Programmierkurs Delphi

Grundlagen und objektorientierte Programmierung
Anwendungs- und Komponentenentwicklung

zu finden im Regal Programmierung auf
http://de.wikibooks.org

Copyright © 2007 Autor.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

Vorwort ... 5
  Delphi, was ist das? ... 5
  Aufbau des Buches ... 5
    Die Grundlagen ... 5
    Der Schnelleinstieg ... 5
    Die RAD Umgebung ... 5
  Warum Pascal ... 6
  Warum Delphi? ... 6
Pascal ... 6
  Grundlagen ... 6
    Die Konsole ... 6
    Die Programmvorlage ... 7
  Variablen ... 9
  Verschiedene Variablentypen ... 10
    Zeichen-Variablen ... 10
    Ganzzahlige Variablen ... 11
    Wahrheitswerte ... 12
    Initialisierung von Variablen bei der Deklaration ... 13
    Typisierte Konstanten ... 13
  Eingabe und Ausgabe ... 13
    Eingaben erfassen ... 13
    Variablen ausgeben ... 14
  Verzweigungen ... 15
    If-Abfrage (Wenn-Dann-Sonst) ... 15
  Schachtelungen ... 18
  Schleifen ... 19
    Die while-Schleife ... 19
    Die repeat-until-Schleife ... 19
    Die for-Schleife ... 20
    Vorzeitiger Abbruch einer Schleife ... 21
    Überspringen von Werten ... 22
  Arrays ... 23
    Was sind Arrays? ... 23
    Arrays anlegen ... 23
    Auf Arrays zugreifen ... 23
    Dynamische Arrays ... 24
    Mehrdimensionale Arrays ... 26
  Prozeduren und Funktionen ... 27
    Prozeduren ohne Parameter ... 27
    Prozeduren mit Wertübergabe (call by value) ... 28
    Prozeduren mit Variablenübergabe (call by reference) ... 29
    Prozeduren/Funktionen mit Const Parameter ... 30
    Erste Zusammenfassung ... 30
    Funktionen ... 30
    Unterprozeduren / Unterfunktionen ... 31
    forward Deklaration ... 32
    Überladene Prozeduren / Funktionen ... 33
  Fortgeschritten ... 33
    Typdefinition ... 33
    Records ... 35
    Zeiger ... 36

Methodenzeiger 38
Rekursionen 38
Objektorientierung 39
Klassen 39
Konstruktoren und Destruktoren 42
Eigenschaften 46
Vererbung 50
Überladene Methoden 50
Interfaces 51
Exceptions 52
Schnelleinstieg 52
Einstieg 52
Was ist Delphi 52
Warum Delphi? 52
Die Oberfläche 52
Das erste Programm (Hello world) 53
Erweitertes Programm 53
Die Syntax 54
Die Strukturen von Delphi 54
Prozeduren und Funktionen 56
Datentypen (Array, Records und Typen) 56
Arrays 56
Typen 58
Records 58
Pointer 59
Einleitung 59
Grundlagen 59
Dynamische Datenstrukturen 61
Die klassische Methode: Listen 61
Die moderne Methode: Klassen 62
DLL-Programmierung 65
Was ist eine DLL? 65
Das Grundgerüst einer DLL 65
Assembler und Delphi 66
Allgemeines 66
Syntax 67
RAD-Umgebung 68
Warum eine RAD-Umgebung? 68
Erstellung einer graphischen Oberfläche 69
Anhänge 69
Befehlsregister 69
Glossar 69
Lizenz 69
Autoren 69
GNU Free Documentation License 71

Vorwort

Delphi, was ist das?

Gut, so wird wohl keiner fragen, der sich bis hierher durchgeklickt hat, aber trotzdem sollen hier einige Begrifflichkeiten geklärt werden:

Delphi, oder noch korrekter "Object Pascal" bzw. "Delphi Language" (seit Erscheinen von Delphi 7), ist eine Programmiersprache, die auf der in den 70ern entwickelten Sprache "Pascal" basiert. Pascal wurde dabei, wie der Name "Object Pascal" schon vermuten lässt, um die objektorientierte Programmierung (OOP) erweitert.

Für Object Pascal gibt es mehrere Compiler mit integrierter Entwicklungumgebung. Die meisten Programmierer benutzen je nachdem, ob sie Windows oder Linux benutzen die Compiler von Borland "Delphi" oder "Kylix". Ein weiterer Compiler stammt vom Open Source-Project Free Pascal, die Entwicklungsumgebung dazu ist Lazarus.

Aufbau des Buches

Dieses Buch soll sowohl Anfängern den Einstieg in die Programmierung mit Pascal, als auch Umsteigern von anderen Sprachen (z.B.: C) einen schnellen Einstieg in Pascal ermöglichen. Um dies zu gewährleisten, teilt sich das Buch in 3 Bereiche auf:

Die Grundlagen

Dieser Abschnitt richtet sich an totale Programmieranfänger und führt schrittweise und anhand von Beispielen in die Pascal-Programmierung ein. Neben der Einführung in die Sprache Pascal werden grundlegende Prinzipien des allgemeinen Programmierens anhand von praktischen Beispielen vermittelt.

Der Schnelleinstieg

Wer schon eine andere Programmiersprache beherrscht und/oder seine Pascal-Kenntnisse ein bisschen auffrischen will, sollte mit diesem Abschnitt beginnen. Er erläutert alle Features der Sprache und ihrer Syntax, setzt dabei aber grundlegendes Verständnis voraus.

Die RAD Umgebung

In diesem Abschnitt geht es um die Programmierumgebungen Delphi und Lazarus. Es soll ein Einstieg in die Entwicklung graphischer Oberflächen und die damit verbundenen Programmiertechniken gegeben werden, die allerdings das Verständnis von Pascal voraussetzen. Weiterhin werden der Aufbau und die dahinter liegenden Prinzipien der graphischen Klassenbibliotheken (VCL,LCL) erläutert.

Warum Pascal

Pascal wurde speziell zu Lernzwecken entwickelt. Insofern kann Pascal für Programmieranfänger empfohlen werden. Eine der hervorstehensten Eigenschaften von Pascal ist die gute Lesbarkeit des Quellcodes, verglichen mit Programmiersprachen wie z.B. C oder C++. Von der Geschwindigkeit der ausführbaren Programme her zieht Pascal fast mit C++ gleich. Obwohl bereits 1970 der erste Pascal-Compiler verfügbar war, gelang Pascal erst mit Borlands Turbo Pascal Mitte der 80er Jahre der Durchbruch. Seit Turbo Pascal 5.5 gibt es auch Möglichkeiten für die objektorientierte Programmierung. Der bekannteste aktuelle, kostenlose Pascalcompiler **FreePascal** ist in vielen Punkten nicht nur zu Turbo Pascal 7, sondern sogar zu Delphi kompatibel und wird ständig weiterentwickelt. Pascal ist also keine tote Sprache, obwohl dies oft behauptet wird.

Warum Delphi?

Es gibt viele Gründe Delphi zu benutzen. Es gibt aber wahrscheinlich auch genauso viele dagegen. Es ist also mehr Geschmacksache, ob man Delphi lernen will oder nicht. Wenn man allerdings Gründe für Delphi sucht, so fällt sicher zuerst auf, dass Delphi einfach zu erlernen ist, vielleicht nicht einfacher als Basic aber doch viel einfacher als C/C++. Für professionelle Programmierer ist es sicher auch wichtig zu wissen, dass die Entwicklung von eigenen Komponenten unter Delphi einfach zu handhaben ist. Durch die große Delphi-Community mangelt es auch nicht an Funktionen und Komponenten.

Ein besonderer Vorteil von Delphi ist hohe Typsicherheit. Viele Fehler werden also schon beim Kompilieren bemerkt und müssen nicht durch langwieriges Debuggen entdeckt werden.

Erstellt man größere Projekte mit Borlands Delphi Compiler, so ist die Geschwindigkeit beim Kompilieren sicher ein entscheidender Vorteil. Auch die einfache Modularisierung, durch Units, Functions und Procedures ist sicherlich ein Vorteil der Sprache gegenüber einfachen Sprachen wie Basic.

Mit Delphi lässt sich zudem so ziemlich alles entwickeln, abgesehen von Systemtreibern. Dennoch ist es nicht unmöglich teilweise auch sehr hardwarenahe Programme zu entwickeln.

Pascal

Grundlagen

Die Konsole

Obwohl die schnelle Entwicklung grafischer Oberflächen eine der größten Stärken von Delphi ist, eignet sie sich nur bedingt zum Einstieg in die Programmierung, da die GUI-Entwicklung ein eher komplexeres und vielseitiges Thema ist. Außerdem setzt sie ein grobes Verständnis allgemeiner Programmiertechniken (wie z.B.: OOP) voraus. In diesem Buch wird als Einstieg die Konsolenprogrammierung genutzt, um dem Leser auf einfache Art und Weise in die verschiedenen Sprachelemente des Software Engineerings einzuführen. Unter Windows auch

als „DOS-Fenster“ bekannt, stellt eine Konsole grundlegende Methoden zur Ein- und Ausgabe von Daten dar, die heutzutage zwar oft als veraltet gilt, aber dennoch nicht wegzudenken ist.

Die Programmvorlage

Startet man Delphi so öffnet es direkt ein leeres Projekt zur Erstellung eines grafischen Programms. Da wir zunächst aber ein Konsolenprogramm erstellen wollen, müssen wir dieses Projekt schließen und ein anderes erstellen.

Delphi:

Datei->Neu->Weitere...->Konsolen-Anwendung

```
[1]  program Project1;
[2]
[3]  {$APPTYPE CONSOLE}
[4]
[5]  uses
[6]    SysUtils;
[7]
[8]  begin
[9]    { TODO -oUser -cConsole Main : Hier Code einfügen }
[10] end.
```

Dies ist nun die Vorlage für unser erstes Programm. Zeile 1: Jedes Programm beginnt mit dem Schlüsselwort **program** gefolgt von dem Programmnamen (der übrigens identisch mit dem Dateinamen sein muss) gefolgt von einem Semikolon.

Die Zeilen 3-6 sollen uns zunächst nicht interessieren, sie sorgen für die nötigen Rahmenbedingungen die unser Programm benötigt.

In Zeile 8 leitet das **begin** nun den Hauptanweisungsblock ein in dem sich später unser Quelltext befinden wird. In Zeile 10 endet sowohl der Anweisungsblock mit **end.**. Dieser Anweisungsblock wird beim Programmstart ausgeführt und danach beendet sich das Programm wieder.

Zeile 9 enthält nur einen Kommentar, der für uns unwichtig ist und bedenkenlos entfernt werden kann.

Lazarus:

Datei->Neu...->Project->Program

```
 [1] program Project1;
 [2]
 [3] {$mode objfpc}{$H+}
 [4]
 [5] uses
 [6]   {$IFDEF UNIX}{$IFDEF UseCThreads}
 [7]   cthreads,
 [8]   {$ENDIF}{$ENDIF}
 [9]   Classes
[10]   { add your units here };
[11]
```

```
[12] begin
[13] end.
```

Die Zeilen 3-10 sollen uns auch hier zunächst nicht interessieren.

Der Befehl Readln

Delphi stellt mit der Programmvorlage ein zwar funktionierendes aber funktionsloses Programm zur Verfügung. Wenn wir einmal dieses Programm starten (mittels [F9]) so sehen wir im besten Fall für den Bruchteil einer Sekunde ein Konsolenfenster. Denn da unser Programm noch leer ist, wird es sofort beendet. Um dies zu verhindern, fügen wir den Befehl **Readln** ein:

```
begin
  Readln;
end.
```

Readln steht für „read line“ (deutsch: lies Zeile). Das heißt, Readln macht nichts anderes als eine Zeile Text von der Konsole zu lesen. Die Eingabe einer Zeile wird mit der [ENTER]-Taste beendet. Bis die [ENTER]-Taste gedrückt wurde, liest der Readln-Befehl alle Zeichen die man eingibt ein. Also müsste unser Konsolen-Fenster nun solange geöffnet bleiben bis wir die [ENTER]-Taste drücken.

Tipp:

Pascal und Delphi sind *case-**in**sensitive*. Das heißt, dass sie nicht zwischen Groß- und Kleinschreibung unterscheiden. Es spielt also keine Rolle, ob man „program“, „Program“, „PROGRAM“ oder „PrOgRaM“ schreibt. Sinnvoll ist allerdings, sich für eine Schreibweise zu entscheiden, damit der Quelltext lesbar bleibt (siehe StyleGuide (folgt))

Wie man die durch Readln eingelesenen Zeichen verarbeitet, erfahren Sie später.

Die Ausgabe

Nachdem unser Programm nun geöffnet bleibt, damit wir die Ausgaben des selbigen betrachten können, wird es nun Zeit Ausgaben hinzuzufügen. Dazu verwenden wir den Befehl *Writeln*:

```
begin
  Writeln('Hello World');
  Readln;
end.
```

Analog zu Readln steht Writeln für „write line“ (deutsch: schreibe Zeile). Wie sich daher bereits erahnen lässt, führt dieser Befehl dazu, dass der Text „Hello world“ auf der Konsole ausgegeben wird.

Tipp: Die Routine Writeln ist nicht auf Zeichenketten begrenzt, sondern kann alle Arten von Daten aufnehmen. So ergibt z.B.: Writeln(42); dass die Zahl 42

auf der Konsole ausgegeben wird.

Exkurs: Zeichenketten

In Pascal werden Zeichenketten durch ein einfaches Apostroph (') gekennzeichnet und im Allgemeinen als String bezeichnet. Will man allerdings ein ' in seinem String verwenden so ist dies durch '' möglich.

Übersicht:	
Zeichenkette in Pascal	**Entsprechung**
'Hello World'	Hello World
'Hello '' World'	Hello ' World
'Hello '+' World'	Hello World

Näheres siehe Strings.

Nun lassen sich selbstverständlich noch beliebig viele weitere Ausgaben hinzufügen. Allerdings entbehrt die reine Ausgabe von Text auf der Konsole, dem eigentlichen Sinn von Programmen, die ja Informationen verarbeiten können sollen. Genau dies ist also Thema des nächsten Kapitels: Wenn Dann Sonst, aber zunächst beschäftigen wir uns erst mal mit Variablen und Konstanten.

Variablen

Eine Variable ist eine Möglichkeit, Daten innerhalb eines Programms zu speichern und zu verwenden. Eine Variable steht repräsentativ für einen Bereich im Speicher, in dem der Wert der Variablen gespeichert ist. Über den Variablennamen kann dann einfach auf diese Speicherstelle zugegriffen werden. Eine Variable besteht in Delphi also immer aus einem Namen, einem Typ und einer Adresse im Speicher.

Die Größe dieses Bereichs hängt vom Typ der Variablen ab. Es gibt vordefinierte (eingebaute) Variablentypen wie Integer, Real oder String und auch die Möglichkeit selbst neue Variablentypen zu definieren. Variablen müssen deklariert, d.h. bekannt gemacht werden, bevor sie im Programm verwendet werden können. Dies geschieht im sogenannten Deklarationsabschnitt (vor **begin**), der mit dem Schlüsselwort **var** eingeleitet wird:

```
var
  s: string;
begin
  ...
end.
```

Dies würde die Variable **s** vom Typ `String` erzeugen. Nun können wir mittels **s** auf die Zeichenkette zugreifen.

Es ist ebenfalls möglich, mehrere Variablen eines Typs gleichzeitig zu deklarieren, indem die Namen der Variablen mit Kommata getrennt werden.

```
var
  s, s2: string;
  i: Integer;
begin
  ...
end.
```

Dies deklariert sowohl zwei Variablen vom Typ `String` (**s** und **s2**), als auch eine (**i**) vom Typ `Integer`.

Verschiedene Variablentypen

Zeichen-Variablen

Zeichen-Variablen sind nicht, wie der Name vermuten lässt zum Zeichnen da, sondern enthalten **Texte**, d.h. Buchstaben, Zahlen (damit kann man aber nicht rechnen), Sonderzeichen und Ähnliches.

[bearbeiten] Der Variablentyp String

Der Typ String bezeichnet eine Zeichenkette mit variabler Länge. Das heißt, er kann keine Zeichen, ein Zeichen oder auch mehrere beinhalten.

Für Strings wird der Zuweisungsoperator **:=** verwendet, Strings müssen im Quelltext immer von Apostrophen umschlossen werden:

```
s := 'Dies ist ein String';
```

Strings werden von Delphi standardmäßig blau hervorgehoben.

Mehrere Strings lassen sich miteinander mit Hilfe des +-Operators zu einem größeren Text verknüpfen (konkatenieren). Wenn wir beispielsweise an "Dies ist ein String" noch " und wird in s gespeichert." hängen wollen, so können wir das mit dem so genannten Verknüpfungs- oder Verkettungsoperator tun. Dieser Operator ist ein Pluszeichen (+). Doch genug zur Theorie. Unser Beispiel sähe, wenn "Dies ist ein String" schon in s gespeichert wäre, so aus:

```
s := s + ' und wird in s gespeichert.';
```

Um nicht druckbare Zeichen in einem String zu speichern, oder einfach ein Zeichen mit einem bestimmten ASCII-Code einzufügen, kann das #-Zeichen, gefolgt von einer Ganzzahl genutzt werden:

```
s := s + #10; // Linefeed an s anhängen (Dezimale Schreibweise)
s := s + #$20; // Leerzeichen an s anhängen (Hexadezimale Schreibweise)
```

So wird im Prinzip jedes Zeichen im String gespeichert. Dies erklärt auch, warum man mit Zahlen in einem String nicht rechnen kann, obwohl sie im String gespeichert werden können.

Um ein einzelnes Zeichen eines Strings anzusprechen, genügt es, in eckigen Klammern dahinter zu schreiben, welches Zeichen man bearbeiten oder auslesen möchte.

```
Writeln('Der erste Buchstabe des Strings ist ein "'+s[1]+'".');
```

Der Variablentyp Char

Der Typ Char dient dazu, ein einzelnes Zeichen zu speichern. Die Zuweisung funktioniert hierbei genauso wie beim Typ String (allerdings kann natürlich nur ein einzelnes Zeichen zugewiesen werden):

```
var
  c: Char;
begin
  c := 'a'; // c den Buchstaben a zuweisen
end.
```

Auch Steuerzeichen können genauso wie bei Strings verwendet werden:

```
var
  c: Char;
begin
  c := #64; // c das Zeichen mit dem ASCII-Code 64 zuweisen (= '@')
end.
```

Ganzzahlige Variablen

Ganzzahlige Variablen können, wie der Name schon sagt, ganze oder natürliche Zahlen speichern. Der am häufigsten gebräuchliche Typ in dieser Variablenart ist "Integer", was nichts mit dem deutschen Wort "*integer*" zu tun hat. Mit diesen Zahlen lässt sich dann auch rechnen.

Der Variablentyp Integer / Variablentypen zum Umgang mit Ganzzahlen

Eine Variable vom Typ „Integer“ bezeichnet eine 32 Bit große Ganzzahl mit Vorzeichen, es können somit Werte von -2^{31} bis $2^{31}-1$ dargestellt werden.

Variablen werden in Pascal mit dem Zuweisungsoperator := beschrieben:

Das nachfolgende Beispiel zeigt einige Möglichkeiten zur Verwendung von Variablen. Nach jedem Schritt wird mittels Writeln der Name der Variable und danach der Wert der selbigen ausgegeben.

```
var
  i, j: Integer;
begin
  i := 10;
  WriteLn('i ', i);
  j := i;
  WriteLn('j ', j);
  i := 12;
  WriteLn('i ', i);
```

```
  WriteLn('j ', j);
  ReadLn;
end.
```

Ausgabe:

```
i 10
j 10
i 12
j 10
```

Hier wird deutlich, dass bei der Zuweisung einer Variablen an eine andere nur der Wert übertragen (kopiert) wird.

Außerdem kann wie gewohnt mit +, **-** und * gerechnet werden. Eine Division wird für Ganzzahltypen durch **div** dargestellt. Um den Rest einer Division zu erhalten, wird der **mod**-Operator verwendet. Der Compiler beachtet auch Punkt-vor-Strich Rechnung und Klammerung.

```
i := 10;         // i ist jetzt 10
i := i * 3 + 5; // i ist jetzt 35
i := 10 div 3;  // i ist jetzt 3
i := 10 mod 3;  // i ist jetzt 1
```

Weitere ganzzahlige Variablentypen

Name	Wertbereich
Shortint	-128..127
Smallint	-32768..32767
Longint (entspricht **Integer**)	-2^31..2^31-1
Int64	-2^63..2^63-1
Byte	0..255
Word	0..65535
Cardinal/Longword	0..2^32-1

Wahrheitswerte

Der Variablentyp Boolean

Der Typ Boolean ermöglicht es, einen Wahrheitswert zu speichern. Mögliche Werte sind true (wahr) und false (falsch). Auch der Wert kann wiederum über den Zuweisungsoperator der Variable zugewiesen werden:

```
var
  a: Boolean;
begin
  a := True;
end.
```

Auf diesen Variablentyp wird im Kapitel Verzweigungen näher eingegangen, doch erst mal sind jetzt Input und Output dran.

Initialisierung von Variablen bei der Deklaration

Es ist sogar möglich (und auch ziemlich elegant) Variablen schon bei ihrer Deklaration einen Startwert mitzugeben. Damit kann man sich große Initialisierungsorgien nach dem begin ersparen.

```
var
  i: Integer = 42;
begin
      ...
end.
```

Die Initialisierung ist allerdings nur bei globalen Variablen möglich. Lokale Variablen von Prozeduren und Funktionen können dagegen auf diese Weise nicht mit Startwerten belegt werden.

Typisierte Konstanten

Typisierten Konstanten wird, wie der Name schon sagt, ein definierter Typ zugewiesen (statt dass der Typ aus dem Wert entnommen wird). Damit ist es möglich, einer Konstanten z.B. auch Records und Arrays zuzuweisen. Sie werden genau wie initialisierte Variablen definiert:

```
const
  a: Boolean = False;
```

Standardmäßig werden typisierte Konstanten wie normale Konstanten behandelt, d.h. dass diese während der Laufzeit nicht geändert werden können. Man kann Delphi jedoch mit der Compiler-Direktive **{$J+}** anweisen, diese wie Variablen zu behandeln. Free Pascal unterstützt Zuweisungen an typisierte Konstanten ohne weiteres.

Eingabe und Ausgabe

Um mit dem Benutzer zu kommunizieren, muss der Computer die eingegebenen Daten (Input) speichern, verarbeiten und dann später ausgeben (Output). Um das Verarbeiten kümmert sich das nächste Kapitel; in diesem Kapitel dreht sich alles um das Ein- und Ausgeben von Daten.

Eingaben erfassen

Kommen wir nun zu unserem eigentlichen Ziel, dem Einlesen und Verarbeiten von Eingaben auf der Konsole. Unser Ausgangs-Programm:

```
program Eingabeerfassung;

{$APPTYPE CONSOLE}
```

```
uses
  SysUtils;

var
 ...
begin
  ...
  Readln;
end.
```

Nun erweitern wir den Anweisungsblock um die Ausgabe der Nachricht: 'Bitte geben Sie ihren Vornamen ein:'; Dies sollte kein größeres Problem darstellen. Als nächstes legen wir eine Variable an, in der wir den Vornamen des Benutzers speichern wollen. Wir fügen also

```
Vorname: string;
```

in den "Var-Abschnitt" ein. Nun wollen wir den Vornamen von der Konsole einlesen und in der Variable **Vorname** speichern dies geschieht mittels:

```
Readln(Vorname);
```

Vorname wird hier der Methode *Readln* als Parameter übergeben. Da es sich bei der Methode *Readln* um eine sehr spezielle Funktion handelt, gehen wir nicht näher auf dieses Konstrukt ein. Uns genügt die Tatsache, dass sich der in die Konsole eingegebene Text, nach dem Ausführen des Befehls in der Variable **Vorname** befindet.

Variablen ausgeben

Ähnlich der Eingabe mit Readln, kann man mit Writeln Variablen ausgeben. Beispielsweise so:

```
Writeln(Vorname);
```

Nun wird der gerade eingegebene Vorname auf dem Bildschirm ausgegeben.

Schreiben wir jetzt das erste Programm, das aus der Eingabe eine Ausgabe „berechnet“, denn dazu sind Programme im Allgemeinen da.

Unser Programm soll den Benutzer fragen, wie er heißt und ihn dann mit dem Namen begrüßen, und zwar so, dass er die Begrüßung auch sieht.

```
program Eingabeerfassung;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  Vorname: string;

begin
  Writeln('Wie heisst du?');
  Readln(Vorname);
  Writeln('Hallo '+Vorname+'! Schoen, dass du mal vorbeischaust.');
```

```
  Readln;
end.
```

Hier wird deutlich, dass Writeln irgendeinen String benötigt. Dabei ist es egal, ob dieser aus einer Variablen stammt, direkt angegeben wird oder wie hier eine Kombination von beidem darstellt.

Verzweigungen

In diesem Kapitel wollen wir uns mit dem Verarbeiten von Benutzer-Eingaben mittels If-Abfragen beschäftigen.

If-Abfrage (Wenn-Dann-Sonst)

Aufbau

Bisher hat der Computer auf jede Eingabe in gleicher Art reagiert. Nun wollen wir beispielsweise, dass der Computer nur Hugo begrüßt. Dazu brauchen wir eine so genannte "If-Abfrage". Dabei überprüft der Computer, ob ein bestimmter Fall wahr (true) oder falsch (false) ist. Die Syntax sieht, da in Pascal ja alles im Englischen steht, so aus:

```
if Bedingung then Anweisung
```

Die Bedingung ist dabei eine Verknüpfung von Vergleichen, oder ein einzelner Wahrheitswert. Aber dazu später mehr. Erstmal wollen wir uns dem Vergleichen widmen:

Es gibt in Delphi die folgenden sechs Vergleichsoperatoren:

Operator(en)	Diese(r) Operator(en) prüft/en, ob...
=	die beiden Werte gleich sind.
> und <	einer der Werte größer ist (bei Strings die Länge).
>= und <=	einer der Werte größer **oder** gleich ist.
<>	die beiden Werte unterschiedlich sind.

Einfache if-Abfragen

Nun aber erstmal wieder ein Beispiel, bevor wir zu den Verknüpfungen weitergehen:

Wir wollen jetzt nachgucken, ob Hugo seinen Namen eingegeben hat. Dazu setzen wir vor den Befehl "Writeln('Hallo '+Vorname+'! Schoen, dass du mal vorbeischaust.');" die folgende If-Abfrage:

```
if Vorname = 'Hugo' then
```

Der Fall ist hier gegeben, wenn der eingegebene Vorname 'Hugo' ist. Dabei steht Vorname nicht in Apostrophen, da es der Name der Variablen ist. Hugo ist jedoch der Wert, mit dem die Variable verglichen werden soll, so dass hier die Apostrophe zwingend sind.

Das gesamte Programm sieht jetzt also so aus:

```
program Eingabeerfassung;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  Vorname: string;

begin
  Writeln('Wie heisst du?');
  Readln(Vorname);
  if Vorname = 'Hugo' then
    Writeln('Hallo '+Vorname+'! Schoen, dass du mal vorbeischaust.');
  Readln;
end.
```

Erweiterte if-Abfragen mit „sonst“

Wer dieses Programm einmal getestet hat, wird festgestellt haben, dass es keine weitere Ausgabe gibt, wenn der eingegebene Name ungleich „Hugo“ ist. Dies wollen wir nun ändern. Dazu erweitern wir unsere Syntax ein klein wenig:

```
if Bedingung then Anweisung else Anweisung
```

Die auf **else** folgende Anweisung wird dann aufgerufen wenn die Bedingung der if-Abfrage nicht zutrifft (also false ist). Unser so erweitertes Programm sieht nun also in etwa so aus:

```
program Eingabeerfassung;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  Vorname: string;

begin
  Writeln('Wie heisst du?');
  Readln(Vorname);
  if Vorname = 'Hugo' then
    Writeln('Hallo '+Vorname+'! Schoen, dass du mal vorbeischaust.')
  else
    Writeln('Hallo '+Vorname+'! Du bist zwar nicht Hugo, aber trotzdem
willkommen.');
  Readln;
end.
```

Zusammenhang mit dem Typ Boolean

If-Abfragen hängen eng mit dem Typ Boolean zusammen: So können Booleans den Wahrheitswert der Abfrage speichern, indem ihnen die Bedingung „zugewiesen“ wird:

```
var
  IstHugo: Boolean;
  ...
begin
  ...
  IstHugo := Vorname = 'Hugo';
  ...
end.
```

Diese Variable kann jetzt wiederum in if-Abfragen verwendet werden:

```
if IstHugo then
  ...
```

Für Boolean-Variablen ist also kein Vergleich nötig, da sie bereits einen Wahrheitswert darstellen.

Verknüpfung von Bedingungen

Wollen wir jetzt mehrere Bedingungen gleichzeitig abfragen, wäre es natürlich möglich, mehrere if-Abfragen zu schachteln:

```
program Eingabeerfassung2;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  Vorname, Nachname: string;
begin
  Writeln('Wie ist dein Vorname?');
  Readln(Vorname);
  Writeln('Wie ist dein Nachname?');
  Readln(Nachname);
  if Vorname = 'Hugo' then
    if Nachname = 'Boss' then
      Writeln('Hallo '+Vorname+' '+Nachname+'! Schoen, dass du mal
vorbeischaust.');
  Readln;
end.
```

Hier fallen aber gleich mehrere Probleme auf:

- Für komplizierte Abfragen wird dies sehr unübersichtlich
- Wollten wir eine Nachricht an alle abgeben, die nicht „Hugo Boss“ heißen, müssten wir für jede der if-Abfragen einen eigenen else-Zweig erstellen

Um diese Probleme zu lösen, wollen wir die beiden Abfragen in eine einzelne umwandeln. Dazu bietet Delphi den **and**-Operator, mit dem wir zwei Vergleiche mit „und“ verknüpfen können. Alles was wir tun müssen, ist um jeden der Vergleiche Klammern und ein „and“ dazwischen setzen:

```
  ...
  if (Vorname = 'Hugo') and (Nachname = 'Boss') then
    Writeln('Hallo '+Vorname+' '+Nachname+'! Schoen, dass du mal
vorbeischaust.')
  else
    Writeln('Hallo '+Vorname+' '+Nachname+'! Du bist zwar nicht Hugo Boss,
aber trotzdem willkommen.')
  ...
```

and ist nicht der einzige Operator:

Operator	Funktion
and	beide Bedingungen müssen erfüllt sein
or	mindestens eine Bedingung muss erfüllt sein
xor	genau eine Bedingung muss erfüllt sein
not	kehrt das Ergebnis um

Schachtelungen

Bisher haben wir bei unseren Abfragen immer nur einen einzelnen Befehl ausgeführt. Natürlich ist es aber auch möglich, mehrere Befehle nacheinander auszuführen, wenn die Bedingung erfüllt ist.

Dabei wäre es unübersichtlich, für jeden Befehl eine neue If-Abfrage zu starten, wie hier:

```
if IstHugo then Writeln('Hallo Hugo!');
if IstHugo then Writeln('Wie geht es dir?');
```

Man kann die Befehle, die zusammen gehören auch schachteln. Dann ist nur noch eine If-Abfrage nötig.

Die zu schachtelnden Befehle werden dabei zwischen ein **begin** und ein **end** geschrieben. Diese Schachtel wird nun als ein eigenständiger Befehl angesehen. Deshalb muss das **end** oftmals mit einem Semikolon geschrieben werden.

Beispiel:

```
var
  ErsterAufruf: Boolean;
begin
  ...

  if ErsterAufruf then
  begin
    WriteLn('Dies ist der erste Aufruf!');
    ErsterAufruf := False;
  end;

  ...
```

```
end.
```

In diesem Fall wird, sofern "ErsterAufruf" wahr (True) ist, eine Nachricht ausgegeben **und** ErsterAufruf auf False gesetzt. Schachtelungen werden vor allem in Zusammenhang mit if-Abfragen und Schleifen angewandt.

Wie man in dem Beispiel auch erkennen kann, ist es üblich, die geschachtelten Befehle einzurücken, um die Übersichtlichkeit zu verbessern.

Schleifen

Die while-Schleife

Die while-Schleife ermöglicht es, einen Programmcode so oft auszuführen, *solange* eine Bedingung erfüllt ist. Ist die Bedingung schon vor dem ersten Durchlauf nicht erfüllt, wird die Schleife übersprungen.

```
while <Bedingung> do
  <Programmcode>
```

Hierbei wird vor jedem Durchlauf die Bedingung erneut überprüft.

Beispiel:

```
var
  a, b: Integer;
begin
  a := 100;
  b := 100;
  while a >= b do
  begin
    WriteLn('Bitte geben Sie einen Wert < ', b, ' ein.');
    Readln(a);
  end;
end.
```

Die Schleife wird mindestens einmal durchlaufen, da zu Beginn die Bedingung 100 = 100 erfüllt ist.

Die repeat-until-Schleife

Die repeat-until-Schleife ähnelt der while-Schleife. Hier wird die Bedingung jedoch nach jedem Durchlauf überprüft, so dass sie mindestens einmal durchlaufen wird.

```
repeat
  <Programmcode>
until <Bedingung>;
```

Des Weiteren wird die repeat-until-Schleife im Gegensatz zur while-Schleife so lange durchlaufen, *bis* die Bedingung erfüllt ist.

Das obere Beispiel:

```
var
```

```
  a, b: Integer;
begin
  b := 100;
  repeat
    WriteLn('Bitte geben Sie einen Wert < ', b, ' ein.');
    Readln(a);
  until a < b;
end;
```

Hier ist dieser Schleifentyp von Vorteil, da die Initialisierung von `a` entfällt, weil die Schleife mindestens einmal durchlaufen wird.

Ein weiterer Vorteil der repeat-until-Schleife ist, dass sie die einzige Schleife ist, die ohne Schachtelung auskommt, da der Programmcode von **repeat** und **until** bereits eingegrenzt ist. Bei den anderen Schleifen folgt der Programmcode immer der Schleifendeklaration.

Die for-Schleife

Im Gegensatz zu den anderen beiden Schleifentypen basiert die for-Schleife nicht auf einer Bedingung, sondern auf einer bestimmten Anzahl an Wiederholungen.

```
for variable := <untergrenze> to <obergrenze> do
  <Programmcode>
for variable := <obergrenze> downto <untergrenze> do
  <Programmcode>
```

Der Unterschied zwischen den beiden Schleifen liegt darin, dass die erste von unten nach oben, die zweite jedoch von oben nach unten zählt. Falls die Untergrenze größer als die Obergrenze ist, wird jedoch nichts ausgeführt. (Beinahe) derselbe Effekt lässt sich mit einer while-Schleife erreichen:

Beispiele:

```
for i := 0 to 4 do
  WriteLn(i);
Ausgabe:
0
1
2
3
4
for i := 4 downto 0 do
  WriteLn(i);
Ausgabe:
4
3
2
1
0
for i := 4 to 3 do
  WriteLn(i);
Ausgabe:
-keine-
for i := 4 to 4 do
  WriteLn(i);
Ausgabe:
4
```

Es ist auch möglich, als Ober- und/oder Untergrenze Variablen anzugeben:

```
for i := a to b do
  WriteLn(i);
```

Die Schleife verhält sich hierbei genauso wie die Variante mit Konstanten.

Als Variablentyp ist nicht nur Integer erlaubt: for-Schleifen unterstützen alle Ordinalen Typen, also Integer, Aufzählungen und Buchstaben.

Beispiele:

```
var
  i: Integer;
begin
  for i := 0 to 10 do
    WriteLn(i);
end.
var
  c: Char;
begin
  for c := 'a' to 'z' do
    WriteLn(c);
end.
type
  TAmpel = (aRot, aGelb, aGruen);
var
  ampel: TAmpel;
begin
  for ampel := aRot to aGruen do
    WriteLn(Ord(ampel));
end.
```

Im Gegensatz zu while- und repeat-until-Schleifen darf in einer for-Schleife die Bedingungsvariable nicht geändert werden. Daher führt folgender Abschnitt zu einer Fehlermeldung:

```
for i := 0 to 5 do
begin
  i := i - 1;  // <-- Fehler
  WriteLn(i);
end;
```

Vorzeitiger Abbruch einer Schleife

In manchen Fällen ist es hilfreich, dass nicht jeder Schleifendurchgang komplett ausgeführt wird, oder dass die Schleife bereits früher als von der Schleifenbedingung vorgegeben verlassen werden soll. Hierfür stehen die Routinen Continue und Break zur Verfügung.

Beispiel: Sie suchen in einer Liste von Strings eine bestimmte Zeichenfolge:

```
var
  a: array[1..5] of string;
  i: Integer;
  e: Boolean;

begin
```

```
  a[1] := 'Ich';
  a[2] := 'werde';
  a[3] := 'dich';
  a[4] := 'schon';
  a[5] := 'finden!';

  e := False;
  i := 1;

  repeat
    if a[i] = 'dich' then e := True;
    Inc(i);
  until e;

  if i <= 5 then
    WriteLn('Das Wort "dich" wurde an Position ' + IntToStr(i - 1 ) + '
gefunden.')
  else
    WriteLn('Das Wort "dich" wurde nicht gefunden.');
end.
```

In diesem Beispiel steht das gesuchte Wort „dich“ an der Position 3 in der Liste. Um nicht alle Elemente durchzusuchen, haben wir hier eine repeat-until-Schleife verwendet, die dann beendet wird, sobald das Wort gefunden wird.

Einfacher wäre eine for-Schleife. Diese durchsucht jedoch grundsätzlich alle Elemente und gibt dann, wenn sie nicht vorher abgebrochen wird, immer den letzten Zählwert zurück. Um den rechtzeitigen Abbruch zu erreichen, verwenden wir Break:

```
  for i := 1 to 6 do
  begin
    if i = 6 then
      Break;
    if a[i] = 'dich' then
      Break;
  end;
```

Hier benötigen wir weder eine Prüfvariable, noch muss i vorher initialisiert werden. Zuerst wird geprüft, ob wir schon alle Elemente durchsucht haben und gegebenenfalls abgebrochen, falls nicht, erfolgt der Abbruch, wenn das Wort gefunden wurde. In diesem Falle erfolgen die Schleifendurchläufe mit den Werten 4, 5 und 6 gar nicht.

Überspringen von Werten

Ein Schleifendurchlauf kann mit dem Befehl Continue an der entsprechenden Stelle abgebrochen und wieder am Anfang begonnen werden.

Beispiel: Ausgabe aller geraden Zahlen zwischen 1 und 10:

```
var
  i: Integer;

begin
  for i := 1 to 10 do
    if i mod 2 = 0 then
      Writeln(IntToStr(i));
end.
```

oder:

```
var
  i: Integer;

begin
  for i := 1 to 10 do
  begin
    if i mod 2 <> 0 then
      Continue;
    Writeln(IntToStr(i));
  end;
end.
```

Im oberen Beispiel werden die Werte vor der Ausgabe „gefiltert“. Das zweite Beispiel startet hingegen einen neuen Durchlauf, wenn eine Zahl nicht ohne Rest durch 2 teilbar ist. Der Vorteil der zweiten Variante liegt darin, dass mit einer geringeren Verschachtelungsebene gearbeitet werden kann. Die Verarbeitung von i erfolgt noch immer im Programmcode der for-Schleife, während sie beim ersten Beispiel im Programmcode der if-Anweisung erfolgt.

Arrays

Was sind Arrays?

Ein Array ist vereinfacht gesagt, eine **Liste** von Variablen.

Arrays anlegen

Wir wollen eine Gästeliste mit 10 Gästen anfertigen. Bisher hätten wir in etwa folgendes gemacht:

```
var
  gast1, gast2, gast3, gast4, gast5, gast6, gast7, gast8, gast9, gast10:
string;
```

Der Nachteil dieses Verfahrens liegt auf der Hand.

Nun erzeugen wir einfach ein Array vom Datentyp String mit 10 Elementen:

```
var
  gast: array[1..10] of string;
```

Die genaue Struktur der Array Deklaration ist:

```
array [startindex .. endindex] of Datentyp;
```

startindex..endindex ist dabei eine so genannte Bereichsstruktur mit dem wir den Bereich zwischen Startwert und Endwert angeben (Randwerte werden mit eingeschlossen). Näheres siehe Typdefinition. Es ist auch möglich, einen Bereich wie -3..5 anzugeben.

Auf Arrays zugreifen

Um nun auf die einzelnen Elemente zuzugreifen, verwenden wir folgende Syntax:

```
gast[1] := 'Axel Schweiß';
gast[2] := 'Peter Silie';
gast[3] := 'Jack Pot';
...
```

Die Zahl in den eckigen Klammern ist der so genannte Index. Er gibt an, auf welche Variable des Arrays wir zugreifen wollen. Gültige Werte sind hier die Zahlen 1 bis 10. Ein weiterer Vorteil von Arrays ist, dass wir anstatt eines fixen Indexes auch einen ordinalen Datentyp angeben können. Das heißt z.B. eine Integer-Variable. Die Abfrage der Namen von 10 Gästen ließe sich also so sehr einfach implementieren:

```
var
  index: Integer;
  gast: array[1..10] of string;
begin
  for index := 1 to 10 do
  begin
    Writeln('Bitte geben Sie den Namen des ', index, '. Gastes ein:');
    Readln(gast[index]);
  end;
end.
```

Dynamische Arrays

Ändern wir unser Szenario so ab, dass wir eine Gästeliste erstellen wollen, aber nicht wissen, wieviele Gäste diese beinhalten soll. Nun könnten wir zwar ein Array erzeugen, das auf jedenfall groß genug ist um alle Gäste aufzunehmen. Allerdings wäre dies eine Verschwendung von Speicher und nicht gerade effektiv. Hier kommen uns die dynamischen Arrays zu Hilfe. Dabei handelt es sich, wie man vielleicht vermuten kann, um Arrays, deren Länge man zur Laufzeit verändern kann. Erstellt werden sie praktisch genauso wie normale Arrays, nur geben wir diesmal keinen Wertebereich an:

```
var
  gast: array of string;
```

Der Wertebereich eines dynamischen Arrays ist zwar dynamisch, aber er beginnt zwingend immer mit 0. Zu Beginn hat dieser Array die Länge 0, d.h. er beinhaltet momentan keine Variablen.

Länge des Arrays verändern

Nun verändern wir die Länge des Arrays auf 10:

```
SetLength(gast, 10);
```

Unser Array hat nun eine Länge von 10. Das bedeutet, wir können 10 Strings in ihm verstauen. Allerdings hat das höchste Element im Array den Index 9. Das liegt daran, dass das erste Element den Index 0 hat und wir daher mit dem Index 9 schon 10 Elemente zusammen haben.

Nun könnten wir zum Einlesen unserer Gästeliste so vorgehen:

```
var
  index, anzahlgaeste: Integer;
  gast: array of string;
begin
  Writeln('Bitte geben Sie die Anzahl der Gäste ein:');
  Readln(anzahlgaeste);
  SetLength(gast, anzahlgaeste);
  for index := 0 to anzahlgaeste-1 do
  begin
    Writeln('Bitte geben Sie den Namen des ', index, '. Gastes ein:');
    Readln(gast[index]);
  end;
end.
```

Dies würde zwar zum gewünschten Erfolg führen, allerdings benötigen wir so ständig eine weitere Variable, die die Länge unseres Arrays angibt. Um dies zu umgehen, bedienen wir uns der Routinen High() und Low().

Erster und letzter Index

Die Routine High() liefert den höchsten Index des übergeben Arrays zurück:

```
SetLength(gast, 10);
Writeln(High(gast));  // Ausgabe: 9

SetLength(gast, 120);
Writeln(High(gast));  // Ausgabe: 119
```

Die Methode Length() gibt, wie sich vermuten lässt, die Länge des Arrays zurück:

```
SetLength(gast, 10);
Writeln(Length(gast));  // Ausgabe: 10
```

Die Methode Low() liefert den ersten Index des übergebenen Arrays zurück. Bei einem dynamischen Array wäre dies immer 0. Daher benötigt man diese Methode in einem realen Programm eigentlich nicht. Lediglich bei Arrays mit festen Wertebereichen erhält diese Funktion einen tieferen Sinn.

Nun können wir unser Programm ein weiteres bisschen vereinfachen:

```
var
  index, anzahlgaeste: Integer;
  gast: array of string;
begin
  Writeln('Bitte geben Sie die Anzahl der Gäste ein:');
  Readln(anzahlgaeste);
  SetLength(gast, anzahlgaeste);
  for index := 0 to High(gast) do
  begin
    Writeln('Bitte geben Sie den Namen des ', index, '. Gastes ein:');
    Readln(gast[index]);
  end;
end.
```

Der entstehende Vorteil ist hier zugegebener Maßen eher gering, aber in der Praxis erspart man sich so leidige Tipparbeit.

Array freigeben

Da wir beim Erstellen des Arrays Speicher belegt haben, müssen wir diesen noch freigeben. Das geschieht ganz einfach mittels:

```
SetLength(gast, 0);
```

Dabei wird die Länge des Arrays wieder auf 0 gesetzt und er beansprucht so keinen weiteren Platz im Speicher mehr. Dies sollte man allerdings immer dann ausführen, wenn der verwendete Array nicht mehr benötigt wird. Unser finales Programm sieht also so aus:

```
var
  index, anzahlgaeste: Integer;
  gast: array of string;
begin
  Writeln('Bitte geben Sie die Anzahl der Gäste ein:');
  Readln(anzahlgaeste);
  SetLength(gast, anzahlgaeste);
  for index := 0 to High(gast) do
  begin
    Writeln('Bitte geben Sie den Namen des ', index, '. Gastes ein:');
    Readln(gast[index]);
  end;
  SetLength(gast, 0);
end.
```

Mehrdimensionale Arrays

Bis jetzt haben wir uns nur mit eindimensionalen Arrays beschäftigt. Wir haben in Pascal aber auch die Möglichkeit, mehrdimensionale Arrays anzulegen. Z.B. wollen wir nun den Vornamen und den Nachnamen auf unserer Gästeliste getrennt voneinander abspeichern. Dazu erzeugen wir zuerst ein Array mit zwei Elementen, eins für den Vornamen eins für den Nachnamen:

```
type
  TName = array[0..1] of string; // Index 0 = Vorname; 1 = Nachname

var
  gast: array of TName;
```

Und so einfach haben wir einen mehrdimensionalen Array erzeugt. Man kann sich diesen Array nun auch als Tabelle vorstellen. Es gibt eine Spalte für den Vornamen und eine weitere für den Nachnamen. Die Anzahl der Einträge (Zeilen) in dieser Tabelle ist dabei beliebig, da der zweite Array dynamisch ist.

Natürlich können wir das Ganze auch in einer einzelnen Zeile deklarieren:

```
var
  gast: array of array[0..1] of string;
```

Nun wollen wir unsere Gästeliste erneut einlesen:

```
var
  index, anzahlgaeste: Integer;
  gast: array of array[0..1] of string;
begin
```

```
  Writeln('Bitte geben Sie die Anzahl der Gäste ein:');
  Readln(anzahlgaeste);
  SetLength(gast, anzahlgaeste);
  for index := 0 to High(gast) do
  begin
    Writeln('Bitte geben Sie den Vornamen des ', index, '. Gastes ein:');
    Readln(gast[index, 0]);
    Writeln('Bitte geben Sie den Nachnamen des ', index, '. Gastes ein:');
    Readln(gast[index, 1]);
  end;
  SetLength(gast, 0);
end.
```

Wie zu erkennen ist, werden die Indizes innerhalb der Array-Klammern durch ein Komma getrennt.

Prozeduren und Funktionen

Prozeduren ohne Parameter

Oft benötigt man ein und den selben Code mehrere Male in einem Programm, - was liegt also näher als ihn nur ein mal zu schreiben, unter einem Namen zu speichern und diesen dann im Programm aufzurufen? Wohl nichts, und deshalb leisten Prozeduren genau dies. Will man etwa die Anweisungen aus dem Eingangsbeispiel

```
var
  Vorname: string;
begin
  Writeln('Wie heisst du?');
  Readln(Vorname);
  Writeln('Hallo '+Vorname+'! Schoen, dass du mal vorbeischaust.');
  Readln;
end.
```

gleich 2 mal aufrufen und dafür eine Prozedur verwenden, so sähe das Ergebnis so aus:

```
var
  Vorname: string;

procedure Beispielprozedur;
begin
  Writeln('Wie heisst du?');
  Readln(Vorname);
  Writeln('Hallo '+Vorname+'! Schoen, dass du mal vorbeischaust.');
  Readln;
end;  // Ende procedure Beispielprozedur

begin // Beginn des Hauptprogramms
  Beispielprozedur;
  Beispielprozedur;
```

```
end.  // Ende Hauptprogramm
```

Das Schlüsselwort **procedure** leitet dabei den Prozedurkopf ein, der im Beispiel nur aus dem Prozedurnamen besteht. Der Anfang des Codes der Prozedur wird durch **begin** gekennzeichnet, und im Anschluss daran folgt **end;** um das Prozedurende zu kennzeichnen. Der String Vorname ist hierbei global, also am Programmanfang außerhalb der Prozedur deklariert, wodurch er im gesamten Programm samt aller Prozeduren Gültigkeit besitzt, also an jeder Stelle gelesen und beschrieben werden kann. Dies hat jedoch Nachteile und birgt Gefahren, da verschiedene Prozeduren eventuell ungewollt gegenseitig den Inhalt solcher globaler Variablen verändern. Die Folge wären Fehler im Programm, die schwer zu erkennen, aber glücklicherweise leicht zu vermeiden sind. Man kann nämlich auch schreiben:

```
procedure Beispielprozedur;
var
  Vorname: string;
begin
  Writeln('Wie heisst du?');
  Readln(Vorname);
  Writeln('Hallo '+Vorname+'! Schoen, dass du mal vorbeischaust.');
  Readln;
end;  // Ende procedure Beispielprozedur

begin // Beginn des Hauptprogramms
  Beispielprozedur;
  Beispielprozedur;
end.  // Ende Hauptprogramm
```

Jetzt ist die Variable innerhalb der Prozedur deklariert, und sie besitzt auch nur dort Gültigkeit, kann also außerhalb weder gelesen noch beschrieben werden. Man nennt sie daher eine lokale Variable. Mit Erreichen der Stelle `end; // Ende procedure Beispielprozedur` verliert sie ihren Wert, sie ist also auch temporär.

Prozeduren mit Wertübergabe (call by value)

Oft ist es jedoch nötig, der Prozedur beim Aufruf Werte zu übergeben, um Berechnungen auf unterschiedliche Daten anzuwenden oder individuelle Nachrichten auszugeben. Ein Beispiel:

```
var
  MusterName: string;
  GeschoentesAlter: Integer;

procedure NameUndAlterAusgeben(Vorname: string; AlterMonate: Integer);
var
  AlterJahre: Integer;
begin
  AlterJahre := AlterMonate div 12;
  AlterMonate := AlterMonate mod 12;
  Writeln(Vorname+ ' ist '+ IntToStr(AlterJahre)+' Jahre und '+
IntToStr(AlterMonate)+ ' Monate alt');
  Readln;
```

```
end;

begin
  NameUndAlterAusgeben('Konstantin', 1197);
  MusterName := 'Max Mustermann';
  GeschoentesAlter := 386;
  NameUndAlterAusgeben(MusterName, GeschoentesAlter)
end.
```

Der Prozedur NameUndAlterAusgeben werden in runden Klammern zwei durch Semikolon getrennte typisierte Variablen (Vorname: **string**; AlterMonate: Integer) bereitgestellt, die beim Prozeduraufruf mit Werten belegt werden müssen. Dies kann wie im Beispiel demonstriert mittels Konstanten oder auch Variablen des geforderten Typs geschehen. Diese Variablen werden genau wie die unter var deklarierten lokalen Variablen behandelt, nur dass sie eben mit Werten vorbelegt sind. Sie verlieren also am Ende der Prozedur ihre Gültigkeit und sind außerhalb der Prozedur nicht lesbar. Veränderungen an ihnen haben auch keinen Einfluss auf die Variablen mit deren Werten sie belegt wurden, im Beispiel verändert also AlterMonate := AlterMonate **mod** 12; den Wert von GeschoentesAlter nicht.

Prozeduren mit Variablenübergabe (call by reference)

Was aber, wenn man der Prozedur nicht nur Werte mitteilen will, sondern sie dem Hauptprogramm auch ihre Ergebnisse übermitteln soll? In diesem Fall benötigt man im Hauptprogramm eine Variable, deren Werte beim Prozeduraufruf übergeben werden und in die am Ende das Ergebnis geschrieben wird. Außerdem muss man im Prozedurkopf einen Platzhalter definieren, der die Variable aufnehmen kann (eine Referenz darauf bildet). Dies geschieht indem man genau wie bei der Wertübergabe hinter dem Prozedurnamen in runden Klammern eine typisierte Variable angibt, ihr aber das Schlüsselwort var voranstellt. Ein Beispiel:

```
var
  eingabe: Double;

procedure kubik(var zahl: Double);
begin
  zahl := zahl * zahl * zahl;
end;

begin
  eingabe := 2.5;
  kubik(eingabe);

  ...

end.
```

Als Platzhalter dient hier die Variable zahl, ihr wird beim Aufruf der Wert von eingabe übergeben, am Ende wird der berechnete Wert zurückgeliefert, im Beispiel erhält eingabe also den Wert 15.625. Bei der Variablenübergabe darf keine Konstante angegeben werden, da diese ja das Ergebnis nicht aufnehmen könnte, es muss immer eine Variable übergeben werden.

Objekte werden in Delphi immer als Zeiger übergeben. Man kann also auf Eigenschaften und Methoden eines Objekts direkt zugreifen:

```
procedure ClearLabel(label: TLabel);
begin
   label.Caption := ''
end;
```

Prozeduren/Funktionen mit Const Parameter

Parameter können mit **Const** gekennzeichnet sein, um zu verhindern, dass ein Parameter innerhalb einer Prozedur oder Funktion geändert werden kann.

```
function StrSame(const S1, S2: AnsiString): Boolean;
begin
  Result := StrCompare(S1, S2) = 0;
end;
```

Bei String-Parametern ergibt sich ein Geschwindigkeitsvorteil, wenn diese mit ***Const*** *gekennzeichnet werden.*

Erste Zusammenfassung

Prozeduren können ohne Ein- und Ausgabe, mit einer oder mehreren Eingaben und ohne Ausgabe, oder mit Ein- und Ausgabe beliebig vieler Variablen deklariert werden. Wertübergaben und Variablenübergaben können selbstverständlich auch beliebig kombiniert werden. Bei Variablenübergaben ist zu beachten, dass die Prozedur immer den Wert der Variable einliest. Will man sie allein zur Ausgabe von Daten verwenden, darf deren Wert nicht verwendet werden, bevor er nicht innerhalb der Prozedur überschrieben wurde, beispielsweise mit einer Konstanten oder dem Ergebnis einer Rechnung.

Prozeduren können natürlich auch aus anderen Prozeduren oder Funktionen heraus aufgerufen werden.

Werden Variablen im Hauptprogramm deklariert, so nennt man sie global, und ihre Gültigkeit erstreckt sich demnach über das gesamte Programm (genauer: die gesamte Unit); somit besitzen auch Prozeduren Zugriff darauf.
Innerhalb von Prozeduren deklarierte Variablen besitzten nur dort Gültigkeit, man nennt sie daher lokale Variablen.

Aber Vorsicht: Konnte man sich bei Zahlen (vom Typ Integer, Int64, Single, Double, ...) in globalen Variablen noch darauf verlassen, dass sie zu Anfang den Wert null haben, so ist dies in lokalen Variablen nicht mehr der Fall, sie tragen Zufallswerte. Außerdem kann man keine typisierten Konstanten innerhalb von Prozeduren deklarieren.

Funktionen

Funktionen liefern gegenüber Prozeduren immer genau ein Ergebnis. Dieses wird sozusagen an Ort und Stelle geliefert, genau dort wo der Funktionsaufruf im Programm war, steht nach dem Ende ihr Ergebnis. Das hat den Vorteil, dass man mit den Funktionsausdrücken rechnen kann als ob man es bereits mit ihrem Ergebnis zu tun hätte, welches erst zur Laufzeit des Programms berechnet wird.

Sie werden durch das Schlüsselwort function eingeleitet, darauf folgt der Funktionsname, in runden Klammern dahinter ggf. typisierte Variablen zur Wertübergabe, gefolgt von einem Doppelpunkt und dem Ergebnis-Typ. Innerhalb der Funktion dient ihr Name als Ergebnisvariable. Ein Beispiel:

```
var
  ergebnis: Double;

function Kehrwert(zahl: Double): Double;
begin
  Kehrwert := 1/zahl;   // oder: Result := 1/zahl;
end;

begin
  ergebnis := Kehrwert(100)*10; // wird zu   ergebnis := 0.01*10;

  ...

  Kehrwert(100)*10;
end.
```

Der Funktion wird also der (Konstanten-) Wert 100 übergeben, Sie liefert ihr Ergebnis, es wird mit 10 multipliziert, und in ergebnis gespeichert. Der nächste Aufruf bleibt ohne Wirkung, und verdeutlicht, dass man mit Funktionsausdrücken zwar beliebig weiterrechnen kann, aber am Ende das Ergebnis immer speichern oder ausgeben muss, da es sonst verloren geht.

Funktionen erfordern nicht notwendigerweise Wertübergaben. Z.B. wäre es möglich, dass eine Funktion mit Zufallszahlen arbeitet oder ihre Werte aus globalen Variablen oder anderen Funktionen bezieht. Eine weitere Anwendung wäre eine Funktion ähnlich einer normalen Prozedur, die jedoch einen Wert als Fehlermeldung zurückgibt. Tatsächlich gibt es z.B. in der Windows-Programmierung die Funktion GetForegroundWindow, die einen Zeiger auf das aktive Fenster zurückgibt und dafür keine Werte vom Benutzer benötigt.

Unterprozeduren / Unterfunktionen

Prozeduren und ebenso Funktionen können bei ihrer Deklaration auch in einander verschachtelt werden (im Folgenden wird nur noch von Prozeduren gesprochen, alle Aussagen treffen aber auch auf Funktionen zu). Aus dem Hauptprogramm oder aus anderen Prozeduren kann dabei nur die Elternprozedur aufgerufen werden, die Unterprozeduren sind nicht zu sehen. Eltern- und Unterprozeduren können sich jedoch gegenseitig aufrufen.
Die Deklaration dazu an einem Beispiel veranschaulicht sieht folgendermaßen aus:

```
procedure Elternelement;
var
  Test: string;

  procedure Subprozedur;
  begin
    Writeln(Test);
```

```
    Readln(Test);
  end;

...
Beliebig viel weitere Prozeduren
...


begin  // Beginn von Elternelement
  Test := 'Ein langweiliger Standard';
  Subprozedur;
  Writeln(Test);
  Readln;
end;
```

Nach dem Prozedurkopf folgen optional die Variablen der Elternprozedur, dann der Prozedurkopf der Unterprozedur, ihr Rumpf, wenn noch nicht geschehen die Variablen der Elternprozedur und schließlich der Rumpf der Elternprozedur. Sofern die Variablen der Elternprozedur vor den Unterprozeduren deklariert werden, können diese darauf ähnlich einer globalen Variable zugreifen, wodurch deren Einsatz sich dadurch noch weiter reduzieren lässt. Im Zusammenhang mit Rekursionen wird ein solcher Einsatz manchmal angebracht sein.
Unterprozeduren können auch selbst wieder Unterprozeduren haben, die dann nur von ihrem Elternelement aufgerufen werden können, es sind beliebige Verschachtelungen möglich.

forward Deklaration

Alle folgenden Aussagen treffen auch auf Funktionen zu: Eigentlich muss der Code einer Prozedur vor ihrem ersten Aufruf im Programm stehen, manchmal ist dies jedoch eher unpraktisch, etwa wenn man viele Prozeduren alphabetisch anordnen will, oder es ist gar unmöglich, nämlich wenn Prozeduren sich gegenseitig aufrufen. In diesen Fällen hilft die forward Deklaration, die dem Compiler am Programmanfang mitteilt, dass später eine Prozedur unter angegebenen Namen definiert wird. Man schreibt dazu den gesamten Prozedurkopf ganz an den Anfang des Programms, gefolgt von der Anweisung **forward;** . Der Prozedur-Rumpf folgt dann im Implementation-Teil der Unit. Hierbei kann der vollständige Prozedurkopf angegeben werden oder man lässt die Parameter weg. Am Beispiel der Kehrwertfunktion sähe dies so aus:

```
function Kehrwert(zahl: Double): Double; forward;

var
  ergebnis: Double;

function Kehrwert;
begin
  Kehrwert := 1/zahl;
end;

...
```

Überladene Prozeduren / Funktionen

Alle Bisher betrachteten Prozeduren / Funktionen verweigern ihren Aufruf wenn man ihnen nicht genau die Zahl an Werten und Variablen wie in ihrer Deklaration gefordert übergibt. Auch deren Reihenfolge muss beachtet werden.
Will man beispielsweise eine Prozedur mit ein und demselben Verhalten auf unterschiedliche Datentypen anwenden, so wird dies durch die relativ strikte Typisierung von Pascal verhindert. Versucht man also, einer Prozedur die, Integer-Zahlen in Strings wandelt, eine Gleitkommazahl als Eingabe zu übergeben, so erhält man eine Fehlermeldung. Da dies aber eigentlich ganz praktisch wäre, gibt es eine Möglichkeit, dies doch zu tun. Man muss jedoch die Prozedur in allen benötigten Varianten verfassen und diesen den gleichen Namen geben. Damit der Compiler von dieser Mehrfachbelegung des Namens weiß, wird jedem der Prozedurköpfe die Anweisung **overload;** angefügt. Ein Beispiel:

```
procedure NumberToString(zahl: Int64; var Ausgabe: string); overload;
begin
  Ausgabe := IntToStr(zahl);
end;

procedure NumberToString(zahl: Double; var Ausgabe: string); overload;
begin
  Ausgabe := FloatToStr(zahl);
end;

procedure NumberToString(zahl: Double; var Ausgabe: TEdit); overload;
begin
  Ausgabe.Text := FloatToStr(zahl);
end;
```

Die erste Version kann Integer-Werte in Strings wandeln, die zweite wandelt Fließkommawerte, die dritte ebenso, speichert sie dann jedoch in einer Delphi Edit Komponenten.
Wenn Typen zuweisungskompatibel sind, braucht man keine separate Version zu schreiben, so nimmt z.B. Int64 auch Integer (Longint) Werte auf, genau wie Double auch Single Werte aufnehmen kann.
Noch weitergehende Fähigkeiten kann man mit dem Typ Variant realisieren.

Fortgeschritten

Typdefinition

Bei einer Typdefinition wird einem Variablentyp ein neuer Bezeichner zugeordnet. Die Typdefinitionen stehen vor den Variablendefinitionen und werden mit dem Schlüsselwort **Type** eingeleitet. Die Syntax zeigen die folgenden Beispiele.

```
type
  int = Integer;
  THandle = Word;
var
  zaehler: int;
  fensterhandle: THandle;
```

In den obigen Beispielen wird den in Delphi definierten Typen *Integer* sowie *Word* ein zusätzlicher Bezeichner, hier *int* sowie *THandle* gegeben. In Variablendefinitionen können danach die neuen Bezeichner verwendet werden.

Mögliche weitere Typdefinitionen sind

Teilbereiche von Ganzzahlen

```
type
  TZiffer = 0..9;  // Die Zahlen 0 bis 9
  TSchulnote = 1..6;
var
  a, b: TZiffer;
begin
  a := 5;      // zulässig
//  a := 10;   Diese Zuweisung erzeugt beim Compilieren eine
Fehlermeldung!!!
  b := 6;
  a := a + b; // Dies führt bei Laufzeit zu einer Fehlermeldung, da das
              // Ergebnis außerhalb des Wertebereiches von TZiffer
liegt!!!
end;
```

Aufzählungen

```
type
  TWochentag = (Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag,
Sonntag);
  TSchulnote = (sehr_gut, gut, befriedigend, ausreichend, mangelhaft,
ungenuegend);
var
  tag: TWochentag;
begin
  tag := Mittwoch;
  if tag = Sonntag then Writeln('Sonntag');
end;
```

Aufzählungen sind identisch mit Teilbereichen von Ganzzahlen, die mit 0 beginnen. Sie machen aber den Code besser lesbar, wie im letzten Beispiel verdeutlicht.

Records, Arrays

```
type
  TKomplex = record
    RealTeil: Double;     // Single, Double, Extended und Real sind in
                          // Delphi definierte Fließkommazahlen
    ImaginaerTeil: Double;
  end;
  TNotenArray = array[1..6] of Integer;

var
  a, b: TKomplex;
  Noten: TNotenArray;

function Plus(a1, a2: TKomplex): TKomplex;
begin
  Result.RealTeil := a1.RealTeil + a2.RealTeil;
  Result.ImaginaerTeil := a1.ImaginaerTeil + a2.ImaginaerTeil;
end;
```

```
begin
  a.RealTeil := 3;
  a.ImaginaerTeil := 1;
  b := a;
  a := Plus(a, b);
end.
```

Records oder Arrays als Typdefinition haben den Vorteil, dass sie direkt einander zugewiesen werden können (a := b) und dass sie als Parameter sowie Rückgabewert einer Funktion zulässig sind. Vergleiche (if a = b then...) sind allerdings nicht möglich.

Zeiger

```
type
  PWord = ^Word;
  PKomplex = ^TKomplex;  // nach obiger Typdefinition von TKomplex
var
  w: PWord;
begin
  ...
  w^ := 128;
  ...
end.
```

Zeiger als Typdefinion sind ebenfalls als Parameter sowie Rückgabewert von Funktionen zulässig. Da ein Zeiger immer eine Adresse speichert, unabhängig vom Typ, auf den der Zeiger zeigt, kann die Typdefinition auch nach der Definition des Zeigers erfolgen, wie im nachfolgenden Beispiel.

```
type
  PKettenglied = ^TKettenglied;
  TKettenglied = record
    Glied: Integer;
    Naechstes: PKettenglied;
  end;
```

Klassen

Die Deklarierung von Klassen ist ebenfalls eine Typdefinition.

Es wird als guter Stil empfunden, wenn Typdefinitionen, die einen Zeiger (englisch: **P**ointer) definieren, mit **P** beginnen und alle anderen **T**ypdefinitionen mit **T**.

Records

Was sind Records?

Records ermöglichen es, mehrere Variablen zu gruppieren. Dies ist beispielsweise dann hilfreich, wenn oft die gleiche Menge an Variablen benötigt wird, oder eine Menge Variablen logisch zusammengefasst werden soll. Eine weitere Situation in der Records unverzichtbar sind ist, wenn im Programm mehrere Datensätze gespeichert und verwaltet werden sollen, beispielsweise in einem Adressbuch

Wie funktionieren Records?

Um zu unserem Beispiel vom Adressbuch zurückzukommen: Wir wollen alle Daten, also Vorname, Nachname, etc. in einem Record speichern. Dazu legen wir einen neuen Typ **TPerson** an, in dem wir alle Variablen auflisten:

```
type
  TPerson = record
    Vorname: string;
    Nachname: string;
    Anschrift: string;
    TelNr: string;
  end;
```

Wenn jetzt eine Variable vom Typ **TPerson** deklariert wird, enthält diese all diese Variablen:

```
var
  Person: TPerson;
begin
  Person.Vorname := 'Hans';
  Person.Nachname := 'Müller';
  ...
end;
```

Die Variablen im Record verhalten sich genauso wie „normale“ Variablen.

Die with-Anweisung

Falls Sie mit mehreren Record-Feldern nacheinander arbeiten wollen, ist es sehr mühselig, immer den Namen der Variablen vornweg zu schreiben. Diese Aufrufe lassen sich mithilfe der with-Anweisung ebenfalls logisch gruppieren:

```
with Person do
begin
  Vorname := 'Hans';
  Nachname := 'Müller';
  Anschrift := 'Im Himmelsschloss 1, 12345 Wolkenstadt';
  TelNr := '03417/123456';
end;
```

Zeiger

Was sind Zeiger?

Ein Zeiger bzw. Pointer ist eine Variable, die auf einen Speicherbereich des Computers verweist. Zum besseren Verständnis hier ein kleines Beispiel:

Nehmen wir einmal an, wir haben einen Schrank mit 50 verschiedenen Fächern. Jedes Fach ist mit einer Nummer gekennzeichnet, dazu hat er in jedem Fach einen Gegenstand.

Im Sinne von Zeiger in einer Programmiersprache ist der Schrank der Speicherbereich des Computers, welcher dem Programm zur Verfügung gestellt wird. Jedes einzelne Fach des

Schrankes repräsentiert eine Adresse auf diesem Speicher. Auf jedem Speicher kann nun auch eine beliebe Bitreihenfolge abgespeichert werden.

Wozu dienen Zeiger?

Es gibt mehrere Gründe, warum man Zeiger benötigt. Zum einen sind hinter den Kulissen von Pascal sehr viele Zeiger versteckt, die große dynamische Speicherblöcke benötigen. Beispielsweise ist ein String ein Zeiger, wie auch eine Klasse nur ein einfacher Zeiger ist.

In vielen Fällen sind aber auch Zeiger dazu da, um die strikte Typisierung von Pascal zu umgehen. Dazu benötigt man einen untypisierten Zeiger, der nur den Speicherblock enthält, ohne Information dazu, was sich auf dem Speicherblock befindet.

Anm.: Es geht eher darum, dass man ohne Pointer immer einen ganzen Schrank haben muss, obwohl sich darin nur 3 Paar Socken befinden. Ein Zeiger würde eben darauf verweisen, wo genau die Socken zu finden sind und man braucht nicht den ganzen Schrank im Zimmer zu haben. Auf Programmieren zurückübertragen bedeutet dies, dass man nicht wie bei einem Array die Größe des Speichers vorher festlegen muss (-> 50 Schubladen im Schrank), sondern die Größe dynamisch angepasst wird (2 Schubladen bei ein paar Paaren, 50 - und eben nur diese 50 - wenn alles voll ist).

Anwendung

Deklaration

Es gibt zwei Arten, wie man einen Zeiger deklarieren kann. Eine typisierte und eine untypisierte Variante.

```
var
  Zeiger1: ^Integer;   // typisiert
  Zeiger2: Pointer;    // untypisiert
```

Speicheradresse auslesen

Im Code kann man nun entweder die Adresse oder den Inhalt des Speicherblockes auslesen. Das Auslesen der Adresse des Speicherblockes funktioniert gleich bei der un- wie auch bei der typisierten Variante.

```
begin
  Zeiger2 := @Zeiger1;
  Zeiger2 := Addr(Zeiger1);
end;
```

Wie ihr seht, gibt es für das Auslesen der Adresse zwei Varianten. Entweder den @-Operator oder auch mit der Funktion Addr(), wobei zu bemerken ist, dass der @-Operator schneller ist. Hinter einem @-Operator kann jede beliebige Variable stehen, aber nur einem untypisierten Pointer kann jede Adresse jedes Speicherblock zugeordnert werden, ohne auf den Typ zu achten.

Inhalt auslesen

Hier gibt es einen Unterschied zwischen typisierten und untypisierten Zeiger.

```
var
  i, j: Integer;
  p1: ^Integer;
  p2: Pointer;
begin
  i := 1;
  {typisiert}
  p1 := @i;     // dem Pointer wird die Adresse der Integer-Variable
übergeben
  p1^ := p1^+1; // hier wird der Wert um eins erhöht
  j := p1^;     // typisiert: der Variable j wird 2 übergeben
  {untypisiert}
  p2 := @i;     // analog oben
  Integer(p2^) := i+1;
  j := Integer(p2^);
end;
```

Bei einem untypisierten Zeiger muss immer der Typ angegeben werden, welcher aus dem Speicher ausgelesen werden soll. Dies geschieht durch die so genannte Typumwandlung: Typ(Zeiger).

Methodenzeiger

Methodenzeiger werden intern als Doppel-Zeiger gespeichert.

```
type
  TMethod = record
    Code, Data: Pointer;
  end;
```

Bei der Zuweisung an einen Methodenzeiger ist also immer ein Objekt(-zeiger) und eine Methode beteiligt. Über Methodenzeiger werden die in Delphi so nützlichen Ereignisse abgebildet.

Beim Aufruf eines Methodenzeigers springt das Programm über Zeiger *Code* in die Methode. Dabei wird *Data* als der versteckte *Self* Parameter übergeben.

Rekursionen

Definition

Unter Rekursion versteht man die bei Delphi übliche Methode einer Prozedur oder Funktion mit Baumstruktur, sich selbst aufzurufen. Diese Variante ist sehr ressourcensparend.

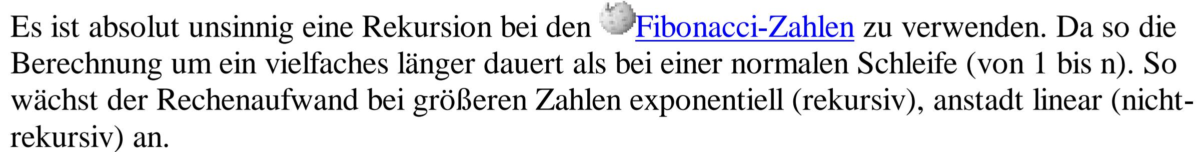
Beispiel einer Rekursionsprozedur

```
function Quersumme(n: Integer): Integer;
begin
  if n = 0 then
    Result := 0
  else
    Result := (n mod 10) + Quersumme(n div 10);
end;
```

Beispiele für Rekursionen

Das wohl berühmteste Beispiel für eine absolut notwendige und elegante Rekursion sind die Türme von Hanoi.

Wann man Rekursionen NICHT verwendet

Es ist absolut unsinnig eine Rekursion bei den Fibonacci-Zahlen zu verwenden. Da so die Berechnung um ein vielfaches länger dauert als bei einer normalen Schleife (von 1 bis n). So wächst der Rechenaufwand bei größeren Zahlen exponentiell (rekursiv), anstadt linear (nicht-rekursiv) an.

Überhaupt sind Rekursive Algorithmen zwar sehr elegant, aber wenn es nicht-rekursive Algorithmen gibt, sind diese meistens schneller und einfacher zu programmieren.
Deshalb sollten rekursive Algothmen, wenn es nicht dringend erforderlich oder sinnvoll ist, nicht benutzt werden.

Links

Wikipedia Artikel zu rekursiver Programmierung: Rekursive Programmierung

Objektorientierung

Klassen

Einleitung

Die Grundlage für die objektorientierte Programmierung, kurz OOP, bilden Klassen. Diese kapseln, ähnlich wie Records, verschiedene Daten zu einer Struktur. Gleichzeitig liefern sie Methoden mit, welche die vorhandenen Daten bearbeiten.

Der Vorteil von Klassen besteht also darin, dass man mit ihnen zusammengehörige Variablen, Funktionen und Prozeduren zusammenfassen kann. Weiterhin können - bei entsprechender Implementation - die Daten einer Klasse nicht „von außen“ geändert werden.

Aufbau einer Klasse

Allgemeiner Aufbau

Die einfachste Klasse entspricht in ihrem Aufbau einem Record:

```
program Klassentest;

type
  TMyRec = record
    EinByte: Byte;
    EinString: string;
  end;
```

```
  TMyClass = class
    FEinByte: Byte;
    FEinString: string;
  end;

var
  MyRec: TMyRec;
  MyClass: TMyClass;

begin
  MyRec.EinByte := 15;
  MyClass := TMyClass.Create;
  MyClass.FEinString := 'Hallo Welt!';
  MyClass.Free;    // bzw. MyClass.Destroy;
end.
```

Hierbei kann man bereits einen Unterschied zu Records erkennen: Während man jederzeit auf die Daten eines Records zugreifen kann, muss bei einer Klasse zunächst Speicher angefordert und zum Schluss wieder freigeben werden. Die speziellen Methoden `Create` und `Destroy` sind in jeder Klasse enthalten und müssen nicht gesondert programmiert werden. Hierzu mehr im Kapitel Konstruktoren und Destruktoren.
Es sollte immer Free anstatt Destroy verwendet werden. Free prüft, ob überhaupt die Klasse initialisiert wurde, was zu keiner Exception führen kann.

Weiterhin hat es sich durchgesetzt, die Variablen einer Klasse, Felder genannt, immer mit dem Buchstaben F zu beginnen. So werden Verwechslungen mit globalen und lokalen Variablen vermieden.

Sichtbarkeit der Daten

Im oben gezeigten Beispiel kann auf die Felder wie bei einem Record zugegriffen werden. Dies sollte man unter allen Umständen vermeiden!

Hierfür gibt es eine ganze Reihe von Möglichkeiten. Zunächst einmal können Felder, sowie Methoden und Eigenschaften einer Klasse nach außen hin „versteckt“ werden. Das erreicht man mit folgenden Schlüsselwörtern:

- **`private`** - auf diese Daten kann außerhalb der Klasse nicht zugegriffen werden
- **`protected`** - hierauf kann man nur innerhalb desselben Packages zugreifen
- **`public`** - diese Daten sind uneingeschränkt zugänglich
- **`published`** - zusätzlich zu public können diese Daten auch im Objekt-Inspektor von Delphi™ bearbeitet werden (nur bei Eigenschaften von Komponenten sinnvoll).

Das entsprechende Schlüsselwort wird der Gruppe von Daten vorangestellt, für die diese Sichtbarkeit gelten soll. Um die Felder unserer Klasse zu verstecken, schreiben wir also:

```
type
  TMyClass = class
  private
    FEinByte: Byte;
    FEinString: string;
  end;
```

Wenn man jetzt versucht, wie oben gezeigt, einem Feld einen Wert zuzuweisen oder ihn auszulesen, wird bereits bei der Kompilierung eine Fehlermeldung ausgegeben.

Methoden

Wie erhält man nun aber Zugriff auf die Daten? Dies erreicht man über öffentlich zugängliche Methoden, mit denen die Daten ausgelesen und geändert werden können.

Eine Methode ist eine fest mit der Klasse verbundene Funktion oder Prozedur. Daher wird sie auch wie Felder direkt innerhalb der Klasse definiert:

```
TMyClass = class
public
  function GetString: string;
  procedure SetString(NewStr: string);
end;
```

Die Ausführung der Methoden erfolgt direkt über die Variable dieses Klassentyps:

```
var
  MyClass: TMyClass;

...

MyClass.SetString('Hallo Welt!');
WriteLn(MyClass.GetString);
```

In der Typdefinition werden nur der Name und die Parameter von Methoden definiert. Die Implementation, also die Umsetzung dessen, was eine Methode tun soll, erfolgt außerhalb der Klasse, genau wie globale Funktionen und Prozeduren. Allerdings muss der Klassenname vorangestellt werden:

```
function TMyClass.GetString: string;
begin
  Result := FEinString;
end;

procedure TMyClass.SetString(NewStr: string);
begin
  FEinString := NewStr;
end;
```

Da die Methoden GetString und SetString Mitglieder der Klasse sind, können diese auf das private Feld FEinString zugreifen.

Eigenschaften

Für unser Beispiel erscheint der Zugriff etwas umständlich. Daher gibt es noch eine andere einfache Möglichkeit, Daten einer Klasse auszulesen und zu ändern: Eigenschaften (engl. *properties*).

Eigenschaften lassen sich wie Variablen behandeln, das heißt, man kann sie (wenn gewünscht) auslesen oder (wenn gewünscht) ändern. Die interne Umsetzung bleibt dabei verborgen. So kann eine Eigenschaft zum Beispiel direkt auf ein Feld oder über den Umweg einer Methode darauf zugreifen:

```
type
  TMyClass = class
```

```
  private
    FEinInt: Integer;
    function HoleDoppelteZahl: Integer;
    procedure SpeichereHalbeZahl(DoppZahl: Integer);    // Name des
Parameters ist egal
  public
    property Zahl: Integer read FEinInt write FEinInt;
    property DoppelZahl: Integer read HoleDoppelteZahl write
SpeichereHalbeZahl;
  end;
```

Sowohl das Feld als auch die Methoden sind versteckt. Die einzige Verbindung zur Außenwelt besteht über die Eigenschaften. Hier greift die Eigenschaft „Zahl" beim Lesen und Schreiben direkt auf das Feld zu, während „DoppelZahl" in beiden Fällen auf Methoden zurückgreift. Da DoppelZahl immer das Doppelte von Zahl sein soll (so sagt es zumindest der Name), werden die Methoden wie folgt implementiert:

```
function HoleDoppelteZahl: Integer;
begin
  Result := FEinInt * 2;
end;

procedure SpeichereHalbeZahl(DoppZahl: Integer);
begin
  FEinInt := DoppZahl div 2;
end;
```

Diese Klasse führt Berechnungen durch, ohne dass es der Benutzer mitbekommt. Weist man Zahl einen Wert zu, erhält man aus DoppelZahl automatisch das Doppelte. Andersherum kann man DoppelZahl einen Wert zuweisen und erhält mit Zahl die Hälfte. Von außen betrachtet erscheinen beide Eigenschaften jedoch wie Felder:

```
with MyClass do
begin
  Zahl := 4;
  WriteLn(DoppelZahl);  // Schreibt eine 8
  DoppelZahl := 20;
  WriteLn(Zahl);        // ergibt 10 und nicht etwa 4
end;
```

Soll DoppelZahl einen Schreibschutz erhalten, lässt man in der Definition den Teil „write ..." weg. Damit kann auch die entsprechende Methode SpeichereHalbeZahl entfernt werden.

Die verschiedenen Arten von Eigenschaften werden ausführlicher in einem eigenen Kapitel behandelt.

Konstruktoren und Destruktoren

Allgemeines

Wie bereits im Kapitel über Klassen beschrieben, muss der Speicher für die Datenstruktur einer Klasse angefordert werden, bevor man mit ihr arbeiten kann. Da eine Klasse mehr als nur eine Datenansammlung ist - nämlich ein sich selbst verwaltendes Objekt - müssen gegebenenfalls weitere Initialisierungen durchgeführt werden. Genauso kann nach der Arbeit

mit einer Klasse noch Speicher belegt sein, der von Delphi's Speicherverwaltung nicht erfasst wird.

Für zusätzliche anfängliche Initialisierungen stehen die Konstruktoren („Errichter“) zur Verfügung, wogegen die Destruktoren („Zerstörer“) die abschließende Aufräumarbeit übernehmen. Diese sind spezielle Methoden einer Klasse, die nur zu diesem Zweck existieren.

Konstruktoren

Für die Arbeit mit einfachen Klassen, die keine weitere Initialisierung benötigen, braucht kein spezieller Konstruktor verwendet werden. In Delphi stammt jede Klasse automatisch von der Basisklasse TObject ab und erbt von dieser den Konstruktor Create. Folgender Aufruf ist daher gültig, obwohl keine Methode Create definiert wurde:

```
type
  TErsteKlasse = class
  end;

var
  Versuch: TErsteKlasse;

begin
  Versuch := TErsteKlasse.Create;
end.
```

Wie zu sehen ist, erfolgt der Aufruf des Konstruktors über den Klassentyp. Eine Anweisung wie `Variable.Create` führt zu einer Fehlermeldung.

Das „einfache“ Create erzeugt ein Objekt des Typs TErsteKlasse und gibt einen Zeiger darauf zurück.

Verwendet man Klassen, welche die Anfangswerte ihrer Felder einstellen oder z.B. weiteren Speicher anfordern müssen, so verdeckt man die geerbte Methode mit seiner eigenen:

```
type
  TErsteKlasse = class
    constructor Create;
  end;

constructor TErsteKlasse.Create;
begin
  inherited;
  { Eigene Anweisungen }
end;
```

Das Schlüsselwort **inherited** ruft dabei die verdeckte Methode TObject.Create auf und sorgt dafür, dass alle notwendigen Initialisierungen durchgeführt werden. Daher muss dieses Schlüsselwort *immer* zu Beginn eines Konstruktors stehen.

Beispiel: Sie wollen eine dynamische Adressliste schreiben, bei der der Name Ihrer Freundin immer als erster Eintrag erscheint.

```
type
  TAdresse = record
    Vorname, Nachname: string;
```

```
    Anschrift: string;
    TelNr: string;
  end;

  TAdressListe = class
    FListe: array of TAdresse;
    constructor Create;
  end;
```

In diesem Falle erstellen Sie den Konstruktor wie folgt:

```
constructor TAdressListe.Create;
begin
  inherited;
  SetLength(FListe, 1);  // Speicher für 1. Eintrag anfordern
  FListe[0].Vorname   := 'Barbie';
  FListe[0].Nachname  := 'Löckchen';
  FListe[0].Anschrift := 'Puppenstube 1, Kinderzimmer';
  FListe[0].TelNr     := '0800-BARBIE';
end;
```

Wenn Sie jetzt eine Variable mit diesem Konstruktor erstellen, enthält diese automatisch immer den Namen der Freundin:

```
var
  Adressen: TAdressListe;

begin
  Adressen := TAdressListe.Create;
  with Adressen.FListe[0] do
    {Ausgabe: "Barbie Löckchen wohnt in Puppenstube 1, Kinderzimmer und ist
erreichbar unter 0800-BARBIE."}
    Writeln(Vorname+' '+Nachname+' wohnt in '+Anschrift+' und ist
erreichbar unter '+TelNr+'.')
end.
```

Destruktoren

Destruktoren dienen, wie schon oben beschrieben dazu, den von einer Klasse verwendeten Speicher wieder freizugeben. Auch hierfür stellt die Basisklasse TObjekt bereits den Destruktor Destroy bereit. Dieser gibt grundsätzlich zwar den Speicher einer Klasse wieder frei, aber nur den des Grundgerüstes. Wenn eine Klasse zusätzlichen Speicher anfordert, z.B. weil sie mit Zeigern arbeitet, wird nur der Speicher für den Zeiger freigegeben, nicht aber der Speicherplatz, den die Daten belegen.

Es sollte jedoch immer Free anstatt Destroy benutzt werden, da Free noch prüft, ob überhaupt die Klasse initialisiert wurde und Free so zu keiner Exception führen kann; im Gegensatz zu Destroy.

Eine Klasse sollte daher immer dafür sorgen, dass keine Datenreste im Speicher zurückbleiben. Dies erfolgt durch Überschreiben des Destruktors TObject.Destroy mit einem eigenen:

```
type
  TZweiteKlasse = class
    destructor Destroy; override;
  end;
```

```
destructor TZweiteKlasse.Destroy;
begin
  { Anweisungen }
  inherited;
end;

var
  Versuch: TZweiteKlasse;

begin
  Versuch := TZweiteKlasse.Create;
  Versuch.Destroy
end.
```

Eine eigene Implementation von Destroy muss immer mit dem Schlüsselword **override** versehen werden, da Destroy eine virtuelle Methode ist (mehr dazu unter Virtuelle Methoden). Auch hier kommt wieder das Schlüsselwort **inherited** zum Einsatz, welches den verdeckten Destruktor TObject.Destroy aufruft. Da dieser Speicher frei gibt, muss inherited innerhalb eines Destruktors immer zuletzt aufgerufen werden.

Wenn wir unser Beispiel der Adressliste erweitern und auf das Wesentliche reduzieren, ergibt sich folgendes Programm:

```
type
  TAdresse = record
    Vorname, Nachname: string;
    Anschrift: string;
    TelNr: string;
  end;

  TAdressListe = class
    FListe: array of TAdresse;
    destructor Destroy; override;
  end;

destructor TAdressListe.Destroy;
begin
  SetLength(FListe, 0);  // Speicher der dynamischen Liste freigeben
  inherited;             // Objekt auflösen
end;

var
  Adressen: TAdressListe;

begin
  Adressen := TAdressListe.Create;
  { Anweisungen }
  Adressen.Destroy;
end.
```

Überladen

Das Überladen von Konstruktoren und Destruktoren ist genauso möglich wie bei anderen Methoden auch. Für das Beispiel unserer Adressliste könnten zum Beispiel zwei Konstruktoren bestehen: einer, der eine leere Liste erzeugt, und einer, der eine Liste aus einer Datei lädt. Genauso könnte ein Destruktor die Liste einfach verwerfen, während ein anderer die Liste vorher in einer Datei speichert.

Sie können die Konstruktoren und Destruktoren jeweils gegenseitig aufrufen. Sie sollten hierbei jedoch darauf achten, dass nur *genau einer* das Schlüsselwort inherited aufruft.

```
type
  TAdressListe = class
    constructor Create; overload;
    constructor Create(Dateiname: string); overload;
    procedure DatenLaden(Dateiname: string);
  end;

procedure TAdressListe.DateiLaden(Dateiname: string);
begin
  {...}
end;

constructor TAdressListe.Create;
begin
  inherited;
  {...}
end;

constructor TAdressListe.Create(Dateiname: string);
begin
  Create;  // ruft TAdressListe.Create auf
  Dateiladen(Dateiname);
end;
```

Weiteres zum Thema Überladen im Kapitel „Überladene Methoden“

Eigenschaften

Einleitung

Objekte haben Eigenschaften. Dies ist eine Regel, die in der freien Natur wie auch in der objektorientierten Programmierung unter Delphi zutrifft.

Eigenschaften stellen einen Zwischenweg zwischen Feldern und Methoden einer Klasse dar. So kann man zum Beispiel beim Auslesen einer Eigenschaft einfach den Wert eines Feldes zurück erhalten, löst aber beim Ändern dieser Eigenschaft eine Reihe von Folgeaktionen aus. Dies liegt daran, dass man bei Eigenschaften genau festlegen kann, was im jeweiligen Falle passieren soll. Ebenso kann jeweils das Lesen oder das Schreiben einer Eigenschaft unterbunden werden.

Wie man bereits sieht, kann ein Programm über Eigenschaften sicherer und gleichzeitig flexibler gestaltet werden.

Eigenschaften definieren

In der Klassendefinition legt man Eigenschaften mit folgender Syntax an:

```
property Name: <Typ> [read <Feld oder Methode>] [write <Feld oder
Methode>];
```

Die Schlüsselwörter **read** und **write** sind so genannte Zugriffsbezeichner. Sie sind beide optional, wobei jedoch mindestens einer von beiden angegeben werden muss. Die folgenden Beispiele sind alle gültig:

```
property Eigenschaft1: Integer read FWert;
property Eigenschaft2: Integer write ZahlSetzen;
property Eigenschaft3: string read FText write TextSetzen;
```

Zur kurzen Erklärung:

?? mit Zugriff auf Eigenschaft1 wird der Wert des Feldes FWert zurück gegeben, eine Zuweisung an Eigenschaft1 ist jedoch nicht möglich (Nur-Lesen-Eigenschaft)
?? Eigenschaft2 dürfen nur Werte zugewiesen werden, diese werden dann an die Methode ZahlSetzen weitergereicht. Das Auslesen des Wertes ist über Eigenschaft2 nicht möglich (Nur-Schreiben-Eigenschaft)
?? Eigenschaft3 ermöglicht hingegen sowohl das Schreiben als auch das Lesen von Daten (Lesen-Schreiben-Eigenschaft)

Im Normalfall wird man überwiegend Lese-Schreib-Eigenschaften verwenden und ggf. einige Nur-Lesen-Eigenschaften. Nur-Schreib-Eigenschaften sind hingegen sehr selten anzutreffen, können aber im Einzelfall einen ebenso nützlichen Dienst erweisen.

Verwendet man für den Zugriff ein Feld, muss dieses lediglich den gleichen Typ haben wie die Eigenschaft. Sollen jedoch Methoden beim Zugriff zum Einsatz kommen, müssen diese bestimmten Regeln folgen: Für den Lesezugriff benötigt man eine Funktion ohne Parameter, deren Ergebnis vom Typ der Eigenschaft ist. Beim Schreibzugriff muss eine Prozedur verwendet werden, deren einziger Parameter den gleichen Typ wie die Eigenschaft besitzt.

Gut, das alles hört sich etwas kompliziert an. Gestalten wir doch mal ein kleines Gerüst für die Verwaltung eines Gebrauchtwagenhändlers:

```
type
  TFarbe = (fSchwarz, fWeiß, fSilber, fBlau, fGruen, fRot, fGelb);
  TAutoTyp = (atKlein, atMittelStufe, atMittelFliess, atKombi, atLuxus,
atGelaende);

  TAuto = class
  private
    FFarbe: TFarbe;
    FTyp: TAutoTyp;
    procedure FarbeSetzen(Farbe: TFarbe);
    function PreisBerechnen: Single;
  public
    property Farbe: TFarbe read FFarbe write FarbeSetzen;
    property Typ: TAutoTyp read FTyp;
    property Preis: Single read PreisBerechnen;
    constructor Create(Farbe: TFarbe; Typ: TAutoTyp);
  end;
```

Wie man bereits ohne das vollständige Programm erkennen kann, haben wir mit dieser Klasse ähnliche Möglichkeiten wie in der realen Welt. Zum Beispiel ist es dem Händler möglich, das Auto umzulackieren. Hierbei wird eine Methode aufgerufen, die möglicherweise Folgeaktionen auslöst (z.B. das Auto für eine Weile als „nicht im Bestand" markiert). Andererseits kann der Händler zwar abrufen, um welchen Autotyp es sich handelt, kann diesen aber selbstverständlich nicht verändern. Der Preis ist nicht starr festgelegt, sondern

wird über eine Funktion berechnet (die sich wahrscheinlich unter anderem an der Farbe und am Typ des Autos orientiert).

Weiterhin ist diese Klasse so angelegt, dass man „von außen“ nur Zugriff auf die Eigenschaften und den Konstruktor von TAuto hat. Die Felder und Methoden dienen nur der internen Verwaltung der Eigenschaften und sind daher im private-Abschnitt verborgen.

Array-Eigenschaften

Eigenschaften können auch als Arrays definiert werden. Dies ist immer dann sinnvoll, wenn man Daten in Form einer Tabelle bzw. Auflistung speichern möchte. Der Zugriff erfolgt wie bei Variablen über Indizes in eckigen Klammern. Hinter den Zugriffsbezeichnern darf bei Array-Eigenschaften allerdings kein Feld angegeben werden, hier sind nur Methoden zulässig:

```
property Zeile[Nr: Integer]: string read HoleZeile write SchreibeZeile;
property Punkt[X, Y: Integer]: Integer read HolePunkt;
```

Die zugehörigen Methoden werden dann wie folgt definiert:

```
function HoleZeile(Nr: Integer): string;
procedure SchreibeZeile(Nr: Integer; Text: string);
function HolePunkt(X, Y: Integer): Integer;
```

Als erste Parameter werden also immer die Felder des Arrays angesprochen, dann folgen die Parameter analog den einfachen Eigenschaften. Die Bezeichner der Variablen müssen mit den Indizes der Eigenschaften nicht notwendigerweise übereinstimmen. Um die Anwendung jedoch übersichtlich zu gestalten, sollte man jedoch gleiche Bezeichner verwenden. Grundsätzlich wäre also auch folgende Notation gültig:

```
function HoleZeile(ZeilenNr: Integer): string;
property Zeile[X: Integer]: string read HoleZeile;
```

Überschreiben

Sichtbarkeit ändern

Bei der Vererbung von Klassen kann es vorkommen, dass man Eigenschaften mit der Sichtbarkeit **protected** erbt, diese aber in der eigenen Klasse als **public** kennzeichnen möchte. Hierfür ist es möglich, die Eigenschaften mit der gewünschten Sichtbarkeit neu zu definieren.

Beispiel: Wir haben von der Klasse TBasis die als **private** gekennzeichnete Eigenschaft „Name“ geerbt. Um diese nun allgemein zugänglich zu machen, schreiben wir einfach:

```
public
  property Name;
```

Damit wurde an der Funktionsweise der Eigenschaft nichts geändert, sie ist jetzt lediglich „besser sichtbar“. Damit ist der Programmcode bereits vollständig. Es müssen keine weiteren Zugriffsfelder oder -methoden hierfür geschrieben werden.

Auf diese Weise ist es ebenfalls möglich, öffentlich zugängliche Eigenschaften als privat zu kennzeichnen.

Neu definieren

Wenn Sie nun von einer Klasse eine Eigenschaft vererbt bekommen, die Sie anders verarbeiten wollen, können Sie diese wie eine gewöhnliche Eigenschaft mit dem gleichen Namen neu definieren. Hierzu sind ebenfalls die Angaben der Zugriffsbezeichner erforderlich. So lässt sich zum Beispiel eine Lese-Schreib-Eigenschaft als Nur-Lesen-Eigenschaft neu definieren:

```
type
  TBasis = class
    FFarbe: string;
    property Farbe: string read FFarbe write FFarbe;
  end;

  TErbe = class(TBasis)
    property Farbe: string read FFarbe;
  end;
```

„FFarbe“ muss in der Klasse „TErbe“ nicht neu eingeführt werden, es wird das Feld aus der Basis-Klasse verwendet. Falls ein Zugriffsbezeichner in der Basisklasse als **private** gekennzeichnet wurde, gibt Delphi bei der Kompilierung eine entsprechende Warnung aus. Für reine Delphi-Programme kann diese Warnung ignoriert werden. Sie ist für die Portierung des Programms in andere Programmiersprachen gedacht, da dort eine solche Schreibweise unter Umständen nicht erlaubt ist.

Falls eine Eigenschaft eine Methode für den Zugriff verwendet und Sie das Verhalten ändern wollen, müssen Sie lediglich die geerbte Methode überschreiben. Mehr hierzu im nächsten Kapitel „Vererbung“.

Zugriff auf Eigenschaften

Der Zugriff auf Eigenschaften erfolgt wie bei Feldern und Methoden:

```
Objekt.Eigenschaft := Wert;
Wert := Objekt.Eigenschaft;
```

Hierbei ist jedoch zusätzlich zur Sichtbarkeit auch die Zugriffsmöglichkeit der Eigenschaft zu beachten, das heißt, ob Sie eine Nur-Lesen-, Nur-Schreiben- oder eine Lese-Schreib-Eigenschaft verwenden wollen. Die Eigenschaften verhalten sich wie eine Mischung aus Variablen und Methoden, z.B. kann man Werte auslesen und übergeben. Nicht möglich ist dagegen die Übergabe einer Eigenschaft als Variablen-Parameter einer Funktion. So erzeugt dieser Programmschnipsel eine Fehlermeldung:

```
type
  TKlasse = class
  private
    FZahl: Integer;
  public
    property Zahl: Integer read FZahl write FZahl;
  end;

var
  Test: TKlasse;

begin
  Test := TKlasse.Create;
```

```
  Test.Zahl := 0;
  Inc(Test.Zahl);  // <-- Fehlermeldung
  Test.Destroy;
end.
```

Obwohl hier die Daten letztlich in der Klassenvariablen FZahl gespeichert werden, darf die dazugehörige Eigenschaft Zahl nicht an die Routine Inc übergeben werden. Hier bleibt letztlich nur der direkte Weg:

```
Test.Zahl := Test.Zahl + 1;
```

Standard-Eigenschaften

Wie oben beschrieben muss beim Zugriff auf eine Eigenschaft immer der Name der Eigenschaft selbst angegeben werden. Für Array-Eigenschaften gibt es eine praktische Besonderheit: die Standard-Eigenschaft. Wenn eine Array-Eigenschaft als Standard definiert wird, kann der Name dieser Eigenschaft beim Zugriff weggelassen werden. Dies erreicht man durch Angabe der Direktive **default** hinter der Eigenschaften-Definition:

```
 property Eigenschaft[X: Integer]: string read LeseMethode write
SchreibMethode; default;
```

Jetzt sind diese beiden Angaben gleichbedeutend:

```
 Wert := Objekt.Eigenschaft[Index];
 Wert := Objekt[Index];
```

Selbstverständlich kann nur jeweils eine Eigenschaft je Klasse als Standard definiert werden.

Vererbung

Überladene Methoden

Einleitung

Der Begriff **Überladen** (overload) beschreibt das mehrmalige Einführen einer **Function** oder **Procedure** mit gleichem Namen, aber unterschiedlichen Parametern. Der Compiler erkennt hierbei an den Datentypen der Parameter, welche Version er nutzen soll. Das Überladen ist sowohl in Klassen, für die dort definierten Methoden, als auch in normalen Prozeduren und Funktionen möglich.

Deklaration

Um eine Methode, Funktion oder Prozedur zu überladen, wird das Wort **overload;** hinter jede Deklaration der betroffenen Prozedur innerhalb des **Interface** Blockes gesetzt. Wichtig ist es zu beachten, dass die Parameter tatsächlich in der Reihenfolge ihrer Datentypen unterschiedlich sind. Es reicht nicht, unterschiedliche Parameternamen zu verwenden.

[bearbeiten] Beispiel

Aufbau der Klasse

```
TAnderes = class
public
  function anzStellen(zahl: Cardinal): Integer; overload;
  function anzStellen(zahl: string): Integer; overload;
end;
```

Die Methoden bestimmen die Länge, und damit auch die Stellenanzahl der Zahl oder des Strings.

Code der Methoden

```
function TAnderes.anzStellen(zahl: Cardinal): Integer;
begin
  Result := Length(IntToStr(zahl));
end;
```

Die Zahl wird erst in einen String umgewandelt (mit der Funktion IntToStr) und dann wird mit Hilfe der Funktion Length die Länge des Strings bestimmt.

```
function TAnderes.anzStellen(zahl: string): Integer;
begin
  Result := Length(zahl);
end;
```

Die Länge des Strings wird mit Hilfe der Funktion Length bestimmt.

Aufruf der Methoden

```
var
  s: string;
  c: Cardinal;
begin
  s := '1024';
  c := 16384;

  Writeln(anzStellen(s));
  Writeln(anzStellen(c));
end.
```

Im ersten Fall – `Writeln(anzStellen(s));` – wird die Methode `TAnderes.anzStellen(zahl: string): Integer` aufgerufen, da der Übergabeparameter ein String ist und dementsprechend der Wert 4 ausgegeben.

Im zweiten Fall – `Writeln(anzStellen(c));` – wird die Methode `TAnderes.anzStellen(zahl: Cardinal): Integer` aufgerufen, da der Übergabeparameter vom Typ Cardinal ist. Es wird 5 ausgegeben, da die Stellenanzahl von c bestimmt wurde.

Überladene Prozeduren und Funktionen

Man kann auch außerhalb einer Klasse definierte Funktionen oder Prozeduren überladen. Siehe hierzu Überladene Prozeduren / Funktionen.

Interfaces

Exceptions

Schnelleinstieg

Einstieg

Was ist Delphi

Borland Delphi ist eine RAD-Programmierumgebung von Borland. Sie basiert auf der Programmiersprache Object Pascal. Borland benannte Object Pascal jedoch 2003 in Delphi-Language um, mit der Begründung, dass sich bereits soviel in der Sprache verändert habe, dass man es nicht mehr mit Pascal vergleichen könne. RAD-Umgebungen für Object Pascal bzw. Delphi existieren nur wenige. Der bekannteste ist der Delphi-Compiler von Borland. Für Linux gibt es Kylix, das ebenfalls von Borland stammt und mit Delphi (ggf. mit wenigen Code-Änderungen) kompatibel ist. Darüber hinaus gibt es das freie Projekt Lazarus, das versucht eine ähnliche Entwicklungsumgebung bereitzustellen. Dieses nutzt FreePascal für die Kompilierung und ist für verschiedene Betriebssysteme erhältlich.

Warum Delphi?

Es gibt viele Gründe, Delphi zu benutzen. Es gibt aber wahrscheinlich auch genauso viele dagegen. Es ist also mehr Geschmacksache, ob man Delphi lernen will oder nicht. Wenn man allerdings Gründe für Delphi sucht, so fällt sicher zuerst auf, dass Delphi einfach zu erlernen ist, vielleicht nicht einfacher als Basic aber doch viel einfacher als C/C++. Gegenüber Basic hat Delphi allerdings den Vorteil, dass Delphi viel mächtiger ist. Für professionelle Programmierer ist es sicher auch wichtig zu wissen, dass die Entwicklung von eigenen Komponenten unter Delphi einfach zu handhaben ist. Durch die große Delphi-Community mangelt es auch nicht an Funktionen und Komponenten.

Erstellt man größere Projekte mit Borlands Delphi Compiler, so ist die Geschwindigkeit beim Kompilieren sicher ein entscheidender Vorteil. Auch die einfache Modularisierung, durch Units, Functions und Procedures ist sicherlich ein Vorteil der Sprache gegenüber einfachen Sprachen wie Basic.

Mit Delphi lässt sich zudem so ziemlich alles entwickeln, abgesehen von Systemtreibern. Dennoch ist es nicht unmöglich teilweise auch sehr hardwarenahe Programme zu entwickeln.

Die Oberfläche

Startet man Delphi zum ersten Mal, dann erscheint ein Fenster, das in mehrere Bereiche unterteilt ist. Oben sieht man die Menüleiste, links befindet sich der Objektinspektor und in der Mitte ist eine so genannte Form. Im Hintergrund befindet sich eine Art Texteditor. Aus der Menüleiste kann man verschiedene Aktionen, wie Speichern, Laden, Optionen, und anderes ausführen. Unten rechts in der Menüleiste (bzw. bei neueren Delphi-Versionen am rechten Bildschirmrand) kann man die verschiedenen Komponenten auswählen, die dann auf der Form platziert werden. Im Objektinspektor kann man die Eigenschaften und Funktionen der Komponenten ändern. Der Texteditor dient später einmal dazu den Quellcode des Programms einzugeben.

Das erste Programm (Hello world)

Wenn man nun mit dem Mauszeiger auf die Komponente Label (Menükarte „Standard“) und dann irgendwo auf die Form klickt, dann erscheint die Schrift „Label1“. Das so erzeugte Label kann man nun überall auf der Form verschieben. Um die Schrift zu ändern, wählt man das Label an und sucht im Objektinspektor die Eigenschaft „Caption“ und ändert den Wert von „Label1“ in „Hello world!“. Wenn man nun auf den grünen Pfeil in der Menüleiste klickt (oder F9 drückt), kompiliert Delphi die Anwendung, d.h. Delphi erstellt eine ausführbare Datei und startet diese. Nun sieht man ein Fenster mit der Schrift „Hello world!“. Das ist unsere erste Anwendung in Delphi! War doch gar nicht so schwer, oder?

Erweitertes Programm

Nachdem wir unser „Hello world“-Programm mit Datei->Projekt speichern... gespeichert haben, erstellen wir mit Datei->Neu...->Anwendung eine neue Anwendung. Auf die neue Form setzt man nun einen Button und ein Label (beides Menü Standard). Wenn man nun doppelt auf den Button klickt, dann öffnet sich das Code-Fenster. Hier geben sie folgendes ein:

```
Label1.Caption := 'Hello world';
```

Wenn man nun das Programm mit F9 startet und dann auf den Button klickt sieht man, dass sich das Label verändert und nun „Hello world“ anzeigt. Damit haben wir eine Möglichkeit, Labels (und eigentlich alles) während der Laufzeit zu verändern.

Beenden Sie nun die Anwendung und löschen Sie den eben getippten Text wieder. Nun bewegen Sie den Cursor vor das `begin` und geben folgendes ein:

```
var
  x, y, Ergebnis: Integer;
```

Wenn Sie wieder unter dem `begin` sind, dann geben Sie

```
x := 40;
y := 40;
Ergebnis := x + y;
Label1.Caption := IntToStr(Ergebnis);
```

ein. Nun kompilieren Sie mit F9 und sehen sich das Ergebnis an. Sie sehen, wenn man auf den Button klickt, verändert sich das Label und zeigt nun das Ergebnis der Addition an. Dies kann man natürlich auch mit anderen Zahlen oder Operationen durchführen (es sind Zahlen von -2147483648 bis 2147483647 und die Operanden + - * möglich).

Nun fassen wir mal zusammen, was Sie bis jetzt gelernt haben könnten bzw. jetzt lernen:

?? In Delphi fungiert das „`:=`“ als so genannter Zuweisungsoperator, d.h. die linke Seite enthält nach der Operation den Inhalt der rechten Seite. z.B.:

```
x := 40;
x := y;
Label1.Caption := 'Text';
```

?? Es gibt unter Delphi Variablen, die vorher in einem speziellen **var**-Abschnitt definiert werden, z.B.: `x: Integer;`, so kann x Zahlen von -2147483648 bis 2147483647 aufnehmen oder `x, y: Integer;` definiert x und y als Integer (spart Schreibarbeit!)

Die Syntax

(Fast) jeder Programmcode-Befehl wird mit einem Semikolon/Strichpunkt (;) abgeschlossen. Ausnahmen davon sind ein nachfolgendes **end**, **else** oder **until**.

```
ShowMessage('Hallo Welt');
```

Strings werden von einfachen Anführungszeichen eingeschlossen

```
var
  s: string;
...
  s := 'ich bin ein String';
```

Ebenso wie einzelne Zeichen (Char)

```
var
  c: Char;
...
  c := 'A';
```

Die Strukturen von Delphi

Delphi verfügt über die aus Pascal bekannten Kontrollstrukturen zur Ablaufsteuerung: Reihenfolge, Auswahl und Wiederholung. Als **Auswahlkonzepte** stehen zur Verfügung: ein-, zwei- und mehrseitige Auswahl (Fallunterscheidung). Als **Wiederholstrukturen** verfügt Delphi über abweisende, nicht abweisende Schleifen und Zählschleifen. Die Syntax der Strukturen stellt sich wie folgt dar:

Reihenfolge

```
<Anweisung>;
<Anweisung>;
<Anweisung>;
<...>
```

Einseitige Auswahl

```
if <Bedingung> then <Anweisung>;
```

Zweiseitige Auswahl

```
if <Bedingung> then
  <Anweisung>
else
  <Anweisung>;
```

Fallunterscheidung

```
case <Fall> of
  <wert1> : <Anweisung>;
  <wert2> : <Anweisung>;
  <.....> : <Anweisung>
else
  <Anweisung>
end; // von case
```

Abweisende Schleife

```
while <Wiederhol-Bedingung> do
  <Anweisung>;
```

Nicht abweisende Schleife

```
repeat
  <Anweisung>;
  <Anweisung>;
  ...
until <Abbruch-Bedingung>;
```

Zählschleife

```
for <Laufvariable> := <Startwert> to <Endwert> do
  <Anweisung>;
```

Strukturen können ineinander verschachtelt sein: Die Auswahl kann eine Reihenfolge enthalten, in einer Wiederholung kann eine Auswahl enthalten sein oder eine Auswahl kann auch eine Wiederholung enthalten sein. Enthält die Struktur `<Anweisung>` mehr als eine Anweisung, so sind Blöcke mittels **`begin`** und **`end`** zu bilden. Ausnahme davon ist die Repeat-Schleife: Hier wird zwischen **`repeat`** und **`until`** automatisch ein Block gebildet.

Programmbeispiel mit allen Strukturen

```
program example001;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  i       : Integer;
  Zahl    : Real;
  Antwort: Char;
begin
  WriteLn('Programmbeispiel Kontrollstrukturen');
  WriteLn;
  repeat                  // nicht abweisende Schleife
    Write('Bitte geben Sie eine Zahl ein: ');
    ReadLn(Zahl);
    if Zahl <> 0 then     // einseitige Auswahl
      Zahl := 1 div Zahl;
    for i := 1 to 10 do   // Zählschleife
      Zahl := Zahl * 2;
    while Zahl > 1 do     // abweisende Schleife
      Zahl := Zahl div 2;
    i := Round(Zahl) * 100;
    case i of             // Fallunterscheidung
      1: Zahl := Zahl * 2;
      2: Zahl := Zahl * 4;
```

```
        4: Zahl := Zahl * 8
      else
        Zahl := Zahl * 10
      end;
      if Zahl <> 0 then     // zweiseitige Auswahl
        WriteLn(Format('Das Ergebnis lautet %.2f', [Zahl]))
      else
        Writeln('Leider ergibt sich der Wert von 0.');
      Write('Noch eine Berechnung (J/N)? ');
      ReadLn(Antwort)
    until UpCase(Antwort) = 'N'
  end.
```

Prozeduren und Funktionen

Grundsätzlich wird zwischen Prozeduren und Funktionen unterschieden. Dabei liefert eine Funktion immer einen Rückgabewert zurück, hingegen eine Prozedur nicht.

Der eigentliche Code einer Prozedur oder Funktion (in OOP auch als Methode bezeichnet) beginnt immer nach dem `begin` und endet mit `end;`

```
procedure testmethode;
begin
  ShowMessage('TestMethode!');
end;
```

Die folgende Funktion liefert beispielsweise einen String zurück.

```
function testmethode: string;
begin
  Result := 'ich bin der Rückgabewert';
end;
```

Parameter können wie folgt verwendet werden:

```
procedure testmitparameter1(Parameter1: string);
```

Mehrere Parameter werden mit „`;`“ getrennt.

```
procedure testmitparameter2(Parameter1: string; Parameter2: Integer);
```

Mehrere Parameter vom gleichen Typ können auch mit „`,`“ getrennt werden:

```
procedure testmitparameter3(Parameter1, Parameter2: string);
```

Datentypen (Array, Records und Typen)

Arrays

Ein Array ermöglicht es, Daten zu indizieren und Tabellen zu erstellen. Es gibt zwei verschiedene Arten von Arrays:

Eindimensionale Arrays

Statische Arrays

?? Ein **statisches Array** besitzt eine in der Variablendeklarierung festgelegte Größe, die in eckigen Klammern angegeben wird:

```
var
  test_array: array[1..10] of Byte;
```

definiert ein Array, das mit 10 Byte-Werten gefüllt werden kann. Man unterscheidet in Delphi zwischen 0-basierten und 1-basierten Arrays, wobei der Unterschied eigentlich nur in der Zählweise liegt:

```
var
  array_0: array[0..9] of Byte;
```

definiert ein gleich großes Array mit 0 als ersten Index. Dies ist eigentlich Geschmacksache, aber wer nebenbei C/C++ programmiert, dem wird die 0-basierte Schreibweise sicher vertrauter sein.

Dynamische Arrays

?? Ein **dynamisches Array** ist in seiner Größe dynamisch (wie der Name schon sagt). Das heißt, dass man die Größe während der Laufzeit verändern kann, um beliebig große Datenmengen aufzunehmen. Die Deklaration erfolgt mit

```
var
  array_d: array of Byte;
```

Will man zur Laufzeit dann die Größe verändern, so erfolgt die über die procedure `SetLength(array, neue_Laenge)`. Mit den Funktionen `High(array)` und `Low(array)` erhält man den höchsten und den niedrigsten Index des Arrays. Bei dynamischen Arrays ist zu beachten, dass diese immer 0-basiert sind.
Mit `Length(tabelle1)` bekommt man die Länge des Arrays heraus. Hier ist zu beachten, dass bei einem Array von 0 bis 2, 3 Elemente vorhanden sind und darum die Zahl 3 zurückgegeben wird.

Der Zugriff auf ein Array erfolgt mit `arrayname[5] := 10;`. Der Vorteil von Arrays ist, dass man sie einfach in Schleifen einbauen kann:

```
var
  arrayname: array[1..10] of Byte;
begin
  for i := 1 to 10 do
    arrayname[i] := 10;
end;
```

Mehrdimensionale Arrays

Außerdem kann man mit Arrays mehrere Dimensionen definieren:

Statische Arrays

```
var
  tabelle1: array[1..10] of array[1..10] of Byte;    // dies ist
gleichbedeutend mit:
  tabelle2: array[1..10, 1..10] of Byte;
```

definiert beides eine Tabelle mit der Größe 10x10. Der Zugriff erfolgt über

```
tabelle[5, 8] := 10;
```

Damit wird die Zelle in der 8. Reihe und in der 5. Spalte mit 10 belegt.

Dynamische Arrays

```
var
  tabelle1: array of array of Byte;
```

Natürlich muss man auch hier mit `SetLength` die Länge des Arrays bestimmen. Und das geht so:

```
SetLength(tabelle1, 9, 9);
```

Dies definiert eine 2-dimensionale Tabelle mit der Größe 10x10. Der Zugriff kann nun wie bei einem statischen Array erfolgen.

Typen

Mit einer Typendeklaration kann man eigene Datentypen festlegen, dies erfolgt z.B. durch

```
type
  Zahl = Integer;
```

Nach dieser Deklaration kann man überall anstatt Integer Zahl einsetzen:

```
var
  x: Zahl;
```

Interessant werden Typen, wenn man zum Beispiel eine Arraydefinition, z.B. eine bestimmte Tabelle öfter einsetzen will:

```
type
  Tabelle = array[1..10, 1..10] of Byte;
```

oder wenn man Records oder Klassen benutzt.
Natürlich funktioniert das mit allen Arten von Arrays, also auch mehrdimensional, dynamisch, etc.

Records

Mit einem Record kann man einer Variable mehrere Untervariablen geben, dies ist z.B. bei Datensammlungen der Fall. Man definiert einen Record fast immer über einen Typen:

```
type
  THighscoreEintrag = record
    Nr: Byte;
```

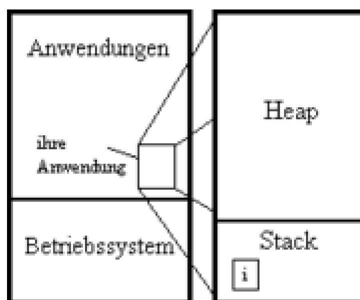
```
    Name: string;
    Punkte: Integer;
  end;
```

Wenn man nun eine 10-stellige Highscoreliste erstellen will, dann benutzt man die Anweisung:

```
var
  Hscr: array[1..10] of THighscoreEintrag;
begin
  Hscr[1].Nr := 1;
  Hscr[1].Name := 'Der Erste';
  Hscr[1].Punkte := 10000;
end;
```

Pointer

Einleitung

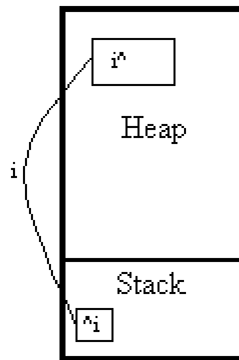


Das RAM als Modell

Betrachtet man das RAM, so fällt folgende Struktur auf:

Das RAM ist unter den verschiedenen Anwendungen und dem Betriebssystem aufgeteilt und den RAM, den eine Anwendung verbraucht kann man in Heap und Stack unterteilen. Im Stack werden alle Variablen gepeichert, die z.B. mit `var i: Integer;` deklariert worden sind. Das Problem am Stack ist, dass er in der Größe sehr beschränkt ist. Bei großen Datenstrukturen sollte man daher auf den Heap zurückgreifen und dazu benötigt man **Zeiger (Pointer)**.

Grundlagen



Ein Zeiger in den Heap

Ein Pointer ist in der Regel 4 Bytes groß, d.h. es lohnt sich normalerweise nicht, einen Pointer auf einen Bytewert zu erzeugen, da dieser nur einen Byte belegt! Ein Pointer auf eine bestimmte Datenstruktur wird folgenderweise definiert:

```
var
  zgr: ^TDaten;
```

Einem Pointer muss immer ein Wert zugewiesen werden! Für den Fall, dass ein Pointer vorerst nicht gebraucht wird, kann ein Zeiger auch ins Leere zeigen, dies geschieht mit der Zuweisung:

```
zgr := nil; // nil = not in list
```

Bevor man den Zeiger allerdings auf Daten zeigen lassen kann muss man zuerst Speicher im Heap reservieren:

```
New(zgr);
```

Nun kann auf die Daten zugreifen wie bei einer normalen Variable, allerdings hängt man dem Zeigernamen beim Zugriff das Dach hinten an:

```
zgr^ := 50000;
```

Dies ist nötig, da in zgr selber die Anfangsadresse von zgr^ im Heap gespeichert ist. Wenn TDaten allerdings ein Record ist, so kann man theoretisch beim Zugriff auf die einzelnen Unterpunkte das Dach weglassen:

```
zgr.nummer := 1;
```

Wenn man einen Pointer im weiterem Programmverlauf nicht mehr braucht, dann sollte man unbedingt den belegten Speicher wieder freigeben:

```
Dispose(zgr);
```

Mit diesen Grundlagen sollte das Kapitel über Listen, Bäume und Schlangen eigentlich kein Problem mehr darstellen!

Dynamische Datenstrukturen

Die klassische Methode: Listen

```
Bitte beachten: Die hier gezeigte Vorgehensweise ist stark veraltet,
fehleranfällig und umständlich.
Seit Delphi 4 gibt es bessere und sicherere Methoden, dynamische
Datenstrukturen zu handhaben.
Siehe hierzu Die moderne Methode: Klassen. Die klassische Vorgehensweise
ist nur für erfahrene
Programmierer geeignet, die aus dieser Laufzeit-Geschwindigkeitsvorteile
ziehen wollen.
```

Eine Liste ist wie alle dynamischen Datenstrukturen eine rekursive Datenstruktur, d.h.: sie referenziert sich selbst. Schauen wir uns mal folgendes Beispiel für eine Typendefinition für Listen an:

```
type
  PListItem = ^TListItem;
  TListItem = record
    data: Integer;
    next: PListItem;  // Verweis auf das nächste Item
  end;
```

Man sieht, dass der Zeiger in PListItem auf ein Objekt gelegt wird, das noch nicht definiert ist. Eine solche Definition ist nur bei rekursiven Strukturen möglich.

Möchte man nun eine neue, leere Liste erzeugen, so reicht folgender Code:

```
var
  Liste: PListItem;
begin
  New(Liste);
  Liste^.data := 0;
  Liste^.next := nil; // WICHTIG!!!
end;
```

Vergessen Sie bitte niemals, einen nicht benötigten Zeiger (wie in diesem Fall) auf **nil** zu setzen, da das gerade bei dynamischen Datenstrukturen zu nicht vorhersehbaren Fehlern führen kann.

Wollen sie nun ein Item der Liste hinzufügen, ist folgender Code zu benutzen:

```
New(Liste^.next);
```

und die entsprechende Belegung mit Nichts:

```
Liste^.next^.data := 0;
Liste^.next^.next := nil;
```

Es ist natürlich lästig und aufwändig, die Liste auf diese Weise zu vergrößern. Deshalb benutzt man eine Prozedur, um die Liste bis zu ihrem Ende zu durchlaufen und an ihrem Ende etwas anzuhängen:

```
procedure AddItem(List: PListItem; data: Integer);
```

```
var
  tmp: PListItem;
begin
  tmp := List;
  while tmp^.next <> nil do
    tmp := tmp^.next;
  New(tmp^.next);
  tmp^.next^.data := data;
  tmp^.next^.next := nil;
end;
```

Da dies aber sehr zeitaufwändig ist, sollte immer das letzte Element der Liste gespeichert werden, um nicht zuerst die gesamte Liste durchlaufen zu müssen, bevor das neue Element angehängt werden kann.

Um eine Liste wieder aus dem Speicher zu entfernen, genügt es nicht, die Variable Liste auf **nil** zu setzen. Dabei würde der verbrauchte Speicherplatz belegt bleiben und auf die Dauer würde das Programm den Speicher „zumüllen“. Um die Objekte wieder freizugeben, muss `Dispose` verwendet werden:

```
Dispose(Item);
```

Da hierbei jedoch das Element in `next` (falls vorhanden) nicht ordnungsgemäß freigegeben würde, muss man mit einer Schleife alle Elemente einzeln freigeben:

```
procedure DisposeList(var List: PListItem);
var
  current, temp: PListItem;
begin
  current := List;
  while current <> nil do
  begin
    temp := current^.next;
    Dispose(current);
    current := temp;
  end;
  List := nil;
end;
```

Hier wird in jedem Schleifendurchlauf das aktuelle Element freigegeben und das nächste Element zum aktuellen gemacht. Hier wäre zwar prinzipiell auch eine rekursive Funktion möglich (und eventuell auch die zunächst offensichtliche Lösung), diese würde aber bei sehr großen Listen einen Stack Overflow auslösen.

Die moderne Methode: Klassen

Nachteile der Listenmethode sind vor allem die Fehleranfälligkeit und die umständliche Referenzierung. Sinnvoller - und natürlich auch komplexer - ist hier der Einsatz einer sich selbst verwaltenden Klasse. Hierbei hat man als Anwender lediglich Zugriff auf Methoden, nicht jedoch auf die Datenstruktur selbst. Dies bewirkt einen höchstmöglichen Schutz vor Fehlern.

Grundgerüst

Als Basis für die dynamische Liste wird ein offenes Array verwendet. Der Speicherbedarf hierfür lässt sich dem Bedarf entsprechend anpassen. Weiterhin werden Methoden benötigt, um neue Daten hinzuzufügen, nicht mehr benötigte zu löschen, vorhandene auszulesen oder zu ändern. Dies alles wird in einer Klasse gekapselt.

```
type
  TMyList = class
  private
    FFeld: array of Integer;
  public
    constructor Create;
    procedure Free;
    function Count: Integer;
    procedure Add(NewValue: Integer);
    procedure Delete(Index: Integer);
    procedure Clear;
    function GetValue(Index: Integer): Integer;
    procedure SetValue(Index: Integer; NewValue: Integer);
  end;
```

Implementation

Im Constructor muss dem Feld Speicher zugewiesen werden, ohne diesen jedoch bereits mit Daten zu füllen:

```
constructor TMyList.Create;
begin
  inherited;
  SetLength(FFeld, 0);
end;
```

Die Methode `Free` sollte auf gleiche Weise den Speicher wieder freigeben:

```
procedure TMyList.Free;
begin
  SetLength(FFeld, 0);
  inherited;
end;
```

Für die Verwendung der Liste ist es oftmals notwendig, deren Größe zu kennen, um nicht eventuell über deren Ende hinaus Daten auszulesen. Das wird mit der simplen Methode `Count` verwirklicht.

```
function TMyList.Count: Integer;
begin
  Result := Length(FFeld);
end;
```

Um nun Daten hinzuzufügen, wird eine weitere Methode benötigt: `Add`. Hierbei ist zu beachten, dass der erste Index der Liste immer 0 (Null) ist. Wenn nur ein Eintrag enthalten ist (`Count = 1`), dann ist dieser in `FFeld[0]` zu finden. Durch den Aufruf von `SetLength` wird das Feld um einen Eintrag erweitert, womit sich auch das Ergebnis von `Count` um 1 erhöht. Demnach muss beim Speichern des Wertes wieder 1 subtrahiert werden.

```
procedure TMyList.Add(NewValue: Integer);
begin
  SetLength(FFeld, Count+1);     // Größe des Feldes erhöhen
```

```
  FFeld[Count-1] := NewValue;    // Neuen Eintrag ans Ende der Liste setzen
end;
```

Das Löschen eines Eintrages gestaltet sich etwas schwieriger, da dieser sich am Anfang, in der Mitte oder am Ende der Liste befinden kann. Je nachdem müssen die Daten gegebenenfalls umkopiert werden.

```
procedure TMyList.Delete(Index: Integer);
var
  i: Integer;
begin
  if (Index < 0) or (Index >= Count) then
    Exit;

  if Index = Count - 1 then          // letzter Eintrag
    SetLength(FFeld, Count-1)        // Es reicht, den Speicher vom letzten
Eintrag freizugeben
  else                               // erster Eintrag oder irgendwo in der
Mitte
  begin
    for i := Index to Count - 2 do
      FFeld[i] := FFeld[i+1];
    SetLength(FFeld, Count-1);
  end;
end;
```

Man sollte auch die Liste leeren können, ohne jeden einzelnen Eintrag zu löschen. Hierfür muss nur wieder die Größe des Feldes auf 0 gesetzt werden.

```
procedure TMyList.Clear;
begin
  SetLength(FFeld, 0);
end;
```

Das Auslesen und Ändern vorhandener Daten gestaltet sich ebenfalls sehr einfach.

```
function TMyList.GetValue(Index: Integer): Integer;
begin
  if (Index < 0) or (Index >= Count) then
    Result := -1                        // Oder ein anderer Wert, der einen
Fehler darstellt
  else
    Result := FFeld[Index];
end;
procedure TMyList.SetValue(Index: Integer; NewValue: Integer);
begin
  if (Index < 0) or (Index >= Count) then
    Exit
  else
    FFeld[Index] := NewValue;
end;
```

Erweiterungsmöglichkeit

Um einen noch komfortableren Zugriff auf die Daten zu erhalten, können diese über eine Eigenschaft aufgerufen werden. Hierzu werden die Methoden `GetValue` und `SetValue`

„versteckt“, das heißt, im Bereich **private** der Klasse untergebracht. Im Bereich **public** kommt folgendes hinzu:

```
public
  property Items[I: Integer]: Integer read GetValue write SetValue;
default;
```

Auf diese Weise ist ein wirklich einfacher Zugriff auf die Daten möglich:

```
var
  MyList: TMyList;
  i: Integer;
begin
  MyList := TMyList.Create;
  MyList.Add(2);        // MyList[0] = 2
  MyList.Add(3);
  MyList.Add(5);

  i := MyList[2];       // i = 5
  MyList[0] := 13;      // MyList[0] = 13
end;
```

Anmerkung

Die hier gezeigte Lösung stellt nur den einfachsten Ansatz einer dynamischen Datenverwaltung dar. Sie kann jedoch Daten jeden Typs speichern, es muss lediglich der Typ von `FFeld` und aller betroffenen Methoden angepasst werden. Auch andere Klassen können so gespeichert werden.

Dieses Grundgerüst ist auf einfachste Weise erweiterbar. Zum Beispiel können Methoden zum Suchen und Sortieren von Daten eingebaut werden. Auch das Verschieben von Einträgen wäre denkbar.

DLL-Programmierung

Was ist eine DLL?

Eine DLL (Dynamic Link Libary) kann Routinen und Forms beinhalten, die dann in ein beliebiges Programm eingebunden werden können, auch in andere Sprachen. Dazu wird die DLL in den Speicher geladen, von wo aus alle Anwendungen auf sie zurückgreifen können. Aus diesen Eigenschaften ergeben sich folgende Vorteile:

- Programmiersprachenunabhängiges Programmieren
- Beim Benutzen durch mehrere Anwendungen wird der Code nur einmal geladen
- Die DLL kann dynamisch in den Code eingebunden werden

Das Grundgerüst einer DLL

Klickt man auf Datei->Neu->DLL erhält man folgendes Grundgerüst für eine DLL:

```
library Project1;
{ ...Kommentare (können gelöscht werden)...}
uses
  SysUtils,
  Classes;

begin
end.
```

Wie man sieht, wurde das für ein Programm übliche **program** durch **library** ersetzt, dadurch „weiß" der Compiler, dass er eine DLL kompilieren soll und dieser die Endung .dll gibt.

In den Bereich zwischen dem Ende des **uses** und dem **begin** können nun beliebig viele Prozeduren und Funktionen eingegeben werden. Wenn man die DLL nun kompiliert, kann man allerdings noch nichts mit ihr anfangen! Man muss die Funktionen/Prozeduren, die man aus der DLL nutzen soll, exportieren! Dazu benutzt man vor dem Abschließenden **begin end** folgenden Code:

```
exports
  Funktion1 index 1,
  Funktion2 index 2;
```

Dadurch kann man diese Funktionen aus jedem Windows-Programm nutzen. Da andere Sprachen allerdings andere Aufrufkonventionen besitzen, muss der Funktionsdeklaration ein **stdcall**; hinzugefügt werden, das für den Standard-Aufruf von anderen Sprachen steht. Damit erhält man z.B. folgenden Code:

```
library DemoDLL;

uses
  SysUtils,
  Classes;

function Addieren(x, y: Byte): Word; stdcall;
begin
  Result := x + y;
end;

function Subtrahieren(x, y: Byte): ShortInt; stdcall;
begin
  Result := x - y;
end;

exports
  Addieren index 1,
  Subtrahieren index 2;

begin
end.
```

Assembler und Delphi

Allgemeines

In Delphi wird der Assemblerteil immer mit **`asm`** eröffnet und mit dem obligatorischen **`end`** geschlossen. Die Register edi, esi, esp, ebp und ebx müssen innerhalb des Assemblerteils unverändert bleiben! Will man sie trotzdem verwenden, muss man sie vorher auf dem Stack ablegen:

```
push edi
push esi
push esp
push ebp
push ebx
```

Und am Ende der Assembler-Anweisungen wieder zurückholen:

```
pop ebx
pop ebp
pop esp
pop esi
pop edi
```

Syntax

Es können wahlweise Assemblerblocks mit Delphicode vermischt werden, oder ganze Prozeduren/Funktionen in Assembler verfasst werden.

Beispiel für Assembler-Funktion

```
function Add(a, b: Integer): Integer; assembler;
asm
  add eax, edx
end;
```

Das Schlüsselwort **assembler** ist hierbei nicht verpflichtend.

Beispiel für Assembler in Delphi

```
function Add(a, b: Integer): Integer;
var
  temp: Integer;
begin
  asm
    mov eax, a
    mov temp, eax
  end;
  Result := temp + b;
end;
```

In der Regel wird in Assembler-Abschnitten nicht mehr als ein Befehl pro Zeile untergebracht. Mehrere Befehle können durch ein Semikolon oder einem Kommentar voneinander getrennt werden.

Assembler-Zeile:

```
[Label] [Befehl] [Operand(en)]
```

Beispiele:

```
xor eax, eax
mov variable, eax
xor eax, eax; mov variable, eax
xor eax, eax {eax leeren} mov variable, eax
```

Alle drei Varianten sind gleichwertig, die erste Variante mit einem Befehl pro Zeile ist jedoch die übliche und sollte vorgezogen werden. Kommentare dürfen in Assembler-Blocks nie zwischen Befehl und Operanden stehen, zwischen Label und Befehl jedoch schon.

Labels müssen als lokale Labels definiert sein:

```
@xxx:
{ ... }
jmp @xxx
```

Einige unterstützte Befehle gibt es in der Delphi-Hilfe unter dem Index „Assembler-Anweisungen“. Der integrierte Assembler unterstützt die Datentypen Byte(db), Word(dw) und DWord(dd), ein Pointer entspricht einem DWord. Wenn Sie eine Variable deklarieren, die denselben Namen hat wie ein reserviertes Wort, dann hat das reservierte Wort Vorrang. Auf die deklarierte Variable kann durch Voransetzen eines kaufmännischen Unds zugegriffen werden:

```
var
  ax: Integer;
asm
  mov ax, 4  {verändert das Register ax}
  mov &ax, 4 {verändert die Variable ax}
end;
```

RAD-Umgebung

Warum eine RAD-Umgebung?

Um diese Frage zu beantworten, muss man erst einmal den Begriff RAD definieren: RAD ist ein Akronym für Rapid Application Development (dt. etwa: „Schnelle Anwendungsentwicklung“). Sprich, es behauptet, man könne schnell Anwendungen entwickeln. Verwirklicht wird dies durch:

- IDE: Die integrierte Entwicklungsumgebung („Integrated Development Environment“) bietet Design, Coding, Compiler und Debugging in einer Oberfläche
- Automatisierung: per Knopfdruck (bei Delphi „F9“) wird das Projekt geprüft, kompiliert, gelinkt und dann ausgeführt. Das geschieht meist innerhalb von wenigen Sekunden, man kann also eine kleine Änderung vornehmen und sofort sehen, wie sich diese im Programm auswirkt.

Nun mag vielleicht die Frage aufkommen „Was gibt es denn dann noch außer RAD?“. Antwort: Es gibt zum Beispiel Kommandozeilen-Compiler, bei denen der Vorgang so abläuft:

- Code komplett in eine Textdatei schreiben
- Compiler mit der Datei ausführen

- Wenn ein Compiler-Fehler auftritt, zurück zum Anfang und korrigieren, dann kommt der nächste, denn der Compiler spuckt immer nur einen Fehler auf einmal aus
- Dann die Objektfiles linken - bei Linkerfehler das selbe Spiel wie beim Kompilieren
- Und dann endlich sollte ein ausführbares Programm rauskommen, bei dem man dann überprüfen kann, ob man alles richtig programmiert hat. Wenn das nicht der Fall ist, zurück zum Anfang.

Der Vorteil eines solchen Systems: Man gewöhnt sich bald an, sehr, sehr sauber zu programmieren - Die Einsteigerfreundlichkeit ist aber gleich Null.

Die Antwort lautet also:

- Mit einer RAD-Umgebung kann man schnell und flexibel programmieren.
- Eine RAD-Umgebung ist anfängerfreundlich.
- Eine RAD-Umgebung ist kompakt und in sich stimmig.

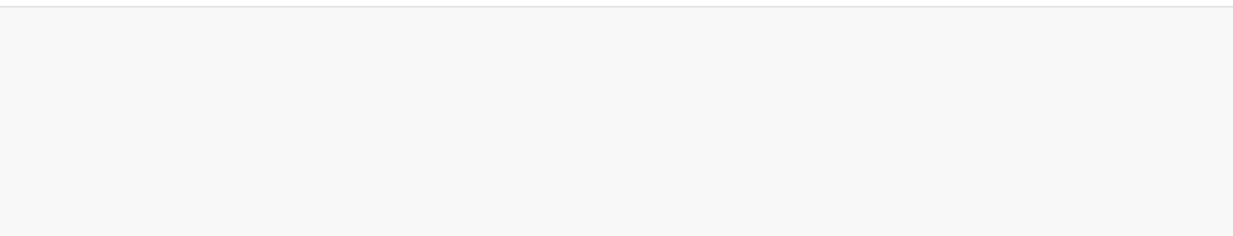
Erstellung einer graphischen Oberfläche

Anhänge

Befehlsregister

Glossar

Lizenz



Autoren

Diese Autoren haben an diesem Wikibook mitgearbeitet (IP-Adressen nicht aufgelistet):

- Alpha
- Axelf
- Bastie
- Daniel B
- Dannys9
- Delphi-himmel.de
- Delphimann
- Derbeth

?? DieFel
?? Dosenfant
?? Drumlin
?? Gbg
?? Gnushi
?? Gordon Freeman
?? Hollerbach
?? I.MacLeod
?? Jonathan Hornung
?? Luke
?? MGla
?? Muffin
?? Progman
?? Shivan
?? Sliver
?? Stefan Majewsky
?? ThePacker

Letzte Aktualisierung: 23.6.2006, 23:23:48

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

?? **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be

listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

?? **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

?? **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.

?? **D.** Preserve all the copyright notices of the Document.

?? **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

?? **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

?? **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

?? **H.** Include an unaltered copy of this License.

?? **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

?? **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

?? **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

?? **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

?? **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

?? **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

?? **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.