8 Datentypen, Operatoren und Standardroutinen.

Überblick. Die Typen Integer, Real, Char und Boolean. Priorität. Die selbstdefinierten Typen.

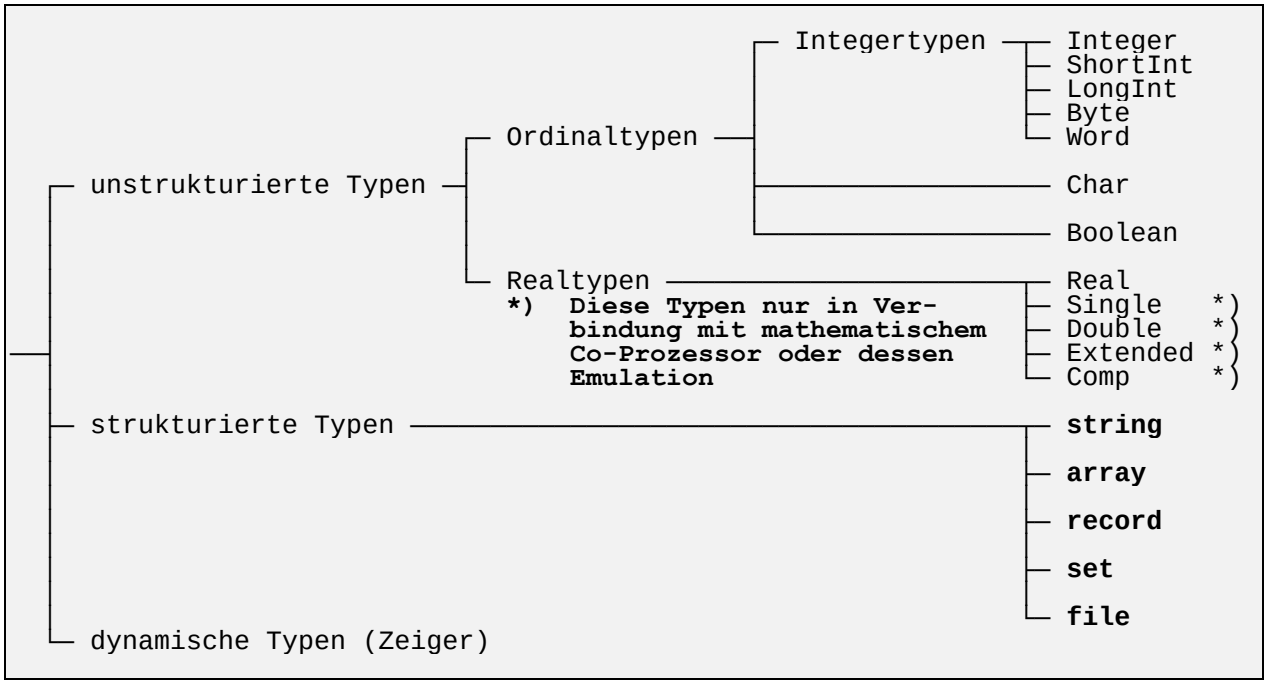
Gliederung

Vorbemerkungen:

Routine ist der Oberbegriff für *Prozeduren* und *Funktionen*. Routinen führen gewisse Aktivitäten aus. Eine *Funktion* liefert im Gegensatz zur *Prozedur* an die aufrufende Stelle einen Wert zurück, z.B. den Logarithmus einer Zahl. Der Wert kann beliebig verwendet werden, z.B. auch in einem Ausdruck. Man unterscheidet zwischen Standardroutinen (eingebaute Routinen), die Pascal in vordefinierter Form zur Verfügung stellt und selbsterstellten Routinen. Letztere werden in einem eigenen Kapitel behandelt. Prozeduren werden in Pascal wie Funktionen nur mit ihrem Namen aufgerufen.

Dieses Kapitel enthält viele Details, die nicht alle sofort für die Behandlung der Folgekapitel gebraucht werden. Bei Bedarf wird in diesem Kapitel "nachgearbeitet".

8.1 Die Datentypen in Turbo-Pascal. Ein Überblick



Datentypen können in Pascal direkt in der Variablen-Deklaration oder indirekt mit **type** deklariert werden.

Beispiel für direkte Deklaration (vermeiden):

```
....
var
  Temperatur:   Real;
  KundenNummer: Word;
....
```

Beispiel für indirekte Deklaration (bevorzugen):

```
....
type
  Grad             = Real;
  PositiverInteger = Word;
  ....
var
  Temperatur:   Grad;
  KundenNummer: PositiverInteger;
....
```

Die indirekte Deklaration mit **type** erhöht u.U. die Lesbarkeit des Programms und ist vor allem dann angebracht, wenn der gleiche Typ für verschiedene Variablen gebraucht wird und vor allem dann, wenn bestimmte Datentypen an Routinen übergeben werden müssen, wenn man nicht von den *Offenen Arrays* Bebrauch macht, die ab Turbo Pascal 7.0 (siehe Kap. 11) zur Array- und String-Übergabe verwendet werden können.

Turbo-Pascal kennt mehrere Integer- und Real-Datentypen, die Datentypen Char, Boolean, **string**, **array**, **record**, **set** und **file**, sowie den dynamischen Datentyp (Zeigertyp). Hinzu kommen noch die selbstdefinierten Datentypen (Aufzählungstypen und Teilbereichstypen).

Alle Integer- und Real-Datentypen, die Datentypen *Char* und *Boolean* zählen zu den unstrukturierten (einfachen) Datentypen. Die Datentypen *string*, *array*, *record*, *set* und *file* sind strukturierte Datentypen. Unstrukturiert heißt, daß die Werte nicht weiter unterteilt werden können. Beim strukturierten Datentyp liegt dagegen eine Ansammlung von Werten vor.

Alle Integer-Typen, die Datentypen *Char*, *Boolean* und die selbstdefinierten Datentypen (Aufzählungstypen und Teilbereichstypen) sind **ordinale** Datentypen. Sie besitzen endliche und nach einem Ordnungsschema geordnete Werte. Die Real-Typen sind keine Ordinal-Typen! Auf ordinale Typen sind die Standardfunktionen *Ord* (Ordnungsnummer), *Pred* (Predecessor, Vorgänger) und *Succ* (Successor, Nachfolger) anwendbar. Ordinale Typen können weiter in einer **case**-Selektion und als Laufvariable in einer **for**-Anweisung verwendet werden. Außerdem können nur ordinale Typen beim strukturierten Datentyp **array** als Index verwendet werden.

Turbo-Pascal kennt fünf Integer-Typen (Ganzzahl-Typen) und zwar:

- Integer
- ShortInt
- LongInt
- Byte
- Word

Turbo-Pascal kennt fünf Real-Typen (Komma-Typen) und zwar:

- Real
- Single nur mit Coprozessor oder Emulation

- Double nur mit Coprozessor oder Emulation
- Extended nur mit Coprozessor oder Emulation
- Comp nur mit Coprozessor oder Emulation

Standard-Pascal kennt dagegen nur je einen Integer- und Real-Typ.

Wenn nichts anders ausgeführt ist, dann ist mit "Integer-Typ" die Gesamtheit aller Turbo-Pascal-Integer-Typen gemeint. Entsprechendes gilt dann auch für "Real-Typ".

- Zum Datentypen Integer: Behandlung in den Unterpunkten 8.2
- Zum Datentyp Real: Behandlung im Unterpunkt 8.3
- Zum Datentyp Char: Behandlung im Unterpunkt 8.4
- Zum Datentyp Boolean: Behandlung im Unterpunkt 8.5

- **Die strukturierten Datentypen:**

  - **string** Zeichenkette. In Standard-Pascal gibt es den Datentyp **string** nicht. Dort ist eine Zeichenkette als **array of** Char zu betrachten. In Turbo-Pascal ist die Länge der Zeichenkette auf maximal 255 Zeichen begrenzt. Wenn keine Stringlänge deklariert ist, wird der Maximalwert angenommen. Um Speicherplatz zu sparen, sollte man die benötigte String-Länge deklarieren. Ab Turbo-Pascal 7.0 können mit der Unit Strings *nullterminierte Strings* mit einer Länge von bis zu 2^{16} = 65536 Zeichen verwendet werden. In Delphi ist die Stringlänge nur noch durch die Speicherkapazität gegrenzt.

    Beispiel: `s: string[25]`

    deklariert einen String *s* mit maximal 25 Zeichen Länge. Pro Zeichen wird ein Byte benötigt. Hinzu (genauer davor) kommt ein Byte, in dem die aktuelle Länge des Strings gespeichert ist, das Längenbyte. Strings werden im Kapitel 14 ausführlicher behandelt.

  - **array** Reihung gleicher Datentypen. Behandlung in einem Kapitel 12.
  - **record** Verbunde, Reihung verschiedener Datentypen. Behandlung im Kapitel 16.
  - **set** Mengen. Behandlung im Kapitel 15.
  - **file** Dateien. Behandlung im Kapitel 18.

- **Zu den dynamischen Datentypen:**

Zeigertyp (Pointer). Ein Zeiger hat keinen Wert, sondern enthält die Speicheradresse eines Wertes. Zeiger werden im Behandlung im Kapitel 19.

- **Zu den selbstdefinierten Typen**

Selbstdefinierte Typen (Aufzählungstypen und Teilbereichstypen) werden im Unterpunkt 8.7 behandelt.

8.2 Integer-Typen. Operatoren und Standardroutinen

Integer-Typ	Wertebereich			Speicherbedarf
Integer	-32768..+32767	=	$-2^{15}..+2^{15}-1$	2 Byte
ShortInt	-128..+127	=	$-2^{7}..+2^{7}-1$	1 Byte
LongInt	-2147483648..+2147483647	=	$-2^{31}..+2^{31}-1$	4 Byte
Byte	0..255	=	$0..2^{8}-1$	1 Byte
Word	0..65535	=	$0..2^{16}-1$	2 Byte

Wenn nichts anderes ausgeführt ist, dann ist mit "Integer-Typ" die Gesamtheit aller fünf Integer-Typen von Turbo-Pascal gemeint.

Turbo-Pascal kennt zwei Integer-Konstanten:

- `MaxInt` mit Wert 32767
- `MaxLongInt` mit Wert 2 147 483 647

Arithmetische Operatoren für Integer-Typen:

+ Addition
- Subtraktion und monadisches Minus
***** Multiplikation
div ganzzahlige Division. Beispiel: 10 div 4 = 2
mod Restwert der ganzzahligen Division (Modulo-Rechnung). Beispiel: 17 mod 5 = 2

Man beachte, daß der Schrägstrich (slash) "/" als Divisionsoperator für Realtypen dient. Das Ergebnis ist dann immer ein Realtyp, auch wenn die Operanden Integer-Typen sind und das Ergebnis ganzzahlig ist.

Die Vergleichsoperatoren für Integertypen:

= gleich
> größer
< kleiner
>= größer oder gleich
<= kleiner oder gleich
<> ungleich

Logische Operatoren für Integer-Typen:

Folgende logische Operatoren können auf Integer-Typen angewendet werden. Die Verarbeitung erfolgt bitweise für das ganze Bit-Muster der Integer-Operanden.

Hinweis: Logische Operatoren liefern Integer-Zahlenwerte; die booleschen Operatoren (*not*, *and*, *or* und *xor*) dagegen die Wahrheitswerte *True* oder *False* (Kap. 8.5).

not bitweise Negation
and bitweises UND
or bitweises ODER
xor bitweises Exklusiv-ODER (ausschließendes ODER, Antivalenz)
shl Bit-Muster um *n* bits nach links verschieben (shift left)
shr Bit-Muster um *n* bits nach rechts verschieben (shift right)

Die Wirkung der logischen Operatoren **not**, **and**, **or** und **xor** kann anschaulich mit Wahrheitstabellen dargestellt werden. Für "Bit gesetzt" steht "1" und für Bit nicht gesetzt steht "0".

Wahrheitstabelle:

Bit X	Bit Y	**not** X	X **and** Y	X **or** Y	X **xor** Y
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Die Tabelle gilt ansonsten auch für die Boolschen Operatoren **not**, **and**, **or** und **xor**, wenn man "1" durch "True" und "0" durch "False" ersetzt.

Beispiel: 85 **and** 7

```
Bit-Muster für 85:         0101 0101
Bit-Muster für  7:         0000 0111
-----------------------------------
Bit-Muster für (85 and 7): 0000 0101
Das Ergebnis-Bit-Muster hat das Dezimal-Äquivalent 5.
Somit:  (85 and 7) ====> 5
```

Logische Operationen mit Integer-Typen werden überwiegend bei systemnaher Programmierung benötigt.

Mengen-Operator für Integer-Typen:

Der Mengen-Operator "**in**" (Vorgriff auf strukturierten Datentyp **set,** Kap. 15) kann auch auf Integer-Typen angewendet werden.

Beispiel:

```
program Pas08021; { Demo: Mengen-Operator "in" mit Integer Typen }
uses
  CRT;
var
  z: Byte;
begin
  repeat
    Write('Drücken Sie eine der Zifferntasten 3 bis 7: ');
    ReadLn(z);
  until z in [3..7];
end.
```

Standardfunktionen für Integer-Typen:

In der folgenden Tabelle sind die wichtigsten Standardfunktionen aufgeführt, die entweder ein Integer-Argument oder ein Integer-Ergebnis haben. Manche der Funktionen sind in gleicher Weise für Integer- und auch für Real-Argumente definiert oder auch für alle Ordinaltypen. Der Ergebnistyp ist dann mit dem Typ des Arguments identisch. Das Argument ist in der Tabelle mit dem Zeichen a symbolisiert. Die mathematischen Funktionen wie Sin, Cos, ArcTan, Exp und Ln liefern ein Real-Ergebnis und sind primär auch für Real-Argumente gedacht. Da Turbo-Pascal gegebenenfalls die Integer-Argumente automatisch in Real umwandelt, können diese Funktionen auch für Integer-Argumente benutzt werden.

Standard-Funktion	Datentyp Argument	Datentyp Ergebnis	Bemerkungen
`Abs(a)`	Integer, Real	Integer, Real	Absolutwert
`Pred(a)`	Integer, ordinal	Integer, ordinal	Vorgänger, predecessor
`Succ(a)`	Integer, ordinal	Integer, ordinal	Nachfolger, successor
`Random(a)`	Word *)	Word	Zufallszahl 0..a-1
`Sqr(a)`	Integer, Real	Integer, Real	Quadrat a*a
`Odd(a)`	Integer	Boolean	Prüfung auf ungerade
`Lo(a)`	Word/Integer	Byte	Niederwertiges Byte
`Hi(a)`	Word/Integer	Byte	Höherwertiges Byte
`Sqrt(a)`	Integer, Real	Real	(Quadrat-)Wurzel
`Sin(a)`	Integer, Real	Real	Sinus,Winkel Bogenmaß
`Cos(a)`	Integer, Real	Real	Cosinus
`ArcTan(a)`	Integer, Real	Real	Arcustangens
`Exp(a)`	Integer, Real	Real	Exponentialfunktion
`Ln(a)`	Integer, Real	Real	natürl. Logarithmus
`Round(a)`	Real	Integer	ganzzahlige Rundung
`Trunc(a)`	Real	Integer	ganzzahliger Teil
`Chr(a)`	Byte	Char	Character, Zeichen
`Ord(a)`	Integer, ordinal	Integer, ordinal	Ordnungsnummer

*) Das Argument bei der Funktion "Random" ist optional und kann somit auch entfahlen. Die Funktion liefert dann eine Real-Zufallszahl aus dem Bereich 0..<1; die Grenze 1 ist also nicht eingeschlossen. Um unterschiedlichen Reihen von Zufallszahlen zu erhalten, muß der Zufallszahlen-Generator mit der Standardprozedur "Randomize" initalisiert werden. Diese Initialisierung wird im Demoprogramm Pas08022.PAS gezeigt.

Im folgenden Demo-Programmen und auch in weiteren wird die Bildschirmausgabe durch Kommentare im Quelltext simuliert. Der Senkrechtstrich soll den linken Bildschirmrand darstellen.

```
program Pas08022;   { Demo: Operatoren und Funktionen
                            für Integer-Typen }
uses
  CRT;
const
  u = 4711;
  v = 4712;
  w = 4713;
  x = 0;
  y = 1;
  z = 2;      { Alle Konstanten haben Typ (normalen) Integer }

var
  r:               Real;
  NormalerInteger: Integer;
  KurzerInteger:   ShortInt;
  ByteInteger:     Byte;
  WordInteger:     Word;

begin
  ClrScr;
  NormalerInteger := 0;
  KurzerInteger   := 0;
  ByteInteger     := 0;
  WordInteger     := 0;

  Randomize;   { Prozedur zum Initialisieren des Zufallszahlen-
                 Generators }

  r := x + (2 - y)*z/2;   { Typ Real wegen Real-Division mit /   }
  WriteLn('REAL: ', r);           { |REAL:  1.0000000000E+00    }
  WriteLn('+-*:  ', u + (1 - z)*y);{ |+-*:  4710  }    { Integer }

  { Es folgen die speziellen Integer-Operatoren:                 }
  WriteLn('DIV:  ', w div v);      { |DIV:  1     }
  WriteLn('MOD:  ', w mod u);      { |MOD:  2     }
  WriteLn('NOT:  ',   not x);      { |NOT:  -1    }
  WriteLn('NOT:  ',   not y);      { |NOT:  -2    }
  WriteLn('NOT:  ',   not -2);     { |NOT:  1     }
  WriteLn('AND:  ', y and z);      { |AND:  0     }
  WriteLn('OR:   ', x or  y);      { |OR:   1     }
  WriteLn('XOR:  ', x xor y);      { |XOR:  1     }
  WriteLn('SHL:  ', z shl 2);      { |SHL:  8     }  { 2 bit links  }
  WriteLn('SHR:  ', z shr 1);      { |SHR:  1     }  { 1 bit rechts }

  { Es folgt Demo der verschiedenen Integer-Typen mit Operator "not" }
  WriteLn('nINT: ', not NormalerInteger); { |-1   }  { negierte 0   }
  WriteLn('kINT: ', not KurzerInteger);   { |-1   }  { negierte 0   }
```

```
  WriteLn('bINT: ', not ByteInteger);     { |255  }  { negierte 0   }
  WriteLn('wINT: ', not WordInteger);     { |65535}  { negierte 0   }

  ReadLn;

{ Es folgen Funktionen mit Integer-Argument und Integer-Ergebnis: }
  WriteLn('ABS:  ', Abs(-u));       { |ABS:  4711  }  { Auch f. Real }
  WriteLn('PRED: ', Pred(u));       { |PRED: 4710  }  { Vorgänger    }
  WriteLn('SUCC: ', Succ(u));       { |SUCC: 4712  }  { Nachfolger   }
  WriteLn('RND:  ', Random(100*z));{ |RND:  73    }  { 0..199       }
  WriteLn('SQR:  ', Sqr(z));        { |SQR:  4     }  { Quadrat      }
{ Es folgt Funktion mit Integer-Argument und Boolean-Ergebnis:    }
  WriteLn('ODD:  ', Odd(u));        { |ODD:  TRUE  }  { ungerade     }
  WriteLn('ODD:  ', Odd(u - 1));    { |ODD:  FALSE }

  { Es folgen Funktionen m. Integer/Real-Argument und Real-Ergebnis }
  WriteLn('SQRT: ', Sqrt(z):6:3);  { |SQRT:  1.414 } { Quadr.-Wurzel}
  WriteLn('SIN:  ', Sin(z):6:3);   { |SIN:   0.909 } { Sinus,        }
  WriteLn('COS:  ', Cos(z):6:3);   { |COS:  -0.416 } { Cosinus,      }
  WriteLn('ATN:  ', ArcTan(z):6:3);{ |ATN:   1.107 } { Arcustangens }
  WriteLn('EXP:  ', Exp(y):6:3);   { |EXP:   2.718 } { Exponential  }
  WriteLn('LN:   ', Ln(z):6:3);    { |LN:    0.693 } { nat. Logar.  }

  { Es folgen Funktionen mit Real-Argument und Integer-Ergebnis:    }
  WriteLn('RUND: ', Round(3.4));   { |RUND: 3   }  { Rundung  }
  WriteLn('RUND: ', Round(3.5));   { |RUND: 4   }  { Rundung  }
  WriteLn('RUND: ', Round(3.6));   { |RUND: 4   }  { Rundung  }
  WriteLn('RUND: ', Round(-3.4));  { |RUND: -3  }  { Rundung  }
  WriteLn('RUND: ', Round(-3.5));  { |RUND: -4  }  { Rundung  }
  WriteLn('RUND: ', Round(-3.6));  { |RUND: -4  }  { Rundung  }
  WriteLn('TRUNC: ', Trunc(-3.6)); { |TRUNC: -3 }  { ganzzahl. Teil }

  { Es folgt Funktion mit Char-Argument und Integer-Ergebnis        }
  WriteLn('ORD:  ', Ord('A'));     { |65          }  { nach ASCII   }

  { Es folgt Funktion mit Integer-Argument und Char-Ergebnis        }
  WriteLn('CHR:  ', Chr(65));      { |A           }  { nach ASCII   }
  WriteLn('CHR:  ', #65;           { |A           }  { nach ASCII   }

  { Es folgen die Integer-Konstanten                                }
  WriteLn('MaxInt:     ', MaxInt);     { |MaxInt:     32767         }
  WriteLn('MaxLongInt: ', MaxLongInt); { |MaxLongInt: 2147483647    }
  ReadLn;
end.
```

Zur Negierung von Integertypen (im Beispiel i) und Bytetypen (im Beispiel j):

i	-128	-127	-126		-3	-2	-1	0	+1	+2		+125	+126	+127
not i	+127	+126	+125		+2	+1	0	-1	-2	-3		-126	-127	-128
j	0	+1	+2									+253	+254	+255
not j	+255	+254	+253									+2	+1	0

Standardprozeduren für Integer-Typen:

Turbo-Pascal besitzt viele Prozeduren mit Integer-Argumenten; sie werden bei den einschlägigen Kapiteln behandelt.

Wegen der besonderen Bedeutung werden an dieser Stelle nur die Standard-Prozeduren *Inc* (inkrementiere, erhöhe) und *Dec* (dekrementiere, erniedrige) vorgestellt. Beide Prozeduren sind für alle ordinalen Typen zulässig, also nicht nur für Integer.

Formate:
`Inc(`*x*`)`	Erhöhe *x* um 1
`Inc(`*x*`, `*n*`)`	Erhöhe *x* um *n*
`Dec(`*x*`)`	Erniedrige *x* um 1
`Dec(`*x*`, `*n*`)`	Erniedrige *x* um *n*

x ist eine Variable mit ordinalem Typ.
Der optionale Parameter *n* ist ein Integer-Ausdruck

Inc(x) ist ein etwas schnellerer Ersatz für die Anweisung `x := x + 1`, aber erst bei vielen Schleifendurchläufen.

8.3 Die Realtypen. Operatoren und Standardfunktionen

Realtypen werden durch eine Mantisse und einen Exponenten dargestellt. Turbo-Pascal kennt fünf Real-Typen, Standard-Pascal nur einen.

Real-Typ	Wertebereich, und signifikante Stellenzahl	Speicherbedarf
Real	±2.9E-39..±1.7E+38, 11 bis 12 Stellen	6 Byte
Single	±1.5E-45..±3.4E+38, 7 bis 8 Stellen	4 Byte
Double	±5.0E-324..±1.7E+308, 15 bis 16 Stellen	8 Byte
Extended	±1.9E-4951..±1.1E+4932, 19 bis 20 Stellen	10 Byte
Comp	-9.2E+18..+9.2E+18, 18 bis 19 Stellen	8 Byte

Allerdings sind die Real-Datentypen *Single*, *Double*, *Extended* und *Comp* nur mit dem mathematischen Coprozessor verfügbar. Diese haben die Bezeichnungen i8087, i80287 oder i80387, ab dem i80486 DX ist der Coprpzessor im Hauptprozessor integeriert, somit auch beim Pentium und seinen Nachfolgern. Der Coprozessor kann aber auch software-mäßig emuliert werden, siehe Compilerschalter **E** und **N** im Kap. 5.12.1 bzw. Menüpunkt "Option/Compiler.../Gleitkommaberechnungen". Der Typ *Comp* ist streng genommen kein Realtyp, muß aber als solcher behandelt werden.

Real-Typen werden standardmäßig in Gleitkomma-Schreibweise mit einer Schreibbreite von 17 Zeichen (Datentyp Real) oder 23 Zeichen (Datentyp Double) ausgegeben, auch wenn der Wert im Einzelfall ganzzahlig ist. Mit Hilfe einer Formatierung können Real-Typen aber auch in Fixkomma-Schreibweise ausgegeben werden, siehe Kapitel 7.1. Die Anzahl der Nachkommastellen kann gewählt werden. Im Gegensatz zu Integerdaten wird bei Realdaten für das positive Vorzeichen ein Leerzeichen gedruckt. Als Dezimaltrennzeichen dient der Punkt und nicht das Komma. Die Eingabe von Realtypen kann wahlweise in Gleitkomma- oder in Fixkomma-Schreibweise erfolgen. Die führende Null vor dem Dezimalpunkt ist im Gegensatz zu einigen anderen Programmiersprachen in Pascal bei Zuweisungen, Rechenoperationen und Ausgaben anzugeben; lediglich beim

Einlesen von der Tastatur mit *Read* bzw. *ReadLn* kann man schlampern und die führende Null weglassen.

Die Zahl `47.11` in Gleitkomma-Schreibweise (unterhalb der Abzählleiste):

`12345678901234567890123`	Nur Abzählleiste
`4.7110000000E+01`	Ohne Coprozessor, Datentyp Real
`4.71099999999860E+0001`	Mit Coprozessor, Datentyp Real
`4.71100000000000E+0001`	Mit Coprozessor, Datentyp Double

Realtypen sind keine Ordinaltypen, da sie keine endliche Wertemenge umfassen. Die Funktionen *Ord*, *Pred*, *Succ* sind somit nicht für Realtypen zugelassen, ebenso nicht die Prozeduren *Inc* und *Dec*.

Die Vergleichsoperatoren für Realtypen:

=	gleich
>	größer
<	kleiner
>=	größer oder gleich
<=	kleiner oder gleich
<>	ungleich

Bei Realtypen ist immer mit Fehlern zu rechnen, die z.B. bei der Differenzbildung von großen Zahlen beträchtlich sein können. Zudem können gewisse Dezimalzahlen, wie z.B. 0.1, nicht exakt im Dualsystem dargestellt werden, was auch zu Fehlern führen kann. Deshalb sollte man Realtypen nie auf Gleichheit prüfen, sondern auf >= oder <= .

Arithmetische Operatoren für Realtypen:

+	Addition
-	Subtraktion und monadisches Minus
*	Multiplikation
/	Division

Man beachte, daß Pascal keinen Potenz-Operator besitzt.

Bei ganzzahligem Exponenten *y* (Integer-Typ) kann der Ausdruck x^y z.B. mit Hilfe der Standardfunktion *Sqr* (Quadratbildung) wie folgt dargestellt werden:

```
x^2 --->    Sqr(x)
x^3 --->    x*Sqr(x)
x^4 --->    Sqr(Sqr(x))
x^5 --->    x*Sqr(Sqr(x))
```

usw. Für höhere Potenzen sollte man aber eigene Funktionen programmieren. Für Polynome wähle man die Darstellung nach HORNER, bei der nur Additionen und Multiplikationen vorkommen.

Bei nichtganzzahligem Exponenten *y* (Realtyp) muß der Term x^y mit den Standardfunktionen "Exp" (Exponentialfunktion) und "Ln" (natürlicher Logarithmus) wie folgt dargestellt werden:

$$x^y = e^{y\ \ln(x)} = \mathrm{Exp}(y * \mathrm{Ln}(x))$$

wobei x > 0 sein muß. Auch diese Ersatzdarstellung hat ihre Tücken im Fall x <= 0. Eine universelle Lösung, zumindest für Realtypen, ist in der selbstdefinierten Funktion "Potenz" im Programm "Pas11021.PAS" im Kap. 11.02 angegeben.

Die Standardfunktionen für Realtypen:

Fast alle Standardfunktionen für Realtypen sind auch für Integertypen zulässig, da Turbo-Pascal gegebenenfalls die Integertypen automatisch in Realtypen umwandelt. Über die Zuordnung Datentyp Argument zu Datentyp Ergebnis gibt die folgende Tabelle Auskunft. Das Argument *a* kann ein beliebiger Real-Ausdruck oder (bei den meisten Funktionen!) auch ein beliebiger Integer-Ausdruck sein. Siehe auch das frühere Demo-Programm Pas08022.PAS.

Standard-Funktion	Datentyp Argument	Datentyp Ergebnis	Bemerkungen
`Abs(a)`	Real, Integer	Real, Integer	Absolutwert
`Random`	*)	Real	Zufallszahl 0..<1
`Sqr(a)`	Real, Integer	Real, Integer	Quadrat a*a
`Sqrt(a)`	Real, Integer	Real	(Quadrat-)Wurzel
`Pi`		Real	Kreiszahl 3.14159...
`Sin(a)`	Real, Integer	Real	Sinus, Winkel Bogenmaß
`Cos(a)`	Real, Integer	Real	Cosinus
`ArcTan(a)`	Real, Integer	Real	Arcustangens
`Exp(a)`	Real, Integer	Real	Exponentialfunktion
`Ln(a)`	Real, Integer	Real	natürl. Logarithmus
`Round(a)`	Real	Integer	ganzzahlige Rundung
`Frac(a)`	Real	Real	Nachkommateil
`Trunc(a)`	Real	Integer	ganzzahliger Teil

*) Die Funktion »Random« kann optional auch einen Integer-Ausdruck als Argument haben. Sie liefert dann Integer-Zufallszahlen. Siehe Unterpunkt 8.2.

Um unterschiedliche Reihen von Zufallszahlen zu erhalten, muß der Zufallfszahlen-Generator mit der Standardprozedur `"Randomize"` initialisiert werden. Diese Initialisierung wird im früheren Demo-Programm Pas08022.PAS gezeigt.

Die Funktion `ArcTan` liefert den Hauptwert (Bereich: -Pi/2 ... +Pi/2)

Für die Umrechnung von Gradmaß in Bogenmaß gilt bekanntlich:

`Winkel_in_Grad = Winkel_in_Bogen/Pi*180,` mit `Pi` = π

Die mathematische Funktion **Tangens** ist standardmäßig nicht in Pascal enthalten. Der Tangens des Winkels Alpha muß wie folgt berechnet werden:

```
Tan(Alpha) = Sin(Alpha) / Cos(Alpha)          Alpha <> ±Pi/2
```

Der **Arcussinus** ist standardmäßig ebenfalls nicht in Pascal enthalten. Bei der Berechnung sind drei Fälle zu unterscheiden:

```
                      ┌─ +Pi/2,                              wenn x =  +1
Arcussinus von x:   ──┼─  -Pi/2,                              wenn x =  -1
                      └─  ArcTan(x / Sqrt(1 - Sqr(x))),      wenn x <> ±1
```

Der **Arcuscosinus** ist standardmäßig ebenfalls nicht in Pascal enthalten. Bei der Berechnung sind zwei Fälle zu unterscheiden:

```
                      ┌─ +Pi,                                wenn x =  -1
Arcuscosinus von x: ──┤
                      └─ 2*ArcTan(Sqrt((1 - x)/(1 + x))),    wenn x <> -1
```

Der **dekadische Logarithmus** ist ebenfalls nicht in Pascal enthalten. Für die Umrechnung gilt:

```
Ln(x) / Ln(10)
```

Für die **Kreiszahl Pi** = 3.14159.... steht in Turbo-Pascal die gleichnamige Standardfunktion zur Verfügung, was in Standard-Pascal und einigen anderen Programmiersprachen nicht der Fall ist. Mit einer Wertzuweisung an eine "Variable" Pi mit:

```
Pi := 4 * ArcTan(1)
```

kann man sich aber leicht behelfen. In Pascal besteht natürlich auch die Möglichkeit, eine Konstante Pi mit dem Zahlenwert 3.14159... zu deklarieren. Wegen Fehlermöglichkeit vermeiden!

Zur Schreibweise von mathematischen Ausdrücken:

Bei der mathematischen Schreibweise wird das Multiplikationszeichen häufig nicht angeschrieben. In Pascal muß das aber unbedingt geschehen. In den Beispielen seien x, y und z Variablen.

Mathematisch	In Pascal (und ähnl. Sprachen)
$x + yz$	`x + y*z`
$x - \frac{y}{z}$	`x - y/z`
$\frac{xy}{z}$	`x*y/z` oder `x/z*y`
$\frac{x+y}{z}$	`(x + y)/z`

$\frac{x}{yz}$	`x/y/z` oder `x/(y*z)`

Das folgende Demo-Programm zeigt die Wirkung des mathematischen Coprozessors bei Gleitkommaberechnungen in Abhängigkeit der Compilerschalter **N** und **E**. Für die Demo muß aber ein Vorgriff auf die **for**-Schleife gemacht werden:

```
{$N+,E+}          { 4 Kombinationen "N-,E-", "N-,E+", }
                  {                 "N+,E-", "N+,E+"  }
program Pas08031; { "Pas08031.PAS", Demo Coprozessor  }
                  { 37300398, Dr. K. Haller           }
uses
  CRT, DOS;       { Unit DOS wegen "GetTime"          }

const
  iMax = 100000;

var
  x, y, T:   Double;  { Hier steht fallweise "Real" oder "Double" }
  i:         LongInt;
  hh, mm,
  ss, ss100: Word;

begin
  ClrScr;
  GetTime(hh, mm, ss, ss100);
  T := hh*3600.0 + mm*60 + ss + ss100/100;   { "3600.0" !!! }

  for i := 1 to iMax do
    begin
      x := Sin(47.11);
      y := Exp(47.11);
    end;

  GetTime(hh, mm, ss, ss100);
  T := hh*3600.0 + mm*60 + ss + ss100/100 - T;
  WriteLn('Die Ausführungszeit: ', T:8:4);
  ReadLn;
  (*
  | Compiler- | Zeit     | Zeit        | EXE-File   | EXE-File    |
  | schalter  | Real-Typ | Double-Typ  | Real-Typ   | Double-Typ  |
  +===========+==========+=============+============+=============+
  | {$N-,E-}  | 3.90 s   | nicht mögl. | 12090 Byte | nicht mögl. |
  | {$N-,E+}  | 3.90 s   | nicht mögl. | 12090 Byte | nicht mögl. |
  | {$N+,E-}  | 0.99 s   | 0.99 s      | 12042 Byte | 11946 Byte  |
  | {$N+,E+}  | 0.99 s   | 0.93 s      | 21802 Byte | 21706 Byte  |

  Die Ausführungszeiten und EXE-Dateigrößen gelten für ein System
  mit Prozessor Pentium 166 MHz und den hier nicht aufgeführten
  weiteren Compilerschaltern. Die Daten sind nur für diese Test-
  umgebung gültig und können nicht einfach verallgemeinert werden.

  Die Compilerschalter-Kombination {$N+,E+} ist auf allen Systemen
```

```
  lauffähig, egal ob Coprozessor vorhanden ist oder nicht.

  Die Compilerschalter-Kombination {$N+,E-} ist nur auf Systemen
  mit Coprozessoren lauffähig.

  *)
end.
```

8.4 Der Typ Char. Operatoren und Standardfunktionen

Der Datentyp *Char* (Character, Zeichen) umfaßt alle vom Computer darstellbaren Zeichen, also nicht nur Buchstaben, sondern auch Ziffern, Satz- und Sonderzeichen, Graphikzeichen und Steuerzeichen. Letztere können aber nicht sichtbar dargestellt werden, sondern sind für die Steuerung von Computerfunktionen vorgesehen. Jedes Zeichen wird im Speicher mit 1 Byte = 8 bit dargestellt. Damit ergeben sich 2^8 = 256 verschiedene Bit-Muster, die den Zeichen mit den Ordnungsnummern von 0 bis 255 zugeordnet werden. Die Ordnungsnummern sind das Dezimal-Äquivalent der Bit-Muster. Die Zuordnung (Codierung) ist im Prinzip willkürlich, sinnvollerweise wird man aber die Codierung so wählen, daß die Buchstaben dem gewöhnlichen Alphabet entsprechend aufeinanderfolgen. Bei Ziffern wird man die gleiche Vorgehensweise wählen. Bei Mikrocomputern ist fast ausschließlich die Codierung nach ASCII gebräuchlich (ASCII: American Standard Code for Information Interchange); allerdings nicht mehr in der ursprünglichen 7-Bit-Form, sondern in der erweiterten 8-Bit-Form (Zeichen 128 bis 255), für die es noch keine Norm, dafür aber einen mittlerweile weit verbreiteten "Industrie-Standard" gibt, den sogenannten "IBM-Zeichensatz für Mikrocomputer". Nach ASCII beginnen die Großbuchstaben mit der Codenummer 65 (Zeichen A), die Kleinbuchstaben haben um 32 verschobene Codenummern, beginnen also bei 97 (Zeichen a). Die Ziffernzeichen beginnen mit der "0" bei 48, in hex 30 (leicht merkar), das Leerzeichen hat die Codenummer 32. Die Zeichen für die Codenummern >= 128 sind nationale Sonderzeichen, Graphikzeichen und mathematische Zeichen, z.B. nach dem "IBM-Zeichensatz". Weitere Details siehe Kapitel 13. Den IBM-Zeichensatz gibt es zudem in verschiedenen nationalen Ausgestaltungen, Im Betriebssystem MS-DOS Codepages genannt. Für Deutschland empfiehlt sich die Codepage 437 (auf dem Arbeitsblatt dargestellt) oder die Codepage 850.
Hinweis: Das Windows- und das Apple-Betriebsssytem arbeiten mit dem Ansi-Zeichensatz, der im Codebereich 0 bis 127 identisch ist mit dem Ascii-Zeichensatz, sich aber im Codebereich >= 128 gänzlich vom IBM-Zeichensatz unterscheidet.

Operatoren für Char-Typen:

Char-Typen können mit *Read* bzw. *ReadLn* eingelesen und mit *Write* bzw. *WriteLn* ausgegeben werden. Die Vergleichsoperatoren (= > < >= <= <>) und der Mengenoperator **in** sind auch für Char-Typen definiert, ebenso die Wertezuweisung. Es gibt aber keine Operatoren für die Manipulation von Char-Typen.

Schreibweise der Konstanten vom Typ Char:

Konstanten vom Typ *Char* sind in Hochkommas zu setzen. Wenn das Hochkomma selbst als Zeichen gebraucht wird, dann muß es doppelt in Hochkommas geschrieben werden. In Turbo-Pascal können Char-Konstanten auch durch die Codenummer des Zeichens und dem vorausgestellten Nummernzeichen # dargestellt werden.

Beispiele:	`'A'`	`'a'`	`'7'`	`'+'`	`' '`	`''''`
In Turbo-Pascal auch:	`#65`	`#97`	`#55`	`#43`	`#32`	`#39`

Das vorletzte Zeichen ist ein Blank (Space, Leerzeichen), das letzte ein Hochkomma.

Standardfunktionen für Char-Typen:

- Die Standardfunktion Chr(*a*) liefert das Zeichen mit der Ordungsnummer *a*, wobei *a* einen Byte-Ausdruck darstellt. In Turbo-Pascal statt Chr(*a*) auch zulässig, wenn *a* eine Byte-Konstante ist: *#a*

  Beispiele:

```
Write(Chr(65));      { |A    }
Write(#66);          { |B    }
Write(Chr(97));      { |a    }
Write(Chr(98));      { |b    }
Write(Chr(48));      { |0    }
Write(Chr(49));      { |1    }
Write(Chr(32));      { |     }  { Blank, Space, Leerzeichen }
Write(Chr(39));      { |'    }  { Hochkomma          }
```

- Die Standardfunktion Ord(*Ch*) ist die Umkehrfunktion zu Chr(a). *Ch* ist eine Konstante oder Variable vom Datentyp *Char*. Die Funktion *Ord* liefert bei Char-Typen die Ordnungsnummer des Zeichens, also die Code-Nummer nach ASCII, bzw. nach dem IBM-Zeichensatz. Das Ergebnis hat den Datentyp *Byte* und liegt somit im Bereich von 0 bis 255.

  Beispiele:

```
Write(Ord('A'));     { |65   }
Write(Ord('B'));     { |66   }
Write(Ord('a'));     { |97   }
Write(Ord('b'));     { |98   }
Write(Ord('0'));     { |48   }
Write(Ord('1'));     { |49   }
Write(Ord(' '));     { |32   }  { Blank, Space, Leerzeichen }
Write(Ord(''''));    { |39   }  { Hochkomma          }
```

- Die Standardfunktionen Pred(*Ch*) (Vorgänger) und Succ(*Ch*) (Nachfolger) sind auch für Char-Typen definiert. Ist Ch eine Konstante oder Variable vom Datentyp *Char*, dann hat das Ergebnis ebenfalls den Datentyp *Char*. Man beachte, daß der Nachfolger des letzten Wertes und der Vorgänger des ersten Wertes nicht definiert sind.

Beispiele:

```
Write(Pred('B'));     { |A    }
Write(Pred('b'));     { |a    }
Write(Pred('1'));     { |0    }
Write(Succ('A'));     { |B    }
Write(Succ('a'));     { |b    }
Write(Succ('0'));     { |1    }
```

- Die Standardfunktion *ReadKey* liest ein Zeichen von der Tastatur (genauer: aus dem Tastaturpuffer) ohne das Zeichen auf dem Bildschirm anzuzeigen. Diese Funktion hat kein Argument. Sie benötigt die Unit *CRT*. Weitere Details siehe Kapitel 7 "Ein- und Ausgaben".
- Die Standardfunktionen UpCase(*Ch*) (Upper Case) liefert den Großbuchstaben des Zeichens. *Ch* ist eine Konstante oder eine Variable vom Typ *Char*. Die Funktion wirkt nur für den Kleinbuchstabenbereich 'a'..'z'. Alle anderen Zeichen werden nicht verändert, somit leider auch nicht 'ä', 'ö' und 'ü'. Für die Umwandlung in Kleinbuchstaben gibt es in Turbo-Pascal keine Standardfunktion.

  Beispiele:

```
Write(UpCase('a'));     { |A    }
Write(UpCase('b'));     { |B    }
Write(UpCase('A'));     { |A    }
Write(UpCase('1'));     { |1    }
Write(UpCase('ä'));     { |ä    }
Write(UpCase('Ä'));     { |Ä    }
Write(UpCase('+'));     { |+    }
```

Demo-Programm:

```
program Pas08041;   { Demo: Typ Char, Ja-/Nein-Eingabe }
uses
  CRT;              { Unit CRT wegen ReadKey }
var
  Antwort: Char;
begin
  ClrScr;
  repeat
    Write('Wiederholung (j/n): ');
    repeat
      Antwort := Chr(Ord('a') - Ord('A') + Ord(UpCase(ReadKey)));
      { Bleibt in der Schleife, bis 'j' oder 'J' oder 'n' oder 'N'
        von der Tastatur eingegeben wird. Die Zeichen werden in
        Kleinbuchstaben umgewandelt.
        Der Teil-Ausdruck "Ord('a') - Ord('A')" ergibt beim
        ASCII-Code den Wert 32.
       }
    until Antwort in ['j', 'n'];
    { oder:   until (Antwort = 'j') or (Antwort = 'n');  }
    Writeln(Antwort);
  until Antwort = 'n';
end.
```

8.5 Der Typ Boolean. Operatoren und Standardfunktionen

Boolesche Variablen und Ausdrücke können nur die vordefinierten Werte "True" oder "False" annehmen.

Boolesche Ausdrücke werden vorrangig als Bedingung in if-Anweisungen, und als Bedingung in den Schleifenanweisungen **while** und **until** verwendet.

Die Ergebnisse von Boolean-Ausdrücken können außerdem mit dem Zuweisungsoperator := an boolesche Variablen zugewiesen werden.

Eine Eingabe von Boolean-Typen (z.B. mit *Read*) ist nicht möglich, wohl aber eine Ausgabe mit Write, wobei die Zeichenfolge "TRUE" bzw. "FALSE" ausgegeben wird.

Man unterscheidet einfache und zusammengesetzte boolesche Ausdrücke. Einfache Ausdrücke enthalten nur Relationen. Zusammengesetzte Ausdrücke können außer Relationen auch die booleschen Operatoren **not**, **and**, **or** und **xor** enthalten.

Beispiele für einfache boolesche Ausdrücke mit Verwendung von if-Anweisungen und bei Zuweisungen:

```
program ....;
var
  Note:      Integer;
  Bestanden: Boolean;
  Zeichen:   Char;
  ....;
begin
  ....;
  Note    := ....;
  Zeichen := ....;
  ....
  if Note = 1 then ....;
   { └──┬──┘
      └── Boolescher Ausdruck, nur True oder False,
          wenn True, dann wird folgende Anweisung ausgeführt }

  if Zeichen <> 'J' then ....;

  Bestanden := (Note < 5);
            { └──┬──┘
               └── Boolescher Ausdruck, nur True oder False.
                   Zuweisung an eine boolsche Variable     }
  ....;
  Write(Bestanden);    {|TRUE      oder: |FALSE   }

  if Bestanden then ....;  { Nicht schön und ineffizient:  }
                           { if Bestanden = True then .... }
  ....;
end.
```

Operatoren für Boolean-Typen:

Die Boolean-Operatoren sind mit:

- **not** logische Negation
- **and** logisches UND
- **or** logisches ODER
- **xor** logisches Exklusiv-ODER (ausschließendes ODER, Antivalenz)

eine Untermenge der logischen Operatoren für Integer-Typen. Die Wirkung der booleschen Operatoren kann anschaulich mit Wahrheitstabellen dargestellt werden:

X	Y	**not** X	X **and** Y	X **or** Y	X **xor** Y
False	False	True	False	False	False
False	True	True	False	True	True
True	False	False	False	True	True
True	True	False	True	True	False

Beispiel für einen zusammengesetzten booleschen Ausdruck mit Zuweisung an eine boolesche Variable mit dem Bezeichner "Test":

```
Test := (x = 3) and not (Zeichen = 'J') or (y < 4711);
```

Hat *x* den Wert 7, *y* den Wert 2 und *Zeichen* den Wert 'N', dann erhält die boolesche Variable *Test* den Wert *True*.

Die Priorität aller Operatoren wird im Unterpunkt 8.6 erklärt. Es wurde aber im Beispiel vorweggenommen, daß unter den logischen Operatoren **not** die höchste Priorität hat, in der Stufe 2 befindet sich das **and**, wogegen **or** und **xor** in der 3. Stufe sind. Die relationalen Operatoren haben die geringste Priorität. Durch Klammerung mit runden Klammern kann aber jede gewünschte Abarbeitungsreihenfolge erzwungen werden.

Zur Umformung von booleschen Ausdrücken:

not (**not** x)	ist gleichwertig mit:	x
not (x **and** y)	ist gleichwertig mit:	(**not** x) **or** (**not** y)
not (x **or** y)	ist gleichwertig mit:	(**not** x) **and** (**not** y)
not (x = y)	ist gleichwertig mit:	x <> y

Zur Auswertung von zusammengesetzten booleschen Ausdrücken:

Ein zusammengesetzter boolescher Ausdruck, der z.B. die Form hat: *b1* **and** *b2*

wird standardmäßig von Turbo-Pascal nicht vollständig ausgewertet, wenn die Auswertung von *b1* den Wert *False* ergibt, da ja damit bereits das Gesamtergebnis False feststeht.

Ein ähnlicher Fall liegt vor bei: *b1* **or** *b2*

Wenn hier *b1* den Wert *True* hat, dann wird *b2* nicht mehr ausgewertet, da das Gesamtergebnis mit *True* ebenfalls bereits feststeht, unabhängig von *b2*.

Dieses effiziente *Kurzschlußverfahren* kann aber in Sonderfällen zu Problemen führen, wenn z.B. *b2* Aufrufe von selbstdefinierten Funktionen enthält und in diesen Funktionen Variablen manipuliert werden (bzw. in diesem Falle nicht), auf die an anderer Stelle des Programms zugegriffen wird. Die Variablen können dann undefiniert sein.

In Turbo-Pascal kann man eine vollständige Auswertung eines zusammengesetzten booleschen Ausdruckes erzwingen. Dazu dient der Compilerbefehl {$B}. Dieser ist standardmäßig auf {$B-} gesetzt (Kurzschlußverfahren). Mit {$B+} wird eine vollständige Auswertung eines booleschen Ausdruck erzwungen. Diese Compilerbefehle werden in das Programm geschrieben werden und bei Bedarf lokal. Eine entsprechende Einstellung ist auch über den Menüpunkt "*Option/Compiler.../Boolesche Ausdrücke vollständig"* in der IDE des Turbo-Pascal-Systems möglich, siehe Kap. 5.12.1.

Standardfunktionen für Boolean-Typen:

Da der Typ *Boolean* ordinal ist, können auf ihn die Standardfunktionen

- Ord (Ordnungsnummer)
- Pred (Predecessor, Vorgänger) und
- Succ (Successor, Nachfolger)

angesetzt werden, wenn auch ohne großen praktischen Nutzen. Die Ordnungsnummer von "False" ist 0, die von "True" ist 1. Somit gilt:

- True > False
- Succ(False) ⟹ True
- Pred(True) ⟹ False

Weitere Standardfunktionen mit Boolean-Ergebnis:

- Odd(*a*) Prüfung auf ungerade. Liefert *True*, wenn die Auswertung des Integer-Ausdrucks *a* eine ungerade Zahl ergibt, sonst *False*. Beispiel: Odd(4711) liefert *True*.
- KeyPressed Liefert *True*, wenn der Tastaturpuffer noch Zeichen enthält, sonst False. Diese Funktion hat kein Argument und benötigt die Unit *CRT*.
- EoLn(*f*) End of Line. Liefert *True*, wenn der Positionszeiger innerhalb der Datei *f* auf das Zeilenende zeigt oder das Dateiende erreicht ist, sonst *False*.
- SeekEoLn(*f*) Liefert *True*, wenn sich zwischen dem Positionszeiger und dem nächsten Zeilenende der Datei *f* noch lesbare Zeichen befinden, sonst *False*.
- EoF(*f*) End of File. Liefert *True*, wenn das Ende der Datei *f* erreicht ist oder die Datei keine Daten enthält oder wenn die Datei eine

Geräte-Datei ist, von der keine Daten gelesen werden können (z.B. Drucker), sonst *False*.

- SeekEof(*f*) Liefert *True*, wenn sich zwischen dem Positionszeiger und dem Ende der Datei *f* noch lesbare Zeichen befinden, sonst *False*.

Demo-Programm:

```
program Pas08051;  { Demo: Typ Boolean, Priorität der Operatoren }
uses
  CRT;
var
  x, y,
  Note:      Integer;
  Zeichen:   Char;
  Test1,
  Test2,
  Bestanden: Boolean;
begin
  ClrScr;
  x         := 7;
  y         := 2;
  Note      := 2;
  Zeichen   := 'N';
  Bestanden := (Note < 5);
  Test1     := (x = 3) and not (Zeichen = 'J') or (y < 4711);
  Test2     := (x div y > 3) and ((Zeichen <> 'J') xor (x >= 4711));
  Writeln(Test1);                  { |TRUE       }
  Writeln(not Test1);              { |FALSE      }
  Writeln(Test2);                  { |FALSE      }
  Writeln(Bestanden xor Test2);    { |TRUE       }
  Writeln(not True);               { |FALSE      }
  Writeln(not False);              { |TRUE       }
  Writeln(x in [3..5]);            { |FALSE      }
  ReadLn;
end.
```

8.6 Die Priorität der Operatoren

Die Priorität der arithmetischen und logischen Operatoren wird in der folgenden Liste gezeigt. Eine andere Prioritätsfolge kann durch Klammerung mit runden Klammern erzwungen werden.

```
not  @                              1. Stufe (unär, höchste Stufe)
/  *  div  mod  and  shl  shr       2. Stufe (multiplizierend)
+  -  or  xor                       3. Stufe (addierend)
=  <  >  <>  <=  >=  in             4. Stufe (relational)
```

Das Symbol **@** ist der Adress-Operator. Mit ihm kann die Adresse einer Variablen bestimmt werden und einem Zeiger zugeordnet werden. Siehe Kapitel über dynamische Datentypen.

Die Symbole **+**, **-** und ***** dienen auch als Mengen-Operatoren; das Symbol **+** zudem auch noch als String-Operator. Siehe die einschlägigen Kapitel.

8.7 Die selbstdefinierten Typen

Pascal hat gegenüber den meisten anderen Programmiersprachen den Vorzug, daß eigene Datentypen definiert werden können. Die Deklaration eigener Typen kann ebenfalls wieder direkt bei der Variablendeklaration oder indirekt mit "**type**" erfolgen. Bei eigenen Datentypen sollte man die indirekte Dekleration vorziehen. Sie ist sogar notwendig, wenn Variablen mit selbstdefinierten Typen als Parameter an Funktionen oder Prozeduren übergeben werden (Vorgriff), wenn auch diese Notwendigkeit ab Turbo-Pascal 7.0 bei Verwendung der Offenen Arrays bei Array- und Stringübergabe an Routinen nicht mehr besteht (Vorgriff).

Die selbstdefinierten Datentypen unterteilen sich in
1. **Aufzählungstypen** (enumerated types) und
2. **Teilbereichstypen** (Ausschnittstyp, subrange types).

Beide Datentypen sind ordinal, d.h. sie besitzen endliche und geordnete Werte. Die Standardfunktionen *Ord* (Ordnungsnummer), *Pred* (Predecessor, Vorgänger) und *Succ* (Successor, Nachfolger) sind somit auch bei selbstdefinierten Typen anwendbar.

Darüber hinaus sind mit "**type**" auch Umbennenungen von Standardtypen möglich.

Beispiel:

```
type
  Grad = Real;
var
  Temperatur: Grad;
```

1. Der Aufzählungstyp:

Die möglichen Werte dieses Typs werden mit dem Komma als Trennzeichen in runde Klammern geschrieben. Die Werte können nur durch Bezeichner dargestellt werden. Die Schreibweise der Bezeichner bezüglich Groß- und Kleinbuchstaben wird, wie bei allen Bezeichnern, nicht beachtet und ist somit beliebig.

Beispiel:

```
....
type
  Grundfarben = (Cyan, Magenta, Yellow, Black);
  ....
```

```
var
  Farbe: Grundfarben;
  ....
begin
  ....
  Farbe := Magenta;
  ....
  if (Farbe = Magenta)
     then Write('Die Farbe ist Magenta.');
  ....
end.
```

Wichtig: Aufzählungstypen können weder eingegeben noch ausgegeben werden.

Im vorstehenden Beispiel würden die Anweisungen "Read(Farbe)" oder "Write(Farbe)" die Fehlmeldung 64 "Cannot Read or Write variables of this type" zur Folge haben. Im Dialogfenster "Auswerten und Ändern" des integrierten Debuggers können an den Haltepunkten mit **Strg+F4** auch Aufzählungstypen abgefragt und gegebenenfall auch geändert werden.

Zuweisungen und Vergleichsoperatoren (=, >, <, >=, <=, <>) sind für Aufzählungstypen zulässig. Die Ordnungsnummer der Werte ergibt sich aus der Reihenfolge der Auflistung. Der erste Wert hat die Ordungsnummer 0, der zweite die Ordnungsnummer 1 usw. Somit ist im Beispiel "Magenta" größer als "Cyan" und kleiner als "Yellow". Die Ordungsnummer kann mit der Standardfunktion "Ord" ermittelt werden. Im vorstehenden Beispiel würde "Ord(Farbe)" bei "Magenta" den Wert 1 liefern.

Die Standardfunktionen »Pred« (Predecessor, Vorgänger) und »Succ« (Succesor, Nachfolger) sind ebenfalls für Aufzählungstypen zulässig. Zu beachten ist aber, daß der Nachfolger des letzten Wertes und der Vorgänger des ersten Wertes nicht definiert sind

2. Der Teilbereichstyp:

Teilbereichstypen sind ein Rückgriff auf einen ordinalen Grundtyp. Der Grundtyp kann ein vordefinierter Typ sein (alle Integer-Typen, Char und Boolean) oder ein bereits deklarierter Aufzählungstyp. Der Wertebereich des Teilbereichstypen ist gegenüber dem Grundtyp eingeschränkt und kann somit an die Aufgabenstellung angepaßt werden.

Real-Typen sind somit nicht für Teilbereichstypen zulässig.

Die Definition erfolgt durch Nennung je einer Konstanten für die Untergrenze und für die Obergrenze des Teilbereiches. Beide Werte sind mit zwei aufeinanderfolgenden Punkten zu trennen.

Beispiel: "Wochentage" ist ein Aufzählungstyp, die anderen sind Teilbereichstypen. "Arbeitstage" ist ein Teilbereichstyp des vorher deklarierten Aufzählungstyp "Wochentage".

```
....
type
  KontoNummernBereich = 100000..999999;
```

```
  KontoBereich        = -6000..9000;
  Grossbuchstaben     = 'A'..'Z';
  Wochentage          = (Mon, Die, Mit, Don, Fre, Sam, Son);
  Arbeitstage         = Mon..Fre; { Grundtyp ist der vorher
                        deklarierte Aufzählungstyp »Wochentage«}
var
  KontoNummer: KontoNummernBereich;
  Konto:       KontoBereich;
  Zeichen:     Grossbuchstaben;
  Tag:         Arbeitstage;
```

Die Eigenschaften des Teilbereichstyps sind bezüglich Operatoren, Funktionen und Eingabe und Ausgabe identisch mit denen des Grundtyps. Zum Beispiel können Integer und Character ein- und ausgegeben werden. Boolean können nur ausgegeben werden; bei Aufzählungstypen ist weder Eingabe noch Ausgabe möglich.

Bei Turbo-Pascal wird standardmäßig nicht auf Einhaltung der Grenzen geprüft. Trotz Deklaration von Teilbereichstypen können z.B. Werte eingegeben werden, die außerhalb der Grenzen liegen. Um dies zu verhindern, muß der Compiler mit dem Compilerschalter **{$R+}** angewiesen werden, zusätzlichen Prüfcode zu erzeugen, was auch in der IDE mit dem Menüpunkt "**Option/Compiler.../Bereichsüberprüfung**" global geschehen kann. In diesem Fall gilt die Prüfung für das gesamte Programm; die Ausführungsgeschwindigkeit kann abfallen. Bei einer fehlerhaften Eingabe bricht das Programm mit der Fehlermeldung "Runtime error 201 at ....." (Range check error) ab, was mit Sicherheit besser ist als fehlerhaftes weiteres Ausführen. Da der Compilerschalter R lokal wirkt, kann er auch wieder abgeschaltet werden und zwar mit **{$R-}**. Compilerschalter sind im Kapitel 5.12.1 beschrieben.

Demo-Programm:

```
program Pas08071;  { Demo: Aufzählungstypen und Teilbereichstypen.
                         Vorgriff auf "repeat/until", "case ... of"
                         und "if/then/else"}
uses
  CRT;   { Unit CRT wegen ClrScr und GotoXY }

type        { Indirekte Deklaration von Aufzählungs- und Teilbereichs-
              typen mit "type" }
  Wochentage  = (Mon, Die, Mit, Don, Fre, Sam, Son); { Aufzählungstyp}
  Arbeitstage =  Mon..Fre; { Teilbereichstyp eines vorher deklarierten
                            Aufzählungstypen }
  Monatstage  = 1..31;                          { Teilbereichstyp }
  Noten       = (Fuenf, Vier, Drei, Zwei, Eins);   { Aufzählungstyp  }

var
  Zahl:      1..5;     { direkte Deklaration eines Teilbereichstypen }
  Freude:    (klein, maessig, mittelpraechtig, gross);
                       { direkte Deklaration eines Aufzählungstypen  }
  Tag:       Arbeitstage;
  Monatstag: Monatstage;
  Note:      Noten;
  Bestanden: Boolean;

begin
```

```
  ClrScr;
  repeat
    GotoXY(5, 2);
    Write('Eingabe einer Zahl für Note (1 bis 5): ');
    Readln(Zahl);
  until Zahl in [1..5];     { Vorgriff: Mengenoperator "in"}
  case Zahl of              { Vorgriff: Selektion mit "case ... of"}
    1: begin Note := Eins;  Freude := gross;            end;
    2: begin Note := Zwei;  Freude := gross;            end;
    3: begin Note := Drei;  Freude := mittelpraechtig;  end;
    4: begin Note := Vier;  Freude := maessig;          end;
    5: begin Note := Fuenf; Freude := klein;            end;
  end;
  if Note > Vier     { Man beachte die Reihenfolge in der
                       Deklaration des Typen "Noten"}
     then Bestanden := True
     else Bestanden := False;
  if Freude >= mittelpraechtig
     then Writeln('Hurra !!')
     else Writeln('Na ja ..');
  Monatstag  := 13;
  Tag        := Fre;

  if (Freude <= maessig) and (Monatstag = 13) and (Tag = Fre)
      then Writeln('Man ist ja nicht abergläubisch, aber ....');

  ReadLn;
end.
```