Hacker Basics

ISBN 3-8272-5796-4

Kapitel 2

Einführung in QBASIC

In diesem Kapitel lernst du endlich deine erste Programmiersprache kennen, QBASIC. Obwohl du später wahrscheinlich nicht mehr mit QBASIC programmieren wirst, sondern mit leistungsfähigeren Programmiersprachen wie Visual Basic oder C++, ist QBASIC wahrscheinlich am besten für den Einstieg geeignet, weil es sehr leicht zu verstehen und ziemlich unkompliziert ist. Und ich versichere dir, dass dir alles, was du in diesem Kapitel lernst, später nützlich sein wird, weil sich alle Programmiersprachen im Grunde ziemlich ähnlich sind.

Installation

Normalerweise ist QBASIC auf dem Computer mit dem Betriebssystem Windows schon installiert. Du kannst das ganz einfach testen:

- Klicke im Startmenü auf *Suchen* und dann auf *Dateien/Ordner.*
- Gib den Suchbegriff »qbasic« ein, und bestätige mit der ↵-Taste.

Nach kurzer Zeit sollten zwei Dateien namens »qbasic.exe« und »qbasic.hlp« gefunden werden.

Mach am besten gleich eine Verknüpfung auf dem Desktop zur Datei »qbasic.exe«, damit du sie später wieder findest. Dazu klickst du die angezeigte Datei einmal mit der rechten Maustaste an und ziehst sie einfach mit gedrückter Maustaste auf den Desktop. Klicke im erscheinenden Menü auf den Befehl *Verknüpfung hier erstellen*, und du bist fertig!

Wenn das Suchprogramm die beiden Dateien nicht gefunden hat, musst du sie dir erst von der Windows-CD kopieren. Dazu legst du die Windows-CD in das CD-ROM-Laufwerk und kopierst die Dateien vom Ordner *\tools\oldmsdos* auf der CD in das Verzeichnis *\Dos* auf der Festplatte.

Damit hast du alles, was du brauchst um mit QBASIC Programme zu schreiben.

QBASIC starten

Um QBASIC zu starten, machst du einen Doppelklick auf die Verknüpfung auf deinem Desktop, die du zuvor erstellt hast. Nach einer letzten Parameter-Abfrage (unter Windows 98) sollte der QBASIC Bildschirm erscheinen.

Abbildung 2.1: Der QBASIC-Bildschirm

Drücke jetzt die Esc-Taste, und der QBASIC-Editor erscheint.

Der QBASIC-Editor

Abbildung 2.2: Der Editor ist gestartet.

Das große Feld, das fast den ganzen Bildschirm ausfüllt, ist das Codefenster. Dort gibst du den Programmcode ein.

Der untere Teil des Bildschirms ist das Direktfenster. Im Direktfenster kannst du ganz unabhängig davon, was im Codefenster steht, arbeiten, zum Beispiel Werte ausrechnen oder Befehle ausprobieren, oder, während du ein Programm ausführst Variablen abfragen (du wirst bald erfahren, was Variablen sind):

- Schreibe den Befehl, und drücke [↵].
- Gib die Anweisung PRINT oder ein Fragezeichen ein und dahinter die Variable, deren Wert du wissen willst. Das Direktfenster gibt den Wert der Variable aus.
- Schreibe PRINT oder ? und die Rechnung, deren Ergebnis du wissen willst.

Beispiele:

```
PRINT "Das ist ein Test"
```

Oder besser:

```
? "Das ist ein Test"
? 20 * 3 / 1.5
? 125 * 16%
```

Die Umgebung ist für dich vielleicht ungewohnt, wenn du vorher mit Windows gearbeitet hast, aber du wirst dich bald daran gewöhnen, da es ähnlich wie ein normales Windows-Fenster ist:

Oben siehst du die Titelleiste, wie es sie in fast jedem Windows Programm auch gibt. Dort kannst du z. B. deine Programme speichern oder öffnen.

Du hast auch einen Cursor, den du mit der Maus oder den Pfeiltasten steuern kannst.

Der Rest des Bildschirms ist ein großes Feld, in das der Programmcode eingegeben wird.

Mit den Tasten [Bild↑] und [Bild↓] blätterst du nach oben und unten, wenn dein Programm nicht auf den Bildschirm passt.

Wenn du ein Programm geschrieben hast, kannst du es mit [F5] ausführen.

Fensterdarstellung und Bildschirm

Im DOS-Modus sieht das QBASIC-Fenster nicht gerade einladend aus, und als Windows-Anwender wird dich die hölzerne Pixel-Auflösung schockieren. Du kannst aber das Programmfenster auch wie alle Windows-Fenster benutzen. Dazu musst du nur die Eigenschaften der aufgerufenen EXE-Datei umstellen:

- Klicke mit der rechten Maustaste auf das Verknüpfungssymbol von QBASIC.EXE.
- Wähle *Eigenschaften* aus dem Kontextmenü.
- Stelle im Register *Bildschirm* des Eigenschaftenfensters die Option *Fenster* ein, wenn du QBASIC im Windows-Fenster sehen willst.
- Überprüfe auch die übrigen Einstellungen, und passe die Schriftart und -größe nach deinen Wünschen, besser nach den Fähigkeiten deiner Grafikkarte an.

Abbildung 2.3: Bildschirm und Fenster anpassen

Die Menüs

In den Menüs findest du eine Reihe nützlicher Befehle:

Tabelle 2.1: Die Menüs im QBASIC

Datei	Hier kannst du eine neue Datei anlegen, gespeicherte Dateien mit Programmen öffnen, Dateien speichern und drucken und das Programm beenden.
Bearbeiten	Die Befehle *Ausschneiden* und *Kopieren* beziehen sich auf markierte Teile des Fensters, sie schneiden den Teil aus oder kopieren ihn in die Zwischenablage. Um etwas zu markieren, ziehst du einfach die Maus mit gedrückter Maustaste über den Code. *Einfügen* holt den kopierten Teil aus der Zwischenablage an die Cursorposition. Mit *Löschen* wird die Markierung gelöscht. *Neue Sub* erzeugt ein neues Programm, *Neue Function* eine neue Funktion (siehe unten).
Ansicht	Mit *Subs* erhältst du die Liste aller angelegten Subprocedures und Functions. *Aufteilen* macht, was der Befehl ausdrückt, er teilt das Fenster in zwei Hälften. So kannst du zwei Teile eines Programms gleichzeitig ansehen und bearbeiten, was besonders nützlich ist bei sehr langen Listings. Ein Doppelklick auf die Teilungslinie oder ein zweiter Aufruf der Menüoption entfernt die Aufteilung wieder. Mit *Ausgabebildschirm* schaltest du auf den Bildschirm um, auf dem dein Programm die Ausgaben hinterlegt (zum Beispiel die Ergebnisse der PRINT-Anweisung). Drücke eine beliebige Taste, um zum Editor zurückzukehren.

Tabelle 2.1: Die Menüs im QBASIC

Suchen	In diesem Menü findest du Suchen- und Ersetzen-Anweisungen. Gib unter *Suchen* einen Begriff ein, und starte die Suche mit *OK*. Der gefundene Begriff wird markiert, du kannst mit *Weitersuchen* den nächsten orten. Unter *Ändern* lässt sich der Suchbegriff auch durch einen anderen Begriff ersetzen.
Ausführen	Der *Start*-Befehl aktiviert das Programm, in dem der Cursor blinkt. Mit *Neustart* wird ein unterbrochenes oder laufendes Programm neu gestartet, und *Weiter* setzt das Programm nach einer Unterbrechung (z.B. Einzelschritt) fort.
Debug	Dieses Menü enthält nützliche Anweisungen zum Testen der Programme. Mit *Einzelschritt* wird eine einzelne Programmzeile gestartet, du kannst das ganze Programm schrittweise testen (drücke einfach F8 nach jedem Befehl). *Prozedurschritt* testet nur das Hauptprogramm, aufgerufene Unterprogramme werden im normalen Modus ausgeführt. *Rückverfolgung ein* solltest du einschalten, damit auch optisch angezeigt wird, welche Anweisung als nächste ausgeführt wird. *Haltepunkte* setzt du an den Anweisungen, an denen das Programm stoppen und in den Einzelschrittmodus wechseln soll. *Nächste Anweisung festlegen* sorgt dafür, dass die markierte Programmzeile als nächste ausgeführt wird.
Optionen	Unter *Bildschirmanzeige* findest du einige Optionen für Bildschirmfarben und Hervorhebungen. *Pfad für Hilfe* verweist auf den Ordner, in dem die Hilfedatei für QBASIC zu finden ist, und *Syntax überprüfen* sollte markiert sein (lässt sich ein- und ausschalten), damit jede Anweisung bei der Eingabe bereits auf Fehler geprüft wird.
Hilfe	Sieh dir den Inhalt der Hilfedatei und den Index an, wenn du alles über QBASIC erfahren willst, was du in diesem Kapitel nicht gelernt hast (wäre ja auch zu viel verlangt!). Die Programmfenster verschwinden beim Aufruf der Hilfe nach unten, du kannst sie mit einem Doppelklick auf die Trennlinie wieder aktivieren. Mit Esc verschwindet die Hilfe ebenfalls vom Bildschirm. Ein Tipp: Setze die Markierung auf den Befehl, zu dem du Hilfe suchst, und drücke die Funktionstaste F1. Die Hilfe wird prompt zu diesem Menübefehl oder zur markierten Anweisungen die passenden Texte anzeigen.

Dein erstes Programm

Du wirst jetzt dein erstes QBASIC-Programm programmieren. Es ist kein besonderes Programm, es macht nämlich nichts anderes, als dich nach deinem Namen zu fragen, um dich dann mit diesem zu begrüßen, aber du kannst dich dadurch etwas besser mit der Umgebung vertraut machen.

Starte einfach QBASIC und gib im Editor den folgenden Text ein. Achte darauf, genau das zu schreiben, was hier steht. Oft wirst du dich fragen, warum ein Programm nicht funktioniert, du wirst Gott, die Welt und Bill Gates dafür verantwortlich machen (nicht unbedingt in dieser Reihenfolge), und dann nach ewig langer Suche erkennen, dass du dich nur vertippt hat. Also: Genauigkeit ist alles.

Du musst allerdings nicht auf Groß- oder Kleinschreibung achten, wenn ein Wort klein geschrieben wird, das groß geschrieben werden müsste, ändert QBASIC es automatisch um in Großbuchstaben.

Außerdem kannst du zwischen den Wörtern so viele Leerzeichen und zwischen den Zeilen so viele Leerzeilen machen, wie du willst.

Schreib in das große Fenster des QBASIC-Editors folgendes Programm, drücke am Ende jeder Zeile die ↵-Taste:

```
REM Mein erstes QBASIC Programm
CLS
PRINT "Wie heißt du?"
INPUT Name$
PRINT "Hallo "; Name$; "!"
```

Wenn du jetzt F5 drückst, wird das Programm gestartet.

Abbildung 2.4: Das erste QBASIC-Programm

Wenn das Programm abgelaufen ist, zeigt QBASIC die Meldung *Eine beliebige Taste drücken um fortzusetzen* an, also drücke irgendeine Taste, um zum Editor zurückzukehren.

Um das Programm zu speichern, sodass du es später noch einmal ausführen kannst, klicke auf *Datei/Speichern*. Jetzt musst du noch einen Dateinamen eingeben, zum Beispiel »hallo.bas« und auf *OK* klicken.

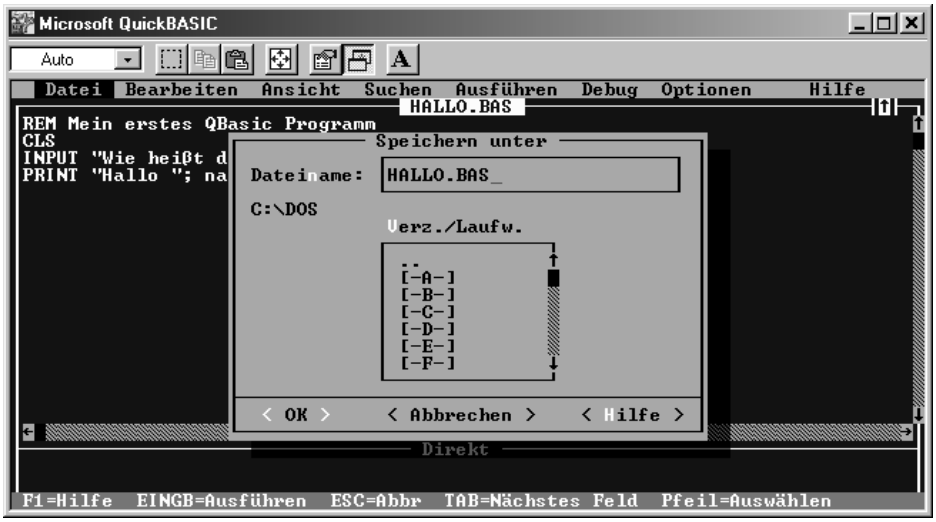

Abbildung 2.5: Das Programm wird gespeichert.

QBASIC-Programme sollten immer die Endung `.BAS` haben. Du musst diese Endung aber nicht angeben, QBASIC ergänzt die Dateiendung automatisch, es würde also auch genügen, wenn du nur »hallo« als Dateinamen angibst, die Datei würde trotzdem also »hallo.bas« gespeichert werden.

Weil wir hier unter DOS arbeiten, darf der erste Teil des Namens nicht länger als 8 Zeichen sein. Wenn du einen Dateinamen verwendest, der länger ist als 8 Zeichen, schneidet QBASIC die restlichen Zeichen ab.

Wie funktioniert das Beispielprogramm?

Du hast gerade dein erstes QBASIC-Programm erfolgreich eingegeben und ausgeführt, doch höchstwahrscheinlich verstehst du noch nicht besonders viel, von dem, was du da gerade geschrieben hast.

Also gehen wir jetzt das Programm noch einmal Schritt für Schritt durch und ich erkläre dir, wie es funktioniert:

```
REM Mein erstes QBASIC Programm
```

Diese Zeile ist so etwas wie eine Überschrift, mit dem Titel des Programms. Du könntest diese Zeile auch weglassen, weil sie keinen Einfluss auf das Programm hat und das Programm ohne sie ganz genauso läuft. Alle Zeilen, vor denen `REM` steht, werden von QBASIC völlig ignoriert. `REM` steht für *remark*, und das heißt Kommentar. Der Befehl `REM` leitet also einen Kommentar ein.

```
CLS
```

CLS ist die Abkürzung für »Clear Screen« = Lösche den Bildschirm(inhalt). Mit diesem Befehl, löscht du alles, was von vorher benutzten Programmen noch auf dem Bildschirm stand, sodass dein Programm den ganzen Platz für sich allein hat.

```
PRINT "Wie heißt du?"
```

Mit der PRINT-Anweisung wird etwas auf den Bildschirm geschrieben: das, was dahinter in Anführungszeichen steht.

```
INPUT Name$
```

Die INPUT-Anweisung nimmt Eingaben des Benutzers über die Tastatur an. Danach kommt der Ort, an dem der Computer das speichern soll, was der Benutzer hier eingibt. Das ist eine so genannte Variable mit dem Namen »Name$«. Über Variablen wirst du später mehr erfahren.

```
PRINT "Hallo "; Name$; "!"
```

Hier wird wieder mit der PRINT-Anweisung etwas auf dem Bildschirm ausgegeben. In diesem Fall ist es ein Text, »Hallo« und der Name des Benutzers, den er vorher eingegeben hat und der in der Variable *Name$* gespeichert wurde. Danach kommt noch ein Ausrufezeichen, und weil das wieder ein festgelegter Text ist, muss dieser wieder in Anführungszeichen stehen. Die einzelnen Elemente sind durch Semikolons getrennt.

Variablen

Eine Variable ist dazu da, Daten zu speichern, die später wieder abgerufen oder verarbeiten werden. Du kannst dir eine Variable wie eine Schachtel vorstellen, in die du etwas reinlegst, damit es nicht verloren geht. Du kannst in einem Programm beliebig viele Variablen benutzen.

Damit der Computer die Variablen unterscheiden kann, musst du ihnen Namen zuweisen. Du kannst die Variablen nennen, wie du willst, vorausgesetzt, du hältst dich an die folgenden Regeln:

- Sie müssen mit einem Buchstaben beginnen.
- Sie dürfen nicht länger als 40 Zeichen sein.
- Sie dürfen keine Leerzeichen oder eins der folgenden Sonderzeichen enthalten: ,; / * – + , .
- Umlaute (ä, ö, ü) sind in Variablennamen zwar erlaubt, die solltest du aber vermeiden. Mit Umlauten gibt es auf Systemebene immer Ärger.
- Es dürfen keine Schlüsselwörter sein, das sind Wörter, die von QBASIC reserviert sind, wie z.B. Befehlsnamen (du darfst eine Variable zum Beispiel nicht PRINT nennen, weil QBASIC sonst denkt, du meinst den gleichnamigen Befehl, wenn die Variable zum Einsatz kommt).

Du solltest einer Variable immer einen Namen geben, der auch ihrer Funktion entspricht, sonst kommst du in größeren Programmen schnell mit den vielen verschiedenen Variablen durcheinander. Nenne eine Variable z.B. nie »Variable1« oder »Dings«, weil du dann später nicht weißt, wozu sie gut ist, wenn du dir das Programm nach längerer Zeit mal wieder durchsiehst, um etwa einen Fehler zu korrigieren oder um es zu verbessern.

Außerdem wird unterschieden zwischen Variablen mit Zahlen als Inhalt und Variablen, deren Inhalt Zeichenketten, sogenannte Strings sind. Bei Variablen, die Zeichenketten aufnehmen sollen (String-Variablen), muss der Name am Ende das $-Zeichen haben.

Die folgenden Variablen können also Zeichen, Wörter oder sogar Sätze enthalten:

```
Name$
Haarfarbe$
Inhalt$
Adresse$
```

Werte zuweisen

Variablen müssen natürlich auch einen Wert erhalten, sonst sind sie im Programm ja nutzlos. Der Fachausdruck heißt Zuweisung, die Variable wird wie in einer mathematischen Gleichung mit einem Wert versehen, und diese Wertezuweisungen haben immer den gleichen Aufbau:

Variablenname = Ausdruck

Ein Ausdruck kann ein genauer Wert sein oder eine andere Variable.

Hier einige Beispiele für Zuweisungen an Variablen:

```
Zahl = 23
Name$ = "Bob"
Superbonus = Megagewinn
```

Das Gleichheitszeichen ist hier nicht das mathematische »ist gleich« sondern ein Zuweisungsoperator, also ein »wird zu«, das heißt, das was links vom Gleichheitszeichen steht, erhält den gleichen Wert wie das, was auf der rechten Seite steht.

Ist ein Wert eine Zeichenkette, musst du ihn in Anführungszeichen setzen, wie im zweiten Beispiel, Zahlen dürfen nicht in Anführungszeichen stehen. Wenn du eine Zahl in Anführungszeichen setzt, wird sie wie ein Zeichen behandelt und kann später nicht für Rechenoperationen benutzt werden.

Berechnungen

Ähnlich wie Wertezuweisungen funktionieren auch Berechnungen, nur dass hier der Ausdruck aus einer Kombination von Werten oder Variablen besteht, die mit Rechenzeichen, so genannten Operatoren verknüpft sind.

Das funktioniert eigentlich wie im Matheunterricht, nur, dass das Gleichheitszeichen wie vorher schon erwähnt wieder ein Zuweisungsoperator ist.

Die verschiedenen Rechenoperatoren sind:

Operator	Bedeutung
+	Addieren (Zusammenzählen)
-	Subtrahieren (Abziehen)
*	Multiplizieren (Mal-Nehmen)
/	Dividieren (Teilen)
^	Potenzrechnen (Hoch-Rechnen)

Hier ein paar Beispiele für Berechnungen:

```
Ergebnis = 2 + 3
AnzahlSpieler = 11 * 2
Benzinverbrauch = Fahrleistung / Durchschnittsverbrauch
Hypothenusenquadrat = b ^ 2 + a ^ 2
```

Du fragst dich vielleicht, warum es keine Wurzelrechnung gibt. Die gibt es auch: Wenn du einen Wert hoch 0,5 rechnest, erhältst du die Wurzel davon. Wenn du rechnest:

```
Test = 9 ^ 0.5
```

Erhältst du dieses Ergebnis:

```
Test = 3
```

Du kannst nur mit Variablen rechnen, deren Wert eine Zahl ist. Du kannst aber auch Variablen vom Typ String addieren, dann werden die Inhalte aneinandergereiht (das nennt der Fachmann eine Stringverkettung).

```
A$ = "Toast"
B$ = "Brot"
C$ =  a$ + b$
```

Bei diesem Beispiel ist `C$ = "ToastBrot"`. Andere Rechenarten kannst du aber auf String-Variablen nicht anwenden.

Daten ausgeben

Jetzt weißt du schon, wie man mit Variablen rechnet, aber wie teilt man dem Benutzer dann das Ergebnis (oder etwas anderes) mit?

Das geht über die PRINT-Anweisung. Die PRINT Anweisung schreibt Daten auf den Bildschirm.

Dazu schreibst du einfach an der entsprechenden Stelle im Programm PRINT und danach das, was ausgegeben werden soll. Das kann ein Text in Anführungszeichen, eine Variable oder eine Rechnung sein. Bei einer Variable wird der Inhalt der Variable ausgegeben und bei einer Rechnung das Ergebnis.

Das folgende Beispiel zeigt dir die Unterschiede:

```
Zahl = 23
Wort$ = "Hallo"
PRINT Zahl
PRINT Wort$
PRINT "Das ist ein Test"
PRINT "2 + 3"
PRINT 2 + 3
```

Wenn du dieses Programm startest, erhältst du folgendes Ergebnis:

```
23
Hallo
Das ist ein Test
2 + 3
5
```

Wie du siehst, interpretiert QBASIC in der vorletzten Zeile das »2 + 3« als Zeichenkette (String), weil es in Anführungszeichen gesetzt ist, und schreibt es so, wie es angegeben ist, während in der letzten Zeile eine Rechenoperation ausgeführt wird.

Eine PRINT-Anweisung schreibt immer eine Zeile auf den Bildschirm, die nächste PRINT-Anweisung schreibt in eine neue Zeile. Wenn du willst, dass mehrere Daten in einer Zeile ausgegeben werden, musst du sie in einer PRINT-Anweisung mit Semikolons trennen:

```
Wort$ = "Hallo!"
Satz$ = "Das ist ein Test."
PRINT Wort$; Satz$
```

Wenn du dieses Programm startest, erhältst du das:

```
Hallo! Das ist ein Test.
```

Du kannst natürlich auch Strings und String-Variablen kombinieren:

```
Satz$ = "Das ist ein Test."
PRINT  "Hallo!"; Satz$
```

Auch hier erhältst du:

```
Hallo! Das ist ein Test.
```

Eine andere Möglichkeit ist, am Ende der PRINT-Anweisung ein Semikolon zu setzen, dann startet die nächste PRINT-Anweisung in der selben Zeile:

```
PRINT "Toast";
PRINT "Brot"
```

Hier ist die Ausgabe:

```
ToastBrot
```

Du kannst statt PRINT auch ein Fragezeichen schreiben. Das Ergebnis ist das Gleiche, der Schreibaufwand ist geringer, und QBASIC wandelt das ? eh in PRINT um, wahrscheinlich würde das Programm sonst zu sehr verwirrt werden.

```
? "Hallo"
```

Eingaben vom Benutzer anfordern

Das Gegenteil von PRINT ist INPUT. INPUT nimmt Eingaben von der Tastatur entgegen. Diese vom Benutzer eingegebenen Daten werden in einer Variable abgelegt, also musst du nach INPUT die Variable schreiben, in der diese Daten gespeichert werden. Hier ein kleines Beispielprogramm:

```
PRINT "Schreib was!"
INPUT Eingabe$
PRINT "Du hast "; Eingabe$; " geschrieben"
```

Wenn du das Programm startest, schreibt es zuerst »Schreib was!« auf den Bildschirm und wartet dann mit einem Fragezeichen, das soll dir sagen, dass du etwas eingeben sollst, so lange, bis du etwas schreibst und mit ↵ bestätigst. Dann speichert es das, was du geschrieben hast in der Variable Eingabe$ und gibt zur Kontrolle das aus, was du geschrieben hast.

Du kannst auch mit einer INPUT-Anweisung mehrere Werte aufnehmen, dazu musst du ihr auch mehrere Variablen, mit Kommas getrennt, übergeben:

```
PRINT "Wie heißt du und wie alt bist du?"
INPUT Name$, Alter$
```

Wichtig ist hier, dass der Benutzer zwischen den Werten Kommas setzt, sonst erhält er in einer Fehlermeldung den Hinweis, er solle noch mal von vorn anfangen.

Also, das Programm fordert dich auf, Name und Alter einzugeben, und wartet dann in der nächsten Zeile mit einem Fragezeichen. Dann musst du deinen Namen, dann ein Komma und dann dein Alter eingeben.

Das mit dem Komma führt beim Benutzer leicht zur Verwirrung, deshalb ist es wohl besser, wenn du ihn in solchen Fällen eins nach dem anderem abfragst:

```
PRINT "Wie heißt du? "
INPUT Name$
PRINT "Wie alt bist du? "
INPUT Alter$
```

Das ist zwar ein bisschen mehr Tipparbeit, aber für den Benutzer ist das Programm viel einfacher zu verstehen.

Kombinationen aus PRINT und INPUT

Wie du dir denken kannst, ist es wichtig, vor eine INPUT-Anweisung immer eine PRINT-Anweisung zu setzen, die dem Benutzer sagt, was er überhaupt schreiben soll.

Deshalb haben die Programmierer von QBASIC eine recht praktische Funktion von INPUT eingebaut, die die PRINT-Anweisung mit einbindet.

Dazu musst du nach INPUT einfach in Anführungszeichen eine Eingabeaufforderung oder Frage schreiben, dann ein Semikolon und dann wie gewohnt die Variable, in der die Eingabe gespeichert werden soll.

Hier ein Beispiel:

```
INPUT "Wie heißt du? "; Name$
```

Die Eingabe, die der Benutzer auf diese Anforderung macht, wird in der Variable Name$ gespeichert. Wenn QBASIC diese Programmzeile erreicht, sieht der Benutzer das:

```
Wie heißt du??
```

Wie du siehst, setzt die INPUT-Anweisung auch hier automatisch ein Fragezeichen an das Ende der Zeile, sodass dort dann zwei Fragezeichen ausgegeben werden. Weil man nicht verhindern kann, dass QBASIC automatisch ein Fragezeichen setzt, lassen wir einfach das Fragezeichen am Ende der Frage weg:

```
INPUT "Wie heißt du"; Name$
```

Dann ist die Ausgabe richtig:

```
Wie heißt du?
```

Kommentare

In großen Programmen verliert man oft sehr leicht den Überblick, und dann ist es für einen Programmierer ziemlich nützlich, wenn er sich im Programm einige Notizen und Anmerkungen machen kann. Diese Notizen heißen Kommentare.

In QBASIC wird ein Kommentar entweder durch REM oder einen Apostroph (') eingeleitet. Den Apostroph schreibst du mit der #Taste und gedrückter ⇧-Taste (nicht verwechseln mit den Accents links neben der ←-Taste!).

Alles was nach REM oder einem Apostroph steht, wird von QBASIC einfach ignoriert.

Der Vorteil beim Apostroph ist, dass du mit ihm auch einen Kommentar am Ende einer Programmzeile machen kannst, und für REM eine eigene Zeile brauchst.

```
REM Das ist ein Kommentar mit REM
'Das ist ein Kommentar mit Apostroph
PRINT "Moin" 'Kommentar mit ' hinter einer Programmzeile
```

Die Ausgabe dieses Programms ist nur »Moin«.

Entscheidungen

Entscheidungen sind Befehle, die einen oder mehrere Befehle nur dann ausführen, wenn eine bestimmte Bedingung erfüllt ist.

Entscheidungen mit IF

Der Standard-Entscheidungsbefehl bei QBASIC ist der IF-Befehl.

Der If-Befehl hat diesen Aufbau:

```
IF Bedingung THEN
   Befehl
ELSE
   Anderer Befehl
END IF
```

Die Bedingung ist immer eine Gleichung oder Ungleichung. Wenn die Bedingung wahr ist, werden die Befehle nach THEN ausgeführt, wenn sie falsch ist, werden die Befehle nach ELSE ausgeführt.

Das ELSE ist dabei nicht unbedingt notwendig, wenn nichts passieren soll, wenn der Ausdruck falsch ist, lässt du ELSE einfach weg.

Hier ein Beispiel:

```
INPUT "Wie viel Sekunden hat eine Minute"; Antwort$
IF Antwort$ = "60" THEN
   PRINT "RICHTIG! "
ELSE
   PRINT "FALSCH! "
END IF
```

Wenn du 60 eingibst, wird »RICHTIG!« ausgegeben, ansonsten springt das Programm zu ELSE und »FALSCH!« wird ausgegeben.

Hier ist die Bedingung eine Gleichung. Übrigens ist hier das Gleichheitszeichen kein Zuweisungsoperator, sondern das mathematische »Ist gleich«, weil diesmal ja zwei Werte verglichen werden und nichts zugewiesen wird.

Die Bedingung kann aber auch eine Ungleichung sein, d.h. ein Vergleich mit Größer oder Kleiner:

```
INPUT "Wie alt bist du"; Antwort
IF Antwort < 18 THEN
   PRINT "Schade, du darfst noch nicht Auto fahren. "
ELSE
   PRINT "Du darfst schon Auto fahren."
END IF
```

Wenn du als Antwort eine Zahl eingibst, die kleiner ist als 18, macht das Programm in der nächsten Zeile mit der Meldung weiter: »Du darfst noch nicht Auto fahren.«, ansonsten springt es zu der Meldung »Du darfst schon Auto fahren.«

< ist einer von vielen Vergleichsoperatoren. Diese Vergleichsoperatoren kannst du verwenden:

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

In den letzten zwei Beispielen war die Bedingung immer ein Vergleich einer Variable mit einem Wert. Aber eine Bedingung kann auch ein Vergleich von zwei Variablen sein:

```
PRINT "Denk dir zwei Zahlen"
INPUT "Was ist die erste Zahl"; Zahl1
INPUT "Was ist die zweite Zahl"; Zahl2
IF Zahl1 = Zahl2 THEN
   PRINT "Beide Zahlen sind gleich"
ELSE
   PRINT "Die Zahlen sind verschieden. "
END IF
```

Hier wäre es doch noch toll, wenn das Programm dir noch sagen würde, welche Zahl größer ist, wenn sie nicht gleich sind, oder? Kein Problem, dafür gibt es die ELSEIF-Anweisung:

```
PRINT "Denk dir zwei Zahlen"
INPUT "Was ist die erste Zahl"; Zahl1
INPUT "Was ist die zweite Zahl"; Zahl2
```

```
IF Zahl1 = Zahl2 THEN
   PRINT "Beide Zahlen sind gleich"
ELSEIF Zahl1 > Zahl2 THEN
   PRINT "Die erste Zahl ist größer"
ELSE
   PRINT "Die zweite Zahl ist größer"
END IF
```

Wenn also die Zahlen gleich sind, schreibt das Programm »Beide Zahlen sind gleich«. Wenn sie nicht gleich sind, springt es zu ELSEIF und überprüft, ob Zahl1 größer ist als Zahl2. Wenn das zutrifft, gibt es die Meldung aus: »Die erste Zahl ist größer«, wenn das nicht zutrifft, springt es zu ELSE und gibt die Meldung aus: »Die zweite Zahl ist größer«.

Ich gebe ja zu, es gibt sicher aufwändigere Programme als den Vergleich zweier Zahlen, aber am Anfang sollte man die Dinge immer möglichst einfach halten (schwieriger wird's von selbst). Und so musst du dich noch eine Zeit lang durch dieses Beispiel in zig Variationen quälen. Später wird's dann schon interessanter!

Die SELECT CASE-Anweisung

Die SELECT CASE-Anweisung eignet sich dazu, mehrere Bedingungen gleichzeitig zu überprüfen. Eigentlich hat sie die gleiche Funktion wie die ELSEIF-Anweisung, ist aber leichter zu benutzen und übersichtlicher.

Das ist der Aufbau einer SELECT CASE-Anweisung:

```
SELECT CASE Var
CASE Wert1
   Befehle1
CASE  Wert2
   Befehle2
CASE Wert3
   Befehle3
CASE ELSE
   Befehle4
END SELECT
```

Wenn die Variable »Var« den Wert »Wert1« hat, führt das Programm die Befehle »Befehle1« aus, wenn sie den Wert »Wert2« hat, führt es die Befehle »Befehle2« aus usw. Wenn der Wert der Variable weder »Wert1« noch »Wert2« noch »Wert3« ist, führt das Programm die Befehle aus, die unter CASE ELSE stehen, also »Befehle4«.

Wenn zwei Werte mit Hilfe der Vergleichsoperatoren <, >, <=, >= oder <> verglichen werden sollen, schreibst du CASE IS, dann den Vergleichsoperator und den Wert, mit dem verglichen werden soll.

Wir können SELECT CASE auch in dem Beispielprogramm von vorhin benutzen:

```
PRINT "Denk dir zwei Zahlen"
INPUT "Was ist die erste Zahl"; Zahl1
INPUT "Was ist die zweite Zahl"; Zahl2
```

```
SELECT CASE Zahl1
CASE Zahl2 ' Zahl1 = Zahl2
   PRINT "Beide Zahlen sind gleich"
CASE IS > Zahl2
   PRINT "Zahl1 ist größer"
CASE IS < Zahl2
   PRINT "Zahl2 ist größer"
END SELECT
```

Dieses Programm macht das Gleiche wie das vorherige, der Code ist aber etwas übersichtlicher und du hast weniger zu schreiben.

Hinweis

Benutze am besten immer die IF-Anweisung, wenn du nur eine oder zwei Bedingungen hast. Wenn mehrere Bedingungen zum Vergleich anstehen, ist SELECT CASE in der Regel besser geeignet.

Schleifen

Schleifen sind Anweisungen, die einen oder mehrere Befehle immer wieder wiederholen.

Die FOR...NEXT-Schleife

Mit der FOR...NEXT-Schleife kannst du eine Gruppe von Befehlen beleibig oft wiederholen.

Angenommen, du möchtest zehn Mal »Hallo« auf den Bildschirm schreiben. Dazu müsstest du zehn Mal untereinander schreiben:

```
PRINT "Hallo"
```

Mit der FOR...NEXT-Schleife ersparst du dir die Schreibarbeit:

```
FOR i = 1 TO 10
   PRINT "HALLO"
NEXT
```

Auch hier werden zehn Hallos ausgegeben, aber du musst nicht zehn Mal den gleichen Befehl hinschreiben.

Mit »i =« führst du eine Variable ein, die am Anfang den Wert 1 erhält. Dann wird der Befehl

```
PRINT "Hallo"
```

ausgeführt, und sobald das Programm

```
NEXT
```

erreicht, springt es wieder zur ersten Zeile, erhöht i um 1 und führt den PRINT-Befehl wieder aus. Das geht solange weiter, bis i = 10 ist, dann führt das Programm noch ein letztes Mal den Befehl `PRINT "Hallo"` aus, springt aber dann nicht mehr nach oben, wenn es `NEXT` erreicht.

Du kannst natürlich die Variable, die jedes Mal erhöht wird, auch innerhalb der Schleife, also zwischen FOR und NEXT, verwenden, du darfst nur ihren nicht Wert verändern, weil die Schleife sonst nicht mehr so weitermacht, wie sie sollte.

Wenn du willst, dass sich »i« bei jedem Durchgang um einen anderen Wert erhöht, kannst du das mit der STEP-Option ändern:

```
FOR i = 1 TO 10 STEP 2
   PRINT "Hallo"
NEXT
```

Hier wird nur fünf Mal »Hallo« geschrieben, weil »i« jedes Mal, wenn das Programm FOR erreicht nicht um 1, sondern um 2 erhöht wird. Auf diese Weise kannst du zum Beispiel alle ungeraden Zahlen zwischen 1 und 100, das heißt jede zweite Zahl ausgeben:

```
FOR i = 1 TO 100 STEP 2
   PRINT i
NEXT
```

Wenn du STEP einen negativen Wert gibst, wird »i« bei jedem FOR verkleinert. So kannst du zum Beispiel einen Countdown programmieren:

```
FOR i = 10 TO 1 STEP -1
   PRINT i
NEXT
PRINT "Feuer"
```

Abbildung 2.6: Countdown mit der For...Next-Schleife

Wenn du aus irgendeinem Grund die FOR...NEXT-Schleife verlassen willst, bevor alle Durchgänge ausgeführt wurden, kannst du den EXIT FOR Befehl benutzen. Findet das Programm in der Schleife einen EXIT FOR-Befehl, verlässt es diese sofort und springt zum ersten Befehl nach NEXT.

```
FOR i = 1 TO 100   'würde normalerweise 100mal durchlaufen
   PRINT i
   INPUT "Weiter (J/N) "; Antwort$
'prüft, ob der Benutzer weitermachen will
   IF Antwort$ = "N" THEN
      EXIT FOR
   END IF
NEXT
```

Dieses Programm gibt nacheinander alle Zahlen von 1 bis 100 aus und fragt den Benutzer nach jeder Zahl, ob er weitermachen will. Wenn er N drückt, verlässt das Programm die Schleife.

Die FOR...NEXT-Schleife ist dazu da, eine ganz bestimmte Anzahl von Schleifendurchgängen auszuführen. Aber nicht bei allen Schleifen weiß man vorher, wie oft man sie durchlaufen muss, für manche Situationen braucht man eine Schleife, die durchlaufen wird, solange oder bis eine bestimmte Bedingung zutrifft.

Auch dafür gibt es zwei verschiedene Schleifen: die DO...WHILE – und die DO...UNTIL-Schleife.

Die DO...WHILE-Schleife

Die DO...WHILE-Schleife führt einen oder mehrere Befehle solange aus, wie eine bestimmte Bedingung zutrifft. Wenn diese Bedingung nicht mehr zutrifft, verlässt das Programm die Schleife.

Das ist ihr Aufbau:

```
DO
   Befehle
LOOP WHILE Bedingung
```

Sobald das Programm die Schleife erreicht, führt es die Befehle innerhalb der Schleife aus. Wenn es alle Befehle zwischen DO und LOOP ausgeführt hat, überprüft es, ob die Bedingung nach WHILE zutrifft, und wenn sie zutrifft, durchläuft es die Schleife noch einmal. Wenn die Bedingung nicht zutrifft, verlässt es die Schleife.

Die DO...UNTIL-Schleife

Die DO...UNTIL-Schleife, ist ähnlich wie die DO...WHILE-Schleife, nur dass die DO...UNTIL-Schleife verlassen wird, wenn die Bedingung zutrifft und die DO...WHILE Schleife verlassen wird, wenn die Bedingung nicht zutrifft. Das ist der Aufbau der DO...UNTIL-Schleife:

```
DO
   Befehle
LOOP UNTIL Bedingung
```

Sobald das Programm die Schleife erreicht, führt es die Befehle innerhalb der Schleife aus. Wenn es alle Befehle zwischen DO und LOOP ausgeführt hat, überprüft es, ob die Bedingung nach UNTIL zutrifft, und wenn sie zutrifft, wird die Schleife verlassen. Trifft die Bedingung nicht zu, durchläuft es die Schleife noch einmal.

Hinweis

Achte in einer DO...WHILE – oder DO...UNTIL-Schleife immer darauf, dass die Bedingung auch sicher innerhalb der Schleife unerfüllt bzw. erfüllt wird, sodass das Programm die Schleife irgendwann verlässt, weil sie sonst unendlich (oder bis der Computer abstürzt) weiterläuft.

Das folgende Programm ist ein Zahlenratespiel, bei dem du eine vom Computer erzeugte Zufallszahl erraten sollst:

```
CLS
RANDOMIZE TIMER 'initialisiert den Zufallszahlengenerator
Zufallszahl = INT((10 * RND) + 1)
' erzeugt die Zufallszahl zwischen 1 und 10
PRINT "Ich habe mir eine Zahl zwischen 1 und 10 gedacht. "
PRINT "Errate sie!"
DO
   INPUT Ratezahl
LOOP UNTIL Ratezahl = Zufallszahl
PRINT "Richtig! "
```

Dieses Programm lässt in einer DO...UNTIL-Schleife den Benutzer die Zahl so lange raten, bis die geratene Zahl gleich der Zufallszahl ist und gibt dann die Meldung »Richtig!«

Abbildung 2.7: Zahlen raten mit der Do-Schleife

Hinweis

Der RANDOMIZE-Befehl liefert eine »Zufallszahl« zwischen 1 und 0 für die RND-Funktion, dazu muss man ihr aber eine Startzahl geben, aus der sie die »Zufallszahl« berechnet. TIMER ist eine Variable, die automatisch die Zahl der Sekunden enthält, die seit Mitternacht vergangen sind. Weil sich dieser Wert dauernd verändert, ist er gut dazu geeignet aus ihm eine zufällige Zahl zu berechnen, und deshalb wird er meistens dem RANDOMIZE-Befehl übergeben. Die Variable RND steht für die Zahl, die von RANDOMIZE berechnet wurde, und weil es das eine Zahl zwischen 0 und 1 ist, müssen wir sie mit 10 multiplizieren und 1 dazuzählen, damit wir eine Zahl von 1 bis 10 haben. Mit INT wird diese Zahl, die ja noch ein Dezimalbruch sein könnte, auf eine ganze Zahl gerundet. Fertig ist unsere Zufallszahl. Natürlich ist das keine Zufallszahl, denn sie wurde nicht willkürlich und zufällig gewählt, sondern nur auf einem komplizierten Weg, sodass sie kaum vorhersehbar ist.

Weil LOOP WHILE genau entgegengesetzt ist zu LOOP UNTIL, kannst du eigentlich mit beiden das Gleiche bezwecken, wenn du die Bedingung umkehrst. So könnte z.B. die letzte Zeile des Zahlenratenprogramms auch so lauten:

```
LOOP WHILE Ratezahl <> Zufallszahl
```

Das Ergebnis wäre das Gleiche. Im ersten Fall, fragt das Programm den Benutzer so lange nach einer Zahl, bis die Zahl des Benutzers gleich der Zufallszahl ist und im zweiten Fall fragt das Programm den Benutzer so lange, wie die Zahl des Benutzers sich von der Zufallszahl unterscheidet.

Arrays

Oft kommt es vor, dass man so viele Variablen desselben Typs braucht, dass es zu viel wäre, jede Variable extra einzulesen oder jeder einzelnen Variable einen Wert zuzuordnen.

Stell dir zum Beispiel mal vor, dein Lehrer hat ein Programm, mit dem er die Noten seiner Schüler verwaltet. Bei sagen wir mal 30 Schülern wäre so ein Programm ganz schön aufwändig. Erst einmal müssten die Namen aller Schüler in Variablen abgespeichert werden. Die Variablen könnten SName1$, SName2$, SName3$ usw. heißen. Dann müsste der Lehrer eingeben, welche Note welcher Schüler bekommen hat, also müssen noch einmal 30 separate Variablen, beispielsweise Noten1$, Note2$, Noten3$ usw. erstellt werden. Dieses Programm könnte dann so aussehen:

```
REM Programm, das geradezu nach der Verwendung
REM von Arrays schreit
CLS
INPUT "Name des nächsten Schülers"; SName1$
INPUT "Prüfungsergebnis von diesem Schüler"; Note1$
PRINT 'leere Zeile ausgeben
INPUT "Name des nächsten Schülers"; SName2$
INPUT "Prüfungsergebnis von diesem Schüler"; Note2$
```

```
PRINT 'leere Zeile ausgeben
INPUT "Name des nächsten Schülers"; SName3$
INPUT "Prüfungsergebnis von diesem Schüler"; Note3$
PRINT 'leere Zeile ausgeben
INPUT "Name des nächsten Schülers"; SName4$
INPUT "Prüfungsergebnis von diesem Schüler"; Note4$
PRINT 'leere Zeile ausgeben
INPUT "Name des nächsten Schülers"; SName5$
INPUT "Prüfungsergebnis von diesem Schüler"; Note5$
PRINT 'leere Zeile ausgeben
...
```

So würde das noch weiter gehen, bis endlich 2 mal 30 Variablen mit den Namen und den Noten der Schüler zusammen sind.

Hinweis

Natürlich ist dieses Programm völlig praxisfern, da ein Lehrer bekanntlich die Noten seiner Schüler grundsätzlich mit der RND-Funktion (Zufallszahl) ermittelt. ☺

Wie du siehst, müssen hier ziemlich oft immer die gleichen drei Zeilen geschrieben werden, die sich nur in einer Stelle in den Variablennamen unterscheiden. Außerdem ist diese eine Stelle auch noch eine Zahl, die bei jedem Befehl um 1 höher ist. Es wäre doch toll, wenn man hier eine FOR...NEXT-Schleife benutzen könnte, und einfach an dieser Stelle im Variablennamen die Variable einsetzt, die bei jedem Durchgang erhöht wird. Dann wäre nämlich diese seitenlange Befehlskette nur fünf Zeilen lang.

Leider kann man in einen Variablennamen nichts einbauen, auch keine andere Variable. Es gibt aber doch eine einfache Möglichkeit, diesen Vorschlag zu realisieren, nämlich mit einem Array.

Ein Array ist eine Gruppe ähnlicher Variablen, die alle den gleichen Namen haben und sich nur durch einen Index unterscheiden. Dieser Index ist eine Zahl, die hinter dem Variablennamen in Klammern steht. Und dieser Index lässt sich über andere Variablen ansprechen.

Wir müssen die Variablen in diesem Programm also nicht SName1$, SName2$, SName3$ usw. nennen, sondern SName$(1), SName$(2), SName$(3) usw.

Jetzt können wir diesen langen Programmausschnitt in eine FOR...NEXT-Schleife packen:

```
REM Das Programm von vorher unter Verwendung von Arrays
CLS
FOR i = 1 TO 30
   INPUT "Name des nächsten Schülers"; SName$(i)
   INPUT "Prüfungsergebnis von diesem Schüler"; Note$(i)
   PRINT' leere Zeile ausgeben
NEXT
' Rest des Programms folgt
```

Beim ersten Durchlauf der FOR...NEXT-Schleife ist i gleich 1. Der Lehrer gibt den Wert für SName$(i), also SName$(1) ein und dann den Wert für Note$(i), also Note$(1). Anschließend erhöht die Schleife i um 1, sodass i jetzt 2 ist und fängt wieder von vorne an, so lange bis der Lehrer alle 30 Namen und Noten eingegeben hat. So ist der Code viel übersichtlicher, und der Programmierer muss viel weniger schreiben.

Häufig muss QBASIC im Voraus wissen, wie groß ein Array sein soll. Ohne die DIM-Anweisung stellt QBASIC nur Platz für zehn Arrayelemente bereit. DIM reserviert Arrayspeicher für dein Programm.

So sieht sie DIM-Anweisung für die 30 Schülernamen und Ergebnisse aus:

```
DIM SName$(30)' reserviert Speicher für 30 Elemente
DIM Note$(30)
```

Zugriff auf Dateien

Variablen und Arrays sind praktisch, um Daten in einem Programm abzulegen und zu einem späteren Zeitpunkt im Programm wieder abzurufen, aber wenn das Programm beendet oder der Computer ausgeschaltet wird, sind diese Daten verloren. Wenn du Daten dauerhaft speichern willst, um sie z.B. beim nächsten Programmstart wieder abrufen zu können, musst du sie auf der Festplatte in Dateien speichern.

Dateien bleiben auf der Platte, auch wenn du den Computer ausschaltest. Jede Datei auf der Platte hat einen eindeutigen Dateinamen, über den du von deinem Programm aus auf die Dateien zugreifen kannst, um aus ihnen zu lesen oder in sie zu schreiben.

QBASIC unterstützt keine langen Dateinamen, die es erst seit Windows 95 gibt. Lange Dateinamen sind alle Dateinamen, die (vor dem Punkt) länger als acht Zeichen und/oder nach dem Punkt mehr als drei Zeichen lang sind. Wenn eine DOS-Anwendung wie QBASIC auf eine Datei mit einem langem Dateinamen zugreift, beispielsweise

Datenspeicher 204.DAT

erzeugt DOS eine Abkürzung, indem es die ersten sechs Buchstaben des Dateinamens nimmt und ~1 anhängt, etwa so:

Datens~1.DAT.

Wenn mehrere Dateien mit den gleichen sechs Buchstaben beginnen, wird die Endung erhöht: ~2, ~3 usw.

Wenn du von QBASIC aus auf Dateien mit langen Dateinamen zugreifst, musst du also die abgekürzte Version des Dateinamens verwenden; eine mit QBASIC angelegte Datei kann nur entsprechend dem 8.3 Format gespeichert werden, Das heißt, du darfst nicht mehr als acht Zeichen vor dem Punkt und nicht mehr als drei Zeichen nach dem Punkt benutzen.

Du solltest keine Sonderzeichen verwenden. Einige der Sonderzeichen sind zwar erlaubt, viele aber nicht, deshalb ist es am besten, überhaupt keine Sonderzeichen in Dateinamen zu verwenden.

Zugriffsarten

Es gibt drei Dinge, die du mit einer Datei tun kannst:

- Eine neue Datei anlegen und etwas in diese schreiben.
- Eine vorhandene Datei lesen.
- Daten an das Ende eine vorhandenen Datei anfügen.

Der OPEN-Befehl

Das alles kannst du mit einem Befehl machen, der OPEN-Anweisung. Du musst der OPEN-Anweisung allerdings sagen, mit welcher Zugriffsart du auf eine Datei zugreifen willst. Außerdem brauchst du noch eine Dateinummer. Die Dateinummer ist normalerweise 1, wenn du eine zweite Datei öffnen willst, solltest du ihr dann die Dateinummer 2 geben. Die Dateinummer wird beim Öffnen zur Bezeichnung für diese Datei. Wenn eine Datei geöffnet ist, sprichst du sie nur noch über die Dateinummer und nicht über ihren Namen an. Wenn du z.B. die Datei »test.dat« auf Laufwerk C im Ordner »Speicher« öffnen willst, um aus ihr zu lesen, lautet der Befehl dafür

```
OPEN "C:\Speicher\test.dat" FOR INPUT AS #1
```

Wenn du eine Datei zur Eingabe öffnen willst, die es gar nicht gibt, erhältst du eine Fehlermeldung. Eine Datei kann nur gelesen werden, wenn sie existiert.

Allerdings wird eine Datei, die du für eine Ausgabe, also zum Hineinschreiben, öffnen willst, automatisch erstellt, wenn sie nicht existiert.

Die folgende OPEN-Anweisung öffnet eine Datei für den Ausgabemodus, sodass du etwas hineinschreiben kannst.

```
OPEN "C:\Speicher\test2.dat" FOR OUTPUT AS #2
```

Bei OUTPUT wird vorausgesetzt, das die Datei noch nicht existiert, weil sie überschreiben wird, wenn sie schon vorhanden ist.

Also, mit INPUT liest man aus einer Datei und mit OUTPUT legt man eine neue Datei an und schreibt dann in diese. Es kommt aber auch vor, dass etwas in eine Datei geschrieben wird, die es schon gibt, ohne dass ihr Inhalt verloren geht. Dafür gibt es den Einfügemodus. In diesem Modus werden alle Schreibvorgänge an das Ende dieser Datei angefügt, ohne dass der bestehende Inhalt gelöscht wird. Der Befehl dafür sieht so aus:

```
OPEN "C:\Speicher\test3.dat" FOR APPEND AS #3
```

Also, wenn du eine Datei geöffnet hast, sprichst du sie nur über die Dateinummer an, die du ihr zugeordnet hast. Wie du sie ansprechen kannst, hängt von dem Modus ab, in dem du sie geöffnet hast.

- Aus Dateien, die du im Eingabemodus, also mit INPUT, geöffnet hast, kannst du nur lesen.
- Dateien, die du im Ausgabemodus öffnest, werden neu angelegt, oder, wenn sie schon existieren, überschrieben und du kannst nur in sie schreiben.
- Wenn du in Dateien, die du im Einfügemodus geöffnet hast, etwas schreiben willst, wird das automatisch an das Ende der Datei geschrieben, ihr ursprünglicher Inhalt wird nicht gelöscht.

Schreiben in Dateien

Nachdem du eine Datei im Ausgabemodus oder im Einfügemodus geöffnet hast, schreibst du mit dem Befehl WRITE # Daten in diese Datei. Nach WRITE # muss die Dateinummer der Datei stehen, in die du schreiben willst.

WRITE # ist für eine Datei das, was PRINT für den Bildschirm ist. Aber anders als PRINT, das Daten so ausgibt, wie du sie eingibst, macht WRITE # mehr, als die Daten einfach nur hineinzuschreiben. WRITE # trennt die Daten durch Kommas und setzt auch noch alle Stringdaten in Anführungszeichen. So ist es später viel leichter, die einzelnen Daten mit INPUT # wieder in Variablen zurückzuholen.

Statt die Daten einfach zu einer Kette zusammenzuklatschen, trennt WRITE # die Daten voneinander, damit man sie später separat wieder auslesen kann.

Das nächste Programm zeigt dir, wie du Daten mit WRITE # in Dateien speichern kannst. Die ersten paar Befehle weisen den Variablen Werte zu, die letzte Zeile schreibt diese Daten auf die Platte.

```
CLS
Name$ = "Bart Simpson"
Alter = 10
Adresse$ = "742 Evergreen Terrace"
Stadt$ = "Springfield"
OPEN "C:\Data.dat" FOR OUTPUT AS #1
WRITE #1, Name$, Alter, Adresse$, Stadt$
CLOSE 1
```

Und so sieht die neu angelegte Datei »Data.dat« aus, nachdem das Programm sie erzeugt hat:

```
"Bart Simpson",10,"742 Evergreen Terrace","Springfield"
```

Lesen aus einer Datei

Es ist ganz einfach, aus einer Datei zu lesen. Daten, die du mit WRITE # geschrieben hast, kannst du wieder mit INPUT # in separate Variablen einlesen. INPUT # ist der entgegengesetzte Befehl zu WRITE #.

Die Daten aus dem vorigen Programm kannst du also mit INPUT wieder aus der Datei heraus in ihre ursprünglichen Variablen lesen:

```
CLS
OPEN "C:\Data.dat" FOR INPUT AS #1
INPUT #1, Name$, Alter, Adresse$, Stadt$
CLOSE 1
PRINT "Name = "; Name$
PRINT "Alter = "; Alter
PRINT "Adresse = "; Adresse$
PRINT "Stadt = "; Stadt$
```

Zuerst wird die Datei, in der wir vorher die Daten gespeichert haben, im Eingabemodus geöffnet. Dann werden nacheinander die einzelnen Daten eingelesen und den entsprechenden Variablen übergeben. Als Ausgabe erhalten wir also:

```
Name = Bart Simpson
Alter = 10
Adresse = 742 Evergreen Terrace
Stadt = Springfield
```

So könnten wir also ein Programm schreiben, das beim Beenden die Werte aller Variablen in einer Datei speichert und beim nächsten Start wieder aus der Datei liest und in die Variablen schreibt.

Es gibt noch ein Problem beim Lesen von Datendateien. Das Programm, das die Datei liest, weiß vielleicht nicht genau, wie viele Datensätze sich in der Datei befinden. Deshalb gibt es in QBASIC eine Funktion, die dir mitteilt, wann das Ende der Datei erreicht ist, damit dein Programm weiß, dass es aufhören muss, zu lesen.

Diese Funktion zum Erkennen des Dateiendes heißt EOF( ), das bedeutet *End of File*, also Ende der Datei. In den Klammern musst du die Dateinummer der Datei angeben. EOF( ) ist gleich 0, wenn es noch weitere Datensätze zu lesen gibt, und wird -1, wenn der letzte Datensatz der Datei gelesen wurde.

Im Allgemeinen verwendet man eine DO...UNTIL-Schleife mit der Bedingung

```
EOF( ) = -1
```

um eine Datei bis zum Ende zu lesen:

```
DO
   INPUT #1, Variablen
LOOP UNTIL EOF(1) = -1
```

So werden so lange die Variablen eingelesen, bis das Ende der Datei erreicht ist. Am besten ist es hier, wenn du zum Einlesen ein Array verwendest und innerhalb der Schleife eine Variable erhöhst, mit der du den Index des Arrays ansprichst, sodass du immer in eine neue Variable einliest:

```
DO
   Zähler = Zähler + 1
   INPUT #1, Variablen(Zähler)
LOOP UNTIL EOF(1) = -1
```

So wird jedes Datenfeld aus der Datei in eine eigene Variable eingelesen.

Offene Dateien schließen

Jede Datei, die mit QBASIC geöffnet wurde, sollte auch wieder geschlossen werden, wenn sie nicht mehr gebraucht wird. Das geht ganz einfach, nämlich mit dem CLOSE-Befehl. Du schreibst einfach CLOSE und danach die Dateinummer der Datei, die du schließen willst.

```
CLOSE 1
```

Dieser Befehl schließt die Datei, der du beim Öffnen die Dateinummer 1 zugeordnet hast. Du kannst auch mehrere Dateien gleichzeitig schließen, indem du die Dateinummern mit Kommas voneinander trennst:

```
CLOSE 1, 2, 3
```

Wenn du CLOSE ohne einen Zusatz benutzt, werden alle geöffneten Dateien geschlossen.

```
CLOSE
```

Daten an eine Datei anfügen

Das Anhängen der Daten an eine Datei funktioniert genauso, wie das normale Schreiben in eine Datei, nämlich mit dem WRITE # -Befehl. Der einzige Unterschied befindet sich im OPEN-Befehl.

Wenn du einen WRITE # -Zugriff auf eine Datei machst, die im OUTPUT-Modus geöffnet wurde, wird die Datei neu angelegt bzw. überschrieben.

Wenn du einen WRITE # -Zugriff auf eine Datei machst, die im APPEND-Modus geöffnet wurde, werden die Daten einfach ans Ende der Datei angehängt.

Übrigens, wenn du eine Datei im APPEND-Modus öffnest, die noch nicht existiert, wird sie auch erstellt. Du darfst nur keine nicht existierenden Dateien im INPUT-Modus öffnen, das führt zu einer Fehlermeldung.

Das folgende Programm öffnet die bereits existierende Datei »Data.dat« aus den vorigen Beispielprogrammen und fordert den Benutzer auf, zusätzliche Werte einzugeben.

```
CLS
OPEN "C:\Data.dat" FOR APPEND AS #1
DO
   PRINT
   INPUT "Name"; Name$
   INPUT "Alter"; Alter
   INPUT "Adresse"; Adresse$
   INPUT "Wohnort"; Stadt$
   WRITE #1, Name$, Alter, Adresse$, Stadt$
   INPUT "Noch eine Person hinzufügen(J/N)"; Antwort$
LOOP UNTIL Antwort$ = "N"
CLOSE 1
```

Hier kannst du jetzt so viele Personen mit ihrem Alter und ihrer Adresse hinzufügen, wie du willst, und das Programm speichert die Daten in der Datei »Data.dat«.

So kannst du dir eine Datenbank mit all deinen Freunden und Bekannten anlegen.

Wir brauchen jetzt nur noch ein Programm, mit dem man den Inhalt der Datei ansehen kann. Du könntest die Datei zwar mit dem Windows-Notepad öffnen, aber dann sind die Felder nebeneinander und mit Kommas getrennt, das sieht nicht so toll aus. Schreiben wir uns doch schnell ein Programm, das die Daten wieder ausliest.

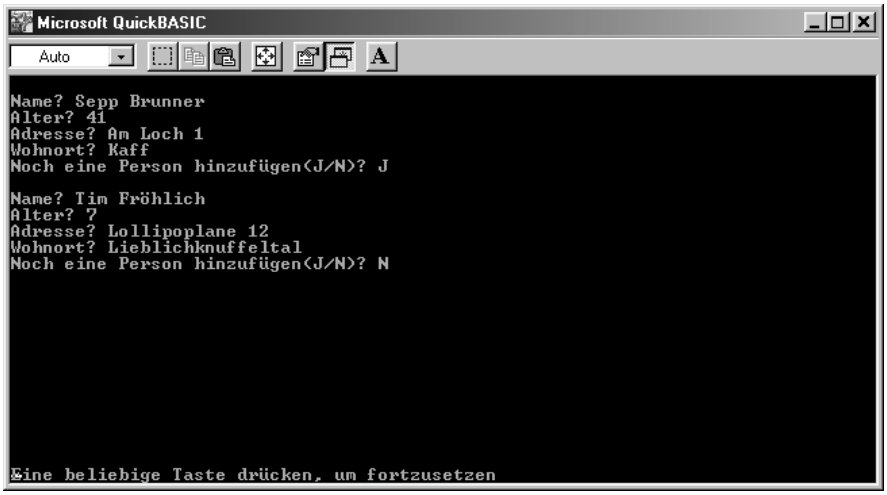

Abbildung 2.8: Die Datendatei wird angelegt.

Abbildung 2.9: Datendatei wird ausgegeben.

```
DIM Name$(100), Alter(100), Adresse$(100), Stadt$(100)
CLS
OPEN "C:\Data.dat" FOR INPUT AS #1
DO
   i = i + 1
   INPUT #1, Name$(i), Alter(i), Adresse$(i), Stadt$(i)

   PRINT "Name:"; Name$(i)
   PRINT "Alter:"; Alter(i)
   PRINT "Adresse:"; Adresse$(i)
```

```
   PRINT "Wohnort:"; Stadt$(i)
   PRINT
LOOP UNTIL EOF(1) = -1
CLOSE 1
```

Dieses Programm liest in einer DO...UNTIL-Schleife so lange die Datenfelder in Arrayelemente und gibt diese aus, bis das Ende der Datei erreicht ist.

Dateien kopieren

Wir könnten doch mal probieren, zwei Dateien gleichzeitig zu öffnen und den Inhalt der einen Datei in die andere schreiben. Das ist gar nicht so schwer: Das nächste Programm liest in einer DO...UNTIL-Schleife jeweils einen Datensatz aus der Datei »Data.dat« und schreibt ihn in die Datei »Backup.dat«. So kannst du ganz schnell Backups, also Sicherungskopien von wichtigen Dateien erzeugen.

```
CLS
PRINT "Kopiere Datei..."
OPEN "C:\data.dat" FOR INPUT AS #1
OPEN "C:\Backup.dat" FOR OUTPUT AS #2
DO
   INPUT #1, Name$, Alter, Adresse$, Stadt$
   WRITE #2, Name$, Alter, Adresse$, Stadt$
LOOP UNTIL EOF(1) = -1
CLOSE 1
CLOSE 2
PRINT "Datei ist gesichert."
```

Mit PRINT Daten in Dateien schreiben

Es gibt auch noch eine andere Möglichkeit, Daten in Dateien zu schreiben, als mit WRITE. Die Anweisung WRITE eignet sich gut, um Daten für dein Programm zu speichern, die du später wieder einfach auslesen kannst.

Wenn du allerdings Dateien erstellen willst, die der Benutzer auch lesen soll, ist der WRITE-Befehl eher fehl am Platz, weil er die Daten nacheinander und mit Kommas getrennt ausgibt. In diesem Fall ist der PRINT-Befehl besser geeignet, denn er schreibt die Daten (wie der PRINT-Befehl für die Bildschirmausgabe) zeilenweise in die Datei.

```
PRINT #1, "dumm ist der, der Dummes tut(Forrest Gump's Mama)"
```

Dieser Befehl schreibt den Text in die Datei, die mit der Dateinummer 1 geöffnet wurde. Wenn du jetzt noch einmal einen Text mit PRINT in diese Datei schreibst, wird er in die nächste Zeile geschrieben.

Mit LINE INPUT Daten zeilenweise auslesen

Das Gegenstück zu PRINT ist die Anweisung LINE INPUT. Mit ihr kannst du eine Zeile einer Datei in eine Variable einlesen. Der folgende Befehl speichert den Text der ersten Zeile der Datei, die mit der Dateinummer 1 geöffnet wurde, in der Variable »ersteZeile« ab:

```
LINE INPUT #1, ersteZeile
```

Der nächste LINE INPUT-Befehl würde den Text der nächsten Zeile speichern.

Töne auf dem Lautsprecher

Viele PCs haben einen eingebauten Lautsprecher, den PC-Speaker. Dieser Lautsprecher wurde früher oft in Programmen und Spielen verwendet, aber da heute fast jeder Computer mit zwei Stereoboxen ausgestattet ist und die Tonqualität des PC-Speakers ziemlich schlecht ist, wird er kaum mehr benutzt und in viele Systeme gar nicht mehr eingebaut. Wenn du noch so einen Lautsprecher dein Eigen nennst, kannst du mit QBASIC ein bisschen damit rumprobieren und sehen, was man mit ihm alles machen kann.

BEEP

Die vielleicht einfachste Methode, mit QBASIC einen Ton herauszuholen, ist der Befehl BEEP. Dieser Befehl gibt einen kurzen Ton über den PC-Speaker aus. Mit dem folgenden Programm kannst du so viele Beeps hören, wie du willst:

```
CLS
BEEP
PRINT "Das war der Lautsprecher"
INPUT "Willst du es noch einmal hören(J/N)"; Antwort$
IF Antwort$ = "J" THEN
   INPUT "Wie oft"; Anzahl
   For n = 1 TO Anzahl
      BEEP
   NEXT
END IF
```

SOUND

Der Befehl BEEP war ja schon ganz lustig, aber leider kannst du mit ihm nur einen einzigen Ton erzeugen. Mit dem Befehl SOUND hast du viel mehr Kontrolle über den PC-Speaker, denn du kannst bestimmen, welcher Ton gespielt werden soll, und wie lange er anhalten soll.

Hier ein Beispiel für den SOUND-Befehl:

```
SOUND 880, 36.4
```

Hinweis

Dezimal musst du grundsätzlich mit einem Punkt für die Dezimalstelle schreiben (36.4, nicht 36,4!).

Der Wert nach SOUND ist die Hertzzahl des Tons, die Zahl danach gibt die Dauer des Tons an.

Hinweis

Hertz (Hz) gibt die Anzahl der Zyklen pro Sekunde an, mit der eine Note vibriert. Das ist eine ziemlich komplizierte Sache, du brauchst eigentlich nur das zu wissen: Je höher der Hertzwert ist, desto höher klingt die Note.

Leider wird auch die Dauer ein bisschen kompliziert angegeben. Die Dauer wird angegeben als Anzahl von CPU-Takten, während derer der Sound zu hören ist. Insgesamt liegen in jeder Sekunde 18,2 CPU-Takte. Wenn du also willst, dass ein Ton eine Sekunde lang gespielt wird, gib als zweiten Wert für SOUND 18.2 an.

Das nächste Programm spielt alle Töne in einer FOR...NEXT-Schleife nacheinander ab, fängt also beim tiefsten an und wird dann immer höher. Weil es zu lange dauert, alle Töne zu spielen, und sich die Töne ziemlich ähnlich sind, setzen wir STEP auf 25, so wird nur jeder fünfundzwanzigste Ton gespielt.

```
FOR n = 40 TO 4000 STEP 25
   SOUND n, 1
NEXT
```

So können wir auch eine Sirene programmieren, die zuerst höher und dann tiefer wird:

```
FOR n = 450 TO 750 STEP 5
   SOUND n, 2
NEXT
FOR n = 750 TO 450 STEP -5
   SOUND n, 2
NEXT
```

PLAY

QBASIC besitzt noch einen Befehl, um Sound über den PC-Speaker auszugeben: PLAY. Mit PLAY kann man wirklich richtige Lieder spielen. PLAY hat sogar so etwas wie seine eigene Programmiersprache, aber keine Angst, die ist nicht sehr schwer zu lernen.

So sieht ein PLAY-Befehl aus:

```
PLAY "L4 C2 E G < B. > L16 C D L2 C"
```

Dieser Befehl sieht jetzt auf den ersten Blick etwas seltsam aus. Der String nach PLAY ist der Befehlsstring. Er weist QBASIC an, wie es Musik spielen soll.

L4 (für eine Länge von 4) teilt dem Computer mit, wie lange er die folgenden Noten abspielen soll. Das gilt für alle Noten nach diesem L, bis zum nächsten L, das eine andere Länge festlegt. 4 bezeichnet eine Viertelnote, 3 eine halbe Note, 2 eine punktierte halbe Note und 1 eine ganze Note.

Die Buchstaben hinter dem L sind die Noten auf der Tonleiter. Wenn neben einem Buchstaben eine Zahl steht, bezeichnet sie die Note für diesen Buchstaben, aber nur für diesen, die Buchstaben danach haben wieder die Länge, die ihnen das L vorschreibt.

- Das Größer-Symbol (>) und das Kleiner-Symbol (<) wechseln eine Oktave nach oben bzw. unten.
- Ein Punkt hinter einer Note »punktiert« diese, d.h. sie wird um die Hälfte ihrer Dauer verlängert.

Auf diese Weise kannst du fast jede Melodie spielen, die es gibt. Leider ist die Qualität ziemlich schlecht. Aber ohne Soundkarte lässt sich nicht mehr machen, und in QBASIC gibt es leider keinen Befehl, um die Soundkarte anzusteuern. Aber wenn du solche Programme schreiben willst, brauchst du eh ein besseres Programmierwerkzeug.

Subs und Functions

Du hast jetzt an vielen Beispielen gesehen, dass QBASIC-Programme einfach eingetippt und gestartet werden können. In der Praxis wird so natürlich nicht programmiert. Die Anweisungen werden in abgeschlossenen Programmen zusammengefasst, sogenannten Subprocedures oder abgekürzt *Subs* (subprocedure = Unterprogramm, denn im Prinzip kann jedes Programm das Unterprogramm eines anderen sein). *Functions* sind programmierte Funktionen. Im Unterschied zu Subprocedures können Funktionen nicht direkt gestartet werden, sie werden in der Praxis aus einem Programm heraus aufgerufen (Call Function).

Im *Bearbeiten*-Menü findest du Befehle, um Subs oder Functions zu erzeugen. Eine Sub ist ein Programm, das mit

```
Sub (Programmname)
```

beginnt und mit

```
End Sub
```

endet. Der Vorteil: du kannst viele solcher Subs erzeugen und diese wahlweise abrufen. Dazu setzt du einfach den Cursor in die gewünschte Sub und startest mit F5.

Functions beginnen mit

```
Function (Programmname)
```

und enden mit

```
End Function
```

Wie geht's weiter mit QBASIC?

Wie am Anfang schon erwähnt, wirst du mit dem »alten« QBASIC sicher nicht dein ganzen Leben lang programmieren, dazu ist die Sprache zu überholt, und die Möglichkeiten speziell im grafischen Bereich zu beschränkt. Wir haben noch gar nicht erwähnt, dass QBASIC nur Pro-

gramme unterstützt, die maximal 160 Kbyte Speicherplatz brauchen – unsere gigantischen Beispiele sind weit entfernt von dieser Byteverschwendung, aber in der Praxis sind solche Werte schnell erreicht, und da hilft nur ein Wechsel zu Visual Basic oder einer anderen Entwicklungsumgebung.

Aber du hast gesehen, dass die grundlegenden Strukturen einer Programmiersprache vorhanden sind, und für den Einstieg ist der alte BASIC-Veteran gar nicht so übel. Es gibt sogar QBASIC-Compiler, die aus Programmen ablauffähige EXE-Dateien machen, die müssen aber zusätzlich gekauft werden.

Im nächsten Kapitel wirst du Visual Basic kennen lernen, eine echte Windows-Programmiersprache. Was du mit QBASIC gelernt hast, kannst du vertiefen und wiederholen, und viele neue Überraschungen warten auf dich.