DELPHI 02

Sprachelemente und Strukturen

© Herbert Paukert

[2] KLASSISCHE SPRACHELEMENTE

Hinter jedem Programm in DELPHI steht ein Quellcode in der Programmiersprache OBJECT PASCAL. Der ursprüngliche Programmkern von TURBO PASCAL für MSDOS wurde um objekt- und ereignisorientierte Bestandteile von WINDOWS erweitert. Wir wollen hier nur die klassischen Sprachelemente vorstellen, um die Grundlagen zu vermitteln und den Einsteiger nicht mit dem gewaltig angestiegenen Sprachumfang von OBJECT PASCAL zu belasten.

Der Zeichenvorrat einer Sprache umfasst Buchstaben, Ziffern und Sonderzeichen. Aus diesen werden die Sprachelemente (Ausdrücke und Befehle bzw. Anweisungen) gebildet. Dabei ist zu beachten, dass als Zeichen keine "Umlaute" und kein "ß" zugelassen sind. Hingegen ist die Groß- oder Kleinschreibung ohne Belang. Trotzdem sollte, wegen der besseren Lesbarkeit, ein und derselbe Bezeichner immer gleichartig geschrieben werden. Jede Anweisung muss von einem Strichpunkt beendet werden. **Grundelemente** einer Hochsprache sind Konstanten, Variablen und Funktionen. Durch die Verknüpfung dieser Basiselemente mit zulässigen Operatoren (z.B.: +) werden **Ausdrücke** gebildet. Die **Anweisungen** werden mit vorgegebenen Schlüsselwörtern bezeichnet.

[2.01] Programmgliederung

Bei der klassischen Gliederung eines Programmes oder einer Prozedur werden prinzipiell zwei Programmteile unterschieden: Zuerst kommt der Definitionsteil und dahinter dann der Ablaufteil. So ergibt sich folgendes grundsätzliche Schema (der Programm- bzw. Prozedur-Name sei *PName*):

program *PName; (bzw.* ***procedure*** *PName;)*

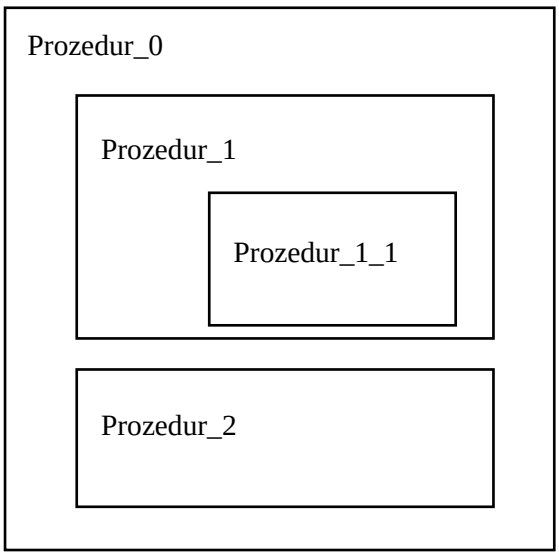
> **Definitionsteil:**
> Hier erfolgt die Definition der Variablen, die im Ablaufteil verwendet werden. Dabei wird für sie ein Name und ein entsprechend strukturierter Speicherbereich reserviert.

begin

> **Ablaufteil:**
> Hier erfolgt die Eingabe, Verarbeitung und Ausgabe (EVA) jener Daten, die vorher im Definitionsteil als Variablen deklariert worden sind.

end. *(bzw.* ***end;****)*

Ein Programm oder eine Prozedur kann mehrere Unterprogramme (z.B. Prozeduren) enthalten:



Für den Datenzugriff in hierarchisch gegliederten Programmen gilt folgende Regel: In einer Routine definierte Variablen sind innerhalb der Routine selbst und in allen ihren untergeordneten Routinen bekannt und zugreifbar. Das ist die **globale** Sichtweise. In den übergeordneten Routinen sind sie nicht bekannt und daher nicht zugreifbar. Das ist die **lokale** Sichtweise.

Speicherintern wird diese Datenkapselung dadurch erreicht, dass alle lokalen Daten und auch die Werteparameter einer Routine auf einem so genannten Stapelspeicher (**Stack**) abgelegt werden. Beim Verlassen der Routine werden diese Daten wieder vom Stack entfernt.

[2.02] Datentypen

Grundsätzlich kann man zwischen numerischen und nicht numerischen Daten unterscheiden. Beim ersten Datentyp werden die Speicherbytes als ganze Zahlen (***Integer***) oder als Dezimalzahlen (***Real***) interpretiert. Beim zweiten Datentyp werden die Speicherbytes als Zeichencodes entsprechend der internationalen ANSI-Norm interpretiert. Solche Zeichenketten nennt man auch ***Strings***. Jeder Text besteht aus einer Folge von Strings.

Die verschiedenen Daten werden im Speicher durch **Konstanten** und **Variablen** repräsentiert.

Variablen sind Sprachelemente, denen während des Programmablaufs verschiedene Werte zugewiesen werden können. Um eine Variable im Programm ansprechen zu können, muss ihr vorher ein Name zugeordnet werden (Designation). Außerdem muss der Wertebereich der Variablen festgelegt, d.h. ein entsprechender Bereich im Hauptspeicher reserviert werden. Der Name (z.B. *VName*) steht dann symbolisch für die Speicheradresse des reservierten Bereiches. All das erfolgt durch die **Variablendefinition**, welche mit dem reservierten Wort ***var*** eingeleitet wird:

```
var  VName :  Datentyp;
```

So wird zum Beispiel durch die Programmanweisung ***var*** *Ergebnis : Integer;* einer Variablen mit dem Namen *Ergebnis* eine freie Speicheradresse zugeteilt, welche nur ganze Zahlen aufnehmen soll. Jedesmal wenn die Variable durch ihren Namen im Programm aufgerufen wird, ersetzt der Compiler diesen Namen durch die zugeteilte Speicheradresse. Durch die Typisierung der Variablen als Integer werden vier Byte des Speichers reserviert und der Compiler weiß, dass das erste Bit als Vorzeichen (0 = +, 1 = –) und die restlichen 31 Bit als Betrag der Zahl zu interpretieren sind. Daraus ergibt sich für die Werte solcher Integer-Zahlen n ein Bereich von $-2^{31} \le n < +2^{31}$, wobei 2^{31} gleich 2 147 483 648 ist.

Beispiel:

```
var  n : Integer;                 // Definition einer globalen Integer-Variablen
                                  // außerhalb einer Prozedur
procedure Test;                   // Kopfzeile mit dem Namen der Prozedur
var  s : String;                  // Definition einer lokalen String-Variablen
                                  // in der Prozedur. Nur dort ist sie zugreifbar.
begin                             // Beginn des Prozeduren-Codes mittels begin
   n := 53;                       // einfache Wertzuweisung mittels :=
   s := 'Herbert';                // einfache Wertzuweisung mittels :=
end;                              // Ende des Prozeduren-Codes mittels end
```

Konstanten sind Sprachelemente, deren Werte sich zur Laufzeit des Programms nicht ändern können. Durch das reservierte Wort ***const*** erkennt der Compiler, dass es sich um die Definition einer Konstanten handelt. Der Befehl, mit dem eine Konstante definiert wird, lautet:

```
const  CName = Wert;
```

CName steht für einen beliebig wählbaren Namen und *Wert* kann eine Wertangabe, wie zum Beispiel 734.50 für eine numerische Konstante oder 'Herbert' für eine Textkonstante (String), sein.

Beispiel:

```
const breite  =  734.50;          // Dezimalwertige Konstante (Real)
      laenge  =  567;             // Ganzzahlige Konstante (Integer)
      name    =  'Herbert';       // String-Konstante (von Hochkommas begrenzt)
      alfa    =  #65;             // Textzeichen 'A'  (ANSI-Code 65)
      blank   =  #32;             // Textzeichen ' '  (Blank, ANSI-Code 32)
```

Einen Sonderfall stellen die **typisierten Konstanten** dar, welche eigentlich Variablen sind, denen schon bei der Typisierung ein konstanter Anfangswert zugeordnet wird (Initialisierung). Dieser kann im Gegensatz zu den echten Konstanten im Programmverlauf durch Wertzuweisungen geändert werden.

Beispiel:
```
const z : Real   = -0.73;
      s : String = 'Herbert';
      t : String = '';          // Dadurch wird t ein leerer String zugewiesen!
```

Grundsätzlich unterscheidet man zwischen **direkter** oder **indirekter Typisierung**. Im ersten Fall ist der Datentyp einer Variablen *VName* bereits durch das System vordefiniert, im zweiten Fall ist er benutzerdefiniert, d.h., er wird durch den Programmierer mit der Anweisung ***type*** festgelegt:

```
type TName = Datentyp;
var  VName : TName;
```

Im Folgenden werden im System **vordefinierte Datentypen** mit ihrer Speichergröße aufgelistet:

a) ***Numerische Typen:***

Byte	Vorzeichenlose Ganzzahl von	0	bis	255	(1 Byte).
Word	Vorzeichenlose Ganzzahl von	0	bis	65 535	(2 Byte).
LongInt	Ganzzahl von	-2 147 483 648	bis	+2 147 483 647	(4 Byte).
Integer	Entspricht ab DELPHI Version 4 dem Datentyp *LongInt*.				
int64	Ganzzahl von	-10^{18}	bis	$+10^{18}$	(8 Byte).
Single	Dezimalzahl von	$1.5*10^{-45}$	bis	$3.4*10^{+38}$	(4 Byte), 8 Stellen genau.
Double	Dezimalzahl von	$5.0*10^{-324}$	bis	$1.7*10^{+308}$	(8 Byte), 16 Stellen genau.
Extended	Dezimalzahl von	$3.4*10^{-4932}$	bis	$1.1*10^{+4932}$	(10 Byte), 20 Stellen genau.
Real	Entspricht ab DELPHI Version 4 dem Datentyp *Double*.				

b) ***Nicht numerische Typen:***

Boolean	Ein Byte mit Werten 1 oder 0, die als TRUE oder FALSE interpretiert werden. Solche Variablen bezeichnet man auch als logische Variablen.
Char	Ein Byte mit Werten 0 bis 255, die als ANSI-Zeichencode interpretiert werden. Beispielsweise hat der Buchstabe 'A' den Code 65.
String	Eine Folge von Bytes, die als Zeichenkette (String) interpretiert wird. Konstante Strings müssen immer von Hochkommas begrenzt werden (z.B. *S := 'Berg'*). Der Zugriff auf einzelne Zeichen im String erfolgt über einen Index (z.B. liefert *S[4]* das Zeichen *'g'* und die Wertzuweisung *S[2] := 'u'* erzeugt eine *'Burg'*).

Ein Beispiel eines **benutzerdefinierten Datentyps** ist der **Aufzählungstyp**, dessen Wertebereich durch eine linear geordnete Liste von Wertenamen festgelegt ist (begrenzt von runden Klammern). Solche vom Programmierer festgelegte Datentypen müssen mit dem Wort ***type*** deklariert werden.

Beispiel:
```
type Tag = (mo,di,mi,do,fr,sa,so);
var  Geburt : Tag;

Geburt := mi;          // Der Variablen Geburt wird der Wert mi zugewiesen.
```

Der so genannte **Ausschnittstyp** definiert einen zusammenhängenden Teilbereich einer vorangehenden Aufzählung:

Beispiel:
```
type  Tag     = (mo,di,mi,do,fr,sa,so);
      Werktag = mo .. sa;
```

Ausschnitte aus Integer- oder Char-Daten können auch ohne vorangehende Aufzählung der Grundelemente definiert werden:

```
type  Kleinbuch = 'a' .. 'z';
      Jahre     = 1938 .. 1945;
```

Neben diesen **einfachen Datentypen** gibt es noch **strukturierte Datentypen**. Diese bestehen aus einzelnen Elementen, die entweder einfache Datentypen oder selber wieder strukturierte Datentypen sind. Die wichtigsten Typen sind: ***RECORD***, ***ARRAY*** und ***SET***.

RECORD

Das reservierte Wort ***record*** kennzeichnet so genannte **Strukturvariablen**. Diese haben eine feste Speicherstruktur mit verschiedenen Datenfeldern.

Beispiel:
```
type  TPerson = record
          Name : String:        // Erstes Record-Feld
          Alter  : Integer;     // Zweites Record-Feld
        end;
```

Hier wird ein Datentyp zur Speicherung von Namen und Alter einer Person deklariert. Die Definition einer entsprechenden Variablen ist dadurch jedoch noch nicht erfolgt. Dazu dient beispielsweise die Anweisung

```
var  Person1, Person2 : TPerson;
```

Durch diese Anweisung wird die Datenspeicherung von zwei Personen möglich. Der Zugriff im Programm erfolgt über den Namen der Recordvariablen, gefolgt von einem Punkt (Qualifizierer) und dem Feldnamen.

```
Person1.Name  := 'Berger';
Person1.Alter  :=  53;

Person2.Name  :=  'Meier';
Person2.Alter  :=  Person1.Alter - 8;
```

Eine alternative Zugriffsmöglichkeit bietet die Anweisung ***with.*** Diese hat den Vorteil, dass der Name der Record-Variablen nicht immer angeschrieben werden muss. Es genügen die Feldnamen.

```
with Person1 do begin
    Name := 'Berger';
    Alter  := 53;
end;
```

```
with Person2 do begin
    Name := 'Meier';
    Alter  := Person1.Alter - 8;
end;
```

Ein weiterer Datentyp ist der so genannte **variante Record**, dessen Feldstruktur verschiedenartig festgelegt werden kann.

ARRAY

Datentypen für **ein-** und **mehrdimensionale Bereiche** werden durch das reservierte Wort ***array*** definiert. Dieser Typ definiert Datenbereiche, welche nur Elemente von der gleichen Art enthalten. Auf die Elemente eines Arrays wird über einen ganzahligen Index (0,1, ..., n) zugegriffen. Ein Array ist mit einem Kasten vergleichbar, dessen Schubladen fortlaufend und eindeutig nummeriert sind. Weil bei solchen Arrays die Speichergröße vor ihrer Verwendung definiert werden muss, was durch die kleinste und größte Indexzahl geschieht, nennt man sie auch **statische Arrays**. Daneben gibt es auch **dynamische Arrays**, deren Größe veränderlich ist.

Das nachfolgende Beispiel stellt ein benutzerdefiniertes, eindimensionales Array von maximal vier reellen Zahlen dar. Dabei wird dem dritten Arrayelement ein reeller Variablenwert zugewiesen.

Beispiel:
```
type  TBereich = array[1..4] of Real;
var   Spalte : TBereich;
      i : Integer;
      x, y : Real;

i  := 3;
x := 5.25;
Spalte[i] := x;
y := 2 * Spalte[i]  - 10;
```

Wie am Beispiel ersichtlich, muss hinter jedem Array-Element sein Index in eckigen Klammern stehen. Hier wird die Integervariable ***i*** als Index verwendet. Natürlich hätte man auch kürzer die Anweisung *Spalte[3] := 5.25;* schreiben können.

Folgende Deklaration definiert in direkter Weise ein **zweidimensionales Array** von reellen Zahlen mit 4 Spalten und 3 Zeilen: ***var** Matrix : array [1**..**4,1**..**3] of Real;*

SET

Schließlich muss noch der Typ ***Menge (set)*** erwähnt werden. Dabei werden einfache Elemente (Integer oder Char) innerhalb von eckigen Klammern aufgezählt. Zur Prüfung der Zugehörigkeit eines Elementes zu einer definierten Menge wird der logische Bezugsoperator ***in*** verwendet, der TRUE oder FALSE sein kann. Mit den Operatoren +, * und – können Vereinigung, Durchschnitt und Differenz von zwei Mengen gebildet werden.

Beispiel:
```
type   TBuch  =  set of Char;
const  Vokale  :  TBuch =  ['a','e','i','o','u'];      // typisierte Konstante
       Konso   :  TBuch =  ['b','c','d'];
var    Test    :  Boolean;
       Letter  :  Char;
       Misch   :  TBuch;

Letter  := 'b';                                         // einfache Wertzuweisung
Test    := Letter in  Vokale;                           // wird hier zu FALSE!
Misch   := Vokale + Konso;                              // Mengen-Vereinigung
Test    := Letter in  Misch;                            // wird hier zu TRUE!
```

Wichtiger Hinweis: Bei Dateneingaben über die Tastatur (mit Inputboxen oder Editfeldern) und bei Datenausgaben am Bildschirm (mit Messageboxen oder Editfeldern) können nur Zeichenketten (Strings) verwendet werden. Diese können dann im Programm in Zahlen umgewandelt werden.

[2.03] Operatoren

Zur Bildung von Ausdrücken werden Operatoren benötigt. Je nach Datentyp unterscheidet man verschiedene Arten von Operatoren: arithmetische Operatoren (+, -, *, /), logische Operatoren (not, and, or, xor), Mengen-Operatoren (+, -, *) und schließlich auch Zeichenketten-Operatoren zum Verknüpfen von Strings (*'Albert' + ' ' + 'Berger'* ergibt *'Albert Berger'*).

Wichtige zweistellige arithmetische Operationen sind: *die Addition (a + b), die Subtraktion (a - b), die Multiplikation (a * b), die reellzahlige Division (a / b), die ganzzahlige Division (a div b) und der Divisionsrest (a mod b).* Der Typ des Ergebnisses hängt von den Typen der Operanden ab. Die ganzzahlige Division rundet den Quotienten nicht, sondern schneidet die Nachkommastellen ab. Grundsätzlich sollten nur **kompatible Datentypen** miteinander verknüpft werden.

[2.04] Entscheidungen

Die wichtigsten Entscheidungen sind die **Alternativ-Auswahl** und die **Mehrfach-Auswahl**. Dabei werden **Bedingungen** definiert, von denen es dann abhängt, welcher Anweisungsblock ausgeführt wird. Entscheidungen stellen daher echte Programmverzweigungen dar. Erst dadurch kann ein Programmverlauf den individuellen Wünschen des Programmierers bzw. der jeweiligen Programmlogik angepasst werden. Was versteht man nun unter einer Bedingung?

Variablen vom Typ **Boolean** werden in einem Byte gespeichert. Je nach dem Wert dieses Bytes (1 oder 0) wird die Variable entweder als TRUE oder als FALSE bestimmt.

Solche Boolean-Variablen können mit Hilfe **logischer Operatoren** (not, or, and, xor) miteinander verknüpft werden. Die jeweilige Bestimmung des Wahrheitsgehaltes dieser logischen Ausdrücke erfolgt vom System entsprechend den Konventionen der Aussagenlogik.

a	b	not a	a and b	a or b	a xor b
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Beispiel:

```
var  a, b, c : Boolean;

a := TRUE;
b := not a;                    // ergibt hier FALSE
c := not (a and b);            // ergibt hier TRUE
```

Unter einer **Bedingung** versteht man einen Wertevergleich. Die Resultate der zweistelligen Vergleichsrelationen (=, <, >, <=, >=, <>) zwischen passenden Variablen oder Ausdrücken werden intern als **Boolean**-Werte abgespeichert, die entweder TRUE oder FALSE sind.

Beispiel:

```
var  a, b, c : Integer;
     d : Char;
     e : Boolean;

a := 3; b := 4; c := 5; d := 'B';

e := (a < b);                  // ergibt hier TRUE
e := (a > b) or (a > c);       // ergibt hier FALSE
e := (d = 'A');                // ergibt hier FALSE
```

[2.04.1] Die Alternativ-Auswahl

```
if Bedingung then
  begin
    Anweisungsblock_1
  end
else
  begin
    Anweisungsblock_2
  end;
```

Der erste Anweisungsblock wird nur dann ausgeführt, wenn die **Bedingung erfüllt** (TRUE) ist. Der zweite Anweisungsblock hingegen wird nur ausgeführt, wenn die **Bedingung nicht erfüllt** (FALSE) ist. Es können beliebig viele solcher Entscheidungen ineinander geschachtelt werden, wobei eine Bedingung jeweils nur dann abgefragt wird, wenn die vorangegangene Bedingung erfüllt ist. Der ***else***-Teil kann auch weggelassen werden, er ist optional. Vor ***else*** darf kein "**;**" stehen! Grundsätzlich muss jeder Anweisungsblock, der aus mehreren aufeinander folgenden Anweisungen besteht, mit den reservierten Worten ***begin*** und ***end*** eingegrenzt werden. Wenn der Block nur eine einzige Anweisung enthält, dann können ***begin*** und ***end*** auch weggelassen werden.

Beispiel:

```
if note = 1 then
  begin
    Canvas.TextOut(20,20,'Sehr Gut');          // Positionierte Textausgabe
    Canvas.TextOut(20,40,'Gratuliere');        // auf der Formularleinwand.
  end                                          // Dabei geben die beiden
else                                           // Zahlenparameter die X-
  begin                                        // und Y-Koordinaten in Bezug
    if note = 2 then                           // auf die linke, obere Ecke des
      Canvas.TextOut(20,20,'Gut')              // Formulares an (in Pixeln).
    else
      Canvas.TextOut(20,20,'Mehr Lernen');
  end;
```

[2.04.2] Die Mehrfach-Auswahl

Die Differenz zu der Einfach-Auswahl liegt darin, dass bei der Mehrfach-Auswahl, wie es der Name schon besagt, mehrere Wertemöglichkeiten einer Variablen (*selekto*r) hintereinander abgefragt und unterschiedlich beantwortet werden können.

```
case selektor of
   Wert_1: begin
              Anweisungsblock_1
           end;
   Wert_2: begin
              Anweisungsblock_2
           end;
   . . . . . .
   . . . . . .
else  begin
         Anweisungsblock_0
      end;
end;
```

Welcher Anweisungsblock durchgeführt wird, hängt bei dieser Entscheidungsstruktur vom Wert des Selektors ab. Dieser Selektor ist eine Variable, deren Datentyp einfach und abzählbar sein muss (ordinal). Der abgefragte Wert kann ein einzelner Wert oder eine Wertaufzählung sein.

Beispiel:
```
case note of
      1 : begin
            Canvas.TextOut(20,20,'Sehr Gut');
            Canvas.TextOut(20,40,'Gratuliere');
          end;
      2 : Canvas.TextOut(20,20,'Gut');
  3,4,5 : Canvas.TextOut(20,20,'Mehr Lernen')
else
     Canvas.TextOut(20,20,'Das ist keine Note');
end;
```

[2.05] Wiederholungen

Mit Hilfe von Wiederholungsschleifen wird ein Anweisungsblock, **abhängig** von einer bestimmten **Bedingung**, **mehrmals** durchlaufen. Als Beispiele sind hier die "For-Schleife", die "While-Schleife" und die "Repeat-Schleife" angeführt.

[2.05.1] Die zählende Schleife (For-Schleife)

```
for ZVar := Wert_1 to Wert_2 do
begin
  Anweisungsblock
end;
```

Die einfache Zählvariable *ZVar* wird zu Beginn auf *Wert_1* gesetzt. Nach jeder Ausführung des Anweisungsblockes wird die Zählvariable um Eins erhöht (inkrementiert). Wenn die Zählvariable den *Wert_2* überschritten hat, dann wird der Programmablauf beim ersten Befehl nach der Schleife fortgesetzt. Wird das Wort ***to*** durch ***downto*** ersetzt, dann erniedrigt sich die Zählvariable bei jedem Schleifendurchgang um Eins (Dekremetierung, absteigende Zählschleife).

Beispiel: Es sollen in einer kleinen Programmroutine die ersten zehn natürlichen Zahlen summiert werden. Dazu definiert man zunächst zwei Variablen vom Integer-Typ. Die Variable ***i*** dient als Zählvariable und in der Variablen ***sum*** soll die Summe gespeichert werden.

```
var i, sum : Integer;

sum := 0;
for i := 1 to 10 do
begin
   sum := sum + i;
end;
```

[2.05.2] Die kopfgesteuerte Schleife (While-Schleife)

```
while Bedingung do
begin
  Anweisungsblock
end;
```

Die Ausführung des Anweisungsblockes wird **so lange** wiederholt, **als** die Bedingung erfüllt ist (Laufbedingung). Die Prüfung der Bedingung erfolgt jeweils am Schleifenanfang. Ist die Bedingung nicht erfüllt, so wird der Programmablauf beim ersten Befehl nach der Schleife fortgesetzt. Damit dies auch eintritt, muss im Anweisungsblock eine in der Bedingung abgefragte Variable entsprechend verändert werden - andernfalls wird die Schleife nie abgebrochen (Endlos-Schleife)!

Beispiel:

```
var i, sum : Integer;

sum := 0;
i := 0;
while (i < 10) do
begin
  i := i + 1;
  sum := sum + i;
end;
```

[2.05.3] Die fußgesteuerte Schleife (Repeat-Schleife)

```
repeat
  Anweisungsblock
until Bedingung;
```

Die Ausführung des Anweisungsblockes wird **so lange** wiederholt, **bis** die Bedingung erfüllt ist (Abbruchbedingung). Die Prüfung der Bedingung erfolgt jeweils am Schleifenende. Ist die Bedingung erfüllt, dann wird der Programmablauf beim ersten Befehl nach der Schleife fortgesetzt. Damit dies auch eintritt, muss im Anweisungsblock eine, in der Bedingung abgefragte Variable entsprechend verändert werden - andernfalls wird die Schleife nie abgebrochen (Endlos-Schleife)!

Beispiel:

```
var i, sum : Integer;

sum := 0;
i := 1;
repeat
   sum := sum + i;
   i := i + 1;
until (i > 10);
```

Erwähnenswert ist die Tatsache, dass der wiederholte Anweisungsblock innerhalb der Schleife nicht durch ***begin*** und ***end*** eingegrenzt werden muss. Es genügt die Begrenzung durch ***repeat*** und ***until***. Schließlich gibt es noch die nützliche Anweisung ***break***, welche den unbedingten Ausstieg aus einer Wiederholungsschleife erzwingt.

[2.06] Unterprogramme

Prozeduren oder Funktionen sind Unterprogramme (Routinen) eines Programmes. Sie werden in einer Unit codiert und können dann beliebig oft mit ihren Namen aufgerufen werden, sofern die Unit in das Programm eingebunden ist. Die Kopfzeile von Prozeduren (*Header*) beginnt mit dem reservierten Wort ***procedure,*** gefolgt von dem ***Namen*** der Routine. In Klammer können nach dem Namen bestimmte Variablen geschrieben werden. Diese Variablen werden als **Parameter** bezeichnet. Sie dienen der Datenübermittlung zwischen Haupt- und Unterprogramm. Beim Aufruf einer Prozedur müssen die aktuell übergebenen Parameter in Anzahl und Datentyp mit den vorher formal vereinbarten Parametern übereinstimmen. Die Parameter werden im Unterprogramm mit ihren Namen angesprochen und verarbeitet.

Codierung der Prozedur PName:

```
procedure PName (formale Parameter);
begin
  Anweisungsblock
end;
```

Aufruf der Prozedur im Programm:

```
PName (aktuelle Parameter);
```

Eine **Funktion** unterscheidet sich von einer **Prozedur** zunächst dadurch, dass genau ein Wert an das aufrufende Programm zurückgegeben wird. Dazu muss innerhalb des Funktionscodes dem Funktionsnamen oder der Variablen ***Result*** ein passender Wert zugewiesen werden. Der Aufruf der Funktion schließlich ist keine eigenständige Anweisung, sondern kann nur immer im rechts stehenden Teil einer Wertzuweisung oder als Teil eines Ausdruckes vorkommen.

Codierung der Funktion FName:

```
function FName (formale Parameter): VTyp;
begin
   Anweisungsblock
  FName  :=  . . . . . ( oder Result := . . . . . )
end;
```

Aufruf der Funktion im Programm:

```
var VarName: VTyp;
VarName := FName (aktuelle Parameter);
```

Die Parameter von Prozeduren und Funktionen sind grundsätzlich lokale Variablen, die nur innerhalb des Unterprogrammes zur Verfügung stehen. Sie dienen der Übergabe von Werten und werden daher auch **Werteparameter** genannt. ("*call by value*"). Ihre Abspeicherung erfolgt vorübergehend auf einem Stapelspeicher (Stack), der nach Beendigung der Prozedur wieder abgeräumt wird. Soll der Wert einer Variablen auch nach Beendigung des Unterprogrammes im Hauptprogramm erhalten bleiben, so muss bei der Codierung der Routine in der Parameterliste vor dem entsprechenden Parameternamen das reservierte Wort **var** gesetzt werden. Dadurch wird der Parameter nicht auf dem Stackspeicher abgelegt, sondern eine Adressierung zu einer schon vorher im Hauptprogramm reservierten Variablen hergestellt. Solche Parameter werden demgemäß **Variablenparameter** genannt. ("*call by reference*"). Zur Illustration sollen einer Prozedur ein Werteparameter **a** und ein Variablenparameter **b** übergeben werden. In der Prozedur werden beide Werte abgeändert. Nach dem Aufruf im Hauptprogramm ist jedoch nur die zweite übergebene Variable **b** verändert.

Codierung einer Prozedur *PName*:

```
type VType  =  Integer;
var  a,b,c,d  :  VType;

procedure PName(x : VType; var  y :  VType);
begin
   x := 1;
   y := 1;
end;
```

Aufruf der Prozedur im Programm:

```
a := 0;
b := 0;
PName(a,b);
c := a;
d := b;
```

Auf den globalen Variablen ***c*** und ***d*** liegen jetzt die Werte 0 und 1.

Als Übungsbeispiel soll im ersten Fall die Fläche eines Rechtecks mit den Seiten 8 und 5 in Form einer Funktion berechnet werden. Im zweiten Fall wird die Fläche eines Rechtecks mit Hilfe einer Prozedur ermittelt.

Fall 1, Codierung:

```
function flaeche(a, b : Real) : Real;
begin
  Result := a * b;
end;
```

Fall 1, Aufruf:

```
var  f : Real;
f := flaeche(8,5);
```

Fall 2, Codierung:
```
procedure rechteck(a,b : Real; var fla : Real);
begin
  fla := a * b;
end;
```

Fall 2, Aufruf:
```
var flaeche : Real;
rechteck(8, 5, flaeche);
```

In DELPHI ist jede **Ereignis-Behandlungsroutine** eine Prozedur, wobei sich der Programmierer normalerweise nicht um den Rahmencode (*Kopfzeile; begin ... end;*) der Prozedur kümmern muss. Dieser wird vom System automatisch in den Quellcode der Unit geschrieben.

Außer den vom User **selbst definierten Routinen** gibt es viele im System **vordefinierte Funktionen und Prozeduren**, welche von DELPHI zur Verfügung gestellt werden. Diese Routinen sind in verschiedenen Units verpackt. Erwähnenswert ist die Unit ***Math***, die zusätzliche mathematische Funktionen bereitstellt. Die Standardroutinen lassen sich in folgende Hauptkategorien einteilen:

(1) Arithmetische Routinen: Sie dienen vor allem für Zahlenberechnungen (Unit *System*)

function **Pi**: Extended;	Liefert die Zahl Pi (3.1415...).
function **Abs**(x: Real/Integer): Real/Integer;	Liefert den Absolutwert von x.
function **Int**(x: Extended): Extended;	Liefert den ganzzahlig abgeschnittenen Wert von x.
function **Frac**(x: Extended): Extended;	Liefert den Nachkommateil (x - int(x)) der Zahl x.
function **Trunc**(x: Extended): Integer;	Liefert den Vorkommateil (x - frac(x)) der Zahl x.
function **Round**(x: Extended): Integer;	Liefert den ganzzahlig gerundeten Wert von x.
function **Sqrt**(x: Extended): Extended;	Liefert die Quadratwurzel von x.
function **Sin**(x: Extended): Extended;	Liefert den Sinus-Wert von x (Bogenmaß).
function **Cos**(x: Extended): Extended;	Liefert den Cosinus-Wert von x (Bogenmaß).
function **ArcTan**(x: Extended): Extended;	Liefert den ArcusTangens von x.
function **Exp**(x: Extended): Extended;	Liefert den Exponentialwert von x (e^x).
function **Ln**(x: Extended): Extended;	Liefert den natürlichen Logarithmus von x.
function **Random**(n: Word): Word;	Liefert eine ganze Zufallszahl x mit 0 <= x < n.
function **Odd**(x: LongInt): Boolean;	Liefert TRUE, wenn die ganze Zahl x ungerade ist.
procedure **Inc**(var x [; n] : LongInt);	Erhöht die Zahl x um Eins (bzw. optional um n).

(2) Stringfunktionen: Verarbeitungsroutinen von Zeichenketten (Units *System*, *SysUtils*)

function **Length**(S: String): Integer;	Liefert die Länge des Strings S.
function **Pos**(T,S: String): Integer;	Liefert die erste Position von String T im String S.
function **Trim**(S: String): String;	Entfernt führende und nachfolgende Leerzeichen.
function **UpperCase**(S: String): String;	Umwandlung in Goßbuchstaben.
function **LowerCase**(S: String): String;	Umwandlung in Kleinbuchstaben.
function **Concat**(S,T: String); String;	Verkettung von S und T (auch mittels R := S + T).
function **Copy**(S: String; P,L: Integer): String;	Liefert von dem String S jenen Teilstring, der an der Position P beginnt und L Zeichen lang ist.
function **Delete**(var S: String; P,L: Integer);	Löscht im String S ab Position P genau L Zeichen.
function **Insert**(T: String; var S: String; P: Integer);	Fügt T in S an der Position P ein.
function **Val**(S: String; var Z: Real; var P: Integer);	Wandelt den String S in eine reelle Zahl Z um und liefert für P entweder 0 oder eine Fehlernummer > 0. Bei einem Fehler ist Z immer Null.
procedure **Str**(Z:n:d: Real; var S: String);	Wandelt die reelle Zahl Z in einen String S um, mit n Stellen und d Dezimalstellen.
function **StrToInt**(S: String): Integer;	Ungeprüfte Umwandlung von String in Zahl.
function **IntToStr**(N: Integer): String;	Unformatierte Umwandlung von Zahl in String.
function **UpCase**(C: Char): Char;	Umwandlung des Zeichens C in Goßbuchstaben.
function **Ord**(C: Char): Integer;	Liefert den ANSI-Code des Zeichens C.
function **Chr**(N: Integer): Char;	Liefert das Zeichen C mit dem ANSI-Code C. (auch mittels C := #N, z.B. Blank := #32).

(3) Weitere Routinen für Datum, Uhrzeit, für Datei- und Speicherverwaltung usw.

[Anhang] DREI VARIATIONEN VON EINEM PROGRAMM

Die Aufgabenstellung

Ziel ist es, eine Unit zu schreiben, welche selbst definierte Funktionen zur Berechnung von Fläche und Umfang eines Dreiecks beinhaltet. Dazu ist es notwendig, eine Benutzeroberfläche für die Eingabe der drei Seitenlängen des Dreiecks und die Ausgabe des Umfangs und der Fläche zu gestalten.

Es soll eine eigene Unit erzeugt werden. Diese wird dann in die dem Hauptformular zugehörige Unit eingebunden. Die Unit für die Berechnung von Umfang und Fläche eines Dreiecks könnte natürlich auch in jedes andere Projekt eingebunden werden. Da jeweils nur ein Ergebnis, nämlich Fläche bzw. Umfang, zurückgeliefert wird, ist es sinnvoller, Funktionen anstelle von Prozeduren zu schreiben. Als Parameter sind die drei Seitenlängen des Dreiecks zu übergeben.

Das gestellte Ziel soll in drei Programm-Variationen erreicht werden. Die erste Variation "***drei01***" enthält weder eine selbst definierte Unit noch selbst definierte Funktionen. Sie dient nur der einfachen Eingabe, Verarbeitung und Ausgabe von Daten. Die zweite Variation "***drei02***" hingegen enthält bereits selbstdefinierte Funktionen und die dritte Variation "***drei03***" schließlich stellt eine eigene zusätzliche Unit "***drei03_uu***" zur Verfügung.

Der prinzipielle Aufbau einer Unit

Jede Unit ist eine eigenständige Untereinheit (Modul) des Programmprojektes. Sie enthält eine Sammlung von Datentypen, Konstanten, Variablen, Prozeduren und Funktionen, welche in verschiedene andere Programme eingebunden werden können. Eine Unit setzt sich aus bestimmten Abschnitten zusammen, welche in einer vorgeschriebener Reihenfolge angeführt werden müssen. Der Aufbau ist folgendermaßen strukturiert:

```
unit NAME
        { Unit-Kopf  }
interface
        { Schnittstellenteil }
. . . . . .
. . . . . .

implementation
        {  Implementierungsteil }
. . . . . .
. . . . . .

initialization
        {  optionaler Initialisierungsteil }
finalization
        {  optionaler Finalisierungsteil }
end.    { Ende der Unit }
```

Unit-Kopf: Hier wird definiert, unter welchem Namen die Unit im System angesprochen werden kann. Eine Unit beginnt immer mit dem reservierten Wort ***unit***, nach welchem ihr Name folgt.

Interface-Abschnitt: Durch das Schlüsselwort ***interface*** wird der Schnittstellenteil eingeleitet. Hier werden Datentypen, Variablendefinitionen und die **Kopfzeilen** von Prozeduren und Funktionen angegeben. Alle diese Sprachelemente können von außen angesprochen werden. Sie wirken **global**.

Implementation-Abschnitt: In diesem, mit dem Wort ***implementation*** eingeleiteten Teil befindet sich der eigentliche Programmcode jener Prozeduren und Funktionen, deren Kopfzeilen im Schnittstellenteil beschrieben sind. Außerdem können hier auch **lokale** Variable und Prozeduren definiert werden.

Initialisierung-Abschnitt: Dieser Teil ist optional. Falls es notwendig sein sollte, eine Unit beim Programmstart zu initialisieren, d.h. bestimmte Variable mit Anfangswerten zu belegen oder zusätzlichen Speicher vom Betriebssystem anzufordern, so kann das hier geschehen.

Finalisierung-Abschnitt: Dieser Abschnitt ist das Gegenstück zur Initialisierung. Er ist ebenfalls optional und wird am Programmende aufgerufen. Hier können nötige Aufräumarbeiten erledigt werden.

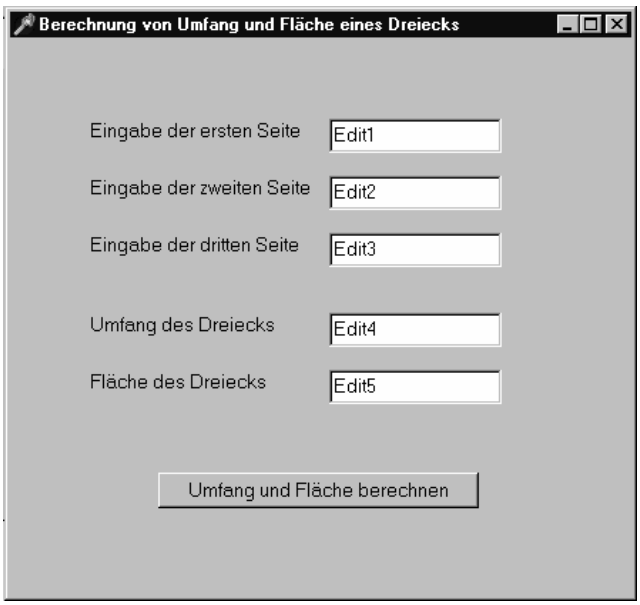
Mit dem Schlüsselwort ***end*** und dem folgenden ***Punkt*** ist die Codierung der Unit abgeschlossen. Die Unit kann mit Hilfe des Schlüsselwortes ***uses*** in jedes andere Programm eingebunden werden.

[Version 1] Das Programm "*drei01*" (Eingabe, Verarbeitung und Ausgabe von Daten)

Die Formulargestaltung

Wir fangen wieder mit der **ersten Phase** der Programmentwicklung an, dem visuellen Entwurf der Bedienungsoberfläche. Wir brauchen für unsere Dreiecksberechnungen drei Eingabefelder und zwei Ausgabefelder. Wir platzieren also insgesamt fünf Editierfelder auf das Startformular und setzen jeweils eine Beschriftung (Label) davor, um dem Benutzer die Bedeutung der Editierfelder anzuzeigen. Außerdem ist ein Schaltknopf (Button) zu platzieren, sodass erst durch einen einfachen Mausklick darauf die Berechnungen ausgelöst werden.

In der **zweiten Phase** der Programmentwicklung weisen wir nun wieder den Objekten die gewünschten Eigenschaften zu. Wir verändern die Maße der Objekte und schreiben in die ***Labels*** über die Eigenschaft ***Caption*** die gewünschte Beschriftung. Sinnvollerweise sollte die Eigenschaft ***ReadOnly*** der beiden Ausgabefelder auf TRUE gesetzt werden, um die Eingabe eines Textes durch den User in diesen Feldern zu verhindern (Schreibschutz). Der Titel des Formulars kann ebenfalls über die Eigenschaft ***Caption*** geändert werden. Die Beschriftung des ***Buttons*** kann auf gleiche Weise erzeugt werden. Die Oberfläche erhält dann folgendes Aussehen:



Wie ersichtlich, enthält das Formular insgesamt zwölf Komponenten (das Formular *Form1* selbst, die fünf Anzeigefelder *Label1* bis *Label5*, die fünf Editierfelder *Edit1* bis *Edit5* und noch einen Schaltknopf *Button1*). Von diesen Komponenten können im Objektinspektor gewünschte Eigenschaften eingestellt bzw. verändert werden.

Nun kommen wir zur **dritten Phase** der Programmentwicklung, dem Verknüpfen von Objekten mit Ereignissen. Durch das ***OnClick***-Ereignis des ***Buttons*** sollen die Berechnungen ausgelöst und die Ergebnisse in entsprechenden Ausgabefeldern angezeigt werden. In die Programmschablone der Ereignisbehandlungsroutine des ***OnClick***-Ereignisses im Quelltext gelangt man durch einen Doppelklick auf den Schaltknopf oder mit Hilfe des Objektinspektors.

Die einzelnen visuellen Objekte haben dann folgende Eigenschaften (properties):

Objektname	Eigenschaft	Wert
Form1	Caption	'Berechnung von Umfang und Fläche eines Dreiecks'
	Color	clInfoBk
	Font	MS SansSerif, 10, Standard
	Position	poDesktopCenter
Label1	Caption	'Eingabe der ersten Seite'
Label2	Caption	'Eingabe der zweiten Seite'
Label3	Caption	'Eingabe der dritten Seite'
Label4	Caption	'Umfang des Dreiecks'
Label5	Caption	'Fläche des Dreiecks'
Edit1	Text	
Edit2	Text	
Edit3	Text	
Edit4	Text	
	ReadOnly	True
Edit5	Text	
	ReadOnly	True
Button1	Caption	'Umfang und Fläche berechnen'

Der Komponente ***Button1*** wird folgendes Ereignis zugeordnet:

Benutzer-Aktion	Windows-Ereignis	Programm-Routine	Programm-Reaktion
Mausklick auf Button1	OnClick	Button1Click	Berechnungen ausführen

Im Folgenden wird das komplette Listing sowohl des Hauptprogrammes als auch der Unit wiedergegeben. Die neben den Anweisungen stehenden Kommentarzeilen (beginnend mit //) dienen zur Erklärung der einzelnen Programmelemente.

```
program drei01;
// Hauptprogramm (Projektdatei "drei01.dpr")

uses Forms,
     drei01_u in 'drei01_u.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

```
unit drei01_u;
// Dereicksberechnungen, Version 01 (c) H.Paukert

interface

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

```
type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Edit4: TEdit;
    Edit5: TEdit;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
    private { Private declarations }
    public  { Public declarations }
  end;

var Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
// Eingabe, Verarbeitung, Ausgabe und Fehlerprüfung
var a,b,c : Real;
    s,t   : Real;
    Umfang,Flaeche: Real;
begin
  try
    a := StrToFloat(Edit1.Text);
    b := StrToFloat(Edit2.Text);
    c := StrToFloat(Edit3.Text);

    if ((a+b)<=c) or ((a+c)<=b) or ((b+c)<=a) then Umfang := 0
    else Umfang := a+b+c;

    s := (a+b+c)/2;
    t := s*(s-a)*(s-b)*(s-c);
    if t > 0 then Flaeche := Sqrt(t) else Flaeche := 0;

    Edit4.Text := FloatToStr(Umfang);
    Edit5.Text := FloatToStr(Flaeche);
  except
    MessageBox(0,'Falsche Eingabe','Fehler',16);
    Edit4.Text := '';
    Edit5.Text := '';
  end;
end;

end.
```

Zuerst werden die in den Edit-Feldern eingegebenen Texte auf die entsprechenden numerischen Variablen abgespeichert. Ein Problem dabei ist, dass die verwendeten Variablenwerte vom Zahlentyp ***Real*** sind, die Editierfelder aber nur Daten vom Typ ***String*** aufnehmen können. DELPHI stellt verschiedene Funktionen für Typkonvertierungen zur Verfügung. In unserem Fall benötigen wir die vom System vordefinierten Funktionen ***StrToFloat*** und ***FloatToStr***.

Die Anweisung ***a := StrToFloat(Edit1.Text)*** wandelt zunächst den im Editierfeld von ***Edit1*** eingegebenen Text in eine Gleitkommazahl um und weist diese Zahl der reellen Variablen ***a*** zu. Die Anweisung ***Edit4.Text := FloatToStr(Umfang)*** funktioniert in umgekehrter Weise. Sie wandelt den Zahlenwert der Variablen ***Umfang*** in einen String um und schreibt diesen in ***Edit4***.

Bei der Berechnung des Umfanges muss geprüft werden, ob die drei Seiten ***a, b, c*** wirklich ein Dreieck bilden, d.h. die Summe von zwei Seiten muss länger als die dritte Seite sein. Dies wird mit Hilfe einer ***if***-Anweisung realisiert. Dabei sind die einzelnen Wertvergleiche durch ein logisches Oder (***or***) miteinander verknüpft.

Für die Berechnung der ***Flaeche*** ist es sinnvoll, die zwei Hilfsvariablen ***s*** und ***t*** einzuführen. Die Berechnung erfolgt mit der Heron'schen Flächenformel. Zuvor wird noch durch eine Entscheidung sichergestellt, dass nur aus einer positiven Zahl die Wurzel gezogen wird. Für die Berechnung einer Wurzel stellt DELPHI die vordefinierte Funktion ***Sqrt*** zur Verfügung. Dieser Funktion muss als Parameter die Zahl, von welcher die Wurzel gezogen wird, übergeben werden.

Die Fehlerbehandlung (Exception-Handling)

Was passiert, wenn der User unerlaubte Werte eingibt? Wenn also zum Beispiel eine Seitenlänge nicht eingegeben wird oder wenn die Eingabe nicht numerisch ist? Oder wenn die Berechnung der Fläche nicht möglich ist, weil eine Seite länger ist als die beiden anderen zusammen? In diesen Fällen wird eine der ***StrToFloat***-Funktionen oder eine der ***FloatToStr***-Funktionen mit einer Fehlermeldung abbrechen. Ein solcher Fehler, auch Ausnahme (Exception) genannt, sollte abgefangen werden, bevor es zu einem Abbruch des Programmes kommt. Eine Möglichkeit, Fehler abzufangen, sind so genannte ***Try-Except***-Blöcke. Damit das Ganze auch in der Entwicklungsumgebung reibungslos funktioniert, muss man in dem Menü <*Tools/Debugger-Optionen/Sprach-Exceptions*> den Schalter <*Bei Delphi-Exceptions anhalten*> deaktivieren.

Try
Programmcode, der einen Fehler auslösen kann;
Except
Fehlerbehandlungsroutine, welche nur durchlaufen wird, wenn vorher ein Fehler auftritt;
End;

In dieser Fehlerbehandlungsroutine sollte eine entsprechende Meldung an den User erfolgen, was beispielsweise über eine einfache ***Dialogbox*** geschehen kann. Der entsprechende Befehl lautet:

MessageBox(0,'Falsche Eingaben','Fehler',16);

Dabei ist der erste Wert die interne Formular-Bezugsnummer (Formular-Handle), der zweite die ausgegebene Meldung, der dritte die Box-Beschriftung, und der vierte Wert ist die Kennnummer des angezeigten Symbols. (16 = Stoppsymbol "X").

Hinweis: Die Fehlerbehandlung bezieht sich nicht nur auf Fehler, welche im direkt eingegrenzten Befehlsblock auftreten, sondern auch auf dort aufgerufene Prozeduren und Funktionen, auch wenn diese in einer anderen Unit codiert sind. Es bleibt noch anzumerken, dass zur Weiterarbeit an einem Programm nach einem Feher das Menü <*Start/Programm-Reset*> in der Entwicklungsumgebung aufgerufen werden muss.

Zum Abschluss wird die ***Unit1*** über das Menü <*Datei/Speichern unter*> unter dem neuen Namen ***drei01_u*** abgespeichert, das gesamte Projekt hingegen unter dem Namen ***drei01***. Sodann muss das Programm mittels Menü <*Start/Start*> noch getestet und falsch geschriebene Anweisungen ausgebessert werden. Wenn alles fehlerfrei funktioniert, muss noch einmal alles abgespeichert und schließlich der Quelltext des Programmes in die ausführbare Programmdatei (***drei01.exe***) compiliert werden. Dies geschieht mittels Menü <*Projekt/Compilieren*>.

[Version 2] Das Programm "*drei02*" (Verwendung selbst definierter Funktionen)

```
unit drei02_u;
// Dereicksberechnungen, Version 02 (c) H.Paukert

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

```
type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Edit4: TEdit;
    Edit5: TEdit;
    Button1: TButton;

    procedure Button1Click(Sender: TObject);

    private { Private declarations }
    public  { Public declarations }
  end;

var Form1: TForm1;


implementation
{$R *.DFM}


function Umfang(x,y,z: Real): Real;
// Umfang des Dreiecks
begin
  if ((x+y)<=z) or ((x+z)<=y) or ((y+z)<=x) then Result := 0
  else Result := x+y+z;
end;

function Flaeche(x,y,z: Real): Real;
// Fläche des Dreiecks
var s,t: Real;
begin
  s := (x+y+z)/2;
  t := s*(s-x)*(s-y)*(s-z);
  if t > 0 then Result := Sqrt(t) else Result := 0;
end;


procedure TForm1.Button1Click(Sender: TObject);
// Eingabe, Verarbeitung, Ausgabe und Fehlerprüfung
var a,b,c: Real;
begin
  try
    a := StrToFloat(Edit1.Text);
    b := StrToFloat(Edit2.Text);
    c := StrToFloat(Edit3.Text);
    Edit4.Text := FloatToStr(Umfang(a,b,c));
    Edit5.Text := FloatToStr(Flaeche(a,b,c));
  except
    MessageBox(0,'Falsche Eingabe','Fehler',16);
    Edit4.Text := '';
    Edit5.Text := '';
  end;
end;

end.
```

Wie aus dem Listing ersichtlich ist, werden Umfang und Fläche in dieser Programmvariation mit Hilfe von selbst geschriebenen Funktionen ermittelt. Dabei muss der Quellcode der Funktionen vor deren Aufruf programmiert werden. Der Aufruf wird mit Hilfe des Funktionsnamens durchgeführt, wobei die formalen Parameter *x*, *y*, *z* durch die aktuellen Parameter *a*, *b*, *c* zu ersetzen sind.

[Version 3] Das Programm "*drei03*" (Aufbau und Verwendung selbst definierter Units)

Bevor wir mit der Entwicklung unseres eigentlichen Programms beginnen, müssen wir die Unit, welche die gewünschten Funktionen anbietet, schreiben. Um eine neue Unit anzulegen, gehen wir in das Menü <*Datei / Neu*> und klicken dort auf <*Unit*>. Wir brauchen dann nur mehr unsere Anweisungen in die vorgefertigte Programmschablone schreiben. Zum Schluss muss mit Hilfe des Menüs <*Projekt / Dem Projekt hinzufügen*> die Unitdatei ***drei03_uu*** zum Projekt ***drei03*** hinzugefügt werden. Mit <*Strg*><*F12*> kann zwischen den Quellcodes der Units gewechselt werden.

```
unit drei03_uu;
// Dereicksberechnungen, Version 03 (c) H.Paukert
// Zweite Unit von "drei03"

interface

  function Umfang(x,y,z : single): Real;
  function Flaeche(x,y,z : single): Real;

implementation

function Umfang(x,y,z: Real): Real;
// Umfang des Dreiecks
begin
  if ((x+y)<=z) or ((x+z)<=y) or ((y+z)<=x) then Result := 0
  else Result := x+y+z;
end;

function Flaeche(x,y,z: Real): Real;
// Fläche des Dreiecks
var s,t: Real;
begin
  s := (x+y+z)/2;
  t := s*(s-x)*(s-y)*(s-z);
  if t > 0 then Result := Sqrt(t) else Result := 0;
end;

end.
```

Zuerst wird es sinnvoll sein, den Namen der Unit auf einen aussagekräftigeren umzuändern. In unserem Fall nennen wir die Unit ***drei03_uu***. Dann muss im Interface-Abschnitt, also nach dem Schlüsselwort ***interface***, die Schnittstelle nach außen definiert werden. Dazu wird zuerst mit ***function*** angegeben, dass es sich um Funktionen handelt. Dann folgt der Funktionsnamen und dahinter werden, in Klammern und mit Beistrichen getrennt, die notwendigen Eingabeparameter und deren Typ festgelegt. In unserem Fall sind die notwendigen Parameter die drei Seitenlängen, welche vom reellwertigen Typ ***real*** sind. Zum Schluss muss noch der Typ des Ergebnisses der Funktion definiert werden, ebenfalls ***real***. Im nachfolgenden Implementierungsteil wird der Programmcode der Funktionen erstellt. Dieser ist identisch mit dem Quellcode aus Variation "***drei02***". Die Unit ist mit dem Schlüsselwort ***end.*** abzuschließen. Nun muss noch die Unit mit Hilfe des Menüs <*Datei/Speichern unter*> unter ihrem Namen ***drei03_uu*** abgespeichert werden.

Wie im folgenden Listing ersichtlich ist, muss in diese erste Unit die zweite Unit ***drei03_uu*** mit Hilfe der Anweisung ***uses*** eingebunden werden.

```
program drei03;
// Hauptprogramm (Projektdatei "drei03.dpr")

uses Forms,
     drei03_u in 'drei03_u.pas' {Form1},
     drei03_uu in 'drei03_uu.pas';

{$R *.RES}
```

```
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

unit drei03_u;

```
// Dereicksberechnungen, Version 03 (c) H.Paukert
// Erste Unit von "drei03"

interface

uses Windows, Messages, SysUtils, Classes,
     Graphics, Controls, Forms, Dialogs, StdCtrls,
     drei03_uu;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Edit4: TEdit;
    Edit5: TEdit;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
    private { Private declarations }
    public  { Public declarations }
  end;

var Form1: TForm1;


implementation
{$R *.DFM}

// Die Unit "drei03_uu" könnte mit "uses" auch hier eingebunden werden!

procedure TForm1.Button1Click(Sender: TObject);
// Eingabe, Verarbeitung, Ausgabe und Fehlerprüfung
var a,b,c: Real;
begin
  try
    a := StrToFloat(Edit1.Text);
    b := StrToFloat(Edit2.Text);
    c := StrToFloat(Edit3.Text);
    Edit4.Text := FloatToStr(Umfang(a,b,c));
    Edit5.Text := FloatToStr(Flaeche(a,b,c));
  except
    MessageBox(0,'Falsche Eingabe','Fehler',16);
    Edit4.Text := '';
    Edit5.Text := '';
  end;
end;


end.
```