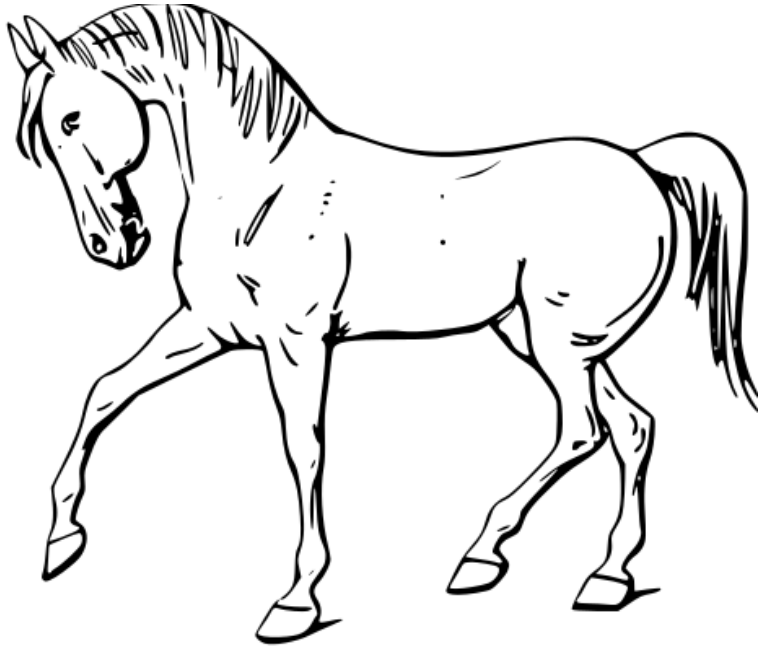
# FreeBASIC-Einsteigerhandbuch

**Grundlagen der Programmierung in FreeBASIC**



von S. Markthaler

Stand: 30. August 2015

# 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.

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. Wenn Ihnen 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 Programms 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. Alle fünf Zeilen wird die aktuelle Zeilennummer angezeigt; diese ist selbst nicht Bestandteils des Programms und soll dementsprechend auch nicht mit abgetippt werden.

Die Ausgabe des Programms wird ebenfalls in nichtproportionaler Schrift aufgelistet und, wie unten angezeigt, durch eine graue Box umschlossen. Diese Ausgabe kann mit der eigenen verglichen werden, um zu überprüfen, ob das Programm korrekt eingegeben wurde.

Ausgabe

```
Hallo FreeBasic-Welt!
```

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
http://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. Besonders erwähnen möchte ich (TODO: Helfer suchen :D)

# Inhaltsverzeichnis

# 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 für den Computer zugänglich zu sein. 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 und DOS 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 drei 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 im Kapitel **??**.

- **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 muss, 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**[1], **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 kann mit dem Befehl eine mehrfache Variablenzuweisung für die Records eines UDTs vorgenommen werden.

- **numerische Labels**
  Labels, die nur aus einer Zahl 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:

[1] **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, Operator-Überladung, 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.
- Objektorientierte Programmierung wird nur in Grundzügen unterstützt.
- 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.[2] 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 unter der Adresse:
http://www.freebasic-portal.de/downloads/aktuelle-compiler/
Die IDE wxFBE kann hier heruntergeladen werden:
http://www.freebasic-portal.de/projekte/wxfbe-69.html [3]
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

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

[3] Außerdem werden für wxFBE unter Windows die Microsoft Visual C++ 2008 Laufzeitkomponenten (oder höher) benötigt. Sollten diese noch nicht installiert sein, können sie von der Microsoft-Homepage heruntergeladen werden:
http://www.microsoft.com/de-de/download/details.aspx?id=5582

genau genommen zu hoch gegriffen – der Ordner muss lediglich an eine passende Stelle 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.

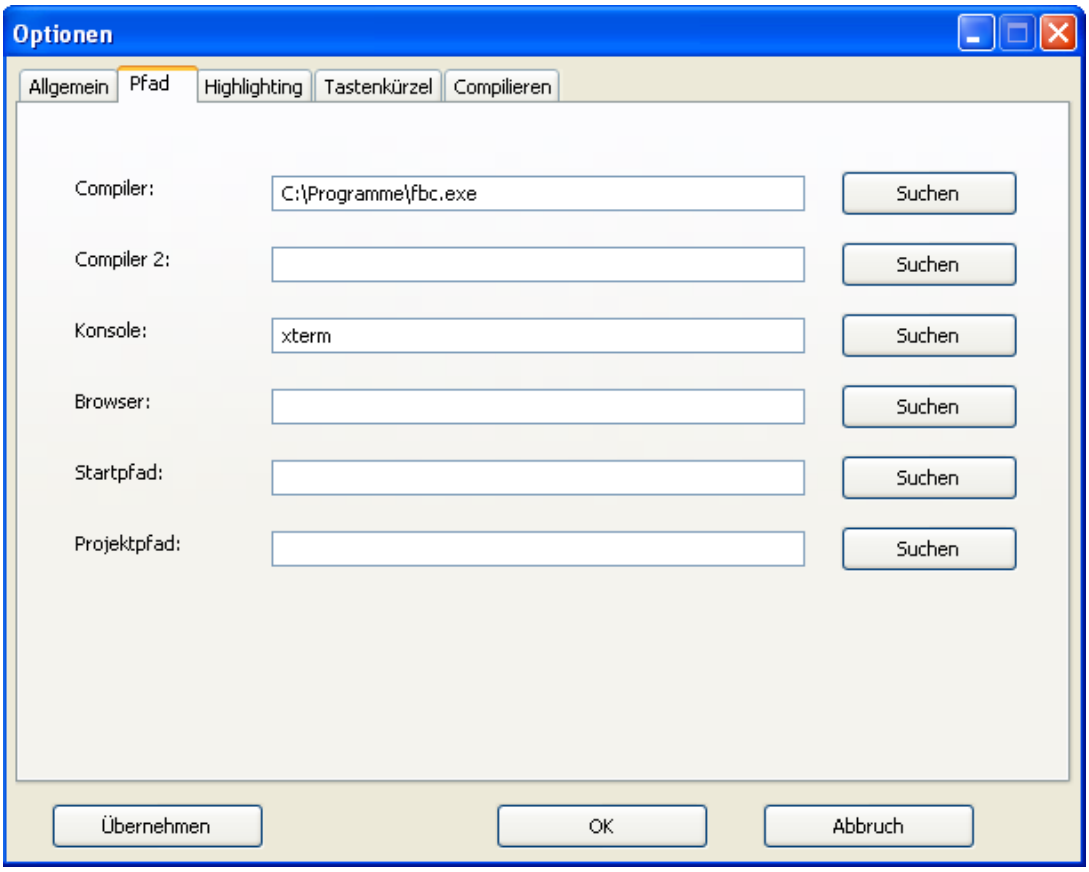
Allerdings ist bei der Installation etwas mehr Arbeit erforderlich. Die Einzelinstallation von wxFBE läuft zunächst genauso wie die Installation des Komplettpakets. Zusätzlich wird jedoch der Compiler benötigt, der hier zu finden ist:
`http://www.freebasic-portal.de/downloads/aktuelle-compiler/`
Der 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, empfielt 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 Bibiotheken 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 Programms 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 bzw. in Kapitel 2.1.

Sollte wxFBE unter Linux nicht starten, so empfielt 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.exeC:\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.exeC:\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 Programms auf 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.

Wenn 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 nach 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 Programms benötigt – die erstellte EXE (bzw. ELF) ist für sich allein lauffähig. wxFBE erleichert 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-Programms

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-Programms 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 Programms, 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 auch 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 Kapitel **??**.

## 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 zur Hinzufuegung 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).

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. Im folgenden Beispiel 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

Um sich innerhalb des Programms 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 Variablentypen folgt in den nächsten Kapiteln.

Wertzuweisungen erfolgen über das Gleichheitszeichen =

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

*variable* bekommt hier den Wert 3 zugewiesen – zu einem späteren Zeitpunkt des Programms 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), die Ziffern 0-9 und den Unterstrich _ enthalten. 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!

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 angegebende Stelle vor. Es bestehen also Ähnlichkeiten zu **LOCATE** mit ausgelassenem ersten Parameter. Beachten Sie jedoch, dass tätsächlich *vor*gerückt 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. 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.

### 4.3.4. 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 eingestellten 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 Programms 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; zunächst dient die Tastatur als Eingabemedium.

## 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 selbst eine Meldung ausgeben. Diese wird als String vor die Speichervariable gesetzt:

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

Soweit, 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 einfach durch 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. Wenn Sie sich für genauere Details interessieren, finden Sie diese in der Befehlsbeschreibung zu **WRITE**. Desweiteren 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 Stringkonkatierung (dazu später mehr) zusammengesetzt worden sein – es muss sich um einen einzelnen String handeln, der mit Anführungszeichen beginnt und mit Anführungszeichen endet.

Noch eine weitere Besonderheit hält **INPUT** 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: Stephan
Hallo, Stephan. Wie findest du FreeBASIC? Super!
Vielen Dank. Mit einem Tastendruck beendest du das Programm.
```

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 um keine Variable und auch nicht 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 einzulesenden Variablen. Die einzelnen Werte werden vom Benutzer ebenfalls durch Komma getrennt eingegeben.

## 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 ein-

gegeben 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 schreiben, 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.

Beachten Sie unbedingt den Datentyp des Rückgabewertes! Im Falle von **INPUT()** ist der Rückgabewert 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 eines Zeichens gewartet wird, oder genauer gesagt: Die Funktion holt ein Zeichen aus dem Tastaturpuffer. Immer, wenn Sie eine Taste drücken, wird diese in den Tastaturpuffer gelegt. Von dort aus kann sie, 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!

### 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.

Ein Vorteil von **`INKEY()`** ist, dass Sonderzeichen komplett übergeben werden. Bei einem Druck bspw. auf eine Pfeiltaste ist der Rückgabewert zwei Zeichen lang. Allerdings werden wir im Augenblick noch nicht viel mit der Funktion anfangen können, da unsere Programm 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.

Der Vollständigkeit halber werden noch zwei weitere Befehle zur Tastaturabfrage genannt: **`GETKEY()`** wird vorwiegend dazu 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 eingangen.

## 5.3. Fragen zum Kapitel

Das Kapitel beschäftigte sich mit der Eingabe über 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()`**?

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 drei verschiedene Typen von Variablen:

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

Außerdem ist es möglich aus den vorgegebenen Datentypen eigene benutzerdefinierte zusammenzubauen – eine Sache, die in Kapitel 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 32bit-Version des Compilers verwenden, belegt ein **`INTEGER`** 32 Bit (entspricht also einem **`LONG`**); in

der 64bit-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 vorzeichenbehaf. | -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, 32bit- bzw. 64bit-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 32bit-Umgebung eine **BYTE**-Variable einsetzen, belegt diese nur acht Bit einer 32-Bit-Speicherstelle. 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, wird zur Verwendung von **INTEGER** bzw. **UINTEGER** geraten.

Einen Nachteil dieser flexiblen Anpassung sollte man allerdings noch erwähnen: 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.

### 6.1.2. 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. Folgende Zeichen werden verwendet:

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

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.

Quelltext 6.1: Rechnen über den Speicherbereich

```
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 kann durchaus erwünscht sein, 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.

### 6.1.3. 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 der 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 Operanten – 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: Wenn 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 *Nachkommastellen*. 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 $\infty$)** | **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 6.2.: 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. 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“ ($\infty$) bzw. „unendlich klein“ ($-\infty$) behandelt, dargestellt als *inf* bzw. *-inf*.

**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.

### 6.2.2. Rechengenauigkeit bei Gleitkommazahlen

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

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 die 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. 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.

## 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 zuende ist, reserviert es intern – vom Benutzer versteckt – eine weitere Speicherstelle für die Länge der Zeichenkette. Dies sieht, 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 verwendet ein reserviertes „Stoppzeichen", 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 B entnehmen.

**Achtung:**
Für Konsole und Grafikfenster werden verschiedene ANSI-Codepages verwendet.

**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 das „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 der **STRINGs** beschränken. Auf den Umgang mit **ZSTRING** und **WSTRING** wird in **??** 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 32bit-Version des Compilers bis zu 2 GB groß werden, in der 64bit-Version sogar 8 388 607 TB.

**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 +

```
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 &

```
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 Interessantes anstellen, wie z. B. nur einen Teil der Zeichenkette ausgeben, in Groß- oder Kleinbuchstaben umwandeln und vieles mehr. Dazu erfahren Sie mehr in **??**.

## 6.4. 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 Programms 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 $\pi$ als Konstanten fest. Die Werte können anschließend genauso wie Variablen ausgegeben oder für Berechnungen 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 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 Ganzzahl 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.

## 6.5. 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.

## 6.6. 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. Wo liegt der Unterschied zwischen einem **`STRING`** und einem **`ZSTRING`**?
5. Was ist der Unterschied zwischen einer Variablen und einer Konstanten?

# 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 Verablen 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 Träger der objektorientierten Programmierung, die zu komplex ist, um sie in ein einziges Kapitel zu pressen, und die daher einen eigenen Abschnitt im Buch erhält.

## 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 sie 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“, auch *Records* genannt. Auf die Records kann, sowohl lesend als auch schreibend, nach dem Muster `udtname.recordname` zugegriffen werden. Dazu folgt in Quelltext 7.1 ein Beispiel. Die Records, z. B. `vorname`, existieren nur innerhalb des UDTs, es wird also keine allgemein verfügbare Variable mit dem Namen `vorname` angelegt.

Ihnen wird sicher aufgefallen sein, dass die Deklaration der Records 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 Records 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 Record-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 Records 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 einen UDT-Record 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. Recordzugriff 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 Record dieses UDTs kann anschließend innerhalb des Blocks der UDT-Name weggelassen werden. Der Record 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 adresse
WITH adresse
  .strasse = "Milchstr."   ' Kurzform von adresse.strasse
  .hausnummer = 39         '     bzw. von adresse.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.

## 7.3. Speicherverwaltung

Die Speicherplätze für die Records eines UDTs liegen direkt hintereinander (ein weiterer Vorteil von UDTs, weil sich ein UDT dadurch oft komplett an einem Stück speichern und laden lässt[4]). 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*.

Besteht ein UDT aus drei **BYTE**-Records, 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.

Das Padding-Verhalten soll in Quelltext 7.4 veranschaulicht werden. Zur Bestimmung der Speichergröße eines Elements wird **SIZEOF** verwendet. Man kann damit die Anzahl der Bytes ausgeben lassen, die von einer angegebenen Variablen, aber auch von Variablen eines angegebenen Datentyps belegt werden.

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

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

Quelltext 7.4: Standard-Padding bei UDTs

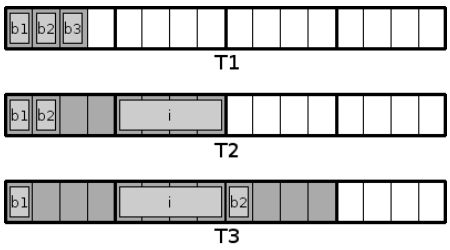
```
TYPE T1
  AS BYTE b1, b2, b3
END TYPE
TYPE T2
  AS BYTE b1, b2
  AS INTEGER i
END TYPE
TYPE T3
  AS BYTE b1
  AS INTEGER i
  AS BYTE b2
END TYPE

PRINT SIZEOF(T1), SIZEOF(T2), SIZEOF(T3)
SLEEP
```

Ausgabe

```
 3              8              12
```

Diese Ausgabe gilt für die 32bit-Version des Compilers. Wie der Speicherbedarf 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 (32bit-)Integer, findet ein Padding

auf die Größe dieses Datentyps statt.[5] Das wird vor allem beim UDT `T3` deutlich: Auch 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 7.5: 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 SIZEOF(T4), SIZEOF(T5), SIZEOF(T6)
SLEEP
```

Ausgabe

```
6               8               6
```

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


Durch eine geringere Ausdehnung wird Speicherplatz gespart, Sie sehen jedoch, dass

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

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.

**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 Record-Namen ein Doppelpunkt geschrieben, gefolgt von der gewünschten Bit-Zahl.

Quelltext 7.6: Deklaration von Bitfeldern

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

Hier werden drei Records 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 Record 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 Records.

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 32bit-Integers (sehr wohl aber eines 64bit-Integers). Außerdem wird **LONGINT** nur in der 64bit-Version des Compilers unterstützt.

Desweiteren wird ein UDT immer vollständige Bytes als Speicherplatz belegen. Ein UDT mit einem einzigen Bitfeld-Record wird keine Speicherersparnis mit sich bringen. Selbst wenn der Record nur ein einziges Bit belegt, wird die Größe des UDTs durch die Größe des gewählten Ganzzahl-Datentyps festgelegt. Eine Platzersparnis tritt erst ein, wenn der Speicherplatz des gewählten Datentyps auf mehrere Records 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 7.7: 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**, wordurch 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`.

Gern eingesetzt 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 QBasic-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 QBasic-Format gespeicherten Bilder noch unterstützt werden sollen, wurde der neue Header folgendermaßen definiert:

Quelltext 7.8: 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`**-Records 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."
```

## 7.6. Fragen zum Kapitel

Zum Kapitel folgen diesmal nur wenige Fragen, die allerdings etwas Denkarbeit erfordern. Wenn Sie das Kapitel aufmerksam gelesen haben, sollten Sie die Fragen allerdings ohne größere Schwierigkeiten beantworten können.[6]

1. Spielt bei der UDT-Deklaration die Reihenfolge, in der die Records angegeben werden, eine Rolle? Wenn ja, warum? Wenn nein, warum nicht?

2. In Quelltext 7.8 wurde die Deklaration des FreeBASIC-internen Grafik-Headers vorgestellt. Warum verwendet der Header das Padding `FIELD=1`?

[6] Wenn Sie das Kapitel nicht aufmerksam gelesen haben, lesen Sie es noch einmal.

# 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 Namen hinzukommen. Stattdessen bietet sich ein Array an, das über einen einzigen Namen 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 Programms 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 kann der Befehl **REDIM** verwendet werden. **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
' Array vergroessern
REDIM PRESERVE arr(1 TO 6)
PRINT arr(3), arr(4), arr(5)
' Grenzen verschieben
REDIM PRESERVE arr(3 TO 9)
PRINT arr(3), arr(4), arr(5)
' Neudimensionierung ohne PRESERVE
REDIM arr(3 TO 9)
PRINT arr(3), arr(4), arr(5)
SLEEP
```

Ausgabe

```
3             4             0
1             2             3
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 Speicherverwaltung bei Arrays 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 dimensioniert ist.

### 8.4.2. 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.

### 8.4.3. 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 Programms 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 Programms, 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 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 Programms, 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 den Variablennamen ein @ gesetzt wird.

```
DIM AS INTEGER x
PRINT "x befindet sich an der Speicherstelle "; @x
SLEEP
```

Die Variable wird bei jedem Start des Programms an einer anderen Stelle abgelegt. Bei der Adresse handelt es sich, je nach Betriebssystem, um einen 32bit- oder 64bit-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 32bit in einem 32bit-System und 64bit in einem 64bit-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 Variablennamen 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`[7] auslösen. Man kann der Pointer-Variablen jedoch zuvor die Adresse einer anderen Variablen zuweisen.

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

Pointer werden vor allem dann benötigt, wenn eigene Speicherbereiche verwaltet werden

[7] 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“.

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.

# 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 Ferseher an.
```

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 Programms 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 – doch auch diese Werte *wahr* und *falsch* muss sich das Programm in irgendeiner Form merken. In vielen Programmiersprachen gibt es dafür einen eigenen Datentyp, in FreeBASIC ist das jedoch nicht der Fall. Stattdessen werden alle Zahlen als Wahrheitswert interpretiert: Ist die Zahl 0, so gilt sie als *false (falsch)*; in allen anderen Fällen gilt sie als *true (wahr)*.

Üblicherweise werden in FreeBASIC die Werte 0 =*false* und −1 =*true* verwendet – bei der Zahl −1 sind intern alle Bits gesetzt (vgl. dazu **??**), während bei der Zahl 0 kein Bit gesetzt ist. −1 ist vom Wert her also genau das „Gegenteil“ von 0. Jedoch werden auch alle anderen Zahlen außer 0 als *true* interpretiert – ein Umstand, der in Kapitel 10.2.2 noch eine wichtige Rolle spielt.

### 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
SLEEP
```

Folgende Vergleichsoperatoren können verwendet werden:

| Operator | Bedeutung | Beispiel | Wert |
|---|---|---|---|
| $>$ | größer als | $5 > 5$ | *false* |
| | | $3 > 8$ | *false* |
| $<$ | kleiner als | $5 < 5$ | *false* |
| | | $3 < 8$ | *true* |
| $=$ | gleich | $5 = 5$ | *true* |
| | | $3 = 8$ | *false* |
| $>=$ | größer oder gleich | $5 >= 5$ | *true* |
| | | $3 >= 8$ | *false* |
| $<=$ | kleiner oder gleich | $5 <= 5$ | *true* |
| | | $3 <= 8$ | *true* |
| $<>$ | ungleich | $5 <> 5$ | *false* |
| | | $3 <> 8$ | *true* |

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-Wert* des Zeichens (vgl. dazu **??** und Anhang B). 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 **??** 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 D 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 nur die Wahrheitswerte 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. 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.

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 *true* interpretiert, `2 AND 4` jedoch ist 0 und damit *false*. 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` *true* 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 *false* ist, kann der Gesamtausdruck nicht mehr *true* sein, und die rechte Seite wird daher gar nicht mehr ausgewertet. Wenn umgekehrt bei **ORELSE** die linke Seite *true* ist, dann ist auch der Gesamtausdruck *true*. 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
```

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. 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 *true* 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 ein Argument angewandt. `NOT 0` z. B. ist -1 und `NOT -1` ist 0.

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. Dazu eine tabellarische Übersicht:

| 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 10.1.: Bit-Operatoren

Sofern diese Operatoren ausschließlich für die Wahrheitswerte -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.

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 ügedrckt"
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 ügedrckt"
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.

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 11.4.2). 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.
- In den 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.

Der Hintergrund ist, dass 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). Die Syntax lautet hier

```
SELECT CASE AS CONST Ausdruck
```

## 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 *true* oder *false* 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 *true* und welche als *false* 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?

Und 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 *Schleifenbedingung* 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.

## 11.1. 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.[8] 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 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
```

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

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 gewöhnugsbedürftig 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.

## 11.2. 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
```

## 11.3. 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.

### 11.3.1. 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. 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 11.4 sehen, wie dieses Verhalten sinnvoll genutzt werden kann.

### 11.3.2. `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
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.

### 11.3.3. 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 Speicherpaltz 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 Skope 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.

### 11.3.4. 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). Analog dazu wird die Schleife natürlich auch übersprungen, wenn die Schrittweite negativ und der Startwert kleiner als der Endwert ist.

### 11.3.5. 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, also z. B. bei 0.5 abwärts auf 0 und bei 1.5 aufwärts auf 2. Dies ist für Statistiker vorteilhaft, weil die Werte x.5 dann im Schnitt gleich oft auf- wie abgerundet werden.

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:[9]

[9] 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.

## 11.4. 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.

### 11.4.1. 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 ür i=5 uebersprungen
  PRINT i
NEXT
PRINT "Nach der Schleife: i ="; i
SLEEP
```

Ausgabe

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

### 11.4.2. 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 im Kapitel 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
  ' Abbruchmoeglichkeit
  IF INKEY <> "" THEN EXIT FOR
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!

### 11.4.3. 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.

## 11.5. 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?

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 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 Hauptprogramms überschreibt. Trotzdem können Daten zwischen Haupt- und Unterprogramm ausgetauscht werden. Das geschieht in der Regel durch den Einsatz von Parametern.

## 12.1. Einfache Prozeduren

Eine Prozedur wird in FreeBASIC **SUB**[10] 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 Produktion genannt. Die drei Zeilen zwischen **SUB** und **END SUB** stellen den *Rumpf* der Prozedur dar.

[10] 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 Programms 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 Hauptprogramms 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 wird man vermutlich nicht so häufig die Welt grüßen, sondern stattdessen möglicherweise einen anderen Text ausgeben wollen. 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 Unterprogramms 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 Unterprogramms 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 bevorzugen. 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 Unterprogramms 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 Unterprogramms). Aber auch im Hauptprogramm definierte Variablen sind lokal – sie gelten nur innerhalb des Hauptprogramms. 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**.

```
DECLARE SHARED AS datentyp variable1, variable2
DECLARE 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. In Quelltext 12.3 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 Hauptprogramms. `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 Unterprogramms gültig sind, deren Wert aber nach Beendigung des Unterprogramms

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 Unterprogramms – 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 Programms 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 Unterprogramms, nur mit vorangestelltem Schlüsselwort **DECLARE**. Allerdings wird der Inhalt des Unterprogramms an dieser Stelle noch nicht festgelegt. Die **DECLARE**-Zeile dient nur dazu, dem Programm mitzuteilen: Irgendwann später wird die Definition eines Unterprogramms mit diesem Namen und dieser Parameterliste folgen. Ein Aufruf kann dann auch schon erfolgen, bevor das Unterprogramm festgelegt wird – das Programm 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 Unterprogramms 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 Programms 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 gegenseitig aufrufen. Quelltext 12.8 ist kein übermäßig sinnvolles Programm, sollte aber das Prinzip veranschaulichen.

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 Programm 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 Unterprogramms 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). Wichtig ist nur, dass 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 Funktionen (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, damit 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 in Frage 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.

**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 selbstverständlich 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 Rüchgabewert in einer Variablen gespeichert oder anderweitig ausgewertet wird.

**Hinweis:**
Während beim Aufruf von Prozeduren keine Klammern um die Parameterliste gesetzt werden muss, sind diese 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. 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. Die Funktion reagiert sehr flexibel auf das übergebene Array. Zumindest funktioniert das

gut, solange ein *eindimensionales* Array übergeben wird. Die Funktion erkennt die Anzahl der Dimensionen nicht. Bei Übergabe eines mehrdinemsionalen 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 der Funktion mit der korrekten Anzahl an **ANY** versehen werden.

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 einen genau definierten 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 Unterprogramms 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 Unterprogramms 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 12.13: 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, wird automatisch festgelegt, wie die Parameter übergeben werden:

- 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 QBasic:**
Das Standardverhalten von FreeBASIC unterscheidet sich von QBasic 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 Unterprogramms 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 Unterprogramms erfolgen sollte.

### 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 Unterprogramms nur lesend zugegriffen werden. Sollten Sie dennoch versuchen, den Wert des Parameters zu

ändern, quittiert der Compiler dies durch eine Fehlermeldung.[11] **AS CONST** ist jedoch nicht auf Arrays beschränkt.

Quelltext 12.14: 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 12.14 einen korrekten Quellcode dar. Wenn Sie jedoch das Kommentarzeichen in der vorletzten Zeile entfernen, können Sie den Codeschnipsel nicht mehr compilieren.

### 12.5.4. Variable Parameterlisten

Sie können, wenn Sie wollen, auch eine vollkommen freie Parameterliste verwenden, womit Sie beim Aufruf des Unterprogramms weder in der Anzahl noch in 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 Unterprogramms sieht z. B. folgendermaßen aus:

```
SUB variabel(x AS INTEGER, y AS INTEGER, ...)
```

Die drei Punkte, auch *Ellipse* genannt, 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 Unterprogramms lediglich die Lage der Parameter im Speicher ermitteln können, jedoch keine Möglichkeit haben, die Datentypen oder auch nur die Anzahl der Parameter festzustellen.

Der Pointer auf den ersten „variablen“ Parameter kann mit **VA_FIRST** bestimmt werden. Wenn Sie den Datentyp dieses Parameters kennen, können Sie mit **VA_ARG** seinen Wert ermitteln. Danach müssen Sie sich nach und nach durch die Parameterliste

[11] 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.

hangeln, indem Sie mit **VA_NEXT** den Pointer auf den nächsten Parameter bestimmen.

Eine Mittelwertbestimmung mit variabler Parameterliste könnte folgendermaßen aussehen:

Quelltext 12.15: Mittelwertsbestimmung mit variabler Parameterliste

```
FUNCTION mittelwert (anz AS INTEGER, ...) AS DOUBLE
  DIM AS INTEGER PTR argument = VA_FIRST  ' Pointer zum ersten variablen Argument holen
  DIM AS INTEGER summe = 0

  FOR i AS INTEGER = 1 TO anz
    summe += VA_ARG(argument, INTEGER)     ' aktuellen Wert holen und aufaddieren
    argument = VA_NEXT(argument, INTEGER) ' Pointer auf den ächsten Wert setzen
  NEXT
  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. `argument` ist ein Pointer, der mit `VA_FIRST` auf den ersten Parameter der variablen Liste gesetzt wird. Wenn sichergestellt ist, dass es sich bei allen Werten um **INTEGER** handelt, liefert `VA_ARG(argument, INTEGER)` den **INTEGER**-Wert dieses Parameters. Wird ein anderer Datentyp verwendet, muss natürlich dieser statt **INTEGER** eingesetzt werden.

Die Speicherposition des nächsten Parameters ermittelt man mit `VA_NEXT(argument, INTEGER)`. Auch hier ist die korrekte Angabe des (Vorgänger-)Datentyps erforderlich, weil anhand dessen die Datengröße bestimmt wird, um die der Zeiger weitergesetzt werden muss. Angegeben wird also nicht der Datentyp des *nächsten* Parameters, zu dem man gelangen möchte, sondern des *aktuellen* Parameters, den man verlassen will.

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

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 12.15 bereits eine Umsetzung 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 12.16.

Quelltext 12.16: Variable Parameterliste mit Formatstring

```
SUB varlist(formatstring AS STRING, ...)
  DIM AS ANY PTR argument = VA_FIRST
  DIM AS INTEGER i
  DIM AS ZSTRING PTR p

  FOR i = 1 TO LEN(formatstring)
    SELECT CASE MID(formatstring, i, 1)
      CASE "i"
        PRINT "INTEGER:  " & VA_ARG(argument, INTEGER)
        argument = VA_NEXT(argument, INTEGER)
      CASE "d"
        PRINT "DOUBLE:   " & VA_ARG(argument, DOUBLE)
        argument = VA_NEXT(argument, DOUBLE)
      CASE "s"
        PRINT "STRING:   " & *VA_ARG(argument, ZSTRING PTR)
        argument = VA_NEXT(argument, ZSTRING PTR)
    END SELECT
  NEXT
END SUB

DIM s AS STRING = "String2"

varlist "idss", 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 **??** ausführlich behandelt.

# Teil III.

# 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 **LONGINT** nötig, oder bei der Verwendung des 64bit-Compilers 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 möglichen Problemen bei der Genauigkeit nicht empfohlen.

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.

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.

## Fragen zu Kapitel 7

1. Wenn Sie das UDT nur nutzen, um über den Memberzugriff `udtname.recordname` die Records zu lesen bzw. zu schreiben, spielt die Reihenfolge bei der Deklaration keine Rolle. Das Programm erkennt selbst, auf welche Speicherstellen es zugreifen muss, und Sie müssen sich um die genaue Struktur des Speicheraufbaus keine Sorgen machen. Schließlich verwenden wir ja auch gerade deshalb eine höhere Programmiersprache, um uns nicht ständig mit den Internas herumschlagen zu müssen.
Wenn Sie sich aus irgendeinem Grund mit dem Speicheraufbau beschäftigen wollen oder müssen, wird die Reihenfolge aber durchaus interessant. Zum einen kann durch eine ungünstig gewählte Reihenfolge der Speicherbedarf unnötig wachsen. Zum anderen macht es Sinn, logisch zusammengehörende Elemente auch nebeneinander zu stellen. Das wird in der Regel auch zu einem leichter verständlichen Quelltext führen, als wenn alle Records bunt durcheinander gewürfelt werden.

2. Der Header soll sowohl das neue FreeBASIC-Format als auch das alte QBasic-Format gleichermaßen unterstützen. QBasic verwendet jedoch ausschließlich das

Padding 1. Um eine Kompatibilität zu einer im QBasic-Format gespeicherten Grafik zu erreichen, ist daher die Angabe `FIELD=1` erforderlich.

## 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 eingschlossen.

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

TODO ...

## 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. In FreeBASIC gibt es keinen eigenen Bool-Datentyp. Stattdessen werden alle Zahlenwerte außer 0 (Null) als *true* interpretiert, 0 dagegen als *false*. Strings können nicht als Bool-Werte interpretiert werden.
   Werden die Werte von Programm selbst vergeben, verwendet FreeBASIC $-1$ für *true* und 0 für *false*.

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. 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 = "Stephan" 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 = "Stephan" 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 = "Stephan" AND passwort = "supersicher") _
   OR (benutzername = "Maria" 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 saht 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 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.

```
' 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."

' Namensausgabe
FOR i AS INTEGER = UBOUND(nachname) TO 0 STEP -1
  PRINT nachname(i)
NEXT
SLEEP
```

# B. 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 | ∈ |
| 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 | | |

# C. 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 | Konstante | hex | dez | Konstante | hex | dez |
|---|---|---|---|---|---|---|---|---|
| SC_ESCAPE | 01 | 1 | SC_A | 1E | 30 | SC_F1 | 3B | 59 |
| SC_1 | 02 | 2 | SC_S | 1F | 31 | SC_F2 | 3C | 60 |
| SC_2 | 03 | 3 | SC_D | 20 | 32 | SC_F3 | 3D | 61 |
| SC_3 | 04 | 4 | SC_F | 21 | 33 | SC_F4 | 3E | 62 |
| SC_4 | 05 | 5 | SC_G | 22 | 34 | SC_F5 | 3F | 63 |
| SC_5 | 06 | 6 | SC_H | 23 | 35 | SC_F6 | 40 | 64 |
| SC_6 | 07 | 7 | SC_J | 24 | 36 | SC_F7 | 41 | 65 |
| SC_7 | 08 | 8 | SC_K | 25 | 37 | SC_F8 | 42 | 66 |
| SC_8 | 09 | 9 | SC_L | 26 | 38 | SC_F9 | 43 | 67 |
| SC_9 | 0A | 10 | SC_SEMICOLON | 27 | 39 | SC_F10 | 44 | 68 |
| SC_0 | 0B | 11 | SC_QUOTE | 28 | 40 | SC_NUMLOCK | 45 | 69 |
| SC_MINUS | 0C | 12 | SC_TILDE | 29 | 41 | SC_SCROLLLOCK | 46 | 70 |
| SC_EQUALS | 0D | 13 | SC_LSHIFT | 2A | 42 | SC_HOME | 47 | 71 |
| SC_BACKSPACE | 0E | 14 | SC_BACKSLASH | 2B | 43 | SC_UP | 48 | 72 |
| SC_TAB | 0F | 15 | SC_Z | 2C | 44 | SC_PAGEUP | 49 | 73 |
| SC_Q | 10 | 16 | SC_X | 2D | 45 | SC_LEFT | 4B | 75 |
| SC_W | 11 | 17 | SC_C | 2E | 46 | SC_RIGHT | 4D | 77 |
| SC_E | 12 | 18 | SC_V | 2F | 47 | SC_PLUS | 4E | 78 |
| SC_R | 13 | 19 | SC_B | 30 | 48 | SC_END | 4F | 79 |
| SC_T | 14 | 20 | SC_N | 31 | 49 | SC_DOWN | 50 | 80 |
| SC_Y | 15 | 21 | SC_M | 32 | 50 | SC_PAGEDOWN | 51 | 81 |
| SC_U | 16 | 22 | SC_COMMA | 33 | 51 | SC_INSERT | 52 | 82 |
| SC_I | 17 | 23 | SC_PERIOD | 34 | 52 | SC_DELETE | 53 | 83 |
| SC_O | 18 | 24 | SC_SLASH | 35 | 53 | SC_F11 | 57 | 87 |
| SC_P | 19 | 25 | SC_RSHIFT | 36 | 54 | SC_F12 | 58 | 88 |
| SC_LEFTBRACKET | 1A | 26 | SC_MULTIPLY | 37 | 55 | SC_LWIN | 7D | 125 |
| SC_RIGHTBRACKET | 1B | 27 | SC_ALT | 38 | 56 | SC_RWIN | 7E | 126 |
| SC_ENTER | 1C | 28 | SC_SPACE | 39 | 57 | SC_MENU | 7F | 127 |
| SC_CONTROL | 1D | 29 | SC_CAPSLOCK | 3A | 58 | | | |

# D. 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 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 | Stelligkeit | Bedeutung | Assoziativität |
|---|---|---|---|---|
| **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 | Stelligkeit | Bedeutung | Assoziativität |
|---|---|---|---|---|
| **@** | 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 |

# E. 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 v1.01 Verwendung finden (Dialektversion *fb*) sowie die in dieser Version reservierten, aber nicht verwendbaren Schlüsselwörter.

## E.1. Schlüsselwörter

**ABS**
gibt den Absolutbetrag der angegebenen Zahl zurück.

**ACCESS**
wird zusammen mit **OPEN** verwendet und legt die Zugriffsrechte auf die Datei fest.

**ACOS**
gibt den Arcuskosinus einer Zahl zurück.

**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.
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.

**ANY**
wird zusammen mit **DECLARE**, **DIM**, **INSTR**, **TRIM**, **LTRIM** und **RTRIM** benutzt. Je nach Einsatzart hat es eine andere Bedeutung.

**APPEND**
wird zusammen mit **OPEN** verwendet, um eine Datei zu öffnen, an die weitere Daten angehängt werden sollen.

**AS**
wird zusammen mit verschiedenen Befehlen verwendet, um den Datentyp einer Variablen oder eines Records anzugeben oder die Dateinummer einer zu öffnenden Datei festzulegen.

**ASC**
liefert den ASCII-Code des Zeichens.

**ASIN**
gibt den Arcussinus einer Zahl zurück.

**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.

**ATN**
gibt den Arcustangens einer Zahl zurück.

**BASE**
gibt im Zusammenhang mit Vererbung die Möglichkeit, auf ein Record der Eltern-Klasse zuzugreifen, selbst wenn die eigene Klasse ein Record mit gleichem Namen besitzt.

**BEEP**
weist das System an, ein Tonsignal auszugeben.

**BIN**
gibt den binären Wert eines Ausdrucks zurück.

**BINARY**
wird zusammen mit **OPEN** verwendet, um eine Datei im Binary-Modus zu öffnen.

**BIT**
prüft, ob in einem Ausdruck das Bit an der angegebenen Stelle gesetzt ist.

**BITRESET**
gibt den Wert eines *Ausdruck* zurück, bei dem das Bit an der angegebenen Stelle gelöscht wurde.

**BITSET**
gibt den Wert eines *Ausdruck* zurück, bei dem das Bit an der angegebenen Stelle gesetzt wurde.

**BLOAD**
lädt einen Block binärer Daten (z. B. Bilder) aus einer Datei.

**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.

**BYTE**
eine vorzeichenbehaftete 8-bit-Ganzzahl.

**BYVAL**
legt fest, dass ein Parameter als Wert statt als direkte Referenz übergeben werden soll.

**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.

**CBYTE**
konvertiert einen numerischen Ausdruck zu einem **BYTE**.

**CDBL**
konvertiert einen numerischen Ausdruck zu einem **DOUBLE**.

**CDECL**
setzt die Aufrufkonvention der Parameter auf `C-DECLARE` (Übergabe von rechts nach links.

**CHAIN**
übergibt die Kontrolle an ein anderes Programm und startet dieses.

**CHDIR**
ändert das aktuelle Arbeitsverzeichnis oder -laufwerk.

**CHR**
verwandelt einen ASCII-Code in seinen Character.

**CINT**
konvertiert einen numerischen Ausdruck zu einem **INTEGER**.

**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**.

**CLNGINT**
verwandelt einen numerischen Ausdruck zu einem **LONGINT**.

**CLOSE**
schließt Dateien, die zuvor mit **OPEN** geöffnet wurden.

**CLS**
löscht den Bildschirm und füllt ihn mit der Hintergrundfarbe.

**COLOR**
als Anweisung: setzt die Vorder- und Hintergrundfarbe.
als Funktion: gibt Informationen über die verwendeten Textfarben zurück.

**COM**
**OPEN COM** bereitet den Zugriff auf einen COM-Port vor.

**COMMAND**
enthält die Kommandozeilenparameter an das Programm.

**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*.

**CONST**
erzeugt eine Konstante. Kann auch bei der Parameterübergabe und in **SELECT CASE** verwendet werden, um die Verwendung konstanter Werte zu erzwingen.

**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.

**COS**
gibt den Kosinus eines Winkels im Bogenmaß zurück.

**CPTR**
verwandelt einen 32bit-Ausdruck in einen Pointer eines beliebigen Typs.

**CSHORT**
konvertiert einen numerischen Ausdruck zu einem **SHORT**.

**CSIGN**
verwandelt eine vorzeichenlose Zahl in eine vorzeichenbehaftete Zahl desselben Datentyps.

**CSNG**
konvertiert einen numerischen Ausdruck zu einem **SINGLE**.

**CSRLIN**
gibt die aktuelle Zeile des Cursors zurück.

**CUBYTE**
konvertiert einen numerischen Ausdruck zu einem **UBYTE**.

**CUINT**
konvertiert einen numerischen Ausdruck zu einem **UINTEGER**.

**CULNG**
konvertiert einen numerischen Ausdruck zu einem **ULONG**.

**CULNGINT**
konvertiert einen numerischen Ausdruck zu einem **ULONGINT**.

**CUNSG**
konvertiert eine vorzeichenbehaftete Zahl in eine vorzeichenlose Zahl desselben Datentyps.

**CURDIR**
gibt das aktuelle Arbeitsverzeichnis aus.

**CUSHORT**
konvertiert einen numerischen Ausdruck zu einem **USHORT**.

**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.

**CVD**
konvertiert einen 8-Byte-String in eine **DOUBLE**-Zahl. Umkehrung von **MKD**.

**CVI**
konvertiert einen 4-Byte-String in eine **INTEGER**-Zahl. Umkehrung von **MKI**.

**CVL**
konvertiert einen 4-Byte-String in eine **LONG**-Zahl. Umkehrung von **MKL**.

**CVLONGINT**
konvertiert einen 8-Byte-String in eine **LONGINT**-Zahl. Umkehrung von **MKLONGINT**.

**CVS**
konvertiert einen 4-Byte-String in ein **SINGLE**.

**CVSHORT**
konvertiert einen 2-Byte-String in ein **SHORT**. Umkehrung zu **MKSHORT**.

**DATA**
speichert Konstanten im Programm, die mit **READ** eingelesen werden können.

**DATE**
gibt das aktuelle Datum im Format `mm-dd-yyyy` zurück.

**DATEADD**
rechnet zu einem Datum ein bestimmtes Zeitintervall hinzu und gibt das neue Datum zurück *(benötigt datetime.bi)*.

**DATEDIFF**
berechnet den zeitlichen Abstand zweier Datumsangaben *(benötigt datetime.bi).*

**DATEPART**
extrahiert eine Teilangabe aus einer gegebenen Serial Number *(benötigt datetime.bi).*

**DATESERIAL**
wandelt ein angegebenes Datum in eine Serial Number um *(benötigt datetime.bi).*

**DATEVALUE**
verwandelt einen String mit einer Datumsangabe in eine Serial Number um *(benötigt datetime.bi).*

**DAY**
extrahiert den Tag aus einer Serial Number *(benötigt datetime.bi).*

**DEALLOCATE**
gibt einen mit **ALLOCATE** reservierten Speicher wieder frei.

**DECLARE**
deklariert eine neue Prozedur, um sie dem Compiler bekannt zu machen.

**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.

**DIR**
gibt die Dateien im aktuellen Arbeitsverzeichnis oder im angegebenen Pfad zurück.

**DO...LOOP**
wiederholt einen Anweisungsblock, während bzw. bis eine Bedingung erfüllt ist.

**DOUBLE**
eine Gleitkommazahl mit doppelter Genauigkeit (64 Bit).

**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.

**ELSEIF**
wird im Zusammenhang mit **IF...THEN** verwendet. Es beschreibt einen vom ersten Prüffall unterschiedlichen Fall.

**ENCODING**
wird zusammen mit **OPEN** verwendet, um festzulegen, in welcher Zeichenkodierung die Daten behandelt werden sollen.

**END**
beendet eine Blockstruktur (Prozedur, Schleife ...) oder das Programm.

**ENDIF**
kann aus Gründen der Kompatibilität zu anderen BASIC-Dialekten anstelle von **END IF** verwendet werden.

**ENUM**
erzeugt eine Liste von **INTEGER**-Konstanten vom Typ **INTEGER**.

**ENVIRON**
gibt den Wert einer Systemumgebungsvariablen zurück.

**EOF**
gibt -1 zurück, wenn der Dateizeiger das Ende einer geöffneten Datei erreicht hat.

**EQV**
vergleicht zwei Werte Bit für Bit und setzt im Ergebnis nur dann ein Bit, wenn die entsprechenden Bits in beiden Ausdrücken gleichwertig waren.

**ERASE**
löscht dynamische Arrays aus dem Speicher oder setzt alle Elemente eines statischen Arrays auf den Initialisationswert.

**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.

**ERROR**
simuliert einen Fehler.

**EXEC**
startet eine ausführbare Datei mit den übergebenen Argumenten.

**EXEPATH**
gibt das Verzeichnis zurück, in dem sich das gerade ausgeführte Programm befindet.

**EXIT**
verlässt eine Blockstruktur (Prozedur, Schleife ...).

**EXP**
gibt die Potenz einer angegebenen Zahl zur eulerschen Zahl e zurück.

**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 Records 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.

**FIELD**
wird zusammen mit **TYPE** und **UNION** verwendet und legt dort das Padding-Verhalten fest.

**FILEATTR**
gibt Informationen über eine mit **OPEN** geöffnete Datei zurück *(benötigt file.bi)*.

**FILECOPY**
kopiert die Datei *(benötigt file.bi)*.

**FILEDATETIME**
gibt Datum und Uhrzeit, an dem die Datei zuletzt bearbeitet wurde, als Serial Number zurück *(benötigt file.bi)*.

**FILEEXISTS**
überprüft, ob eine Datei existiert oder nicht *(benötigt file.bi)*.

**FILELEN**
gibt die Länge von einer Datei in Bytes zurück *(benötigt file.bi)*.

**FIX**
schneidet den Nachkommateil einer Zahl ab.

**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.
Das Schlüsselwort **FOR** wird außerdem bei **OPEN** benötigt.

**FORMAT**
wandelt einen numerischen Ausdruck anhand der angegebenen Formatierung in einen **STRING** um.

**FRAC**
gibt die Nachkommastellen inklusive Vorzeichen einer Zahl zurück.

**FRE**
gibt den verfügbaren RAM-Speicher in Bytes zurück.

**FREEFILE**
gibt die nächste unbenutzte Dateinummer zurück.

**FUNCTION**
definiert ein Unterprogramm mit Rückgabewert.

**GET**
als Dateioperation: liest Daten aus einer Datei.
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.

**HEX**
gibt den hexadezimalen Wert eines numerischen Ausdrucks als **STRING** zurück.

**HIBYTE**
gibt das obere Byte eines Ausdrucks als **UINTEGER** zurück.

**HIWORD**
gibt das obere Word eines Ausdrucks als **UINTEGER** zurück.

**HOUR**
extrahiert die Stunde einer Serial Number *(benötigt datetime.bi).*

**IF...THEN**
führt einen Codeteil nur dann aus, wenn eine Bedingung erfüllt ist.

**IIF**
liefert einen von zwei Werten zurück, abhängig davon, ob die Bedingung erfüllt ist oder nicht.

**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.

**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.

**INP**
liest ein Byte von einem Port.

**INPUT**
als Anweisung: liest eine Eingabe von der Tastatur oder aus einer Datei.
als Funktion: liest eine Anzahl an Zeichen von der Tastatur oder aus einer Datei.
als Dateimodus: wird zusammen mit **OPEN** verwendet, um eine Datei zum sequentiellen Lesen zu öffnen.

**INSTR**
liefert die Stelle, an der ein String das erste Mal in einem anderen String1 vorkommt.

**INSTRREV**
liefert die Stelle, an der ein String das letzte Mal in einem anderen String vorkommt.

**INT**
rundet einen numerischen Ausdruck auf die nächstkleinere Ganzzahl ab.

**INTEGER**
eine vorzeichenbehaftete 32-bit-Ganzzahl.

**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.

**ISDATE**
überprüft, ob ein String einem korrekten Datumsformat entspricht.

**KILL**
löscht eine Datei von einem Datenträger.

**LBOUND**
gibt den kleinsten Index des angegebenen Arrays zurück.

**LCASE**
wandelt einen Stringausdruck in Kleinbuchstaben.

**LEFT**
gibt die ersten Zeichen eines Strings zurück.

**LEN**
gibt die Größe eines Stringausdrucks oder eines Datentyps zurück.

**LET**
ermöglicht eine mehrfache Variablenzuweisung, bei der einer Liste von Variablen die Werte der einzelnen Records 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.

**LOBYTE**
gibt das niedere Byte eines Ausdrucks als **UINTEGER** zurück.

**LOC**
gibt die Position des Zeigers innerhalb einer mit **OPEN** geöffneten Datei zurück.

**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.

**LOCK**
sperrt den Zugriff auf eine Datei.
*Achtung: funktioniert zur Zeit nicht wie vorgesehen!*

**LOF**
gibt die Länge einer geöffneten Datei in Bytes zurück.

**LOG**
gibt den natürlichen Logarithmus (zur Basis e) zurück.

**LONG**
eine vorzeichenbehaftete 32-bit-Ganzzahl.

**LONGINT**
eine vorzeichenbehaftete 64-bit-Ganzzahl.

**LOOP**
beendet einen **DO...LOOP**-Block.

**LOWORD**
gibt das niedere Word eines Ausdrucks als **UINTEGER** zurück.

**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.

**LPT**
**OPEN LPT** bereitet den Drucker zum Datenempfang vor.

**LSET**
befüllt einen Zielstring mit dem Inhalt eines Quellstrings, behält aber die Länge des Zielstrings bei. Kann auch mit UDTs verwendet werden.

**LTRIM**
entfernt bestimmte führende Zeichen aus einem String.

**MID**
gibt einen Ausschnitt einer Zeichenkette zurück oder ersetzt einen Teil einer Zeichenkette durch eine andere.

**MINUTE**
extrahiert die Minute einer Serial Number *(benötigt datetime.bi)*.

**MKD**
verwandelt ein **DOUBLE** in einen 8-Byte-**STRING**. Umkehrung von **CVD**.

**MKDIR**
erstellt einen Ordner.

**MKI**
verwandelt ein **INTEGER** in einen 4-Byte-**STRING**. Umkehrung von **CVI**.

**MKL**
verwandelt ein **LONG** in einen 4-Byte-**STRING**. Umkehrung von **CVL**.

**MKLONGINT**
verwandelt ein **LONGINT** in einen 8-Byte-**STRING**. Umkehrung von **CVLONGINT**.

**MKS**
verwandelt ein **SINGLE** in einen 4-Byte-**STRING**. Umkehrung von **CVS**.

**MKSHORT**
verwandelt ein **SHORT** in einen 2-Byte-**STRING**. Umkehrung von **CVSHORT**.

**MOD**
gibt den Rest der Division zurück (Modulo).

**MONTH**
extrahiert den Monat einer Serial Number *(benötigt datetime.bi)*.

**MONTHNAME**
extrahiert den Namen des Monats einer Serial Number *(benötigt datetime.bi)*.

**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.

**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.

**NOW**
gibt die aktuelle Systemzeit als Serial Number aus *(benötigt datetime.bi).*

**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.

**OFFSETOF**
gibt den Offset (Abstand in Byte zur Adresse des UDTs) eines Records innerhalb eines UDTs zurück.

**ON...GOTO**
verzweigt zu verschiedenen Labels, abhängig vom Wert des Ausdrucks.

**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.

**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.
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.

**OUT**
schreibt ein Byte an einen Ausgabeport.

**OUTPUT**
wird zusammen mit **OPEN** verwendet, um eine Datei zum sequentiellen Schreiben zu öffnen.

**OVERLOAD**
ermöglicht die Definition von Prozeduren mit unterschiedlicher Parameterliste, aber gleichem Prozedurnamen.

**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.

**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.

**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**.

**POKE**
schreibt einen Wert direkt in den RAM.

**POS**
gibt die horizontale Position des Cursors zurück.

**PRESERVE**
wird zusammen mit **REDIM** benutzt, um ein Array zu redimensionieren, ohne seine Elemente zu löschen.

**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.

**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**.

**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.
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.

**RANDOMIZE**
initialisiert den Zufallsgenerator.

**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.

**REM**
leitet einen Kommentar ein.

**RESET**
schließt alle geöffneten Dateien oder setzt die Standardeingabe bzw. -ausgabe zurück.

**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**.

**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.

**RMDIR**
löscht ein leeres Verzeichnis aus dem Dateisystem.

**RND**
gibt ein zufälliges **DOUBLE** im Intervall $[0; 1[$ zurück.

**RSET**
befüllt einen Zielstring mit dem Inhalt eines Quellstrings, behält aber die Länge des Zielstrings bei.

**RTRIM**
entfernt bestimmte angehängte Zeichen aus einem String.

**RUN**
startet eine ausführbare Datei.

**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.

**SECOND**
extrahiert die Sekunde einer Serial Number *(benötigt datetime.bi)*.

**SEEK**
setzt die Position des Zeigers innerhalb einer Datei oder gibt die Position zurück.

**SELECT CASE**
führt, abhängig vom Wert eines Ausdrucks, bestimmte Codeteile aus.

**SETDATE**
ändert das Systemdatum.

**SETENVIRON**
verändert die Systemumgebungsvariablen.

**SETMOUSE**
setzt die Koordinaten des Mauscursors und bestimmt, ob die Maus sichtbar oder unsichtbar ist.

**SETTIME**
setzt die neue Systemzeit.

**SGN**
gibt einen Wert aus, der das Vorzeichen eines Ausdrucks identifiziert.

**SHARED**
bewirkt im Zusammenhang mit **DIM**, **REDIM**, **COMMON** und **STATIC**, dass Variablen sowohl auf Modulebene als auch innerhalb von Prozeduren verfügbar sind.

**SHELL**
führt ein Systemkommando aus und gibt die Kontrolle an das aufrufende Programm zurück, sobald das aufgerufene Kommando abgearbeitet wurde.

**SHL**
verschiebt alle Bits in der Variablen um eine bestimmte Stellenzahl nach links.

**SHORT**
eine vorzeichenbehaftete 16-bit-Ganzzahl.

**SHR**
verschiebt alle Bits in der Variablen um eine bestimmte Stellenzahl nach rechts.

**SIN**
gibt den Sinus eines Winkels im Bogenmaß zurück.

**SINGLE**
eine Gleitkommazahl mit einfacher Genauigkeit (32 Bit).

**SIZEOF**
gibt die Größe einer Struktur im Speicher in Bytes aus.

**SLEEP**
wartet eine bestimmte Zeit oder bis eine Taste gedrückt wird.

**SPACE**
gibt einen **STRING** zurück, der aus Leerzeichen besteht.

**SPC**
wird im Zusammenhang mit **PRINT** verwendet, um Texteinrückungen zu erzeugen.

**SQR**
gibt die Quadratwurzel eines Wertes aus.

**STATIC**
erlaubt in Prozeduren die Verwendung von Variablen, deren Wert beim Beenden der Prozedur gespeichert wird und beim nächsten Prozeduraufruf wieder verfügbar sind. 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).

**STEP**
gibt in einer **FOR**-Schleife die Schrittweite der Zählvariablen an.
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**.

**STRING**
eine Zeichenkette variabler oder fester Länge.
Als Funktion gibt **STRING** eine Zeichenkette mit lauter gleichen Zeichen zurück.

**STRPTR**
gibt einen Pointer zu einer String-Variablen zurück.

**SUB**
definiert ein Unterprogramm.

**SWAP**
tauscht die Werte zweier Variablen vom selben Typ.

**SYSTEM**
beendet das Programms; sollte durch **END** ersetzt werden.

**TAB**
wird zusammen mit **PRINT** benutzt, um die Ausgabe in der angegebenen Spalte fortzusetzen.

**TAN**
gibt den Tangens eines Winkels im Bogenmaß zurück.

**THEN**
wird zusammen mit **IF...THEN** verwendet.

**THIS**
wird in Prozeduren eines UDTs verwendet, um auf die Records 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.

**TIMER**
gibt die Zahl der Sekunden zurück, die seit dem Systemstart (unter DOS/Windows) bzw. seit der Unix-Epoche (unter Unix/Linux) vergangen sind.

**TIMESERIAL**
wandelt eine angegebene Uhrzeit in eine Serial Number um *(benötigt datetime.bi)*.

**TIMEVALUE**
wandelt einen **STRING** mit einer Zeitangabe in eine Serial Number um *(benötigt datetime.bi)*.

**TO**
gibt Bereiche für die Befehle **FOR...NEXT**, **DIM**, **SELECT CASE** und **LOCK** an.

**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.

**TYPE**
erstellt ein UDT (user defined type).

**TYPEOF**
gibt zur Compilierzeit den Datentypen zurück.

**UBOUND**
gibt den größten Index des angegebenen Arrays zurück.

**UBYTE**
eine vorzeichenlose 8-bit-Ganzzahl.

**UCASE**
wandelt einen Stringausdruck in Großbuchstaben.

**UINTEGER**
eine vorzeichenlose 32-bit-Ganzzahl.

**ULONG**
eine vorzeichenlose 32-bit-Ganzzahl.

**ULONGINT**
eine vorzeichenlose 64-bit-Ganzzahl.

**UNION**
definiert einen **UDT**, dessen Elemente sich eine Speicherstelle teilen.

**UNLOCK**
entsperrt eine mit **LOCK** gesperrte Datei.

**UNSIGNED**
erzeugt einen ganzzahligen Datentyp, der vorzeichenlos ist.

**UNTIL**
wird mit **DO...LOOP** verwendet.

**USHORT**
eine vorzeichenlose 16-bit-Ganzzahl.

**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.

**VA_FIRST**
gibt einen Pointer auf den ersten Parameter einer variablen Parameterliste zurück.

**VA_NEXT**
setzt den Pointer, der auf ein Argument einer variablen Parameterliste zeigt, auf das nächste Argument.

**VAL**
konvertiert einen **STRING** zu einer Zahl.

**VALINT**
konvertiert einen **STRING** zu einem **INTEGER**.

**VALLNG**
konvertiert einen **STRING** zu einem **LONGINT**.

**VALUINT**
konvertiert einen **STRING** zu einem **UINTEGER**.

**VALULNG**
konvertiert einen **STRING** zu einem **ULONGINT**.

**VAR**
deklariert eine Variable, deren Typ aus dem initialisierenden Ausdruck abgeleitet wird.

**VARPTR**
gibt die Adresse einer Variablen im Speicher zurück.

**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.

**WCHR**
verwandelt einen Unicode-Wert in seinen Character.

**WEEKDAY**
extrahiert den Wochentag (1 bis 7) aus einer Serial Number *(benötigt datetime.bi)*.

**WEEKDAYNAME**
gibt den Namen eines Wochentags aus *(benötigt datetime.bi)*.

**WEND**
beendet einen **WHILE...WEND**-Block.

**WHEX**
gibt den hexadezimalen Wert eines numerischen Ausdrucks als **WSTRING** zurück.

**WHILE**
wird mit **DO...LOOP** und bei einer **WHILE...WEND**-Schleife verwendet.

**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.

**WITH**
erlaubt es, auf die Records und Methoden eines UDTs zuzugreifen, ohne den Namen des UDTs mit angeben zu müssen.

**WOCT**
gibt den oktalen Wert eines Ausdrucks als **WSTRING** zurück.

**WRITE**
gibt einen Text auf dem Bildschirm oder in eine Datei aus, formatiert dabei aber anders als **PRINT**.

**WSPACE**
gibt einen **WSTRING** zurück, der aus Leerzeichen besteht.

**WSTR**
verwandelt einen numerischen Ausdruck in einen **WSTRING**.

**WSTRING**
eine nullterminierte wide-chars-Zeichenkette.
Als Funktion gibt **WSTRING** eine Zeichenkette mit lauter gleichen wide-chars zurück.

**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.
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)*.

**ZSTRING**
eine nullterminierte Zeichenkette.

## E.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

**#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

## E.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 0.24.0*

**__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. 0

**__FB_VER_MINOR__**
enthält die Version des Compilers enthält, z. B. 24

**__FB_VER_PATCH__**
enthält die Version des Compilers enthält, z. B. 0

**__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

# Liste der Quelltexte