Aufrufkonvention für Schnittstellen ist **register.** Bei Schnittstellen, die von verschiedenen Modulen gemeinsam benutzt werden, sollten alle Methoden mit **stdcall** deklariert werden. Dies gilt insbesondere dann, wenn diese Module in verschiedenen Programmiersprachen erstellt wurden. Methoden von dualen Schnittstellen (siehe »Duale Schnittstellen« auf Seite 10-13) und CORBA-Schnittstellen werden mit **safecall** implementiert.

Ausführliche Informationen über Aufrufkonventionen finden Sie im Abschnitt »Aufrufkonventionen« auf Seite 6-5.

Schnittstelleneigenschaften

Auf Eigenschaften, die in einer Schnittstelle deklariert werden, kann nur über Ausdrücke vom Typ der Schnittstelle zugegriffen werden. Der Zugriff über Variablen vom Typ der Klasse ist nicht möglich. Außerdem sind Schnittstelleneigenschaften

nur in Programmen sichtbar, in denen die Schnittstelle compiliert wird. COM-Objekte haben keine Eigenschaften.

Da in Schnittstellen keine Felder verfügbar sind, müssen die Zugriffsattribute für Eigenschaften (**read** und **write**) Methoden sein.

Vorwärtsdeklarationen

Eine Schnittstellendeklaration, die mit dem reservierten Wort **interface** und einem Strichpunkt endet und keinen Vorfahr, keine GUID und keine Elementliste enthält, wird als *Vorwärtsdeklaration* bezeichnet. Eine Vorwärtsdeklaration muß durch eine *definierende Deklaration* derselben Schnittstelle innerhalb desselben Typdeklarationsabschnitts aufgelöst werden. Das bedeutet, daß zwischen einer *Vorwärtsdeklaration* und ihrer definierenden Deklaration ausschließlich andere Typdeklarationen stehen dürfen.

Vorwärtsdeklarationen ermöglichen voneinander abhängige Schnittstellen. Ein Beispiel:

```
type
  IControl = interface;
  IWindow = interface
   ['{00000115-0000-0000-C000-000000000044}']
   function GetControl(Index: Integer): IControl;
   ⋮
  end;
  IControl = interface
   ['{00000115-0000-0000-C000-000000000049}']
   function GetWindow: IWindow;
   ⋮
  end;
```

Es ist nicht möglich, Schnittstellen gegenseitig voneinander abzuleiten (z.B. *IWindow* von *IControl* und *IControl* von *IWindow*).

Schnittstellen implementieren

Nach der Deklaration muß die Schnittstelle in einer Klasse implementiert werden, bevor sie verwendet werden kann. Die in einer Klasse implementierten Schnittstellen werden in der Klassendeklaration nach dem Namen der Vorfahrklasse angegeben. Die Deklaration sieht folgendermaßen aus:

```
type Klassenname = class (Vorfahrklasse, Schnittstelle1, ..., Schnittstellen)
  Elementliste
  end;
```

Im folgenden Beispiel wird eine Klasse namens *TMemoryManager* deklariert, die die Schnittstellen *IMalloc* and *IErrorInfo* implementiert:

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
   ⋮
```

```
  end;
```

Wenn eine Klasse eine Schnittstelle implementiert, muß sie alle in der Schnittstelle deklarierten Methoden implementieren (oder eine Implementation jeder Methode erben).

Hier die Deklaration von *TInterfacedObject* in der Unit *System*:

```
type
  TInterfacedObject = class(TObject, IUnknown)
  protected
   FRefCount: Integer;
   function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
   function _AddRef: Integer; stdcall;
   function _Release: Integer; stdcall;
  public
   property RefCount: Integer read FRefCount;
  end;
```

TInterfacedObject implementiert die Schnittstelle *IUnknown*. Daher deklariert und implementiert *TInterfacedObject* jede der drei Methoden von *IUnknown*.

Klassen, die Schnittstellen implementieren, können auch als Basisklassen verwendet werden. (Im ersten der obigen Beispiele wird *TMemoryManager* als direkter Nachkomme von *TInterfacedObject* deklariert.) Da jede Schnittstelle die Methoden von *IUnknown* erbt, muß eine Klasse, die Schnittstellen implementiert, auch die Methoden *QueryInterface*, *_AddRef* und *_Release* implementieren. *TInterfacedObject* in der Unit *System* implementiert diese Methoden und eignet sich aus diesem Grund als Basis für weitere Klassen, die Schnittstellen implementieren.

Nach der Implementierung einer Schnittstelle wird jede ihrer Methoden einer Methode der implementierenden Klasse zugeordnet, die denselben Ergebnistyp, dieselbe Aufrufkonvention und dieselbe Anzahl von Parametern hat (wobei entsprechende Parameter auch identische Typen haben müssen). Standardmäßig wird jede Methode der Schnittstelle der gleichnamigen Methode der implementierenden Klasse zugewiesen.

Methodenzuordnung

Das Standardverfahren der Methodenzuordnung in einer Klassendeklaration kann mit Hilfe von *Methodenzuordnungsklauseln* außer Kraft gesetzt werden. Wenn eine Klasse zwei oder mehr Schnittstellen mit identischen Methoden implementiert, können Sie die Namenskonflikte, die sich daraus ergeben, mit Hilfe von Methodenzuordnungsklauseln lösen.

Eine Methodenzuordnungsklausel sieht folgendermaßen aus:

```
procedure Schnittstelle.Schnittstellenmethode = implementierndeMethode;
```

oder

```
function Schnittstelle.Schnittstellenmethode = implementierendeMethode;
```

Dabei ist *implementierendeMethode* eine in der Klasse deklarierte Methode oder eine Methode eines der Klassenvorfahren. Es kann sich dabei um eine an späterer Stelle

deklarierte Methode handeln, nicht aber um eine als **private** deklarierte Methode einer Vorfahrklasse, die in einem anderen Modul deklariert ist.

In der folgenden Klassendeklaration werden beispielsweise die Methoden *Alloc* und *Free* von *IMalloc* den Methoden *Allocate* und *Deallocate* von *TMemoryManager* zugeordnet:

```
type
   TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
    ⋮
   end;
```

Die in einer Vorfahrklasse festgelegten Zuordnungen können in Methodenzuordnungsklauseln nicht geändert werden.

Geerbte Implementationen ändern

Untergeordnete Klassen können die Implementation einer bestimmten Schnittstellenmethode durch Überschreiben ändern. Bei der implementierenden Methode muß es sich dabei um eine virtuelle oder dynamische Methode handeln.

Eine Klasse kann auch eine gesamte, von einer Vorfahrklasse geerbte Schnittstelle neu implementieren. Dazu muß die Schnittstelle in der Deklaration der untergeordneten Klasse nochmals aufgeführt werden. Ein Beispiel:

```
type
   IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
    procedure Draw;
    ⋮
   end;
   TWindow = class(TInterfacedObject, IWindow)  // TWindow implementiert IWindow
    procedure Draw;
    ⋮
   end;
   TFrameWindow = class(TWindow, IWindow)  // TFrameWindow implementiert IWindow neu
    procedure Draw;
    ⋮
   end;
```

Bei der Neuimplementation einer Schnittstelle wird die geerbte Implementation derselben Schnittstelle verborgen. Aus diesem Grund haben Methodenzuordnungsklauseln in einer untergeordneten Klasse keinerlei Auswirkung auf die neu implementierte Schnittstelle.

Schnittstellen delegieren

Die Direktive **implements** ermöglicht es, die Implementation einer Schnittstelle an eine Eigenschaft der implementierenden Klasse zu *delegieren*. Im folgenden Beispiel wird eine Eigenschaft namens *MyInterface* deklariert, die die Schnittstelle *IMyInterface* implementiert:

```
property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
```

Die Direktive **implements** muß der letzte Bezeichner in der Eigenschaftsdeklaration sein und kann mehrere Schnittstellen enthalten, die durch Kommas voneinander getrennt sind. Die »beauftragte« Eigenschaft muß folgende Bedingungen erfüllen:

- Sie muß vom Typ der Klasse oder Schnittstelle sein.
- Sie darf keine eine Array-Eigenschaft sein und keinen Index-Bezeichner verwenden.
- Sie muß über einen **read**-Bezeichner verfügen. Wenn es für die Eigenschaft eine **read**-Methode gibt, muß diese die standardmäßige Aufrufkonvention **register** verwenden. Außerdem darf die Methode nicht dynamisch, wohl aber virtuell sein.

Delegieren an eine Eigenschaft vom Typ einer Schnittstelle

Wenn die »beauftragte« Eigenschaft den Typ einer Schnittstelle hat, muß diese Schnittstelle bzw. die übergeordnete Schnittstelle in der Vorfahrenliste der Klasse enthalten sein, in der die Eigenschaft deklariert wird. Die beauftragte Eigenschaft muß ein Objekt zurückgeben, dessen Klasse die mit der Direktive **implements** angegebene Schnittstelle vollständig implementiert. Dabei dürfen keine Methodenzuordnungsklauseln verwendet werden. Ein Beispiel:

```
type
   IMyInterface = interface
    procedure P1;
    procedure P2;
   end;
   TMyClass = class(TObject, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
   end;
var
   MyClass: TMyClass;
   MyInterface: IMyInterface;
begin
   MyClass := TMyClass.Create;
   MyClass.FMyInterface := ...  // Ein Objekt, dessen Klasse IMyInterface implementiert
   MyInterface := MyClass;
   MyInterface.P1;
end;
```

Delegieren an eine Eigenschaft vom Typ einer Klasse

Wenn die »beauftragte« Eigenschaft den Typ einer Klasse hat, werden diese Klasse und ihre Vorfahren nach Methoden durchsucht, welche die angegebene Schnittstelle implementieren. Danach werden bei Bedarf die umgebende Klasse und ihre Vorfahren durchsucht. Es ist daher möglich, einige Methoden in der durch die Eigenschaft bezeichneten Klasse zu deklarieren, andere dagegen in der Klasse, in der die Eigenschaft deklariert ist. Methodenzuordnungsklauseln können wie üblich verwendet werden, um Mehrdeutigkeiten aufzulösen oder um eine bestimmte Methode anzuge-

ben. Eine Schnittstelle kann immer nur durch eine einzige Eigenschaft mit dem Typ einer Klasse implementiert werden. Dazu ein Beispiel:

```
type
   IMyInterface = interface
    procedure P1;
    procedure P2;
   end;
   TMyImplClass = class
    procedure P1;
    procedure P2;
   end;
   TMyClass = class(TInterfacedObject, IMyInterface)
    FMyImplClass: TMyImplClass;
    property MyImplClass: TMyImplClass read FMyImplClass implements IMyInterface;
    procedure IMyInterface.P1 = MyP1;
    procedure MyP1;
   end;
procedure TMyImplClass.P1;
   ⋮
procedure TMyImplClass.P2;
   ⋮
procedure TMyClass.MyP1;
   ⋮
var
   MyClass: TMyClass;
   MyInterface: IMyInterface;
begin
   MyClass := TMyClass.Create;
   MyClass.FMyImplClass := TMyImplClass.Create;
   MyInterface := MyClass;
   MyInterface.P1;        // Ruft TMyClass.MyP1 auf.
   MyInterface.P2;        // Ruft TImplClass.P2 auf.
end;
```

Schnittstellenreferenzen

Eine mit einem Schnittstellentyp deklarierte Variable kann Instanzen jeder Klasse referenzieren, die von der Schnittstelle implementiert wird. Mit Hilfe solcher Variablen können Schnittstellenmethoden auch aufgerufen werden, wenn zur Compilierzeit noch nicht bekannt ist, wo die Schnittstelle implementiert wird. Variablen, die Schnittstellen referenzieren, unterliegen folgenden Einschränkungen:

- Mit Ausdrücken vom Typ einer Schnittstelle kann nur auf Methoden und Eigenschaften zugegriffen werden, die in der Schnittstelle deklariert sind. Ein Zugriff auf andere Elemente der implementierenden Klasse ist nicht möglich.
- Ein Ausdruck vom Typ einer Schnittstelle kann nur dann ein Objekt referenzieren, dessen Klasse eine abgeleitete Schnittstelle implementiert, wenn die Klasse (bzw. eine übergeordnete Klasse) die abgeleitete Schnittstelle explizit implementiert.

Dazu ein Beispiel:

```
type
   IAncestor = interface
   end;
   IDescendant = interface(IAncestor)
    procedure P1;
   end;
   TSomething = class(TInterfacedObject, IDescendant)
    procedure P1;
    procedure P2;
   end;
   ⋮
var
   D: IDescendant;
   A: IAncestor;
begin
   D := TSomething.Create;  // Funktioniert!
   A := TSomething.Create;  // Fehler
   D.P1;  // Funktioniert!
   D.P2;  // Fehler
end;
```

- *A* ist in diesem Beispiel als Variable des Typs *IAncestor* deklariert. Da *IAncestor* nicht in den von *TSomething* implementierten Schnittstellen enthalten ist, kann *A* keine Instanz von *TSomething* zugewiesen werden. Um dieses Problem zu umgehen und eine gültige Zuweisung zu ermöglichen, muß *TSomething* folgendermaßen deklariert werden:

```
TSomething = class (TInterfacedObject, IAncestor, IDescendant)
   ⋮
```

- *D* ist als Variable des Typs *IDescendant* deklariert. Solange *D* eine Instanz von *TSomething* referenziert, kann damit nicht auf die Methode *P2* von *TSomething* zugegriffen werden (*P2* ist keine Methode von *IDescendant*). Um dieses Problem zu umgehen und einen gültigen Methodenaufruf zu erzeugen, muß *D* wie folgt deklariert werden:

```
D: TSomething;
```

Schnittstellenreferenzen werden über einen Referenzzähler verwaltet, der auf den von *IUnknown* geerbten Methoden *_AddRef* und *_Release* basiert. Ein Objekt, das ausschließlich durch Schnittstellen referenziert wird, braucht nicht manuell freigegeben zu werden. Seine Freigabe erfolgt automatisch, wenn der Referenzzähler den Wert Null erreicht.

Für globale Variablen vom Typ einer Schnittstelle ist nur der Initialisierungswert **nil** zulässig.

Um festzustellen, ob ein Ausdruck vom Typ einer Schnittstelle ein Objekt referenziert, übergeben Sie ihn an die Standardfunktion *Assigned*.

Zuweisungskompatibilität von Schnittstellen

Ein Klassentyp ist zu jedem Schnittstellentyp zuweisungskompatibel, der von der Klasse implementiert wird. Ein Schnittstellentyp ist zu jedem Schnittstellentyp kom-

patibel, von dem er abgeleitet ist. Jeder Variable vom Typ einer Schnittstelle kann der Wert **nil** zugewiesen werden.

Ein Ausdruck vom Typ einer Schnittstelle kann einer Variante zugewiesen werden. Wenn eine Schnittstelle vom Typ *IDispatch* oder ein Nachkomme von *IDispatch* ist, hat die resultierende Variante den Typencode *varDispatch*, andernfalls den Typencode *varUnknown*.

Eine Variante mit dem Typencode *varEmpty*, *varUnknown* oder *varDispatch* kann einer *IUnknown*-Variablen zugewiesen werden. Varianten mit dem Typencode *varEmpty* und *varDispatch* können an *IDispatch*-Variablen zugewiesen werden.

Schnittstellenumwandlungen

Für Schnittstellentypen gelten die gleichen Regeln wie bei Typumwandlungen von Variablen und Werten vom Typ einer Klasse. Der Wert eines Klassentyps kann mit einem Ausdruck der Form IMyInterface(SomeObject) in einen Schnittstellentyp umgewandelt werden. Voraussetzung dafür ist, daß die Schnittstelle von der Klasse implementiert wird.

Ein Ausdruck vom Typ einer Schnittstelle kann in den Typ Variant konvertiert werden. Wenn die Schnittstelle vom Typ *IDispatch* oder ein Nachkomme von *IDispatch* ist, hat die resultierende Variante den Typencode *varDispatch*, andernfalls den Typencode *varUnknown*.

Ein Variant-Wert mit dem Typencode *varEmpty*, *varUnknown* oder *varDispatch* kann in den Typ *IUnknown* umgewandelt werden. Varianten mit dem Typencode *varEmpty* oder *varDispatch* können in den Typ *IDispatch* konvertiert werden.

Schnittstellenabfragen

Mit Hilfe des Operators as können Schnittstellenumwandlungen durchgeführt werden. Dieser Vorgang wird als *Schnittstellenabfrage* bezeichnet. Eine Schnittstellenabfrage generiert auf der Basis des (zur Laufzeit vorliegenden) Objekttyps aus einer Objektreferenz oder einer anderen Schnittstellenreferenz einen Ausdruck vom Typ einer Schnittstelle. Die Syntax für eine Schnittstellenabfrage lautet

```
Objekt as Schnittstelle
```

Objekt ist entweder ein Ausdruck vom Typ einer Schnittstelle oder Variante, oder er bezeichnet eine Instanz einer Klasse, die eine Schnittstelle implementiert. *Schnittstelle* kann jede mit einer GUID deklarierte Schnittstelle sein.

Wenn *Objekt* den Wert **nil** hat, liefert die Schnittstellenabfrage als Ergebnis **nil**. Andernfalls wird die GUID der Schnittstelle an die Methode *QueryInterface* von Objekt übergeben. Wenn *QueryInterface* einen Wert ungleich Null zurückgibt, wird eine Exception ausgelöst. Ist der Rückgabewert dagegen Null (d.h. die *Schnittstelle* ist in der Klasse von *Objekt* implementiert), ergibt die Schnittstellenabfrage eine Referenz auf *Objekt*.

Automatisierungsobjekte

Ein Objekt, dessen Klasse die in der Unit *System* deklarierte *IDispatch*-Schnittstelle implementiert, ist ein Automatisierungsobjekt.

Dispatch-Schnittstellen

Über Dispatch-Schnittstellen werden die Methoden und Eigenschaften definiert, die ein Automatisierungsobjekt mit Hilfe der *IDispatch*-Schnittstelle implementiert. Aufrufe der Methoden einer Dispatch-Schnittstelle werden zur Laufzeit über die Methode *Invoke* von *IDispatch* weitergeleitet. Eine Dispatch-Schnittstelle kann nicht von einer Klasse implementiert werden.

Die Syntax zur Deklaration einer Dispatch-Schnittstelle lautet

```
type Schnittstellenname = dispinterface
   ['{GUID}']
   Elementliste
end;
```

Die Angabe von ['{GUID}'] optional. *Elementliste* besteht aus Eigenschafts- und Methodendeklarationen. Eine Dispatch-Schnittstellendeklaration hat große Ähnlichkeit mit der Deklaration einer normalen Schnittstelle, mit dem Unterschied, daß kein Vorfahr angegeben werden kann. Das folgende Codefragment zeigt ein Beispiel für die Deklaration einer Dispatch-Schnittstelle:

```
type
   IStringsDisp = dispinterface
    ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
    property ControlDefault[Index: Integer]: OleVariant dispid 0; default;
    function Count: Integer; dispid 1;
    property Item[Index: Integer]: OleVariant dispid 2;
    procedure Remove(Index: Integer); dispid 3;
    procedure Clear; dispid 4;
    function Add(Item: OleVariant): Integer; dispid 5;
    function _NewEnum: IUnknown; dispid -4;
   end;
```

Methoden für Dispatch-Schnittstellen

Die Methoden für eine Dispatch-Schnittstelle stellen Prototypen für Aufrufe der Methode *Invoke* dar, die zur zugrundeliegenden Implementation der IDispatch-Schnittstelle gehört. Um für eine Methode eine Dispatch-ID für die Automatisierung festzulegen, nehmen Sie in die Methodendeklaration die Direktive **dispid** und eine Integer-Konstante auf. Die Angabe einer bereits verwendeten Dispatch-ID führt zu einem Fehler.

Eine Methode, die in einer Dispatch-Schnittstelle deklariert wird, darf neben **dispid** keine weiteren Direktiven enthalten. Die Typen aller Funktionsergebnisse und Parameter müssen kompatibel zur Automatisierung sein. Folgende Typen sind zulässig: *Byte, Currency, Real, Double, Longint, Integer, Single, Smallint, AnsiString, WideString, TDateTime, Variant, OleVariant, WordBool* und alle Schnittstellentypen.

Eigenschaften für Dispatch-Schnittstellen

Die Eigenschaften einer Dispatch-Schnittstelle enthalten keine Zugriffsattribute. Sie können entweder mit der Direktive **readonly** oder mit der Direktive **writeonly** deklariert werden. Zu Festlegung einer Dispatch-ID für eine Eigenschaft nehmen Sie die Direktive **dispid** in die Eigenschaftsdeklaration auf. Auf **dispid** muß eine Integer-Konstante folgen. Die Angabe einer Dispatch-ID, die bereits verwendet wird, führt zu einem Fehler. Mit Hilfe der Direktive **default** können Sie eine Array-Eigenschaft als Standardeigenschaft für die Schnittstelle festlegen. Eigenschaftsdeklarationen für Dispatch-Schnittstellen dürfen keine anderen als die genannten Direktiven enthalten.

Zugriff auf Automatisierungsobjekte

Der Zugriff auf Automatisierungsobjekte erfolgt über Varianten. Wenn ein Automatisierungsobjekt über eine Variante referenziert wird, können die Eigenschaften des Objekts durch einen Aufruf der entsprechenden Objektmethoden über die Variante gelesen und geändert werden. Zu diesem Zweck muß *ComObj* in die **uses**-Klausel einer der Units bzw. des Programms oder der Bibliothek aufgenommen werden.

Da Aufrufe von Methoden des Automatisierungsobjekts zur Laufzeit gebunden werden, sind keine Methodendeklarationen erforderlich. Die Gültigkeit solcher Aufrufe wird vom Compiler nicht überprüft.

Das folgende Beispiel demonstriert Aufrufe von Automatisierungsmethoden. Die (in *ComObj* definierte) Funktion *CreateOleObject* gibt eine *IDispatch*-Referenz auf ein Automatisierungsobjekt zurück und ist zuweisungskompatibel zur Variante *Word*.

```
var
   Word: Variant;
begin
   Word := CreateOleObject('Word.Basic');
   Word.FileNew('Normal');
   Word.Insert('Erste Zeile'#13);
   Word.Insert('Zweite Zeile'#13);
   Word.FileSaveAs('c:\temp\test.txt', 3);
end;
```

Parameter vom Typ einer Schnittstelle können an Automatisierungsmethoden übergeben werden.

Die Übergabe binärer Daten zwischen Automatisierungs-Controllern und Automatisierungs-Servern erfolgt bevorzugt über variante Arrays mit dem Elementtyp *varByte*. Da die Daten derartiger Arrays nicht übersetzt werden, können Sie mit den Routinen *VarArrayLock* und *VarArrayUnlock* effizient darauf zugreifen.

Syntax für Aufrufe von Automatisierungsmethoden

Aufrufe von Methoden eines Automatisierungsobjekts und Zugriffe auf die Eigenschaften des Objekts folgen einer ähnlichen Syntax wie normale Methodenaufrufe und Eigenschaftszugriffe. Beim Aufruf einer Automatisierungsmethode können aber sowohl Positionsparameter als auch benannte Parameter verwendet werden. (Be-

nannte Parameter werden allerdings nicht von allen Automatisierungs-Servern unterstützt.)

Während ein Positionsparameter aus einem Ausdruck besteht, setzt sich ein benannter Parameter aus einem Parameterbezeichner, dem Symbol **:=** und einem Ausdruck zusammen. In einem Methodenaufruf müssen Positionsparameter immer vor benannten Parametern angegeben werden. Die Reihenfolge der benannten Parameter ist beliebig.

Bei bestimmten Automatisierungs-Servern können in Methodenaufrufen Parameter weggelassen werden. In diesem Fall werden Standardwerte verwendet. Dazu ein Beispiel:

```
Word.FileSaveAs('test.doc');
Word.FileSaveAs('test.doc', 6);
Word.FileSaveAs('test.doc',,,'secret');
Word.FileSaveAs('test.doc', Password := 'secret');
Word.FileSaveAs(Password := 'secret', Name := 'test.doc');
```

Sie können Automatisierungsmethoden neben Parametern vom Typ Integer und String auch reelle, Boolesche und variante Typen übergeben. Parameterausdrücke, die nur aus einer Variablenreferenz bestehen, werden als Referenz übergeben. Dies gilt allerdings nur, wenn die Variablenreferenz einen der folgenden Typen hat: *Byte, Smallint, Integer, Single, Double, Currency, TDateTime, AnsiString, WordBool oder Variant*. Hat der Ausdruck einen anderen Typ oder handelt es sich nicht um eine Variable, wird der Parameter als Wert übergeben. Wird ein Parameter als Referenz an eine Methode übergeben, die einen Wert-Parameter erwartet, ermittelt COM den Wert aus der Referenz. Dagegen führt die Übergabe eines Wert-Parameters an eine Methode, die eine Übergabe als Referenz erwartet, zu einem Fehler.

Duale Schnittstellen

Eine duale Schnittstelle ist eine Schnittstelle, die sowohl die Bindung zur Compilierzeit als auch die Laufzeitbindung mittels Automatisierung unterstützt. Eine duale Schnittstelle muß ein Nachkomme von *IDispatch* sein.

Alle Methoden einer dualen Schnittstelle (mit Ausnahme der von *IUnknown* und *IDispatch* geerbten Methoden) müssen die Aufrufkonvention **safecall** verwenden. Alle Methodenparameter und Ergebnistypen müssen den Anforderungen der Automatisierung entsprechen. Folgende Typen können für die Automatisierung verwendet werden: *Byte, Currency, Real, Double, Real48, Integer, Single, Smallint, AnsiString, ShortString, TDateTime, Variant, OleVariant*, und *WordBool*.

Kapitel

11

Speicherverwaltung

Dieses Kapitel beschäftigt sich mit der Speicherverwaltung. Es beschreibt, wie Programme den Speicher verwenden, und erläutert die internen Datenformate der Datentypen von Object Pascal.

Der Speichermanager von Delphi

Der Speichermanager ist für alle Operationen zuständig, mit denen eine Delphi-Anwendung dynamisch Speicher zuweist oder freigibt. Er wird von den Standardprozeduren *New*, *Dispose*, *GetMem*, *ReallocMem* und *FreeMem* und bei der Zuweisung von Speicher an alle Objekte und lange Strings verwendet.

Der Speichermanager von Delphi ist speziell auf Anwendungen zugeschnitten, die sehr viele Blöcke kleiner bis mittlerer Größe belegen. Dies ist typisch für objektorientierte Anwendungen und für Anwendungen, die String-Daten verarbeiten. Andere Speichermanager wie die Implementationen von *GlobalAlloc* und *LocalAlloc* und die Unterstützung des privaten Heap in Windows sind in solchen Situationen weniger geeignet und verlangsamen eine Anwendung spürbar, wenn sie direkt verwendet werden.

Um die bestmögliche Leistung zu erzielen, arbeitet der Speichermanager direkt mit der Win32-API für virtuellen Speicher zusammen (über die Funktionen *VirtualAlloc* und *VirtualFree*). Er fordert vom Betriebssystem Speicher in 1-MB-Blöcken an und weist ihn bei Bedarf in 16-KB-Blöcken zu. Umgekehrt gibt er nicht verwendeten Speicher in 16-KB- und 1-MB-Blöcken frei. Wenn kleinere Blöcke benötigt werden, wird der zugewiesene Speicher weiter unterteilt.

Die Blöcke des Speichermanagers belegen immer ein Vielfaches von vier Byte und enthalten immer einen vier Byte großen Header, in dem die Größe des Blocks und weitere Statusbits gespeichert sind. Die Blöcke sind daher immer an Double-Word-Grenzen ausgerichtet, was eine optimale Geschwindigkeit bei ihrer Adressierung gewährleistet.

Der Speichermanager verwaltet die beiden Statusvariablen *AllocMemCount* und *AllocMemSize*, die die Anzahl der gegenwärtig zugewiesenen Speicherblöcke und ihre Gesamtgröße enthalten. Diese Variablen können in Anwendungen als Statusinformationen beim Debuggen angezeigt werden.

Die Unit System stellt die zwei Prozeduren *GetMemoryManager* und *SetMemoryManager* zur Verfügung, mit denen Sie in einer Anwendung die Low-Level-Aufrufe des Speichermanagers abfangen können. In der gleichen Unit ist die Funktion *GetHeapStatus* definiert, die einen Record mit detaillierten Statusinformationen über den Speichermanager zurückgibt. Weitere Informationen über diese Routinen finden Sie in der Online-Hilfe.

Variablen

Globale Variablen werden im Datensegment der Anwendung zugewiesen und bleiben bis zur Beendigung des Programms erhalten. Lokale Variablen, die innerhalb von Prozeduren und Funktionen deklariert sind, werden auf dem Stack der Anwendung abgelegt. Wenn eine Prozedur oder Funktion aufgerufen wird, reserviert sie auf dem Stack Speicherplatz für ihre lokalen Variablen. Bei der Beendigung der Prozedur oder Funktion werden die lokalen Variablen wieder freigegeben. Variablen können aber aufgrund von Optimierungsaktionen des Compilers auch zu einem früheren Zeitpunkt freigegeben werden.

Der Stack einer Anwendung wird durch eine Mindestgröße und eine Maximalgröße festgelegt. Diese Werte werden über die Compiler-Direktiven **$MINSTACKSIZE** und **$MAXSTACKSIZE** gesteuert. Die Voreinstellung lautet 16.384 (16 KB) bzw. 1.048.576 (1 MB). Eine Anwendung hat an Stack nie weniger als die Mindestgröße und nie mehr als die Maximalgröße zur Verfügung. Wenn beim Start einer Anwendung weniger Speicher zur Verfügung steht, als es der Wert für die Mindestgröße des Stack vorschreibt, gibt Windows eine entsprechende Fehlermeldung aus.

Wenn eine Anwendung mehr Stack benötigt als die angegebene Mindestgröße, wird ihr in Blöcken von vier KB automatisch weiterer Speicher zugewiesen. Schlägt die Zuweisung fehl, weil nicht mehr Speicher vorhanden ist oder die Maximalgröße des Stack überschritten würde, wird eine *EStackOverflow-Exception* ausgelöst. (Die Stack-Überlaufprüfung wird automatisch durchgeführt. Die Compiler-Direktive **$S** zum Ein- und Ausschalten dieser Prüfung wurde aber aus Gründen der Abwärtskompatibilität beibehalten.)

Der Speicher für dynamische Variablen, die Sie mit den Prozeduren *GetMem* oder *New* erzeugen, wird auf dem Heap reserviert. Die Variablen bleiben bis zu einem entsprechenden *FreeMem*- bzw. *Dispose*-Aufruf erhalten.

Lange Strings, Wide-Strings, dynamische Arrays, variants, und interfaces werden auf dem Heap zugewiesen. Ihr Speicher wird aber dennoch automatisch verwaltet.

Interne Datenformate

Die folgenden Abschnitte erläutern die internen Formate der Datentypen von Object Pascal.

Integer-Typen

Das Format, in dem eine Variable eines Integer-Typs dargestellt wird, hängt von ihren Bereichsgrenzen ab.

- Liegen beide Grenzen im Bereich –128..127 (*Shortint*), wird die Variable als Byte mit Vorzeichen gespeichert.
- Liegen beide Grenzen im Bereich 0..255 (*Byte*), wird die Variable als Byte ohne Vorzeichen gespeichert.
- Liegen beide Grenzen im Bereich –32768..32767 (*Smallint*), wird die Variable als Word mit Vorzeichen gespeichert.
- Liegen beide Grenzen im Bereich 0..65535 (*Word*), wird die Variable als Word ohne Vorzeichen gespeichert.
- Liegen beide Grenzen im Bereich –2147483648..2147483647 (*Longint*), wird die Variable als Double Word mit Vorzeichen gespeichert.
- Liegen beide Grenzen im Bereich 0..4294967295 (*Longword*), wird die Variable als Double Word ohne Vorzeichen gespeichert.
- In allen anderen Fällen wird die Variable als Vierfach-Word mit Vorzeichen (*Int64*) gespeichert.

Zeichentypen

Ein Zeichen vom Typ *Char, AnsiChar* oder einem Teilbereich von *Char* wird als Byte ohne Vorzeichen gespeichert. Eine Variable vom *Typ WideChar* wird als Word ohne Vorzeichen gespeichert.

Boolesche Typen

Eine Variable vom Typ *Boolean* oder *ByteBool* wird als Byte, eine Variable vom Typ *WordBool* als Word und eine Variable vom Typ *LongBool* als Double Word gespeichert.

Eine Variable vom Typ *Boolean* kann die Werte 0 *(False)* und 1 *(True)* annehmen. Variablen vom Typ ByteBool, WordBool und LongBool können die Werte 0 *(False)* und ungleich 0 *(True)* annehmen.

Aufzählungstypen

Eine Variable eines Aufzählungstyps wird als Byte ohne Vorzeichen gespeichert, wenn die Aufzählung nicht mehr als 256 Werte umfaßt und der Typ im **Status {$Z1}** (Voreinstellung) deklariert wurde. Enthält der Aufzählungstyp mehr als 256 Werte oder wurde er im Status **{$Z2}** deklariert, wird die Variable als Word ohne Vorzeichen gespeichert. Wird ein Aufzählungstyp im Status **{$Z4}** deklariert, wird die Variable als Double Word ohne Vorzeichen gespeichert.

Reelle Typen

Reelle Typen speichern die binäre Entsprechung des Vorzeichens (**+** oder **–**), eines *Exponenten* und einer *Mantisse*. Ein reeller Wert wird in folgender Form dargestellt:

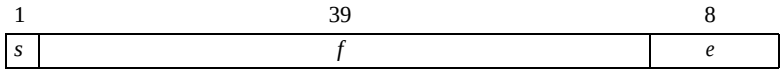
$$+/- \ Mantisse * 2^{Exponent}$$

Dabei wird für die *Mantisse* links des binären Dezimalpunkts ein einziges Bit verwendet (d.h. 0 <= *Mantisse* < 2).

In den folgenden Abbildungen befindet sich das höchstwertige Bit immer auf der linken Seite, das niedrigstwertige Bit immer auf der rechten Seite. Die Zahlen am oberen Rand geben die Breite jedes Feldes in Bit an. Die Elemente ganz links werden an den höchsten Adressen gespeichert. Bei einem Real48-Wert wird beispielsweise *e* im ersten Byte, *f* in den nächsten fünf Bytes und *s* im höchstwertigen Bit des letzten Byte gespeichert.

Der Typ Real48

Eine Real48-Zahl mit sechs Bytes (48 Bit) wird in drei Felder unterteilt:



Wenn gilt 0 < *e* <= 255, ergibt sich der Wert *v* der Zahl folgendermaßen:

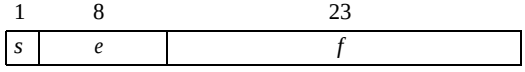
$v = (-1)^s * 2^{(e-129)} * (1.f)$

Wenn *e* = 0, ist *v* = 0.

Der Typ *Real48* eignet sich nicht zum Speichern von NaN-Werten (Not a Number = keine Zahl) und unendlichen Werten. Das Speichern von NaN-Werten und unendlichen Werten in Real48-Variablen führt zu einem Überlauffehler.

Der Typ Single

Eine *Single*-Zahl mit vier Byte (32 Bit) wird in drei Felder unterteilt:



Der Wert *v* der Zahl ergibt sich folgendermaßen:

Wenn $0 < e < 255$, ist $v = (-1)^s * 2^{(e-127)} * (1.f)$

Wenn $e = 0$ und $f <> 0$, ist $v = (-1)^s * 2^{(-126)} * (0.f)$

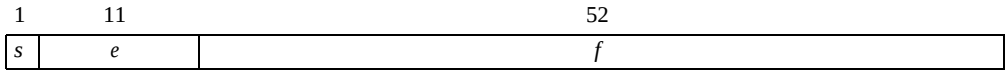
Wenn $e = 0$ und $f = 0$, ist $v = (-1)^s * 0$

Wenn $e = 255$ und $f = 0$, ist $v = (-1)^s *$ Inf

Wenn $e = 255$ und $f <> 0$, ist v ein NaN-Wert

Der Typ Double

Eine *Double*-Zahl mit acht Bytes (64 Bit) wird in drei Felder unterteilt:



Der Wert *v* der Zahl ergibt sich folgendermaßen:

Wenn $0 < e < 2047$, ist $v = (-1)^s * 2^{(e-1023)} * (1.f)$

Wenn $e = 0$ und $f <> 0$, ist $v = (-1)^s * 2^{(-1022)} * (0.f)$

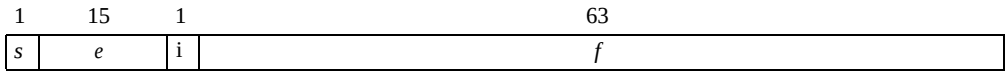
Wenn $e = 0$ und $f = 0$, ist $v = (-1)^s * 0$

Wenn $e = 2047$ und $f = 0$, ist $v = (-1)^s *$ Inf

Wenn $e = 2047$ und $f <> 0$, ist v ein NaN-Wert

Der Typ Extended

Eine *Extended*-Zahl mit zehn Bytes (80 Bit) wird in vier Felder unterteilt.



Der Wert v der Zahl ergibt sich folgendermaßen:

Wenn $0 <= e < 32767$, ist $v = (-1)^s * 2^{(e-16383)} * (i.f)$

Wenn $e = 32767$ und $f = 0$, ist $v = (-1)^s *$ Inf

Wenn $e = 32767$ und $f <> 0$, ist v ein NaN-Wert

Der Typ Comp

Eine *Comp*-Zahl mit acht Bytes (64 Bit) wird als 64-Bit-Integer mit Vorzeichen gespeichert.

Der Typ Currency

Eine *Currency*-Zahl mit acht Bytes (64 Bit) wird als skalierter 64-Bit-Integer mit Vorzeichen gespeichert. Dabei stehen die vier niedrigstwertigen Ziffern implizit für vier Dezimalstellen.

Zeigertypen

Eine Zeigervariable wird als 32-Bit-Adresse gespeichert und belegt vier Byte. Der Zeigerwert **nil** wird als Null gespeichert.

Kurze String-Typen

Die Anzahl der Bytes eines Strings ergibt sich aus seiner maximalen Länge + 1. Das erste Byte enthält die aktuelle dynamische Länge des Strings, die folgenden Bytes seine Zeichen.

Das Längenbyte und die Zeichen werden als Werte ohne Vorzeichen betrachtet. Die maximale Länge eines Strings beträgt 255 Zeichen plus ein Längenbyte (**string**[255]).

Lange String-Typen

Eine lange String-Variable belegt vier Byte, die einen Zeiger auf einen dynamisch zugewiesenen String enthalten. Wenn eine lange String-Variable leer ist (also einen String der Länge 0 enthält), ist der String-Zeiger **nil**, und der Variable wird kein dynamischer Speicher zugewiesen. Andernfalls referenziert der String-Zeiger einen dynamisch zugewiesenen Speicherblock, der neben dem String-Wert eine Längenangabe und einen Referenzzähler von je 32 Bit enthält. Die nachstehende Tabelle zeigt den Aufbau eines Speicherblocks für einen langen String.

Tabelle 11.1 Aufbau des Speichers für einen langen String

Offset	Inhalt
–8	32-Bit-Referenzzähler
–4	32-Bit-Längenangabe
0..Länge – 1	Zeichen-String
Länge	NULL-Zeichen

Das NULL-Zeichen am Ende des Speicherblocks eines langen Strings wird vom Compiler und den integrierten String-Routinen automatisch verwaltet. Durch das NULL-Zeichen kann ein langer String direkt in einen nullterminierten String umgewandelt werden.

Für String-Konstanten und Literale erzeugt der Compiler einen Speicherblock mit demselben Aufbau wie bei einem dynamisch zugewiesenen String. Der Referenzzähler ist jedoch –1. Wird einer langen String-Variable eine String-Konstante zugewiesen, wird der String-Zeiger mit der Adresse des Speicherblocks der String-Konstante be-

legt. Die integrierten String-Routinen ändern keine Blöcke mit einem Referenzzähler von –1.

Wide-String-Typen

Eine Wide-String-Variable belegt vier Byte, die einen Zeiger auf einen dynamisch zugewiesenen String enthalten. Wenn eine Wide-String-Variable leer ist (also einen String der Länge 0 enthält), ist der String-Zeiger **nil**, und der Variablen wird kein dynamischer Speicher zugewiesen. Andernfalls referenziert der String-Zeiger einen dynamisch zugewiesenen Speicherblock, der neben dem String-Wert eine Längenangabe von 32 Bit enthält. Die folgende Tabelle zeigt den Aufbau eines Speicherblocks für einen Wide-String.

Tabelle 11.2 Aufbau des dynamischen Speichers für einen Wide-String

Offset	Inhalt
–4	32-Bit-Längenangabe in Byte
0..Länge – 1	Zeichen-String
Länge	NULL-Zeichen

Die Länge des Strings ergibt sich aus der Anzahl der Bytes und ist damit doppelt so groß wie die Anzahl der Wide-Zeichen, die er enthält.

Das NULL-Zeichen am Ende des Speicherblocks eines Wide-Strings wird vom Compiler und den integrierten String-Routinen automatisch verwaltet. Durch das NULL-Zeichen kann ein Wide-String direkt in einen nullterminierten String umgewandelt werden.

Mengentypen

Eine Menge ist ein Array von Bits. Jedes Bit zeigt an, ob ein Element in der Menge enthalten ist oder nicht. Da die maximale Anzahl der Elemente einer Menge 256 beträgt, belegt eine Menge nie mehr als 32 Bytes. Die Anzahl der Bytes, die eine bestimmte Menge belegt, ergibt sich wie folgt:

(Max div 8) – (*Min* div 8) + 1

Max und *Min* sind die obere und die untere Grenze des Basistyps der Menge. Die Position des Byte eines speziellen Elements *E* ergibt sich wie folgt:

(*E* div 8) – (*Min* div 8)

Die Position des Bit in diesem Byte ergibt sich aus

E mod 8

Dabei ist *E* der ordinale Wert des Elements.

Statische Array-Typen

Ein statisches Array wird als fortlaufende Folge von Variablen des Komponententyps des Arrays gespeichert. Die Komponenten mit den niedrigsten Indizes werden an der niedrigsten Speicheradresse abgelegt. Bei einem mehrdimensionalen Array liegt die äußerste rechte Dimension an der Basis des belegten Speicherblocks.

Dynamische Array-Typen

Eine Variable für ein dynamisches Array belegt vier Byte, die einen Zeiger auf ein dynamisch zugewiesenes Array enthalten. Wenn eine Variable leer ist (also nicht initialisiert), ist der Zeiger **nil**, und der Variablen wird kein dynamischer Speicher zugewiesen. Andernfalls referenziert die Variable einen dynamisch zugewiesenen Speicherblock, der neben dem Array eine Längenangabe und einen Referenzzähler von je 32 Bit enthält. Die folgende Tabelle zeigt den Aufbau eines Speicherblocks für ein dynamisches Array.

Tabelle 11.3 Aufbau des Speichers für ein dynamisches Array

Offset	Inhalt
–8	32-Bit-Referenzzähler
–4	32-Bit-Längenangabe (Anzahl der Elemente)
0..Länge * (Elementgröße) – 1	Array-Elemente

Record-Typen

Wenn ein Record-Typ mit dem voreingestellten Status **{$A+}** deklariert wird und die Deklaration nicht den Modifizierer packed enthält, handelt es sich um einen *ungepackten* Record-Typ. Die Felder des Records werden so ausgerichtet, daß die CPU möglichst effizient darauf zugreifen kann. Die Ausrichtung hängt von den Typen der einzelnen Felder ab. Jeder Datentyp besitzt eine implizite Ausrichtungsmaske, die vom Compiler automatisch berechnet wird. Sie kann die Werte 1, 2, 4 oder 8 haben und entspricht dem Byte-Raster, in dem ein Wert dieses Typs für den optimalen Zugriff im Speicher angeordnet werden muß. Die folgende Tabelle enthält die Ausrichtungsmasken für alle Datentypen.

Tabelle 11.4 Ausrichtungsmasken für Typen

Typ	Ausrichtungsmaske
Ordinaltypen	Größe des Typs (1, 2, 4 oder 8)
Reelle Typen	2 für *Real* und *Extended*, 4 für alle anderen reellen Typen
Kurze Strings	1
Array-Typen	Wie der Typ der Array-Elemente
Record-Typen	Die größte Ausrichtungsmaske der Record-Felder
Mengentypen	Größe des Typs bei 1, 2 oder 4, andernfalls 1
Alle anderen Typen	4

Um die korrekte Ausrichtung der Felder in einem ungepackten Record-Typ zu gewährleisten, fügt der Compiler vor den Feldern mit der Ausrichtungsmaske 2 ein leeres Byte ein. Bei Feldern mit der Ausrichtungsmaske 4 werden nach Bedarf bis zu drei leere Bytes eingefügt. Schließlich erweitert der Compiler die gesamte Größe des Records bis zu der Byte-Grenze, die sich aus der größten Ausrichtungsmaske der enthaltenen Felder ergibt.

Wenn ein Record-Typ mit dem Status **{$A–}** deklariert wird oder die Deklaration den Modifizierer **packed** enthält, werden die Felder des Records nicht ausgerichtet, sondern einfach an aufeinanderfolgenden Offsets abgelegt. Die Gesamtgröße eines solchen gepackten Records ergibt sich aus der Größe aller Felder.

Dateitypen

Dateitypen werden im Speicher als Records dargestellt. Für jede typisierte und untypisierte Datei wird ein Record mit 332 Bytes angelegt, die sich wie folgt verteilen:

```
type
   TFileRec = record
    Handle: Integer;
    Mode: Integer;
    RecSize: Cardinal;
    Private: array[1..28] of Byte;
    UserData: array[1..32] of Byte;
    Name: array[0..259] of Char;
   end;
```

Der Record für Textdateien umfaßt 460 Bytes, die sich wie folgt verteilen:

```
type
   TTextBuf = array[0..127] of Char;
   TTextRec = record
    Handle: Integer;
    Mode: Integer;
    BufSize: Cardinal;
    BufPos: Cardinal;
    BufEnd: Cardinal;
    BufPtr: PChar;
    OpenFunc: Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;
    UserData: array[1..32] of Byte;
    Name: array[0..259] of Char;
    Buffer: TTextBuf;
   end;
```

Handle enthält das Handle der Datei (wenn die Datei geöffnet ist).

Das Feld *Mode* kann einen der folgenden Werte annehmen:

```
const
   fmClosed = $D7B0;
   fmInput  = $D7B1;
   fmOutput = $D7B2;
   fmInOut  = $D7B3;
```

Dabei zeigt *fmClosed* an, daß die Datei geschlossen ist. *fmInput* und *fmOutput* zeigen an, daß es sich bei der Datei um eine Textdatei handelt, die zurückgesetzt *(fmInput)* oder neu geschrieben *(fmOutput)* wurde. *fmInOut* zeigt an, daß die Dateivariable eine typisierte oder eine untypisierte Datei ist, die zurückgesetzt oder neu geschrieben wurde. Jeder andere Wert zeigt an, daß die Dateivariable nicht zugeordnet und damit nicht initialisiert wurde.

Das Feld *UserData* wird für benutzerdefinierte Routinen freigehalten, um Daten zu speichern.

Name enthält den Dateinamen. Dieser besteht aus einer Folge von Zeichen, die mit NULL (#0) abgeschlossen ist.

Bei typisierten und untypisierten Dateien enthält *RecSize* die Record-Länge in Byte. Das Feld *Private* ist reserviert und wird nicht verwendet.

Bei Textdateien ist *BufPtr* ein Zeiger auf einen Puffer mit *BufSize* Bytes, *BufPos* ist der Index des nächsten zu lesenden oder zu schreibenden Zeichens des Puffers. *BufEnd* entspricht der Anzahl der gültigen Zeichen im Puffer. *OpenFunc, InOutFunc, FlushFunc* und *CloseFunc* sind Zeiger auf die E/A-Routinen, welche die Datei verwalten. Weitere Informationen zu diesem Thema finden Sie unter »Gerätetreiberfunktionen« auf Seite 8-5.

Prozedurale Typen

Ein Prozedurzeiger wird als 32-Bit-Zeiger auf den Eintrittspunkt einer Prozedur oder Funktion gespeichert. Ein Methodenzeiger wird als 32-Bit-Zeiger auf den Eintrittspunkt einer Methode gespeichert, dem ein 32-Bit-Zeiger auf ein Objekt folgt.

Klassentypen

Der Wert eines Klassentyps wird als 32-Bit-Zeiger auf eine Instanz der Klasse gespeichert. Die Instanz einer Klasse wird auch als *Objekt* bezeichnet. Das interne Datenformat eines Objekts gleicht dem eines Records. Die Felder eines Objekts werden in der Reihenfolge ihrer Deklaration als fortlaufende Folge von Variablen gespeichert. Die Felder werden wie bei einem ungepackten Record-Typ immer ausgerichtet. Alle von einer übergeordneten Klasse geerbten Felder werden vor den neuen Feldern gespeichert, die in der abgeleiteten Klasse definiert wurden.

Das erste 4-Byte-Feld eines jeden Objekts ist ein Zeiger auf die *Tabelle der virtuellen Methoden* (VMT) der Klasse. Es gibt nur eine VMT pro Klasse (und nicht eine für jedes Objekt). Zwei verschiedene Klassentypen können eine VMT jedoch nicht gemeinsam benutzen. VMTs werden automatisch vom Compiler erstellt und nie direkt von einem Programm bearbeitet. Ebenso werden die Zeiger auf VMTs automatisch von den Konstruktormethoden in den erstellten Objekten gespeichert und nie direkt von einem Programm bearbeitet.

Die folgende Tabelle zeigt die Struktur einer VMT. Bei positiven Offsets besteht eine VMT aus einer Liste mit 32-Bit-Methodenzeigern. Für jede benutzerdefinierte virtuelle Methode des Klassentyps ist ein Zeiger vorhanden. Die Zeiger sind in der Reihenfolge der Deklaration angeordnet. Jeder Eintrag enthält die Adresse des Eintritts-

punktes der entsprechenden virtuellen Methode. Dieses Layout ist zur V-Tabelle von C++ und zu COM kompatibel. Bei negativen Offsets enthält eine VMT die Anzahl der Felder, die in Object Pascal intern implementiert sind. In einer Anwendung sollten diese Informationen mit den Methoden von *TObject* abgerufen werden, da sich dieses Layout bei zukünftigen Implementierungen von Object Pascal ändern kann.

Tabelle 11.5 Struktur der virtuellen Methodentabelle

Offset	Typ	Beschreibung
–64	*Pointer*	Zeiger auf virtuelle Methodentabelle (oder **nil**)
–60	*Pointer*	Zeiger auf Schnittstellentabelle (oder **nil**)
–56	*Pointer*	Zeiger auf Informationstabelle zur Automatisierung (oder **nil**)
–52	*Pointer*	Zeiger auf Instanzen-Initialisierungstabelle (oder **nil**)
–48	*Pointer*	Zeiger auf Informationstabelle des Typs (oder **nil**)
–44	*Pointer*	Zeiger auf die Tabelle der Felddefinitionen (oder **nil**)
–40	*Pointer*	Zeiger auf die Tabelle der Methodendefinitionen (oder **nil**)
–36	*Pointer*	Zeiger auf die Tabelle der dynamischen Methoden (oder **nil**)
–32	*Pointer*	Zeiger auf einen kurzen String, der den Klassennamen enthält
–28	*Cardinal*	Instanzgröße in Byte
–24	*Pointer*	Zeiger auf einen Zeiger auf die übergeordnete Klasse (oder **nil**)
–20	*Pointer*	Zeiger auf den Eintrittspunkt der Methode ***SafecallException*** (oder **nil**)
–16	*Pointer*	Eintrittspunkt der Methode *DefaultHandler*
–12	*Pointer*	Eintrittspunkt der Methode *NewInstance*
–8	*Pointer*	Eintrittspunkt der Methode *FreeInstance*
–4	*Pointer*	Eintrittspunkt des Destruktors *Destroy*
0	*Pointer*	Eintrittspunkt der ersten benutzerdefinierten virtuellen Methode
4	*Pointer*	Eintrittspunkt der zweiten benutzerdefinierten virtuellen Methode
⋮	⋮	⋮

Klassenreferenztypen

Der Wert einer Klassenreferenz wird als 32-Bit-Zeiger auf die virtuelle Methodentabelle (VMT) einer Klasse gespeichert.

Variante Typen

Eine Variante wird als 16-Byte-Record gespeichert, der einen Typencode und einen Wert (oder eine Referenz auf einen Wert) des durch den Code bezeichneten Typs enthält. Die Unit *System* definiert Konstanten und Typen für Varianten.

Der Typ *TVarData* steht für die interne Struktur einer Variante. Die interne Struktur entspricht dem Typ Variant, der von COM und der *Win32-API* verwendet wird. Mit dem Typ *TVarData* kann bei einer Typumwandlung von Varianten auf die interne Struktur einer Variablen zugegriffen werden.

Das Feld *VType* eines *TVarData*-Records enthält den Typencode der Variante in den niederwertigen zwölf Bits (die Bits, die von der Konstante *varTypeMask* definiert werden). Zusätzlich zeigt das Bit *varArray* an, ob es sich bei der Variante um ein Array handelt. Das Bit *varByRef* gibt an, ob die Variante eine Referenz oder einen Wert enthält.

Die Felder *Reserved1, Reserved2* und *Reserved3* eines *TVarData*-Records werden nicht verwendet.

Der Inhalt der restlichen acht Bytes eines *TVarData*-Records hängt vom Feld *VType* ab. Wenn keines der Bits *varArray* und *varByRef* gesetzt ist, enthält die Variante einen Wert des gegebenen Typs.

Wenn *varArray* gesetzt ist, enthält die Variante einen Zeiger auf eine *TVarArray*-Struktur, die das Array definiert. Der Typ eines jeden Array-Elements ist durch die Bits *varTypeMask* des Feldes *VType* festgelegt.

Wird das Bit *varByRef* gesetzt, enthält die Variante eine Referenz auf einen Wert des Typs, der durch die Bits *varTypeMask* und *varArray* im Feld *VType* definiert ist.

Der Typencode *varString* gilt nur in Delphi. Varianten, die einen *varString*-Wert enthalten, sollten nicht an externe Funktionen, also an Funktionen außerhalb von Delphi, übergeben werden. Die Automatisierungsunterstützung von Delphi sorgt dafür, daß *varString*-Varianten vor der Übergabe als Parameter automatisch in *varOleStr*-Varianten umgewandelt werden.

Kapitel

12

Ablaufsteuerung

In diesem Kapitel wird beschrieben, wie Parameter und Funktionsergebnisse übergeben und gespeichert werden. Der letzte Abschnitt befaßt sich mit Exit-Prozeduren.

Parameter und Funktionsergebnisse

Auf welche Weise Parameter und Funktionsergebnisse übergeben werden, ist von verschiedenen Faktoren abhängig. Dazu gehören die Aufrufkonventionen, die Parametersemantik und der Typ und die Größe des zu übergebenden Wertes.

Parameter

Die Übergabe von Parametern an Prozeduren und Funktionen erfolgt entweder über CPU-Register oder über den Stack. Welche Übergabemethode verwendet wird, hängt von der Aufrufkonvention der Routine ab. Informationen über Aufrufkonventionen finden Sie unter »Aufrufkonventionen« auf Seite 6-5.

Variablenparameter **(var)** werden immer als Referenz übergeben, also als 32-Bit-Zeiger auf die tatsächliche Speicherposition.

Wert- und Konstantenparameter (**const**) werden abhängig vom Typ und der Größe des Parameters als Wert oder als Referenz übergeben:

- Ein Parameter ordinalen Typs wird als 8-Bit-, 16-Bit-, 32-Bit- oder 64-Bit-Wert übergeben. Dabei wird dasselbe Format verwendet wie bei einer Variablen des entsprechenden Typs.
- Ein Parameter reellen Typs wird immer im Stack übergeben. Ein *Single*-Parameter benötigt vier Byte, ein *Double*-, *Comp*-, oder *Currency*-Parameter belegt acht Byte. Auch ein *Real48*-Parameter belegt acht Byte. Dabei wird der *Real48*-Wert in den niederwertigen sechs Byte gespeichert. Ein *Extended*-Parameter belegt zwölf Byte, wobei der *Extended*-Wert in den niederwertigen zehn Byte gespeichert wird.

- Ein kurzer String-Parameter wird als 32-Bit-Zeiger auf einen kurzen String übergeben.
- Lange String-Parameter und dynamische Array-Parameter werden als 32-Bit-Zeiger auf den dynamischen Speicherblock übergeben, der für den langen String reserviert wurde. Für einen leeren langen String wird der Wert **nil** übergeben.
- Ein Zeiger, eine Klasse, eine Klassenreferenz oder ein Prozedurzeiger wird als 32-Bit-Zeiger übergeben.
- Ein Methodenzeiger wird immer als zwei 32-Bit-Zeiger im Stack übergeben. Der Instanzzeiger wird vor dem Methodenzeiger auf dem Stack abgelegt, so daß der Methodenzeiger die niedrigere Adresse erhält.
- Mit den Konventionen **register** und **pascal** wird ein Variantenparameter als 32-Bit-Zeiger auf einen Variant-Wert übergeben.
- Mengen, Records und statische Arrays aus einem, zwei oder vier Byte werden als 8-Bit-, 16-Bit- und 32-Bit-Werte übergeben. Größere Mengen, Records und statische Arrays werden als 32-Bit-Zeiger auf den Wert übergeben. Eine Ausnahme von dieser Regel ist, daß bei den Konventionen **cdecl**, **stdcall** und **safecall** die Records immer direkt im Stack übergeben werden. Die Größe eines auf diese Weise übergebenen Records wird immer bis zur nächsten Double-Word-Grenze erweitert.
- Ein offener Array-Parameter wird in Form zweier 32-Bit-Werte übergeben. Der erste Wert ist ein Zeiger auf die Array-Daten. Der zweite Wert enthält die Anzahl der Array-Elemente minus eins.

Bei der Übergabe zweier Parameter im Stack belegt jeder Parameter immer ein Vielfaches von vier Byte (also eine ganzzahlige Anzahl von Double Words). Ein 8-Bit- oder 16-Bit-Parameter wird auch dann als Double Word übergeben, wenn er nur ein Byte oder ein Word belegt. Der Inhalt der nicht verwendeten Bytes des Double Word ist nicht definiert.

Bei Verwendung der Konventionen **pascal**, **cdecl**, **stdcall** und **safecall** werden alle Parameter im Stack übergeben. Bei der Konvention **pascal** werden die Parameter in der Reihenfolge ihrer Deklaration (von links nach rechts) übergeben, so daß der erste Parameter im Stack an der obersten Adresse und der letzte Parameter an der untersten Adresse gespeichert wird. Bei den Konventionen **cdecl**, **stdcall** und **safecall** werden die Parameter in der entgegengesetzten Reihenfolge ihrer Deklaration (von rechts nach links) übergeben, so daß der erste Parameter im Stack an der untersten Adresse und der letzte an der obersten Adresse gespeichert wird.

Bei der Konvention **register** werden maximal drei Parameter in den CPU-Registern übergeben, der Rest im Stack. Die Parameter werden in der Reihenfolge ihrer Deklaration übergeben (wie bei der Konvention **pascal**). Die ersten drei geeigneten Parameter stehen in den Registern EAX, EDX und ECX (in dieser Reihenfolge). Nur reelle Typen und Methodenzeigertypen sind als Registerparameter ungeeignet. Sind mehr als drei mögliche Registerparameter vorhanden, werden die ersten drei in EAX, EDX und ECX übergeben. Die restlichen Parameter werden in der Reihenfolge ihrer Deklaration im Stack abgelegt. Betrachten Sie folgende Deklaration:

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E: Pointer);
```

Hier übergibt die Prozedur *Test* den Parameter *A* in EAX als 32-Bit-Integer, *B* in EDX als Zeichenzeiger und *D* in ECX als Zeiger auf einen Speicherblock für einen langen String. *C* wird im Stack in Form zweier Double Words und *E* als 32-Bit-Zeiger (in dieser Reihenfolge) abgelegt.

Konventionen zur Speicherung in Registern

Prozeduren und Funktionen dürfen die Register EBX, ESI, EDI und EBP nicht verändern. Die Register EAX, EDX und ECX stehen jedoch zur freien Verfügung. Außerdem werden Prozeduren und Funktionen immer unter der Voraussetzung aufgerufen, daß das Richtungsflag der CPU nicht gesetzt ist (entsprechend einer CLD-Anweisung). Auch nach Beendigung der Routine darf das Richtungsflag nicht gesetzt sein.

Funktionsergebnisse

Für die Rückgabe von Funktionsergebnissen gelten folgende Konventionen.

- Funktionsergebnisse ordinalen Typs werden in einem CPU-Register zurückgegeben. Bytes werden in AL, Words in AX und Double Words in EAX zurückgegeben.
- Die Funktionsergebnisse der Real-Typen werden im Top-of-Stack-Register des Coprozessors für Gleitkommazahlen (ST(0)) zurückgegeben. Bei Funktionsergebnissen vom Typ *Currency* wird der Wert von ST(0) mit dem Faktor 10000 skaliert. Beispielsweise wird der *Currency*-Wert 1.234 in ST(0) als 12340 zurückgegeben.
- Strings, dynamische Arrays, Methodenzeiger oder variante Typen werden so zurückgegeben, als ob das Funktionsergebnis als zusätzlicher **var**-Parameter nach den übrigen Parametern deklariert worden wäre. Die aufrufende Routine übergibt also einen zusätzlichen 32-Bit-Zeiger auf eine Variable, über die das Funktionsergebnis zurückgeliefert wird.
- Zeiger, Klassen, Klassenreferenzen und Prozedurzeiger werden in EAX zurückgegeben.
- Statische Arrays, Records und Mengen werden in AL zurückgegeben, wenn der Wert ein Byte belegt, in AX, falls der Wert zwei Byte belegt, und in EAX, falls vier Byte benötigt werden. Andernfalls wird der Funktion nach den deklarierten Parametern ein zusätzlicher **var**-Parameter übergeben, über den die Funktion das Ergebnis zurückliefert.

Methoden

Für Methoden werden dieselben Aufrufkonventionen wie für normale Prozeduren und Funktionen verwendet. Jede Methode verfügt jedoch über den zusätzlichen Parameter *Self*. Dabei handelt es sich um eine Referenz auf die Instanz oder Klasse, in der die Methode aufgerufen wird. Der Parameter *Self* wird als 32-Bit-Zeiger übergeben.

- Bei der Konvention **register** verhält sich der Parameter *Self*, als ob er *vor* allen anderen Parametern deklariert worden wäre. Er wird somit immer im Register EAX übergeben.
- Bei der Konvention **pascal** verhält sich der Parameter *Self*, als ob er nach allen anderen Parametern (einschließlich dem zusätzlichen **var**-Parameter für das Funktionsergebnis) deklariert worden wäre. Er wird somit als letzter Parameter übergeben und hat von allen Parametern die niedrigste Adresse.
- Bei den Konventionen **cdecl**, **stdcall** und **safecall** verhält sich der Parameter *Self*, als ob er vor allen anderen Parametern, aber nach dem zusätzlichen **var**-Parameter für das Funktionsergebnis deklariert worden wäre. Er wird daher als letzter Parameter, aber vor dem zusätzlichen **var**-Parameter (falls vorhanden) übergeben.

Konstruktoren und Destruktoren

Konstruktoren und Destruktoren verwenden dieselben Aufrufkonventionen wie die anderen Methoden. Es wird jedoch ein zusätzlicher Boolescher Flag-Parameter übergeben, der den Kontext des Konstruktor- oder Destruktoraufrufs anzeigt.

Der Wert *False* im Flag-Parameter eines Konstruktoraufrufs zeigt an, daß der Konstruktor über ein Instanzobjekt oder mit dem Schlüsselwort **inherited** aufgerufen wurde. In diesem Fall verhält sich der Konstruktor wie eine gewöhnliche Methode.

Der Wert *True* im Flag-Parameter eines Konstruktoraufrufs zeigt an, daß der Konstruktor über eine Klassenreferenz aufgerufen wurde. In diesem Fall erzeugt der Konstruktor eine Instanz der mit *Self* referenzierten Klasse und gibt eine Referenz auf das neu erzeugte Objekt in EAX zurück.

Der Wert *False* im Flag-Parameter eines Destruktoraufrufs zeigt an, daß der Destruktor mit dem Schlüsselwort **inherited** aufgerufen wurde. In diesem Fall verhält sich der Destruktor wie eine normale Methode.

Der Wert *True* im Flag-Parameter eines Destruktoraufrufs zeigt an, daß der Destruktor über ein Instanzobjekt aufgerufen wurde. In diesem Fall gibt der Destruktor die mit *Self* bezeichnete Instanz frei, bevor er beendet wird.

Der Flag-Parameter verhält sich so, als ob er vor allen anderen Parametern deklariert worden wäre. Bei der Konvention **register** wird er im Register DL übergeben, bei pascal erfolgt seine Übergabe vor allen anderen Parametern. Bei den Konventionen **cdecl**, **stdcall** und **safecall** wird er direkt vor dem Parameter *Self* übergeben.

Exit-Prozeduren

Mit *Exit-Prozeduren* können Sie sicherstellen, daß vor Beendigung eines Programms bestimmte Aktionen (z.B. das Aktualisieren und Schließen von Dateien) eingeleitet werden. Mit Hilfe der Zeigervariable *ExitProc* kann eine Exit-Prozedur »installiert« werden, die bei jeder Beendigung des Programms aufgerufen wird. Dabei ist es gleichgültig, ob das Programm normal, über einen Aufruf von *Halt* oder aufgrund eines Laufzeitfehlers beendet wird. Einer Exit-Prozedur werden keine Parameter übergeben.

Hinweis Es empfiehlt sich, alle Abläufe bei der Programmbeendigung nicht über Exit-Prozeduren, sondern über **finalization**-Abschnitte zu steuern. (Einzelheiten finden Sie unter »Der finalization-Abschnitt« auf Seite 3-5.) Exit-Prozeduren können nur verwendet werden, wenn eine EXE- oder DLL-Datei erzeugt wird. Bei der Verwendung von Packages muß das gewünschte Verhalten in einem **finalization**-Abschnitt implementiert werden. Alle Exit-Prozeduren werden aufgerufen, bevor die **finalization**-Abschnitte ausgeführt werden.

Exit-Prozeduren können von Units und von Programmen installiert werden. Eine Unit kann eine Exit-Prozedur im **initialization**-Abschnitt installieren. Die Prozedur ist dann für die erforderlichen Aufräumarbeiten verantwortlich (z.B. das Schließen von Dateien).

Bei korrekter Implementierung ist jede Exit-Prozedur ein Glied in einer Kette von Exit-Prozeduren. Alle Prozeduren in der Kette werden in der umgekehrten Reihenfolge ihrer Installation ausgeführt. Dadurch ist sichergestellt, daß der Beendigungscode einer bestimmten Unit nicht vor dem Beendigungscode der Units ausgeführt wird, die von ihr abhängen. Um die Beendigungskette nicht zu unterbrechen, müssen Sie den aktuellen Inhalt von *ExitProc* speichern, bevor Sie ihr die Adresse Ihrer eigenen Beendigungsprozedur zuweisen. Außerdem muß der gespeicherte Wert von *ExitProc* in der ersten Anweisung Ihrer Beendigungsprozedur wiederhergestellt werden.

Das folgende Beispiel skizziert die Implementierung einer solchen Prozedur:

```
var
   ExitSave: Pointer;
procedure MyExit;
begin
   ExitProc := ExitSave;  // Zuerst den alten Vektor wiederherstellen
   .
   .
   .
end;
begin
   ExitSave := ExitProc;
   ExitProc := @MyExit;
   .
   .
   .
end.
```

Zuerst wird der Inhalt von *ExitProc* in *ExitSave* gespeichert. Anschließend wird die Exit-Prozedur *MyExit* installiert. Nachdem die Prozedur als Teil des Beendigungsvorgangs aufgerufen wurde, wird zuerst die bisherige Exit-Prozedur wiederhergestellt.

Die Beendigungsroutine der Laufzeitbibliothek ruft Exit-Prozeduren auf, bis *ExitProc* den Wert **nil** annimmt. Um Endlosschleifen zu vermeiden, wird *ExitProc* vor jedem Aufruf auf **nil** gesetzt. Die nächste Exit-Prozedur wird somit nur aufgerufen, wenn ExitProc in der aktuellen Exit-Prozedur eine Adresse zugewiesen wird. Tritt in einer Exit-Prozedur ein Fehler auf, wird sie nicht erneut aufgerufen.

Eine Exit-Prozedur kann die Ursache einer Beendigung feststellen, indem sie die Integer-Variable *ExitCode* und die Zeigervariable *ErrorAddr* auswertet. Bei einer normalen Beendigung hat *ExitCode* den Wert Null und *ErrorAddr* den Wert **nil.** Wird ein Programm durch einen Aufruf von *Halt* beendet, enthält *ExitCode* den der Funktion *Halt* übergebenen Wert und *ErrorAddr* den Wert **nil.** Wird das Programm aufgrund

eines Laufzeitfehlers beendet, enthält *ExitCode* den Fehlercode und *ErrorAddr* die Adresse der ungültigen Anweisung.

Die letzte Exit-Prozedur (die von der Laufzeitbibliothek installiert wird) schließt die Ein- und Ausgabedateien. Hat *ErrorAddr* nicht den *Wert* **nil**, wird eine Meldung über den Laufzeitfehler ausgegeben. Wenn Sie selbst Meldungen zu Laufzeitfehlern ausgeben wollen, installieren Sie eine Exit-Prozedur, die *ErrorAddr* auswertet und eine Meldung ausgibt, wenn die Variable nicht den Wert **nil** hat. Zusätzlich müssen Sie vor dem Ende der Prozedur den Wert von *ErrorAddr* auf **nil** setzen, so daß der Fehler nicht in anderen Exit-Prozeduren erneut ausgegeben wird.

Nachdem die Laufzeitbibliothek alle Exit-Prozeduren aufgerufen hat, wird der in *ExitCode* gespeicherte Wert an Windows zurückgegeben.

Kapitel

13

Der integrierte Assembler

Der integrierte Assembler ermöglicht es, Intel-Assembler-Code direkt in Object-Pascal-Programme zu integrieren. Er implementiert eine umfangreiche Teilmenge der Syntax, die von Turbo Assembler und Macro Assembler von Microsoft unterstützt wird. Dazu gehören alle 8086/8087- und 80386/80387-Opcodes sowie einige der Ausdrucksoperatoren von Turbo Assembler. Zudem können Sie die Object-Pascal-Bezeicher in Assembler-Anweisungen verwenden.

Mit Ausnahme von DB, DW und DD (Define Byte, Word und Double Word) unterstützt der integrierte Assembler keine weiteren Direktiven von Turbo Assembler (z.B. EQU, PROC, STRUC, SEGMENT und MACRO). Operationen, die mit Turbo-Assembler-Direktiven implementiert werden, sind aber weitgehend mit äquivalenten Object-Pascal-Konstruktionen vergleichbar. Beispielsweise entsprechen die meisten EQU-Direktiven Konstanten-, Variablen- und Typdeklarationen, während die PROC-Direktive Prozedur- und Funktionsdeklarationen entspricht. Die STRUC-Direktive findet ihre Entsprechung in Datensatztypen.

Alternativ zur Verwendung des integrierten Assemblers können Sie OBJ-Datei hinzulinken, die als external deklarierte Prozeduren und Funktionen enthalten. Informationen hierzu finden Sie im Abschnitt »OBJ-Dateien linken« auf Seite 6-7.

Die Anweisung asm

Auf den integrierten Assembler greifen Sie über **asm**-Anweisungen zu, die folgende Syntax haben:

```
asm Anweisungsliste end
```

Dabei steht *Anweisungsliste* für eine Folge von Assembler-Anweisungen, die durch Strichpunkte, Zeilenendezeichen oder Object-Pascal-Kommentare voneinander getrennt werden.

Kommentare in einer **asm**-Anweisung müssen dem Object-Pascal-Stil entsprechen. Ein Strichpunkt besagt hier nicht, daß es sich beim Rest der Zeile um einen Kommentar handelt.

Das reserviert Wort **inline** und die Direktive **assembler** werden aus Gründen der Abwärtskompatibilität mitgeführt und haben keinerlei Auswirkung auf den Compiler.

Register

Im allgemeinen sind die Regeln für die Verwendung von Registern in einer asm-Anweisung identisch mit denjenigen für eine external-Prozedur oder -Funktion. In einer asm-Anweisung muß der Inhalt der Register EDI, ESI, ESP, EBP und EBX erhalten bleiben, während die Register EAX, ECX und EDX beliebig geändert werden können. Beim Eintritt in eine asm-Anweisung zeigt BP auf den aktuellen Stackframe, SP auf den Beginn des Stack, SS enthält die Segmentadresse des Stack-Segments und DS die Segmentadresse des Datensegments. Zu Beginn der Ausführung einer asm-Anweisung ist der Registerinhalt unbekannt. Eine Ausnahme bilden die Register EDI, ESI, ESP, EBP und EBX.

Syntax für Assembler-Anweisungen

Die Syntax für eine Assembler-Anweisung lautet:

Label: Präfix Opcode $Operand_1$, $Operand_2$

Label repräsentiert einen Label-Bezeichner, *Präfix* einen Assembler-Präfix-Opcode (Operationscode), *Opcode* einen Assembler-Anweisungscode oder eine Direktive, und *Operand* steht für einen Assembler-Ausdruck. *Label* und *Präfix* sind optional. Es gibt Opcodes mit nur einem Operanden, während andere überhaupt keine Operanden haben.

Kommentare sind nur zwischen, nicht aber innerhalb von Assembler-Anweisungen erlaubt:

```
MOV AX,1 {Anfangswert}     { OK }
MOV CX,100 {Zähler}        { OK }

MOV {Anfangswert} AX,1;    { Fehler! }
MOV CX, {Zähler} 100       { Fehler! }
```

Label

Label werden in Assembler auf die gleiche Weise definiert wie in Object Pascal: Vor einer Anweisung wird ein Label und ein Doppelpunkt eingefügt. Obwohl es für Label keine Längenbeschränkung gibt, sind nur die ersten 32 Zeichen signifikant. Wie in Object Pascal müssen auch in Assembler alle Label im label-Deklarationsabschnitt des Blocks definiert werden, der die asm-Anweisung enthält. Von dieser Regel gibt es eine Ausnahme: *lokale Label*.

Lokale Label beginnen immer mit dem Zeichen @. Sie setzen sich aus folgenden Zeichen zusammen: dem Zeichen @, gefolgt von einem oder mehreren Buchstaben, Ziffern, Unterstrichen oder @-Zeichen. Ein lokales Label ist auf asm-Anweisungen beschränkt. Der Gültigkeitsbereich eines lokalen Label erstreckt sich vom Schlüsselwort asm bis zum Schlüsselwort end in der asm-Anweisung, in der sich das Label befindet. Ein lokales Label braucht nicht deklariert zu werden.

Anweisungs-Opcodes

Der integrierte Assembler unterstützt die folgenden Opcodes:

LOCK	REP	REPE	REPZ	REPNE
REPNZ	SEGES	SEGCS	SEGSS	SEGDS
SEGFS	SEGGS	ADC,mLeft	ADD,mLeft	AND,mLeft
AAA,mAX	AAS,mAX	AAD,mAX	AAM,mAX	BOUND, mNONE
BSF,mLeft	BSR,mLeft	BT	BTC,mLeft	BTR,mLeft
BTS,mLeft	CALL,mNONE	CMP	CBW,mAX	CWDE,mAX
CWD, <mAX,mDX>	CDQ, <mAX,mDX>	CLC	CLD	CLI
CMC	CMPSB, <mSIDI>	CMPSW, <mSIDI>	CMPSD, <mSIDI>	DAA,mAX
DAS,mAX	DEC,mLeft	DIV,mLeft	ENTER, mNONE	HLT
IDIV,mLeft	IMUL,mLeft	IN,mLeft	INC,mLeft	INSB,mDI
INSW,mDI	INSD,mDI	INT	INTO	IRET
IRETD	JMP	JO	JNO	JC
JB	JNAE	JNC	JAE	JNB
JE	JZ	JNE	JNZ	JBE
JNA	JA	JNBE	JS	JNS
JP	JPE	JNP	JPO	JL
JNGE	JGE	JNL	JLE	JNG
JG	JNLE	JCXZ	JECXZ	LAHF,mAX
LEA,mLeft	LEAVE, mNONE	LDS,mSpecial	LES,mSpecial	LFS,mSpecial
LGS,mSpecial	LSS,mSpecial	LODSB, <mAX,mDI>	LODSW, <mAX,mDI>	LODSD, <mAX,mDI>
LOOP,mCX	LOOPE,mCX	LOOPZ,mCX	LOOPNE,mCX	LOOPNZ,mCX
LOOPD,mCX	LOOPDE,mCX	LOOPDZ,mCX	LOOPDNE, mCX	LOOPDNZ, mCX

MOV,mLeft	MOVSX,mLeft	MOVZX,mLeft	MOVSB, <mSIDI>	MOVSW, <mSIDI>
MOVSD, <mSIDI>	MUL,mLeft	NEG,mLeft	NOP	NOT,mLeft
OR,mLeft	OUT	OUTSB,mSI	OUTSW,mSI	OUTSD,mSI
POP,mLeft	POPF	POPA,mSpecial	POPAD, mSpecial	POPFD, mSpecial
PUSH	PUSHF	PUSHA	PUSHAD	PUSHFD
RET	RETN	RETF	SUB,mLeft	SBB,mLeft
RCL,mLeft	RCR,mLeft	ROL,mLeft	ROR,mLeft	SAL,mLeft
SHL,mLeft	SAR,mLeft	SHR,mLeft	SHLD,mLeft	SHRD,mLeft
SAHF	SCASB,mDI	SCASW,mDI	SCASD,mDI	STC
STD	STI	STOSB,mDI	STOSW,mDI	STOSD,mDI
TEST	WAIT	XCHG,<mLeft, mRight>	XLAT,mAX	XOR,mLeft
SETA,mLeft	SETAE,mLeft	SETB,mLeft	SETBE,mLeft	SETC,mLeft
SETE,mLeft	SETG,mLeft	SETGE,mLeft	SETL,mLeft	SETLE,mLeft
SETNA,mLeft	SETNAE,mLeft	SETNB,mLeft	SETNBE,mLeft	SETNC,mLeft
SETNE,mLeft	SETNG,mLeft	SETNGE,mLeft	SETNL,mLeft	SETNLE,mLeft
SETNO,mLeft	SETNP,mLeft	SETNS,mLeft	SETNZ,mLeft	SETO,mLeft
SETP,mLeft	SETPE,mLeft	SETPO,mLeft	SETS,mLeft	SETZ,mLeft
ARPL	LAR,mLeft	CLTS	LGDT	SGDT
LIDT	SIDT	LLDT	SLDT	LMSW
SMSW	LSL,mLeft	LTR,mLeft	STR,mLeft	VERR
VERW	BSWAP,mLeft	XADD,mLeft	CMPXCHG, <mLeft,mAX>	INVD
WBINVD	INVLPG	FLD,m87	FILD,m87	FST,m87
FSTP,m87	FIST,m87	FISTP,m87	FADD,m87	FADDP,m87
FIADD,m87	FSUB,m87	FSUBP,m87	FSUBR,m87	FSUBRP,m87
FISUB,m87	FISUBR,m87	FMUL,m87	FMULP,m87	FIMUL,m87
FDIV,m87	FDIVP,m87	FDIVR,m87	FDIVRP,m87	FIDIV,m87
FIDIVR,m87	FCOM,m87	FCOMP,m87	FCOMPP,m87	FICOM,m87
FICOMP,m87	F2XM1,m87	FABS,m87	FBLD,m87	FBSTP,m87
FCHS,m87	FDECSTP,m87	FFREE,m87	FINCSTP,m87	FLD1,m87
FLDCW,m87	FLDENV,m87	FLDL2E,m87	FLDL2T,m87	FLDLG2,m87
FLDLN2,m87	FLDPI,m87	FLDZ,m87	FNOP,m87	FPREM,m87

FPATAN,m87	FPTAN,m87	FRNDINT,m87	FRSTOR,m87	FSCALE,m87
FSETPM,m87	FSQRT,m87	FTST,m87	FWAIT,m87	FXAM,m87
FXCH,m87	FXTRACT,m87	FYL2X,m87	FYL2XP1,m87	FCLEX,m87
FNCLEX,m87	FDISI,m87	FNDISI,m87	FENI,m87	FNENI,m87
FINIT,m87	FNINIT,m87	FSAVE,m87	FNSAVE,m87	FSTCW,m87
FNSTCW,m87	FSTENV,m87	FNSTENV,m87	FSTSW,m87	FNSTSW,m87
FUCOM,m87	FUCOMP,m87	FUCOMPP,m87	FPREM1,m87	FCOS,m87
FSIN,m87	FSINCOS,m87			

Eine vollständige Beschreibung aller Anweisungen finden Sie in der Dokumentation Ihres Mikroprozessors.

Der Befehl RET

Der Befehl RET bewirkt immer ein Near-Return.

Sprungbefehle

Wenn nicht anders angegeben, optimiert der integrierte Assembler Sprunganweisungen durch automatische Auswahl der kürzesten und damit effektivsten Form einer Sprunganweisung. Diese automatische Anpassung wird bei der nicht bedingten Sprunganweisung (JMP) und allen bedingten Sprunganweisungen angewendet, wenn es sich bei dem Ziel um ein Label (und nicht um eine Prozedur oder Funktion) handelt.

Bei einer nicht bedingten Sprunganweisung (JMP) erzeugt der integrierte Assembler einen kurzen Sprung (ein Byte Opcode und ein Byte mit Angabe der Sprungweite), wenn die Adreßdifferenz zum Ziel-Label im Bereich von -128 bis 127 Byte liegt. Andernfalls wird ein Near-Sprung generiert (ein Byte Opcode und zwei Byte für die Sprungweite).

Bei einer bedingten Sprunganweisung wird ein kurzer Sprung (ein Byte Opcode und ein Byte mit Angabe der Sprungweite) erzeugt, wenn der Adreßabstand zum Ziel-Label im Bereich von -128 bis 127 Byte liegt. Andernfalls generiert der integrierte Assembler einen kurzen Sprung mit der inversen Bedingung, bei dem über einen Near-Sprung zum Ziel-Label gesprungen wird (insgesamt fünf Byte). Nehmen wir an, in der folgenden Assembler-Anweisung liegt *Stop* nicht innerhalb der Reichweite eines kurzen Sprungs:

```
JC      Stop
```

Diese Anweisung wird in eine Maschinencode-Sequenz wie die folgende umgewandelt:

```
JNC     Skip
JMP     Stop
Skip:
```

Sprünge zu Eintrittspunkten von Prozeduren und Funktionen sind immer Near-Sprünge.

Assembler-Direktiven

Der integrierte Assembler unterstützt die drei Direktiven DB (Define Byte), DW (Define Word) und DD (Define Double Word). Jede dieser Direktiven erzeugt Daten, die den durch Kommas voneinander getrennten, nachgestellten Operanden entsprechen.

Die Direktive DB erzeugt eine Byte-Folge. Jeder Operand kann ein konstanter Ausdruck mit einem Wert von -128 bis 255 oder ein String von beliebiger Länge sein. Konstante Ausdrücke erzeugen Code mit einer Länge von einem Byte. Strings definieren eine Byte-Folge, die den ASCII-Codes der enthaltenen Zeichen entsprechen.

Die Direktive DW erzeugt eine Word-Sequenz. Jeder Operand kann ein konstanter Ausdruck mit einem Wert von -32.768 bis 65.535 oder ein Adreßausdruck sein. Für einen Adreßausdruck erzeugt der integrierte Assembler einen Near-Zeiger, d.h. ein Word, das den Offset-Anteil der Adresse enthält.

Die Direktive DD erzeugt eine Folge von Double Words. Jeder Operand kann ein konstanter Ausdruck mit einem Wert von -2.147.483.648 bis 4.294.967.295 oder ein Adreßausdruck sein. Bei einem Adressausdruck erzeugt der integrierte Assembler einen Far-Zeiger, d.h. ein Word, das den Offset-Anteil der Adresse enthält, gefolgt von einem Word, das die Segment-Komponente der Adresse enthält.

Die durch die Direktiven DB, DW und DD erzeugten Daten werden wie der Code, der von anderen Anweisungen des integrierten Assemblers erzeugt wird, immer im Code-Segment gespeichert. Zur Erstellung nichtinitialisierter Daten im Datensegment müssen Sie die **var**- und **const**-Deklarationen von Object Pascal verwenden.

Hier einige Beispiele für die Direktiven DB, DW und DD:

```
asm
   DB      0FFH                        { Ein Byte }
   DB      0,99                        { Zwei Byte }
   DB      'A'                         { Ord('A') }
   DB      'Hello world...',0DH,0AH    { String gefolgt von CR/LF }
   DB      12,"Delphi"                 { String im Object-Pascal-Stil }
   DW      0FFFFH                      { Ein Word }
   DW      0,9999                      { Zwei Words }
   DW      'A'                         { Identisch mit DB 'A',0 }
   DW      'BA'                        { Identisch mit DB 'A','B' }
   DW      MyVar                       { Offset von MyVar }
   DW      MyProc                      { Offset von MyProc }
   DD      0FFFFFFFFH                  { Ein Double Word }
   DD      0,999999999                 { Zwei Double Words }
   DD      'A'                         { Identisch mit DB 'A',0,0,0 }
   DD      'DCBA'                      { Identisch mit DB 'A','B','C','D' }
   DD      MyVar                       { Zeiger auf MyVar }
   DD      MyProc                      { Zeiger auf MyProc }
end;
```

Wenn in Turbo Assembler vor einem Bezeichner eine DB-, DW- oder DD-Direktive steht, führt dies zur Deklaration einer Variablen mit einem Byte, einem Word bzw. ei-

nem Double Word an der Position der Direktive. Die folgenden Anweisungen sind beispielsweise in Turbo Assembler zulässig:

```
ByteVar      DB      ?
WordVar      DW      ?
IntVar       DD      ?
   ⋮
             MOV     AL,ByteVar
             MOV     BX,WordVar
             MOV     ECX,IntVar
```

Der integrierte Assembler unterstützt diese Variablendeklarationen nicht. Das einzige Symbol, das in einer Anweisung des integrierten Assemblers definiert werden kann, ist ein Label. Alle Variablen müssen mit Hilfe der Object-Pascal-Syntax deklariert werden. Die obige Konstruktion entspricht folgender Deklaration:

```
var
   ByteVar: Byte;
   WordVar: Word;
   IntVar: Integer;
   ⋮
asm
   MOV     AL,ByteVar
   MOV     BX,WordVar
   MOV     ECX,IntVar
end;
```

Operanden

Operanden im integrierten Assembler sind Ausdrücke, die aus Konstanten, Registern, Symbolen und Operatoren bestehen.

Die folgenden reservierten Wörter haben bei ihrer Verwendung in Operanden eine vordefinierte Bedeutung.

Tabelle 13.1 Reservierte Wörter im integrierten Assembler

AH	BX	DI	EBX	ESP	PTR	SS
AL	BYTE	DL	ECX	HIGH	QWORD	ST
AND	CH	DS	EDI	LOW	SHL	TBYTE
AX	CL	DWORD	EDX	MOD	SHR	TYPE
BH	CS	DX	EID	NOT	SI	WORD
BL	CX	EAX	ES	OFFSET	SP	XOR
BP	DH	EBP	ESI	OR		

Reservierte Wörter haben immer Vorrang vor benutzerdefinierten Bezeichnern. Im folgenden Code-Fragment wird 1 nicht in die Variable *CH*, sondern in das Register CH geladen:

```
var
   Ch: Char;
   ⋮
asm
   MOV     CH, 1
end;
```

Wenn Sie auf ein benutzerdefiniertes Symbol zugreifen wollen, das den Namen eines reservierten Wortes trägt, müssen Sie den Operator **&** zum Überschreiben des Bezeichners verwenden:

```
MOV     &Ch, 1
```

Benutzerdefinierte Bezeichner sollten möglichst nie mit den Namen reservierter Wörter belegt werden.

Ausdrücke

Der integrierte Assembler wertet alle Ausdrücke als 32-Bit-Integer aus. Gleitkomma- und String-Werte werden mit Ausnahme von String-Konstanten nicht unterstützt.

Ausdrücke im Assembler setzen sich aus *Ausdruckselementen* und *Operatoren* zusammen und gehören zu einer bestimmten Ausdrucksklasse und zu einem bestimmten Ausdruckstyp.

Unterschiede zwischen Ausdrücken in Object Pascal und Assembler

Der größte Unterschied zwischen Object-Pascal-Ausdrücken und Ausdrücken des integrierten Assemblers besteht darin, daß alle Assembler-Ausdrücke einen konstanten *Wert* ergeben müssen, d.h. einen Wert, der während der Compilierung berechnet werden kann. Beispielsweise ist für die Deklarationen

```
const
   X = 10;
   Y = 20;
var
   Z: Integer;
die folgende Assembler-Anweisung zulässig:
asm
   MOV     Z,X+Y
end;
```

Da *X* und *Y* Konstanten sind, ist der Ausdruck X + Y nur eine andere Möglichkeit zur Darstellung der Konstante 30. Die resultierende Anweisung bewirkt eine direkte Speicherung des Wertes 30 in der Word-Variablen *Z*. Wenn *X* und *Y* aber Variablen sind, kann der integrierte Assembler den Wert von X + Y nicht während der Compilierung berechnen:

```
var
  X, Y: Integer;
```

In diesem Fall müßten Sie folgende Anweisung verwenden, um die Summe von *X* und *Y* in *Z* zu speichern:

```
asm
   MOV     EAX,X
   ADD     EAX,Y
   MOV     Z,EAX
end;
```

In einem Object-Pascal-Ausdruck wird eine Referenz auf eine Variable als Inhalt der Variablen interpretiert, im integrierten Assembler dagegen als *Adresse* der Variablen. Beispielsweise bezieht sich in Object Pascal der Ausdruck X + 4, in dem *X* eine Variable ist, auf den Inhalt von X + 4. Im integrierten Assembler bedeutet derselbe Ausdruck, daß sich der Inhalt des Word an einer Adresse befindet, die um vier Byte höher ist als die Adresse von *X*. Dazu ein Beispiel:

```
asm
   MOV     EAX,X+4
end;
end;
```

Obwohl dieser Code zulässig ist, würde er nicht den Wert von X + 4 in AX laden, sondern den Wert eines Word, das vier Byte über *X* liegt. Um 4 zum Inhalt von *X* zu addieren, müssen Sie folgende Anweisung verwenden:

```
asm
   MOV     EAX,X
   ADD     EAX,4
end;
```

Ausdruckselemente

Zu den Elementen eines Ausdrucks gehören *Konstanten*, *Register* und *Symbole.*

Konstanten

Der integrierte Assembler unterstützt zwei Konstantentypen: *numerische Konstanten* und *String-Konstanten.*

Numerische Konstanten

Numerische Konstanten müssen Integer-Zahlen sein, deren Wert im Bereich von -2147483648 bis 4294967295 liegt.

Per Voreinstellung wird bei numerischen Konstanten die dezimale Notation verwendet. Der integrierte Assembler unterstützt aber auch die binäre, die oktale und die hexadezimale Notation (Basis 16). Zur Kennzeichnung der binären Notation wird der Zahl der Buchstabe B nachgestellt. In oktaler Notation steht nach der Zahl der Buchstabe O. Zur Kennzeichnung einer Hexadezimalzahl kann entweder nach der Zahl der Buchstabe H oder vor der Zahl das Zeichen $ stehen.

Numerische Konstanten müssen mit einer Ziffer von 0 bis 9 oder dem Zeichen $ beginnen. Wenn Sie eine hexadezimale Konstante mit dem Suffix *H* angeben und die erste signifikante Ziffer eine hexadezimale Ziffer zwischen A und F ist, müssen Sie eine zusätzliche Null an den Beginn der Zahl stellen. Beispielsweise handelt es sich bei

0BAD4H und $BAD4 um hexadezimale Konstanten, bei BAD4H aber um einen Bezeichner, weil der Ausdruck mit einem Buchstaben beginnt.

String-Konstanten

String-Konstanten müssen in halbe oder ganze Anführungszeichen eingeschlossen werden. Zwei aufeinanderfolgende Anführungszeichen, die vom selben Typ wie die umgebenden Anführungszeichen sind, werden als ein einzelnes Zeichen interpretiert. Hier einige Beispiele für String-Konstanten:

```
'Z'
'Delphi'
"Das ist alles, Leute "
'"Das war''s Leute," sagte er.'
'100'
'"'
"'"
```

In DB-Direktiven sind String-Konstanten von beliebiger Länge erlaubt. Sie bewirken die Zuweisung einer Byte-Folge mit den ASCII-Werten der String-Zeichen. In allen anderen Fällen darf eine String-Konstante maximal vier Zeichen umfassen und einen numerischen Wert angeben, der in einem Ausdruck zugelassen ist. Der numerische Wert einer String-Konstante ergibt sich wie folgt:

```
Ord(Ch1) + Ord(Ch2) shl  8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

Dabei ist *Ch1* das am weitesten rechts stehende (letzte) Zeichen und *Ch4* das am weitesten links stehende (erste) Zeichen. Wenn der String weniger als vier Zeichen umfaßt, werden die am weitesten links stehenden Zeichen als Null vorausgesetzt. Die folgende Tabelle enthält einige Beispiele für String-Konstanten und die entsprechenden numerischen Werte:

Tabelle 13.2 Beispiele für String-Konstanten und ihre Werte

String	Wert
'a'	00000061H
'ba'	00006261H
'cba'	00636261H
'dcba'	64636261H
'a '	00006120H
'a’	20202061H
'a' * 2	000000E2H
'a'-'A'	00000020H
not 'a'	FFFFFF9EH

Register

Die reservierten Symbole in der folgenden Tabelle bezeichnen CPU-Register.

Tabelle 13.3 CPU-Register

32-Bit-Allzweckregister	EAX EBX ECX EDX	32-Bit-Zeiger- oder Indexregister	ESP EBP ESI EDI
16-Bit-Allzweckregister	AX BX CX DX	16-Bit-Zeiger- oder Indexregister	SP BP SI DI
Untere 8-Bit-Register	AL BL CL DL	16-Bit-Segmentregister	CS DS SS ES
Obere 8-Bit-Register	AH BH CH DH	Coprozessor-Registerstapel	ST

Wenn ein Operand nur aus einem Registernamen besteht, wird er als *Register-Operand* bezeichnet. Als Register-Operanden können alle Register verwendet werden. Einige Register lassen sich auch in einem anderen Kontext einsetzen.

Die Basisregister (BX und BP) und die Indexregister (SI und DI) werden zur Kennzeichnung der Indizierung in eckigen Klammern angegeben. Folgende Kombinationen von Basis-/Index-Registern sind erlaubt: [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI] und [BP+DI]. Sie können auch mit allen 32-Bit-Registern indizieren, z.B. [EAX+ECX], [ESP] und [ESP+EAX+5].

Die 16-Bit-Segmentregister (ES, CS, SS und DS) werden unterstützt (im 32-Bit-Code gibt es allerdings keine Segmente).

Das Symbol ST bezeichnet das oberste Register im 8087-Gleitkommaregister-Stack. Jedes der acht Gleitkommaregister kann mit ST(*X*) referenziert werden, wobei *X* eine Konstante von 0 bis 7 ist, die den Abstand vom oberen Ende des Stack angibt.

Symbole

Der integrierte Assembler ermöglicht den Zugriff auf nahezu alle Object-Pascal-Bezeichner in Assembler-Ausdrücken, einschließlich Konstanten, Typen, Variablen, Prozeduren und Funktionen. Außerdem ist im integrierten Assembler das spezielle Symbol *@Result* implementiert, das der Variablen *Result* im Anweisungsteil einer Funktion entspricht. Die Funktion

```
function Sum(X, Y: Integer): Integer;
begin
  Result := X + Y;
end;
```

wird im Assembler folgendermaßen angegeben:

```
function Sum(X, Y: Integer): Integer; stdcall;
begin
  asm
    MOV     EAX,X
    ADD     EAX,Y
    MOV     @Result,EAX
  end;
end;
```

Die folgenden Symbole dürfen in **asm**-Anweisungen nicht verwendet werden:

- Standardprozeduren und -funktionen (z.B. *WriteLn* und *Chr*).
- Die speziellen Arrays *Mem*, *MemW*, *MemL*, *Port* und *PortW*.
- String-, Gleitkomma- und Mengenkonstanten.
- Label, die nicht im aktuellen Block deklariert sind.
- Das Symbol *@Result* außerhalb einer Funktion.

Die folgende Tabelle faßt die Symbolarten zusammen, die in **asm**-Anweisungen verwendet werden können.

Tabelle 13.4 Im integrierten Assembler verwendbare Symbole

Symbol	Wert	Klasse	Typ
Label	Adresse des Label	Speicherreferenz	SHORT
Konstante	Wert der Konstanten	Direkter Wert	0
Typ	0	Speicherreferenz	Größe des Typs
Feld	Offset des Feldes	Speicher	Größe des Typs
Variable	Adresse der Variablen	Speicherreferenz	Größe des Typs
Prozedur	Adresse der Prozedur	Speicherreferenz	NEAR
Funktion	Adresse der Funktion	Speicherreferenz	NEAR
Unit	0	Direkter Wert	0
@Code	Codesegment-Adresse	Speicherreferenz	0FFF0H
@Data	Datensegment-Adresse	Speicherreferenz	0FFF0H
@Result	Ergebnisvariablen-Offset	Speicherreferenz	Größe des Typs

Bei deaktivierter Optimierung werden lokale (also in Prozeduren und Funktionen deklarierte) Variablen immer auf dem Stack abgelegt. Der Zugriff erfolgt immer relativ zu EBP. Der Wert eines lokalen Variablensymbols besteht in seinem mit Vorzeichen versehenen Offset von EBP. Der Assembler addiert zu Referenzen auf lokale Variablen automatisch [EBP] hinzu. Beispielsweise wird für die Deklaration

```
var Count: Integer;
```

in einer Funktion oder Prozedur die Anweisung

```
MOV     EAX,Count
```

in MOV EAX,[EBP–4] assembliert.

Der integrierte Assembler behandelt einen **var**-Parameter immer als 32-Bit-Zeiger. Die Größe eines **var**-Parameters beträgt immer 4 Byte. Die Syntax für den Zugriff auf einen **var**-Parameter unterscheidet sich von derjenigen für einen Wert-Parameter. Für den Zugriff auf den Inhalt eines **var**-Parameters müssen Sie zuerst den 32-Bit-Zeiger laden und dann auf die Position zugreifen, auf die er zeigt:

```
function Sum(var X, Y: Integer): Integer; stdcall;
begin
  asm
    MOV     EAX,X
    MOV     EAX,[EAX]
```

```
    MOV     EDX,Y
    ADD     EAX,[EDX]
    MOV     @Result,AX
  end;
end;
```

Bezeichner können in **asm**-Anweisungen qualifiziert werden. So lassen sich für die Deklarationen

```
type
  TPoint = record
    X, Y: Integer;
  end;
  TRect = record
    A, B: TPoint;
  end;
var
  P: TPoint;
  R: TRect;
```

die folgenden Konstruktionen für den Zugriff auf Felder in einer **asm**-Anweisung angeben:

```
MOV     EAX,P.X
MOV     EDX,P.Y
MOV     ECX,R.A.X
MOV     EBX,R.B.Y
```

Typbezeichner können zur einfachen und schnellen Konstruktion von Variablen verwendet werden. Alle folgenden Anweisungen erzeugen denselben Maschinencode, der den Inhalt von EDX in EAX lädt:

```
MOV     EAX,(TRect PTR [EDX]).B.X
MOV     EAX,TRect(EDX]).B.X
MOV     EAX,TRect[EDX].B.X
MOV     EAX,[EDX].TRect.B.X
```

Ausdrucksklassen

Der integrierte Assembler unterteilt Ausdrücke in drei Klassen: *Register, Speicherreferenzen* und *direkte Werte.*

Ausdrücke, die nur aus einem Registernamen bestehen, nennt man *Registerausdrücke (z.B. AX, CL, DI* oder *ES). Registerausdrücke*, die als Operanden verwendet werden, veranlassen den Assembler zur Erzeugung von Anweisungen, die auf die CPU-Register zugreifen.

Ausdrücke, die Speicheradressen bezeichnen, nennt man Speicherreferenzen. Zu dieser Kategorie gehören Label, Variablen, typisierte Konstanten, Prozeduren und Funktionen von Object Pascal.

Ausdrücke, bei denen es sich nicht um Register handelt und die auch nicht auf Speicheradressen zeigen, werden als direkte Werte bezeichnet. Zu dieser Gruppe gehören untypisierte Konstanten und Typbezeichner von Object Pascal.

Wenn direkte Werte und Speicherreferenzen als Operanden verwendet werden, führt dies zu unterschiedlichem Code. Ein Beispiel:

```
const
   Start = 10;
var
   Count: Integer;
   ⋮
asm
   MOV     EAX,Start            { MOV EAX,xxxx }
   EBX,Count                    { MOV EBX,[xxxx] }
   MOV     ECX,[Start]          { MOV ECX,[xxxx] }
   MOV     EDX,OFFSET Count     { MOV EDX,xxxx }
end;
```

Da *Start* ein direkter Wert ist, wird das erste MOV in eine Move-Immediate-Anweisung assembliert. Das zweite MOV wird in eine Move-Memory-Anweisung übersetzt, weil *Count* eine Speicherreferenz ist. Im dritten MOV wird *Start* wegen der ekkigen Klammern in eine Speicherreferenz umgewandelt (in diesem Fall handelt es sich um das Word mit dem Offset 10 im Datensegment). Im vierten MOV sorgt der Operator OFFSET für die Konvertierung von *Count* in einen direkten Wert (mit dem Offset von *Count* im Datensegment).

Die eckigen Klammern und der Operator OFFSET ergänzen einander. Die folgende **asm**-Anweisung erzeugt denselben Maschinencode wie die ersten beiden Zeilen der obigen Anweisung:

```
asm
   MOV     EAX,OFFSET [Start]
   MOV     EBX,[OFFSET Count]
end;
```

Bei Speicherreferenzen und direkten Werten findet eine weitere Unterteilung in *verschiebbare* und *absolute Ausdrücke* statt. Unter einer Verschiebung versteht man den Vorgang, bei dem der Linker Symbolen eine absolute Adresse zuweist. Ein verschiebbarer Ausdruck bezeichnet einen Wert, für den beim Linken eine Verschiebung (Relokation) erforderlich ist. Dagegen bezeichnet ein absoluter Ausdruck einen Wert, bei dem dies nicht nötig ist. In der Regel handelt es sich bei Ausdrücken, die ein Label, eine Variable, eine Prozedur oder eine Funktion referenzieren, um verschiebbare Ausdrücke, weil die endgültige Adresse dieser Symbole zur Compilierungszeit nicht bekannt ist. Absolut sind dagegen Ausdrücke, die ausschließlich Konstanten bezeichnen.

Im integrierten Assembler kann mit absoluten Werten jede Operation ausgeführt werden. Mit verschiebbaren Ausdrücken ist dagegen nur die Addition und Subtraktion von Konstanten möglich.

Ausdruckstypen

Jedem Assembler-Ausdruck ist ein bestimmter Typ (genauer gesagt eine bestimmte Größe) zugeordnet, weil der Assembler den Typ eines Ausdrucks einfach aus der Größe seiner Speicherposition abliest. Beispielsweise hat eine Integer-Variable den Typ vier, weil sie vier Byte Speicherplatz belegt. Der integrierte Assembler führt, wenn möglich, immer eine Typenprüfung durch. Aus den folgenden Anweisungen ergibt sich beispielsweise, daß die Größe von *QuitFlag* eins (ein Byte) und die Größe von *OutBufPtr* zwei (ein Word) beträgt:

```
var
   QuitFlag: Boolean;
   OutBufPtr: Word;
   ⋮
asm
   MOV     AL,QuitFlag
   MOV     BX,OutBufPtr
end;
```

Die folgende Anweisung führt zu einem Fehler:

```
MOV     DL,OutBufPtr
```

Das Problem liegt darin, daß DL nur ein Byte groß ist, während *OutBufPtr* ein Word ist. Der Typ einer Speicherreferenz kann durch eine Typumwandlung geändert werden. Die obige Anweisung müßte also folgendermaßen formuliert werden:

```
MOV     DL,BYTE PTR OutBufPtr
MOV     DL,Byte(OutBufPtr)
MOV     DL,OutBufPtr.Byte
```

Diese MOV-Anweisungen referenzieren das erste (niederwertige) Byte der Variablen *OutBufPtr*.

Es gibt Fälle, in denen eine Speicherreferenz untypisiert ist. Ein Beispiel hierfür ist ein direkter Wert, der in eckige Klammern gesetzt ist:

```
MOV     AL,[100H]
MOV     BX,[100H]
```

Der integrierte Assembler läßt beide Anweisungen zu, da der Anweisung [100H] kein Typ zugeordnet ist (sie bezeichnet einfach den Inhalt der Adresse 100H im Datensegment) und der Typ anhand des ersten Operanden feststellbar ist (Byte für AL, Word für BX). Falls sich der Typ nicht über einen anderen Operanden ermitteln läßt, verlangt der integrierte Assembler eine explizite Typumwandlung:

```
INC     BYTE PTR [100H]
IMUL    WORD PTR [100H]
```

Die folgende Tabelle enthält die vordefinierten Typensymbole, die der integrierte Assembler zusätzlich zu den aktuell deklarierten Object-Pascal-Typen bereitstellt.

Tabelle 13.5 Vordefinierte Typensymbole

Symbol	Typ
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

Ausdrucksoperatoren

Der integrierte Assembler stellt eine Vielzahl von Operatoren bereit. Die Regeln für die Rangfolge der Operatoren unterscheiden sich von den Regeln in Object Pascal. Beispielsweise haben die Operatoren für Addition und Subtraktion in einer asm-Anweisung Vorrang gegenüber AND. Die folgende Tabelle enthält die Ausdrucksoperatoren des integrierten Assemblers. Die Operatoren sind nach ihrer Rangfolge sortiert.

Tabelle 13.6 Rangfolge der Ausdrucksoperatoren des integrierten Assemblers

Operatoren	Erläuterungen	Rangfolge
&		Höchste Stufe
(), **[]**, **.**, **HIGH**, **LOW**		
+, -	Unäres **+** und **-**	
:		
OFFSET, **SEG**, **TYPE**, **PTR**, *, /, **MOD**, **SHL**, **SHR**, +,	Binäres **+** und **-**	
NOT, **AND**, **OR**, **XOR**		Niedrigste Stufe

Die folgende Tabelle faßt die Ausdrucksoperatoren des integrierten Assemblers zusammen.

Tabelle 13.7 Erläuterung der Ausdrucksoperatoren des integrierten Assemblers

Operator	Beschreibung
&	**Überschreiben von Bezeichnern**. Der Bezeichner, der unmittelbar auf das Zeichen & folgt, wird als benutzerdefiniertes Symbol betrachtet. Dies gilt auch dann, wenn er mit einem reservierten Symbol des integrierten Assemblers identisch ist.
(...)	**Unterausdruck**. Ausdrücke in Klammern werden vollständig ausgewertet und als einzelnes Ausdruckselement betrachtet. Optional kann dem Ausdruck in Klammern ein weiterer Ausdruck vorangestellt werden. Das Resultat ist in diesem Fall die Summe der Werte der beiden Ausdrücke. Der Typ des ersten Ausdrucks bestimmt den Ergebnistyp.

Tabelle 13.7 Erläuterung der Ausdrucksoperatoren des integrierten Assemblers (Fortsetzung)

Operator	Beschreibung
[...]	**Speicherreferenz**. Der Ausdruck in eckigen Klammern wird vollständig ausgewertet und dann als einzelnes Ausdruckselement betrachtet. Der Ausdruck in den eckigen Klammern kann über den Plus-Operator (+) mit dem Register BX, BP, SI oder DI kombiniert werden, um einen CPU-Registerindex zu bilden. Optional kann dem Ausdruck in eckigen Klammern ein weiterer Ausdruck vorangestellt werden. Das Resultat ist in diesem Fall die Summe der Werte der beiden Ausdrücke. Der Typ des ersten Ausdrucks bestimmt den Ergebnistyp. Das Ergebnis ist immer eine Speicherreferenz.
.	**Selektor für Strukturelemente**. Das Resultat ergibt sich aus der Addition der Ausdrücke vor und nach dem Punkt. Der Typ des Ausdrucks nach dem Punkt bestimmt den Ergebnistyp. Im Ausdruck nach dem Punkt kann auf Symbole, die zum Gültigkeitsbereich des Ausdrucks vor dem Punkt gehören, zugegriffen werden.
HIGH	Gibt die höherwertigen acht Bits des Word-Ausdrucks zurück, der auf den Operator folgt. Der Ausdruck muß ein absoluter direkter Wert sein.
LOW	Gibt die niederwertigen acht Bits des Word-Ausdrucks zurück, der auf den Operator folgt. Der Ausdruck muß ein absoluter direkter Wert sein.
+	**Unäres Plus**. Liefert den auf das Pluszeichen folgenden Ausdruck ohne Änderungen zurück. Der Ausdruck muß ein absoluter direkter Wert sein.
-	**Unäres Minus**. Liefert den negativen Wert des Ausdrucks zurück, der auf das Minuszeichen folgt. Der Ausdruck muß ein absoluter direkter Wert sein.
+	**Addition**. Die Ausdrücke können direkte Werte oder Speicherreferenzen sein. Nur einer der Ausdrücke darf aus einem verschiebbaren Wert bestehen. Handelt es sich bei einem der Ausdrücke um einen verschiebbaren Wert, ist das Ergebnis ebenfalls ein verschiebbarer Wert. Ist einer der Ausdrücke eine Speicherreferenz, ist auch das Ergebnis eine Speicherreferenz.
-	**Subtraktion**. Der erste Ausdruck kann zu einer beliebigen Klasse gehören, der zweite muß ein absoluter direkter Wert sein. Das Ergebnis gehört zur gleichen Klasse wie der erste Ausdruck.
:	**Überschreiben von Segmenten**. Teilt dem Assembler mit, daß der Ausdruck nach dem Doppelpunkt zu dem Segment gehört, das über den Segmentregisternamen (CS, DS, SS, FS, GS oder ES) vor dem Doppelpunkt angegeben ist. Das Ergebnis ist eine Speicherreferenz mit dem Wert des Ausdrucks nach dem Doppelpunkt. Wenn in einem Anweisungsoperanden das Überschreiben eines Segments verwendet wird, wird der Anweisung eine entsprechende Präfixanweisung zur Segmentüberschreibung vorangestellt. Dies stellt sicher, daß das angegebene Segment ausgewählt ist.
OFFSET	Liefert den Offset-Anteil (Double Word) des Ausdrucks zurück, der auf den Operator folgt. Das Ergebnis ist ein direkter Wert.
TYPE	Liefert den Typ (die Größe in Byte) des Ausdrucks zurück, der auf den Operator folgt. Der Typ eines direkten Wertes ist 0.
PTR	**Typumwandlungsoperator**. Das Ergebnis ist eine Speicherreferenz mit dem Wert des Ausdrucks, der auf den Operator folgt, und mit dem Typ des Ausdrucks, der dem Operator vorangeht.
*****	**Multiplikation**. Beide Ausdrücke müssen absolute, direkte Werte sein. Das Ergebnis ist ebenfalls ein absoluter direkter Wert.
/	**Integerdivision**. Beide Ausdrücke müssen absolute, direkte Werte sein. Das Ergebnis ist ebenfalls ein absoluter direkter Wert.

Tabelle 13.7 Erläuterung der Ausdrucksoperatoren des integrierten Assemblers (Fortsetzung)

Operator	Beschreibung
MOD	**Rest einer Integerdivision**. Beide Ausdrücke müssen absolute, direkte Werte sein. Das Ergebnis ist ebenfalls ein absoluter direkter Wert.
SHL	**Logische Linksverschiebung**. Beide Ausdrücke müssen absolute, direkte Werte sein. Das Ergebnis ist ebenfalls ein absoluter direkter Wert.
SHR	**Logische Rechtsverschiebung**. Beide Ausdrücke müssen absolute, direkte Werte sein. Das Ergebnis ist ebenfalls ein absoluter direkter Wert.
NOT	**Bitweise Negation**. Der Ausdruck muß ein absoluter, direkter Wert sein. Das Ergebnis ist ebenfalls ein absoluter direkter Wert.
AND	**Bitweises AND**. Beide Ausdrücke müssen absolute, direkte Werte sein. Das Ergebnis ist ebenfalls ein absoluter direkter Wert.
OR	**Bitweises OR**. Beide Ausdrücke müssen absolute, direkte Werte sein. Das Ergebnis ist ebenfalls ein absoluter direkter Wert.
XOR	**Bitweises exklusives OR**. Beide Ausdrücke müssen absolute, direkte Werte sein. Das Ergebnis ist ebenfalls ein absoluter direkter Wert.

Assembler-Prozeduren und -Funktionen

Mit dem integrierten Assembler können Sie komplette Prozeduren und Funktionen schreiben, für die keine begin...end-Anweisung erforderlich ist:

```
function LongMul(X, Y: Integer): Longint;
asm
   MOV     EAX,X
   IMUL    Y
end;
```

Der Compiler führt für diese Routinen verschiedene Optimierungen durch:

- Der Compiler erstellt keinen Code zum Kopieren von Wert-Parametern in lokale Variablen. Dies betrifft alle Wert-Parameter vom Typ String und alle anderen Wert-Parameter, deren Größe nicht ein, zwei oder vier Byte beträgt. Innerhalb der Routine müssen derartige Parameter als **var**-Parameter behandelt werden.
- Der Compiler weist keine Funktionsergebnis-Variable zu, und eine Referenz auf das Symbol *@Result* ist ein Fehler. Eine Ausnahme bilden Funktionen, die eine Referenz auf einen String, eine Variante oder eine Schnittstelle zurückliefern. Die aufrufende Routine weist diesen Typen immer einen *@Result*-Zeiger zu.
- Der Compiler generiert keinen Stackframe für Routinen, die nicht verschachtelt sind und weder Parameter noch lokale Variablen haben.
- Der automatisch erzeugte Eintritts- und Austrittscode für Routinen sieht folgendermaßen aus:

```
PUSH    EBP             ; Vorhanden, wenn Locals <> 0 oder Params <> 0
MOV     EBP,ESP         ; Vorhanden, wenn Locals <> 0 oder Params <> 0
SUB     ESP,Locals      ; Vorhanden, wenn Locals <> 0
   :
MOV     ESP,EBP         ; Vorhanden, wenn Locals <> 0
POP     EBP             ; Vorhanden, wenn Locals <> 0 oder Params <> 0
RET     Params          ; Immer vorhanden
```

Wenn lokale Variablen Varianten, lange Strings oder Schnittstellen enthalten, werden sie mit Null initialisiert, aber nach der Verarbeitung nicht finalisiert.

- *Locals* ist die Größe der lokalen Variablen, *Params* die Größe der Parameter. Wenn sowohl *Locals* als auch *Params* Null ist, existiert kein Eintrittscode, und der Austrittscode besteht nur aus einer RET-Anweisung.

Assembler-Funktionen liefern ihre Ergebnisse folgendermaßen zurück:

- Ordinale Werte werden in AL (8-Bit-Werte), AX (16-Bit-Werte) oder EAX (32-Bit-Werte) zurückgeliefert.
- Reelle Werte werden in ST(0) über den Register-Stack des Coprozessors zurückgegeben. (*Currency*-Werte werden mit dem Faktor 10000 skaliert.)
- Zeiger einschließlich langer Strings werden in EAX zurückgeliefert.
- Kurze Strings und Varianten werden an die temporären Adresse zurückgegeben, auf die *@Result* zeigt.

Anhang

A

Die Grammatik von Object Pascal

```
Ziel -> (Programm | Package  | Bibliothek  | Unit)

Programm -> [PROGRAM Bezeichner ['('Bezeichnerliste')'] ';']
            Programmblock '.'

Unit -> UNIT Bezeichner ';'
        interface-Abschnitt
        implementation-Abschnitt
        initialization-Abschnitt '.'

Package -> PACKAGE Bezeichner ';'
           [requires-Klausel]
           [contains-Klausel]
           END '.'

Bibliothek -> LIBRARY Bezeichner ';'
              Programmblock '.'

Programmblock -> [uses-Klausel]
                 Block

uses-Klausel -> USES Bezeichnerliste ';'

interface-Abschnitt -> INTERFACE
                       [uses-Klausel]
                       [interface-Deklaration]...

interface-Deklaration -> const-Abschnitt
                         -> type-Abschnitt
                         -> var-Abschnitt
                         -> exported-Kopf

exported-Kopf -> Prozedurkopf ';' [Direktive]
                 -> Funktionskopf ';' [Direktive]

implementation-Abschnitt -> IMPLEMENTATION
                            [uses-Klausel]
                            [Deklarationsabschnitt]...

Block -> [Deklarationsabschnitt]
```

```
      Verbundanweisung

Deklarationsabschnitt -> Label-Deklarationsabschnitt
                      -> const-Abschnitt
                      -> type-Abschnitt
                      -> var-Abschnitt
                      -> Prozedurdeklarationsabschnitt

Label-Deklarationsabschnitt -> LABEL Label-Bezeichner

const-Abschnitt -> CONST (Konstantendeklaration ';')...

Konstantendeklaration -> Bezeichner '=' Konstanter Ausdruck
                       -> Bezeichner ':' Typbezeichner '=' Typisierte Konstante

type-Abschnitt -> TYPE (Typdeklaration ';')...

Typdeklaration -> Bezeichner '=' Typ
               -> Bezeichner '=' Eingeschränkter Typ

Typisierte Konstante -> (Konstanter Ausdruck | Array-Konstante | Record-Konstante)

Array-Konstante -> '(' Typisierte Konstante/','... ')'

Record-Konstante -> '(' Recordfeld-Konstante/';'... ')'

Recordfeld-Konstante -> Bezeichner ':' Typisierte Konstante

Typ -> Typbezeichner
    -> Einfacher Typ
    -> Strukturierter Typ
    -> Zeigertyp
    -> String-Typ
    -> Prozedurtyp
    -> Variantentyp
    -> Klassenreferenztyp

Eingeschränkter Typ -> Objekttyp
                    -> Klassentyp
                    -> Schnittstellentyp

Klassenreferenztyp -> CLASS OF Typbezeichner

EinfacherTyp -> (Ordinaler Typ | Reeller Typ)

Reeller Typ -> REAL48
            -> REAL
            -> SINGLE
            -> DOUBLE
            -> EXTENDED
            -> CURRENCY
            -> COMP

Ordinaler Typ -> (Unterbereichstyp | Aufzählungstyp | Ordinalbezeichner)

Ordinalbezeichner -> SHORTINT
            -> SMALLINT
            -> INTEGER
            -> BYTE
            -> LONGINT
            -> INT64
            -> WORD
            -> BOOLEAN
```

```
         -> CHAR
         -> WIDECHAR
         -> LONGWORD
         -> PCHAR

Variantentyp -> VARIANT
          -> OLEVARIANT

Unterbereichstyp -> Konstanter Ausdruck '..' Konstanter Ausdruck

Aufzählungstyp -> '(' Bezeichnerliste ')'

String-Typ -> STRING
         -> ANSISTRING
         -> WIDESTRING
         -> STRING '[' Konstanter Ausdruck ']'

Strukturierter Typ -> [PACKED] (Array-Typ | Mengentyp | Dateityp | Record-Typ)

Array-Typ -> ARRAY ['[' Ordinaler Typ/','... ']'] OF Typ

Record-Typ -> RECORD [Felderliste] END

Felderliste ->  Felddeklaration/';'... [Variantenabschnitt] [';']

Felddeklaration -> Bezeichnerliste ':' Typ

Variantenabschnitt -> CASE [Bezeichner ':'] Typbezeichner OF Record-Variante/';'...

Record-Variante -> Konstanter Ausdruck/','...  ':' '(' [Felderliste] ')'

Mengentyp -> SET OF Ordinaler Typ

Dateityp -> FILE OF Typbezeichner

Zeigertyp -> '^' Typbezeichner

Prozedurtyp -> (Prozedurkopf | Funktionskopf) [OF OBJECT]

var-Abschnitt -> VAR (Variablendeklaration ';')...

Variablendeklaration -> Bezeichnerliste ':' Typ [(ABSOLUTE (Bezeichner | Konstanter
                        Ausdruck)) | '=' Konstanter Ausdruck]

Ausdruck -> Einfacher Ausdruck [Vergleichsoperator Einfacher Ausdruck]...

Einfacher Ausdruck -> ['+' | '-'] Term [Additiver Operator Term]...

Term -> Faktor [Multiplikativer Operator Faktor]...

Faktor -> Designator ['(' Ausdrucksliste ')']
       -> '' Designator
       -> Zahl
       -> String
       -> NIL
       -> '(' Ausdruck ')'
       -> NOT Faktor
       -> Mengenkonstruktor
       -> Typbezeichner '(' Ausdruck ')'

Vergleichsoperator -> '>'
                   -> '<'
                   -> '<='
                   -> '>='
                   -> '<>'
```

```
                    -> IN
                    -> IS
                    -> AS

Additiver Operator  -> '+'
                    -> '-'
                    -> OR
                    -> XOR

Multiplikativer Operator  -> '*'
                          -> '/'
                          -> DIV
                          -> MOD
                          -> AND
                          -> SHL
                          -> SHR

Designator -> Qualifizierter Bezeichner ['.' Bezeichner | '[' Ausdrucksliste ']' | '^']...

Mengenkonstruktor -> '[' [Mengenelement/','...] ']'

Mengenelement -> Ausdruck ['..' Ausdruck]

Ausdrucksliste -> Ausdruck/','...

Anweisung -> [Label-Bezeichner ':'] [Einfache Anweisung | Strukturierte Anweisung]

Anweisungsliste -> Anweisung/';'...

Einfache Anweisung -> Designator ['(' Ausdrucksliste ')']
                   -> Designator ':=' Ausdruck
                   -> INHERITED
                   -> GOTO Label-Bezeichner

Strukturierte Anweisung -> Verbundanweisung
                -> Bedingte Anweisung
                -> Schleifenanweisung
                -> with-Anweisung

Verbundanweisung -> BEGIN Anweisungsliste END

Bedingte Anweisung -> if-Anweisung
                   -> case-Anweisung

if-Anweisung -> IF Ausdruck THEN Ausdruck [ELSE Ausdruck]

case-Anweisung -> CASE Ausdruck OF case-Selektor/';'... [ELSE Ausdruck] [';'] END

case-Selektor -> case-Label/','... ':' Anweisung

case-Label -> Konstanter Ausdruck ['..' Konstanter Ausdruck]

Schleifenanweisung -> repeat-Anweisung
                   -> while-Anweisung
                   -> for-Anweisung

repeat-Anweisung -> REPEAT Anweisung UNTIL Ausdruck

while-Anweisung -> WHILE Ausdruck DO Anweisung

for-Anweisung -> FOR Qualifizierter Bezeichner ':=' Ausdruck (TO | DOWNTO) Ausdruck DO
                 Anweisung

with-Anweisung -> WITH Bezeichnerliste DO Anweisung
```

Prozedurdeklarationsabschnitt -> Prozedurdeklaration
-> Funktionsdeklaration

Prozedurdeklaration -> Prozedurkopf ';' [Direktive]
Block ';'

Funktionsdeklaration -> Funktionskopf ';' [Direktive]
Block ';'

Funktionskopf -> FUNCTION Bezeichner [Formale Parameter] ':' (Einfacher Typ | STRING)

Prozedurkopf -> PROCEDURE Bezeichner [Formale Parameter]

Formale Parameter -> '(' Formaler Parameter/';'... ')'

Formaler Parameter -> [VAR | CONST | OUT] Parameter

Parameter -> Bezeichnerliste [':' ([ARRAY OF] Einfacher Typ | STRING | FILE]
-> Bezeichner ':' Einfacher Typ '=' Konstanter Ausdruck

Direktive -> CDECL
-> REGISTER
-> DYNAMIC
-> VIRTUAL
-> EXPORT
-> EXTERNAL
-> FAR
-> FORWARD
-> MESSAGE
-> OVERRIDE
-> OVERLOAD
-> PASCAL
-> REINTRODUCE
-> SAFECALL
-> STDCALL

Objekttyp -> OBJECT [Objektvererbung] [Objektfelderliste] [Methodenliste] END

Objektvererbung -> '(' Qualifizierter Bezeichner ')'

Methodenliste -> (Methodenkopf [';' VIRTUAL])/';'...

Methodenkopf -> Prozedurkopf
-> Funktionskopf
-> Konstruktorkopf
-> Destruktorkopf

Konstruktorkopf -> CONSTRUCTOR Bezeichner [Formale Parameter]

Destruktorkopf -> DESTRUCTOR Bezeichner [Formale Parameter]

Objektfelderliste -> (Bezeichnerliste ':' Typ)/';'...

initialization-Abschnitt -> INITIALIZATION Anweisungsliste [FINALIZATION Anweisungsliste] END
-> BEGIN Anweisungsliste END
-> END

Klassentyp -> CLASS [Klassenvererbung]
[Klassenfelderliste]
[Klassenmethodenliste]
[Klasseneigenschaftenliste]
END

Klassenvererbung -> '(' Bezeichnerliste ')'

Klassensichtbarkeit -> [PUBLIC | PROTECTED | PRIVATE | PUBLISHED]

Klassenfelderliste -> (Klassensichtbarkeit Objektfelderliste)/';'...

Klassenmethodenliste -> (Klassensichtbarkeit Methodenliste)/';'...

Klasseneigenschaftenliste -> (Klassensichtbarkeit Eigenschaftenliste ';')...

Eigenschaftenliste -> PROPERTY Bezeichner [Eigenschaftsschnittstelle]
Eigenschaftsbezeichner

Eigenschaftsschnittstelle -> [Eigenschaftsparameterliste] ':' Bezeichner

Eigenschaftsparameterlis te -> '[' (Bezeichnerliste ':' Typbezeichner)/';'... ']'

Eigenschaftsbezeichner -> [INDEX Konstanter Ausdruck]
[READ Ident]
[WRITE Bezeichner]
[STORED Bezeichner | Konstante)]
[(DEFAULT Konstanter Ausdruck) | NODEFAULT]
[IMPLEMENTS Typbezeichner]

Schnittstellentyp -> INTERFACE [Schnittstellenvererbung]
[Klassenmethodenliste]
[Klasseneigenschaftenliste]
END

Schnittstellenvererbung -> '(' Bezeichnerliste ')'

requires-Klausel -> REQUIRES Bezeichnerliste... ';'

contains-Klausel -> CONTAINS Bezeichnerliste... ';'

Bezeichnerliste -> Bezeichner/','...

Qualifizierter Bezeichner -> [Unit-Bezeichner '.'] Bezeichner

Typbezeichner -> [Unit-Bezeichner '.'] <Typbezeichner>

Ident -> <Bezeichner>

ConstExpr -> <Konstanter Ausdruck>

UnitId -> <Unit-Bezeichner>

LabelId -> <Label-Bezeichner>

Number -> <Nummer>

String -> <String>

Index

Symbole

Zahlen

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z