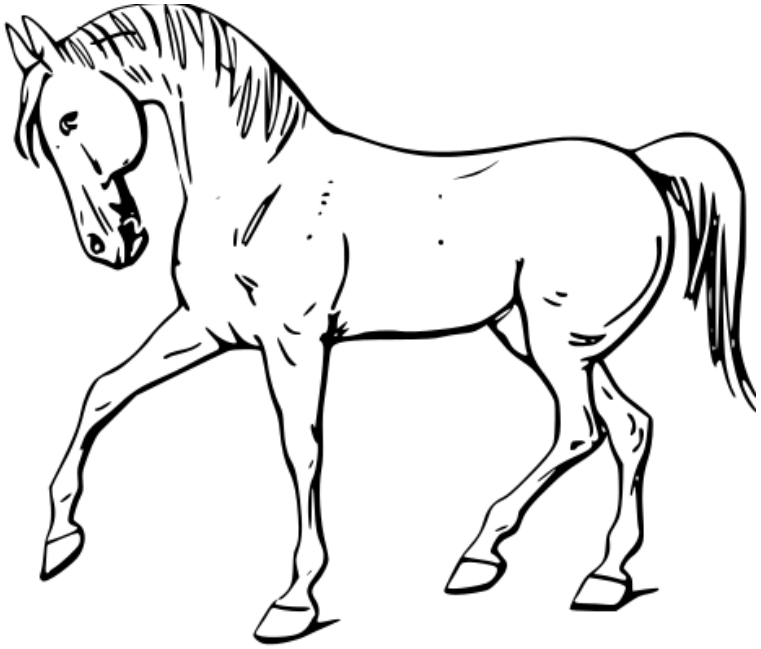
FreeBASIC-Einsteigerhandbuch

Grundlagen der Programmierung in FreeBASIC



von S. Markthaler

Stand: 2. April 2022

Einleitung

1. Über das Buch

Dieses Buch ist für Programmieranfänger gedacht, die sich mit der Sprache FreeBASIC beschäftigen wollen. Es setzt keine Vorkenntnisse über die Computerprogrammierung voraus. Sie sollten jedoch wissen, wie man einen Computer bedient, Programme installiert und startet, Dateien speichert usw. Kenntnisse über die Bedienung der Konsole (auch unter dem Namen Shell, Eingabeaufforderung u. a. bekannt) ist zwar nicht notwendig, zur Bedienung des Compilers aber durchaus nützlich.

Das Buch will einen möglichst umfassenden Überblick über die Sprache FreeBASIC geben. Konzepte, die zwar noch existieren, aber als veraltet gelten, werden dabei nur kurz angeschnitten, ebenso Konzepte, die sich zwar über FreeBASIC nutzen lassen, selbst aber nicht zur Sprache gehören. Ich bitte um Verständnis, dass dies den Rahmen des Buches deutlich sprengen würde. Außerdem handelt es sich bei diesem Buch um ein Handbuch und nicht um ein Tutorial. Die einzelnen Elemente sind so aufeinander aufgebaut, dass sie auch ohne Vorkenntnisse nacheinander erarbeitet werden können. Nutzen Sie jedoch ausgiebig die Möglichkeit, die erworbenen Kenntnisse in eigenen Programmen zu vertiefen! Programmieren lernen Sie nicht aus Büchern, sondern ausschließlich durch eigenes Anwenden. (Dennoch hoffe ich, dass ich mit diesem Buch eine gute Hilfestellung an die Hand geben kann.)

Wenn Sie bereits mit Q(uick)BASIC gearbeitet haben, finden Sie in Kapitel 1.3 eine Zusammenstellung der Unterschiede zwischen beiden Sprachen. Sie erfahren dort auch, wie Sie Q(uick)BASIC-Programme für FreeBASIC lauffähig machen können.

Wenn Sie noch über keine Programmiererfahrung verfügen, empfiehlt es sich, die Kapitel des Buches in der vorgegebenen Reihenfolge durchzuarbeiten. Manche Inhalte sind möglicherweise beim ersten Lesen noch schwer zu verstehen und erschließen sich erst später im Zusammenhang mit anderen Kapiteln. In diesem Fall ist es auch nicht schlimm, wenn Sie ein Kapitel zunächst nur überfliegen und später bei Bedarf wieder darauf zurückkommen. Wenn Ihnen dagegen einige Konzepte bereits bekannt sind, können Sie auch direkt zu den Kapiteln springen, die Sie interessieren.

2. In diesem Buch verwendete Konventionen

In diesem Buch tauchen verschiedene Elemente wie Variablen, Schlüsselwörter und besondere Textabschnitte auf. Damit Sie sich beim Lesen schnell zurechtfinden, werden diese Elemente kurz vorgestellt. Befehle und Variablen, die im laufenden Text auftauchen, werden in `nichtproportionaler Schrift` dargestellt. Schlüsselwörter wie **`PRINT`** werden in Fettdruck geschrieben, während für andere Elemente wie `variablenname` die normale Schriftstärke eingesetzt wird.

Quelltexte werden vollständig in nichtproportionaler Schrift gesetzt und mit einem Begrenzungsrahmen dargestellt. Auch hier werden Schlüsselwörter fett gedruckt. Der Dateiname des Programmes wird oberhalb des Quelltextes angezeigt.

Quelltext 1.1: Hallo Welt

```
' Kommentar: Ein gewoehnliches Hallo-Welt-Programm
CLS
PRINT "Hallo FreeBASIC-Welt!"
SLEEP
END
```

Es empfiehlt sich, die Programme abzutippen und zu testen. Die meisten Programme sind sehr kurz und können schnell abgetippt werden – auf der anderen Seite werden Sie Codebeispiele, die Sie selbst getippt haben, leichter behalten, und Sie können (und sollen!) mit dem Quelltext experimentieren. Alle fünf Zeilen wird die aktuelle Zeilennummer angezeigt; diese ist selbst nicht Bestandteil des Programmes und soll dementsprechend auch nicht mit abgetippt werden. Nichtsdestotrotz liegen auch alle Quelltexte als ZIP-Datei vor.

Die Ausgabe des Programmes wird ebenfalls in nichtproportionaler Schrift aufgelistet und durch eine graue Box umschlossen:

Ausgabe

```
Hallo FreeBASIC-Welt!
```

Diese Ausgabe kann mit der eigenen verglichen werden, um zu überprüfen, ob das Programm korrekt eingegeben wurde.

Zusätzliche Informationen zum aktuellen Thema werden in Hinweisboxen dargestellt.

Hinweis:
Diese Boxen enthalten ergänzende Informationen, die mit dem aktuell behandelten Thema zusammenhängen.

Informationen, die speziell für Umsteiger von QuickBASIC oder für fortgeschrittenere Programmierer gedacht sind, werden durch ein Werkzeug-Bild gekennzeichnet.

Hinweis:
Hier können z. B. Unterschiede zu QuickBASIC oder zu früheren FreeBASIC-Versionen aufgeführt werden. Für die Box wird dann auch die Überschrift entsprechend angepasst.

Ein mögliches Problem wird in einer Warnung angezeigt. Diese wird ähnlich dargestellt wie die Hinweisbox, jedoch mit einem Warnsymbol.

Achtung:
Warnungen sollten auf jeden Fall beachtet werden, da es sonst im Programm zu Problemen kommen kann!

3. Rechtliches

4. Weitere Informationen

Sämtliche längeren Quelltexte stehen auf der Projektseite
https://www.freebasic-portal.de/projekte/99
zum Download zur Verfügung. Dort erhalten Sie auch weitere Informationen.

5. Danksagung

Mein Dank gilt allen FreeBASICern, die direkt oder indirekt an der Entstehung des Buches mitgewirkt haben. Einige Inhalte wurden mit mehr oder weniger großen Anpassungen aus der FreeBASIC-Referenz übernommen. Daher geht ein besonderer Dank an alle Mitstreiter, die mit mir zusammen am Entstehen dieser großartigen Refernz mitgewirkt haben. Vielen Dank auch an Tom, der einige gute Verbesserungsvorschläge eingebracht hat.

Inhaltsverzeichnis

Einleitung **ii**
1. Über das Buch . . . ii
2. In diesem Buch verwendete Konventionen . . . iii
3. Rechtliches . . . iv
4. Weitere Informationen . . . v
5. Danksagung . . . v

I. Einführung **1**

1. Eine kurze Einführung in FreeBASIC **2**
1.1. Was ist eine Programmiersprache? . . . 2
1.2. Was ist FreeBASIC? . . . 3
1.3. Unterschiede zu QuickBASIC . . . 3
1.4. Vor- und Nachteile von FreeBASIC . . . 5

2. Installation und Inbetriebnahme **8**
2.1. Installation . . . 8
2.1.1. Installation unter Windows . . . 8
2.1.2. Installation unter Linux . . . 10
2.2. Erstes Programm . . . 11
2.3. Problembehebung . . . 12
2.3.1. Probleme beim Start . . . 12
2.3.2. Probleme beim Compilieren . . . 12
2.4. Wie wird aus meinem Quelltext ein Programm? . . . 13

II. Grundkonzepte **15**

3. Aufbau eines FreeBASIC-Programmes **16**
3.1. Grundaufbau und Ablauf . . . 16
3.2. Kommentare . . . 17

3.3. Zeilenfortsetzungszeichen . . . 18
3.4. Trennung von Befehlen mit Doppelpunkt . . . 18

4. Bildschirmausgabe 20
4.1. Der **PRINT**-Befehl . . . 20
4.2. Ausgabe von Variablen . . . 22
4.2.1. Deklaration von Variablen . . . 22
4.2.2. Wertzuweisung . . . 23
4.2.3. Wertvertauschung . . . 25
4.2.4. Ausgabe des Variablenwertes . . . 26
4.3. Formatierung der Ausgabe . . . 26
4.3.1. Direkte Positionierung mit **LOCATE** . . . 27
4.3.2. Positionierung mit **TAB** und **SPC** . . . 28
4.3.3. Cursor-Position ermitteln: **POS** und **CSRLIN** . . . 28
4.3.4. Farbige Ausgabe . . . 29
4.3.5. Bildschirm löschen mit **CLS** . . . 30
4.4. Fragen zum Kapitel . . . 31

5. Tastatureingabe 33
5.1. Der **INPUT**-Befehl . . . 33
5.2. Eingabe einzelner Zeichen . . . 36
5.2.1. **INPUT()** als Funktion . . . 36
5.2.2. **INKEY()** . . . 37
5.3. Fragen zum Kapitel . . . 38

6. Variablen und Konstanten 39
6.1. Ganzzahlen . . . 39
6.1.1. Verfügbare Ganzzahl-Datentypen . . . 39
6.1.2. Alternative Schreibweise: **INTEGER<n>** . . . 41
6.1.3. Rechnen mit Ganzzahlen . . . 42
6.1.4. Kurzschreibweisen . . . 44
6.2. Gleitkommazahlen . . . 44
6.2.1. Darstellungsbereich von Gleitkommazahlen . . . 44
6.2.2. Rechengenauigkeit bei Gleitkommazahlen . . . 46
6.3. Zeichenketten . . . 48
6.3.1. Arten von Zeichenketten . . . 49
6.3.2. Aneinanderhängen zweier Zeichenketten . . . 50
6.4. Wahrheitswerte . . . 52

6.5. Konstanten . . . 53
6.6. Nummerierungen (Enumerations) . . . 54
6.7. Weitere Speicherstrukturen . . . 56
6.8. Fragen zum Kapitel . . . 56

7. Benutzerdefinierte Datentypen (UDTs) 57
7.1. Deklaration . . . 57
7.2. Mitgliederzugriff mit **WITH** . . . 59
7.3. Speicherverwaltung . . . 60
7.4. Bitfelder . . . 64
7.5. UNIONs . . . 65
7.6. Fragen zum Kapitel . . . 67

8. Datenfelder (Arrays) 69
8.1. Deklaration und Zugriff . . . 69
8.2. Mehrdimensionale Arrays . . . 70
8.3. Dynamische Arrays . . . 71
8.3.1. Deklaration und Redimensionierung . . . 72
8.3.2. Werte-Erhalt beim Redimensionieren . . . 73
8.4. Weitere Befehle . . . 74
8.4.1. Grenzen ermitteln: **LBOUND()** und **UBOUND()** . . . 74
8.4.2. Nur die Dimensionenzahl festlegen: **ANY** . . . 75
8.4.3. Implizite Grenzen . . . 76
8.4.4. Löschen mit **ERASE** . . . 77
8.5. Fragen zum Kapitel . . . 78

9. Pointer (Zeiger) 79
9.1. Speicheradresse ermitteln . . . 79
9.2. Pointer deklarieren . . . 80
9.3. Speicherverwaltung bei Arrays . . . 81
9.4. Fragen zum Kapitel . . . 82

10. Bedingungen 83
10.1. Einfache Auswahl: **IF** ... **THEN** . . . 83
10.1.1. Einzeilige Bedingungen . . . 83
10.1.2. Mehrzeilige Bedingungen (**IF**-Block) . . . 84
10.1.3. Alternativauswahl . . . 86

10.2. Bedingungsstrukturen . . . 87
10.2.1. Vergleiche . . . 87
10.2.2. Logische Operatoren . . . 89
10.2.3. Das Binärsystem . . . 90
10.2.4. **AND** und **OR** als Bit-Operatoren . . . 91
10.2.5. **ANDALSO** und **ORELSE** . . . 92
10.2.6. Weitere Bit-Operatoren: **XOR**, **EQV**, **IMP** und **NOT** . . . 94
10.3. Mehrfachauswahl: **SELECT CASE** . . . 94
10.3.1. Grundaufbau . . . 96
10.3.2. Erweiterte Möglichkeiten . . . 97
10.3.3. **SELECT CASE AS CONST** . . . 98
10.4. Bedingungsfunktion: **IIF()** . . . 99
10.5. Welche Möglichkeit ist die beste? . . . 99
10.6. Fragen zum Kapitel . . . 100

11. Schleifen und Kontrollanweisungen **101**
11.1. Sprunganweisungen . . . 101
11.2. **DO** ... **LOOP** . . . 103
11.3. **WHILE** ... **WEND** . . . 105
11.4. **FOR** ... **NEXT** . . . 105
11.4.1. Einfache **FOR**-Schleife . . . 106
11.4.2. **FOR**-Schleife mit angegebener Schrittweite . . . 107
11.4.3. **FOR** i **AS** datentyp . . . 109
11.4.4. Übersprungene Schleifen . . . 110
11.4.5. Fallstricke . . . 110
11.5. Kontrollanweisungen . . . 112
11.5.1. Fortfahren mit **CONTINUE** . . . 112
11.5.2. Vorzeitiges Verlassen mit **EXIT** . . . 113
11.5.3. Kontrollstrukturen in verschachtelten Blöcken . . . 115
11.6. Systemauslastung in Dauerschleifen . . . 116
11.7. Fragen zum Kapitel . . . 117

12. Prozeduren und Funktionen **119**
12.1. Einfache Prozeduren . . . 119
12.2. Verwaltung von Variablen . . . 120
12.2.1. Parameterübergabe . . . 121
12.2.2. Globale Variablen . . . 123
12.2.3. Statische Variablen . . . 125

12.3. Unterprogramme bekannt machen . . . 127
12.3.1. Die Deklarationszeile . . . 127
12.3.2. Optionale Parameter . . . 129
12.3.3. **OVERLOAD** . . . 130
12.4. Funktionen . . . 132
12.5. Weitere Eigenschaften der Parameter . . . 134
12.5.1. Übergabe von Arrays . . . 134
12.5.2. **BYREF** und **BYVAL** . . . 136
12.5.3. Parameterübergabe **AS CONST** . . . 138
12.5.4. Aufrufkonvention für die Parameter . . . 139
12.5.5. Variable Parameterlisten . . . 141
12.6. Fragen zum Kapitel . . . 146

III. Datenverarbeitung **148**

13. Datentypen umwandeln **149**
13.1. Allgemeine Umwandlung: **CAST()** . . . 149
13.2. Umwandlung in einen String . . . 151
13.3. Implizite Umwandlung . . . 152
13.4. ASCII-Code . . . 153
13.4.1. Ursprung und Bedeutung des ASCII-Codes . . . 153
13.4.2. **ASC()** und **CHR()** . . . 154
13.4.3. Binäre Kopie erstellen . . . 156
13.5. Fragen zum Kapitel . . . 158

14. Verarbeitung von Zahlen **159**
14.1. Mathematische Funktionen . . . 159
14.1.1. Quadratwurzel, Absolutbetrag und Vorzeichen . . . 159
14.1.2. Winkelfunktionen . . . 160
14.1.3. Exponentialfunktion und Logarithmus . . . 162
14.1.4. Rundung . . . 163
14.1.5. Modulo-Berechnung . . . 165
14.1.6. Zufallszahlen . . . 167
14.2. Zahlensysteme . . . 169
14.2.1. Darstellung in Nicht-Dezimalsystemen . . . 170
14.2.2. **BIN()**, **OCT()** und **HEX()** . . . 170

14.3. Bit-Manipulationen . . . 172
14.3.1. Manipulation über Bitoperatoren . . . 173
14.3.2. Einzelne Bit lesen und setzen . . . 176
14.3.3. Vergleich . . . 178
14.3.4. **LOBYTE()** und seine Freunde . . . 178
14.3.5. Bit-Verschiebung . . . 179
14.4. Fragen zum Kapitel . . . 181

15. Stringmanipulation **182**
15.1. Speicherverwaltung . . . 182
15.1.1. Verwaltung eines **ZSTRING** . . . 182
15.1.2. Verwaltung eines **WSTRING** . . . 183
15.1.3. Verwaltung eines **STRING** . . . 184
15.2. Stringfunktionen . . . 185
15.2.1. **LEN()**: Länge eines Strings ermitteln . . . 185
15.2.2. Die Funktionen **SPACE()** und **STRING()** . . . 186
15.2.3. **LEFT()**, **RIGHT()** und **MID()** . . . 186
15.2.4. Zeichenketten modifizieren . . . 188
15.2.5. Führende/angehängte Zeichen entfernen: **LTRIM()**, **RTRIM()** und **TRIM()** . . . 190
15.2.6. Umwandlung in Groß- bzw. Kleinbuchstaben: **UCASE()** und **LCASE()** . . . 192
15.2.7. Teilstring suchen: **INSTR()** und **INSTRREV()** . . . 192
15.3. Umlaute und andere Sonderbuchstaben . . . 194
15.3.1. Problemstellung: Darstellung von Sonderzeichen . . . 194
15.3.2. Erster Lösungsversuch: Speicherformat anpassen . . . 195
15.3.3. Zweiter Lösungsversuch: Sonderzeichen als Konstanten . . . 196
15.4. Escape-Sequenzen . . . 197
15.5. Fragen zum Kapitel . . . 199

16. Datei-Zugriff **200**
16.1. Datei öffnen und schließen . . . 200
16.1.1. Dateinummer festlegen . . . 201
16.1.2. Der Dateimodus . . . 203
16.1.3. Zugriffsart beschränken . . . 205
16.1.4. Datei für andere Programme sperren . . . 206
16.1.5. Text-Encoding . . . 206
16.1.6. Datei(en) schließen . . . 207

16.2. Lesen und Schreiben sequentieller Daten . . . 209
16.2.1. PRINT# und WRITE# . . . 209
16.2.2. INPUT#, LINE INPUT# . . . 210
16.2.3. **INPUT()** und **WINPUT()** für Dateien . . . 211
16.3. Zugriff auf binäre Daten . . . 211
16.3.1. Binäre Daten speichern . . . 212
16.3.2. Binäre Daten einlesen . . . 213
16.3.3. Binäre Speicherung von Strings variabler Länge . . . 216
16.3.4. Position des Dateizeigers . . . 217
16.4. Dateiende ermitteln . . . 219
16.4.1. End Of File (EOF) . . . 219
16.4.2. Length Of File (LOF) . . . 219
16.5. Datei löschen . . . 220
16.6. Standard-Datenströme . . . 221
16.6.1. Eine kurze Einführung . . . 221
16.6.2. Öffnen und Schließen eines Standard-Datenstroms . . . 222
16.6.3. OPEN PIPE . . . 224
16.7. Hardware-Zugriffe: **OPEN COM** und **OPEN LPT** . . . 224
16.8. Fragen zum Kapitel . . . 226

17. Betriebssystem-Anweisungen **228**
17.1. Ordnerfunktionen . . . 228
17.1.1. Kurze Einführung in das Ordnersystem . . . 228
17.1.2. Arbeitsverzeichnis anzeigen und wechseln: **CURDIR()** und **CHDIR()** 230
17.1.3. Programmpfad anzeigen: **EXEPATH()** . . . 231
17.1.4. Ordner anlegen und löschen: **MKDIR()** und **RMDIR()** . . . 232
17.2. Dateifunktionen . . . 233
17.2.1. Dateien umbenennen und verschieben: **NAME()** . . . 234
17.2.2. Dateien kopieren: **FILECOPY()** . . . 235
17.2.3. Dateiinformationen abrufen: **FILEATTR()**, **FILELEN()** und **FILEDATETIME()** . . . 235
17.2.4. Auf Existenz prüfen: **FILEEXISTS()** . . . 238
17.2.5. Dateien suchen: **DIR()** . . . 238
17.3. Externe Programme starten . . . 241
17.3.1. **CHAIN()**, **EXEC()** und **RUN()** . . . 241
17.3.2. **SHELL()** . . . 242
17.4. Kommandozeilenparameter auswerten . . . 243
17.5. Umgebungsvariablen abfragen und setzen . . . 245

17.6. Fragen zum Kapitel . . . 246

18. Datum und Zeit **248**
18.1. Zeitmessung . . . 248
18.1.1. **SLEEP** . . . 248
18.1.2. **TIMER()** . . . 249
18.2. Abruf und Änderung der Systemzeit . . . 250
18.3. Serial Numbers . . . 251
18.3.1. Aktuelles Datum mit **NOW()** und **FORMAT** . . . 252
18.3.2. Setzen einer *Serial Number* . . . 253
18.3.3. Teilinformationen einer *Serial Number* . . . 256
18.3.4. Namen des Wochentage und Monate . . . 258
18.3.5. Rechnen mit Datum und Zeit . . . 259
18.4. Fragen zum Kapitel . . . 261

IV. Anhang **262**

A. Antworten zu den Fragen **263**

B. Compiler-Optionen **280**
B.1. Grundlegende Compileroptionen . . . 280
B.2. Dialekt wählen . . . 281
B.3. Fehlerbehandlung einstellen . . . 281
B.4. Unterdrüchung des Konsolenfensters . . . 282
B.5. Weitere Optionsschalter . . . 282
B.5.1. Erzeugung von Bibliotheken (Libraries) mit privaten/öffentlichen Prozeduren . . . 283
B.5.2. Behandlung von Programmdateien (*.bas und *.bi) . . . 283
B.5.3. Bedingtes Compilieren und Präprozessor . . . 284
B.5.4. Fehlerbehandlung . . . 285
B.5.5. Programmerstellung . . . 285
B.5.6. Informationen . . . 288

C. ASCII-Zeichentabelle **289**

D. MULTIKEY-Scancodes **290**

E. Konstanten und Funktionen der *vbcompat.bi* **291**
E.1. Datum und Zeit . . . 291
E.1.1. Verfügbare Funktionen . . . 291
E.1.2. Definierte Konstanten . . . 292
E.2. Formatierungsmöglichkeiten durch `FORMAT()` . . . 294
E.3. Konstanten für die Attribute von `DIR` . . . 295
E.4. Dateifunktionen . . . 295
E.4.1. Verfügbare Funktionen . . . 295
E.4.2. Definierte Konstanten . . . 295

F. Vorrangregeln (Hierarchie der Operatoren) **296**

G. FreeBASIC-Schlüsselwörter **298**
G.1. Schlüsselwörter . . . 298
G.2. Metabefehle . . . 330
G.3. Vordefinierte Symbole . . . 331

Index **340**

Liste der Quelltexte **340**

Teil I.

Einführung

1. Eine kurze Einführung in FreeBASIC

Das Kapitel will Ihnen die Grundbegriffe zum Thema Programmierung und Compilierung nahe bringen. In Kapitel 1.3 erfahren Sie außerdem, welche Unterschiede zwischen Q(uick)BASIC und FreeBASIC bestehen.

1.1. Was ist eine Programmiersprache?

Computer besitzen eine eigene Maschinensprache, in der ihre Programme ablaufen. Da ein Computer in Binärcode „denkt“, der für Menschen sehr schwer verständlich ist, werden Programmiersprachen eingesetzt, um Computerprogramme in einem für Menschen leichter verständlichen Programmcode, sogenannten Quelltext, niederzuschreiben. Nun muss jedoch dieser Programmcode zuerst in Maschinensprache übersetzt werden, um vom Computer verarbeitet werden zu können. Ursprünglich übernahm bei BASIC diese Aufgabe ein Interpreter, der die Anweisungen bei jeder Ausführung Zeile für Zeile übersetzte. Heute werden vielfach *Compiler* eingesetzt, die den Quelltext einmal komplett in Maschinencode übersetzen. Diese übersetzten Programme – unter Windows handelt es sich dabei um die bekannten .exe-Dateien – können anschließend jederzeit ausgeführt werden, ohne dass man sie jedes Mal aufs Neue übersetzen muss.

Wurde ein Quelltext erfolgreich z. B. für Windows übersetzt, so kann das erzeugte Programm von allen binärkompatiblen Windowssystemen ausgeführt werden – nicht jedoch von anderen Systemen wie z. B. Linux. Die Programmiersprache selbst ist dagegen nicht an das Betriebssystem gebunden. Der Quelltext eines Programmes kann also für mehrere verschiedene Betriebssysteme übersetzt werden, sofern für dieses System ein Compiler existiert. Man spricht dabei von *Maschinenunabhängigkeit*. Während das compilierte Programm problemlos an andere Nutzer desselben Betriebssystems weitergegeben werden kann, ohne dass diese den Compiler besitzen müssen, kann der Quelltext ohne weiteres zwischen verschiedenen Betriebssystemen ausgetauscht werden, muss dann aber noch für das neue System compiliert werden.

1.2. Was ist FreeBASIC?

Wie der Name schon sagt, ist FreeBASIC ein BASIC-Dialekt. BASIC steht für *Beginner's All-purpose Symbolic Instruction Code* (symbolische Allzweck-Programmiersprache für Anfänger) und wurde 1964 mit dem Ziel entwickelt, eine einfache, für Anfänger geeignete Programmiersprache zu erschaffen.

Der Befehlssatz von FreeBASIC baut auf QuickBASIC der Firma Microsoft auf, welches in der abgespeckten Version namens QBasic noch unter Windows 98 mit der Installations-CD ausgeliefert wurde. Vorrangiges Ziel bei der Entwicklung von FreeBASIC war die Kompatibilität zu QuickBASIC. Jedoch gibt es für FreeBASIC zur Zeit keinen Interpreter (jedenfalls keinen, der den vollen Befehlssatz unterstützt), sondern stattdessen den FreeBASIC-Compiler *fbc* zur Erstellung von 32- bzw. 64-Bit-Anwendungen. Der Compiler steht für Microsoft Windows, Linux, DOS und Xbox zur Verfügung. In FreeBASIC erstellte Programme sind – von wenigen Ausnahmen abgesehen – auf allen drei Plattformen ohne Änderung lauffähig. Oder, etwas genauer ausgedrückt: Quelltexte, die in FreeBASIC geschrieben wurden, können – von wenigen Ausnahmen abgesehen – für alle genannten Plattformen ohne Änderung compiliert werden.

Der Compiler ist außerdem Open Source, was bedeutet, dass der Quelltext frei betrachtet und an die eigenen Bedürfnisse angepasst werden kann.

1.3. Unterschiede zu QuickBASIC

Dieser Abschnitt behandelt QuickBASIC und dessen abgespeckte Version QBASIC gleichermaßen; der Einfachheit halber wird nur von QuickBASIC gesprochen. Wer noch nie mit QuickBASIC in Berührung gekommen ist, kann den Abschnitt getrost überspringen – er ist eher für Programmierer interessant, die von QuickBASIC zu FreeBASIC wechseln wollen.

Trotz hoher Kompatibilität zu QuickBASIC gibt es eine Reihe von Unterschieden zwischen beiden BASIC-Dialekten. Einige davon beruhen auf der einfachen Tatsache, dass QuickBASIC für MS-DOS entwickelt wurde und einige Elemente wie beispielsweise direkte Hardware-Zugriffe unter echten Multitaskingsystemen wie höhere Windows-Systeme oder Linux nicht oder nur eingeschränkt laufen. Des Weiteren legt FreeBASIC größeren Wert auf eine ordnungsgemäße Variablendeklaration, womit Programmierfehler leichter vermieden werden können.

Kompatibilitätsmodus:
Mit der Compiler-Option `-lang qb` kann eine größere Kompatibilität zu QuickBASIC erzeugt werden. Verwenden Sie diese Option, um alte Programme zum Laufen zu bringen. Mehr dazu erfahren Sie in Anhang B.

- **Nicht explizit deklarierte Variablen (`DEF###`)**
  In FreeBASIC müssen alle Variablen und Arrays explizit (z. B. durch **`DIM`**) deklariert werden. Die Verwendung von **`DEFINT`** usw. ist nicht mehr zulässig.

- **`OPTION BASE`**
  Die Einstellung der unteren Array-Grenze mittels **`OPTION BASE`** ist nicht mehr zulässig. Sofern die untere Grenze eines Arrays nicht explizit angegeben wird, verwendet FreeBASIC den Wert 0.

- **Datentyp `INTEGER`**
  QuickBASIC verwendet 16 Bit für die Speicherung eines Integers. In FreeBASIC sind es, je nach Compilerversion, 32 bzw. 64 Bit. Verwenden Sie den Datentyp **`SHORT`**, wenn Sie eine 16-Bit-Variable verwenden wollen.

- **Funktionsaufruf**
  Alle Funktionen und Prozeduren, die aufgerufen werden, bevor sie definiert wurden, müssen mit **`DECLARE`** deklariert werden. Der Befehl **`CALL`** wird in FreeBASIC nicht mehr unterstützt.

- **Verwendung von Suffixen**
  Suffixe (z. B. ein $ am Ende des Variablennamens, um die Variable als String zu kennzeichnen) werden nicht mehr unterstützt. Jede Variable und ihr Typ müssen, z. B. durch **`DIM`**, explizit deklariert werden.
  Damit dürfen Variablen auch keinen Namen erhalten, der bereits von einem Schlüsselwort belegt ist.

- **Padding von `TYPE`-Feldern**
  Unter QuickBASIC wird kein Padding durchgeführt. In FreeBASIC werden UDTs standardmäßig auf ein Vielfaches von 4 oder 8 Byte ausgedehnt, abhängig vom System. Dieses Verhalten kann durch das Schlüsselwort **`FIELD`** angepasst werden.

- **Strings**
  Unter FreeBASIC wird an das Ende des Strings intern ein **`CHR(0)`** angehängt.

Strings können in der 32-Bit Version des Compilers eine maximale Länge von 2 GB besitzen, in der 64-Bit-Version eine maximale Länge von 8.388.607 TB.

- **BYREF**
  Alle Zahlen-Variablen werden standardmäßig **BYVAL** übergeben. Arrays werden immer **BYREF** übergeben; hier ist eine Übergabe mittels **BYVAL** nicht möglich.

- **Punkte in Symbolnamen**
  Punkte in Symbolnamen sind nicht mehr zulässig, da die Punkte für die objektorientierte Programmierung eine neue Bedeutung besitzen.

- **nicht mehr unterstützte Befehle**
  GOSUB/**RETURN**[2], **ON...GOSUB**, **ON...GOTO**, **ON ERROR** und **RESUME** sind nicht mehr erlaubt. Es gibt jedoch andere Befehlskonstrukte, um die gewünschten Ergebnisse zu erzielen.
  In Kommentare eingebundene Meta-Befehle **'$DYNAMIC**, **'$STATIC**, **'$INCLUDE** und **'$INCLIB** existieren nicht mehr. Verwenden Sie **#INCLUDE** bzw. **#INCLIB**, um diese Befehle zu ersetzen.
  CALL existiert nicht mehr und kann einfach weggelassen werden.
  LET kann nicht mehr für eine einfache Variablenzuweisung verwendet werden; lassen Sie dazu **LET** einfach weg. Stattdessen erlaubt der Befehl eine mehrfache Variablenzuweisung für die Attribute eines UDTs.

- **numerische Labels**
  Labels, die nur aus Ziffern bestehen, werden nicht mehr unterstützt.

- **globale Symbole, die den Namen eines internen Schlüsselworts besitzen**
  Möchten Sie diese weiterhin benutzen, müssen Sie sie in einem **NAMESPACE** deklarieren.

1.4. Vor- und Nachteile von FreeBASIC

Jede Programmiersprache hat ihre Vor- und Nachteile. Je nachdem, welche Ansprüche der Programmierer an sein Programm stellt, ist die eine oder die andere Sprache besser für ihn geeignet. Für die Wahl der richtigen Programmiersprache ist es daher wichtig, ihre Vor- und Nachteile zu kennen.

Als 32- bzw. 64-Bit-BASIC-Compiler besitzt *fbc* eine Reihe von Vorzügen:

[2] **RETURN** kann stattdessen eingesetzt werden, um Prozeduren vorzeitig zu verlassen.

- FreeBASIC ist eine leicht erlernbare Sprache mit einem einprägsamen Befehlssatz, der an die englische Sprache angelehnt ist. Die Funktionsweise der Grundbefehle lässt sich damit auch bereits mit wenigen Programmierkenntnissen recht leicht erschließen.
- Der Compiler erzeugt ausführbare 32- bzw. 64-Bit-Anwendungen, die auf anderen Computern mit gleichem Betriebssystem ohne Zusatzprogramme (Interpreter, Virtual Machine o. ä.) ausgeführt werden können.
- Sofern keine speziellen systemabhängigen Bibliotheken eingesetzt werden, kann der Quelltext ohne Änderungen für Windows und Linux compiliert werden. Durch den Einsatz von Präprozessoren kann die Portabilität weiter erhöht werden.
- FreeBASIC kann alle in C geschriebenen Bibliotheken einbinden. Viele davon können ohne zusätzliche Vorarbeit direkt eingesetzt werden.
- Der Quelltext kann ohne Schwierigkeiten in mehrere Module aufgeteilt werden, wodurch einzelne Codeabschnitte leicht wiederverwertet werden können. Je nach Deklaration sind die in den Modulen definierten Variablen und Prozeduren nur im jeweiligen Modul oder im ganzen Programm aufrufbar.
- Die eingebaute *gfx*-Grafikbibliothek erlaubt eine einfache Erstellung grafischer Anwendungen. Auch Windows-Bitmaps können direkt eingesetzt werden. Für den Einsatz weiterer Grafikformate wie PNG und JPEG gibt es eine Reihe von unterstützenden Bibliotheken.
- FreeBASIC unterstützt Grundlagen der Objektorientierung. Dazu gehören benutzerdefinierte Datentypen, Überladung von Operatoren, Kapselung und Vererbung.
- Der eingebauten Inline-Assembler erlaubt die direkte Einbindung von Assemblerprogrammen in den Quelltext. Zeitkritische Operationen können damit unter Umständen optimiert werden.

Jedes Programmierkonzept besitzt jedoch auch Nachteile. Für FreeBASIC sind zu nennen:

- Ein FreeBASIC-Programm kann, im Gegensatz zu Interpretersprachen, erst ausgeführt werden, wenn es compiliert wurde. Nach jeder Änderung im Programm ist ein neuer Compiliervorgang nötig. Da dieser jedoch recht schnell durchgeführt wird, ist das nur bei sehr umfangreichen Quelltexten mit entsprechend langer Compilierzeit ein Nachteil.

- Der für Anfänger leicht verständlich gehaltene Befehlssatz sorgt dafür, dass viele Befehlsstrukturen etwas ausschweifender formuliert werden müssen. Abstraktere Sprachen erlauben hier oft eine knappere und elegantere (aber für Anfänger schwerer verstehbare) Form.
- So wie viele BASIC-Dialekt besitzt FreeBASIC, im Vergleich zu anderen Sprachen, eine sehr große Zahl an fest integrierten Schlüsselwörtern, die (wie bereits erwähnt) nicht ohne Weiteres als Namen für Variablen oder Funktionen verwendet werden können. Diese Situation wird durch das Ziel der Rückwärtskompatilbilität noch verschärft.
- Einige Konzepte, die aus der objektorientierten Programmierung bekannt sind, sind in FreeBASIC nicht implementiert. Weder Mehrfachvererbung noch Interfaces werden unterstützt. Auch generische Typen können nicht eingesetzt werden.
- Im Gegensatz zu einem BASIC-Interpreter führt FreeBASIC keine Laufzeitüberprüfung der Variablen durch. Der Überlauf einer Variablen wird nicht als Fehler erkannt. Das kann genau so gewünscht sein, kann bei fehlender Sorgfalt aber auch zu schwer lokalisierbaren Fehlern führen.

2. Installation und Inbetriebnahme

Um mit FreeBASIC Programme zu schreiben und zu compilieren, benötigen Sie zwei Dinge: einen Editor und den Compiler. Als Editor können Sie jeden beliebigen Texteditor verwenden, der in der Lage ist, in ASCII-Code zu speichern. Allerdings gibt es einige Editoren, die speziell für FreeBASIC ausgelegt sind und eine Reihe von Annehmlichkeiten bieten, welche die Programmierung vereinfachen. Man spricht hierbei von einer *integrierten Entwicklungsumgebung*, auf englisch *Integrated Development Environment* oder kurz *IDE*.

Im Folgenden wird die Installation und Verwendung der IDE *wxFBE* erläutert, auch wenn Sie selbstverständlich einen anderen Editor verwenden können.[3] wxFBE besitzt gegenüber anderen FreeBASIC-IDEs den Vorteil, dass es unter Windows und Linux gleichermaßen eingesetzt werden kann.

2.1. Installation

Den Compiler finden Sie auf der (englischsprachigen) Herstellerseite:
https://www.freebasic.net
Die IDE wxFBE kann hier heruntergeladen werden:
https://www.freebasic-portal.de/projekte/wxfbe-69.html [4]
Für Windows steht ein Komplettpaket (wxFBE + fbc) zur Verfügung. Wenn Sie sich dafür entscheiden, ist kein getrennter Download des Compilers nötig.

2.1.1. Installation unter Windows

Komplettpaket

Das Komplettpaket installiert den Compiler und die IDE, außerdem wird wxFBE bereits auf den richtigen Dateipfad des Compilers einstellt. Der Begriff „Installation“ ist hier genau genommen zu hoch gegriffen – der Ordner muss lediglich an eine passende Stelle

[3] Eine Liste und Download-Möglichkeit gängiger FreeBASIC-IDEs finden Sie unter
http://www.freebasic-portal.de/downloads/ides-fuer-freebasic/

[4] Außerdem werden für wxFBE unter Windows die Microsoft Visual C++ 2008 Laufzeitkomponenten (oder höher) benötigt, die in neueren Windows-Versionen standardmäßig installiert ist

auf Ihrem Rechner oder auch auf einem USB-Stick entpackt werden; es finden keine Eingriffe in das System statt. Dementsprechend müssen die beiden Programme auch nicht deinstalliert werden (warum auch immer jemand auf den Gedanken kommen sollte, so etwas tun zu wollen), sondern können bei Bedarf einfach gelöscht werden.

Getrennte Installation

Ich persönlich bevorzuge eine getrennte Installation von IDE und Compiler, weil dadurch zum einen klarer wird, dass es sich um zwei verschiedene Komponenten handelt, und zum anderen das spätere Update einer der beiden Komponenten leichter wird, ohne dass die andere dabei angetastet werden muss. Vielfach wird z. B. von Neueinsteigern die Versionsnummer der IDE mit der Versionsnummer des Compilers verwechselt. Wenn die Entwicklung einer IDE eingestellt wird, werden oftmals die Komplettpakete nicht mehr aktualisiert, d. h. mit der letzten IDE-Version wird ein veralteter Compiler ausgeliefert. Bei wxFBE ist das leider bereits der Fall.

Allerdings ist bei der Installation etwas mehr Arbeit erforderlich. Die Einzelinstallation von wxFBE läuft zunächst genauso wie die Installation des Komplettpakets. Zusätzlich wird jedoch der Compiler benötigt, der auf http://freebasic.net zu finden ist (genauer gesagt finden Sie dort einen Link auf die entsprechende Sourceforge-Seite).

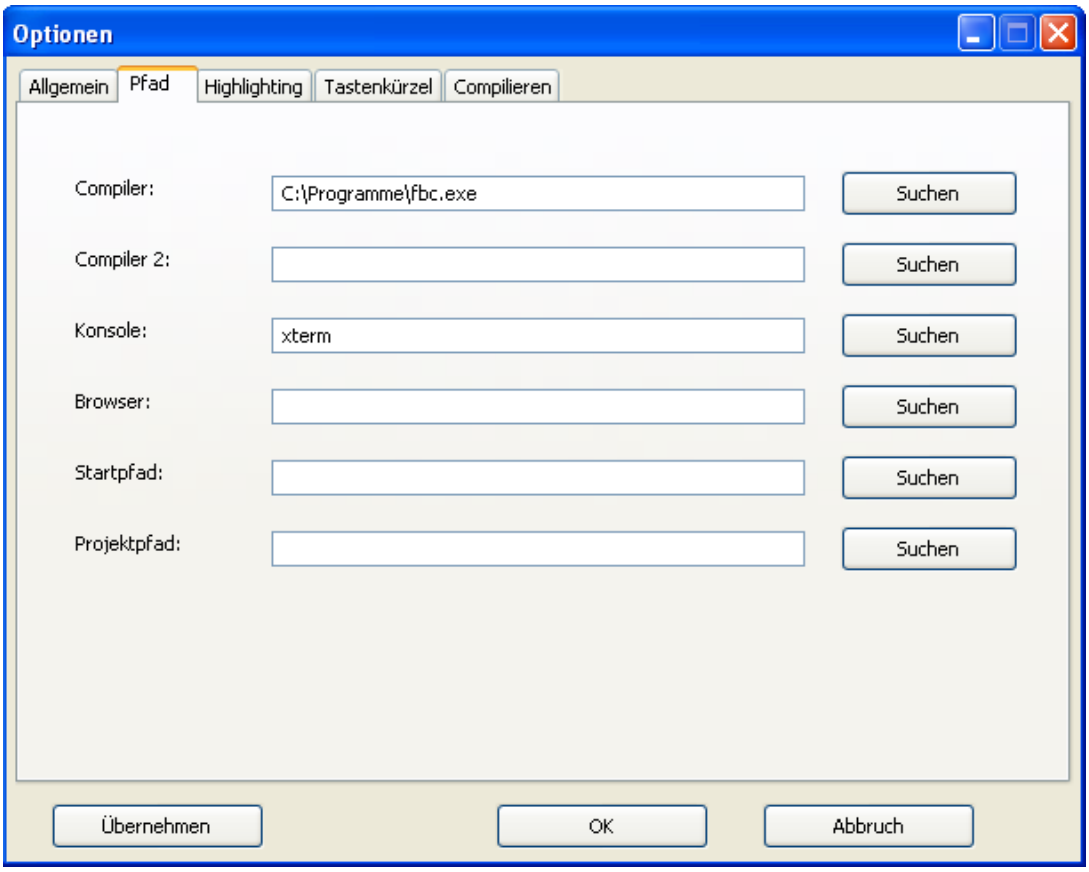
Der Download enthält eine Setup-Datei, welche den Compiler und alle benötigten Komponenten installiert (z. B. im Ordner `C:\Programme\FreeBASIC` – merken Sie sich bitte den Installationspfad; Sie werden ihn gleich noch einmal brauchen).

Anschließend muss wxFBE noch eingerichtet werden. Starten Sie die IDE über das Programm *wxFBE.exe*. Sofern Sie nicht das Komplettpaket installiert haben, erhalten Sie folgende Warnung:

```
Can't open wxFBE.xml - default settings will be used instead.
```

Dies sollte Sie nicht beunruhigen, es bedeutet lediglich, dass wxFBE zum ersten Mal gestartet wurde und noch keine Konfigurationsdatei existiert. Sie erhalten noch eine weitere Meldung und gelangen anschließend direkt in das Optionenmenü des Editors. Sollten Sie des Englischen nicht mächtig sein, empfiehlt es sich, als erstes unter „Language“ die gewünschte Sprache einzustellen und den Editor neu zu starten. Das Optionenmenü finden Sie anschließend im Menü „Optionen“ wieder. Öffnen Sie dort den Reiter „Pfad“ und tragen Sie den Installationspfad des Compilers ein.

Wundern Sie sich übrigens nicht über den eingetragenen Wert im Feld „Konsole“. Dieser ist nur für Linux-Nutzer interessant und wird unter Windows ignoriert.



2.1.2. Installation unter Linux

Für Linux gibt es kein wxFBE-Komplettpaket. Stattdessen laden Sie, wie oben im Abschnitt „Getrennte Installation“ beschrieben, Compiler und IDE einzeln.

FreeBASIC versucht so weit wie möglich, nur auf Bibliotheken zuzugreifen, bei denen davon ausgegangen werden kann, dass sie auf dem Rechner installiert sind. Zum Compilieren werden jedoch die Entwicklerpakete der Bibliotheken benötigt, die in vielen Systemen nicht vorinstalliert sind, jedoch in den Paketquellen vorliegen. Je nach Distribution können sich die Paketnamen unterscheiden. Lesen Sie bitte in der dem Compiler beigelegten Datei `readme.txt` nach, welche Pakete benötigt werden.

wxFBE greift für seine GUI auf wxWidget zurück und benötigt dazu die Bibliothek wx-c 0.9. Glücklicherweise liegt diese dem Download von wxFBE bereits bei. Das Pro-

gramm kann direkt auf diese Dateien zugreifen, ein Eingriff in das System ist also nicht nötig. Dies funktioniert allerdings nur, wenn beim Start das Arbeitsverzeichnis mit dem Programmverzeichnis identisch ist. Wenn Sie den Editor z. B. unter `~/wxfbe/` gespeichert haben, können Sie ihn nicht von `~/` aus aufrufen, da dann die Bibliothek nicht gefunden wird.

Wenn Sie wxFBE das erste Mal starten, werden Sie, wie schon weiter oben beschrieben, zwei Meldungen erhalten und anschließend zum Einstellungsmenü weitergeleitet. Geben Sie als Compilerpfad `fbc` ein und passen Sie ggf. den Pfad der Konsole an. Als Voreinstellung wurde eine Konsole gewählt, die mit großer Wahrscheinlichkeit in allen Distributionen verfügbar ist; es steht Ihnen selbstverständlich frei, hier Ihre Lieblingskonsole einzutragen.

2.2. Erstes Programm

Um zu testen, ob die Installation erfolgreich war, starten Sie wxFBE und öffnen Sie eine neue Datei (Menü „Datei“, Eintrag „Neu“ oder über das Icon ganz links, oder aber über das Tastaturkürzel `[Strg]+[N]`). Tippen Sie Ihr erstes Programm ein – im Laufe der Zeit hat sich dafür das „Hallo-Welt-Programm“ eingebürgert:

Quelltext 2.1: hallowelt.bas

```
PRINT "Hallo Welt!"
SLEEP
```

wxFBE verfügt über eine sogenannte *Schnellstart*-Funktion (Quickrun). Dazu klicken Sie auf das Icon mit dem doppelten grünen Pfeil; Sie können die Funktion auch über das Menü „Erzeugen“ erreichen. Der Quelltext wird beim Schnellstart temporär gespeichert, compiliert und ausgeführt. Es öffnet sich ein Konsolenfenster mit der Ausgabe:

Ausgabe

```
Hallo Welt!
```

Nach einem Tastendruck schließt sich das Fenster wieder und das Programm ist beendet.

Nach dem Programmende werden die temporären Dateien automatisch gelöscht. Schnellstart bietet sich also an, wenn Sie eben schnell die Funktionstüchtigkeit Ihres Programmes testen wollen. Um das Programm „dauerhaft“ zur Verfügung zu haben, speichern Sie es mit der Dateiendung `.bas` (z. B. unter dem Namen `halloWelt.bas`) und compilieren es anschließend. In **??** erhalten Sie eine Einführung in wxFBE, in der die verschiedenen Möglichkeiten der IDE kurz vorgestellt werden.

2.3. Problembehebung

Sollte es beim Compilieren zu unerwarteten Fehlern kommen, dann finden Sie hier einen kurzen Überblick über mögliche Fehlerquellen. Eine ausführlichere Zusammenstellung zur Problembehandlung finden Sie im Anhang in **??**.

2.3.1. Probleme beim Start

Erhalten Sie beim Start von wxFBE unter Windows eine Fehlermeldung wie

```
This application failed to initialize properly
```

bedeutet dies, dass die benötigte Visual C++ Runtime nicht installiert ist. Beachten Sie dazu die Hinweise auf der Projektseite von wxFBE.

Sollte wxFBE unter Linux nicht starten, so empfiehlt es sich, das Programm aus der Konsole aus aufzurufen, da Sie dann die möglicherweise aufgetretenen Fehler angezeigt bekommen. Prüfen Sie, ob alle notwendigen Bibliotheken installiert sind. Wenn die Bibliothek `libwx-c-0-9-0-2.so` nicht geladen werden kann, liegt dies wahrscheinlich daran, dass Sie sich im falschen Arbeitsverzeichnis befinden.

2.3.2. Probleme beim Compilieren

Nach Eingabe des Quelltext 2.1 und einem Klick auf Schnellstart sollte im Meldungsfenster unterhalb des Quelltextes etwa folgendes zu lesen sein:

```
"C:\Programme\FreeBASIC\fbc.exe" "C:\Dokumente und
Einstellungen\meinName\FBTEMP.bas"


Compilation ausgeführt
```

wobei sich die Ausgabe je nach Betriebssystem leicht unterscheiden wird. Außerdem öffnet sich die Konsole mit der Meldung `Hallo Welt!` In diesem Fall ist alles in Ordnung und Ihr Programm wurde ordnungsgemäß compiliert und ausgeführt. Erscheint jedoch keine Konsole, dann lässt sich im Meldungsfenster ablesen, welcher Fehler aufgetreten ist. Erscheint die Meldung:

```
"C:\Programme\FreeBASIC\fbc.exe" "C:\Dokumente und
Einstellungen\meinName\FBTEMP.bas"


Build-Fehler: Datei nicht gefunden.


Compilation ausgeführt
```

bedeutet dies, dass der Compiler nicht am angegebenen Ort gefunden wurde. Die Meldung ist etwas irreführend, da in diesem Fall das Programm *nicht* compiliert werden konnte. Entweder ist bei der Installation etwas schiefgelaufen (bzw. der Compiler wurde noch gar nicht installiert) oder er befindet sich in einem anderen Verzeichnis. Den richtigen Pfad können Sie im Menü „Optionen“ – „Editor-Optionen“ einstellen. Wählen Sie dazu den Reiter „Pfad“ und geben Sie den korrekten Pfadnamen (incl. Dateinamen) des Editors an.

Da wxFBE zum Aufruf des Compilers und des compilierten Programmes auf die Windows Eingabeaufforderung zugreift, kommt es zu Problemen bei der Verwendung von Netzwerkpfaden. Wenn Sie den Ordner, auf dem wxFBE liegt, nur über einen Netzwerkpfad ansprechen können, ist es möglicherweise empfehlenswert, den Ordner aus der Komplettinstallation auf einem USB-Stick zu speichern und von dort aus auf das Programm zuzugreifen.

Falls eine andere Fehlermeldung auftaucht, stellen Sie sicher, dass Sie das Programm korrekt abgetippt haben.

Hinweis:
Wenn Sie unter Linux zwar fehlerlos compilieren können, aber sich dennoch keine Ausgabe öffnet, prüfen Sie bitte, ob in den Editor-Optionen ein korrekter Pfad für die Konsole eingetragen ist.

2.4. Wie wird aus meinem Quelltext ein Programm?

Auch wenn Sie das wxFBE-Komplettpaket installieren, sollten Sie sich immer über den Unterschied zwischen Editor (bzw. IDE) und Compiler im Klaren sein. Das Programmieren läuft in folgenden Schritten ab:

- Mit dem Editor schreiben Sie Ihr Programm und speichern es mit der Dateiendung *.bas* ab. Diese für Menschen lesbare Form von Programmen wird *Quelltext* oder

auch *Quellcode* genannt; die Datei heißt dementsprechend *Quelltextdatei*.

- Der Quelltext muss nun in ein für den Computer lesbares Format, den sogenannten *Maschinencode* übersetzt werden. Dazu wird er zuerst vom *Compiler* in Assemblercode und dieser dann vom *Assembler* in Maschinencode übersetzt.
- Mit dem *Linker* werden die compilierten Objektdateien zu einem ausführbaren Programm zusammengefügt. Unter Windows besitzt dieses Programm die Dateiendung *.exe*. Unter Linux haben ausführbare Dateien in der Regel keine Endung; man spricht hier jedoch häufig von einer *ELF-Datei* (*Executable and Linkable Format*). Der Linker wird vom Compiler automatisch aufgerufen. Sie müssen sich also vorerst keine Gedanken über den Linker machen.

Mit wxFBE können Sie nicht nur den Quelltext erstellen, sondern auch direkt den Compiler aufrufen und anschließend das Programm starten. Allerdings wird wxFBE nicht für die Ausführung des Programmes benötigt – die erstellte EXE (bzw. ELF) ist für sich allein lauffähig. wxFBE erleichtert Ihnen lediglich den Umgang mit dem Compiler und dem compilierten Programm. Damit können Sie den kompletten Weg vom Quelltext über die Compilierung bis hin zum Programmtest über wxFBE steuern.

Bei einem Schnellstart passieren im Prinzip der Reihe nach die gleichen Schritte, jedoch übernimmt hier wxFBE zusätzlich die (temporäre) Speicherung des Quelltextes und löscht das erstellte Programm nach Beendigung.

Teil II.

Grundkonzepte

3. Aufbau eines FreeBASIC-Programmes

In den folgenden Kapiteln lernen Sie die grundlegenden Elemente der Sprache FreeBASIC kennen. Wenn Sie bereits Programmiererfahrung besitzen, brauchen Sie vermutlich das ein oder andere Kapitel nur zu überfliegen. Ansonsten sollten Sie besonders die nächsten Kapitel gründlich lesen, da sie die Grundlage der weiteren Programmierung bilden.

Zunächst lernen Sie den syntaktischen Aufbau eines FreeBASIC-Programmes kennen. Sollten Sie im Augenblick noch nicht jede Einzelheit dieses Kapitels verstehen, brauchen Sie deshalb nicht zu verzagen – fahren Sie anschließend mit Kapitel 4 fort, wo Sie die ersten Programme kennen lernen. Wenn Sie dann im Laufe des Buches ein Gespür dafür entwickelt haben, wie FreeBASIC funktioniert, können Sie noch einmal hierher zurückkehren, um die Feinheiten des Aufbaus zu vertiefen.

3.1. Grundaufbau und Ablauf

FreeBASIC ist eine imperative Sprache. Der Programmablauf wird durch eine Reihe aufeinander folgender Anweisungen beschrieben. Das bedeutet, dass das Programm von oben nach unten Befehl für Befehl abgearbeitet wird. Für gewöhnlich steht in jeder Zeile eine neue Anweisung.

Der reguläre Programmablauf kann durch Bedingungen, Schleifen und Verzweigungen unterbrochen werden. Bei einer Bedingung werden bestimmte Anweisungen nur unter gewissen Voraussetzungen ausgeführt. Eine Schleife erlaubt es, eine Gruppe von Anweisungen mehrmals hintereinander durchzuführen. Verzweigungen schließlich springen zu einem anderen Abschnitt des Programmes, führen den dortigen Programmcode aus und springen anschließend wieder zurück. Mehr zum Thema Bedingungen, Schleifen und Verzweigungen erfahren Sie in Kapitel 10, Kapitel 11 und Kapitel 12.

Wie in BASIC-Dialekten üblich, ignoriert FreeBASIC die Groß- und Kleinschreibung von Befehlen. Ob Sie den Befehl zur Abfrage des Tastaturpuffers nun **`GETKEY`** oder **`getkey`** oder auch **`gETkeY`** schreiben, ist völlig egal, abgesehen davon, dass die letzte Schreibweise sehr schwer lesbar und daher nicht empfehlenswert ist. Neben der klassischen kompletten Großschreibung bzw. der kompletten Kleinschreibung hat sich auch ein gemischter Stil durchgesetzt, in dem nur der erste Buchstabe des Wortes (**`Getkey`**) oder

auch der erste Buchstabe jedes Wortbestandteils (**GetKey**) groß geschrieben wird. Welchen Stil Sie wählen, müssen Sie selbst entscheiden; es wird jedoch wärmstens empfohlen, den gewählten Stil für das ganze Programm durchzuhalten.

Mehrfache aufeinander folgende Leer- oder Tabulatorzeichen werden wie ein einziges Leerzeichen behandelt. Sie können dieses Verhalten dazu nutzen, im Quelltext sinnvolle Einrückungen vorzunehmen, um ihn lesbarer zu gestalten.

3.2. Kommentare

Programme sollen nicht nur für den Computer verständlich sein, sondern auch für diejenigen, die sich später den Quelltext ansehen wollen. Selbst wenn der Quelltext nicht für fremde Augen bestimmt ist, muss zumindest der Programmierer sein eigenes Programm später wieder verstehen können. Bei einem kurzen Programm ist das meist noch sehr einfach. Bei umfangreicheren Programmen können Kommentare helfen: diese werden vom Compiler ignoriert und dienen daher lediglich dem Programmierer als Hilfe. Sie können in einem Kommentar jeden beliebigen Text einbauen, von Erklärungen zum Programmcode bis hin zu Lizenzbestimmungen – den Programmablauf wird das nicht beeinflussen.

Früher diente der Befehl **REM** als Einleitung einer Kommentarzeile. **REM** wird unter FreeBASIC weiterhin unterstützt, jedoch gibt es mit dem Hochkomma **'** eine wesentlich bequemere Alternative (ASCII-Code 39; nicht zu verwechseln mit den beiden Apostroph-Zeichen).

```
REM Eine Kommentarzeile, die (einschliesslich des Befehls REM) ignoriert wird
' dasselbe mit Hochkomma: die Zeile wird (einschliesslich Hochkomma) ignoriert
PRINT "Hallo Welt" ' Ein Kommentar kann auch nach einer Anweisung folgen
```

Soll ein mehrzeiliger Kommentar verfasst werden – etwa um einleitende Informationen zum Programm oder zu den Lizenzbedingungen zu geben – dann wird dieser mit **/'** eingeleitet und mit **'/** beendet.

```
PRINT "Diese Zeile wird normal ausgegeben."
/' Hier beginnt der Kommentarblock
   Alles innerhalb des Blocks wird ignoriert.
   Nach dem Abschluss geht es normal weiter. '/
PRINT "Diese Zeile wird wieder ausgegeben."
```

Achten Sie bei der Verwendung von Kommentaren darauf, die möglicherweise unklaren Abschnitte zu erläutern. Die Erklärung offensichtlicher Dinge ist dagegen wenig sinnvoll. Mag für einen kompletten Anfänger der Kommentar

```
PRINT "Hallo Welt"   ' gibt den Text 'Hallo Welt' aus
```

noch hilfreich sein, ist er für jeden, der sich auch nur ein wenig mit Programmierung auskennt, selbstverständlich und wird viel eher als Gängelung angesehen. Durch eine Flut solcher Kommentare wird das Programm eher undurchsichtiger als verständlicher, da wirklich wichtige Informationen untergehen.
Hinweise zu einem sinnvollen Umgang mit Kommentaren finden Sie in **??**.

3.3. Zeilenfortsetzungszeichen

In der Regel entspricht jede Programmzeile einer Anweisung. Das Zeilenende legt damit auch das Ende der Anweisung fest. Dies kann bei langen Anweisungen hinderlich sein. Überlange Zeilen erschweren die Lesbarkeit des Codes; bei den Codebeispielen in diesem Buch wären sie sogar gar nicht sinnvoll abdruckbar. Zum Glück stellt FreeBASIC eine Möglichkeit bereit, überlange Zeilen auf mehrere Zeilen aufzuteilen.

Wenn der Compiler auf einen allein stehenden Unterstrich _ stößt (also auf einen Unterstrich, der nicht Teil eines Befehls, einer Variablen oder eines Strings ist), dann wird die folgende Zeile an die aktuelle angefügt – beide Zeilen werden also so behandelt, als ob sie eine einzige wären. Selbstverständlich können auf diese Art auch mehrere Zeilen zusammengefügt werden.

Der Unterstrich selbst und alles, was in der alten Zeile auf ihn folgt, wird vom Compiler ignoriert. Dies erlaubt Ihnen beispielsweise, am Ende der Zeile einen Kommentar unterzubringen, etwa um in der Definition einer Funktion (vgl. Kapitel 12) die vorkommenden Parameter zu erklären, wie im folgenden Beispiel:

```
FUNCTION eingabe( BYVAL s AS STRING, _          ' ausgegebene Meldung
                  BYVAL Sys AS STRING = "*", _  ' Typ der Eingabe
                  BYVAL Upper AS INTEGER = 1, _ ' Umwandlung in Grossbuchstaben
                  BYVAL pw AS STRING = "", _    ' Zeichen zur Passwortmaskierung
                  BYVAL AddLf AS INTEGER = 1, _ ' Flag zum Hinzufuegen eines LF
                  BYVAL Edit AS STRING = "" _   ' zu editierender String
                ) AS STRING
```

Abgesehen davon, dass die Funktion nicht in einer Zeile hätte abgedruckt werden können, ist auch eine Kommentierung ohne dem Zeilenfortsetzungszeichen schwer möglich.

3.4. Trennung von Befehlen mit Doppelpunkt

Andererseits ist es auch möglich, mehrere kurze Anweisungen in eine Zeile zu packen. Die Anweisungen werden dazu durch einen Doppelpunkt **:** getrennt.

```
PRINT "Hallo!" : PRINT "Willkommen im Programm."
```

Der Quelltext wird vom Compiler so behandelt, als ob statt des Doppelpunktes ein Zeilenumbruch stehen würde. Sie sollten sich jedoch darüber im Klaren sein, dass die Verwendung des Doppelpunktes Ihr Programm in den meisten Fällen schwerer lesbar macht. In aller Regel sollten Sie nur eine Anweisung pro Zeile verwenden.

4. Bildschirmausgabe

Nach diesen doch recht theoretischen Vorüberlegungen wollen wir nun aber mit den ersten Programmen beginnen. Eine der wichtigsten Aufgaben von Computerprogrammen ist die Verarbeitung von Daten: Das Programm liest Daten ein, verarbeitet sie und gibt das Ergebnis aus. Zunächst werden wir uns die Bildschirmausgabe ansehen.

4.1. Der `PRINT`-Befehl

Der Befehl **`PRINT`** wurde in der Einführung bereits vorgestellt. Mit ihm ist es möglich, einen Text, auch Zeichenkette oder String genannt, auf dem Bildschirm auszugeben:

```
PRINT "Hallo FreeBASIC-Welt!"
SLEEP
```

Der abschließende Befehl **`SLEEP`** dient übrigens dazu, das Programm so lange geöffnet zu halten, bis eine Taste gedrückt wurde. Nach Ausführung des letzten Befehls ist das Programm beendet; das Programmfenster schließt sich damit wieder. Ohne dem **`SLEEP`** würde zwar der Text angezeigt, danach aber das Programm sofort geschlossen werden, sodass der Benutzer keine Möglichkeit mehr hat, den Text zu lesen. **`SLEEP`** verhindert das sofortige Beenden und erlaubt es dem Benutzer, die Ausgabe anzusehen.

Der Text, der ausgegeben werden soll, wird in "doppelte Anführungszeichen" gesetzt (das Zeichen mit dem ASCII-Code 34; auf den meisten westeuropäischen Tastaturen über `[Shift]+[2]` erreichbar). Wenn der Text selbst ein doppeltes Anführungszeichen enthalten sollen, müssen Sie an die entsprechende Stelle einfach nur zwei dieser Zeichen hintereinander setzen – der Compiler weiß dann, dass die Zeichenkette nicht endet, sondern ein Anführungszeichen geschrieben werden soll.

```
PRINT "Wilkommen bei meinem ""tollen"" Programm!"
SLEEP
```

Ausgabe

```
Wilkommen bei meinem "tollen" Programm!
```

Im Gegensatz dazu werden Zahlen ohne Anführungszeichen geschrieben. FreeBASIC kann so auch als Taschenrechner „missbraucht“ werden.

```
PRINT 2
PRINT 5 + 3 * 7
SLEEP
```

Ausgabe

```
2
26
```

Wie Sie sehen, hält sich das Programm an die Rechenregel „Punkt vor Strich“. Vielleicht sollte noch kurz erwähnt werden, dass sich in der Programmierung der Stern ***** als Malzeichen eingebürgert hat, während als Divisionszeichen der Schrägstrich **/** verwendet wird. Potenzen werden mit dem Zeichen **^** geschrieben. Außerdem können auch Klammern eingesetzt werden, jedoch sind für Berechnungen nur runde Klammern erlaubt. In Quelltext 4.1l werden zur Veranschaulichung einige Berechnungen durchgeführt.

Normalerweise wird die Ausgabe nach jedem **PRINT** in einer neuen Zeile fortgesetzt. Um mehrere Ausgaben (z. B. ein Text und eine Rechnung) direkt nacheinander zu erhalten, dient der Strichpunkt **;** als Trennzeichen. Steht der Strichpunkt ganz am Ende der Anweisung, so findet die nächste **PRINT**-Ausgabe in derselben Zeile statt.

Auch das Komma **,** kann als Trennzeichen eingesetzt werden. Dabei wird jedoch ein Tabulator gesetzt; die Ausgabe setzt nicht direkt hinter der letzten Ausgabe fort, sondern an der nächsten Tabulatorposition.

Quelltext 4.1: PRINT-Ausgabe

```
PRINT "Addition:       5 + 2 ="; 5+2      ' zwei Ausgaben hintereinander
PRINT "Subtraktion:    5 - 2 ="; 5-2
PRINT "Multiplikation: 5 * 2 =";          ' die naechste Ausgabe findet in
PRINT 5*2                                 ' derselben Zeile statt
PRINT "Division:       5 / 2 ="; 5/2
PRINT "Exponent:       5 ^ 2 ="; 5^2

PRINT                                     ' Leerzeile ausgeben

PRINT "gemischt:", "4 + 3*2 ="; 4 + 3*2   ' Komma zum Tabulator-Einsatz
PRINT , "(4 + 3)*2 ="; (4 + 3)*2          ' geht auch zu Beginn der Ausgabe
SLEEP
```

Ausgabe

```
Addition:       5 + 2 = 7
Subtraktion:    5 - 2 = 3
Multiplikation: 5 * 2 = 10
Division:       5 / 2 = 2.5
Exponent:       5 ^ 2 = 25

gemischt:     4 + 3*2 = 10
              (4 + 3)*2 = 14
```

Noch einmal zusammengefasst: Zeichenketten bzw. Strings müssen in Anführungszeichen gesetzt werden, Berechnungen stehen ohne Anführungszeichen. Nach der Ausgabe wird in der folgenden Zeile fortgesetzt, außer der Zeilenumbruch wird durch einen Strichpunkt oder ein Komma unterdrückt. Dabei dient der Strichpunkt dazu, die Ausgabe an der aktuellen Stelle fortzusetzen. Das Komma setzt bei der nächsten Tabulatorstelle fort.

Achtung:
An der Ausgabe der Divisionsaufgabe können Sie erkennen, dass als Dezimaltrennzeichen kein *Komma* verwendet wird, sondern ein *Punkt*. Wenn Sie selbst Dezimalzahlen einsetzen wollen, müssen Sie ebenfalls den Punkt als Trennzeichen verwenden.

4.2. Ausgabe von Variablen

4.2.1. Deklaration von Variablen

Um sich innerhalb des Programmes Werte merken zu können, werden Variablen verwendet. Variablen sind Speicherbereiche, in denen alle möglichen Arten von Werten abgelegt werden können. Man kann sie sich gewissermaßen wie „Schubladen“ vorstellen, in die der gewünschte Wert hineingelegt und wieder herausgeholt werden kann. Dabei kann jede „Schublade“ nur eine bestimmte Art von Werten speichern. Es gibt beispielsweise **`INTEGER`**-Variablen, in denen Ganzzahlen (also ohne Nachkommastellen) abgelegt werden können, während **`STRING`**-Variablen Zeichenketten speichern. Jeder Variablentyp kann nur die zugehörige Art an Daten speichern – in einer Zahlenvariablen kann keine Zeichenkette abgelegt werden und in einer **`STRING`**-Variablen keine Zahl (wohl aber eine Zeichenkette,

die eine Zahl repräsentiert). Welche Variablentypen zur Verfügung stehen und wo die genauen Unterschiede liegen, wird in Kapitel 6 erklärt.

Bevor eine Variable verwendet werden kann, muss dem Programm zunächst bekannt gemacht werden, dass die gewünschte Variable existiert. Man spricht dabei vom *Deklarieren* der Variable. Üblicherweise wird dazu der Befehl **DIM** verwendet.

```
DIM variable1 AS INTEGER
DIM AS INTEGER variable2
```

Der oben stehende Code deklariert zwei INTEGER-Variablen mit den Namen `variable1` bzw. `variable2`. Beide genannten Schreibweisen sind möglich; welche Sie verwenden, hängt zum einen vom persönlichen Geschmack ab, hat zum anderen aber auch praktische Gründe: Sollen mehrere Variablen deklariert werden, dann kann das oft mit einem einzigen Befehl erledigt werden.

```
DIM variable1 AS INTEGER, variable2 AS SHORT, variable3 AS STRING
```

deklariert drei Variablen unterschiedlichen Typs auf einmal. Wenn alle drei Variablen demselben Datentyp angehören, bietet sich hier die zweite Schreibweise an:

```
DIM AS INTEGER variable1, variable2, variable3
```

Dies ist wesentlich kürzer, setzt jedoch voraus, dass der Datentyp aller drei Variablen gleich ist. Wollen Sie drei Variablen unterschiedlichen Typs gleichzeitig deklarieren, dann müssen Sie auf die erste Schreibweise zurückgreifen. Die Bedeutung der einzelnen Datentypen folgt in den nächsten Kapiteln.

4.2.2. Wertzuweisung

Wertzuweisungen erfolgen über das Gleichheitszeichen =

```
DIM AS INTEGER variable
variable = 3
```

`variable` bekommt hier den Wert 3 zugewiesen – zu einem späteren Zeitpunkt des Programmes kann dieser Wert wieder ausgelesen und verwendet werden. Da man einer neuen Variablen sehr oft gleich zu Beginn einen Wert zuweisen muss, gibt es eine Kurzschreibweise für Deklaration und Zuweisung:

```
DIM AS INTEGER variable1 = 3, variable2 = 10
DIM variable3 AS INTEGER = 8, variable4 AS INTEGER = 7
```

In den beiden Zeilen werden je zwei **INTEGER**-Variablen deklariert und sofort mit einem Wert belegt. Beachten Sie bitte die Schreibweise: die Zuweisung steht immer am

Ende, also in der zweiten Zeile erst nach dem Variablentyp.

Bevor die verschiedenen Datentypen besprochen werden, muss noch geklärt werden, welche Variablennamen überhaupt verwendet werden dürfen. Ein Variablenname darf nur aus Buchstaben (a-z, ohne Sonderzeichen wie Umlaute oder ß), den Ziffern 0-9 und dem Unterstrich _ bestehen. Allerdings darf das erste Zeichen keine Ziffer sein. Außerdem kann nur ein Name benutzt werden, der noch nicht in Verwendung ist. Beispielsweise können keine FreeBASIC-Befehle als Variablennamen verwendet werden, aber selbstverständlich können auch nicht zwei verschiedene Variablen denselben Namen besitzen.

Obwohl als Variablenname eine Bezeichnung wie `gzyq8r` zulässig ist, werden Sie später, wenn Sie auf diesen Namen stoßen, wohl keine Ahnung mehr haben, was die Variable für eine Bedeutung hat. Sie sollten daher lieber Namen wählen, mit denen Sie auf den Inhalt der Variable schließen können. Wollen Sie z. B. das Alter des Benutzers speichern, bietet sich der Variablenname `alter` weitaus mehr an als der Name `gzyq8r`. Die Verwendung von „sprechenden Namen“ ist eines der obersten Gebote des guten Programmierstils!

Einer Variablen kann auch der Wert einer anderen Variablen zugewiesen werden, oder auch ein Wert, der mithilfe anderer Variablen berechnet wird.

```
DIM AS INTEGER var1 = 3, var2 = 10                ' Deklaration
DIM AS INTEGER mittelwert = (var1 + var2) / 2     ' Deklaration mit Wertzuweisung
var1 = var1 + 1                                   ' Wert um 1 erhoehen
```

In der letzten Zeile sehen Sie sehr schön, dass es sich um eine *Zuweisung* und nicht um eine mathematische *Gleichung* handelt. Mathematisch gesehen ist `var1=var1+1` keine sinnvolle Aussage. Bei der Zuweisung dagegen berechnet das Programm den Wert der rechten Seite und weist das Ergebnis der Variablen auf der linken Seite zu.

(Nachdem ich jetzt eindrücklich die Verwendung sprechender Namen angemahnt habe, kommt gleich im Anschluss ein Beispiel mit so nichtssagenden Namen wie `var1` und `var2` ... Allerdings haben diese Variablen in den kurzen Codeabschnitten tatsächlich keine besondere Bedeutung, weshalb ich die aus Platzgründen möglichst kurz gewählten Namen zu entschuldigen bitte.)

Hinweis:
Wie Sie nun wissen, können Sie einer Variablen direkt bei der Deklaration einen Wert zuweisen. Wenn Sie dies nicht tun, wird die Variable mit einem Standardwert belegt (bei Zahlen ist das der Wert `0`). Es gibt aber auch noch eine dritte Möglichkeit: Die Befehlszeile
`DIM AS INTEGER zahl = ANY`
deklariert eine **`INTEGER`**-Variable, *ohne* ihr einen Wert zuzuweisen – das bedeutet, die Variable behält den Wert, der sich zuvor bereits an der verwendeten Speicherstelle befunden hat. Dieser Wert ist, praktisch gesehen, im Großen und Ganzen zufällig, da Sie ja nicht wissen können, welcher Speicherplatz für Ihre neue Variable verwendet wird.

4.2.3. Wertvertauschung

Um den Wert zweier Variablen (z. B. `var1` und `var2`) zu vertauschen, kann nun nicht einfach der zweiten Variablen der Wert der ersten zugewiesen werden und umgekehrt – mit der ersten Zuweisung geht nämlich der ursprüngliche Wert der zweiten Variablen verloren. Zur Veranschaulichung sehen Sie in den folgenden Codeschnipsel im Kommentar jeweils die Wertbelegungen, welche die Variablen nach Ausführung der Zeile besitzen. Außer der Wertzuweisungen machen diese „Prográmmchen“ allerdings nichts, weshalb Sie beim Ausführen des Codes nichts sehen würden.

```
' das funktioniert nicht ...
DIM AS INTEGER var1 = 3, var2 = 10              ' var1 = 3; var2 = 10
var2 = var1                                    ' var1 = 3; var2 = 3
var1 = var2                                    ' var1 = 3; var2 = 3
```

Stattdessen muss der Wert der ersten Variable zwischengespeichert werden, damit er anschließend weiter zur Verfügung steht. Das lässt sich mit einer zusätzlichen temporären Variablen erreichen.

```
' das funktioniert dagegen:
DIM AS INTEGER var1 = 3, var2 = 10, temp        ' var1 = 3;  var2 = 10; temp = 0
temp = var2                                    ' var1 = 3;  var2 = 10; temp = 10
var2 = var1                                    ' var1 = 3;  var2 = 3;  temp = 10
var1 = temp                                    ' var1 = 10; var2 = 3;  temp = 10
```

Wie Sie sehen, wurden die Werte von `var1` und `var2` jetzt korrekt vertauscht. Allerdings ist die Vorgehensweise recht aufwändig. FreeBASIC stellt daher einen Befehl zur Verfügung, der die Angelegenheit deutlich schneller regelt:

```
' Werte vertauschen mit SWAP
DIM AS INTEGER var1 = 3, var2 = 10              ' var1 = 3;  var2 = 10
SWAP var1, var2                                 ' var1 = 10; var2 = 3
```

4.2.4. Ausgabe des Variablenwertes

Um nun den Wert einer Variablen auszugeben, wird sie einfach in eine **PRINT**-Anweisung eingefügt. Sie kann auch für Berechnungen verwendet werden.

Quelltext 4.2: Ausgabe von Variablen

```
DIM vorname AS STRING, alter AS INTEGER
vorname = "Otto"
alter = 17
PRINT vorname; " ist"; alter; " Jahre alt."
PRINT "In drei Jahren wird "; vorname; alter+3; " Jahre alt sein."
SLEEP
```

Ausgabe

```
Otto ist 17 Jahre alt.
In drei Jahren wird Otto 20 Jahre alt sein.
```

In den Zeilen 4 und 5 wird die Ausgabe jeweils aus mehreren Teilen zusammengesetzt. Die Zeichenketten, welche exakt wie angegeben auf dem Bildschirm erscheinen sollen, sind wieder in Anführungszeichen gesetzt. Die Variablen stehen genauso wie die Zahlenwerte ohne Anführungszeichen.

Der aufmerksame Leser wird bemerkt haben, dass bei der Ausgabe vor den Zahlen – und ebenso vor dem Wert einer Zahlenvariablen – ein Leerzeichen eingefügt wird. Genauer gesagt handelt es sich dabei um den Platzhalter für das Vorzeichen der Zahl. Ein positives Vorzeichen wird nicht angezeigt und bleibt daher leer. Bei negativen Zahlen dagegen wird dieser Platz durch das Minuszeichen belegt. Wenn Sie in Quelltext 4.2 für `alter` den Wert `-17` eingeben, werden Sie sehen, dass die zusätzlichen Leerzeichen wegfallen.

4.3. Formatierung der Ausgabe

Das Komma im **PRINT**-Befehl ist zur Formatierung der Ausgabe ganz nett, aber nicht sehr flexibel. Um die Ausgabe an einer bestimmten Stelle fortzusetzen, gibt es mehrere Möglichkeiten.

4.3.1. Direkte Positionierung mit LOCATE

Mit dem Befehl **LOCATE** wird eine Zeile und eine Spalte angegeben, an die der Cursor gesetzt wird. Die nächste Textausgabe beginnt an dieser Position. Die Nummerierung der Zeilen und Spalten beginnt mit 1 – die Anweisung `LOCATE 1, 1` setzt den Cursor also in das linke obere Eck. Die beiden durch Komma getrennten Zahlen werden Parameter genannt.

Quelltext 4.3: Positionierung mit LOCATE

```
LOCATE 1, 20              ' erste Zeile, zwanzigste Spalte
PRINT "Der LOCATE-Befehl"
LOCATE 2, 20              ' zweite Zeile, zwanzigste Spalte
PRINT "=================="
LOCATE 4, 1               ' Beginn der vierten Zeile
PRINT "Mit dem Befehl LOCATE wird der Cursor positioniert."
SLEEP
```

Ausgabe

```
                    Der LOCATE-Befehl
                    ==================

 Mit dem Befehl LOCATE wird der Cursor positioniert.
```

Es ist auch möglich, nur einen der beiden Parameter anzugeben. Mit `LOCATE 5` beispielsweise wird der Cursor in die fünfte Zeile gesetzt, während die aktuelle Spalte nicht verändert wird. Wenn umgekehrt nur die Spaltenposition geändert werden soll, fällt der erste Parameter weg, das Komma muss jedoch stehen bleiben. Ansonsten könnte der Compiler Zeilen- und Spaltenparameter ja nicht auseinanderhalten. Das oben verwendete Beispiel könnte damit auch folgendermaßen aussehen, ohne dass sich die Ausgabe ändert:

```
LOCATE 1, 20              ' erste Zeile, zwanzigste Spalte
PRINT "Der LOCATE-Befehl"
' Nun befindet sich der Cursor am Anfang der zweiten Zeile
LOCATE , 20               ' in die zwanzigste Spalte setzen
PRINT "=================="
' Nun befindet sich der Cursor am Anfang der dritten Zeile
LOCATE 4                  ' in die vierte Zeile setzen
PRINT "Mit dem Befehl LOCATE wird der Cursor positioniert."
SLEEP
```

4.3.2. Positionierung mit TAB und SPC

Im Gegensatz zu **LOCATE** werden die Befehle **TAB** und **SPC** innerhalb der **PRINT**-Anweisung gesetzt:

```
PRINT "Spalte 1"; TAB(20); "Spalte 20"
```

Die Befehle bewirken Ähnliches wie ein im **PRINT** eingebautes Komma, nämlich das Vorrücken zu einer bestimmten Spaltenposition, nur dass diese Position frei gewählt werden kann. Beachten Sie auch, dass der Parameter in diesem Fall in Klammern geschrieben werden muss.

TAB rückt den Cursor an die angegebene Stelle vor. Es bestehen also Ähnlichkeiten zu **LOCATE** mit ausgelassenem ersten Parameter. Beachten Sie jedoch, dass tatsächlich *vorgerückt* wird. Notfalls wechselt der Cursor in die nächste Zeile.

```
PRINT "Zeile 1, Spalte 1"; TAB(5); "Zeile 2, Spalte 5"
```

Da sich der Cursor nach der ersten Ausgabe bereits in Spalte 18 befindet, muss er, um die Spalte 5 zu erreichen, in die zweite Zeile vorrücken.

SPC rückt den Cursor um die angegebene Zeichenzahl weiter. Der Unterschied zu **TAB** besteht also darin, dass `TAB(20)` den Cursor genau in die Spalte 20 setzt, während `SPC(20)` den Cursor *um 20 Zeichen weiter* rückt (also z. B. von Spalte 15 nach Spalte 35).

```
PRINT "Spalte 1"; TAB(20); "Spalte 20"
PRINT "Spalte 1"; SPC(20); "Spalte 29"
```

Bei Bedarf wird in die nächste Zeile gewechselt. `SPC(400)` etwa wird mehrere Zeilen weiter rücken – bei einer Konsolenbreite von 80 Zeichen werden es fünf Zeilen sein, um die der Cursor nach unten rutscht.

Unterschiede zu QuickBASIC:
Im Gegensatz zu QuickBASIC werden in FreeBASIC die übersprungenen Zeichen nicht mit Leerzeichen überschrieben. Der bereits vorhandene Text bleibt erhalten.

4.3.3. Cursor-Position ermitteln: POS und CSRLIN

Um umgekehrt herauszufinden, wo sich der Textcursor gerade befindet, helfen die beiden Befehle **POS** (*POSition*) und **CSRLIN** (*CurSoR LINe*). **POS** gibt die Spalte zurück, in

der sich der Cursor im Augenblick befindet, und **CSRLIN** die aktuelle Zeile.

Es sollte nicht überraschen, dass eine Abfrage der Cursor-Position innerhalb einer **PRINT**-Anweisung das Ergebnis beeinflussen kann. Im folgenden Quelltext wird das anhand zweier **PRINT**-Anweisungen demonstriert. In der ersten Anweisung in Zeile 1 wird **POS** erst abgefragt, nachdem der vorherige Text bereits ausgegeben wurde und sich der Cursor in der 44. Spalte befindet. Dagegen wird die Cursorposition für die zweite Ausgabe zuerst gespeichert und dann dieser gespeicherte Wert ausgegeben.

```
PRINT "Der Cursor befindet sich in Zeile"; CSRLIN; ", Spalte"; POS
DIM AS INTEGER zeile = CSRLIN, spalte = POS
PRINT "Der Cursor befindet sich in Zeile"; zeile; ", Spalte"; spalte
SLEEP
```

Ausgabe

```
Der Cursor befindet sich in Zeile 1, Spalte 44
Der Cursor befindet sich in Zeile 2, Spalte 1
```

4.3.4. Farbige Ausgabe

Nun wollen wir etwas Farbe ins Spiel bringen. Im Konsolenfenster stehen uns 16 Farben zur Verfügung. Wir können jeweils die Vorder- und Hintergrundfarbe des Textes auf eine dieser Farben setzen. Dazu dient der Befehl **COLOR**, dem zwei Parameter mitgegeben werden können, nämlich der Farbwert der Vordergrundfarbe und der Farbwert der Hintergrundfarbe.

```
COLOR 1, 4
```

setzt die Vordergrundfarbe auf den Wert 1 und die Hintergrundfarbe auf den Wert 4. Die Farbwerte nützen selbstverständlich nur etwas, wenn man weiß, welche Farben sich dahinter verbergen. Es handelt sich dabei um die CGA-Farbwerte (siehe Tabelle 4.1). Das obige Beispiel würde also beim nächsten **PRINT** einen blauen Text auf rotem Hintergrund ausgeben.

Farbnr.	Farbe	Farbnr.	Farbe
0	Schwarz	8	Grau
1	Blau	9	Hellblau
2	Grün	10	Hellgrün
3	Cyan	11	Hell-Cyan
4	Rot	12	Hellrot
5	Magenta	13	Hell-Magenta
6	Braun	14	Gelb
7	Hellgrau	15	Weiß

Tabelle 4.1.: Farbwerte (Konsole)

Quelltext 4.4: Farbige Textausgabe

```
COLOR 1, 4
PRINT "Dieser Text ist in blauer Schrift auf rotem Hintergrund."
COLOR 4, 1
PRINT "Dieser Text ist in roter Schrift auf blauem Hintergrund."
COLOR 14
PRINT "Dieser Text ist in gelber Schrift auf blauem Hintergrund."
COLOR , 0
PRINT "Dieser Text ist in gelber Schrift auf schwarzem Hintergrund."
SLEEP
```

Wie Sie sehen, können Sie auch einen der beiden Parameter auslassen – in diesem Fall wird nur der angegebene Wert geändert und der andere beibehalten. Außerdem sollte Ihnen aufgefallen sein, dass die gewählte Hintergrundfarbe nur an den Stellen angezeigt wird, an denen Text ausgegeben wurde. In der Regel wird man sich aber wünschen, dass der Hintergrund im kompletten Fenster die gleiche Farbe besitzt, also z. B. ein blaues Fenster mit gelber Schrift. Hierzu kann der Befehl **CLS** dienen.

4.3.5. Bildschirm löschen mit CLS

CLS ist eine Abkürzung von *CLear Screen* und löscht den Inhalt des Fensters. Mehr noch: das Fenster wird vollständig auf die eingestellte Hintergrundfarbe gesetzt. Dazu muss *erst* die Hintergrundfarbe über **COLOR** eingestellt und *anschließend* der Bildschirm mit **CLS** gelöscht werden.

Quelltext 4.5: Fenster-Hintergrundfarbe setzen

```
COLOR 6, 1
CLS
PRINT "Der ganze Bildschirm ist jetzt blau."
PRINT "Die Schrift wird gelb dargestellt."
SLEEP
```

Das Löschen des Fensterinhalts fällt in diesem Beispiel nicht ins Gewicht, weil das Fenster ja bisher noch nichts enthält. Das einzig Interessante ist im Augenblick das Setzen des Fensterhintergrunds.

Bitte bedenken Sie jedoch beim Einsatz von Farben, dass manche Farbkombinationen für einige Menschen sehr schwer lesbar sind! Außerdem sollte man es, wie bei allen Spezialeffekten, nicht übertreiben – oft gilt: weniger ist mehr. In der Testphase dürfen Sie sich natürlich erst einmal austoben.

4.4. Fragen zum Kapitel

Am Ende der meisten Kapitel werden Sie einige Fragen und/oder kleine Programmieraufgaben finden, mit denen Sie das bisher Gelernte überprüfen können. Die Antworten erhalten Sie im Anhang A.

1. Welche Bedeutung besitzt innerhalb einer **PRINT**-Anweisung der Strichpunkt, welche Bedeutung besitzt das Komma?
2. Was bedeutet das Komma in anderen Anweisungen?
3. Wann werden Anführungszeichen benötigt, wann nicht?
4. Welche Zeichen sind für einen Variablennamen erlaubt?
5. Was versteht man unter sprechenden Namen?
6. Was sind die Unterschiede zwischen **TAB** und **SPC**?

Und noch eine kleine Programmieraufgabe für den Schluss:
Schreiben Sie ein Programm, das den gesamten Hintergrund gelb färbt und dann in roter Schrift (ungefähr) folgende Ausgabe erzeugt:

Ausgabe

```
              Mein erstes FreeBASIC-Programm
              ==============================

Heute habe ich gelernt, wie man mit FreeBASIC Text ausgibt.
Ich kann den Text auch in verschiedenen Farben ausgeben.
```

Anschließend soll das Programm auf einen Tastendruck warten.

5. Tastatureingabe

Wenn Sie nur Daten verarbeitet könnten, die beim Schreiben des Programmes bereits feststehen, wäre das ziemlich eintönig. In der Regel erhalten die Programme Daten von außen, etwa über eine externe Datei, die Tastatur und andere Steuergeräte wie Maus oder Joystick, und passen den Programmablauf an diese Daten an. Wir werden später im Buch auf das Einlesen von Dateien und auf die Mausabfrage zu sprechen kommen. Als Eingabemedium verwenden wir zunächst nur die Tastatur.

5.1. Der `INPUT`-Befehl

`INPUT` ist eine recht einfache Art, Tastatureingaben einzulesen. Der Befehl liest so lange von der Tastatur, bis ein Zeilenumbruch (`Return`-Taste) erfolgt, und speichert die eingegebenen Zeichen in einer Variablen. Diese Variable muss dem Programm zuvor bekannt sein, also mit **`DIM`** deklariert worden sein. Im weiteren Programmablauf kann der Variablenwert dann z. B mit **`PRINT`** wieder ausgegeben werden.

```
DIM AS INTEGER alter
INPUT alter
```

Um zu kennzeichnen, dass eine Eingabe erwartet wird, gibt das Programm ein Fragezeichen an der Stelle aus, an der die Eingabe stattfinden wird. Da dies nicht immer gewünscht ist, kann der Programmierer stattdessen einen selbst gewählten Text anzeigen lassen. Dieser wird als String vor die Speichervariable gesetzt:

```
DIM AS INTEGER alter
INPUT "Gib dein Alter ein: ", alter
```

So weit, so gut – beim **`INPUT`**-Befehl steckt der Teufel jedoch im Detail: Es ist nämlich möglich, mit einem einzigen **`INPUT`**-Befehl mehrere Variablen abzufragen. Diese werden im Befehl, durch jeweils ein Komma getrennt, hintereinander angegeben. In der Eingabe dient dann ebenfalls das Komma zum Trennen der einzelnen Daten.

```
DIM AS STRING vorname, nachname
INPUT "Gib, durch Komma getrennt, deinen Vor- und deinen Nachnamen ein: ", _
      vorname, nachname
PRINT "Hallo "; vorname; " "; nachname
SLEEP
```

Zur Erinnerung: Die Zeilen 2 und 3 werden wegen des Unterstrichs wie eine einzige Zeile behandelt. Die Trennung in zwei Zeilen erfolgt hier nur aus Platzgründen.

Die Möglichkeit der Mehrfacheingabe bedeutet natürlich, dass die eingegebenen Daten selbst kein Komma enthalten dürfen, da sie ja sonst bei diesem Komma abgeschnitten würden. Dennoch hält FreeBASIC eine Möglichkeit bereit, auch Kommata einzugeben. Dazu muss der Benutzer den komplette Wert in Anführungszeichen setzen.

Des Weiteren darf es sich bei der ausgegebenen Meldung wirklich nur um einen String handeln und nicht etwa um eine Variable, die einen String enthält. Nehmen wir folgendes Beispiel:

```
DIM AS INTEGER alter
DIM AS STRING frage = "Gib dein Alter ein: "
INPUT frage, alter                          ' tut nicht das Erwuenschte
```

Das Programm denkt nun, dass zwei Werte eingelesen werden sollen, und zwar in die beiden Variablen `frage` und `alter`. Außerdem darf die Meldung nicht durch Stringkonkatenation (dazu später mehr) zusammengesetzt worden sein – es muss sich um einen einzelnen String handeln, der mit Anführungszeichen beginnt und mit Anführungszeichen endet.

INPUT hält noch eine weitere Besonderheit parat: Die Trennung zwischen der Meldung und der ersten Variablen kann statt durch ein Komma auch durch einen Strichpunkt erfolgen. In diesem Fall wird dann zusätzlich zur Meldung auch noch ein Fragezeichen ausgegeben – etwa wie in folgendem Fall:

```
DIM AS INTEGER alter
INPUT "Wie alt bist du"; alter
```

Inwieweit diese Schreibweise der Lesbarkeit des Quelltextes dient, mag der Leser selbst entscheiden.

Was tut man nun aber, wenn in der Frage der Wert einer Variablen einbunden werden soll? Nehmen wir an, Sie wollen die Frage nach dem Alter etwas persönlicher gestalten und den Benutzer mit seinem (zuvor eingegebenen) Namen ansprechen. Das lässt sich ganz leicht bewerkstelligen: Geben Sie die Frage einfach zuerst mit **PRINT** aus und hängen Sie dann mit **INPUT** die Abfrage an.

```
DIM AS STRING vorname, meinung
PRINT "Willkommen bei meinem Programm!"
INPUT "Gib zuerst deinen Vornamen ein: ", vorname
PRINT "Hallo "; vorname; ". Wie findest du FreeBASIC? ";
INPUT meinung
PRINT "Vielen Dank. Mit einem Tastendruck beendest du das Programm."
SLEEP
```

Sieht schon recht gut aus – allerdings gibt **INPUT** ein zusätzliches Fragezeichen aus, das sich hier störend auswirkt. Mit einer kleinen Änderung ist auch dieses Problem behoben:

Quelltext 5.1: Benutzereingabe mit INPUT

```
DIM AS STRING vorname, meinung
PRINT "Willkommen bei meinem Programm!"
INPUT "Gib zuerst deinen Vornamen ein: ", vorname
PRINT "Hallo "; vorname; ". Wie findest du FreeBASIC? ";
INPUT "", meinung
PRINT "Vielen Dank. Mit einem Tastendruck beendest du das Programm."
SLEEP
```

Ausgabe

```
Willkommen bei meinem Programm!
Gib zuerst deinen Vornamen ein: Rose
Hallo Rose. Wie findest du FreeBASIC? Echt cool!
Vielen Dank. Mit einem Tastendruck beendest du das Programm.
```

Um dem Benutzer zu ermöglichen, eine komplette Eingabezeile ohne Rücksicht auf mögliche Kommatas einzugeben, können Sie **LINE INPUT** verwenden. Dieser Befehl liest immer bis zum Ende der Zeile, also bis zu einem Zeilenumbruch. Dementsprechend darf für die Eingabe auch nur eine einzige Variable angegeben werden – der Benutzer kann ja auch nur einen einzigen Wert übermitteln. Diese muss jetzt aber eine **STRING**-Variable sein. Da auf der anderen Seite die Meldung und die Eingabevariable klar voneinander getrennt werden können, erlaubt **LINE INPUT** für die Meldung auch Variablen und zusammengesetzte Strings. Ansonsten ist **LINE INPUT** genauso aufgebaut wie **INPUT**.

```
DIM AS STRING benutzername, eingabe
LINE INPUT "Gib deinen Namen ein: ", benutzername
LINE INPUT "Hallo " & benutzername & ". Willst du noch etwas sagen"; eingabe
```

Ausgabe

```
Gib deinen Namen ein: Jack, König der Welt
Hallo Jack, König der Welt. Willst du noch etwas sagen? Ja, klar.
```

Dieser Quelltext verwendet bereits das Zusammensetzen mehrerer Ausdrücke zu einem String. Die dazu verwendete Stringkonkatenation wird in Kapitel 6.3.2 besprochen.

Kurz zusammengefasst:

Mit **INPUT** werden Werte von der Tastatur in Variablen eingelesen. Zuerst kann ein String angegeben werden, der vor der Eingabe angezeigt wird. Es darf sich dabei weder um eine Variable noch um einen zusammengesetzten String handeln. Wird kein solcher String angegeben oder folgt dem String ein Strichpunkt statt eines Kommas, dann wird vor der Eingabe (ggf. zusätzlich) ein Fragezeichen angezeigt.

Anschließend folgen, durch Komma getrennt, die Namen der einzulesenden Variablen. Die einzelnen Werte werden vom Benutzer ebenfalls durch Komma getrennt eingegeben.

LINE INPUT erlaubt nur eine Eingabevariable, die ein **STRING** sein muss, dafür aber auch das Einlesen einer kompletten Zeile einschließlich Kommata ermöglicht.

5.2. Eingabe einzelner Zeichen

5.2.1. INPUT() als Funktion

Manchmal ist eine so freie Eingabe, wie sie durch die Anweisung **INPUT** ermöglicht wird, nicht gewünscht. Stattdessen wollen Sie vielleicht gezielt angeben, wie viele Zeichen eingegeben werden sollen. Dazu kann **INPUT** als Funktion eingesetzt werden. Eine Funktion ist weitgehend dasselbe wie eine Anweisung, nur dass eine Funktion einen Rückgabewert liefert. Einige FreeBASIC-Befehle, wie z. B. **INPUT**, existieren als Anweisung und als Funktion. Um beide voneinander zu unterscheiden, werden wir in diesem Buch alle Funktionen mit anschließenden runden Klammern benennen, also in diesem Fall **INPUT()**. Der Hintergrund für diese Schreibweise ist, dass die Parameter, die einer Funktion mitgegeben werden, stets mit runden Klammern umschlossen sein müssen. Eine Funktion ohne Parameter benötigt keine Klammern, weshalb diese dann im Quelltext in der Regel weggelassen werden (hier wird der Funktionen-Charakters durch die verwendete Zuweisung deutlich). Wenn jedoch im laufenden Text von einer Funktion gesprochen wird, werden wir zur Verdeutlichung die Klammern schreiben.

Beachten Sie unbedingt den Datentyp des Rückgabewertes! Im Falle von **INPUT()** ist der Rückgabewert immer und ausschließlich ein **STRING**. Wenn Sie ihn also in einer Variablen speichern wollen (siehe Quelltext 5.2), muss es sich dabei um eine

Stringvariable handeln.

Hinweis:
Viele FreeBASIC-Funktionen können genauso wie eine Anweisung verwendet werden, indem man einfach den Rückgabewert auslässt (**INPUT()** gehört allerdings nicht dazu).

Quelltext 5.2: Einzelzeichen mit INPUT()

```
DIM AS STRING taste
PRINT "Druecke eine beliebige Taste."
taste = INPUT(1)
PRINT "Du hast '"; taste; "' gedrueckt."
SLEEP
```

INPUT(1) bedeutet, dass auf die Eingabe genau eines Zeichens gewartet wird, oder genauer gesagt: Die Funktion holt genau ein Zeichen aus dem Tastaturpuffer. Immer, wenn Sie eine Taste drücken, wird der dieser Taste zugeordnete Wert in den Tastaturpuffer gelegt. Von dort aus kann er, z. B. mit **INPUT()**, ausgelesen werden. Ist der Puffer leer, dann wartet **INPUT()** so lange, bis eine Taste gedrückt wird. Wollen Sie mehrere Zeichen auslesen, müssen Sie nur den Parameter entsprechend anpassen. **INPUT(3)** beispielsweise wartet auf drei Zeichen.

Achtung:
Sondertasten wie z. B. die Funktions- oder Pfeiltasten belegen zwei Zeichen im Puffer, weil ihnen eine Kombination aus zwei Werte zugeordnet sind! In Kapitel 13.5 finden Sie dazu eine Übungsaufgabe, die aber zum jetzigen Zeitpunkt noch einiges an Vorwissen benötigt.

5.2.2. INKEY()

Die Funktion **INKEY()** erfüllt einen ähnlichen Zweck wie **INPUT()**. Es gibt jedoch zwei entscheidende Unterschiede: Erstens erwartet **INKEY()** keine Parameter; es wird immer genau eine Taste eingelesen. Bei einer solchen leeren Parameterliste können die Klammern auch weggelassen werden. Zweitens wartet die Funktion nicht auf einen Tastendruck. Ist der Tastaturpuffer leer, wird ein Leerstring zurückgegeben. **INKEY()** bietet sich damit auch an, um den Puffer zu leeren (etwa um weitere Abfragen nicht mit „Altlasten“ aus

einem bereits gefüllten Tastaturpuffer zu beginnen).

INKEY() besitzt den Vorteil, dass Sonderzeichen komplett übergeben werden. Beispielsweise ist der Rückgabewert bei einem Druck auf eine Pfeiltaste zwei Zeichen lang. Allerdings werden wir im Augenblick noch nicht viel mit der Funktion anfangen können, da unsere Programme viel zu schnell vorbei sind, um rechtzeitig eine Taste zu drücken. **INKEY()** wird erst dann interessant, wenn in das Programm Schleifen eingebaut werden – aber dazu kommen wir erst später. **INKEY()** wird beispielsweise in Quelltext 11.9 eingesetzt.

Der Vollständigkeit halber werden noch zwei weitere Befehle zur Tastaturabfrage genannt: **GETKEY()** wird vorwiegend verwendet, um auf einen Tastendruck zu warten, liefert jedoch ebenfalls einen Rückgabewert. Allerdings wird ein **INTEGER** zurückgegeben. Es handelt sich dabei um den ASCII-Wert des Zeichens bzw. bei Sondertasten um einen kombinierten Wert.

Wenn weniger der Tastaturpuffer als vielmehr der Status einer Taste – gedrückt oder nicht gedrückt – interessiert, bietet sich **MULTIKEY()** an. Der Name dieser Funktion deutet bereits an, dass damit mehrere Tasten gleichzeitig überprüft werden können. Auf beide Befehle wird später genauer eingegangen.

5.3. Fragen zum Kapitel

Das Kapitel beschäftigte sich mit der Eingabe über die Tastatur. Dazu einige Fragen:

1. Welche Bedeutung hat bei der Anweisung **INPUT** der Strichpunkt, welche Bedeutung hat das Komma?

2. Was ist der Unterschied zwischen der Anweisung **INPUT** und der Funktion **INPUT()**?

3. Worin liegt der Unterschied zwischen **INPUT()** und **INKEY()**?

4. Und noch eine Programmieraufgabe: Schreiben Sie ein kleines Rechenprogramm. Es soll den Benutzer zwei Zahlen eingeben lassen und anschließend die Summe beider Zahlen berechnen und ausgeben.

6. Variablen und Konstanten

In Kapitel 4 wurden ja bereits die ersten Variablen eingeführt. Eine Variable ist, wie schon erwähnt, eine Speicherstelle, an der Werte abgelegt werden können. Welche Art von Werten eine Variable speichern kann, hängt vom Variablentyp ab.

Grundsätzlich gibt es in FreeBASIC vier verschiedene Typen von Variablen:

- Ganzzahlen (also ohne Nachkommastellen)
- Gleitkommazahlen (Zahlen mit Nachkommastellen)
- Zeichenketten (Strings)
- Wahrheitswerte

Außerdem ist es möglich aus den vorgegebenen Datentypen eigene benutzerdefinierte zusammenzubauen – eine Sache, die in Kapitel 7 beschrieben wird.

6.1. Ganzzahlen

6.1.1. Verfügbare Ganzzahl-Datentypen

Die Ganzzahl-Typen in FreeBASIC unterscheiden sich in ihrer Größe, d. h. sowohl im verfügbaren Zahlenbereich als auch im Speicherbedarf. Der kleinstmögliche Typ ist **`BYTE`**, welcher – der Name verrät es bereits – ein Byte Speicherplatz benötigt. Ein Byte besteht aus acht Bit, daher kann ein Zahlenbereich von $2^8 = 256$ Werten gespeichert werden. Dieser muss noch so gleichmäßig wie möglich auf die negativen und positiven Zahlen aufgeteilt werden. Letztendlich ergibt sich ein Zahlenbereich von -128 bis 127.

Der nächstgrößere Datentyp, das **`SHORT`**, hat bereits zwei Byte, also 16 Bit, zur Verfügung und bietet damit Platz für $2^{16} = 32768$ Werte – wieder zur Hälfte negativ. Der Datentyp **`LONG`** schließlich verbraucht vier Byte Speicherplatz (2^{32} Werte). Wem das noch nicht ausreicht, kann sich mit dem acht Byte großen **`LONGINT`** einen Bereich von 2^{64} Werten sichern.

Etwas komplizierter ist die Speicherplatzangabe beim Datentyp **`INTEGER`**: dessen Größe hängt davon ab, für welche Plattform compiliert wird. Wenn Sie die 32-Bit-Version

des Compilers verwenden, belegt ein **INTEGER** 32 Bit (entspricht also einem **LONG**); in der 64-Bit-Version belegt es 64 Bit (entspricht einem **LONGINT**). Ein **INTEGER** belegt also immer den Speicherplatz, der von der Architektur direkt in einem Schritt angesprochen werden kann.

Da nicht immer negative Wertebereiche benötigt werden, gibt es zu jedem der vorgestellten Datentypen noch eine vorzeichenlose Variante. „Vorzeichenlos“ heißt im englischen „unsigned“, weshalb zur Kennzeichnung der vorzeichenlosen Datentypen ein U am Anfang des Schlüsselwortes verwendet wird. Ein **UBYTE** ist ebenso wie ein **BYTE** acht Bit groß, muss die `256` möglichen Werte jedoch nicht auf den positiven und negativen Bereich aufteilen – daher kann ein **UBYTE** Zahlen von `0` bis `255` speichern. Entsprechend gibt es auch die Datentypen **USHORT**, **ULONG**, **ULONGINT** und **UINTEGER**.

Zusammengefasst gibt es also folgende Ganzzahl-Typen:

Datentyp	Größe in Bit	Grenzen	Suffix
BYTE	8 vorzeichenbehaftet	-128 bis +127	
UBYTE	8 vorzeichenlos	0 bis +255	
SHORT	16 vorzeichenbehaftet	-32 768 bis +32 767	
USHORT	16 vorzeichenlos	0 bis +65 535	
LONG	32 vorzeichenbehaftet	-2 147 483 648 bis +2 147 483 647	&, l
ULONG	32 vorzeichenlos	0 bis +4 294 967 295	ul
LONGINT	64 vorzeichenbehaftet	-9 223 372 036 854 775 808 bis +9 223 372 036 854 775 807	ll
ULONGINT	64 vorzeichenlos	0 bis +18 446 744 073 709 551 615	ull
INTEGER	32/64 vorzeichenbehaftet	siehe LONG bzw. LONGINT	%
UINTEGER	32/64 vorzeichenlos	siehe ULONG bzw. ULONGINT	

Tabelle 6.1.: Datentypen für Ganzzahlen

Das angegebene Suffix kann bei den Zahlendatentypen an die Zahl (nicht an den Variablennamen!) angehängt werden, um für den Compiler deutlich zu machen, dass es sich um den gewünschten Datentyp handelt. Interessant ist das vor allem dann, wenn der Datentyp aus dem Wert heraus automatisch ermittelt wird (z. B. bei **CONST** oder **VAR**).

Achtung:
Die Variablennamen dürfen, im Gegensatz zu QuickBASIC und vielen anderen BASIC-Dialekten, keine Suffixe erhalten. Alle Variablen müssen explizit über **DIM** o. ä. deklariert werden.

Bei der Entscheidung, welcher Datentyp nun der beste ist, spielen hauptsächlich zwei Faktoren eine Rolle: der benötigte Speicherplatz und die Verarbeitungsgeschwindigkeit. Dabei ist die Geschwindigkeit häufig das wichtigere Kriterium. Dass die Datentypen nun unterschiedlich viel Speicherplatz benötigen, ist offensichtlich – aber warum erfolgt ihre Verarbeitung unterschiedlich schnell?

FreeBASIC erstellt, je nach Architektur, 32-Bit- bzw. 64-Bit-Programme, das bedeutet, dass 32 bzw. 64 Bit gleichzeitig (innerhalb eines Taktes) verarbeitet werden können. Umgekehrt heißt das aber auch, dass Variablen anderer Größe nicht direkt verwendet werden können. Wenn Sie z. B. in einer 32-Bit-Umgebung eine **BYTE**-Variable einsetzen, belegt diese nur acht Bit einer 32-Bit-Speicherstelle.[5]. Um den Wert der Variablen zu ändern, müssen diese acht Bit zuerst „extrahiert“ und am Ende wieder so zurückgeschrieben werden, dass die restlichen 24 Bit davon nicht betroffen werden. Das geschieht alles automatisch im Hintergrund, und Sie werden von den Operationen nichts mitbekommen – aber dennoch kosten sie ein wenig Zeit. Gerade wenn Sie sehr aufwändige Berechnungen durchführen müssen, kann das durchaus eine Geschwindigkeitseinbuße von bis zu 10% ausmachen! Sofern also der Speicherplatzbedarf keine allzu große Rolle spielt, bietet sich die Verwendung von **INTEGER** bzw. **UINTEGER** besser an.

Einen Nachteil dieser flexiblen Anpassung sollte man allerdings beachten: Wenn Sie **INTEGER**-Variablen in einer Datei speichern und später wieder auslesen, wird das nur dann fehlerfrei funktionieren, wenn das schreibende und das lesende Programm dieselbe **INTEGER**-Größe verwenden. Insbesondere der Datenaustausch zwischen verschiedenen Architekturen wird dadurch erschwert. Das ist bei weitem keine unüberwindliche Hürde, sollte allerdings im Hinterkopf behalten werden. Am besten verzichten Sie immer dann auf **INTEGER** bzw. **UINTEGER**, wenn Sie eine (wie auch immer geartete) feste Größe des Datentyps erwarten. Auf die besonders kritischen Operationen werden Sie im Laufe des Buches ausdrücklich hingewiesen.

6.1.2. Alternative Schreibweise: INTEGER<n>

Die hohe Zahl an Schlüsselwörtern – wie wir gesehen haben, gibt es allein für Ganzzahl-Datentypen bereits zehn Schlüsselwörter – ist nicht unbedingt ein Vorteil der BASIC-Sprachfamilie. Für die Datentypen **LONG** und **INTEGER** kommt erschwerend hinzu, dass diese in unterschiedlichen Sprachen (und sogar plattformabhängig) unterschiedlich behandelt werden. Während (sprachübergreifend) in einem 32-Bit-System sowohl **LONG** als auch **INTEGER** weitestgehend einheitlich als 32-Bit-Datentyp behandelt werden, bleibt im

[5] Nichtsdestotrotz wird das System oft eine komplette **INTEGER**-Stelle reservieren, d. h. bei einzelnen, nicht in größeren Strukturen eingebundenen **BYTE** kommen Sie unter Umständen gar nicht in den Genuss der Speicherersparnis, müssen aber trotzdem mit dem Geschwindigkeitsverlust leben.

64-Bit-FreeBASIC die Länge von **LONG** gleich, während sich **INTEGER**-Länge verdoppelt. In C ist das Verhalten compilerabhängig, in der Regel arbeiten aber C-Compiler unter Windows immer mit einer **long**-Größe von 32 Bit, während unter einem 64-Bit-Linux 64 Bit belegt werden (nicht jedoch unter einem 32-Bit-Linux). Java wiederum verwendet plattformunabhängig immer 32 Bit für den Datentyp **int** und 64 Bit für den Datentyp **long**. Wie Sie sehen, gibt es in der Computerwelt keine einheitliche Bezeichnung. Gerade wenn man in mehreren Programmiersprachen gleichzeitig aktiv ist, kann das durchaus Verwirrung stiften.

Um das „Chaos“ etwas zu entschärfen, wurde in FreeBASIC mit der Compilerversion v0.90 eine weitere mögliche Schreibweise eingeführt: Sie können nun Ganzzahlen immer mit **INTEGER** bzw. **UINTEGER** kennzeichnen, gefolgt von der gewünschten Größe in Spitzklammern. Kurz gesagt bedeutet das:

- **[U]INTEGER<8>** ist gleichbedeutend mit **[U]BYTE**.
- **[U]INTEGER<16>** ist gleichbedeutend mit **[U]SHORT**.
- **[U]INTEGER<32>** ist gleichbedeutend mit **[U]LONG**.
- **[U]INTEGER<64>** ist gleichbedeutend mit **[U]LONGINT**.

Andere Zahlen als `8`, `16`, `32` und `64` können nicht verwendet werden.

6.1.3. Rechnen mit Ganzzahlen

In FreeBASIC stehen die gewohnten Grundrechenarten und eine große Zahl mathematischer Funktionen zur Verfügung. Der Compiler folgt dabei den gewohnten Rechenregeln wie Punkt vor Strich und Vorrang der Klammern. Syntaktisch sind jedoch, bedingt durch den zur Verfügung stehenden Zeichensatz, einige Besonderheiten zu beachten. U. a. können für die Berechnung nur runde Klammern verwendet werden (eckige und geschweifte Klammern haben syntaktisch eine andere Bedeutung), diese können aber beliebig oft ineinander verschachtelt werden. Auch Exponenten a^b und der Malpunkt im Produkt $c \cdot d$ müssen auf eine andere Art geschrieben werden. Tabelle 6.2 fasst die Rechenzeichen noch einmal zusammen.

Da alle Datentypen nur einen eingeschränkten Wertebereich besitzen, kann es vorkommen, dass das Ergebnis einer Rechnung nicht mehr im erlaubten Bereich liegt. Wird auf diese Weise „über den Speicherbereich hinaus“ gerechnet, dann meldet das Programm keinen Fehler, sondern rechnet „auf der anderen Seite“ des Bereichs weiter. Wird z. B. zu einem **BYTE** mit dem Wert `127` noch `1` hinzugezählt, dann springt der Wert hinunter auf `-128`. In die andere Richtung gilt dasselbe.

$a + b$	Addition
$a - b$	Subtraktion
$a * b$	Multiplikation
$a \ / \ b$	Division
$a \setminus b$	Integerdivision (ohne Rest)
$a \ \hat{} \ b$	Potenz a^b

Tabelle 6.2.: Grundrechenarten

Quelltext 6.1: Rechnen über den Speicherbereich hinaus

```
DIM AS BYTE start = 100, neu
neu = start + 27
PRINT "100 + 27  = "; neu
neu = start + 28
PRINT "100 + 28  = "; neu
neu = start + 30
PRINT "100 + 30  = "; neu
neu = start - 300
PRINT "100 - 300 = "; neu
SLEEP
```

Ausgabe

```
100 + 27   =  127
100 + 28   = -128
100 + 30   = -126
100 - 300  =  56
```

Im oben stehenden Beispiel wurde nur der Datentyp **BYTE** verwendet, da dies der kleinste Datentyp ist und sich bei ihm das Verhalten am einfachsten nachvollziehen lässt. Prinzipiell ist das Verhalten jedoch bei allen Ganzzahl-Datentypen gleich: Sobald die Grenze des Speicherbereichs über- oder unterschritten wird, wird „auf der anderen Seite“ des Bereichs weitergerechnet. Dieses Verhalten (man nennt es auch *Überlauf*, engl.: *overflow*) kann durchaus erwünscht sein,[6] manchmal aber auch zu unerwarteten Fehlern führen. Sie sollten sich diese Tatsache daher immer vor Augen halten und ggf. Prüfungen einbauen, ob über den Zahlenbereich hinaus gerechnet wird.

Die Zeichen **+**, **-** usw. werden *Operatoren* genannt (in diesem Fall Rechenoperatoren), die beiden Zahlen links und rechts vom Operator sind die *Operanden*. Wir werden aber

[6] Der Versuch vieler Staatsregierungen, durch ausreichend hohe Verschuldung einen Overflow zu erreichen, sind leider zum Scheitern verurteilt.

im Laufe dieses Buches noch auf eine ganze Reihe weiterer Operatoren stoßen.

6.1.4. Kurzschreibweisen

Zu den oben genannten Rechenoperatoren gibt es Kurzschreibweisen, welche, auch in diesem Buch, gern genutzt werden. Will man zu einer gegebenen Zahl einen bestimmten Wert addieren, lässt sich das auch in verkürzter Form als `zahl += wert` schreiben.

```
n += x    ' ist identisch mit n = n+x
n -= x    ' ist identisch mit n = n-x
n *= x    ' ist identisch mit n = n*x
n /= x    ' ist identisch mit n = n/x
n \= x    ' ist identisch mit n = n\x
n ^= x    ' ist identisch mit n = n^x
```

Vor allem den Kurzformen `n += 1` und `n -= 1` werden Sie häufiger begegnen, wenn ein Wert inkrementiert (also um `1` erhöht) oder dekrementiert (um `1` vermindert) werden soll. Die Kurzformen existieren aber nicht nur für die Rechenoperatoren, sondern für nahezu alle Operatoren mit zwei Operanden – Ausnahmen sind offensichtliche Fälle wie Vergleichsoperatoren, bei denen eine Kurzschreibweise keinen Sinn ergibt.

6.2. Gleitkommazahlen

6.2.1. Darstellungsbereich von Gleitkommazahlen

Um mit Gleitkommazahlen zu rechnen, gibt es die beiden Datentypen **`SINGLE`** und **`DOUBLE`**. Die Schlüsselwörter stehen für *single* bzw. *double precision*, also einfache bzw. doppelte Genauigkeit. Ein **`SINGLE`** belegt vier Byte, während ein **`DOUBLE`** mit acht Byte den doppelten Speicherplatz belegt und daher die Werte auch genauer speichern kann.

Das Problem ist nämlich folgendes: Die meisten Dezimalzahlen lassen sich gar nicht mit einer endlichen Anzahl an Nachkommastellen schreiben. Wenn z. B. eine Rechnung wie „1 geteilt durch 3“ durchgeführt wird, erhält man ein periodisches Ergebnis mit unendlich vielen Nachkommastellen, mathematisch exakt ausgedrückt durch $\frac{1}{3} = 0.\overline{3} = 0.33333333\ldots$

Der Computer kann natürlich nicht unendlich viele Nachkommastellen speichern – dazu bräuchte er ja unendlich viel Speicherplatz – und muss daher das Ergebnis irgendwo abschneiden bzw. runden (ein Taschenrechner macht übrigens dasselbe). Wird dann mit den gerundeten Werten weitergerechnet, ergeben sich dadurch möglicherweise weitere Rundungsfehler. Die Anzahl der gespeicherten Nachkommastellen ist dabei ausschlaggebend für die Rechengenauigkeit.

Ein **SINGLE** kann etwa 7 Stellen, ein **DOUBLE** etwa 16 Stellen speichern. Es handelt sich dabei nicht notwendigerweise um *Nachkomma*stellen. Hier kommt die Bedeutung des Begriffs *Gleitkommazahl* zu tragen. Ein **SINGLE** kann z. B. den Wert `1234567` oder `12.34567` oder `12345670000` speichern – alle drei Mal gibt es sieben signifikante Stellen, nur die Position des Dezimalpunkts unterscheidet sich. Deutlicher wird das in der Exponentialdarstellung (auch als *wissenschaftliche Notation* bezeichnet): $12345670000 = 1.234567 \cdot 10^{10}$ und $123.4567 = 1.234567 \cdot 10^{2}$. Auf diese Art und Weise können sowohl sehr große als auch sehr kleine Zahlen (nahe `0`) gespeichert werden, ohne dass die Anzahl der signifikanten Stellen davon betroffen ist. Wie groß bzw. wie klein die Zahlen sein dürfen, können Sie der Tabelle entnehmen:

Datentyp	**kleinster Wert (nahe 0)**	**größter Wert (nahe ∞)**	**Suffix**
SINGLE	±1.401 298 e-45	±3.402 823 e+38	!, f
DOUBLE	±4.940 656 458 412 465e-324	±1.797 693 134 862 316e+308	#, d

Tabelle 6.3.: Datentypen für Gleitkommazahlen

Die Angabe `1.401298e-45` ist die in FreeBASIC (und vielen anderen Programmiersprachen) übliche wissenschaftliche Notation $1.401298 \cdot 10^{-45}$. Das Komma ist also um 45 Stellen nach links verschoben. Oder genauer gesagt der Dezimal*punkt* – wie in den meisten Programmiersprachen wird als Dezimaltrennzeichen der Punkt verwendet. Entsprechend bedeutet `3.402823e+38` eine Verschiebung des Dezimalpunktes um 38 Stellen nach rechts. Damit kann eine Zahl bis zu 340 Sextillionen gespeichert werden.[7] Für den Normalbedarf sollte das ausreichen.

Wird eine Zahl betragsmäßig kleiner als der erlaubte kleinste Wert, so wird sie als `0` gespeichert. Wird sie größer als der erlaubte größte Wert, wird sie intern als „unendlich groß“ (∞) bzw. „unendlich klein“ ($-\infty$) behandelt, dargestellt als `inf` bzw. `-inf`.

[7] Beachten Sie hierzu aber, dass die Genauigkeit eines **DOUBLE** weiterhin bei etwa 16 Stellen liegt. Wenn Sie zu 340 Sextillionen eine weitere Milliarde addieren, bleiben Sie nach wie vor bei 340 Sextillionen – der Summand ist hier zu klein, um eine Veränderung herbeizuführen.

Hinweis:
Beim Rechnen mit „unendlich großen“ Zahlen wird kein Runtime-Error ausgegeben – mit `inf` und `-inf` wird gerechnet, als ob es sich um normale Zahlen handeln würde. Kann der Wert einer Rechnung nicht exakt bestimmt werden, wird stattdessen der Wert `nan` (oder `-nan`; steht für „not a number“) verwendet. Das kann z. B. bei den Rechnungen `inf-inf` oder `inf*0` oder bei der Division durch `0` geschehen. Das genaue Ausgabeformat unterscheidet sich dabei je nach System – unter Windows werden Sie möglicherweise die Ausgabe `1.#IND` statt `nan` und `1.#INF` statt `inf` zu lesen bekommen.
`nan` und `inf` arbeiten unter FreeBASIC allerdings nicht ganz so, wie von der Norm IEEE 754 vorgesehen. Daher reicht es, von ihrer Existenz zu wissen; ein korrekter Umgang mit den Werten ist mit der Standardbibliothek leider nicht möglich.

6.2.2. Rechengenauigkeit bei Gleitkommazahlen

Während sich die Genauigkeit von Ganzzahlberechnungen exakt angeben lässt, müssen wir bei Berechnungen mit Gleitkommazahlen mit einer gewissen Rechenungenauigkeit zurechtkommen. Das folgende Beispiel zeigt an, wie die Ergebnisse von $\frac{1}{3}$, $\frac{3}{7}$ und $\frac{100}{3}$ gespeichert werden.

Quelltext 6.2: Rechnen mit Gleitkommazahlen

```
DIM s AS SINGLE, d AS DOUBLE
' Berechnung von 1/3
s = 1 / 3
PRINT "1/3 als SINGLE: "; s
d = 1 / 3
PRINT "1/3 als DOUBLE: "; d

' Berechnung von 3/7
s = 3 / 7
PRINT "3/7 als SINGLE: "; s
d = 3 / 7
PRINT "3/7 als DOUBLE: "; d

' Berechnung von 100/3
s = 100 / 3
PRINT "100/3 als SINGLE: "; s
d = 100 / 3
PRINT "100/3 als DOUBLE: "; d
SLEEP
```

Ausgabe

```
1/3 als SINGLE:  0.3333333
1/3 als DOUBLE:  0.3333333333333333
3/7 als SINGLE:  0.4285714
3/7 als DOUBLE:  0.4285714285714285
100/3 als SINGLE:  33.33333
100/3 als DOUBLE:  33.33333333333334
```

Die ersten beiden Ausgabezeilen werfen keine weiteren Probleme auf. Gleitkommazahlen haben nur eine gewisse Genauigkeit, daher muss ab einer bestimmten Stelle gerundet werden. Beim **SINGLE** findet die Rundung an der siebten Nachkommastelle statt, beim **DOUBLE** nach der sechzehnten. Die vierte Ausgabezeile zeigt jedoch schon eine Unregelmäßigkeit: Die letzte Ziffer wurde offenbar falsch gerundet. In der sechsten Ausgabezeile wird das noch deutlicher: Die letzte Ziffer hätte eine 3 statt einer 4 sein müssen.

Bevor Panik wegen falsch rechnender Computer ausbricht: Der Grund an den Unstimmigkeiten liegt im Unterschied zwischen dem Zahlensystem des Computers und unserem. Der Computer rechnet üblicherweise im Binärsystem, also einem Zahlensystem, das nur die Ziffern `1` und `0` kennt. Damit kann er nur diejenigen Brüche exakt behandeln, welche als Nenner eine Zweierpotenz besitzen ($\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$ usw.). Andere Bruchzahlen werden wie gewohnt gerundet, nur dass die Rundung im Binärsystem etwas anders ausfallen kann und das Ergebnis bei der Übersetzung in unser Dezimalsystem falsch erscheinen mag. Der Fehler passiert also genau genommen nicht bei der Division, sondern bei der Übersetzung des Ergebnisses in das andere Zahlensystem. Bei dieser Übersetzung kann es dann auch passieren, dass einmal eine angezeigte Stelle mehr oder weniger herauskommt.

In den letzten beiden Ausgabezeilen erkennt man auch die eben erwähnte Rechengenauigkeit nach signifikanten Stellen. Da die Zahlen mehr Stellen vor dem Komma besitzen, sinkt die Anzahl der Nachkommastellen. Der Computer speichert lediglich die Ziffernfolge sowie die Position des Dezimalpunktes. Die Ausgabe erfolgt ab einer bestimmten Größe in Exponentialschreibweise:

```
PRINT ((123 / 1000000) / 1000000) / 1000000
SLEEP
```

Ausgegeben wird der Wert `1.23e-16`. Gemeint ist damit $1.23 \cdot 10^{-16}$, also der Wert 0.000000000000000123, der durch eine Verschiebung des Dezimalpunktes um 16 Stellen nach links entsteht.

Um noch einmal darauf zurückzukommen: auch bei doppelter Rechengenauigkeit

müssen Sie eine gewisse Ungenauigkeit in Kauf nehmen. Probieren Sie einmal folgendes Programm aus:

```
PRINT 0.5 - 0.4 - 0.1
SLEEP
```

Auf den meisten Plattformen wird man folgende oder eine ähnliche Ausgabe erhalten, die auf den ersten Blick überraschen mag:

Ausgabe

```
-2.775557561562891e-17
```

Das Ergebnis hat eine Größenordnung von 10^{-17}; sein Betrag ist also kleiner als ein Zehnbilliardstel. Dennoch hätten wir das Ergebnis `0` erwartet. Dieser, wenn auch sehr kleine, Unterschied kommt wieder daher, dass Computer nicht wie wir im Dezimalsystem, sondern im Binärsystem rechnen. Im Binärsystem sind die Werte der Dezimalzahlen `0.4` und `0.1` periodisch und werden entsprechend gerundet. Der Rundungsfehler wird meistens nicht groß ins Gewicht fallen, aber je nach Rechnung kann er auch mehr oder weniger starke Auswirkungen haben. Sie werden gegen diese Rechenfehler nichts tun können; seien Sie sich einfach immer darüber im Klaren, dass es absolute Genauigkeit bei Gleitkommazahl-Rechnungen nicht gibt.

Im Buch verwenden wir für Gleitkommazahlen immer **DOUBLE** (außer es soll etwas Spezielles mit **SINGLE**-Werten demonstriert werden). Auf heutigen Computerarchitekturen gibt es in der Regel keinen Geschwindigkeitsunterschied zwischen Gleitkommazahlen einfacher und doppelter Genauigkeit (viele Prozessoren arbeiten bei Gleitkommazahlen nicht mit 32 oder 64, sondern mit 80 Bit), weshalb es wenig Gründe gibt, auf die mit **DOUBLE** verbundene höhere Genauigkeit zu verzichten.

6.3. Zeichenketten

Als dritten Standard-Datentyp stellt FreeBASIC Zeichenketten (Strings) zur Verfügung. Dabei handelt es sich um eine beliebige Aneinanderreihung von Zeichen (Buchstaben, Ziffern, Sonderzeichen); eine Zeichenkette kann aber auch komplett leer sein (ein sogenannter Leerstring). Im Quelltext müssen Strings immer in "Anführungszeichen" stehen. Wir haben solche Zeichenketten bereits in den vorigen Kapiteln kennen gelernt.

6.3.1. Arten von Zeichenketten

Ein **STRING** besteht also aus einer Aneinanderreihung von erweiterten ASCII-Zeichen. Damit FreeBASIC weiß, wo die Zeichenkette endet, reserviert es intern – vom Benutzer versteckt – eine weitere Speicherstelle für die Länge der Zeichenkette. Dies sieht, stark vereinfacht dargestellt, etwa folgendermaßen aus:

```
<es folgen 006 Zeichen>WETTER
<es folgen 013 Zeichen>WETTERBERICHT
```

Daneben gibt es noch zwei weitere Typen: den **ZSTRING** und den **WSTRING**.
ZSTRING steht für *zero-terminated string*, also zu Deutsch eine nullterminierte Zeichenkette. Ein **ZSTRING** besitzt keine Speicherstelle für die Länge, sondern hängt ein reserviertes „Stoppzeichen“ an, um das Ende der Zeichenkette zu markieren: das *Nullbyte* mit dem ASCII-Wert `0`.

```
WETTER<Stopp>
WETTERBERICHT<Stopp>
```

Im **ZSTRING** darf nun aber kein Nullbyte vorkommen, da es ja als Stoppzeichen interpretiert werden würde. Würde die Zeichenkette folgendermaßen aussehen:

```
WETTER<Stopp>BERICHT<Stopp>
```

dann würde sie lediglich als `WETTER` interpretiert werden und der hintere Teil `BERICHT` ginge verloren. Ein **STRING** unterliegt dieser Einschränkung nicht. Bis auf das Nullbyte sind in einem **ZSTRING** jedoch alle Zeichen erlaubt, die auch in einem **STRING** verwendet werden können. Welche Zeichen das sind, können Sie Anhang C entnehmen.

Achtung:
Für Konsole und Grafikfenster werden verschiedene ANSI-Codepages verwendet (siehe Anhang C).

ZSTRING wird vor allem benötigt, um einen Datenaustausch mit externer Bibliotheken (z. B. solchen, die in C geschrieben wurden) zu ermöglichen. Ein **STRING** verwendet intern ein eigenes Speicherformat, das nicht ohne weiteres auf andere Programmiersprachen übertragen werden kann. FreeBASIC-intern ist jedoch der Datentyp **STRING** in der Regel leichter zu handhaben.

Ein **WSTRING** ist wie ein **ZSTRING** nullterminiert, nutzt also „Null-Zeichen“ als Stoppzeichen. Er speichert jedoch keine ASCII-, sondern Unicode-Zeichen, besitzt also einen

wesentlich größeren Zeichenvorrat. Dafür wird pro Zeichen natürlich auch mehr Speicherplatz benötigt. Da FreeBASIC bei der Unicode-Unterstützung auf die C runtime library zurück greift, die plattformbedingt variiert, gibt es zwischen den verschiedenen Betriebssystem kleine Unterschiede: Unter DOS wird Unicode nicht unterstützt. Unter Windows werden **WSTRINGs** in UCS-2 codiert (ein Zeichen belegt 2 Bytes), während sie unter Linux in UCS-4 codiert werden (ein Zeichen belegt 4 Bytes). Entsprechend besteht natürlich auch das terminierende „Null-Zeichen“ nicht aus einem, sondern aus zwei bzw. vier Byte.

Im Augenblick werden wir uns auf die Verwendung von **STRINGs** beschränken. Auf den Umgang mit **ZSTRING** und **WSTRING** wird in Kapitel 15 genauer eingegangen; dort erfahren Sie auch Näheres über den internen Aufbau eines **STRINGs** und darüber, was es mit Zeichenketten fester Länge auf sich hat.

Ein **STRING** kann in der 32-Bit-Version des Compilers bis zu 2 GB groß werden, in der 64-Bit-Version sogar 8 388 607 TB (die Größe entspricht also dem plattformabhängigen Wertebereich eines **INTEGERs**).

Hintergrundinformation:
Auch beim Datentyp **STRING** wird intern am Ende ein Nullbyte angehängt, um kompatibler zu externen Bibliotheken zu sein. Dieses Nullbyte ist jedoch innerhalb von FreeBASIC nicht terminierend.
Achtung: In QuickBASIC wird kein solches Nullbyte angehängt. Insbesondere sind UDTs, die **STRINGs** enthalten, zwischen QuickBASIC und FreeBASIC nicht kompatibel!

6.3.2. Aneinanderhängen zweier Zeichenketten

Selbstverständlich kann mit Zeichenketten nicht „gerechnet“ werden – dies bleibt den Zahlen vorbehalten – jedoch gibt es auch für sie Bearbeitungsmethoden. Auch für Zeichenketten ist das Pluszeichen definiert, besitzt dort jedoch eine etwas andere Bedeutung: mit ihm werden zwei Zeichenketten einfach aneinander gehängt. Man spricht dabei von einer Stringverkettung oder Konkatenation der Zeichenketten.

Quelltext 6.3: Stringverkettung mit Pluszeichen

```
DIM AS STRING vorname = "Max", nachname = "Muster", gesamtname
gesamtname = vorname + " " + nachname
PRINT gesamtname

PRINT  123  +  456
PRINT "123" + "456"

SLEEP
```

Ausgabe

```
Max Muster
 579
123456
```

Zunächst einmal wurde der Vorname mit einem Leerzeichen und dem Nachnamen zusammengehängt. Das Ergebnis ist wohl sehr einleuchtend. Dasselbe passiert auch, wenn die Zeichenketten Zahlenzeichen enthalten – sie werden einfach aneinander gehängt. Dem Compiler ist dabei egal, was die Zeichenketten enthalten; er behandelt Ziffern in einer Zeichenkette genauso wie Buchstaben oder Sonderzeichen. Während in Zeile 5 eine normale Addition zweier Zahlen durchgeführt wird, handelt es sich in Zeile 6 um zwei Zeichenketten, die aneinandergehängt werden. Das Pluszeichen erzielt also für die verschiedenen Datentypen zwei völlig unterschiedliche Ergebnisse.

Eine „Addition“ einer Zahl mit einer Zeichenkette ist nicht möglich und wird zu einem Compiler-Fehler führen. Stattdessen muss zuerst die Zahl in einen String umgewandelt werden, um beide Zeichenketten miteinander zu verketten, oder der String in eine Zahl, um eine Rechenoperation durchzuführen. Wie eine solche Umwandlung durchgeführt wird, folgt später. Für eine String-Konkatenation gibt es jedoch noch eine andere Möglichkeit: die *et-Ligatur* **&** (auch *kaufmännisches Und* oder *Ampersand* genannt). Der Vorteil bei **&** ist, dass die verwendeten Werte automatisch zu Zeichenketten umgewandelt werden. Sie können es also auch zur Verkettung eines Strings mit einer Zahl oder sogar zur Verkettung zweier Zahlen verwenden – diese werden zuerst zu Zeichenketten umgewandelt und anschließend aneinandergehängt.

Quelltext 6.4: Stringverkettung mit et-Ligatur

```
DIM AS STRING vorname = "Max", nachname = "Muster", gesamtname
gesamtname = vorname & " " & nachname
' arbeitet genauso wie vorname + " " + nachname
PRINT gesamtname

PRINT  123  +  456
PRINT "123" + "456"
PRINT  123  &  456
PRINT "ABC" &  456
' PRINT "ABC" + 456   ' FEHLER: Das ist nicht erlaubt!

SLEEP
```

Ausgabe

```
Max Muster
 579
123456
123456
ABC456
```

Natürlich kann man mit Zeichenketten noch allerhand andere interessante Dinge anstellen, wie z. B. nur einen Teil der Zeichenkette ausgeben, in Groß- oder Kleinbuchstaben umwandeln und vieles mehr. Dazu erfahren Sie mehr in Kapitel 15.

6.4. Wahrheitswerte

Als letzten Standarddatentyp steht das *Boolean* zur Verfügung (benannt nach dem englischen Mathematiker George Boole). Ein **BOOLEAN** kann nur zwei Werte annehmen: den Wert `true` (*wahr*) und den Wert `false` (*falsch*).

```
DIM AS BOOLEAN wahrheitswert = true
PRINT wahrheitswert
SLEEP
```

Die Bedeutung des **BOOLEAN** wird erst ab Kapitel 10 wirklich interessant. Allerdings sollte bereits jetzt erwähnt werden, dass FreeBASIC auch alle Zahlenwerte als Wahrheitswerte interpretiert: Ist die Zahl `0`, so gilt sie als *falsch*; in allen anderen Fällen gilt sie als *wahr*. Die Trennungslinie zwischen **BOOLEAN** und Zahlen läuft daher nicht immer so sauber, wie es zu wünschen wäre; wenn sie aber zu grob missachtet wird, erhalten Sie zumindest eine Warnung vom Compiler.

6.5. Konstanten

Konstanten sind den Variablen sehr ähnlich, allerdings kann ihnen nur bei der Deklaration ein Wert zugewiesen werden. Eine spätere Änderung des Wertes ist nicht möglich. Konstanten können z. B. eingesetzt werden, wenn zu Beginn des Programmes bestimmte Einstellungen festgelegt werden sollen, die sich später nicht mehr ändern dürfen. Eine weitere Möglichkeit ist die Definition mathematischer oder physikalischer Konstanten, die im Programm benötigt werden. Konstanten werden üblicherweise komplett in Großbuchstaben geschrieben, um sie sofort als solche erkennen zu können (auch wenn dem Compiler die Groß- und Kleinschreibung nach wie vor egal ist).

Außerdem sind Konstanten im ganzen Programm gültig – eine Eigenschaft, die z. B. beim Einsatz von Prozeduren interessant wird (vgl. Kapitel 12).

Das folgende Programm legt den Programmnamen und die Kreiszahl π als Konstanten fest. Die Werte können anschließend genauso wie Variablen ausgegeben oder für Berechnungen, bei **STRINGs** auch für Konkatenationen verwendet werden.

Quelltext 6.5: Konstanten

```
CONST TITEL = "Kreisberechnungsprogramm"
CONST PI = 3.14    ' zwar recht ungenau, reicht aber fuer unsere Zwecke
DIM AS INTEGER radius

PRINT TITEL
PRINT
INPUT "Gib den Kreisradius an: ", radius
PRINT "Der Kreis hat etwa den Umfang " & (2*radius*PI);
PRINT " und den Flaecheninhalt " & (radius^2*PI)
SLEEP
```

Ausgabe

```
Kreisberechnungsprogramm

Gib den Kreisradius an: 12
Der Kreis hat etwa den Umfang 75.36 und den Flaecheninhalt 452.16
```

Wie Sie sehen, muss bei der Anweisung **CONST** nicht angegeben werden, um welchen Datentypen es sich handelt. FreeBASIC entscheidet selbständig, welcher Datentyp für die angegebenen Daten am praktischsten ist. Ganzzahlen werden als **INTEGER** gespeichert (wegen der Rechengeschwindigkeit) oder (in 32-Bit-Systemen) als **LONGINT**, wenn ein **INTEGER** zu klein für den angegebenen Wert ist. Gleitkommazahlen werden als **DOUBLE** gespeichert (wegen der Rechengenauigkeit). Ob es sich bei dem Wert um eine Ganz-

zahl oder eine Gleitkommazahl handelt, wird ganz einfach daran unterschieden, ob ein Dezimalpunkt vorkommt: `1` ist eine Ganzzahl und `1.0` eine Gleitkommazahl.

Allerdings kann der Datentyp einer Konstanten auch direkt festgelegt werden:

```
CONST PI AS SINGLE = 3.14
```

Damit wird PI nicht mehr als **DOUBLE**, sondern als **SINGLE** verarbeitet.

Auch die Verwendung der in Tabelle 6.1 angeführten Suffixes ist erlaubt, z. B.:

```
CONST x = 45ul
CONST y = 45%
CONST z = 45LL   ' Auch bei Suffixen wird die Gross-/Kleinschreibung ignoriert
```

6.6. Nummerierungen (Enumerations)

Eine besondere Art der Konstanten sind die Enumerations. Hierbei handelt es sich um eine einfache Nummerierung von Konstanten-Werten: Sie geben lediglich eine Folge von Namen für die gewünschten Konstanten an, und der Compiler übernimmt automatisch die Wertzuweisung. Als Datentyp wird sinnvollerweise immer **INTEGER** verwendet.

Eine Enumeration bietet sich an, wenn Sie verschiedene unterscheidbare Werte mit sprechenden Namen belegen wollen, der tatsächliche Wert dabei aber unerheblich ist. Denken Sie dabei z. B. an verschiedene Optionen in einem Auswahldialog:

Quelltext 6.6: Verwendung von ENUM

```
ENUM AUSWAHL
  einkauf, verkauf, ausbau, preisaenderung
END ENUM

DIM AS AUSWAHL meineWahl = verkauf
'...
```

Die Verwendung eines Listennamen (hier `AUSWAHL`) ist optional, aber empfehlenswert. Unter anderem können Sie dann wie in Quelltext 6.6 diese Enumeration auch als Typ einer Variablen verwenden (der Compiler überprüft aber leider nicht, ob anschließende Wertzuweisungen wirklich Werte aus der Enumeration sind). Außerdem kann der Name vor die Variable gesetzt werden, um die Zugehörigleit eindeutig festzulegen:

```
meineWahl = AUSWAHL.verkauf
```

Das bringt einige Vorteile mit sich – der entscheidenste ist sicherlich, dass verschiedene Enumerations dieselben Variablennamen verwenden können, ohne sich gegenseitig in die Quere zu kommen, und dass Sie bei der Namenswahl „normaler“ Variablen weniger

eingeschränkt werden. Immerhin kann es in größeren Projekten leicht vorkommen, dass Sie verschiedene externe Bibliotheken verwenden, welche dieselben Namen definieren wollen. Solange nun die Enumerations unterschiedlich benannt sind, stören gleiche Variablennamen nicht. (Über Bereichsnamen, die in **??** behandelt werden, können auch Konflikte gleichnamiger Enumerations verhindert werden.)

Wenn Sie die Verwendung des Listennamens erzwingen wollen, geben Sie dahinter das Schlüsselwort **EXPLICIT** an:

```
ENUM AUSWAHL EXPLICIT
  einkauf, verkauf, ausbau, preisaenderung
END ENUM

' DIM AS AUSWAHL meineWahl = verkauf        ' funktioniert nicht mehr
DIM AS AUSWAHL meineWahl = AUSWAHL.verkauf  ' funktioniert dagegen
'...
```

Sie sehen, dass Sie dadurch für mehr Schreibarbeit sorgen; dafür wird aber auch die Lesbarkeit deutlich erhöht, da jederzeit klar ist, woher der Wert `einkauf` stammt.

Ein umfangreiches Beispiel mit der Verwendung von **ENUM** finden Sie in Quelltext 14.14.

Noch eine Information zur Wertzuweisung:
FreeBASIC weist innerhalb der Enumeration der ersten Variablen den Wert `0` zu und zählt dann jeweils um `1` nach oben. In Quelltext 6.6 ist dann also `AUSWAHL.einkauf=0`, `AUSWAHL.verkauf=1`, `AUSWAHL.ausbau=2` und `AUSWAHL.preisaenderung=4`. Dieses Verhalten können Sie auch beeinflussen, indem Sie einer Variablen den gewünschten Wert zuweisen. Alle folgenden Variablen werden anschließend normal weiternummeriert.

Quelltext 6.7: ENUM mit Wertzuweisung

```
ENUM AUSWAHL
  einkauf, verkauf, ausbau=7, preisaenderung
END ENUM

PRINT AUSWAHL.einkauf, AUSWAHL.verkauf, AUSWAHL.ausbau, AUSWAHL.preisaenderung
SLEEP
```

Ausgabe

```
 0             1             7             8
```

Enumerations sind jedoch weiterhin Konstanten. Eine Wertzuweisung ist ausschließlich bei der Definition zulässig, später ist keine Änderung mehr möglich.

6.7. Weitere Speicherstrukturen

Neben den bisher behandelten Standard-Datentypen (also Ganzzahlen, Gleitkommazahlen und Zeichenketten) stehen noch weitere Möglichkeiten der Datenspeicherung zur Verfügung, die aber in gesonderten Kapiteln behandelt werden. Zum einen lassen sich eigene Datentypen definieren, die sich aus den vorhandenen zusammensetzen, zum anderen gibt es die Arrays, die eine ganze Gruppe von Daten gleichzeitig beinhalten.

6.8. Fragen zum Kapitel

1. Welcher Datentyp bietet sich an, um eine Ganzzahl in der Größenordnung von ±1 000 000 000 zu speichern?
2. Welcher Datentyp bietet sich an, um eine positive Ganzzahl bis `255` zu speichern?
3. Welche Datentypen können für Gleitkommazahlen verwendet werden? Welche Unterschiede gibt es zwischen ihnen?
4. Worin liegt der Unterschied zwischen einem **`STRING`** und einem **`ZSTRING`**?
5. Was ist der Unterschied zwischen einer Variablen und einer Konstanten?
6. Was ist der Unterschied zwischen **`LONG`** und **`INTEGER`**?

7. Benutzerdefinierte Datentypen (UDTs)

Wenn die vorgegebenen Datentypen nicht ausreichen (und das ist in der fortgeschrittenen Programmierung sehr häufig der Fall), lassen sich eigene Datentypen definieren. Im Englischen spricht man von *user defined types*, kurz *UDT*. Ein UDT setzt sich, vereinfacht gesagt, aus mehreren Variablen bereits bekannter Datentypen zusammen. Auf diesen einfachen Fall wollen wir uns in diesem Kapitel beschränken. UDTs können aber deutlich mehr, als nur ein paar Variablen zusammenzufassen: Sie sind die Basis der objektorientierten Programmierung, die zu komplex ist, um sie in ein einziges Kapitel zu pressen, und die daher in einen eigenen Bereich des Buches besprochen wird.

7.1. Deklaration

Ein beliebtes Beispiel für die (hier behandelte vereinfachte) Verwendung eines UDTs ist ein Datentyp zur Speicherung von Adressen. Zu einer Adresse gehören unter anderem der Vorname, Nachname, Straßenname und Hausnummer, die Postleitzahl und der Wohnort. Diese sechs Angaben können selbstverständlich in sechs verschiedenen Variablen gespeichert werden, aber spätestens bei der gleichzeitigen Verwendung mehrerer Adressen wird das unangenehm. Stattdessen werden die Angaben jetzt in einem UDT zusammengefasst und sind dadurch leichter zu verwalten.

```
TYPE Adresse
  AS STRING  vorname, nachname, strasse, ort
  AS INTEGER hausnummer, plz
END TYPE
```

Der Quelltext-Ausschnitt legt einen neuen Datentyp mit dem Namen `Adresse` an, der ab sofort für die Deklaration neuer Variablen verwendet werden kann. Er beinhaltet sechs *Mitglieder* (engl.: *member*), die auch *Attribute* genannt werden.[8] Auf die Attribute kann, sowohl lesend als auch schreibend, nach dem Muster `udtname.attributname` zugegriffen werden. Dazu folgt in Quelltext 7.1 ein Beispiel. Die Attribute, z. B. `vorname`, existieren nur innerhalb des UDTs, es wird also keine allgemein verfügbare Variable mit

[8] In **??** werden wir lernen, dass es zwei Arten von Mitgliedern gibt: Die Attribute (Member-Variablen) und die Methoden (Member-Funktionen).

dem Namen `vorname` angelegt.

Ihnen wird sicher aufgefallen sein, dass die Deklaration der Attribute sehr ähnlich aussieht wie eine Variablendeklaration, nur dass das Schlüsselwort **`DIM`** nicht erforderlich ist (erlaubt ist es allerdings). Der UDT-Typname wird sehr oft groß geschrieben, um ihn optisch von einem Variablennamen abzuheben. Viele Programmierer kennzeichnen Typnamen auch durch ein Prefix, wie z. B. ein vorangestelltes `Typ` oder `T`. Der Typname würde dann `TypAdresse` oder `TAdresse` lauten.

Quelltext 7.1: Anlegen eines UDTs

```
' UDT deklarieren
TYPE TAdresse
  AS STRING  vorname, nachname, strasse, ort
  AS INTEGER hausnummer, plz
END TYPE

' neue Adressenvariablen anlegen
DIM AS TAdresse adresse1, adresse2

' Werte zuweisen
adresse1.vorname  = "Simon"
adresse1.nachname = "Mustermann"
adresse2.vorname  = "Sandra"

' Werte auslesen
PRINT adresse1.vorname
SLEEP
```

Der Vorteil eines UDTs wird auch in diesem kurzen Beispiel bereits deutlich. Für das komplette Set `vorname`, `nachname` usw. ist nur ein einziger Variablenname nötig. Statt für die beiden angelegten Adressen zwölf Variablen deklarieren zu müssen, reichen lediglich zwei. Außerdem besteht eine feste Zuordnung zwischen Vor- und Nachnamen derselben Adresse; sie können nicht versehentlich mit anderen Adressen vertauscht werden. Besonders interessant wird diese Zusammenlegung, wenn mit Arrays oder Parameterübergabe gearbeitet wird, doch dazu später mehr.

Für die Attribute können Sie jeden Datentyp verwenden, der dem Compiler zu diesem Zeitpunkt bekannt ist, d. h. also alle Standard-Datentypen und andere UDTs, die bereits deklariert wurden. Nicht möglich ist die Verwendung von UDTs, die erst später deklariert werden und die der Compiler daher zu diesem Zeitpunkt noch nicht kennt. Falls Sie so etwas benötigen, schafft das *forward referencing* Abhilfe; diese Methode werden wir aber erst in **??** behandeln.

Quelltext 7.2 zeigt die Einbindung eines bereits bekannten UDTs als Attribut-Datentyp. Dazu wird erst ein Datentyp für die Speicherung von Vor- und Nachnamen deklariert. Dieser kann anschließend im Adress-UDT verwendet werden.

Quelltext 7.2: Verschachtelte UDT-Struktur

```
' UDTs deklarieren
TYPE TName
  AS STRING vorname, nachname
END TYPE
TYPE TAdresse
  AS TName   name_
  AS STRING  strasse, ort
  AS INTEGER hausnummer, plz
END TYPE

' Zugriff auf die Elemente
DIM AS TAdresse adresse
adresse.hausnummer    = 32
adresse.name_.vorname = "Sonja"
PRINT adresse.name_.vorname
SLEEP
```

UDTs können beliebig ineinander verschachtelt werden, jedoch wird der Zugriff auf die Attribute durch die langen Bezeichnungsketten mühseliger. Den Namen in ein eigenes UDT auszulagern macht vor allem dann Sinn, wenn dieses UDT auch außerhalb des Adress-UDTs eine Bedeutung hat, z. B. weil auch Namen ohne zugehörige Adresse gespeichert werden müssen.

Hinweis:
NAME ist ein FreeBASIC-Schlüsselwort und kann nicht als Variablenname verwendet werden. Viele Schlüsselwörter, darunter auch **NAME**, können jedoch als Bezeichnung für ein Mitglied eines UDTs verwendet werden, da sie innerhalb der UDT-Deklaration keine eigenständige Bedeutung besitzen. Dennoch wurde in Quelltext 7.2 die Variante mit dem abschließenden Unterstrich gewählt, die in keiner Situation ein Schlüsselwort ist.

7.2. Mitgliederzugriff mit `WITH`

Ein einfacherer Zugriff auf die Mitglieder eines UDTs ist durch den **WITH**-Block möglich. Zusammen mit **WITH** wird der Name des UDTs angegeben, auf dessen Elemente zugegriffen werden soll. Beim Zugriff auf ein Mitglied dieses UDTs kann anschließend innerhalb des Blocks der UDT-Name weggelassen werden. Das Mitglied beginnt dann mit einem Punkt.

Quelltext 7.3: Vereinfachter UDT-Zugriff mit WITH

```
TYPE TAdresse
  AS STRING  vorname, nachname, strasse, ort
  AS INTEGER hausnummer, plz
END TYPE

DIM AS TAdresse adresse1, adresse2
WITH adresse1
  .strasse = "Einbahnstr."         ' Kurzform von adresse1.strasse
  adresse2.strasse = "Milchstr."   ' geht natuerlich immer noch
  .hausnummer = 39                 ' Kurzform von adresse1.hausnummer
END WITH
```

WITH-Blöcke können ineinander verschachtelt sein. Es gilt dann immer das UDT des innersten Blocks, in dem sich das Programm zur Zeit befindet. **END WITH** beendet den aktuellen **WITH**-Block.

Quelltext 7.4: Verschachteltes WITH

```
TYPE TAdresse
  AS STRING  vorname, nachname, strasse, ort
  AS INTEGER hausnummer, plz
END TYPE

DIM AS TAdresse adresse1, adresse2
WITH adresse1
  ' Kurzschreibweisen beziehen sich jetzt auf adresse1
  .strasse = "Einbahnstr."         ' Kurzform von adresse1.strasse
  WITH adresse2
    ' Kurzschreibweisen beziehen sich jetzt auf adresse2
    .strasse = "Milchstr."         ' Kurzform von adresse2.strasse
  END WITH
  ' Kurzschreibweisen beziehen sich wieder auf adresse1
  .hausnummer = 39                 ' Kurzform von adresse1.hausnummer
END WITH
```

7.3. Speicherverwaltung

Die Speicherplätze für die Attribute eines UDTs liegen direkt hintereinander (ein weiterer Vorteil von UDTs, weil sie sich dadurch oft komplett an einem Stück speichern und laden lassen[9]). Die Speicherstellen werden in der Regel „dicht aneinander“ gepackt, wobei jedoch „Füllstellen“ freigelassen werden können, wenn dadurch ein einfacherer Speicherzugriff möglich wird. Man spricht hier vom *Padding*.

[9] Das funktioniert nur, wenn alle Attribute eine feste Länge besitzen, also u. a. keine Strings variabler Länge eingesetzt werden.

Besteht ein UDT aus drei **BYTE**-Attributen, so benötigt es drei Byte Speicherplatz. Besteht es jedoch aus einem **BYTE** und einem **INTEGER**, dann wäre es unpraktisch, den **INTEGER**-Wert direkt hinter das **BYTE** zu hängen. Ein **INTEGER** hat ja den Vorteil, dass es direkt in einem Arbeitsschritt gelesen bzw. geschrieben werden kann. Das funktioniert aber nicht, wenn sein Speicherbereich über die Grenze einer **INTEGER**-Speicherstelle hinausragt. An dieser Stelle kommt das Padding ins Spiel: Der Speicherbereich wird automatisch so ausgeweitet, dass ein schneller Datenzugriff ermöglicht wird. Allerdings können Sie dieses Verhalten auch beeinflussen.

Hinweis:
Über Padding müssen Sie sich vorerst nur in drei Fällen Gedanken machen:
- Sie arbeiten mit sehr großen Datenmengen und müssen auf den Speicherverbrauch achten, oder
- Sie verwenden Datentypen aus anderen (Nicht-FreeBASIC-)Quellen, die ein anderes Padding verwenden als FreeBASIC, oder
- Sie wollen nicht nur programmieren, sondern auch verstehen, was im Hintergrund passiert.

Wenn mindestens einer der drei Fälle zutrifft, sollten Sie weiterlesen. Wenn nicht, können Sie diesen Abschnitt getrost überspringen, mit Kapitel 7.4 fortfahren und bei Bedarf auf diesen Abschnitt zurückkommen.

Das Padding-Verhalten soll in Quelltext 7.5 veranschaulicht werden. Wir nutzen dazu zwei Funktionen: Mit **SIZEOF()** lässt sich die Speichergröße eines Elements ausgeben. Man kann damit die Anzahl der Bytes ausgeben lassen, die von einer angegebenen Variablen, aber auch von Variablen eines angegebenen Datentyps belegt werden. Um das Offset, also die Speicherposition eines Attributs relativ zu seinem UDT zu ermitteln, dient **OFFSETOF()**. Das erste Attribut, das in einer UDT-Definition angegeben wird, hat immer das Offset `0`.

Achtung:
Bei Strings variabler Länge beträgt die von **SIZEOF()** ausgegebene Speichergröße je nach Architektur immer 12 Byte (32-Bit-Rechner) oder 24 Byte (64-Bit-Rechner). Das entspricht der Größe des Headers, in dem u. a. auch die Position und Länge des Stringinhalts gespeichert wird.

Quelltext 7.5: Standard-Padding bei UDTs

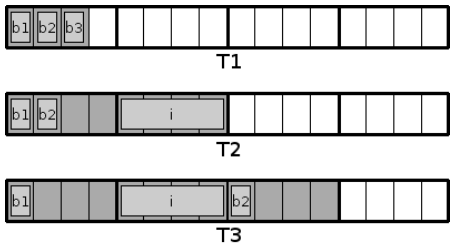
```
TYPE T1
  AS BYTE b1, b2, b3
END TYPE
TYPE T2
  AS BYTE b1, b2
  AS INTEGER i
END TYPE
TYPE T3
  AS BYTE b1
  AS INTEGER i
  AS BYTE b2
END TYPE

PRINT "UDT", "SIZEOF",   "Offset 1",       "Offset 2",       "Offset 3"
PRINT "T1", SIZEOF(T1), OFFSETOF(T1, b1), OFFSETOF(T1, b2), OFFSETOF(T1, b3)
PRINT "T2", SIZEOF(T2), OFFSETOF(T2, b1), OFFSETOF(T2, b2), OFFSETOF(T2, i)
PRINT "T3", SIZEOF(T3), OFFSETOF(T3, b1), OFFSETOF(T3, i),  OFFSETOF(T3, b2)
SLEEP
```

Ausgabe

```
UDT           SIZEOF        Offset 1      Offset 2      Offset 3
T1             3             0             1             2
T2             8             0             1             4
T3             12            0             4             8
```

Diese Ausgabe gilt für die 32-Bit-Version des Compilers. Wie die Speicherbelegung zustande kommt, lässt sich schematisch folgendermaßen darstellen:



Die grau gefärbten Bereiche geben den belegten Speicher an. Sobald ein größerer Datentyp verwendet wird, wie in diesem Beispiel ein (32-Bit-)Integer, findet ein Padding auf die Größe dieses Datentyps statt.[10] Das wird vor allem beim UDT `T3` deutlich: Auch

[10] Genauer gesagt wird das Standard-Padding durch den größten Datentyp festgelegt, beträgt aber höchstens 4 (unter 32-Bit x86 Linux/BSD) bzw. 8 (bei allen anderen Systemen).

der Platz hinter dem letzten einzelnen Byte wird auf vier Bytes aufgefüllt.

Das Standard-Paddingverhalten kann durch das Schlüsselwort **FIELD** verändert werden. Sie können das Padding dadurch allerdings nur verkleinern, nicht vergrößern.

Quelltext 7.6: Benutzerdefiniertes Padding

```
TYPE T4 FIELD=1
  AS BYTE b1
  AS INTEGER i
  AS BYTE b2
END TYPE
TYPE T5 FIELD=2
  AS BYTE b1
  AS INTEGER i
  AS BYTE b2
END TYPE
TYPE T6 FIELD=2
  AS BYTE b1, b2
  AS INTEGER i
END TYPE

PRINT "UDT", "SIZEOF",   "Offset 1",       "Offset 2",       "Offset 3"
PRINT "T4", SIZEOF(T4), OFFSETOF(T4, b1), OFFSETOF(T4, i),  OFFSETOF(T4, b2)
PRINT "T5", SIZEOF(T5), OFFSETOF(T5, b1), OFFSETOF(T5, i),  OFFSETOF(T5, b2)
PRINT "T6", SIZEOF(T6), OFFSETOF(T6, b1), OFFSETOF(T6, b2), OFFSETOF(T6, i)
SLEEP
```

Ausgabe

UDT	SIZEOF	Offset 1	Offset 2	Offset 3
T4	6	0	1	5
T5	8	0	2	6
T6	6	0	1	2

`T5` unterscheidet sich von `T4` durch das größere Padding. `T6` verwendet zwar dasselbe Padding wie `T5`, belegt aber trotzdem weniger Speicher, da die Attribute platzsparender angeordnet sind. Schematisch dargestellt sieht das so aus:

Durch eine geringere Ausdehnung wird Speicherplatz gespart, Sie sehen jedoch, dass das Integer nun nicht mehr eine vollständige Integer-Stelle belegt, sondern sich über eine der Grenzen erstreckt. Der Bonus bei der Zugriffgeschwindigkeit geht damit verloren. In der Regel wird **FIELD** nur verwendet, um eine Kompatibilität zu externen Bibliotheken herzustellen. Wenn dort die Speicherverwaltung anders abläuft als unter FreeBASIC, kommt es unweigerlich zu Problemen. Ansonsten lohnt sich der Einsatz von Padding in der Regel nicht – sinnvoller ist es, die Attribute so in der Deklaration anzuordnen, dass die durch das Padding entstehenden Lücken nicht unnötig groß werden.

Unterschiede zu QuickBASIC:
In QuickBASIC existiert kein Padding. Die Feldbreite beträgt dort immer 1 Byte. Das muss vor allem dann beachtet werden, wenn Sie in FreeBASIC Daten einlesen wollen, die mit QuickBASIC erstellt wurden.

7.4. Bitfelder

Manchmal benötigt man deutlich kleinere Datentypen als ein **BYTE**. In einem Formular mit mehreren Kontrollkästchen werden eine Reihe von Variablen benötigt, die lediglich den Zustand „angeklickt“ oder „nicht angeklickt“ speichern müssen. Es reicht also jeweils ein Bit pro Kontrollkästchen.

So etwas lässt sich über Bitfelder lösen. Dazu wird eine Ganzzahl-Variable in eine Anzahl von einzelnen Bits aufgeteilt. Wir erinnern uns: Jedes Bit kann zwei Zustände annehmen (1 oder 0). Mit zwei Bits sind dann $2^2 = 4$ Zustände möglich, mit drei Bits $2^3 = 8$ usw. Eine solche Aufsplittung ist nur innerhalb eines UDTs erlaubt. Dazu wird hinter dem Attributnamen ein Doppelpunkt geschrieben, gefolgt von der gewünschten Bit-Zahl.

Quelltext 7.7: Deklaration von Bitfeldern

```
TYPE TFormular
  AS INTEGER button1 : 1
  button2 : 1 AS INTEGER  ' alternative Schreibweise
  AS INTEGER radio   : 3
END TYPE
```

Hier werden drei Attribute deklariert, von denen zwei ein Bit lang sind und eines drei Bit lang. Das bedeutet: `button1` und `button2` können jeweils nur zwei verschiedene Werte annehmen (0 oder 1), `radio` dagegen acht (von 0 bis 7). Wenn Sie einem Attribut

einen zu großen Wert zuweisen wollen, wird dieser automatisch „zurechtgestutzt“. Wenn Sie also z. B. in `radio` den Wert `10` speichern wollen (binär 1010), werden nur die hinteren drei Bit verwendet und der Rest verworfen – gespeichert wird damit der Wert `2`. Abgesehen davon verhalten sich Bitfelder jedoch ganz genauso wie andere Attribute.

Es sollte noch ergänzt werden, dass die gewählte Bitzahl nicht größer sein kann als die Bitzahl des zugrunde liegenden Datentyps. Es können z. B. keine 9 Bit eines **BYTE** verwendet werden oder 33 Bit eines 32-Bit-Integers (sehr wohl aber eines 64-Bit-Integers). Außerdem wird **LONGINT** nur in der 64-Bit-Version des Compilers unterstützt.

Des Weiteren wird ein UDT immer vollständige Bytes als Speicherplatz belegen. Ein UDT mit einem einzigen Bitfeld-Attribut wird keine Speicherersparnis mit sich bringen. Selbst wenn das Attribut nur ein einziges Bit belegt, wird die Größe des UDTs durch die Größe des gewählten Datentyps festgelegt. Eine Platzersparnis tritt erst ein, wenn der Speicherplatz des gewählten Datentyps auf mehrere Attribute aufgeteilt wird.

7.5. UNIONs

Eine **UNION** ist ein UDT, dessen Elemente sich dieselbe Speicheradresse teilen. Abhängig vom Einsatzbereich kann der Inhalt der Speicherstelle auf verschiedene Arten interpretiert werden, eine Änderung des Inhalts wirkt sich aber natürlich auch auf die anderen Elemente aus.

Quelltext 7.8: Einfache UNION-Verwendung

```
UNION testunion
  AS BYTE  byteVar
  AS SHORT shortVar
END UNION

DIM AS testunion wert

wert.shortVar = 26
PRINT wert.shortVar, wert.byteVar

wert.shortVar = 260
PRINT wert.shortVar, wert.byteVar

wert.byteVar = 3
PRINT wert.shortVar, wert.byteVar
SLEEP
```

Ausgabe

```
26              26
260             4
259             3
```

`wert.byteVar` interpretiert den Speicherinhalt nur als **BYTE**, wodurch sich der ausgegebene Wert `4` ergibt. In diesem Fall hätte natürlich eine einfache Typumwandlung von **SHORT** zu **BYTE** ausgereicht. In den Zeilen 14 und 15 sieht man aber, dass eine Änderung von `wert.byteVAR` nur das erste Byte von `wert.shortVar` verändert und das andere Byte unverändert lässt.

- `26` besitzt den Binärwert `00000000 00011010`. Das höherwertige Byte ist `0`, weshalb `wert.byteVar` und `wert.shortVar` denselben Wert ausgeben.
- `260` besitzt den Binärwert `00000001 00000100`. Das niedere Byte besitzt den Wert `4`, der von `wert.byteVar` ausgegeben wird.
- Bei der dritten Änderung wird in das niedere Byte der Dezimalwert `3` bzw. der Binärwert `00000011` gelegt. Im Speicher liegt nun der Binärwert `00000001 00000011` bzw. als Dezimalwert `259`. Etwas genauer werden wir auf das Binärsystem in Kapitel 10.2.3 eingehen.

Gern verwendet wird **UNION** innerhalb eines UDTs, das für mehrere Zwecke eingesetzt werden soll. Elemente, die nicht gleichzeitig zum Einsatz kommen, können sich eine Speicherstelle teilen, da sie sich ja nicht gegenseitig in die Quere kommen können. FreeBASIC selbst nutzt das bei der Deklaration von Grafik-Headern. Die alten QuickBASIC-Header sind folgendermaßen aufgebaut:

```
TYPE _OLD_HEADER FIELD = 1
  bpp     :  3 AS USHORT
  width   : 13 AS USHORT
  height       AS USHORT
END TYPE
```

Für die Ansprüche von FreeBASIC ist eine Beschränkung auf 8191 Pixel Bildbreite nicht unbedingt wünschenswert. Deswegen wurde ein neues Header-Format gewählt, das zusätzlich noch Platz für weitere Informationen bietet. Da aber auch die im alten QuickBASIC-Format gespeicherten Bilder noch unterstützt werden sollen, wurde der neue Header folgendermaßen definiert:

Quelltext 7.9: UNION innerhalb einer UDT-Deklaration

```
TYPE Image FIELD = 1
  UNION
    old               AS _OLD_HEADER
    type              AS ULONG
  END UNION
  bpp                 AS LONG
  width               AS ULONG
  height              AS ULONG
  pitch               AS ULONG
  _reserved(1 to 12) AS UBYTE
END TYPE
```

`type` gibt die Versionsnummer des Headers zurück; beim neuen Headerformat ist das immer `7`. Der alte Header kann dagegen niemals den Wert `7` annehmen. Abhängig von `type` kann FreeBASIC also entscheiden, ob die weiteren Daten nach dem alten oder neuen Header-Format interpretiert werden müssen.

Eine innerhalb von **`TYPE`**-Deklarationen verwendete **`UNION`** darf keinen eigenen Bezeichner erhalten. Stattdessen werden die **`UNION`**-Attribute direkt über den Bezeichner des UDTs angesprochen, also z. B. im oben stehenden Grafik-Header folgendermaßen:

```
DIM AS Image meinBild
IF meinBild.type <> 7 THEN PRINT "Der alte Header wird verwendet."
```

Ein weiteres Beispiel direkt aus dem FreeBASIC-internen Fundus stellt das UDT `Event` dar, das von der Funktion **`SCREENEVENT()`** verwendet wird. Wir werden uns mit diesem Befehl in **??** genauer beschäftigen. Kurz gesagt geht es um die Abfrage bestimmter Ereignisse wie Mausbewegung, Tastendruck, Betreten oder Verlassen des Grafikfensters usw. Dabei liefern die verschiedenen Ereignisse teils unterschiedliche Informationen – z. B. ist es für die Mausbewegung wichtig, wohin die Maus bewegt wurde, während ein Tastendruck die gedrückte Taste zurückgeben soll. Da weder die Maus einen Tastendruck zurückgibt noch die Tastatur eine Mausposition, können sich beide Informationen den Speicherbereich teilen.

7.6. Fragen zum Kapitel

Die folgenden Programmieraufgaben bauen aufeinander auf. Sie können also mit dem Ergebnis aus Aufgabe 1 in Aufgabe 2 weiterarbeiten und mit diesem Ergebnis in Aufgabe 3.

1. Als Einzelhandelsunternehmen wollen Sie Ihre Produkte katalogisieren. Erstellen Sie ein UDT, in dem Sie die Informationen eines Produktes speichern können:

Name, Einkaufspreis, Verkaufspreis und vorhandene Stückzahl. Wählen Sie jeweils passende Datentypen.

2. Lassen Sie den Benutzer über **`INPUT`** für zwei Produkte die Attributwerte eingeben. Wenn die Eingabe unsinnig ist, soll eine Warnung ausgegeben werden – beispielsweise soll der Verkaufspreis nicht negativ sein und nicht unterhalb des Einkaufspreises liegen.

3. Geben Sie für die beiden eingegebenen Produkte jeweils den Namen und den zu erwartenden Gewinn pro verkauftes Stück aus (der Gewinn berechnet sich aus dem Unterschied zwischen Verkaufs- und Einkaufspreis).

8. Datenfelder (Arrays)

Ein Array ist nichts anderes als eine Gruppe gleichartiger Variablen, die im Speicher direkt hintereinander liegen. Sie werden nicht über viele verschiedenen Namen angesprochen, sondern besitzen einen gemeinsamen Namen und unterscheiden sich durch einen Index.

8.1. Deklaration und Zugriff

Stellen Sie sich z. B. vor, Sie haben eine Liste mit den Nachnamen Ihrer Kunden. Jeden Namen in einer eigenen Variablen zu speichern, ist aus mehreren Gründen unpraktisch. Zum einen ist es umständlich, die Variablen zu verwalten, zum anderen ist das Konzept unflexibel, wenn neue Kunden hinzukommen. Stattdessen bietet sich ein Array an, das über einen einzigen Bezeichner angesprochen werden kann.

```
' STRING-Datenfeld fuer 5 Nachnamen anlegen
DIM AS STRING nachname(1 TO 5)
```

Die Syntax ist weitgehend aus den vorigen Kapiteln bekannt; sie ist fast identisch mit der Variablendeklaration. Neu ist die Angabe innerhalb der Klammern. Dadurch werden Speicherplätze für fünf String-Variablen angelegt, die von 1 bis 5 durchnummeriert sind. Ebenfalls fünf Speicherplätze, nun aber von 4 bis 8 nummeriert, würden durch folgende Zeile festgelegt werden:

```
DIM AS STRING nachname(4 TO 8)
```

In beiden Fällen spricht man von der *Array-Länge* 5. Die einzelnen Werte können nun über ihren Index angesprochen werden:

Quelltext 8.1: Array-Deklaration

```
' STRING-Datenfeld fuer 5 Nachnamen anlegen
DIM AS STRING nachname(1 TO 5)
' Array-Werte speichern
nachname(1) = "Huber"
nachname(2) = "Schmid"
' Array-Wert abrufen
PRINT nachname(2)
SLEEP
```

Zu Beginn, in Zeile 2, werden alle Speicherstellen auf einen leeren Wert gesetzt, also bei Strings auf einen Leerstring, bei Zahlendatentypen auf den Wert `0`. Jeder Array-Stelle in einer neuen Zeile einen Wert zuzuweisen kann recht aufwendig werden. Stattdessen gibt es, analog zur Variablendeklaration, eine Kurzschreibweise, mit der direkt bei der Deklaration auch die Wertzuweisung erfolgt.

Quelltext 8.2: Array-Deklaration mit direkter Wertzuweisung

```
' STRING-Datenfeld fuer 5 Nachnamen anlegen
DIM AS STRING nachname(1 TO 5) = {"Huber", "Schmid", "Mayr", "Mueller", "Schuster"}
' Array-Wert abrufen
PRINT nachname(2)
SLEEP
```

Ausgabe

```
Schmid
```

Die Liste muss mit geschweiften Klammern umschlossen werden und genauso viele Elemente enthalten wie das Array. Die einzelnen Werte werden durch Komma voneinander getrennt. Natürlich müssen die Elemente auch alle den richtigen Datentyp besitzen.

Achtung:
Wird beim Arrayzugriff ein Index angegeben, der außerhalb der bei der Deklaration festgelegten Grenzen liegt, dann greift das Programm auf einen Speicherbereich zu, der nicht zum Array gehört. Dies führt zu unvorhersagbaren Ergebnissen und muss daher auf jeden Fall vermieden werden. Mehr dazu erfahren Sie in Kapitel 9.3.
Wenn Sie mit der Compileroption `-exx` compilieren, bricht das Programm mit einer Fehlermeldung ab, sobald auf einen Arrayindex außerhalb der erlaubten Grenzen zugegriffen wird. Dadurch lassen sich falsche Speicherzugriffe deutlich leichter aufspüren.

8.2. Mehrdimensionale Arrays

Während die bisher behandelten eindimensionalen Arrays in etwa einer aneinandergereihten Folge von Datenwerten entsprechen, kann man sich ein zweidimensionales Array ähnlich wie ein Schachbrett vorstellen. Jedes Feld kann eindeutig über die Reihe und

Spalte angesprochen werden, z. B. als `a4`. Der Unterschied besteht nur darin, dass in FreeBASIC keine Buchstaben-Indizes verwendet werden. Das Feld in der ersten Spalte der vierten Reihe könnte also z. B. über `feld(1, 4)` angesprochen werden.

Auch die Initialisierung, also eine Wertzuweisung direkt bei der Deklaration, ist möglich. Dazu müssen die einzelnen Dimensionen extra durch geschweifte Klammern eingeschlossen werden.

Quelltext 8.3: Mehrdimensionales Array

```
' 2x3-Integerfeld
DIM AS INTEGER array(1 TO 2, 1 TO 3) = { { 1, 2, 3 }, _
                                        { 4, 5, 6 } }
PRINT array(2, 2)
SLEEP
```

Ausgabe

```
 5
```

Wie Sie sehen, besteht die Wertzuweisungen aus zwei Blöcken mit je drei Einträgen, entsprechend den Länge 2 in der ersten Dimension und der Länge 3 in der zweiten Dimension.
Arrays sind nicht auf zwei Dimensionen begrenzt. Es können bis zu acht Dimensionen genutzt werden.

Für Fortgeschrittene:
Bei mehrdimensionalen Arrays folgen im Speicher die Werte aufeinander, deren erster Index gleich ist. FreeBASIC unterscheidet sich hier von QuickBASIC, welches die Werte aufeinander folgen lässt, deren letzter Index gleich ist.

8.3. Dynamische Arrays

FreeBASIC kennt statische und dynamische Arrays. Bei einem statischen Array werden die Anzahl der Dimensionen und die Länge einmal mit **DIM** festgelegt und können dann innerhalb seines Gültigkeitsbereichs nicht mehr verändert werden. Ein dynamisches Array dagegen erlaubt eine spätere Größenänderung. Das ist vor allem dann praktisch, wenn während der Laufzeit des Programmes immer wieder neue Werte zum Array hinzugefügt werden müssen und daher zu Beginn die benötigte Länge noch nicht feststeht.

8.3.1. Deklaration und Redimensionierung

Zur Deklaration eines dynamischen Arrays gibt es zwei Möglichkeiten: zum einen wie gewohnt mit **DIM**, jedoch ohne Angabe der Arraygrenzen (zur Unterscheidung von normalen Variablen müssen jedoch die Klammern angegeben werden). Zum anderen können Sie den Befehl **REDIM** verwenden. **REDIM** dient auch zur späteren Änderung der Arraygrenzen. In der ersten Version mit **DIM** ist das Array zunächst noch undimensioniert, muss also später vor der Verwendung noch mit **REDIM** auf seine Dimensionen festgelegt werden. Wird gleich zu Beginn **REDIM** verwendet, dann können sofort die gewünschten Array-Grenzen angegeben werden.

Hinweis:
Wir unterscheiden hier die *Deklaration* und die *Dimensionierung* (= Zuweisung der Dimensionen) eines Arrays. Während ein statisches Array bei der Deklaration auch gleich dimensioniert wird und diese Dimensionierung nicht mehr verliert, kann bei einem dynamischen Array Deklaration und Dimensionierung getrennt voneinander stattfinden, und das Array kann später vergrößert oder verkleinert werden.

Dynamische Arrays können nicht gleich bei der Deklaration initialisiert werden; eine Wertzuweisung ist erst nach der Deklaration möglich.

Quelltext 8.4: Dynamische Arrays anlegen

```
' dynamische Arrays deklarieren
DIM AS INTEGER array1()           ' dynamisches Array mit DIM
REDIM AS INTEGER array2(1 TO 5)   ' dynamisches Array mit REDIM

' dynamische Arrays neu dimensionieren
REDIM array1(1 TO 10), array2(1 TO 10)
```

Auch bei der ersten Dimensionierung mit **REDIM** können die Grenzen zunächst weggelassen werden; in diesem Fall muss das Array, genauso wie bei der dynamischen Version von **DIM**, erst noch festgelegt werden, bevor auf das Array zugegriffen werden kann.

Bei einer Redimensionierung eines bereits deklarierten dynamischen Arrays braucht kein Datentyp mehr angegeben werden, da dieser bereits feststeht. Wird er trotzdem angegeben, *muss* er mit dem ursprünglichen Datentyp übereinstimmen. Auch die Anzahl der Dimensionen kann, wenn sie einmal festgelegt wurde, nicht mehr verändert werden – veränderbar sind nur die Array-Grenzen innerhalb der Dimensionen.

8.3.2. Werte-Erhalt beim Redimensionieren

Bei der Verwendung von **REDIM** werden die Array-Werte, genauso wie bei **DIM**, neu initialisiert, d. h. Zahlenwerte werden auf 0 gesetzt und Zeichenketten auf einen Leerstring. Sie können also davon ausgehen, dass Sie nach jedem **REDIM** wieder ein „frisches“ Array vor sich haben.

Wollen Sie allerdings die alten Werte behalten, benötigen Sie das Schlüsselwort **PRESERVE**. Nehmen wir an, Sie wollen eine Reihe von Messdaten in einem Array speichern und, falls der Platz nicht ausreicht, das Array vergrößern. In diesem Fall wäre ein Zurücksetzen der Array-Werte auf 0 nicht wünschenswert. **PRESERVE** weist das Programm an, bisherige Daten zu erhalten. Wird das Array vergrößert, werden an sein Ende leere Elemente angefügt (also mit dem Wert 0 oder Leerstring). Bei einer Verkleinerung werden am hinteren Ende Daten abgeschnitten und gehen damit verloren.

Quelltext 8.5: REDIM mit und ohne PRESERVE

```
REDIM AS INTEGER arr(1 TO 4)
arr(1) = 1 : arr(2) = 2 : arr(3) = 3 : arr(4) = 4
PRINT "Startbelegung des Arrays arr(1 TO 4) mit den Werten:"
PRINT arr(1), arr(2), arr(3), arr(4)
' Array vergroessern
REDIM PRESERVE arr(1 TO 6)
PRINT "REDIM aus arr(1 TO 6); Ausgabe der Werte arr(3), arr(4) und arr(5):"
PRINT arr(3), arr(4), arr(5)
' Grenzen verschieben
REDIM PRESERVE arr(3 TO 9)
PRINT "REDIM auf arr(3 TO 9); Ausgabe der Werte arr(3), arr(4) und arr(5):"
PRINT arr(3), arr(4), arr(5)
' Neudimensionierung ohne PRESERVE
REDIM arr(3 TO 9)
PRINT "REDIM ohne PRESERVE; Ausgabe der Werte arr(3), arr(4) und arr(5):"
PRINT arr(3), arr(4), arr(5)
SLEEP
```

Ausgabe

```
Startbelegung des Arrays arr(1 TO 4) mit den Werten:
 1             2             3             4
REDIM aus arr(1 TO 6); Ausgabe der Werte arr(3), arr(4) und arr(5):
 3             4             0
REDIM auf arr(3 TO 9); Ausgabe der Werte arr(3), arr(4) und arr(5):
 1             2             3
REDIM ohne PRESERVE; Ausgabe der Werte arr(3), arr(4) und arr(5):
 0             0             0
```

Beachten Sie, dass beim Verschieben der Grenzen die Werte nicht ebenfalls verschoben werden. Zuvor hatte der dritte Eintrag den Wert `3`. Nach der Anweisung `REDIM PRESERVE arr(3 TO 9)` ist jedoch `arr(3)` der erste Eintrag und `arr(5)` der dritte – dementsprechend kommt es bei der Ausgabe zu den Werten, die Sie oben sehen können.

Achtung:
Das Schlüsselwort **`PRESERVE`** kann nur bei eindimensionalen Arrays ohne Probleme verwendet werden, bei mehrdimensionalen Arrays bleibt wegen der Array-Speicherverwaltung nur der erste Index erhalten.

8.4. Weitere Befehle

8.4.1. Grenzen ermitteln: `LBOUND()` und `UBOUND()`

Mit den Funktionen **`LBOUND()`** (*Lower BOUND*) und **`UBOUND()`** (*Upper BOUND*) kann die untere bzw. obere Grenze eines Arrays ausgelesen werden. Der Name des Arrays, dessen Grenzen bestimmt werden sollen, wird als Argument übergeben; dabei fallen die zum Array gehörenden Klammern weg. Bei mehrdimensionalen Arrays kann die gewünschte Dimension als zweites Argument übergeben werden.

Interessant ist auch die Rückgabe, die erfolgt, wenn als zweiter Parameter der Wert `0` angegeben wird. Doch dazu sehen wir uns erst einmal den Quellcode an:

Quelltext 8.6: Verwendung von LBOUND und UBOUND

```
DIM AS INTEGER a(1 TO 5), b(0 TO 9, 4 TO 7)
PRINT "Grenzen von a():", LBOUND(a), UBOUND(a)
PRINT "1. Dim. von b():", LBOUND(b, 1), UBOUND(b, 1)
PRINT "2. Dim. von b():", LBOUND(b, 2), UBOUND(b, 2)
PRINT "LBOUND(b)    = ";  LBOUND(b), "UBOUND(b)    = "; UBOUND(b)
PRINT
PRINT "LBOUND(a, 0) = "; LBOUND(a, 0), "UBOUND(a, 0) = "; UBOUND(a, 0)
PRINT "LBOUND(b, 0) = "; LBOUND(b, 0), "UBOUND(b, 0) = "; UBOUND(b, 0)
SLEEP
```

Ausgabe

```
Grenzen von a():              1              5
1. Dim. von b():              0              9
2. Dim. von b():              4              7
LBOUND(b)     =  0           UBOUND(b)     =  9

LBOUND(a, 0) =  1            UBOUND(a, 0) =  1
LBOUND(b, 0) =  1            UBOUND(b, 0) =  2
```

Die ersten drei Ausgabezeilen sind selbsterklärend. In der vierten Zeile fällt auf, dass **LBOUND()** und **UBOUND()** ohne zweiten Parameter offenbar die Grenzen in der ersten Dimension zurückgeben – ein ausgelassener Parameter wird also als Wert `1` behandelt. Dieses Verhalten ist auch sinnvoll, da bei eindimensionalen Arrays logischerweise die 1. Dimension interessant ist.

Eine `0` als zweiter Parameter macht dagegen etwas ganz anderes: hier wird die Anzahl der Dimensionen zurückgegeben. `LBOUND(array, 0)` gibt immer `1` zurück, weil das immer „die untere Dimensionenzahl“ ist. `UBOUND(array, 0)` liefert bei eindimensionalen Arrays den Wert `1`, bei zweidimensionalen den Wert `2` usw. Ist das Array noch nicht dimensioniert (beim Deklarieren dynamischer Arrays durch **DIM** ohne Angabe der Grenzen), dann gibt `UBOUND(array, 0)` den Wert `0` zurück.

Wenn Sie die Länge einer Dimension abfragen, die überhaupt nicht existiert (wenn Sie also z. B. `LBOUND(array, 4)` abfragen, obwohl `array()` weniger als vier Dimensionen besitzt), gibt **LBOUND()** `0` und **UBOUND()** `-1` zurück. Auch dieses Verhalten kann dazu genutzt werden, um zu überprüfen, ob ein Array bereits dimensioniert ist.

8.4.2. Nur die Dimensionenzahl festlegen: ANY

Bei dynamischen Arrays kann es vorkommen, dass Sie zwar noch keine konkrete Dimensionierung vornehmen können, aber bereits festschreiben wollen, *wie viele* Dimensionen es geben soll. Geben Sie dann statt der Grenzen einfach das Schlüsselwort **ANY** an. Bei mehrdimensionalen Arrays ist allerdings keine Mischung zwischen **ANY** und Arraygrenzen erlaubt – entweder wird für jede Dimension **ANY** angegeben oder für keine.

```
DIM   AS STRING array1(ANY, ANY)      ' dynamisches zweidimensionales Array
REDIM AS STRING array2(ANY, ANY)      ' alternative Schreibweise zur 1. Zeile
REDIM AS STRING array3(ANY, ANY, ANY) ' dynamisches dreidimensionales Array
' REDIM AS STRING array4(1 TO 4, ANY) ' nicht erlaubt!
```

Beachten Sie, dass in allen drei Fällen die Arrays noch nicht dimensioniert wurden.

`UBOUND(array1, 0)` wird daher `0` ausgeben. Die Deklarationen verhindern lediglich, dass später eine Dimensionierung mit einer anderen als der vorgesehenen Dimensionenzahl stattfindet.

8.4.3. Implizite Grenzen

In der Regel werden von den Programmierern Arrays bevorzugt, die mit dem Index `0` beginnen. Daher kann bei der Dimensionierung die Angabe des Startindex auch entfallen, wenn `0` als Startindex verwendet werden soll.

```
DIM AS INTEGER array(5)
' ist identisch mit: DIM AS INTEGER array(0 TO 5)
```

Auf diese Weise wird ein Array mit sechs Elementen erzeugt. Auch die Angabe der oberen Grenze kann in einem besonderen Fall offen gelassen werden, nämlich wenn bei der Dimensionierung gleichzeitig Werte zugewiesen werden. Die obere Grenze wird dann durch drei Punkte, auch *Ellipsis* (*Auslassung*) genannt, ersetzt. Die Ellipsis verhindert, dass bei einer Quelltextänderung durch Hinzufügen weiterer Initialwerte zwei verschiedene Stellen angepasst werden müssen.

Quelltext 8.7: Implizite obere Grenze bei Arrays

```
DIM AS INTEGER a(0 TO ...) = { 1, 2, 3, 4, 5 }
DIM AS INTEGER b(...)      = { 1, 2, 3, 4, 5 }
```

Beide Codezeilen definieren ein Array mit unterer Grenze `0` und oberer Grenze `4`.

Die Ellipsis kann auch bei mehrdimensionalen Arrays angewendet werden; jede obere Grenze kann durch drei Punkte ersetzt werden. Wichtig ist nur, dass bei der Dimensionierung auch gleich die Wertzuweisung erfolgt, da ja die Anzahl der Werte entscheidend für die Länge des Arrays ist. Außerdem funktioniert das Vorgehen nur bei statischen Arrays – einem dynamischen Array können, wie oben bereits erwähnt, bei der Dimensionierung keine Startwerte übergeben werden.

Unterschiede zu QuickBASIC:
In QuickBASIC kann mit **`OPTION BASE`** die standardmäßig verwendete untere Grenze zwischen `0` und `1` umgestellt werden. In FreeBASIC steht **`OPTION BASE`** nicht zur Verfügung. Wenn Sie eine andere untere Grenze als `0` verwenden wollen, müssen Sie sie explizit angeben.

8.4.4. Löschen mit ERASE

Arrays können auch wieder gelöscht werden, wobei unter dem Löschen eines statischen Arrays etwas anderes verstanden wird als unter dem Löschen eines dynamischen Arrays. Mit **ERASE** werden die Werte eines statischen Arrays auf den leeren Wert zurückgesetzt (also auf 0 bei Zahlenwerten und auf einen Leerstring bei Zeichenketten). Ein dynamisches Array wird dagegen tatsächlich aus dem Speicher gelöscht und anschließend als uninitialisiertes Array behandelt. Mit **LBOUND()** und **UBOUND()** lässt sich das leicht veranschaulichen.

Auch bei **ERASE** werden keine Array-Klammern angegeben. Um mehrere Arrays gleichzeitig zu löschen, listen Sie diese durch Komma getrennt hintereinander auf.

Quelltext 8.8: Arrays löschen

```
DIM   AS INTEGER a(9)  ' statisches  Array a
REDIM AS INTEGER b(9)  ' dynamisches Array b
a(0) = 1338            ' Testwert zuweisen

PRINT "vor dem ERASE:"
PRINT "Grenzen von a:", LBOUND(a), UBOUND(a)
PRINT "Grenzen von b:", LBOUND(b), UBOUND(b)
PRINT "a(0) besitzt den Wert"; a(0)
PRINT
ERASE a, b
PRINT "nach dem ERASE:"
PRINT "Grenzen von a:", LBOUND(a), UBOUND(a)
PRINT "Grenzen von b:", LBOUND(b), UBOUND(b)
PRINT "a(0) besitzt den Wert"; a(0)
SLEEP
```

Ausgabe

```
vor dem ERASE:
Grenzen von a:                0             9
Grenzen von b:                0             9
a(0) besitzt den Wert 1338

nach dem ERASE:
Grenzen von a:                0             9
Grenzen von b:                0            -1
a(0) besitzt den Wert 0
```

Die Länge des statischen Arrays bleibt erhalten, es ist also nach wie vor im Speicher vorhanden, nur seine Werte werden zurückgesetzt. Anhand der Ausgabe zum zweiten Array können Sie dagegen sehen, dass hier seine Dimensionierung nicht mehr existiert

und das Array gelöscht wurde. Nichtsdestotrotz – wenn Sie das Array neu dimensionieren wollen, muss es sowohl im Datentyp als auch in der Anzahl der Dimensionen mit der ursprünglichen Deklaration übereinstimmen.

Achtung:
Wird ein statisches Array als Parameter an eine Prozedur übergeben, so wird es dort als dynamisches Array angesehen. Die Verwendung von **`ERASE`** führt dann zu einem Speicherzugriffsfehler.

8.5. Fragen zum Kapitel

1. Wie wird ein statisches Array deklariert, und welche Unterschiede bestehen zur Deklaration einer Variablen?
2. Wie wird ein dynamisches Array deklariert?
3. Was sind die Unterschiede zwischen statischen und dynamischen Arrays?
4. Wie viele Dimensionen kann ein Array maximal haben?
5. Ein dynamisches Array wurde im Laufe des Programmes mit Werten gefüllt und soll nun vergrößert werden. Worauf ist zu achten?
6. Wie kann bei einem dynamischen Array festgestellt werden, ob es dimensioniert wurde?

9. Pointer (Zeiger)

Der Umgang mit Pointern ist nicht besonders anfängerfreundlich – bei falscher Verwendung können Sie einen Programmabsturz oder Schlimmeres verursachen. Es ist also äußerste Vorsicht geboten, und Sie sollten nur dann eigene Experimente mit Pointern anstellen, wenn Sie wissen, was Sie tun.

Dieses Kapitel ist sehr kurz und soll Ihnen nur einen schnellen Einblick in den Umgang mit Pointern geben, da in späteren Artikeln grundlegende Kenntnisse über Pointern hilfreich sind.

9.1. Speicheradresse ermitteln

Variablen sind, wie wir uns erinnern, Speicherplätze, in denen Werte „aufbewahrt“ werden können. Sie besitzen einen Variablennamen, über den sie angesprochen werden können, und einen Wert, der im Speicherplatz hinterlegt wurde. Nicht zuletzt hat die Variable aber auch einen Platz im Speicherbereich des Programmes, also eine Adresse, an der die Variable gefunden werden kann. Das Anlegen einer Variablen ist eigentlich nicht ganz so trivial, wie es bisher den Eindruck hatte – zunächst muss ein ausreichend großer freier Speicherbereich reserviert werden, da ja nicht etwa zwei verschiedene Variablen an dieselbe Stelle schreiben und damit den alten Wert der anderen Variable zerstören sollen. Wenn die Variable nicht mehr benötigt wird (spätestens am Ende des Programmes, oft aber schon wesentlich früher) ist wieder eine Freigabe des Speicherbereichs nötig, damit der Speicherbedarf nicht ständig anwächst. Wenn Sie **DIM** verwenden, kümmert sich zum Glück der Compiler um die korrekte Reservierung, Verwaltung und Freigabe des Speicherbereichs.

Dennoch – jede Variable besitzt eine Adresse, die den ihr zugeordneten Speicherbereich angibt. Diese Adresse lässt sich über einen *Zeiger*, oder auf englisch *Pointer*, ansprechen. Dieser Pointer wird ausgegeben, wenn vor dem Variablennamen ein @ gesetzt wird. Alternativ dazu lässt sich der Pointer auch mit **VARPTR()** ermitteln.

```
DIM AS INTEGER x
PRINT "x befindet sich an der Speicherstelle "; @x
' PRINT "x befindet sich an der Speicherstelle "; VARPTR(x)   ' Alternative
SLEEP
```

Die Variable wird bei jedem Start des Programmes an einer anderen Stelle abgelegt. Bei der Adresse handelt es sich, je nach Betriebssystem, um einen 32-Bit- oder 64-Bit-Wert.

9.2. Pointer deklarieren

Es können auch Variablen direkt als Pointer deklariert werden, was bedeutet, dass sie eine Adresse speichern anstatt eines „normalen“ Wertes. Pointer-Variablen besitzen denselben Wertebereich und Speicherplatzbedarf wie eine **UINTEGER**-Variable (also 32-Bit in einem 32-Bit-System und 64-Bit in einem 64-Bit-System). Insofern könnten Pointer wie **UINTEGER** behandelt werden. FreeBASIC unterscheidet die beiden Datentypen jedoch intern voneinander und gibt eine Warnung aus, wenn sie im Programm nicht säuberlich voneinander getrennt werden, da ein falscher Pointerzugriff zu erheblichen Problemen führen kann.

Eine Pointer-Variable wird wie eine normale Variable deklariert, wobei auf den Variablentypen das Schlüsselwort **POINTER** oder häufiger (da kürzer) **PTR** folgt. **POINTER** und **PTR** sind in diesem Zusammenhang gleichbedeutend, und es macht keinen Unterschied, welches der beiden Schlüsselwörter Sie verwenden.

```
DIM AS SINGLE PTR x
DIM y AS INTEGER PTR
```

Um auf den Wert zuzugreifen, der an einer bestimmten Adresse hinterlegt ist, wird vor die Pointer-Variable ein Stern * gestellt. Nach der Deklaration besitzt die Variable standardmäßig den Wert `0` (in diesem Fall spricht man dann von einem *Nullpointer*). Ein lesender oder schreibender Zugriff auf diese Adresse würde den Laufzeitfehler `Segmentation fault` [11] auslösen. Man kann der Pointer-Variablen jedoch zuvor die Adresse einer anderen Variablen zuweisen.

[11] Unter Windows war ein `Segmentation fault` früher unter dem Namen „Allgemeine Schutzverletzung“ bekannt; bei aktuellen Versionen erscheint in diesem Fall eine Meldung wie „<Programmname> funktioniert nicht mehr“.

Quelltext 9.1: Pointerzugriff

```
DIM AS INTEGER x = 5       ' normaler INTEGER-Wert
DIM AS INTEGER PTR p       ' Pointer auf einen INTEGER-Wert

' Zuweisung einer Adresse
p = @x
PRINT "x liegt bei der Adresse "; p;
PRINT " und besitzt den Wert"; *p

' Variable x ueber ihren Pointer veraendern
*p = 12
PRINT "x besitzt nun den Wert "; x
SLEEP
```

Pointer werden vor allem dann benötigt, wenn eigene Speicherbereiche verwaltet werden sollen, etwa ein Grafikspeicher. In diesem Fall muss der Speicher jedoch selbst reserviert und auch wieder freigegeben werden. Auch bei der Übergabe von Speicherbereichen an externe Bibliotheken werden gern Pointer verwendet. Das beliebteste Austauschformat für Zeichenketten ist der **ZSTRING PTR**, weil für das Speicherformat nur bekannt sein muss, dass die Zeichenkette mit einem Nullbyte endet.

9.3. Speicherverwaltung bei Arrays

Bei einem Array werden, wie in Kapitel 8 erwähnt, die Einträge im Speicher direkt hintereinander abgelegt.

Quelltext 9.2: Speicherverwaltung bei Arrays

```
DIM AS DOUBLE  d(2)         ' 3 DOUBLE-Eintraege
DIM AS SHORT   s(1, 1)      ' 2x2 SHORT-Werte

PRINT "Position der DOUBLE-Werte d():"
PRINT @d(0), @d(1), @d(2)
PRINT
PRINT "Position der SHORT-Werte s():"
PRINT @s(0, 0), @s(0, 1), @s(1, 0), @s(1, 1)
SLEEP
```

Ausgabe

```
Position der DOUBLE-Werte d():
3214830252    3214830260    3214830268

Position der SHORT-Werte s():
3214830212    3214830214    3214830216    3214830218
```

Die Ausgabe ist natürlich nur ein mögliches Beispiel. Sie sehen jedoch mehreres:

- Die **DOUBLE**-Werte folgen im Abstand von 8 Bit aufeinander, die **SHORT**-Werte im Abstand von 2 Bit. Das entspricht der Größe des jeweiligen Datentyps.
- Bei mehrdimensionalen Arrays folgen im Speicher zuerst die Werte aufeinander, die denselben ersten Indexwert besitzen.
- Die beiden Speicherbereiche der Arrays müssen nicht nebeneinander liegen, auch wenn die Arrays direkt nacheinander definiert wurden. Das Programm sucht sich jeweils eine passende „Lücke“ im Speicherblock, in die das Array am Stück untergebracht werden kann.

9.4. Fragen zum Kapitel

Das Verständnis von Pointern ist für Einsteiger sicher nicht so einfach, und es ist nicht schlimm, wenn im Moment noch ein paar Fragen offen bleiben. Ein paar davon habe ich hier zusammengestellt:

1. Pointer – was ist das überhaupt?
2. Wozu brauche ich sowas?
3. Wie groß ist der Pointer einer **SHORT**-Variablen? Wie groß ist der Pointer auf einen **STRING**?

10. Bedingungen

10.1. Einfache Auswahl: `IF ... THEN`

10.1.1. Einzeilige Bedingungen

Während das Computerprogramm zunächst einmal der Reihe nach von der ersten Zeile bis zur letzten abgearbeitet wird, ist es häufig nötig, bestimmte Programmteile nur unter besonderen Voraussetzungen auszuführen. Denken Sie an folgendes Beispiel aus dem Alltag: Wenn es acht Uhr abends ist – und nur dann – soll der Fernseher für die Nachrichten eingeschalten werden. (Ja, es soll tatsächlich Haushalte geben, in denen der Fernseher nicht den ganzen Tag über läuft ...)

```
WENN es 20:00 Uhr ist DANN schalte den Fernseher ein.
```

Zunächst tauchen in dem Satz die beiden Schlüsselwörter **Wenn** und **Dann** auf. Zwischen den beiden Wörtern steht eine Bedingung (`es ist 20:00 Uhr`). Diese Bedingung kann erfüllt sein oder nicht – aber nur dann, wenn sie erfüllt ist, wird die anschließende Handlung (`schalte den Fernseher ein`) durchgeführt.

In FreeBASIC-Syntax sieht das ähnlich aus:

```
IF bedingung THEN anweisung
```

Nehmen wir einmal ein einfaches Beispiel: Der Benutzer wird nach seinem Namen gefragt. Wenn dem Programm der Name besonders gut gefällt, findet es dafür lobende Worte.

Quelltext 10.1: Einfache Bedingung (einzeilig)

```
DIM AS STRING eingabe
INPUT "Gib deinen Namen ein: ", eingabe
IF eingabe = "Stephan" THEN PRINT "Das ist aber ein besonders schoener Name!"
PRINT "Danke, dass du dieses Programm getestet hast."
SLEEP
```

Die Ausgabe "Das ist aber ein besonders schoener Name!" erscheint nur, wenn die zuvor genannte Bedingung erfüllt ist.

10.1.2. Mehrzeilige Bedingungen (`IF`-Block)

Nun passiert es häufig, dass unter einer bestimmten Bedingung nicht nur ein einziger Befehl, sondern eine ganze Reihe von Anweisungen ausgeführt werden soll. Alle Anweisungen in eine Zeile zu quetschen, wird schnell unübersichtlich. Wenn daher auf das **THEN** keine Anweisung, sondern ein Zeilenumbruch erfolgt, bedeutet das für den Compiler, dass die folgenden Zeilen nur unter der gegebenen Bedingung ausgeführt werden sollen. Allerdings muss dann explizit festgelegt werden, wo dieser Anweisungsblock enden soll. Dazu dient der Befehl **END IF**.

In „Alltagssprache“ könnte das etwa so aussehen:

```
WENN du Hunger hast DANN
  OEFFNE Keksdose
  NIMM Keks
  SCHLIESSE Keksdose
  ISS Keks
ENDE WENN
```

Für die FreeBASIC-Umsetzung bauen wir Quelltext 10.1 ein klein wenig um:

Quelltext 10.2: Einfache Bedingung (mehrzeilig)

```
DIM AS STRING eingabe
INPUT "Gib deinen Namen ein: ", eingabe
IF eingabe = "Stephan" THEN
  PRINT "Das ist aber ein besonders schoener Name!"
  ' Hier koennten jetzt noch weitere Befehle folgen
END IF
PRINT "Danke, dass du dieses Programm getestet hast."
SLEEP
```

Diese Variante wird auch als **IF**-Block bezeichnet. Natürlich kann sie auch genutzt werden, wenn nur eine einzige Anweisung ausgeführt werden soll. Manche Programmierer bevorzugen es, **IF**-Abfragen ausschließlich mit Blöcken umzusetzen, weil dadurch Bedingungsabfrage und resultierende Anweisung klar voneinander getrennt sind. Dies kann sich positiv auf die Lesbarkeit des Quelltextes auswirken.

Apropos Lesbarkeit: Sicher sind Ihnen die Einrückungen in den beiden letzten Beispielen aufgefallen. Solche Einrückungen spielen für den Compiler keine Rolle; ihm ist es völlig egal, ob Sie Ihre Zeilen einrücken und wie weit. Wichtig sind sie jedoch für den Leser des Quelltextes – also für alle, welche die Funktionsweise des Programmes verstehen und vielleicht auch verbessern und weiterentwickeln wollen, und natürlich für den Programmierer selbst!

Um den Quelltext lesbar zu gestalten, werden alle Zeilen, die auf derselben „logischen Ebene“ stehen, gleich weit eingerückt. In Quelltext 10.2 werden die ersten drei Zeilen immer ausgeführt (auch die Bedingungsabfrage findet immer statt). Daher befinden sie

sich auf der „obersten“ Ebene, sind also nicht eingerückt. Zeilen 4 und 5 werden nur unter der genannten Bedingung ausgeführt. Sie befinden sich eine Ebene „tiefer“ und werden eingerückt. Da sie sich beide auf derselben Ebene befinden – ist die Bedingung erfüllt, werden beide Zeilen berücksichtigt – wurden sie auch gleich weit eingerückt. **END IF** beendet den **IF**-Block und steht damit wieder in derselben Einrückungsebene wie Zeile 3, ebenso wie die restlichen Zeilen, die unabhängig von der Bedingung in jedem Fall ausgeführt werden.

Wie stark die Zeilen eingerückt werden, wird von Programmierer zu Programmierer unterschiedlich gehandhabt. Üblich sind Einrückungen von zwei bis vier Leerzeichen pro Ebene oder, alternativ dazu, die Verwendung von Tabulatoren. Bleiben Sie jedoch innerhalb einer Quelltext-Datei bei *einer* dieser Möglichkeiten: verwenden Sie entweder *immer* zwei Leerzeichen oder *immer* vier Leerzeichen oder *immer* Tabulatoren. Die Arten zu mischen sorgt lediglich für Verwirrung, da man dann nicht mehr auf den ersten Blick einwandfrei die Ebenentiefe erkennen kann. Leerzeichen und Tabulatoren zu mischen ist sowieso eine schlechte Idee, allein schon, weil unterschiedliche Editoren einen Tabulator auch unterschiedlich weit eingerückt darstellen können. Was in Ihrem Editor dann gleichmäßig aussieht, ist bei einem anderen möglicherweise sehr ungleichmäßig.

Ein sauberer Umgang mit Einrückungen ist vor allem wichtig, wenn mehrere Blöcke ineinander verschachtelt sind:

Quelltext 10.3: Verschachtelte Bedingungen

```
DIM username AS STRING, alter AS INTEGER
INPUT "Gib deinen Namen ein: ", username
IF username = "Stephan" THEN
  PRINT "Das ist aber ein besonders schoener Name!"
  INPUT "Wie alt bist du? ", alter
  IF alter = 0 THEN
    PRINT "Dann bist du ja vor nicht einmal einem Jahr geboren worden!"
  END IF
END IF
PRINT "Danke, dass du dieses Programm getestet hast."
SLEEP
```

Die Benutzereingabe des Alters und die daraus resultierende Bedingungsabfrage erfolgen nur, wenn bereits der richtige Name eingegeben wurde. Ansonsten springt das Programm schon von vornherein in die Zeile 10. **IF**-Blöcke können beliebig ineinander verschachtelt werden. Jedoch muss jeder **IF**-Block auch mit einem **END IF** beendet werden.

Hinweis:
Statt **END IF** findet man in manchen Quelltexten auch die zusammengeschriebene Form **ENDIF**. Diese wird in einigen anderen BASIC-Dialekten für den Abschluss des Blockes verwendet und wird in FreeBASIC aus Kompatibilitätsgründen ebenso unterstützt.

10.1.3. Alternativauswahl

In der bisherigen einfachen Form wird der Code nur ausgeführt, wenn die Bedingung erfüllt ist; ansonsten wird das Programm nach dem **END IF** fortgesetzt. Man kann allerdings auch Alternativen festlegen, die nur dann ausgeführt werden, wenn die Bedingung *nicht* erfüllt ist. Dazu dienen die beiden Schlüsselwörter **ELSEIF** und **ELSE**.

```
IF bedingung1 THEN
  anweisungen
ELSEIF bedingung2 THEN
  anweisungen
ELSEIF bedingung3 THEN
  anweisungen
' ...
ELSE
  anweisungen
END IF
```

Zuerst wird *bedingung1* geprüft. Ist sie erfüllt, werden die Anweisungen nach dem **THEN** ausgeführt. Die **ELSEIF**-Blöcke werden in diesem Fall übersprungen. Ist *bedingung1* dagegen nicht erfüllt, werden der Reihe nach die Bedingungen hinter den **ELSEIF**-Zeilen geprüft, bis das Programm auf die erste zutreffende Bedingung stößt. Der darauf folgende Anweisungsblock wird dann ausgeführt. Die Anweisungen nach dem **ELSE** kommen nur dann zum Tragen, wenn keine einzige der abgearbeiteten Bedingungen erfüllt war.

Ein **IF**-Block kann beliebig viele (auch keine) **ELSEIF**-Zeilen enthalten, aber höchstens eine **ELSE**-Zeile.

Quelltext 10.4: Mehrfache Bedingung

```
DIM AS INTEGER alter
INPUT "Gib dein Alter ein: ", alter
IF alter < 0 THEN
  ' Das Alter ist kleiner als 0
  PRINT "Das Alter kann nicht negativ sein!"
ELSEIF alter < 3 THEN
  ' Das Alter ist mind. 0, aber kleiner als 3
  PRINT "Bist du nicht noch zu jung?"
ELSEIF alter < 18 THEN
  ' Das Alter ist mind. 3, aber kleiner als 18
  PRINT "Du bist noch nicht volljaehrig!"
ELSE
  ' Das Alter ist mind. 18
  PRINT "Dann bist du ja volljaehrig!"
END IF
SLEEP
```

Wenn Sie Quelltext 10.4 mehrmals mit verschiedenen Alterseingaben ausführen, werden Sie sehen, dass immer nur der erste Abschnitt ausgeführt wird, dessen Bedingung erfüllt ist.

10.2. Bedingungsstrukturen

Bedingungen können, wie gesagt, *wahr* oder *falsch* sein. Nun würde sich zur Verwaltung von Wahrheitswerten natürlich ein **BOOLEAN** anbieten. FreeBASIC steht hier jedoch stark in der Tradition alter BASIC-Dialekte, die es ermöglichen, Wahrheitswerte auch in aufwändige Rechnungen einzubauen. Daher verwendet FreeBASIC – wie schon sein Vorgänger QuickBASIC – für Vergleiche die Ergebnisse `0` für *falsch* und `-1` für *wahr*. Allerdings wird jeder beliebige Zahlenwert ungleich `0` als *wahr* interpretiert. Beispielsweise würde auch dies hier funktionieren:

```
IF 2+3 THEN PRINT "Der Ausdruck '2+3' ist wahr."
```

`2+3` ist nicht `0`, und daher wird der anschließende Text angezeigt. Aus diesem Grund wird in diesem Buch zwischen dem Wahrheitswert *wahr* und dem **BOOLEAN** `true` unterschieden. Nichtsdestotrotz kann der Wert eines Vergleiches (`0` oder `-1`) auch einem **BOOLEAN** zugewiesen und damit in einen der Werte `false` oder `true` konvertiert werden.

10.2.1. Vergleiche

Bisher haben wir einfache Vergleiche wie = (ist gleich) und < (ist kleiner als) verwendet. Dass bei beiden Vergleichen Zahlenwerte zurückgegeben werden, kann man übrigens

leicht mithilfe von **PRINT** überprüfen:

```
PRINT 5 = 5     ' offensichtlich wahr; also wird -1 ausgegeben
PRINT 5 < 5     ' falsch; die Ausgabe ist 0
PRINT 6-2*(5=5) ' Rechenausdruck mit dem Ergebnis 8
SLEEP
```

Folgende Vergleichsoperatoren können verwendet werden:

Operator	Bedeutung	Beispiel	Wert
$>$	größer als	$5 > 5$	0
		$3 > 8$	0
$<$	kleiner als	$5 < 5$	0
		$3 < 8$	−1
$=$	gleich	$5 = 5$	−1
		$3 = 8$	0
$>=$	größer oder gleich	$5 >= 5$	−1
		$3 >= 8$	0
$<=$	kleiner oder gleich	$5 <= 5$	−1
		$3 <= 8$	−1
$<>$	ungleich	$5 <> 5$	0
		$3 <> 8$	−1

Tabelle 10.1.: Liste der Vergleichsoperatoren

Nicht nur Zahlen können miteinander verglichen werden, sondern auch Zeichenketten. Während die Prüfung auf Gleichheit bzw. auf Ungleichheit noch sehr einleuchtend ist, benötigen die Operatoren $>$ und $<$ bei Zeichenketten noch eine genauere Erläuterung. Die Zeichenketten werden hierbei von vorn nach hinten Zeichen für Zeichen verglichen (solange bis das Ergebnis feststeht). Ausschlaggebend ist dabei der *ASCII-Code* des Zeichens (vgl. dazu Kapitel 13.4 und Anhang C). Zu beachten ist dabei, dass Großbuchstaben einen kleineren ASCII-Code besitzen als Kleinbuchstaben.

Quelltext 10.5: Vergleich von Zeichenketten

```
PRINT "Ist 'b' kleiner als 'a'?         "; "b"    < "a"
PRINT "Ist 'Anna' kleiner als 'Anne'?   "; "Anna" < "Anne"
PRINT "Ist 'Kind' kleiner als 'Kinder'? "; "Kind" < "Kinder"
PRINT "Ist 'Z' kleiner als 'a'?         "; "Z"    < "a"
SLEEP
```

Ausgabe

```
Ist 'b' kleiner als 'a'?          0
Ist 'Anna' kleiner als 'Anne'?   -1
Ist 'Kind' kleiner als 'Kinder'? -1
Ist 'Z' kleiner als 'a'?         -1
```

Gerade das letzte Beispiel scheint den Vergleichsoperator für eine alphabetische Sortierung (für die Programmierung eines Telefonbuchs o. ä.) ungeeignet zu machen. Doch dafür gibt es Abhilfe: Die zu vergleichenden Zeichenketten können zuvor komplett in Kleinbuchstaben (oder alternativ natürlich auch komplett in Großbuchstaben) umgewandelt werden. Darauf werden wir in Kapitel 15.2.6 zu sprechen kommen.

Für Fortgeschrittene:
Die Vergleichsoperatoren überprüfen in FreeBASIC lediglich die Gleichheit der Werte. Die Gleichheit der Datentypen kann z. B. mit **`TYPEOF()`** (zur Compilier-Zeit) oder **`IS`** (bei UDTs mit RTTI-Funktionalität) geprüft werden.

10.2.2. Logische Operatoren

Bedingungen setzen sich manchmal aus mehreren Teilaspekten zusammen. Im folgenden Beispiel wird der Zutritt ab 18 Jahren gestattet. Jugendliche dürfen nur eintreten, wenn sie mindestens 14 Jahre alt sind und von ihren Eltern begleitet werden.

```
WENN Alter >= 18 ODER (Alter > 13 UND Begleitung = "Eltern") DANN Zutritt
```

In FreeBASIC heißen die entsprechenden Operatoren **`OR`** (oder) bzw. **`AND`** (und). Wenn Sie in einer Bedingung beide Operatoren mischen, sollten Sie zusammengehörige Ausdrücke wie im obigen Beispiel mit Klammern umschließen, um sicher zu gehen, dass sie in der richtigen Reihenfolge ausgewertet werden. Wären die Klammern anders gesetzt, wäre auch eine andere Bedingung gefordert.

In FreeBASIC-Syntax sieht die Bedingung folgendermaßen aus:

```
IF Alter >= 18 OR (Alter > 13 AND Begleitung = "Eltern") THEN Zutritt
```

Hinweis:
Es gibt bei der Abarbeitung der Operatoren eine feste Reihenfolge, ähnlich wie die Rechenvorschrift „Punkt vor Strich“. Sie wird durch die sogenannten Vorrangregeln bestimmt, die in Anhang F aufgelistet sind, und kann durch den Einsatz von runden Klammern geändert werden. Wenn Sie unsicher sind, welche Operatoren Vorrang haben, sollten Sie auf Klammern zurückgreifen, auch um eine bessere Lesbarkeit des Quelltextes zu gewährleisten.

AND prüft also, ob *beide* Bedingungen (links und rechts vom **AND**) *wahr* sind, während **OR** überprüft, ob *mindestens eine* der beiden Bedingung zutrifft. Ganz richtig ist das allerdings noch nicht – verglichen werden nämlich die einzelnen Bits der beiden Ausdrücke. Dazu ist es wichtig zu wissen, dass der Computer im Binärsystem rechnet.

10.2.3. Das Binärsystem

Werte werden im Prozessor durch zwei Zustände (z. B. Strom an/Strom aus) festgehalten; im übertragenen Sinn kann man von zwei Werten 0 und 1 sprechen. Dies entspricht einer Speichereinheit bzw. *1 Bit*. 8 Bit werden zusammengefasst zu *1 Byte*, welches $2^8 = 256$ Zustände annehmen kann, je nachdem, welche Bit gesetzt sind und welche nicht.
Ein möglicher Zustand wäre z. B. folgender:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	1	0	1	1

Wird dieser Zustand als Ganzzahl interpretiert, dann zählt von rechts nach links jedes Bit doppelt so viel wie das davor. Bit 0 zählt also als `1`, Bit 1 als `2`, Bit 2 als `4` usw. Dies entspricht im wesentlichen der Darstellung einer Zahl im Dezimalsystem, nur dass die einzelnen Stellen nicht mehr Einer, Zehner, Hunderter, Tausender usw. repräsentieren, sondern Einer, Zweier, Vierer, Achter usw. Die oben dargestellte Zahl hätte also (von rechts gelesen) den Wert

$$1 \cdot 1 + 1 \cdot 2 + 0 \cdot 4 + 1 \cdot 8 + 0 \cdot 16 + 0 \cdot 32 + 1 \cdot 64 + 0 \cdot 128 = 75$$

Man spricht hierbei vom Binärsystem oder, mathematisch etwas genauer, vom Dualsystem (Programmierer verwenden beide Begriffe synonym). Dass die Nummerierung der Bits mit `0` beginnt, hat übrigens einen ganz praktischen Grund, denn dadurch lässt sich die Wertigkeit eines Bits ganz einfach als Zweierpotenz berechnen: $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, ...

Mit diesem System können mit einem Byte Zahlen von 0 bis 255 dargestellt werden. Für größere Zahlen werden mehrere Byte „zusammengesetzt“. Auch negative Zahlen können so dargestellt werden – dazu wird vereinbart, dass das höchstwertige Bit kennzeichnet, ob die Zahl positiv oder negativ ist. Für den Wert `-1` sind alle Bit gesetzt, für `-2` alle außer Bit `0` usw. In Quelltext 10.6 wird die Funktion **`BIN()`** verwendet, um die Binärdarstellung von Dezimalzahlen zu erhalten. Sie können etwas experimentieren, wie negative Werte binär dargestellt werden.

Hintergrundinformation:
Die von FreeBASIC verwendete Darstellung für Ganzzahlen heißt Zweierkomplementdarstellung. Es ist die heute am häufigsten verwendete Art, positive und negative Zahlen in binärer Form darzustellen.

Auch andere Informationen wie Farbwerte auf dem Monitor oder Zeichen innerhalb einer Zeichenkette werden letztendlich über das Binärsystem in Bytes gespeichert.

10.2.4. **`AND`** und **`OR`** als Bit-Operatoren

Auch wenn wir zuvor so getan haben, also ob es sich bei **`AND`** und **`OR`** um logische Operatoren handelt, ist das nicht ganz richtig. Sofern ausschließlich **`BOOLEAN`** bzw. nur die Werte `0` und `-1` verwendet werden, ist alles in Ordnung, allerdings verknüpfen die beiden Operatoren die Werte bitweise. Das bedeutet: Die Werte werden Bit für Bit miteinander verglichen und im Rückgabewert entsprechend gesetzt oder nicht. **`AND`** setzt das Bit im Rückgabewert genau dann, wenn das Bit auch in *beiden* Operanden gesetzt ist. **`OR`** setzt das Bit genau dann, wenn es in *mindestens einem* der Operanden gesetzt ist.

Das soll in folgendem Codebeispiel verdeutlicht werden. Dazu bedienen wir uns der Funktion **`BIN()`**, die den Binärwert einer Zahl als String zurückgibt. **`BIN()`** kann neben der umzuwandelnden Zahl optional noch ein zweiter Parameter mitgegeben werden, der angibt, wie viele Stellen der Binärzahl zurückgegeben werden sollen – ggf. wird die Zahl abgeschnitten oder mit Nullen aufgefüllt. Durch eine einheitliche Länge lässt sich die Ausgabe besser formatieren und damit leichter lesen. Im folgenden Quelltext werden alle Binärwerte mit vier Stellen ausgegeben.

Quelltext 10.6: Bit-Operatoren AND und OR

```
DIM AS INTEGER z1 = 6, z2 = 10

PRINT "Verknuepfung AND"
PRINT z1, BIN(z1, 4)
PRINT z2, BIN(z2, 4)
PRINT "---", "----"
PRINT z1 AND z2, BIN(z1 AND z2, 4)

PRINT "Verknuepfung OR"
PRINT z1, BIN(z1, 4)
PRINT z2, BIN(z2, 4)
PRINT "---", "----"
PRINT z1 OR z2, BIN(z1 OR z2, 4)
SLEEP
```

Ausgabe

```
Verknuepfung mit AND
 6             0110
 10            1010
---            ----
 2             0010

Verknuepfung mit OR
 6             0110
 10            1010
---            ----
 14            1110
```

Wenn Sie **AND** zur Bestimmung eines Wahrheitswertes verwenden wollen, stoßen Sie hier auf ein Problem: die Zahlen `2` und `4` z. B. werden beide als *wahr* interpretiert, `2 AND 4` jedoch ist `0` und damit *falsch*. Sie sollten in diesem Fall unbedingt auf die Vergleichoperatoren $=$, $<$ usw. zurückgreifen oder sicherstellen, dass nur die Werte `0` und `-1` verglichen werden.

10.2.5. ANDALSO und ORELSE

Alternativ dazu existieren die (nun tatsächlich logischen) Operatoren **ANDALSO** und **ORELSE**. Diese arbeiten jedoch etwas anders: Zunächst werden beide Seiten immer als Wahrheitswert definiert, womit `2 ANDALSO 4 = -1` (*wahr*) ist. Zum anderen wird die Auswertung nur so lange durchgeführt, bis das Ergebnis feststeht. In der Praxis bedeutet

das für **ANDALSO**: Wenn die linke Seite bereits *falsch* ist, kann der Gesamtausdruck nicht mehr *wahr* sein, und die rechte Seite wird daher gar nicht mehr ausgewertet. Wenn umgekehrt bei **ORELSE** die linke Seite *wahr* ist, dann ist auch der Gesamtausdruck *wahr*. Auch hier wird die rechte Seite nicht mehr ausgewertet.

Ein einfaches Beispiel:

Quelltext 10.7: ANDALSO und ORELSE

```
DIM AS INTEGER z1 = 3, z2 = 5
IF z1 < 2 ANDALSO z2 > 4 THEN PRINT "Hoppla ..."
SLEEP
```

Wenn beide Operanden vom Typ **BOOLEAN** sind, liefern **ANDALSO** und **ORELSE** als Rückgabewert ein **BOOLEAN**, ansonsten einen der Werte `-1` und `0`. Wenn Sie die Operatoren allerdings nur zur Bedingungsabfrage einsetzen und nicht innerhalb von Rechenausdrücken, macht das keinen Unterschied.

Sinnvoll sind die beiden Operatoren z. B. dann, wenn auf der rechten Seite aufwendige Berechnungen durchgeführt werden müssen, die dann, wenn die linke Seite nicht erfüllt ist, komplett wegfallen können. Andererseits kann es auch passieren, dass die rechte Seite gar nicht ausgewertet werden *darf*, da die erforderlichen Bedingungen nicht erfüllt sind und die Auswertung zu einem Fehler führen würde. **ANDALSO** findet man daher auch oft im Zusammenhang mit Pointerzugriffen, vor denen geprüft wird, ob der Pointer überhaupt korrekt gesetzt ist.

```
' Vermeidung der aufwendigen Berechnung, wenn Vorbedingung nicht erfuellt
IF bedingung ANDALSO aufwendigeBerechnung() > 0 THEN tueWas

' Vermeidung eines unerlaubten Null-Pointer-Zugriffs
IF meinPointer ANDALSO *meinPointer = 10 THEN tueWas
```

In beiden Fällen wird zunächst überprüft, ob der erste Ausdruck *wahr* ist, also ob `bedingung` bzw. `meinPointer` einen Wert ungleich Null besitzen. Ob `meinPointer` tatsächlich ein korrekter Pointer auf eine Zahl ist, kann damit zwar nicht sichergestellt werden, aber zumindest wird so ein Nullpointer abgefangen. Die Abfrage des gespeicherten Wertes erfolgt auf jeden Fall nur dann, wenn `meinPointer` kein Nullpointer ist.

Achtung:
ANDALSO und **ORELSE** führen nicht automatisch zu einem schnelleren Code! Falsch angewendet kann es im Extremfall sogar zu einer Verlangsamung kommen.

10.2.6. Weitere Bit-Operatoren: XOR, EQV, IMP und NOT

Neben **AND** und **OR** wird auch der Operator **XOR** (*eXclusive OR*) häufig verwendet, der umgangssprachlich als *entweder – oder* aufgefasst werden kann. Es muss also *genau eines* der verglichenen Bits gesetzt sein. Sind beide Bits gesetzt, ergibt sich der Wert `0`, genauso wenn keines der beiden Bits gesetzt ist.

Etwas exotischer sind die beiden Operatoren **EQV** (*EQuivalent Value*) und **IMP** (*IMPlication*). **EQV** prüft die Gleichheit beider Bits, also ob *beide* gesetzt sind oder *beide* nicht. Bei **IMP** ist der Hintergrund etwas komplizierter: Es wird geprüft, ob aus der ersten Aussage die zweite folgt. Praktisch bedeutet es, dass das Bit immer gesetzt wird, außer wenn es im linken Ausdruck gesetzt war, im rechten jedoch nicht.

NOT schließlich dreht die Bitwerte einfach um: wo das Bit gesetzt war, ist es im Ergebnis nicht gesetzt und umgekehrt. Im Gegensatz zu den bisherigen Operatoren wird **NOT** lediglich auf einen Operanden angewandt. `NOT 0` z. B. ist `-1` und `NOT -1` ist `0` (auf den Gesamtwert betrachtet).

Noch einmal zusammengefasst: Die Bit-Operatoren **AND**, **OR**, **XOR**, **EQV**, **IMP** und **NOT** vergleichen die beiden übergebenen Ausdrücke (bei **NOT** nur ein übergebener Ausdruck) Bit für Bit und setzen das entsprechende Bit im Ergebnis je nachdem auf den Wert `0` oder `1`. Tabelle 10.2 stellt das Verhalten der Operatoren in einer Übersicht zusammen.

Sofern diese Operatoren ausschließlich für **BOOLEAN** bzw. ausschließlich für `-1` und `0` verwendet werden (aber nur dann), verhalten sie sich wie logische Operatoren.

10.3. Mehrfachauswahl: SELECT CASE

Wenn eine Variable auf mehrere verschiedene Werte geprüft werden soll, ist das Konstrukt **IF ... THEN ... ELSEIF ... ELSE ... END IF** oft recht umständlich. Ein denkbarer Fall wäre eine Tastaturabfrage, auf die in Abhängigkeit von der gedrückten Taste reagiert werden soll. Im Folgenden soll auf die Tasten W-A-S-D bzw. 8-4-5-6 zur Steuerung einer Spielfigur reagiert werden. Da es egal sein soll, ob dabei die Shift-Taste gedrückt wurde oder nicht, wird der Tastenwert zuerst mit **LCASE()** in Kleinbuchstaben umgewandelt.

Schlüsselwort	Ausdruck1	Ausdruck2	Ergebnis
AND	0	0	0
	0	1	0
	1	0	0
	1	1	1
OR	0	0	0
	0	1	1
	1	0	1
	1	1	1
XOR	0	0	0
	0	1	1
	1	0	1
	1	1	0
EQV	0	0	1
	0	1	0
	1	0	0
	1	1	1
IMP	0	0	1
	0	1	1
	1	0	0
	1	1	1
NOT	0	-	1
	1	-	0

Tabelle 10.2.: Bit-Operatoren

Quelltext 10.8: Mehrfache Bedingung (2) mit IF

```
DIM AS STRING taste
PRINT "Druecke eine Steuerungstaste."
taste = INPUT(1)

IF LCASE(taste) = "w" or taste = "8" THEN
  PRINT "Taste nach oben gedrueckt"
ELSEIF LCASE(taste) = "s" or taste = "5" THEN
  PRINT "Taste nach unten gedrueckt"
ELSEIF LCASE(taste) = "a" or taste = "4" THEN
  PRINT "Taste nach links gedrueckt"
ELSEIF LCASE(taste) = "d" or taste = "6" THEN
  PRINT "Taste nach rechts gedrueckt"
ELSE
  PRINT "keine Steuerungstaste gedrueckt"
END IF
SLEEP
```

Wie Sie sehen, steckt in diesem Quellcode eine Menge Schreibarbeit – und damit verbunden ein erhöhtes Risiko an Tippfehlern.

10.3.1. Grundaufbau

Eine alternative Bedingungsstruktur bietet **SELECT CASE**. Hier wird nur einmal die Bedingung angegeben; danach folgen nur noch die gewünschten Vergleichswerte. **SELECT CASE** hat folgenden Grundaufbau:

```
SELECT CASE Ausdruck
  CASE Ausdrucksliste1
    Anweisung1
  CASE Ausdrucksliste2
    Anweisung2
  ' ...
  CASE ELSE
    Anweisung
END SELECT
```

Mithilfe von **SELECT CASE** kann man Quelltext 10.8 folgendermaßen umformulieren:

Quelltext 10.9: Mehrfache Bedingung (2) mit SELECT CASE

```
DIM AS STRING taste
PRINT "Druecke eine Steuerungstaste."
taste = INPUT(1)

SELECT CASE LCASE(taste)
  CASE "w", "8"
    PRINT "Taste nach oben gedrueckt"
  CASE "s", "5"
    PRINT "Taste nach unten gedrueckt"
  CASE "a", "4"
    PRINT "Taste nach links gedrueckt"
  CASE "d", "6"
    PRINT "Taste nach rechts gedrueckt"
  CASE ELSE
    PRINT "keine Steuerungstaste gedrueckt"
END SELECT
SLEEP
```

Der Ablauf ist ähnlich wie bei **IF**. Zuerst wird der Ausdruck ausgewertet und dann in die erste **CASE**-Zeile gesprungen, auf welche der Ausdruck zutrifft. Alle Zeilen zwischen dieser und der nächsten **CASE**-Zeile werden ausgeführt, danach fährt das Programm am Ende des **SELECT**-Blockes fort. Trifft keine der Ausdruckslisten zu, werden die Zeilen hinter dem **CASE ELSE** ausgeführt, sofern vorhanden.

In jeder Ausdrucksliste können ein oder auch mehrere Vergleichswerte stehen, die dann durch Komma voneinander getrennt werden. Diese Liste kann beliebig lang werden.

Es gibt auch noch weitere Möglichkeiten zur Angabe der Vergleichswerte, auf die in Kapitel 10.3.2 eingegangen wird.

Da der auszuwertende Ausdruck – hier der in Kleinbuchstaben umgewandelte Wert der Tastatureingabe – nur einmal angegeben werden muss, wird das Programm deutlich übersichtlicher. Allerdings gibt es auch einen programmtechnischen Vorteil: Die Umwandlung in Kleinbuchstaben, die ja durchaus etwas Rechenzeit kostet, muss nur einmal erledigt werden statt viermal wie in Quelltext 10.8. Und sollte sich der auszuwertende Ausdruck einmal ändern, muss dies auch nur an einer Stelle geschehen anstatt in jeder **ELSEIF**-Zeile. Dafür sind Sie aber auf einen einzigen Ausdruck für den gesamten Block eingeschränkt.

Zu beachten ist, dass immer nur einer der **CASE**-Abschnitte ausgeführt wird, auch wenn der Ausdruck bei mehreren Abschnitten angegeben wird. In einem solchen Fall wird immer der erste passende Abschnitt gewählt.

Vergleich mit anderen Sprachen:
Im Gegensatz zur entsprechenden C-Funktion `switch` wird der **SELECT**-Block nach Abarbeitung des passenden **CASE**-Abschnitts verlassen; die Verwendung einer `break`-Anweisung ist nicht nötig. Allerdings lassen sich auch in FreeBASIC die **SELECT**-Blöcke durch Kontrollanweisungen vorzeitig verlassen (siehe dazu Kapitel 11.5.2). Es ist dagegen nicht möglich, in den folgenden **CASE**-Abschnitten fortzufahren!

10.3.2. Erweiterte Möglichkeiten

Neben der einfachen Auflistung aller passenden Werte gibt es für die Ausdrucksliste noch weitere Möglichkeiten:

- Bereichsangaben:
  Mit `wert1 TO wert2` kann überprüft werden, ob der Ausdruck im Bereich von `wert1` bis `wert2` liegt. Er muss also größer oder gleich `wert1` sowie kleiner oder gleich `wert2` sein. Bereichsangaben sind auch bei Zeichenketten möglich, jedoch sind hier die Besonderheiten von String-Vergleichen zu beachten (siehe Kapitel 10.2.1).

- Vergleichsoperatoren:
  Die üblichen Vergleichsoperatoren <, >, <= und >= können eingesetzt werden, indem vor den Vergleichsoperator das Schlüsselwort **IS** gesetzt wird.

- Alle drei Varianten (Einzelwerte, Bereichsangaben, Vergleichsoperatoren) können beliebig kombiniert werden, indem man sie durch Kommata getrennt auflistet.

Dazu ein Beispiel aus der FreeBASIC-Referenz (dient nur zur Veranschaulichung und ist nicht ohne Änderung lauffähig):

Quelltext 10.10: Ausdruckslisten bei SELECT CASE

```
SELECT CASE a
 ' ist a = 5?
 CASE 5

 ' ist a ein Wert von 5 bis 10?
 ' Die kleinere Zahl muss zuerst angegeben werden
 CASE 5 TO 10

 ' ist a groesser als 5?
 CASE IS > 5

 ' ist a gleich 1 oder ein Wert von 3 bis 10?
 CASE 1, 3 TO 10

 ' ist a gleich 1, 3, 5, 7 oder b + 8?
 CASE 1, 3, 5, 7, b + 8
END SELECT
```

10.3.3. SELECT CASE AS CONST

Unter bestimmten Bedingungen kann der Compiler angewiesen werden, den **SELECT**-Block effizienter umzusetzen und dadurch die Ausführungsgeschwindigkeit zu erhöhen (was natürlich nur dann ins Auge fällt, wenn der Block während des Programmablaufs sehr oft ausgeführt werden muss).

- Der Ausdruck muss eine Ganzzahl sein.
- Die Ausdruckslisten dürfen nur Konstanten oder einfache Ausdrücke enthalten, also Ausdrücke, die keine Variablen enthalten. Die FreeBASIC-internen mathematischen Funktionen dürfen ebenfalls verwendet werden.
- Der Operator **IS** steht nicht zur Verfügung.
- Die Differenz zwischen dem kleinsten und dem größten Vergleichswert kann maximal 4096 betragen. Möglich sind also z. B. Werte im Bereich von 0 bis 4096 oder von 4 bis 4100.

Die Syntax lautet hier

```
SELECT CASE AS CONST Ausdruck
```

Der Hintergrund für die Einschränkungen ist, dass bei diesem Aufruf die durch die Ausdruckslisten festgelegten Sprungmarken bereits beim Compilieren festgelegt werden. Deswegen dürfen die Ausdruckslisten nur Werte enthalten, die bereits beim Compilieren feststehen (also keine Variablen).

10.4. Bedingungsfunktion: `IIF()`

Als dritte Möglichkeit steht die Funktion **`IIF()`** zur Verfügung. Diese funktioniert ein gutes Stück anders als die beiden vorher genannten Bedingungsblöcken: Je nachdem, ob die übergebene Bedingung *wahr* oder *falsch* ist, wird einer von zwei Werten zurückgegeben.

```
Rueckgabe = IIF(Bedingung, Rueckgabewert_Wenn_Wahr, Rueckgabewert_Wenn_Falsch)
```

Es ist eine Kurzform von

```
IF Bedingung THEN
  Rueckgabe = Rueckgabewert_Wenn_Wahr
ELSE
  Rueckgabe = Rueckgabewert_Wenn_Falsch
END IF
```

Die beiden möglichen Rückgabewerte müssen denselben Datentyp besitzen und zum Datentyp der Variablen *Rueckgabe* passen. Die Stärke von **`IIF()`** besteht darin, dass es auch in Ausdrücken eingebaut werden kann, z. B. in eine Berechnung oder einer Textausgabe. Dazu ein paar Beispiele:

Quelltext 10.11: Bedingungen mit IIF

```
' Geld um 100 reduzieren, aber nicht unter 0
geld = IIF(geld > 100, geld-100, 0)
' Gehaltstruktur abhaengig von den Dienstjahren
gehalt = 1000 + IIF(dienstjahre > 10, dienstjahre*50, dienstjahre*20)
' Rueckmeldung: Einlass erst ab 18
PRINT "Du darfst hier " & IIF(alter>=18, "selbstverstaendlich", "nicht") & " rein!"
```

10.5. Welche Möglichkeit ist die beste?

Wenn eine feststehende Variable oder ein Rechenausdruck auf mehrere verschiedene Ergebnisse hin geprüft werden soll, ist meistens **`SELECT CASE`** die beste Wahl. Der Quelltext ist damit am einfachsten zu lesen und zu warten. Allerdings kann **`SELECT`**

CASE nur einen einzigen Ausdruck abprüfen; auf den Inhalt zweier oder mehr verschiedener Variablen kann nicht gleichzeitig geprüft werden. In diesem Fall bleibt nur der Einsatz von **IF** (verschachtelte **SELECT**-Blöcke sind natürlich möglich, und ob sie sinnvoll sind, muss im Einzelfall geprüft werden). Die Stärke von **IF** besteht in der hohen Flexibilität.

Als Beispiel für einen Einsatz von **SELECT CASE**: Sie wollen ein tastaturgesteuertes Menü programmieren. Am Bildschirm werden fünf verschiedene Menüeinträge angezeigt, jeder mit einer Angabe versehen, mit welcher Taste er aufgerufen werden kann. In diesem Fall ist **SELECT CASE** ideal geeignet. Als weiteres Beispiel kann eine „Übersetzung“ von Schulnoten in ihre Textbedeutung dienen (was sich jedoch leichter über ein Array regeln lässt).

Dagegen lässt sich allein schon bei einer Abfrage von Benutzernamen und Passwort ein **SELECT CASE** nicht sinnvoll einsetzen. Sowohl **IF** als auch **SELECT CASE** haben also, je nach konkreten Fall, ihre Stärken.

IIF() wiederum lässt sich (ausschließlich) dann gewinnbringend einsetzen, wenn ein Rückgabewert in Abhängigkeit von der Bedingung gefordert ist. Ein **IIF()** lässt sich immer auch durch einen **IF**-Block ersetzen, benötigt dann aber oft mehr Schreib- und Speicheraufwand, da der Rückgabewert in der Regel zusätzlich in einer Variablen zwischengespeichert werden muss.

10.6. Fragen zum Kapitel

1. Welchen Sinn haben Einrückungen im Quelltext?
2. Welche Werte werden in FreeBASIC als *wahr* und welche als *falsch* interpretiert?
3. Nach welchem System werden Zeichenketten verglichen?
4. Was ist der Unterschied zwischen einem logischen Operator und einem Bit-Operator?
5. Wie muss eine Bedingung formuliert werden, wenn der Wert der Variablen `a` zwischen 3 und 8 oder aber zwischen 12 und 20 liegen soll?
6. Wieder eine kleine Programmieraufgabe: Das Programm soll den Benutzer nach Namen, Passwort und Alter fragen. Wenn der Name und das Passwort richtig sind, wird je nach Alter (verschiedene Altersbereiche wie „höchstens 14“, „zwischen 14 und 18“ und „mindestens 18“) eine andere Meldung ausgegeben.
   Das Programm soll dann dahingehend erweitert werden, dass es nicht nur *einen* Benutzernamen mit zugehörigem Passwort erkennt, sondern zwei verschiedene Benutzernamen, jedes mit einem eigenen Passwort.

11. Schleifen und Kontrollanweisungen

Oft soll eine Reihe von Anweisungen nicht nur einmal, sondern mehrmals hintereinander ausgeführt werden. Dazu gibt es das Konstrukt der Schleife.

Schleifen besitzen eine *Laufbedingung* und den *Schleifenrumpf*, auch *Schleifenkörper* genannt. Der Rumpf ist ein Anweisungsblock, der wiederholt ausgeführt werden soll. Die Bedingung steuert, unter welchen Voraussetzungen der Rumpf wiederholt werden soll; es handelt sich um eine Bedingungsstruktur, wie sie in Kapitel 10.2 behandelt wurde. Wie alle Kontrollstrukturen können Schleifen beliebig verschachtelt werden.

Schleifen sind bereits das erste Element der strukturierten Programmierung. Bevor sich das Programmierparadigma der strukturierten Programmierung durchsetzte, konnten Unterbrechungen im linearen Ablauf des Programms meist nur durch Sprunganweisungen durchgeführt werden. Die in den frühen BASIC-Jahren beliebte Sprunganweisung **`GOTO`** kann (und soll!) heute problemlos durch bessere Konstrukte ersetzt werden, nichtsdestotrotz ist sie in FreeBASIC weiterhin vorhanden und einsetzbar.

11.1. Sprunganweisungen

Eine Sprunganweisung im weiteren Sinn ist jede Art von Anweisung, die das Programm nicht in der nächsten Zeile, sondern an einer beliebigen Stelle fortsetzen lässt. Im engeren Sinn sind speziell die Funktionen **`GOTO`** und **`GOSUB`** gemeint, wovon letzterer jedoch von FreeBASIC nicht mehr unterstützt wird.

`GOTO` benötigt die Angabe einer Sprungmarke, auch Label genannt, und selbstverständlich muss die Position dieser Sprungmarke ebenfalls im Programm vermerkt sein. Der Name einer Sprungmarke folgt denselben Regeln wie Variablennamen. Die Sprungmarke wird mit einem Doppelpunkt `:` abgeschlossen.

```
PRINT "Hallo ";
GOTO welt
PRINT " du da!"
ende:
PRINT "Und ade!"
SLEEP
END
welt:
PRINT "Welt!"
GOTO ende
```

Ausgabe

```
Hallo Welt!
Und ade!
```

Das Programm springt von Zeile 2 zum Label `welt` in Zeile 8, fährt dann mit Zeile 9 und 10 fort, wo es wiederum zum Label `ende` in Zeile 4 springt und mit den Zeilen 5 und 6 fortsetzt. Zeile 3 wird in diesem Beispiel gar nicht ausgeführt.

Der Code ist nicht wirklich sinnvoll; aufgrund seines munteren Springens ist er sogar einfach nur schlecht (aber er sollte ja auch lediglich das Prinzip veranschaulichen). Mit **GOTO** lassen sich aber auch sinnvollere Dinge wie z. B. Schleifen umsetzen:

```
DIM AS STRING eingabe, passwort="schwertfisch8"
schleifenanfang:
INPUT "Gib das Passwort ein: ", eingabe
IF eingabe <> passwort THEN GOTO schleifenanfang
SLEEP
```

GOTO ist sehr mächtig, weil man damit sehr frei durch das Programm springen kann, aber darin liegt auch sein größter Nachteil. Es ist auf diesem Weg nämlich sehr einfach, den Programmcode unüberschaubar zu machen – und damit ist nicht nur die schwere Lesbarkeit für den menschlichen Programmierer, sondern auch maschinelle Auswertbarkeit des Codes gemeint. In den frühen BASIC-Jahren entstand dadurch eine Menge an nahezu nicht mehr wartbarem „Spaghetti-Code“. Der Einsatz von **GOTO** ist daher in nahezu allen höheren Programmiersprachen verpönt.[12] Einige Sprachen haben diese Art von Sprunganweisung sogar komplett aus dem Sprachschatz verbannt.

Wie oben bereits erwähnt lassen sich Programmierprobleme auch ohne **GOTO** lösen. Die oben verwendete **GOTO**-Schleife zur Passwortabfrage werden wir in Quelltext 11.1 durch eine **DO**-Schleife ersetzen.

[12] Und das seit schon über 50 Jahren – denken Sie über diesen Sachverhalt nach, bevor Sie ernsthaft in Erwägung ziehen, **GOTO** einzusetzen.

11.2. DO ... LOOP

Die **DO**-Schleife ist gewissermaßen der Allrounder der Schleifen. Sie beginnt mit **DO** und endet mit **LOOP** – alles dazwischen gehört zum Schleifenrumpf.

```
DO
  ' Anweisungen im Schleifenrumpf
LOOP
```

Die oben aufgeführte Schleife enthält keine Laufbedingung; sie wird immer wieder ohne Ende durchgeführt. In diesem Fall spricht man von einer Endlosschleife. Natürlich ist es in der Regel nicht erwünscht, dass das Programm endlos in der Schleife hängen bleibt. Deshalb gibt es zwei Möglichkeiten, eine Laufbedingung einzufügen. Man unterscheidet die *kopfgesteuerte* und die *fußgesteuerte* Schleife.

```
' kopfgesteuerte Schleife
DO { UNTIL | WHILE } Bedingung
  ' Anweisungen im Schleifenrumpf
LOOP

' fussgesteuerte Schleife
DO
  ' Anweisungen im Schleifenrumpf
LOOP { UNTIL | WHILE } Bedingung
```

Bei einer kopfgesteuerten Schleife wird die Bedingung überprüft, bevor die Schleife das erste Mal durchlaufen wird. Ist die Bedingung zu Beginn nicht erfüllt, dann wird der Schleifenrumpf überhaupt nicht ausgeführt, und das Programm fährt direkt am Ende der Schleife fort. Eine fußgesteuerte Schleife dagegen wird mindestens einmal durchlaufen, da die Bedingung erst nach dem ersten Durchlauf überprüft wird.

Die Schreibweise `DO { UNTIL | WHILE } Bedingung` ist eine Kurzschreibweise – die geschweiften Klammern bedeuten, dass genau eine der darin aufgeführten Möglichkeiten verwendet werden muss. Es lautet also entweder `DO UNTIL Bedingung` oder `DO WHILE Bedingung`. Diese Art der Schreibweise finden Sie auch sehr häufig in der Befehlsreferenz.[13] Die beiden Versionen besitzen einen kleinen, aber feinen Unterschied:

- **DO UNTIL** `Bedingung` führt die Schleife aus, *bis* die Bedingung erfüllt ist. Man spricht hier auch von einer Abbruchbedingung – ist sie erfüllt, wird die Schleife abgebrochen.
- **DO WHILE** `Bedingung` führt die Schleife aus, *solange* die Bedingung erfüllt ist. Die Schleife wird also verlassen, sobald die Bedingung das erste Mal nicht erfüllt

[13] Die deutschsprachige Befehlsreferenz finden Sie unter http://www.freebasic-portal.de/befehlsreferenz

ist. Dies ist der eigentliche Fall einer Laufbedingung – die Bedingung muss erfüllt sein, um weiterhin die Schleife zu durchlaufen.

Das gilt analog auch für fußgesteuerte Schleifen.

Grundsätzlich kann immer frei gewählt werden, ob eine Laufbedingung oder eine Abbruchbedingung verwendet werden soll; man muss lediglich die Bedingung passend formulieren. Die Schleife „Iss etwas, bis du satt bist“ (**UNTIL**-Version) lässt sich auch formulieren als „Iss etwas, solange du hungrig bist“ (**WHILE**-Version). Welche der beiden Möglichkeiten verwendet wird, ist zum Teil Geschmacksache, manchmal sieht auch eine Version eleganter aus als die andere.

Ob jedoch eine kopf- oder fußgesteuerte Schleife verwendet wird, ist durch die Aufgabenstellung bereits vorgegeben. Manchmal muss der Schleifenrumpf erst einmal ausgeführt werden, bevor überhaupt eine Abbruchbedingung feststeht. Quelltext 11.1 fragt so lange nach dem Passwort, bis die Eingabe richtig ist (natürlich öffnet diese Vorgehensweise Brute-Force Tor und Tür ...); eine Überprüfung macht jedoch erst nach der ersten Eingabe Sinn. Eine kopfgesteuerte Schleife wird dagegen z. B. benötigt, wenn Sie eine Datei bis zum Ende auslesen wollen – sollte die Datei leer sein (d. h. das Dateiende ist bereits zu Beginn erreicht), soll auch nichts ausgelesen werden.

Quelltext 11.1: Passwortabfrage in einer Schleife

```
DIM AS STRING eingabe, passwort="schwertfisch8"
DO
  INPUT "Gib das Passwort ein: ", eingabe
LOOP UNTIL eingabe = passwort
```

Mit den Schleifen lässt sich auch die Stärke der Arrays hervorragend ausspielen. Mit unserem Wissen aus Kapitel 8 können wir jetzt eine Schleife programmieren, die den Benutzer so lange Namen eingeben lässt, bis er mit einer Leereingabe beendet. Da im Vorfeld nicht feststeht, wie viele Eingaben erfolgen werden, benötigen wir ein dynamisches Array, das während der Eingabe ständig erweitert wird.

Quelltext 11.2: Wiederholte Namenseingabe

```
DIM AS STRING nachname(), eingabe  ' dynamisches Array deklarieren
DIM AS INTEGER i = 0               ' Zaehlvariable fuer die Array-Laenge
PRINT "Geben Sie die Namen ein - Leereingabe beendet das Programm"
DO
  PRINT "Name"; i+1; ": ";
  INPUT "", eingabe
  IF eingabe <> "" THEN
    REDIM PRESERVE nachname(i)
    nachname(i) = eingabe
    i += 1
  END IF
LOOP UNTIL eingabe = ""
PRINT "Sie haben"; UBOUND(nachname)+1; " Namen eingegeben."
SLEEP
```

Die Zählweise „von `0` bis `(Anzahl-1)`“ statt „von `1` bis `Anzahl`“ mag vielleicht etwas gewöhnungsbedürftig sein, ist jedoch durchaus üblich. Sie können stattdessen selbstverständlich auch `1` als untere Grenze wählen; dazu muss das Programm ein einigen wenigen Stellen angepasst werden.

11.3. WHILE ... WEND

Ein Sonderfall ist die **WHILE**-Schleife:

```
WHILE bedingung
  ' Anweisungen im Schleifenrumpf
WEND
```

Es handelt sich dabei lediglich um einen Spezialfall einer kopfgesteuerten Schleife und kann durch eine **DO**-Schleife ersetzt werden:

```
DO WHILE bedingung
  ' Anweisungen im Schleifenrumpf
LOOP
```

11.4. FOR ... NEXT

Während die **DO**-Schleife eine sehr frei definierbare Abbruchbedingung besitzt, führt die **FOR**-Schleife eine eigene Zählvariable mit. Diese wird bei jedem Schleifendurchlauf um einen festen Wert verändert. Überschreitet sie einen zu Beginn festgelegten Wert, dann wird die Schleife verlassen.

```
FOR zaehlvariable = startwert TO endwert [STEP schrittweite]
  ' Anweisungen im Schleifenrumpf
NEXT
```

`zaehlvariable` wird zu Beginn auf den Wert `startwert` gesetzt und dann nach jedem Schleifendurchlauf um den Wert `schrittweite` erhöht. Wenn sie dadurch den Wert `endwert` überschreitet, wird die Schleife verlassen. Die **FOR**-Schleife bietet sich also vor allem dann an, wenn eine Schleife eine genau festgelegte Anzahl von Durchläufen haben soll. Praktischerweise lässt sich die Zählvariable aber auch innerhalb der Schleife jederzeit abfragen.

Die eckigen Klammern bei `[STEP schrittweite]` bedeuten, dass diese Angabe optional ist, d. h. es ist nicht notwendig, eine Schrittweite anzugeben. Ohne anderweitige Angabe wird die Schrittweite `1` verwendet.

Hinweis:
Hinter dem **NEXT** kann noch einmal der Name der Zählvariablen angegeben werden. Dies war vor allem in älteren BASIC-Dialekten notwendig. In FreeBASIC ist diese Angabe nicht nötig; sie kann bei verschachtelten Schleifen aber der Übersichtlichkeit dienen.

11.4.1. Einfache FOR-Schleife

Das Prinzip lässt sich am einfachsten anhand von Beispielen verstehen. Quelltext 11.3 zählt einfach nur von `10` bis `20` nach oben:

Quelltext 11.3: Einfache FOR-Schleife

```
DIM AS INTEGER i
PRINT "Variablenwert vor der Schleife:"; i
FOR i = 10 TO 20
  PRINT i
NEXT
PRINT "Variablenwert nach der Schleife:"; i
SLEEP
```

Ausgabe

```
Variablenwert vor der Schleife: 0
 10
 11
 12
 13
 14
 15
 16
 17
 18
 19
 20
Variablenwert nach der Schleife: 21
```

Zunächst einmal wurde eine Zählvariable `i` angelegt. `i` mag ein sehr kurzer und nicht sehr aussagekräftiger Name sein, aber er hat sich für Zählvariablen eingebürgert, womit er bereits wieder einiges an Aussagekraft besitzt. Da der Variablen zu Beginn kein anderer Wert zugewiesen wurde, wird sie automatisch mit dem Wert `0` belegt.

Nun wird die Schleife betreten. `i` erhält den Startwert `10`, und die Schleife wird bis zum **`NEXT`** durchlaufen. Danach wird `i` auf `11` erhöht und mit dem Endwert (hier `20`) verglichen. Da `i` noch zu klein ist, springt das Programm wieder zurück zu Zeile 4, also in die Zeile nach dem **`FOR`**.

Dies geht so lange, bis `i` den Wert `20` erreicht hat. Nun wird die Schleife noch ein letztes Mal durchlaufen. Auch jetzt wird beim Erreichen des **`NEXT`** der Wert um 1 erhöht, weshalb `i` nun den Wert `21` besitzt. `i` wird wiederum mit der Laufbedingung verglichen. Da der Wert jetzt größer ist als `20`, wird die Schleife verlassen. Es mag auf den ersten Blick ungewöhnlich erscheinen, dass die Laufvariable am Ende größer ist als der Endwert, aber so arbeitet die **`FOR`**-Schleife nun einmal. Außerdem werden wir in Kapitel 11.5 sehen, wie dieses Verhalten sinnvoll genutzt werden kann.

11.4.2. `FOR`-Schleife mit angegebener Schrittweite

Mithilfe von **`STEP`** kann das Programm veranlasst werden, bei jedem Schleifendurchgang nicht um 1, sondern um einen beliebigen anderen Wert weiterzuzählen. Dieser Wert kann sogar negativ sein. Tatsächlich wird die Angabe `STEP -1` recht häufig verwendet, eben um im Einerschritt abwärts zu zählen. Bei negativer Schrittweite muss der Endwert dann auch kleiner sein als der Startwert, um einen sinnvollen Durchlauf starten zu können.

In Quelltext 11.4 wird ein kleiner Countdown programmiert, der schrittweise von 10 herunterzählt, immer mit einer Wartezeit von einer Sekunde. Mit **SLEEP** kann ein Parameter angegeben werden, der das Programm veranlasst, so viele Millisekunden zu warten. `SLEEP 1000` etwa wartet eine Sekunde (1000 Millisekunden). Allerdings kann die Wartezeit auch durch einen Tastendruck übersprungen werden. Wenn der Countdown genau zehn Sekunden dauern soll und nicht durch Tastendruck verkürzt werden darf, kann man **SLEEP** noch einen zweiten Parameter übergeben. Als zweiter Parameter ist nur `0` (Wartezeit kann übersprungen werden; Standard) oder `1` (Wartezeit kann nicht übersprungen werden) sinnvoll.

Quelltext 11.4: Countdown

```
DIM AS INTEGER i
PRINT "Der Countdown laeuft!"
FOR i = 10 TO 1 STEP -1
  PRINT i; " ..."
  SLEEP 1000, 1         ' Warte 1 Sekunde; nicht unterbrechbar
NEXT
PRINT "START!"
SLEEP
```

Ausgabe

```
Der Countdown laeuft!
 10 ...
 9 ...
 8 ...
 7 ...
 6 ...
 5 ...
 4 ...
 3 ...
 2 ...
 1 ...
START!
```

Sie können auch testweise den zweiten Parameter von **SLEEP** entfernen (also `SLEEP 1000` statt `SLEEP 1000, 1`), um zu testen, wie sich Tastendrücke auf den Programmablauf auswirken.

11.4.3. `FOR i AS datentyp`

Da die Zählvariable oft tatsächlich nur zum Zählen verwendet wird und außerhalb der Schleife keine Bedeutung hat, wird sie häufig nur für die Dauer der Schleife angelegt. Dazu schreibt man, ähnlich wie bei der Variablen-Deklaration mit **`DIM`**, hinter der Zählvariablen ein **`AS`** gefolgt vom gewünschten Datentyp. Das kann z. B. folgendermaßen aussehen:

```
' ausserhalb der Schleife - hier ist i nicht bekannt
FOR i AS INTEGER = 1 TO 10
  ' innerhalb der Schleife - hier ist i bekannt
  PRINT i
NEXT
' wieder ausserhalb - hier ist i nicht mehr bekannt
SLEEP
```

Sobald die Schleife verlassen wird, „vergisst“ das Programm die Variable – sie kann jetzt nicht mehr angesprochen werden (ein Versuch würde zum Compilier-Fehler *Variable not declared* führen). Sie belegt jetzt auch keinen Speicherplatz mehr.

Praktisch ist auch, dass sich der Programmierer keine Gedanken darüber machen muss, ob bereits eine Variable mit demselben Namen existiert und ob ihr Wert später noch gebraucht wird. Das Programm überschreibt einfach kurzfristig die alte Belegung und stellt sie nach Beendigung der Schleife wieder her. Man spricht hier vom Sichtbarkeitsbereich bzw. vom Scope der Variablen: die (neue) Laufzeitvariable ist nur innerhalb der Schleife sichtbar.

Quelltext 11.5: Sichtbarkeit der Zählvariablen

```
DIM AS INTEGER i = 10, k = 20
PRINT "ausserhalb:", i, k
FOR i AS INTEGER = 1 TO 3
  PRINT "innerhalb:", i, k
NEXT
PRINT "ausserhalb:", i, k
SLEEP
```

Ausgabe

```
ausserhalb:    10            20
innerhalb:     1             20
innerhalb:     2             20
innerhalb:     3             20
ausserhalb:    10            20
```

Normalerweise können Sie innerhalb der Schleife auf eine Variable, die außerhalb deklariert wurde, zugreifen (wie hier anhand von `k` demonstriert). Da aber für die Schleife

die Zählvariable `i` neu deklariert wird, ist die alte Belegung von `i` kurzzeitig unsichtbar. Auf sie kann erst nach Beendigung der Schleife wieder zugegriffen werden.

Das Thema „Gültigkeitsbereich von Variablen“ wird noch ausführlich in **??** besprochen.

11.4.4. Übersprungene Schleifen

Um es noch einmal zu präzisieren: Bei einer positiven Schrittweite wird die Schleife verlassen, sobald die Zählvariable größer ist als der Endwert. Bei negativer Schrittweite muss sie sinnvollerweise kleiner werden als der Endwert. Diese Bedingung wird bereits vor dem ersten Durchlauf geprüft. Das bedeutet: Wenn die Schrittweite positiv ist und der Startwert größer ist als der Endwert, dann wird die Schleife überhaupt nicht durchlaufen. Dies kann sinnvoll sein, wenn Start- und/oder Endwert variabel sind (z. B. abhängig von Benutzereingaben). An einem konkreten Beispiel: Wenn Sie aufwärts von `8` bis `3` zählen wollen, gibt es natürlich überhaupt nichts zu zählen. Analog dazu wird die Schleife auch übersprungen, wenn die Schrittweite negativ und der Startwert kleiner als der Endwert ist.

11.4.5. Fallstricke

Gleitkomma-Schrittweite

Es gibt zwei Fallstricke, die Sie beim Verwenden der **`FOR`**-Schleife im Kopf behalten sollten. Die erste betrifft die Verwendung einer Schrittweite kleiner oder gleich `0.5`, wenn zugleich eine Ganzzahl-Laufvariable angegeben wird. Die Schrittweite wird dann zuerst auf eine Ganzzahl gerundet, in diesem Fall also auf `0` abgerundet. Es findet damit effektiv keine Erhöhung der Laufvariablen statt und die Schleife läuft endlos.

Warum überhaupt „Schrittweite kleiner oder gleich `0.5`“? Wird bei `0.5` denn nicht aufgerundet?
Nein, wird es nicht. FreeBASIC rundet, wie auch sehr viele andere Programmiersprachen, nicht kaufmännisch, sondern mathematisch. Das bedeutet: bei `x.5` wird in Richtung *der nächsten geraden Zahl* gerundet. Mit Rundungen werden wir uns in Kapitel 14.1.4 etwas näher beschftigen.

Das Problem mit der Rundung kann leicht vermieden werden, indem Sie immer, wenn Sie eine Gleitkommazahl als Schrittweite verwendet wollen, auch eine Gleitkommazahl als Zählvariable verwenden. Meist ist es sogar besser, auf Gleitkomma-Schrittweiten komplett zu verzichten – denken Sie hier auch an die bereits in Kapitel 6.2.2 erwähnten Probleme mit der Genauigkeit – und bei Bedarf den benötigten Gleitkommawert im Schleifenrumpf zu berechnen.

Quelltext 11.6: FOR: Probleme der Gleitkomma-Schrittweite

```
PRINT "mit Gleitkomma-Schrittweite"
FOR i AS DOUBLE = -1 TO 0 STEP .2
  PRINT i
NEXT
PRINT

PRINT "mit Integer-Schrittweite"
FOR i AS INTEGER = -5 TO 0
  PRINT i/5
NEXT
SLEEP
```

Ausgabe

```
mit Gleitkomma-Schrittweite
-1
-0.8
-0.6000000000000001
-0.4000000000000001
-0.2000000000000001
-5.551115123125783e-17

mit Integer-Schrittweite
-1
-0.8
-0.6
-0.4
-0.2
 0
```

Überschreitung des Wertebereichs

Bei Zählvariablen, die einen ganzzahligen Datentyp besitzen, ist es zudem nicht möglich, bis zum letzten Wert des Wertebereichs zu zählen, da sonst eine Endlosschleife entsteht. Ein Beispiel:[14]

[14] Die Ausgabe kann aus Platzgründen nicht im Buch abgedruckt werden.

Quelltext 11.7: FOR: Probleme mit dem Wertebereich

```
' Achtung: erzeugt eine Endlosschleife!
FOR i AS UBYTE = 0 TO 255
  PRINT i
NEXT
SLEEP
```

Was ist passiert? Nach dem „letzten“ Durchlauf mit `i=255` wird die Zählvariable, wie gewohnt, noch einmal um `1` erhöht. Der neue Wert `256` liegt aber bereits außerhalb des Wertebereichs eines **`UBYTEs`**. In der Variablen `i` wird also stattdessen der Wert `0` gespeichert – und die Schleife fährt munter fort.

Oft gibt es gar keinen triftigen Grund, als Zählvariable einen platzsparenden Datentyp zu verwenden, zumal ein **`INTEGER`** deutlich schneller verarbeitet werden kann. Bei der Verwendung eines **`INTEGERs`** werden Sie andererseits deutlich schwerer an die Grenzen des Wertebereichs stoßen. Sie sehen also, dass man auch diesen Fallstrick in den Griff bekommen kann.

11.5. Kontrollanweisungen

Der reguläre Ablauf einer Schleife kann durch Kontrollanweisungen verändert werden. Zwei Kontrollanweisungen stehen zur Verfügung: mit der ersten springt das Programm vorzeitig an das Ende der Schleife (und wiederholt sie gegebenenfalls), mit der zweiten wird die Schleife sofort verlassen.

11.5.1. Fortfahren mit `CONTINUE`

Die Anweisung **`CONTINUE DO`** bzw. **`CONTINUE WHILE`** oder **`CONTINUE FOR`** springt an das Ende der aktuellen **`DO`**-, **`WHILE`**- bzw. **`FOR`**-Schleife. Es wird dann ganz regulär überprüft, ob die Schleife weiter ausgeführt werden muss oder nicht. Ist die Abbruchbedingung erfüllt, wird die Schleife verlassen, sonst fährt sie mit dem nächsten Durchlauf fort. Selbstverständlich kann **`CONTINUE`** nur innerhalb einer entsprechenden Schleife eingesetzt werden.

In Quelltext 11.8 werden die Zahlen von `1` bis `5` ausgegeben, allerdings soll die `3` und die `5` übersprungen werden. Natürlich macht das Überspringen der `5` programmiertechnisch wenig Sinn (man hätte ja auch gleich nur bis `4` zählen können), aber die Abfrage der Laufvariablen nach Ende der Schleife zeigt sehr schön, dass auch hier ordentlich zum Schleifenende gesprungen und die Variable noch einmal wie gewohnt um `1` erhöht wird.

Quelltext 11.8: CONTINUE FOR

```
DIM AS INTEGER i
FOR i = 1 TO 5
  ' Dieser Bereich wird noch fuer jedes i ausgefuehrt
  IF i = 3 OR i = 5 THEN CONTINUE FOR
  ' Dieser Bereich wird fuer i=3 und fuer i=5 uebersprungen
  PRINT i
NEXT
PRINT "Nach der Schleife: i ="; i
SLEEP
```

Ausgabe

```
 1
 2
 4
Nach der Schleife: i = 6
```

11.5.2. Vorzeitiges Verlassen mit `EXIT`

Um eine Befehlsstruktur vorzeitig zu verlassen, dient die Kontrollanweisung **`EXIT`**, also z. B. **`EXIT DO`** zum Verlassen einer **`DO`**-Schleife. Im Gegensatz zu **`CONTINUE`** gibt es hier jedoch deutlich mehr Strukturen, die diese Anweisung einsetzen können:

1. Schleifen (**`EXIT DO`**, **`EXIT WHILE`**, **`EXIT FOR`**)
2. **`SELECT`**-Blöcke (**`EXIT SELECT`**)
3. Unterprogramme (**`EXIT SUB`**, **`EXIT FUNCTION`**)
4. Blockstrukturen im Zusammenhang mit UDTs (**`EXIT CONSTRUCTOR`**, **`EXIT DESTRUCTOR`**, **`EXIT OPERATOR`**, **`EXIT PROPERTY`**)

Unterprogramme werden noch in Kapitel 12 zur Sprache kommen.

In Quelltext 11.4 wurde ein Countdown vorgestellt, der bis zum bitteren Ende nach unten zählt. Der Benutzer kann das Programm nicht unterbrechen (außer durch das Schließen der Konsole). In der Regel werden über einen längeren Zeitraum laufende Sequenzen, die ohne erkennbaren Grund nicht unterbrochen werden können, von Benutzerseite nicht gern gesehen. Deswegen erlauben wir dem Benutzer jetzt, den Countdown durch einen Tastendruck abzubrechen.

Quelltext 11.9: Countdown mit Abbruchbedingung

```
DIM AS INTEGER i
' Tastenpuffer leeren
DO UNTIL INKEY = ""
LOOP

PRINT "Der Countdown laeuft!"
FOR i = 10 TO 1 STEP -1
  PRINT i; " ..."
  SLEEP 1000
  IF INKEY <> "" THEN EXIT FOR     ' Abbruchmoeglichkeit
NEXT

' War der Start erfolgreich?"
IF i > 0 THEN
  PRINT "*** START ABGEBROCHEN! ***"
ELSE
  PRINT "START!"
END IF
SLEEP
```

Die Funktion **INKEY()** wurde bereits in Kapitel 5.2.2 kurz vorgestellt. Sie ruft, falls vorhanden, eine Taste aus dem Tastenpuffer ab. Um sicherzustellen, dass sich beim Start der Schleife nicht noch Informationen im Puffer befinden, wird dieser in den Zeilen 3 und 4 geleert – er wird einfach so lange abgefragt, bis er leer ist.

Dem **SLEEP** in Zeile 9 wurde sein zweiter Parameter genommen. Dadurch muss sich der Benutzer nach einem Tastendruck nicht noch bis zum Ablauf der Wartesekunde gedulden. Allerdings leert **SLEEP** den Tastaturpuffer nicht, d. h. in Zeile 10 wird die gedrückte Taste dann registriert und die Schleife verlassen.

Interessant ist auch die Abfrage am Ende, in der festgestellt wird, ob der Countdown ordentlich heruntergezählt wurde. Nach dem letzten kompletten Schleifendurchlauf wird die Laufvariable ja noch einmal um 1 reduziert und enthält damit den Wert 0. Wird die Schleife aber im letzten Durchlauf abgebrochen, besitzt die Variable den Wert 1 (bei einem früheren Abbruch natürlich auch einen höheren Wert).

Eine Ausgabe zu Quelltext 11.9 könnte z. B. folgendermaßen aussehen:

Ausgabe

```
Der Countdown laeuft!
 10 ...
 9 ...
 8 ...
 7 ...
*** START ABGEBROCHEN! ***
```

Das Prüfen der Laufvariablen nach Beendigung der Schleife kann z. B. auch hilfreich sein, wenn eine Reihe von Daten durchlaufen werden soll, bis der erste passende Wert gefunden wurde. Außerhalb der Schleife kann dann überprüft werden, ob ein Wert gefunden wurde, und wenn ja, welcher. Diese Möglichkeit einer Abfrage der Laufvariablen außerhalb der Schleife funktioniert selbstverständlich nicht mit einem Konstrukt wie `FOR i AS INTEGER`, da ja in einer solchen Version die Laufvariable nur innerhalb der Schleife existiert und anschließend zerstört wird!

11.5.3. Kontrollstrukturen in verschachtelten Blöcken

Wenn mehrere Schleifen ineinander verschachtelt sind, ist es manchmal erforderlich, aus dem gesamten Schleifenkonstrukt herauszuspringen, also nicht nur die aktuelle Schleife, sondern eine der umgebenden Schleifen zu verlassen. `EXIT FOR` (als Beispiel) verlässt die innerste **FOR**-Schleife, die gefunden werden kann. Wenn sich nun z. B. innerhalb der **FOR**-Schleife eine **DO**-Schleife befindet und Sie darin ein `EXIT FOR` aufrufen, werden natürlich beide Blöcke gleichzeitig verlassen.

```
' Codeschnipsel 1
FOR i as INTEGER = 1 TO 10
  FOR k as INTEGER = 1 TO 10
    EXIT FOR
  NEXT
  ' Hier geht es nach dem "EXIT FOR" weiter
NEXT

' Codeschnipsel 2
FOR i as INTEGER = 1 TO 10
  DO
    EXIT FOR
  LOOP
NEXT
' Hier geht es nach dem "EXIT FOR" weiter
```

Wenn zwei oder mehr gleichartige Blöcke gleichzeitig verlassen werden sollen, kann man den Blocknamen mehrmals durch Komma getrennt angeben. Das sieht dann folgendermaßen aus:

```
' Codeschnipsel 3
FOR i as INTEGER = 1 TO 10
  FOR k as INTEGER = 1 TO 10
    EXIT FOR, FOR
  NEXT
NEXT
' Hier geht es nach dem "EXIT FOR, FOR" weiter

' Codeschnipsel 4
FOR i as INTEGER = 1 TO 10
  FOR k as INTEGER = 1 TO 10
    DO
      FOR l as INTEGER = 1 TO 10
        EXIT FOR, FOR
      NEXT
    LOOP
  NEXT
  ' Hier geht es nach dem "EXIT FOR, FOR" weiter
NEXT
```

Analog dazu können z. B. drei **DO**-Schleifen gleichzeitig durch `EXIT DO, DO, DO` verlassen werden. Die Mehrfach-Angabe ist natürlich auch für alle anderen Strukturen möglich, bei denen Kontrollanweisungen erlaubt sind (**SELECT**-Blöcke, Unterprogramme usw.), und sie kann auch bei **CONTINUE** angewendet werden.

11.6. Systemauslastung in Dauerschleifen

Bisher haben alle in diesem Kapitel verwendeten Schleifen eine sehr überschaubare Laufzeit gehabt, dabei ist ein wichtiger Punkt bisher nicht zur Sprache gekommen. Moderne Betriebssysteme sind Multitasking-Systeme, in denen sich viele Prozesse die zur Verfügung stehenden Ressourcen teilen müssen (selbst wenn Ihr Programm aktuell „das einzige“ aktuell laufende Programm ist, laufen doch auf jeden Fall auch die Steuerungsprozesse des Betriebssystem, aber wahrscheinlich auch eine Menge an anderen Programmen im Hintergrund). Das System regelt das, in dem es der Reihe nach den einzelnen Prozessen Ressourcen zuteilt und diese Prozesse dann immer wieder ihre Ausführung kurz unterbrechen, um die Kontrolle an das System zurückzugeben, damit andere Prozesse nicht zu kurz kommen. Solche Unterbrechungen finden in der Regel im Millisekundenbereich statt.

Sobald Sie einen Befehl wie **INPUT** oder **SLEEP** aufrufen, der auf eine Benutzeraktion wartet, kümmert sich Ihr Programm selbständig um die Unterbrechung der Ausführung. Ansonsten aber werden die Befehle ohne Unterbrechung nacheinander abgearbeitet (was aber von verschiedenen Betriebssystemen unterschiedlich gelöst wird). Wenn Ihr Programm nun sehr kurz ist, spielt das keine Rolle. Wenn die Programmausführung aber länger dauert (und mit „länger“ ist schon der Sekundenbereich gemeint), sollte

man sich Gedanken über die Unterbrechung des Programmablaufs machen, um die Systemauslastung zu reduzieren.

Die einfachste Möglichkeit dazu ist ein simples `SLEEP 1`. Die Unterbrechung ist so kurz, dass sie im Programm nur selten eine Rolle spielt[15], gibt aber dem Betriebssystem die Möglichkeit, auch andere Prozesse zum Zug kommen zu lassen. Besonders in Dauerschleifen ist das mehr als angebracht.

Quelltext 11.10: Dauerschleife mit SLEEP 1

```
' Die Schleife arbeitet bis zum Abbruch ueber die Escape-Taste
DIM AS STRING taste
PRINT "Brechen Sie das Programm mit ESC ab!"
DO
  ' hier koennen verschiedene Anweisungen stehen, die wiederholt werden sollen
  ' anschliessend: Abfrage fuer den Schleifenabbruch
  taste = INKEY
  ' Unterbrechung zur Reduzierung der Systemauslastung
  SLEEP 1
LOOP UNTIL taste = CHR(27)
```

WICHTIG:
Verwenden Sie in Schleifen mit langer Ausführungsdauer immer zumindest `SLEEP 1`, um die Rechnerperformance aufrechtzuerhalten!

11.7. Fragen zum Kapitel

1. Was versteht man unter einer kopf- bzw. fußgesteuerten Schleife?
2. Wann kann eine **DO**-Schleife eingesetzt werden, wann eine **FOR**-Schleife?
3. Was ist eine Endlosschleife?
4. Welche Datentypen eignen sich für die Zählvariable der **FOR**-Schleife?
5. Welche Problemfälle sind bei einer **FOR**-Schleife zu beachten?
6. Welche Möglichkeiten gibt es, in den normalen Ablauf einer Schleife einzugreifen?

[15] Bei aufwendigen Berechnungen werden Sie vermutlich durchaus einen messbaren Unterschied wahrnehmen.

7. In Quelltext 11.2 wurde der Benutzer aufgefordert, eine (beliebig lange) Reihe an Namen einzugeben. Erweitern Sie das Programm: Es soll nach abgeschlossener Eingabe alle eingegebenen Namen wieder ausgeben, und zwar in umgekehrter Reihenfolge.

12. Prozeduren und Funktionen

Bei Prozeduren und Funktionen handelt es sich um sogenannte Unterprogramme. Programmteile, die an mehreren Stellen ausgeführt werden sollen, werden in einen eigenen Bereich ausgelagert. Bei Bedarf springt der Programmablauf in das Unterprogramm, führt es aus und springt wieder zurück.

Der Begriff Unterprogramm ist so zu verstehen, dass der Bereich auch eine völlig eigene Speicherverwaltung besitzt. Auf die Variablen, die im Hauptprogramm definiert sind, kann im Unterprogramm in der Regel nicht zugegriffen werden und umgekehrt. Das ist vorteilhaft, weil sowohl Haupt- als auch Unterprogramm nicht wissen müssen, welche Variablen im jeweils anderen Bereich definiert werden, und sich trotzdem nicht versehentlich in die Quere kommen, indem z. B. das Unterprogramm eine wichtige Variable des Hauptprogrammes überschreibt. Trotzdem können Daten zwischen verschiedenen Programmteilen ausgetauscht werden. Das geschieht in der Regel durch den Einsatz von Parametern.

12.1. Einfache Prozeduren

Eine Prozedur wird in FreeBASIC **SUB**[16] genannt. Jede Prozedur benötigt einen Namen; dabei gelten dieselben Regeln wie für die Namen der Variablen, insb. darf eine Prozedur nicht den Namen einer bereits definierten Variablen erhalten (oder umgekehrt, je nachdem ...). Eine sehr einfache Prozedur könnte folgendermaßen aussehen:

```
SUB halloWelt
  PRINT
  PRINT TAB(20); "| Hallo Welt!"
  PRINT TAB(20); "========================="
END SUB
```

Beginn und Ende der Prozedur wird durch **SUB** und **END SUB** definiert. Hinter dem **SUB** folgt der Name der Prozedur, in diesem Fall `halloWelt`. Diese erste Zeile wird auch *Kopf* der Prozedur genannt. Die drei Zeilen zwischen **SUB** und **END SUB** stellen den *Rumpf* der Prozedur dar.

[16] aus dem Lateinischen und auch im Englischen: *sub* = unter

Wenn Sie das Programm abtippen und starten, sehen Sie – dass Sie nichts sehen. Und das liegt nicht nur an einem fehlenden **SLEEP**. Die Prozedur wurde zwar definiert und steht von nun an im Programm zur Verfügung, sie wurde aber bisher noch nicht aufgerufen. Nach der Definition der Prozedur wird `halloWelt` genauso wie ein FreeBASIC-Befehl behandelt und kann dementsprechend an beliebiger Stelle des Programmes aufgerufen werden.

Quelltext 12.1: Hallo Welt als Prozedur

```
SUB halloWelt
  PRINT
  PRINT TAB(20); "| Hallo Welt!"
  PRINT TAB(20); "========================="
END SUB

halloWelt
PRINT "Gut, dass es dich gibt."
PRINT "Daher gruesse ich dich gleich noch einmal:"
halloWelt
SLEEP
```

Ausgabe

```
                    | Hallo Welt!
                    =========================
Gut, dass es dich gibt.
Daher gruesse ich dich gleich noch einmal:

                    | Hallo Welt!
                    =========================
```

12.2. Verwaltung von Variablen

Wie eingangs erwähnt, kann auf Variablen, die im Hauptprogramm deklariert wurden, im Unterprogramm nicht zugegriffen werden und umgekehrt. Dadurch läuft das Unterprogramm in einer gesicherten Umgebung, und es kann nicht passieren, dass versehentlich der Ablauf des Hauptprogrammes durcheinander gebracht wird. Wie dennoch eine Kommunikation zwischen beiden Programmteilen stattfinden kann, werden wir uns in diesem Abschnitt ansehen.

12.2.1. Parameterübergabe

Die in Quelltext 12.1 genutzte Möglichkeit, die Hallo-Welt-Ausgabe zu formatieren, ist ja schon einmal recht praktisch – immerhin sind in Zukunft statt drei Zeilen Code nur noch eine Zeile nötig. Allerdings will man vermutlich nicht so häufig die Welt grüßen, sondern stattdessen möglicherweise einen anderen Text ausgeben. Diesen Text werden wir jetzt als Parameter übergeben.

Parameter sind Werte, die beim Aufruf der Prozedur an das Unterprogramm übergeben werden und dort in einer Variablen zur Verfügung stehen. Bei der Definition der Prozedur wird hinter dem Prozedurnamen in Klammern die Parameterliste übergeben. Das ist eine durch Komma getrennte Liste aller Parameter, die beim Aufruf der Prozedur übergeben werden müssen, einschließlich Datentyp. Die Prozedur aus Quelltext 12.1 soll nun so verwendet werden, dass sowohl der auszugebende Text als auch die Einrückung vor der Ausgabe von Aufruf zu Aufruf variiert werden kann. Außerdem wird für die Prozedur auch noch ein passenderer Name ausgewählt.

Quelltext 12.2: Prozedur mit Parameterübergabe

```
SUB ueberschrift(text AS STRING, einrueckung AS INTEGER)
  PRINT
  PRINT TAB(einrueckung); "| "; text
  PRINT TAB(einrueckung); "=========================="
END SUB

ueberschrift("1. Unterprogramme", 1)
PRINT "Ein Unterprogramm kann von jedem beliebigen Programmpunkt aus"
PRINT "aufgerufen werden."
ueberschrift("1.1 Parameteruebergabe", 5)
PRINT "An ein Unterprogramm koennen auch Parameter uebergeben werden."
ueberschrift("1.2 Nutzung der Parameter", 5)
PRINT "Parameter sind innerhalb des gesamten Unterprogrammes gueltig."
SLEEP
```

Die beiden übergebenen Parameter – beim Prozedur-Aufruf in Zeile 7 ist es die Zeichenkette `"1. Unterprogramme"` und die Zahl `1` – werden innerhalb der Prozedur den Variablen `text` und `einrueckung` zugewiesen und können nun genutzt werden. Jedoch nur innerhalb der Prozedur; außerhalb sind die beiden Variablen dem Programm nicht bekannt.

Ausgabe

```
| 1. Unterprogramme
========================
Ein Unterprogramm kann von jedem beliebigen Programmpunkt aus
aufgerufen werden.

    | 1.1 Parameteruebergabe
    ========================
An ein Unterprogramm koennen auch Parameter uebergeben werden.

    | 1.2 Nutzung der Parameter
    ========================
Parameter sind innerhalb des gesamten Unterprogrammes gueltig.
```

Beim Aufruf einer Prozedur müssen die Parameter nicht in Klammern stehen (sehr wohl aber bei der Definition). Möglich ist also auch ein Aufruf in der Form:

```
ueberschrift "1.2 Nutzung der Parameter", 5
```

Ob die Klammern gesetzt werden oder nicht, ist weitgehend Geschmacksache – ohne Klammern entspricht es eher der gewohnten BASIC-Syntax, während in vielen anderen Programmiersprachen die Klammern grundsätzlich gesetzt werden müssen und eine Reihe an Programmierern daher die Klammern auch in FreeBASIC bevorzugt. Die Klammern sind aber nur beim Prozedur*aufruf* optional; im Prozedurkopf müssen sie gesetzt werden.

Der Vorteil an der Prozedur ist, neben der Einsparung von zwei Zeilen bei jedem Aufruf, dass eine Änderung nur noch an einer einzigen Stelle vorgenommen wird anstatt an vielen verschiedenen Stellen. Nehmen wir etwa an, wir entschließen uns irgendwann, alle Überschriften gelb zu schreiben. Dazu müssen wir lediglich die Prozedur anpassten; im Hauptprogramm kann alles so bleiben, wie es ist.

```
SUB ueberschrift(text AS STRING, einrueckung AS INTEGER)
  COLOR 14        ' Schriftfarbe gelb
  PRINT
  PRINT TAB(einrueckung); "| "; text
  PRINT TAB(einrueckung); "========================="
  COLOR 15        ' zurueck zu Schriftfarbe weiss
END SUB
```

Diese Anpassung an jeder Stelle vorzunehmen, an der eine solche Überschrift angezeigt werden soll, wäre deutlich mühsamer – abgesehen davon, dass eine erhöhte Gefahr besteht, eine Stelle zu vergessen.

Hinweis:
Auch wenn Prozeduren eine eigene Verwaltung ihrer Variablen besitzen, können sie dennoch Seiteneffekte aufweisen. Eine Änderung der Schriftfarbe innerhalb der Prozedur etwa bleibt auch außerhalb der Prozedur erhalten.

Grundsätzlich kann jeder Datentyp an eine Prozedur übergeben werden, auch UDTs und Pointer. Bei Pointern wird an den Datentypen ein **PTR** angehängt, wie es auch schon von der Variablendeklaration bekannt ist. Sogar ganze Prozeduren und Funktionen können als Parameter übergeben werden.

12.2.2. Globale Variablen

Innerhalb eines Unterprogrammes können wie gewohnt mit **DIM** neue Variablen deklariert werden. Es handelt sich bei ihnen um sogenannte *lokale Variablen*, die nur in der Umgebung gültig sind, in der sie definiert wurden (also innerhalb des Unterprogrammes). Aber auch im Hauptprogramm definierte Variablen sind lokal – sie gelten nur innerhalb des Hauptprogrammes. Allerdings bietet FreeBASIC auch die Möglichkeit, globale Variablen zu deklarieren, auf die sowohl im Hauptprogramm als auch in allen Unterprogrammen uneingeschränkt zugegriffen werden kann. Diese erhalten bei der Deklaration das zusätzliche Schlüsselwort **SHARED**.

```
DIM SHARED AS datentyp variable1, variable2
DIM SHARED variable1 AS datentyp1, variable2 AS datentyp2
```

Die erste Zeile deklariert wieder mehrere Variablen desselben Datentyps, während die Variablen in der zweiten Zeile unterschiedliche Datentypen besitzen dürfen. **SHARED** kann jedoch nur im Hauptprogramm angegeben werden, nicht in Unterprogrammen. Das bedeutet kurz gesagt: Wenn Sie in einem Unterprogramm globale Variablen verwenden wollen, müssen diese im Hauptprogramm mit **SHARED** deklariert werden.

Gerade wenn Sie eine Variable in nahezu allen Ihren Unterprogrammen benötigen, ist **SHARED** ein praktisches Mittel, eine ständige Übergabe als Parameter zu umgehen. Bei Unachtsamkeit kann es jedoch schnell zu großen Problemen kommen. Quelltext 12.3 demonstriert ein solches Problem: In diesem Beispiel wird davon ausgegangen, dass die Laufvariable `i` in so vielen Unterprogrammen benötigt wird, dass sie als globale Variable deklariert werden kann. Warum das keine gute Idee ist, sieht man in der Ausgabe.

Quelltext 12.3: Probleme mit unachtsamer Verwendung von SHARED

```
DIM SHARED i AS INTEGER         ' i ist jetzt global (Haupt- und Unterprogramme)

SUB schreibeSumme1bisX(x AS INTEGER)
  DIM AS INTEGER summe = 0     ' summe ist jetzt lokal (nur im Unterprogramm)
  FOR i = 1 TO x
    summe += i
  NEXT
  PRINT summe
END SUB

PRINT "Berechne die Summe aller Zahlen von 1 bis ..."
FOR i = 1 TO 5
  PRINT i; ": ";
  schreibeSumme1bisX i
NEXT
SLEEP
```

Ausgabe

```
Berechne die Summe aller Zahlen von 1 bis ...
 1:  1
 3:  6
 5:  15
```

Was ist passiert? Im Hauptprogramm startet die Schleife mit `i=1`. Während des ersten Schleifendurchlaufs wird das Unterprogramm aufgerufen und auch dort eine Schleife durchlaufen – in diesem Fall von `1` bis `1`. Wir erinnern uns, dass *nach* dem Durchlauf die Laufvariable noch einmal erhöht wird; sie hat nun also den Wert `2`. Mit diesen Voraussetzungen kehrt das Programm aus der Prozedur zurück in die Schleife des Hauptprogrammes. `i` hat sich also während des Aufrufs der Prozedur verändert, und mit diesem veränderten Wert arbeitet die Schleife nun weiter.

Quelltext 12.3 arbeitet also nicht alle Werte ab, die es eigentlich hätte abarbeiten sollen. Es hätte bei „geeigneter“ Programmierung der Prozedur sogar passieren können, dass sich das Programm in einer Endlosschleife verfängt und immer dieselben Werte ausgibt. Bei Verzicht auf die globale Variable `i` wäre das Problem gar nicht aufgetaucht. Natürlich muss dann `i` in der Prozedur neu deklariert werden.

Quelltext 12.4: Korrekte Berechnung aller Summen

```
DIM i AS INTEGER                ' i ist jetzt lokal (Hauptprogramm)

SUB schreibeSumme1bisX(x AS INTEGER)
  DIM AS INTEGER summe = 0, i  ' summe und i sind lokal (Unterprogramm)
  FOR i = 1 TO x
    summe += i
  NEXT
  PRINT summe
END SUB

PRINT "Berechne die Summe aller Zahlen von 1 bis ..."
FOR i = 1 TO 5
  PRINT i; ": ";
  schreibeSumme1bisX i
NEXT
SLEEP
```

Ausgabe

```
Berechne die Summe aller Zahlen von 1 bis ...
 1:  1
 2:  3
 3:  6
 4:  10
 5:  15
```

Im Übrigen sind solche Fallstricke der Grund, warum die Laufvariablen in diesem Buch meist im Schleifenkopf neu deklariert werden (`FOR i AS INTEGER`). Auch dadurch lässt sich sicherstellen, dass sie nicht versehentlich in Konflikt mit einer gleichnamigen Variablen gerät.

Selbst wenn Sie eine Variable global deklarieren, können Sie sie in einem Unterprogramm oder einer anderen Blockstruktur neu deklarieren. In diesem Fall „vergisst“ das Programm die globale Variable solange, bis die Blockstruktur verlassen wird.

Noch eine letzte Anmerkung: Im Hauptprogramm definierte Konstanten gelten global, sind also auch im Unterprogramm verfügbar. Da Konstanten nicht mehr verändert werden können, kann es hier auch nicht zum Konflikt zwischen zwei Programmteilen kommen.

12.2.3. Statische Variablen

Statische Variablen sind ein weiteres Konzept, das bei Unterprogrammen zum Tragen kommen kann. Man versteht darunter Variablen, die wie lokale Variablen nur innerhalb des Unterprogrammes gültig sind, deren Wert aber nach Beendigung des Unterprogrammes

nicht verloren geht, sondern gespeichert wird. Wenn dasselbe Unterprogramm erneut aufgerufen wird, erhält die Variable den zuletzt gespeicherten Wert wieder zurück. In einem einfachen Beispiel soll eine statische Variable eingesetzt werden, um lediglich zu zählen, wie oft die Prozedur bereits aufgerufen wurde.

Quelltext 12.5: Statische Variable in einem Unterprogramm

```
SUB zaehler
  STATIC AS INTEGER i = 0
  i += 1
  PRINT "Das ist der"; i; ". Aufruf der Prozedur."
END SUB

' Prozedur dreimal aufrufen
zaehler
zaehler
PRINT "und ein letztes Mal:"
zaehler
SLEEP
```

Ausgabe

```
Das ist der 1. Aufruf der Prozedur.
Das ist der 2. Aufruf der Prozedur.
und ein letztes Mal:
Das ist der 3. Aufruf der Prozedur.
```

Um eine statische Variable zu deklarieren, wird das Schlüsselwort **STATIC** statt **DIM** verwendet. Hier besteht die einzige Möglichkeit, der Variablen einen Initialwert zuzuweisen (tut man es nicht, wird die Variable standardmäßig mit dem Wert 0 belegt). Diese Zuweisung wird nur beim allerersten Aufruf der **STATIC**-Zeile durchgeführt und später ignoriert – aus diesem Grund wird in Quelltext 12.5 `i` nicht ständig wieder auf 0 gesetzt. Sie können die Auswirkung des Initialwerts gern testen, indem Sie den Wert verändern. Spätere Zuweisungen erfolgen dagegen bei jedem Durchlauf, weshalb folgende Variante nicht wie gewünscht funktioniert:

```
' ...
  STATIC AS INTEGER i
  i = 0
' ...
```

Da `i` bei jedem Durchlauf gleich nach der Deklaration auf `0` gesetzt wird, ist der eigentliche Zweck, den Wert zwischen den Prozedur-Aufrufen zu speichern, hinfällig.

Wenn alle im Unterprogramm deklarierten Variablen statisch sein sollen, kann man der Einfachheit halber das Unterprogramm selbst als statisch deklarieren. Quelltext 12.5

lässt sich dann auch folgendermaßen schreiben:

Quelltext 12.6: Statisches Unterprogramm

```
SUB zaehler STATIC
  DIM AS INTEGER i = 0
  i += 1
  PRINT "Das ist der"; i; ". Aufruf der Prozedur."
END SUB

' Prozedur dreimal aufrufen
zaehler
zaehler
PRINT "und ein letztes Mal:"
zaehler
SLEEP
```

STATIC steht in dieser Version ganz am Ende der Kopfzeile des Unterprogrammes – also ggf. hinter der Parameterliste. Die Variablendeklaration kann nun über **DIM** erfolgen (allerdings ist auch **STATIC** möglich), und wieder wird der Initialwert nur beim ersten Durchlauf berücksichtigt.

Hinweis:
Da statische Variablen ständig im Speicher bereitgehalten werden müssen, sollten Sie nur solche Variablen als statisch deklarieren, die tatsächlich statisch sein sollen.

12.3. Unterprogramme bekannt machen

Wie bereits gesagt wurde, muss ein Unterprogramm im Programm bekannt sein, bevor es aufgerufen werden kann. Wenn die Prozedur, wie in den obigen Beispielen, ganz zu Beginn des Programmes definiert wird, ist sie anschließend auch verfügbar. Es gibt jedoch einige Fälle, in denen das Definieren „ganz am Anfang“ nicht möglich ist.

12.3.1. Die Deklarationszeile

Jedes Unterprogramm vor seinem ersten Aufruf zu definieren, stößt sehr schnell an eine logische Grenze. Wenn das Unterprogramm A das Unterprogramm B aufruft, muss Unterprogramm B zuerst definiert werden, um beim Aufruf in Unterprogramm A bereits bekannt zu sein. Was passiert nun aber, wenn sich beide Unterprogramme gegenseitig aufrufen wollen?

An dieser Stelle kommt die **DECLARE**-Zeile ins Spiel. Sie ist vom Aufbau identisch mit der Kopfzeile des Unterprogrammes, nur mit vorangestelltem Schlüsselwort **DECLARE**. Allerdings wird der Inhalt des Unterprogrammes an dieser Stelle noch nicht festgelegt. Die **DECLARE**-Zeile dient nur dazu, dem Compiler mitzuteilen: Irgendwann später wird die Definition eines Unterprogrammes mit diesem Namen und dieser Parameterliste folgen. Ein Aufruf kann dann auch schon erfolgen, bevor das Unterprogramm festgelegt wird – der Compiler kennt ja bereits seinen Namen.

Quelltext 12.7: Deklarieren einer Prozedur

```
DECLARE SUB ueberschrift(text AS STRING, einrueckung AS INTEGER)

ueberschrift("1. Unterprogramme", 1)
PRINT "Ein Unterprogramm kann von jedem Programmpunkt aus aufgerufen werden."
ueberschrift("1.1 Parameteruebergabe", 5)
PRINT "An ein Unterprogramm koennen auch Parameter uebergeben werden."
ueberschrift("1.2 Nutzung der Parameter", 5)
PRINT "Parameter sind innerhalb des gesamten Unterprogrammes gueltig."
SLEEP

SUB ueberschrift(text AS STRING, einrueckung AS INTEGER)
  PRINT
  PRINT TAB(einrueckung); "| "; text
  PRINT TAB(einrueckung); "========================="
END SUB
```

Quelltext 12.7 bewirkt dasselbe wie Quelltext 12.2. In der ersten Zeile wird die Prozedur deklariert (aber noch nicht definiert, d. h. noch nicht inhaltlich festgelegt). Die Definition der Prozedur kann nun (fast) an beliebiger Stelle des Programmes stattfinden: zu Beginn (nach der **DECLARE**-Zeile) oder ganz am Ende wie in Quelltext 12.7 – sogar irgendwann mitten im Hauptprogramm, was aber im Sinne der Übersichtlichkeit keinesfalls empfehlenswert wäre. Eine sehr häufige Vorgehensweise ist die Deklaration möglichst früh im Programm und die Definition der Unterprogramme ganz am Ende.

Nun lassen sich auch Prozeduren realisieren, die sich wechselseitig aufrufen.[17] Quelltext 12.8 ist kein übermäßig sinnvolles Programm, sollte aber das Prinzip veranschaulichen.

[17] Man spricht hier von einer verschränkten Rekursion.

Quelltext 12.8: PingPong

```
DECLARE SUB ping(anzahl AS INTEGER)
DECLARE SUB pong(anzahl AS INTEGER)

ping 3

SUB ping(anzahl AS INTEGER)
  PRINT "ping ist bei"; anzahl
  IF anzahl < 1 THEN                  ' Abbruchbedingung, damit der gegen-
    PRINT "Kein Aufruf von pong"      ' seitige Aufruf nicht ewig laeuft
  ELSE
    PRINT "pong wird aufgerufen"
    pong anzahl-1
  END IF
END SUB
SUB pong(anzahl AS INTEGER)
  PRINT "pong ist bei"; anzahl
  IF anzahl < 1 THEN
    PRINT "Kein Aufruf von ping"
  ELSE
    PRINT "ping wird aufgerufen"
    ping anzahl-1
  END IF
END SUB
```

Ausgabe

```
ping ist bei 3
pong wird aufgerufen
pong ist bei 2
ping wird aufgerufen
ping ist bei 1
pong wird aufgerufen
pong ist bei 0
Kein Aufruf von ping
```

12.3.2. Optionale Parameter

Uns sind bisher bereits FreeBASIC-eigene Anweisungen begegnet, bei denen nicht alle Parameter angegeben werden mussten; z. B. bei **LOCATE** und **COLOR**. Solche optionalen Parameter sind auch bei eigenen Unterprogrammen möglich. Dazu muss dem Compiler mitgeteilt werden, welcher Wert für den Parameter verwendet werden soll, wenn dieser beim Aufruf nicht mit angegeben wurde. Beispielsweise könnte man in Quelltext 12.2 einen zusätzlichen Parameter für die Überschriftsfarbe einführen. Wird sie nicht angegeben,

verwendet die Prozedur den Standardwert 14 (gelb).

Quelltext 12.9: Optionale Parameter

```
DECLARE SUB ueberschrift(txt AS STRING, einrueck AS INTEGER, farbe AS INTEGER = 14)

ueberschrift "1. Unterprogramme",      1, 10  ' Ueberschrift in gruen
ueberschrift "1.1 Parameteruebergabe", 5      ' Ueberschrift in gelb
SLEEP

SUB ueberschrift(text AS STRING, einrueckung AS INTEGER, farbe AS INTEGER = 14)
  COLOR farbe     ' angegebene Schriftfarbe
  PRINT
  PRINT TAB(einrueckung); "| "; text
  PRINT TAB(einrueckung); "========================="
  COLOR 15        ' zurueck zu Schriftfarbe weiss
END SUB
```

Die Wertzuweisung für den bzw. die optionalen Parameter muss sowohl in der **DECLARE**-Zeile als auch in der Kopfzeile des Unterprogrammes stehen, und natürlich sollte in beiden Zeilen derselbe Wert angegeben werden.

Optionale Parameter bieten sich vor allem am Ende der Parameterliste an, weil sie dort am einfachsten weggelassen werden können. Um einen optionalen Parameter auszulassen, der mitten in der Liste steht, müssen (genauso wie z. B. beim Auslassen des ersten Parameters von **COLOR**) die korrekte Anzahl an Kommata gesetzt werden.

Hinweis:
In Quelltext 12.9 wurden in der **DECLARE**-Zeile zwei Parameternamen gekürzt, um den Seitenrand nicht zu sprengen. Die Parameternamen in der **DECLARE**-Zeile müssen nicht mit denen des Prozedurkopfs übereinstimmen (sie könnten sogar ganz weggelassen werden). Für die Variablen im Funktionsrumpf sind lediglich die Namen im Kopf entscheidend. Wichtig ist aber, dass in der **DECLARE**-Zeile die korrekten Datentypen angegeben werden.

12.3.3. OVERLOAD

Häufig benötigt man zwei oder mehrere völlig verschiedene Parameterlisten, will jedoch bei einem einheitlichen Namen für das Unterprogramm bleiben. Wir wollen eine weitere Überschrift-Prozedur bauen, bei der die Einrückung nicht durch einen Zahlenwert, sondern durch einen Einrückungs-String festgelegt werden soll, der dunkelgrau ausgegeben wird (das Beispiel ist zugegebenermaßen etwas konstruiert, ist aber zur Veranschaulichung gut

geeignet). Die Prozedur soll weiterhin `ueberschrift` heißen, da es sich ja prinzipiell um dieselbe Art von Anweisung handelt.

```
SUB ueberschrift(text AS STRING, einrueckung AS STRING, farbe AS INTEGER = 14)
  PRINT
  COLOR 8     : PRINT einrueckung;    ' Einrueckungs-String in dunklem Grau
  COLOR farbe : PRINT "| "; text      ' Text in angegebener Schriftfarbe
  COLOR 8     : PRINT einrueckung;
  COLOR farbe : PRINT "========================="
  COLOR 15        ' zurueck zu Schriftfarbe weiss
END SUB
```

Wenn Sie nun beide Versionen der Prozedur untereinander schreiben und zu compilieren versuchen, erhalten Sie die Fehlermeldung:

```
error 4: Duplicated definition
```

Auch ein Hinzufügen der **DECLARE**-Zeilen behebt das Problem noch nicht. Sie müssen dem Compiler mitteilen, dass er mehrere Unterprogramme desselben Namens zu erwarten hat. Dazu fügen Sie in der ersten **DECLARE**-Zeile hinter dem Prozedurnamen das Schlüsselwort **OVERLOAD** (zu deutsch: Überladung) hinzu. Egal, wie viele Unterprogramme mit gleichem Namen Sie letztlich bereitstellen wollen: Das Schlüsselwort **OVERLOAD** muss und darf nur in der **DECLARE**-Zeile des ersten dieser Unterprogramme auftauchen.

Da bei uns beide Überschrifts-Prozeduren größtenteils dasselbe machen, lohnt es sich übrigens, die Arbeit der zweiten Prozedur in die erste auszulagern.

Quelltext 12.10: Überladene Prozeduren (OVERLOAD)

```
DECLARE SUB ueberschrift OVERLOAD(t AS STRING, e AS INTEGER, f AS INTEGER = 14)
DECLARE SUB ueberschrift(txt AS STRING, einrueck AS STRING, farbe AS INTEGER = 14)

ueberschrift "1. Unterprogramme",      1, 10    ' Ueberschrift in gruen
ueberschrift "1.1 Parameteruebergabe", "~~~~~"  ' Ueberschrift in gelb
SLEEP

SUB ueberschrift(text AS STRING, einrueckung AS INTEGER, farbe AS INTEGER = 14)
  ueberschrift text, SPACE(einrueckung), farbe
  ' SPACE(x) erzeugt einen String, der aus x Leerzeichen besteht.
  ' Diese Prozedur macht also nichts weiter, als die folgende Prozedur
  ' anzuweisen, zur Einrueckung Leerzeichen zu verwenden.
END SUB
SUB ueberschrift(text AS STRING, einrueckung AS STRING, farbe AS INTEGER = 14)
  PRINT
  COLOR 8     : PRINT einrueckung;    ' Einrueckungs-String in dunklem Grau
  COLOR farbe : PRINT "| "; text      ' Text in angegebener Schriftfarbe
  COLOR 8     : PRINT einrueckung;
  COLOR farbe : PRINT "========================="
  COLOR 15        ' zurueck zu Schriftfarbe weiss
END SUB
```

Hinweis: Die andere Parameterbezeichnung in der ersten **DECLARE**-Zeile wurde nur deshalb so gewählt, dass der Quelltext nicht den Seitenrahmen sprengt.

Der erste `ueberschrift`-Aufruf in Zeile 4 springt in das erste der beiden Unterprogramme, der zweite Aufruf in Zeile 5 dagegen in das zweite. Dies ist klar festgelegt, da sich der Datentyp des zweiten Parameters unterscheidet und daher immer nur eine der beiden Prozeduren infrage kommt. Bei der Verwendung von **OVERLOAD** ist immer darauf zu achten, dass der Compiler zweifelsfrei entscheiden kann, wann er welches der Unterprogramme aufzurufen hat. Das geschieht durch eine unterschiedliche Parameterzahl und/oder durch verschiedene Datentypen. Der Name des Unterprogramms zusammen mit der Parameterliste wird auch als *Signatur* des Unterprogramms bezeichnet – diese Signatur muss innerhalb eines Programmes eindeutig sein.

Hinweis:
Achten Sie auch bei gleichzeitiger Verwendung von **OVERLOAD** und optionalen Parametern darauf, dass die Eindeutigkeit gewahrt bleibt.

12.4. Funktionen

Funktionen sind, wie Prozeduren, Unterprogramme, nur mit einem Unterschied: Eine Funktion gibt einen Wert zurück. Die Stelle mit dem Funktionsaufruf wird dann durch diesen Rückgabewert ersetzt. Der Funktionsaufruf kann daher an jeder Stelle stehen, an der überhaupt ein Wert mit dem Datentyp des Rückgabewerts stehen kann.

Als Beispiel soll die Funktion `mittelwert` definiert werden, der zwei Zahlen übergeben werden; die Funktion gibt dann den Wert zurück. der in der Mitte beider Parameter liegt (arithmetisches Mittel). Zuerst müssen wir uns entscheiden, welche Datentypen wir verwenden wollen; zunächst einmal erlauben wir für die Parameter nur die Übergabe von **INTEGER**-Werten. Der Mittelwert kann natürlich eine Gleitkommazahl sein. Einfache Genauigkeit reicht jedoch aus, da als Nachkomma-Anteil nur `.0` oder `.5` herauskommen kann. Die Deklaration der Funktion sieht dann so aus:

```
DECLARE FUNCTION mittelwert(a AS INTEGER, b AS INTEGER) AS SINGLE
```

Statt **SUB** steht jetzt **FUNCTION**, und am Ende hinter der schließenden Klammer ist noch `AS datentyp` nötig, womit der Datentyp des Rückgabewerts angegeben wird. Innerhalb der Funktion muss der Rückgabewert nun noch festgelegt werden. Das kann z. B. mit **RETURN** geschehen.

```
FUNCTION mittelwert(a AS INTEGER, b AS INTEGER) AS SINGLE
  RETURN (a + b) / 2
END FUNCTION
```

Mit **RETURN** geschieht zweierlei: Einerseits wird der Rückgabewert zugewiesen, andererseits wird die Funktion verlassen. Weitere Zeilen hinter dem **RETURN** würden also nicht mehr ausgeführt werden. Es gibt noch zwei weitere Möglichkeiten, den Rückgabewert zuzuweisen: durch die Anweisung `FUNCTION = (a + b) / 2` und durch `mittelwert = (a + b) / 2`. Die erste Version ist ein feststehender Ausdruck, während in `mittelwert =...` ggf. der Funktionsname angepasst werden muss. In beiden Fällen wird die Funktion zunächst noch *nicht* verlassen. Diese beiden Zuweisungsmöglichkeiten sind historisch bedingt; sie werden aus Kompatibilitätsgründen weiter unterstützt, aber nicht mehr häufig eingesetzt.

Es fehlt noch ein Aufruf der Funktion, und das Beispielprogramm ist fertig. Da ein **SINGLE**-Wert zurückgegeben wird, kann dieser in einer passenden Variablen gespeichert oder aber gleich mit **PRINT** ausgegeben werden.

Quelltext 12.11: Arithmetisches Mittel zweier Werte

```
DECLARE FUNCTION mittelwert(a AS INTEGER, b AS INTEGER) AS SINGLE
DIM AS INTEGER w1, w2

INPUT "Geben Sie, durch Komma getrennt, zwei Ganzzahlen ein: ", w1, w2
PRINT "Der Mittelwert von"; w1; " und"; w2; " ist"; mittelwert(w1, w2)
SLEEP

FUNCTION mittelwert(a AS INTEGER, b AS INTEGER) AS SINGLE
  RETURN (a + b) / 2
END FUNCTION
```

Ausgabe

```
Geben Sie, durch Komma getrennt, zwei Ganzzahlen ein: 10,3
Der Mittelwert von 10 und 3 ist 6.5
```

Sämtliche Möglichkeiten, die für Prozeduren zur Verfügung stehen, gibt es auch für Funktionen; der einzige Unterschied zwischen diesen beiden Arten von Unterprogrammen ist der, dass die Funktion einen Wert zurückgeben kann (bzw. muss). Wenn innerhalb der Funktion keine Zuweisung des Rückgabewerts erfolgt, gibt der Compiler eine Warnung aus. Allerdings wird dazu keine Laufzeitüberprüfung durchgeführt, d. h. der Compiler prüft lediglich, ob im Funktionsrumpf eine Zuweisung wie etwa `RETURN wert` vorkommt. Ob diese Zuweisung dann im speziellen Fall tatsächlich *aufgerufen* wird, kann er nicht überprüfen.

Der Rückgabewert einer Funktion kann auch verworfen werden. Dazu wird die Funktion genauso aufgerufen wie eine Prozedur, also ohne dass ihr Rückgabewert in einer Variablen gespeichert oder anderweitig ausgewertet wird.

Hinweis:
Während beim Aufruf von Prozeduren keine Klammern um die Parameterliste gesetzt werden müssen, sind Klammern bei Funktionen unbedingt erforderlich – außer Sie verwerfen den Rückgabewert.

12.5. Weitere Eigenschaften der Parameter

12.5.1. Übergabe von Arrays

Die Übergabe einer großen Datenmenge – insbesondere auch, wenn die Anzahl der Elemente nicht von vornherein feststeht – kann recht bequem über ein Array erfolgen. Dazu muss in der Parameterliste an den Array-Namen lediglich ein Klammer-Paar angehängt werden – allerdings ohne Angabe der Array-Grenzen. Es ist dabei egal, wie viele Dinensionen das Array besitzt; die Übergabe erfolgt immer nur durch den Array-Namen mit dem angehängten Klammer-Paar.

Den Mittelwert aller Werte eines Arrays könnte man folgendermaßen berechnen:

Quelltext 12.12: Arithmetisches Mittel aller Werte eines Arrays

```
DECLARE FUNCTION mittelwert(w() AS INTEGER) AS SINGLE

DIM AS INTEGER werte(...) = { 25, 412, -19, 32, 112 }
PRINT "Der Mittelwert der festgelegten Werte ist "; mittelwert(werte())

FUNCTION mittelwert(w() AS INTEGER) AS SINGLE
  IF UBOUND(w) < LBOUND(w) THEN RETURN 0  ' Fehler: Array nicht dimensioniert
  DIM AS INTEGER summe = 0
  FOR i AS INTEGER = LBOUND(w) TO UBOUND(w)
    summe += w(i)
  NEXT
  RETURN summe / (UBOUND(w) - LBOUND(w) + 1)
END FUNCTION
```

Ausgabe

```
Der Mittelwert der festgelegten Werte ist 112.4
```

Die Funktion hat den Vorteil, dass Start- und Endwert nicht festgelegt sind. Sie reagiert sehr flexibel auf das übergebene Array. Zumindest funktioniert das gut, solange sichergestellt ist, dass ein *eindimensionales* Array übergeben wird. Die Funktion erkennt die Anzahl der Dimensionen nicht. Bei Übergabe eines mehrdimensionalen Array wird kein Compilerfehler erzeugt, sondern die Funktion behandelt die im Array-Speicher liegenden Werte so, als ob es ein eindimensionales Array wäre. Das zurückgegebene Resultat ist dann mit großer Wahrscheinlichkeit sinnlos.

Natürlich kann die Funktion vor dem Start der Berechnung prüfen, ob die Anzahl der Dimensionen `UBOUND(w, 0)=1` beträgt. Man kann aber auch den Compiler anweisen, nur eine bestimmte Anzahl an Dimensionen zuzulassen. Bei einer falschen Dimensionenzahl würde das Programm dann gar nicht compilieren. Wenn man diese Möglichkeit nutzen will, muss für jede Dimension ein **ANY** angegeben werden.

```
DECLARE FUNCTION f1(w(ANY) AS datentyp) AS datentyp       ' nur eindimensional
DECLARE FUNCTION f2(w(ANY, ANY) AS datentyp) AS datentyp  ' nur zweidimensional
' und so weiter
```

Analog dazu muss dann auch die Kopfzeile des Unterprogrammes mit der korrekten Anzahl an **ANY** versehen werden. Nun kann an die Funktion `f1` nur noch ein eindimensionales Array übergeben werden und an die Funktion `f2` nur noch ein zweidimensionales Array; eine falsche Dimensionenzahl führt zu einem Compilierfehler.

Als komplettes Programm könnte das dann so aussehen:

Quelltext 12.13: Arithmetisches Mittel für ein eindimensionales(!) Array

```
DECLARE FUNCTION mittelwert(w(ANY) AS INTEGER) AS SINGLE

DIM AS INTEGER werte(...) = { 25, 412, -19, 32, 112 }
PRINT "Der Mittelwert der festgelegten Werte ist "; mittelwert(werte())

FUNCTION mittelwert(w(ANY) AS INTEGER) AS SINGLE
  IF UBOUND(w) < LBOUND(w) THEN RETURN 0  ' Fehler: Array nicht dimensioniert
  DIM AS INTEGER summe = 0
  FOR i AS INTEGER = LBOUND(w) TO UBOUND(w)
    summe += w(i)
  NEXT
  RETURN summe / (UBOUND(w) - LBOUND(w) + 1)
END FUNCTION
```

Ausgabe

```
Der Mittelwert der festgelegten Werte ist 112.4
```

Es ist also nicht möglich, die Länge der Array-Parameter festzulegen – nur die Anzahl

der Dimensionen kann vorgegeben werden. Wenn es notwendig ist, dass das Array eine genau definierte untere bzw. obere Grenze besitzt, überprüfen Sie die Grenzen mit **`LBOUND()`** bzw. **`UBOUND()`** und brechen Sie das Unterprogramm ab, wenn die Grenzen nicht passen.

Während die Übergabe eines oder mehrerer Arrays kein Problem darstellt, ist die *Rückgabe* eines Arrays als Funktionswert nicht möglich. Beim Rückgabewert darf es sich aber selbstverständlich um ein UDT handeln (vgl. Kapitel 7), welches auch ein Array beinhalten kann. Eine weitere Lösung werden wir uns im nächsten Abschnitt ansehen.

12.5.2. `BYREF` und `BYVAL`

Über eine wichtige Frage haben wir uns bisher noch keine Gedanken gemacht: Was passiert eigentlich mit einem Parameter, wenn er innerhalb des Unterprogrammes verändert wird? Ändert sich dann auch sein Wert außerhalb der Funktion? Hierbei spielt es eine entscheidende Rolle, ob der Parameter nur als Wert übergeben wird oder als Referenz auf den Variablenwert. Im ersten Fall wird gewissermaßen eine Kopie der Variablen erstellt und diese übergeben. Was auch immer das Unterprogramm mit dieser Kopie anstellt, das Original bleibt davon unberührt. Wird aber die Variable selbst übergeben (oder genauer gesagt eine Referenz auf seine Speicherstelle), dann bleiben Änderungen innerhalb des Unterprogrammes auch nach dessen Beendigung erhalten. Es wurde ja direkt das Original verändert.

FreeBASIC unterstützt beide Konzepte. Parameter können mit den Schlüsselwörtern **`BYREF`** oder **`BYVAL`** versehen werden, um zu kennzeichnen, ob die Übergabe *by reference* (also als Original) oder *by value* (als Wert, also als Kopie) übergeben werden sollen.

Quelltext 12.14: BYREF und BYVAL

```
SUB doppel(BYREF a AS INTEGER, BYVAL b AS INTEGER, c AS INTEGER)
  ' Werte verdoppeln
  a *= 2 : b *= 2 : c *= 2
  ' Zwischenergebnis ausgeben
  PRINT "In der Prozedur:"
  PRINT "a = "; a,
  PRINT "b = "; b,
  PRINT "c = "; c
  PRINT
END SUB

DIM AS INTEGER a = 3, b = 5, c = 7
PRINT "Vor der Prozedur:"
PRINT "a = "; a,
PRINT "b = "; b,
PRINT "c = "; c
PRINT

doppel a, b, c
PRINT "Nach der Prozedur:"
PRINT "a = "; a,
PRINT "b = "; b,
PRINT "c = "; c
SLEEP
```

Ausgabe

```
Vor der Prozedur:
a =  3         b =  5         c =  7

In der Prozedur:
a =  6         b =  10        c =  14

Nach der Prozedur:
a =  6         b =  5         c =  7
```

Jeder Parameter wird innerhalb der Funktion verdoppelt. Wie Sie sehen, „vergisst“ das Programm die Änderung von `b`, wenn die Prozedur verlassen wird, während die Verdopplung von `a` beibehalten wird. Wird weder **`BYREF`** noch **`BYVAL`** angegeben, dann wird die Art der Parameterübergabe automatisch ermittelt:

- Zahlen-Datentypen (Ganzzahlen, Gleitkommazahlen) werden **`BYVAL`** übergeben. Das entspricht dem in der Regel gewünschten Verhalten, dass Unterprogramme eine eigene Speicherumgebung verwenden.

- Strings und UDTs werden **BYREF** übergeben. Das liegt daran, dass das Kopieren eines Strings bzw. eines UDTs deutlich mehr Arbeit erfordert als das Kopieren eines Zahlen-Datentyps.
- Arrays, egal welcher Art, werden *immer* **BYREF** übergeben. Für sie ist eine Übergabe *by value* nicht möglich; die Angabe von **BYVAL** (und auch von **BYREF**) führt zu einem Compiler-Fehler.

Unterschiede zu QuickBASIC:
Das Standardverhalten von FreeBASIC unterscheidet sich von QuickBASIC und auch von älteren FreeBASIC-Versionen (bis v0.16). Dort wurden alle nicht explizit ausgezeichneten Parameter **BYREF** übergeben. In den Dialektformen `-lang qb`, `-lang deprecated` und `-lang fblite` wird dieses ältere Verhalten gewählt.

Wenn Sie eine größere Kompatibilität zu anderen Dialektformen oder anderen Sprachen erreichen wollen, empfiehlt es sich, unabhängig vom Standardverhalten alle Parameter, bei denen eine bestimmte Übergabeart erforderlich ist, entsprechend mit **BYREF** oder **BYVAL** auszuzeichnen (außer Arrays, bei denen diese Angabe ja nicht erlaubt ist). Betroffen sind dabei alle Parameter, deren Wert innerhalb des Unterprogrammes verändert wird. Parameter, die im Unterprogramm nicht verändert werden, brauchen nicht gesondert ausgezeichnet zu werden.

Beachten Sie auch, dass die Angaben zu **BYREF** und **BYVAL** sowohl in der **DECLARE**-Zeile als auch in der Kopfzeile des Unterprogrammes erfolgen sollten.

12.5.3. Parameterübergabe `AS CONST`

Auch wenn ein Array nicht **BYVAL** übergeben werden kann, könnten Sie ein berechtigtes Interesse daran haben, eine Veränderung durch das Unterprogramm zu verhindern. Wenn Sie hinter einem Parameternamen, zwischen dem **AS** und der Angabe für den Datentyp, ein **CONST** einfügen, kann auf diesen Parameter innerhalb des Unterprogrammes nur lesend zugegriffen werden. Sollten Sie dennoch versuchen, den Wert des Parameters zu ändern, quittiert der Compiler dies durch eine Fehlermeldung.[18] **AS CONST** ist jedoch

[18] Sie könnten sich jetzt natürlich fragen, warum ein Unterbinden der Wertänderung überhaupt erforderlich ist, wenn Sie stattdessen auch einfach einen schreibenden Zugriff unterlassen können. Bedenken Sie aber, dass oft mehrere Personen an einem Projekt mitarbeiten und dass ein festgeschriebenes **CONST** eine Garantie dafür ist, dass tatsächlich keine Wertänderung stattfindet.

nicht auf Arrays beschränkt.

Quelltext 12.15: Parameterübergabe AS CONST

```
SUB prozedur(a AS INTEGER, b AS CONST INTEGER)
  ' Lesender Zugriff ist auf jeden Fall moeglich.
  PRINT a, b

  ' Auf a kann auch schreibend zugegriffen werden.
  a = 3
  ' Versucht man jedoch, b einen Wert zuzuweisen, erfolgt ein Fehler.
  ' b = 4
END SUB
```

In der abgedruckten Form stellt Quelltext 12.15 einen korrekten Quellcode dar. Wenn Sie jedoch das Kommentarzeichen in der vorletzten Zeile entfernen, können Sie den Codeschnipsel nicht mehr compilieren.

12.5.4. Aufrufkonvention für die Parameter

Damit die Parameter vom aufrufenden Befehl zum aufgerufenen Unterprogramm gelangen, werden sie zunächst der Reihe nach in den Stack gelegt (das ist ein Stapelspeicher, aus dem immer das zuletzt hineingelegte Element als erstes wieder herausgenommen wird – vergleichbar mit einem Stapel Teller, auf den man immer nur oben einen Teller hinzufügen oder herunternehmen kann) und innerhalb des Unterprogrammes wieder ausgelesen. Da sich FreeBASIC selbständig um die Verwaltung des Stacks kümmert, brauchen wir uns meistens mit den Details nicht auseinanderzusetzen. Gelegentlich werden sie jedoch wichtig.

Zunächst einmal ist wichtig, welcher Parameter zuerst in den Stack gelegt wird (und damit als letztes wieder ausgelesen werden kann). Dazu gibt es drei verschiedene Aufrufkonventionen:

- **STDCALL** legt die Parameter von rechts nach links in den Stack, d. h. der letzte Parameter liegt dann ganz unten im Stack. **STDCALL** ist die Standard-Aufrufkonvention der WinAPI und daher auch von FreeBASIC in Windows-Betriebssystemen.
- **CDECL** legt ebenfalls die Parameter von rechts nach links in den Stack. Allerdings muss die Prozedur in dieser Konvention den Stack nicht selbständig abbauen – das ist Aufgabe des aufrufenden Codes. **CDECL** ist die Standard-Aufrufkonvention unter Linux, BSD und DOS und wird daher auf diesen Plattformen auch von FreeBASIC verwendet.
- **PASCAL** legt die Paramter von links nach rechts in den Stack, d. h. der letzte Parameter liegt dann ganz oben im Stack. Auch hier muss die Prozedur den

Stack nicht selbständig abbauen. **PASCAL** ist die Standard-Aufrufkonvention unter QuickBASIC.

Wichtig werden die Unterschiede vor allem bei der Verwendung externer Bibliotheken, da hier natürlich sichergestellt werden muss, dass sich die Bibliothek und das eigene Programm über die Konvention einig sind. Es gibt FreeBASIC-intern aber ebenfalls einen Fall, in dem die Aufrufkonvention wichtig wird, nämlich bei der Verwendung variabler Parameterlisten, auf die wir gleich zu sprechen kommen.

Da nun zwischenzeitlich doch schon eine ganze Menge an Informationen zu den Unterprogrammen zusammengekommen ist (und das sind noch nicht alle), hier noch einmal zusammenfassend der Zwischenstand zum Aufbau der Syntax:

```
' Deklaration einer Prozedur bzw. einer Funktion
DECLARE SUB Name [Aufrufkonvention] [OVERLOAD] _
        ([{BYVAL|BYREF}] Parameter AS Typ [= Wert] [, ...])
DECLARE FUNCTION Name [Aufrufkonvention] [OVERLOAD] _
        ([{BYVAL|BYREF}] Parameter AS Typ [= Wert] [, ...]) AS Typ

' Definition einer Prozedur bzw. einer Funktion
SUB Name [Aufrufkonvention] ([{BYVAL|BYREF}] Parameter AS Typ [= Wert] [, ...])
  ' Prozedur-Rumpf
END SUB
FUNCTION Name [Aufrufkonvention] ([{BYVAL|BYREF}] Parameter AS Typ [= Wert] _
        [, ...]) AS Typ
  ' Funktions-Rumpf
END FUNCTION
```

- `Name` ist der Name des Unterprogramms, unter dem es angesprochen werden kann.
- `Aufrufkonvention` ist einer der Schlüsselwörter **STDCALL**, **CDECL** und **PASCAL**. Wird sie ausgelassen, verwendet FreeBASIC plattformabhängig **STDCALL** oder **CDECL**.
- **OVERLOAD** wird benötigt, wenn Sie mehrere Unterprogramme mit gleichem Namen definieren wollen. Die Signatur (bestehend aus dem Namen des Unterprogramms, der Anzahl und der Typen der Parameter sowie bei Funktionen der Typ des Rückgabewerts) muss jedoch eindeutig sein.
- **BYREF** bzw. **BYVAL** geben an, ob der folgende Parameter *by reference* oder *by value* übergeben wird. Ohne Angabe hängt die Art der Übergabe vom Datentyp ab.
- Es können beliebig viele Parameter übergeben werden (auch keiner). Die Wertzuweisung ist optional; sie bewirkt, dass der Parameter beim Aufrauf auch ausgelassen werden kann.

- Bei Funktionen muss noch der Datentyp des Rückgabewertes angegeben werden.

12.5.5. Variable Parameterlisten

Abfrage variabler Parameterlisten

Sie können, wenn Sie wollen, auch eine vollkommen freie Parameterliste verwenden, womit Sie beim Aufruf des Unterprogrammes weder in der Anzahl noch bei den Datentypen der Parameter eingeschränkt sind. Der Umgang mit einer variablen Parameterliste ist allerdings deutlich schwerer als mit den bisher behandelten „normalen“ Parameterlisten, und er ist definitiv nicht anfängerfreundlich.

Die Kopfzeile eines solchen Unterprogrammes sieht z. B. folgendermaßen aus:

```
SUB variabel CDECL (x AS INTEGER, y AS INTEGER, ...)
```

Achtung:
Beachten Sie die Aufrufkonvention **CDECL**: Sie ist für die Verwendung von variablen Parameterlisten dringend erforderlich, da bei **STDCALL** nicht sichergestellt werden kann, dass die Prozedur den erforderten Abbau des Stacks tatsächlich durchführt.

Die drei Punkte (*Ellipsis*) kennzeichnen den variablen Teil der Parameterliste. Das bedeutet, dass in diesem Beispiel die beiden Parameter x und y festgelegt sind und angegeben werden müssen. Danach kann eine beliebige Anzahl an Parametern folgen (es können auch null sein). Ein Nachteil ist, dass Sie innerhalb des Unterprogrammes lediglich die Lage der Parameter im Speicher ermitteln können, jedoch keine Möglichkeit haben, die Datentypen oder auch nur die Anzahl der Parameter festzustellen.

FreeBASIC stellt für variable Parameterlisten einen eigenen Datentyp **CVA_LIST** zur Verfügung.[19] Zuerst muss innerhalb der Prozedur diese **CVA_LIST** initialisiert werden. Dazu dient **CVA_START**, dem zwei Parameter übergeben werden: zum einen die neu zu initialisierende **CVA_LIST**, zum anderen der letzte Parameter der Parameterliste, der noch nicht zum variablen Anteil gehört.

Anschließend können mit **CVA_ARG** der Reihe nach die variablen Parameter abgerufen werden. **CVA_ARG** benötigt zwei Parameter, nämlich die Variable, in die der Parameterwert

[19] Genau genommen handelt es sich hier – abhängig von der Architektur, für die compiliert wird – um verschiedene Datentypen. Genauere Details sind hier jedoch nicht von Belang, denn es ist für uns nur entscheidend, wie der Datentyp angesprochen wird.

gelegt werden soll, und der Datentyp des Parameters (den kann das Programm ja, wie oben bereits erwähnt, nicht selbst aus der Parameterliste herauslesen).

Zuletzt muss die **CVA_LIST** mittels **CVA_END** wieder freigegeben werden. In Kurzform sehen diese drei Schritte folgendermaßen aus:

```
SUB variabel CDECL (x AS INTEGER, y AS INTEGER, ...)
  DIM AS CVA_LIST liste          ' zur Speicherung des variablen Listenteils
  DIM p AS LONG, q AS DOUBLE     ' zur Speicherung der variablen Parameter
  CVA_START(liste, y)            ' y ist der letzte feste Parameter der Liste
  p = CVA_ARG(liste, LONG)       ' ersten variablen Parameter holen
  q = CVA_ARG(liste, DOUBLE)     ' zweiten variablen Parameter holen
  ' ...
  CVA_END(liste)                 ' Parameterliste freigeben
END SUB
```

In diesem Codeschnipsel wird in der vierten Zeile die variable Liste initialisiert und später in der achten Zeile wieder freigegeben. Achten Sie darauf, dass jede Initialisierung mit **CVA_START** auch wieder mit **CVA_END** freigegeben werden muss! Bei der Initialisierung wurde im Codeschnipsel `y`, der zweite Parameter der Parameterliste, übergeben (also der letzte festgelegte Parameter vor dem variablen Anteil). Nun weist ein interner Zeiger auf den Speicherbereich hinter `y` und damit auf den ersten Parameter des variablen Anteils.

Unter der Annahme, dass es sich beim ersten variablen Parameter um ein **LONG** handelt, wird nun in der fünften Zeile mit **CVA_ARG** der erste variable Parameter ausgelesen. Dabei geschieht zweierlei: Zum einen wird der Speicherbereich, auf den der interne Zeiger verweist, als **LONG** interpretiert und entsprechend ausgelesen. Zum anderen rückt der Zeiger auf den nächsten Parameter weiter. Dazu muss er um die Länge eines **LONG** weitergesetzt werden. Der übergebene Datentyp ist also sowohl für die richtige Interpretation des Parameterinhalts als auch für das korrekte Weiterrücken des Zeigers nötig.

Wenn Sie die obige Funktion etwa über den Befehl `variabel(1, 2, 3, 4.5, 6)` aufrufen, wird im Parameter `p` der Wert `3` und im Parameter `q` der Wert `4.5` landen. Weitere Variablen würden in diesem Beispiel ignoriert, da sie in der Prozedur nicht abgefragt werden.

Beachten Sie noch zwei Dinge:

1. Damit variable Parameterlisten verwendet werden können, müssen die Parameter zwingend **CDECL** übergeben werden. Die Reihenfolge der Parameterübergabe ist hier entscheidend, da ja nur so korrekt von Parameter zu Parameter gesprungen werden kann.

2. Bei **CVA_START**, **CVA_ARG** und **CVA_END** handelt es sich weder um Funktionen noch Anweisungen, sondern um Makros. Was das genau bedeutet, werden wir später noch besprechen. Für den Augenblick ist nur wichtig, dass die um die

Parameter gesetzten Klammern unbedingt erforderlich sind – auch bei **CVA_START** und **CVA_END**.

Die entscheidende Frage ist nun, wie der Prozedur Kenntnis über Art und Anzahl der variablen Parameter Kenntnis erlangt. Da sowieso mindestens ein Paramter fest übergeben werden muss, ist es naheliegend, hier die Anzahl der nun folgenden Parameter zu übergeben. Sofern deren Datentypen festgelegt sind – im einfachsten Fall müssen alle denselben festgelegten Datentyp besitzen – reicht das bereits aus. Eine Mittelwertbestimmung mit variabler Parameterliste könnte dann folgendermaßen aussehen:

Quelltext 12.16: Mittelwertsbestimmung mit variabler Parameterliste

```
FUNCTION mittelwert CDECL (anz AS INTEGER, ...) AS DOUBLE
  DIM AS CVA_LIST liste
  DIM AS INTEGER summe
  CVA_START(liste, anz)

  FOR i AS INTEGER = 1 TO anz
    summe += CVA_ARG(liste, INTEGER)    ' aktuellen Wert holen und aufaddieren
  NEXT
  CVA_END(liste)
  RETURN summe / anz
END FUNCTION

PRINT mittelwert(3, 15, 17, 20)
SLEEP
```

Der erste Parameter `3` gehört nicht zur Mittelwertsbestimmung dazu, sondern gibt lediglich die Anzahl der zu mittelnden Werte an. `liste` ist die **CVA_LIST**, die durchlaufen werden soll. Die einzelnen aus der **CVA_LIST** gelesenen Werte werden selbst nicht benötigt, sonden können direkt aufsummiert werden. Wenn sichergestellt ist, dass es sich bei allen Werten um **INTEGER** handelt, liefert `CVA_ARG(liste, INTEGER)` den Wert dieses Parameters und setzt den Zeiger auf den nächsten Parameter. Wird ein anderer Datentyp verwendet, muss natürlich dieser statt **INTEGER** eingesetzt werden.

Beim Funktionsaufruf muss auf die korrekten Datentypen geachtet und ggf. auch die automatische Typbestimmung von FreeBASIC berücksichtigt werden. Wenn Sie statt `15` den Wert `15.0` übergeben, liegt er als **DOUBLE** im Speicher. Da der in der Funktion vorgefundene Speicherwert aber als **INTEGER** behandelt wird, das eine völlig andere Speicherverwaltung verwendet als Gleitkommazahlen, erhalten Sie falsche Daten.

Wenn Sie sinnvoll mit variablen Parameterlisten arbeiten wollen, bieten sich zwei Konzepte an:

- Der Datentyp aller Parameter in der variablen Liste muss gleich sein. Zudem übergeben Sie als ersten Parameter die Länge der variablen Liste. Diese Methode

ist relativ leicht umzusetzen, und Sie haben in Quelltext 12.16 bereits ein Beispiel dazu gesehen.

- Die Datentypen können sich unterscheiden. In diesem Fall greift man häufig auf einen Formatstring zurück, der als erster Parameter übergeben wird und der die Anzahl und die Datentypen der verwendeten Parameter enthält. Dazu sehen Sie ein Beispiel in Quelltext 12.17.

Quelltext 12.17: Variable Parameterliste mit Formatstring

```
SUB varlist CDECL (formatstring AS STRING, ...)
  DIM AS CVA_LIST liste
  CVA_START(liste, formatstring)

  FOR i AS INTEGER = 1 TO LEN(formatstring)
    SELECT CASE MID(formatstring, i, 1)
      CASE "l"
        PRINT "LONG:   " & CVA_ARG(liste, LONG)
      CASE "d"
        PRINT "DOUBLE: " & CVA_ARG(liste, DOUBLE)
      CASE "s"
        PRINT "STRING: " & *CVA_ARG(liste, ZSTRING PTR)
    END SELECT
  NEXT
  CVA_END(liste)
END SUB

DIM s AS STRING = "String2"

varlist "ldss", 1, 3.4, "String1", s
SLEEP
```

Ausgabe

```
INTEGER:  1
DOUBLE:   3.4
STRING:   String1
STRING:   String2
```

Mit dem Formatstring `"idss"` wird der Funktion mitgeteilt, dass der Reihe nach ein **`INTEGER`**, ein **`DOUBLE`** und zwei **`STRING`** übergeben werden. Das in der Funktion verwendete **`MID()`** dient zum Auslesen des Formatstrings – es gibt einen (in diesem Fall ein Zeichen langen) Teilstring zurück. **`MID()`** wird in Kapitel 15.2.3 ausführlich behandelt.

Beachten Sie:
Wie Sie in Quelltext 12.17 sehen, wird nicht der Datentyp **STRING**, sondern ein **ZSTRING PTR** eingesetzt. Das hat mit der internen Speicherverwaltung bei der Übergabe von Zeichenketten an Unterprogramme zu tun.

Kopie erstellen

Manchmal ist es nötig, eine Kopie der variablen Parameterliste anzulegen. Sie können dann innerhalb der Prozedur gleichzeitig mit zwei Versionen der Liste arbeiten – wichtiger ist aber die Möglichkeit, die Liste an ein anderes Unterprogramm weiterzugeben. Zum Erstellen der Kopie benötigen Sie das Makro **CVA_COPY**. Auch die Kopie muss am Ende mit **CVA_END** freigegeben werden.

Quelltext 12.18: Variable Parameterliste kopieren

```
DECLARE FUNCTION mittelwert CDECL (anz AS INTEGER, ...) AS SINGLE
DECLARE FUNCTION summe CDECL (anz AS INTEGER, varList as CVA_LIST) AS INTEGER

FUNCTION mittelwert CDECL (anz AS INTEGER, ...) AS SINGLE
  DIM AS CVA_LIST liste
  CVA_START(liste, anz)
  DIM AS INTEGER s = summe(anz, liste)    ' Auslagerung der Summenberechnung
  CVA_END(liste)                          ' CVA_LIST freigeben
  RETURN s / anz
END FUNCTION

FUNCTION summe CDECL (anz AS INTEGER, varList AS CVA_LIST) AS INTEGER
  DIM AS CVA_LIST liste
  DIM AS INTEGER rueckgabe = 0
  ' Da nicht sicher ist, dass die aufrufende Prozedur die uebergebene Liste nicht
  ' selbst verwendet (in diesem Fall kommen sich die verschiedenen Aufrufe von
  ' CVA_ARG gegenseitig in die Quere), wird lieber eine Kopie angelegt.
  CVA_COPY(liste, varList)
  ' Mit dieser Kopie wird nun weitergearbeitet:
  FOR i AS INTEGER = 1 TO anz
    rueckgabe += CVA_ARG(liste, INTEGER)
  NEXT
  ' CVA_LIST freigeben und Summenwert zurueckgeben:
  CVA_END(liste)
  RETURN rueckgabe
END FUNCTION

PRINT mittelwert(3, 15, 17, 20)
SLEEP
```

In diesem Beispiel verwendet die Funktion `mittelwert` die Liste nicht selbst aus,

daher hätte summe auch direkt mit der übergebenen Liste arbeiten können. Sicherer ist es aber, stattdessen eine Kopie zu erstellen, mit dieser zu arbeiten und sie anschließend wieder freizugeben. Natürlich hätte auch in mittelwert bereits eine Kopie erstellt werden können, die dann an summe übergeben (und am Ende ebenfalls wieder durch **CVA_END** freigegeben) wird – auch dann hätte mittelwert auf die (originale) Liste zugreifen können, ohne den Zugriff durch summe (auf die Kopie) zu beeinträchtigen.

Frühere Methoden

CVA_START und die anderen damit zusammenhängenden Makros sind in FreeBASIC recht neu. Viele im Umlauf befindliche Programme, die variable Parameterlisten nutzen, verwenden noch eine ältere Methode, die ich hier der Vollständigkeit halber kurz vorstellen möchte. Auch hier muss die Parameterliste aus mindestens einem festen Parameter bestehen. Der erste variable Parameter wird mit **VA_FIRST()** bestimmt, der Wert des aktuellen Parameters mit **VA_ARG()** ausgelesen. Im Unterschied zu **CVA_ARG** aktualisiert **VA_ARG()** den Zeiger nicht, sondern der Zeiger muss mit **VA_NEXT()** weitergerückt werden. Das Erstellen eines eigenen Listenobjekts ist für diese Funktionen nicht nötig.

Der große Nachteil von **VA_FIRST()**, **VA_ARG()** und **VA_NEXT()**: Die hierfür verwendete Methode greift auf spezielle Mechaniken des 32-Bit-Assemblers zurück; insbesondere bedeutet das, dass sie für die 64-Bit-Version von FreeBASIC nicht zur Verfügung steht. Die **CVA_LIST** arbeitet dagegen plattformübergreifend.

Quelltext 12.19: Mittelwertsbestimmung mit variabler Parameterliste (alte Methode)

```
' Dieser Quelltext funktioniert nicht unter 64 Bit!
FUNCTION mittelwert CDECL (anz AS INTEGER, ...) AS DOUBLE
  DIM AS ANY PTR param = VA_FIRST     ' Pointer auf den ersten Parameter
  DIM AS INTEGER summe

  FOR i AS INTEGER = 1 TO anz
    summe += VA_ARG (param, INTEGER)  ' aktuellen Wert holen und aufaddieren
    param =  VA_NEXT(param, INTEGER)  ' Zeiger auf den naechsten Parameter setzen
  NEXT
  RETURN summe / anz
END FUNCTION

PRINT mittelwert(3, 15, 17, 20)
SLEEP
```

12.6. Fragen zum Kapitel

1. Stellen Sie eine Liste von Argumenten zusammen, die für einen Einsatz von Unterprogrammen sprechen.

2. Was ist der Unterschied zwischen Prozeduren und Funktionen? Wann bietet sich welcher Typ an?

3. Welche Möglichkeiten gibt es, Variablen aus dem Hauptprogramm im Unterprogramm zu verwenden?

4. Welche Arten der Parameterübergabe gibt es und worin unterscheiden sie sich?

Teil III.

Datenverarbeitung

13. Datentypen umwandeln

Die meisten Befehle erfordern für ihre Parameter einen bestimmten Datentyp. Eine Multiplikation z. B. kann nur mit Zahlen durchgeführt werden, nicht aber mit Strings. Wenn Sie eine Zahl als String vorliegen haben und sie mit einer anderen Zahl multiplizieren wollen, müssen Sie sie vorher in einen geeigneten Datentyp umwandeln.

13.1. Allgemeine Umwandlung: `CAST()`

Der allgemeine Befehl zur Umwandlung von einem Datentyp in einen anderen heißt **`CAST()`**. Sie können damit einen Wert in jeden beliebigen Zahlendatentyp oder Pointertyp umwandeln.

```
neueVariable = CAST(neuerDatentyp, alteVariable)
```

Als ersten Parameter geben Sie den Datentyp an, in den Sie die (als zweiten Parameter angegebene) Variable umwandeln wollen. Für eine Umwandlung von einem String in eine Ganzzahl kann man dann schreiben:

Quelltext 13.1: Einsatz von CAST

```
DIM AS STRING zahlString = "246.8"
' Zahlenwert verdoppeln und ausgeben
PRINT 2*CAST(INTEGER, zahlString)
SLEEP
```

Ausgabe

```
492
```

Es wird Sie sicher nicht verwundern, dass bei einer Änderung von `CAST(INTEGER, zahlString)` zu `CAST(DOUBLE, zahlString)` der Nachkomma-Anteil erhalten bleibt. Erwähnenswert ist allerdings, dass bei einer Umwandlung in einen Ganzzahlentyp der Nachkomma-Anteil abgeschnitten, also immer in Richtung `0` gerundet wird.

CAST() löst eine Reihe an Befehlen ab, die ursprünglich von QuickBASIC her kommen und in früheren FreeBASIC-Versionen erweitert wurden, um auch vorzeichenlose Ganzzahlen und **(U)LONGINT** zu ermöglichen:

Befehl	Umwandlung in ...		Befehl	Umwandlung in ...
CBYTE()	**BYTE**		**CUBYTE()**	**UBYTE**
CSHORT()	**SHORT**		**CUSHORT()**	**USHORT**
CINT()	**INTEGER**		**CUINT()**	**UINTEGER**
CLNG()	**LONG**		**CULNG()**	**ULONG**
CLNGINT()	**LONGINT**		**CULNGINT()**	**ULONGINT**
CSIGN()	vorzeichenbehaftete Zahl		**CUNSG()**	vorzeichenlose Zahl
CSNG()	**SINGLE**		**CDBL()**	**DOUBLE**
CBOOL()	**BOOLEAN**		**CPTR()**	Pointer

Tabelle 13.1.: Befehle zur Typumwandlung

Allein schon an der puren Fülle der Befehle sehen Sie, warum man in den letzten Jahren von diesen Bezeichnungen abgekommen ist und alles kompakt in den Befehl **CAST()** gepackt wurde. Die alten Befehle werden aber weiter unterstützt; sie besitzen vor allem den Vorteil, deutlich kürzer zu sein. Aufgrund der Kürze des Befehls werden Sie im Buch auch gelegentlich auf **CINT()** stoßen; ansonsten empfehle ich eher die Verwendung der „Langform“. Zur Ergänzung: **CSIGN()** und **CUNSG()** wandeln die Zahl in die vorzeichenbehaftete bzw. vorzeichenlose Version des gleich großen Datentyps um, also z. B. ein **BYTE** in ein **UBYTE** bzw. umgekehrt.

Auch UDTs können mittels **CAST** in andere Datentypen umgewandelt werden, benötigen dazu aber zuerst die passende Definition einer **CAST**-Funktion. Dazu erfahren Sie in **??** mehr.

Und dann gibt es da noch ...

Ja, dann gibt es da noch **VAL()** und seine Familie. Ich erwähne die Befehle nur deshalb, weil es leicht sein kann, dass Sie in (vor allem älteren) Quelltexten darauf stoßen. **VAL()** wandelt einen String in ein **DOUBLE** um, macht also dasselbe wie **CDBL()**, außer dass es auf Strings beschränkt ist. Ähnlich ist es mit den anderen Befehlen:

- **VALINT()**: Umwandlung in ein **LONG** (!)
- **VALUINT()**: Umwandlung in ein **ULONG**
- **VALLNG()**: Umwandlung in ein **LONGINT** (!)
- **VALULNG()**: Umwandlung in ein **ULONGINT**

In allen Fällen werden die Nachkommastellen abgeschnitten. Die Funktionen arbeiten also genauso wie die entsprechenden **CAST**-Funktionen (genauer gesagt: die **CAST**-Funktionen rufen zur Umwandlung von Strings zuvor eine der Funktionen aus der **VALxxx**-Familie auf). Es spricht also nichts dagegen, gleich von vornherein **CAST()** zu verwenden.

13.2. Umwandlung in einen String

Für die Umwandlung einer Zahl in einen String ist **CAST()** nicht vorgesehen; stattdessen kommt hier **STR()** zum Einsatz. Liegt der Zahlenwert in einem **WSTRING** vor, verwenden Sie stattdessen die Variante **WSTR()**.

Quelltext 13.2: Zahl in einen String umwandeln

```
DIM AS INTEGER alter
INPUT "Gib dein Alter ein: ", alter
PRINT "In einem Jahr bist du " + STR(alter+1) + " Jahre alt."
SLEEP
```

Ausgabe

```
Gib dein Alter ein: 17
In einem Jahr bist du 18 Jahre alt.
```

Mit dem Operator **+** können Strings nicht mit Zahlen verkettet werden. Daher ist vorher eine explizite Umwandlung des Zahlenwertes in einen String erforderlich. Natürlich hätten Sie die Werte für die **PRINT**-Ausgabe auch einfach mit Strichpunkten aneinanderhängen können. Dazu müssen aber zwei Dinge bedacht werden: Zum einen ist eine Aneinanderreihung mit Strichpunkten nur bei **PRINT** zulässig, nicht aber bei einer Zuweisung in eine Variable oder bei Parametern eines Unterprogrammes oder anderer FreeBASIC-interner Befehle. Zum anderen formatiert **STR()** etwas anders. Es wird nämlich kein Platzhalter (in Form eines Leerzeichens) für das fehlende positive Vorzeichen gelassen; ein Umstand, der zum Tragen kommt, wenn sowohl positive als auch negative Werte auftauchen können.

```
DIM AS INTEGER alter
INPUT "Gib dein Alter ein: ", alter
PRINT "mit STR:         In einem Jahr bist du " + STR(alter+1) + " Jahre alt."
PRINT "mit Strichpunkt: In einem Jahr bist du ";  alter+1;       " Jahre alt."
SLEEP
```

Ausgabe

```
Gib dein Alter ein: 17
mit STR:         In einem Jahr bist du 18 Jahre alt.
mit Strichpunkt: In einem Jahr bist du  18 Jahre alt.
```

Im zweiten Fall wird der positive Wert `18` vorn mit einem Leerzeichen erweitert, wodurch in der Ausgabe ein überflüssiges Leerzeichen steht. Bei negativen Zahlen passt die Ausgabe allerdings:

Ausgabe

```
Gib dein Alter ein: -22
mit STR:         In einem Jahr bist du -21 Jahre alt.
mit Strichpunkt: In einem Jahr bist du -21 Jahre alt.
```

Das bedeutet: In der zweiten Ausgabe einfach ein Leerzeichen wegzulassen, um den Platzhalter bei positiven Zahlen zu kompensieren, ist nicht unbedingt eine passende Lösung.

13.3. Implizite Umwandlung

Während an manchen Stellen eine explizite Umwandlung in den richtigen Datentyp nötig ist, wandelt der Compiler die Daten oft auch automatisch in das passende Format um. Ein Beispiel: Der Operator **+** hat für Strings und Zahlen vollkommen verschiedene Bedeutung, und der Compiler kann bei einer Vermischung nicht entscheiden, ob der Programmierer nun eigentlich eine String-Addition oder eine Zahl-Addition durchführen wollte. Dagegen können eine Ganzzahl und eine Gleitkommazahl problemlos addiert werden. Zwar läuft auch hier die Addition unterschiedlich ab, was an den vollkommen verschiedenen Speicherformaten beider Zahlentypen liegt, jedoch kann der Compiler davon ausgehen, dass der Programmierer eine Gleitkommaberechnung wünscht. Die Ganzzahl wird daher kurzerhand in eine Gleitkommazahl umgewandelt und dann mit der anderen Zahl addiert.

Diese automatische Umwandlung findet bei Zahlen immer dann (und erst dann) statt, wenn sie benötigt wird. Ganzzahlen werden so lange als Ganzzahlen behandelt, bis ein Rechenschritt eine Gleitkommazahl erfordert. Umgekehrt kann aber auch eine Gleitkommazahl implizit in eine Ganzzahl umgewandelt werden, nämlich dann, wenn als Operanden Ganzzahlen gefordert sind. Das ist z. B. bei der Ganzzahldivision oder bei

MOD der Fall.

Ein weiterer Fall der impliziten Umwandlung wurde bereits in Kapitel 6.3.2 angesprochen: die Stringverkettung mit **&**. Da **&** nur für Zeichenketten definiert ist, weiß der Compiler, dass gegebenenfalls eine Konvertierung zu einer Zeichenkette durchgeführt werden muss.[20] Die explizite Umwandlung mittels **STR()** kann daher weggelassen werden.

13.4. ASCII-Code

13.4.1. Ursprung und Bedeutung des ASCII-Codes

Computer speichern alle Daten in Speicherzuständen, auch Texte, Bilder usw. Über das Binärsystem können wir diese Zustände mit Zahlenwerten identifizieren. Auch Texte werden gewissermaßen als Zahlenfolgen behandelt. Dazu wird jedem Schriftzeichen ein Zahlenwert zugeordnet. Eine der ältesten Zuordnungstabellen (im Bereich der Computertechnologie) ist der *American Standard Code for Information Interchange* ASCII, der schnell zu einem weltweiten Standard der Zeichencodierung wurde. Der ASCII-Code verwendet 7 Bit, kann also 128 Zeichen darstellen. Bei den ersten 32 Zeichen handelt es sich um Steuerbefehle, z. B. der Tabulatorvorschub oder das Zeilenende. Die weiteren beinhalten unter anderem die 26 Klein- und ebenso vielen Großbuchstaben des lateinischen Alphabets, die zehn Ziffern 0-9 sowie Leerzeichen und diverse Satzzeichen.

Selbstverständlich ist bei einem Zeichenvorrat von 128 Zeichen nicht genug Platz, um alle länderspezifischen Sonderzeichen wie etwa die deutschen Umlaute mit aufzunehmen. Sprachen, die nicht auf dem lateinischen Alphabet beruhen – als Beispiele seien nur einmal das griechische und das kyrillische Alphabet genannt – benötigen sogar eine beachtliche Menge an zusätzlichen Zeichen. Da sich allerdings schnell die Byte-Größe von 8 Bit durchsetzte und ein ASCII-Zeichen damit nur die Hälfte des Byte-Zahlenbereichs nutzte, lag es nahe, die restlichen 128 Werte mit den Zeichen zu füllen, die man gerade brauchte. Es entstanden eine Reihe an ANSI-Codepages[21] mit den national benötigten bzw. gewünschten Zusatzzeichen. Bei uns am bekanntesten sind die Codierungen ISO-8859-1 und ISO-8859-15, die unter anderem die deutschen Umlaute und eine Reihe an Vokalen mit Akzentzeichen enthält und den Zeichenvorrat eines großen Teils der westeuropäischen Sprachen abdeckt. ISO-8859-5 dagegen enthält sämtliche Schriftzeichen des kyrillischen Alphabets, ISO-8859-6 die arabischen Zeichen usw.

Konkret bedeutet das: Wenn Sie einen ANSI-codierten Text vorliegen haben, ohne die konkrete Codepage zu kennen, können Sie die Zeichen ab Nr. 128 nicht korrekt darstellen.

[20] **&** wandelt beide Operanden in einen String um, außer einer der beiden ist ein **WSTRING**; in diesem Fall wird auch der andere Operand in einen **WSTRING** umgewandelt.

[21] ANSI = American National Standards Institute

Hierin liegt auch der Grund, warum es bei der Verwendung von Umlauten in FreeBASIC zu Problemen kommen kann. Im Gegensatz zum ASCII-Code ist der Begriff ANSI-Code nicht genormt. Und es gibt noch weitere Probleme: In manchen Sprachen gibt es deutlich mehr Schriftzeichen, als in der ANSI-Codierung Platz hätten. Die chinesische Schrift etwa kennt mehrere Tausend Zeichen, selbst wenn die selten verwendeten Zeichen nicht mitgezählt werden. Für eine international geeignete Codierung ist daher ein deutlich größerer Zeichenvorrat nötig. Deshalb wurde *Unicode* entwickelt, welches heute weit über 100 Schriftsysteme und über 120 000 Zeichen unterstützt.

Wir werden im Folgenden mehrfach auf den Begriff ASCII-Code zu sprechen kommen. Es sei noch einmal darauf hingewiesen, dass diese Codierung lediglich die Zeichen mit den Nummern 0 bis 127 beinhaltet. Der Einfachheit halber werden wir aber auch die Zeichen 128 bis 255 als „erweiterten ASCII-Code“ bezeichnen, um eine ständige Differenzierung zwischen den ASCII-Zeichen und den weiteren ANSI-Zeichen zu vermeiden.

13.4.2. `ASC()` und `CHR()`

Den ASCII-Code eines Zeichens können Sie mit **`ASC()`** ermitteln. **`ASC()`** erhält als Parameter eine Zeichenkette, es wird davon jedoch nur das erste Zeichen ausgewertet.

Quelltext 13.3: Hallo Welt in ASCII-Werten (1)

```
PRINT "Die einzelnen ASCII-Codes des Strings 'Hallo Welt' lauten:"
DIM AS STRING halloWelt(...) = { "H", "a", "l", "l", "o", " ", "W", "e", "l", "t" }
FOR i AS INTEGER = 0 TO UBOUND(halloWelt)
  PRINT ASC(halloWelt(i)),
NEXT
SLEEP
```

Ausgabe

```
 72          97          108         108         111
 32          87          101         108         116
```

Um beispielsweise den ASCII-Code des dritten Zeichens eines Strings auszugeben, gab es früher nur die Möglichkeit, einen Teilstring ab dem dritten Zeichen zu erstellen und davon den ASCII-Wert zu bestimmen. Inzwischen geht das auch einfacher – **`ASC()`** erlaubt nämlich in FreeBASIC die Angabe eines zweiten Parameters, der die Position des gewünschten Zeichens bestimmt. Das dritte Zeichen des Strings "Hallo Welt" lässt sich also auch über `ASC("Hallo Welt",3)` ansprechen. Eine Alternative zu Quelltext 13.3 wäre damit:

Quelltext 13.4: Hallo Welt in ASCII-Werten (2)

```
PRINT "Die einzelnen ASCII-Codes des Strings 'Hallo Welt' lauten:"
DIM AS STRING halloWelt = "Hallo Welt"
FOR i AS INTEGER = 1 TO LEN(halloWelt)
  PRINT ASC(halloWelt, i),
NEXT
SLEEP
```

Hier greife ich schon die in Kapitel 15.2.1 vorgestellte Funktion **LEN()** zur Ermittlung der Länge eines Strings voraus.

Für die tatsächlichen ASCII-Werte – also alle Zeichen bis zum Wert 127 – ist die Zuordnung eindeutig, Sie werden also immer, wenn Sie `ASC("a")` aufrufen, das Ergebnis `97` erhalten. Anders sieht es mit den Zeichen der ANSI-Erweiterung aus. Wenn Sie z. B. `ASC("ä")` ermitteln wollen, hängt das Ergebnis von der in Ihrer IDE verwendeten Codepage ab, also davon, nach welcher Code-Tabelle die IDE das Zeichen `"ä"` speichert. Erhalten Sie die Zahl 228, dann verwendet die IDE vermutlich ISO-8859-1 oder ISO-8859-15.

Die Konsole verwendet möglicherweise eine vollkommen andere Codepage als Ihre IDE. Am einfachsten sehen Sie das, wenn Sie den Schritt rückwärts gehen. Mit **CHR()** wird eine ASCII-Nummer in das zugehörige Zeichen umgewandelt.

```
PRINT "Das ASCII-Zeichen 228 ist ein "; CHR(228)
SLEEP
```

Ausgabe

```
Das ASCII-Zeichen 228 ist ein õ.
```

Die Konsole unter Windows verwendet in der Regel Codepage 850; dort ist unter der Nummer `228` das `õ` gespeichert. Unter Linux hängt die Codepage von den System- und Konsoleneinstellungen ab. Viele Konsolen unter Linux arbeiten mit Unicode; in diesem Fall ist die Angabe `228` an dieser Stelle überhaupt nicht interpretierbar. Im Grafikfenster (darauf kommen wir in **??** zu sprechen) kommt dagegen Codepage 437 zu tragen, diesmal unabhängig vom Betriebssystem.

Die verfügbaren Zeichen finden Sie in Anhang C aufgelistet; beachten Sie jedoch, dass die Zeichen im Konsolen-Modus von der dort eingesetzten Codepage abhängen und daher abweichen können. Wenn Sie selbst testen wollen, wie Ihr Konsolenfenster die Zeichen von 128 bis 255 interpretiert, kann Quelltext 13.5 helfen. Sollten Sie nur „undefinierte Zeichen“ erhalten, wird Ihre Konsole vermutlich gar keinen ANSI-Satz verwenden, sondern mit Unicode arbeiten.

Quelltext 13.5: ANSI-Zeichen von 128 bis 255

```
FOR i AS INTEGER = 128 TO 252 STEP 4   ' immer vier Zeichen pro Zeile
  PRINT i; ") "; CHR(i); TAB(20); i+1; ") "; CHR(i+1);
  PRINT TAB(40); i+2; ") "; CHR(i+2); TAB(60); i+3; ") "; CHR(i+3)
NEXT
SLEEP
```

Wenn Sie mehrere Nummern gleichzeitig in ihren ASCII-Code umwandeln wollen, können Sie die Werte als Parameterliste von **CHR()** angeben.

```
PRINT CHR(72, 97, 108, 108, 111, 32, 87, 101, 108, 116)
SLEEP
```

Zur Nutzung von Unicode ist **CHR** nicht geeignet, aber es gibt dafür eine Ersatzfunktion namens **WCHR()**. Diese gibt zu einer Unicode-Nummer das passende Zeichen zurück.

```
PRINT WCHR(1055, 1088, 1080, 1074, 1077, 1090, 33)
```

Ausgabe

```
Привет!
```

Sie können diese Ausgabe natürlich nur dann korrekt sehen, wenn die verwendete Konsole Unicode unterstützt.

13.4.3. Binäre Kopie erstellen

ASC() und **CHR()** geben zwei verschiedene Interpretationen desselben Speicherzustandes wieder – einmal als **UBYTE** und einmal als **STRING** der Länge 1. Das lässt sich auch für größere Datentypen überlegen: Der Datenteil einer zwei Zeichen langen Zeichenkette belegt zwei Byte, genauso wie ein **SHORT**. Bei einer vier Zeichen langen Zeichenkette belegt der Datenteil denselben Speicherplatz wie ein **LONG**, usw. Da liegt es nahe, eine direkte Umwandlung von Zeichenketten in Zahlen zu definieren, die einer identischen Speicherbelegung entsprechen. Natürlich gibt es dazu auch eine Rückumwandlung. Sie können die Funktionen Tabelle 13.2 entnehmen.

Für **CVI()** kann auch explizit die Bit-Zahl angegeben werden, die verwendet werden soll. Erlaubt sind die Werte 16, 32 und 64, und die Angabe erfolgt in spitzen Klammern direkt hinter dem Befehl. Ohne eine solche Angabe wird die Größe eine **INTEGER** verwendet, also 32 Bit beim 32-Bit-Compiler und 64 Bit beim 64-Bit-Compiler.

Als Beispiel soll **CVSHORT()** dienen, da es mit den kleinsten Werten arbeitet und dadurch am leichtesten nachzurechnen ist.

Befehl	Stringgröße	Umwandlung in ...	Rückwandlung
CVSHORT()	2	**SHORT**	**MKSHORT()**
CVI()	4 bzw. 8	**INTEGER**	**MKI()**
CVL()	4	**LONG**	**MKL()**
CVLONGINT()	8	**LONGINT**	**MKLONGINT()**
CVS()	4	**SINGLE**	**MKS()**
CVD()	8	**DOUBLE**	**MKD()**

Tabelle 13.2.: Befehle für binäre Speicherkopien

```
PRINT CVSHORT("Hi")
SLEEP
```

"H" besitzt den ASCII-Code `72`, "i" den ASCII-Code `105`. Da das "i" an zweiter Stelle steht, hat seine Codenummer den 256-fachen Stellenwert. $72 + 105 \cdot 256$ ergibt den vom Programm ausgegebenen Wert `26952`. Dasselbe hätte auch **CVI<16>()** ergeben:[22]

```
PRINT CVI<16>("Hi")
SLEEP
```

Eine Rückumwandlung erfolgt über **MKSHORT()**:

```
PRINT MKSHORT(26952)     ' oder PRINT MKI<16>(26952)
SLEEP
```

Übrigens: Die hier genannte Befehl machen nichts anderes als das, was Sie mit einer **UNION** eines Strings und einer Zahl erreichen würden. Zur Erinnerung: **UNION** legt mehrere Datentypen in denselben Speicherbereich. Sie können daher ein und denselben Speicherwert als verschiedene Datentypen interpretieren.

[22] Auch CVI("Hi") ohne Bitangabe würde `26952` ergeben, da zwar je nach System vier bzw. acht Zeichen erwartet würden, die fehlenden Zeichen jedoch als ASCII-Wert `0` interpretiert würden. Bei längeren Strings würde das aber durchaus einen Unterschied machen.

Quelltext 13.6: Binäre Kopie mit MKLONGINT und mit UNION

```
UNION LongintString
  AS STRING*8 s
  AS LONGINT  i
END UNION

DIM AS LONGINT testZahl = 2410100309775106630
DIM AS LongintString testUnion
testUnion.i = testZahl
PRINT testUnion.s
PRINT MKLONGINT(testZahl)
PRINT MKI<64>(testZahl)
SLEEP
```

Ausgabe

```
FB4ever!
FB4ever!
FB4ever!
```

(Leider musste ich hier schon wieder einem späteren Kapitel vorgreifen. Was es mit **`STRING*8`** auf sich hat und warum Quelltext 13.6 mit einem normalen String nicht funktioniert, erfahren Sie in Kapitel 15.)

13.5. Fragen zum Kapitel

1. Nennen Sie zwei Fälle, in denen eine implizite Umwandlung des Datentyps stattfindet, sowie einen Fall, in dem eine explizite Umwandlung notwendig ist.

2. Was ist der Unterschied zwischen ASCII und ANSI, und warum ist die Unterscheidung wichtig?

3. Sondertasten in FreeBASIC: Fragen Sie (innerhalb einer Schleife) mit `INKEY` Tasten ab und lassen Sie sich dazu die ASCII-Codes ausgeben. Beachten Sie dabei, dass Sondertasten (z. B. Pfeiltasten) einen String der Länge `2` zurückliefern. Lassen Sie in diesem Fall *beide* ASCII-Werte ausgeben und erstellen Sie sich eine Liste der (für Sie) wichtigsten Sondertasten.

14. Verarbeitung von Zahlen

Die folgenden Kapitel werden ausführlicher auf die Möglichkeiten eingehen, vorhandene Daten zu verarbeiten. Für die Zahlendatentypen kommen dazu natürlich erst einmal eine Reihe eingebauter mathematischer Funktionen in Betracht, aber wir wollen auch auf den Umgang mit verschiedenen Zahlensystemen sowie auf das direkte Lesen oder Setzen einzelner Bit eingehen.

14.1. Mathematische Funktionen

14.1.1. Quadratwurzel, Absolutbetrag und Vorzeichen

Neben den Grundrechenarten `+`, `-`, `*`, `/`, `\` und `^` gibt es noch weitere Standardfunktionen der Mathematik. Eine davon ist die Quadratwurzel, auf englisch *square route*, woraus sich der Funktionsname **`SQR()`** ableitet. `SQR(n)` ist diejenige nichtnegative Zahl, die mit sich selbst multipliziert `n` ergibt. Die Quadratwurzel einer negativen Zahl ist nicht definiert, da kein Quadrat negativ sein kann.[23] Wenn Sie versuchen, die Quadratwurzel aus einer negativen Zahl zu ziehen, liefert FreeBASIC den Rückgabewert `nan` (*not a number*) oder auch `-nan`.

Achtung:
`nan` (*not a number*) und `inf` (*infinite*, unendlich) sind besondere Speicherzustände bei Gleitkommazahlen, die für Ganzzahlen nicht existieren. Wenn Sie das Ergebnis einer nicht berechenbaren Aufgabe wie `SQR(-1)` in einer Ganzzahl-Variablen zu speichern versuchen, werden Sie unsinnige Ergebnisse erhalten.

Der absolute Betrag einer Zahl `n` ist ihr Abstand vom Wert `0`, das bedeutet: Bei nichtnegativen Zahlen ist der Betrag der Zahl gleich der Zahl selbst. Bei negativen Zahlen fällt durch die Betragsbildung lediglich das negative Vorzeichen fort. Die Funktion wird in FreeBASIC mit **`ABS()`** aufgerufen.

[23] Einen Standard-Datentyp der komplexen Zahlen gibt es in FreeBASIC nicht.

Eng damit verbunden ist die Vorzeichenfunktion **SGN()**, vom lateinischen *signum* (*Zeichen*). Auch im Deutschen wird sie häufig als Signumfunktion bezeichnet. Sie gibt für positive Zahlen den Wert 1 zurück, für negative Zahlen den Wert `-1` und für `0` den Wert `0`.

Quelltext 14.1: Quadratwurzel; Betragsfunktion; Signumfunktion

```
PRINT " n         SQR(n)     ABS(n)     SGN(n)"
PRINT "=================================="

PRINT 9;    TAB(10); SQR(9);    TAB(20); ABS(9);    TAB(30); SGN(9)
PRINT 1.44; TAB(10); SQR(1.44); TAB(20); ABS(1.44); TAB(30); SGN(1.44)
PRINT -3;   TAB(10); SQR(-3);   TAB(20); ABS(-3);   TAB(30); SGN(-3)
PRINT 0;    TAB(10); SQR(0);    TAB(20); ABS(0);    TAB(30); SGN(0)
SLEEP
```

Ausgabe

```
 n          SQR(n)     ABS(n)     SGN(n)
==================================
 9          3          9           1
 1.44       1.2        1.44        1
-3         -nan        3          -1
 0          0          0           0
```

14.1.2. Winkelfunktionen

Auch die Winkelfunktionen **SIN()** (Sinus), **COS()** (Kosinus) und **TAN()** (Tangens) sind im Standard-Befehlssatz enthalten. Den mathematischen Hintergrund dieser Funktionen hier zu beleuchten, würde deutlich zu weit führen. Für diejenigen, welche mit Winkelfunktionen bisher noch nichts zu tun hatten, sei vereinfacht zusammengefasst: Diese Funktionen ermöglichen in einem rechtwinkligen Dreieck, einen direkten Zusammenhang zwischen den Winkelgrößen und den Seitenverhältnissen herzustellen. Etwas weiter gefasst eignen sie sich z. B. hervorragend für Berechnungen in einem kartesischen Koordinatensystem.

FreeBASIC rechnet ausschließlich in Bogenmaß, d. h. ein Vollwinkel beträgt 2π. Wenn Sie in Gradmaß rechnen wollen, müssen Sie die Werte vorher umrechnen.

Quelltext 14.2: Trigonometrische Berechnungen

```
CONST PI AS DOUBLE = 3.141592653589793
DIM AS DOUBLE deg, rad

INPUT "Bitte geben Sie einen Winkel in Grad (DEG) ein: ", deg
rad = deg * PI / 180   ' Grad in Bogenmass umrechnen
PRINT ""
PRINT "SIN(" & deg & " Grad) = " & SIN(rad)
PRINT "COS(" & deg & " Grad) = " & COS(rad)
PRINT "TAN(" & deg & " Grad) = " & TAN(rad)
SLEEP
```

Ausgabe

```
Bitte geben Sie einen Winkel in Grad (DEG) ein: 60

SIN(60 Grad) =  0.8660254037844386
COS(60 Grad) =  0.5000000000000001
TAN(60 Grad) =  1.732050807568877
```

Beachten Sie in diesem Beispiel, dass es beim Umgang mit Gleitkommazahlen unvermeidlich zu Rundungs-Ungenauigkeiten kommt.

Die Umkehrfunktionen, also der Arkussinus, Arkuskosinus und Arkustangens, werden über die Befehle **ASIN()**, **ACOS()** und **ATN()** aufgerufen. `ASIN(x)` gibt eine Zahl zurück, deren Sinus `x` ergibt. Da die Winkelfunktionen periodisch sind, ist die Umkehrung nicht eindeutig festgelegt. **ASIN()** und **ATN()** liefern den Wert, der im Bereich von $-\pi/2$ bis $\pi/2$ liegt, während **ACOS()** einen Wert im Bereich von 0 bis π zurückgibt.

Für **ATN()** gibt es noch den Alternativbefehl **ATAN2()**, dem zwei Parameter übergeben werden und der den Arkustangens des Quotienten beider Parameter zurückgibt. Ein entscheidender Unterschied zu **ATN()** ist, dass die Vorzeichen beider Parameter bei der Berechnung berücksichtigt werden: `ATAN2(2, 3)` ist nicht dasselbe wie `ATAN2(-2, -3)`.

Quelltext 14.3: Arkusfunktionen

```
CONST PI AS DOUBLE = 3.141592653589793

PRINT "Einige Arkus-Berechnungen:"
PRINT "ASIN(0.5) = " & ASIN(.5) & " (" & ASIN(.5)/PI*180 & " Grad)"
PRINT "ACOS(0.5) = " & ACOS(.5) & " (" & ACOS(.5)/PI*180 & " Grad)"
PRINT

PRINT "ATAN2( 7,  24) = " & ATAN2(7, 24)
' ergibt das gleiche wie
PRINT "ATN ( 7 /  24) = " & ATN(7 / 24)
' aber etwas anderes als
PRINT "ATAN2(-7, -24) = " & ATAN2(-7, -24)   ' = ATAN2(7, 24) - PI
SLEEP
```

Ausgabe

```
Einige Arkus-Berechnungen:
ASIN(0.5) = 0.5235987755982989 (30 Grad)
ACOS(0.5) = 1.047197551196598 (60.00000000000001 Grad)

ATAN2( 7,  24) = 0.2837941092083279
ATN ( 7 /  24) = 0.2837941092083279
ATAN2(-7, -24) = -2.857798544381466
```

`ATAN(-7, -24)` ist der um π „verschobene“ Wert von `ATAN(7, 24)` (also entsprechend einer Verschiebung um 180°). Der Tangens beider Werte ist gleich.

14.1.3. Exponentialfunktion und Logarithmus

`EXP()` ruft die Exponentialfunktion zur Basis e auf (die Eulersche Zahl $e \approx 2.718$). Die Umkehrfunktion dazu ist der natürliche Logarithmus, der in FreeBASIC über den Befehl **`LOG()`** angesprochen wird. Während mittels `a^x` die Potenz zu jeder beliebigen Basis direkt berechnet werden kann, gibt es keine vorgefertigte Funktion, die unmittelbar den Logarithmus zur Basis a zurück gibt. Hier bieten jedoch die Logarithmusgesetze eine schnelle und unkomplizierte Lösung.

Quelltext 14.4: Exponentialberechnungen

```
PRINT "Eulersche Zahl e ="; EXP(1)
PRINT "e^2     ="; EXP(2)
PRINT "LN(e^2) ="; LOG(EXP(2))
PRINT
PRINT "5 ^ 3   ="; 5^3
PRINT "Logarithmus von 125 zur Basis 5:"; LOG(125)/LOG(5)
SLEEP
```

Ausgabe

```
Eulersche Zahl e = 2.718281828459045
e^2     = 7.38905609893065
LN(e^2) = 2

5 ^ 3   = 125
Logarithmus von 125 zur Basis 5: 3
```

14.1.4. Rundung

Zum Runden eines Zahlenwertes gibt es verschiedene Konzepte, je nachdem, was Sie erreichen wollen. Der einfachste Fall ist das Abrunden auf die nächstkleinere Ganzzahl (sofern es sich bei der Eingabe nicht schon um eine Ganzzahl handelt). Hierzu wird **`INT()`** verwendet. Etwas Ähnliches, jedoch nicht völlig Gleiches, macht **`FIX()`**: hier wird der Nachkomma-Anteil abgeschnitten. Der Unterschied beider Befehle liegt in der Behandlung negativer Zahlen. Während **`INT()`** immer abrundet, rundet **`FIX()`** immer „in Richtung `0`“; negative Zahlen werden also aufgerundet.

Wenn Sie, wie man das üblicherweise gewohnt ist, bis `.5` ab- und ab `.5` aufrunden wollen, können Sie es mit **`CINT()`** versuchen. Sie erinnern sich: `CINT(x)` ist eine Kurzform von `CAST(INTEGER, x)`, also eine Umwandlung zwischen zwei Datentypen. Wird eine Gleitkommazahl in eine Ganzzahl umgewandelt, muss sie natürlich gerundet werden.

Aber: Wie viele andere Programmiersprachen auch, rundet FreeBASIC nicht kaufmännisch, sondern *mathematisch*! Das bedeutet, dass bei `.5` zur nächsten *geraden* Zahl gerundet wird – also z. B. bei `1.5` aufwärts auf `2`, bei `4.5` aber abwärts auf `4`. Hintergrund ist die bessere Verwertbarkeit gerundeter Werte für statistische Aufgaben, aber für ein Buchhaltungsprogramm wirft dieses Vorgehen Probleme auf.[24]

[24] Es sei aber daran erinnert, dass ein Buchhaltungsprogramm, das sich auf Gleitkommazahlen verlässt, noch mit ganz anderen Problemen zu kämpfen haben wird!

Glücklicherweise gibt es mit ein wenig Einfallsreichtum auch dazu eine Lösung. Wenn Sie zu einer Zahl `x` die „Hälfte des Vorzeichens“ addieren und anschließend mit **`FIX()`** die Nachkommastellen abschneiden, erhalten Sie eine kaufmännische Rundung. Hatte ein positives `x` einen Nachkomma-Anteil, der kleiner war als `.5`, dann erreichen Sie durch die Addition von `0.5` noch nicht die nächste ganze Zahl; `x` wird also abgerundet. Ab einem Nachkommaanteil von (einschließlich) `.5` landen Sie durch die Addition im Bereich der nächsten Ganzzahl. Überlegen Sie, warum das beschriebene Verfahren auch für negative Zahlen funktioniert.

Quelltext 14.5 gibt einen Überblick über die vier vorgestellten Möglichkeiten:

Quelltext 14.5: Rundungsverfahren

```
PRINT " x        INT(x)    FIX(x)    CINT(x)   FIX(x+SGN(x)/2)"
PRINT "=================================================="
DIM AS DOUBLE wert(...) = { 1.2, 1.8, -1.8, 2.5, 3.5, -3.5 }

FOR i AS INTEGER = 0 TO UBOUND(wert)
  PRINT wert(i); TAB(12); INT(wert(i)); TAB(22); FIX(wert(i));
  PRINT TAB(32); CINT(wert(i)); TAB(42); FIX(wert(i)+SGN(wert(i))/2)
NEXT
SLEEP
```

Ausgabe

```
 x         INT(x)    FIX(x)    CINT(x)   FIX(x+SGN(x)/2)
==================================================
 1.2         1         1         1         1
 1.8         1         1         2         2
-1.8        -2        -1        -2        -2
 2.5         2         2         2         3
 3.5         3         3         4         4
-3.5        -4        -3        -4        -4
```

Beachten Sie den Unterschied zwischen `INT(-1.8)` und `FIX(-1.8)` sowie die Besonderheit bei `CINT(2.5)`. Selbstverständlich können Sie das Array in Zeile 3 um eigene Werte ergänzen, um die Rundungsverfahren weiter zu erforschen.

Wenn Sie umgekehrt nur den Nachkommateil einer Zahl erhalten wollen, können Sie **`FRAC()`** verwenden. Das Vorzeichen der Zahl bleibt dabei erhalten, d. h. `FRAC(3.14)` gibt den Wert `0.14` zurück und `FRAC(-3.14)` den Wert `-0.14`.

14.1.5. Modulo-Berechnung

Während die Integerdivision \ den ganzzahligen Anteil einer Division zweier Integer zurückgibt, bestimmt die Modulo-Berechnung **MOD** den Rest, der bei dieser Division bleibt. `7 MOD 3` z. B. ist der Rest der Division `7/3`, also `1`. Ist die erste Zahl negativ, dann bleibt das Vorzeichen erhalten, d. h. `-7 MOD 3` ist `-1`. Das Vorzeichen der zweiten Zahl spielt dagegen keine Rolle.

Wenn Sie die Modulo-Berechnung bei Gleitkommazahlen anwenden, werden die Zahlen, genauso wie bei der Integerdivision, zuerst mit **CINT()** gerundet.

MOD kann bei manchen Aufgaben sehr praktisch sein. Es bietet etwa eine sehr bequeme Möglichkeit, zu überprüfen, ob eine Zahl gerade oder ungerade ist. Schließlich bleibt bei ungeraden Zahlen bei einer Teilung durch `2` der Rest `1` – bei geraden Zahlen nicht. Außerdem können Sie es verwenden, wenn Sie sich in einem zyklischen Datenfeld bewegen wollen, z. B. wenn Sie ein mit Daten gefülltes Array haben, bei dem Sie beim Überschreiten einer Array-Grenze wieder auf der anderen Seite fortfahren wollen. Auch die Minutenangabe einer Zeitberechnung fällt in diesen Bereich: nach der 59. Minute wird wieder bei 0 begonnen.

Quelltext 14.6: Rechungen mit Modulo

```
DIM AS STRING monat(...) = { "Januar", "Februar", "Maerz", "April", "Mai", "Juni", _
                  "Juli", "August", "September", "Oktober", "November", "Dezember" }
DIM AS INTEGER zahl
INPUT "Geben Sie eine Zahl ein: ", zahl

IF zahl MOD 2 THEN
  PRINT "Die Zahl ist ungerade!"
ELSE
  PRINT "Die Zahl ist gerade!"
END IF

PRINT "Der 6. Monat im Jahr ist der "; monat(5)
PRINT zahl; " Monate spaeter ist "; monat((5+zahl) MOD 12)
SLEEP
```

Ausgabe

```
Geben Sie eine Zahl ein: 17
Die Zahl ist ungerade!
Der 6. Monat im Jahr ist der Juni
 17 Monate spaeter ist November
```

Zu Zeile 6: `zahl MOD 2` wird als Wahrheitswert interpretiert und ist daher genau dann erfüllt, wenn bei der Division ein Rest bleibt. Sie hätten auch direkt das Ergebnis

der Modulo-Berechnung überprüfen können:

```
IF (zahl MOD 2) = 0 THEN PRINT "Die Zahl ist gerade."
' oder
IF (zahl MOD 2) = 1 THEN PRINT "Die Zahl ist ungerade."
' Da Zeile 3 nicht mit negativen Zahlen funktioniert, waere besser:
' IF (zahl MOD 2) <> 0 THEN PRINT "Die Zahl ist ungerade."
```

Beachten Sie beim Monats-Array, dass die Zählung, bei `0` beginnt. `monat(0)` ist Januar und `monat(11)` ist Dezember. Das ist vielleicht eine ungewöhnliche Bezeichnung, aber für die Modulo-Berechnung äußerst praktisch. Bei der Berechung `MOD 12` erhalten Sie ja einen Wert von `0` bis `11`.

Leider funktioniert zwar das Addieren, beim Subtrahieren stoßen wir aber auf das Problem mit dem möglicherweise negativen Ergebnis.

Quelltext 14.7: Subtraktion mit Modulo

```
DIM AS STRING monat(11) = { "Januar", "Februar", "Maerz", "April", "Mai", "Juni", _
                  "Juli", "August", "September", "Oktober", "November", "Dezember" }
DIM AS INTEGER aktuell = 5

' funktioniert problemlos:
PRINT "8 Monate nach "; monat(aktuell) " kommt "; monat((aktuell+8) MOD 12)
' funktioniert nicht (Zugriff ausserhalb des Speicherbereichs!):
' PRINT "8 Monate vor "; monat(aktuell) " kommt "; monat((aktuell-8) MOD 12)
' stattdessen funktioniert:
PRINT "8 Monate vor "; monat(aktuell) " kommt "; monat((aktuell+4) MOD 12)
```

Acht Monate zurück ist dasselbe wie vier Monate nach vorn, zumindest was den Monatsnamen angeht. Etwas aufwändiger wird es, wenn Sie, wie in Quelltext 14.6 bei der Addition, den zu subtrahierenden Wert nicht kennen. In diesem Fall können Sie das Ergebnis der Modulo-Berechnung zunächst zwischenspeichern und bei einem negativen Ergebnis den zugehörigen positiven Wert berechnen. Wenn sichergestellt ist, dass Sie weniger als 12 subtrahieren (bzw. weniger als den Wert rechts von **MOD**), können Sie die Sache vereinfachen, indem Sie links noch einmal 12 addieren.

```
DIM AS INTEGER zwischenwert = alterWert - abzug
DO UNTIL zwischenwert >= 0                ' Solange der Zwischenwert zu klein ist,
  zwischenwert += 12                      '     addiere 12 zum Zwischenwert.
LOOP
PRINT monat(zwischenwert MOD 12)              ' klappt fuer jeden 'abzug'
PRINT monat((alterWert - abzug + 12) MOD 12)  ' klappt nur, wenn 'abzug' <= 12
```

Da sich die Zahlenwerte der Berechnung `MOD 12` immer nach zwölf Zahlen wiederholt, können Sie links beliebig oft 12 addieren, ohne dass sich das Ergebnis ändert.

14.1.6. Zufallszahlen

Auch die Computer-Zufallszahlen gehören in den Bereich der mathematischen Funktionen. FreeBASIC kann nämlich (wie andere Programmiersprachen auch) keine echten Zufallszahlen erzeugen, sondern verwendet einen sogenannten Pseudozufallszahlengenerator, der anhand bestimmter Voraussetzungen die nächste „Zufallszahl“ berechnet. Der dabei verwendete Mersenne-Twister-Algorithmus wurde so geschickt ausgetüftelt, dass die erzeugten Zahlen für den Anwender wie Zufallszahlen erscheinen. Unter anderem heißt das, dass die kommende Zahlenfolge für den Benutzer unvorhersehbar ist.

Diese Unvorhersehbarkeit hat natürlich ihre Grenzen. Immerhin handelt es sich immer noch um einen (zudem bekannten) Algorithmus, und was der Algorithmus berechnet, muss sich auch auf anderem Weg berechnen lassen. Wenn ausreichend viele Zahlen der Zufallsfolge vorliegen, lässt sich theoretisch tatsächlich die weitere Folge berechnen. Für sicherheitsrelevante Anwendungen ist der Algorithmus daher nur bedingt geeignet. Für andere Programme, etwa für zufallsbedingte Spiele, ist die Pseudozufälligkeit aber mehr als ausreichend.

Auch wenn Sie die Funktionsweise des Mersenne-Twister gar nicht so genau kennen müssen, ist ein klein wenig Hintergrundwissen dennoch hilfreich. Der Algorithmus benötigt eine Gruppe von Startwerten, aus denen er dann die weiteren Zufallszahlen berechnet. Das bedeutet: Bevor man den Zufallsgenerator sinnvoll nutzen kann, muss er zuerst initialisiert werden. Dazu dient der Befehl **RANDOMIZE**.

```
RANDOMIZE [initialwert] [, algorithmus]
```

Anhand von `initialwert` (im Englischen *seed* genannt) werden die Startwerte für den Algorithmus bestimmt. Daher werden Sie immer, wenn Sie denselben Startwert verwenden, auch dieselbe Zufallsfolge erhalten. Das ist natürlich in den meisten Fällen nicht gewünscht. Wenn Sie jedoch `startwert` einfach weglassen, wählt FreeBASIC selbständig einen Initialwert, der auf der systeminternen Zeitmessung **TIMER()** basiert (siehe Kapitel 18.1.2). Möglicherweise finden Sie auch Quelltexte, in denen **TIMER()** als Startwert angegeben wird. Das war früher ein beliebtes Mittel, um einen „zufälligen“ Initialwert zu erhalten. Bei FreeBASIC ist es dagegen empfehlenswert, `initialwert` wegzulassen, um die bestmögliche Initialisierung zu erhalten.

`algorithmus` legt den Algorithmus für die Zufallszahl-Berechnung fest. Wie bereits gesagt, wird normalerweise der Mersenne-Twister verwendet. Sie können aber auch z. B. gezielt einen speichereffizienteren Algorithmus wählen (auf Kosten der hohen Zufälligkeit) oder den Algorithmus von QuickBASIC, der zwar weitaus schwächer ist, aber für die Kompatibilität eines älteren Programmes interessant sein könnte. Wenn Sie einfach nur einen guten, soliden Zufallsgenerator verwenden wollen, lassen Sie diesen Parameter weg.

Hinweis:
RANDOMIZE wird in der Regel nur ein einziges Mal aufgerufen, nämlich bevor Sie die erste Zufallszahl abfragen. Es ist keinesfalls nötig und auch nicht zu empfehlen, **RANDOMIZE** vor jeder weiteren Zufallszahl erneut aufzurufen!

Um anschließend auf die Zufallszahlen zuzugreifen, benötigen Sie **RND()**. Sie erhalten eine **DOUBLE**-Zahl von `0` bis `1` (genauer: von einschließlich `0` bis ausschließlich `1`). Bei jedem Aufruf von **RND()** erhalten Sie eine neue Zahl vom Generator.

Quelltext 14.8: Zufallszahlen ausgeben

```
RANDOMIZE
FOR i AS INTEGER = 1 TO 10  ' zehn Zufallszahlen ausgeben
  PRINT RND
NEXT
SLEEP
```

Wenn Sie das Programm mehrmals starten, werden Sie sehen, dass Sie jedesmal eine neue Folge an Zahlen erhalten. Wenn Sie allerdings die erste Zeile weglassen oder einen festen Initialwert angeben (z. B. `RANDOMIZE 18`), bekommen Sie bei jedem Start dieselbe Folge.

Hinweis:
Die Angabe eines Initialwertes wird notwendig, wenn Sie zufällige Werte generieren wollen, die reproduzierbar sein sollen – etwa wenn das Programm zufallsgesteuert ein Spielfeld erstellen soll, das auf einem anderen Rechner durch Übermittlung des Initialwertes ganz genauso nachgebaut werden soll.

Da man in den seltensten Fällen Zufallswerte im Bereich von `0` bis `1` benötigt, werden Sie die mit **RND()** ermittelte Zahl in der Regel noch so modifizieren, dass eine Zufallszahl in einem von Ihnen gewünschten Bereich herauskommt. Wir wollen das an der Simulation eines Würfelwurfes demonstrieren. Der gewünschte Zahlenbereich geht von `1` bis `6`, außerdem werden nur Ganzzahlen benötigt. Wenn **RND()** mit `6` multipliziert wird und man anschließend die Nachkommastellen abschneidet, erhält man Ganzzahlen von `0` bis `5`. Nun nur noch `1` addiert, und schon ist der Würfelwurf-Simulator fertig.

Quelltext 14.9: Würfelwurf simulieren

```
RANDOMIZE
PRINT "Ich werde zehnmal fuer Sie wuerfeln. Die Ergebnisse lauten:"
FOR i AS INTEGER = 1 TO 10  ' zehn Zufallszahlen ausgeben
  PRINT INT(RND*6) + 1
NEXT
SLEEP
```

Entscheidend bei der Simulation eines Würfelwurfes ist, dass jede Zahl mit derselben Wahrscheinlichkeit errechnet wird. Versuchen Sie nachzuvollziehen, warum das bei der hier gewählten Berechnung der Fall ist.

Der Vollständigkeit halber sei noch erwähnt, dass Sie **`RND()`** einen Parameter mitgeben können. Ist der Parameter `0`, gibt der Befehl ein weiteres Mal die letzte ausgegebene Zufallszahl zurück. Bei allen anderen Parameterwerten erhalten Sie die nächste „neue“ Zufallszahl.

Unterschiede zu QuickBASIC:
In QuickBASIC hat der optionale Parameter eine andere Bedeutung. Die Dialektform `-lang qb` arbeitet hier genauso wie QuickBASIC.

14.2. Zahlensysteme

Neben dem uns geläufigen Dezimalsystem werden im Computerbereich noch drei weitere Zahlensysteme verwendet: das Binär-, das Oktal- und das Hexadezimalsystem. Allen gemeinsam ist, dass es sich um Stellenwertsysteme handelt, d. h. die Position einer Ziffer innerhalb der Zahl gibt ihre Wertigkeit an. Während im Dezimalsystem die zweite Ziffer von rechts die Wertigkeit `10` besitzt, die dritte Ziffer von rechts die Wertigkeit $10 \cdot 10 = 100$ usw., ist die Basis des Binärsystem die `2`, die Basis des Oktalsystems ist `8` und die Basis des Hexadezimalsystems `16`. Die Oktalzahl 123_8 hätte also den dezimalen Wert $1 \cdot 64 + 2 \cdot 8 + 3 = 83$ (die tiefgestellte `8` in 123_8 zeigt an, dass es sich um eine Oktalzahl handelt). Grundsätzlich wären auch Zahlensysteme zu ganz anderen Basen möglich.[25]

Für das Binärsystem stehen nur die zwei Ziffern `0` und `1` zur Verfügung, für das Oktalsystem die acht Ziffern von `0` bis `7`. Für das Hexadezimalsystem werden 16 Ziffern

[25] Weitere im Alltag auftretende Zahlensysteme sind das Sechziger-System der Zeitmessung (1 Stunde = 60 Minuten = 3600 Sekunden) und das noch gelegentlich verwendete Zwölfer-System (1 Gros = 12 Dutzend = 144).

benötigt. Da wir im Dezimalsystem nur zehn Ziffern zur Verfügung haben, behilft man sich für die fehlenden sechs Ziffern mit den Buchstaben `A` bis `F`. D_{16} ist damit der Dezimalwert `13` und $1A_{16}$ der Dezimalwert `26` $(1 \cdot 16 + 10)$.

14.2.1. Darstellung in Nicht-Dezimalsystemen

Im Quelltext auftretende Zahlen interpretiert FreeBASIC zunächst als Dezimalzahlen. So, wie oben die tiefgestellten Werte verwendet wurden, um das verwendete Zahlensystem zu kennzeichnen (wie D_{16} für das Hexadezimalsystem), muss es auch im Programm explizit kenntlich gemacht werden, wenn Sie nicht das Dezimalsystem meinen. FreeBASIC verwendet dazu die et-Ligatur **`&`**, die, zusammen mit einem Kennbuchstaben für das gewünschte System, vor die Zahl geschrieben wird:

- **`&b`** für Binärzahlen (z. B. `&b1101=13`)
- **`&o`** für Oktalzahlen (z. B. `&o37=31`)
- **`&h`** für Hexadezimalzahlen (z. B. `&h1F=31`)

Sowohl für den Kennbuchstaben als auch die Ziffern `A-F` im Hexadezimalsystem ist Groß- und Kleinschreibung nicht von Bedeutung.

Mit dieser neuen Schreibweise können Zahlen anderer Systeme genauso im Programm verwendet werden, wie Sie es von Dezimalzahlen gewohnt sind. Nur Gleitkommazahlen anderer Systeme sind in FreeBASIC nicht möglich.

```
PRINT (420 + &hfe) / &b10
' identisch mit PRINT (420 + 254) / 2
SLEEP
```

14.2.2. `BIN()`, `OCT()` und `HEX()`

FreeBASIC selbst zeigt Zahlen ausschließlich im Dezimalsystem an. Wenn Sie eine Anzeige in einem anderen Zahlensystem wünschen, kann diese nur über einen String erfolgen. Die Umwandlung läuft über die Funktionen **`BIN()`** (Binärsystem), **`OCT()`** (Oktalsystem) und **`HEX()`** (Hexadezimalsystem). Der Rückgabewert ist ein String, der die Ziffernfolge im jeweiligen System enthält. **`BIN()`** wurde ja schon in Kapitel 10.2 eingesetzt.

Wenn Sie übrigens den Zahlenwert als **`WSTRING`** haben wollen, verwenden Sie stattdessen die Funktionen **`WBIN()`**, **`WOCT()`** und **`WHEX()`**. Wir werden uns allerdings erst einmal weiter mit normalen Strings beschäftigen.

Quelltext 14.10: Vom Dezimalsystem in ein anderes System umrechnen

```
DIM AS INTEGER zahl
INPUT "Geben Sie eine Zahl ein: ", zahl
PRINT "in Binaerschreibweise: "; BIN(zahl)
PRINT "im Oktalsystem         "; OCT(zahl)
PRINT "im Hexadezimalsystem:  "; HEX(zahl)
SLEEP
```

Ausgabe

```
Geben Sie eine Zahl ein: 42
in Binaerschreibweise: 101010
im Oktalsystem         52
im Hexadezimalsystem:  2A
```

Wenn Sie nachrechnen wollen: Für das Binärsystem ergibt sich 32+8+2 = 42, im Oktalsystem $5 \cdot 8 + 2 = 42$ und im Hexadezimalsystem $2 \cdot 16 + 10 = 42$. Interessant ist sicher auch der Umgang mit negativen Zahlen:

Ausgabe

```
Geben Sie eine Zahl ein: -42
in Binaerschreibweise: 11111111111111111111111111010110
im Oktalsystem         37777777726
im Hexadezimalsystem:  FFFFFFD6
```

Beim Wert `-1` sind alle Bits gesetzt, das entspricht (in einem 32-Bit-System) der maximalen Oktalzahl `37777777777` bzw. der maximalen Hexadezimalzahl `FFFFFFFF`. Für jeden Schritt weiter ins Negative wird dieser Maximalwert um 1 reduziert.

BIN(), **OCT()** und **HEX()** erlauben einen zusätzlichen zweiten Parameter, der die Länge des zurückgegebenen Strings angibt. Das kann genutzt werden, um ihn vorn bei Bedarf mit Nullen aufzufüllen und dadurch eine einheitliche Länge zu erhalten, aber auch, um führende Ziffern abzuschneiden, die für Sie nicht interessant sind.

```
PRINT "Binaerer Vergleich von 10 und -10"
DIM AS UBYTE x = 10
PRINT BIN( x, 8)
PRINT BIN(-x, 8)
```

`BIN(-x)` würde, je nach System, 32 oder 64 Stellen ausgeben. Da Sie aber mit einem **UBYTE** arbeiten, sind für Sie vermutlich nur die hintersten acht Ziffern von Belang.

Da die mit **BIN()** usw. zurückgegebenen Werte Zeichenketten sind, können Sie mit

ihnen natürlich nicht einfach so weiterrechnen. Das ist allerdings nicht schlimm, da sich die Berechnungen in anderen Zahlensystemen nicht voneinander unterscheiden; es handelt sich lediglich um eine andere Darstellungsform der Werte bzw. der Ergebnisse. Sie können also getrost die Rechnung im Dezimalsystem durchführen (tatsächlich nutzt der Prozessor intern ja das Binärsystem) und das Ergebnis in das von Ihnen gewünschte System umwandeln.

Um mit Zahlen, die nach der Verwendung von **BIN()** usw. in einem String vorliegen, weiterzurechnen, können Sie sie mit **CINT()** (oder einer anderen Funktion aus dieser Gruppe) in das Dezimalsystem umwandeln. Beachten Sie aber, dass die Information, um welches System es sich aktuell handelt, im String nicht gespeichert wurde. Sie müssen sie also wieder „nachtragen“, und zwar wie oben durch ein **&** und den Kennbuchstaben, die vor dem Zahlenwert eingefügt werden müssen.

Quelltext 14.11: Umwandlung ins Binärsystem und zurück

```
DIM AS INTEGER dezimal = 19
DIM AS STRING  binaer  = BIN(dezimal)
PRINT "Die Dezimalzahl " & dezimal & " lautet binaer " & binaer
PRINT "Die Binaerzahl " & binaer & " lautet dezimal " & CINT("&b" & binaer)
SLEEP
```

Dem Binärzahl-String wird also vorn ein `"&b"` angefügt und das Ganze dann in die Funktion **CINT()** eingefügt.

Ausgabe

```
Die Dezimalzahl 19 lautet binaer 10011
Die Binaerzahl 10011 lautet dezimal 19
```

Für Oktalzahlen erlaubt FreeBASIC hier – und nur hier – auch das Weglassen des Kennbuchstaben o. Die et-Ligatur allein würde reichen. Mit dem o wird es jedoch deutlicher, welches Zahlensystem genau gemeint ist.

14.3. Bit-Manipulationen

Wie Sie ja bereits wissen, werden alle Daten intern als eine Ansammlung von Bits verwaltet. Umgekehrt könnte man auch sagen: Eine Gruppe von Bitwerten lässt sich auch als anderes Datenformat interpretieren, z. B. als Ganzzahl. Tatsächlich werden gern Ganzzahl-Datentypen verwendet, wenn mit Bits gearbeitet werden soll. So können die Werte sehr kompakt zwischen verschiedenen Programmteilen oder auch verschiedenen Programmen ausgetauscht werden.

Falls Sie sich fragen, was man mit einzelnen Bits denn so Interessantes anstellen kann: Jede Information, die auf zwei Zuständen beruht (wahr oder falsch, an oder aus, ja oder nein) entspricht einer Bit-Information. Wenn Sie etwa eine Eingabemaske bereitstellen, in der der Benutzer wählen kann, welche Optionen er nutzen will und welche nicht, stellt die Wahl für jede einzelne Option (angewählt oder abgewählt) ein Bit an Information dar. Die Bit-Manipulation von Ganzzahlen ist dann eine einfache Möglichkeit zur Verwaltung, wenn Bitfelder für Ihre Zwecke zu aufwendig sind (vgl. dazu Kapitel 7.4).

Im Übrigen werden natürlich auch Gleitkommazahlen binär gespeichert; hier macht der Zugriff auf einzelne Bit jedoch in aller Regel keinen Sinn, da sich das verwendete Speicherformat[26] nicht sehr für Direktmanipulationen eignet.

14.3.1. Manipulation über Bitoperatoren

In Kapitel 10.2.4 wurde bereits das Verhalten der Operatoren **AND**, **OR**, **XOR** usw. besprochen. Diese können gezielt dazu verwendet werden, den Status bestimmter Bits abzufragen. Ein einfacher Fall ist die bereits in Kapitel 14.1.5 angedachte Überprüfung, ob eine Zahl gerade ist. Da bei ungeraden Zahlen das niedrigste Bit gesetzt ist und bei geraden Zahlen nicht, kann man die Geradzahligkeit auch über **AND** prüfen.

Quelltext 14.12: Geradzahligkeit mit AND prüfen

```
DIM AS INTEGER x
INPUT "Geben Sie eine ganze Zahl ein: ", x
PRINT "Binaere Schreibweise der Zahl: "; BIN(x)
IF x AND 1 THEN
  PRINT "Das niedrigste Bit ist gesetzt, also ist"; x; " ungerade."
ELSE
  PRINT "Das niedrigste Bit ist nicht gesetzt, also ist"; x; " gerade."
END IF
SLEEP
```

Ausgabe

```
Geben Sie eine ganze Zahl ein: 42
Binaere Schreibweise der Zahl: 101010
Das niedrigste Bit ist nicht gesetzt, also ist 42 gerade.
```

Hilfreich sind die Bitoperatoren, wenn die einzelnen Bit einer Zahl jeweils eigene Informationen ausdrücken sollen. Nehmen wir als Beispiel eine Variable, die verschiedene Anzeigeoptionen eines Textes beinhalten soll. Wir legen (willkürlich) fest,

[26] FreeBASIC verwendet die allgemein übliche Norm IEEE 754.

dass Bit 0[27] angibt, ob der Text fett gedruckt sein soll. Bit 1 gibt eine Unterstreichung des Textes an, Bit 2 eine Kursivschrift. In Dezimalwerten ausgedrückt heißt das dann `1=fett`, `2=unterstrichen`, `4=kursiv`. Kombinationen lassen sich durch eine Verknüpfung mehrerer Werte erreichen, z. B. `3=fett+unterstrichen`, `6=unterstrichen+kursiv` und `7=alle Attribute`.

Ich habe hier bewusst allgemein von *Verknüpfung* gesprochen und nicht von *Addition*. Denn auch wenn der Wert für `fett+unterstrichen` durch die Addition von `fett` und `unterstrichen` entsteht, würde eine Addition von `fett` und `fett` den Wert `unterstrichen` ergeben – was natürlich keinen Sinn ergibt. Stattdessen kann die **OR**-Verknüpfung eingesetzt werden.

```
DIM AS INTEGER formatierung = 1 OR 4  ' fett und kursiv
PRINT "Formatierungsmuster: "; BIN(formatierung, 3)

IF (formatierung AND 1) = 1 THEN PRINT "fett"
IF (formatierung AND 2) = 2 THEN PRINT "unterstrichen"
IF (formatierung AND 4) = 4 THEN PRINT "kursiv"
SLEEP
```

Hier weicht die Verknüpfungslogik vom allgemeinen Sprachgefühl des Alltags ab. Wenn die Formatierung auf fett **UND** kursiv gesetzt werden soll, werden die zugehörigen Zahlen mit **OR** verknüpft. Das mag auf den ersten Blick verwirren. Sie sollten sich jedoch klar machen, dass **AND** und **OR** auf Bit-Ebene arbeiten und dass **OR** im Ergebnis immer dann ein Bit setzt, wenn bei mindestens einem der Operanden das Bit gesetzt ist. `1 OR 4` setzt daher Bit 0 (wegen der 1) und Bit 2 (wegen der 4).

AND dagegen prüft, welche Bit bei beiden Operanden gleichzeitig gesetzt sind. Mit `formatierung AND 2` können Sie feststellen, ob das Bit für die Unterstreichung gesetzt ist oder nicht, unabhängig davon, wie die anderen beiden Formatierungsarten eingestellt sind. Sie können auch prüfen, ob z. B. fett und kursiv gleichzeitig aktiviert sind oder ob beide deaktiviert sind – oder vieles mehr.

Bevor wir dazu ein Beispiel ansehen, gibt es noch einen Verbesserungsvorschlag zur Lesbarkeit. Da das Programm sehr kurz ist, kann man sich die Zuordnung `1=fett` usw. noch recht gut merken. Bei größeren Projekten und längerer Entwicklungsdauer kann das aber schon schwierig werden, gerade weil Sie mit der Zeit sicher auch noch andere Werte aus anderen Bereichen ins Programm einführen, deren Bedeutungen man sich alle merken müsste. Stattdessen ist es einfacher, die Werte in Konstanten mit sprechenden Namen abzulegen.

[27] Zur Erinnerung: Bit 0 ist das niedrigste Bit, das in der Binärschreibweise der Zahl ganz rechts steht.

Quelltext 14.13: Formateigenschaften mit OR und AND setzen und lesen

```
CONST FETT = 1, UNTERSTRICH = 2, KURSIV = 4
DIM AS INTEGER formatierung = FETT OR KURSIV
PRINT "Formatierungsmuster: "; BIN(formatierung, 3)

' Ist Fettschrift aktiviert?
IF (formatierung AND FETT) = FETT THEN PRINT "fett"

' Ist sowohl Fett- als auch Kursivschrift aktiviert?
IF (formatierung AND (FETT OR KURSIV)) = (FETT OR KURSIV) THEN
  PRINT "Fett und kursiv? Uebertreiben Sie es bitte nicht!"
END IF

' Sind alle drei Formatierungsformen deaktiviert?
IF (formatierung AND (FETT OR UNTERSTRICH OR KURSIV)) = 0 THEN
  PRINT "Es wurde keine Formatierung gewaehlt."
END IF

' nachtraeglich Fettschrift aktivieren
formatierung OR= FETT
PRINT "Fettschrift wurde aktiviert - neues Formatierungsmuster: ";
PRINT BIN(formatierung, 3)
SLEEP
```

Ausgabe

```
Formatierungsmuster: 101
fett
Fett und kursiv? Uebertreiben Sie es bitte nicht!
Fettschrift wurde aktiviert - neues Formatierungsmuster: 101
```

Da Fettschrift schon von Beginn an aktiviert ist, darf sich durch eine „nachträgliche“ Aktivierung nichts ändern. Mit **OR** ist das kein Problem, während eine Addition von FETT zu einem völlig falschen Ergebnis geführt hätte. Sie sehen in Quelltext 14.13 übrigens, dass auch für die Bitoperatoren die Kurzschreibweise **OR=** usw. zulässig ist.

Sie sehen, dass man mit logischen Operatoren eine ganze Menge anstellen kann, allerdings dauert es vielleicht anfangs eine Weile, bis man sich an die Arbeitsweise gewöhnt hat. Um effektiv programmieren zu können, ist allerdings das Verständnis der Bitmanipulationen unerlässlich.

Hinweis:
Wenn Sie mit der Funktionsweise der Bitoperatoren herumspielen, lassen Sie sich die Ergebnisse mit **BIN()** ausgeben. Dadurch sehen Sie am besten, welche Wirkung die einzelnen Operatoren erzielen.

Um – unabhängig von der aktuellen Einstellung – eine der Formatierungsbefehle umzustellen, bietet sich **XOR** an. Zur Erinnerung: **XOR** setzt im Ergebnis genau dann ein Bit, wenn sich die beiden entsprechenden Bit der beiden Operatoren unterscheiden, also genau dann, wenn es in einem der beiden Operatoren gesetzt ist und im anderen nicht.

```
CONST FETT = 1, UNTERSTRICH = 2, KURSIV = 4
DIM AS INTEGER formatierung = FETT OR KURSIV
PRINT "Formatierungsmuster vorher: "; BIN(formatierung, 3)

' Kursivschrift umstellen
formatierung XOR= KURSIV
PRINT "Formatierungsmuster danach: "; BIN(formatierung, 3)
```

Ausgabe

```
Formatierungsmuster vorher: 101
Formatierungsmuster danach: 001
```

Auch wenn Bitfelder meist einfacher in der Handhabung sind, gibt es doch Gründe für den Einsatz der Bitoperatoren. Um Bitfelder zu verwenden, ist die Deklaration eines UDTs nötig, und das gleichzeitige Setzen mehrerer Formatangaben wäre nicht möglich. Wenn Sie etwa ein Unterprogramm einsetzen wollen, dem die gewünschte Formatierung übergeben werden soll, ist es deutlich leichter, diese schnell über einen Zahlenwert zusammenzusetzen, als zuerst das passende UDT zu bauen (auch dazu gibt es aber programmiertechnische Lösungen). Ein schlagendes Argument ist aber die bessere Kompatibilität zu anderen Programmiersprachen, da Ganzzahldatentypen in nahezu jeder Sprache zur Verfügung stehen.

14.3.2. Einzelne Bit lesen und setzen

Wenn Sie gezielt ein einzelnes Bit auslesen wollen, kommt auch der Befehl **BIT()** infrage.

```
PRINT "Bit-Status des dritten Bit der Zahl 123:";
PRINT BIT(123, 3)
SLEEP
```

Die Nummerierung der Bits beginnt wie üblich bei `0` für das niederste Bit. Die Funktion gibt `-1` (=*wahr*) zurück, wenn das Bit gesetzt ist, und `0`, wenn das nicht der Fall ist. Um ein bestimmtes Bit zu setzen, dient **BITSET()**, um es zu löschen (also auf `0` zu setzen) **BITRESET()**.

```
DIM AS INTEGER zahl = 123
PRINT zahl; " mit gesetztem 2. Bit:  "; BITSET(zahl, 2)
PRINT zahl; " mit geloeschtem 3. Bit:"; BITRESET(zahl, 3)
SLEEP
```

Beachten Sie, dass **BITSET()** und **BITRESET()** nicht den Variablenwert selbst verändern, sondern lediglich den Wert zurückgeben, der durch die Veränderung entsteht. Wollen Sie in der Variablen selbst ein Bit verändern, müssen Sie das Ergebnis von **BITSET()** bzw. **BITRESET()** wieder der Variablen zuweisen.

Die in Quelltext 14.13 eingesetzte Einstellung von Formatangaben lässt sich mit den drei Befehlen ebenfalls umsetzen. Da nun die Position des Bits gefragt ist und nicht seine Wertigkeit, geht die Nummerierung nun von 0 bis 2 statt über die Werte 1, 2 und 4. Deswegen lässt sich hier sehr gut **ENUM** einsetzen. Ein Vorteil ist zudem, dass möglicherweise später neu hinzukommende Formateigenschaften nur zur Liste hinzugefügt werden müssen, ohne eine Berechnung der Wertbelegung vornehmen zu müssen.

Quelltext 14.14: Formateigenschaften mit BITSET() und BIT() setzen und lesen

```
ENUM FORMATE
  fett, unterstrich, kursiv
END ENUM
DIM AS INTEGER formatierung
formatierung = BITSET(formatierung, FORMATE.fett)   ' Fettschrift aktivieren
formatierung = BITSET(formatierung, FORMATE.kursiv) ' Kursivschrift aktivieren
PRINT "Formatierungsmuster: "; BIN(formatierung, 3)

' Ist Fettschrift aktiviert?
IF BIT(formatierung, FORMATE.fett) THEN PRINT "fett"

' Ist sowohl Fett- als auch Kursivschrift aktiviert?
IF BIT(formatierung, FORMATE.fett) AND BIT(formatierung, FORMATE.kursiv) THEN
  PRINT "Fett und kursiv? Uebertreiben Sie es bitte nicht!"
END IF

' Sind alle drei Formatierungsformen deaktiviert?
IF BIT(formatierung, FORMATE.fett) = 0 _
   AND BIT(formatierung, FORMATE.unterstrich) = 0 _
   AND BIT(formatierung, FORMATE.kursiv) = 0 THEN
   PRINT "Es wurde keine Formatierung gewaehlt."
END IF

' nachtraeglich Fettschrift aktivieren
formatierung = BITSET(formatierung, FORMATE.fett)
PRINT "Fettschrift wurde aktiviert - neues Formatierungsmuster: ";
PRINT BIN(formatierung, 3)
SLEEP
```

Die Ausgabe ist identisch mit der Ausgabe zu Quelltext 14.13.

14.3.3. Vergleich

Beide Varianten – Bitoperatoren und **BITSET()**/**BITRESET()** – haben ihre Vorteile. Bitoperatoren sind nicht nur auf 1-Bit-Werte beschränkt, und es ist auch ein gleichzeitiger Zugriff auf mehrere Formatangaben möglich. Auf der anderen Seite ist das Setzen eines einzelnen Bits mit **BITSET()** deutlich anschaulicher. Auch das Löschen eines Bits mit Bitoperatoren sieht recht umständlich aus, während es mit **BITRESET()** sehr klar ist. Wenn Sie allerdings die Bitrechnung beherrschen, stellt auch das kein großes Problem dar.

```
formatierung = BITRESET(formatierung, FORMATE.fett)   ' keine Fettschrift mehr
formatierung AND= NOT FETT                            ' Alternative mit AND
```

14.3.4. LOBYTE() und seine Freunde

Ähnlich wie **BIT()**, aber auf einen größeren Datenbereich bezogen, wirkt **LOBYTE()**. Wie der Name andeutet, wird hier nicht ein Bit-, sondern ein Byte-Zustand abgefragt, und zwar das untere Byte der übergebenen Zahl (*LOw*: niedrig). `LOBYTE(zahl)` ist identisch mit `zahl AND &hFF`, d. h. alles außer den unteren acht Bit (oder eben des unteren Byte) der Zahl wird ignoriert. Die eng verwandte Funktion **HIBYTE()** (*HIgh*: hoch) gibt das Byte oberhalb des **LOBYTE()** zurück.

Wenn mit Byte-Zuständen gearbeitet wird, verwendet man gern das Hexadezimalsystem, da dort ein Byte genau aus zwei Ziffern besteht. Hexadezimal gesprochen sind **LOBYTE()** die untersten zwei Ziffern und **HIBYTE()** die beiden Ziffern links davon.

Noch größere Bereiche decken **LOWORD()** und **HIWORD()** ab. Ein Word bzw. Datenwort ist eine Dateneinheit von zwei Byte. Wie sich jetzt leicht erschließen lässt, enthält ein **LOBYTE()** die unteren beiden Byte des übergebenen Parameters und **HIBYTE()** die zwei Byte darüber.

Falls sich das noch etwas kompliziert anhört – ein Beispiel sollte hier Klarheit schaffen.

Quelltext 14.15: Byte- und Word-Zugriff

```
DIM AS LONGINT zahl(2) = { &h123, &hC0DEBA5E, &hFEEDBABEF00D }
PRINT "Vorgabewert  | HIBYTE | LOBYTE | HIWORD | LOWORD"
PRINT "=============+========+========+========+======="
FOR i AS INTEGER = 0 TO 2
  PRINT HEX(zahl(i), 12); " |   ";
  PRINT HEX(HIBYTE(zahl(i)), 2); "   |   ";
  PRINT HEX(LOBYTE(zahl(i)), 2); "   |  ";
  PRINT HEX(HIWORD(zahl(i)), 4); "  |  ";
  PRINT HEX(LOWORD(zahl(i)), 4)
NEXT
SLEEP
```

Ausgabe

```
Vorgabewert  | HIBYTE | LOBYTE | HIWORD | LOWORD
=========================================
000000000123 |   01   |   23   |  0000  |  0123
0000C0DEBA5E |   BA   |   5E   |  C0DE  |  BA5E
FEEDBABEF00D |   F0   |   0D   |  BABE  |  F00D
```

14.3.5. Bit-Verschiebung

Es gibt noch zwei Operatoren, die eigentlich in den Bereich der mathematischen Operatoren fallen, aber vorwiegend zur Bit-Manipulation eingesetzt werden: **SHL** (*SHift Left*) schiebt in einer Ganzzahl alle Bit um eine angegebene Anzahl an Stellen nach links, **SHR** (*SHift Right*) schiebt sie nach rechts.

An einem konkreten Beispiel bedeutet das: Die Zahl `27` hat die Binärdarstellung 11011_2. Schiebt man alle Bit um eins nach links, erhält man 110110_2 bzw. die Dezimalzahl `54`. Schiebt man dagegen alle Bit um eins nach rechts, erhält man 1101_2 bzw. die Dezimalzahl `13`. Ein Bit, das „über den Rand hinaus“ geschoben wird, verschwindet also – und zwar auch bei **SHL**, wenn der Datentyp, in dem das Ergebnis gespeichert werden soll, zu klein ist.

Die Syntax für **SHL** (und selbstverständlich analog dazu auch für **SHR**) sieht folgendermaßen aus:

```
' Zuweisung in eine neue Variable
Ergebnis = Ausgangswert SHL Anzahl
' Kurzschreibweise
Wert SHL= Anzahl   ' ist identisch mit 'Wert = Wert SHL Anzahl'
```

Ausgangswert ist der Wert, für den die Verschiebung stattfinden soll, und *Anzahl* ist die Zahl der Stellen, um die verschoben werden soll. In der Kurzschreibweise `SHL=` wird der Wert des Ergebnisses direkt wieder in der Ausgangszahl gespeichert; ansonsten verändern **SHL** und **SHR** den Wert ihrer Operanden nicht. Natürlich können die Operatoren auch in Berechnungen eingebaut werden, ebenso wie z. B. der Operator + oder **OR**.

Tatsächlich erreichen Sie mit **SHL** dasselbe wie bei der Multiplikation mit einer Zweierpotenz und mit **SHR** entsprechend dasselbe wie bei der Ganzzahl-Division. Allerdings arbeiten **SHL** und **SHR** in der Regel schneller. Bei der Verwendung vorzeichenbehafteter Typen und einer Verschiebung über die Datentypgrenzen ist das Ergebnis möglicherweise überraschend, aber auch hier unterscheidet es sich nicht vom Ergebnis einer passenden Multiplikation.

Quelltext 14.16: SHL und SHR

```
DIM b AS BYTE, u AS UBYTE
PRINT "12 SHL 3 = "; 12 SHL 3,     "12*(2^3) = "; 12*(2^3)
PRINT "12 SHL 4 = "; 12 SHL 4,     "12*(2^4) = "; 12*(2^4)
PRINT "12 SHL 5 = "; 12 SHL 5,     "12*(2^5) = "; 12*(2^5)
PRINT

b = 12 SHL 4
PRINT "12 SHL 4 (als BYTE)  = " & b
u = 12 SHL 4
PRINT "12 SHL 4 (als UBYTE) = " & u
b = 12 SHL 5
PRINT "12 SHL 5 (als BYTE)  = " & b
u = 12 SHL 5
PRINT "12 SHL 5 (als UBYTE) = " & u
PRINT

PRINT "12 SHR 2 = "; 12 SHR 2,,    "12\(2^2) = "; 12^2)
PRINT "12 SHR 3 = "; 12 SHR 3,,    "12\(2^3) = "; 12^3)
```

Ausgabe

```
 12 SHL 3 =  96              12*(2^3) =  96
 12 SHL 4 =  192             12*(2^4) =  192
 12 SHL 5 =  384             12*(2^5) =  384

 12 SHL 4 (als BYTE)  = -64
 12 SHL 4 (als UBYTE) = 192
 12 SHL 5 (als BYTE)  = -128
 12 SHL 5 (als UBYTE) = 128

 12 SHR 2 =  3               12\(2^2) =  3
 12 SHR 3 =  1               12\(2^3) =  1
```

Da in Quelltext 14.16 alle zur Berechnung verwendeten Werte Konstanten sind, kann der Compiler schon während der Compilierzeit feststellen, dass in Zeile 11 und Zeile 13 über den verfügbaren Bereich hinausgeschoben wird, also dass das Ergebnis zu groß ist, um in ein **BYTE** bzw. **UBYTE** zu passen. In diesem Fall gibt der Compiler eine Warnung aus. Der Code wird trotzdem compiliert und kann anschließend ausgeführt werden, trotzdem sollten Sie auf Warnungen des Compilers achten (und sie am besten beheben).

14.4. Fragen zum Kapitel

1. Schreiben Sie eine Funktion zur Erzeugung von Zufallszahlen: Die Funktion erhält als Parameter einen Start– und einen Endwert und gibt eine Zufallszahl im Bereich dieser beiden Werte zurück.

2. Schreiben Sie ein Programm, das in einer vorgegebenen Zahl die unten angegebenen Modifikationen durchführt. Versuchen Sie dabei möglichst mehrere verschiedene Ansätze durchzuführen.
   a) Setze die niedrigsten 4 Bit alle auf 0 (alle anderen Bit bleiben gleich).
   b) Drehe alle Bitwerte um – aus 1 wird 0 und umgekehrt.
   c) Drehe die Bitwerte der niedrigsten 4 Bit um (alle anderen Bit bleiben gleich).

3. Überlegen Sie sich eine Umsetung der in Quelltext 14.13 und Quelltext 14.14 verwendeten Formateigenschaften mithilfe der in Kapitel 7.4 vorgestellten Bitfeldern.

15. Stringmanipulation

Wie bereits aus Kapitel 6.3.1 bekannt ist, gibt es drei verschiedene Arten von Zeichenketten: den **STRING**, der eine zusätzliche Speicherung seiner Länge enthält, der **ZSTRING**, welcher durch ein Nullbyte beendet wird, und der **WSTRING**, der ebenfalls durch Nullbytes beendet wird, aber Unicode-Zeichen speichern kann. Viele der in diesem Kapitel behandelten Befehle für die Stringmanipulation können sowohl bei einem **STRING** als auch bei einem **ZSTRING** verwendet werden, ohne dass Sie sich viele Gedanken um die Unterschiede zwischen beiden Speicherarten machen müssen. Einige sind auch für einen **WSTRING** einsetzbar, allerdings nicht alle – oft gibt es dafür eine eigene Variante des Befehls. Gehen Sie bei den folgenden Befehlen davon aus, dass die Angaben für alle drei Arten von Zeichenketten gelten, außer es ist anderweitig beschrieben.

15.1. Speicherverwaltung

15.1.1. Verwaltung eines ZSTRING

Die Speicherverwaltung bei einem **ZSTRING** ist die einfachste der drei Varianten. Für die Zeichenkette wird ein bestimmter Speicherbereich reserviert. Dort werden die einzelnen Byte hintereinander abgelegt. Wegen der Einfachheit in der Verwaltung verwendet man diesen Datentyp auch sehr gern zum Austausch zwischen Programmteilen, die in verschiedenen Programmiersprachen geschrieben sind.

Damit FreeBASIC weiß, wie viel Speicher für den **ZSTRING** belegt werden soll, muss dies im Programm explizit mitgeteilt werden, z. B. über einen **ZSTRING** fester Länge. Eine zweite Möglichkeit ist die Verwendung von Pointern, verbunden mit einer Speicherreservierung „von Hand“; darauf werden wir aber erst später zu sprechen kommen. Ein **ZSTRING** variabler Länge kann dagegen nicht angelegt werden.

„**ZSTRING** fester Länge“ bedeutet im Übrigen nur, dass die *maximale* Länge der Zeichenkette festgelegt wird – und damit verbunden der reservierte Speicherplatz. Die Zeichenkette, die tatsächlich gespeichert wird, darf durchaus kürzer sein. Die Auswertung des **ZSTRING**s wird immer beim ersten auftretenden Nullbyte gestoppt, alle folgenden Zeichen werden ignoriert. Allerdings macht es keinen Sinn, einen 1000 Zeichen langen

ZSTRING zu reservieren, wenn man nur vorhat, höchstens zehn Zeichen lange Ketten zu speichern, da man so unnötig Speicherplatz verschwenden würde.

```
' DIM AS ZSTRING varlength          ' nicht moeglich (ZSTRING variabler Laenge)
DIM AS ZSTRING*5 fixlength          ' ZSTRING fester Laenge: 4 Zeichen + Nullbyte
DIM AS ZSTRING PTR zeiger           ' Pointer; Speicherreservierung noch erforderlich
```

Während `fixlength` sofort einsatzbereit ist und mit einem Wert belegt werden kann, wurde in der letzten Zeile noch keine Speicherreservierung durchgeführt. `zeiger` steht daher noch auf dem Ausgangswert `0`, und der Versuch einer Wertzuweisung wird scheitern. Beachten Sie bei `fixlength` auch, dass das abschließende Nullbyte ebenfalls mit einberechnet werden muss. Um einen Text wie `"Hallo"` zu speichern, ist mindestens ein **ZSTRING*6** erforderlich.

15.1.2. Verwaltung eines WSTRING

Bei einem **WSTRING** gelten für die Speicherverwaltung im Großen und Ganzen dieselben Regeln wie beim **ZSTRING**. Da ein **WSTRING** jedoch deutlich mehr (Unicode-)Zeichen beinhaltet, müssen die einzelnen Zeichen auch mehr Speicherplatz belegen. Davon merken Sie beim Programmieren allerdings nur etwas, wenn Sie sich gezielt mit der Speicherverwaltung beschäftigen, z. B. wenn Sie den Dateninhalt direkt über Speicherzugriffe manipulieren wollen oder die Gesamtgröße der Datentypen eine entscheidende Rolle spielt.

Unterschiede zwischen den Plattformen:
Wie viel Platz ein Unicode-Zeichen im Speicher belegt, hängt vom Betriebssystem ab. Windows codiert die Zeichen in UCS-2 und belegt zwei Byte pro Zeichen, unter Linux kommt UCS-4 zum Einsatz, wodurch vier Byte pro Zeichen belegt werden. Das betrifft auch das abschließende Nullbyte, das in Wirklichkeit aus zwei bzw. viel Nullbyte besteht.

Der Umgang mit **WSTRING** hält am Anfang einige Schwierigkeiten bereit, die nicht immer leicht zu durchschauen sind. Zunächst einmal kann FreeBASIC nur dann korrekt mit Unicode-Dateien umgehen, wenn die entsprechende Datei in Unicode mit *Byte Order Mark* (BOM) gesetzt ist. Eine Ausgabe über Konsole funktioniert natürlich nur, wenn dies von der Konsole auch unterstützt wird, und eine korrekte Ausgabe im Grafikfenster ist ohne externe Bibliotheken gar nicht erst möglich. Daher empfehle ich Ihnen, sich zuerst gründlich mit **STRING** und **ZSTRING** zu beschäftigen, bevor Sie sich an **WSTRING** wagen.

15.1.3. Verwaltung eines STRING

Ein **STRING** *fester* Länge wird im Prinzip genauso verwaltet wie ein **ZSTRING**, nur dass ein Nullbyte nicht das Ende markiert. Ein **STRING*10** ist *immer* 10 Byte lang, egal wie sein Inhalt lautet. Ein **ZSTRING*10** dagegen belegt immer einen Speicherbereich von 10 Byte, kann aber 0 bis 9 Byte lang sein.

Achtung:
Wenn Sie einen **STRING** fester Länge als Parameter einer Funktion oder Prozedur verwenden, wird er in aller Regel als **ZSTRING** übergeben, und die Terminierung durch das Nullbyte schlägt wieder zu.

Bei einem **STRING** *variabler* Länge muss, im Gegensatz zu den vorigen Datentypen, die aktuelle Länge mitgespeichert werden. Außerdem läuft die Reservierung und Freigabe des benötigten Speicherbereichs vollkommen automatisch. Dies ist nur möglich, weil der Zeiger auf den **STRING** gar nicht auf den **STRING**-Inhalt selbst zeigt, sondern auf einen Bereich mit Metadaten. Damit ist der **STRING** zum einen intern der komplizierteste Datentyp für Zeichenketten, zum anderen aber auch der am leichtesten zu bedienende.

Der **STRING**-Pointer zeigt auf einen drei **INTEGER** großen Header-Bereich (also je nach Betriebssystem 12 bzw. 24 Bit). Dort ist der Reihe nach die Adresse der Zeichenkette, die aktuelle Länge der Zeichenkette und die Größe des aktuell für die Zeichenkette reservierten Speicherbereichs hinterlegt. Wenn Sie eine **STRING**-Variable anlegen (und noch keine Wert-Zuweisung vornehmen), benötigt der **STRING** den Speicherplatz von drei **INTEGER** für die Metadaten, die alle drei den Wert `0` besitzen. Sobald dem **STRING** eine Zeichenkette zugewiesen wird, reserviert FreeBASIC einen zusätzlichen Speicherbereich für den Inhalt der Zeichenkette und aktualisiert die Metadaten entsprechend.

Bei einer sich häufig ändernden **STRING**-Länge hätte dieses Verfahren den Nachteil, dass der Speicherbereich für die Zeichenkette immer wieder gewechselt werden müsste und es dadurch schnell zu einer Speicherfragmentierung käme. Um dem entgegenzuwirken, reserviert FreeBASIC in der Regel mehr Speicherplatz für die Zeichenkette, als dringend erforderlich. Insbesondere bei einer Verkleinerung des **STRING**s wird kein neuer Speicherplatz gesucht, sondern der bisher verwendete Bereich beibehalten. Nur wenn Sie der Variablen einen Leerstring zuweisen, wird der Speicherplatz für die Zeichenkette wieder komplett freigegeben.

Hinweis:
Um eine bessere Kompatibilität zwischen **STRING** und **ZSTRING** (und damit zu externen Bibliotheken) zu erhalten, hängt FreeBASIC – im Gegensatz zu QuickBASIC – an das Ende eines **STRING**s fester und variabler Länge ein zusätzliches Nullbyte an. Dieses terminiert die Zeichenkette jedoch nicht und tritt nur in Erscheinung, wenn Sie direkt auf die Speicherbereiche zugreifen oder wenn Sie einen Datenaustausch zwischen FreeBASIC und QuickBASIC vornehmen wollen.

15.2. Stringfunktionen

15.2.1. LEN(): Länge eines Strings ermitteln

Kurz und bündig: **LEN()** gibt zurück, wie viele Zeichen ein String enthält. Dass ein **WSTRING** mehrere Byte Speicherplatz pro Zeichen benötigt, spielt dabei keine Rolle; zurückgegeben wird die Anzahl der Zeichen, nicht der Speicherbedarf.

Quelltext 15.1: LEN() zur Ermittlung der String-Länge

```
DIM AS STRING     halloString  = "Hallo, Welt!"
DIM AS WSTRING*99 halloWString = "Hallo, Welt1"
PRINT "STRING:                "; LEN(halloString)
PRINT "WSTRING fester Laenge: "; LEN(halloWString)

' und weil das nun doch etwas zu einfach gewesen waere
DIM AS STRING*99  halloFixString  = "Hallo, Welt!"
PRINT "STRING fester Laenge:  "; LEN(halloFixString)
SLEEP
```

Ausgabe

```
STRING:                 12
WSTRING fester Laenge:  12
STRING fester Laenge:   99
```

Wäre auch zu schön gewesen. Aber **STRING**s fester Länge haben eben wirklich eine feste Länge, ganz egal, welchen Inhalt sie besitzen. **WSTRING** und **ZSTRING** dagegen enden immer beim ersten auftretenden Nullbyte.

15.2.2. Die Funktionen SPACE() und STRING()

Ja, es gibt eine Stringfunktion namens **STRING()**. Sie gibt ganz einfach einen String zurück, der aus lauter gleichen Zeichen besteht. In Kapitel 12.3.3 wurde bereits **SPACE()** vorgestellt, das eine Zeichenkette aus einer beliebigen Anzahl an Leerzeichen erstellt. **STRING()** bewirkt dasselbe, nur dass Sie sich das Füllzeichen selbst aussuchen können. Dieses Füllzeichen kann als ASCII-Code oder als String angegeben werden (von einem solchen String wird aber nur das erste Zeichen verwendet).

Quelltext 15.2: STRING() und SPACE()

```
' erzeuge eine Zeichenkette aus 20 Leerzeichen
DIM AS STRING leer20
leer20 = STRING(20, " ")
' bewirkt dasselbe wie    leer20 = SPACE(20)
' oder auch               leer20 = STRING(20, 32)
' Der ASCII-Code des Leerzeichens ist 32.

' Textunterstreichung
DIM AS STRING satz = "Dieser Satz ist unterstrichen."
PRINT satz
PRINT STRING(LEN(satz), "=")
SLEEP
```

Ausgabe

```
Dieser Satz ist unterstrichen.
==============================
```

Die Länge der Unterstreichung wird automatisch aus der Länge des Satzes ermittelt. Eine solche Art von Textunterstreichung wäre auch in Quelltext 12.10 recht praktisch.

Wenn Sie statt eines Strings einen **WSTRING** erstellen wollen, der aus lauter gleichen Zeichen besteht, greifen Sie stattdessen auf die Funktionen **WSTRING()** bzw. **WSPACE()** zurück.

15.2.3. LEFT(), RIGHT() und MID()

FreeBASIC stellt einige Funktionen bereit, um einen Teilstring auszulesen. Die gängigsten sind **LEFT()**, **RIGHT()** und **MID()**. Mit **LEFT()** erhalten Sie die ersten x Zeichen des Strings, mit **RIGHT()** die letzten x Zeichen – mit **MID()** lesen Sie eine Zeichenkette mitten aus einem String aus.

LEFT() erwartet als Parameter zunächst einmal den Originalstring und des Weiteren die Anzahl der Zeichen, die zurückgegeben werden sollen. **RIGHT()** funktioniert prinzipiell

genauso, nur dass nicht die vorderen, sondern die hinteren Zeichen zurückgegeben werden.

Quelltext 15.3: LEFT() und RIGHT()

```
DIM AS STRING halloWelt = "Hallo, Welt!"
PRINT LEFT(halloWelt, 5)                    ' die ersten 5 Zeichen
DIM AS STRING welt = RIGHT(halloWelt, 5)    ' die letzten 5 Zeichen
PRINT welt
PRINT ">"; LEFT(halloWelt, 50); "<"         ' die ersten 50 Zeichen (bricht
SLEEP                                       '   nach dem letzten Zeichen ab)
```

Ausgabe

```
Hallo
Welt!
>Hallo, Welt!<
```

Die beiden Funktionen führen eine automatische Überprüfung der Stringlänge durch. Wenn Sie mehr Zeichen auslesen wollen, als der String Zeichen hat, erhalten Sie den kompletten String zurück. Sie können also nicht versehentlich über den Speicherbereich hinaus lesen und dadurch einen Programmabsturz riskieren. Die dadurch erhaltene Sicherheit wird natürlich mit Geschwindigkeit bezahlt: durch die notwendige Überprüfung sind die beiden Funktionen relativ langsam (was aber nur ins Gewicht fällt, wenn der Aufruf sehr häufig wiederholt wird).

Auch **MID()** führt eine solche Sicherheitsüberprüfung durch. **MID()** erwartet drei Parameter: den Originalstring, die Startposition für den Lesevorgang und die Anzahl der Zeichen, die ausgelesen werden sollen. Wenn Sie mehr Zeichen lesen wollen, als zur Verfügung stehen, gibt **MID()** ganz einfach den Rest des Strings ab der Startposition bis zum Ende aus. Sie können sich aber auch von Vornherein alle Zeichen von einem Startpunkt bis zum Stringende ausgeben lassen – lassen Sie dazu einfach die Längenangebe weg oder geben Sie eine negative Länge an.

Quelltext 15.4: MID

```
DIM AS STRING halloWelt = "Hallo, Welt!"
PRINT MID(halloWelt, 5, 4)                  ' 4 Zeichen ab dem 5. Zeichen
DIM AS STRING welt = MID(halloWelt, 8)      ' alles ab dem 8. Zeichen
PRINT welt
PRINT MID(halloWelt, 8, 50)                 ' alles ab dem 8. Zeichen
SLEEP                                       '   (weil der String zu kurz ist)
```

Ausgabe

```
o, W
Welt!
Welt!
```

Wenn Sie „unsinnige“ Werte einsetzen (z. B. indem Sie bei **LEFT()** oder **RIGHT()** eine negative Anzahl an Zeichen auslesen wollen oder bei **MID** als Startwert keine positive Zahl eingeben), erhalten Sie als Rückgabe einen Leerstring. Negative Zahlen sind nur als Längenangabe für **MID()** erlaubt, um den kompletten Reststring ab einem bestimmten Startwert zu erhalten.

15.2.4. Zeichenketten modifizieren

Die oben genannten Funktionen geben einen Teilstring zurück; der ursprüngliche String wird dabei nicht verändert. **MID** gibt es jedoch auch als Anweisung, um einen Teil des Strings durch einen anderen zu ersetzen. Weitere Befehle zur direkten Modifikation einer Zeichenkette sind **LSET** und **RSET**

Die drei Befehle stammen aus frühen QBasic-Zeiten und sind aus Kompatibilitätsgründen immer noch in FreeBASIC verfügbar. Persönlich würde ich von einer Verwendung abraten und sie durch flexiblere Prozeduren ersetzen (dazu später ein Beispiel in Quelltext 15.5). Der Vollständigkeit halber werden sie hier aber aufgeführt.

MID als Anweisung

Als Anweisung besitzt **MID** einen etwas anderen Aufbau:

```
MID(Zeichenkette, Startzeichen, Zeichenzahl) = Ersetzungsstring
```

`Zeichenzahl` kann auch weggelassen werden; dann richtet sich die Anzahl der ersetzten Zeichen nach der Länge des Ersetzungsstrings.

Quelltext 15.5: Teilstring ersetzen mit MID

```
DIM AS STRING halloWelt = "Hallo, Welt!"
MID(halloWelt, 9, 3) = "ald"              ' Teilstring ersetzen
' MID(halloWelt, 9) = "ald"               ' bewirkt dasselbe
PRINT halloWelt
MID(halloWelt, 9, 3) = "ienerle"          ' Teilstring ersetzen
PRINT halloWelt
SLEEP
```

Ausgabe

```
Hallo, Wald!
Hallo, Wien!
```

Gut, das war jetzt nicht besonders originell. Aber es sollte deutlich geworden sein, was passiert ist: Ab dem neunten Zeichen wurden drei Zeichen (ursprünglich “elt“) durch “ald“ ersetzt. Leider hat diese Methode eine deutliche Einschränkung: Die Länge des Originalstrings kann damit nicht verändert werden. Das heißt: Sie können immer nur einen Teilstring durch einen *gleich langen* String ersetzen. Das sieht man im zweiten Ersetzungsversuch: Da nur drei Zeichen ersetzt werden sollen, werden auch nur die ersten drei Zeichen aus dem Ersetzungsstring genommen. Wenn übrigens der Ersetzungsstring zu kurz ist, werden auch nur so viele Zeichen ersetzt, wie vorhanden sind.

```
DIM AS STRING text = "abcdefgh"
MID(text, 3, 5) = "XY"
PRINT text
SLEEP
```

Ausgabe

```
abXYefgh
```

Meiner Meinung nach muss man sinnvolle Anwendungsmöglichkeiten für die Anweisung **MID** schon ziemlich konstruieren. Als Beispiel dazu eine Möglichkeit zur Textzensur:

Quelltext 15.6: Textzensur mit MID

```
DIM AS STRING sternchen = STRING(500, "*")
DIM AS STRING anweisung = "Geben Sie den Benutzernamen admin " _
                          & "und das Passwort abc123 ein."
MID(anweisung, 29, 5) = sternchen
MID(anweisung, 52, 6) = sternchen
PRINT anweisung
SLEEP
```

Ausgabe

```
Geben Sie den Benutzernamen ***** und das Passwort ****** ein.
```

Hier spielt es bei der Erstellung des Ersetzungsstrings `sternchen` keine Rolle, wie viele Zeichen später gebraucht werden – `sternchen` stellt einfach ausreichend viele

Zeichen für alle Fälle bereit. Wie viele dann tatsächlich im Ausgangsstring eingesetzt werden, entscheidet sich direkt in der Anweisung **MID**.

LSET und RSET

LSET und **RSET** füllen einen String links- bzw rechtsbündig durch einen anderen String auf.

```
LSET Ziel, Quelle
```

Der Inhalt von `Ziel` wird durch `Quelle` ersetzt, jedoch bleibt die Länge von `Ziel` gleich. Ist `Quelle` länger als `Ziel`, dann werden nur die ersten Zeichen von `Quelle` verwendet (so viele, wie in `Ziel` hineinpassen). Ist dagegen `Ziel` länger, dann werden die fehlenden Zeichen durch Leerzeichen aufgefüllt, und zwar bei **LSET** hinten (der neue Inhalt von `Ziel` ist dann linksbündig) und bei **RSET** vorn (der neue Inhalt von `Ziel` ist dann rechtsbündig).

Quelltext 15.7: LSET und RSET

```
DIM AS STRING quelle, ziel = "0123456789"
PRINT ziel
' linksbuendiger Text
quelle = "links"
LSET ziel, quelle
PRINT ziel
' rechtsbuendiger Text
quelle = "rechts"
RSET ziel, quelle
PRINT ziel
SLEEP
```

Ausgabe

```
0123456789
links
    rechts
```

15.2.5. Führende/angehängte Zeichen entfernen: LTRIM(), RTRIM() und TRIM()

Kommen wir wieder zu deutlich praktischeren Funktionen. **LTRIM()** (*Left TRIM*) entfernt bestimmte Zeichen vom Anfang der Zeichenkette. Zunächst ist die Funktion dazu gedacht, führende Leerzeichen zu entfernen, jedoch kann auch jedes beliebige Zeichen oder auch

eine bestimmte Zeichenfolge entfernt werden. **RTRIM()** (*Right TRIM*) arbeitet analog dazu von rechts und entfernt Zeichen oder Zeichenfolgen am Ende der Zeichenkette.

Quelltext 15.8: LTRIM() und RTRIM()

```
DIM AS STRING eingabe = "     Die besten Reden sind die schnell endenden"

' Ohne weitere Angabe werden die Leerzeichen entfernt:
PRINT LTRIM(eingabe)

' Es kann auch ein anderes Zeichen entfernt werden:
PRINT RTRIM(eingabe, "n")

' Entfernung der Zeichenfolge 'den':
PRINT RTRIM(eingabe, "den")
SLEEP
```

Ausgabe

```
Die besten Reden sind die schnell endenden
     Die besten Reden sind die schnell endende
     Die besten Reden sind die schnell en
```

In der letzten Zeile wurde die Zeichenfolge "den" zweimal abgeschnitten, weil sie zweimal vorkam. Jetzt endet der Text zwar wieder auf "en", aber nicht auf "den", womit nicht weiter gekürzt wird. Beachten Sie auch, dass die Variable `eingabe` das ganze Programm hinweg denselben Wert behält. Die Kürzung findet nur beim zurückgegebenen Funktionswert statt (der in diesem Fall nicht gespeichert, sondern direkt über **PRINT** ausgegeben wird).

Wenn Sie nicht die genaue Zeichenfolge "den" abschneiden wollen, sondern jedes Auftreten eines der Buchstaben d, e und n, können Sie das Schlüsselwort **ANY** benutzen.

Quelltext 15.9: RTRIM() mit Schlüsselwort ANY

```
DIM AS STRING eingabe = "     Die besten Reden sind die schnell endenden"

' entfernt am Ende jedes d, e, l, n oder Leerzeichen
PRINT RTRIM(eingabe, ANY "deln ")
SLEEP
```

Ausgabe

```
      Die besten Reden sind die sch
```

ANY bietet sich auch an, wenn Sie jede Art von Whitespace entfernen wollen. Dazu

gehören Tabulatoren (ASCII-Code 9), Zeilenumbruch (je nach Betriebssystem ASCII-Codes 10 und 13) und Leerzeichen (ASCII-Code 32).

```
DIM AS STRING eingabe
' alle Whitespace-Zeichen am Anfang entfernen
PRINT LTRIM(eingabe, ANY CHR(9, 10, 13, 32))
SLEEP
```

TRIM() vereinigt die beiden Funktionen **LTRIM()** und **RTRIM()** und entfernt die gewünschten Zeichen sowohl am Anfang als auch am Ende der Zeichenkette. Auch hier kann **ANY** eingesetzt werden.

15.2.6. Umwandlung in Groß- bzw. Kleinbuchstaben: UCASE() und LCASE()

UCASE() (*Upper CASE*) wandelt einen String komplett in Großbuchstaben um, **LCASE()** (*Lower CASE*) in Kleinbuchstaben. Natürlich können nur die Zeichen umgewandelt werden, zu denen ein Groß- bzw. Kleinbuchstabe existiert – das wären zunächst einmal die lateinischen Buchstaben a-z. Umlaute oder Buchstaben mit Akzent werden nicht umgewandelt. Sie bleiben, wie auch alle Ziffern und Sonderzeichen, unverändert.

Quelltext 15.10: Alles gross oder alles klein

```
DIM AS STRING text = "Mal GROSS, mal klein!"
' alles in Gross
PRINT UCASE(text)
' alles in Klein:
PRINT LCASE(text)
SLEEP
```

Umlaute gehören genauso wie andere länderspezifische Sonderbuchstaben nicht zum ASCII-Zeichensatz. Die Verwendung in einem String wirft daher Probleme auf. Einen kurzen Abriss über das Thema Umlaute lesen Sie in Kapitel 15.3.

15.2.7. Teilstring suchen: INSTR() und INSTRREV()

INSTR() durchsucht einen String nach einem Teilstring und gibt die erste Fundstelle zurück. Optional kann eine Startposition angegeben werden, ab der die Suche beginnt. Wie bei **TRIM()** kann der Suchstring aus mehreren Zeichen bestehen, und die Angabe des Schlüsselwortes **ANY** ermöglicht die Suche nach einem beliebigen Zeichen aus dem Suchstring.

Quelltext 15.11: Teilstring suchen

```
DIM AS STRING text = "Ich bin ein Satz mit sechs Leerzeichen."
DIM AS INTEGER position1, position2

' erstes Leerzeichen suchen
position1 = INSTR(text, " ")
PRINT "Das erste  Leerzeichen befindet sich an Position"; position1
' zweites Leerzeichen suchen
PRINT "Das zweite Leerzeichen befindet sich an Position";
PRINT INSTR(position1+1, text, " ")
PRINT

' Verwendung von ANY
PRINT "Suche Zeichenkette 'ein' - Fundstelle:"; INSTR(text, "ein")
PRINT "Suche nach e, i oder n   - Fundstelle:"; INSTR(text, ANY "ein")
SLEEP
```

Ausgabe

```
Das erste  Leerzeichen befindet sich an Position 4
Das zweite Leerzeichen befindet sich an Position 8

Suche Zeichenkette 'ein' - Fundstelle: 9
Suche nach e, i oder n   - Fundstelle: 6
```

Das erste Leerzeichen findet sich beim vierten Zeichen des Strings. Danach sucht das Programm nach einem Leerzeichen ab der fünften Stelle. Hätten Sie ab der vierten Stelle gesucht, hätten Sie dasselbe Leerzeichen ein zweites Mal gefunden.

Bei der Suche nach dem String "ein" wird das Programm ab dem neunten Zeichen fündig, da hier die komplette Zeichenkette zum ersten Mal auftritt. Mit dem Schlüsselwort **ANY** erhalten Sie dagegen die Position des ersten auftretenden "i", denn "e" und "n" treten später auf.

INSTRREV() arbeitet ähnlich wie **INSTR()**, durchsucht den String jedoch von hinten (gibt also zurück, wann der Suchstring *das letzte Mal* auftritt). Die beiden Befehle unterscheiden sich außerdem in der Angabe des optionalen Parameters für die Startposition. Während dieser Parameter bei **INSTR()** vorn steht, wird er bei **INSTRREV()** nach hinten gestellt.

Hinweis:
Die Angabe eines optionalen Parameters am Beginn der Parameterliste ist untypisch für FreeBASIC. Genau genommen handelt es sich bei **INSTR()** nicht um eine Funktion mit optionalem Parameter, sondern um eine überladene Funktion mit zwei verschiedenen Parameterlisten. Das liegt an der Vorgabe von QuickBASIC, die in FreeBASIC übernommen wurde. **INSTRREV()** ist dagegen neu in FreeBASIC, und es verwendet das Standard-Verhalten von FreeBASIC.

Quelltext 15.12: Teilstring rückwärts suchen

```
DIM AS STRING text = "Ich bin ein Satz mit sechs Leerzeichen."
DIM AS INTEGER position1, position2

' letztes Leerzeichen suchen
position1 = INSTRREV(text, " ")
PRINT "Das letzte Leerzeichen befindet sich an Position"; position1
' vorletztes Leerzeichen suchen
PRINT "Das vorletzte Leerzeichen befindet sich an Position";
PRINT INSTRREV(text, " ", position1-1)
PRINT

' Verwendung von ANY
PRINT "Suche Zeichenkette 'ein' - Fundstelle:"; INSTRREV(text, "ein")
PRINT "Suche nach e, i oder n   - Fundstelle:"; INSTRREV(text, ANY "ein")
SLEEP
```

Ausgabe

```
Das letzte Leerzeichen befindet sich an Position 27
Das vorletzte Leerzeichen befindet sich an Position 21

Suche Zeichenkette 'ein' - Fundstelle: 9
Suche nach e, i oder n   - Fundstelle: 38
```

15.3. Umlaute und andere Sonderbuchstaben

15.3.1. Problemstellung: Darstellung von Sonderzeichen

Die Verwendung von Umlauten wirft leider einige Schwierigkeiten auf. Wie in Kapitel 13.4 bereits erwähnt, ist nur der ASCII-Zeichensatz genormt. Ob und an welcher Stelle

ein bestimmtes Sonderzeichen gespeichert ist, hängt von der jeweiligen Codepage ab. Leider verwenden die gängigen Editorn in der Regel eine andere Codepage als die Programmkonsole.

```
' Problemlose Umwandlung in Grossbuchstaben:
PRINT UCASE("unproblematische Umsetzung!")
' Umlaute bereiten Schwierigkeiten:
PRINT UCASE("ärgerliche Schwierigkeiten?")
SLEEP
```

Welche Ausgabe Sie beim zweiten Befehl erhalten, hängt von zwei Faktoren ab. Am wahrscheinlichsten lesen Sie etwas wie

Ausgabe

```
õRGERLICHE SCHWIERIGKEITEN?
```

Wurde das "ä" falsch umgewandelt? Nein, es wurde ohne Änderung übernommen, aber Sie sehen es in einer falschen Zeichencodierung. Unter Windows arbeitet die Konsole in der Zeichencodierung `IBM850` (sie unterstützt inzwischen aber auch Unicode), der Editor wird dagegen in aller Regel mit `ISO-8859-1` oder `ISO-8859-15` arbeiten (aber auch immer mehr moderne Editoren unterstützen Unicode).

15.3.2. Erster Lösungsversuch: Speicherformat anpassen

Da sich hier zwei unterschiedliche Formate in die Quere kommen, ist eine naheliegende Lösung, eines der beiden Formate anzupassen. Wenn Sie die Datei im Format `IBM850` speichern und kompilieren, erhalten Sie unter Windows zumindest folgende Ausgabe:

Ausgabe

```
äRGERLICHE SCHWIERIGKEITEN?
```

Das "ä" wurde schon einmal richtig erkannt, allerdings weiß FreeBASIC nicht, dass es sich um ein Zeichen handelt, für das auch ein Großbuchstabe existiert – geschweige denn, dass es den zugehörigen Großbuchstaben kennen würde. **`UCASE()`** und **`LCASE()`** schlägt daher für solche Zeichen fehl, Sie müssten sich dazu eine eigene Umwandlungsroutine schreiben.

Interessant wird es aber, wenn Sie den Quelltext in UTF-8 mit BOM speichern (die Byte Order Mark ist hier von entscheidender Bedeutung!) Dann nämlich wird die Zeichenkette nicht als **`STRING`**, sondern als **`WSTRING`** behandelt, und hier kennt der Compiler für Umlaute und Buchstaben mit Akzenten die korrekte Zuordnung zwischen

Groß- und Kleinbuchstaben. Dann erhalten Sie also tatsächlich die korrekte Ausgabe

Ausgabe

```
ÄRGERLICHE SCHWIERIGKEITEN?
```

Allerdings mit einem großen Aber. Eine Wertzuweisung in eine **STRING**-Variable macht das wieder zunichte. Zum Glück funktioniert jedoch eine Zuweisung in eine **WSTRING**-Variable:

```
' Das funktioniert nicht, wie es soll:
DIM AS STRING     test1 = "ärgerliche Schwierigkeiten?"
' Das funktioniert dagegen:
DIM AS WSTRING*28 test2 = "ärgerliche Schwierigkeiten?"

PRINT UCASE(test2)
SLEEP
```

Leider unterstützen die gängigen FreeBASIC-IDEs keine Speicherung im UTF-Format. Empfehlenswert ist in diesem Bereich der Editor Geany[28], der zwar keine spezielle FreeBASIC-Funktionionalität besitzt, aber zumindest ein Syntax-Highlighting für FreeBASIC sowie einige allgemein nützliche Programmierhilfen bereitstellt. Ein Problem bleibt damit aber immer noch: Wenn Sie Ihren Quelltext veröffentlichen, können Sie nicht sicher sein, dass ihn der Benutzer genau mit der Zeichencodierung speichert, die Sie gerne hätten.

15.3.3. Zweiter Lösungsversuch: Sonderzeichen als Konstanten

Wenn Sie sicher gehen wollen, dass Ihnen die Speichermethode nicht dazwischenfunkt, sollten Sie auf Sonderzeichen im Quelltext verzichten. Stattdessen kann jedes in der Codetabelle vorhandene Zeichen über seine Codenummer angesprochen werden. Um eine einheitliche Lösung für Windows und Linux zu bieten,[29] greifen wir etwas vor und nutzen für das folgende Beispiel das Grafikfenster, das Sie in **??** kennenlernen werden. **PRINT**-Ausgaben im Grafikfenster nutzen den FreeBASIC-internen Grafikfont und damit festdefiniert und unveränderlich die Codepage 437 (DOS-US, der Original-Zeichensatz des IBM-PC ab 1981).

[28] https://geany.org

[29] Selbst unter Windows ist nicht sichergestellt, dass der Benutzer die Standardkonsole verwendet und dass Codepage 850 eingestellt ist. Die Ausgabe von ANSI-Zeichen in der Konsole ist in jedem Fall mit dem Risiko einer falschen Darstellung verbunden.

```
SCREENRES 400, 300
PRINT "Sonderzeichen ohne Fehlerm" & CHR(148) & "glichkeit"
SLEEP
```

Ausgabe

```
Sonderzeichen ohne Fehlermöglichkeit
```

Das Beispiel funktioniert auf allen unterstützten Betriebssystemen und bei jeder Speichercodierung, denn der Quellcode nutzt nur ASCII-Zeichen. Natürlich gibt es sehr wohl eine Fehlermöglichkeit, und zwar, dass Sie sich bei der Codenummer `CHR(148)` vertun oder schlichtweg vertippen. Wenn Sie nicht alle wichtigen Codenummern auswendig im Kopf haben, ist es schwer, auf die Schnelle zu erkennen, ob der richtige Buchstabe verwendet wurde. Etwas leichter wird es mit Verwendung von Konstanten:

```
CONST UML_AE = CHR(132), UML_OE = CHR(148), UML_UE = CHR(129)

SCREENRES 400, 300
PRINT "So w" & UML_AE & "re es " & UML_UE & "berhaupt sch" & UML_OE & "ner."
SLEEP
```

Sicherlich auch etwas holprig zu lesen, aber einfacher als mit den direkten **CHR**-Befehlen. Einen weiteren Vorteil haben die Konstanten: Wenn sich die Werte einmal ändern, beispielsweise weil Sie eine Ausgabe in der Konsole wünschen, müssen Sie die Anpassung nur an einer Stelle vornehmen, anstatt den kompletten Quelltext nach Änderungen abzusuchen. Beachten Sie dabei die vorhin erwähnten Probleme der Plattformabhängigkeit bei der Konsolenausgabe: Mithilfe von Präprozessoren (siehe **??**) können Sie die Werte der Konstanten passend zum verwendeten Betriebssystem gestalten.

15.4. Escape-Sequenzen

Unter Escape-Sequenzen versteht man Zeichenkombinationen, die eine Sonderfunktion ausführen. In FreeBASIC werden sie eingesetzt, um in einem String auf spezielle ASCII-Zeichen wie den Zeilenumbruch oder die Tabulatortaste zuzugreifen.

Escape-Sequenzen werden in einem String nur dann interpretiert, wenn dem beginnenden Anführungszeichen des Strings ein Ausrufezeichen vorausgeht. Das Escape-Zeichen ist der Backslash \ (ASCII-Code 92). Tritt im String ein Backslash auf, so wird der folgende Ausdruck zuerst interpretiert. Die einfachste Form ist ein Backslash gefolgt von einer Zahl – in diesem Fall wird das Zeichen ausgegeben, dessen ASCII-Codenummer mit der Zahl übereinstimmt. Die Zahl kann dabei in jedem Zahlensystem angegeben

werden, wenn Sie dafür das zugehörige Präfix verwenden (vgl. Kapitel 14.2.1). Umlaute (im Grafikfenster) lassen sich damit auch folgendermaßen schreiben:

```
SCREENRES 400, 300
' mit ASCII-Codes im Dezimalsystem:
PRINT !"So w\132re es \129berhaupt sch\148ner."
' mit ASCII-Codes im Hexadezimalsystem:
PRINT !"So w\&h84re es \&h81berhaupt sch\&h94ner."
' mit ASCII-Codes im Oktalsystem:
PRINT !"So w\&o204re es \&o201berhaupt sch\&o224ner."
SLEEP
```

Durch das Ausrufezeichen wird die Interpretation der Escape-Sequenzen aktiviert, der Umlaut "ä" (ASCII-Code `132`, hexadezimal `&h84`, oktal `&o204`) kann nun mit Hilfe des Backslash und der Codenummer aufgerufen werden.

Einige besondere Steuerungszeichen besitzen eine spezielle Escape-Sequenz:

- `\r`: wie `CHR(13)` (carriage-return: Wagenrücklauf)
- `\n` und `\l`: wie `CHR(10)` (line-feed: Zeilenvorschub).
  `\r\n` ergibt ein CRLF, unter Windows die EDV-Version eines Zeilenumbruchs.
- `\a`: wie `CHR(7)` (Bell: Glocke)
- `\b`: wie `CHR(8)` (Backspace: Rücktaste)
- `\t`: wie `CHR(9)` (Tab: Tabulator)
- `\v`: wie `CHR(11)` (vtab: vertikaler Tabulator)
- `\f`: wie `CHR(12)` (formfeed: Seitenumbruch)
- `\"`: wie `CHR(34)` (doppeltes Anführungszeichen ")
- `\'`: wie `CHR(39)` (einfaches Anführungszeichen ')

Alle anderen Zeichen hinter dem Escape-Zeichen werden so interpretiert, wie sie im Ausdruck stehen. \\ wird also zu \.

```
PRINT !"C:\\\\Programme\\meinTest.exe\tTestprogramm \"Alpha\"\r\nViel Erfolg!"
```

Ausgabe

```
C:\\Programme\meinTest.exe      Testprogramm "Alpha"
Viel Erfolg!
```

Escape-Sequenzen funktionieren ausschließlich als Präprozessor, d. h. sie werden direkt vor dem Compilieren übersetzt. Das bedeutet konkret, dass die Markierung eines Escape-Strings durch das Ausrufezeichen ausschließlich vor dem beginnenden Anführungszeichen möglich ist. Eine Variable kann nicht nachträglich durch ein Ausrufezeichen zu einem Escape-String umgebaut werden.

```
DIM AS STRING test1 = !"Text mit\nZeilenumbruch"
PRINT test1                                          ' funktioniert

DIM AS STRING test2 = "Text mit\nZeilenumbruch"
PRINT !test2                                         ' erzeugt einen Compiler-Fehler
```

15.5. Fragen zum Kapitel

1. Schreiben Sie ein Programm, das den Benutzer auffordert, seinen Vor- und Nachnamen durch Leerzeichen getrennt einzugeben. Zerlegen Sie dann die Eingabe und speichern Sie Vor- und Nachname in zwei getrennten Variablen.

2. Schreiben Sie eine Funktion, die in einem übergebenen String nach einer bestimmten Zeichenkette sucht und sie durch eine andere ersetzt. Als Anregung könnten Sie alle auftretenden `"ä"` durch `"ae"` ersetzen.

3. Im BASIC-Dialekt *Omikron BASIC* gab es eine Funktion namens **`MIRROR$()`**, die einen String entgegennahm und den rückwärts geschriebenen String zurückgab. Auch wenn es für diese Funktion nicht allzu viele Einsatzszenarien gibt, stellt sie doch eine hervorragende Übungsaufgabe dar.

16. Datei-Zugriff

Ebenfalls ein wichtiger Punkt bei der Verarbeitung von Daten ist der Zugriff auf externe Ressourcen, insbesondere auf Dateien. Zunächst beschäftigen wir uns mit den klassischen Dateien, die auf einem Datenträger gespeichert werden; aber auch Hardwarezugriffe über COM-Port oder Drucker werden wie Dateizugriffe geregelt.

16.1. Datei öffnen und schließen

Beim Öffnen einer Datei gibt es bereits eine Menge Dinge, die Sie Ihrem Programm mitteilen müssen. Bevor wir die Informationen im Einzelnen besprechen, werfen wir erst einmal einen Blick auf die Syntax:

```
OPEN dateiname FOR dateimodus [ACCESS zugriffsart] [LOCK sperrart] _
     [ENCODING format] AS [#]dateinummer [LEN = recordlaenge]
```

Der Dateiname muss dabei einschließlich der Dateiendung angegeben werden[30] und darf auch eine Pfadangabe enthalten. Auf die Pfadangaben geht Kapitel 17.1.1 genauer ein.

Achtung:
Beachten Sie bei allen Datei- und Pfadnamen die korrekte Groß- und Kleinschreibung! Windows arbeitet *case insensitive*, d. h. dem System ist Groß- und Kleinschreibung egal. Anders ist das jedoch unter Linux, welches *case sensitive* arbeitet. Schon aus Gründen der Portabilität ist die korrekte Schreibweise dringend anzuraten – sonst stoßen Anwender auf einer anderen Plattform möglicherweise auf Probleme, die vom Entwickler nicht nachvollzogen werden können.

In QuickBASIC kam es zu einem Laufzeitfehler, wenn die Datei aus irgendwelchen Gründen nicht geöffnet werden konnte. Bei einem compilierten Programm wäre es aber sehr unvorteilhaft, wenn es wegen einer fehlenden Datei gleich zum Absturz käme.

[30] Die Dateiendungen werden unter Windows leider standardmäßig ausgeblendet. Nichtsdestotrotz müssen sie angegeben werden, um die korrekte Datei anzusprechen.

Daher wird ein möglicherweise auftretender Fehler ignoriert und die Datei einfach nicht geöffnet. Anschließende Lese- und Schreibzugriffe laufen dann ins Leere. Um dennoch einen missglückten Öffnen-Versuch abzufangen, kann **OPEN** auch als Funktion eingesetzt werden. Wenn die Datei ohne Fehler geöffnet werden konnte, liefert die Funktion den Wert `0`. Ansonsten entspricht der Rückgabewert der Fehlernummer.

In der Funktions-Variante des Befehls müssen die Parameter eingeklammert werden:

```
wert1 = OPEN(dateiname FOR dateimodus [ACCESS zugriffsart] [LOCK sperrart] _
           [ENCODING format] AS [#]dateinummer [LEN = recordlaenge])
```

Mögliche Fehlernummern sind:

- **1:** 'Illegal function call' (ungültiger Funktionsaufruf)
  Dies ist der Fall, wenn Sie eine Dateinummer auswählen, unter der bereits eine Datei geöffnet ist.
- **2:** File not found (Datei nicht gefunden)
  Die gewünschte Datei existiert nicht, oder Sie besitzen nicht die erforderlichen Rechte, um die Datei zu öffnen. Der Fehler kann auch auftreten, wenn z. B. mit `ENCODING "UTF-8"` eine Datei geöffnet werden soll, die nicht UTF-8-codiert ist oder bei der das Byte Order Mark nicht gesetzt ist.
  'File not found' ist in der Regel der am häufigsten auftretende Fehler.
- **3:** 'File I/O error' (Datei-Ein-/Ausgabe-Fehler)
  Es ist ein allgemeiner Lese- oder Schreibfehler aufgetreten, etwa weil kein Speicherplatz mehr zur Verfügung steht.

Hinweis:
Gerade im Umgang mit externen Ressourcen ist es sinnvoll, Fehler (wie in diesem Fall Lese- und Schreibfehler bei einer Datei) abzufangen, da Sie ja keine darüber hinausgehende Kontrolle haben, ob die Ressourcen wirklich existieren und zugreifbar sind. Wir werden in diesem Kapitel ein paar solcher Abfangroutinen einsetzen.

16.1.1. Dateinummer festlegen

Der Zugriff auf Dateien erfolgt über eine Dateinummer. Wenn Sie eine Datei öffnen, legen Sie eine Nummer im Bereich von 1 bis 255 fest, unter der die Datei in Zukunft

angesprochen werden soll. Das bedeutet zum einen, dass Sie später für die Dateizugriffe den Dateinamen nicht mehr zu kennen brauchen (es spielt auch keine Rolle mehr, ob es sich überhaupt um eine Datei handelt oder aber um einen Drucker oder eine Konsole), zum anderen können Sie aber auch nicht zwei Dateien zur gleichen Zeit unter derselben Nummer öffnen – das Programm wüsste dann ja nicht mehr, welche Datei denn nun gemeint ist.

Es ist also nötig, eine Dateinummer zu wählen, die im Augenblick noch nicht in Verwendung ist. Für kurze, überschaubare Programme ist das noch kein Problem. Spätestens aber, wenn Sie **OPEN** in einem Unterprogramm verwenden wollen, können Sie nicht sicher sein, ob die von Ihnen frei gewählte Dateinummer nicht bereits vom Hauptprogramm oder einem anderen Unterprogramm belegt wurde.

Glücklicherweise steht Ihnen mit **FREEFILE()** eine Funktion zur Verfügung, die Ihnen die nächste verfügbare Dateinummer mitteilt. Am besten gewöhnen Sie sich – auch bei kurzen Programmen – gleich an, die Dateinummer über **FREEFILE()** zu ermitteln.

Quelltext 16.1: Freie Dateinummer ermitteln

```
DIM AS INTEGER dateinummer = FREEFILE
IF OPEN("test.txt" FOR INPUT AS #dateinummer) THEN
  ' Datei konnte erfolgreich geoeffnet werden
END IF
```

Zum oben verwendeten Dateimodus **INPUT** kommen wir gleich. Zunächst noch kurz zum verwendeten Rautezeichen (Hash) #: Bei den Lese- und Schreibzugriffen ist es notwendig, dem Compiler mitzuteilen, dass der Zugriff nicht über Tastatur bzw. Ausgabefenster, sondern über die gewünschte Datei erfolgen soll. Um die Dateinummer von einer ganz gewöhnlichen auszugebenden Zahl zu unterscheiden, muss das Rautezeichen vorangestellt werden. Im **OPEN**-Befehl ist das Rautezeichen optional, wird aber empfohlen, um deutlich zu machen, dass es sich um eine Dateinummer handelt.

Achtung:
Wenn Sie mehrere Dateien öffnen wollen, müssen Sie die erste Datei öffnen, *bevor* Sie eine **FREEFILE**-Abfrage für die zweite Datei machen. **FREEFILE()** fragt immer nach der aktuell nächsten freien Dateinummer. Solange zwischenzeitlich keine Datei geöffnet wird, wird sich der Rückgabewert von **FREEFILE()** nicht ändern!

16.1.2. Der Dateimodus

FreeBASIC unterstützt fünf Dateimodi, die wir uns nun genauer ansehen werden. Bei den ersten drei handelt es sich um sequentielle Modi, die letzten beiden arbeiten mit einem wahlfreien Zugriff.

`INPUT` sequentieller Zugriff: Daten lesen

`OUTPUT` sequentieller Zugriff: Daten schreiben

`APPEND` sequentieller Zugriff: Daten anhängen

`RANDOM` Random-Access (lesen/schreiben)

`BINARY` wahlfreier Binärzugriff (lesen/schreiben)

Sequentieller Zugriff

Die „klassischen“ Dateien unter BASIC arbeiten mit einem sequentiellen Zugriff. Das bedeutet: Wenn eine Datei z. B. zum Lesen geöffnet wird, befindet sich ein Datenzeiger (Datei-Cursor) am Anfang dieser Datei. Beim Einlesen der Daten wandert dieser Zeiger weiter und markiert immer die Stelle, an welcher der nächste Lesezugriff erfolgen wird. Hat der Zeiger das Dateiende erreicht (**`EOF`**: End Of File), kann nicht weiter gelesen werden. Ein Lesezugriff erfolgt über den Dateimodus **`INPUT`**. In Quelltext 16.1 wurde eine Datei in diesem Lesemodus geöffnet.

Wenn Sie schreibend auf die Datei zugreifen wollen, verwenden Sie den Dateimodus **`OUTPUT`**. Auch hier wird der Datenzeiger an den Anfang der Datei gesetzt. Beim Schreiben rückt er weiter, sodass die geschriebenen Daten hintereinander in der Datei zu liegen kommen. Im Gegensatz zu **`INPUT`**, das sinnvollerweise nur aus bereits existierenden Dateien lesen kann, wird bei **`OUTPUT`** die Datei ggf. neu angelegt. Öffnen Sie allerdings eine bestehende Datei im Dateimodus **`OUTPUT`**, wird ihr bisheriger Inhalt gelöscht.

Achtung:
Wenn Sie eine Datei im Dateimodus **`OUTPUT`** öffnen, spielt es keine Rolle, ob Sie tatsächlich Daten schreiben oder nicht – bestehende Daten werden auf jeden Fall gelöscht. Wenn Sie eine Datei zum Schreiben öffnen, ohne Daten zu schreiben, haben Sie nach Programmende eine leere Datei vorliegen.

Als dritten sequentiellen Modus gibt es dann noch **`APPEND`**. Hierbei öffnen Sie die Datei ebenfalls zum Schreiben, der Dateizeiger wird jedoch *ans Ende* der Datei gesetzt. Wenn

die Datei zuvor nicht existiert hat, arbeitet **APPEND** genauso wie **OUTPUT**. Wenn Sie jedoch eine bereits bestehende Datei öffnen, wird ihr Inhalt nicht gelöscht, sondern die Ausgabe an die bisherigen Inhalte angehängt. **APPEND** eignet sich damit hervorragend dazu, eine fortlaufende Log-Datei zu schreiben.

Und wenn Sie aus einer Datei sowohl lesen als auch in sie hineinschreiben wollen? Für diese Vorgehensweise sind sequentielle Zugriffe nicht gedacht. Stattdessen können Sie die Datei zuerst lesend öffnen, alle Inhalte in den Speicher laden und die Datei wieder schließen. Anschließend öffnen Sie die Datei schreibend. Sie können stattdessen natürlich auch mit einer temporären Ausgabedatei arbeiten. Wenn Sie gleichzeitig an verschiedenen Stellen lesen und schreiben wollen, empfiehlt sich der Einsatz des wahlfreien Zugriffs.

Random-Access und BINARY

Ein sequentieller Zugriff ist sehr praktisch, wenn eine komplette Datei in den Speicher gelesen werden soll, nicht jedoch, wenn Sie in einer sehr großen Datei (etwa einer Datenbank) gezielt auf bestimmte Einträge zugreifen wollen (z. B. auf den Datensatz Nr. 2448). Für solche Anwendungsfälle wurde der *Random-Access-Zugriff* eingeführt, der aber in FreeBASIC vollständig durch den *wahlfreien Zugriff über* **BINARY** ersetzt werden kann.

Der Gedanke hinter der Zugriffsart **RANDOM** ist die Einteilung der Daten in gleich große Blöcke. Nehmen wir an, Sie wollen eine Datenbank anlegen, in der Vor- und Nachname, Wohnort und Telefonnummer Ihrer Kunden festgehalten werden. In der Regel werden Sie dazu ein UDT mit den entsprechenden Attributen anlegen. Dieses UDT besitzt eine bestimmte Größe, sagen wir 120 Byte. Diese Größe wird beim **OPEN**-Befehl als Record-Länge hinter dem Schlüsselwort **LEN** angegeben – **RANDOM** ist der einzige Dateimodus, in dem **LEN** sinnvoll eingesetzt werden kann.

Über *Random-Access* können Sie nun direkt den 2448. Eintrag auslesen oder aber den 731. Eintrag überschreiben. Das Programm springt dazu selbständig an die richtige Stelle ($(2448-1)\cdot 120$ bzw. $(731-1)\cdot 120$) und nimmt den Lese- bzw. Schreibzugriff vor. Gerade um Datensätze mitten in der Datei zu schreiben, ist *Random-Access* deutlich besser geeignet als der sequentielle Zugriff – bei letzterem müsste man sämtliche Daten einlesen und anschließend (mit Veränderung) neu schreiben.

Trotzdem gilt *Random-Access* inzwischen als veraltet und wird kaum noch eingesetzt. Grund dafür ist die mangelnde Flexibilität: Jeder Datensatz benötigt *genau* die gleiche Größe. Es ist damit auch nicht möglich, mehrere verschiedenartige Datensätze gemeinsam in einer Datei zu speichern. Als deutlich bessere Alternative dient der Dateimodus **BINARY**. Mit ihm lassen sich zum einen sämtliche **RANDOM**-Zugriffe komplett ersetzen, zum anderen ermöglicht er aber auch einen tatsächlich *wahlfreien* Zugriff auf jede beliebige

Stelle innerhalb der Datei.

Vergleich der Dateimodi

Bei der Entscheidung, welchen Dateimodus man benutzen will, gibt es an sich nur eine Frage: Wollen Sie einen sequentiellen Dateizugriff nutzen oder einen wahlfreien Zugriff? Wenn Sie sich für einen sequentiellen Zugriff entscheiden, ergibt sich der benötigte Dateimodus von selbst, je nachdem, ob Sie lesen, (neu) schreiben oder an bestehende Daten anhängen wollen. Bei einem wahlfreien Zugriff sollten Sie auf jeden Fall **BINARY** verwenden, denn **RANDOM** bietet keine Vorteile, die Sie mit **BINARY** nicht genauso nutzen können (außer vielleicht, dass man es „von früher so gewohnt“ ist).

Sequentielle Dateimodi bieten sich vor allem dann an, wenn Sie eine Datei zeilenweise einlesen oder schreiben wollen – also wenn die Daten durch Zeilenumbrüche getrennt sind und Sie im Vorfeld nicht wissen, wie lang die einzelnen Zeilen sein werden. Auch das Komma dient in der Regel als Trennzeichen zwischen den einzelnen Daten. Aber auch hier können die Daten unterschiedlich lang sein. Da Sie mit **BINARY** die exakte Stelle ansteuern können, von der aus Sie lesen bzw. schreiben wollen, müssen Sie dazu natürlich diese exakten Stellen kennen, was bei verschieden langen Zeilen nicht der Fall ist. Im Grunde genommen ist jede Art von textbasierter Datei (CSV-Dateien, FreeBASIC-Quellcodes ...) ein gutes Anwendungsziel für einen sequentiellen Dateimodus.

Für Binärdateien sind sequentielle Modi dagegen in der Regel ungeeignet. Ein Binärformat können Sie z. B. wählen, damit die Daten von außen nicht so ohne Weiteres sichtbar sind, oder aber, um die Informationen deutlich kompakter zu speichern. Und selbstverständlich liegen auch Bilder, Audiodateien usw. üblicherweise nicht als Textdatei vor.[31] Wenn Sie also z. B. die Abmessungen eines BMP-Bildes auslesen wollen, ist das Öffnen mittels **BINARY** die einzig sinnvolle Möglichkeit. Quelltext 16.7 wird ein solches Auslesen demonstrieren.

16.1.3. Zugriffsart beschränken

Während in den Dateimodi **INPUT**, **OUTPUT** und **APPEND** jeweils nur der Lese- bzw. nur der Schreibzugriff erlaubt ist, können mit **RANDOM** oder **BINARY** geöffnete Dateien sowohl ausgelesen als auch beschrieben werden. Manchmal ist es praktisch, die Zugriffsmöglichkeiten einzuschränken und beispielsweise nur das Lesen zu erlauben. Dazu dient das Schlüsselwort **ACCESS**.

[31] Bilder lassen sich durchaus auch im Textformat speichern – etwa beim Format *X PixMap* (XPM)

Quelltext 16.2: Dateizugriff auf das Lesen beschränken

```
DIM AS INTEGER dateinummer = FREEFILE
IF OPEN("test.txt" FOR BINARY ACCESS READ AS #dateinummer) THEN
  ' aus der Datei kann jetzt nur gelesen werden
END IF
```

Anstelle von **READ** ist auch **WRITE** (nur Schreibzugriff) und **READ WRITE** (Lese- und Schreibzugriff; Standard) möglich. Auswirkung hat diese Angabe nur in den Dateimodi **RANDOM** und **BINARY**.

16.1.4. Datei für andere Programme sperren

Die Syntax von **OPEN** sieht mit dem Schlüsselwort **LOCK** die Möglichkeit vor, eine Datei während des Öffnens für andere Programme zum Lesen und/oder Schreiben zu sperren. Diese Sperre funktioniert jedoch nicht wie vorgesehen, und es scheint auch auf Dauer so zu bleiben. Daher wird **LOCK** hier nur der Vollständigkeit halber erwähnt.

Hinweis für Experten:
Wenn Sie mehrere Programme haben, die auf dieselbe Datei zugreifen, können Sie eine zusätzliche Sperrdatei verwenden, um gleichzeitigen Zugriff zu vermeiden: Das Programm, welches auf die Datei zugreifen will, muss zuvor die Sperrdatei umbenennen; schlägt das Umbenennen fehl, dann hat bereits ein anderes Programm den Dateizugriff gestartet. Nach Beendigung des Zugriffs wird die Datei dann wieder zurückbenannt, und ein anderes Programm kann sich den Zugriff sichern. Ein Problem beim Einsatz von Sperrdateien ist, dass bei einem Absturz des sperrenden Programmes die angelegte Sperre dauerhaft erhalten bleibt und händisch behoben werden muss. Wenn der Benutzer (der ja nicht der Programmierer sein muss) mit der Sperrpraxis nicht vertraut ist, wird dies voraussichtlich für viel Frust sorgen.

16.1.5. Text-Encoding

Zur Unterstützung von Unicode kann beim Öffnen sequentieller Dateien angegeben werden, in welcher Codierung die Datei vorliegt. FreeBASIC speichert einen **WSTRING** je nach Plattform in UCS-2 bzw. UCS-4 (siehe Kapitel 6.3.1), und eine Übertragung zwischen Dateiformat und FreeBASIC-internem Format benötigt eine korrekte Umwandlung. Wenn Sie beim Öffnen die richtige Zeichenkodierung angeben, nimmt FreeBASIC die

Umwandlung automatisch vor. Da dieses Verfahren nur bei sequentiellem Zugriff sinnvoll ist, kann **ENCODING** im Dateimodus **RANDOM** und **BINARY** nicht eingesetzt werden.

Folgende UTF-Codierungen werden unterstützt:

- `"ASCII"`
- `"UTF-8"`
- `"UTF-16"`
- `"UTF-32"`

Damit die Datei erfolgreich geöffnet werden kann, muss das *Byte Order Mark (BOM)* gesetzt sein. Das BOM wird in UTF-16- und UTF-32-codierten Dateien benötigt, um die Byte-Reihenfolge (Big-Endian oder Little-Endian) zu kennzeichnen. In UTF-8 steht die Byte-Reihenfolge zwar fest, FreeBASIC braucht das BOM aber dennoch. Ist das BOM nicht enthalten, wird beim Versuch, die Datei im UTF-Format zu öffnen, der Laufzeitfehler 2 (File not found) zurückgegeben.

Unterschiede zu QuickBASIC:
Unicode wird in QuickBASIC nicht unterstützt. In der Dialektform `-lang qb` steht **ENCODING** nicht zur Verfügung.

16.1.6. Datei(en) schließen

Wenn alle Lese- oder Schreibvorgänge beendet sind, muss die Datei wieder geschlossen werden. Tatsächlich werden die Daten in der Regel erst dann physikalisch auf den Datenträger geschrieben, wenn die Datei geschlossen wird. Der dazugehörige Befehl lautet **CLOSE**, gefolgt von der Dateinummer oder den Dateinummern der gewünschten Dateien.

Quelltext 16.3: Dateien schließen

```
' Oeffne zwei Dateien: eine mit Lese-, eine mit Schreibzugriff
DIM AS INTEGER dateinummerEin, dateinummerAus
' erste freie Dateinummer ermitteln
dateinummerEin = FREEFILE
IF OPEN("eingabe.txt" FOR INPUT AS #dateinummerEin) <> 0 THEN
  ' Oeffnen fehlgeschlagen - Programm beenden
  PRINT "eingabe.txt konnte nicht geoeffnet werden."
  END
END IF
' naechste freie Dateinummer ermitteln (NACHDEM die erste verwendet wurde)
dateinummer2 = FREEFILE
IF OPEN("ausgabe.txt" FOR OUTPUT AS #dateinummer) <> 0 THEN
  ' Oeffnen fehlgeschlagen - Programm beenden
  PRINT "ausgabe.txt konnte nicht geoeffnet werden."
  ' Die Eingabedatei ist noch geoeffnet und wird nun geschlossen.
  ' Die Ausgabedatei wurde ja nicht erfolgreich geoeffnet.
  CLOSE #dateinummerEin
  END
END IF

' Dateien konnte erfolgreich geoeffnet werden. Es folgt das Hauptprogramm.
' Am Ende werden beide Dateien geschlossen:
CLOSE #dateiEin, #dateiAus
```

Achtung: Eine ggf. bereits existierende Datei `ausgabe.txt` wird durch dieses Programm überschrieben!

In der letzten Zeile sehen Sie, wie mehrere Dateien gleichzeitig geschlossen werden können. Das Rautezeichen vor den Dateinummern ist auch hier optional. Alternativ kann **CLOSE** auch ohne Parameter verwendet werden, um sämtliche noch offenen Dateien zu schließen. Dazu müssen Sie aber sicher sein, dass keine von anderen Programmteilen geöffnete Dateien mehr existieren, die noch benötigt werden.

Nach dem Schließen einer Datei ist die von ihr verwendete Dateinummer wieder frei und kann erneut eingesetzt werden. Schon aus diesem Grund empfiehlt es sich, Dateien nicht länger als notwendig geöffnet zu halten.

Beim Programmende werden alle noch geöffneten Dateien automatisch geschlossen. Trotzdem sollten Sie Dateien immer explizit schließen. Das gehört zum guten Programmierstil und verhindert, dass Sie im entscheidenen Moment ein notwendiges **CLOSE** vergessen. Wenn Sie innerhalb einer häufig aufgerufenen Prozedur Dateien öffnen und nicht wieder schließen, werden Sie bald an die Grenze der maximal 255 gleichzeitig geöffneten Dateien stoßen, und ein weiteres Öffnen wird scheitern. Wenn Sie sich von vorn herein ein explizites **CLOSE** angewöhnen, können Sie solche Fehler vermeiden.

16.2. Lesen und Schreiben sequentieller Daten

Lesen und Schreiben sequentieller Daten funktioniert nahezu genauso wie beim Lesen von der Tastatur bzw. das Schreiben in das Ausgabefenster. Der einzige Unterschied ist, dass die Dateinummer mit angegeben werden muss.

16.2.1. PRINT# und WRITE#

Mit **PRINT#** können Dateien sequentiell beschrieben werden. Der Befehl folgt dabei denselben Regeln wie **PRINT** (siehe Kapitel 4.1), einschließlich der Komma- und Strichpunkt-Regeln. Auch **TAB** wird unterstützt. Die Ausgabe in der Datei entspricht genau dem, was Sie sonst im Programmfenster erwarten würden.

PRINT# eignet sich vor allem dann, wenn das Lesen der gespeicherten Daten über den Texteditor angenehm gestaltet werden soll, z. B. wenn Sie eine schön formatierte Ausgabe erreichen wollen (etwa eine tabellarische Darstellung) oder wenn ein spezielles textbasiertes Format, etwa HTML, genutzt werden soll. Wenn Ihr vorrangiges Ziel dagegen ist, die geschriebenen Daten mit geringem Aufwand wieder einzulesen, ist **WRITE#** möglicherweise die bessere Wahl. Hier werden die Parameter durch Kommata getrennt (ein Semikolon ist nicht zulässig), und diese Kommata werden auch in die Datei geschrieben. Außerdem werden Strings mit Anführungszeichen umgeben.

Quelltext 16.4: Vergleich zwischen PRINT# und WRITE#

```
DIM AS INTEGER dateinummer = FREEFILE
DIM AS STRING txt = "b"
OPEN "ausgabe.txt" FOR OUTPUT AS #dateinummer
PRINT #dateinummer, 1; 2, "a"; txt
WRITE #dateinummer, 1, 2, "a", txt
CLOSE #dateinummer
END
```

Für beide Befehle ist das Rautezeichen notwendig, da sonst nicht zwischen der Datei-Eingabe und der Tastatur-Eingabe unterschieden werden könnte. Aus diesem Grund werden wir bei der Datei-Ausgabe auch von **PRINT#** und **WRITE#** sprechen, auch wenn der Befehl tatsächlich nur **PRINT** bzw. **WRITE** lautet. In der Datei `ausgabe.txt` steht nun folgendes:

Ausgabe

```
 1 2          ab
1,2,"a","b"
```

Beachten Sie bei der Dateiausgabe in der ersten Zeile die Auswirkung des Strichpunktes

(direktes Aneinanderhängen) und des Kommas (Einrückung). Bei **WRITE#** werden die Daten, durch Komma getrennt, ohne Einrückungen direkt aneinandergehängt. **WRITE#** eignet sich daher nicht dazu, eine schön formatierte Ausgabe zu erhalten, sondern hat einen rein praktischen Nutzen. Die Daten können nämlich mit **INPUT#** problemlos wieder einzeln eingelesen werden.

Hinweis:
WRITE existiert auch als normaler Schreibbefehl im Programmfenster, hat dort aber keinen großen Nutzen.

16.2.2. INPUT#, LINE INPUT#

Auch **INPUT#** und **LINE INPUT#** funktionieren grundsätzlich genauso wie die in Kapitel 5.1 behandelten Befehle zur Tastatureingabe. Wenn Sie die in Quelltext 16.4 gespeicherten Daten wieder einlesen wollen, können Sie folgendermaßen vorgehen:

Quelltext 16.5: INPUT-Methode bei Dateien

```
DIM AS INTEGER dateinummer = FREEFILE
DIM AS INTEGER wert1, wert2
DIM AS STRING zeile1, wert3, wert4
OPEN "ausgabe.txt" FOR INPUT AS #dateinummer
LINE INPUT #dateinummer, zeile1                ' gesamte Zeile einlesen
INPUT #dateinummer, wert1, wert2, wert3, wert4  ' Einzelwerte einlesen
CLOSE #dateinummer
PRINT "erste Zeile: """; zeile1; """"
PRINT "wert1 = "; wert1
PRINT "wert2 = "; wert2
PRINT "wert3 = """; wert3; """"
PRINT "wert4 = """; wert4; """"
SLEEP
```

Ausgabe

```
erste Zeile: " 1 2          ab"
wert1 =  1
wert2 =  2
wert3 = "a"
wert4 = "b"
```

INPUT# folgt denselben Begrenzungsregeln durch Kommata. Wenn Sie also eine Zeile

einlesen wollen, die ein Komma beinhaltet, wird der Lesevorgang vor dem Komma beendet.[32] Das kann so gewollt sein; wenn Sie aber Textdateien zeilenweise einlesen wollen, weichen Sie besser auf **LINE INPUT#** aus.

16.2.3. INPUT() und WINPUT() für Dateien

Auch die Funktionsversion **INPUT()** zum Auslesen einer festgelegten Zeichenzahl kann auf eine zum Lesen geöffnete Datei angewendet werden. Dazu muss allerdings die Dateinummer als zweiter Parameter angegeben werden (anstatt als erster), und es darf ihr auch kein Hash vorangestellt werden. Wenn Sie dagegen aus einer Unicode-codierten Datei lesen wollen, verwenden Sie stattdessen **WINPUT()**.

Quelltext 16.6: INPUT() bei Dateien

```
DIM AS INTEGER dateinummer = FREEFILE
DIM AS STRING einlesen
OPEN "ausgabe.txt" FOR INPUT ENCODING "UTF-8" AS #dateinummer
PRINT "Die ersten fuenf Zeichen der Datei lauten:"
einlesen = WINPUT(5, dateinummer)
PRINT """ & einlesen & """
CLOSE #dateinummer
SLEEP
```

Denken Sie daran, dass die Datei nur dann korrekt mit `ENCODING "UTF-8"` (oder anderen Unicode-Arten) geöffnet werden kann, wenn in der Datei das BOM gesetzt ist.

Falls Sie sich übrigens fragen, was in der sechsten Zeile die vier aufeinanderfolgenden Anführungszeichen bedeuten sollen: Auf diese Art und Weise können Sie ein Anführungszeichen auf der Konsole ausgeben. Das vorderste und hinterste Anführungszeichen der Vierergruppe begrenzt den String, und die beiden mittleren werden zusammen als ein Anführungszeichen interpretiert.

16.3. Zugriff auf binäre Daten

Binäre Dateien sind auf der einen Seite deutlich flexibler und kompakter, auf der anderen Seite erfordern sie aber eine deutlich bessere Planung. Im Gegensatz zum sequentiellen Zugriff lesen und speichern Sie hier keine Datenzeilen, sondern direkt einzelne Datentypen. Daher ist es notwendig, zu wissen, welche Daten sich wo befinden.

[32] Das gilt natürlich nicht, wenn das Komma korrekt von Anführungszeichen umschlossen wird. Tatsächlich ist das Verhalten bei Dateien ohne korrekter Formatierung etwas komplizierter, aber das lernen Sie am besten durch Ausprobieren.

16.3.1. Binäre Daten speichern

Zum Speichern binärer Daten verwenden wir den Befehl **PUT#**:

```
PUT #dateinummer, [position], daten [, menge]
```

- `dateinummer` ist wieder die Nummer, die der Datei mit **OPEN** zugewiesen wurde.
- `position` gibt die Stelle an, an die geschrieben werden soll. Der Parameter kann ausgelassen werden. Dann schreibt der Befehl an der Stelle, an der sich der Dateizeiger gerade befindet. Direkt nach dem Öffnen einer Datei ist das die Position `1`.
- `daten` enthält den zu schreibenden Wert. In der Regel wird es sich dabei um eine Variable oder einen Pointer handeln, denn hier ist es wichtig, dass der Datentyp klar erkennbar ist. **PUT#** benötigt eine exakte Angabe über die Länge der geschriebenen Daten, also die Größe des Datentyps.
  Auch Strings sind möglich. Die Datenlänge berechnet sich dann über die Länge des Strings. Zahlenwerte oder mathematische Berechnungen funktionieren dagegen nicht.
- Handelt es sich bei `daten` um einen Pointer, dann kann mit `menge` angegeben werden, wie viele Elemente des entsprechenden Datentyps geschrieben werden sollen. Sie können damit in einem **ALLOCATE**-Puffer z. B. zehn **INTEGER** hintereinander ablegen und diese zehn Werte dann in einem Rutsch in die Datei schreiben. Ohne Angabe von `menge` wird immer ein einzelner Wert geschrieben.

  Wenn Sie ein Array schreiben, gibt `menge` an, wie viele Einträge geschrieben werden sollen.

PUT# existiert auch als Funktion. Dann muss die Parameterliste mit Klammern umgeben werden; der Rückgabewert ist eine FreeBASIC-Fehlernummer (oder 0, wenn kein Fehler aufgetreten ist).

Die folgenden Beispiele dienen nur zur Demonstration der Syntax; der Code ist in dieser Form nicht lauffähig. Insbesondere kann der Pointer-Zugriff nicht ohne Reservierung des Speicherplatzes stattfinden.

```
DIM AS STRING   varlength = "123"
DIM AS STRING*5 fixlength = "456"
DIM AS SHORT    shortVal, array(10)
DIM AS LONG PTR longPtr
PUT #dateinummer,, varlength            ' schreibt 3 Byte (aktuelle Stringlaenge)
PUT #dateinummer,, fixlength            ' schreibt 5 Byte (feste Stringlaenge)
PUT #dateinummer, 8, shortVal           ' schreibt 2 Byte ab Position 8
PUT #dateinummer,, CAST(LONG, shortVal) ' schreibt 4 Byte (Laenge eines LONG)
PUT #dateinummer,, shortVal+1           ' schreibt ein INTEGER!!!
PUT #dateinummer,, 123                  ' funktioniert nicht (Datentyp unklar)
PUT #dateinummer,, CAST(LONG, 123)      ' funktioniert auch nicht (trotz CAST)
PUT #dateinummer,, *longPtr             ' schreibt 4 Byte (Laenge von LONG)
PUT #dateinummer,, *longPtr, 5          ' schreibt 20 Byte (5 LONG-Werte)
PUT #dateinummer,, array(3), 4          ' schreibt die Werte array(3) bis array(6)
```

Nicht alle Speichergrößen sind intuitiv. Verwenden Sie auf jeden Fall keine Rechenausdrücke, oder setzen Sie sie in ein **CAST**, um den Datentyp sicherzustellen. Auch bei Strings variabler Länge müssen Sie aufpassen, damit Sie beim Einlesen wieder die richtige Länge verwenden.

Besonders praktisch ist, dass Sie mit **PUT#** auch komplette UDTs schreiben können. Es gibt dabei jedoch eine Einschränkung: Strings variabler Länge funktionieren genauso wenig wie Pointer-Inhalte. Der Grund dafür ist ganz einfach: Ein Pointer selbst ist lediglich ein **INTEGER**-Wert mit der Adresse auf den Inhalt. Diesen zu speichern bringt nichts, da der tatsächliche Inhalt verloren geht. Bei Strings variabler Länge ist es dasselbe, denn auch dieser verweist nur auf den tatsächlichen String-Inhalt. Wenn Sie im UDT aber nur Zahlenwerte und Strings fester Länge verwenden, ist ein Speichern und Laden des gesamten UDTs problemlos möglich. Ein Beispiel dafür finden Sie in Quelltext 16.8.

Hinweis:
Im Grafikmodus gibt es ebenfalls einen Befehl **PUT** (und **GET**), der in diesem Fall Bilddaten auf den Bildschirm schreibt (bzw. davon liest). Siehe dazu **??**

16.3.2. Binäre Daten einlesen

Der Lesebefehl **GET#** ist bis auf einen weiteren optionalen Parameter genauso aufgebaut wie **PUT#** und kann ebenfalls als Funktion eingesetzt werden.

```
GET #dateinummer, [position], daten [, menge[, gelesen]]
```

`gelesen` ist eine Variable, in der gespeichert wird, wie viele Byte tatsächlich eingelesen

wurden. Damit können Sie überprüfen, ob alle Daten korrekt gelesen wurde oder ob versucht wurde, hinter dem Dateiende zu lesen.

Als Beispiel wollen wir die Abmessungen eines BMP-Bildes auslesen. Wie viele andere Dateitypen besitzt BMP einen Header, in dem u. a. Informationen zu Bildabmessung, Auflösung, Farbtiefe, Komprimierung usw. gespeichert sind. Diese Daten besitzen eine feste Position innerhalb der Datei und können daher gezielt angesteuert werden.

Quelltext 16.7: Bildabmessungen eines BMP auslesen

```
DIM AS LONG breit, hoch, dateinummer = FREEFILE
OPEN "testbild.bmp" FOR BINARY ACCESS READ AS #dateinummer
GET #dateinummer, 19, breit                  ' Breite aus der BMP-Datei auslesen
GET #dateinummer, 23, hoch                   ' Hoehe aus der BMP-Datei auslesen
CLOSE #dateinummer                           ' schliesse die Datei
PRINT "Das Bild ist"; breit; "px breit und"; ABS(hoch); "px hoch."
SLEEP
```

Hinweis: Die Höhe des Bildes kann (selten) negativ angegeben sein, um damit eine Datenspeicherung von oben nach unten (statt von unten nach oben) zu kennzeichnen.

Die Verwendung des richtigen Datentyps ist von entscheidender Bedeutung. In Quelltext 16.7 beispielsweise muss ein **LONG** (bzw. ein **INTEGER<32>**) verwendet werden; ein **INTEGER** würde in einem 64-Bit-System zu falschen Werten führen, da es sich dort um ein **INTEGER<64>** und damit um einen zu großen Datentyp handelt.

Achtung:
Wenn Ihr Programm sowohl unter 32-Bit- als auch 64-Bit-Rechnern verwendet werden kann, sollten Sie beim Dateizugriff den Datentyp **INTEGER** unbedingt vermeiden und stattdessen **LONG**, **INTEGER<32>**, **INTEGER<64>** o. ä. verwenden.

Sehen wir uns zum Thema Datentyp-Größen noch folgendes Beispiel an:

```
DIM AS USHORT shortVarEin = 6440, shortVarAus
DIM AS UBYTE  byteVarEin  =   86, byteVarAus
DIM AS INTEGER dateinummer = FREEFILE
OPEN "test.dat" FOR BINARY AS #dateinummer
' in der Reihenfolge USHORT - UBYTE schreiben
PUT #dateinummer, 1, shortVarEin
PUT #dateinummer,,   byteVarEin
' in der Reihenfolge UBYTE - USHORT lesen
GET #dateinummer, 1, byteVarAus
GET #dateinummer,,   shortVarAus
' und auf dem Bildschirm ausgeben
PRINT shortVarAus, byteVarAus
SLEEP
```

Ausgabe

```
22034          52
```

Was genau schiefgelaufen ist, sieht man, wenn man sich die Hexadezimalwerte ansieht:

```
PRINT HEX(4660),  HEX(86)     ' Werte der Eingabe
PRINT HEX(22034), HEX(52)     ' Werte der Ausgabe
SLEEP
```

Ausgabe

```
1234           56
5612           34
```

Gespeichert wurde erst das **USHORT** `&h1234` (zwei Byte) – und zwar, da es sich um ein Little-Endian-System[33] handelt, in der Reihenfolge `&h34 &h12` – und dann das **UBYTE** `&h56`. In der Datei steht nun die Bytefolge `&h341256`. Beim Auslesen wird erst ein **UBYTE** extrahiert, nämlich `&h34`, und als zweites ein **USHORT** mit der Byte-Reihenfolge `&h12 &h56`. Das **USHORT** wird, wieder wegen des Little-Endian-Systems, als `&h5612` zusammengesetzt.
Hätten Sie den Inhalt der Datei in einen ausreichend langen **STRING** gelesen, wären die Byte-Werte als ASCII-Codes interpretiert worden (in diesem Fall als `CHR(&h34, &h12, &h56)`).

Auch die Verwendung eigener UDTs soll kurz demonstriert werden. Wir verwenden ein UDT mit Unter-UDT, denn auch das stellt keine Schwierigkeit dar, solange in keiner der Ebenen Strings variaber Länge oder Pointer vorkommen.

[33] Big-Endian und Little-Endian stehen für die Byte-Reihenfolge, in der einfache Zahlenwerte in den Speicher gelegt werden. Bei Little-Endian wird das kleinstwertige Byte an der Anfangsadresse gespeichert, danach das zweitkleinste usw.

Quelltext 16.8: UDT speichern und auslesen

```
TYPE TypKoordinaten
  AS INTEGER x, y
END TYPE
TYPE TypTest
  AS UBYTE          id
  AS TypKoordinaten position
END TYPE

DIM AS TypTest testfeld(1 TO 4), testwert
DIM AS INTEGER dateinummer = FREEFILE

' testfeld() mit Daten befuellen
FOR i AS INTEGER = 1 TO 4
  testfeld(i).id = i
  testfeld(i).position.x = 10
  testfeld(i).position.y = 5 + i*2
NEXT

' testfeld() speichern
OPEN "test.dat" FOR BINARY AS #dateinummer
PUT #dateinummer,, testfeld(1), 4

' dritten Datensatz auslesen und ausgeben
GET #dateinummer, 1 + 2*SIZEOF(TypTest), testwert
PRINT testwert.id, testwert.position.x, testwert.position.y
CLOSE #dateinummer
SLEEP
```

Ausgabe

```
 3              10             11
```

Sie sehen, dass Sie in eine mit **BINARY** geöffnete Datei sowohl schreiben als auch aus ihr lesen können, was insbesondere die Verwendung mit Datensätzen sehr bequem macht. Noch eine kurze Erläuterung zur **GET#**-Zeile: Der erste Datensatz startet an Position 1 und benötigt `SIZEOF(TypTest)` Byte an Speicherplatz. Dementsprechend startet der zweite Datensatz an Position `1+SIZEOF(TypTest)` und der dritte bei `1+2*SIZEOF(TypTest)`. Zur Erinnerung: **SIZEOF()** gibt die Größe einer Variablen oder eines Datentyps an.

16.3.3. Binäre Speicherung von Strings variabler Länge

Um eine Zeichenkette korrekt aus der Datei auszulesen, müssen Sie ihre Länge kennen. Und ebenso wichtig: *Das Programm* muss wissen, wie lang der auszulesende String sein

soll. Das können Sie auf zwei Wegen erreichen:

- durch die Verwendung von Strings fester Länge: Die Lesemenge ist durch die Größe des Datentyps festgelegt.
- bei Strings variabler Länge: Auch hier wird die Lesemenge durch die Länge des Strings bestimmt. Vor dem Lesevorgang müssen Sie daher den String auf die richtige Länge bringen.

Die Länge des Strings wird nicht automatisch mitgespeichert. Bei variabler Länge sind Sie daher selbst dafür verantwortlich, die Länge zu speichern. Ich empfehle dazu, erst die Länge als Zahlenwert zu speichern und dann den String dahinter abzulegen. Beim Lesevorgang lesen Sie dann wieder erst die Länge aus, präparieren dann die Stringvariable so, dass sie die richtige Länge besitzt, und lesen den String ein.

Quelltext 16.9: Strings variabler Länge speichern und laden

```
DIM AS STRING speicherstring = "Hallo Welt!", ladestring
DIM AS LONG speicherlaenge, ladelaenge
DIM AS INTEGER dateinummer = FREEFILE
OPEN "test.dat" FOR BINARY AS #dateinummer
' Laenge und Stringinhalt speichern
speicherlaenge = LEN(speicherstring)
PUT #dateinummer,, speicherlaenge
PUT #dateinummer,, speicherstring
' Laenge und Stringinhalt lesen
GET #dateinummer, 1, ladelaenge       ' Laenge lesen
ladestring = SPACE(ladelaenge)        ' ladestring auf die richtige Laenge bringen
GET #dateinummer,, ladestring         ' String lesen
PRINT ladestring                      ' Testausgabe
SLEEP
```

In Zeile 11 wird in `ladestring` eine Zeichenkette aus der benötigten Anzahl an Leerzeichen erzeugt. Das ist sehr effektiv, aber lediglich eine von mehreren Möglichkeiten. Wichtig ist nur, dass `ladestring` vor dem Lesevorgang die richtige Länge hat.

16.3.4. Position des Dateizeigers

Die Position des Dateizeigers können Sie jederzeit mit **SEEK** setzen oder lesen. Im ersten Fall wird **SEEK** als Prozedur eingesetzt, im zweiten Fall **SEEK()** als Funktion.

Quelltext 16.10: Dateizeiger setzen und lesen

```
DIM AS INTEGER dateinummer = FREEFILE
OPEN "test.dat" FOR BINARY AS #dateinummer
' Zeiger setzen
SEEK #dateinummer, 100
' Beim Schreiben wird die Position veraendert.
PUT #dateinummer,, "Testeingabe"
' neue Position lesen und ausgeben
PRINT "Zeigerposition:"; SEEK(dateinummer)
CLOSE #dateinummer
SLEEP
```

Ausgabe

```
Zeigerposition: 111
```

Das erste Setzen des Zeigers hätte natürlich auch über den zweiten Parameter von **PUT** erfolgen können. Oft ist es programmiertechnisch jedoch sinnvoller, zunächst die gewünschte Ausgangsposition herzustellen und anschließend die Schreibvorgänge durchzuführen.

Eine weitere Funktion zum Ermitteln (jedoch nicht zum Setzen) des Dateizeigers ist **LOC()**. Diese gibt nicht die aktuelle Position an, sondern das zuletzt gelesene Element. Bei Dateien, die mit **BINARY** geöffnet wurden, ist daher der Rückgabewert von **LOC()** immer um genau 1 kleiner als **SEEK()**. Bei mit **RANDOM** geöffneten Dateien wird der zuletzt gelesene *Datensatz* zurückgegeben, und bei sequentiellen Dateien nimmt der Compiler eine Datensatzlänge von 128 Byte an (`LOC(dateinummer) = (SEEK(dateinummer)-1)\128.`) Allein schon wegen dieser eher unhandlichen Berechnungsformel sollten Sie auf **LOC()** verzichten und stattdessen **SEEK()** verwenden. **LOC()** dient lediglich der Rückwärtskompatibilität zu QuickBASIC.

Dem aufmerksamen Leser sollte nicht entgangen sein, dass vor dem Parameter `dateinummer` in der Funktion **SEEK()** kein Rautenzeichen steht. Aufgrund der Compiler-Syntax können innerhalb von Funktionen keine Parameter mit Rautezeichen eingesetzt werden. Dies betrifft auch **LOC()** und die im Folgenden aufgeführten Funktionen **EOF()** und **LOF()**, aber auch selbst definierten Unterprogramme. **OPEN**, **CLOSE**, **GET#**, **INPUT#** usw. werden beim Compiliervorgang speziell behandelt.

16.4. Dateiende ermitteln

Insbesondere dann, wenn Sie eine komplette Datei auslesen wollen, brauchen Sie eine Möglichkeit, das Ende der Datei zu ermitteln. Dazu stehen Ihnen zwei Alternativen zur Verfügung.

16.4.1. End Of File (EOF)

Die klassische Variante für sequentielle Daten ist **EOF()**. Diese Funktion gibt zurück, ob das Dateiende erreicht wurde (`-1`) oder nicht (`0`). Auch wenn der Dateizeiger mit **SEEK** hinter das Dateiende gesetzt wurde, liefert **EOF()** `-1` zurück.

Besonders interessant ist **EOF()** im Zusammenhang mit dem Dateimodus **INPUT**. Bei den Dateimodi **OUTPUT** und **APPEND** gibt **EOF()** immer `-1` zurück, unabhängig von der tatsächlichen Position des Zeigers.

Quelltext 16.11 demonstriert das Einlesen einer kompletten sequentiellen Datei. Wenn Sie den Quellcode unter dem Namen `test.bas` speichern, können Sie damit genau diesen Quellcode auf dem Bildschirm ausgeben.

Quelltext 16.11: Sequentielle Datei vollständig einlesen

```
DIM AS STRING zeile
DIM AS INTEGER dateinummer = FREEFILE
OPEN "test.bas" FOR INPUT AS #dateinummer

DO UNTIL EOF(dateinummer)           ' solange das Dateiende nicht erreicht wurde
  LINE INPUT #dateinummer, zeile    ' eine Zeile einlesen ...
  PRINT zeile                       ' ... und auf dem Bildschirm ausgeben
LOOP
CLOSE #dateinummer
SLEEP
```

16.4.2. Length Of File (LOF)

EOF() kann zwar auch bei mit **BINARY** geöffneten Dateien eingesetzt werden, allerdings bietet sich dort **LOF()** in der Regel eher an. **LOF()** gibt die Länge der geöffneten Datei in Byte an, und vor dem Zugriff auf die gewünschten Daten kann dann überprüft werden, ob der benötigte Datenbereich überhaupt innerhalb der Datei liegt.

Quelltext 16.12: Datensatz aus einer binären Datei einlesen

```
TYPE TypDatensatz
  AS LONG      id
  AS STRING*10 inhalt
END TYPE
DIM AS TypDatensatz datensatz
DIM AS INTEGER p = 5

DIM AS INTEGER dateinummer = FREEFILE
OPEN "test.dat" FOR BINARY AS #dateinummer
' versuche, Datensatz an Position p zu lesen
IF LOF(dateinummer) >= p*SIZEOF(TypDatensatz) THEN
  ' Die Datei ist lang genug, um den Datensatz 'p' vollstaendig zu enthalten
  GET #dateinummer, 1+(p-1)*SIZEOF(TypDatensatz), datensatz
ELSE
  ' Die Datei ist zu kurz - Datensatz 'p' ist nicht enthalten
  PRINT "Fehler: Datensatz konnte nicht gefunden werden."
END IF
CLOSE #dateinummer
SLEEP
```

Zur Erläuterung: Wie schon in Quelltext 16.8 beginnt Datensatz 1 an der Stelle `1` und ist `SIZEOF(TypDatensatz)` lang (hier 16 Byte; siehe auch Kapitel 7.3). Der fünfte Datensatz beginnt dementsprechend an der Stelle `1+(5-1)*SIZEOF(TypDatensatz)`. Ist die Datei kürzer als `5*SIZEOF(TypDatensatz)`, dann ist der fünfte Datensatz nicht vorhanden oder unvollständig. Anstatt zu versuchen, den Datensatz einzulesen – der Lesevorgang schlägt dann fehl und `datensatz` wird auf den Initialwert gesetzt – fängt das Programm den Fehler mit einer Meldung ab.

16.5. Datei löschen

Zum Entfernen einer Datei aus dem Dateisystem dient der Befehl **`KILL`**. Dem Befehl wird eine Zeichenkette übergeben, die den Namen der zu löschenden Datei enthält – einschließlich Dateiendung. So wie **`OPEN`** kann auch **`KILL`** als Funktion eingesetzt werden und gibt dann im Erfolgsfall `0` und ansonsten eine Fehlernummer zurück.

Achtung: Mit **`KILL`** gelöschte Dateien werden nicht in den Papierkorb geschoben, sondern wirklich gelöscht!

Quelltext 16.13: Datei löschen

```
SELECT CASE KILL("LoeschMich.txt")
  CASE 0 : PRINT "Die Datei wurde geloescht."
  CASE 2 : PRINT "Fehler: Die Datei existiert nicht!"
  CASE 3 : PRINT "Fehler: Unzureichende Zugriffsrechte oder Ordnerzugriff!"
END SELECT
SLEEP
```

16.6. Standard-Datenströme

Auch die Standard-Datenströme werden wie Dateien behandelt. Sie sind ein unter Unix und Linux allgemein bekanntes Konzept, welches den meisten Windows-Nutzern jedoch eher unbekannt sein dürfte – daher im Vorfeld ein paar kurze Erläuterungen.

16.6.1. Eine kurze Einführung

Insgesamt gibt es drei Standard-Datenströme:

- die Standardeingabe (*stdin*): Über sie werden Daten in das Programm eingelesen. Sie ist normalerweise mit der Tastatur verbunden, die Benutzereingaben finden also über die Tastatur statt. *stdin* hat den Datei-Deskriptor `0`. (Zur Frage, wozu der Datei-Deskriptor benötigt wird, kommen wir gleich.)
- die Standardausgabe (*stdout*): Über sie gibt das Programm Daten aus. Die Ausgabe erfolgt normalerweise über die Programmkonsole auf dem Bildschirm. (Wenn Sie ein Grafikfenster nutzen, wird dieses als *stdout* festgelegt.) *stdout* hat den Datei-Deskriptor `1`.
- die Standardfehlerausgabe (*stderr*): Über sie werden Status- und Fehlermeldungen ausgegeben. Normalerweise erfolgt auch sie über die Programmkonsole (bzw. über das Grafikfenter). *stderr* hat den Datei-Deskriptor `2`.

Interessant sind die Standard-Datenströme, weil sie auf ein anderes Ein- bzw. Ausgabemedium umgeleitet werden können. Sie können Daten z. B. statt von der Tastatur auch aus einer Datei einspeisen oder umgekehrt die Ausgabe in eine Datei umleiten. Nehmen wir an, Sie haben ein Programm namens `test.exe`, das von *stdin* liest, nach *stdout* ausgibt und Fehlermeldungen nach *stderr* schreibt. Sie können dieses Programm nun über die Konsole starten:

```
test.exe
```

Die Eingabe erfolgt, wie gewohnt, über die Tastatur, und sowohl die Ausgabe als auch die Fehlermeldungen erscheinen im Konsolenfester. Um die Ausgabe in eine Datei umzuleiten, können Sie stattdessen folgendermaßen starten:

```
test.exe > ausgabe.txt
test.exe >> ausgabe.txt
```

Die Spitzklammer legt die Ausgabedatei fest. Fehlermeldungen (*stderr*) werden weiterhin auf der Konsole ausgegeben, die regulären Meldungen (*stdout*) dagegen in die Datei `ausgabe.txt` geschrieben. Mit der einfachen Spitzklammer (erste Befehlszeile) legen Sie die Datei neu an und überschreiben die alten Inhalte. Die doppelte Spitzklammer (zweite Befehlszeile) bewirkt, dass die Ausgabe an die bereits bestehende Datei angehängt wird. Wenn Sie die Ausgabe und die Fehlermeldungen getrennt in zwei verschiedene Dateien schreiben wollen, können Sie folgendermaßen vorgehen:

```
test.exe > ausgabe.txt 2> fehler.txt
```

Die `2` steht für den Datei-Deskriptor von *stderr*, gibt also an, dass nicht *stdout*, sondern *stderr* in die Datei `fehler.txt` umgeleitet werden soll. (Sie hätten bei der ersten Spitzklammer auch den Datei-Deskriptor `1` hinzuschreiben können, aber da das der Standard ist, kann es weggelassen werden. Der Datei-Deskriptor wird also beim Aufruf benötigt, um *stdout* von *stderr* zu unterscheiden.) Um die Eingabedatei umzuleiten, d. h. um die Eingabe aus einer Datei zu lesen, drehen Sie die Spitzklammer um (auch hier kann der Datei-Deskriptor `0` dazugeschrieben werden):

```
test.exe < eingabe.txt
```

Eine Umleitung von *stdin* findet oft auch über eine *Pipe* statt – das ist die Verkettung zweier Programme hintereinander. Die Ausgabe des ersten Programmes wird direkt als Eingabe für das zweite Programm weitergereicht:

```
programm1.exe | programm2.exe
```

All diese Umleitungen der Standard-Datenströme funktionieren in Ihrem Programm allerdings nur, wenn Sie dort einen oder mehrere dieser Datenströme öffnen.

16.6.2. Öffnen und Schließen eines Standard-Datenstroms

stdin und *stdout* werden über **`OPEN CONS`** geöffnet, im ersten Fall im Dateimodus **`INPUT`**, im zweiten im Dateimodus **`OUTPUT`**. Wenn Sie sowohl *stdin* als auch *stdout* öffnen wollen, müssen Sie zwei verschiedene Dateinummern verwenden.

Zum Öffnen von *stderr* greifen Sie auf **OPEN ERR** zurück. Auf den Datenstrom kann nur schreibend zugegriffen werden, weshalb kein Dateimodus angegeben werden muss – die Angabe wird ignoriert.

Quelltext 16.14 verwendet zur Demonstration alle drei Standard-Datenströme:

Quelltext 16.14: Standard-Datenströme nutzen

```
' Standard-Datenstroeme oeffnen
DIM AS INTEGER stdin  = FREEFILE
OPEN CONS FOR INPUT AS #stdin
DIM AS INTEGER stdout = FREEFILE
OPEN CONS FOR OUTPUT AS #stdout
DIM AS INTEGER stderr = FREEFILE
OPEN ERR FOR OUTPUT AS #stderr

' Zeile aus stdin lesen und nach stdout ausgeben - Statusmeldung nach stderr
DIM AS STRING zeile
PRINT #stderr, "Zeile wird gelesen und geschrieben"
LINE INPUT #stdin, zeile
PRINT #stdout, "Gelesene Zeile: "; zeile
CLOSE #stdin, #stdout, #stderr
SLEEP
```

Wenn Sie das Programm „normal“ starten, d. h. ohne Umleitungsangaben, gibt es zunächst auf der Konsole die Statusinformation aus (Zeile 11), wartet dann auf eine Benutzereingabe (diese wird während der Eingabe nicht angezeigt!) und gibt diese anschließend auf der Konsole aus. Um die Umleitung zu testen, können Sie z. B. folgenden Aufruf verwenden:

```
test.exe < eingabe.txt > ausgabe.txt 2> status.txt
```

(wobei davon ausgegangen wird, dass das Programm zu `test.exe` compiliert wurde und eine Eingabedatei namens `eingabe.txt` existiert – eine eventuell existierende `ausgabe.txt` und `status.txt` wird beim Aufruf überschrieben).

Standard-Datenströme können zwar mit **CLOSE** geschlossen, dann aber nicht mehr geöffnet werden. Um sie komplett zurückzusetzen (damit können Sie sie theoretisch wieder neu öffnen), verwenden Sie den Befehl **RESET** `0` für *stdin* bzw. **RESET** `1` für *stdout*. Allerdings gibt es in der Regel keinen Grund dafür, mehrmals denselben Standard-Datenstrom zu öffnen. Sie können ihn problemlos bis zum Ende des Programmes geöffnet halten.

Der Vollständigkeit halber soll an dieser Stelle auch **OPEN SCRN** erwähnt werden, das ebenfalls die Standardausgabe öffnet, jedoch nicht die Standardeingabe. Wenn Sie versuchen, von einer mit **OPEN SCRN** geöffneten Datei zu lesen, werden Sie dafür mit einer Endlosschleife belohnt. Der Befehl existiert hauptsächlich zur Rückwärtskompatibilität und bietet keine Vorteile im Vergleich zu **OPEN CONS**.

16.6.3. OPEN PIPE

Für die direkte Kommunikation Ihres Programma mit einem anderen Programm unterstützt FreeBASIC unidirektionale Pipes. Das sind Datenströme, die jedoch nur in eine Richtung laufen, nämlich vom aufgerufenen Programm aus in Richtung des aufrufenden Programmes. **OPEN PIPE** startet ein Programm (bzw. einen Shell-Befehl) und leitet dessen Ausgabe in einen Datenpuffer um, der über **INPUT#** gelesen werden kann. Dies kann folgendermaßen aussehen:

Quelltext 16.15: Unidirektionale Pipe

```
' Pipe oeffnen
DIM AS INTEGER dateinummer = FREEFILE
DIM AS STRING zeile
OPEN PIPE "test.exe" FOR INPUT AS #dateinummer

' Ausgabe zeilenweise einlesen
DO UNTIL EOF(dateinummer)
  LINE INPUT #dateinummer, zeile
  PRINT zeile
LOOP
CLOSE #dateinummer
SLEEP
```

Hilfreich kann dieser Befehl sein, wenn Sie mit Ihrem Programm einen Konsolenbefehl aufrufen und dessen Rückgabe verarbeiten wollen. Über **EXEC** (siehe Kapitel 17.3) ist es ebenfalls möglich, einen Rückgabewert eines Programmes abzurufen, dieser ist jedoch sehr eingeschränkt und eigentlich nur als Fehler-Rückgabe gedacht. **OPEN PIPE** erlaubt Ihnen dagegen jede beliebige Rückgabe, die dann nur noch entsprechend ausgewertet werden muss.

16.7. Hardware-Zugriffe: OPEN COM und OPEN LPT

Nach wie vor existiert der Befehl **OPEN COM**, um einen Zugriff auf einen COM-Port zu öffnen. Moderne Computer besitzen allerdings meist keinen COM-Port mehr, weshalb dieser wiederum vom Betriebssystem über einen speziellen Treiber simuliert werden muss. COM-Port-Zugriffe sind an sich nur für Spezialisten interessant, die bestimmte Geräte ansteuern wollen – für einen Einsteiger (ohne spezielle COM-Port-Neigungen) führt das Thema deutlich zu weit. Bei Interesse kann der dazu gehörige Referenzeintrag zu Rate gezogen werden:

http://www.fb-referenz.de/OPEN%20COM

Auch **OPEN LPT** öffnet eine Hardware-Verbindung, und zwar zu einem Drucker, der im Betriebssystem registriert ist. Dazu müssen Sie den Druckernamen kennen. Der

vollständige Befehl lautet:

```
OPEN LPT "LPT:Druckername,TITLE=Dokumenttitel,EMU=TTY" [FOR Dateimodus] AS #Dateinr
```

- `Druckername` gibt den Namen des Druckers an, unter dem er im System angemeldet ist. Möglich ist auch die Ansprache des Standarddruckers, ohne dass dessen Name bekannt sein muss – dazu kann `Druckername` einfach weggelassen werden.
- `Dokumenttitel` legt fest, unter welchem Namen der Druckauftrag im Spooler angezeigt wird. `TITLE=Dokumenttitel` kann auch weggelassen werden; dann legt FreeBASIC automatisch einen Namen fest.
- `EMU=TTY` ermöglicht die Verwendung von Steuerzeichen wie `CHR(13)` (Carriage Return – Wagenrücklauf), `CHR(10)` (Line Feed – Zeilenvorschub), `CHR(8)` (Backspace), `CHR(9)` (Tabulator) und `CHR(12)` (Form Feed – Seitenvorschub). Wenn `EMU=TTY` ausgelassen wird, müssen die Daten in einer Druckersprache gesendet werden, z. B. ESC/P, HPGL oder PostScript. Allerdings funktioniert `EMU=TTY` nur unter Windows. Unter Linux können also generell nur in einer Druckersprache gesendete Daten gedruckt werden.

Eine einfache Kommunikation mit dem Drucker könnte unter Windows folgendermaßen aussehen: Die Datei "test.txt" wird geöffnet, zeilenweise ausgelesen und zum Ausdruck an den Drucker namens "ReceiptPrinter" geschickt.

Quelltext 16.16: Einfaches Druckbeispiel

```
DIM AS STRING eingabe

' Datei zum Einlesen oeffnen
DIM AS INTEGER dateiNr = FREEFILE
OPEN "test.txt" FOR INPUT AS #dateiNr

' Drucker oeffnen
DIM AS INTEGER druckerNr = FREEFILE
OPEN LPT "LPT:ReceiptPrinter,TITLE=ReceiptWinTitle,EMU=TTY" AS #druckerNr

' Daten zeilenweise einlesen und drucken
WHILE NOT EOF(dateiNr)
  LINE INPUT #dateiNr, eingabe
  PRINT #druckerNr, eingabe
WEND

CLOSE #druckerNr, #dateiNr

PRINT "Beliebige Taste zum Beenden druecken..."
SLEEP
```

Die meisten Druckertreiber werden erst dann mit dem Druck beginnen, wenn sie einen Seitenvorschub (`CHR(12)`) empfangen. Wenn der Drucker über **`CLOSE`** geschlossen wird, sendet FreeBASIC automatisch einen Seitenvorschub, sodass in Quelltext 16.16 der Druck wie geplant durchgeführt wird.

Das Drucken simpler Texte ohne jegliche Formatierung ist in der Regel nicht sehr befriedigend, jedoch können komplexere Ausgaben nur mit detailierter Kenntnis einer der Druckersprachen getätigt werden (oder durch Einsatz einer passenden Bibliothek). Wenn Sie in Ihren Programmen ernsthaft den Einsatz einer Druckfunktion in Erwägung ziehen, sollten Sie sich ausgiebig z. B. mit PostScript auseinandersetzen. Für Anwendungen im rein privaten Bereich ist es oft einfacher, die Daten in einer Datei zu speichern und diese dann seperat auszudrucken.

Alternative Druckmöglichkeit in früheren Versionen:
Eine deutlich einfachere, aber dafür auch eingeschränktere Methode zum Drucken (die auch möglicherweise nicht auf allen Systemen funktioniert) stellt der Befehl **`LPRINT`** dar, der wie **`PRINT`** funktioniert, nur dass er auf dem Standarddrucker ausgibt anstatt auf dem Bildschirm. Mit diesem Befehl ist ausschließlich der Standarddrucker ansteuerbar. **`LPRINT`** steht in der Compiler-Option `-lang fb` (also im Standard) nicht zur Verfügung.

16.8. Fragen zum Kapitel

1. Welche Datei-Modi gibt es? Fassen Sie kurz die Besonderheiten der einzelnen Modi zusammen.

2. Zum Ende des Programmes werden noch geöffnete Dateien automatisch geschlossen. Nennen Sie Gründe dafür, als Programmierer trotzdem alle über **`OPEN`** geöffneten Dateien auch explizit über **`CLOSE`** wieder zu schließen.

3. Welche Datentypen sind für eine binäre Datenspeicherung besser, welche weniger gut geeignet?

4. Schreiben Sie ein Programm, das sich den Namen des letzten Benutzers über eine externe Datei merkt. Wenn die Namensdatei noch nicht existiert, soll der Benutzer nach seinem Namen gefragt und dieser gespeichert werden. Ansonsten wird der zuletzt geschriebene Name eingelesen und der Benutzer mit diesem Namen begrüßt.

Allerdings soll er dann die Möglichkeit erhalten, seinen Namen zu ändern (was dann natürlich wiederum gespeichert werden muss).

17. Betriebssystem-Anweisungen

In Kapitel 16 haben wir bereits Dateien ausgelesen und beschrieben. Jetzt wollen wir weitere Befehle zum Umgang mit Dateien und Ordner kennenlernen. Zunächst geht es um das Navigieren im Dateiensystem und das Anlegen und Löschen von Ordnern, danach um das Umbenennen, Verschieben und Kopieren von Dateien sowie das Abrufen von Dateiinformationen. Außerdem erfahren Sie, wie Sie externe Programme aufrufen können.

17.1. Ordnerfunktionen

17.1.1. Kurze Einführung in das Ordnersystem

Dateien werden in einer hierarchischen Ordnerstruktur gespeichert. Dabei arbeiten verschiedene Betriebssysteme leicht unterschiedlich, es gibt aber auch deutliche Gemeinsamkeiten. Darunter ist zunächst einmal der Dateipfad zu nennen. Pfade können absolut oder relativ angegeben werden, die durchlaufenen Ordner werden durch ein betriebssystemabhängiges Trennzeichen getrennt.

Absolute Pfadangaben

Eine absolute Pfadangabe verweist auf den gewünschten Ordnerpfad, unabhängig davon, von welchem Pfad aus sie aufgerufen wird.

- Unter Windows beginnt der absolute Pfad mit einem Laufwerksbuchstaben gefolgt von einem Doppelpunkt, oder aber bei einem Netzwerkordner mit zwei Backslashes gefolgt vom Netzwerkordner-Namen (Uniform Naming Convention). Das Trennzeichen zwischen den Ordnern ist der Backslash, wobei Windows jedoch auch den Slash akzeptiert. Ein absoluter Pfad könnte also folgendermaßen aussehen:
  `C:\Users\Peter\Documents\BASIC\` (Ordner auf dem Laufwerk `C:`)
  `\\PC-WOHNZIMMER\Users\Paul\Documents\BASIC\` (Ordner auf dem Computer namens `PC-WOHNZIMMER`)
- Unter Linux sind sämtliche Ressourcen im Root-Verzeichnis `/` eingebunden. Es gibt also keine Laufwerksbuchstaben, sondern alle Dateien, egal ob Festplatte, DVD-Laufwerk oder Drucker (ja, auch Drucker und andere Hardware werden als Dateien

angesprochen) sind über das Root-Verzeichnis eingebunden. Als Trennzeichen wird / verwendet. Ein möglicher Pfad könnte lauten:
`/home/Mary/Dokumente/BASIC/`

Da der Slash als Trennzeichen von Linux benötigt und von Windows akzeptiert wird, ist auch unter Windows die Verwendung des Slashes anzuraten. Dadurch bleibt der Quelltext systemunabhängig (natürlich nicht bei Verwendung absoluter Pfade). Nur wenn Sie Programme für DOS schreiben wollen, müssen Sie den Backslash verwenden.
Das Buch verwendet im Folgenden immer den Slash als Trennzeichen. Außerdem werden Ordnerpfade im Buch immer mit einem Trennzeichen abgeschlossen, um deutlich zu machen, dass es sich um einen Ordner und nicht um eine Datei handelt.[34]

Sowohl unter Linux als auch unter Windows sind in jedem Ordner automatisch zwei Ordner fest angelegt, nämlich `./` (verweist auf den Ordner selbst, was insb. unter Linux wichtig ist) und `../` (verweist auf den ihm übergeordneten Ordner, also den Elternordner). Im Dateibrowser werden sie meist nicht angezeigt, sie sind jedoch für die Pfadnavigation wichtig.

Relative Pfadangaben

Eine relative Pfadangabe ist abhängig vom Pfad, von dem aus sie aufgerufen wird. Eine Pfadangabe wie `inc/freetype2/` etwa gibt an, dass *vom aktuellen Ordner ausgehend* der Unterordner `inc/` und dort der Unterordner `freetype2/` aufgerufen werden soll. Befindet man sich z. B. gerade im FreeBASIC-Installationsordner, dann verweist der Pfad `inc/Lua/` auf die Include-Dateien von Lua, die sich in Unterordnern des FreeBASIC-Installationsordners befinden.

In der Regel werden wir immer mit relativen Pfadangaben arbeiten. Sie werden z. B. des Öfteren auf Dateien innerhalb Ihres Projekt zugreifen wollen, die in einem Unterodner Ihres Projektordners liegen. Wenn Sie ausschließlich relative Pfadangaben verwenden, können Sie das komplette Projekt problemlos an einen anderen Ort bewegen – etwa auf den USB-Stick ziehen oder auf einen anderen Rechner kopieren – ohne dass die Pfadbezüge verloren gehen. Bei Verwendung absoluter Pfade ist das für gewöhnlich nicht möglich.

An dieser Stelle kommt auch der Elternordner ins Spiel. Dazu gleich ein konkretes Beispiel: Sie befinden sich im Projektordner
`C:/Users/Peter/Documents/BASIC/einsteigerhandbuch`
und rufen von dort aus den Pfad `../gemeinsameDateien/Bilder/` auf. Damit

[34] Unter Linux werden auch Ordner als (spezielle) Dateien behandelt. Das Buch verwendet der Einfachheit halber den Begriff *Datei* nicht für Ordner.

landen Sie im Ordner
`C:/Users/Peter/Documents/BASIC/gemeinsameDateien/Bilder`
Die zwei Punkte wechseln in den Elternordner `C:/Users/Peter/Documents/BASIC/`, und von dort aus geht es weiter in den Ordner `gemeinsameDateien/` und `Bilder/`. Beachten Sie jedoch, dass durch die Einbindung des Elternordners und dessen Unterordner die Pfadunabhängigkeit des Projekts verloren gehen kann!

17.1.2. Arbeitsverzeichnis anzeigen und wechseln: `CURDIR()` und `CHDIR()`

Auch FreeBASIC-Programme werden von einem Pfad aus ausgeführt, d. h. alle relativen Pfadangaben innerhalb des Programmes werden von diesem Pfad aus berechnet. Man spricht hierbei vom Arbeitsverzeichnis. In der Regel ist das der Pfad, von dem aus das Programm gestartet wurde, und das wiederum ist häufig der Pfad, in dem sich das ausgeführte Programm befindet. Allerdings muss das nicht so sein! Sie können Ihre Programme über Konsole von jedem beliebigen Pfad aus starten, womit sich dann natürlich das Arbeitsverzeichnis ändert.

Das aktuelle Arbeitsverzeichnis erhalten Sie über die Funktion **`CURDIR()`** (*CURrent DIRectory*). Mithilfe des Befehls **`CHDIR()`** (*CHange DIRectory*) können Sie das Arbeitsverzeichnis ändern, wobei hier, wie üblich, relative und absolute Pfadangaben erlaubt sind. Der Rückgabewert gibt an, ob der Verzeichniswechsel erfolgreich war (dazu unten mehr). Wie die meisten Funktionen lässt sich **`CHDIR()`** auch als Anweisung aufrufen.

Quelltext 17.1: Arbeitsverzeichnis anzeigen und wechseln

```
PRINT "Sie befinden sich aktuell im Verzeichnis " & CURDIR
PRINT "Ich wechsle nun in den Elternordner."
CHDIR ".."
PRINT "Das neue Arbeitsverzeichnis lautet jetzt " & CURDIR
PRINT "Ich wechsle in einen Ordner, der nicht existiert."
CHDIR "../ordner/pfad/der/nicht/vorhanden/ist"
PRINT "Das neue Arbeitsverzeichnis lautet jetzt " & CURDIR
PRINT "Noch ein Versuch, diesmal mit Fehlerkontrolle:"
IF CHDIR("../ordner/pfad/der/nicht/vorhanden/ist") THEN
  PRINT "Wechsel fehlgeschlagen - der Ordner existiert wohl nicht."
ELSE
  PRINT "Das neue Arbeitsverzeichnis lautet jetzt " & CURDIR
END IF
PRINT "Ich wechsle nun in den Ordner /.."
CHDIR "/.."
PRINT "Das neue Arbeitsverzeichnis lautet jetzt " & CURDIR
SLEEP
```

Ausgabe

```
Sie befinden sich aktuell im Verzeichnis D:\Users\Ich\BASIC
Ich wechsle nun in den Elternordner.
Das neue Arbeitsverzeichnis lautet jetzt D:\Users\Ich
Ich wechsle in einen Ordner, der nicht existiert.
Das neue Arbeitsverzeichnis lautet jetzt D:\Users\Ich
Noch ein Versuch, diesmal mit Fehlerkontrolle:
Wechsel fehlgeschlagen - der Ordner existiert wohl nicht.
Ich wechsle nun in den Ordner /..
Das neue Arbeitsverzeichnis lautet jetzt D:\
```

Durch die Verwendung der Trennzeichens `/` ist das Programm sowohl unter Windows als auch Linux lauffähig, auch wenn Windows bei der Ausgabe selbst `\` verwendet. Die angezeigten Pfadnamen werden sich bei Ihnen natürlich unterscheiden. Ein paar wichtige Informationen können wir jedoch aus der Ausgabe herauslesen:

- Pfade, die mit Slash (oder unter Windows auch mit Backslash) beginnen, werden als absolute Pfade interpretiert. Unter Linux ist der Pfad `/` das Root-Verzeichnis des Systems. Unter Windows landen Sie im obersten Ordner des aktuellen Laufwerks. Der mit dem Befehl `CHDIR "/.."` zusätzlich durchgeführte Wechsel in den nächsthöheren Ordner bleibt wirkungslos, da `/` bereits der oberste Ordner ist (dennoch ist er erlaubt, da auch das Root-Verzeichnis den Ordner `..` enthält).
- Wenn Sie in einen Ordner wechseln wollen, der nicht existiert, wird der Befehl verworfen; es findet auch kein „Teilwechsel“ statt. Sie können aber prüfen, ob der Wechsel erfolgreich war, wenn Sie **`CURDIR()`** als Funktion aufrufen. Bei Erfolg liefert sie `0` zurück, bei Misserfolg `-1`.
- Die Pfadangabe erfordert keinen abschließenden Slash, und **`CURDIR()`** gibt auch keinen abschließenden Slash zurück. Sie können bei **`CHDIR()`** den abschließenden Slash setzen, wenn Sie wollen. er wird dann vom Programm ignoriert. Den fehlenden Slash bei der Rückgabe sollten Sie jedoch im Hinterkopf behalten – dazu sehen wir ein Beispiel in Quelltext 17.2.

17.1.3. Programmpfad anzeigen: `EXEPATH()`

`EXEPATH()` gibt den Programmpfad aus, also den Pfad, in dem sich das ausgeführte Programm befindet. Wenn Sie, wie z. B. in Quelltext 16.3, auf eine Datei zugreifen wollen, die sich im selben Ordner wie das Programm befindet, gelingt das nur, wenn der

Programmpfad mit dem Arbeitsverzeichnis übereinstimmt. Ansonsten wird die Datei im falschen Ordner gesucht und dort nicht gefunden – oder, manchmal noch schlimmer, es wird eine falsche Datei geöffnet, die zufällig denselben Namen hat. Sie können **EXEPATH** aber beim Öffnen der Datei als Pfadangabe mitgeben, um so sicher zu stellen, dass auf die richtige Datei zugegriffen wird.

Quelltext 17.2: Eine im Programmpfad liegende Datei auslesen

```
' Oeffne eine Datei, die im selben Ordner liegt wie das Programm
DIM AS INTEGER dateinummer
DIM AS STRING zeile

PRINT "Programmpfad: " & EXEPATH

dateinummer = FREEFILE
IF OPEN(EXEPATH & "/test.bas" FOR INPUT AS #dateinummer) <> 0 THEN
  PRINT "FEHLER: eingabe.txt konnte nicht geoeffnet werden."
ELSE
  DO UNTIL EOF(dateinummer)
    LINE INPUT #dateinummer, zeile
    PRINT zeile
  LOOP
  CLOSE #dateinummer
END IF
```

Beachten Sie den Slash vor dem Namen der Datei! **EXEPATH()** liefert, ebenso wie **CURDIR()**, kein abschließendes Trennzeichen. Wenn Sie also an **EXEPATH()** einen Pfad oder einen Dateinamen anhängen, müssen Sie dazwischen ein Trennzeichen hinzufügen.

Nun liegt es nahe, mit `CHDIR EXEPATH` das Arbeitsverzeichnis auf den Programmpfad zu setzen und so weitere Zugriffe zu erleichtern. Das ist jedoch nicht immer von Vorteil, denn dadurch gehen die Informationen über das ursprüngliche Arbeitsverzeichnis verloren. Oft ist es gerade wünschenswert, wenn der Benutzer das Arbeitsverzeichnis wählen kann, um z. B. mit bestimmten Konfigurationsdateien zu arbeiten. Ein Wechsel auf den Programmpfad ist höchstens dann empfehlenswert, wenn Sie ausschließlich auf eigene Programmressourcen zurückgreifen müssen.

17.1.4. Ordner anlegen und löschen: MKDIR() und RMDIR()

MKDIR() (*MaKe DIRectory*) legt einen neuen Ordner an. Als Funktion aufgerufen gibt der Befehl im Erfolgsfall `0` und ansonsten `-1` zurück – das kann z. B. vorkommen, wenn Sie nicht über ausreichende Schreibrechte verfügen, der Ordner bereits existiert oder der gewählte Ordnername ungültige Zeichen enthält.

Um einen vorhandenen Ordner wieder zu löschen, gibt es den Befehl **RMDIR()**, der, als Funktion eingesetzt, wiederum im Erfolgsfall `0` und bei einem Fehler `-1` zurückgibt.

Ein Order kann nur gelöscht werden, wenn er leer ist, und er wird auch nicht in den Papierkorb verschoben, sondern wirklich gelöscht.

Mit **MKDIR()** und **RMDIR()** lassen sich hervorragend temporäre Ordner anlegen, die bei Programmende wieder gelöscht werden. Quelltext 17.3 legt in der Mitte des Programmes eine kurze Pause ein, sodass Sie den Zwischenstand in Ihrem Dateibrowser kontrollieren können.

Quelltext 17.3: Temporären Ordner anlegen

```
IF MKDIR("tempOrdner") THEN
  PRINT "tempOrdner kann nicht angelegt werden - existiert der Ordner bereits?"
  END
END IF
IF CHDIR("tempOrdner") THEN
  PRINT "tempOrdner kann nicht geoeffnet werden. ;("
  END
END IF
' Das Programm befindet sich nun in tempOrdner. Nun koennen die Dateien
' angelegt und beschrieben werden.
DIM AS INTEGER dateiNr = FREEFILE
OPEN "test.dat" FOR OUTPUT AS #dateiNr
CLOSE #dateiNr
PRINT "Das Anlegen der temporaeren Dateien ist abgeschlossen."
PRINT "Druecken Sie eine beliebige Taste, um fortzufahren."
GETKEY
KILL "test.dat"
CHDIR ".."
IF RMDIR("tempOrdner") THEN
  PRINT "tempOrdner kann nicht geloescht werden. ;("
ELSE
  PRINT "tempOrdner wurde wieder erfolgreich geloescht."
END IF
SLEEP
```

17.2. Dateifunktionen

QuickBASIC stellte mit **NAME** zur Umbenennung von Dateien nur noch einen weiteren Dateien-Befehl zur Verfügung. Inspiriert von Visual Basic wurden in FreeBASIC noch weitere Funktionen eingebaut, etwa um die Existenz einer Datei zu überprüfen oder Dateiattribute auszulesen. Da diese Befehle nicht im QuickBASIC-Sprachschatz vorkommen, müssen sie dem Compiler zunächst bekannt gemacht werden.

Die Deklarationen der genannten Funktionen sind in der Datei *file.bi* niedergelegt, die sich im Ordner `inc/` des FreeBASIC-Installationsordner befindet. Um sie nutzen zu können, muss diese Datei in das Programm eingebunden werden. Dies geschieht über die Befehlszeile

```
#INCLUDE "file.bi"
```

file.bi wird wiederum von *vbcompat.bi* eingebunden, welche letztendlich eine Sammlung VB-kompatibler Funktionen darstellt. Statt *file.bi* können Sie also auch *vbcompat.bi* einbinden:

```
#INCLUDE "vbcompat.bi"
```

Das Einbinden „externer Befehle“ mag einen Puristen vielleicht abschrecken. Es handelt sich bei diesen Funktionen jedoch um regulären Sprachbestandteil von FreeBASIC, nur dass die Namen im Programmcode nicht von vornherein bekannt sind.[35] In vielen anderen Programmiersprachen ist das selektive Einbinden der benötigten Befehle im Übrigen vollkommen üblich.

17.2.1. Dateien umbenennen und verschieben: NAME()

Mit **NAME()** wird eine Datei umbenannt. Die Funktion nimmt zwei Strings als Argumente entgegen, nämlich den ursprünglichen Dateinamen und den neuen Namen, in den umbenannt werden soll. Als Funktion verwendet, liefert **NAME()** wie gewohnt `0`, wenn kein Fehler auftritt, und `-1` bei einem Fehler (z. B. bei fehlenden Schreibrechten oder wenn bereits eine Datei mit dem neuen Namen existiert).

Quelltext 17.4: Dateien umbenennen und verschieben

```
' Benenne die Datei 'alterName.txt' zu 'neuerName.txt' um
' ohne Erfolgskontrolle
NAME "alterName.txt", "neuerName.txt"

' Verschiebe die Datei TODO.doc vom Unterordner 'zuErledigen' in den
' Unterordner 'erledigt', ohne sie dabei umzubenennen
IF NAME ("zuErledigen/TODO.doc", "erledigt/TODO.doc") THEN
  PRINT "Fehler: Die Datei konnte nicht verschoben werden."
END IF
SLEEP
```

Im zweiten Teil des Quelltextes sehen Sie, dass sich auch die Pfadangabe im alten und neuen Dateinamen unterscheiden kann. In diesem Fall versucht das Programm, die Datei zu verschieben. Wenn sich zudem auch der Dateiname unterscheidet, wird mit der Verschiebung auch eine Umbenennung durchgeführt. Umgekehrt bedeutet das aber auch:

[35] Ein Grund für diese Vorgehensweise ist die Abwärtskompatibilität. Beispielsweise enthält *file.bi* die Deklaration von **FILEEXISTS** zur Überprüfung der Existenz einer Datei. Hätte nun ein Anwender bereits selbst eine Funktion namens **FILEEXISTS** definiert, wäre sein Programm wegen des Namenskonflikts nicht mehr lauffähig. Daher versucht FreeBASIC, das von QuickBASIC sowieso bereits eine große Menge an Schlüsselwörtern erbt, neue Schlüsselwörter nur sehr sparsam einzuführen.

Wenn Sie eine Datei umbenennen wollen, die sich nicht im Arbeitsverzeichnis befindet, müssen Sie sowohl im alten als auch im neuen Namen den Pfad angeben, da die Datei sonst verschoben wird.

17.2.2. Dateien kopieren: `FILECOPY()`

Mit dem QuickBASIC-Sprachumfang war das Kopieren einer Datei nur möglich, indem sie komplett eingelesen und die Werte in eine neue Datei geschrieben wurden. FreeBASIC greift zu diesem Zweck die Funktion **`FILECOPY()`** auf. Auch diesem Befehl wird der Quell- und der Zielname übergeben, und als Funktion aufgerufen gibt er im Erfolgsfall `0` zurück. Etwas ungewöhnlich ist, dass bei einem Fehler `1` zurückgegeben wird und nicht `-1`.

Quelltext 17.5: Datei kopieren

```
' Kopiere die Datei 'test.bas' in einen temporaeren Unterordner.
' Dort kann sie bearbeitet werden, ohne die originale Datei zu beeinflussen.

' Notwendige INCLUDE-Datei einbinden
#INCLUDE "vbcompat.bi"

' tempOrdner anlegen, falls er noch nicht existiert. Der Ordner soll am Ende nur
' geloescht werden, wenn er vor Programmstart noch nicht existierte - daher muss
' gespeichert werden, ob das Anlegen erfolgreich war.
DIM AS BOOLEAN tempOrdnerExistierte = MKDIR("tempOrdner")

' Datei kopieren
IF FILECOPY("test.bas", "tempOrdner/test.bas") = 0 THEN
  ' jetzt kann die neue Datei bearbeitet werden
  KILL "tempOrdner/test.bas"
END IF

' ggf. tempOrdner wieder loeschen
IF NOT tempOrdnerExistierte THEN RMDIR "tempOrdner"
SLEEP
```

17.2.3. Dateiinformationen abrufen: `FILEATTR()`, `FILELEN()` und `FILEDATETIME()`

Über eine geöffnete Datei können mit **`FILEATTR()`** verschiedene Informationen abgerufen werden:

```
Rueckgabe = FILEATTR(Dateinummer[, Ausgabe])
```

`Dateinummer` ist dabei die Nummer, unter der die Datei mit **`OPEN`** geöffnet wurde. Als `Ausgabe` kann einer der Werte `1`, `2` oder `3` angegeben werden. Statt sich jedoch eine

recht nichtssagende Nummernaufteilung merken zu müssen, stellt *file.bi* Konstanten zur Verfügung, die stattdessen verwendet werden können. Wird `Ausgabe` nicht angegeben, verwendet FreeBASIC den Wert `fbFileAttrMode`.

Ausgabe	zurückgegebene Information	mögliche Rückgabe

`fbFileAttrMode`	`Dateimodus, unter dem die Datei geöffnet wurde`	`fbFileModeInput`
		`fbFileModeOutput`
		`fbFileModeRandom`
		`fbFileModeAppend`
		`fbFileModeBinary`

`fbFileAttrHandle`	`Handle für den Zugriff`	*(Handle)*

`fbFileAttrEncoding`	`Codierungsvariante, mit der die Datei behandelt wird`	`fbFileEncodASCII`
		`fbFileEncodUTF8`
		`fbFileEncodUTF16`
		`fbFileEncodUTF32`

Der Rückgabewert für die Ausgabeoption `fbFileAttrHandle` ist der Handle, über den das System auf die Datei zugreift. Er ist identisch mit dem, den auch die C-Runtime-Library verwendet. Der Handle kann damit an andere C-Funktionen übergeben werden. Dieses Thema führt hier allerdings zu weit. Zu den anderen Optionen folgt ein Beispiel in Quelltext 17.6.
Kann eine Information nicht abgefragt werden, z. B. weil die angegebene Dateinummer nicht belegt ist, so ist der Rückgabewert `0`.

Man könnte sich nun fragen, wozu der Dateimodus oder die Codierungsvariante einer geöffneten Datei abgefragt werden muss, wenn man diese Informationen doch selbst über **`OPEN`** festgelegt hat. Allerdings kann ein Programm in mehrere Teile gegliedert sein, die unabhängig voneinander agieren. Das einfachste Szenario dazu ist ein Unterprogramm, dem die Dateinummer einer Datei übergeben wird, mit der es arbeiten soll (das Unterprogramm muss dann etwa den Namen der Datei nicht kennen, sondern nur den „Einhängepunkt“). Über **`FILEATTR()`** kann es nun beispielsweise die Codierungsvariante ermitteln und so sicherstellen, dass die Daten in korrekter Form gelesen bzw. geschrieben werden.

Zur Veranschaulichung der Syntax hier noch ein (minimal geändertes) Beispiel aus der deutschsprachigen FreeBASIC-Referenz:

Quelltext 17.6: Dateiattribute abfragen

```
#INCLUDE "vbcompat.bi"
DIM f AS INTEGER = FREEFILE
OPEN "test.bas" FOR INPUT AS #f

SELECT CASE FILEATTR(f, fbFileAttrMode)
CASE fbFileModeInput
   PRINT "Die Datei wurde im INPUT-Modus geoeffnet"
CASE fbFileModeOutput
   PRINT "Die Datei wurde im OUTPUT-Modus geoeffnet"
CASE fbFileModeAppend
   PRINT "Die Datei wurde im APPEND-Modus geoeffnet"
CASE fbFileModeRandom
   PRINT "Die Datei wurde im RANDOM-Modus geoeffnet"
CASE fbFileModeBinary
   PRINT "Die Datei wurde im BINARY-Modus geoeffnet"
END SELECT

PRINT "Das System-Handle zur Datei ist"; FILEATTR(f, fbFileAttrHandle)

SELECT CASE FILEATTR(f, fbFileAttrEncoding)
CASE fbFileEncodASCII
   PRINT "Die Datei ist als ASCII-Datei geoeffnet"
CASE fbFileEncodUTF8
   PRINT "Die Datei ist als UTF-8-Datei geoeffnet"
CASE fbFileEncodUTF16
   PRINT "Die Datei ist als UTF-16-Datei geoeffnet"
CASE fbFileEncodUTF32
   PRINT "Die Datei ist als UTF-32-Datei geoeffnet"
END SELECT

CLOSE #f
SLEEP
```

Ausgabe

```
Die Datei wurde im INPUT-Modus geoeffnet
Das System-Handle zur Datei ist 1971013216
Die Datei ist als ASCII-Datei geoeffnet
```

Für **FILELEN()** und **FILEDATETIME()** ist es dagegen nicht erforderlich, die Datei vorher zu öffnen. **FILELEN()** gibt die Länge einer Datei in Byte an, **FILEDATETIME()** liefert den Zeitpunkt, an dem die Datei das letzte Mal geändert wurde. Dieser Zeitpunkt wird als *Serial Number* zurückgegeben, auf die Kapitel 18 ausführlich eingeht. Auch die formatierte Ausgabe einer *Serial Number* mittels **FORMAT()** wird dort ausführlich besprochen.

```
#INCLUDE "vbcompat.bi"
DIM AS STRING dateiname = "C:\Programme\FreeBASIC\fbc.exe"
PRINT "Die Datei " & dateiname & " ist " & FILELEN(dateiname) & " Byte lang."
PRINT "Zeitpunkt der letzten Aenderung: ";
PRINT FORMAT(FILEDATETIME(dateiname), "dd.mm.yyyy, hh:mm:ss")
SLEEP
```

Wenn die Datei nicht existiert oder keine Zugriffsberechtigung besteht, liefert **FILELEN()** den Wert `0` zurück. Achtung: Der Rückgabewert `0` kann auch bedeuten, dass die Datei tatsächlich eine Länge von 0 Byte besitzt!

17.2.4. Auf Existenz prüfen: FILEEXISTS()

Wenn Sie einfach nur überprüfen wollen, ob eine Datei existiert, können Sie **FILEEXISTS()** verwenden. Der Rückgabewert ist `-1`, wenn die Datei existiert, oder `0`, wenn keine Datei mit diesem Namen existiert. Auch wenn es sich beim angegebenen Namen um einen Ordner statt um eine Datei handelt, wird `0` zurückgegeben.

```
#INCLUDE "vbcompat.bi"
IF FILEEXISTS("suchmich.bas") THEN
  PRINT "Die Datei existiert!"
ELSE
  PRINT "Die Datei existiert nicht!"
END IF
SLEEP
```

17.2.5. Dateien suchen: DIR()

Bisher war es immer nötig, den genauen Dateinamen der gewünschten Datei zu kennen. Das ist nicht immer möglich. Vielleicht benötigen Sie einmal eine Liste aller PDF-Dateien in Ihrem Ordner, oder Sie wollen alle Dateien löschen, die mit *temp_* beginnen. Für die Suche nach Dateien, die einem bestimmten Muster entsprechen, eignet sich **DIR()**.

```
DIR([Dateiangabe][, Attributnummer[, Rueckgabeattribut]])
```

- *Dateiangabe* ist der String eines Datei- oder Ordnernamens, der auch Wildcards enthalten kann, nämlich `*` für eine Folge beliebig vieler Zeichen (auch 0) und `?` für kein oder ein beliebiges Zeichen. Beispielsweise würde `"bsp*.ba?"` unter anderem nach den Dateien *bsp.bas*, *bsp123.bak* und *bsp_neu.ba* suchen.
  Dateiangabe kann auch eine Pfadangabe enthalten, dort sind jedoch keine Wildcards erlaubt.

- *Attributnummer* erlaubt die Angabe verschiedener Attribute, welche die Dateien erfüllen müssen – etwa ob sie schreibgeschützt sind oder ob es sich um Ordner handelt.
- In die Variable *Rueckgabeattribut* werden die Attribute gespeichert, welche für die gefundene Datei *tatsächlich* vorliegen.

DIR() gibt den Namen der gefundenen Datei zurück. Der Pfadname entfällt bei der Rückgabe. Eine leere Rückgabe zeigt an, dass keine Datei gefunden wurde, die dem Suchmuster entspricht.

Um den Sinn hinter *Rueckgabeattribut* zu verstehen, müssen wir etwas in die Funktionsweise von **DIR()** hineinschauen. *Attributnummer* setzt sich aus einer Kombination verschiedener Werte zusammen, die in der Datei *file.bi* mit sprechenden Namen belegt wurden:

- `fbReadOnly`: schreibgeschützte Datei
- `fbHidden`: versteckte Datei
- `fbSystem`: Systemdatei
- `fbDirectory`: Ordner
- `fbArchive`: archivierbare Datei

Außerdem ist `fbNormal = fbReadOnly OR fbArchive`. Sie können jede gewünschte Kombination von Attributen über eine **OR**-Verknüpfung herstellen, also z. B. mit `fbNormal OR fbHidden OR fbDirectory` nach Einträgen suchen, die auch versteckt und/oder Ordner sein können.

Wird *Attributnummer* nicht angegeben, wird `fbNormal` verwendet. **DIR** sucht also nach allen „normalen“ Dateien einschließlich schreibgeschützter und archivierbarer Dateien, aber keine versteckten Dateien oder Ordner. Der Schalter `fbReadOnly` bedeutet also, dass *auch* nach schreibgeschützten Dateien gesucht wird, aber *nicht nur* schreibgeschützte Dateien. Wenn Sie nun lediglich die schreibgeschützten Dateien ermitteln wollen, jedoch nicht die beschreibbaren, müssen Sie die Rückgabe anhand von *Rueckgabeattribut* filtern, also nachsehen, ob dort `fbReadOnly` gesetzt ist oder nicht.

Quelltext 17.7: Suche nach einer Datei

```
#INCLUDE "vbcompat.bi"
DIM AS INTEGER attribute
DIM AS STRING dateiname = DIR("suchmich.*", fbNormal, attribute)
IF dateiname = "" THEN
  PRINT "Datei nicht gefunden"
ELSE
  PRINT "Gefundene Datei: " & dateiname
  IF attribute AND fbReadOnly THEN
    PRINT "Die Datei ist schreibgeschuetzt."
  ELSE
    PRINT "Die Datei kann beschrieben werden."
  END IF
END IF
SLEEP
```

DIR() benötigt zwar keine Einbindung von *file.bi* bzw. *vbcompat.bi*, die verwendeten Attributvariablen `fbNormal` und `fbReadOnly` dagegen schon.
Die Umsetzung der Dateiattribute wird unter den verschiedenen Betriebssystemen unterschiedlich gehandhabt. Beispielsweise markiert Windows versteckte Dateien über ein spezielles Dateienattribut, während Linux alle Dateien als versteckt behandelt, die mit einem Punkt beginnen. Windows stellt eine Option zur Verfügung, Dateien als nicht archivierbar zu kennzeichnen, was dann aber bei einem Schreibzugriff großzügig annulliert wird; Linux dagegen unterstützt dieses Attribut nicht ... Im Zweifelsfall müssen Sie mit den Attributen etwas herumexperimentieren.

Nun hat **DIR()** in Quelltext 17.7 zwar Wildcards unterstützt, aber nur eine einzige Datei zurückgeliefert. Allerdings wird das letzte Suchmuster immer intern gespeichert. Jetzt kann **DIR()** erneut aufgerufen werden, diesmal jedoch mit einem Leerstring als *Dateiangabe*. Dadurch liefert es den nächsten passenden Treffer. Diesen „leeren“ Aufruf können Sie auch in eine Schleife packen und damit alle passenden Treffer nacheinander abarbeiten. Wenn Sie *Rueckgabeattribut* nutzen wollen, müssen Sie das erneut angeben (ansonsten wird der Wert nicht aktualisiert). *Dateiangabe* und *Attributnummer* entfällt jedoch und wird aus der letzten Abfrage übernommen.

Quelltext 17.8: Auflisten aller normalen Dateien und Ordner

```
#INCLUDE "vbcompat.bi"
DIM AS INTEGER attribute
DIM AS STRING dateiname = DIR("*.*", fbNormal OR fbDirectory, attribute)
DO WHILE LEN(dateiname)             ' solange ein Eintrag gefunden wurde
  ' Kennzeichne Ordner durch einen Stern
  IF attribute and fbDirectory THEN
    PRINT "* ";
  ELSE
    PRINT "  ";
  END IF
  ' Gib den Dateinamen (bzw. Ordnernamen) aus
  PRINT dateiname
  ' Suche nach dem naechsten Eintrag
  dateiname = DIR(attribute)
LOOP
SLEEP
```

Wenn Sie auf die Auswertung von *Rueckgabeattribut* verzichten, können Sie sich am Ende der Schleife sogar einfach mit einem `dateiname = DIR` begnügen.

17.3. Externe Programme starten

Externe Dateien können Sie nicht nur zum Lesen und Schreiben öffnen, Sie können sie auch ausführen – jedenfalls wenn es sich um eine ausführbare Datei handelt. Wie in BASIC üblich, gibt es dazu gleich mehrere Befehle, die sich im Detail unterscheiden und die in erster Linie aufgrund der Abwärtskompatibilität beibehalten wurden.

17.3.1. `CHAIN()`, `EXEC()` und `RUN()`

```
' Drei verschiedene Varianten, ein externes Programm zu starten
Rueckgabewert = CHAIN(Programm)
Rueckgabewert = EXEC(Programm, Argumente)
Rueckgabewert = RUN(Programm[, Argumente])
```

Die drei Funktionen **`CHAIN()`**, **`EXEC()`** und **`RUN()`** übergeben jeweils die Kontrolle an ein anderes Programm und startet dieses. *Programm* ist ein String mit dem Dateinamen des zu startenden Programmes (eventuell mit Pfadangabe). *Argumente* ist ein String, mit den Argumenten, die an das Programm übergeben werden sollen. Während **`CHAIN()`** keine Argumentübergabe erlaubt, wird sie von **`EXEC()`** erzwungen – Sie können aber auch einen Leerstring `""` übergeben. Bei **`RUN()`** ist die Übergabe der Argumente optional. Der *Rueckgabewert* der drei Funktionen ist der *Errorlevel*, der vom aufgerufenen Programm

zurückgegeben wurde, oder `-1`, falls das Programm nicht aufgerufen werden konnte. (Beachten Sie aber, dass auch der zurückgegebene Errorlevel `-1` sein könnte!)

`RUN()` unterscheidet sich von den beiden anderen Funktionen dadurch, was nach der Ausführung des aufgerufenen Programmes weiter passiert. Während bei **`CHAIN()`** und **`EXEC()`** anschließend die Kontrolle an das aufrufende Programm zurückgegeben wird und dieses normal weiterläuft, beendet **`RUN`** nach erfolgreicher Ausführung und gibt die Kontrolle an das Betriebssystem zurück. Entsprechend können Sie bei **`RUN()`** auch nur den Rückgabewert `-1` bei missglücktem Aufruf erhalten – bei einem erfolgreichen Aufruf haben Sie ja anschließend keine Möglichkeit mehr, die Rückgabe auszuwerten.

Sie sehen, dass die Unterschiede zwischen den drei Funktionen sehr gering sind. **`CHAIN()`** kann problemlos durch **`EXEC()`** ersetzt werden, und auch **`RUN()`** unterscheidet sich von **`EXEC()`** im Prinzip nur durch das Verhalten nach Beendigung des aufgerufenen Programmes.

Natürlich können Sie auch in Ihren eigenen Programmen am Ende ein Errorlevel setzen. Dies geschieht ganz einfach mit der Anweisung **`END`**, gefolgt von einem Integerwert. In Kapitel 17.4 wird dazu ein Beispiel präsentiert.

Kompatibilität zu älteren BASIC-Dialekten:
Neben **`END`** können auch die Befehle **`SYSTEM`** und **`STOP`** verwendet werden, um ein Programm zu beenden. Beide Befehle arbeiten identisch zu **`END`** und bestehen nur aus Kompatibilitätsgründen zu älteren BASIC-Dialekten. Es wird empfohlen, stattdessen **`END`** zu verwenden.

17.3.2. `SHELL()`

```
Rueckgabe = SHELL(Kommando)
```

Mit **`SHELL()`** steht noch eine vierte Funktion zur Verfügung. Diese unterscheidet sich jedoch tatsächlich etwas von den anderen, denn zum einen wird ihr statt eines Programmes ein Shell-Befehl übergeben (der seinerseits aber auch ein Programmname sein kann), zum anderen findet eine Parameterübergabe innerhalb dieses Befehls statt. **`SHELL()`** ist ungleich mächtiger als die zuvor genannten Befehle, da über den Shell-Befehl z. B. auch Programmaufrufe verknüpft und Ausgaben umgeleitet werden können.

Beachten Sie, dass sich zwei nacheinander ausgeführte **`SHELL()`**-Aufrufe nicht gegenseitig beeinflussen! Sie können beispielsweise als Kommando auch einen Ordnerwechsel (*cd*) durchführen; dieser ist in einem folgenden **`SHELL()`**-Aufrufe aber nicht mehr vorhanden.

Dazu ein einfaches Beispiel – der Befehl *cd* wechselt das Verzeichnis und *dir* listet den Inhalt des Verzeichnisses auf.

Quelltext 17.9: Ordnerinhalt anzeigen mit SHELL()

```
' Inhalt des uebergeordneten Verzeichnisses anzeigen

' So funktioniert es nicht:
SHELL "cd .."          ' Verzeichniswechsel
SHELL "dir"            ' Ordnerinhalt auflisten
SLEEP

' So klappt es aber:
SHELL "cd .. && dir"   ' beides hintereinander
SLEEP
```

Natürlich kann hier nicht ansatzweise auf die Möglichkeiten der Shell-Kommandos eingegangen werden. Es ist jedoch lohnenswert, sich näher damit zu beschäftigen, da mit den zur Verfügung stehenden Werkzeugen Programme effizient miteinander verknüpft werden können.

17.4. Kommandozeilenparameter auswerten

In den vorigen Abschnitten wurde gezeigt, wie man ein Programm zusammen mit Argumenten aufrufen kann. Das hat denselben Effekt wie wenn Sie das Programm von Konsole aus starten und dabei die Argumente hinter dem Programmnamen aufführen. Bei dieser Eingabe spricht man von einer Kommandozeile. Der Programmablauf kann dann an die übergebenen Parameter angepasst werden. Ein Beispiel dafür ist der FreeBASIC-Compiler: Beim Aufruf werden die benötigten Informationen – insb. welche Datei(en) kompiliert werden soll(en) – übergeben. Auch wenn Sie Ihre Programme ausschließlich über die IDE compilieren, nutzen Sie damit bereits intensiv den Vorteil der Kommandozeilenparameter, denn die IDE teilt dem Compiler über die Kommandozeile alle nötigen Informationen zum Compiliervorgang mit.

Die Kommandozeilenparameter sind natürlich nur dann sinnvoll, wenn sie vom aufgerufenen Programm auch abgefragt werden können. In FreeBASIC kann das über die Funktion **`COMMAND()`** erfolgen. **`COMMAND()`** kann mit und ohne Parameter aufgerufen werden, wobei sich die Bedeutung etwas unterscheidet.

- `COMMAND` (ohne Parameter) liefert einen String mit allen Kommandozeilenparametern. Dasselbe geschieht bei Übergabe eines negativen Parameters.
- `COMMAND(0)` gibt den im Aufruf angegebenen Pfad und Dateinamen zurück.

- `COMMAND(i)` gibt für einen positven Wert `i` das i-te Argument der Kommandozeilenparameter zurück.

Die Anzahl der übergebenen Argumente kann auch mit dem Symbol **__FB_ARGC__** abgefragt werden. **__FB_ARGC__** beginnt mit der Zählung beim Argument `0` (dem Programmnamen), d. h. wenn Sie neben dem Programmnamen noch weitere drei Argumente übergeben, hat **__FB_ARGC__** den Wert `4`.

Das folgende Beispiel besteht aus zwei Programmen: das erste ruft das zweite mit einer Argumentenliste auf, die vom zweiten der Reihe nach ausgewertet werden. Beide Programme müssen in compilierter Form im selben Ordner vorliegen, und der Name des zweiten Programmes muss mit dem Namen im **SHELL**-Aufruf des ersten Programmes übereinstimmen.

Hinweis: Wenn Sie im Dateinamen des übergebenen Programmes die Dateierweiterung weglassen, versucht Windows zunächst, eine Datei dieses Namens ohne Dateiendung zu finden. Scheitert dies, wird nach einer Datei mit passender Endung wie `.exe` oder `.bat` gesucht. Unter Linux besitzen Programme in der Regel keine Endung, und auch die von `fbc` erzeugten Programme sind dort ohne Dateiendung. Quelltext 17.10 verwendet deshalb ebenfalls keine Erweiterung, sodass die Quellcodes auf beiden Systemen funktionieren. Beachten Sie in Quelltext 17.10 außerdem die Pfadangabe "./" vor dem Namen des aufzurufenden Programmes. In Linux-Systemen können ausführbare Dateien, die nicht in einem der speziellen Systemordner liegen, aus Sicherheitsgründen nur mit Pfadangabe aufgerufen werden (so wird verhindert, dass ein Systemprogramm von einem lokalen Script gleichen Namens überlagert wird). Der Pfad "./" weist explizit darauf hin, dass sich die ausführbare Datei im aktuellen Arbeitsverzeichnis befindet.

Quelltext 17.10: COMMAND() - aufrufendes Programm

```
' Programm "shell_1.bas"
' ruft das Programm "shell_2" auf
DIM AS INTEGER rueckgabe

rueckgabe = SHELL("./shell_2 shell param1 param2 endParam")
PRINT "Rueckgabe des SHELL-Befehl: " & rueckgabe
PRINT
rueckgabe = EXEC("./shell_2", "exec param1 param2 endParam")
PRINT "Rueckgabe des EXEC-Befehl: " & rueckgabe

SLEEP
```

Quelltext 17.11: COMMAND() - aufgerufenes Programm

```
' Programm "shell_2.bas"
' wird von "shell_1" aufgerufen
PRINT "========================="
PRINT "Programmstart von shell_2"
SELECT CASE COMMAND(1)
  CASE "shell"
    PRINT "gestartet durch SHELL"
  CASE "exec"
    PRINT "gestartet durch EXEC"
END SELECT

PRINT "Kommandozeile: " & COMMAND
PRINT "Uebergebene Parameter:"
FOR i AS INTEGER = 0 TO __FB_ARGC__-1
  PRINT i & ". Argument: " & COMMAND(i)
NEXT
PRINT "========================="

END 123      ' mit Errorlevel 123 beenden
```

Wie Sie sehen, können Sie den Programmablauf an den Inhalt der übergebenen Argumente anpassen. Dies geschieht in Quelltext 17.11 in den Zeilen 5-10, auch wenn die dort verwendete Unterscheidung äußerst einfach ist.

Achtung:
Als Errorlevel, das als Parameter zu **END** angegeben wird, kann zwar ein **INTEGER** gewählt werden, unter Linux wird aber nur ein Wert von `-1` bis `254` verwendet, da hier nur ein **UBYTE** weitergereicht wird. Sie sollten daher auf andere Errorlevel verzichten.

17.5. Umgebungsvariablen abfragen und setzen

ENVIRON() erlaubt es, eine Umgebungsvariable abzufragen. Beispiele für solche Umgebungsvariablen sind die Variable `PATH`, die eine Liste aller Pfade enthält, in denen nach ausführbaren Programmen gesucht wird, oder die Variable `HOME` (Linux) bzw. `HOMEPATH` (Windows) mit dem Pfad zum persönlichen Verzeichnis des aktuellen Nutzers. Diese Variablen sind „global“ in dem Sinne, dass sie im gesamten Betriebssystem zur Verfügung stehen. Allerdings wird in der Regel für jeden Prozess eine eigene Kopie dieser Umgebungsvariablen angelegt. Eine Änderung wirkt sich damit nur für den eigenen Prozess und für von diesem Prozess gestartete Unterprozesse aus. Ändern können Sie eine

Umgebungsvariable (lokal) mit **SETENVIRON**. Damit können Sie aber auch eigene Umgebungsvariablen anlegen, die dann z. B. in der Befehlszeile von **SHELL()** genutzt werden können. Auch FreeBASIC selbst legt innerhalb seiner Programme Umgebungsvariablen an, etwa den verwendeten Grafiktreiber.

SETENVIRON erwartet einen String in der Form:

```
SETENVIRON "Variable=Wert"
```

Wenn Sie die Umgebungsvariable `Variable` in **SHELL** aufrufen wollen, verwenden Sie unter Linux `$Variable` und unter Windows `%Variable%` (dazu gleich noch ein Beispiel). Beachten Sie auch hier, dass die Angaben unter Linux *case sensitive* sind.

Quelltext 17.12 funktioniert aufgrund der gewählten Variablen- und Pfadnamen nur unter Windows, kann aber problemlos an Linux-Systeme angepasst werden.

Quelltext 17.12: ENVIRON() und SETENVIRON

```
' Benutzerverzeichnis anzeigen
PRINT "Homeverzeichnis: " & ENVIRON("HOMEPATH")

' Datenordner des Programmes zu PATH hinzufuegen
DIM AS STRING meinOrdner = ENVIRON("APPDATA") & "\meinProgramm"
SETENVIRON "PATH=" & ENVIRON("PATH") & ";" & meinOrdner
PRINT ENVIRON("PATH")

' neue Variable anlegen und mit SHELL nutzen
SETENVIRON("meinPfad=C:\Users")
SHELL "cd %meinPfad% && dir"
SLEEP
```

Unterschiede zu QuickBASIC:
In QuickBASIC dient **ENVIRON** sowohl zum Abfragen als auch zum Setzen einer Umgebungsvariablen. In FreeBASIC kann mit **ENVIRON()** nur abgefragt werden.

17.6. Fragen zum Kapitel

1. Wodurch kann man eine absolute und eine relative Pfadangabe voneinander unterscheiden?
2. Sie wollen mit **NAME** eine Datei verschieben und erhalten den Rückgabewert `-1`. Welche Gründe kann das haben?

3. Erweitern Sie Quelltext 17.8, sodass es auch versteckte Dateien anzeigt und entsprechend markiert (z. B. durch Änderung der Schriftfarbe in grau). Außerdem soll zu jeder Datei die Dateigröße angezeigt werden.

4. Schreiben Sie ein Programm, das einem anderen Programm über die Befehlszeile Parameter übergibt, die dort auf sinnvolle Weise ausgewertet werden. Zu dieser Aufgabe gibt es keine Lösung im Buch; Sie können sich jedoch an Quelltext 17.10 und Quelltext 17.11 orientieren.

18. Datum und Zeit

Dieses Kapitel teilt sich in drei Abschnitte: Zeitmessung innerhalb des Programms, einfache Befehle zum Abrufen und Setzen der Systemzeit und – als umfangreichster Abschnitt – der Umgang mit dem FreeBASIC-internen Datumsformat.

18.1. Zeitmessung

18.1.1. SLEEP

Einen sehr einfachen Befehl zur zeitlichen Kontrolle haben wir bereits kennen gelernt: **SLEEP** erlaubt es, das Programm für eine festgelegte Anzahl an Millisekunden anzuhalten.

```
SLEEP           ' wartet auf Tastendruck
SLEEP 3000      ' wartet 3 Sekunden; durch Tastendruck abbrechbar
SLEEP 3000, 1   ' wartet 3 Sekunden; nicht durch Tastendruck abbrechbar
```

Nachteil von **SLEEP** ist, dass die Programmausführung angehalten wird, also nicht anderweitig auf Benutzereingaben reagiert werden kann. Oft ist es dagegen das Ziel, das Programm weiterlaufen zu lassen und trotzdem nach einer festgelegter Zeit mit einer bestimmten Aktion zu reagieren. Hierzu ist **SLEEP** nicht geeignet.

Eine weitere Sache, über die man sich im Klaren sein sollte: Auch wenn **SLEEP** eine auf Millisekunden genaue Angabe erlaubt, arbeiten die Betriebssysteme gar nicht mit einer so hohen zeitlichen Auflösung. Wie genau der Zeitraum gemessen wird, hängt vom System ab. Als grobe Richtlinie können Sie unter Linux mit 10 ms und unter Windows mit 15 ms rechnen (unter älteren Windowssystemen sogar 50 ms und unter DOS 55 ms). Das bedeutet, dass ein `SLEEP 1` unter Linux etwas kürzer wartet als unter Windows. Für Präzessionsmessungen ist der Befehl nicht gedacht.

Um eben einmal schnell und unkompliziert eine Pause einzufügen, eignet sich **SLEEP** dagegen schon. Der wichtigste Einsatzbereich ist allerdings die Unterbrechung innerhalb einer dauerhaft ausgeführten Schleife (vgl. Kapitel 11.6).

18.1.2. TIMER()

Eine deutlich zielgenauere Zeitkontrolle kann mit **TIMER()** erreicht werden. Die Funktion gibt ein **DOUBLE** zurück, welches die Anzahl der seit einem bestimmten Referenzpunkt vergangenen Sekunden angibt. Welcher Zeitpunkt den Start der Messung festlegt, hängt vom Betriebssystem ab. Unter Linux beginnt die Messung mit dem 01.01.1970, 00:00 Uhr (*Unix-Epoche*), unter Windows mit dem Systemstart – wobei ein „reguläres Herunterfahren“ nicht notwendigerweise zu einem Neustart führt. Allerdings ist der genaue Startzeitpunkt gar nicht entscheidend. Interessant ist **TIMER()** deshalb, weil damit die Zeitdauer zwischen zwei Code-Abschnitten gemessen werden kann.

Quelltext 18.1: Beispiele für den Einsatz von TIMER()

```
DIM AS DOUBLE start

' LOCATE und PRINT sind recht zeitaufwaendige Befehle - Messung der Dauer:
PRINT "Messung der Dauer von LOCATE und PRINT (10 000 Durchlaeufe)"
start = TIMER                    ' aktuellen TIMER-Stand merken
FOR i AS INTEGER = 1 TO 10000
  LOCATE 2, 1
  PRINT "Durchlauf"; i
NEXT
PRINT "Vergangene Zeit: ";
PRINT TIMER - start; " Sekunden"  ' Abstand zwischen "Jetzt" und "Vorhin"

PRINT "Das Programm endet in drei Sekunden."
start = TIMER
DO
  SLEEP 1
LOOP UNTIL TIMER > start + 3      ' Schleifenende, wenn 3 Sekunden vergangen
```

Im zweiten Programmteil soll die Ausführung nicht drei Sekunden lang den Prozess blockieren. Daher wird das `SLEEP 1` eingesetzt. Die Ausführungsgeschwindigkeit des zweiten Teils wird dadurch nicht beeinträchtigt, denn die Schleife läuft ja sowieso drei Sekunden lang – ob da noch ein paar Millisekunden dazu kommen oder nicht, spielt nun wirklich keine Rolle.

Etwas anders sieht die Situation im ersten Programmteil aus. Auch hier nimmt die Ausführung einige Zeit in Anspruch (in meinem Test waren es etwa 1,5 Sekunden, was für einen Prozessor eine sehr lange Zeit ist). Deshalb sollte auch in dieser Schleife ein `SLEEP 1` eingefügt werden. Probieren Sie es aus! Sie werden allerdings wahrscheinlich feststellen, dass die Programmausführung nun wesentlich länger dauert. Das ist auch kein Wunder: Immerhin wird die Ausführung des ersten Teils nun zehntausend Mal unterbrochen.

In dieser Situation würde sich ein Mittelweg anbieten, etwa dass `SLEEP 1` nur bei jedem hundertsten Durchlauf eingesetzt wird. Das ist mittels Modulo-Rechnung tatsäch-

lich möglich. Dazu benötigt die Schleife nur eine zusätzliche bedingte Anweisung, die ausgeführt wird, wenn sich die Laufvariable ohne Rest durch 100 teilen lässt. Natürlich verbraucht auch diese bedingte Anweisung etwas Rechenzeit, allerdings ist diese im Vergleich zu den (wirklich langsamen) Anweisungen **LOCATE** und **PRINT** vernachlässigbar.

Quelltext 18.2: SLEEP 1 nur jeden hundertsten Durchlauf

```
DIM AS DOUBLE start

' LOCATE und PRINT sind recht zeitaufwaendige Befehle - Messung der Dauer:
PRINT "Messung der Dauer von LOCATE und PRINT (10 000 Durchlaeufe)"
start = TIMER                    ' aktuellen TIMER-Stand merken
FOR i AS INTEGER = 1 TO 10000
  LOCATE 2, 1
  PRINT "Durchlauf"; i
  IF (i MOD 100) = 0 THEN SLEEP 1 ' jeden hundertsten Durchlauf
NEXT
PRINT "Vergangene Zeit: ";
PRINT TIMER - start; " Sekunden"  ' Abstand zwischen "Jetzt" und "Vorhin"
SLEEP
```

Experimentieren Sie mit den Werten etwas herum und verändern Sie ggf. auch die Anzahl der Schleifendurchläufe – die gemessenen Zeitintervalle hängen auch entscheidend von der Leistung Ihres Computers ab. Außerdem dürfen Sie die sonstigen laufenden Prozesse in Ihrem System nicht vergessen! Die hier behandelte Messung kann lediglich als Orientierung dienen; die Ausführungsgeschwindigkeit hängt auch sehr stark von der sonstigen Auslastung Ihres Systems ab.

Unterschiede zwischen den Betriebssystemen:
Auf manchen, insbesondere älteren, Plattformen wird **TIMER()** um Mitternacht auf 0 zurückgesetzt. Wenn diese Rücksetzung zwischen Start- und Endpunkt der Zeitmessung stattfindet, ist die berechnete Differenz zwischen beiden Zeitpunkten negativ. Das Programm kann dadurch ein unerwartetes Verhalten aufweisen. Als Lösung dieses Problems kann bei negativer Differenz 86400 addiert werden – dies entspricht der Anzahl der Sekunden eines Tages.

18.2. Abruf und Änderung der Systemzeit

Mit den Funktionen **TIME()** und **DATE()** erhalten Sie einen String mit der aktuellen Systemzeit bzw. dem aktuellen Systemdatum. Beide haben ein fest vorgegebenes For-

mat: Die Systemzeit wird im Format `"hh:mm:ss"` zurückgegeben, das Systemdatum im Format `"mm-dd-yyyy"`. Falls Ihnen diese Schreibweise unbekannt sein sollte: Die Buchstaben bedeuten

- im Zeitformat: `h` = Stunde (*hour*), `m` = Minute, `s` = Sekunde
  `"hh"` steht damit für eine Angabe der Stunde mit zwei Ziffern, also ggf. mit führender Null. Das Format `"h"` wäre dagegen eine Stundenanzeige ohne führende Null (bei Werten größer als 9 werden aber selbstverständlich zwei Stellen angezeigt).
- im Datumsformat: `d` = Tag (*day*), `m` = Monat, `y` = Jahr
  Beachten Sie hier die amerikanische Formatierung, in der die Monatsangabe vor der Angabe des Tages steht.

Die Bedeutung des Buchstaben `m` muss sich hier leider aus dem Kontext erschließen. Der Kontext mag Ihnen hier vollkommen klar sein – für den Computer ist das nicht unbedingt so einfach. Wie FreeBASIC mit dem Problem umgeht, beleuchten wir in Kapitel 18.3.1. Dort werden wir uns eine deutlich flexiblere Möglichkeit ansehen, Datum und Zeit anzuzeigen.

```
PRINT "Heute ist der "; DATE; ". Es ist " & TIME & " Uhr."
SLEEP
```

Um Systemzeit und -datum neu zu setzen, verwenden Sie **`SETTIME`** und **`SETDATE`**. Die Zeitangabe muss in einem der Formate `"hh:mm:ss"` oder `"hh:mm"` oder `"hh"` sein, die Datumsangabe in einem der Formate `"mm/dd/yyyy"`, `"mm-dd-yyyy"`, `"mm/dd/yy"` oder `"mm-dd-yy"`. Beachten Sie aber dabei, dass das Betriebssystem möglicherweise keine Änderung zulässt. Ich empfehle, auf den Einsatz dieser beiden Befehle zu verzichten.

18.3. Serial Numbers

`TIME()` und **`DATE()`** sind zum einen in ihrem Format sehr unflexibel, und um bestimmte Informationen wie etwa den aktuellen Tag des Monats zu extrahieren, muss zusätzlich mit Teilstrings gearbeitet werden. Zum anderen lässt sich damit nur der aktuelle Zeitpunkt bestimmen. Für weiter reichende Datumsberechnungen sind die beiden Funktionen nicht geeignet. Stattdessen werden Zeitangaben in FreeBASIC über *Serial Numbers* realisiert.

Eine Serial Number ist, kurz gesagt, ein **`DOUBLE`**, dessen ganzzahliger Anteil die Anzahl der Tage angibt, die seit dem 30.12.1899 (00:00 Uhr) vergangen sind. Der Nachkommaanteil gibt den entsprechenden Bruchteil des Tages an – so bedeutet der Nachkommateil `.5` einen halben Tag bzw. 12 Stunden und der Nachkommateil `.25` einen viertel Tag bzw.

6 Stunden. Negative Zahlen geben dann natürlich Zeitpunkte an, die vor dem 30.12.1899 liegen.

Das Format der *Serial Numbers* bringt einige Vorteile mit sich. Interessiert man sich nur für das Datum, muss lediglich der ganzzahlige Anteil der Zahl betrachtet werden. Die (ganzzahlige) Differenz zweier *Serial Numbers* ist zugleich die Differenz beider Daten in Tagen. Umgekehrt interessiert für die repräsentierte Uhrzeit lediglich der Nachkommaanteil. Ein nicht zu ignorierender Nachteil ist dagegen die eingeschränkte Genauigkeit von Gleitkommazahlen: Bei sehr großen Datumswerten leidet die Genauigkeit der Nachkommastellen. Allerdings werden Sie bei einem Datum innerhalb des „normalen" Anwendungsbereichs nicht an diese Grenzen stoßen. Um etwa eine Genauigkeit im Sekundenbereich zu verlieren, müssten Sie mehrere Millionen Jahre in die Zukunft oder in die Vergangenheit rechnen.

FreeBASIC stellt eine Reihe an Funktionen zur Verfügung, um bequem mit *Serial Numbers* umzugehen. Um diese Funktionen nutzen zu können, müssen Sie mittels **#INCLUDE** die Datei *datetime.bi* einbinden, oder aber wieder *vbcompat.bi*, welche unter anderem auch *datetime.bi* einbindet.

18.3.1. Aktuelles Datum mit NOW() und FORMAT

Die einfachste Funktion in Bezug auf *Serial Numbers* ist **NOW()**. Die Funktion gibt eine *Serial Number* zurück, die den aktuellen Zeitpunkt repräsentieren.

```
#INCLUDE "vbcompat.bi"
PRINT "Aktueller Zeitpunkt: "; NOW
SLEEP
```

Ausgabe

```
Aktueller Zeitpunkt:  44056.55517361111
```

Das ist jetzt noch nicht besonders hilfreich. Glücklicherweise lässt sich der Wert auf recht einfache Weise in ein für Menschen lesbares Format übertragen. Wir greifen dazu auf die Funktion **FORMAT()** zurück, die auf vielfältige Weise zur Formatierung von Zahlenwerten eingesetzt werden kann. Bevor wir ins Detail gehen, hier ein paar Beispiele:

Quelltext 18.3: Aktueller Zeitpunkt mit NOW() und FORMAT()

```
#INCLUDE "vbcompat.bi"
DIM AS DOUBLE zeitpunkt = NOW
PRINT "Heute ist der " & FORMAT(zeitpunkt, "d.m.yyyy")
PRINT "Es ist " & FORMAT(zeitpunkt, "hh:nn:ss") & " Uhr ";
PRINT "(" & FORMAT(zeitpunkt, "hh:nn:ss AM/PM)")v
PRINT "Morgen ist schon wieder " & FORMAT(zeitpunkt+1, "dddd")
SLEEP
```

Ausgabe

```
Heute ist der 13.8.2020
Es ist 14:19:27 Uhr (02:19:27 PM)
Morgen ist schon wieder Freitag
```

Wie Sie sehen, ist eine Aussage über „morgen“ sehr einfach zu treffen, indem Sie zum aktuellen Zeitpunkt lediglich `1` addieren.

Aus der kurzen Demonstration können Sie schon ein paar Details über den Formatierungsstring herauslesen. `d` wird offenbar als Tag interpretiert, `yyyy` als vierstellige Jahreszahl usw. Da `m` als Monatsnummer interpretiert wird, wurde `n` (bzw. hier `nn`) zur Anzeige der Minuten verwendet. Tatsächlich wird ein hinter einem `h` stehendes `m` ebenfalls als Minutenangabe interpretiert anstatt als Monatsnummer. Ich würde trotzdem zur Verwendung von `n` raten, damit Sie nicht in Schwierigkeiten geraten, wenn Sie einmal nur die Minuten und Sekunden anzeigen lassen wollen, ohne Stundenangabe.

Neben Datums- und Zeitformatierungen dient **`FORMAT()`** auch zu allgemeinen Zahlenformatierungen. Der Formatierungsstring wird dazu analysiert und bei Auftreten bestimmter Zeichen oder Zeichenkombinationen entsprechend interpretiert. Die Möglichkeiten sind, wie gesagt, sehr vielfältig, und sind bei Weitem nicht auf die Ausgabe von Zeit und Datum beschränkt. Eine Zusammenstellung der möglichen Formatierungsangaben finden Sie in Kapitel E.2. Auf den folgenden Seiten werden aber noch ein paar weitere Beispiele zum Einsatz von **`FORMAT()`** auftauchen.

18.3.2. Setzen einer *Serial Number*

Sie können natürlich nicht nur die *Serial Number* des aktuellen Zeitpunkts ermitteln, sondern die zu jedem beliebigen Zeitpunkt. Dazu gibt es zwei Funktionen zur Bearbeitung einer Zeitangabe und zwei Funktionen für Datumsangaben. Eine *Zeit-Serial* enthält nur die Zeitangabe ohne Datum, d. h. ihr Wert ist immer kleiner als `1`. Eine *Datums-Serial* dagegen enthält keine Zeitangabe, ist also immer ganzzahlig. Selbstverständlich können

Sie eine *Zeit-Serial* und eine *Datums-Serial* durch einfache Addition zu einer kompletten *Serial Number* kombinieren.

Zum Ermitteln einer *Zeit-Serial* gibt es folgende zwei Funktionen:

- **TIMESERIAL()** erlaubt die Übergabe dreier Parameter, welche die Stunde (0-23), Minute (0-59) und Sekunde (0-59) der gewünschten Zeit repäsentieren. Beispielsweise gibt `TIMESERIAL(1,30,0)` die *Serial Number* zur Uhrzeit 1:30:00 zurück (da es sich hierbei um das Sechzehntel eines Tages handelt, ist der Rückgabewert exakt `1/16=0.0625`).
  Die Funktion überprüft nicht den Gültigkeitsbereich der einzelnen Parameter – Sie können also durchaus versuchen, welche *Serial Number* zur Uhrzeit „0 Uhr 89 und 60 Sekunden“ gehört. In einem solchen Fall rechnet **TIMESERIAL()** mit: aus den 89 Minuten wird 1 Stunde und 29 Minuten, und aus den 60 Sekunden werden 1 Minute und 0 Sekunden.[37]

- **TIMEVALUE()** erlaubt dagegen die Übergabe eines Strings, der ein Zeitformat in der Form `"h:m:s"` enthalten muss. Hier muss es sich nun um eine korrekte Zeitangabe handeln – die Eingabe von 89 Minuten o. ä. wird nicht akzeptiert, der Rückgabewert wäre in diesem Fall `0`. Allerdings darf jedes der drei Datensegmente (Stunden, Minuten, Sekunden) vorn beliebig mit Nullen aufgefüllt werden, und vor und hinter den Datensegmenten dürfen jeweils beliebig viele Leerzeichen stehen (die Ziffern eines Datensegments müssen aber jeweils direkt beisammen stehen).

Analog dazu gibt es folgende zwei Funktionen zur Bestimmung einer *Datums-Serial*:

- **DATESERIAL()** erlaubt die Übergabe dreier Parameter, welche das Jahr, den Monat und den Tag des gewünschten Datums repäsentieren. Beispielsweise gibt `DATESERIAL(2020,2,29)` die *Serial Number* des Datums 29.02.2020 zurück. Auch hier wird notfalls passend umgerechnet: Wäre das Jahr 2020 kein Schaltjahr gewesen, wäre der 29.02. korrekt als 01.03. interpretiert worden.

- **DATEVALUE()** erlaubt die Übergabe eines Strings, der ein Zeitformat in der Form `"d.m.y"` enthalten muss. Als Trennzeichen zwischen den Datensegmenten (Tag, Monat, Jahr) kann anstelle des Punktes `.` auch der Bindestrich – oder der Slash / verwendet werden. Ansonsten gelten die gleichen Regeln wie für **TIMEVALUE()**: Jedes der drei Datensegmente darf vorn beliebig mit Nullen aufgefüllt werden, vor

[37] Durch entsprechend große bzw. durch negative Zeitangaben können Sie auf diesem Weg auch *Serial Numbers* erzeugen, die größer als `1` bzw. kleiner als `0` sind. Falls das ein Problem darstellt, müssen Sie die Parameter vor der Eingabe selbständig prüfen.

und hinter den Datensegmenten dürfen jeweils beliebig viele Leerzeichen stehen, und es muss sich um eine korrekte Datumsangabe handeln (der 29.02.2021 wäre nicht erlaubt).

Wenn Sie prüfen wollen, ob ein String eine gültige Datumsangabe enthält, können Sie **ISDATE()** verwenden. Die Funktion gibt `-1` zurück, wenn ein wie bei **DATEVALUE()** angegebenes Datumsformat voliegt, und `0`, wenn dies nicht der Fall ist.

Quelltext 18.4: TIMESERIAL() und Co.

```
#INCLUDE "vbcompat.bi"
DIM AS INTEGER stunden, minuten, sekunden
DIM AS STRING datum
DIM AS DOUBLE serial

PRINT "Geben Sie die Uhrzeit ein:"
INPUT "Stunden:  ", stunden
INPUT "Minuten:  ", minuten
INPUT "Sekunden: ", sekunden
INPUT "Geben Sie das Datum ein (dd.mm.yyyy): ", datum

IF ISDATE(datum) THEN
  serial = TIMESERIAL(stunden, minuten, sekunden) + DATEVALUE(datum)
  PRINT "Die zu den Daten gehoerige Serial Number lautet"; serial
  PRINT "Der "; FORMAT(serial, "d. mmmm yyyy, hh:mm AM/PM"); " ist ein ";
  PRINT FORMAT(serial, "dddd.")
ELSE
  PRINT "Es handelt sich nicht um eine gueltige Datumsangabe!"
END IF
```

Ausgabe

```
Geben Sie die Uhrzeit ein:
Stunden:  37
Minuten:  37
Sekunden: 11
Geben Sie das Datum ein (dd.mm.yyyy): 28  .  02  .  2020
Die zu den Daten gehörige Serial Number lautet 43890.56748842593
Der 29. Februar 2020, 01:37 PM ist ein Samstag.
```

Beachten Sie in diesem Beispiel die automatische Umwandlung der zu hohen Uhrzeitangabe. Außerdem hängt die Ausgabe der Wochen- und Monatsnamen von den Systemeinstellungen ab.

18.3.3. Teilinformationen einer *Serial Number*

Um aus einer *Serial Number* bestimmte Informationen zu extrahieren, ist **FORMAT** nicht immer die einfachste Methode. Sie können dafür auf eine große Anzahl an Funktionen mit leicht zu merkenden Namen zurückgreifen:

- **SECOND()** liefert die Sekunde,
- **MINUTE()** liefert die Minute,
- **HOUR()** liefert die Stunde,
- **DAY()** liefert den Tag des Jahres,
- **WEEKDAY()** liefert den Tag der Woche,
- **MONTH()** liefert den Monat,
- **YEAR()** liefert das Jahr

die in der *Serial Number* gespeichert sind. Der Rückgabewert ist jeweils ein **INTEGER**.

Außerdem gibt es noch **DATEPART()**, um eine beliebige Teilinformation zu extrahieren. Mit der Funktion können die zuvor genannten Funktionen ersetzt werden, aber auch weitere Informationen wie die Rückgabe der Kalenderwoche und des Quartals sind möglich.

Quelltext 18.5: Informationen aus einer Serial Number ermitteln

```
#INCLUDE "vbcompat.bi"
DIM AS DOUBLE serial = NOW
' Gesamtinformation ausgeben
PRINT "Heute ist der "; FORMAT(serial, "dd.mm.yyyy, hh:nn:ss"); " Uhr."
' Teilinformationen ausgeben
PRINT "Tag:             "; DAY(serial)      ' = DATEPART("d",    serial)
PRINT "Monat:           "; MONTH(serial)    ' = DATEPART("m",    serial)
PRINT "Jahr:            "; YEAR(serial)     ' = DATEPART("yyyy", serial)
PRINT "Stunden:         "; HOUR(serial)     ' = DATEPART("h",    serial)
PRINT "Minuten:         "; MINUTE(serial)   ' = DATEPART("n",    serial)
PRINT "Sekunden:        "; SECOND(serial)   ' = DATEPART("s",    serial)
PRINT "Tag der Woche:   "; WEEKDAY(serial)  ' = DATEPART("w",    serial)
' Weitere Teilinformationen
PRINT "Quartal:         "; DATEPART("q",  serial)
PRINT "Kalenderwoche:   "; DATEPART("ww", serial)
PRINT "Tag des Jahres:  "; DATEPART("y",  serial)
SLEEP
```

Ausgabe

```
Heute ist der 13.08.2020, 14:19:27 Uhr.
Tag:             13
Monat:           8
Jahr:            2020
Stunden:         14
Minuten:         19
Sekunden:        27
Tag der Woche:   5
Quartal:         3
Kalenderwoche:   33
Tag des Jahres:  226
```

Die durch **`WEEKDAY()`** und **`DATEPART()`** ermittelten Werte über den Tag der Woche und über die Kalenderwoche hängen davon ab, mit welchem Tag die Woche beginnt (üblich ist Sonntag oder Montag) bzw. mit welcher Woche die Zählung der Kalenderwoche beginnt. Dazu können beim Aufruf der Funktion weitere Angaben gemacht werden:

```
DATEPART (Intervall, Serial [, erster_Tag_der_Woche [, erste_Woche_des_Jahres]])
```

Intervall ist eine der in Quelltext 18.5 verwendeten Intervallangaben, *Serial* ist die *Serial Number*. Für *erster_Tag_der_Woche* kann eine der in der Datei *datetime.bi* definierten Konstanten verwendet werden:
`fbUseSystem`, `fbSunday`, `fbMonday`, `fbTuesday`, `fbWednesday`, `fbThursday`, `fbFriday`, `fbSaturday`
`fbUseSystem` verwendet das in den Systemeinstellungen festgelegte Verhalten, `fbSunday` legt den Sonntag als den ersten Tag der Woche fest, usw. Wird der Parameter ausgelassen, verwendet FreeBASIC die Systemeinstellungen.

Für *erste_Woche_des_Jahres* sind ebenfalls passende Konstanten definiert:

- `fbUseSystem`: Verwende die Systemeinstellung.
- `fbFirstJan1`: Beginne mit der Woche des ersten Januar.
- `fbFirstFourDays`: Beginne mit der ersten Woche, die mindestens vier Tage hat.
- `fbFirstFullWeek`: Beginne mit der ersten ganzen Woche des Jahres.

Auch hier wird standardmäßig das durch die Systemeinstellungen festgelegte Verhalten gewählt.

18.3.4. Namen des Wochentage und Monate

Die letzten beiden Funktionen im Zusammenhang mit dem Datum sind **WEEKDAYNAME()** und **MONTHNAME()**. Sie geben die Namen der Wochentage bzw. der Monate zurück. `WEEKDAYNAME(1)` liefert den Namen des ersten Wochentages, `WEEKDAYNAME(7)` entsprechend den Namen des letzten Wochentages. Für **MONTHNAME()** sind naürlich Parameterwerte von `1` bis `12` zulässig. Für Werte außerhalb des zugelassenen Bereichs wird ganz einfach ein Leerstring zurückgegeben.

Normalerweise liefern diese Funktionen den vollen Namen des Wochentages bzw. des Monats, also z. B. `"Donnerstag"` oder `"August"`. Beide Funktionen erlauben aber auch die Ausgabe einer Kurzform wie `"Do"` oder `"Aug"`. Dazu muss ein zweiter Parameter angegeben werden – ist dieser nicht `0`, erfolgt die Ausgabe in Kurzform. Anders gesagt: Wenn Sie die Langform erhalten wollen, müssen Sie den zweiten Parameter `0` setzen oder gleich ganz weglassen.

WEEKDAYNAME() erlaubt außerdem einen dritten Parameter, um festzulegen, mit welchem Tag die Woche beginnt. Das funktioniert genauso wie bei **DATEPART()**. Wenn also etwa als dritter Parameter der Wert `fbFriday` angegeben wird, ist Freitag der erste Tag der Woche, der zweite Tag ist Samstag und die Woche endet am Donnerstag.

Beachten Sie auch bei diesen Funktionen, dass die Ausgabe (also insbesondere die Sprache, in der die Wochentage und Monate ausgegeben werden) von den Systemeinstellungen abhängt.

Quelltext 18.6: Namen der Wochentage und Monate

```
#INCLUDE "vbcompat.bi"
' aktuellen Wochentag und Monat ermitteln
DIM AS INTEGER wochentag = WEEKDAY(NOW), monat = MONTH(NOW)
' Ausgabe der Namen
PRINT "Heute ist der"; wochentag; ". Tag der Woche, also ";
PRINT WEEKDAYNAME(wochentag); "."
PRINT "In Kurzform: "; WEEKDAYNAME(wochentag, -1)
PRINT "Es ist ein Tag im "; MONTHNAME(monat); " ("; MONTHNAME(monat, -1); ")"
' Die Woche beginnt jetzt mit Freitag
wochentag = WEEKDAY(NOW, fbFriday)
PRINT "Wuerde die Woche am Freitag beginnen, waere heute der";
PRINT wochentag; ". Wochentag."
PRINT "Es waere trotzdem "; WEEKDAYNAME(wochentag,, fbFriday); "."
SLEEP
```

Ausgabe

```
Heute ist der 5. Tag der Woche, also Donnerstag.
In Kurzform: Do
Es ist ein Tag im August (Aug)
Wuerde die Woche am Freitag beginnen, wäre heute der 7. Wochentag.
Es waere trotzdem Donnerstag.
```

Für den zweiten Parameter von **WEEKDAYNAME** bzw. **MONTHNAME** hätte statt `-1` auch jeder andere Wert ungleich `0` verwendet werden können. Ich verwende in solchen Situationen aber gern `-1`, weil es am deutlichsten den Wert *wahr* repräsentiert.

Da im unteren Codeabschnitt sowohl die Ermittlung der Wochentagsnummer als auch des Wochentagsnamens von Freitag an zu zählen beginnen, gleicht sich das im Endeffekt wieder aus, und Sie erhalten auch hier den korrekten Wochentagsnamen. In der vorletzten Zeile hätte statt des ausgelassenen mittleren Parameters natürlich auch `0` stehen können.

18.3.5. Rechnen mit Datum und Zeit

Serial Numbers können sehr einfach zum Berechnen von Zeitunterschieden und ähnlichem eingesetzt werden, allerdings halten die Zeitintervalle noch einige Schwierigkeiten bereit. So sind z. B. die Monate unterschiedlich lang, und auch bei Berechnungen mit Jahren müssen die Schaltjahre berücksichtigt werden. Glücklicherweise gibt es zwei Funktionen, welche die Rechenarbeit deutlich vereinfachen.

DATEADD() berechnet aus einer *Serial Number* ein neues Datum (wobei hier mit Datum immer die Kombination von Datum und Zeit gemeint ist), das aus der Addition eines bestimmten Zeitintervalls entsteht. Sie müssen dazu die gewünschte Intervall-Einheit sowie die Anzahl dieser Einheiten angeben.

```
neues_Datum = DATEADD(Intervall, Anzahl, altes_Datum)
```

Sie können also z. B. zum alten Datum zwei Wochen addieren, oder mit einer negativen `Anzahl` auch subtrahieren.

Um den Abstand zwischen zwei Datumsangaben zu bestimmen – also z. B. um herauszufinden, wie viele Wochen zwischen den beiden Angaben liegen – dient die Funktion **DATEDIFF()**:

```
differenz = DATEADIFF(Intervall, erstes_Datum, zweites_Datum _
                    [, erster_Tag_der_Woche [, erste_Woche_des_Jahres]])
```

erster_Tag_der_Woche und *erste_Woche_des_Jahres* haben dieselbe Bedeutung wie bei **DATEPART** in Kapitel 18.3.3. Die Angabe des Intervalls in **DATEADD()** und

DATEDIFF() ist der dort angegebenen Intervallangabe sehr ähnlich, weist jedoch kleine Unterschiede auf, die aus Tabelle 18.1 ersichtlich sind. Die Unterschiede erklären sich dadurch, dass in den drei Funktionen nicht immer alle Intervallangaben sinnvoll sind.

Intervall	DATEPART()	DATEADD()	DATEDIFF()
"yyyy"	Jahre	Jahre	Jahre
"q"	Quartale	Quartale	Quartale
"m"	Monate	Monate	Monate
"ww"	Wochen im Jahr	Wochen	Kalenderwochen
"w"	Tage in der Woche	Tage	Wochen
"d"	Tage im Monat	Tage	Tage
"y"	Tage im Jahr	Tage	Tage
"h"	Stunden	Stunden	Stunden
"n"	Minuten	Minuten	Minuten
"s"	Sekunden	Sekunden	Sekunden

Tabelle 18.1.: Befehle für binäre Speicherkopien

Bei *Quartale* handelt es sich, wie oben bereits erwähnt, um Einheiten von drei Monaten. Eine *Woche* ist eine Einheit von sieben Tagen, wogegen eine *Kalenderwoche* von der Einstellung des ersten Wochentags abhängig ist.

Ein paar Beispiele zu den beiden Funktionen:

Quelltext 18.7: DATEDIFF() und DATEADD()

```
#INCLUDE "vbcompat.bi"
DIM AS DOUBLE Feb1_19 = DATESERIAL(1900, 2, 1), Mar1_19 = DATESERIAL(1900, 3, 1)
DIM AS DOUBLE Feb1_20 = DATESERIAL(2000, 2, 1), Mar1_20 = DATESERIAL(2000, 3, 1)
PRINT "Der Februar 1900 hatte " & DATEDIFF("d", Feb1_19, Mar1_19) & " Tage (";
PRINT DATEDIFF("w", Feb1_19, Mar1_19) & " Wochen)"
PRINT "Zwischen 01.03.2020 und 01.02.2020 liegen ";
PRINT DATEDIFF("d", Mar1_20, Feb1_20) & " Tagen (";
PRINT DATEDIFF("w", Mar1_20, Feb1_20) & " Wochen)"

DIM AS DOUBLE neuesDatum19 = DATEADD("h", 100000, Feb1_19)
DIM AS DOUBLE neuesDatum20 = DATEADD("h", 100000, Feb1_20)
PRINT "100.000 Std. nach 01.02.1900 0:00 Uhr liegt ";
PRINT FORMAT(neuesDatum19, "dd.mm.yyyy, h:nn ""Uhr""")
PRINT "100.000 Std. nach 01.02.2000 0:00 Uhr liegt ";
PRINT FORMAT(neuesDatum20, "dd.mm.yyyy, h:nn ""Uhr""")
SLEEP
```

Ausgabe

```
Der Februar 1900 hatte 28 Tage (4 Wochen)
Zwischen 01.03.2020 und 01.02.2020 liegen -29 Tagen (-4 Wochen)
100.000 Std. nach 01.02.1900 0:00 Uhr liegt 30.06.1911 16:00 Uhr.
100.000 Std. nach 01.02.2000 0:00 Uhr liegt 29.06.2011 16:00 Uhr.
```

Wie Sie sehen, gibt **DATEDIFF()** einen positiven Wert zurück, wenn das zweite Datum zeitlich nach dem ersten liegt, und ansonsten einen negativen Wert. Außerdem wurden die Schaltjahre bei beiden Funktionen korrekt berücksichtigt: Die Jahre 1904, 1908, 2000, 2004 und 2008 waren Schaltjahre, nicht jedoch das Jahr 1900 (da 1900 ohne Rest durch 100, aber nicht durch 400 teilbar ist). Schaltsekunden können von FreeBASIC allerdings nicht berücksichtigt werden, da diese aufgrund der Unregelmäßigkeiten in der Rotationsgeschwindigkeit der Erde nicht vorhersagbar sind.

18.4. Fragen zum Kapitel

1. Quelltext 11.9 erzeugt einen Countdown, der durch Tastendruck vorzeitig beendet werden kann. Schreiben Sie nun einen Countdown, der nur durch die ESC-Taste abgebrochen werden kann. Ein Tastendruck soll natürlich die Ablaufgeschwindigkeit nicht verändern. Nutzen Sie daher eine **TIMER**-gesteuerte Warteschleife, innerhalb derer Sie eine Tastenabfrage mit Überprüfung der ESC-Taste einbauen.

2. Schreiben Sie ein Programm, das ausgibt, wie viele Tage, Wochen bzw. Minuten es noch bis Weihnachten dauert (25.12., 00:00 Uhr). Außerdem soll der Wochentag des 25.12. ausgegeben werden.

Teil IV.

Anhang

A. Antworten zu den Fragen

Fragen zu Kapitel 4

1. Mit dem Strichpunkt in einer **PRINT**-Anweisung können mehrere Ausdrücke aneinandergereiht werden. Sie werden hintereinander ausgegeben. Ein Strichpunkt am Ende der **PRINT**-Anweisung verhindert den Zeilenumbruch.
   Der Unterschied zwischen einem Komma und einem Strichpunkt ist bei **PRINT**, dass ein Komma die Einrückung zur nächsten Tabulatorposition bewirkt.

2. Das Komma dient bei den meisten Anweisungen zur Trennung der einzelnen Parameter.

3. Anführungszeichen werden benötigt, wenn Zeichenketten unverändert ausgegeben werden sollen. Um Variablenwerte und Ergebnisse von Berechnungen auszugeben, werden die Anführungszeichen weggelassen.

4. Erlaubt sind alle Buchstaben von a-z (Groß- und Kleinbuchstaben), Ziffern 0-9 und der Unterstrich _. Eine Variable darf jedoch nicht mit einer Ziffer beginnen.

5. „Sprechende Namen“ sind Bezeichnungen, die dem besseren Verständnis des Programmablaufs dienen. Beispielsweise kann man am „sprechenden Namen“ einer Variablen sofort erkannt werden, welche Bedeutung diese Variable besitzt.

6. **TAB** setzt den Textcursor *auf* die angegebene Position vor. **SPC** rückt den Textcursor *um* die angegebene Zahl an Stellen vorwärts.

```
COLOR 4, 14                                                    ' rot auf gelb
CLS
LOCATE 1, 15
PRINT "Mein erstes FreeBASIC-Programm"
LOCATE 2, 15
PRINT "============================="
PRINT                                                          ' Leerzeile
PRINT "Heute habe ich gelernt, wie man mit FreeBASIC Text ausgibt."
PRINT "Ich kann den Text auch in verschiedenen Farben ausgeben."
SLEEP
```

Fragen zu Kapitel 5

1. In der Anweisung **INPUT** kann der Strichpunkt nur direkt nach der Meldung stehen, die als Frage ausgegeben werden soll. In diesem Fall wird an die Frage ein zusätzliches Fragezeichen angehängt. Folgt auf die Frage ein Komma statt eines Strichpunkts, dann wird kein zusätzliches Fragezeichen ausgegeben.
   Außerdem dient das Komma zum Trennen verschiedener Variablen, falls Sie mehrere Eingaben gleichzeitig abfragen wollen. Auch der Benutzer muss dann seine Eingabe durch Kommata trennen.

2. Die Anweisung **INPUT** erwartet vom Benutzer die Eingabe einer Zeile – d. h. es werden so viele Zeichen von der Tastatur gelesen, bis Return gedrückt wird. Bis dahin hat der Benutzer die Möglichkeit, die Eingabe z. B. durch Backspace und Delete zu bearbeiten.
   Die Funktion **INPUT()** fragt eine festgelegte Anzahl an Zeichen ab. Dabei ist es egal, ob es sich um normale Zeichen oder Sondertasten wie Return, Backspace oder Pfeiltasten handelt (dabei ist zu beachten, dass eine Reihe von Sonderzeichen, z. B. die Pfeiltasten, als zwei Zeichen behandelt werden). Unter anderem gibt es also für den Benutzer keine Möglichkeit, seine Eingabe zu korrigieren oder vorzeitig zu beenden.

3. **INPUT()** wartet auf die Eingabe einer festgelegten Anzahl an Zeichen. Das Programm wird währenddessen angehalten. **INKEY()** ruft genau eine Taste aus dem Tastaturpuffer ab (dabei kann es sich auch um eine Taste handeln, die zwei Zeichen belegt), wartet jedoch nicht auf einen Tastendruck.

```
DIM AS INTEGER zahl1, zahl2
PRINT "Gib zwei Zahlen ein - ich werde sie addieren!"
INPUT "1. Zahl: ", zahl1
INPUT "2. Zahl: ", zahl2
PRINT
PRINT "Die Summe aus"; zahl1; " und"; zahl2; " ist";
PRINT zahl1 + zahl2; "."
PRINT
PRINT "Druecke eine Taste, um das Programm zu beenden."
SLEEP
```

Fragen zu Kapitel 6

1. Für Ganzzahlen im Bereich ±1 000 000 000 ist ein **LONG** oder **LONGINT** nötig (oder auch ein **INTEGER**). Vorzeichenlose Datentypen bieten sich wegen des negativen

Zahlenbereichs nicht an.
Prinzipiell könnten auch **SINGLE** und **DOUBLE** verwendet werden, dies wird jedoch für reine Ganzzahlbereichnungen aufgrund der Geschwindigkeit und (hier insbesondere im Falle von **SINGLE**) möglichen Problemen bei der Genauigkeit nicht empfohlen.[39]

2. Wenn die Speichergröße ein ausschlaggebendes Argument ist, genügt hier der Datentyp **UBYTE**. Wenn auf hohe Verarbeitungsgeschwindigkeit Wert gelegt wird, ist die Verwendung eines **INTEGER** empfehlenswert. *Möglich* ist allerdings jeder Zahlendatentyp außer **BYTE**.

3. Für Gleitkommazahlen stehen die Datentypen **SINGLE** und **DOUBLE** zur Verfügung. **DOUBLE** rechnet mit höherer Genauigkeit, belegt dafür aber auch den doppelten Speicherplatz.

4. Ein **ZSTRING** ist nullterminiert und kann daher kein Nullbyte enthalten. Ein **STRING** unterliegt dieser Einschränkung nicht. Der Vorteil von **ZSTRING** liegt in seiner Kompatibilität zu externen Bibliotheken.

5. Einer Konstanten wird bei der Deklaration ein Wert zugewiesen, der zugleich ihren Datentypen festlegt und der später nicht mehr geändert werden kann. Im Gegensatz dazu können Variablen im späteren Programmverlauf geändert werden. Daher muss die Wertzuweisung auch nicht gleich bei der Deklaration erfolgen.

6. **LONG** ist ein Datentyp mit der festen Größe 2^32. Die Größe eines **INTEGERs** ist dagegen plattformabhängig. In 32-Bit-Systemen ist ein **INTEGER** genauso groß wie ein **LONG** (wird aber dennoch als eigenständiger Datentyp behandelt).
Vorteil des **INTEGERs** gegenüber einem **LONG** ist, dass es unter jeder Plattform die schnellste Zugriffszeit besitzt. Dafür kann die variable Größe zu Problemen bei der Portierung des Programms von 32 Bit zu 64 Bit oder umgekehrt führen.

Fragen zu Kapitel 7

Der Quelltext für alle drei Aufgaben steht unten in einem gemeinsamen Programm – die einzelnen Aufgabenteile wurden entsprechend gekennzeichnet. Das Programm weist viele Dopplungen auf. Mit den Kenntnissen aus dem folgenden Kapitel können Sie seinen Umfang fast um die Hälfte reduzieren.

[39] Wenn Sie die Genauigkeitsprobleme bei **SINGLE** nachprüfen wollen, legen Sie doch einmal zwei **SINGLE**-Variablen mit den Werten 1 000 000 000 und 999 999 999 an und berechnen Sie die Differenz.

```
' Aufgabe 1: Deklaration des UDT
TYPE TProdukt
  AS STRING  name_
  AS DOUBLE  einkaufspreis, verkaufspreis
  AS INTEGER stueckzahl
END TYPE

' Aufgabe 2: Produkteingabe
DIM AS TProdukt produkt1, produkt2
PRINT "Geben Sie, durch Komma getrennt, folgende Produktinformationen ein:"
PRINT "Produktname, Einkaufspreis, Verkaufspreis, Stueckzahl"
PRINT
WITH produkt1
  INPUT "1. Produkt: ", .name_, .einkaufspreis, .verkaufspreis, .stueckzahl
  IF .einkaufspreis < 0 OR .verkaufspreis < 0 THEN
    PRINT "WARNUNG: Negative Preise sind nicht vorgesehen!"
  END IF
  IF .verkaufspreis < .einkaufspreis THEN
    PRINT "WARNUNG: Sie wollen billiger verkaufen, als Sie eingekauft haben!"
  END IF
  IF .stueckzahl < 0 THEN
    PRINT "WARNUNG: Sie koennen keine negative Anzahl besitzen!"
  END IF
END WITH
WITH produkt2
  INPUT "2. Produkt: ", .name_, .einkaufspreis, .verkaufspreis, .stueckzahl
  IF .einkaufspreis < 0 OR .verkaufspreis < 0 THEN
    PRINT "WARNUNG: Negative Preise sind nicht vorgesehen!"
  END IF
  IF .verkaufspreis < .einkaufspreis THEN
    PRINT "WARNUNG: Sie wollen billiger verkaufen, als Sie eingekauft haben!"
  END IF
  IF .stueckzahl < 0 THEN
    PRINT "WARNUNG: Sie koennen keine negative Anzahl besitzen!"
  END IF
END WITH

' Aufgabe 3: Ausgabe
WITH produkt1
  PRINT "Das Produkt " & .name_ & " erzielt einen Gewinn von ";
  PRINT (.verkaufspreis + .einkaufspreis) & " pro Stueck."
END WITH
WITH produkt2
  PRINT "Das Produkt " & .name_ & " erzielt einen Gewinn von ";
  PRINT (.verkaufspreis + .einkaufspreis) & " pro Stueck."
END WITH
```

Fragen zu Kapitel 8

1. Ein statisches Array wird, wie eine Variable, mit **DIM** und der Angabe des Datentyps deklariert. Im Unterschied zur Deklaration einer Variablen folgen auf den Array-Namen geschweifte Klammern, in denen die Dimensionen des Arrays festgelegt werden. Mögliche Deklarationen wären also
   `DIM AS STRING array1(untereGrenze TO obereGrenze)`
   `DIM array2(untereGrenze TO obereGrenze) AS INTEGER`
   Die untere Grenze muss nicht angegeben werden, wenn sie 0 sein soll:
   `DIM AS DOUBLE array3(obereGrenze)`
   Sollen die Werte des Arrays gleich initiiert werden, dann werden die Werte, die zusammen zur selben Dimension gehören, in geschweiften Klammern eingeschlossen.

2. Im Gegensatz zu statischen Arrays werden bei der Deklaration dynamischer Arrays in den Klammern hinter dem Array-Namen keine Grenzen angegeben, oder man verwendet **REDIM** statt **DIM**. Die Werte eines dynamischen Arrays können nicht sofort bei der Deklaration initiiert werden.

3. Statische Arrays besitzen eine feste Länge, während die Länge eines dynamischen Arrays im Programmverlauf verändert werden kann. Mit **ERASE** wird ein dynamisches Array gelöscht, d. h. es wird in den Zustand eines nicht dimensionierten Arrays zurückgesetzt (wobei auch jetzt die Anzahl der Dimensionen nicht mehr verändert werden kann). Bei einem statischen Array werden durch **ERASE** lediglich die Werte „auf Null“ gesetzt.

4. Ein Array, ob statisch oder dynamisch, kann maximal acht Dimensionen besitzen.

5. Wenn die bisherigen Werte erhalten bleiben sollen, ist beim Einsatz von **REDIM** das Schlüsselwort **PRESERVE** notwendig. Ohne **PRESERVE** werden die Werte des Arrays zurückgesetzt.
   Beachten Sie, dass bei einer Verkleinerung eines Arrays Werte verloren gehen. Bei einer Veränderung der unteren Grenze verschieben sich außerdem die Indizes.

6. `UBOUND(array, 0)` gibt die Anzahl der Dimensionen zurück; bei nicht dimensionierten Arrays ist das der Wert `0`. Außerdem liefert bei nichtdimensionierten Arrays `UBOUND(array)=-1` und `LBOUND(array)=0`

Fragen zu Kapitel 9

1. Variablen werden an Speicheradressen abgelegt. Pointer sind ganz einfach Zeiger auf diesen Speicherbereich. Ein Pointer enthält also nicht den Variablenwert, sondern die Adresse, unter der der Variablenwert gefunden werden kann.

2. Bisher können wir noch keine sinnvollen Anwendungen für Pointer umsetzen; dazu sind erst fundierte Kenntnisse über die Speicherverwaltung nötig. Wir werden in **??** Pointer einsetzen, um Grafikpuffer zu verwalten. Ein beliebter Einsatzbereich für Pointer ist auch die Kommunikation mit externen Bibliotheken.

3. Ganz unabhängig vom Datentyp: Ein Pointer hat immer die Größe eines **INTEGERs**, also je nach Architektur 32 Bit oder 64 Bit. Die Größe des Speicherbereichs, auf den der Pointer verweist, unterscheidet sich natürlich je nach Datentyp.

Fragen zu Kapitel 10

1. Einrückungen werden vom Compiler ignoriert und dienen nur dem Programmierer bzw. jedem, der einen Blick in den Quelltext wirft. Ziel ist eine übersichtliche Strukturierung: Alle Zeilen, die sich bei (ggf. verschachtelten) Blöcken auf derselben Ebene befinden, werden gleich weit eingerückt. Dadurch wird die Verschachtelungstiefe sofort auf einem Blick erkennbar.

2. Der Zahlenwert `0` wird als *falsch* interpretiert, jeder andere Zahlenwert als *wahr*. Strings besitzen keinen Wahrheitswert. Natürlich werden auch die **BOOLEAN** `true` als *wahr* und `false` als *falsch* interpretiert.
   Zur Auswertung von Bedingungen verwendet FreeBASIC `-1` für *wahr* und `0` für *falsch*.

3. Zeichenketten werden von links nach rechts Zeichen für Zeichen verglichen, solange bis der erste Unterschied auftritt. Entscheidend ist der ASCII-Code des verglichenen Zeichens. Das bedeutet unter anderem, dass Großbuchstaben kleiner sind als Kleinbuchstaben.

4. Ein logischer Operator vergleicht die Wahrheitswerte der übergebenen Ausdrücke und liefert einen Wahrheitswert, also `-1` oder `0` bzw. `true` oder `false`. Ein Bit-Operator vergleicht die Ausdrücke Bit für Bit, wodurch prinzipiell jeder Zahlenwert als Ergebnis in Frage kommt. Nur **ANDALSO** und **ORELSE** sind echte logische Operatoren; bei **AND**, **OR**, **XOR**, **EQV**, **IMP** und **NOT** handelt es sich um Bit-Operatoren, die unter speziellen Bedingungen wie logische Operatoren verwendet werden können.

5. `(a > 3 AND a < 8) OR (a > 12 AND a < 20)`
   Beachten Sie, dass „*zwischen* 3 und 8“ die Zahlen 3 und 8 nicht einschließt.

Zum Programmier-Auftrag will ich zwei Lösungen vorstellen; einmal nur mit **IF**:

```
DIM benutzername AS STRING, passwort AS STRING, alter AS INTEGER
INPUT "Gib deinen Namen ein:  "; benutzername
INPUT "Gib dein Passwort ein: "; passwort
INPUT "Gib noch das Alter an: "; alter
IF benutzername = "Jerry" AND passwort = "supersicher" THEN
  ' korrekte Eingabe
  IF alter < 14 THEN
    PRINT "Du bist leider noch zu jung."
  ELSEIF alter < 18 THEN
    PRINT "Du darfst weiter, wenn du versprichst, dass ein Erwachsener dabei ist."
  ELSE
    PRINT "Willkommen!"
  END IF
ELSE
  ' falsche Eingabe
  PRINT "Benutzername und/oder Passwort waren leider falsch."
END IF
SLEEP
```

Als zweites eine Altersüberprüfung mit **SELECT CASE** (nur der relevante Teil)

```
IF benutzername = "Jerry" AND passwort = "supersicher" THEN
  ' korrekte Eingabe
  SELECT CASE alter
  CASE IS < 14
    PRINT "Du bist leider noch zu jung."
  CASE 14 TO 17
    PRINT "Du darfst weiter, wenn du versprichst, dass ein Erwachsener dabei ist."
  CASE ELSE
    PRINT "Willkommen!"
  END IF
ELSE
  ' falsche Eingabe
  PRINT "Benutzername und/oder Passwort waren leider falsch."
END IF
```

Für zwei Benutzernamen mit zugehörigen Passwörtern könnte Zeile 5 folgendermaßen aussehen:

```
IF (benutzername = "Jerry" AND passwort = "supersicher") _
   OR (benutzername = "Tom" AND passwort = "strenggeheim") THEN
```

An dieser Stelle sei aber noch einmal darauf hingewiesen, dass eine Klartext-Angabe des Passworts im Quelltext nur zu Übungszwecken sinnvoll ist und keinesfalls einen sicheren Passwortschutz darstellt!

Fragen zu Kapitel 11

1. Bei einer kopfgesteuerten Schleife erfolgt die Abfrage der Lauf- oder Abbruchbedingung vor dem Schleifendurchlauf, bei einer fußgesteuerten Schleife erfolgt sie danach. Insbesondere bedeutet das, dass eine fußgesteuerte Schleife auf jeden Fall mindestens einmal durchlaufen wird.

2. Eine **FOR**-Schleife bietet sich an, wenn eine feste Anzahl an Durchgängen feststeht. Auch z. B. beim Durchlaufen eines Arrays ist sie hilfreich, da seine Länge bekannt ist (sie kann ggf. mit **LBOUND()** und **UBOUND()** bestimmt werden) und die Zählvariable als Index für den Array-Zugriff dienen kann. Eine **DO**-Schleife hat dagegen eine flexible Laufdauer und kann verwendet werden, wenn die letztendliche Anzahl der Durchläufe zu Beginn unbekannt ist.

3. Eine **DO**-Schleife ohne Lauf- und Abbruchbedingung ist eine Endlosschleife. Sie kann nur durch den Befehl `EXIT DO` verlassen werden. Allerdings spricht man auch bei Schleifen, deren Abbruchbedingung nie erfüllt werden kann (z. B. `DO UNTIL 2>3`) oder deren Laufbedingung immer erfüllt ist, von einer Endlosschleife.

4. Eine Zählvariable benötigt einen Datentyp, der um eine Schrittweite erhöht werden kann (man sagt dazu, dass sie einen Iterator besitzen). Also kommen alle Zahlendatentypen in Frage, nicht jedoch Zeichenketten.
   In **??** werden eigene Datentypen behandelt, für die ein Iterator definiert werden kann.

5. Gleitkommazahlen können bei **FOR**-Schleifen problematisch sein. Zum einen kommt es bei der Erhöhung um eine Gleitkkommazahl leicht zu Rundungsfehlern, zum anderen sollte eine Gleitkomma-Schrittweite nicht zusammen mit einer Ganzzahl-Laufvariablen verwendet werden, da die Schrittweite dann sowieso erst gerundet wird.
   Ein anderes Problem tritt auf, wenn bis an die Grenze des Wertebereichs der (Ganzzahl-)Laufvariablen gezählt werden soll. Nach der letzten Erhöhung wird dann der Wertebereich überschritten, und die Laufvariable liegt wieder am unteren Ende des Wertebereichs. Analog gilt das natürlich auch bei negativer Schrittweite, wenn bis zum unteren Ende des Wertebereichs gezählt werden soll.

6. Mit **CONTINUE DO** bzw. **CONTINUE FOR** springt das Programm an das Ende der Schleife und überprüft, ob ein weiterer Durchlauf stattfinden muss oder nicht. **EXIT DO** bzw. **EXIT FOR** springt sofort aus der Schleife heraus.
   CONTINUE existiert auch für andere Blockstrukturen, z. B. für den **SELECT**-Block.

7. Programmieraufgabe:

```
' Namenseingabe
DIM AS STRING eingabe, nachname()  ' dynamisches Array deklarieren
DIM AS INTEGER i = 0               ' Zaehlvariable fuer die Array-Laenge
PRINT "Geben Sie die Namen ein - Leereingabe beendet das Programm"
DO
  PRINT "Name"; i+1; ": ";
  INPUT "", eingabe
  IF eingabe <> "" THEN
    REDIM PRESERVE nachname(i)
    nachname(i) = eingabe
    i += 1
  END IF
LOOP UNTIL eingabe = ""
PRINT "Sie haben"; UBOUND(nachname)+1; " Namen eingegeben."

' Namensausgabe
FOR i AS INTEGER = UBOUND(nachname) TO 0 STEP -1
  PRINT nachname(i)
NEXT
SLEEP
```

Fragen zu Kapitel 12

1. **Wiederverwertbarkeit:** Ein häufig benötigter Programmteil muss nur einmal geschrieben werden und ist dann beliebig oft aufrufbar. Durch die **Verwendung von Parametern** können auch unterschiedliche Verläufe im selben Unterprogramm erreicht werden.
   Wartbarkeit: Durch die Reduktion des Quellcodes müssen Veränderungen nur noch an einer Stelle vorgenommen werden statt an mehreren Stellen gleichzeitig. Damit einher geht auch die **Verringerung der Fehleranfälligkeit.**
   Lokale Verwendung der Variablen: Da Unterprogramme ihren Speicher selbständig verwalten, müssen Sie sich (bei korrekter Umsetzung) im Unterprogramm keine Gedanken über die Variablenbelegungen im Hauptprogramm oder anderen Unterprogrammen machen.
   Rekursion: Unterprogramme können sich auch selbst rekursiv aufrufen. Damit lassen sich Probleme lösen, die auf anderem Weg deutlich schwerer zu bewältigen wären.

2. Funktionen liefern einen Rückgabewert, Prozeduren dagegen nicht. Dementsprechend bieten sich Prozeduren an, wenn Sie keinesfalls einen Rückgabewert erwarten. Umgekehrt können aber alle Prozeduren auch als Funktionen definiert werden, die

dann z. B. eine Erfolgsmeldung (oder Fehlermeldung) zurückgeben.

3. Mit **SHARED** definierte Variablen können im Hauptprogramm und allen Unterprogrammen verwendet werden – dies läuft jedoch dem Gedanken der lokalen Variablen zuwider. Die gängige Methode ist, benötigte Variablen als Parameter zu übergeben. Konstanten sind im gesamten Programm zugreifbar.

4. Parameter können **BYVAL** (als Wert) oder **BYREF** (als Referenz) übergeben werden. Ein mit **BYVAL** übergebener Wert wird in der Prozedur lokal gespeichert. Änderungen an dieser Variablen innerhalb des Unterprogramms haben keine Auswirkungen auf das Hauptprogramm. Dagegen wird mit **BYREF** eine Variable „direkt“ übergeben. In diesem Fall wirkt sich eine Veränderung der Variablen im Unterprogramm auch auf den Wert der Variablen im Hauptprogramm aus.

Fragen zu Kapitel 13

1. Zahlendatentypen werden untereinander immer implizit umgewandelt. Wenn Sie z. B. einer Ganzzahl-Variablen den Wert einer Gleitkommazahl zuweisen oder umgekehrt, wird der Wert automatisch an den richtigen Datentyp angepasst. Eine weitere Form der impliziten Umwandlung ist die String-Konkatenation mit **&** – hierbei werden die Werte ggf. in Strings umgewandelt.
   Abgesehen von der String-Konkatenation findet jedoch keine automatische Umwandlung zwischen Strings und Zahlenwerten statt. Wenn Sie etwa mit den in Zeichenketten vorliegenden Zahlenwerten rechnen oder einen Zahlenwert als String speichern wollen, müssen Sie die notwendigen Umwandlungen explizit durchführen.

2. ASCII ist eine standardisierte Zeichencode-Tabelle mit 128 Einträgen (die Werte von 0 bis 127). In einem ASCII-codierten Text können Sie sicher sein, dass ein bestimmter Zahlenwert immer dasselbe Zeichen repräsentiert. ANSI ist eine Erweiterung des ASCII-Codes auf 256 Einträge (die Werte von 0 bis 255). Während die Werte von 0 bis 127 mit den ASCII-Werten identisch sind, hängt die Bedeutung der weiteren Zeichen davon ab, welche Codierung verwendet wurde. Ein ANSI-codierter Text (der auch Nicht-ASCII-Zeichen verwendet) kann daher nur dann korrekt gelesen werden, wenn auch bekannt ist, welche Codierung verwendet wurde.

3. Unten ist ein mögliches Programm aufgeführt. Sondertasten werden aus `CHR(255)` und einem weiteren Wert zusammengesetzt. Beispiele:
   `CHR(8)`, `CHR(9)`, `CHR(13)`, `CHR(27)`: Backspace, Tab, Return, Esc
   `CHR(255, 72)`, `CHR(255, 75)`, `CHR(255, 77)`, `CHR(255, 80)`: Pfeiltasten

oben, links, rechts, unten
`CHR(255, 59)`, ... `CHR(255, 68)`: Funktionstasten F1, ... F10
`CHR(255, 71)`, `CHR(255, 79)`, `CHR(255, 73)`, `CHR(255, 81)`: Pos1, Ende, BildAuf, BildAb

```
DIM AS STRING taste
PRINT "Tastenabfrage - Beenden mit ESC"
DO
  taste = INKEY
  SELECT CASE LEN(taste)
    CASE 1 : PRINT ASC(taste)
    CASE 2 : PRINT ASC(taste), ASC(taste, 2)
  END SELECT
  SLEEP 1     ' Ich hoffe, Sie haben daran gedacht!
LOOP UNTIL taste = CHR(27)   ' Escape-Taste
```

Fragen zu Kapitel 14

1. Zufallszahl innerhalb eines Bereiches
   Für den Fall, dass der Endwert `bis` kleiner ist als der Startwert `von`, werden die beiden Werte in der Berechnungsformel einfach vertauscht.

```
FUNCTION zufallszahl(von AS INTEGER, bis AS INTEGER) AS INTEGER
  IF bis > von THEN
    RETURN INT(RND*(bis-von+1)) + von
  ELSE
    RETURN INT(RND*(von-bis+1)) + bis   ' vertauschte Werte
  END IF
END FUNCTION
```

2. Bit-Manipulation
   Die Varianten mit **BITSET()** und **BITRESET()** sind deutlich aufwändiger, allerdings muss man sich an die sehr eleganten Möglichkeiten mit **AND**, **XOR** usw. erst gewöhnen. lassen Sie sich am besten immer auch die Binärdarstellung der Zahlen ausgeben, um ein Gefühl für die Funktionsweise zu bekommen.

```
DIM AS LONG alt, neu      ' ohne Wertzuweisung; der Wert ist im Programm egal

' a) niedrigste vier Bit auf 0
' mit BITRESET
neu = alt
FOR i AS INTEGER = 0 TO 3
  neu = BITRESET(neu, i)
NEXT
' mit AND
neu = alt AND &hfffffff0  ' sehen Sie sich dazu bei Bedarf die Binaerwerte an

' b) alle Bitwerte umdrehen
' mit BIT(RE)SET
neu = alt
FOR i AS INTEGER = 0 TO 31
  neu = IIF(BIT(neu, i), BITRESET(neu, i), BITSET(neu, i))
NEXT
' mit XOR
neu = alt XOR &hffffffff
' mit NOT (sicherlich die beste Loesung fuer diese Aufgabe)
neu = NOT alt

' c) niedrigste vier Bitwerte umdrehen
' mit BIT(RE)SET
neu = alt
FOR i AS INTEGER = 0 TO 3
  neu = IIF(BIT(neu, i), BITRESET(neu, i), BITSET(neu, i))
NEXT
' mit XOR
neu = alt XOR 15
```

3. Programmieraufgabe:

```
TYPE TFormat
  AS INTEGER fett        : 1
  AS INTEGER unterstrich : 1
  AS INTEGER kursiv      : 1
END TYPE

DIM AS TFormat formatierung
formatierung.fett   = 1
formatierung.kursiv = 1

' Ist Fettschrift aktiviert?
IF formatierung.fett THEN PRINT "fett"

' Ist sowohl Fett- als auch Kursivschrift aktiviert?
IF formatierung.fett AND formatierung.kursiv THEN
  PRINT "Fett und kursiv? Uebertreiben Sie es bitte nicht!"
END IF

' Sind alle drei Formatierungsformen deaktiviert?
IF formatierung.fett = 0 AND formatierung.kursiv = 0 _
        AND formatierung.unterstrich = 0 THEN
  PRINT "Es wurde keine Formatierung gewaehlt."
END IF
SLEEP
```

Fragen zu Kapitel 15

1. Programm zur Eingabe des Namens

```
DIM AS STRING eingabe, vorname, nachname
DIM AS INTEGER suchposition
' Namenseingabe
PRINT "Geben Sie, durch Leerzeichen getrennt, Ihren Vor- und Nachnamen ein: "
INPUT "", eingabe
' letztes Leerzeichen finden (funktioniert auch bei Eingabe mehrerer Vornamen,
' solange der Nachname nicht aus mehreren Einzelwoertern besteht)
suchposition = INSTRREV(eingabe, " ")
' Eingabe aufsplitten und Ergebnis ausgeben
IF suchposition THEN                                ' Leerzeichen gefunden
  vorname  = LEFT(eingabe, suchposition-1)
  nachname = MID (eingabe, suchposition+1)
  PRINT "Vorname:  " & vorname
  PRINT "Nachname: " & nachname
ELSE
' kein Leerzeichen gefunden
  PRINT "Die Eingabe war fehlerhaft!"
END IF
SLEEP
```

2. Ersetzung eines Teilstrings

```
FUNCTION ersetzeUmlautA(eingabe AS STRING) AS STRING
  ' Bei einer Parameterbergabe BYREF kann direkt mit eingabe gearbeitet werden
  ' mir ist die Beibehaltung der originalen Uebergabe jedoch lieber.
  DIM AS STRING rueckgabe = eingabe
  DIM AS INTEGER suchposition =INSTR(rueckgabe, "ä")
  ' Solange ein Treffer erzielt wurde, ersetze und suche weiter.
  DO WHILE suchposition
    rueckgabe = LEFT(rueckgabe, suchposition-1) & "ae" _
        & MID(rueckgabe, suchposition+1)
    suchposition = INSTR(suchposition+2, rueckgabe, "ä")
  LOOP
  RETURN rueckgabe
END FUNCTION
```

3. MIRROR-Funktion

```
FUNCTION mirror(eingabe AS STRING) AS STRING
  DIM AS STRING rueckgabe = ""
  FOR i AS INTEGER = LEN(eingabe) TO 1 STEP -1
    rueckgabe &= MID(eingabe, i, 1)
  NEXT
  RETURN rueckgabe
END FUNCTION
```

Zusatzaufgabe zu Aufgabe 2: Erweitern Sie die Funktion so, dass sie noch einen Such- und einen Ersetzungsstring entgegennimmt und jedes Vorkommen des Suchstrings durch den Ersetzungsstring ersetzt.

Fragen zu Kapitel 16

1. Es gibt sequentielle Modi und binäre Modi.
   Die sequentiellen Modi unterteilen sich noch einmal auf in den Lesezugriff (**FOR INPUT**), den Schreibzugriff (**FOR OUTPUT**) und einem Schreibzugriff zum Anhängen von Daten (**FOR APPEND**). Der Lesezugriff erfordert eine bereits existierende Datei, wogegen die Schreibzugriffe bei Bedarf eine neue Datei anlegen. Beim Modus **FOR OUTPUT** werden bereits existierende Daten gelöscht.
   Als binären Modus gibt es das (starre und nicht mehr empfohlene) **FOR RANDOM** und das universell einsetzbare **FOR BINARY**. Im binären Modus kann auf die Daten an beliebiger Stelle sowohl lesend als auch schreibend zugegriffen werden.

2. Es ist guter Programmierstil, alle Ressourcen, die man selbst öffnet, auch selbst wieder zu schließen. Besonders wichtig ist das allerdings innerhalb von Unter-

programmen sowie wenn sehr viele Dateien (kurzzeitig) geöffnet werden sollen. Außerdem werden Daten in der Regel erst dann tatsächlich auf den Datenträger geschrieben, wenn die Datei geschlossen wird (was bei einem Programmabsturz zum Datenverlust führt).

3. Der einzige wirklich ungeeignete Datentyp ist **INTEGER**, wenn das Programm sowohl auf 32-Bit- als auch 64-Bit-Systemen laufen soll. Ansonsten sind alle Zahlendatentypen gut geeignet. Zeichenketten fester Länge sind einfacher zu handhaben als Zeichenketten variabler Länge, allerdings lassen sich die Schwierigkeiten hier recht einfach lösen.

4. Programmieraufgabe:

```
DIM AS STRING benutzername
DIM AS INTEGER dateiNr = FREEFILE
IF OPEN("letzerBenutzer.txt" FOR INPUT AS #dateiNr) THEN
  ' Fehler aufgetreten - Datei existiert moeglicherweise nicht
  PRINT "Dies scheint Ihr erster Besuch zu sein. Geben Sie Ihren Namen ein."
  INPUT benutzername
  ' Da die Datei nicht zum Lesen geoeffnet werden konnte, ist #dateiNr noch frei.
  OPEN "letzerBenutzer.txt" FOR OUTPUT AS #dateiNr
  PRINT #dateiNr, benutzername
  CLOSE #dateiNr
ELSE
  ' kein Fehler - Datei konnte korrekt geoeffnet werden
  LINE INPUT #dateiNr, benutzername
  CLOSE #dateiNr
  PRINT "Hallo, " & benutzername & "!"
  PRINT "Wollen Sie Ihren Namen aendern? Geben Sie den neuen Namen ein! "
  PRINT "Wenn Sie den Namen nicht aendern wollen, druecken Sie nur RETURN."
  INPUT benutzername
  IF benutzername <> "" THEN
    ' Die Datei wurde oben geschlossen; #dateiNr ist wieder frei.
    OPEN "letzerBenutzer.txt" FOR OUTPUT AS #dateiNr
    PRINT #dateiNr, benutzername
    CLOSE #dateiNr
  END IF
END IF
```

Fragen zu Kapitel 17

1. Absolute Pfadangabe beginnen unter Linux mit einem Slash /. Auch unter Windows werden Pfade, die mit einem Slash oder einem Backslash \ beginnen, als absolute Pfade ausgehend vom aktuellen Laufwerk interpretiert. Pfade, die mit einem doppelten Slash oder Backslash beginnen, werden als (absoluter) Netzwerkpfad behandelt.
   Ansonsten beginnen unter Windows absolute Pfade mit dem Laufwerksbuchstaben gefolgt von einem Doppelpunkt.

2. Der Rückgabewert `-1` bedeutet, dass das Verschieben fehlgeschlagen ist. Mögliche Gründe dafür sind, dass die Quelldatei nicht existiert, dass die Zieldatei bereits existiert oder auf den Zielpfad nicht zugegriffen werden kann, oder dass die Datei wegen fehlenden Rechten nicht verschoben werden kann.

3. Programmieraufgabe:

```
#INCLUDE "vbcompat.bi"
DIM attribute AS INTEGER, groesse AS STRING, dateiname AS STRING
dateiname = DIR("*.*", fbNormal OR fbHidden OR fbDirectory, attribute)
PRINT "Dateiname"; TAB(40); "Dateigroesse"
DO WHILE LEN(dateiname)             ' solange ein Eintrag gefunden wurde
  ' Kennzeichne versteckte Dateien durch graue Schrift
  IF attribute and fbHidden THEN
    COLOR 8
  ELSE
    COLOR 15
  END IF
  ' ermittle die Dateigroesse (Umwandlung zu einem String)
  groesse = STR(FILELEN(dateiname))
  ' Gib den Dateinamen (bzw. Ordnernamen) aus
  IF attribute and fbDirectory THEN
    PRINT "* "; dateiname
  ELSE
    PRINT "  "; dateiname; TAB(52 - LEN(groesse)); groesse
  END IF
  ' Suche nach dem naechsten Eintrag
  dateiname = DIR(attribute)
LOOP
SLEEP
```

   Weitere Ideen zum Ausbau des Codes:

   - Sortierung nach Ordner und Dateien (erst alle Ordner, dann alle Dateien)
   - Auslagerung des Codes in eine Prozedur und rekursiver Aufruf zur Anzeige aller Unterordner

Fragen zu Kapitel 18

1. Countdown:

```
DIM AS INTEGER restzeit
DIM AS DOUBLE zeit
DIM AS STRING taste

PRINT "Der Countdown laeuft! (Abbruch mit ESC)"
FOR restzeit = 10 TO 1 STEP -1
  PRINT restzeit; "..."
  zeit = TIMER
  DO
    IF INKEY = CHR(27) THEN EXIT FOR  ' INKEY, da nicht gewartet werden soll
    SLEEP 1
  LOOP UNTIL TIMER - zeit > 1         ' eine Sekunde vergangen
NEXT

IF restzeit >= 0 THEN
  PRINT "Countdown abgebrochen mit einer Restzeit von"; restzeit; " s."
ELSE
  PRINT "START!"
END IF
SLEEP
```

2. Weihnachtsberechnungen:

```
#INCLUDE "vbcompat.bi"
DIM AS INTEGER tage, wochen, minuten
DIM AS DOUBLE heute = NOW         ' heutiges Datum
' Berechne das Weihnachtsdatum dieses Jahres
DIM AS DOUBLE weihnachten = DATESERIAL(YEAR(heute), 12, 25)
' Sollte Weihnachten schon vorbei sein, addieren wir ein Jahr hinzu
IF weihnachten < heute THEN weihnachten = DATEADD("yyyy", 1, weihnachten)

' Ausgabe der gesuchten Informationen
PRINT "Bis zum naechsten Weihnachtstag dauert es noch"
PRINT DATEDIFF("d", heute, weihnachten) & " Tage, bzw."
PRINT DATEDIFF("w", heute, weihnachten) & " Wochen, bzw."
PRINT DATEDIFF("n", heute, weihnachten) & " Minuten."
PRINT "Weihnachten faellt auf einen " & WEEKDAYNAME(WEEKDAY(weihnachten))
SLEEP
```

B. Compiler-Optionen

Der Compiler *fbc* besitzt eine Vielzahl an Optionsschaltern, um das Verhalten den Wünschen nach anzupassen. Je nach verwendeter IDE können beim Compilieren die gewünschten Optionen mitgegeben werden. Wenn Sie *fbc* von der Konsole aus eufrufen, um die Datei *meinProgramm.bas* zu compilieren, lautet die Syntax

```
fbc meinProgramm.bas
```

(wobei vorausgesetzt wird, dass Sie sich im Ordner mit *meinProgramm.bas* befinden und dass der Ordner zu *fbc* bei den Systempfaden eingetragen ist[40] – ansonsten sind zusätzlich die Pfadangaben erforderlich).

Die Dateiendung *.bas* ist zwingend erforderlich. *fbc* erkennt die zu compilierende Datei ausschließlich an dieser Endung. Erzeugt wird eine ausführbare Datei namens *meinProgramm.exe* (bzw. *meinProgramm* unter Linux).

B.1. Grundlegende Compileroptionen

Wenn Sie mehrere *.bas*-Dateien angeben, werden alle zusammen zu einem gemeinsamen Programm compiliert, dessen Name durch die erste angegebene *.bas*-Datei festgelegt wird. Auf diesem Weg können Sie Ihr Projekt auf mehrere Dateien aufteilen und nur Teile dieses Projektes im Compiliervorgang einbinden.

```
fbc meinProgramm.bas zusatz1.bas zusatz2.bas
```

erzeugt aus den drei Dateien ein Programm namens *meinProgramm.exe* (bzw. *meinProgramm* unter Linux).

Mit dem Schalter `-x` lässt sich jedoch ein anderer Programmname wählen:

```
fbc -x neuesProgramm.exe meinProgramm.bas zusatz1.bas zusatz2.bas
```

[40] Bei der Installation unter Linux passiert das automatisch. Unter Windows müssen Sie dazu den Pfad in der Systemumgebungsvariablen %PATH% eintragen. Sie finden unter Windows aber im FreeBASIC-Installationsordner auch ein Programm namens *open-console.exe*. Hierbei handelt es sich um eine Konsole mit eingetragenem *fbc*-Pfad.

Neben BASIC-Dateien können weitere Ressourcen eingebunden werden:

- *.a* – Library (statische Sammlung vorcompilierter Prozeduren)
- *.o* – Objektdatei (einzelnes vorcompiliertes Modul)
- *.rc* – Ressourcendatei (Dialoge, Menüs, Icons, Tastaturkürzel; nur unter Windows)
- *.res* – compilierte Ressourcendatei (nur unter Windows)
- *.xpm* – Programmicon (nur unter Linux)

B.2. Dialekt wählen

fbc stellt verschiedene FreeBASIC-Dialekte zur Verfügung. Standardmäßig wird im Dialekt *fb* compiliert, dies kann jedoch auch explizit angegeben werden. Weitere Dialektformen sind *qb* für die beste Unterstützung alter QuickBASIC-Programme sowie *fblite* mit Unterstützung der FreeBASIC-Syntax, aber in einem QuickBASIC-ähnlichen Stil. Die letzte Dialektform lautet *deprecated*, womit Sprachkonstrukte unterstützt werden, die mit der Compiler-Version v0.17 aus dem FreeBASIC-Sprachschatz entfernt wurden. *deprecated* ist jedoch, wie der Name schon sagt, „missbilligt“ und kann möglicherweise in späteren Compiler-Versionen entfernt werden.

Die Dialektform kann mit dem Optionsschalter `-lang` angegeben werden. Die Umstellung etwa auf *fblite* läuft über den Aufruf

```
fbc -lang fblite meinProgramm.bas
```

Wenn im Quellcode die Dialektform mittels `#LANG` umgestellt wird, überschreibt dies die Wahl im Compileraufruf. Sie können die Wahl im Quelltext aber auch übergehen – dazu gibt es den Optionsschalter `-forcelang`. Noch einmal kurz zusammengefasst: *fbc* verwendet als Dialektform die über `-forcelang` eingestellte Option. Existiert diese nicht, wird die im Quelltext eingestellte Option verwendet. Existiert auch diese nicht, kommt `-lang` zum Tragen. Wenn auch das nicht angegeben wurde, verwendet *fbc* die Voreinstellung `-lang fb`.

B.3. Fehlerbehandlung einstellen

Das compilierte Programm soll nach Möglichkeit Laufzeitfehler ignorieren, da ein unkontrollierter Programmabbruch in der Regel nicht gewünscht ist. Nur in wirklich schwerwiegenden Fällen, etwa einem unerlaubten Speicherzugriff, wird das Programm abgebrochen.

Dies kann jedoch dazu führen, dass Fehler schwer aufzufinden sind. *fbc* unterstützt professionelle Fehleranalyse über Debugger, bietet aber zugleich auch einfache Fehlerbehandlungen, mit denen bereits viele der typischen Probleme erkannt werden können.

Der Optionsschalter `-e` schaltet die einfache Fehlerunterstützung ein. Dadurch werden in das fertige Programm Funktionen eingefügt, die beim Erkennen eines Fehlers zum Programmabbruch führen. Diese Fehler können dann auch mit **`ON ERROR`** abgefangen werden.
Der Optionsschalter `-ex` arbeitet wie e", jedoch mit der zusätzlichen Unterstützung von **`RESUME`**, um das Programm nach einem aufgetretenen Fehler wieder fortzusetzen.
Der Optionsschalter `-exx` arbeitet wie `-e` und `-ex`, aber mit zusätzlicher Prüfung auf gültige Array-Grenzen und korrekte Verwendung von Zeigeradressen (Nullpointer-Prüfung). Gerade während der ersten Gehversuche bei Arrays und Pointerzugriffen ist diese Funktion sehr empfehlenswert.

B.4. Unterdrüchung des Konsolenfensters

Wenn Sie unter Windows ein mit *fbc* erstelltes Programm starten, öffnet sich in aller Regel ein Konsolenfenster. Für Programme mit eigenem Grafikfenster ist das jedoch häufig störend. Um das Konsolenfenster zu unterdrücken, können Sie beim Compilieren die Option `-s gui` angeben. Unter Linux hat der Schalter keine Bedeutung – dort wird bei einem Programmstart kein gesondertes Konsolenfenster geöffnet.

Achtung: Verwenden Sie `-s gui` nur, wenn das Programm ein eigenes Grafikfenster verwendet oder automatisiert ohne Benutzereingabe abläuft (auch ohne das Warten auf einen Tastendruck am Ende des Programms). Sollte das Programm im Konsolenmodus auf eine Benutzereingabe warten, wird dies aufgrund des fehlenden Konsolenfenster zu einer Endlosschleife führen.

B.5. Weitere Optionsschalter

Es folgt eine Liste aller möglichen Optionsschalter. Die in diesem Kapitel bereits genannten werden ebenfalls noch einmal aufgeführt, sodass Sie hier eine komplette Zusammenstellung vorliegen haben.

B.5.1. Erzeugung von Bibliotheken (Libraries) mit privaten/öffentlichen Prozeduren

-export <Name> Exportiert Symbole für dynamisches Linken. Der Name einer Prozedur wird explizit für andere Module zugänglich gemacht. Ein anderes Modul kann mit Hilfe der Anweisung **DECLARE** solche öffentlich deklarierten Prozeduren benutzen.

-dll, -dylib Erzeugt eine dynamische Link-Library. So wird unter Windows eine `*.dll` (inklusive der Import-Library `*.dll.a`) bzw. unter Linux eine `*.so` erstellt.

-lib Erzeugt eine statische Library. Es wird eine Sammlung von Prozeduren vorcompiliert, die aber nicht wie bei den Optionen `-dll` und `-dylib` einem Programm separat mitgegeben, sondern direkt in das fertige Programm eingefügt wird. Dies nennt man daher statisches Linken.

[-a] <Name> Fügt eine vorcompilierte Objekt-Datei (`*.o`, `*.a`) zur Linker-Liste hinzu. Das `-a` ist optional, wenn die vorcompilierte Objektdatei die Endung `*.o` oder `*.a` besitzt.

-c Compiliert nur (erzeugt eine Objektdatei), ohne zu linken. Es werden Sammlungen von Prozeduren vorcompiliert, aber nicht zu einem ausführbaren Programm verbunden. Werden mehrere Dateien gleichzeitig oder hintereinander compiliert, so muss die Hauptdatei mit m"markiert werden.

-C Behält Objektdateien bei. Während des Compilierens werden Objektdateien mit der Endung `*.o` erzeugt und am Ende wieder gelöscht. Mit der Option `-C` werden die Objektdateien nicht gelöscht.

-l <Name> Fügt eine Library-Datei zur Linker-Liste hinzu. Eine fertige vorcompilierte statische Sammlung von Prozeduren mit dem Namen `lib<Name>.a` wird statisch dem Programm hinzugefügt. Die ausführbare Programmdatei und das Mitgeben von DLLs oder Shared Librarys kann entfallen.

B.5.2. Behandlung von Programmdateien (*.bas und *.bi)

[-b] <Name> Fügt eine Quelldatei zur Compilierung hinzu. Das `-b` ist optional. Im Allgemeinen ist diese Option nicht nötig, kann aber benutzt werden, wenn die hinzuzufügende Datei nicht die Namenserweiterung .bas besitzt oder aus einem anderen Verzeichnis stammt.

-i <Name> Fügt einen Suchpfad für include-Dateien (`*.bi`) hinzu. Wird die Option `-i` mehrfach verwendet, bestimmt die Reihenfolge ihres Auftretens die Reihenfolge, in der die Verzeichnisse durchsucht werden.

-include <Name> Gibt eine Datei an, die eingebunden wird, bevor die Quelldateien übersetzt werden. Wird die Option `-include` mehrfach verwendet, bestimmt die Reihenfolge ihres Auftretens die Reihenfolge, in der die Dateien eingebunden werden. Die Option bewirkt, dass die mit <Name> angegebenen Dateien an den Anfang des Hauptmoduls eingefügt werden, als wäre dort eine entsprechende **`INCLUDE`**-Anweisung.

-forcelang <fb | fblite | qb | deprecated> Stellt die Dialektform ein, in der compiliert werden soll. Die Option überschreibt die Anweisung **`#LANG`** im Quelltext.

-lang <fb | fblite | qb | deprecated> Stellt die Dialektform ein, in der compiliert werden soll. Diese wird von einem im Quelltext verwendeten **`#LANG`** überschrieben.

-[m | entry] <Quelldatei> Bestimmt den Haupteintrittspunkt einer Quelldatei. Das Argument ist der Name einer Quelldatei ohne ihre Erweiterung. Die Datei <Quelldatei> wird als Startdatei angenommen. Sie enthält den Programmstart Main. Wird `-m` bzw. `-entry` nicht verwendet, wird die erste angegebene Datei mit der Namenserweiterung `*.bas` als Startdatei angesehen. Wird die Option `-c` oder `-r` verwendet, muss `-m` bzw. `-entry` angegeben werden, wenn eine Hauptquelldatei compiliert wird.

-o <Name> Setzt den Namen der Ausgabedatei der compilierten Quelldatei oder Objektdatei (`.o`). `-o` muss unmittelbar nach der Quelldatei stehen. Die Option wirkt allerdings nur zusammen mit der Option `-c`, da Objektdateien normalerweise nicht gespeichert werden. Wird `-o` für eine übergebene Datei nicht benutzt, erhält die Ausgabedatei denselben Namen wie die Quelldatei, nur mit der Dateierweiterung `*.o`

Unabhängig vom Namen der Programmdateien mit der Endung `*.bas` wird ein Programm mit dem Namen `./<Name>` bzw. `<Name>.exe` erzeugt.

B.5.3. Bedingtes Compilieren und Präprozessor

-d <Name=Wert> Fügt ein Präprozessor-Makro allen Quelldateien hinzu. Die Option bewirkt dasselbe wie die Verwendung der Präprozessordirektiven **`#DEFINE`**

oder **#MACRO**. Für bedingtes Compilieren kann man einen Wert definieren, z. B. `DEBUG=1`, der dann mit **#IF** oder **#IFDEF** bzw. **#IFNDEF** abgefragt werden kann.

-pp Nur die vorcompilierte Quelldatei ausgegeben; keine komplette Compilierung.

B.5.4. Fehlerbehandlung

-e Einfache Fehlerunterstützung einschalten. Es werden Funktionen zur Fehlererkennung in das fertige Programm eingefügt, die zum Programmabbruch führen können, wenn ein schwerer Fehler auftritt.

-ex Erweiterte Fehlerunterstützung einschalten. Die Option arbeitet wie `-e`, aber mit der Möglichkeit, eigene Fehlerbehandlung zu implementieren. Mit Hilfe der Anweisung **RESUME** kann das Programm auf einen Fehler reagieren und es muss nicht zwangsläufig beendet werden.

-exx Wie `-e` und `-ex`, aber mit zusätzlicher Prüfung auf gültige Array-Grenzen und korrekte Verwendung von Zeigeradressen (Nullpointer-Prüfung).

-g Debugger-Symbole in die Ausgabedateien einfügen, die von GDB-kompatiblen Debuggern verwendet werden können. Informationen zur Fehlersuche werden in das Programm integriert. Aufgrund der zusätzlichen Informationen ist es möglich, zur Laufzeit das Programm Schritt für Schritt auf seine Funktionsweise hin zu überprüfen. Das schrittweise Ausführen von Programmen wird mit Hilfe von Debuggerprogrammen realisiert. FreeBASIC liegt der GNU Debuger *gdb* bzw. *gdb.exe* bei. Es gibt aber auch frei erhältliche grafische Oberflächen, die das Debuggen erheblich vereinfachen und die Debug-Information visuell aufbereiten.

-noerrline Eine fehlerhafte Stelle im Quellcode nicht anzeigen; nützlich, wenn eine IDE die Fehlermeldungen auswertet.

B.5.5. Programmerstellung

-arch <Typ> Setzt den Ziel-CPU-Typ. Als <Typ> sind folgende Werte möglich:

32bit x86: `386`, `486` (Standard unter x86), `586`, `686`, `athlon`, `athlon-xp`, `athlon-fx`, `k8-sse3`, `pentium-mmx`, `pentium2`, `pentium3`, `pentium4`, `pentium4-sse3`, `native`
64bit x86_64: `x86_64`, `x86-64`, `amd64`
32bit ARM: `armv6`, `armv7-a` (Standard unter ARM)
64bit ARM (AArch64): `aarch64`

weitere: `native` (um für die Architektur zu compilieren, auf der der Compiler läuft), `32`, `64` (für schnelles Cross-Compiling nach 32 Bit bzw. 64 Bit auf der Standardplattform)
`-arch` beeinflusst nur den neu generierten Code, nicht vorcompilierten Code wie die FreeBASIC Runtime-Library oder andere Libraries aus dem Ordner `lib`. Für ein erfolgreiches Cross-Compiling sind noch die zur Zielarchitektur passenden Libraries nötig.

-asm <att | intel> Setzt das verwendete ASM-Format (betrifft nur `-gen gcc`)

-fpu <X87 | SSE> Wird die Option ausgelassen oder `-fpu X87` angegeben, benutzt *fbc* die normalen 387-Assembleranweisungen für mathematische Operationen. Bei der Option `-fpu SSE` werden SSE2-Assembleranweisungen zur Berechnungen von Gleitkommavariablen benutzt. Es wird zuvor geprüft, ob der Prozessor fähig ist, SSE2 Anweisungen auszuführen. Wenn nicht wird die Option ignoriert.

-fpmode <FAST | PRECISE> Gibt die Genauigkeit von Berechnungen mit Nachkommazahlen an. Die Option ist nur in Verbindung mit `-fpu SSE` wirksam.

-gen <gas | gcc | llvm> *fbc* übersetzt den Quelltext für x86-GAS-Assembler (gas), in C für GNU C (gcc) oder für die Low Level Virtual Machine (llvm).

-r Nur Assemblerdateien mit der Endung *.asm erzeugen, nicht compilieren.

-R Während des Compilierens werden Assemblerdateien mit der Endung *.asm erzeugt und am Ende wieder gelöscht. Mit der Option `-R` werden die Assemblerdateien nicht gelöscht und können mit einem Editor eingesehen werden. Dies kann unter anderem dann nützlich sein, wenn man mit der Option `-g` Debug-Informationen erzeugt und sehen will, welche Programmanweisungen der Präprozessor generiert hat. Diese werden in den Assembler-Kommentaren abgelegt. Präprozessor-Makros in diesen Dateien werden ausgewertet.

-RR Erhalte die endgültige .ASM-Datei

-s <gui | console> Nur Windows: Gibt an, ob ein Programm im Fenster oder in der Konsole ausgeführt werden soll. Standardmäßig wird `-s console` benutzt. Wird `-s gui` angegeben, erscheint beim Programmstart kein Konsolenfenster; bei Bedarf muss ein gfx-Grafikfenster initialisiert werden.

-t <Wert> Nur Windows/DOS: Gibt die Größe des verwendeten Stackspeichers in KByte (1024 Byte-Einheiten) an (Standard: 1024 KBytes).

-target <dos | cygwin> Nur Windows: Erstellt ausführbare Dateien für andere Systeme. Die `bin`- und `lib`-Verzeichnisse müssen die Unterverzeichnisse `/dos` bzw. `/linux` der entsprechenden Distribution enthalten. Um die Option einzusetzen, um für andere Betriebssysteme zu compilieren, müssen weitere Voraussetzungen erfüllt sein.

-titel <Name> Nur XBOX: Setzt den XBE-Anzeigetitel.

-mt Erzwingt das Linken mit Thread-sicherer Laufzeitbibliothek für Applikationen mit mehreren Threads. Da normalerweise immer automatisch die Thread-sichere Version der eingebauten FreeBASIC-Threading-Funktionen verwendet werden, kommt diese Option nur bei eigenen Thread-Routinen zum Einsatz.

-nodeflibs Bindet Standard-Libraries nicht ein; alle Libraries müssen manuell mit **`#INCLIB`** geladen und die Prozeduren darin mit **`DECLARE`** deklariert werden.

-O <Wert> Setzt die Optimierungsebene; kann einen der Werte `0` (Standard), `1`, `2`, `3` oder `max` (=3) annehmen.

-p <Librarypfad> Fügt einen Pfad zum Suchen von Libraries hinzu. Standardmäßig werden Libraries im Verzeichnis der FreeBASIC-Systembibliotheken und im aktuellen Verzeichnis gesucht.

-vec <Wert> Setzt die automatische Vektorisierungsebene. Mögliche Werte sind `0` (Standard) bzw. `NONE`, `1` und `2`.

-Wa <Optionen> Übergabe von Optionen an den Assembler GAS bei der Verwendung von `-gen gas` oder `-gen llvm`. Optionen müssen durch Kommata getrennt werden.

-Wc <Optionen> Übergabe von Optionen an GCC bei Verwendung von `-gen gcc` oder `-gen llvm`. Optionen müssen durch Kommata getrennt werden.

-Wl <Optionen> Übergabe von Optionen an den Linker LD. Optionen müssen durch Kommata getrennt werden.

-x <Name> Setzt den Namen der EXE-Datei/Library einschließlich Erweiterung. Standardmäßig wird der Name der ersten Quell-Datei verwendet, die auf der Befehlszeile übergeben wurde. Beim Compilieren von Libraries muß die Datei das Präfix `lib` im Namen haben, sonst kann der Linker sie vielleicht nicht finden.

Beim getrennten Compilieren und Linken darf diese Option nur vom Linker gesetzt werden. Die Option legt demnach den Namen der Enddatei fest, also den der EXE, LIB oder DLL.

-z gosub-setjmp Benutzt setjmp/longjmp zum Implementieren von **GOSUB**

B.5.6. Informationen

-v Aktiviert den ausführlichen Modus. Der Compiler gibt dann das Zielsystem und die Architektur aus und zeigt seine Aktionen Schritt für Schritt an.

-version Zeigt die Programmversion des verwendeten Compilers an.

-w <*level* | all | none | param | escape | pedantic | next | funcptr | constness> Setzt die Mindest-Warnebene. Nur Warnungen, die mindestens das angegebene *level* besitzen, werden angezeigt. Mit Abgaben wie `param` oder `next` können zusätzliche Warnungen angezeigt werden.

-prefix <Pfad> Setzt das Compiler-Präfix (den Ort, wo der Compiler die bin-, lib- und inc-Verzeichnisse sucht). Standardmäßig ist das der Pfad, in dem sich *fbc* befindet, falls das bestimmt werden kann.

-print <fblibdir | host | target | x | sha-1> Gibt verschiedene Informationen aus wie den Library-Pfad des Compilers, den Host- bzw. das Zielsystem, den Dateinamen des erzeugten Programms oder den sha-1-Code des Compiler-Quelltextes. Nur die letzte Angabe wird angezeigt.

-profile Der Compiler erstellt nach Beendigung die Datei profile.txt, die alle Timing-Ergebnisse für Funktionsaufrufe enthält. Diese Technik nennt man auch Profiling.

-maxerr <*Anzahl* | inf> Setzt die Anzahl der Fehler, die der Compiler finden darf, bevor der Vorgang abgebrochen wird. Standard ist 10. Wenn `inf` (unendlich) angegeben wird, macht der Compiler bis zum Ende der Quelldatei weiter. Dies ist nützlich, wenn eine IDE die Fehlermeldungen auswertet.

-map <Name> Link-Map als Datei `Name` speichern

C. ASCII-Zeichentabelle

Zeichencodierung in der Konsole (Codepage 850)

0		26	→	52	4	78	N	104	h	130	é	156	£	182	Â	208	ð	234	Ô
1	☺	27	←	53	5	79	O	105	i	131	â	157	Ø	183	À	209	Ð	235	Ò
2	☻	28	∟	54	6	80	P	106	j	132	ä	158	×	184	©	210	Ê	236	ý
3	♥	29	↔	55	7	81	Q	107	k	133	à	159	ƒ	185	╣	211	Ë	237	Ý
4	♦	30	▲	56	8	82	R	108	l	134	å	160	á	186	║	212	È	238	¯
5	♣	31	▼	57	9	83	S	109	m	135	ç	161	í	187	╗	213	ı	239	´
6	♠	32		58	:	84	T	110	n	136	ê	162	ó	188	╝	214	Í	240	
7		33	!	59	;	85	U	111	o	137	ë	163	ú	189	¢	215	Î	241	±
8		34	"	60	<	86	V	112	p	138	è	164	ñ	190	¥	216	Ï	242	‗
9		35	#	61	=	87	W	113	q	139	ï	165	Ñ	191	┐	217	┘	243	¾
10		36	$	62	>	88	X	114	r	140	î	166	ª	192	└	218	┌	244	¶
11	♂	37	%	63	?	89	Y	115	s	141	ì	167	º	193	┴	219	█	245	§
12	♀	38	&	64	@	90	Z	116	t	142	Ä	168	¿	194	┬	220	▄	246	÷
13		39	'	65	A	91	[	117	u	143	Å	169	®	195	├	221	¦	247	¸
14	♫	40	(	66	B	92	\	118	v	144	É	170	¬	196	─	222	Ì	248	°
15	☼	41	)	67	C	93	]	119	w	145	æ	171	½	197	┼	223	▀	249	¨
16	►	42	*	68	D	94	^	120	x	146	Æ	172	¼	198	ã	224	Ó	250	·
17	◄	43	+	69	E	95	_	121	y	147	ô	173	¡	199	Ã	225	ß	251	¹
18	↕	44	,	70	F	96	`	122	z	148	ö	174	«	200	╚	226	Ô	252	³
19	‼	45	-	71	G	97	a	123	{	149	ò	175	»	201	╔	227	Ò	253	²
20	¶	46	.	72	H	98	b	124	¦	150	û	176	░	202	╩	228	õ	254	■
21	§	47	/	73	I	99	c	125	}	151	ù	177	▒	203	╦	229	Õ	255	
22	▬	48	0	74	J	100	d	126	~	152	ÿ	178	▓	204	╠	230	µ		
23	↨	49	1	75	K	101	e	127	⌂	153	Ö	179	│	205	═	231	Þ		
24	↑	50	2	76	L	102	f	128	Ç	154	Ü	180	┤	206	╬	232	Þ		
25	↓	51	3	77	M	103	g	129	ü	155	ø	181	Á	207	¤	233	Ú		

Zeichencodierung in einem Grafikfenster

0		26	→	52	4	78	N	104	h	130	é	156	£	182	╢	208	╨	234	Ω
1	☺	27	←	53	5	79	O	105	i	131	â	157	¥	183	╖	209	╤	235	δ
2	☻	28	∟	54	6	80	P	106	j	132	ä	158	₧	184	╕	210	╥	236	∞
3	♥	29	↔	55	7	81	Q	107	k	133	à	159	ƒ	185	╣	211	╙	237	ø
4	♦	30	▲	56	8	82	R	108	l	134	å	160	á	186	║	212	╘	238	ϵ
5	♣	31	▼	57	9	83	S	109	m	135	ç	161	í	187	╗	213	╒	239	∩
6	♠	32		58	:	84	T	110	n	136	ê	162	ó	188	╝	214	╓	240	≡
7		33	!	59	;	85	U	111	o	137	ë	163	ú	189	╜	215	╫	241	±
8		34	"	60	<	86	V	112	p	138	è	164	ñ	190	╛	216	╪	242	≥
9		35	#	61	=	87	W	113	q	139	ï	165	Ñ	191	┐	217	┘	243	≤
10		36	$	62	>	88	X	114	r	140	î	166	ª	192	└	218	┌	244	⌠
11	♂	37	%	63	?	89	Y	115	s	141	ì	167	º	193	┴	219	█	245	⌡
12	♀	38	&	64	@	90	Z	116	t	142	Ä	168	¿	194	┬	220	▄	246	÷
13		39	'	65	A	91	[	117	u	143	Å	169	⌐	195	├	221	▌	247	≈
14	♫	40	(	66	B	92	\	118	v	144	É	170	¬	196	─	222	▐	248	°
15	☼	41	)	67	C	93	]	119	w	145	æ	171	½	197	┼	223	▀	249	∙
16	►	42	*	68	D	94	^	120	x	146	Æ	172	¼	198	╞	224	α	250	·
17	◄	43	+	69	E	95	_	121	y	147	ô	173	¡	199	╟	225	ß	251	√
18	↕	44	,	70	F	96	`	122	z	148	ö	174	«	200	╚	226	Γ	252	ⁿ
19	‼	45	-	71	G	97	a	123	{	149	ò	175	»	201	╔	227	π	253	²
20	¶	46	.	72	H	98	b	124	¦	150	û	176	░	202	╩	228	Σ	254	■
21	§	47	/	73	I	99	c	125	}	151	ù	177	▒	203	╦	229	σ	255	
22	▬	48	0	74	J	100	d	126	~	152	ÿ	178	▓	204	╠	230	µ		
23	↨	49	1	75	K	101	e	127	⌂	153	Ö	179	│	205	═	231	τ		
24	↑	50	2	76	L	102	f	128	Ç	154	Ü	180	┤	206	╬	232	Φ		
25	↓	51	3	77	M	103	g	129	ü	155	Ç	181	╡	207	╧	233	Θ		

D. MULTIKEY-Scancodes

Die nachfolgende Liste enthält die Scancodes, die z. B. bei **MULTIKEY** verwendet werden. Sie entsprechen den DOS-Scancodes und funktionieren auch plattformübergreifend. Sie finden diese Liste ebenfalls in der Datei *fbgfx.bi*, die sich in Ihrem inc-Verzeichnis befinden sollte.

Die Liste führt die definierte Konstante sowie den dazu gehörigen Hexadezimal- und den Dezimalwert auf.

Konstante	hex	dez
SC_ESCAPE	01	1
SC_1	02	2
SC_2	03	3
SC_3	04	4
SC_4	05	5
SC_5	06	6
SC_6	07	7
SC_7	08	8
SC_8	09	9
SC_9	0A	10
SC_0	0B	11
SC_MINUS	0C	12
SC_EQUALS	0D	13
SC_BACKSPACE	0E	14
SC_TAB	0F	15
SC_Q	10	16
SC_W	11	17
SC_E	12	18
SC_R	13	19
SC_T	14	20
SC_Y	15	21
SC_U	16	22
SC_I	17	23
SC_O	18	24
SC_P	19	25
SC_LEFTBRACKET	1A	26
SC_RIGHTBRACKET	1B	27
SC_ENTER	1C	28
SC_CONTROL	1D	29

Konstante	hex	dez
SC_A	1E	30
SC_S	1F	31
SC_D	20	32
SC_F	21	33
SC_G	22	34
SC_H	23	35
SC_J	24	36
SC_K	25	37
SC_L	26	38
SC_SEMICOLON	27	39
SC_QUOTE	28	40
SC_TILDE	29	41
SC_LSHIFT	2A	42
SC_BACKSLASH	2B	43
SC_Z	2C	44
SC_X	2D	45
SC_C	2E	46
SC_V	2F	47
SC_B	30	48
SC_N	31	49
SC_M	32	50
SC_COMMA	33	51
SC_PERIOD	34	52
SC_SLASH	35	53
SC_RSHIFT	36	54
SC_MULTIPLY	37	55
SC_ALT	38	56
SC_SPACE	39	57
SC_CAPSLOCK	3A	58

Konstante	hex	dez
SC_F1	3B	59
SC_F2	3C	60
SC_F3	3D	61
SC_F4	3E	62
SC_F5	3F	63
SC_F6	40	64
SC_F7	41	65
SC_F8	42	66
SC_F9	43	67
SC_F10	44	68
SC_NUMLOCK	45	69
SC_SCROLLLOCK	46	70
SC_HOME	47	71
SC_UP	48	72
SC_PAGEUP	49	73
SC_LEFT	4B	75
SC_RIGHT	4D	77
SC_PLUS	4E	78
SC_END	4F	79
SC_DOWN	50	80
SC_PAGEDOWN	51	81
SC_INSERT	52	82
SC_DELETE	53	83
SC_F11	57	87
SC_F12	58	88
SC_LWIN	7D	125
SC_RWIN	7E	126
SC_MENU	7F	127

E. Konstanten und Funktionen der *vbcompat.bi*

vbcompat.bi bindet weitere Dateien ein, welche Funktionalitäten mit Kompatibilität zu Visual Basic bereit stellen:

- *datetime.bi*: Funktionen und Konstanten zur Berechnung von Datum und Zeit
- *string.bi*: Funktion **FORMAT**
- *dir.bi*: Konstanten zur Benutzung mit **DIR**
- *file.bi*: Funktionen und Konstanten zum Umgang mit Dateien

E.1. Datum und Zeit

E.1.1. Verfügbare Funktionen

DATESERIAL wandelt eine Datumsangabe in eine *Serial Number* um

DATEVALUE wandelt einen Datums-String in eine *Serial Number* um

ISDATE überprüft, ob ein String ein gültiges Datum darstellt

YEAR gibt das Jahr einer *Serial Number* zurück

MONTH gibt den Monat einer *Serial Number* zurück

DAY gibt den Tag des Jahres einer *Serial Number* zurück

WEEKDAY gibt den Tag der Woche einer *Serial Number* zurück

TIMESERIAL wandelt eine Zeitangabe in eine *Serial Number* um

TIMEVALUE wandelt einen Zeit-String in eine *Serial Number* um

HOUR gibt die Stunde einer *Serial Number* zurück

MINUTE gibt die Minute einer *Serial Number* zurück

SECOND gibt die Sekunde einer *Serial Number* zurück

NOW gibt die *Serial Number* des aktuellen Zeitpunkts zurück

DATEADD addiert ein bestimmtes Zeitintervall zu einer *Serial Number*

DATEPART gibt eine Teilinformation zu einer *Serial Number* zurück

DATEDIFF gibt die Differenz zwischen zwei *Serial Numbers* zurück

MONTHNAME gibt den Monatsnamen einer Zahl (1-12) zurück

WEEKDAYNAME gibt den Wochennamen einer Zahl (1-7) zurück

E.1.2. Definierte Konstanten

Für **WEEKDAY()**, **WEEKDAYNAME()**, **DATEPART()** und **DATEDIFF()** kann der Wochentag angegeben werden, an dem die Woche beginnt. Wird er nicht angegeben, verwendet FreeBASIC stattdessen *fbUseSystem*.

- `fbUseSystem`: Verwende das lokal eingestellte System.
- `fbSunday`, `fbMonday`, `fbTuesday`, `fbWednesday`, `fbThursday`, `fbFriday`, `fbSaturday`: Die Woche beginnt mit Sonntag, Montag, ...

Für **DATEPART()** und **DATEDIFF()** kann die Woche angegeben werden, mit der das Jahr beginnt. Wird sie nicht angegeben, verwendet FreeBASIC stattdessen *fbUseSystem*.

- `fbUseSystem`: Verwende das lokal eingestellte System.
- `fbFirstJan1`: Beginne mit der Woche des ersten Januar.
- `fbFirstFourDays`: Beginne mit der ersten Woche, die mindestens vier Tage hat.
- `fbFirstFullWeek`: Beginne mit der ersten ganzen Woche des Jahres.

Für **`DATEADD()`**, **`DATEPART()`** und **`DATEDIFF()`** muss ein Intervall angegeben werden. Die drei Funktionen verwenden weitestgehend dieselben Optionen, die sich jedoch in Einzelfällen unterscheiden.

- "yyyy": Jahre
- "q": Quartale (drei Monate)
- "m": Monate
- "ww": Wochen
  Für **`DATEPART`** und **`DATEDIFF`** sind damit die Kalenderwochen gemeint (abhängig von der Einstellung für die erste Woche des Jahres bzw. den ersten Tag der Woche).
- "w": abhängig von der Funktion.
  Für **`DATEPART`**: Tag innerhalb der Woche
  Für **`DATEDIFF`**: Sieben-Tage-Einheiten (Wochen)
  Für **`DATEADD`**: Tage
- "d": Tage
  Für **`DATEPART`** ist der Tag innerhalb des Monats gemeint.
- "y": Tage
  Für **`DATEPART`** ist der Tag innerhalb des Jahres gemeint.
- "h": Stunden
- "n": Minuten
- "s": Sekunden

E.2. Formatierungsmöglichkeiten durch FORMAT()

Die Funktion **FORMAT()** erlaubt die formatierte Ausgabe einer Zahl. Die Ausgabe wird durch die im Formatstring enthaltenen Zeichen und Zeichenfolgen festgelegt.

0	Platzhalter für eine Ziffer; Auffüllung mit führenden Nullen Nach dem Dezimaltrennzeichen werden so viele Ziffern dargestellt wie angegeben.
#	Platzhalter für eine Ziffer; ohne führenden Nullen
.	Platzhalter für ein Dezimaltrennzeichen
%	Der Ausdruck wird mit 100 multipliziert und mit einem Prozent-Zeichen % ausgegeben.
,	Platzhalter für Tausendertrennzeichen. Zwei aufeinanderfolgende Kommata bewirken das Auslassen der drei Ziffern zwischen den Kommata; die Zahl wird dabei korrekt gerundet.
E- e-	Wissenschaftliches Format; nur negatives Vorzeichen des Exponenten anzeigen
E+ e+	Wissenschaftliches Format; Vorzeichen des Exponenten immer anzeigen
: ? + $ ()	Literale; werden so ausgegeben, wie sie im Formatierungsstring stehen.
\	Nächstes Zeichen im Formatierungsstring als Literal ausgeben (also nicht interpretiert).
"Text"	Text wird so ausgegeben, wie er im Formatierungsstring steht.
/	Datumstrennzeichen zur Trennung von Tagen, Monaten und Jahren
d	Tag als Zahl ohne führende Null (0-31)
dd	Tag als Zahl mit führender Null (00-31)
ddd	Tag als Abkürzung seines Namens (So-Sa)*
dddd	Tag als vollen Namen an (Sonntag-Samstag)*
ddddd	Datum als vollständiges Datum, einschließlich Tag, Monat und Jahr*
m	Monat als Zahl ohne führende Null (1-12).**
mm	Monat als Zahl mit führender Null (01-12).**
M, MM	Monat als Zahl ohne bzw. mit führender Null
mmm	Monat als Abkürzung seines Namens (Jan-Dez)*
mmmm	Monat als vollen Namen an (Januar-Dezember)*
y oder yy	Jahr als zweistellige Zahl (00-99)
yyyy	Jahr als vierstellige Zahl (1900-2040)
h	Stunde ohne führende Null (0-23)
hh	Stunde mit führender Null (00-23)
n	Minute ohne führender Null (0-59)
nn	Minute mit führender Null (00-59)
s	Sekunde ohne führende Null (0-59)
ss	Sekunde mit führender Null (00-59)
ttttt	komplette Uhrzeit mit Stunde, Minute und Sekunde*
AM/PM am/pm	Zeit im 12-Stunden-Format mit AM bzw. am (vormittags) / PM bzw. pm (nachmittags)
A/P a/p	Zeit im 12-Stunden-Format mit A bzw. a (vormittags) / P bzw. p (nachmittags)

* Die Anzeige hängt von den Systemeinstellungen ab.

** Wenn m bzw. mm direkt auf h oder hh folgt, zeigt es stattdessen die Minuten an.

E.3. Konstanten für die Attribute von `DIR`

- `fbReadOnly`: Zeige schreibgeschützte Dateien an.
- `fbHidden`: Zeige versteckte Dateien an.
- `fbSystem`: Zeige Systemdateien an.
- `fbDirectory`: Zeige Verzeichnisse an
- `fbArchive`: Zeige archivierbare Dateien an.
- `fbNormal` = `fbReadOnly` **`OR`** `fbArchive`

E.4. Dateifunktionen

E.4.1. Verfügbare Funktionen

`FILECOPY` kopiert eine Datei

`FILEATTR` liefert Informationen über eine geöffnete Datei

`FILELEN` gibt die Länge einer Datei zurück

`FILEEXISTS` prüft, ob eine Datei existiert

`FILEDATETIME` gibt das letzte Änderungsdatum einer Datei zurück

E.4.2. Definierte Konstanten

Durch **`FILEATTR`** abfragbare Daten:
`fbFileAttrMode`, `fbFileAttrHandle`, `fbFileAttrEncoding`

Rückgabewerte von `fbFileAttrMode`:
`fbFileModeInput`, `fbFileModeOutput`, `fbFileModeRandom`,
`fbFileModeAppend`, `fbFileModeBinary`

Rückgabewerte von `fbFileAttrEncoding`:
`fbFileEncodASCII`, `fbFileEncodUTF8`, `fbFileEncodUTF16`,
`fbFileEncodUTF32`

F. Vorrangregeln (Hierarchie der Operatoren)

Mehrere Operatoren in einer Anweisung werden nach einer vordefinierten Reihenfolge abgearbeitet. Ein Operator mit höherer Hierarchie wird vor einem Operator abgearbeitet, der eine niedrigere Hierarchie besitzt. Haben zwei Operatoren die gleiche Hierarchie, werden sie in der Reihenfolge abgearbeitet, in der sie auftreten. Ist eine andere Reihenfolge bei der Abarbeitung gewünscht, kann diese durch eine Klammerung der Ausdrücke erreicht werden.

Die Arität (Stelligkeit) gibt an, wie viele Operanden der Operator besitzt. Unäre Operatoren erwarten lediglich einen Operanden (z. B. `NOT a`), während binäre Operatoren zwei Operanden erwarten (z. B. `a AND b`). Die Abarbeitung von Operatoren der gleichen Hierarchie kann von *links nach rechts* oder aber von *rechts nach links* erfolgen. Z. B. ist `a-b-c` gleichbedeutend mit `(a-b)-c` (Abarbeitung von links), während `**a` gleichbedeutend mit `*(*a)` ist (Abarbeitung von rechts). Ein *'N/A'* gibt an, dass aufgrund der Eigenschaften des Operators keine Richtung existiert.

Die folgende Tabelle enthält alle Operatoren absteigend nach ihrer Hierarchie. Die Hierarchie-Ebenen werden durch Trennstriche markiert; alle Operatoren auf derselben Ebene werden auch gleichberechtigt behandelt. Operatoren mit höherer Hierarchie (also diejenigen, die in der Tabelle weiter oben stehen) binden stärker als Operatoren niedrigerer Ebenen. Beispielsweise ist `a MOD b OR c` gleichbedeutend mit `(a MOD b) OR c` (**MOD** steht in der Hierarchie höher als **OR**).

Operator	Klassifizierung	Arität	Bedeutung	Assoz.
CAST	Funktion	unär	Typumwandlung	N/A
PROCPTR	Funktion	unär	Adressoperator	N/A
STRPTR	Funktion	unär	Adressoperator	N/A
VARPTR	Funktion	unär	Adressoperator	N/A
[]	Zugriffsoperator	/	Stringindex/Pointerindex	von links
()	Indizierung	/	Arrayindex	von links
()	Auswertungsoperator	/	Funktionsaufruf	von links
.	Zugriffsoperator	/	Strukturzugriff	von links
->	Zugriffsoperator	/	Indirektzugriff	von links

Operator	Klassifizierung	Arität	Bedeutung	Assoz.
`@`	Zugriffsoperator	unär	Adressoperator	von rechts
`*`	Zugriffsoperator	unär	Dereferenzierung	von rechts
`NEW`	Datenoperator	unär	Speicher alloziieren	von rechts
`DELETE`	Datenoperator	unär	Speicher dealloziieren	von rechts
`^`	Exponent	unär	Exponent	von links
`-`	Arithmetischer Operator	unär	Negativ	von rechts
`*`	Arithmetischer Operator	binär	Multiplizieren	von links
`/`	Arithmetischer Operator	binär	Dividieren	von links
`\`	Arithmetischer Operator	binär	Integerdivision	von links
`MOD`	Arithmetischer Operator	binär	Modulo Division	von links
`SHL`	Bitoperator	binär	Bitverschiebung nach links	von links
`SHR`	Bitoperator	binär	Bitverschiebung nach rechts	von links
`+`	Arithmetischer Operator	binär	Addieren	von links
`-`	Arithmetischer Operator	binär	Subtrahieren	von links
`&`	Verknüpfungsoperator	binär	Stringverkettung	von links
`=`	Vergleichsoperator	binär	Gleich	von links
`<>`	Vergleichsoperator	binär	Ungleich	von links
`<`	Vergleichsoperator	binär	Kleiner als	von links
`<=`	Vergleichsoperator	binär	Kleiner oder gleich	von links
`>=`	Vergleichsoperator	binär	Größer oder gleich	von links
`>`	Vergleichsoperator	binär	Größer als	von links
`NOT`	Bitweiser Operator	unär	Verneinung	von rechts
`AND`	Bitweiser Operator	binär	Und	von links
`OR`	Bitweiser Operator	binär	Oder	von links
`EQV`	Bitweiser Operator	binär	Äquivalenz	von links
`IMP`	Bitweiser Operator	binär	Implikat	von links
`XOR`	Bitweiser Operator	binär	Exklusives Oder	von links
`ANDALSO`	Logischer Operator	binär	Verkürztes und	von links
`ORELSE`	Logischer Operator	binär	Verkürztes oder	von links
`=`	Zuweisungsoperator	binär	Zuweisung	N/A
`&=`	Zuweisungsoperator	binär	Verkettung + Zuweisung	N/A
`+=`	Zuweisungsoperator	binär	Addition + Zuweisung	N/A
`-=`	Zuweisungsoperator	binär	Subtraktion + Zuweisung	N/A
`*=`	Zuweisungsoperator	binär	Multiplikation + Zuweisung	N/A
`/=`	Zuweisungsoperator	binär	Division + Zuweisung	N/A
`\=`	Zuweisungsoperator	binär	Integerdivision + Zuweisung	N/A
`^=`	Exponent	binär	Exponent + Zuweisung	von links
`MOD=`	Zuweisungsoperator	binär	Modulo + Zuweisung	N/A
`AND=`	Zuweisungsoperator	binär	Und + Zuweisung	N/A
`EQV=`	Zuweisungsoperator	binär	Äquivalenz + Zuweisung	N/A
`IMP=`	Zuweisungsoperator	binär	Implikat + Zuweisung	N/A
`OR=`	Zuweisungsoperator	binär	Oder + Zuweisung	N/A
`XOR=`	Zuweisungsoperator	binär	Exklusives oder + Zuweisung	N/A
`SHL=`	Zuweisungsoperator	binär	Bitverschiebung links + Zuw.	N/A
`SHR=`	Zuweisungsoperator	binär	Bitverschiebung rechts + Zuw.	N/A
`LET`	Zuweisungsoperator	binär	Zuweisung	N/A
`LET ()`	Zuweisungsoperator	binär	Zuweisung	N/A

G. FreeBASIC-Schlüsselwörter

Es folgt eine Liste der FreeBASIC-Schlüsselwörter mit einer kurzen Erläuterung. Genaueres zu den einzelnen Befehlen finden Sie in der FreeBASIC-Referenz. Es werden nur die Schlüsselwörter aufgeführt, die in der zur Drucklegung des Buches aktuellen Compiler-Version v1.07.0 Verwendung finden (Dialektversion `fb`) sowie die in dieser Version reservierten, aber nicht verwendbaren Schlüsselwörter.

G.1. Schlüsselwörter

ABS
gibt den Absolutbetrag der angegebenen Zahl zurück. (siehe S. 159)

ACCESS
wird zusammen mit **OPEN** verwendet und legt die Zugriffsrechte auf die Datei fest. (siehe S. 205)

ACOS
gibt den Arcuskosinus einer Zahl zurück. (siehe S. 161)

ADD
wird im Zusammenhang mit **PUT** und **DRAW STRING** eingesetzt und bestimmt die Art, wie die zu zeichnende Grafik mit den Pixeln auf dem Bildschirm interagieren sollen.

ALIAS
gibt einer Prozedur in einer Library einen neuen Namen, mit dem man auf sie verweisen kann.

ALLOCATE
reserviert eine beliebige Anzahl von Bytes im Speicher (Heap) und liefert einen Pointer zum Anfang dieses Speicherbereichs.

ALPHA
wird im Zusammenhang mit **PUT** und **DRAW STRING** eingesetzt und bestimmt die Art, wie die zu zeichnende Grafik mit den Pixeln auf dem Bildschirm interagieren sollen.

AND
als Operator: vergleicht zwei Werte Bit für Bit und setzt im Ergebnis nur dann ein Bit, wenn die entsprechenden Bits in beiden Ausdrücken gesetzt waren. (siehe S. 89, S. 91, S. 173)
als Methode: wird im Zusammenhang mit **PUT** und **DRAW STRING** eingesetzt und bestimmt die Art, wie die zu zeichnende Grafik mit den Pixeln auf dem Bildschirm interagieren sollen.

ANDALSO
prüft zwei Ausdrücke auf ihren Wahrheitsgehalt und gibt `-1` zurück, wenn beide Ausdrücke wahr sind. Ansonsten wird `0` zurückgegeben. Sind beide Operanden vom Typ **BOOLEAN**, dann ist auch das Ergebnis vom Typ **BOOLEAN**. (siehe S. 92)

ANY
wird zusammen mit **DECLARE**, **DIM**, **INSTR**, **TRIM**, **LTRIM** und **RTRIM** benutzt. Je nach Einsatzart hat es eine andere Bedeutung. (siehe S. 75, S. 135, S. 191, S. 192)

APPEND
wird zusammen mit **OPEN** verwendet, um eine Datei zu öffnen, an die weitere Daten angehängt werden sollen. (siehe S. 203)

AS
wird zusammen mit verschiedenen Befehlen verwendet, um den Datentyp einer Variablen oder eines Attributs anzugeben oder die Dateinummer einer zu öffnenden Datei festzulegen. (siehe S. 23, S. 121, S. 132, S. 201)

ASC
liefert den ASCII-Code des Zeichens. (siehe S. 154)

ASIN
gibt den Arcussinus einer Zahl zurück. (siehe S. 161)

ASM
bindet Maschinensprache-Code ins Programm ein.

ASSERT
beendet das Programm unter der angegebenen Voraussetzung und gibt eine Meldung aus.

ASSERTWARN
gibt unter der angegebenen Voraussetzung eine Meldung aus.

ATAN2
gibt den Arcustangens des Quotienten zweier Zahlen zurück. (siehe S. 161)

ATN
gibt den Arcustangens einer Zahl zurück. (siehe S. 161)

BASE
gibt im Zusammenhang mit Vererbung die Möglichkeit, auf ein Attribut oder eine Methode der Eltern-Klasse zuzugreifen, selbst wenn die eigene Klasse ein Attribut bzw. eine Methode mit gleichem Namen besitzt.

BEEP
weist das System an, ein Tonsignal auszugeben.

BIN
gibt den binären Wert eines Ausdrucks zurück. (siehe S. 91, S. 170)

BINARY
wird zusammen mit **OPEN** verwendet, um eine Datei im Binary-Modus zu öffnen. (siehe S. 204)

BIT
prüft, ob in einem Ausdruck das Bit an der angegebenen Stelle gesetzt ist. (siehe S. 176)

BITRESET
gibt den Wert eines *Ausdruck* zurück, bei dem das Bit an der angegebenen Stelle gelöscht wurde. (siehe S. 176)

BITSET
gibt den Wert eines *Ausdruck* zurück, bei dem das Bit an der angegebenen Stelle gesetzt wurde. (siehe S. 176)

BLOAD
lädt einen Block binärer Daten (z. B. Bilder) aus einer Datei.

BOOLEAN
einer der Werte `true` oder `false` (siehe S. 93)

BSAVE
speichert einen Block binärer Daten in eine Datei.

BYREF
legt fest, dass ein Parameter als direkte Referenz statt nur als Wert übergeben werden soll. (siehe S. 136)

BYTE
eine vorzeichenbehaftete 8-bit-Ganzzahl. (siehe S. 39)

BYVAL
legt fest, dass ein Parameter als Wert statt als direkte Referenz übergeben werden soll. (siehe S. 136)

CALL
veraltet; kann nicht verwendet werden.

CALLOCATE
reserviert einen Bereich im Speicher (Heap) und setzt alle seine Bytes auf `0`.

CASE
wird in Zusammenhang mit **SELECT** verwendet.

CAST
konvertiert einen Ausdruck in einen beliebigen anderen Typ. (siehe S. 149)

CBOOL
konvertiert einen numerischen Ausdruck zu einem **BOOLEAN**. (siehe S. 150)

CBYTE
konvertiert einen numerischen Ausdruck zu einem **BYTE**. (siehe S. 150)

CDBL
konvertiert einen numerischen Ausdruck zu einem **DOUBLE**. (siehe S. 150)

CDECL
setzt die Aufrufkonvention der Parameter auf `C-DECLARE` (Übergabe von rechts nach links. (siehe S. 139)

CHAIN
übergibt die Kontrolle an ein anderes Programm und startet dieses. (siehe S. 241)

CHDIR
ändert das aktuelle Arbeitsverzeichnis oder -laufwerk. (siehe S. 230)

CHR
verwandelt einen ASCII-Code in seinen Character. (siehe S. 155)

CINT
konvertiert einen numerischen Ausdruck zu einem **INTEGER**. (siehe S. 150, S. 163, S. 172)

CIRCLE
zeichnet Kreise, Ellipsen oder Bögen.

CLASS
reserviert, aber noch ohne Funktion.

CLEAR
setzt eine Anzahl an Bytes ab einer bestimmten Adresse auf einen angegebenen Wert.

CLNG
verwandelt einen numerischen Ausdruck zu einem **LONG**. (siehe S. 150)

CLNGINT
verwandelt einen numerischen Ausdruck zu einem **LONGINT**. (siehe S. 150)

CLOSE
schließt Dateien, die zuvor mit **OPEN** geöffnet wurden. (siehe S. 207)

CLS
löscht den Bildschirm und füllt ihn mit der Hintergrundfarbe. (siehe S. 30)

COLOR
als Anweisung: setzt die Vorder- und Hintergrundfarbe. (siehe S. 29)
als Funktion: gibt Informationen über die verwendeten Textfarben zurück.

COM
OPEN COM bereitet den Zugriff auf einen COM-Port vor. (siehe S. 224)

COMMAND
enthält die Kommandozeilenparameter an das Programm. (siehe S. 243)

COMMON
dimensioniert Variablen und Arrays und macht sie mehreren Modulen zugänglich.

CONDBROADCAST
sendet ein Signal an alle Threads, die auf das angegebene Handle warten, dass sie fortgesetzt werden dürfen.

CONDCREATE
erstellt eine *conditional variable* zur Synchronisation von Threads.

CONDDESTROY
zerstört eine mit **CONDCREATE** erstellte *conditional variable*.

CONDSIGNAL
sendet ein Signal an einen einzelnen Thread, dass er fortgesetzt werden kann.

CONDWAIT
wartet mit der Ausführung eines Threads, bis ein **CONDSIGNAL** oder ein **CONDBROADCAST** ein Signal zum Fortsetzen des Threads sendet.

CONS
OPEN CONS öffnet die Standardeingabe *stdin* sowie die Standardausgabe *stdout*. (siehe S. 222)

CONST
erzeugt eine Konstante. Kann auch bei der Parameterübergabe und in **SELECT CASE** verwendet werden, um die Verwendung konstanter Werte zu erzwingen. (siehe S. 53, S. 98, S. 138)

CONSTRUCTOR
erstellt einen Klassen-Konstruktor für ein UDT oder eine Prozedur, die vor Programmstart aufgerufen wird.

CONTINUE
springt in einer **DO**-, **FOR**- oder **WHILE**-Schleife an das Schleifen-Ende, wo dann die Abbruchbedingung geprüft wird. (siehe S. 112)

COS
gibt den Kosinus eines Winkels im Bogenmaß zurück. (siehe S. 160)

CPTR
verwandelt einen 32bit-Ausdruck in einen Pointer eines beliebigen Typs. (siehe S. 150)

CSHORT
konvertiert einen numerischen Ausdruck zu einem **SHORT**. (siehe S. 150)

CSIGN
verwandelt eine vorzeichenlose Zahl in eine vorzeichenbehaftete Zahl desselben Datentyps. (siehe S. 150)

CSNG
konvertiert einen numerischen Ausdruck zu einem **SINGLE**. (siehe S. 150)

CSRLIN
gibt die aktuelle Zeile des Cursors zurück. (siehe S. 28)

CUBYTE
konvertiert einen numerischen Ausdruck zu einem **UBYTE**. (siehe S. 150)

CUINT
konvertiert einen numerischen Ausdruck zu einem **UINTEGER**. (siehe S. 150)

CULNG
konvertiert einen numerischen Ausdruck zu einem **ULONG**. (siehe S. 150)

CULNGINT
konvertiert einen numerischen Ausdruck zu einem **ULONGINT**. (siehe S. 150)

CUNSG
konvertiert eine vorzeichenbehaftete Zahl in eine vorzeichenlose Zahl desselben Datentyps. (siehe S. 150)

CURDIR
gibt das aktuelle Arbeitsverzeichnis aus. (siehe S. 230)

CUSHORT
konvertiert einen numerischen Ausdruck zu einem **USHORT**. (siehe S. 150)

CUSTOM
wird im Zusammenhang mit **PUT** und **DRAW STRING** eingesetzt und bestimmt die Art, wie die zu zeichnende Grafik mit den Pixeln auf dem Bildschirm interagieren sollen.

CVA_ARG
gibt das nächste Argument einer variablen Parameterliste zurück. (siehe S. 141)

CVA_COPY
initialisiert ein Objekt für eine variable Parameterliste aus einem anderen Objekt. (siehe S. 145)

CVA_END
gibt eine Variable vom Typ CVA_LIST frei, die zuvor mit CVA_START oder CVA_COPY angelegt wurde. (siehe S. 142)

CVA_LIST
ist der interne Datentyp für variable Parameterlisten. (siehe S. 141)

CVA_START
initialisiert ein Objekt für variable Parameterlisten. (siehe S. 141)

CVD
konvertiert einen 8-Byte-String in eine **DOUBLE**-Zahl. Umkehrung von **MKD**. (siehe S. 156)

CVI
konvertiert einen 4-Byte-String in eine **INTEGER**-Zahl. Umkehrung von **MKI**. (siehe S. 156)

CVL
konvertiert einen 4-Byte-String in eine **LONG**-Zahl. Umkehrung von **MKL**. (siehe S. 156)

CVLONGINT
konvertiert einen 8-Byte-String in eine **LONGINT**-Zahl. Umkehrung von **MKLONGINT**. (siehe S. 156)

CVS
konvertiert einen 4-Byte-String in ein **SINGLE**. (siehe S. 156)

CVSHORT
konvertiert einen 2-Byte-String in ein **SHORT**. Umkehrung zu **MKSHORT**. (siehe S. 156)

DATA
speichert Konstanten im Programm, die mit **READ** eingelesen werden können.

DATE
gibt das aktuelle Datum im Format `mm-dd-yyyy` zurück. (siehe S. 250)

DATEADD
rechnet zu einem Datum ein bestimmtes Zeitintervall hinzu und gibt das neue Datum zurück *(benötigt datetime.bi)*. (siehe S. 259)

DATEDIFF
berechnet den zeitlichen Abstand zweier Datumsangaben *(benötigt datetime.bi)*. (siehe S. 259)

DATEPART
extrahiert eine Teilangabe aus einer gegebenen Serial Number *(benötigt datetime.bi)*. (siehe S. 256)

DATESERIAL
wandelt ein angegebenes Datum in eine Serial Number um *(benötigt datetime.bi)*. (siehe S. 254)

DATEVALUE
verwandelt einen String mit einer Datumsangabe in eine Serial Number um *(benötigt datetime.bi)*. (siehe S. 254)

DAY
extrahiert den Tag aus einer Serial Number *(benötigt datetime.bi)*. (siehe S. 256)

DEALLOCATE
gibt einen mit **ALLOCATE** reservierten Speicher wieder frei.

DECLARE
deklariert eine neue Prozedur, um sie dem Compiler bekannt zu machen. (siehe S. 128)

DEFINT, **DEFLNG**, **DEFLONGINT**, **DEFSHORT**, **DEFSNG**, **DEFSTR**, **DEFUBYTE**, **DEFUINT**, **DEFULONGINT**, **DEFUSHORT**
veraltet; kann nicht verwendet werden.

DELETE
löscht eine mit **NEW** erstellte Variable und gibt den Speicherplatz frei.

DESTRUCTOR
erstellt einen Klassen-Destruktor für ein **UDT** oder eine Prozedur, die am Programmende aufgerufen wird.

DIM
dimensioniert Variablen und Arrays. (siehe S. 23, S. 69, S. 72)

DIR
gibt die Dateien im aktuellen Arbeitsverzeichnis oder im angegebenen Pfad zurück. (siehe S. 238)

DO...LOOP
wiederholt einen Anweisungsblock, während bzw. bis eine Bedingung erfüllt ist. (siehe S. 103)

DOUBLE
eine Gleitkommazahl mit doppelter Genauigkeit (64 Bit). (siehe S. 44)

DRAW
kann für mehrere verschiedene Zeichenbefehle verwendet werden.

DRAW STRING
gibt einen Text im Grafikmodus an pixelgenauen Koordinaten aus.

DYLIBFREE
gibt den Speicher frei, der durch eine geladene dynamische Bibliothek belegt wurde.

DYLIBLOAD
lädt eine dynamische Bibliothek in den Speicher.

DYLIBSYMBOL
gibt den Pointer auf eine Prozedur oder Variable innerhalb einer dynamischen Bibbliothek zurück.

DYNAMIC
veraltet; kann nicht verwendet werden.

ELSE
wird im Zusammenhang mit **IF...THEN** verwendet. Es beschreibt den Fall, wenn alle anderen definierten Fälle nicht zutreffen. (siehe S. 86)

ELSEIF
wird im Zusammenhang mit **IF...THEN** verwendet. Es beschreibt einen vom ersten Prüffall unterschiedlichen Fall. (siehe S. 86)

ENCODING
wird zusammen mit **OPEN** verwendet, um festzulegen, in welcher Zeichenkodierung die Daten behandelt werden sollen. (siehe S. 206)

END
beendet eine Blockstruktur (Prozedur, Schleife ...) oder das Programm. (siehe S. 84, S. 242)

ENDIF
kann aus Gründen der Kompatibilität zu anderen BASIC-Dialekten anstelle von **END IF** verwendet werden. (siehe S. 83)

ENUM
erzeugt eine Liste von **INTEGER**-Konstanten vom Typ **INTEGER**. (siehe S. 54, S. 177)

ENVIRON
gibt den Wert einer Systemumgebungsvariablen zurück. (siehe S. 245)

EOF
gibt `-1` zurück, wenn der Dateizeiger das Ende einer geöffneten Datei erreicht hat. (siehe S. 203, S. 219)

EQV
vergleicht zwei Werte Bit für Bit und setzt im Ergebnis nur dann ein Bit, wenn die entsprechenden Bits in beiden Ausdrücken gleichwertig waren. (siehe S. 94)

ERASE
löscht dynamische Arrays aus dem Speicher oder setzt alle Elemente eines statischen Arrays auf den Initialisationswert. (siehe S. 77)

ERFN
gibt einen **ZSTRING PTR** auf den Namen der Prozedur zurück, in der ein Fehler aufgetreten ist. Das Programm muss dabei mit der Kommandozeilenoption `-exx` compiliert werden.

ERL
gibt die Zeilennummer des letzten aufgetretenen Fehlers zurück.

ERMN
gibt einen **ZSTRING PTR** auf den Namen des Moduls zurück, in dem ein Fehler aufgetreten ist.

ERR
gibt die Fehlernummer des zuletzt aufgetretenen Fehlers zurück oder setzt die Fehlernummer.
Im Zusammenhang mit **OPEN** wird eine Eingabe von der Standardeingabe *stdin* sowie eine Ausgabe auf die Standardfehlerausgabe *stderr* geöffnet. (siehe S. 223)

ERROR
simuliert einen Fehler.

EXEC
startet eine ausführbare Datei mit den übergebenen Argumenten. (siehe S. 241)

EXEPATH
gibt das Verzeichnis zurück, in dem sich das gerade ausgeführte Programm befindet. (siehe S. 231)

EXIT
verlässt eine Blockstruktur (Prozedur, Schleife ...). (siehe S. 113)

EXP
gibt die Potenz einer angegebenen Zahl zur eulerschen Zahl e zurück. (siehe S. 162)

EXPLICIT
gibt an, dass beim Aufruf der Elemente von **ENUM** der Listenname mit angegeben werden muss. (siehe S. 55)

EXPORT
wird in einer dynamischen Bibliothek für **SUBs** und **FUNCTIONs** verwendet, um sie in externen Programmen einbinden zu können.

EXTENDS
gibt bei der Klassen-Erstellung mit **TYPE** an, dass die Klasse die Attribute und Methoden von einer bereits bestehenden Klasse erbt.

EXTERN
wird benutzt, um auf externe Variablen zuzugreifen, die in anderen Modulen oder DLLs deklariert sind.
Als Blockstruktur wird ein Bereich erstellt, innerhalb dessen alle Deklarationen andere interne Bezeichner erhalten, als sie von FreeBASIC bekommen würden.

false
einer der beiden **BOOLEAN**-Werte; das Gegenteil von **true**

FIELD
wird zusammen mit **TYPE** und **UNION** verwendet; legt dort das Padding-Verhalten fest. (siehe S. 63)

FILEATTR
gibt Informationen über eine mit **OPEN** geöffnete Datei zurück *(benötigt file.bi)*. (siehe S. 235)

FILECOPY
kopiert die Datei *(benötigt file.bi)*. (siehe S. 235)

FILEDATETIME
gibt Datum und Uhrzeit, an dem die Datei zuletzt bearbeitet wurde, als Serial Number zurück *(benötigt file.bi)*. (siehe S. 237)

FILEEXISTS
überprüft, ob eine Datei existiert oder nicht *(benötigt file.bi)*. (siehe S. 238)

FILELEN
gibt die Länge von einer Datei in Bytes zurück *(benötigt file.bi)*. (siehe S. 237)

FIX
schneidet den Nachkommateil einer Zahl ab. (siehe S. 163)

FLIP
kopiert den Inhalt einer Bildschirmseite auf eine andere. Im OpenGL-Modus bewirkt **FLIP** eine Bildschirmaktualisierung.

FOR...NEXT
wiederholt den Codeblock zwischen **FOR** und **NEXT**, wobei eine Variable bei jedem Durchlauf um einen bestimmten Wert erhöht wird. (siehe S. 105)
Das Schlüsselwort **FOR** wird außerdem bei **OPEN** benötigt. (siehe S. 203)

FORMAT
wandelt einen numerischen Ausdruck anhand der angegebenen Formatierung in einen **STRING** um. (siehe S. 237, S. 252, S. 294)

FRAC
gibt die Nachkommastellen inklusive Vorzeichen einer Zahl zurück. (siehe S. 164)

FRE
gibt den verfügbaren RAM-Speicher in Bytes zurück.

FREEFILE
gibt die nächste unbenutzte Dateinummer zurück. (siehe S. 202)

FUNCTION
definiert ein Unterprogramm mit Rückgabewert. (siehe S. 132)

GET
als Dateioperation: liest Daten aus einer Datei. (siehe S. 213)
als Grafik**speichert**einen Ausschnitt aus einem Grafikfenster in einem Bildpuffer.

GETJOYSTICK
gibt die Position des Joysticks und den Status seiner Buttons zurück.

GETKEY
wartet mit der Programmausführung, bis eine Taste gedrückt wird.

GETMOUSE
liefert die Position der Maus und den Status der Buttons, des Mausrads und des Clipping-Status zurück.

GOSUB
veraltet; kann nicht verwendet werden.

GOTO
springt zu einem beliebigen Label. (siehe S. 101)

HEX
gibt den hexadezimalen Wert eines numerischen Ausdrucks als **STRING** zurück. (siehe S. 170)

HIBYTE
gibt das obere Byte eines Ausdrucks als **UINTEGER** zurück. (siehe S. 178)

HIWORD
gibt das obere Word eines Ausdrucks als **UINTEGER** zurück. (siehe S. 178)

HOUR
extrahiert die Stunde einer Serial Number *(benötigt datetime.bi)*. (siehe S. 256)

IF...THEN
führt einen Codeteil nur dann aus, wenn eine Bedingung erfüllt ist. (siehe S. 83)

IIF
liefert einen von zwei Werten zurück, abhängig davon, ob die Bedingung erfüllt ist oder nicht. (siehe S. 99)

IMAGECONVERTROW
kopiert eine bestimmte Anzahl von Pixeln von einem Grafikpuffer in einen anderen und konvertiert dabei die Anzahl der Bits pro Pixel in der Kopie auf einen gewünschten Wert.

IMAGECREATE
reserviert einen Speicherbereich als Datenpuffer für ein Bild.

IMAGEDESTROY
gibt einen mit **IMAGECREATE** reservierten Speicher wieder frei.

IMAGEINFO
gibt Informationen über das angesprochene Image zurück.

IMP
vergleicht zwei Werte Bit für Bit und setzt im Ergebnis nur dann *kein* Bit, wenn das zweite Operanden-Bit nicht gesetzt ist, während das Erste gesetzt ist. (siehe S. 94)

IMPLEMENTS
hat bisher keine Funktion, ist als Schlüsselwort aber bereits geschützt. Zukünfig soll der Befehl eine Klasse angeben, die ein Interface (eine Schnittstelle) implementiert.

IMPORT
wird unter Win32 zusammen mit **EXTERN** verwendet, wenn auf globale Variablen aus DLLs zugegriffen werden muss.

INKEY
gibt einen **STRING** zurück, der die erste Taste im Tastaturpuffer enthält. (siehe S. 37)

INP
liest ein Byte von einem Port.

INPUT
als Anweisung: liest eine Eingabe von der Tastatur oder aus einer Datei. (siehe S. 33, S. 210)
als Funktion: liest eine Anzahl an Zeichen von der Tastatur oder aus einer Datei. (siehe S. 36, S. 211)
als Dateimodus: wird zusammen mit **OPEN** verwendet, um eine Datei zum sequentiellen Lesen zu öffnen. (siehe S. 203)

INSTR
liefert die Stelle, an der ein String das erste Mal in einem anderen String1 vorkommt. (siehe S. 192)

INSTRREV
liefert die Stelle, an der ein String das letzte Mal in einem anderen String vorkommt. (siehe S. 193)

INT
rundet einen numerischen Ausdruck auf die nächstkleinere Ganzzahl ab. (siehe S. 163)

INTEGER
eine vorzeichenbehaftete 32-bit-Ganzzahl. (siehe S. 22, S. 39)

IS
prüft den Typen einer Variable. Kann nur genutzt werden, wenn die Basis-Klasse von **OBJECT** erbt.
IS wird auch im Zusammenhang mit **SELECT CASE** verwendet. (siehe S. 97)

ISDATE
überprüft, ob ein String einem korrekten Datumsformat entspricht. (siehe S. 255)

KILL
löscht eine Datei von einem Datenträger. (siehe S. 220)

LBOUND
gibt den kleinsten Index des angegebenen Arrays zurück. (siehe S. 74)

LCASE
wandelt einen Stringausdruck in Kleinbuchstaben. (siehe S. 94, S. 192)

LEFT
gibt die ersten Zeichen eines Strings zurück. (siehe S. 186)

LEN
gibt die Größe eines Stringausdrucks oder eines Datentyps zurück. (siehe S. 185)
LEN wird auch im Dateimodus **RANDOM** benötigt. (siehe S. 204)

LET
ermöglicht eine mehrfache Variablenzuweisung, bei der einer Liste von Variablen die Werte der einzelnen Attribute eines UDTs zugewiesen werden.

LIB
bindet eine **SUB** oder **FUNCTION** aus einer Bibliothek ein.

LINE
zeichnet eine Strecke von einem angegebenen Punkt zu einem zweiten, oder ein Rechteck, dessen Eckpunkte die beiden angegebenen Punkte sind.

LINE INPUT
liest eine Textzeile von der Tastatur oder aus einer Datei. (siehe S. 35, S. 210)

LOBYTE
gibt das niedere Byte eines Ausdrucks als **UINTEGER** zurück. (siehe S. 178)

LOC
gibt die Position des Zeigers innerhalb einer mit **OPEN** geöffneten Datei zurück. (siehe S. 218)

LOCAL
bewirkt in Zusammenhang mit **ON ERROR**, dass die Fehlerbehandlungsroutine nur für die gerade aktive Prozedur gilt und nicht für das gesamte Modul.

LOCATE
setzt die Position des Cursors in die angegebene Zeile und Spalte oder gibt Informationen über die aktuelle Cursorposition und die Sichtbarkeit des Cursors zurück. (siehe S. 27)

LOCK
sperrt den Zugriff auf eine Datei. (siehe S. 206)
Achtung: funktioniert zur Zeit nicht wie vorgesehen!

LOF
gibt die Länge einer geöffneten Datei in Bytes zurück. (siehe S. 219)

LOG
gibt den natürlichen Logarithmus (zur Basis e) zurück. (siehe S. 162)

LONG
eine vorzeichenbehaftete 32-bit-Ganzzahl. (siehe S. 39)

LONGINT
eine vorzeichenbehaftete 64-bit-Ganzzahl. (siehe S. 39)

LOOP
beendet einen **DO...LOOP**-Block.

LOWORD
gibt das niedere Word eines Ausdrucks als **UINTEGER** zurück. (siehe S. 178)

LPOS
gibt die Anzahl der Zeichen zurück, die seit dem letzten Zeilenumbruch an den Drucker gesendet wurden.

LPRINT, **LPRINT USING**
sendet Daten an den Standarddrucker. (siehe S. 226)

LPT
OPEN LPT bereitet den Drucker zum Datenempfang vor. (siehe S. 224)

LSET
befüllt einen Zielstring mit dem Inhalt eines Quellstrings, behält aber die Länge des Zielstrings bei. Kann auch mit UDTs verwendet werden. (siehe S. 190)

LTRIM
entfernt bestimmte führende Zeichen aus einem String. (siehe S. 190)

MID
gibt einen Ausschnitt einer Zeichenkette zurück oder ersetzt einen Teil einer Zeichenkette durch eine andere. (siehe S. 186, S. 188)

MINUTE
extrahiert die Minute einer Serial Number *(benötigt datetime.bi)*. (siehe S. 256)

MKD
verwandelt ein **DOUBLE** in einen 8-Byte-**STRING**. Umkehrung von **CVD**. (siehe S. 156)

MKDIR
erstellt einen Ordner. (siehe S. 232)

MKI
verwandelt ein **INTEGER** in einen 4-Byte-**STRING**. Umkehrung von **CVI**. (siehe S. 156)

MKL
verwandelt ein **LONG** in einen 4-Byte-**STRING**. Umkehrung von **CVL**. (siehe S. 156)

MKLONGINT
verwandelt ein **LONGINT** in einen 8-Byte-**STRING**. Umkehrung von **CVLONGINT**. (siehe S. 156)

MKS
verwandelt ein **SINGLE** in einen 4-Byte-**STRING**. Umkehrung von **CVS**. (siehe S. 156)

MKSHORT
verwandelt ein **SHORT** in einen 2-Byte-**STRING**. Umkehrung von **CVSHORT**. (siehe S. 156)

MOD
gibt den Rest der Division zurück (Modulo). (siehe S. 165)

MONTH
extrahiert den Monat einer Serial Number *(benötigt datetime.bi)*. (siehe S. 256)

MONTHNAME
extrahiert den Namen des Monats einer Serial Number *(benötigt datetime.bi)*. (siehe S. 258)

MULTIKEY
zeigt an, ob die angegebene Taste gerade gedrückt wird.

MUTEXCREATE
erstellt einen Mutex.

MUTEXDESTROY
löscht einen Mutex.

MUTEXLOCK
sperrt den Zugriff auf einen Mutex.

MUTEXUNLOCK
entsperrt einen Mutex.

NAKED
erstellt Funktionen ohne Handlingcode.

NAME
benennt eine Datei um. (siehe S. 234)

NAMESPACE
definiert einen Codeteil als Namespace, innerhalb dessen Symbole benutzt werden können, die in einem anderem Namespace bereits benutzt wurden.

NEW
weist dynamisch Speicher zu und erzeugt Daten bzw. Objekte.

NEXT
beendet einen **FOR...NEXT**-Block.

NOT
vertauscht die Bits im Quellausdruck; aus `1` wird `0` und aus `0` wird `1`. (siehe S. 94)

NOW
gibt die aktuelle Systemzeit als Serial Number aus *(benötigt datetime.bi)*. (siehe S. 252)

OBJECT
wird in Verbindung mit Vererbung genutzt. Will man über **IS** den Typ einer Variablen erfahren, muss die Basis-Klasse von **OBJECT** erben.

OCT
gibt den oktalen Wert eines numerischen Ausdrucks als **STRING** zurück. (siehe S. 170)

OFFSETOF
gibt den Offset (Abstand in Byte zur Adresse des UDTs) eines Attributs innerhalb eines UDTs zurück. (siehe S. 61)

ON...GOTO
verzweigt zu verschiedenen Labels, abhängig vom Wert des Ausdrucks. GOTO]

ON ERROR
bewirkt einen Programmsprung an ein angegebenes Label, sobald ein Fehler auftritt.

OPEN
öffnet eine Datei oder ein Gerät zum Lesen und/oder schreiben. (siehe S. 200)

OPERATOR
deklariert oder definiert einen überladenen Operator.

OPTION
ermöglicht es, zusätzliche Attribute oder Merkmale zu setzen.

OR
als Operator: vergleicht zwei Werte Bit für Bit und setzt im Ergebnis nur dann ein Bit, wenn mindestens ein Bit der entsprechenden Stelle in den Ausdrücken gesetzt waren. (siehe S. 89, S. 91, S. 173)
als Methode: wird im Zusammenhang mit **PUT** und **DRAW STRING** eingesetzt und bestimmt die Art, wie die zu zeichnende Grafik mit den Pixeln auf dem Bildschirm interagieren sollen.

ORELSE
prüft zwei Ausdrücke auf ihren Wahrheitsgehalt und gibt `-1` zurück, wenn mindestens einer der beiden Ausdrücke wahr ist. Ansonsten wird `0` zurückgegeben. Sind beide Operanden vom Typ **BOOLEAN**, dann ist auch das Ergebnis vom Typ **BOOLEAN**. (siehe S. 92)

OUT
schreibt ein Byte an einen Ausgabeport.

OUTPUT
wird zusammen mit **OPEN** verwendet, um eine Datei zum sequentiellen Schreiben zu öffnen. (siehe S. 203)

OVERLOAD
ermöglicht die Definition von Prozeduren mit unterschiedlicher Parameterliste, aber gleichem Prozedurnamen. (siehe S. 131)

OVERRIDE
gibt an, dass die dazugehörige Methode eine virtuelle oder abstrakte Methode seiner Elternklasse überschreiben muss.

PAINT
füllt in einem Grafikfenster einen Bereich mit einer Farbe.

PALETTE
bearbeitet die aktuelle Farbpalette.

PALETTE GET
speichert den Farbwert eines Palette-Eintrags.

PASCAL
setzt die Aufrufkonvention der Parameter auf die in `PASCAL`-Bibliotheken übliche Konvention (Übergabe von links nach rechts. (siehe S. 139)

PCOPY
kopiert den Inhalt einer Bildschirmseite auf eine andere.

PEEK
liest einen Wert direkt vom RAM.

PIPE
wird zusammen mit **OPEN** verwendet, um ein Programm zu starten und die Eingabe oder Ausgabe auf dieses Programm umzuleiten. (siehe S. 224)

PMAP
wandelt Sichtfensterkoordinaten in physische Koordinaten und umgekehrt.

POINT
gibt Informationen über die Farbe eines Pixels oder über die aktuelle Position des Grafikcursors zurück.

POINTER
wird zusammen mit einem Datentyp verwendet, um einen Pointer dieses Typs zu definieren. **POINTER** ist identisch mit **PTR**. (siehe S. 80)

POKE
schreibt einen Wert direkt in den RAM.

POS
gibt die horizontale Position des Cursors zurück. (siehe S. 28)

PRESERVE
wird zusammen mit **REDIM** benutzt, um ein Array zu redimensionieren, ohne seine Elemente zu löschen. (siehe S. 73)

PRESET
als Anweisung: zeichnet einen einzelnen Pixel, standardmäßig in der Hintergrundfarbe.
als Methode: wird im Zusammenhang mit **PUT** und **DRAW STRING** eingesetzt und bestimmt die Art, wie die zu zeichnende Grafik mit den Pixeln auf dem Bildschirm interagieren sollen.

PRINT
gibt einen Text auf dem Bildschirm oder in eine Datei aus. (siehe S. 20, S. 209)

PRIVATE
Verwendung mit **SUB** und **FUNCTION**: legt fest, dass die Prozedur nur aus dem Modul heraus aufgerufen werden kann, in dem sie sich befinden.
Verwendung in einem UDT: legt fest, dass ein Zugriff auf die folgenden Deklarationen nur von UDT-eigenen Prozeduren aus zulässig ist.

PROCPTR
gibt die Adresse einer Prozedur im Speicher zurück.

PROPERTY
erstellt eine Property einer Klasse.

PROTECTED
wird innerhalb einer UDT-Deklaration verwendet und legt fest, dass ein Zugriff auf die folgenden Deklarationen nur von UDT-eigenen Prozeduren aus zulässig ist.

PSET
als Anweisung: zeichnet einen einzelnen Pixel, standardmäßig in der Vordergrundfarbe.
als Methode: wird im Zusammenhang mit **PUT** und **DRAW STRING** eingesetzt und bestimmt die Art, wie die zu zeichnende Grafik mit den Pixeln auf dem Bildschirm interagieren sollen.

PTR
wird zusammen mit einem Datentyp verwendet, um einen Pointer dieses Typs zu definieren. **PTR** ist identisch mit **POINTER**. (siehe S. 80)

PUBLIC
Verwendung mit **SUB** und **FUNCTION**: legt fest, dass die Prozedur von allen Modulen heraus aufgerufen werden kann.
Verwendung in einem UDT: legt fest, dass ein Zugriff auf die folgenden Deklarationen von jeder Prozedur aus zulässig ist.

PUT
als Dateioperation: schreibt Daten in eine Datei. (siehe S. 212)
als Grafik**überträgt**einen Ausschnitt aus einem Bildpuffer auf das Grafikfenster.

RANDOM
wird zusammen mit **OPEN** verwendet, um eine Datei im Random-Modus zu öffnen. (siehe S. 204)

RANDOMIZE
initialisiert den Zufallsgenerator. (siehe S. 167)

READ
liest Daten, die zuvor mit **DATA** gespeichert wurden.

REALLOCATE
ändert die Größe eines mit **ALLOCATE** oder **CALLOCATE** reservierten Speicherbereichs.

REDIM
erstellt ein dynamisches Array oder ändert dessen Größe. (siehe S. 72, S. 72, S. 104)

REM
leitet einen Kommentar ein. (siehe S. 17)

RESET
schließt alle geöffneten Dateien oder setzt die Standardeingabe bzw. -ausgabe zurück. (siehe S. 223)

RESTORE
gibt an, welche mit **DATA** gespeicherten Variablen von der nächsten **READ**-Anweisung gelesen werden sollen.

RESUME
veraltet; kann nicht verwendet werden.

RETURN
verlässt eine **FUNCTION** oder **SUB**. (siehe S. 132)

RGB
errechnet die gültige 32-Farbnummer bei angegebenem Rot-, Grün- und Blauanteil.

RGBA
errechnet die gültige 32-Farbnummer bei angegebenem Rot-, Grün-, Blau- und Alphawert.

RIGHT
gibt die letzten Zeichen eines Strings zurück. (siehe S. 186)

RMDIR
löscht ein leeres Verzeichnis aus dem Dateisystem. (siehe S. 232)

RND
gibt ein zufälliges **DOUBLE** im Intervall $[0; 1[$ zurück. (siehe S. 168)

RSET
befüllt einen Zielstring mit dem Inhalt eines Quellstrings, behält aber die Länge des Zielstrings bei. (siehe S. 190)

RTRIM
entfernt bestimmte angehängte Zeichen aus einem String. (siehe S. 191)

RUN
startet eine ausführbare Datei. (siehe S. 241)

SADD
gibt einen Pointer auf eine **STRING**-Variable zurück. **SADD** ist identisch mit **STRPTR**.

SCOPE
öffnet einen Block, in dem Variablen temporär (re)dimensioniert und benutzt werden können.

SCREEN
als Funktion: gibt Informationen über den Text im Programmfenster zurück.
als Anweisung: setzt den aktuellen Bildschirm-Grafikmodus; sollte durch **SCREENRES** ersetzt werden.

SCREENCONTROL
ermittelt oder bearbeitet Einstellungen der *gfxlib*.

SCREENCOPY
kopiert den Inhalt einer Bildschirmseite in eine andere.

SCREENEVENT
gibt Informationen zu einem Systemereignis zurück, welches das Grafikfenster betrifft.

SCREENGLPROC
ermittelt die Adresse einer OpenGL-Prozedur.

SCREENINFO
gibt Informationen über den aktuellen Videomodus zurück.

SCREENLIST
gibt eine Liste aller unterstützten Bildschirmauflösungen zurück.

SCREENLOCK
sperrt den Zugriff auf eine Bildschirmseite.

SCREENPTR
gibt einen Pointer zurück, der auf den Datenbereich der aktiven Bildschirmseite zeigt.

SCREENRES
initiiert ein Grafikfenster.

SCREENSET
setzt die aktive und die sichtbare Bildschirmseite.

SCREENSYNC
wartet mit der Programmausführung auf eine Bildschirmaktualisierung.

SCREENUNLOCK
hebt eine Sperrung durch **SCREENLOCK** auf.

SCRN
OPEN SCRN öffnet die Standardausgabe. (siehe S. 223)

SECOND
extrahiert die Sekunde einer Serial Number *(benötigt datetime.bi)*. (siehe S. 256)

SEEK
setzt die Position des Zeigers innerhalb einer Datei oder gibt die Position zurück. (siehe S. 217)

SELECT CASE
führt, abhängig vom Wert eines Ausdrucks, bestimmte Codeteile aus. (siehe S. 96)

SETDATE
ändert das Systemdatum. (siehe S. 251)

SETENVIRON
verändert die Systemumgebungsvariablen. (siehe S. 246)

SETMOUSE
setzt die Koordinaten des Mauscursors und bestimmt, ob die Maus sichtbar oder unsichtbar ist.

SETTIME
setzt die neue Systemzeit. (siehe S. 251)

SGN
gibt einen Wert aus, der das Vorzeichen eines Ausdrucks identifiziert. (siehe S. 160)

SHARED
bewirkt im Zusammenhang mit **DIM**, **REDIM**, **COMMON** und **STATIC**, dass Variablen sowohl auf Modulebene als auch innerhalb von Prozeduren verfügbar sind. (siehe S. 123)

SHELL
führt ein Systemkommando aus und gibt die Kontrolle an das aufrufende Programm zurück, sobald das aufgerufene Kommando abgearbeitet wurde. (siehe S. 242)

SHL
verschiebt alle Bits in der Variablen um eine bestimmte Stellenzahl nach links. (siehe S. 179)

SHORT
eine vorzeichenbehaftete 16-bit-Ganzzahl. (siehe S. 39)

SHR
verschiebt alle Bits in der Variablen um eine bestimmte Stellenzahl nach rechts. (siehe S. 179)

SIN
gibt den Sinus eines Winkels im Bogenmaß zurück. (siehe S. 160)

SINGLE
eine Gleitkommazahl mit einfacher Genauigkeit (32 Bit). (siehe S. 44)

SIZEOF
gibt die Größe einer Struktur im Speicher in Bytes aus. (siehe S. 61)

SLEEP
wartet eine bestimmte Zeit oder bis eine Taste gedrückt wird. (siehe S. 20, S. 108, S. 117, S. 248)

SPACE
gibt einen **STRING** zurück, der aus Leerzeichen besteht. (siehe S. 131, S. 186)

SPC
wird im Zusammenhang mit **PRINT** verwendet, um Texteinrückungen zu erzeugen. (siehe S. 28)

SQR
gibt die Quadratwurzel eines Wertes aus. (siehe S. 159)

STATIC
erlaubt in Prozeduren die Verwendung von Variablen, deren Wert beim Beenden der Prozedur gespeichert wird und beim nächsten Prozeduraufruf wieder verfügbar sind. (siehe S. 125)
Innerhalb einer UDT-Definition deklariert **STATIC** Methoden, die auch aufgerufen werden können, ohne dass eine Instanz des UDTs existieren muss.

STDCALL
setzt die Aufrufkonvention der Parameter auf die in Standard-Aufrufkonvention für FreeBASIC und die Microsoft Win32-API (Übergabe von rechts nach links). (siehe S. 139)

STEP
gibt in einer **FOR**-Schleife die Schrittweite der Zählvariablen an. (siehe S. 106, S. 107)
In Grafikbefehlen legt **STEP** fest, dass die darauf folgenden Koordinaten relativ zum aktuellen Grafikcursor angegeben sind.

STOP
beendet das Programms; sollte durch **END** ersetzt werden.

STR
verwandelt einen numerischen Ausdruck in einen **STRING**. (siehe S. 151)

STRING
eine Zeichenkette variabler oder fester Länge. (siehe S. 22, S. 49, S. 184)
Als Funktion gibt **STRING()** eine Zeichenkette mit lauter gleichen Zeichen zurück. (siehe S. 186)

STRPTR
gibt einen Pointer zu einer String-Variablen zurück.

SUB
definiert ein Unterprogramm. (siehe S. 119)

SWAP
tauscht die Werte zweier Variablen vom selben Typ. (siehe S. 25)

SYSTEM
beendet das Programms; sollte durch **END** ersetzt werden.

TAB
wird zusammen mit **PRINT** benutzt, um die Ausgabe in der angegebenen Spalte fortzusetzen. (siehe S. 28)

TAN
gibt den Tangens eines Winkels im Bogenmaß zurück. (siehe S. 160)

THEN
wird zusammen mit **IF...THEN** verwendet.

THIS
wird in Prozeduren eines UDTs verwendet, um auf die Attribute und Methoden des UDTs zuzugreifen.

THREADCALL
startet eine Prozedur des Programms als eigenständigen Thread.

THREADCREATE
startet eine Prozedur des Programmes als eigenständigen Thread.

THREADWAIT
wartet mit der Fortsetzung des Hauptprogramms, bis eine Prozedur, die mit **THREADCREATE** oder **THREADCALL** als eigener Thread gestartet wurde, beendet ist.

TIME
gibt die aktuelle Uhrzeit im Format `hh:mm:ss` aus. (siehe S. 250)

TIMER
gibt die Zahl der Sekunden zurück, die seit dem Systemstart (unter DOS/Windows) bzw. seit der Unix-Epoche (unter Unix/Linux) vergangen sind. (siehe S. 249)

TIMESERIAL
wandelt eine angegebene Uhrzeit in eine Serial Number um *(benötigt datetime.bi)*. (siehe S. 254)

TIMEVALUE
wandelt einen **STRING** mit einer Zeitangabe in eine Serial Number um *(benötigt datetime.bi)*. (siehe S. 254)

TO
gibt Bereiche für die Befehle **FOR...NEXT**, **DIM**, **SELECT CASE** und **LOCK** an. (siehe S. 69, S. 97, S. 105)

TRANS
wird im Zusammenhang mit **PUT** und **DRAW STRING** eingesetzt und bestimmt die Art, wie die zu zeichnende Grafik mit den Pixeln auf dem Bildschirm interagieren sollen.

TRIM
gibt einen String aus, aus dem bestimmte führende oder angehängte Zeichen entfernt werden sollen. (siehe S. 192)

true
einer der beiden **BOOLEAN**-Werte; das Gegenteil von **false**

TYPE
erstellt ein UDT (user defined type). (siehe S. 57)

TYPEOF
gibt zur Compilierzeit den Datentypen zurück.

UBOUND
gibt den größten Index des angegebenen Arrays zurück. (siehe S. 74)

UBYTE
eine vorzeichenlose 8-bit-Ganzzahl. (siehe S. 40)

UCASE
wandelt einen Stringausdruck in Großbuchstaben. (siehe S. 192)

UINTEGER
eine vorzeichenlose 32-bit-Ganzzahl. (siehe S. 40)

ULONG
eine vorzeichenlose 32-bit-Ganzzahl. (siehe S. 40)

ULONGINT
eine vorzeichenlose 64-bit-Ganzzahl. (siehe S. 40)

UNION
definiert einen **UDT**, dessen Elemente sich eine Speicherstelle teilen. (siehe S. 65, S. 157)

UNLOCK
entsperrt eine mit **LOCK** gesperrte Datei.

UNSIGNED
erzeugt einen ganzzahligen Datentyp, der vorzeichenlos ist.

UNTIL
wird mit **DO...LOOP** verwendet. (siehe S. 103)

USHORT
eine vorzeichenlose 16-bit-Ganzzahl. (siehe S. 40)

USING
bindet die Symbole eines Namespaces in den globalen Namespace ein.
In Zusammenhang mit **PRINT** und **LPRINT** erzeugt **USING** eine formatierte Ausgabe.

VA_ARG
gibt den Wert eines Parameters in der Parameterliste einer Prozedur zurück. (siehe S. 146)

VA_FIRST
gibt einen Pointer auf den ersten Parameter einer variablen Parameterliste zurück. (siehe S. 146)

VA_NEXT
setzt den Pointer, der auf ein Argument einer variablen Parameterliste zeigt, auf das nächste Argument. (siehe S. 146)

VAL
konvertiert einen **STRING** zu einer Zahl. (siehe S. 150)

VALINT
konvertiert einen **STRING** zu einem **INTEGER**. (siehe S. 150)

VALLNG
konvertiert einen **STRING** zu einem **LONGINT**. (siehe S. 150)

VALUINT
konvertiert einen **STRING** zu einem **UINTEGER**. (siehe S. 150)

VALULNG
konvertiert einen **STRING** zu einem **ULONGINT**. (siehe S. 150)

VAR
deklariert eine Variable, deren Typ aus dem initialisierenden Ausdruck abgeleitet wird.

VARPTR
gibt die Adresse einer Variablen im Speicher zurück. (siehe S. 79)

VIEW
setzt die Grenzen des Grafik- oder Textanzeigebereichs (Clipping) oder gibt die Grenzen des Textanzeigebereichs zurück.

VIRTUAL
dient zum deklarieren virtuelle Methoden, die von erbenden Klassen überschrieben werden.

WAIT
liest regelmäßig ein Byte von einem Port und wartet mit der Programmausführung, bis dieses Byte bestimmte Bedingungen erfüllt.

WBIN
gibt den binären Wert eines Ausdrucks als **WSTRING** zurück. (siehe S. 170)

WCHR
verwandelt einen Unicode-Wert in seinen Character. (siehe S. 156)

WEEKDAY
extrahiert den Wochentag (1 bis 7) aus einer Serial Number *(benötigt datetime.bi)*. (siehe S. 256)

WEEKDAYNAME
gibt den Namen eines Wochentags aus *(benötigt datetime.bi)*. (siehe S. 258)

WEND
beendet einen **WHILE...WEND**-Block. (siehe S. 105)

WHEX
gibt den hexadezimalen Wert eines numerischen Ausdrucks als **WSTRING** zurück. (siehe S. 170)

WHILE
wird mit **DO...LOOP** und bei einer **WHILE...WEND**-Schleife verwendet. (siehe S. 103)

WIDTH
legt die Anzahl der Zeilen sowie der Zeichen pro Zeile für die Textausgabe fest oder gibt Informationen über die aktuelle Einstellung zurück.

WINDOW
bestimmt den neuen physischen Darstellungsbereich.

WINDOWTITLE
ändert die Beschriftung eines Grafikfensters.

WINPUT
liest eine Anzahl an Zeichen von der Tastatur oder aus einer Datei und gibt sie als **WSTRING** zurück. (siehe S. 211)

WITH
erlaubt es, auf die Attribute und Methoden eines UDTs zuzugreifen, ohne den Namen des UDTs mit angeben zu müssen. (siehe S. **??**, S. 59)

WOCT
gibt den oktalen Wert eines Ausdrucks als **WSTRING** zurück. (siehe S. 170)

WRITE
gibt einen Text auf dem Bildschirm oder in eine Datei aus, formatiert dabei aber anders als **PRINT**. (siehe S. 209)

WSPACE
gibt einen **WSTRING** zurück, der aus Leerzeichen besteht. (siehe S. 186)

WSTR
verwandelt einen numerischen Ausdruck in einen **WSTRING**. (siehe S. 151)

WSTRING
eine nullterminierte wide-chars-Zeichenkette. (siehe S. 49, S. 183)
Als Funktion gibt **WSTRING()** eine Zeichenkette mit lauter gleichen wide-chars zurück. (siehe S. 186)

XOR
als Operator: vergleicht zwei Werte Bit für Bit und setzt im Ergebnis nur dann ein Bit, wenn eines der beiden Bits in den Ausdrücken gesetzt war, aber nicht beide. (siehe S. 94, S. 176)
als Methode: wird im Zusammenhang mit **PUT** und **DRAW STRING** eingesetzt und bestimmt die Art, wie die zu zeichnende Grafik mit den Pixeln auf dem Bildschirm interagieren sollen.

YEAR
extrahiert das Jahr aus einer Serial Number *(benötigt datetime.bi)*. (siehe S. 256)

ZSTRING
eine nullterminierte Zeichenkette. (siehe S. 49, S. 182)

G.2. Metabefehle

#DEFINE
Symbol definieren

DEFINED
Überprüfen, ob ein Symbol definiert wurde

#ERROR
Fehlermeldung ausgeben

#IF, **#ELSEIF**, **#ELSE**, **#ENDIF**
Bedingung abprüfen

#IFDEF, **#IFNDEF**
Definition eines Symbols testen

#INCLIB
Bibliothek einbinden

#INCLUDE, **#INCLUDE ONCE**
externen Quelltext oder Header-Datei einbinden (siehe S. 233)

#LANG
Dialektform festlegen

#LIBPATH
Pfad für Bibliotheken hinzufügen

#LINE
Zeilennummer und Modulnamen festlegen

#MACRO, **#ENDMACRO**
Makro definieren

#PRAGMA
Compiler-Optionen ändern

#PRINT
Compilermeldung ausgeben

#UNDEF
Symbol löschen

G.3. Vordefinierte Symbole

__DATE__
gibt das Compilier-Datum im Format `mm-dd-yyyy` an.

__DATE_ISO__
gibt das Compilier-Datum im Format `yyyy-mm-dd` an.

__FB_ARGC__
gibt die Anzahl der Argumente an, die in der Kommandozeile für den Programmaufruf verwendet wurden.

__FB_ARGV__
gibt einen Pointer auf einen Speicherbereich zurück, in dem sich weitere **ZSTRING** PTRs befinden, die auf die einzelnen Kommandozeilenparameter im Programmaufruf zeigen.

__FB_BACKEND__
gibt an, ob Maschinencode (`gas`) oder C-Emitter-Code (`gcc`) erzeugt wurde.

__FB_BIGENDIAN__
wird immer dann definiert, wenn für ein System compiliert werden soll, das die big-endian-Regeln anwendet.

__FB_BUILD_DATE__
gibt das Datum (mm-dd-yyyy) aus, an dem der FB-Compiler erstellt wurde.

__FB_CYGWIN__
wird definiert, wenn der Code in der Cygwin-Umgebung umgesetzt werden soll.

__FB_DARWIN__
wird definiert, wenn der Code für Darwin umgesetzt werden soll.

__FB_DEBUG__
enthält den Wert `-1`, wenn beim Compilieren die Kommandozeilenoption `-g` angewandt wurde; andernfalls hat es den Wert `0`.

__FB_DOS__
wird definiert, wenn der Code von der DOS-Version des Compilers umgesetzt wird.

__FB_ERR__
gibt an, welche Art der Fehlerunterstützung beim Compilieren gewählt wurde: `1` für `-e`, `3` für `-ex`, `7` für `-exx`. Wurde keine Fehlerprüfung aktiviert, wird `0` zurückgegeben.

__FB_FPMODE__
enthält den Wert `fast`, wenn der Compiler *SSE floating point arithmetics* compiliert.

__FB_FPU__
enthält den Wert `sse`, wenn mit *SSE floating point arithmetics* compiliert wurde. Ansonsten hat es den Wert `x87`.

__FB_FREEBSD__
wird definiert, wenn der Code für den FreeBSD-Compiler umgesetzt werden soll.

__FB_LANG__
gibt an, nach welchen FB-Dialektregeln compiliert wird: `fb`, `fblite`, `deprecated` oder `qb`.

__FB_LINUX__
wird definiert, wenn der Code von der Linux-Version des Compilers umgesetzt werden soll.

__FB_MAIN__
wird definiert, sobald die Symbole und Makros des Hauptmoduls übersetzt werden.

__FB_MIN_VERSION__
vergleicht die Version des verwendeten Compilers mit den angegebenen Daten. Es gibt `-1` aus, wenn die Version des Compilers größer oder gleich den Spezifikationen ist, bzw. `0`, wenn die Version kleiner ist.

__FB_MT__
gibt an, ob der Code mit der FB-Multithread-Lib umgesetzt wird (`-1`) oder mit der FB-Standard-Lib (`0`).

__FB_NETBSD__
wird definiert, wenn der Code in der Version für den NetBSD-Compiler umgesetzt werden soll.

__FB_OPENBSD__
wird definiert, wenn der Code in der Version für den OpenBSD-Compiler umgesetzt werden soll.

__FB_OPTION_BYVAL__
enthält den Wert `-1`, wenn die Variablenübergabe standardmäßig **BYVAL** geschieht, bzw. `0`, wenn dies nicht der Fall ist.

__FB_OPTION_DYNAMIC__
enthält den Wert `-1`, wenn **OPTION DYNAMIC** verwendet wird, bzw. `0`, wenn dies nicht der Fall ist.

__FB_OPTION_ESCAPE__
enthält den Wert `-1`, wenn **OPTION ESCAPE** verwendet wird, bzw. `0`, wenn dies nicht der Fall ist.

__FB_OPTION_EXPLICIT__
enthält den Wert `-1`, wenn **OPTION EXPLICIT** verwendet wird, bzw. `0`, wenn dies nicht der Fall ist.

__FB_OPTION_GOSUB__
enthält den Wert `-1`, wenn **OPTION GOSUB** verwendet wird, bzw. `0`, wenn dies nicht der Fall ist.

__FB_OPTION_PRIVATE__
enthält den Wert `-1`, wenn **SUBs** und **FUNCTIONs** standardmäßig nur innerhalb des Moduls gültig sind.

__FB_OUT_DLL__
enthält den Wert `-1`, wenn der Code zu einer dynamischen Bibliothek (**.dll bzw.** .so) compiliert wird, bzw. `0`, wenn zu einem anderen Typ compiliert wird.

__FB_OUT_EXE__
enthält den Wert `-1`, wenn der Code zu einer ausführbaren Datei compiliert wird, bzw. `0`, wenn zu einem anderen Typ compiliert wird.

__FB_OUT_LIB__
enthält den Wert `-1`, wenn der Code zu einer statischen Bibliothek (*.lib) compiliert wird, bzw. `0`, wenn zu einem anderen Typ compiliert wird.

__FB_OUT_OBJ__
enthält den Wert `-1`, wenn der Code zu einer Objektdatei (*.obj) compiliert wird, bzw. `0`, wenn zu einem anderen Typ compiliert wird.

__FB_PCOS__
wird definiert, wenn der Code in einem Betriebssystem umgesetzt wird, dessen Dateisystem PC-artig aufgebaut ist.

__FB_SIGNATURE__
gibt einen String zurück, der die Signatur des Compilers enthält, z. B. *FreeBASIC 1.07.1*

__FB_SSE__
wird definiert, wenn der Compiler *SSE floating point arithmetics* compiliert.

__FB_UNIX__
wird definiert, wenn der Code in einem UNIX-artigen Betriebssystem compiliert wurde.

__FB_VECTORIZE__
enthält das durch die Compiler-Option `-vec` eingestellte Level (`0`, `1` oder `2`).

__FB_VER_MAJOR__
enthält die Version des Compilers enthält, z. B. `1`

__FB_VER_MINOR__
enthält die Version des Compilers enthält, z. B. `7`

__FB_VER_PATCH__
enthält die Version des Compilers enthält, z. B. `1`

__FB_VERSION__
enthält die Version des Compilers enthält, z. B. 0.24.0

__FB_WIN32__
wird definiert, wenn der Code von der Win32-Version des Compilers umgesetzt wird.

__FB_XBOX__
wird definiert, wenn der Code für die Xbox umgesetzt werden soll.

__FILE__
enthält den Dateinamen des Moduls, die gerade umgesetzt wird.

__FILE_NQ__
enthält den Dateinamen des Moduls, das gerade umgesetzt wird. Die Zeichenkette wird nicht durch Anführungszeichen eingeschlossen.

__FUNCTION__
enthält den Namen der Prozedur (**FUNCTION** oder **SUB**), die gerade umgesetzt wird.

__FUNCTION_NQ__
enthält den Namen der Prozedur (**FUNCTION** oder **SUB**), die gerade umgesetzt wird. Die Zeichenkette wird nicht durch Anführungszeichen eingeschlossen.

__LINE__
enthält die Zeile, die gerade umgesetzt wird.

__PATH__
enthält den Namen des Verzeichnisses, in dem sich die Quelltext-Datei befindet, die gerade umgesetzt wird.

__TIME__
enthält die Compilier-Uhrzeit im Format `hh:mm:ss`.

Index

& (et-Ligatur)
    String-Konkatenation, 51, 153
    Zahlensysteme, 170

Addition, 42
Adresse, 79
ANSI-Code, 153
Arbeitsverzeichnis, 230
Array
    Grenzen ermitteln, 74
    als Parameter, 134
    Dimensionen festlegen, 75
    löschen, 77
    neu dimensionieren, 72
Arrays, 69
    dynamisch, 71
    statisch, 71
ASCII-Code, 153
ASCII-Zeichentabelle, 289
Attribute, 57
Ausgabe
    Bildschirm, 20
Auslassung, 76

Bildschirm löschen, 30
Binärsystem, 90
Bit, 90
Bit-Operator, 91
Bit-Verknüpfung, 89
Bitfelder, 64
Bogenmaß, 160
Boolean, 52
Boolesche Algebra, 89
Byte, 90

Compiler, 2
Compiler-Optionen, 280
Copyright, iv
Cursor
    Bildschirmcursor, 27

Danksagung, v
Datei
    einbinden, 233
Datenfelder, 69
Deklaration, 72
Dekrementierung, 44
Dezimalpunkt, 45
Dezimalzahlen, 44
Dimensionen, 70
Dimensionierung, 72
Division, 42
Doppelpunkt, 18
Dualsystem, 90

Eingabe
    Tastatur, 33
Einrückung, 84
ELF, 14

Ellipsis, 76, 141
Errorlevel, 241, 245
EXE, 14
Exponentialdarstellung, 45
Exponentialfunktion, 162

Farbe, 29
fbc, 3
Funktion, 119, 132

Genauigkeit, 252

Hierarchie, 296

IDE, 8
inf (unendlich), 159
Initialisierung, 71
Inkrementierung, 44
Integerdivision, 42
Iterator, 270

Klammern, 42
Kommandozeile, 243
Kommentare, 17
Konkatenation, 50
Konstanten, 53, 125
Kopf, 119
Kurzschreibweise, 175, 179

Label, 101
Laufbedingung, 101
Leerstring, 48
Lesbarkeit, 17, 84
Lizenz, iv
Logarithmus, 162

Maschinencode, 14
Maschinenunabhängigkeit, 2
Multiplikation, 42

nan (not a number), 159
Nullbyte, 49, 185
Nullpointer, 80

Operand, 43
Operator, 43
Operatoren, 89
Overflow, 42

Padding, 60
Parameter, 27, 121
    leere Parameterliste, 37
Parameterübergabe, 121
    Pointer, 123
    UDT, 123
Paramter
    optional, 129
Pointer, 79, 80
Potenz, 42
Programm pausieren, 20
Projektseite, v
Prozedur, 119

Quelltext, 13
Quickrun, 11

Rechtliches, iv
Rumpf, 101, 119
Rundung, 110, 163
Rückgabewert, 132

Scancodes, 290
Schleife, 101
Schleifenkörper, 101
Schnellstart, 11
Schrittweite, 107
Scope, 109
seed, 167
Seiteneffekte, 123

Serial Number, 237, 251
Sichtbarkeitsbereich, 109
Signatur, 132
sprechende Namen, 174
Sprungmarke, 101
Stack, 139
Stapelspeicher, 139
String, 20
Strings, 48
Stringverkettung, 50
Subtraktion, 42

Tastaturpuffer, 37

UDTs, 57
Umgebungsvariable, 245
Unicode, 154
Unterprogramm, 119
Unterstrich, 18

Variablen, 22
    deklarieren, 23
    global, 123
    lokal, 123
    Name, 24
    statisch, 125
Variablentyp, 39
Vorrangregeln, 296

Wahrheitswerte, 52
Wildcards, 238
wissenschaftliche Notation, 45
Word, 178

Zeichenkette, 20
Zeichenketten, 48, 49
Zeiger, 79, 80
Zeilenfortsetzung, 18

Überlauf, 42

Liste der Quelltexte

2.1. hallowelt.bas 11

4.1. PRINT-Ausgabe 21
4.2. Ausgabe von Variablen 26
4.3. Positionierung mit LOCATE 27
4.4. Farbige Textausgabe 30
4.5. Fenster-Hintergrundfarbe setzen 31

5.1. Benutzereingabe mit INPUT 35
5.2. Einzelzeichen mit INPUT() 37

6.1. Rechnen über den Speicherbereich hinaus 43
6.2. Rechnen mit Gleitkommazahlen 46
6.3. Stringverkettung mit Pluszeichen 51
6.4. Stringverkettung mit et-Ligatur 52
6.5. Konstanten 53
6.6. Verwendung von ENUM 54
6.7. ENUM mit Wertzuweisung 55

7.1. Anlegen eines UDTs 58
7.2. Verschachtelte UDT-Struktur 59
7.3. Vereinfachter UDT-Zugriff mit WITH 60
7.4. Verschachteltes WITH 60
7.5. Standard-Padding bei UDTs 62
7.6. Benutzerdefiniertes Padding 63
7.7. Deklaration von Bitfeldern 64
7.8. Einfache UNION-Verwendung 65
7.9. UNION innerhalb einer UDT-Deklaration 67

8.1. Array-Deklaration 69
8.2. Array-Deklaration mit direkter Wertzuweisung 70
8.3. Mehrdimensionales Array 71

8.4. Dynamische Arrays anlegen 72
8.5. REDIM mit und ohne PRESERVE 73
8.6. Verwendung von LBOUND und UBOUND 74
8.7. Implizite obere Grenze bei Arrays 76
8.8. Arrays löschen 77

9.1. Pointerzugriff 81
9.2. Speicherverwaltung bei Arrays 81

10.1. Einfache Bedingung (einzeilig) 83
10.2. Einfache Bedingung (mehrzeilig) 84
10.3. Verschachtelte Bedingungen 85
10.4. Mehrfache Bedingung 87
10.5. Vergleich von Zeichenketten 88
10.6. Bit-Operatoren AND und OR 92
10.7. ANDALSO und ORELSE 93
10.8. Mehrfache Bedingung (2) mit IF 95
10.9. Mehrfache Bedingung (2) mit SELECT CASE 96
10.10. Ausdruckslisten bei SELECT CASE 98
10.11. Bedingungen mit IIF 99

11.1. Passwortabfrage in einer Schleife 104
11.2. Wiederholte Namenseingabe 105
11.3. Einfache FOR-Schleife 106
11.4. Countdown 108
11.5. Sichtbarkeit der Zählvariablen 109
11.6. FOR: Probleme der Gleitkomma-Schrittweite 111
11.7. FOR: Probleme mit dem Wertebereich 112
11.8. CONTINUE FOR 113
11.9. Countdown mit Abbruchbedingung 114
11.10. Dauerschleife mit SLEEP 1 117

12.1. Hallo Welt als Prozedur 120
12.2. Prozedur mit Parameterübergabe 121
12.3. Probleme mit unachtsamer Verwendung von SHARED 124
12.4. Korrekte Berechnung aller Summen 125
12.5. Statische Variable in einem Unterprogramm 126
12.6. Statisches Unterprogramm 127

12.7. Deklarieren einer Prozedur 128
12.8. PingPong 129
12.9. Optionale Parameter 130
12.10. Überladene Prozeduren (OVERLOAD) 131
12.11. Arithmetisches Mittel zweier Werte 133
12.12. Arithmetisches Mittel aller Werte eines Arrays 134
12.13. Arithmetisches Mittel für ein eindimensionales(!) Array 135
12.14. BYREF und BYVAL 137
12.15. Parameterübergabe AS CONST 139
12.16. Mittelwertsbestimmung mit variabler Parameterliste 143
12.17. Variable Parameterliste mit Formatstring 144
12.18. Variable Parameterliste kopieren 145
12.19. Mittelwertsbestimmung mit variabler Parameterliste (alte Methode) 146

13.1. Einsatz von CAST 149
13.2. Zahl in einen String umwandeln 151
13.3. Hallo Welt in ASCII-Werten (1) 154
13.4. Hallo Welt in ASCII-Werten (2) 155
13.5. ANSI-Zeichen von 128 bis 255 156
13.6. Binäre Kopie mit MKLONGINT und mit UNION 158

14.1. Quadratwurzel; Betragsfunktion; Signumfunktion 160
14.2. Trigonometrische Berechnungen 161
14.3. Arkusfunktionen 162
14.4. Exponentialberechnungen 163
14.5. Rundungsverfahren 164
14.6. Rechungen mit Modulo 165
14.7. Subtraktion mit Modulo 166
14.8. Zufallszahlen ausgeben 168
14.9. Würfelwurf simulieren 169
14.10. Vom Dezimalsystem in ein anderes System umrechnen 171
14.11. Umwandlung ins Binärsystem und zurück 172
14.12. Geradzahligkeit mit AND prüfen 173
14.13. Formateigenschaften mit OR und AND setzen und lesen 175
14.14. Formateigenschaften mit BITSET() und BIT() setzen und lesen 177
14.15. Byte- und Word-Zugriff 178
14.16. SHL und SHR 180

15.1. LEN() zur Ermittlung der String-Länge 185
15.2. STRING() und SPACE() 186
15.3. LEFT() und RIGHT() 187
15.4. MID 187
15.5. Teilstring ersetzen mit MID 188
15.6. Textzensur mit MID 189
15.7. LSET und RSET 190
15.8. LTRIM() und RTRIM() 191
15.9. RTRIM() mit Schlüsselwort ANY 191
15.10. Alles gross oder alles klein 192
15.11. Teilstring suchen 193
15.12. Teilstring rückwärts suchen 194

16.1. Freie Dateinummer ermitteln 202
16.2. Dateizugriff auf das Lesen beschränken 206
16.3. Dateien schließen 208
16.4. Vergleich zwischen PRINT# und WRITE# 209
16.5. INPUT-Methode bei Dateien 210
16.6. INPUT() bei Dateien 211
16.7. Bildabmessungen eines BMP auslesen 214
16.8. UDT speichern und auslesen 216
16.9. Strings variabler Länge speichern und laden 217
16.10. Dateizeiger setzen und lesen 218
16.11. Sequentielle Datei vollständig einlesen 219
16.12. Datensatz aus einer binären Datei einlesen 220
16.13. Datei löschen 221
16.14. Standard-Datenströme nutzen 223
16.15. Unidirektionale Pipe 224
16.16. Einfaches Druckbeispiel 225

17.1. Arbeitsverzeichnis anzeigen und wechseln 230
17.2. Eine im Programmpfad liegende Datei auslesen 232
17.3. Temporären Ordner anlegen 233
17.4. Dateien umbenennen und verschieben 234
17.5. Datei kopieren 235
17.6. Dateiattribute abfragen 237
17.7. Suche nach einer Datei 240
17.8. Auflisten aller normalen Dateien und Ordner 241

17.9. Ordnerinhalt anzeigen mit SHELL() 243
17.10. COMMAND() - aufrufendes Programm 244
17.11. COMMAND() - aufgerufenes Programm 245
17.12. ENVIRON() und SETENVIRON 246

18.1. Beispiele für den Einsatz von TIMER() 249
18.2. SLEEP 1 nur jeden hundertsten Durchlauf 250
18.3. Aktueller Zeitpunkt mit NOW() und FORMAT() 253
18.4. TIMESERIAL() und Co. 255
18.5. Informationen aus einer Serial Number ermitteln 256
18.6. Namen der Wochentage und Monate 258
18.7. DATEDIFF() und DATEADD() 260