



## **Eine Einführung ins Programmieren von Anfang an**

### **Teil 2: Programmieren unter DOS**

Christof Schatz

Email: [schatz@askos.de](mailto:schatz@askos.de)

[www.askos.de](http://www.askos.de)



<a href="#"><u>DOS</u></a>	Seite 4
<a href="#"><u>Dateiverwaltung</u></a>	Seite 8
<a href="#"><u>Der Editor EDIT</u></a>	Seite 13
<a href="#"><u>Feine DOS-Fähigkeiten</u></a>	Seite 18
<a href="#"><u>QBASIC: Ein Vorwort</u></a>	Seite 24
<a href="#"><u>QBASIC: Erste Schritte</u></a>	Seite 26
<a href="#"><u>QBASIC/Visual Basic: Unterschiede zum C16-Basic</u></a>	Seite 28
<a href="#"><u>Hexadezimalzahlen</u></a>	Seite 37
<a href="#"><u>Grafik und einfache Sprites in QBASIC</u></a>	Seite 38
<a href="#"><u>Textdateien lesen und schreiben</u></a>	Seite 47
<a href="#"><u>Systemsteuerung mittels QBASIC</u></a>	Seite 54
<a href="#"><u>Zeiger</u></a>	Seite 59
<a href="#"><u>Exkurs: Grafiken in Postscript</u></a>	Seite 64
<a href="#"><u>Binäre Dateien</u></a>	Seite 67
<a href="#"><u>Einfache und zusammengesetzte Datentypen</u></a>	Seite 73
<a href="#"><u>Typisierte Dateien und das Datenbankprinzip mit QBASIC</u></a>	Seite 80
<a href="#"><u>Funktionen (SUB und FUNCTION)</u></a>	Seite 84
<a href="#"><u>Animation in 3D</u></a>	Seite 90

---

# DOS

---

## Programmieren unter DOS

Das Programmiersystem C16 ist übersichtlich und einfach, es hat rund zwanzig Befehle, mit denen man schon Einiges machen kann. Aber es ist nicht besonders komfortabel. Der Editor ist sehr umständlich und wenn man einmal ein, zwei Jahre programmiert, hat man ein Wirrwarr von Hunderten von Programmdateien und -versionen unsortiert auf Diskette. Ausserdem ist es langsam und hat keine Ahnung von all den Dingen, die ein PC heute kann.

Gehen wir also einen Schritt weiter, in dem Vorhaben, einen heutigen PC zu programmieren. Dazu werden uns als erstes ein bisschen mit DOS beschäftigen. DOS ist ein sehr einfaches *Betriebssystem*. Was das ist, werden wir später einmal genauer klären. Zunächst einmal reicht uns, darunter ein spezielles Executable zu verstehen, das uns hilft

1. Programme von einem Datenträger (Diskette, Festplatte) zu starten
2. Dateien auf der Festplatte zu verwalten.

In diesem und dem nächsten Kapitel werden wir uns also anhand DOS mit ganz einfachen Begriffen der Dateiverwaltung beschäftigen, denn die Arbeit mit Dateien ist unverzichtbar für das Programmieren.

Was Sie im folgenden lernen, ist allerdings nicht nur das veraltete Betriebssystem DOS. Sie lernen gleichzeitig die Grundzüge der Kommandozeilensteuerung der modernen Windows-Betriebssysteme wie WindowsXP, da diese mit der alten DOS- Kommandosprache identisch sind.

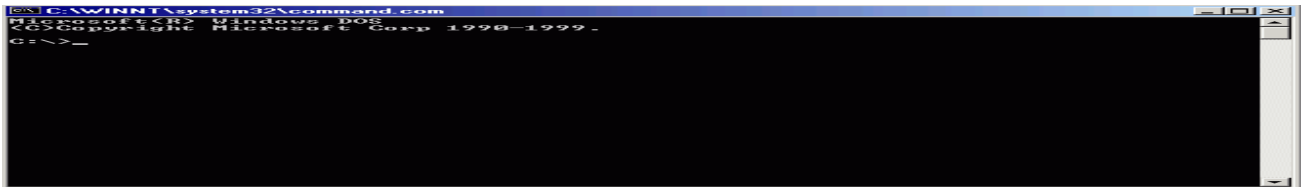
## Öffnen eines DOS-Fensters

Machen Sie einmal folgendes auf Ihrem (gestarten) PC:

- Klicken Sie auf den Start-Button
- Dann auf "Ausführen"
- Geben Sie command.com + ENTER ein.

Nun sollte ein schwarzes Fenster mit weisser Schrift erscheinen.





Die ersten beiden Zeilen brauchen uns nicht zu interessieren. Gehen wir gleich in die Zeile, in der "C:\>" steht. Das ist ein s.g. *Prompt*, der anzeigt, wo getippt wird. Was wir vor uns haben, ist eine Eingabeaufforderung, wie beim C16. Wir können ja versuchsweise mal "?1+1" eingeben und ENTER drücken. Daraufhin kommt eine lange Erklärung "Der Befehl ist entweder falsch geschrieben oder konnte nicht gefunden werden." Beim C16 hiess das noch "Syntax Error" und bedeutete: "Ich verstehe nicht, was du meinst". BASIC versteht dieses Programm offenbar nicht.

## Schliessen eines DOS-Fensters

Geben Sie einmal EXIT ein. Siehe da - das Fenster verschwindet!

EXIT ist aber kein BASIC-Befehl.

## DOS-Kommandos

Legen Sie einmal eine Diskette, vielleicht Ihre Diskette mit den BASIC-Programmen ein. Dann öffnen Sie ein DOS-Fenster und geben ein: **dir A:**. (Wenn Sie kein Diskettenlaufwerk besitzen, können Sie das gleiche mit C: machen: **dir C:**).

Was Sie nun sehen, ist eine Liste von Dateien, die Sie direkt auf Ihrer Diskette gespeichert haben. Links das Datum, dann die Uhrzeit der letzten Änderung. Falls ein DIR auftaucht, ist dies keine Datei. Ansonsten kommt die Grösse der Datei in Bytes und schliesslich der Name.

Wir sehen also: In dem schwarzen Fenster können wir Kommandos eingeben wie im C16-Fenster. Nur versteht das DOS keine BASIC-Kommandos, sondern hat seine eigenen. Wir haben es also mit einem *Befehlsinterpreter* zu tun (Sie erinnern sich? Bits, Bytes und Basic, Abschnitt "Interpreter und Compiler"), einem DOS-Befehlsinterpreter.

Im Gegensatz zum BASIC-Interpreter C16 hat der DOS-Interpreter allerdings nur wenige eingebauten Befehle. Das sind hauptsächlich die Befehle

```
dir    Zeige Inhalt des Verzeichnisses (directories)
mkdir  Erstelle neues Verzeichnis
rmdir  Lösche Verzeichnis
del    Lösche Datei
copy   Kopiere Datei
type   Zeige Datei
ren    Benenne Datei um
path   Setze oder zeige den aktuellen Pfad
```

(Wir gehen im nächsten Kapitel noch auf die einzelnen Befehle ein.)

## Programme starten unter DOS

Trotz diesen wenigen eingebauten Befehlen ist der DOS-Befehlsvorrat fast unerschöpflich. Geben Sie einmal in einem DOS-Fenster den Befehl **mspaint** ein (+ ENTER!). Es öffnet sich ein Windows-Malprogramm! Der Befehl **write** öffnet das Windows-Textprogramm. Wenn Sie **ver** eingeben, erhalten Sie die genaue Version Ihres Betriebssystems.

Woher kennt DOS all diese Befehle? Es sind Executables, die DOS startet, sobald Sie den Namen+ENTER eingeben. Diese Executables sind auf Ihrer **Festplatte** gespeichert. Und wo auf der Festplatte sind die Executables (exe- oder com-Dateien) gespeichert? Und wie findet DOS sie? Wichtige Fragen, die wir nicht alle auf einmal beantworten können. Also immer der Reihe nach...

Geben Sie einmal im DOS-Fenster folgendes ein, falls Sie ein Diskettenlaufwerk haben. (Sie sollten vorher eine Diskette einschieben, am Besten die mit Ihren C16-Programmen.)

```
path A:\  
A:
```

Diese nicht gerade intuitive Sequenz, die wir aber gleich verstehen werden, sorgt dafür, dass DOS die Festplatte sozusagen "vergisst" und nur noch die Diskette kennt.

---

*Falls Sie kein Diskettenlaufwerk haben:*

Falls Sie einen neueren PC ohne Diskettenlaufwerk besitzen, dann können Sie ein Diskettenlaufwerk simulieren. Geben Sie dazu folgende Befehlssequenz ein:

```
mkdir C:\dosworld  
subst A: C:\dosworld
```

Dann haben Sie auch ein Laufwerk A:, das Sie nach Herzenslust beschreiben und "belöschen" können. Geben Sie nun die oben angegebene Befehlssequenz path... ein.

---

Wir haben nun dafür gesorgt, dass DOS die mysteriösen Befehle mspaint, write usw. auf der Festplatte nicht mehr findet. Wenn Sie jetzt "write" abschicken, bekommen Sie eine Meldung der Sorte "Unbekannter Befehl oder unbekannter Dateiname". Sie sehen schon: Nach diesen zwei Dingen sucht der Interpreter, entweder nach einem internen Befehl (siehe obige Tabelle) oder nach einer ausführbaren Datei. Und er sucht, falls nichts anderes vorgegeben wurde, nach ausführbaren Dateien im aktuellen Verzeichnis auf dem aktuellen Laufwerk. Vermutlich befinden sich auf Ihrer Diskette keine ausführbaren Dateien. Das können Sie kontrollieren. Geben Sie einmal den Befehl **dir** ein. **dir** kennt DOS, da es ein in command.com eingebauter Befehl ist.

Dann sehen Sie die Dateiliste. Befindet sich unter ihnen eine Datei mit der Endung .exe oder .com, dann können Sie sie ausführen. (Von "regedit", "format" oder "fdisk" sollten Sie allerdings Abstand nehmen!) Einfach den Dateinamen + ENTER eingeben.

Wahrscheinlich befindet sich allerdings kein Executable auf Ihrer Diskette. Dann kopieren wir einfach eins drauf! Das geht mit dem Befehl **copy**. Es hängt allerdings nun stark davon ab, welche Windowsversion Sie haben. Sie erinnern sich an den **ver**- Befehl? Hier haben Sie festgestellt, welche Windows-Version Sie haben. Jetzt funktioniert er natürlich nicht mehr, da wir ja DOS alles ausser unserem Diskettenlaufwerk "vergessen" haben lassen. Aber kein Problem: Gehen Sie einfach nach START->Ausführen und öffnen Sie eine zweite command.com! Dann haben Sie ein zweites DOS-Fenster. Dieses weiss noch alles und ver funktioniert noch. Schliessen Sie allerdings danach dieses zweite Fenster gleich wieder, sonst gibt'S ein Wirrwarr...

Also, wir wollen ein Executable kopieren. Dazu brauchen wir Ihre Festplatte. Und zwar den Ort, wo

die Datei gespeichert ist. Wir wollen die Datei "edlin.exe" kopieren. Ihr Ort ist

Betriebssystem **DOS**:

C:\DOS

Betriebssystem **Windows 3.x**:

C:\DOS

Betriebssystem **Windows 95/98/ME**:

C:\Windows\command

Betriebssystem **Windows NT/2000/XP**:

C:\winnt\system32

Der Platzhalter für den String in dieser Tabelle (z.B. "C:\DOS") soll ORT heissen. Dann geben Sie folgenden DOS-Befehl ein:

A:\copy ORT\edlin.exe A:

Also z.B. "copy C:\DOS\edlin.exe A:".

Wenn Sie jetzt

A:\edlin.exe test.txt

aufrufen, erhalten, kommt

Neue Datei  
\*

edlin.exe ist hier mehr ein Gag. Es ist einer der allerersten Editoren. Und er funktioniert ganz ähnlich wie der C16-BASIC-Editor. [Hier nebenbei ein paar Hinweise dazu](#), wen's aus historischen Gründen interessieren sollte.

Wichtig ist hier nur: DOS kennt also jetzt den Befehl "EDLIN", da er das Executable im aktuellen Verzeichnis findet.

Wenn Sie "q+ENTER" eingeben und dann ein "j", sind Sie wieder draussen.

Wir können das gleiche mit edit.exe wiederholen.

A:\copy ORT\edit.exe A:

edit.exe ist der Nachfolger von edlin.exe und schon deutlich komfortabler. Mit ihm können Sie sich schon mal vertraut machen. Gleich nach dem Start wird Ihnen ja sogar eine kurze Einführung geboten. In einem der nächsten Kapitel werden wir hier kurz auf ihn eingehen.

Damit dürfte das Prinzip klar geworden sein: DOS kennt seine eingebauten Kommandos und diejenigen, die als Executable im aktuellen Verzeichnis zu finden sind.

---

# Dateiverwaltung

---

## Laufwerke

Wir haben bisher eigentlich nur ein Laufwerk kennengelernt: Das Diskettenlaufwerk. Und aus den bisherigen Erklärungen haben Sie ganz richtig geschlossen, dass das Diskettenlaufwerk unter DOS und Windows einfach A: heisst. Das ist übrigens weithingehend unabhängig vom PC und von der Betriebssystemversion.

Wenn Sie ein zweites Diskettenlaufwerk haben, ist dies aller Wahrscheinlichkeit nach Laufwerk B:. Dass die ersten beiden Buchstaben für so etwas Altmodisches wie Diskettenlaufwerke verwendet werden und die Festplatte erst den Buchstaben C: bekommt, hat historische Gründe. Also:

- A: 1. Diskettenlaufwerk
- B: 2. Diskettenlaufwerk
- C: Festplatte

So die vereinfachte Darstellung. Die dritte Zeile stimmt so schon nicht mehr ganz genau. Aber dazu weiter unten Genaueres.

Wie es nach C: weitergeht, das hängt sehr stark von Ihrem PC ab und wie er konfiguriert ist.

## Festplatten

Was ist eigentlich eine Festplatte? Nichts anderes als eine "Diskette", eine Magnetscheibe, die fest in Ihrem PC eingebaut ist. (Genauer gesagt sind es mehrere Scheiben übereinander, aber das ist hier nicht so wichtig). Sie dient genau dem gleichen Zweck wie ein Diskettenlaufwerk: Daten und Programme, die sich im (flüchtigen) Hauptspeicher befinden, zu sichern. Eine Festplatte ist viel schneller als eine Diskette und sie ist *viel, viel grösser*. Leider eigentlich viel zu gross, für unsere Zwecke hier. Denn selbst die kleinsten Festplatten, die man im Laden kaufen kann, sind heute so gross, dass man unweigerlich sehr schnell den Überblick darüber verliert. Eine heutige Festplatte speichert problemlos 1 Million Dateien(!). Eine Diskette ist nach 100 oder 200 Dateien spätestens voll. Daher ziehe ich es vor, die Dinge anhand einer Diskette zu erklären.

## Partitionen

Eine physikalische Festplatte kann für die Verwaltung in mehrere Teile, s.g. *Partitionen* aufgeteilt werden, die sich von DOS aus so verhalten, als wären sie unterschiedliche Festplatten. Daher bekommen sie auch unterschiedliche Buchstaben. Mein PC, an dem ich gerade sitze, hat zwei Partitionen, die die Buchstaben C: und D: haben. Jetzt verstehen Sie, warum ich vorhin nicht ganz korrekt war, als ich schrieb, C: wäre die Festplatte. C: ist *eine Partition* der Festplatte und oft ist da noch eine zweite oder dritte. Beispiel einer anderen Konfiguration, die ich kenne:

- A: Diskettenlaufwerk
- C: 1. Partition der Festplatte
- D: CD-ROM-Laufwerk



- E: 2. Partition der Festplatte
- F: 3. Partition der Festplatte
- G: Verzeichnis eines Laufwerks, das über Netzwerk verbunden ist.
- H: Wechseldatenträger ZIP

## Aktuelles Laufwerk

Das aktuelle Laufwerk ist dasjenige, auf dem der Interpreter Dateien und Verzeichnisse (s.u.) sucht. Es wird im s.g. **Prompt**, den Zeichen, die anzeigen, wo der Cursor steht, also z.B. **A:\>** angezeigt. Hier ist A: das aktuelle Laufwerk. (Der Doppelpunkt hinter dem Laufwerksbuchstaben dient übrigens dazu, den Laufwerksbuchstaben von einer gleichnamigen Datei zu unterscheiden.)

## Dateien und Verzeichnisse

Ein **Verzeichnis**, engl. **directory**, müssen Sie sich als einen Leitzordner vorstellen, der Ihre einzelnen Dokumente aufbewahrt. Es ist ein Name, unter dem Sie eine Menge von Dateien ablegen und ansprechen können. Die Notwendigkeit für Verzeichnisse kommt, wenn Sie mehr als zehn Dateien auf einem Datenträger (z.B. Diskette) verwalten, denn dann verlieren Sie bei einem DIR-Befehl die Übersicht auf dem Bildschirm.

## Ein paar DOS-Kommandos

### Kommando mkdir

Damit erzeugen Sie ein Verzeichnis

```
A:\>mkdir prog
```

erzeugt ein Verzeichnis namens "prog". Wenn Sie nun einen dir-Befehl eingeben, dann zeigt Ihnen dieser den Namen "prog" und dahinter statt einer Byte-Angabe das Wort <DIR>. Der Befehl "mkdir" entspricht also dem Befehl "Neuer Ordner"/"Neues Verzeichnis" im Kontextmenü des Windows- Explorer.

Mit "A:>dir prog" kann man sich den Inhalt von prog anzeigen lassen. Steht natürlich nichts drin.

### Kommando copy

...kopiert eine Datei. Man muss das Zielverzeichnis angeben.

```
A:\>copy edlin.exe prog
```

...kopiert die Datei edlin.exe nach prog. prog ist Zielverzeichnis. Anschliessend zeigt ein "A:\>dir prog" eine Datei edlin.exe an, von der es ja nun auch ein Exemplar in prog gibt.

Ist das Zielverzeichnis gar kein Verzeichnis, dann nimmt "copy" an, dass es sich um einen Dateinamen handelt, und zwar um den Namen der Kopie, die angelegt wird.

```
A:\>copy edlin.exe edlin2.exe
```

macht von edlin.exe eine Kopie namens edlin2.exe.

## Kommando del

"del edlin2.exe" löscht die Datei edlin2.exe. del kann nur auf Dateien, nicht auf Verzeichnisse angewandt werden.

## Kommando cd

...heisst "change directory" = Wechsle das aktuelle Verzeichnis.

```
A:\>cd prog
```

..wechselt ins Verzeichnis prog. Anschliessend zeigt der Prompt das *aktuelle Verzeichnis*

```
A:\prog>
```

an.

## Pfade

Ein "Pfad" ist der Weg, den man über die Verzeichnisse gehen muss, von der Wurzel des Laufwerks in die Unterverzeichnisse hinein, um eine Datei zu finden. Ein Pfad wird über den *Pfadnamen* definiert. Jede Datei hat einen Pfadnamen. Er besteht aus der Angabe sämtlicher Verzeichnisse und Unterverzeichnisse, in denen die Datei liegt, samt dem Dateinamen selbst. A:\prog\edlin.exe ist der (absolute) Pfadname der Datei edlin.exe. Man beachte

- Verzeichnisse und Unterverzeichnisse werden durch ein \ (backslash) voneinander getrennt.
- Ein führendes \ bezeichnet das *Stammverzeichnis* des aktuellen Laufwerks, also sozusagen das Laufwerk selbst. Mit "\prog" benennt man ein Verzeichnis namens "prog", das im Stammverzeichnis des aktuellen Laufwerks liegt (egal, ob letzteres gerade A:, B: oder K: ist).
- Lässt man das führende \ weg, dann ist der Rest ein Pfad relativ zum aktuellen Verzeichnis. Ist das aktuelle Verzeichnis A:\prog und gibt man als Pfadnamen "edit\edlin.exe" an, so meint man damit "A:\prog\edit\edlin.exe".
- Ein einzelner Punkt bezeichnet das aktuelle Verzeichnis.

```
A:\>cd A:\prog  
A:\prog>copy A:\edlin.exe .
```

kopiert also die Datei edlin.exe ins aktuelle Verzeichnis.

- Zwei Punkte in Folge ("..") bezeichnen das übergeordnete Verzeichnis ("Oberverzeichnis"). \ ist das Oberverzeichnis zu \prog und \prog ist das Oberverzeichnis zu \prog\edit.

```
A:\>cd A:\prog\edit  
A:\prog\edit>cd ..  
A:\prog>
```

## Weitere DOS-Kommandos

### Kommandos abbrechen

Mit Ctrl-C (bzw. Strg-C) können Sie jederzeit die Aktivität eines Programms abbrechen.

### Kommando rmdir

...löscht ein Verzeichnis, allerdings erst dann, wenn es leer ist. Man muss also vorher alle Dateien

des Verzeichnisses mit `del` und alle Unterverzeichnisse mit `rmdir` löschen. Sie werden fragen: Und wie lösche ich ein Verzeichnis mit allen Dateien und Unterverzeichnissen auf einen Satz? Gar nicht. Sie vergessen hier eines: Das Absenden solch eines Kommandos kann Sie um Wochen und Monate Ihrer Arbeit bringen. Deshalb haben es die Schöpfer von DOS nicht vorgesehen. Und noch etwas: Unter Windows können Sie versehentlich gelöschte Dateien aus dem Papierkorb zurückholen. Nicht so hier unter DOS. Was weg ist, ist weg!

## Kommando format

Und was machen Sie, wenn eine Diskette mit Hunderten von Verzeichnissen und kleinen Dateien bespielt ist? Und diese Diskette z.B. von irgendeinem fremden "Müllberg" stammt, so dass keine Gefahr besteht, dass Sie wertvolle Dateien löschen? Nun, dann können Sie die Diskette neu *formatieren*. Aber gehen Sie damit **sehr vorsichtig um**. Sie können nämlich, wenn Sie sich nicht konzentrieren, Ihren ganzen Computer kaputtmachen!

Was heisst "Formatieren"? Die Diskette braucht, damit der Computer damit umgehen kann, eine Einteilung in s.g. Sektoren, das Diskettenformat eben. Und diese Einteilung wird beim Formatieren neu hergestellt. Dabei gehen alle Daten auf der Diskette verloren.

Der Befehl lautet "`format <Laufwerk>`". `<Laufwerk>` kann dabei nicht nur ein Diskettenlaufwerk sein, sondern - und das ist das Gefährliche - auch eine Festplattenpartition! Wenn Sie also eine Diskette neu formatieren, dann schliessen Sie einmal kurz die Augen, entspannen sich, nehmen dann all Ihre Konzentration zusammen und beginnen Sie Ihr Werk in dem Bewusstsein, dass Sie da etwas höllisch Wichtiges und Gefährliches machen.

Falls der Formatierungsvorgang selbst noch nicht gestartet ist (Eine Prozentzahl wird langsam nach oben gezählt), können Sie ihn mit `Ctrl-C` jederzeit abbrechen.

## DOSKEY

DOSKEY verbessert den Komfort des DOS-Interpreters. Einfach `DOSKEY+ENTER` eingeben und Sie haben die DOSKEY-Funktionen zur Verfügung. Wichtigste davon: Mit der Pfeil-nach-oben-Taste können Sie sich vorhergegangene Kommandos zurückholen.

## DOS-Hilfe und Kommandozeilenparameter

Für unsere Programmierzwecke werden uns diese DOS-Kenntnisse ersteinmal reichen. Aber es gibt noch viel mehr Benutzerkomfort unter DOS. Um eine Hilfe zu einem bestimmten Befehl zu erhalten, gibt es zwei Möglichkeiten.

- "`/?`" eintippen.
- "`help`" eintippen.

Ersteres funktioniert immer, letzteres nur bei eingebauten DOS-Kommandos.

Insbesondere bekommen Sie über diese Hilfe auch Informationen über die *Kommandozeilenparameter* eines Befehls. Mit den Parametern können Sie die Wirkungsweise eines Befehls beeinflussen. Beispiel:

```
dir /s
```

listet den Inhalt des aktuellen Verzeichnisses *und aller Unterverzeichnisse* auf. "*..und aller*

Unterverzeichnisse", das bewirkt der Parameter "/s". Wenn Sie

`dir /?`

bekommen Sie eine Liste aller Kommandozeilenparameter von `dir`. Da können Sie auch `-a`, `-d`, `-o` oder andere Parameter eingeben. Entsprechendes gilt für `copy`, `del` usw.

---

# Der Editor EDIT

---

## Überblick: Editoren

Das A und O beim Programmieren ist der Editor, das Werkzeug, mit dem wir die Programmtexte erstellen. Beim C16 war der Editor im BASIC-Interpreter integriert. Es wäre gar nicht möglich gewesen, ein Programm ausserhalb des Interpreters zu erstellen.

Interpreter, bzw. Compiler und Editor - das sind aber eigentlich zwei ganz verschiedene Paar Stiefel. Nur hat das "Paar" Stiefel noch ein drittes Mitglied, das wir später in diesem Teil kennenlernen werden: Den *Debugger*. Alle sind im Prinzip getrennte Programme, die hintereinander oder nebeneinander her laufen. Der Editor ist dabei natürlich besonders wichtig, denn er ist das Tor zum Programmieren, das Programm, innerhalb dem wir uns die meiste Zeit bewegen. Der Editor ist der Schreibtisch, der Bürosessel und der Büroraum des Programmierers zusammen. Hier bewegt er sich und entfaltet sich, auf ihn starrt er stundenlang. Hier muss das "Feeling" stimmen und er muss sich wohlfühlen.

Auf Unix, einer heute immer beliebter werdenden Betriebssystemfamilie, programmieren viele Profis mit dem Editor *Emacs*. (Ich selbst habe bisher noch keine Ahnung davon.) Emacs ist so etwas wie der König der Editoren. Seine Funktionsvielfalt sucht seinesgleichen. Er hat einen eigens eingebauten Interpreter, der nur dazu da ist, Schreibvorgänge zu automatisieren! Man kann von ihm aus Email abrufen, Newsgroups anschauen, den Compiler und den Debugger aufrufen und vieles mehr. Die Programmierer sitzen also den ganzen Tag - nein, nicht vor ihrem PC, vor ihrem Betriebssystem oder ihrem Desktop. Das einzige, was sie wahrnehmen: Sie sitzen vor "ihrem" Emacs...

Für uns ist es noch nicht so zentral, welchen Editor wir benutzen. Aber wir sollten möglichst bald die Grundfunktionen kennenlernen und nutzen lernen, die jeder einigermaßen anständige Editor mitbringt (und erst recht auch jede Textverarbeitung!)

Wir werden uns jetzt mit dem Editor beschäftigen, den das nächste BASIC mitbringt, das wir kennenlernen werden. Da das nächste BASIC noch unter DOS funktioniert, ist auch der Editor ein DOS-Editor. Das macht aber nichts, denn bis auf eine Ausnahme (Copy und Paste) funktionieren die Tasten und Menüs ganz ähnlich, wie bei seinem Windows-Pendant notepad.exe.

## Wozu kann man Editoren verwenden?

Editoren kann man zu einer Menge nützlicher Dinge verwenden, keineswegs nur fürs Programmieren:

- Erstellen von Notizen
- Erstellen von Protokollen, Inhaltsverzeichnissen, Adressverzeichnissen u.a.
- Lesen von readme-Dateien
- Lesen und Verändern von Konfigurations- und Datendateien, die zu Programmen gehören.

## Wo finde ich edit.exe?

- **DOS oder Windows 3.x:** Er ist schon da. Einfach "edit" aufrufen.
- **Neuere Windows-Versionen (9x/ME/NT/2000/XP):** [Hier herunterladen. \(K\)](#) . Die Hilfedatei edit.hlp dazu [gibt's hier](#). (21 K) Beide Dateien müssen in einem Verzeichnis des aktuellen Pfads liegen.

## Erste Schritte

Starten wir erstmal den Editor. Als erstes bietet der Editor gleich eine Hilfe an. Ziemlich komfortabel. Da Sie aber das Tutorial hier schon haben, brauchen Sie die Hilfe vorläufig noch nicht. Drücken Sie also einfach die ESC-Taste.

Nun haben Sie einen leeren Bildschirm vor sich, der auf Eingaben wartet. Schreiben Sie ersteinmal drauf los. Z.B. den Satz

Ich bin eine Datei und warte.

Anhand dieses Satzes können Sie folgende Editorkommandos ausprobieren:

**Backspace** Löschen rückwärts - wie beim C16.  
**Del** Löschen vorwärts  
**Pfeil nach links** Cursor ein Zeichen nach links  
**Pfeil nach rechts** Cursor ein Zeichen nach rechts  
**Pfeil nach oben** Eine Zeile nach oben  
**Pfeil nach unten** Eine Zeile nach unten  
**Bild nach oben** Eine Seite nach oben  
**Bild nach unten** Eine Seite nach unten  
**Strg+Pfeil nach rechts** Ein Wort weiter nach rechts  
**Strg+Pfeil nach links** Ein Wort weiter nach links

Das dürften Sie schnell ausprobiert haben. Edit verhält sich also viel komfortabler wie der Editor des C16 und in etwa so wie eine Textverarbeitung.

## Bedienung der Menüs

Am oberen Rand des DOS-Fensters haben Sie eine graue Leiste mit den Namen "Datei", "Bearbeiten", "Suchen", "Optionen". Das ist die Menüleiste des Editors. Das Menü können Sie mit der Maus anklicken, aber das ist für eine schnelle Bedienung viel zu umständlich. Besser, Sie benutzen die Tastatur.

**Alt+weißer Buchstabe** bringt Sie zum Menü. Alt+D also in das Menü Datei. Dann mit den Pfeiltasten rauf und runter gehen. Sie sehen, dass Sie hier gleich wichtige Befehle, die wir vom C16 auch noch kennen, vor sich haben: "Speichern", bzw. "Speichern unter" entspricht dem BASIC-Befehl "Save" und "Öffnen" entspricht dem BASIC-Befehl "LOAD". Speichern Sie gleich einmal Ihren Text, z.B. unter dem Namen "info.txt". Sie sollten allen Texten den Suffix .txt geben, damit man gleich dem Dateinamen ansieht, ob man die Datei mit einem Editor anschauen kann oder nicht. Ich rate Ihnen nicht, zu versuchen, eine .exe-Datei mit dem Editor anschauen zu wollen! Das macht keinen Spass... Wenn Sie aber nun eine Textdatei mit der Endung .exe versehen, dann werden Sie sie als Textdatei nicht wiedererkennen!

## Markieren, Copy und Paste

Schon auf dem C16 wird es Ihnen so gegangen sein, dass Sie oftmals einen ganzen Textblock oder Textabschnitt hätten bearbeiten wollen. Zeilenweise ging das ja auch: Man konnte einfach einer Zeile eine neue Zeilennummer verpassen und schon war sie kopiert.

Hier im EDIT geht das viel komfortabler. Man markiert einen Textblock und kann diesen dann auf einen Tastendruck duplizieren, verschieben oder löschen.

**Markieren** können Sie einen Textblock, indem Sie während der Cursorbewegung die Shift-Taste (Umschalttaste) gedrückt halten. Also:

- Shift+Pfeil-rechts markiert zeichenweise nach rechts.
- Shift+Pfeil-unten markiert zeilenweise nach unten.

usw.

Und was machen wir, wenn wir markiert haben?

## Die Zwischenablage

Wir können die Markierung in die s.g. *Zwischenablage* (engl. "Clipboard") kopieren. Was ist die Zwischenablage? Das, was der Name sagt: Ein unsichtbarer Speicher im Hintergrund, der den markierten Textblock aufnehmen kann. (Als BASIC-Programmierer ganz verständlich: Einfach eine Stringvariable, die "Zwischenablage" heisst). Man kann den Textblock in diese Zwischenablage kopieren und von dort beliebig oft an verschiedenen Stellen im Text wieder einfügen.

## Cut und Copy

Für das Kopieren gibt's zwei Befehle:

- **Copy:** Strg+Einfg (Ctrl+Ins) kopiert den markierten Block in die Zwischenablage, lässt ihn dabei im Text.
- **Cut:** Umsch+Entf (Shift+Del) kopiert den markierten Block in die Zwischenablage und löscht ihn im Text.

## Paste

Um den Text an anderer Stelle wieder einzufügen, bewegt man den Cursor an die passende Stelle und macht ein **Paste**. Und zwar mit den Tasten Umsch+Einfg (Shift+Ins).

Wenn Sie diese Funktionen ein paar Mal geübt haben, dann möchten Sie sie nicht mehr missen, das kann ich Ihnen versprechen! Und - es ist auf diese Art und Weise gar keine so grosse Kunst, ein, zwei tausend Zeilen Programmcode zu schreiben. Man muss ja nicht jeden Buchstaben einzeln eingetippt haben...

## Zwischenablage von DOS und Windows

Diese sind leider unterschiedlich. Und zwar sowohl der Speicher selbst - Sie können nicht so einfach eine DOS-Zwischenablage in Windowsprogramme einfügen - als auch in der Bedienung. Hier ein paar kurze Informationen dazu.

**Bedienung Copy und Paste unter Windows (z.B. Notepad oder Word):**

- Copy: Strg+C
- Cut: Strg+X
- Paste: Strg+V

### Kopieren vom DOS-Editor nach Windows:

Das können Sie nur mit der Maus. Gehen Sie auf das rot-orange MSDOS-Symbol ganz links oben in der Ecke des DOS-Fensters und klicken Sie es an. Dann gehen Sie auf "Bearbeiten" und "Markieren". Nun können Sie mit der Maus einen Textblock markieren. Wenn Sie dann die ENTER-Taste drücken, ist der Textblock in die Windows-Zwischenablage kopiert und Sie können ihn in Word oder Wordpad o.ä. einfügen. Umgekehrt geht's auch. Genauso auf das MSDOS-Symbol klicken, auf "Bearbeiten" gehen und "Einfügen" anklicken und schon wird an der Stelle, an der der Cursor im DOS-Fenster steht, der Windows-Zwischenablage- text eingefügt.

## Suchen und Ersetzen

Sowohl in "normalen" Texten, als auch in Quelltexten von Programmen kann es sehr nützlich sein, eine Suchfunktion zu benutzen. Die Bedienung ist ganz einfach: Alt+S+Enter und den Suchbegriff einfügen. Dann ENTER drücken. Soll nach der ersten Fundstelle weiter nach diesem String gesucht werden, dann einfach F3 drücken.

Man braucht ein bisschen Erfahrung, um herauszubekommen, was man mit so einer schlichten Suchefunktion alles anstellen kann.

- **Datenbank** Für viele Zwecke reicht die einfache Suchefunktion aus, um einen Namen wiederzufinden. So ist eine normale Textdatei als Adressverzeichnis in vielen Fällen völlig hinreichend. Der Vorteil ist, dass man keine Felder vordefinieren muss und dass man automatisch eine Volltextsuche startet.
- **Retrieval System** Haben Sie das Problem, sich dauernd Notizen zu machen und sie dann nicht wiederzufinden? Hatte ich auch lange. Bis ich auf die Idee kam, alles, was mir "vor die Flinte kommt", Kochrezepte, Ideen, Hinweise, Witze, Tricks usw. und was nicht als Email oder Webadresse automatisch zugeordnet ist, in *eine einzige* Textdatei zu schreiben. Die trage ich immer mit mir herum. Es ist nicht wo wichtig, wo was steht. Wichtig ist nur, dass jeder Eintrag mit einem oder mehreren aussagekräftigen Schlüsselwörtern versehen ist. Dann kann man mittels "Suchen" sehr schnell wiederfinden, was man benötigt.
- **Labels.** Oft wird es vorkommen, dass Sie an einer anderen Textstelle etwas nachschauen und dann an diese hier wieder zurückspringen müssen. Ganz einfach: Sie überlegen sich einen charakteristischen kurzen String, bei mir ist das %%. Daran hänge ich noch eine Ziffer. Also %%1. Den füge ich in den Text ein. Dann suche ich nach der Textstelle, an der ich etwas nachschauen möchte. Und kehre mit SUCHEN nach "%%1" wieder an die vorige Stelle zurück. Auf die gleiche Weise kann man einen Text mit mehreren solchen Labels versehen und sie dann mittels des SUCHEN-Befehls anspringen. So könnten Sie z.B. ein Programm mit drei oder viel Labels in Kommentarzeilen versehen, mit deren Hilfe Sie dann diese drei oder vier wichtigen Stellen (z.B. Unterprogramme) wiederfinden. Aber auch beim oben beschriebenen Retrieval System sind Labels ganz nützlich. Sie können z.B. alle Witze hinter ein Label %%WITZE schreiben. Dann müssen Sie nur SUCHEN "%%WITZE" ausführen und sind bei Ihren Witzen. Und trotzdem ist alles noch in einer Textdatei!

**ERSETZEN** funktioniert analog zu Suchen: Mit ALT-S das "Suchen"- Menü aufrufen und auf "ÄNDERN" gehen. Dann den Such-String eintragen und gegen was er ersetzt werden soll. Zwischen den Kästchen können Sie **mit der Tabulator-Taste hin- und herspringen.**



- SUCHEN und BESTÄTIGEN führt das Ersetzen interaktiv durch. EMPFOHLEN! Sie werden bei jedem Fund gefragt, ob Sie diesen speziellen Suchtext ersetzen wollen oder nicht.
- ALLES ERSETZEN fragt nicht nach, sondern ersetzt alles ungefragt in einem Rutsch. NICHT EMPFOHLEN! In aller Regel geht das schief, da Sie nach dem ersten Ersetzen merken, dass Sie nicht ganz die richtigen Such- oder Ersetzen- Strings eingegeben haben und nun Quatsch dasteht.

Auch ERSETZEN (Ändern) ist eine wichtige Funktion der Editoren, wie Sie bald merken werden. Haben Sie z.B. beim Programmieren einen falschen Variablennamen benutzt oder wollen diesen Variablennamen nachträglich ändern, so ist das mit ERSETZEN gar kein Thema, von Hand wäre die Arbeit aber je nach Länge des Programms mühsam bis fürchterlich.

## **Sonstiges**

Der Editor hat noch ein paar andere Funktionen. So können Sie drucken (ein ganz einfacher "Schreibmaschinen"-Ausdruck)

---

# Feine DOS-Fähigkeiten

---

Im vorletzten Kapitel wurde in die grundlegenden Funktionen von DOS eingeführt: Sich im Verzeichnisbaum bewegen, Dateien erzeugen, kopieren, löschen. Nichts, was man nicht mit einem Dateimanager wie dem Windows-Explorer auch könnte.

In diesem Kapitel kommen nun DOS-Fähigkeiten zur Sprache, die schon näher ans Programmieren heranführen. Und die in Verbindung mit einer Programmiersprache ganz erstaunliche Dinge ermöglichen. Das Prinzip dieser "erstaunlichen Dinge" sei schon vorweggenommen: Wir können später von unseren selbst geschriebenen Programmen aus DOS-Kommandos absetzen. Und damit all die DOS-Funktionen in unseren Programmen nutzen, die wir schon kennengelernt haben und noch kennenlernen werden. Und das ist der wesentliche Unterschied zu ihrem Dateimanager: Den können Sie aus einem eigenen Programm heraus nicht benutzen und automatisieren (jedenfalls nicht so ohne weiteres.)

## Wildcards

### Suffixe

Ein Dateiname bestand unter DOS immer aus bis zu acht Buchstaben Name und 3 Buchstaben Suffix. Also z.B. "command.com" oder "meintext.txt". Diese Regelung wurde durch DOS von seinem Vorläufer CP/M übernommen. Die 8-Buchstaben- Grenze ist längst gestrichen. Heute kann man Dateinamen mit über 200 Buchstaben generieren - wem das Spass macht. Aber die Suffixe werden weiterhin beibehalten. Sie sind sogar elementar geworden. Denn anhand ihnen wird identifiziert, um was für einen Dateityp es sich handelt und mit welchem Programm man ihn bearbeiten kann. Ein paar bekannte Suffixe und die in der Regel zuständigen Programme sind:

Suffix	Zuständiges Programm
doc	Microsoft Word
xls	Microsoft Excel
pdf	Acrobat Reader-Datei
html	WWW-Browser
txt	Editor
gif	Bildbetrachter
jpg	Bildbetrachter
mp3	MP3-Player (Musikprogramm)
mpeg	Video-Player
avi	Video-Player
bas	Basic-Interpreter

u.v.m.

## Auswahl von Suffixen

Suffixe machen es also theoretisch möglich, alle Dateien eines Typs auf einmal zu behandeln oder auszuwählen. Man muss nur alle Dateien mit einem bestimmten Suffix ansprechen. Geht das aber auch praktisch? Klar doch.

DOS beherrscht s.g. "Wildcards". Das sind Symbole im Dateinamen. Entweder ein ? oder ein \*. Wesentlich wichtiger als das ? ist \*. Er steht für einen beliebigen String. Das heisst, es werden durch diesen Namen alle Dateien bezeichnet, deren Name mit den angegebenen Buchstaben übereinstimmt, aber an der Stelle des \* irgendetwas stehen haben.

Ein Beispiel sagt hier mehr als tausend Worte: "n\*t" steht für alle Namen, die mit n beginnen und mit t enden, also "nagt", "näht", "numeriert", "ndjskdjht", "n182kt".

"\*.txt" steht für alle Namen, die mit ".txt" enden, also "a.txt", "meintext.txt", "hallo.txt" usw.

"hallo\*.\*" steht für alle Namen, die mit "hallo" beginnen, also "hallo1.txt", "hallodies.bas", "hallohallo.doc".

Das ? hingegen steht für genau einen beliebigen Buchstaben. "hallo.???" z.B. meint "hallo.txt", "hallo.doc", aber nicht "hallo.html".

## Anwendung

Vielen DOS-Befehlen kann man oder muss man Dateinamen beifügen. Durch Verwendung von Wildcards kann man nun ganz flexibel ganze Dateigruppen mit einem Schlag behandeln. Ein paar Beispiele

dir *.txt	Zeige alle Textdateien
dir /s *.txt	Zeige alle Textdateien in diesem Verzeichnis und allen Unterverzeichnissen.
type *.htm	Zeige den Inhalt aller HTML-Dateien auf dem Bildschirm an.
xcopy /s *.bas A:	Kopiere alle Basic-Programme aus dem aktuellen Verzeichnis und allen Unterverzeichnissen (Option /s) auf eine Diskette
xcopy /s *.* A:	Kopiere ALLE Dateien aus dem aktuellen Verzeichnis und allen Unterverzeichnissen (Option /s) auf eine Diskette
xcopy /s /d:01-01-2003 *.* A:	Kopiere alle Dateien, die seit dem 01.01.2003 verändert oder erstellt wurden, aus dem aktuellen Verzeichnis und allen Unterverzeichnissen (Option /s) auf eine Diskette
ren a*.jpeg a*.jpg	Benenne alle Bilder, die mit a anfangen und die Endung .jpeg tragen, in .jpg um.

## Wildcards für beliebige Befehle

Wildcards sind zunächst keine Eigenschaft von DOS, sondern lediglich eine Konvention unter DOS. Die Interpretation der Wildcards muss das entsprechende Programm selbst übernehmen (ganz im Gegensatz zu Unix, das Wildcards für das entsprechende Programm in explizite Dateinamen übersetzt).

Das heisst: Ausser dir, copy, xcopy und hier und da ein paar Programmen können wir keine

Wildcards verwenden. Denken wir. Es gibt jedoch einen Befehl, der das ermöglicht: Eine FOR-Schleife.

Syntax: **for %1 in () do %1**. Das sieht wilder aus, als es ist. Das "%1" ist einfach ein formaler Platzhalter für die Datei, damit hinten DOS weiss, an welcher Stelle der Befehl den Dateinamen haben möchte. Für können wir irgendeinen Wildcardnamen einsetzen, z.B. \*.\* oder \*.txt oder a\*.\*. kann irgendein DOS-Executable sein. Und %1 setzen wir an die Stelle, an der der Befehl den Dateinamen haben möchte. Anwendungsbeispiel: Der Befehl grep sucht in einer Textdatei nach einem String. z.B. "grep "Hallo!" a.txt" sucht in a.txt nach dem String "Hallo!". Nun setzen wir das in Verbindung mit FOR ein: *for %1 in (\*.txt) do grep "Hallo!" %1*. Dies durchsucht alle Dateien mit der Endung .txt nach Strings "Hallo!" und printet die Fundstellen auf den Bildschirm.

Noch eine kleine lustige Anekdote zu Wildcards, die sich während meiner Diplomarbeitszeit ereignet hat. Das Betriebssystem, auf dem alle arbeiteten, war UNIX. Da aber viele Dinge bei DOS von UNIX abgeschaut sind, (auch die Pipes im folgenden Abschnitt), verwundert es nicht, dass die Wildcards bei UNIX fast genau gleich funktionieren. Nur gibt es dort noch mehr und man kann noch viel kompliziertere Dinge mit ihnen tun. Und: Suffixe gibt es unter Unix nicht zwingend. Wenn man unter DOS alle Dateien ansprechen möchte, muss man "\*.\*)" formulieren, unter UNIX nur "\*". Der del-Befehl heisst unter Unix übrigens "rm".

Eines Tages kam ein Mitarbeiter sehr zerknirscht zum System-Administrator und fragte ihn: "Du, wann ist denn das letzte Backup gemacht worden?". Alle horchten auf. Was war passiert? Der Mitarbeiter war für eine Stunde von seiner Workstation weggegangen. Als er wiederkam, befand sich eine Datei namens "\*" in seinem Verzeichnis. Das störte ihn. Also löschte er sie: "rm \*". Den Rest können Sie sich denken...

## Umleitungen und Pipes

Für DOS ist alles eine Datei. Nicht nur die Datei auf dem Datenträger, sondern auch der Bildschirm und die Tastatur. Schreibt ein Programm auf den Bildschirm, so schreibt es in Wirklichkeit in eine Datei, die laufend auf dem Bildschirm angezeigt wird. Diese nennt man die "Standardausgabe". Entsprechend gibt es auch die "Standardeingabe", die Datei, aus der ein Programm die Tastatureingaben liest.

Der Vorteil dieses Prinzips ist, dass man diese Datei ändern kann. Die Standardeingabe muss nicht zwingend mit der Tastatur verbunden sein. Sondern man kann DOS auch anweisen, ein Programm so auszuführen, dass die Standardeingabe eine Datei ist. Das Programm nimmt also an, es bekäme seine Eingaben über die Tastatur, in Wirklichkeit kommen sie aber aus einer Textdatei. Das ist sehr praktisch. Fragt ein Programm z.B. über die Tastatur eine Menge Informationen ab, so kann ich diese auch einfach mit einem Editor in eine Datei schreiben und dem Programm diese Datei "vor die Nase setzen".

Und wie geht das? <Befehl> > <file> leitet die Standardausgabe auf <file> um. Schreibt <Befehl> also auf den Bildschirm (und das tun die meisten Befehle), dann geht die Ausgabe jetzt in die Datei <file>. "dir > a.lst" schreibt z.B. die Verzeichnisinformationen in die Datei a.lst. Sehr viel mehr sinnvolle Anwendungen der Dateiumleitung mit den elementaren DOS-Befehlen finde ich gerade nicht, aber diese ist sehr praktisch. Praktisch sind auch Dateiumleitungen mit selbst geschriebenen Programmen - dazu später mehr.

Das Zeichen für die Umleitung der Dateieingabe ist übrigens <

Wir können das for-Schleifen-Beispiel von oben noch durch eine Umleitung ergänzen, da es

praktisch wäre, das Ergebnis der Stringssuche in einer Datei zu haben, wo man hin- und herblättern kann. Das geht ganz einfach: *for %I in (\*.txt) do grep "Hallo!" a.txt > a.lst*. Anschliessend sind die Fundstellen in a.txt aufgelistet.

Dann gibt es noch die Pipes. Man kann eine ganze Folge von DOS-Befehlen miteinander verketten, indem man die Standardausgabe des ersten Befehls zur Standardeingabe des nächsten macht. Das nennt man "Piping". ("Pipes" sind auf deutsch "Röhren", hier ist insbesondere die Rohrpost gemeint.) Das einzige Piping, das ich bisher verwendet habe, ist "type <file> | more". "more" ist ein Programm, das von der Standardeingabe liest und das Ergebnis gleich wieder auf dem Bildschirm anzeigt - wenn der Bildschirm voll ist, wartet es auf einen Tastendruck. Auf diese Art und Weise kann man den type-Befehl dazu einsetzen, eine lange Textdatei bildschirmseitenweise auszugeben. Soweit ich weiss, funktioniert das Piping auch nicht so gut unter Windows, da es eigentlich den synchronen Betrieb zweier Programme und das Lesen einer gemeinsamen Datei erfordert, was zumindest unter DOS eigentlich nicht ging.

## Batch-Dateien

Nehmen wir an, wir wollen jeden Tag ein Backup unserer Programmierarbeit von Festplatte auf Diskette machen. Wir arbeiten mit fünf Dateien, die myprog?.bas, descr1?.txt, dat1?.dat, mytab?.dat und toolut?.bas heissen sollen. Das ? steht jeweils für eine Ziffer, die die Version angibt. Es soll jeweils nur die neueste Version gesichert werden. Noch dazu sollen die verschiedenen Dateien in verschiedene Verzeichnisse auf der Diskette: Daten sollen nach A:\data und Programme nach A:\prog. Was tun? Jedesmal fünf copy-Befehle auf DOS-Ebene eingeben? Oder fünfmal Dateien im Explorer packen und in das jeweilige Verzeichnis ziehen? Das muss nicht sein!

Die Lösung ist auch ganz einfach: Wir schreiben die fünf Copy-Befehle in eine Textdatei und speichern sie ab. Dann haben wir ein "DOS-Programm". Und das geht tatsächlich. Nennen wir die Datei nämlich mit der Endung .bat, können wir sie wie einen eigenen DOS-Befehl verwenden. Man nennt das einen *Skript*.

Wir schreiben also mit dem Editor in eine neue Datei

```
copy C:\prog\myprog7.bas A:\prog
copy C:\prog\ toolut7.bas A:\prog
copy C:\prog\dat17.dat A:\data
copy C:\prog\descr17.txt A:\data
copy C:\prog\mytab1.dat A:\data
```

...und speichern sie als "backup.bat" ab. Anschliessend müssen wir nur noch backup als DOS-Befehl eingeben und die Copy-Befehle werden hintereinander ausgeführt.

## Die Skriptsprache von DOS

...ist leider ziemlich primitiv und mit BASIC nicht zu vergleichen. Daher gibt es inzwischen eine ganze Reihe von (meist kostenlosen) Skriptinterpretern - bis hin zum mächtigen Perl - die mehr können. Für den Anfang wird uns jedoch die Standard-DOS-Skriptsprache genügen. Vor allem, wenn wir sie in Verbindung mit BASIC und anderen Programmiersprachen einsetzen können.

Im folgenden also die Sprachelemente

### echo

...gibt einfach eine Meldung aus.

Beispiel: *echo Hallo! Das ist ein Skript!* Keine Anführungszeichen um den Ausgabestring!

Standardmässig gibt DOS ohnehin jeden Skriptbefehl auf den Bildschirm aus. Wenn man das nicht wünscht, muss man als ersten Befehl *@echo off* in den Skript schreiben.

### pause

...hält einen Skript an und fordert zu einer Tasteneingabe auf. Kein Argument.

### %1, %2, %3 ...

...sind Variablen, die man im Skript verwenden kann. Initialisiert werden die Variablen auf der Kommandozeile. Lautet unser Skript *copy %1 A:\* und nennen wir ihn *copytodisk.bat*, dann führt der Aufruf von "*copytodisk myprog.bas*" dazu, dass im Skript "*myprog.bas*" als %1 verwendet und daher nach A: kopiert wird. %1 ist immer der erste übergebene Parameter, %2 der zweite usw.

### goto

...Funktioniert wie bei BASIC. Fast. In Skripten gibt es keine Zeilennummern. Sondern man benutzt s.g. LABELS. Das ist einfach ein Text, der eine bestimmte Stelle im Programm markiert. In DOS-Skripten werden die Labels mit einem Doppelpunkt eingeleitet:

```
echo Das ist ein Skript
echo Dieser Teil wird ausgeführt
goto ende
echo Dieser Teil wird nie ausgeführt
:ende
echo Hier ist das Ende und das wird
echo ausgeführt
```

### If exist

Der If-Befehl in DOS ist sehr spezialisiert. Die erste Variante lautet: *If [not] exist Befehl* Das Not in Klammer bedeutet: Man kann vor das "exist" noch ein "not" setzen. Das Ganze funktioniert ohne NOT so: Wenn DOS findet, wird der Befehl ausgeführt, ansonsten nicht. Mit NOT: Wenn DOS die Datei NICHT findet, wird der Befehl ausgeführt. Beispiel: *if exist %1 copy %1 A:* kopiert das erste Argument nach A:, falls es existiert. Ein bisschen komplizierter:

```
if exist %1 goto docopy
echo Datei existiert nicht!
goto ende
:docopy
copy %1 A:
:ende
```

### If string1==string2

Hier werden zwei Strings verglichen. Wichtig, um Optionen zu verarbeiten. Beispiel:

```
copy %1.bas A:
if %2==1 copy %1.dat A:
```

Der Skript heisse *backup.bat*. Hier kann ich nun auf der Kommandozeile angeben: "*backup myprog 0*". Dann wird nur *myprog.bas* nach A. kopiert. Oder ich gebe "*backup myprog 1*" an. Dann wird auch *myprog.dat* nach A: kopiert.

### if errorlevel

Mit dieser Variante kann man mit (selbstgeschriebenen) Programmen kommunizieren. Wir müssen uns das so vorstellen, dass unser BASIC-Programm mit einem END-Befehl endet. Der END-Befehl erhält jedoch ein Argument, eine einfache Zahl. Also z.B. *END(1)*. Diese Zahl wird nach Ende des Programms DOS übergeben und stellt den ERRORLEVEL da. Der Skript kann unmittelbar nach Aufruf des Programms mit *IF ERRORLEVEL* darauf reagieren. Er folgt dem IF, wenn der zurückgegebene Code des Programms grösser oder gleich dem angegebenen Wert ist. Ein Beispiel:

```
test
if errorlevel 1 goto docopy
```

```
goto ende
:dopcopy
copy %1 A:
:ende
```

Zuerst wird das Programm "test" ausgeführt. Dieses gibt einen Code zurück. Ist dieser Code  $\geq 1$ , dann wird kopiert. Ansonsten wird der docopy-Teil übersprungen.

### **For-Schleifen**

Sie haben wir schon bei den Wildcards kennengelernt. In Skripten können wir sie natürlich auch verwenden. Dabei gibt's zwei Besonderheiten zu beachten. Erstens muss beim Platzhalter ein Doppelprozent stehen, also *for %%1 in () do %%1*. Ansonsten interpretiert der Skript %1 als Übergabeparameter und meldet einen Syntaxfehler. Das %% veranlasst DOS, ein echtes Prozentzeichen zu lesen.

Die zweite Besonderheit ist, dass man dem Platzhalter nicht nur Dateien, sondern auch einfache Zahlen übergeben darf. Man kann also eine Schleife einfach fünfmal durchlaufen, indem man schreibt: *for /l %%i in (1,1,5) do ....* In der Klammer steht (Anfang, Schritt, Ende). Für diese letzte Option müssen Sie allerdings DOS mit dem Befehl "cmd /x" starten und es funktioniert nur auf WinNT/2000/XP.

---

# QBASIC: Vorwort

---

Nun geht's endlich weiter mit dem eigentlichen Programmieren!

## C16-BASIC und QBASIC

Wir werden in den folgenden Kapiteln nach dem C16-BASIC einen weiteren BASIC-Interpreter kennenlernen, QBASIC. Die wesentlichsten Unterschiede zwischen beiden:

- QBASIC ist wesentlich umfangreicher und komfortabler. Es ist sogar in weiten Teilen identisch mit der BASIC- Sprache der heute modernen Windows-Programmiersysteme Visual Basic, Visual Basic Script und Visual Basic for Applications.
- Mit dem C16-BASIC haben Sie einen ziemlich langsamen, simulierten Computer (einen s.g. Emulator) gesteuert, mit QBASIC können Sie Ihren PC direkt ansteuern, d.h. direkt auf Ihre Festplatte, Ihren Hauptspeicher usw. zugreifen. (Na ja, auch nicht so ganz direkt, aber das wird uns noch nicht stören...). Ergebnis: QBASIC ist in etwa so schnell wie Ihr PC. Also vermutlich ziemlich schnell!
- Mit QBASIC können Sie auf DOS zurückgreifen und dessen Funktionen nutzen.

## Mal wieder kleine Historie

- 1981 wurde der erste IBM PC herausgebracht. Er hatte das BASIC fest eingebaut wie der C16. Das BASIC war ähnlich, es hatte aber keine Grafikbefehle wie das C16-BASIC
- Bald brachte die Firma Microsoft, die das Betriebssystem DOS für den IBM PC entwickelte, auch einen unabhängigen BASIC-Interpreter heraus. Das war einfach ein DOS-Binary, das man wie einen DOS-Befehl von einer Diskette oder der Festplatte starten konnte. Dieses BASIC nannte sich GW-BASIC. Es wurde bis 1988 immer weiter entwickelt, bekam auch einige Befehle für Grafik und Sound und umfasste schliesslich über 60K. (Vergleich: Das C16-BASIC hat 32K).
- Schon bald (mit Erscheinen des IBM PC XT 1983) hatten die PCs kein fest eingebautes BASIC mehr. Dafür wurde das GW-BASIC (bei IBM-PCs BASICA genannt) im Rahmen von DOS auf Diskette mitgeliefert, also quasi "kostenlos".
- Um 1986 herum gab Microsoft ein neues BASIC heraus, QUICKBASIC. Dies war nicht nur ein Interpreter, sondern auch ein Compiler, es hatte einige neue Programmierkonzepte aus der *strukturierten Programmierung* , es war viel umfangreicher als GWBASIC und es kostete viel Geld. (Ca. 500 DM).
- Ab 1990 und DOS 5.0 wurde QBASIC statt GW-BASIC in DOS mitgeliefert. QBASIC war eine verkleinerte Version von QUICKBASIC, nämlich nur der Interpreter. Aber sprachlich war QBASIC mit QUICKBASIC praktisch identisch.
- Zwischen 1990 und 1995 waren die beiden Dateien qbasic.exe und qbasic.hlp quasi als DOS-Befehle im DOS-Verzeichnis immer dabei. Weiterentwickelt wurde QBASIC und QUICKBASIC jedoch nicht mehr, da inzwischen Microsoft Windows Rechnung trug. Das BASIC für Windows heisst Visual-BASIC. Mit dem Aufkommen dieser und anderer Programmiersprachen wurde daher QBASIC immer weniger benutzt. Bei Windows95 wurde QBASIC nicht mehr mitinstalliert, befand sich aber noch auf der Installations-CD. So auch noch bei späteren Versionen, z.B. Windows 98. Ob sie bei heutigen Windows-Versionen



immer noch irgendwo auf der CD sind, weiss ich nicht.

- Seit ein paar Jahren hat Microsoft QBASIC und sogar auch QUICKBASIC als Freeware freigegeben und man kann es sich im kostenlos im Internet herunterladen.
- QBASIC ist noch nicht völlig vergessen: Als Einstiegssystem erfreut es sich vielerorts an Schulen noch grosser Beliebtheit

## Neue Perspektiven von QBASIC

In den Zeiten, als DOS das aktuelle Betriebssystem war, war BASIC lediglich eine Anfängersprache und für viele Zwecke, insbesondere für die Spieleprogrammierung kaum zu gebrauchen. Grund: Es war viel zu langsam. Die meisten benutzten aus Kostengründen einen BASIC-Interpreter und interpretierte Programme sind ca. 5 bis 10 mal langsamer als compilierte Programme. Aber selbst QUICKBASIC-Kompilate sind nicht besonders schnell.

Damals wie heute brauchte man für Spiele die höchste Rechnerleistung und einen Rechner voll ausreizen kann man nur, wenn man ihn in Assembler programmiert. Assembler ist allerdings eine schreckliche "Programmiersprache" und daher war es sehr schwierig, wenn man nur BASIC gelernt hatte, ein gutes Spiel zu programmieren.

Die Zeiten haben sich quasi stillschweigend geändert. Schon 1998 war ein durchschnittlicher neuer PC so schnell, dass ein QBASIC-Programm in etwa so schnell lief wie ein in Assembler geschriebenes Programm auf einem leistungsstarken PC von 1988. Und erst heute! Sie können also heute wunderbar Action-Shooter auf QBASIC schreiben! Das glauben Sie nicht? Nun, dann starten wir doch gleich die ersten Schritte!

## Herunterladen und Installieren von QBASIC

- Gehen Sie als erstes auf die grosse [QBASIC- Seite](#). Wie Sie sehen, "lebt" dort eine grosse und aktive Gemeinde! Hier gibt es Tonnen von BASIC-Programmen, Utilities, Einführungen und Nachschlagewerke, so dass man meinen könnte, alle Programme dieser Welt entstünden nur in QBASIC...
- Gehen Sie dort auf der linken Seite nach "Download", dann klicken Sie im rechten Fenster auf "Compiler". (Das ist etwas irreführend, da wir einen Interpreter wollen).
- Nun laden Sie sich QBasic 1.1, Deutsche Vollversion (310 K), herunter.
- Gehen Sie nun eine Seite zurück und dann auf "Spiele".
- Laden Sie hier "Moon35.zip" herunter.
- Erstellen Sie sich ein Verzeichnis "QBASIC" auf Ihrer Festplatte.
- Extrahieren Sie den Inhalt von QB\_1\_1.zip und Moon35.zip in dieses Verzeichnis.
- Machen Sie ein DOS-Fenster auf, wechseln Sie in das QBASIC-Verzeichnis und geben Sie dann das Kommando "qbasic moon35.bas" ein.
- Drücken Sie F5
- Nun haben Sie ein richtig schönes "Ballerspiel" vor sich! Viel Spass dabei! (Und nebenbei haben Sie sich den QBASIC-Interpreter installiert!)

---

# QBASIC: Erste Schritte

---

## Die Entwicklungsumgebung

Wenn Sie genug haben vom Ballern und das Programm wieder beenden, landen Sie wieder in der QBASIC-Entwicklungsumgebung. Was ist eine "Entwicklungsumgebung"? Das ist ein Programm, mit dem man Programme entwickeln kann. Wir haben ja schon gelernt, dass man dazu mindestens einen Editor und einen Interpreter, besser aber noch dazu einen Debugger braucht. Mit dem Editor schreibt man den Quelltext, mit dem Interpreter führt man sie aus und mit dem Debugger testet man ihn auf Fehler. Wenn man nun Interpreter, vielleicht Compiler, Editor und Debugger in *einem* Programm hat, spricht man von einer *Entwicklungsumgebung*. Sowas ist natürlich praktisch und QBASIC ist sowas.

Nach Beendigung des Programms sehen Sie, dass das QBASIC-Fenster zweigeteilt ist und ansonsten fast genauso aussieht wie das EDIT-Fenster, das Sie schon kennen. Wenn Sie im oberen Fenster den Cursor mit der Maus plazieren, können Sie dort ihn auch genauso bewegen, wie in EDIT. Die Bedienung ist identisch und muss hier nicht mehr erklärt werden.

## Zeilennummern, Edit-Fenster und Direkt-Fenster

Schauen Sie sich mal den Code des Moon35-Programms an. Sie werden einiges Bekanntes dort erkennen: DATA-Befehle em masse, DIM-Befehle, IF's, GOSUB's usw. Trotzdem irritiert das Gesamtbild etwas: Es fehlen die Zeilennummern!

Das ist eine ganz wesentlicher Fortschritt von QBASIC gegenüber den Vorgänger-BASIC's: Man braucht keine Zeilennummern mehr. Aus Kompatibilitätsgründen kann man sie noch verwenden, aber es ist nicht ratsam. Überhaupt nicht. Wir haben ja inzwischen EDIT und den überaus grossen Nutzen von Copy und Paste kennengelernt. Und wir wissen, dass QBASICs Editor mit EDIT identisch ist. (Es ist sogar faktisch derselbe.) Wir können daher Copy und Paste auch für die Erstellung unserer Programme nutzen. Das macht aber nur dann einigermassen Sinn, wenn wir nach dem Einfügen von 20 Zeilen Code nicht händisch im ganzen Programm die Zeilennummern ändern müssen! Oder?

Das Wegfallen der Zeilennummern bringt aber zwei Probleme mit sich:

- Wie unterscheidet QBASIC zwischen Direkt-Kommandos und Programmzeilen?
- Wie adressiere ich Sprünge?

Für Direkt-Kommandos ist der untere Fensterteil zuständig. Mit F6 wechsele ich zwischen Edit-Fenster und Direkt-Fenster. Im Direkt-Fenster kann ich ?1+1 eingeben und bekomme sofort mein Ergebnis.

Die zweite Frage beantworten wir gleich, doch jetzt erstmal ein erstes Testprogramm. Gehen Sie im Menü auf Datei/Neu und schreiben Sie ein kleines Programm, das Primzahlen ausrechnet. (Wie am Anfang des C16-Teils). Verwenden Sie das BASIC, das Sie vom C16 bereits kennen: LET, FOR, usw.

Das Programm starten Sie mit F5. Die Syntax wird schon während der Eingabe geprüft.

## LET kann man weglassen

Ach ja, LET: Es war schon beim C16, dass Sie diesen Befehl auch weglassen können. Statt

```
LET A=1  
PRINT A
```

können Sie einfach

```
A=1  
PRINT A
```

schreiben. BASIC weiss von selbst, dass es sich hierbei um eine Zuweisung handeln muss.

## Online-Help

Vielleicht lief alles nicht ganz so glatt und der ein oder andere Befehl hat doch eine geringfügig andere Syntax. Was tun? Wenn Sie ganz rechts im Menü Hilfe/Index aufrufen, erhalten Sie zu jedem Befehl in QBASIC eine exakte Beschreibung. Und meistens sogar noch ein Beispiel. Navigieren müssen Sie innerhalb der Hilfe mit der Maus oder mit der **TAB-Tase**.

Über Hilfe/Inhalt gelangen Sie in das Inhaltsverzeichnis der Hilfe. Dort erhalten Sie

- im linken oberen Kasten eine Beschreibung der Bedienung der Entwicklungsumgebung.
- im rechten oberen Kasten eine Referenz der Tastenkombinationen, die Sie als Ersatz oder zusätzlich zu den Menübefehlen verwenden können.
- im linken unteren Kasten einige technische Hinweise (nicht so wichtig)
- im rechten unteren Kasten alle Fehlermeldungen und ihre Codes.

Schon allein, weil diese Hilfe existiert, soll im Rahmen dieses Tutorials nicht jeder Befehl von QBASIC besprochen werden. Wichtig sind die Programmierkonzepte. In die einzelnen Befehle zur Grafikprogrammierung u.ä. können Sie sich dann selbst einarbeiten.

---

# QBASIC/Visual Basic: Unterschiede zum C16-BASIC

---

In diesem Kapitel wollen wir uns mit einer Auswahl von Eigenschaften von QBASIC/Visual Basic beschäftigen, zum Teil Sprachelemente, die gegenüber dem C16-BASIC gewisse Erweiterungen enthalten, zum Teil Befehle zur Ein- und Ausgabe, die es Ihnen erleichtern sollen, schon mal mit QBASIC loszulegen.

Die Elemente von QBASIC, die ganz neue Konzepte einführen, wie Funktionen, Datentypen und Dateioperationen werden wir nicht in diesem, sondern in den nächsten Kapiteln behandeln.

## Sprünge

Da man in QBASIC/Visual Basic auf Zeilennummern verzichten kann, muss man offensichtlich etwas anderes dafür vorgesehen haben. So ist es auch. Man springt nicht mehr zu Zeilennummern, sondern zu *Sprungmarken*. Das funktioniert genau gleich wie in DOS-Skripten, nur dass die BASIC-Sprungmarken den Doppelpunkt nicht vorne, sondern hinten haben.

```
CLS
PRINT "Taschenrechner"
PRINT "-----"
start:
INPUT "Geben Sie zwei Zahlen ein: ", a, b
PRINT "Summe: "; a + b
PRINT "Differenz: "; a - b
PRINT "Produkt: "; a * b
PRINT "Verhältnis: "; a / b
IF (a <> -1) GOTO start
```

Die Sprungmarken sind ein Riesenvorteil, da "GOTO start" wesentlich aussagekräftiger ist, als "GOTO 1023".

Was wir hier auch gleich noch sehen: Der Befehl zum Bildschirmlöschen heisst hier "CLS", nicht "SCRCLR".

## IF-Anweisung

Im einfachen Fall sieht eine IF-Anweisung so aus:

```
IF THEN
```

Was aber, wenn wir im Fall, dass die Bedingung wahr ist, eine ganze Menge machen wollten? Natürlich können wir die Befehle alle mit Doppelpunkten hinter das THEN packen, aber so richtig toll war das auch nicht:

```
IF antwort$="j" THEN INPUT "Gib noch was ein: ";b$:FOR i=1 TO len(b$):IF
mid$(b$,i,1)="?" THEN b$=left$(b$,i-1)+right$(b$,len(b$)-i-1):NEXT I:
PRINT b$:GOTO 1234
....
```

(Diese Zeile würde ausserdem einen Syntaxfehler verursachen, weil man innerhalb einer

Doppelpunktsequenz nicht nocheinmal einen IF-Befehl bringen kann.) Die andere Möglichkeit ist, sich mit vielen GOTO's zu helfen, die die Übersicht aber auch nicht gerade erleichtern:

```
IF THEN GOTO marke1
IF THEN GOTO marke2
...
GOTO marke3
marke2:
...
GOTO marke3
marke1:
...
marke3:
...
```

Alles klar?

Der IF-Befehl in QBASIC/Visual Basic hat schon den vollen Umfang des If's der *strukturierten Programmierung*:

```
IF THEN
...
ELSEIF
...
ELSE
...
END IF
```

An Stelle von "...", "..." usw. können eine oder mehrere Programmzeilen stehen. Inklusive Schleifen und GOSUBs. So einen ganzen Programmabschnitt, der da steht, wo "normalerweise" nur ein einziger Befehl steht, nennt man einen *Block*. Das ist was sehr Praktisches.

Das Ganze funktioniert also so: Falls Bedingung1 zutrifft, dann führe den Block aus.

Ansonsten(ELSE): Wenn (IF) Bedingung2 zutrifft, dann führe den Block aus. Ansonsten: Führe den Block aus. Und damit man weiss, wo der Block hinter dem ELSE zuende ist, muss man diesen mit einem "END IF" abschliessen.

Man muss nicht immer alle Teile dieser langen IF-Anweisung angeben. Man kann auch das ELSEIF oder ELSE weglassen. Es ginge also auch

```
IF THEN
...
END IF
```

Oder eben:

```
IF THEN
...
ELSE
...
END IF
```

Alle Klarheiten beseitigt? Machen wir noch ein einfaches Übungsbeispiel:

```
INPUT "Geben Sie ein Zahl ein: ",a
IF a=1 THEN
    PRINT "Das ist das Doppelte: ",a*2
    PRINT "Das ist das Vierfache: ",a*4
ELSEIF a=2 THEN
    PRINT "Das ist das Dreifache: ",a*3
    PRINT "Das ist das Sechsfache: ",a*6
ELSE
    PRINT "Das ist weder 1 noch 2"
END IF
```

Und noch ein etwas raffinierteres Beispiel: Hier enthält der erste Block hinter dem IF selbst nochmal ein IF mit Block:

```
CLS
PRINT "Seltsamer Rechner"
PRINT "-----"
start:
INPUT "Geben Sie eine Zahl ein: ", a
INPUT "Wollen Sie wissen, was die Haelfte ist? (j/n)", o$
IF (o$ = "j") THEN
    b = a / 2
    PRINT "Die Haelfte ist: "; b
ELSE
    PRINT "Wollen Sie wissen, was ein Viertel ist? (j/n)", o$
    IF (o$ = "j") THEN
        c = a / 4
        PRINT "Ein Viertel ist: "; c
    END IF
END IF
GOTO start
```

Wie Sie sehen, empfiehlt es sich, die Blöcke nach rechts einzurücken. Immer alle Zeilen, die zu einem Block gehören, sind gleichweit eingezogen. Dadurch kann man die Blöcke mit dem Auge leicht erkennen.

## Schleifen

Soviel Möglichkeiten zum Schleifenbau wie in QBASIC/Visual Basic kenne ich in keiner anderen Sprache. (Schon ein bisschen zuviel und daher etwas unübersichtlich.) Im C16-BASIC haben wir FOR und WHILE kennengelernt. Die gibt's - mit exakt der gleichen Syntax - auch in QBASIC.

### EXIT-Anweisung bei FOR:

Ein wichtiges Extra bei der FOR-Schleife ist allerdings die EXIT-Anweisung. Mit dieser Anweisung bricht man eine FOR-Schleife vorzeitig ab. Dadurch können wir uns auch hier ein GOTO sparen.

<

Variante ohne EXIT:

```
FOR i=1 TO 100
    INPUT "Geben Sie eine Zahl ein: ",x
    IF x=0 THEN GOTO nachschleife
    PRINT 1/x
NEXT i
nachschleife:
...
```

Variante mit EXIT:

```
FOR i=1 TO 100
    INPUT "Geben Sie eine Zahl ein: ",x
    IF x=0 THEN EXIT FOR
    PRINT 1/x
NEXT i
...
```

## WHILE

Aus unerfindlichen Gründen gibt's zwar das EXIT bei FOR-Schleifen, nicht aber bei den häufig

gebrauchten WHILE-Schleifen. Das liegt wohl daran, dass die einfache WHILE-Schleife lediglich aus Kompatibilitätsgründen aufgenommen wurde. Ihr eigentlicher "grosser" Nachfolger ist die DO WHILE-LOOP-Schleife.

## LOOP

Für LOOP-Schleifen gibt es zwei Möglichkeiten. Ich will die eine mal die WHILE-Möglichkeit und die andere die DO-Möglichkeit nennen:

- WHILE-Möglichkeit: Erst wird geprüft und dann ausgeführt.
- DO-Möglichkeit: Erst wird ausgeführt und dann geprüft.

Das sieht dann so aus:

WHILE-Möglichkeit:

```
DO WHILE
...
LOOP
```

DO-Möglichkeit:

```
DO
...
LOOP UNTIL
```

Es gibt sogar noch zwei andere Möglichkeiten, aber die erwähne ich hier nicht, weil sie erstens programmiertechnisch keinen Vorteil bieten, also nur quasi ein Duplikat der anderen beiden Möglichkeiten darstellen (man nennt sowas auch: "Redundanz": Die beiden anderen Möglichkeiten sind redundant). Und weil sie zweitens kein Äquivalent in den Profisprachen der Pascal- und C-Familien haben. Hier gibt es jeweils nur drei Schleifentypen: FOR, WHILE und REPEAT UNTIL (bzw. DO WHILE).

Die DO WHILE-LOOP-Möglichkeit ist also zur einfachen WHILE-Schleife identisch, bis auf die Möglichkeit, bei der WHILE-LOOP-Schleife mit einem EXIT DO abzubrechen.

Bei der DO-LOOP UNTIL-Möglichkeit wird zuerst der Anweisungsblock ausgeführt und dann geprüft. Ist die Bedingung wahr, dann wird die Schleife *abgebrochen*, hier also ein weiterer Unterschied zur WHILE-Schleife, wo eine wahre Bedingung zur *Fortsetzung* der Schleife führt.

Anwendungsbeispiel:

```
'Suche-Funktion

INPUT "Geben Sie einen String ein: ", a$
INPUT "Geben Sie ein Wort ein, das im String gesucht werden soll:", such$

alen = LEN(a$)
slen = LEN(such$)
i = 1
found = 0

DO WHILE (NOT (found = 1)) AND (i <= alen - slen + 1)
    found = 1
    FOR j = 1 TO slen
        a1$ = MID$(a$, i + j - 1, 1)
        s1$ = MID$(such$, j, 1)
        IF UCASE$(a1$) <> UCASE$(s1$) THEN found=0
    NEXT j
```

```

    IF found = 0 THEN i = i + 1
LOOP

IF (found = 1) THEN
    PRINT "Wort an Stelle "; i; " gefunden"
ELSE
    PRINT "Wort wurde nicht gefunden"
END IF

```

Wir hätten dieses Beispiel auch mit einer einfachen WHILE-Schleife formulieren können, da wir hier nirgends die Schleife abbrechen müssen.

## Übung:

1. Wie müsste die Suchfunktion aussehen, wenn man die äussere Schleife als FOR-Schleife formuliert?
2. Wie müsste die Suchfunktion aussehen, wenn man die äussere Schleife als DO-LOOP UNTIL-Schleife formuliert?

Abgesehen von den Schleifen haben sich beim obigen Beispiel zwei weitere Neuerungen eingeschlichen: Das Hochkomma in der ersten Zeile und die Funktion UCASE\$(). Das Hochkomma ist eine Alternative zu REM: Bei QBASIC/ Visual Basic empfiehlt sich, nur noch das Hochkomma als Kommentar zu benutzen, da so der Kommentar viel leichter vom eigentlichen Programmtext zu unterscheiden ist.

UCASE heisst "UpperCASE" und wandelt einen String in Grossbuchstaben um. Das Gegenstück dazu ist LCASE\$(), welches - unschwer zu erraten - einen String in Kleinbuchstaben umwandelt. Durch die Umwandlung in Grossbuchstaben ist unsere Suchfunktion unabhängig von Gross- und Kleinschreibung. Jetzt wissen Sie, wie die Suchfunktion von Textverarbeitungsprogrammen programmiert ist! Nicht schwer, oder?

## SELECT CASE

Ganz nett, in allen "grossen" Programmiersprachen vorhanden, aber nicht unbedingt nötig ist der select-Befehl, ein Spezialfall von IF, gedacht für den Fall, dass nach den einzelnen Werten einer Ganzzahlvariablen verzweigt werden muss. Zunächst der Paredefall ohne SELECT:

```

ende=0
WHILE (ende=0)
    CLS
    PRINT "Ihr Spiele-Menü:"
    PRINT "1. Vier gewinnt"
    PRINT "2. 17 und 4"
    PRINT "3. Formel 1"
    PRINT "4. Superhirn"
    PRINT "5. Millionärsshow"
    PRINT "6. Ende"
    PRINT
    COLOR 14,0
    INPUT "Ihre Wahl: ",a
    IF      a=1 THEN GOSUB viergewinnt
    ELSEIF  a=2 THEN GOSUB 17und4
    ELSEIF  a=3 THEN GOSUB formell1
    ELSEIF  a=4 THEN GOSUB superhirn
    ELSEIF  a=5 THEN GOSUB millionär
    ELSEIF  a=6 THEN ende=1
WEND

```

Den letzten Teil ab INPUT... kann man mit SELECT-Befehl so schreiben:



```

INPUT "Ihre Wahl: ",a
SELECT CASE a
CASE 1
  GOSUB viergewinnt
CASE 2
  GOSUB 17und4
CASE 3
  GOSUB formell
CASE 4
  GOSUB superhirn
CASE 5
  GOSUB millionär
CASE 6
  ende=1
END SELECT

```

Hier bringt das nicht wirklich Vorteile gegenüber IF. Wenn man allerdings sehr viele mögliche Werte berücksichtigen muss, die sich nur auf wenige Entscheidungswege verteilen, dann kann der SELECT-Befehl einiges an Übersicht bringen. Ein

```

SELECT CASE a
CASE 1,4,6,10,18
  b=1
CASE 2,3,12,13,17
  b=2
CASE 5,7,8,9,11
  b=3
CASE ELSE
  b=4
END SELECT

```

ist doch erheblich übersichtlicher als ein

```

IF a=1 OR a=4 OR a=6 OR a=10 OR a=18 THEN b=1
ELSE IF a=2 OR a=3....

```

Damit ist die Syntax von SELECT eigentlich auch schon erklärt:

```

SELECT CASE Variable
CASE Werteliste1
  ...
CASE Werteliste2
  ...
CASE ELSE
  ...
END CASE

```

...heisst: Wenn "Variable" einen Wert der Werteliste1 annimmt, (Werte durch Komma getrennt), dann führe den Anweisungsblock hinter dem entsprechenden CASE aus. Entsprechend die anderen CASE-Teile. Es kann, muss aber kein CASE ELSE folgen. Dessen Anweisungsblock wird durchlaufen, falls kein Wert einer der Wertelisten "gepasst" hat.

## Texteingabe und -ausgabe

Auch hier hat QBASIC anderes oder mehr zu bieten, als das C16-BASIC.

### CLS: Bildschirm löschen

Syntax: CLS

### LOCATE: Cursor auf dem Bildschirm positionieren

Syntax: LOCATE y,x (Kurzversion. Langversion: Siehe Hilfe).

y=Zeile (1..25), x=Spalte (1..80)

### COLOR: Bildschirmfarben setzen

Syntax: COLOR vordergr,hintergr,rahmen.

vordergr,hintergr und rahmen sind Zahlen, die die Farbnummer angeben. Da Sie vermutlich in einem DOS-Fenster unter Windows arbeiten, bleibt "Rahmen" wirkungslos. Weiter unten ein kleines Beispielsprogramm zum Ausprobieren der Farben.

Beispiel zu COLOR:

```
CLS
FOR i = 0 TO 15
  COLOR i, 0
  PRINT "hhhhhhhhhhhhhhhhhhhh"
  REM INPUT a$
NEXT i
CLS
FOR i = 0 TO 15
  COLOR 15, i
  PRINT "hhhhhhhhhhhhhhhhhhhh"
  REM INPUT a$
NEXT i
FOR i = 0 TO 15
  COLOR 15, i
  CLS
  INPUT a$
NEXT i
END
```

## Formatierte Ausgabe mit PRINT USING:

Nehmen wir an, Sie wollen ein Programm zur Erstellung von Rechnungen oder Bilanzen erstellen - für einen Computer sicher ein nicht allzu weit entferntes Anwendungsgebiet. Dieses Programm wird ziemlich viele Zahlen ausspucken. Und es wird nicht so gut aussehen, wenn diese in der Form

```
12398.463723
183262.12
-28322.34762E00
usw.
```

Für eine Rechnung wünschen wir uns die Zahlen rechtsbündig und auf exakt zwei Nachkommastellen gerundet.

Zunächstmal ist das eine sehr gute

**Übung:** Schreiben Sie ein Programm, dass die im Array zahlen() gespeicherten Beträge rechtsbündig und auf zwei Nachkommastellen gerundet auf dem Bildschirm ausgibt!

Wenn Sie nicht weitergekommen sein sollten, hier ein Hinweis auf den Lösungsweg:

1. Vergessen Sie nicht, die Aufgabe in kleine Teilaufgaben zu unterteilen, die Sie in Unterprogrammen lösen.
2. Jede Zahl muss zuerstmal wie gefragt gerundet werden (erste Teilaufgabe).
3. Dann muss sie in einen String umgewandelt werden. (Zweite Teilaufgabe).
4. Dann muss dieser String auf eine feste Zahl von Zeichen verlängert werden. Und zwar von links her. (Dritte Teilaufgabe).

Und schon haben Sie's.

Es ist also nicht so, dass wir das Problem mit C16-BASIC nicht hätten lösen können. Aber so nett das Austüfteln solcher kleinen Lösungen ist - wenn man nur noch damit beschäftigt ist und nicht mehr mit der eigentlich zu lösenden Aufgabe, macht's auch keinen Spass mehr und man kommt

nicht mehr vorwärts.

Daher gilt ganz allgemein: Je umfangreicher der Satz an mitgelieferten Routinen (Unterprogrammen, Befehlen, Ressourcen - darunter können Sie vorläufig dasselbe verstehen) eines Entwicklungssystems ist, desto

1. länger brauchen Sie, bis Sie dieses zu 100% nutzen können.
2. desto produktiver sind Sie dann aber auch im Endeffekt.

Ein gutes Beispiel einer solchen mitgelieferten Ressource ist der PRINT USING- Befehl. Er benutzt einen Extra-String, mit dem angegeben werden kann, in welchem Format die folgenden Zahlen oder Strings ausgegeben werden sollen, d.h. mit wieviel Stellen vor und nach dem Komma, rechts- oder linksbündig die Zahl angezeigt werden soll. Deshalb heisst dieser Extrastring - nicht nur in QBASIC, auch in der wichtigen Sprache C gibt es so etwas - *Formatstring*.

Der String gibt zunächst die Länge des Felds an, in dem die Zahl ausgegeben werden soll. "#####" z.B. gibt jede Zahl in einem fünf Zeichen langen Feld aus und zwar rechtsbündig. (Eine Option für linksbündige Ausgabe gibt es nicht.)

```
FOR i=80 TO 89
  PRINT USING "#####"; i
NEXT i
```

ergibt dann die Ausgabe

```
80
81
82
83
84
85
86
87
88
89
```

Jedes #-Zeichen steht also für eine Ziffer oder - falls die Zahl nicht so lang ist - für ein führendes Leerzeichen. Mit einem Dezimalpunkt kann angegeben werden, auf wieviel Stellen gerundet werden soll:

"#####.##" rundet auf 2 Stellen und gibt das Ganze rechtsbündig in einem 10+1+2=13 Zeichen langen Feld aus.

Wenn man dem String ein Plus voranstellt, druckt es die Zahl immer mit führenden Vorzeichen:

```
FOR i=80 TO 89
  PRINT USING "+#####"; i
NEXT i
```

ergibt dann die Ausgabe

```
+80
+81
+82
+83
+84
+85
+86
+87
+88
+89
```

Wenn man es hintenanstellt ("#####+") druckt PRINT USING das Vorzeichen hinter die Zahl.

Auch für Strings kann man die Länge des Ausgabefelds definieren. Hierzu schreibt man einen String mit der Länge des Ausgabefeldes aus Leerzeichen und führendem und abschliessendem Backslash. "\ \ " z.B. sind 6 Leerzeichen plus die Backslashes=8 Zeichen langes Ausgabefeld. Ist der String länger, so werden nur die die ersten acht Zeichen davon ausgegeben, allgemein: Nur soviel Zeichen, wie der Formatstring lang ist.

**Ganze Syntax:** PRINT USING <Formatstring>Wert1;Wert2;;Wert3;....

Erst kommt also das Schlüsselwort PRINT USING, dann der Formatstring, dann ein Semikolon und dann durch Semikolons getrennt alle Werte, die in diesem Format ausgegeben werden sollen. Dann hinter einem weiteren Semikolon ev. ein neuer Formatstring und dahinter die Werte, die mit dieser neuen Formatangabe gedruckt werden sollen usw. Ganz am Ende gilt die Regelung des normalen Print-Befehls: Ein Semikolon setzt die nächste Ausgabeposition unmittelbar ans Ende dieser Ausgabe. Beim Komma geht's das nächste Mal am nächsten Tabulator weiter und wenn weder Komma noch Semikolon kommt, geht's in der nächsten Zeile weiter.

Im Formatstring von PRINT USING gibt's noch weitere Optionen, die aber nicht so interessant sind, zumindest nicht für Nicht-Bewohner der USA, da diese eher USA-spezifisch sind (Ausgabe des Dollarzeichens, zifferntrennenden Kommata etc.). Wenn Sie neugierig sind: Schauen Sie in der Online-Hilfe von QBASIC die komplette Syntax nach!

---

# Hexadezimalzahlen

---

Im nächsten Kapitel werden wir es mit Hexadezimalzahlen zu tun haben, daher hier eine kleine Erklärung dazu. "Hex" heisst sechs, "dezi" zehn und "Hexadezi" damit sechszehn.

Hexadezimalzahlen sind Zahlen im 16er-System. Die Ziffern gehen nicht wie im Dezimalsystem von 0 bis 9, sondern von 0 bis - ja bis etwas, das im Dezimalsystem 15 heissen würde. Aber es soll ja nur eine Ziffer sein. Man benutzt einfach die Buchstaben A bis F. Die Ziffern sind also 0,1,2,3,4,5,6,7,8, 9,A,B,C,D,E,F. 16 Ziffern. Damit man Dezimal- und Hexadezimalzahlen unterscheiden kann, schreibt man hinter die Hex-Zahlen ein kleines h:  $16 = 10h$ . Wir sehen schon, wo der Vorteil liegt: Die Zahlen, die ein Byte darstellen kann, gehen von 0 bis FFh. Das ist, wie wenn im Dezimalsystem ein Speicher Zahlen von 0 bis 99 speichern würde.

Das Rechnen im Hex-System ist natürlich nicht ganz einfach: Was ist AFh + 3Ch ? Aber wenn wir bestimmte Bits eines Bytes oder eines Byte-Worts setzen wollen, ist das ganz praktisch: 0Fh, das sind die niedrigen 4 Bits, F0h, das sind die hohen vier Bits eines Bytes. Der Darstellungsraum eines 16-Bit-Worts geht von 0 bis FFFFh. Praktisch, nicht?

In QBASIC kann man Hex-Zahlen direkt angeben, ohne sie ins Dezimalsystem umrechnen zu müssen. Allerdings nicht dadurch, dass ein h hinter die Zahl gestellt wird, sondern es muss ein "&h" vor die Zahl gestellt werden.

```
? &hF1D3+&h7EA2
```

..ist also kein Problem. Nur dass QBASIC das Ergebnis leider dezimal ausgibt. Aber auch das kann man ändern:

```
? HEX$( &hF1D3+&h7EA2 )
```

..gibt das Ergebnis hexadezimal zurück. Spätestens hier fällt uns allerdings auf, dass das Ergebnis falsch zu sein scheint. Es erscheint 7075h. Das ist aber kleiner als der erste Summand. Wieso? Nun, die Fähigkeit von QBASIC zum Umgang mit Hex-Zahlen beschränkt sich leider auf 16-Bit-Zahlen. Kommt es bei der Addition zu einem Summanden grösser als FFFFh, so wird ein Überlauf erzeugt, d.h. es wird so getan, als ob  $10000h = 0h$  sei und das Ergebnis entspricht dann also dem Rest. In Wirklichkeit ist unser Rechenergebnis also 17075h.

**Übung:** Schreiben Sie in QBASIC einen Hex-Rechner für 64-Bit-Zahlen!

---

## Grafik und einfache Sprites mit QBASIC

---

Dieses Kapitel können Sie überspringen, falls Sie nur die Programmierprinzipien lernen wollen und zu den weiteren Teilen dieser Einführung streben. Falls Sie jedoch schon ganz eifrig mit QBASIC programmieren und bedauern, bisher keine Zeichnungen, Kurven, grafischen Spiele oder ähnliches erstellen zu können, dann bekommen Sie hier eine kleine Einführung in die Programmierung der "hochauflösenden" Grafik unter DOS.

Es wird Ihnen allerdings auch bei grundsätzlich gedämpften Interesse an Grafikprogrammierung empfohlen, mal hier reinzuschneppern, da im Kapitel "Sprites" einige Beispielprogramme kommen, an denen Sie den "neuen" Programmierstil und die Möglichkeiten sehen können, die Ihnen eine modernere Programmiersprache als das C16-BASIC bietet.

Zur Orientierung in Bezug auf "Grafikprogrammierung auf PCs":

- Grafikprogrammierung unter DOS, besonders unter QBASIC ist technisch einfach, so dass man schnell sich mit der Programmieraufgabe beschäftigen kann und wenig um das technische Drumherum kümmern muss.
- Grafikprogrammierung unter modernen Betriebssystemen wie Linux und Windows ist prinzipiell viel komplizierter, da man dies nur im Rahmen der fensterorientierten Programm- und Speicherstrukturen und Routinen des jeweiligen Betriebssystems tun kann.
- In der Praxis teilt sich die Programmierung von grafischen Benutzeroberflächen wie Windows oder Linux/KDE oder Linux/Gnome in zwei Sorten auf: In die Programmierung langsamer und schneller Grafik. Langsame Grafik für praktisch stehende Zeichnungen ohne Animation usw. kann man mit den geeigneten Programmierwerkzeugen auch relativ einfach hinbekommen. Vorausgesetzt, man hat also z.B. die geeignete BASIC-Version und will nur ein paar Diagramme oder Zeichnungen programmieren, stimmt der zuletzt genannte Punkt nicht: Dann ist es unter Windows/Linux fast so einfach wie unter DOS.
- Ob schnelle Grafikprogrammierung unter Linux überhaupt möglich ist, weiss ich nicht. Unter Windows gibt es die s.g. DirectX-Schnittstelle. Sie zu programmieren, ist aber in jedem Fall aufwändig und heikel, so dass Spieleprogrammierung unter Windows auf jeden Fall nur was für (Quasi-)Profis ist.
- Da die QBASIC-Grafik auf einem schnellen Rechner ausreichend Geschwindigkeit für nette bewegte Grafik bietet, ist gerade die Nutzung der DOS-Grafik ein Vorzug von QBASIC, falls man ohne allzu grossen Einarbeitungsaufwand schnell mal etwas grafisch Bewegtes programmieren will. (Das muss ja kein Spiel sein. Auch die Simulation der Bewegung von Galaxien ist bewegte Grafik).
- Wir werden uns im nächsten Teil mit Visual Basic / Visual Basic for Application beschäftigen. Das, was wir bisher über QBASIC gelernt haben und das was wir in den nächsten Kapiteln darüber lernen, können wir auf diese Systeme 1:1 übertragen. Die Grafikprogrammierung dieses Kapitels ist allerdings QBASIC-spezifisch und hat keine Entsprechung bei den Windows-Varianten von BASIC. (Aus den in den vorigen Punkten genannten Gründen.)

### Bildschirmmodi

Genereller Hinweis: Sie können durch **Alt-Enter** das DOS-Fenster auf den s.g. Vollbildmodus schalten. In der Regel wird das der PC aber automatisch machen, wenn Sie unter Windows in deinem DOS-Fenster den Grafikmodus verwenden. Also nicht erschrecken. Mit **Alt-Enter** kommen Sie wieder zurück in den Fenster-Modus!

Wie beim C16 muss man den Bildschirm unter DOS für die Grafikprogrammierung erstmal vom Textmodus auf den Grafikmodus umschalten. Der Vorteil gegenüber dem C16 ist, dass der herkömmliche PRINT-Befehl auch in den Grafikmodi funktioniert.

Die verfügbaren Grafikmodi in QBASIC spiegeln die Urgeschichte der Grafikkarten wieder. Die beiden Uralt-Grafikkartenstandards CGA und EGA lassen wir hier beiseite. Der VGA-Standard von ca. 1990 hingegen hat noch eine kleine Bedeutung darin, dass auch heutige Linux- und Windows-XP-Systeme in einem VGA-Grafikmodus starten. Es seien hier nur die beiden wichtigsten Grafikmodi aufgezählt:

- 640x480 Bildschirmpunkte in 16 aus 256K Farben
- 320x200 Bildschirmpunkte in 256 aus 256K Farben (Modus 13h)

Zwischen Text- und Grafikmodi schaltet man mit dem **SCREEN**-Befehl um. Für ein grafikorientiertes Programm wird man dies nur einmal und am Anfang machen, da die Grafikmodi gegenüber den Textmodi bei der Darstellung von Text keine Nachteile bieten (sieht man von der geringeren Auflösung des 13h-Modus ab).

**SCREEN 0** Textmodus

**SCREEN 12** 640x480x16-Modus

**SCREEN 13** 320x200x256-Modus

Kleines Schmunzeln am Rande: Schauen Sie mal in der Hilfe unter Screen-Anweisung-> Bildschirmmodi nach. Ganz unten steht ein kleines Beispielprogramm und der Hinweis "Nur für Systeme mit Farbgrafikadapter". "Grafikadapter" ist der alte Name für "Grafikkarte". Und der Hinweis soll heissen: Wenn Ihr PC nur eine Textmodus- Karte und keine Grafikkarte hat, funktioniert das

Beispiel nicht...

## Einfache Grafikbefehle

Grundsätzlich: Wir werden uns hier nur mit 1-Seiten-Grafikprogrammierung beschäftigen. Was heisst "1-Seiten-Programmierung"? Können Sie sich an das Ufo-Beispiel im [Grafik-Kapitel des C16-BASIC-Teils](#) erinnern? Das hoppelte doch so komisch über den Bildschirm. Sah nicht wirklich gut aus. Das lag daran, dass wir versucht haben, mit nur einer s.g. Bildschirmseite Bewegung zu erzeugen. Das klappt nicht. Selbst, wenn wir einen sehr schnellen PC haben. Der Grund ist einfach: Das Auge schaut dem Computer beim Zeichnen der Figur zu. Auch wenn der Computer sehr schnell ist, werden wir im Mittel nur die Hälfte der Figur sehen, wenn der Computer die Figur gleich wieder löscht, sobald er sie gezeichnet hat. Den Effekt abmildern können wir also nur, wenn wir mindestens die zehnfache Zeit warten, die gebraucht wird, um die Figur zu zeichnen, bevor sie wieder gelöscht (und am nächsten Punkt wieder gezeichnet) wird. Das ist aber schlecht, weil wir auf diese Weise extrem Geschwindigkeit verlieren. Insofern ist der Shape-Befehl des C16 nur für sehr grobe Bewegungsraster zu gebrauchen, bei der die Figur sozusagen hüpf statt gleitet.

Die Lösung des Problems, eine Figur gleiten zu lassen, heisst: Mehrere Bildschirmseiten. Das heisst, wir stellen zusätzlichen Speicherplatz bereit, um den Bildschirm zweimal abzuspeichern. Das nennen wir die beiden "Bildschirmseiten". (Wenn Sie jetzt über das Thema "Grafik und Speicherplatz" stolpern, dann schauen Sie nochmal unter [C16: Hochauflösende Grafik](#) im unteren Teil des Kapitels "Grafik und Speicher" nach.) Einstweilen begnügen wir uns mit "Stehgrafiken" oder "Hüpf-Figuren" und brauchen nur 1 Bildschirmseite.

## Übersicht über die Grafikbefehle

Befehl	Syntax	Gebrauch/Bedeutung
CLS	CLS	"Clear Screen". Löscht den Bildschirm, egal in welchem Screen-Modus.
PSET	Einfach: PSET (x,y)	<p>Zeichnet einen Bildschirmpunkt an der Koordinate x,y. Beispielprogramm:</p> <pre>'Sternenhimmel SCREEN 13 FOR i = 1 TO 1000   x = RND * 320   y = RND * 200   PSET (x, y) NEXT i</pre> <p>Ausführliche Syntax: PSET STEP (x,y),Farbe</p> <p>Das Schlüsselwort "STEP" kann weggelassen werden. Wenn es dabei ist, werden x und y als Schrittweite relativ zur letzten gezeichneten PSET- Position gewertet. PSET (50,50) PSET STEP (0,20) PSET STEP (-20,0) PSET STEP (0,-20) PSET STEP (20,0) zeichnet also ein Viereck von der Ecke mit den Absolutkoordinaten (50,50) aus.</p>
LINE	LINE (x1,y1)-(x2,y2)	Zeichnet eine Linie vom Punkt (x1,y1) bis zum Punkt (x2,y2). Funktioniert wie DRAW 1,X1,Y1 TO X2,Y2 beim C16.
Rechtecke und gemusterte Linien mit LINE zeichnen	LINE (x1,y1)-(x2,y2),farbe,B LINE (x1,y1)-(x2,y2),farbe,BF LINE (x1,y1)-(x2,y2),farbe,B,musterwert	<p>LINE (x1,y1)-(x2,y2),farbe,B zeichnet Rechteck von der linken oberen Ecke (x1,y1) bis zur unteren Ecke (x2,y2).</p> <p>LINE (x1,y1)-(x2,y2),farbe,BF Rechteck ist ausgefüllt.</p> <p>Kleines Beispielprogramm</p> <pre>'Modern Art in Vierecken SCREEN 12 xscreenize = 640 yscreenize = 480 FOR i = 1 TO 100   farbe = RND * 16   x1 = (RND - .1) * xscreenize   y1 = (RND - .1) * yscreenize   xsize = RND * xscreenize / 2   ysize = RND * yscreenize / 2   x2 = x1 + xsize   y2 = y1 + ysize   LINE (x1, y1)-(x2, y2), farbe, B,&amp;hFF00   'Verzoegerungsschleife. Wert fuer 500 MHz-PIII.   'Bei schnellerem PC anpassen.   FOR j = 1 TO 10000: NEXT j   'Auskommentiert: Nach jedem Viereck auf Tastendruck warten.   'WHILE INKEY\$ = "": WEND NEXT i</pre> <p>Wie Sie sehen, füllt sich der Bildschirm mit gestrichelten farbigen Vierecken. Die Strichelungen kommt durch die Angabe des letzten Werts zustande. Die Linie wird Abschnitte mit jeweils 16 Pixel eingeteilt und die 16-Bit-Zahl sagt, welche Pixel</p>

		<p>gesetzt sein sollen: Ist ein Bit auf 1, wird das Pixel gesetzt. Im Beispiel sind also immer die ersten 8 Pixel gesetzt und die zweiten 8 Pixel nicht. Eine Angabe von &amp;HF0F0 würde die Strichlänge auf 4 Pixel verkürzen, &amp;HCCCC auf 2 Pixel und &amp;HAAAA auf ein Pixel. (Gelt, das ist jetzt ein bisserl Denksport???)</p>
<b>CIRCLE</b> (Kreise, Ellipsen)	Einfach: CIRCLE (x,y),rad,farbe	<p>Einfache Syntax: Zeichnet eine Kreis mit rad Radius um den Punkt x,y. Zum Zeichnen von Ellipsen und Kreissegmenten gibt es eine ähnlich längliche Syntax wie beim C16. Das kann man aber gut in der Online-Hilfe nachschlagen.</p>
<b>Farben, Farbpalette</b>	PALETTE USING - Befehl	<p>Wahrscheinlich haben Ihnen die 16, bzw. 256 Farben, die Sie mit dem Color-Befehl (siehe Kap. 26) hervorrufen können, noch nicht so ganz gefallen. Die Vorbelegung der 256 Farben ist eigentlich zu nichts zu gebrauchen.</p> <p>Die 16, bzw. 256 auf dem Bildschirm gleichzeitig verfügbaren Farben nennt man auch die Farbregister. (In der QBASIC-Hilfe heissen sie reichlich unverständlich "Attribute"). Welche Farbe ein Farbregister anzeigt, kann man mit dem PALETTE-Befehl definieren. Und zwar aus <math>64 \times 64 \times 64 = 256K</math> möglichen Farben. Die Auswahlfarben sind zusammengemischt aus Rot-, Grün- und Blau-Komponenten. Man nennt das das <i>RGB-Farbmodell</i>, das an vielen Orten in der Computergrafik seine Verwendung findet, auch bei den modernen Grafikkarten und Bildschirmmodi. Auch hier werden die Farben durch ihre RGB-Werte beschrieben, allerdings können hier alle Farben in der Regel gleichzeitig auf dem Bildschirm dargestellt werden, die Arbeit mit Farbregistern ist hier nicht mehr notwendig.</p> <p>Alle Farbregister zusammen heissen auch die "Palette" (daher der QBASIC-Befehl). Jede der 16, bzw. 256 Farbregister der Palette wird also durch 3 64-Bit-Zahlen beschrieben. Jeder Farbwert (R, G oder B) wird allerdings in einem extra Byte abgelegt, so dass wir es bei der Definition der Farbe mit einer 3-Byte-Zahl zu tun haben. Es wäre natürlich nun einfach, wenn der PALETTE-Befehl von QBASIC einfach PALETTE,Farbregister,R-Wert, G-Wert,B-Wert heissen würde. Das wäre viel zu einfach. Microsoft hat es an dieser Stelle richtig schön kompliziert gemacht, so dass Sie bei den ersten Versuchen so gut wie nie mehr als eine Fehlermeldung auf den Bildschirm bekommen werden..</p> <p>Am Besten, Sie versuchen das folgende Programm, das Ihnen eine Reihe von Blauwerten auf den Bildschirm malt:</p> <pre> ' Blautoene DIM farb(256) AS LONG  SCREEN 12 PALETTE 0, 0 f(0) = 0 FOR i = 1 TO 15     farb(i) = 65536 * i * 4 NEXT i PALETTE USING farb(0) GOSUB zeichne END  zeichne: CLS FOR i = 0 TO 15     LINE (1, 1 + i * 20)-(320, 19 + i * 20), i, BF NEXT i RETURN </pre> <p>Wenn alles klappt, bekommen Sie 15 dunkel- bis hellblaue Querbalken.</p> <p>Was passiert hier? Nun, als erstes kommt die Deklaration eines Arrays. Die Ergänzung "as long" kennen wir noch nicht. Sie ist aber wichtig. Systematisch werden wir uns mit solchen Ergänzungen später noch beschäftigen, einstweilen soll uns genügen, dass dies ein speziell "formatiertes" Array ist. Ohne diese Formatierung hagelt es nur Fehlermeldungen.</p> <p>In farb() werden die Farbwerte für die 16 Farbregister gespeichert. Der Palette Using-Befehl weiter unten übernimmt dann die Werte dieses Arrays in die Farbregister. Allerdings stellt sich die Frage, warum farb() dann 256 Zellen enthält und nicht nur 16. Wieder eine böse Falle. Aus welchen technischen Gründen auch immer, der PALETTE-Befehl braucht für die VGA-Modi mindestens 128 Zellen. Da SCREEN 13 ohnehin 256 Zellen braucht, sollte das Paletten-Array am Besten immer 256 Zellen enthalten.</p> <p>Der erste Palette-Befehl 0,0 sichert nur ab, dass Farbregister 0 weiterhin "Schwarz"</p>



ist, da dieses ja für Rand und Hintergrund verwendet wird. Dann geht's mit der Palettendefinitionsschleife los. Wie gesagt: Jeder Farbwert ein Byte, das Ganze eine 3-Byte-Zahl. In Hex ist das eine Zahl zwischen 0 und FFFFFFFh. Wollen wir z.B. Knallrot, dann setzen wir den Rotwert auf FFh, die Grün- und Blauwerte auf null. Der Rotwert ist das niedrigste Byte, also ist das Ganze 0000FFh. Entsprechend ist Knallgrün 00FF00h und Knallblau FF0000h. FFFF00h wäre Türkis und 00FFFFh Gelb. FF00FFh ist Magenta, FFFFFFFh Weiss. 777777h ist Grau, da hier alle Farbwerte gleich stark sind, aber unterhalb des Maximums. Genauso ist 770000h eben ein dunkles Rot.

Leider können wir aufgrund der 16-Bit-Restriktion der Hex-Verarbeitung von QBASIC das Ganze nicht einfach in Hexzahlen angeben. Also geben wir es in Form von  $030201h = 256 * 256 * 3 + 256 * 2 + 1$  an. Soweit so schön. Im Programm steht aber  $65536 * 3 + 256 * 2 + 1$ . Das ist doch das Gleiche, oder? Wieder Falle. Intern erzeugt  $256 * 256$  bei QBASIC einen Überlauf! Um QBASIC also dazu zu bewegen, einen 3-Byte-Wert zu erzeugen, muss man es ihm gleich einen 3-Byte-Wert explizit vor die Nase setzen und das ist  $65536 = 10000h$ . Und damit klappt's dann. Puh!

Weil's so schön war, im SCREEN13 noch ein Beispielprogrammchen:

```
DIM farb(256) AS LONG

SCREEN 13
PALETTE 0, 0
farb(0) = 0
FOR i = 1 TO 63
  'Magenta-Toene
  farb(i) = 65536 * i + i
NEXT i
FOR i = 64 TO 127
  'Tuerkis-Toene
  farb(i) = 65536 * (i - 64) + 256 * (i - 64)
NEXT i
FOR i = 128 TO 191
  'Grau-Toene
  farb(i) = 65536 * (i - 128) + 256 * (i - 128) + i - 128
NEXT i
PALETTE USING farb(0)
GOSUB zeichne
END

zeichne:
CLS
FOR i = 0 TO 191
  LINE (1, i)-(320, i), i, BF
NEXT i
RETURN
```

## Sprites

Das Stichwort "Sprites" war der Stromstoss für den Spieleprogrammierer vor 15 Jahren. Es handelt sich um kleine bewegte Bildchen, die über den Bildschirm gleiten oder hüpfen. Das beste Beispiel für ein "Sprite" ist der Mauscursor. Wahrscheinlich haben Sie sich noch keine Gedanken darüber gemacht, wie Sie einen Mauscursor programmieren würden, aber wenn Sie das Problem näher anschauen, wird es Ihnen nicht mehr so ganz einfach vorkommen:

- Wenn die Maus bewegt wird, muss zunächst der Mauszeiger an der alten Position gelöscht werden.
- Dann muss der Bildschirminhalt, der unter dem Mauszeiger war, wieder hergestellt werden.
- Dann muss der Bildschirminhalt, der an der neuen Position unter dem Mauszeiger sein wird, abgespeichert werden.
- Schliesslich muss der Mauszeiger an der neuen Stelle gezeichnet werden.

Wie wir gesehen haben, müsste dieser ganze Vorgang so schnell passieren, dass er kürzer dauert, als ein Bruchteil der Verweildauer des Zeigers an der neuen Position. Ansonsten kann man den Ab- und Aufbau des Zeigers mit dem Auge mitverfolgen und das Bild "hoppelt". Oder man verwendet eben einen Extra-Speicherbereich, der sehr schnell in das aktuelle Bild ein- und ausgeblendet werden kann.

In den Zeiten des C16 konnte dieses schnelle Ein- und Ausblenden eigentlich nur mit Zusatzhardware realisiert werden, den s.g. Spritegeneratoren. Der beliebteste Homecomputer der damaligen Zeit, der C64, zeichnete sich durch seine leistungsfähigen schon eingebauten Spritegeneratoren aus.

In QBASIC auf einem heutigen PC sind "C64-like" Sprites überhaupt kein Problem. Das eingangs dieses Teils bei der Installation von QBASIC empfohlene Spiel "Mooncrap" illustriert das eindrucksvoll. Aber auch ist Sprite-Programmierung kein ganz einfaches

Kapitel.

## Entwurf von Sprites

Am Besten wäre es natürlich, man könnte Sprites in einem Malprogramm zeichnen und dann in QBASIC verwenden. Das geht auch. Aber wie man gängige Pixelgrafikdateiformate aus Programmiersprachen heraus beherrscht, das gehört noch nicht zu unserem Repertoire. Also werden wir uns einstweilen darauf beschränken, die Pixel per Programm zu speichern oder zu zeichnen.

Möglichkeiten:

1. Wir zeichnen das Sprite mit den Zeichenbefehlen PSET, LINE und CIRCLE
2. Wir speichern die Farbangaben pixelweise in DATA-Zeilen ab und zeichnen diese dann auf den Bildschirm

Die erste Vorgehensweise wurde in [C16-Hochauflösende Grafik](#) im Bereich "Beispielprogramm zu SSHAPE/GSHAPE" schon demonstriert. Die zweite Möglichkeit ist auch nicht schwer. Nehmen wir ein kleines Bild mit 8 x 10 Pixeln. Jedes Pixel hat einen Farbwert zwischen 0 und 255. Jeder Farbwert ist die Nummer des Paletteneintrags. Tipp: Um Ihr erstes Sprite mittels DATA zu entwerfen, schreiben Sie sich zuerst ein kleines Progrämmchen, das Ihnen übersichtlich die Farbe der voreingestellten Paletteneinträge zeigt, falls Sie sich nicht jedes Mal die Palette selbst vordefinieren wollen.

Sie müssen die nun folgenden Programme nicht abtippen. Sie können auch einfach die Programmtexte in Ihrem Browser markieren und in einer Textdatei abspeichern, die Sie dann mit QBASIC wieder öffnen.

Meine Version:

---

```
'Zeigt Screen13-Defaultpalette an von 0 bis 100

SCREEN 13
FOR i = 0 TO 100
COLOR i
PRINT i MOD 10;
IF i MOD 10 = 0 THEN PRINT " "; i
NEXT i
```

---

Nun kanns los gehen. Meine Zeichnung sieht so aus:

---

```
'-----
'Beispielprogramm zum Generieren von Sprites
'mit DATA-Werten
'-----

GOTO Main

'Zeigt Screen13-Defaultpalette an
ZeigePalette:
SCREEN 13
FOR i = 0 TO 100
COLOR i
PRINT i MOD 10;
IF i MOD 10 = 0 THEN PRINT " "; i
NEXT i
END

Spritedaten:
DATA 0,0,0,0,0,0,0,0
DATA 0,0,0,1,1,0,0,0
DATA 0,0,1,1,1,1,0,0
DATA 0,1,2,1,1,2,1,0
DATA 0,1,1,2,1,1,1,0
DATA 0,1,1,2,1,1,1,0
DATA 0,1,4,1,1,4,1,0
DATA 0,0,1,4,4,1,0,0
DATA 0,0,0,1,1,0,0,0
DATA 0,0,0,0,0,0,0,0
xsize = 8
ysize = 10
RETURN

Zeichnesprite:
RESTORE Spritedaten
FOR iy = y TO y + ysize - 1
  FOR ix = x TO x + xsize - 1
    READ farbe
    PSET (ix, iy), farbe
    'PRINT ix, iy, farbe
  NEXT ix
NEXT iy
WHILE (INKEY$ = ""): WEND
RETURN

Main:
```

```

SCREEN 13
x = 100
y = 100
GOSUB Spritedaten
GOSUB Zeichnesprite
END

```

---

Ich habe im Unterprogramm "ZeichneSprite" eine Zeile dringelassen, die auskommentiert ist, die ich aber zum Debuggen verwendet habe. So. Und wie sieht Ihr Sprite aus?

## Bewegung der Sprites auf schwarzem Hintergrund

Das Programm kann man leicht erweitern, um das Sprite herumzubewegen. Prinzipiell. wir berücksichtigen dabei allerdings nicht, was unter dem Sprite eventuell angezeigt war und nun von ihm überschrieben wird. Wir schreiben noch eine zweite Routine namens "LoescheSprite" und rufen beide dann wechselnd auf.

Hier mein Sprite, wie es als Spielball zwischen den Bildschirmrändern hin- und herfliegt:

---

```

'-----
'Beispielprogramm zum einfachen Bewegen von
'Sprites durch Neuzeichnen
'-----

GOTO Main

'Zeigt Screen13-Defaultpalette an
ZeigePalette:
SCREEN 13
FOR i = 0 TO 100
COLOR i
PRINT i MOD 10;
IF i MOD 10 = 0 THEN PRINT " "; i
NEXT i
END

Spritedaten:
DATA 0,0,0,0,0,0,0,0
DATA 0,0,0,1,1,0,0,0
DATA 0,0,1,1,1,1,0,0
DATA 0,1,2,1,1,2,1,0
DATA 0,1,1,2,1,1,1,0
DATA 0,1,1,2,1,1,1,0
DATA 0,1,4,1,1,4,1,0
DATA 0,0,1,4,4,1,0,0
DATA 0,0,0,1,1,0,0,0
DATA 0,0,0,0,0,0,0,0
xsize = 8
ysize = 10
RETURN

Zeichnesprite:
RESTORE Spritedaten
FOR iy = y TO y + ysize - 1
  FOR ix = x TO x + xsize - 1
    READ farbe
    PSET (ix, iy), farbe
    'PRINT ix, iy, farbe
  NEXT ix
NEXT iy
RETURN

LoescheSprite:
LINE (x, y)-(x + xsize - 1, y + ysize - 1), 0, BF
RETURN

Warte:
FOR j = 1 TO 200: z = SIN(1!): NEXT j
RETURN

Main:
SCREEN 13
x = 1
y = 1
xdir = 1
ydir = 1
GOSUB Spritedaten
WHILE (INKEY$ = "")
  GOSUB Zeichnesprite
  GOSUB Warte
  GOSUB LoescheSprite
  x = x + xdir
  y = y + ydir
  IF (x > 320 OR x < 0) THEN xdir = -xdir
  IF (y > 200 OR y < 0) THEN ydir = -ydir

```

```
WEND
END
```

Wenn Sie nichts sehen sollten, es zu langsam oder zu schnell geht, dann müssen Sie die Zahl "200" in der Routine "Warte" anders einstellen.

Sie sehen, dass das Hauptprogramm "Main" ganz einfach geblieben ist - dank Unterprogramme.

## Bewegung der Sprites auf beliebigem Hintergrund

Natürlich ist das noch kein gutes Spriteprogramm. Das Sprite löscht auf seiner Spur alles weg und es bewegt sich zwar recht schnell, aber etwas ruckelig. Als Nächstes lernen wir, wie das Sprite sich den Hintergrund merkt. Dazu gibt es eine einfache Anweisung:

Syntax: GET (x1,y1)-(x2,y2),buf

**GET** buf ist ein Array der Grösse xsize\*yysize, das ganz am Anfang des Programms mit DIM deklariert werden muss. GET speichert nun den Bildschirmbereich (x1,y1)-(x2,y2) in buf.

Syntax: PUT (x,y),buf,flag

**PUT** malt den Bildausschnitt von buf an die Stelle (x,y). "flag" ist ein Schlüsselwort, das den Malmodus bezeichnet. Die möglichen Modi sind: AND, OR, PSET, PRESET oder XOR. Weitere Informationen finden Sie dazu in der Online-Hilfe von Qbasic.

Für unseren Zweck ist bei PUT nur der Modus PSET relevant, der einfach das neue Bild über den alten Hintergrund drübermalt und sich nicht drum schert, was da schon steht. Die anderen Modi verknüpfen in irgendeiner Form die Farben des Hintergrunds mit den Farben von buf.

**Übung 1:** Bauen Sie eine Routine "BewegeSprite" mit den Befehlen Get, Put und der schon bekannten Routinen "ZeichneSprite". Die Routine "LoescheSprite" werden Sie nicht mehr brauchen. Die Routine "BewegeSprite" soll das Sprite von (x1,y1) nach (x2,y2) bewegen. folgendes ist zu tun:

- Den Hintergrund an der alten Position (x1,y1) mit PUT wiederherstellen. Damit ist das alte Sprite auch wieder gelöscht.
- Den Hintergrund an der neuen Stelle (x2,y2) bis (x2+xsize-1,y2+yysize-1) abspeichern.
- Dann das Sprite bei (x2,y2) zeichnen.

Viel Spass!

Meine Lösung lautet:

```
'-----
'Beispielprogramm zum einfachen Bewegen von
'Sprites ueber einen Hintergrund durch Neuzeichnen
'-----

DIM BHG(9 * 11 + 1)

GOTO Main

'Zeigt Screen13-Defaultpalette an
ZeigePalette:
SCREEN 13
FOR i = 0 TO 100
COLOR i
PRINT i MOD 10;
IF i MOD 10 = 0 THEN PRINT " "; i
NEXT i
END

Spritedaten:
DATA 0,0,0,0,0,0,0,0
DATA 0,0,0,1,1,0,0,0
DATA 0,0,1,1,1,1,0,0
DATA 0,1,2,1,1,2,1,0
DATA 0,1,1,2,1,1,1,0
DATA 0,1,1,2,1,1,1,0
DATA 0,1,4,1,1,4,1,0
DATA 0,0,1,4,4,1,0,0
DATA 0,0,0,1,1,0,0,0
DATA 0,0,0,0,0,0,0,0
xsize = 8
ysize = 10
RETURN

ZeichneSprite:
RESTORE Spritedaten
FOR iy = y TO y + ysize - 1
FOR ix = x TO x + xsize - 1
READ farbe
IF farbe = 1 THEN farbe = 53
IF farbe = 2 THEN farbe = 46
PSET (ix, iy), farbe
```

```

        'PRINT ix, iy, farbe
      NEXT ix
    NEXT iy
  RETURN

LoescheSprite:
LINE (x, y)-(x + xsize - 1, y + ysize - 1), 0, BF
RETURN

Warte:
FOR j = 1 TO 200: z = SIN(1!): NEXT j
RETURN

ZeichneHintergrund:
FOR ix = 0 TO 320 STEP 20
  LINE (ix, 1)-(ix, 200), 6
NEXT ix
RETURN

SpeichereBHG:
' BHG = BildHinterGrund
rspeich = 0
IF (x > 0 AND y > 0 AND x + xsize - 1 < 320 AND y + ysize - 1 < 200) THEN
  GET (x, y)-(x + xsize - 1, y + ysize - 1), BHG
  rspeich = 1
END IF
BHGinit = 1'Zeigt, dass BHG sinnvolle Werte enthaelt
RETURN

MaleBHG:
IF (x > 0 AND y > 0 AND x + xsize - 1 < 320 AND y + ysize - 1 < 200) THEN
  PUT (x, y), BHG, PSET
END IF
RETURN

BewegeSprite:
'...von (x1,y1) nach (x2,y2)
x = x1: y = y1
IF (BHGinit = 1) THEN GOSUB MaleBHG
'LOCATE 20, 20: PRINT 1;
'WHILE (INKEY$ = ""): WEND
x = x2: y = y2
GOSUB SpeichereBHG
'LOCATE 20, 20: PRINT 2;
'WHILE (INKEY$ = ""): WEND
IF rspeich = 1 THEN GOSUB ZeichneSprite
'LOCATE 20, 20: PRINT 3;
'WHILE (INKEY$ = ""): WEND
RETURN

Main:
SCREEN 13
BHGinit = 0
GOSUB ZeichneHintergrund
x0 = 1
y0 = 1
xdir = 1
ydir = 1
x1 = x0: y1 = y0
GOSUB Spritedaten
WHILE (INKEY$ = "")
  x2 = x0
  y2 = y0
  GOSUB BewegeSprite
  x1 = x0: y1 = y0 'Zwischenspeichern der alten Koordinaten --> BHG!
  GOSUB Warte
  IF (x0 + xdir > 320 - xsize OR x0 + xdir < 0) THEN xdir = -xdir
  IF (y0 + ydir > 200 - ysize OR y0 + ydir < 0) THEN ydir = -ydir
  x0 = x0 + xdir
  y0 = y0 + ydir
WEND
END

```

Ich habe hier wieder ein paar auskommentierte Debug-Zeilen dringelassen. "Main" habe ich etwas schöner gestaltet, indem nun die Grenzenprüfung die Grösse des Sprites berücksichtigt. Beachten Sie, dass "SpeichereBHG" einen Rückgabewert rspeich liefert, der informiert, ob der Hintergrund überhaupt abgespeichert werden konnte. Ist dies nicht der Fall, dann darf auch nichts Neues gezeichnet werden. Das sichert ab, dass nicht plötzlich die Ordnung von "Speichern" und "Zeichnen" durcheinander kommt.

**Übung 2:** Modifizieren Sie das Programm von oben (oder ihr eigenes!) so, dass das schwarze Viereck um das eigentliche Sprite herum verschwindet. Das ist ganz einfach und erfordert nur eine Halbzeile!

**Übung 3:** Etwas schwieriger, aber nicht so schwierig, wie es scheint: Schreiben Sie ein Programm, das mehrere identische Sprites gleichzeitig über den Bildschirm fliegen lässt!

**Übung 4:** Modifizieren Sie das Programm so, dass Sie die Sprites nicht immer neu mit PSET auf den Bildschirm pixeln, sondern ebenfalls mit PUT übertragen. Gewinnt das Programm an Geschwindigkeit?

**Übung 5:** Fügen Sie nun ein zweites, andersartiges Sprite ein, dessen Bewegung Sie mit den Cursortasten steuern können. Nennen wir dieses Sprite "Janosch". Und die anderen Sprites sind Fliegen. Geben Sie einen Beep aus, (Anweisung BEEP), wenn eine Fliege Janosch trifft. Hurra! Wir haben ein richtiges Action Game!

### **Ausblick**

Wenn Sie sich mehr für Grafikprogrammierung mit QBASIC, insbesondere Sprites, interessieren: Es gibt eine Menge guter Literatur zum Thema auf [qbasic.de](http://qbasic.de)

.

---

# Textdateien lesen und schreiben

---

Auf dem C16 hat sich der Umgang mit Dateien auf das Speichern ("SAVE") und Laden ("LOAD") von Programmdateien beschränkt. Angesichts der Tatsache, dass damals meist nur Audiocassetten als Speichermedium zur Verfügung standen - kein Wunder. Auf einem modernen PC sieht das anders aus. Er bietet mit der Festplatte die Möglichkeit, sehr schnell auf Dateien zugreifen zu können. Und zwar nicht nur auf Programmdateien - bei den paar Kilobyte ist die Geschwindigkeit nicht relevant. Aber insbesondere auf Datendateien, also Dateien, in die unsere Programme Daten reinschreiben und sie wieder auslesen. Wir können also Dateien als eine Art Hauptspeicherersatz benutzen. Und in diesem Zusammenhang sind die Vorteile Legion:

- Dateien sind resistent. Wir können also Speicherinhalte permanent sichern.
- Dateien können bis zu 2 GB gross werden - keine Platzprobleme.
- Dateien können durch mehrere Programme simultan gelesen werden. Wir können also mit einem Programm in Datei X reinschauen, während das andere Programm die Datei X liest. Wir können mit ein bisschen Vorsicht sogar reinschauen, während das andere Programm auf X noch SCHREIBT.
- Wenn wir die Daten in Textform rausschreiben, dann können wir mit jedem beliebigem Editor in unseren Speicher reinschauen.
- Wir können mit dem einfachen QBASIC (theoretisch) alle modernen Windows- programme "beprogrammieren", indem wir ihre Dateien schreiben. Z.B. können wir Grafiken mit grenzenlos hoher Auflösung in True Color erstellen, wenn wir nur Bilddateien erstellen können, die wir dann mit einem Malprogramm oder Viewer anschauen. Auch einfache Worddokumente oder WWW-Seiten können das Ergebnis eines QBASIC-Programms sein.
- Es gibt Dateien mit wahlfreiem Zugriff. Hier kann man ganz gezielt auf Adressen zugreifen, wie beim Hauptspeicher. Wir können also tatsächlich Hauptspeicher auslagern, falls dieser uns zu knapp wird.
- Mit Dateien mit wahlfreiem Zugriff können wir ausserdem ganze Datenbanken realisieren, die aus mehreren Datentabellen, Schlüsseln und Indizes bestehen.

Na, Appetit bekommen? Dann wollen wir mal anfangen!

## Auf welche Datenträger kann ich Dateien schreiben?

Hier ein Tipp, wenn Sie Ihre ersten Versuche starten:

- Verwenden Sie nach Möglichkeit eine Diskette, eine Speicherkarte oder die lokale Festplatte
- Stellen Sie sicher, dass die Diskette, die Speicherkarte oder das Festplattenverzeichnis nicht schreibgeschützt sind. Wenn Sie ein Festplattenverzeichnis verwenden, dann am Besten das, in dem Sie auch Ihre Programme speichern. Das ist auf alle Fälle nicht schreibgeschützt. Machen Sie aber vorher eine Sicherheitskopie Ihrer Programme!
- Versuchen Sie nicht, auf ein CD- oder DVD-Medium zu schreiben. Das wird nicht klappen. (Ausnahme: DVD-RAM)

## Arten des Dateizugriffs: Sequentiell und Random Access

Es gibt zwei Arten, wie man Dateien nutzen kann. Bei der ersten lesen oder schreiben wir eine Datei wie ein Magnetband: Alles hintereinander, von Anfang bis Ende. Das ist der programmiertechnisch einfachere, s.g. *sequentielle* Zugriff. Beim *Random Access*-Zugriff (wahlfreien Zugriff) weisen wir an, irgendwo in der Mitte die Datei zu lesen. Das geht. Denn die Datei besteht zunächst wie alles beim Computer aus Bytes. Und wir können sagen: "Bitte fange mit 200567. Byte an und nicht mit dem ersten!" Byte 200566 (wir zählen immer von null weg!) ist dann die Adresse des Bytes - genau wie im Hauptspeicher. (Siehe [Bits, Bytes und Basic](#))

In der Regel werden wir bezüglich RA-Zugriff allerdings komfortablere Möglichkeiten haben, so auch bei qbasic. Wir können anweisen, nicht das i-te Byte aus der Datei zu lesen, sondern das i-te Element. Dabei kann jedes Element, das wir geschrieben haben, ein ganzes Array sein. Oder zumindest eine Fliesskommazahl, die ja bekanntlich mehr Speicher braucht als nur ein Byte. Mit diesen Möglichkeiten werden wir uns im nächsten Kapitel beschäftigen.

## Binary Dateien und Textdateien

Es gibt zwei grundverschiedene Dateitypen, die haben wir schon bei der Einführung in DOS kennengelernt: Binary-Dateien und Textdateien. Ganz, ganz früher in der PC-Steinzeit gab es den Grundsatz, grosse Dateien als Binary-Dateien zu schreiben. Das sollte man heute bleiben lassen. Heute gilt die dicke Empfehlung: **Schreiben Sie selbst Daten auf die Festplatte und können Sie das Format selbst bestimmen, so schreiben Sie eine Textdatei!** Der Grund ist einfach: Textdateien können Sie mit einem normalen Editor lesen. Und damit auch kontrollieren, wenn Ihr Programm mal Mist gebaut haben sollte. Und Sie können an die Daten auch dann noch rankommen, wenn Sie das Programm gerade mal nicht zur Hand haben. Und Sie werden sehen: Wenn Sie nicht ein ganz, ganz, ganz ordentlicher Mensch sind, werden Sie spätestens nach drei Jahren **nie** das passende Programm zu den Daten mehr haben. Das ist einfach so, glauben Sie mir es. Schauen Sie sich qbasic an: Es dauert nicht mehr lange und Sie müssen ziemliche Klimmzüge machen, qbasic überhaupt noch auftreiben und zum Laufen bringen zu können!

Übrigens sind Sie mit dieser Textdateien-Strategie auf der Höhe der Zeit: Das "modernste" Dateiformat nennt sich XML und ist ein Textformat. Auch Webseiten und das Word-Austauschformat RTF sind Textformate.

Sie natürlich sich auch mit Binary-Dateien befassen müssen, z.B. wenn Sie es mit Bilddateien lesen oder schreiben wollen (oder gar Videos! Das kann ich Ihnen in qbasic aber nur begrenzt empfehlen...) Insbesondere wenn Sie mit qbasic eine Datenbank bauen wollen (was durchaus geht!), müssen Sie RA-Dateizugriffe machen und diese funktionieren nur mit Binaries.

## Schreiben von Textdateien

Das Schreiben oder Lesen Dateien geschieht in 3 Schritten:

1. Öffnen der Datei
2. Schreiben oder Lesen
3. Schliessen der Datei

### Punkt 1: Öffnen einer Datei

Das Kommando sieht ein bisschen kompliziert aus, ist aber ganz einfach:



Lesezugriff: **OPEN "datei.txt" FOR INPUT AS #1**

Löschender Schreibzugriff: **OPEN "datei.txt" FOR OUTPUT AS #1**

Anhängender Schreibzugriff: **OPEN "datei.txt" FOR APPEND AS #1**

Die Befehle unterscheiden sich also nur im Schlüsselwort INPUT, OUTPUT und APPEND.

"Löschender Schreibzugriff" heisst: Wenn die Datei schon existiert, dann wird sie gelöscht, bevor nun neu geschrieben wird. "Anhängender Schreibzugriff" heisst: Die Datei bleibt bestehen und das, was wir nun neu schreiben, wird hinten an gehängt.

Das Ende jedes OPEN-Befehls, dieses "#1", ist ein Name, unter dem Sie die Datei (engl. das "File") ansprechen können, das s.g. Filehandle. Sie werden ihn beim Schreiben oder Lesen brauchen. Es gibt durchaus die Möglichkeit, mehrere Files offen zu haben. Sehr gebräuchlich ist, eines zum Lesen und eines zum Schreiben zu öffnen.

## **Punkt 2: Schreiben von Text in Dateien**

Das ist nun einfacher, als Sie vielleicht denken. Sie schreiben in eine Textdatei genauso, wie Sie auf den Bildschirm schreiben: Mit PRINT. Das Einzige, was Sie machen müssen, ist, den Filehandle davorzuhängen. Wir schreiben also:

```
OPEN "text1.txt" FOR OUTPUT AS #1
PRINT #1,"Hallo Welt! Ich bin in einer Datei!"
CLOSE #1
```

Und schon haben wir unser erstes Programm fertig. (Das Close kommt erst im nächsten Kapitel, ist aber wohl nicht schwer.)

Wenn Sie dieses Programm ausführen, scheint gar nichts zu passieren. Klar, auf dem Bildschirm passiert ja auch nichts. Aber wenn Sie nun in Ihrem Dateimanager oder mit dem dir-Kommando von der Konsole aus das betreffende Verzeichnis anschauen, müsste dort eine neue Datei namens "text1.txt" stehen. Diese können Sie nun mit einem Editor, z.B. edit, notepad oder qbasic selbst öffnen und schauen, was drin steht. Und? Hat alles geklappt?

## **Wenn Sie die Datei nicht finden sollten...**

...keine Panik. Ihr qbasic *kann* Dateien schreiben und Ihr PC auch und Sie auch. Ist nur eine Frage, wo der Fehler steckt...

- Gehen Sie in qbasic in "Datei" und "öffnen". Geben Sie in Dateiname "\*.txt" ein und schauen Sie nun auf das darunter gelistete Verzeichnis. Ist die Datei dabei?
- Wenn nicht: Haben Sie eine Fehlermeldung beim Ausführen des Programms bekommen? Meldet qbasic, dass die Datei nicht angelegt werden konnte? Warum nicht? Haben Sie einen Namen für die Datei angegeben? Haben Sie auf dem Verzeichnis überhaupt Schreibzugriffsrechte?
- Um Ihre Schreibzugriffsrechte zu prüfen, sollten Sie einfach versuchen, mit irgendeinem Editor oder Textverarbeitungsprogramm eine Datei auf diesem Verzeichnis abzuspeichern. Klappt das? Wenn Nein: Dann haben Sie den Fehler. Fragen Sie jemand, der sich damit auskennt, wie man auf einem Verzeichnis einen Vollzugriff einrichtet. Am Besten, Sie wählen für Ihre Tests vorerst ein anderes Verzeichnis, auf dem Sie auf alle Fälle abspeichern können. Sie können als Dateinamen auch einen kompletten Pfad eintragen, z.B.

"C:\meins\basic\text1.txt". Das ist kein Problem.

- Prüfen Sie, ob das Medium, auf das Sie schreiben wollten, schreibgeschützt ist.

## Punkt 2: Lesen der Datei

Das Lesen ist grundsätzlich nicht schwieriger als das Schreiben. Probleme kann es nur dann machen, wenn der Lesevorgang nicht genau weiss, welche Textteile der Datei in welcher Variablen abgespeichert werden sollen. Aber das Problem haben wir erst später. Bisher ist alles noch ganz einfach. Wie beim Schreiben ist auch beim Lesen der Befehl bekannt: INPUT #1,a\$. Dies liest die nächste Zeile der Datei und speichert sie in a\$. Dann mal los!

```
OPEN "text1.txt" FOR INPUT AS #1
INPUT #1,a$
PRINT a$
CLOSE #1
```

Jetzt müsste unser Text auf dem Bildschirm erscheinen.

## Fehlermeldung: "Datei ist schon geöffnet"

Das wird beim Testen unweigerlich passieren: Man startet das Programm neu und plötzlich funktioniert der OPEN-Befehl nicht mehr - mit der obigen Fehlermeldung. Was passiert da? Nun, Dateien bleibt in QBASIC auch dann geöffnet, wenn man den Programmablauf abbricht, das Programm verändert und dann neu startet. Da hilft nichts anderes, als in die Eingabezeile (F6) zu hüpfen und ein manuelles "close #1" abzuschicken, bevor man neu startet, falls man den Lauf zuvor vorzeitig abgebrochen hat.

## Ende der Datei: EOF()-Funktion

Beim Lesen ist es wichtig, zu wissen, ob man das Dateiende schon erreicht hat. Leseversuche, wenn man alles schon "weggelesen" hat, sind nicht sinnvoll. Diese Info gibt uns die Funktion EOF(Filenumber). Sie ist True, falls das Dateiende erreicht wurde. Wir schreiben also z.B.

```
OPEN "test.txt" FOR INPUT as #1
WHILE NOT eof(1)
  input #1,a$
  ? a$
WEND
close(1)
```

um uns die gesamte Datei anzeigen zu lassen, egal wie gross sie ist.

## Umlaute in DOS- und Windows-Texten

Wenn Sie zufällig mit einem Windows-Programm einen Text erstellt haben, in dem Umlaute vorkommen, z.B. "Käse oder Brötchen?", dann werden Sie enttäuscht sein, was dabei herauskommt, wenn Sie das in DOS einlesen. Das liegt daran, dass die deutschen Umlaute nicht zum Umfang des s.g. ASCII-Zeichensatzes gehören und unter DOS anders kodiert sind, als in der unter Windows verwendeten Kodierung, der s.g. ANSI-Kodierung. Wenn man also in einem DOS-Programm Umlaute verwendet, sieht man unter Windows nur Quatsch und umgekehrt. Viele Windowsprogramme haben Optionen, mit denen man den Text explizit als DOS-Text einladen kann. Umgekehrt gibt's diese Option natürlich meist nicht.

Was tun? Nun, wir werden in unseren qbasic-Programmen selten Briefftexte verarbeiten. Sondern eher technischere Daten, z.B. Zahlentabellen u.ä. Hier empfiehlt es sich, falls mal irgendwo Text auftaucht, diesen mit oe, ae usw. zu schreiben. Das kann man überall lesen. (Ist übrigens auch eine Empfehlung für die Programmkommentare!)

Und wenn man es mit Namenslisten o.ä. zu tun hat? Dann gibt es mehrere Möglichkeiten:

- Sich bewusst sein, dass DOS hier nur ein Lernschritt ist und wir bald zur Windowsprogrammierung kommen.
- eine Konvertierung für die Umlaute schreiben.
- Nur DOS-Editoren (edit) und qbasic verwenden. Dann macht die DOS-Kodierung kein Problem.

## Arrays schreiben und lesen: Trennzeichen

In Arrays speichern wir Zahlen- oder Stringtabellen. Oft sind das Daten, die im Programm von vornherein feststehen, s.g. Parameter. Bis jetzt haben wir diese entweder mit LET-Anweisungen direkt im Programmtext festgehalten oder in DATA-Zeilen verpackt. Aber es gibt eine viel bessere und flexiblere Methode: Wir schreiben sie mit einem Editor in eine Textdatei und lesen sie von dort ins Programm!

Unser Übungsbeispiel sollen die Schulnoten einer Klasse sein. Wir haben es mit einer Klasse von 32 Schülern zu tun und sie haben 4 Arbeiten geschrieben, ausserdem gibt es eine mündliche Note. Wir wollen nun sowohl die Note jedes Schülers als auch die Klassendurchschnitte errechnen.

Die Arbeit, ein Programm zu schreiben, das die ganzen Einzelnoten aufnimmt, können wir uns sparen. Das können wir mit einem Editor genauso machen.

Die Tabelle soll folgendes Format haben:

```
Name      Note_P1  Note_P2  Note_P3  Note_P4  Note_muendl
```

P1,P2,P3,P4 sind die Abkürzungen für die Prüfungen. In einer Zeile stehen also insgesamt sechs Angaben. Wie lesen wir diese sechs Angaben ein? Machen wir einmal folgenden Versuch:

```
Mueller 1.5 2.5 3.5 2.5 2
```

Zwischen jeder Zahl lassen wir ein Leerzeichen. Wir speichern das in der Datei "test1.txt" ab und lesen es mit Vierzeiler von oben ein. Was ist in a\$ gespeichert? Die ganze Zeile! Inklusive aller Leerzeichen! qbasic erkennt also nicht, dass die Leerzeichen eigentlich Trennzeichen sein sollen und nach "Mueller" die erste Angabe schon beendet ist. Was jetzt tun? Nun, andere Trennzeichen ausprobieren. Tabulatoren klappen auch nicht. Aber Kommas gehen! Die Angabe

```
Mueller,1.5,2.5,3.5,2.5,2
```

führt dazu, dass nur noch "Mueller" in a\$ gespeichert ist. Und die Abwandlung des Programms in

```
DIM note(5)
OPEN "test1.txt" FOR INPUT AS #1
INPUT #1, name1$
FOR i = 0 TO 4
    INPUT #1, note(i)
NEXT i
PRINT name1$;
FOR i = 0 TO 4
    PRINT TAB(10 + i * 10); note(i);
```

sollte eigentlich das gewünschte Resultat erbringen.

Jetzt gibt es noch eine Kleinigkeit: Wir wollen in die erste Zeile unserer Input-Datei eine Kopfzeile schreiben, die auch schon in der Datei ersichtlich werden lässt, was was ist. Die Datei soll also so aussehen:

```
Name Note_P1 Note_P2 Note_P3 Note_P4 Note_muendl
Mueller,1.5,2.5,3.5,2.5,2
Mayer,3.5,3.5,3,4,3.5,4
```

Hinter Mueller und Mayer kommen natürlich noch die anderen 30 Schüler. Wir müssen nun unser Programm anweisen, die ersten beiden Zeilen zu überspringen. Wie machen wir das? Nun, wir können die ersten beiden Zeilen natürlich einlesen; wir lesen sie einfach in eine x-beliebige Variable ein, deren Inhalt nicht weiters wichtig ist. Wichtig ist nur, dass in der Datei in der ersten und zweiten Zeile *keine* Kommas auftauchen, damit wir die ganze Zeile in einem Rutsch in eine Stringvariable bekommen. Das Ganze sieht dann so aus:

```
DIM note(5)
OPEN "test1.txt" FOR INPUT AS #1
FOR i = 0 TO 1: INPUT #1, a$: NEXT i
INPUT #1, name1$
FOR i = 0 TO 4
    INPUT #1, note(i)
NEXT i
PRINT name1$;
FOR i = 0 TO 4
    PRINT TAB(10 + i * 10); note(i);
NEXT i
PRINT
CLOSE #1
```

## Von einer Tabellenkalkulation nach qbasic

Wenn Sie sich mit einer Tabellenkalkulation wie StarCalc oder Excel auskennen, dann werden Sie sich sagen: Ja, so eine Tabelle kann ich doch viel schöner dort erstellen, als in einem Editor! Gibt es eine Möglichkeit, so eine Tabelle in qbasic einzulesen?

Gibt es zuhauf. Im Prinzip können Sie schon jetzt in qbasic so gut wie alles programmieren, sogar ein DVD-Authoring-Programm...Aber der Reihe nach.

Ich beziehe mich in meiner Erklärung jetzt auf Excel, andere Programme funktionieren ganz ähnlich. Man erstellt also die Tabelle ohne grosse Formatierung. Dann markiert man die Tabelle und speichert die Markierung als Datei des Typs "csv" ab. Das heisst: "Comma Seperated Value". Dieser Name ist irreführend, denn es sind nicht Kommas, die die Werte trennen, sondern Semikolons. Das liegt daran, dass im Deutschen Kommas schon die Dezimaltrennzeichen sind. Und das ist auch das Problem beim Übergang von csv nach qbasic: qbasic ist Englisch und braucht Punkte als Dezimaltrennzeichen und Kommas als Trennzeichen. Was tun?

Nun, jeder Editor hat heute eine "Suchen" und "Ersetzen"-Funktion. Man geht also in den Editor und ersetzt mittels dieser Funktion zunächst alle Kommas durch Punkte. Und dann alle Semikolons durch Kommas. (In dieser Reihenfolge!) Und schon ist die englische csv-Datei fertig.

Jetzt besteht noch eine Stolperfalle: Unsere Kopfzeile. Sie ist natürlich nun auch kommagetrennt.

Deshalb muss unser Programm so angepasst werden, dass es die Kopfzeile "formattreu" einliest, d.h. wirklich sechs Strings pro Zeile. Oder den LINE INPUT-Befehl benutzt. Siehe nächster Abschnitt. Das war's dann aber auch schon.

## LINE-INPUT-Befehl

In manchen Fällen will man tatsächlich, dass die ganze Zeile einer Stringvariablen übergeben wird, egal ob darin ein Komma vorkommt oder nicht. Wie mache ich das? Dafür gibt es einen extra Befehl: LINE INPUT. LINE INPUT #1,a\$ übergibt die ganze Zeile an a\$. Auf diese Weise können Zeilen in die Dummy-Variable a\$ "weggelesen" werden, egal was sie enthalten.

## Width-Befehl

Manches ist bei QBASIC/VBasic nicht so toll gelöst. So die Tatsache, dass der Fileausgabe genau die gleiche physische Spaltenbreite von 80 Zeichen pro Zeile unterstellt wird wie der Bildschirmausgabe. D.h., die Ausgabe bricht auch dann nach 80 Zeichen um, wenn man noch gar keinen Zeilenumbruch geschrieben hat. Ein bisschen lindern kann man das Problem mit dem Width-Befehl: Mit WIDTH #1,255 setze ich die Fileausgabe auf 255 Zeichen pro Zeile Maximalwert für File #1. Auch das ist viel zu wenig. Wenn man mehr braucht, hat man erstmal Pech gehabt. Das gilt übrigens auch für die Eingabe: QBASIC ist nicht in der Lage, Files zu lesen, deren Zeilenbreite 255 Zeichen übersteigt...

## Übungen

1. Schreiben Sie auf der Basis des gerade Gelernten das Notenberechnungsprogramm! Geben Sie die Ergebnisse auf dem Bildschirm aus.
2. Schreiben Sie eine Variante, die die Ergebnisse in die Datei "results.txt" schreibt.
3. Schreiben Sie die Programme aus dem letzten Kapitel so um, dass die Spritedaten aus einer Datei eingelesen werden und nicht aus DATA-Zeilen!
4. Nicht ganz einfach: Schreiben Sie ein Unterprogramm, das in einer Textdatei nach einem frei definierbaren Schlüsselwort sucht, also solange die Textdatei einliest, bis das Schlüsselwort gefunden wurde. Dieses Unterprogramm ist sehr nützlich, um in ein und derselben Datei mehrere Tabellen unterzubringen, die man jeweils durch ein Schlüsselwort kennzeichnet. Man kann dann praktisch wahlfrei auf die einzelnen Tabellen zugreifen. So funktionieren Initialisierungsdateien unter Linux!

## Exkurs: "Reengeneering" von Text-Dateiformaten am Beispiel von Postscript (für den kleinen Gebrauch)

[Hier entlang](#)

---

## Exkurs: Postscriptgrafiken erstellen (für den kleinen Gebrauch)

---

Diesen kleinen Exkurs können Sie nur mal überfliegen. Er ist ein kleiner Appetitanreger. Vielleicht gewinnen Sie Interesse an dem Thema und steigen tiefer ein...

Wir wollen mit QBASIC Vektorgrafiken erstellen. Vektorgrafiken sind Grafiken, die mittels "Steuerdateien" gespeichert werden. Eine solche Steuerdatei besteht aus einer Sequenz von Zeichenanweisungen, z.B. dem Zeichnen von Linien, Kreisen, Vierecken usw. Der Vorteil solcher Vektorgrafiken ist u.a., das sie auf dem Drucker in maximaler Auflösung dargestellt werden und nicht in lauter "Klötzchen" zerfallen, wie das gerne mit Pixel-Grafiken (bmp, gif, jpg) der Fall ist.

Können wir mit QBASIC auch solche schönen vektorgrafiken herstellen? Prinzipiell ja: Einfach nur eine entsprechend formatierte vektorgrafikdatei schreiben, z.B. eine Corel Draw-Datei oder Powerpoint-Datei. Nichts einfacher als das...

Das scheitert gegenwärtig schon daran, dass die aufgezählten Dateiformate BINÄR-Formate sind - und die lernen wir erst später. Zweitens sind diese Formate aber extrem komplex und es würde Hunderte von Seiten Einarbeitung brauchen, um auch nur das erste Viereck auf den Bildschirm zu bekommen. Drittens kennen wir aber die Formate gar nicht, wir müssten also erst einmal irgendwo uns eine Dokumentation beschaffen. Nein, das alles erscheint keine gute Idee.

Nun, es gibt ein Format für die grafische Darstellung, das in Form von Textdateien abgespeichert wird - Postscript. Noch dazu kann jeder Linux- oder Windows-Anwender selbst ohne spezielles Programm dieses Format erzeugen. Dazu geht er in ein Vektorzeichenprogramm (z.B. Power Point oder Zoner Draw oder Corel Draw), zeichnet ein paar Vierecke und druckt das Ganze auf einem Postscript-Drucker aus - allerdings in eine Datei und nicht wirklich auf dem Drucker, den wahrscheinlich nur die wenigstens haben. Man muss sich also als erstes einen Postscript-Drucker installieren. Es macht nichts, dass man den gar nicht angeschlossen hat. Ich empfehle den "Apple Color LW 12/660", das ist ein ziemlicher Standarddrucker mit kompatibeltem Postscript.

Nächster Schritt ist also: Ausdruck in eine Datei, die wir mit der Endung ".ps" versehen. So, nun kommt der einzige Haken: Wir wollen die Datei ansehen, OHNE sie auszudrucken. Wie geht das? Nun, dazu brauchen wir ein Programm namens "Ghostview", das es im Internet kostenlos zum Herunterladen gibt. Wenn Sie das installiert haben, können Sie sich die Datei ansehen - und sogar in ein pdf umwandeln. Und Sie können die Datei auf Ihrem Standarddrucker ausdrucken.

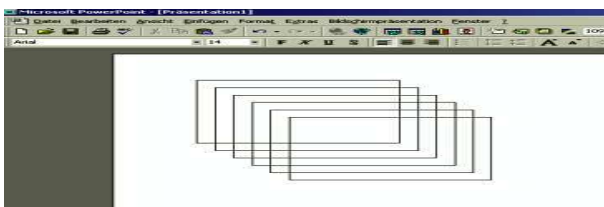
### Der Plan

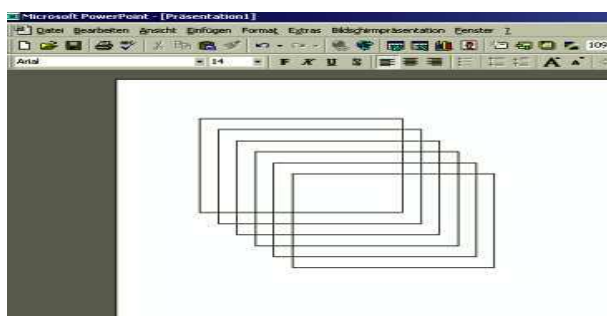
Soweit, so schön. Aber dadurch wissen wir immer noch nicht, wie wir ps-Dateien mit QBASIC erzeugen. Wir kennen das postscript-Format ja nicht und es wird schon hinreichend kompliziert sein.

Das kann schon sein. Aber es geht auch gar nicht darum, nun alles zu verstehen. Es geht darum, erst einmal ein *Muster* zur Hand zu haben und die Teile darin zu identifizieren, die die Vierecke darstellen könnten. Dann schauen wir uns die Viereck-Angaben (oder das, was wir dafür halten) an und ändern den ein oder anderen Wert. Mit Ghostview schauen wir, was das Ergebnis davon ist. Und am Ende wissen wir, wie die grundlegende Syntax für "Viereck zeichnen" in Postscript lautet. Dann lassen wir QBASIC das ganze "Drumherum" wortgetreu schreiben und dazwischen beliebige Viereck-Anweisungen. Mit Viereck-Anweisungen kann man auch Linien und Punkte malen. Wir haben also alles zur Verfügung, um einfache Vektorgrafiken zu erstellen.

### 1. Schritt: Wir malen mit einem Vektorzeichenprogramm Vierecke.

Das geht schon meist mit einer Textverarbeitung wie MS Word oder Open Office Write. Ich benutze hier mal Powerpoint:



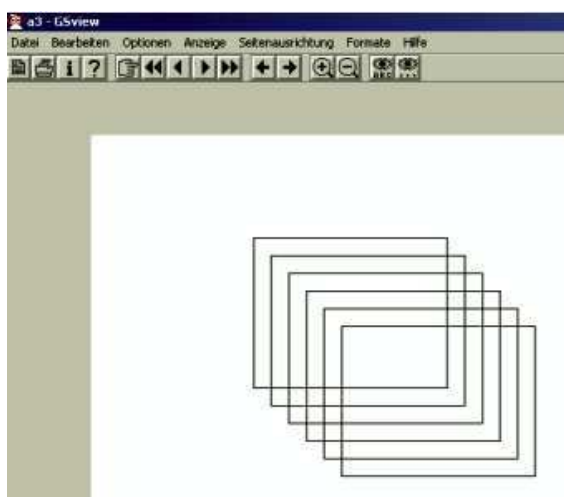


Es ist ganz gut, gleich viele Vierecke zu malen, da dann die charakteristische Struktur im ps-File besser auffällt.

## 2. Schritt: Wir drucken die Zeichnung auf dem ps-Drucker als File aus.



## 3. Schritt: Wir schauen uns die Datei in einem ps-Viewer an:



Das sieht soweit gut aus.

## 4. Schritt: Wir schauen uns die Datei in einem Editor an:

Sie beginnt mit:

```
%!PS-Adobe-3.0
%%Title: (Microsoft PowerPoint - Pr\344sentation1)
%%Creator: PScript5.dll Version 5.2
%%CreationDate: 2/1/2005 16:15:15
```

```
%%For: schatz
%%BoundingBox: (atend)
...
```

Darunter sehen wir schnell, dass Schlüsselworte der Postscript-Sprache mit "%%" eingeleitet werden. Die Datei besteht aus mehreren Abschnitten, die z.B. mit "%%BeginProlog" beginnen und mit "%%EndProlog" enden. Prologe interessieren uns nicht. Viele weitere haben was mit "Resource" zu tun. Ressourcen interessieren uns auch nicht.

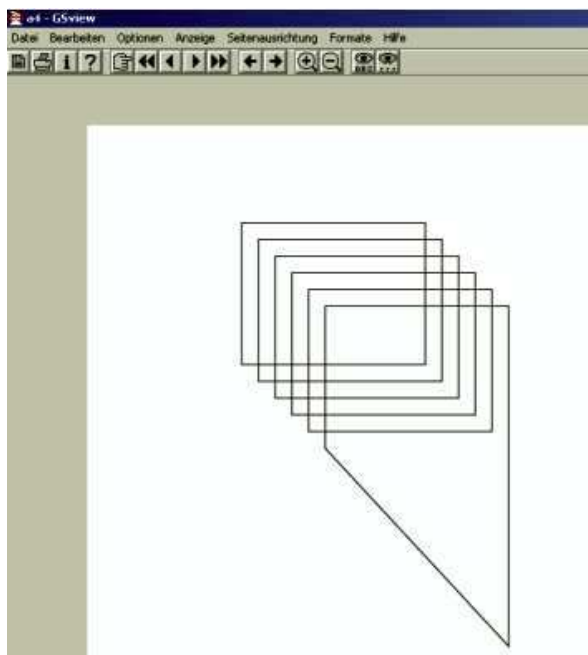
Ziemlich weit unten, kurz vor "\$ § Trailer" sehen wir viele Zahlen. Das sieht ungefähr so aus:

```
: N 350 112 6000 4500 rp C
1 1 1 1 scol L ; 0 0 0 1 scol 0 Lj 1 Lc 6 Lw solid N 1900 462 M 800 462 I 800 1312 I 1900 1312 I C
: [ 1 0 0 1 350 112 ] concat K
; N 2000 562 M 900 562 I 900 1412 I 2000 1412 I C
: [ 1 0 0 1 350 112 ] concat K
; N 2100 662 M 1000 662 I 1000 1512 I 2100 1512 I C
: [ 1 0 0 1 350 112 ] concat K
; N 2200 762 M 1100 762 I 1100 1612 I 2200 1612 I C
: [ 1 0 0 1 350 112 ] concat K
; N 2300 862 M 1200 862 I 1200 1712 I 2300 1712 I C
: [ 1 0 0 1 350 112 ] concat K
; N 2400 962 M 1300 962 I 1300 1812 I 2400 1812 I C
: [ 1 0 0 1 350 112 ] concat K
; LH
```

Betrachten wir einmal jeweils die Zeile mit dem führenden "N". Wir erkennen, dass die hinter den Buchstaben angegebenen Zahlen, wenn wir sie als x/y-Koordinaten interpretieren, jeweils ein Viereck ergeben! Bingo! Machen wir ein Experiment. Verändern wir im letzten Viereck einfach mal eine Zahl. Machen wir z.B. die letzte Zahl des letzten Vierecks, 1812, zu etwas anderem, z.B. "3000":

```
: N 350 112 6000 4500 rp C
1 1 1 1 scol L ; 0 0 0 1 scol 0 Lj 1 Lc 6 Lw solid N 1900 462 M 800 462 I 800 1312 I 1900 1312 I C
: [ 1 0 0 1 350 112 ] concat K
; N 2000 562 M 900 562 I 900 1412 I 2000 1412 I C
: [ 1 0 0 1 350 112 ] concat K
; N 2100 662 M 1000 662 I 1000 1512 I 2100 1512 I C
: [ 1 0 0 1 350 112 ] concat K
; N 2200 762 M 1100 762 I 1100 1612 I 2200 1612 I C
: [ 1 0 0 1 350 112 ] concat K
; N 2300 862 M 1200 862 I 1200 1712 I 2300 1712 I C
: [ 1 0 0 1 350 112 ] concat K
; N 2400 962 M 1300 962 I 1300 1812 I 2400 3000 I C
: [ 1 0 0 1 350 112 ] concat K
; LH
```

Dann speichern wir die Datei ab und schauen sie uns im ps-viewer an:



Siehe da, wir haben unsere erste ps-Datei erfolgreich manipuliert!



## Weiteres Forschen

Mit systematischem weiterem Austesten lässt sich des weiteren folgendes vermuten:

- Die erste Zeile des Blocks ": N 350 112 6000 4500 rp C " gibt die Seitengröße an. Wir haben also Koordinaten  $350 \leq y \leq 6000$  und  $112 \leq x \leq 4500$  zur Verfügung.
- Die ersten drei der vier Zahlen vor dem "scol" sind RGB-Farbparameter zwischen null und eins.
- Danach kommen Parameter "Lj" (keine Ahnung) und Lc (Linienstärke), sowie der Linienstil ("Solid").
- Nun beginnt die Angabe der einzelnen Koordinaten. Hinter "N" ist der Startpunkt einer Linie und hinter "M" der Zielpunkt. Wenn's noch weitere Punkte gibt, werden sie mit "I" angehängt.

Probieren Sie mal, was passiert, wenn Sie die vorletzte Zeile von oben durch

```
" ; 0 0 1 0 scol N 2400 962 M 300 962 I 1300 1812 I 2400 1812 I 3000 3000 I C"
```

ersetzen. Hier wurde die Farbe Blau gewählt ("0 0 1 0" vor scol), der zweite Punkt (hinter "M") wurde nach links verschoben und hinten wurde ein fünfter Punkt angefügt.

## 5. Schritt: Eine ps-Datei in QBASIC

Was brauchen wir dazu? Zunächst die erstellte Datei als Vorlage. Sie wird durch unser QBASIC-Programm bis zum kritischen Punkt ohne Modifikation in die Zielfeile geschrieben. Wir beschränken uns hier einmal auf Schwarzweiß-Grafiken und nehmen daher als Einstieg unserer eigenen Postscript-Zugabe den Punkt hinter "solid", wenn Linienfarbe und -stil festgelegt sind. Hier fügen wir in die Vorlage ein "\$start" ein, um uns das Auffinden dieses Punkts zu erleichtern. Das Ganze sieht also am Ende so aus:

```
end reinitialize
: N 350 112 6000 4500 rp C
1 1 1 1 scol L ; 0 0 0 1 scol 0 Lj 1 Lc 6 Lw solid
$start
; LH
(%[Page: 1]%) =
```

Anstatt von "\$start" schreiben wir dann unsere eigenen Linien im ps-Format. Das heisst, wir legen Start- und Zielkoordinaten hinter "N" und "M" in einem String ab und schliessen das Ganze mit einem "I C" ab. Das könnte in BASIC so aussehen:

```
' Statt line (x1,y1)-(x2,y2):
BUF$="N "+STR$(INT(x1))+ " "+STR$(INT(y1))
BUF$=BUF$+"M "+STR$(INT(x2))+ " "+STR$(INT(y2))+ " I C"
```

Bei der zweiten und allen folgenden Linien müssen wir allerdings noch ein Semikolon voranstellen.

Dann geben wir BUF\$ aus: PRINT #2,BUF\$

Anschliessend müssen wir noch jeweils eine zweite Zeile der Form ": [ 1 0 0 1 350 112 ] concat K" ausgeben. Das ist wirklich kein Problem. Also:

```
' Statt line (x1,y1)-(x2,y2):
BUF$="N "+STR$(INT(x1))+ " "+STR$(INT(y1))
BUF$=BUF$+"M "+STR$(INT(x2))+ " "+STR$(INT(y2))+ " I C"
IF (iline>0) BUF$="; "+BUF$
PRINT #2,BUF$
PRINT #2," : [ 1 0 0 1 350 112 ] concat K"
```

Damit ist das Herzstück unserer QBASIC-Routine, die "psline"-Routine fertig. Am Ende müssen wir den Rest der Vorlage ausgeben und die Datei schliessen.

## Übung 1: Schreiben Sie zum folgenden QBASIC-Programm eine Postscript-Version!

...das dürfte ja jetzt nicht mehr schwer fallen.

```
rad0 = 100
radf = 60
in = 20
pi = 22 / 7
xoff = 320
yoff = 240

screen1:

SCREEN 12
```

```

FOR ic = 0 TO 4
  rad = rad0 + radf * ic
  FOR ir = 0 TO in - 1
    ang = 2 * pi / in * ir
    angl = ang + pi + pi / in
    ang2 = 2 * pi / in * (ir + 1)
    'PRINT ang, angl
    'INPUT a$
    y1 = yoff - rad * COS(ang): x1 = xoff + rad * SIN(ang)
    y2 = yoff - rad * COS(ang1): x2 = xoff + rad * SIN(ang1)
    y3 = yoff - rad * COS(ang2): x3 = xoff + rad * SIN(ang2)
    LINE (x1, y1)-(x2, y2)
    LINE (x2, y2)-(x3, y3)
  NEXT ir
NEXT ic
END

```

[Natürlich gibt's hier auch ne Lösung...](#)

[...und das Resultat sind da so aus...](#)

## Was haben wir gelernt?

1. Mit langweiligen Textdateien lässt sich wesentlich mehr Spannendes erreichen, als man vermuten würde.
2. Man muss nicht immer alles von Grund auf verstehen, um es benutzen zu können.
3. Systematisches Probieren und die richtigen kleinen Werkzeuge helfen, sich fremder Techniken zu bedienen, auch wenn man die technischen Einzelheiten nicht kennt.

---

# Systemsteuerung mittels QBASIC

---

Mittlerweile sind wir in der Lage, mittels Zusammenfügen einiger Bausteine, die wir gelernt haben, doch einiges zu bewerkstelligen. In diesem Kapitel wollen wir das anhand ein paar Systemsteuerungsaufgaben für *Windows XP* (!) mal ankosten. Die Bausteine, die wir benutzen werden, sind:

- Eine Skriptprogrammiersprache, nämlich BASIC
- Unsere Konsolenkenntnisse von DOS, die wir auf die XP-Konsole anwenden werden.
- Unsere neuen Kenntnisse bezüglich Lesen und Schreiben von Textdateien
- Die Möglichkeit, von QBASIC aus Shellkommandos abzusetzen, nämlich der Befehl SHELL von QBASIC

Von diesen vier Bausteinen fehlt uns nur noch der letzte:

## Der SHELL-Befehl

Dieser ist ganz einfach. Mittels SHELL "Befehl" können Sie ein beliebiges DOS-Kommando abschicken. Also z.B. SHELL "copy A:\\*.\* C:\temp" würde aus QBASIC heraus alle Dateien des Diskettenlaufwerks nach C:\temp kopieren. Sogar das Ausführen von Batch-Dateien klappt so: QBASIC "back.bat" führt das Batchskript back.bat aus. Und jetzt machen Sie mal was ganz Lustiges: Starten Sie mal folgendes Programm:

```
SHELL "qbasic"
```

Was passiert? Scheinbar nichts. Ausser, dass der Eingangsbildschirm, bei dem man die ESC-Taste drücken muss, wieder erscheint. Und dann das Programm verschwunden ist. Das ist auch kein Wunder - wir haben aus dem alten QBASIC heraus ein neues QBASIC gestartet! Beenden wir es über Datei->Beenden, dann kommt zuerst der DOS- Bildschirm wieder mit der Aufforderung, eine Taste zu drücken. Und dann kommt wieder unser alter QBASIC-Schirm mit Programm.

## Aufgabe 1: Windowsverzeichnis ermitteln

Nehmen wir an, wir wollen ein Programm schreiben, das ein anderes Programm so installiert, dass es als DOS-Befehl aus jedem Verzeichnis aufgerufen werden kann. Das zu installierende Programm liege als Binary vor, also als Befehl der Form "myprog.exe".

In jedem Windowssystem gibt es ein Verzeichnis, das von vornherein im Pfad der Kommandozeile steht und somit nach Befehlen durchsucht wird, wenn man auf der Kommandozeile einen Namen eingibt: Das Windowsverzeichnis selbst. Nichts einfacher also, als unser myprog.exe ins Windowsverzeichnis zu kopieren (wenn das auch nicht feine englische Art eines Installationsprogramms ist - aber es funktioniert.)

Unser Installationsprogramm soll auf *jeder* Windowsversion bis zur neuesten laufen. Nun sind die Windowsverzeichnisse aber alles andere als einheitlich. Schon deshalb, weil das Systemlaufwerk nicht immer C: sein muss. Was tun? Wie finden wir heraus, wo das Windows-Verzeichnis ist? Das klingt jetzt schon ganz schön kompliziert.

Ist aber nicht so schwer. Als erstes müssen wir wissen, dass es ein Windows/DOS-Kommando namens **SET** gibt. Öffnen Sie mal eine Konsole und geben Sie SET+enter ein. Wenn ich das hier auf meiner Maschine eingebe, dann kommt folgendes:

```
COMSPEC=C:\WINNT\SYSTEM32\COMMAND.COM
COMPUTERNAME=SCHATZ
EDPATH=C:\progra~1\WATCOM\EDDAT
HOMEDRIVE=C:
HOMEPATH=
INCLUDE=C:\progra~1\WATCOM\H;C:\progra~1\WATCOM\H\NT
LOGONSERVER=\\SCHATZ
MXBIN=D:\Programme\FireDaemon
MXHOME=D:\Programme\FireDaemon
NTRESKIT=D:\Programme\reskit1
NUMBER_OF_PROCESSORS=1
OS=Windows_NT
OS2LIBPATH=C:\WINNT\system32\os2\dll;
PATH=C:\winnt;C:\winnt\system32;C:\PROGRA~1\dosprog\tc\bin;C:\progra~1\dosprog\d
os6
PATHEXT=.COM;.EXE;.BAT;.CMD
PROCESSOR_ARCHITECTURE=x86
PROCESSOR_IDENTIFIER=x86 Family 6 Model 7 Stepping 3, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=0703
PROMPT=$P$G
SYSTEMDRIVE=C:
SYSTEMROOT=C:\WINNT
TEMP=C:\TEMP
TMP=C:\TEMP
USERDOMAIN=SCHATZ
USERNAME=schatz
USERPROFILE=C:\WINNT\Profiles\schatz
WATCOM=C:\progra~1\WATCOM
WWINHELP=F:\BINW
```

Damit wissen Sie schon ziemlich viel über mich und meinen PC - sogar meinen Nachnamen. Sie wissen zumindest, dass ich auf einer ollen NT-Maschine mit einer ziemlich schlaffen CPU arbeite (PROCESSOR\_IDENTIFIER). Was sehen wir hier? Wir sehen die voreingestellten *Systemvariablen*. Genau. Das sind Betriebssystem-variablen, die man abfragen und auch neu setzen kann. Die meisten davon sind Strings. Und eine davon enthält das Windows-Verzeichnis: SYSTEMROOT. Bingo.

Die nächste Aufgabe ist es, den Wert für SYSTEMROOT in unser Programm zu bekommen. Nun, wir haben im Kapitel [Feine DOS-Fähigkeiten](#) im Abschnitt "Umleitungen und Pipes" gelernt, dass man die Ausgabe eines DOS-Befehls in eine Datei umleiten kann. Geben wir also in unserem QBASIC-Verzeichnis "SET > set.txt" an, so leiten wir die Liste, was wir gerade noch auf dem Bildschirm gesehen haben, in die Datei set.txt um.

So, die nächste Aufgabe besteht nun darin, den Wert einer bestimmten Systemvariablen aus set.txt auszulesen. Das ist die Sache mit dem Dateien lesen. Hier muss man ein bisschen clever vorgehen, nämlich das Übungsergebnis aus dem vorigen Kapitel benutzen, bei dem es darum ging, eine *search*-Routine für Textdateien zu schreiben. Die die Textdatei nach einem bestimmten Schlüsselwort durchkämmt. Genau so etwas brauchen wir jetzt. Wenn Sie also die Übung noch nicht gemacht haben, dann schreiben Sie jetzt Ihre Routine "searchtextfile"! Ergebnis der Routine soll ein String mit der Zeile sein, in der das Schlüsselwort zum ersten Mal vorkommt. Kommt es in der Datei nicht vor, soll der String leer sein. Die Suche soll Gross- und Kleinschreibung nicht beachten.

Meine Lösung dafür sieht so aus:

```

'-----
'Lookstring:
'Routine fuer das Suchen eines Strings in einer Textdatei
'-----

'Sucht in #1
'#1 muss geoeffnet sein
'Suchstring: such$
'Rueckgabewert 1: ruck$
'Rueckgabewert 2: suchpos
'Enthaelt die Zeile mit dem Suchstring

'by C. Schatz 2003, Programmiertutorial

GOTO main

lookstring:

found = 0
WHILE (NOT EOF(1) AND found = 0)
    LINE INPUT #1, a$
    suchpos = INSTR(LCASE$(a$), LCASE$(such$))
    IF (suchpos > 0) THEN found = 1
WEND
IF (found = 0) THEN ruck$ = "": suchpos = 0 ELSE ruck$ = a$
RETURN

'Test
'-----
main:
SHELL "set > set.txt"
OPEN "set.txt" FOR INPUT AS #1
such$ = "systemroot"
GOSUB lookstring
CLOSE #1
PRINT ruck$
PRINT suchpos

```

Schauen wir uns diese Lösung einmal an. Was fällt Ihnen auf?

- Ich habe mehrere Funktionen von QBASIC benutzt, die wir noch nicht besprochen haben. Unverschämtheit! Nun, das war Absicht. Es gibt zwei Prinzipien, die sich beim Programmieren immer ein bisschen in der Quere sind. Prinzip 1: Nutze Ressourcen, die schon da sind. Das heisst, programmiere nicht etwas selbst, was andere schon längst besser programmiert haben und das du benutzen kannst. Bevor ich also selbst eine Stringbehandlungsroutine schreibe, schaue ich erstmal in QBASIC nach, ob es nicht etwas entsprechendes gibt. Und siehe da: QBASIC kann sowohl in einem String nach einem Suchstring sichten (INSTR()) als auch einen String in Klein- oder Grossbuchstaben umwandeln (LCASE\$, bzw. UCASE\$). Prinzip 2 ist allerdings: Sei kreativ! Denn auf der anderen Seite schreibt man kleine Teillösungen oft schneller, als man sie z.B. im Internet zusammengesucht hat. Zudem muss man eine Lösung fremder Programmierer immer erst genau auf ihre Eigenschaften untersuchen, während man die Eigenschaft der eigenen Routine natürlich kennt (oder kennen sollte!) Das sind so zwei Prinzipien, zwischen denen man immer hängt, wenn man selbst programmiert. Wenn Sie also sich selbst entsprechende Routinen geschrieben haben, dann war das nicht falsch. Aber Sie sollten auch bedenken: Jemand anderes, der QBASIC kennt, wird Ihr Programm besser verstehen können, wenn Sie nicht Ihre eigenen Routinen verwenden, sondern die von QBASIC.
- Ich habe ziemlich viel Kommentar an den Anfang geschrieben, besonders die Übergabewerte genau erklärt. Das hilft, wenn man die Routine später wieder verwenden möchte.
- Am Anfang kommt gleich ein "Goto". Was soll das denn? Nun, viele Dinge, die wir hier

lernen, sollen ja allgemein für das Programmieren gültig sein, nicht nur in QBASIC. Daher sollten wir auch den Programmaufbau so wählen, wie er allgemein üblich ist. Und das ist: Zuerst die Unterprogramme und dann ganz unten das Hauptprogramm. Das hat zwei Gründe: Der erste ist historisch. Die "grossen" Programmiersprachen sind Compilersprachen. Und ein Compiler übersetzt ein Programm Zeile für Zeile von oben nach unten. Wenn ein Compiler auf einen Ausdruck trifft, z.B. einen Unterprogrammnamen, muss er ihn schon kennen, d.h. wissen, dass dies ein Unterprogramm darstellen muss. Deshalb muss die Deklaration einer Routine immer vor ihrer Verwendung erfolgen. Das ist auch heute noch bei "grossen" Programmiersprachen der Fall. Der zweite Grund ist, dass der Leser eines Quelltextes genau das gleiche Problem hat. Auch er liest das Programm von oben nach unten. Also ist es auch gut, in QBASIC die Ordnung so zu halten. Da QBASIC aber ein Programm von oben nach unten durchgeht und nicht zuerst nach dem Hauptprogramm sucht, muss man es erst zum Hauptprogramm "schicken". Für diesen einen Fall müssen wir ein GOTO verwenden, das geht kaum anders.

- In der While-Schleife wird die EOF() mitverwendet. Das sollte man nie vergessen, da es ja auch sein kann, dass der Suchstring NICHT gefunden wird.

So. Das Hauptprogramm zeigt schon, wie wir nun ganz elegant an die Systemvariablen rankommt, auch z.B. an das temporäre Verzeichnis des Rechners. Wir müssen nun nur noch aus der richtigen Zeile des Variablenwert extrahieren. Das ist ziemlich einfach. Wir ermitteln mit INSTR() die Position des Gleichheitszeichens und nehmen dann den String ab einem Zeichen hinter dieser Position:

```
winpath$=MID$(rueck$, INSTR(rueck$, "=")+1, len(rueck$))
```

Die Zeile ist allerdings ein bisschen unübersichtlich. "Sprechender" wäre es, sie in mehrere Zeilen aufzuteilen:

```
posstart=INSTR(rueck$, "=")+1
IF (posstart=1) THEN STOP 'Hups! Kein Gleichheitszeichen!
posend=len(rueck$)
winpath$=MID$(rueck$, posstart, posend)
```

## Das ganze Programm: Kopieren eines Files ins Windowsverzeichnis

Der Rest ist nun einfach. Nehmen wir an, wir wollen das File von Diskette ins Windowsverzeichnis kopieren. Dann bilden wir den ganzen DOS-Befehl als String: cmd\$="copy A:\"+fname\$+" "+winpath\$ und übergeben ihn der SHELL: SHELL(cmd\$). fname\$ enthält den Filenamen, z.B. "myprog.exe". Und das war's dann.

## Was haben wir gelernt?

- Was man mit dem Lesen von Textdateien aus Programmen heraus so machen kann. Die Kombination von DOS-Befehl, Dateiumleitung und Fileauslesen in QBASIC lässt sich auch bezüglich vieler anderer Befehle verwenden, vor allem solchen, die mit dem Netzwerk zu tun haben: ping, nslookup, ipconfig, tracert usw. Die net-Befehle geben ja sehr viele Informationen über das NT-Netzwerk in die Konsole aus und mittels der gelernten Technik können wir sie dann in QBASIC benutzen.
- Ein gewichtiger Vorwand gegen diese Technik könnte sein: Sie ist stark von der

Windowsversion abhängig, da sich unter den verschiedenen Windowsversionen die Ausgaben der genannten Befehle im Format teilweise ziemlich unterscheiden. Das stimmt. Aber der Effekt ist bei unserem Programm nicht besonders tragisch, da wir ja nicht bestimmte Zahlen von Zeilen und Spalten annehmen, sondern immer nach Schlüsselworten suchen. Solange die Schlüsselworte aus den Bildschirmausgaben nicht verschwinden, kann nicht soviel schief gehen.

Es gibt allerdings auch Fälle, wo der Einwand voll zutrifft. Wenn Sie z.B. versuchen, den dir-Befehl umzuleiten, um auf diese Art und Weise eine Operation auf jede Datei im aktuellen Verzeichnis anzuwenden. Das können Sie gut machen, aber die Ausgabe des dir-Befehls ist sehr versionsabhängig. Hier kommen Sie nicht umhin, zuerst die Windowsversion abzufragen (Systemvariable!), um dann für jedes dir-Format eine extra Version ihrer dir-Leseroutine zu schreiben.

- Wir haben aber auch ein bisschen etwas zur allgemeinen Programmiertechnik dazugelernt: Kommentieren von Routinen, Aufsuchen von Ressourcen (schon programmierten Routinen), Aufbau von Programmen, usw.

## Übung:

1. Schreiben Sie ein Programm (einfache Version), das alle Dateien mit der Endung .txt oder .bas in einem beliebig zu wählenden Verzeichnis durchsucht und darin alle DOS-Umlaute in Windows-Umlaute umsetzt - oder umgekehrt. "Einfache Version" heisst: Nur für Ihre Windows-Version. Um zu wissen, welchen Code in DOS- und Windows-Programme jeweils haben, müssen Sie erstmal zwei Auswertungen machen: Einfach jeweils mit einem Windows- und einem DOS-Editor die Umlaute schreiben, abspeichern, mit QBASIC einlesen und mit der ASC()-Funktion schauen, welche Codes sich dahinter verbergen. Das Ergebnis ist dann die "Übersetzungstabelle".
2. Erweitern Sie das Programm so, dass es auch in den Unterverzeichnissen des angegebenen Verzeichnisses die Umwandlung vornimmt!
3. Erweitern Sie die Version so, dass es zuerst die Windowsversion ermittelt und dann die verschiedenen dir-Ausgabeformate der Windowsversionen berücksichtigt.

---

# Zeiger

---

Bevor wir zu den anderen Dateitypen (Binary und Random Access) kommen, müssen wir uns mit "Zeigern" beschäftigen. "Zeiger" sind ein Programmierkonzept, ähnlich wie Unterprogramme oder Arrays. Zeiger tauchen fast überall in der Computerwelt auf - im Zusammenhang mit höheren Programmiersprachen, mit Datenbanken, mit Webseiten. Denn die Links von Webseiten sind im Grunde genommen auch nichts anderes als Zeiger.

In höheren Programmiersprachen gibt es Zeiger auf Variablen als eingebautes Sprachelement. Skriptsprachen fehlt so etwas. QBASIC auch. (Das hat nicht nur Nachteile!) Aber wie gesagt: Zeiger sind ein Programmierkonzept und so kann man Zeigermechanismen auch in QBASIC-Programmen einsetzen. Und das wollen wir jetzt mal tun.

## Was sind Zeiger?

Zeiger sind Adressen von Daten. Sozusagen ihre Ortsangabe. Ein Verweis darauf (daher ist ein Weblink auch nichts anderes als ein Zeiger). Wir können in der Websprache sagen: Wir befassen uns mit "Links". Nur dass diese Links auf Daten und nicht auf Webseiten verweisen.

Eine ganz einfache Form eines Zeigers können wir uns verschaffen, wenn wir Arrays benutzen. Nehmen wir an, wir verwalten eine Postadressentabelle der Form DIM adr\$(N,3). adr\$(i,0) ist der Name, adr\$(i,1) ist die Strasse und adr\$(i,2) ist der Ort der i-ten Postadresse, die wir abspeichern. Nehmen wir nun an, wir wollen ein Unterprogramm schreiben, dass alle Einträge einer Postadresse, also Name, Strasse und Ort in Grossbuchstaben umwandelt. Es gibt nun zwei Möglichkeiten, dem Unterprogramm seine Strings zu übergeben.

1. Wir kopieren dem Unterprogramm die Postadresseinträge in extra Stringvariablen. Also

```
nam$=adr$(i,0)
strass$=adr$(i,1)
ort$=adr$(i,2)
gosub grossbuch
adr$(i,0)=nam$
adr$(i,1)=strass$
adr$(i,2)=ort$
```

Das nennt man eine Übergabe *by value*. Wir übergeben die Werte selbst in eigene Bearbeitungsvariablen. Ändert die Routine etwas an den Bearbeitungsvariablen, müssen wir den Inhalt der Bearbeitungsvariablen erst noch explizit wieder zurück in die eigentlichen Variablen zurückkopieren.

2. Wir übergeben dem Unterprogramm einfach den Index i. i ist dabei selbst so etwas wie eine Hausnummer, die der Routine sagt: "Bitte, hier steht der Postadresseneintrag, mit dem du arbeiten sollst!" Eben ein *Zeiger* auf den richtigen Arrayeintrag. Wir brauchen also nur eine Übergabevariable, nämlich i0 für den zu bearbeitenden Index i:

```
i0=i
gosub grossbuch
```

grossbuch nimmt dann den Eintrag mit Index i0 und bearbeitet ihn. Das nennt man eine Übergabe *by reference*. Die "Referenz" ist praktisch der Zeiger. Der Nachteil dieses



Verfahrens: Der Code von grossbuch muss `adr$()` kennen, ist also nicht universal auf drei Strings anwendbar.

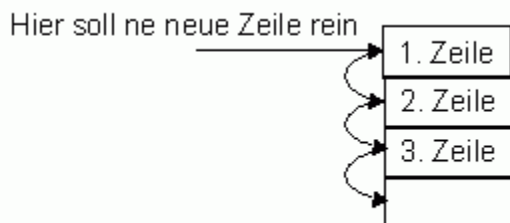
Das Hantieren mit Zeigern u.a. dann von grossem Interesse, wenn man es mit grossen Datenstrukturen zu tun hat und nicht bei jeder Gelegenheit den Inhalt der ganzen Datenstruktur (oder Teile davon) übergeben können, wie wir das mit einzelnen Variablen tun.

Das beste Übungsbeispiel ist die Programmierung eines Editors. Überlegen Sie sich einmal, wie Sie das machen würden!

## Programmierung eines Editors

Einen Editor kann man schon dann gut gebrauchen, wenn man in einem eigenen Programm den Benutzer einen eigenen kleinen Text in ein Feld auf dem Bildschirm eintragen lassen will. Eigentlich ganz easy. Der Text ist natürlich ein String-Array. Die Cursorbewegung bedeutet einfach, die Position innerhalb des Arrays (Zeile) und innerhalb des Strings (Spalte) zu ändern. Sehr schön. Und nun fügen wir am Anfang eine neue Zeile ein...

Merken Sie was? Wenn der User anfängt, irgendwo mitten im Text Zeilen einzufügen, dann muss jedesmal, wenn er einen Zeilenwechsel macht, der Inhalt des gesamten Arrays nach hinten verschoben werden! Damit vorne wieder eine Zeile Platz hat. Das ist natürlich schlecht. Weil das dauert und der User auf seinen zeilenwechsel nicht eine halbe Minute warten müssen soll. Ich hatte mal einen Mini-PC, bei dessen Editor das so war. Der Editor war nur für Texte bis zu 5K zu gebrauchen, also 2, 3 DinA4-Seiten. Danach wurde er bei jedem Zeilenwechsel so langsam, dass man besser eine neue Datei anlegte...



Sie können sich überlegen, wie Sie das lösen wollen. Wenn Sie eine saubere Lösung wollen und lange genug darüber nachgedacht haben, werden Sie schliesslich auf folgenden Weg kommen:

- Das Umsortieren der eigentlichen Strings dauert und muss deshalb verhindert werden.
- Daher darf die Reihenfolge der Strings im Array nichts mit der Reihenfolge der Zeilenfolge auf dem Bildschirm zu tun haben dürfen.
- Also brauchen wir eine zusätzliche Ordnungstabelle, die besagt, welcher Arrayindex gerade welche Zeile darstellen muss.
- Nennen wir diese Ordnungstabelle "Indextabelle". Wenn wir die Reihenfolge der Zeilen neu ordnen müssen, läuft das auf eine Neuordnung der Indextabelle hinaus. Hier müssen wir nur ein paar Integers verschieben - geht viel schneller als Strings!

l	z
8	3
9	4
10	5
0	6
1	7
2	8
11	9
12	10
13	11
14	12
3	13
4	14
5	15

*Beispiel einer Indextabelle: Die rechte Spalte gibt die Indexnummer im String-Array an, die linke Nummer die Zeilennummer, an der der String erscheinen soll. Finden Sie heraus, wie der Text zustande kam! Welche Zeilen hat der User zuerst eingegeben, wo hat er dann Text eingefügt?*

Man könnte die Indextabelle natürlich auch andersherum sortieren: Die Zeilen in der richtigen Reihenfolge und die Arrayindizes entsprechend umsortiert. Das kommt auf das gleiche heraus. Wichtig ist nur: Die Indizes stellen Zeiger auf den eigentlichen Speicherinhalt, die Strings dar.

**Übung:** Schreiben Sie einen kleinen Editor für QBASIC mit bis zu 255 Zeichen Breite und einem schnellen Zeilenmanagement!

# Binäre Dateien

## Hex-Editor

Weiter vorne haben wir Texte, d.h. Strings in Dateien geschrieben. Gewöhnlich enthalten Strings nur ASCII-Zeichen, also Zeichen, die mit Zahlen zwischen 32 und 127. Was in Wirklichkeit in die Datei geschrieben wird, sind diese Zahlenwerte. Dies können Sie leicht überprüfen, und zwar mit einem s.g. *Hex-Editor*. Den werden wir im Folgenden zur Kontrolle des öfteren brauchen.

Ein Hex-Editor ist ein Editor, der die Zahlenwerte einer Datei nicht als Zeichen anzeigt (was ja nur bei Textdateien sinnvoll ist), sondern als Zahlenwerte eben. Und da er das meist im Hexadezimalsystem macht, nennt er sich Hex-Editor. Einen solchen Editor (bzw. Lister) finden Sie oft im Verbund mit komfortablen Editoren oder Dateimanagern. Ich empfehle Ihnen, den [Totalcommander](#) herunterzuladen, ein nicht ganz kostenloser, aber sehr günstiger und mächtiger Dateimanager. Sie müssen ihn nicht gleich bezahlen; er erinnert Sie lediglich bei jedem Programmstart daran. Wenn Sie ihn öffnen, haben Sie zwei Verzeichnisse vor sich. Wenn Sie auf eine Datei gehen und dann "F3" drücken, dann erhalten Sie eine Ansicht der Datei in einem Lister. Und wenn Sie nun hier im Menü auf "optionen" und "Hexadezimal" gehen, dann sehen Sie etwas, das so aussieht:

Lister - [N:\priv\comp\tutorial\KAP2.HTM]																	
Datei	Bearbeiten				Optionen				Hilfe								
00000000:	3C	48	54	4D	4C	3E	0D	0A	3C	48	45	41	44	3E	0D	0A	<HTML>00<HEAD>00
00000010:	20	20	20	3C	4D	45	54	41	20	48	54	54	50	2D	45	51	<META HTTP-EQ
00000020:	55	49	56	3D	22	43	6F	6E	74	65	6E	74	2D	54	79	70	UIV="Content-Type
00000030:	65	22	20	43	4F	4E	54	45	4E	54	3D	22	74	65	78	74	e" CONTENT="text
00000040:	2F	68	74	6D	6C	3B	20	63	68	61	72	73	65	74	3D	69	/html; charset=i
00000050:	73	6F	2D	38	38	35	39	2D	31	22	3E	0D	0A	20	20	20	so-8859-1">00
00000060:	3C	4D	45	54	41	20	4E	41	4D	45	3D	22	41	75	74	68	<META NAME="Auth
00000070:	6F	72	22	20	43	4F	4E	54	45	4E	54	3D	22	43	68	72	or" CONTENT="Chr
00000080:	69	73	74	6F	66	20	53	63	68	61	74	7A	22	3E	0D	0A	istof Schatz">00
00000090:	20	20	20	3C	54	49	54	4C	45	3E	45	69	6E	66	61	63	<TITLE>Einfac
000000A0:	68	65	73	20	50	72	6F	67	72	61	6D	6D	69	65	72	65	hes Programmiere
000000B0:	6E	3C	2F	54	49	54	4C	45	3E	0D	0A	3C	2F	48	45	41	n</TITLE>00</HEA
000000C0:	44	3E	0D	0A	3C	42	4F	44	59	3E	0D	0A	3C	74	61	62	D>00<BODY>00<tab
000000D0:	6C	65	20	77	69	64	74	68	3D	22	31	30	30	25	22	3E	le width="100%">
000000E0:	0D	0A	3C	74	72	3E	0D	0A	3C	74	64	20	77	69	64	74	00<tr>00<td widt
000000F0:	68	3D	22	34	30	25	22	20	61	6C	69	67	6E	3D	63	65	h="40%" align=ce
00000100:	6E	74	65	72	3E	0D	0A	3C	2F	74	64	3E	0D	0A	3C	74	nter>00</td>00<t
00000110:	64	20	77	69	64	74	68	3D	22	31	30	25	22	20	61	6C	d width="10%" al
00000120:	69	67	6E	3D	63	65	6E	74	65	72	3E	0D	0A	3C	61	20	ign=center>00<a
00000130:	68	72	65	66	3D	22	6B	61	70	31	2E	68	74	6D	22	3E	href="kap1.htm">
00000140:	3C	69	6D	67	20	73	72	63	3D	22	70	72	65	76	32	2E	</
00000160:	61	3E	0D	0A	3C	2F	74	64	3E	0D	0A	3C	74	64	20	77	a>00</td>00<td w
00000170:	69	64	74	68	3D	22	31	30	25	22	20	61	6C	69	67	6E	idth="10%" align
00000180:	3D	63	65	6E	74	65	72	3E	0D	0A	3C	61	20	68	72	65	=center>00<a hre
00000190:	66	3D	22	6B	61	70	33	2E	68	74	6D	22	3E	3C	69	6D	f="kap3.htm"><im
000001A0:	67	20	73	72	63	3D	22	6E	65	78	74	32	2E	67	69	66	g src="next2.gif
000001B0:	22	20	62	6F	72	64	65	72	3D	30	3E	3C	2F	61	3E	0D	" border=0></a>
000001C0:	0A	3C	2F	74	64	3E	0D	0A	3C	74	64	20	77	69	64	74	0</td>00<td widt
000001D0:	68	3D	22	34	30	25	22	20	61	6C	69	67	6E	3D	63	65	h="40%" align=ce
000001E0:	6E	74	65	72	3E	0D	0A	3C	2F	74	64	3E	0D	0A	3C	2F	nter>00</td>00</
000001F0:	74	72	3E	0D	0A	3C	2F	74	61	62	6C	65	3E	0D	0A	3C	tr>00</table>00<

Ganz links vor dem Doppelpunkt ist die Adresse (Byteposition) im File angegeben. Jede Zeile umfasst in der Anzeige 16 Bytes. In der Mitte jeder Zeile werden die Zahlenwerte der 16 Bytes angegeben. Ganz rechts sieht man eine Zeichendarstellung der Bytes.

Auf diese Weise können wir im Folgenden auch dann in eine Datei "hineinschauen" und kontrollieren,

was wir gemacht haben, wenn die Datei mit einem Texteditor nicht lesbar ist. Am Ende dieses Kapitels sind wir in der Lage, unseren eigenen Hex-Editor zu schreiben!

## Erste binäre Versuche

Echte binäre Dateien enthalten nicht nur Bytes zwischen 32 und 127, sondern alle Werte zwischen 0 und 255. Schon mit den bisher kennengelernten Mitteln sollte es eigentlich kein Problem sein, solche beliebigen Bytes zu schreiben. Mit der CHR\$(j) Funktion können wir ja jede Zahl zwischen 0 und 255 in ein Char, ein logisches (wenn auch vielleicht nicht lesbares) "Zeichen" verwandeln. Entsprechend enthält das Programm

```
OPEN "a.dat" FOR OUTPUT AS #1
FOR i = 0 TO 255
  FOR j = 0 TO 255
    PRINT #1, CHR$(j);
  NEXT j
NEXT i
CLOSE 1
```

auch keine grossen Überraschungen. Wir schreiben einfach 256 mal alle Bytes zwischen 0 und 255 hintereinander in ein File. Das Semikolon am Ende der PRINT-Anweisung verhindert, dass uns eine Return-Sequenz (Hex 0D0A) dazwischen rutscht.

Hat das funktioniert? Schauen Sie sich's mit dem Hex-Editor an!

Lister - [r:\A.TXT]																																				
Datei		Bearbeiten					Optionen					Hilfe																								
00000000:	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	00000000000000000000000000000000																			
00000010:	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	00000000000000000000000000000000																			
00000020:	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	!"#\$%&'()*+,-./																			
00000030:	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	0123456789:;<=>?																			
00000040:	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	@ABCDEFGHIJKLMNO																			
00000050:	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	PQRSTUVWXYZ[\]^_																			
00000060:	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	`abcdefghijklmnopqrstuvwxyz																			
00000070:	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F	{ }~0																			
00000080:	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F	€0,f,....t†^%\$<00Z0																			
00000090:	90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F	0'.,;""•—~™\$>00ZŸ																			
000000A0:	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF	¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯																			
000000B0:	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF	° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾																			
000000C0:	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î																			
000000D0:	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF	Ë Ì Í Î Ï Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß																			
000000E0:	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF	à á â ã ä å æ ç è é ê ë ì í î ï																			
000000F0:	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF	ð ñ ò ó ô õ ö ÷ ø ù ú û ý þ ÿ																			

Sieht ganz gut aus.

Zur Kontrolle wollen wir das wieder einlesen. Aber wie? Klar, wir öffnen ein File zum Lesen ("FOR INPUT AS..."). Aber mit welcher Anweisung lesen wir die Bytes ein? INPUT? Dann bräuchten wir so etwas wie Zeilen. Aber die gibt es ja hier nicht.

Hier lernen wir eine neue Funktion kennen: **INPUT\$**. Syntax:

INPUT\$(n[, [#]Dateinummer%])

n Die Anzahl der zu lesenden Zeichen (Byte).

Dateinummer Die Nummer einer geöffneten Datei. Wenn Dateinummer weggelassen wird, liest INPUT\$ von der Tastatur.

Mit der INPUT\$-Funktion können wir also einfach eine feste Anzahl Bytes einlesen. INPUT\$(200,1) heisst: "Gib' mir die nächsten 200 Zeichen aus Datei 1." Ideal für unseren Zweck.

Überlegen Sie sich nun ein Programm, mit dem Sie alle 64K, die wir gerade rausgeschrieben haben, wieder einlesen!

Aber ärgern Sie sich nicht, wenn Sie dabei über die Meldung "Eingabe nach Dateiende" stolpern! Sondern lesen Sie gleich hier weiter:

Schauen wir diese Variante an:

```
OPEN "a.dat" FOR INPUT AS #1
OPEN "b.dat" FOR OUTPUT AS #2
FOR i = 0 TO 255
  a$ = INPUT$(256, 1)
  FOR j = 0 TO 255
    k1$ = MID$(a$, j + 1, 1)
    k = ASC(k1$)
    PRINT #2, i, j, k
  NEXT j
NEXT i
```

Beim INPUT\$ hagelt's die oben beschriebene Laufzeitfehlermeldung. Was heisst das? Nun, ganz einfach: Jede Datei hat am Ende ein spezielles Byte, das markiert, dass hier die Datei zuende ist. Wir haben ja schon die Funktion EOF() kennengelernt. Sie tut nichts anderes, als achtgeben darauf, ob dieses spezielle Byte gerade eingelesen wurde. Wenn ja, dann gibt sie TRUE zurück. Wir nennen das Byte daher EOF-Byte. Es handelt sich um den Wert Hex 1A, Dez 26. Wenn nun die Ausführung von INPUT\$ das 27. Bytes ausliest, dann erkennt es den EOF-Marker. INPUT\$ gibt aber die Anweisung, noch weitere Bytes auszulesen, also HINTER dem EOF-Zeichen. Und das ist natürlich verboten. Es sei denn, wir können dem Programm sagen: "Hör mal, liebes Programm, es GIBT KEIN EOF-Zeichen! Es gibt überhaupt keine besonderen Steuerzeichen. Vergiss es!" Und das machen wir, indem wir die Datei im BINARY-Modus öffnen:

```
OPEN "a.dat" FOR BINARY AS #1
```

Und schon funktioniert es und wir können eine makellose Liste in der Datei b.dat abrufen!

## Exkurs: BMP-Grafiken mit QBASIC erstellen

Hups? Was haben Grafiken mit binären Dateien zu tun? Nun, man kann Grafiken nicht nur mittels SCREEN- und LINE-Befehlen auf den Bildschirm zaubern, sondern auch dadurch, dass man Grafikdateien erstellt und diese dann mit einem Bildbetrachter anschaut. Grafikdateien sind Binärdateien - und damit wären wir beim Thema.

Allerdings müssen wir uns dazu mit den Formaten von Grafikdateien auseinandersetzen. Das nimmt ein bisschen Raum in Anspruch, daher [lagern wir das hier aus](#). Aber gönnen Sie sich den Spass, einmal ein selbstgerechnetes Fraktal mit 1200x1200 Punkten in True-Color anzuschauen!

## Random Access auf Binärdateien

"Random Access" heisst nicht "zufälliger Zugriff", sondern "wahlfreier Zugriff", man könnte auch sagen: "Gezielter Zugriff". Es ist das Gegenteil von sequentiellm Zugriff, bei dem immer nur das gerade gelesen werden, was gerade als Nächstes kommt. Beim sequentiellen Zugriff muss man sich die Datei vorstellen wie ein Tonband. Es gibt einen Dateizeiger. Dieser spielt die Rolle des Tonkopfes. Beim Lesen oder Schreiben wird immer ein kleines Stück gelesen, dann rückt der "Tonkopf", also der Dateizeiger vorwärts. Die einzige Beeinflussung auf die Position des Dateizeigers, des "Tonkopfes", können wir dadurch ausüben, dass wir die Datei schliessen und wieder öffnen und damit den Dateizeiger auf den Anfang der Datei verschieben.

Bei Random Access sieht das anders aus: Hier können wir den Dateizeiger auf jedes Byte in der Datei

schieben. Wir können also im Grunde auf die Datei genauso zugreifen wie auf den Hauptspeicher. Dazu müssen wir nur sagen, bei welcher Adresse wir ein Byte aus der Datei lesen oder an welche Adresse wir ein Byte in die Datei schreiben wollen. Und wir müssen höllisch aufpassen, dass die Adresse nicht ausserhalb des Files liegt, also vor Beginn oder nach dem Ende der Datei!

## Erstellen der Swap-Datei

Wenn wir bytewise auf eine Datei zugreifen wollen, müssen wir uns das Ganze wie den Zugriff auf einen Arbeitsspeicher vorstellen. Und damit man auf einen solchen Speicher zugreifen kann, muss er physisch erst einmal existieren: Wir müssen die Datei ersteinmal erstellen. Das machen wir am Besten mit dem altbekannten OPEN FOR OUTPUT-Befehl. Dann schreiben wir eine Anzahl Bytes. Und schliessen die Datei wieder. Diese Datei stellt nun unser "Arbeitsspeicher" da. Man bezeichnet so einen ausgelagerten Speicher übrigens als "Swap"-Speicher oder einfach als "Swap".

## Zugriff auf die Swap-Datei

Um bytewise auf die Swap-Datei zuzugreifen, benutzen wir die Befehle GET und PUT und den Zugriffsmodus BINARY: OPEN FOR BINARY AS #1. Dann schreiben wir mit PUT in die Datei: PUT #,, "nr" ist die Filenummer. "adresse" ist eine Bytezahl. Ist die Datei z.B. 143 Bytes gross, kann die Adresse 1 bis 143 betragen. (Leider nicht 0 bis 142. An sich zählt man in der Informatik immer von null ab, aber Microsoft hält sich nicht gerne an Konventionen...). "variable" ist eine String-Variable, die das zu schreibende Byte enthält.

Bei GET ist es entsprechend, nur dass die String-Variable angibt, in der das zu schreibende Byte gespeichert werden soll.

So, nun ein kleines Beispielprogramm:

```
OPEN "R:\a.txt" FOR OUTPUT AS #1
FOR i = 0 TO 255
  a$ = CHR$(i)
  PRINT #1, a$;
NEXT i
CLOSE (1)
OPEN "R:\a.txt" FOR BINARY AS #1
FOR i = 1 TO 25
  j = i * 10
  GET #1, j + 1, a$
  PRINT j, ASC(a$)
NEXT i
CLOSE (1)
```

Dieses Programmchen schreibt wie vorhin 256 Bytes in die Swap-Datei. Dann wird diese binär geöffnet und wir lesen nun von jeder 10. Adresse ein Byte aus.

## Wozu kann man Swap-Dateien benutzen?

In der modernen Software-Entwicklung haben Swap-Dateien nicht mehr so grosse Bedeutung wie früher, da es ja locker möglich ist, viele hundert Megabyte im Hauptspeicher zu halten. Aber es gibt sie schon noch: In der Messdatenverarbeitung und Statistik oder in der Video- und Audiotbearbeitung.

## Anwendungsbeispiel 1: Maps

QBasic selbst erlaubt nur die Verwendung von bis zu 160K Hauptspeicher und jede Variable, also jedes Array oder jeder String darf nur maximal 64K gross werden. In Spielen werden aber oft viele Grafiken gebraucht, insbesondere oft s.g. "Maps", Landkarten, auf denen sich der Spieler bewegen kann. Der Spieler sieht immer nur einen Ausschnitt, den er dann über die eigentliche Karte bewegt. Die Karte selbst

kann viele Tausend mal viele Tausend Pixel enthalten. Es ist nun problemlos möglich, diese Map in einer Swap-Datei zu speichern und mittels GET und PUT immer genau den Bildausschnitt in den Hauptspeicher zu laden, den der Spieler gerade sieht. Wir wollen das hier nicht weiter verfolgen, aber die folgende Variante des obigen Beispielprogrammchens zeigt, wie man mit QBASIC eine 4 MB grosse Swap-Datei erstellen und benutzen kann:

```
DIM i AS LONG

OPEN "R:\a.txt" FOR OUTPUT AS #1
FOR i& = 0 TO 4* 1024& * 1024-1
  a$ = CHR$(RND(255))
  PRINT #1, a$;
NEXT i
CLOSE (1)

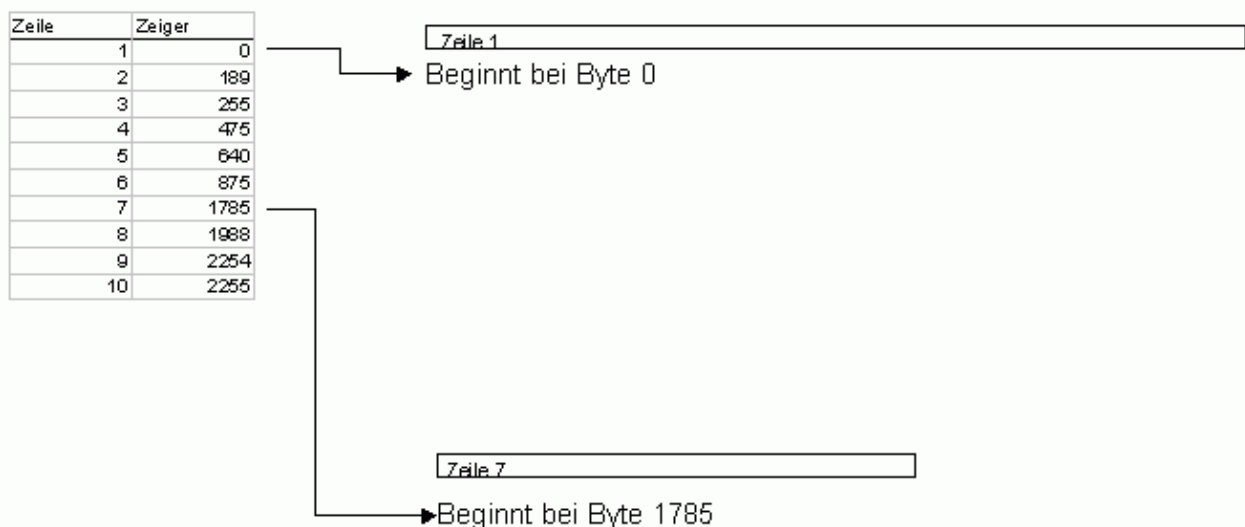
OPEN "R:\a.txt" FOR BINARY AS #1
FOR i = 1 TO 25
  j = i * 10
  GET #1, j + 1, a$
  PRINT j, ASC(a$)
NEXT i
CLOSE (1)
```

Das Besondere an dem Programmchen ist die erste Zeile mit dem "DIM" und die "&"-Zeichen hinter den Zahlen in der dritten Zeile. Nun, beides sorgt dafür, dass für die Variable i nicht nur 2 Bytes Speicher vorgesehen werden, sondern 4 Bytes, was dann einen Zahlenraum von 0 bis rund 4 Mrd. (4 GB) erlaubt und nicht nur 0 bis 65000, also 0 bis 64K. Wir werden auf diese "Typen" und "Deklarationen" später noch ausführlich zu sprechen kommen.

## Anwendungsbeispiel 2/Übung: Textdateien im wahlfreien Zugriff

Wir waren das Problem schon einmal weiter vorne angegangen: Als wir im [vorigen Kapitel](#) dem "schnellen Zeilenmanagement" begegneten: Wie schreibe ich einen Editor, der schnell zwischen die 2000. und 2001. Zeile noch eine einfügen kann? Und dabei nicht Megabyteweise Daten verschieben muss? Die Antwort war: Wir brauchen einen Index, der Zeiger auf jede Zeile enthält. Damals waren die Zeiger Indizes eines Arrays. Jetzt können wir als Zeiger die Byteposition in einer Datei hernehmen. Die Zeiger sammeln wir in einem eigenen Array, dem Index. Hier können wir nachschauen, an welcher Byteposition Zeile 0,1,7 oder 134 beginnt. Die folgende Grafik zeigt das Schema.

Index



Auf diese Art und Weise können wir leicht einen Editor schreiben, der so grosse Texte verwaltet wie der Hauptspeicher gross ist! Machen Sie sich ans Werk!



---

# Einfache und zusammengesetzte Datentypen

---

## Strukturierte Programmierung

Der Begriff "Strukturierte Programmierung" wurde hier und da schon fallengelassen, ohne zu erklären, was überhaupt damit gemeint ist. Wir werden uns hier auch noch nicht richtig damit auseinandersetzen, da QBASIC nur begrenzt die entsprechenden Möglichkeiten aufweist. Aber trotzdem hier schon ein kurzes Wort dazu.

Wir unterscheiden heute im Groben drei "Niveaus" oder "Stile" der Programmierung.

- Die "einfache" Programmierung,
- die "strukturierte Programmierung"
- und die "objektorientierte Programmierung".

Wobei vielfach darauf hingewiesen wird, dass es eigentlich noch eine Zwischenebene zwischen der zweiten und dritten gibt, nämlich die "objektverarbeitende Programmierung".

Beim Anstieg von einem Niveau auf's nächste geht's darum, komplexere Programme schreiben zu können. Vielleicht hat Sie dieser Kurs (oder ein anderer) schon ermutigt und motiviert, einmal ein richtig grosses Programm zu schreiben; eins, das nicht nur zwanzig oder hundert Zeilen hat, sondern, sagen wir zweitausend. Und selbst das ist noch ein vergleichsweise kleines Programm; es gibt Programme, die haben zehn Millionen Zeilen und mehr! Aber schon bei ein, zweitausend Zeilen BASIC werden Sie festgestellt haben, dass es immer schwerer wird, ein fehlerfreies Programm zu bekommen, je mehr Zeilen es hat. Da müssen Sie Stunden und Tage nach einem Fehler suchen. "Warum springt das verflixte Programm jetzt in diese Zeile?" werden Sie sich oftmals gefragt haben. Oder: "Warum hat jetzt die Variable `sum_abc1` null?" Insofern ist es zwar theoretisch möglich, auch ein 100.000-Zeilen-Programm in BASIC zu schreiben, aber man wird sehr, sehr, sehr lange Zeit dafür brauchen.

Schon früh in der Computerentwicklung hat man gemerkt, dass man sehr viel Zeit sparen kann, wenn man Mittel an der Hand hat, das Programm strukturieren zu können. *Kommentare* sind ein solches Mittel. Sie werden vermutlich auch jetzt noch viel zu wenig kommentieren. Ein Kommentar, der in 3 Minuten geschrieben ist, spart im Mittel eine halbe bis eine Stunde Entwicklungszeit ein!

Solche Strukturierungsmittel wie Kommentare, GOTO-freie Programmierung, If-Blöcke usw. sind es, die die Produktivität eines Programmierers vervielfachen können, wenn er damit umgehen kann. Sie sind es auch, die einen einfachen Programmierer von einem erfahrenen Informatiker unterscheidet. Der Informatiker ist (auch) Fachmann für Programmstrukturierung, nicht nur für praktische Programmierung

Ein Tipp an dieser Stelle: Wenn Sie vorhaben sollten, ein Programm von tausend Zeilen oder mehr zu erstellen, dann warten Sie damit noch ein bisschen. Sie können das zwar jetzt schon, aber Sie vertun dabei viel Zeit. Mit den in den folgenden Kapiteln eingeführten Mitteln der strukturierten Programmierung sind Sie viel schneller und haben Sie viel mehr Freude daran.

## Deklarieren

Wir haben inzwischen gelernt, dass der Computer für alle Variablen Speicherplatz benötigt. Wir haben uns aber noch nicht darum gekümmert, wie der Computer erfährt, wieviel Speicherplatz er eigentlich braucht. Am Anfang unserer Programmiersuche wäre die Sorge auch störend gewesen. BASIC ist eine Sprache, die extra für Anfänger entworfen wurde. Und daher nimmt es dem Lernenden erstmal diese Sorge ab. Man schreibt einfach eine Zuweisung hin "x=1+1" und BASIC weiss: "x" ist eine Zahlvariable. Und reserviert die benötigte Anzahl von Bytes. Falls es auf 'x\$="1+1"' trifft, dann weiss es: "x" ist eine Stringvariable. Usw.

Bei Arrays verhielt es sich allerdings anders. Diese muss man *deklarieren*. Das heisst, man muss dem Computer sagen, dass man nun die Variable x benutzen will und welche Grösse x haben soll, z.B. DIM x(100).

Da ein Array in Wirklichkeit *zusammengesetzt* ist aus verschiedenen einzelnen Zahlvariablen, gehören Arrays zu den s.g. "zusammengesetzten Datentypen". Zusammengesetzte Datentypen muss man auch in BASIC deklarieren.

## Automatisches und manuelles Deklarieren

Nun gibt es verschiedene Gründe, warum diese "Deklarationsautomatik" bei einfachen Datentypen für grösste Programme Nachteile bringen kann:

1. Schreibfehler wirken sich z.T. verheerend aus. Wahrscheinlich ist Ihnen das auch schon passiert: Sie wollen die Variable XCOORD verwenden, und schreiben stattdessen an irgendeiner Stelle XCORD. BASIC denkt sich: "Hups! Eine neue Variable!", deklariert sie und initialisiert sie mit null, d.h. gibt ihr den Anfangswert null. Inmitten einer komplizierten Formel fällt der Schreibfehler aber nicht auf. Man stellt nur fest, dass das Formelergebnis nicht stimmt. Dass da plötzlich eine ganz neue Variable auftauchen könnte, darauf kommt man nicht. Und man rechnet und rechnet und grübelt und grübelt und ist am Schluss ziemlich verzweifelt.
2. Geschwindigkeit. Das ist heute kein so grosses Problem mehr wie früher, aber es spielt schon noch eine Rolle. Wenn BASIC automatisch deklariert, dann wird jede Zahlvariable als Flieskommazahl deklariert. Mit dem Ergebnis, dass jede Rechenoperation eine Flieskommaoperation ist. Früher waren diese zehn- oder zwanzigmal langsamer als Ganzzahloperationen. Die Ganzzahlrechnungen hatte der Prozessor ja eingebaut, die Flieskommaoperationen nicht. Heute ist das anders, aber trotzdem sind Ganzzahloperationen noch schneller. Also wird man interessiert daran sein, BASIC zu sagen: "x ist eine Ganzzahl, keine Flieskommazahl!". Am interessantesten ist dies bei FOR-Schleifen. Es macht überhaupt keinen Sinn, FOR I=0 TO 99 mit I als Flieskommazahl durchzuführen. Überlassen wir das Deklarieren aber vollständig BASIC, dann macht es genau das.
3. Ungenaue Vergleiche: Wenn x und y Flieskommazahlen sind und wir schreiben IF (x=y) THEN... hin und wir gehen davon aus, dass in x und y nur Ganzzahlen sind, dann kann das in die Hose gehen. Das sieht man am deutlichsten bei  $x = (((((x/2)*2)/2)*2)/2)*2$ . Sind wir mit x=1 gestartet, ist nun x=0.999999 oder x=1.00001. Wir haben Rundungsfehler. Und wenn wir nun den o.g. Vergleich durchführen, arbeitet dieser nicht so, wie wir uns das gedacht haben. Auch diesen Fehler werden wir ziemlich lange nicht entdecken!
4. Kleines Argument, aber es sei hier erwähnt: Speicherplatz. Nehmen wir an, wir wollen eine Karte, eine "Map" mit 200 x 100 Punkten erstellen. Jeder Punkt soll eine von 16 Farben haben. Wenn wir einfach nur DIM map(200,100) deklarieren, hagelt es eine Fehlermeldung. Warum? In QBASIC kann man Arrays nur bis max. 64K Grösse benutzen. Eine einfache

Fliesskommazahl, die QBASIC für automatisch deklarierte Zahlvariablen hernimmt, benötigt 4 Bytes.  $4 \times 200 \times 100 = 80000 > 64K$ . Wir bräuchten aber nur Ganzzahlen, sogar eigentlich nur ein Byte pro Punkt. Nur 1 Byte für eine Ganzzahl herzunehmen, das sieht QBASIC nicht vor, aber 2 Byte geht. Dieser Datentyp heisst "INTEGER". Deklarieren wir DIM x(200,100) as INTEGER, dann geht's!

Gründe genug, sich anzugewöhnen, bei grösseren Programmen auf die automatische Deklaration zu verzichten. Variablen explizit zu deklarieren, auch das ist ein Element der strukturierten Programmierung.

## Einfache Datentypen

In QBASIC unterscheiden wir folgende einfache Datentypen:

- INTEGER: Ganzzahl zwischen -32767 bis +32766, 2 Bytes. Spezielles Zeichen: %
- LONG: Ganzzahl zwischen  $-2^{32}+1$  (ca. -2 Mrd.) und  $+2^{32}-2$  (ca. + 2 Mrd.), 4 bytes. Spezielles Zeichen: &
- SINGLE: Fliesskommazahl, ca. 5 bis 6 valide Stellen Mantisse, 2 Stellen Exponent; es sind Zahlbeträge bis ca.  $1E038$  möglich. Speicherplatz: 4 Bytes. Zeichen: !
- DOUBLE: Fliesskommazahl, ca. 12 valide Stellen Mantisse, 3 Stellen Exponent. Speicherplatz: 8 Bytes. zeichen: #
- STRING: Zeichenkette, bis 32K Zeichen, Speicherplatz: 1 Byte/Zeichen. Spezielles Zeichen: \$

## Typspezifizierer

Mit dem "speziellen Zeichen" hat es folgendes auf sich: Man kann in QBASIC den Datentypus auch ohne manuelle Deklaration zuweisen, indem man dem Variablennamen das entsprechende "spezielle Zeichen" (Typspezifizierer) anhängt. Wir kennen das schon von den Strings mit dem \$-Zeichen. Das geht aber mit den anderen Typspezifizierern genauso. Beispiel:

```
'Kurzes Programm
sum&=0
FOR i%=0 TO 32766
    sum&=sum&+i%
    PRINT sum&
NEXT i%
```

Die letzten Zeilen der Ausgabe lauten:

32755	536461390
32756	536494146
32757	536526903
32758	536559661
32759	536592420
32760	536625180
32761	536657941
32762	536690703
32763	536723466
32764	536756230
32765	536788995
32766	536821761

Wenn Sie die FOR-Schleifengrenze auf 32767 erhöhen, erhalten Sie die Fehlermeldung "Überlauf". Die Sie übrigens erst dann wieder wegstreichen, wenn Sie im Menü auf Ausführen->Neustart gehen.

Ein Integer kann eben die Zahl 32767 nicht speichern. Hingegen kann sum, das als LONG spezifiziert ist, problemlos 536 Millionen und noch ein paar Zerquetschte speichern.

## Manuelles Deklarieren einfacher Datentypen

Der schöne und empfehlenswertere Weg ist allerdings die manuelle Deklaration. Die Syntax dazu ist: **DIM <Variable> AS <Typ>** . Wobei anstelle von <Typ> einer der fünf einfachen Datentypen INTEGER, LONG, SINGLE, DOUBLE oder STRING zu setzen ist.

Unser kurzes Programm mit manueller Deklaration:

```
'Kurzes Programm
DIM i AS INTEGER
DIM sum AS LONG
sum=0
FOR i=0 TO 32766
    sum=sum+i
    PRINT sum
NEXT i
```

Diese Form macht das Programm auch lesbarer, da die Variablennamen nicht durch Sonderzeichen verunziert werden.

Allerdings hat man damit die "Unsitte" (so ganz eine Unsitte ist das auch wieder nicht; wir kommen noch drauf zurück,) der automatischen Deklaration immer noch nicht abgeschaltet. Wir haben nur einige Variablen manuell deklariert. Wir können weiterhin einfach mitten im Programmtext aus Versehen oder mit Absicht neue Variablennamen erwähnen und BASIC beschwert sich nicht. Bei QBASIC können wir das leider überhaupt nicht abschalten. Bei anderen BASIC-Varianten (z.B. Visual Basic oder Visual Basic for Application (VBA)) geht das aber. Die Anweisung gibt man ganz am Anfang des Programms und sie heisst: "OPTION EXPLICIT". Das nur so nebenbei.

## Noch ein Wort zum "loose typing"

Das, was wir "automatisch Deklarieren" nannte, nennt man in der Fachsprache "loose typing", also lose Typisierung. Was gleichzeitig dann auch "loose declaration" heisst. "Loose typing" ist keineswegs auf dem Rückzug. Vielmehr unterscheidet man heute zwischen *Skriptsprachen* und *Compilersprachen*. *Skripte* sind in der Regel kurze kleine Programme, um Systemverwaltung, Anwendungsverwaltung oder Webverwaltung zu erleichtern und zu automatisieren. Ein Techniker oder Verwalter, der solche Aufgaben zu erledigen hat, hat andere Sorgen, als sich den Typ von Schleifenvariablen zu überlegen. Daher erlauben solche Skriptsprachen das "loose typing". Compilersprachen hingegen werden meist für grössere Entwicklungsprojekte mit vielen tausend Zeilen benutzt. Hier ist loose typing fast überall verboten (z.B. Java, C, C++, C#, Delphi usw.). BASIC ist so ein Zwitter mitten drin. Es gibt BASIC-Skriptsprachen, in jedem modernen Windows-Betriebssystem ist Visual BASIC Script (VBS) eingebaut. Und es gibt BASIC als Compilersprache, z.B. in Form von Gambas, Visual Basic, Freebasic, Purebasic usw. Hier sollte man auf eine strenge Deklarations- und Typisierungspraxis achten.

## Strukturvariablen

Nun kommen wir zu einem äusserst wichtigen neuen Gestaltungsmittel: Strukturvariablen. Es ist deshalb so wichtig, weil es nicht nur in der strukturierten Programmierung eine grosse Rolle spielt, sondern sozusagen die Keimzelle der objektorientierten Programmierung in sich birgt. Das sei hier gleich vorweggenommen.

Was Strukturvariablen sind, ist zum Glück ganz einfach erklärt: Mehrere einfache Variablen werden zu einer "Obervariablen" zusammengefasst. Z.B. bilden Name, Strasse, Hausnummer, Postleitzahl und Wohnort zusammen eine "Adresse". Es wäre nun schön, wenn wir einen Datentyp "Adresse" zur Verfügung hätten, in dem all diese Angaben gespeichert werden könnten. Mittels Strukturvariablen kein Problem. In QBASIC gibt es dazu das Schlüsselwort **TYPE**. Am Besten zuerst ein Beispiel:

```
TYPE adresstyp
  name1 AS STRING * 40
  strasse AS STRING * 40
  hausnr AS INTEGER
  plz AS LONG
  wohnort AS STRING * 40
END TYPE

DIM adressen(100) AS adresstyp
DIM i AS INTEGER
DIM answer AS STRING

i=0
DO
  PRINT "Adresse Nr. "; i
  input "Name: "; adressen(i).name1
  input "Strasse: "; adressen(i).strasse
  input "hausnr: "; adressen(i).hausnr
  input "Postleitzahl: "; adressen(i).plz
  input "Wohnort: "; adressen(i).wohnort
  input "Noch eine Adresse? "; answer
LOOP UNTIL answer="no"
```

Zunächst wird mit TYPE der neue Datentyp definiert. Der Name des Datentyps ist "adresstyp". Merke: Nicht "Adresse". Man sollte immer zwischen den Datentypen und den Variablen selbst gut unterscheiden!.

Dann kommt in den nächsten Zeilen eine Aufzählung der Datentypen, die wir in "adresstyp" reinpacken wollen. Die Syntax ist eigentlich identisch zu der einer DIM-Deklaration, nur ohne das Schlüsselwort DIM. Gleich in der ersten Zeile begegnen wir allerdings einigem Ungewohnten. Erstens: Warum heisst es "name1" und nicht "name"? Und zweitens: Was hat das "\* 40" zu besagen?

Nun, NAME ist eine QBASIC-Anweisung und kann daher nicht für eigene Namen verwendet werden. Beim String besteht das Problem, dass wir - im Gegensatz zur Deklaration ausserhalb von TYPES - angeben müssen, wieviel Bytes denn für den String reserviert werden sollen. Ausserhalb von TYPE müssen wir das nicht, da dann QBASIC von selbst mehr Speicherplatz bereitstellen kann, wenn der String länger wird. Innerhalb von TYPES geht das nicht und wir müssen daher QBASIC gleich von Anfang an sagen, wie lange der String maximal werden soll.

Ansonsten dürfte alles klar sein. Für die Hausnummer reicht ein Integer, da sie selten grösser als 32766 sind. (Sie werden lachen: In Frankreich gibt's tatsächlich auch grössere Hausnummern, da für Häuser, die weit ausserhalb von Ortschaften stehen, oft die Distanz in Metern vom Ortskern als Hausnummer verwendet wird...)

Für die Postleitzahl brauchen wir seit der Postreform einen LONG. Beim Wohnort nehmen wir auch 40 Zeichen, dass solche Namen wie "Villingen-Schwenningen" auch ihren Platz finden.

Wir schliessen die Liste mit dem Schlüsselwort "END TYPE" ab. Damit ist unser neuer Datentyp fertig. Ab jetzt können wir ihn genauso verwenden wie die vordefinierten einfachen Datentypen

auch. Im Beispiel deklarieren wir gleich ein ganzes Array davon. Der Zugriff auf die einzelnen Elemente der Struktur erfolgt mit der Angabe eines Punktes und dann des Elementnamens. Also: <Obervariable>.<Element>

## Übung:

Modifizieren Sie das obige kleine Progrämmchen so, dass es die Adressdaten aus einer Textdatei ausliest, in der die Angaben einer Adresse jeweils durch Tabs oder Leerzeichen getrennt in einer Zeile stehen.

## Komplexere Datenstrukturen

Mithilfe von Strukturvariablen lassen sich ziemlich komplexe Datenstrukturen aufbauen und verwenden. Man kann nämlich

- auch Arrays
- auch Strukturvariablen
- auch Arrays von Strukturvariablen
- auch Strukturvariablen mit Arrays von Strukturvariablen
- usw.

als Elemente von Strukturvariablen verwenden.

Dies sei an einem Beispiel erläutert, das (hoffentlich) selbsterklärend ist:

```
TYPE punkttyp
  x as INTEGER
  y as INTEGER
  z as INTEGER
END TYPE

TYPE linientyp
  startpunkt AS punkttyp
  zielpunkt AS punkttyp
END TYPE

TYPE gittertyp
  linie(100) AS linientyp
END TYPE
```

Angenommen, wir wollen eine Wirtschaftssimulation schreiben. Zentrale Akteure in dieser Simulation sind Firmen. Jeder Firma kann mit einer anderen Firma interagieren. Jede Firma besteht in sich aber wiederum aus Angestellten und Maschinen. Die Maschinen stellen Produkte her. Es gibt Produkte verschiedener Typen. Die Angestellten wiederum treten auch wieder als Käufer der Produkte auf. Maschinen sind übrigens auch Produkte. Und zu all diesen aufgezählten Objekten: Firmen, Angestellten, Maschinen, Käufern, Produkten sind ihre Eigenschaften festzuhalten. Bei Produkten mag das noch einfach sein: Sie haben vielleicht nur einen Preis. Bei Maschinen ist das schon schwieriger: Welche Produkttypen stellt die Maschine her? Welche kann sie herstellen? Wieviel Produkte pro Stunde und Produkttyp werden hergestellt? Wieviel der Maschine nutzt sich dabei ab? Und was ist ihr Preis? Bei den Personen ist das noch vielfältiger: Welche Produkte wollen sie kaufen? Welchen Preis bei welchem Produkt akzeptieren sie? Bei welcher Firma arbeiten sie? An welcher Maschine arbeiten sie? Was verdienen sie dort? Und schliesslich die Firma: Welche Angestellte arbeiten in ihr? Welche Maschinen gehören dazu? Welche andere Firma stellt die Produkte her, die auch diese Firma herstellt? Wie gross sind ihre Einnahmen? Ihre Ausgaben?

## Übung:

Überlegen Sie sich einmal eine passende Datenstruktur mittels TYPE für diese Simulation! So dass ich als Programmierer jederzeit z.B. die Frage beantworten kann "An welchen Maschinen welcher Firmen arbeiten die Personen, die gerade Produkte mit einem Preis kleiner als x gekauft haben?"

Denken Sie auch einmal drüber nach, wie sie die Datenstruktur bilden würden, wenn Sie TYPE nicht zur Verfügung hätten!

---

# Typisierte Dateien und das Datenbankprinzip mit QBASIC

---

## Datentabellen

Wenn schon Strukturvariablen und Dateien für sich alleine jeweils ein sehr mächtiges Programmierwerkzeug darstellen, welches Instrument entsteht erst dann, wenn beide zusammen benutzt werden! Wenn wir dann noch das dritte neue Werkzeug, Zeiger, mit verwenden, dann haben wir das zusammen, was vom Prinzip her die Grundlage mehr als Dreiviertel aller heutigen professionellen Softwaresysteme darstellt: Datenbanken. Von der Datenbank, die Websites oft zugrundeliegt (z.B. MySQL) über Geschäftsdatenbanken (z.B. Oracle) bis zum Wirtschaftssystem a la SAP R/3.

Das Prinzip davon zu verstehen, davon sind wir gar nicht soweit entfernt wie Sie vielleicht denken. Dass wir mit QBASIC soweit kommen, liegt auch daran, dass BASIC in seiner Blütezeit auf PC's oft für kleinere Business-Programme und Datenbanken als Programmiersprache benutzt und von Microsoft, dem BASIC-Hersteller daher entsprechend aufgerüstet wurde.

Datenbanken lassen sich bei weitem nicht nur für Geschäftsanwendungen benutzen. Sie sind genauso in der Fotoarchivierung, in der Statistik oder in Spielen gut anwendbar. Aber zunächst einmal noch einen Schritt zurück zu typisierten Dateien.

Nehmen wir an, wir haben eine umfangreiche Datenstruktur erstellt, wie z.B. adresstyp aus dem vorigen Kapitel, aber adresstyp enthalte noch Tel-Nummer, Fax-Nummer, Email, Homepage und einiges mehr. Nun will ich alles in eine Datei rausschreiben. Natürlich kann ich das machen, indem ich jedes Element einzeln mit einer PRINT-Anweisung als Text rausschreibe. Für viele Zwecke mag das auch ganz nützlich sein, wenn ich die rausgeschriebenen Daten z.B. mit einer Tabellenkalkulation weiter bearbeiten möchte. Aber wenn ich die Daten einfach nur auf der Festplatte zwischenspeichern möchte, dann will ich nicht jedes einzelne Element meiner Struktur anfasseln müssen. Gibt es nicht eine Möglichkeit, einfach PRINT #1,adresse hinzuschreiben? Und dann schreibt BASIC alles raus, was zu adresse gehört? Gibt es. Ist auch nicht kompliziert.

## Datensätze

In so einem Fall schreibt BASIC ein Binary. Wobei dieses Binary in Blöcke fester Länge aufgeteilt wird. Jeder Block speichert eine Strukturvariable, also z.B. eine Adresse. BASIC muss lediglich wissen, wie gross so ein Block ist, wieviel Bytes er umfasst. Das ermitteln wir mit der Funktion LEN, genauso wie bei Strings. Haben wir z.B.

```
TYPE typbeispiel
  a as INTEGER
  b as INTEGER
END TYPE
```

so ist LEN(typbeispiel)=4.

Solche Blockteile von Dateien heissen in der EDV "Datensätze". Ein Datensatz ist immer der Inhalt eines "TYPES", ein Bündel aus Werten, das einer Informationseinheit zugeordnet wird, z.B. einer Person, einem Produkt, einem Zeitpunkt, einem Foto usw.

Jetzt öffnen wir die Datei, dieses mal mit dem Schlüsselwort "RANDOM". Aha, das kennen wir schon: "Random Access" = Wahlfreier Zugriff: OPEN "my.dat" FOR RANDOM AS #1. Wir bekommen also gleich noch den wahlfreien Zugriff auf die Datei als Beigabe.

Das reicht allerdings noch nicht. Nun müssen wir BASIC noch angeben, welche Grösse ein Datensatz hat. Das machen wir mit dem Schlüsselwort LEN ganz am Ende der OPEN-Anweisung:

```
OPEN "my.dat" FOR RANDOM AS #1 LEN=LEN(typbeispiel)
```



Das war auch schon das Schwierigste. Nun geben wir unsere Variablen aus. Da wir wahlfreien Zugriff haben, müssen wir bei der Ausgabe mitangeben, in welchen Datensatz wir die Variable schreiben wollen. Falls wir die Datei neu anlegen, werden wir natürlich ganz vorne beginnen. Den entsprechenden Befehl kennen wir schon: PUT. Damit haben wir in BINARY-Files einzelne Bytes geschrieben. Jetzt schreiben wir ganze Datensätze. Das Ganze könnte z.B. so aussehen:

```
TYPE ta
  a AS INTEGER
  b AS INTEGER
END TYPE

DIM al AS ta

OPEN "R:\a.dat" FOR RANDOM AS #1 LEN = LEN(ta)

FOR i% = 0 TO 9
  al.a = i% * 5: al.b = i% * 8
  PUT #1, i% + 1, al
NEXT i%

CLOSE(1)
```

Das Einlesen funktioniert entsprechend mit GET. Der Modus "RANDOM" bedeutet nicht nur wahlfreien Zugriff, es bedeutet auch, dass wir gleichzeitig Lese- und Schreibzugriff haben. Wir müssen also, wenn wir die Datensätze nach dem Schreiben wieder lesen wollen, die Datei nicht dazwischen schliessen und wieder öffnen.

```
TYPE ta
  a AS INTEGER
  b AS INTEGER
END TYPE

DIM al AS ta

OPEN "R:\a.dat" FOR RANDOM AS #1 LEN = LEN(ta)

FOR i% = 0 TO 9
  al.a = i% * 5: al.b = i% * 8
  PUT #1, i% + 1, al
NEXT i%

FOR i% = 0 TO 9
  GET #1, 10 - i%, al
  PRINT i%, al.a, al.b
NEXT i%

CLOSE (1)
```

Eine solche Sequenz von Datensätzen, wie wir sie im Beispiel geschrieben und gelesen haben, nennt man im allgemeinen in der EDV eine "Datentabelle". In vielen Fällen werden Datentabellen in typisierten Dateien gespeichert, d.h. eine Datentabelle entspricht einer Datei. Es kann aber auch sein, dass viele Datentabellen in einem File gespeichert werden, wie z.B. bei MS Access.

## Schlüssel und Relationen

Wir wollen nun ein neues Programm erstellen. Und zwar ein Super-Fotoalbum. Nicht einfach ein normales Fotoalbum, in dem man die Bilder und vielleicht noch eine Zeile Text sieht. Sondern eins, bei dem man bei Bedarf die Namen der fotografierten Leute, ihre Beziehung zum Fotografen, ihre Adressdaten, Geschichten zum Foto und die technischen Daten der verwendeten Kamera abfragen kann.

Das klingt zunächst weniger nach technischer Revolution, als nach sehr viel Arbeit: Zu jedem Foto muss alles erfasst werden: Die Namen der Leute, die Beziehungen, die Adressen, die Geschichten, die technischen Daten. Oh jeh. Das kann bei ein paar tausend Fotos schnell ziemlich anstrengend werden.

Die technische Revolution besteht darin, dass das gar nicht soviel Arbeit ist, wie man meinen könnte, wenn man s.g. *Relationen* benutzt. Wie man so etwas nutzt und programmiert, damit beschäftigt sich dieser Abschnitt, anhand einer Mini-Version unseres Albums.

### Was ist eine Relation?

Nehmen wir an, wir erstellen zunächst zwei Tabellen. Eine enthält die Dateinamen der Fotos. Die andere enthält die Daten zu den Personen: Namen, Beziehung zum Fotografen ("Cousin" oder "Schulfreund") und die Adresse. Dann müsste ja eigentlich nur noch bei den Fotos notiert werden, welcher der Personen in welchem Foto erscheint. Technisch gesprochen: Welche der Datensätze aus der Personentabelle welchem Datensatz der Fototabelle zugeordnet werden muss. So eine Zuordnung nennt man *Relation*.

Fototabelle		Personentabelle		
Dateiname		Name	Adresse	Beziehung
C:\fotos\2005\sommer\al.bmp	←	Peter Meier	12345 Wollingen	Schulfreund
C:\fotos\2005\sommer\aa2.bmp		Susanne Müller	54321 Strickdingen	Ex-Freundin
C:\fotos\2005\sommer\aa3.bmp	←	usw.		
usw.				

## Schlüssel

Ich muss also in der Fototabelle nur so eine Art Zeilennummer speichern, die auf den richtigen Datensatz in der Personentabelle verweist. Und da auf Fotos mehrere Personen abgebildet sein können, müssen auch mehrere "Zeilennummern" abspeicherbar sein. Eine solcher Verweis auf "Zeilennummern", bzw. genauer Datensatznummern stellt aber nichts anderes dar als das, was wir schon vor einigen Kapiteln kennengelernt haben: Zeiger. Wir müssen also einfach Zeiger auf Datensätze abspeichern. Und ein solcher Zeiger auf einen Datensatz heisst in der Datenbanksprache *Sekundärschlüssel*. Die Frage ist jetzt nur noch, wie man die Adresse selbst, also die Datensatznummer in der Datenbanksprache bezeichnet. Na, dreimal dürfen Sie raten: Richtig, *Primärschlüssel*. Das Ganze sieht also dann so aus:

### Fototabelle

Primärschlüssel	Dateiname	Sekundärschlüssel1	Sekundärschlüssel2
0	C:\fotos\2005\sommer\al.bmp	0	1
1	C:\fotos\2005\sommer\aa2.bmp	-1	-1
2	C:\fotos\2005\sommer\aa3.bmp	0	-1
3	usw.		

Personentabelle			
Primärschlüssel	Name	Adresse	Beziehung
0	Peter Meier	12345 Wollingen	Schulfreund
1	Susanne Müller	54321 Strickdingen	Ex-Freundin
2	usw.		

Das ist dann schon der ganze Zaubertrick. In einem Satz: Wir speichern bei den Fotos noch Datensatznummern anderer Tabellen, die noch weitere Informationen zum Foto enthalten.

Die -1 im Sekundärschlüssel ist ein s.g. Missing-Code. Er zeigt an, dass hier kein Schlüssel vorhanden ist. Er wurde deshalb negativ gewählt, weil er bei der Verwendung in einem Array oder als Datensatznummer zu einem Laufzeitfehler führt. Somit bemerkt man beim Testen mit Sicherheit, wenn einmal auf Schlüssel zugegriffen wird, die gar nicht existieren dürfen.

## Datenbank

Und das Ganze zusammen, also Datentabellen und die Relationen, die sie verknüpfen, nennt man eine *Datenbank*.

Nun dürfte auch klar sein, wie wir das Ganze auf Geschichten erweitern können: Nehmen wir an, es gibt noch ein Urlaubstagebuch. In Tabellenform steht zu jedem Tag des Urlaubs ein Texteintrag. Ein Datensatz enthält also Elemente (ein Element nennt man in der Datenbanksprache ein "Feld"), also als Felder das Datum und einen String

mit dem Tagebuchtext dieses Tages. In der Fototabelle müssen wir dann nur einen Verweis auf den richtigen Tagebuch-Datensatz unterbringen.

### **Übung:**

Nun müssen wir allerdings gut organisieren: Welcher Schlüssel in der Fototabelle gehört zu welcher Tabelle? Überlegen Sie sich einmal eine TYPE-Struktur für die Fototabelle, so dass hier nichts durcheinanderkommt!

### **Umsetzung**

Es gibt nun zwei Art und Weisen, wie wir so etwas in QBASIC umsetzen können. Erste Methode: Wir speichern die Datentabellen in Arrays. Zweite Methode: Wir speichern die Datentabellen in Dateien. Die erste Methode würde ich aus Platzgründen nicht empfehlen. Ein Array kann in QBASIC maximal 64K Umfang haben. Ausserdem muss dann für jede Operation die gesamte Datentabelle in den Hauptspeicher gelesen werden. Keine gute Idee. Damit bleiben also die typisierten Dateien. Wir brauchen also zwei Programme: Ein Einleseprogramm, mit dem wir die Fotoeinträge und Personeneinträge (und Tagebucheinträge) erfassen können. Und das eigentlich Fotoalbum-Programm, in dem wir die Fotos anschauen. Natürlich können wir beide Funktionen auch in ein und demselben Programm unterbringen.

Die Schlüssel stellen die Datensatznummern dar, die wir ins 2. Argument hinter PUT und GET schreiben. Das Erfassungsprogramm sollte uns allerdings die Arbeit abnehmen, die Datensatznummern zu verwalten. Es sollte also z.B. fragen: "Welche Personen befinden sich auf dem Foto?". Wir geben dann ein: "Peter Meier" und "Susanne Müller". Das Programm sucht dann in der Personentabelle im Namensfeld, ob dort solche Einträge zu finden sind und speichert im positiven Fall die entsprechenden Datensatznummern bei den Fotos ab.

### **Grosse Übung**

Nun machen Sie sich ans Werk: Erstellen Sie das Fotoalbum-Programm! Zumindest eine Minimalversion davon. Ach ja: Sie können mit QBASIC natürlich nicht so ohne Weiteres jpg-Dateien darstellen. Es gibt viele Freeware-Programme, z.B. [dieses hier](#), mit dem sie ein paar Fotos ins bmp-Format konvertieren können. Oder Sie lassen in einem eigenen Fenster mittels dem SHELL-Befehl das entsprechende Foto anzeigen. Das ist mit Sicherheit sogar komfortabler, für Sie wie auch für den Benutzer.

---

# Funktionen (SUB und FUNCTION)

---

Viele, die schon gut programmieren und sich diese Einführung anschauen, werden es unmöglich finden, dass hier das Prinzip der Funktionen erst im so ca. 25. Kapitel kommt. Das Prinzip wird meist als so elementar betrachtet, dass es in den meisten anderen Einführungen gleich in der ersten Hälfte des ersten Kapitels gebracht wird.

Nun, dass diese Einführung nicht ist wie andere, darauf wird schon im ersten Kapitel verwiesen. Eine Besonderheit mag dabei die Nutzung des folgenden Prinzips sein: Leiden ist ein guter Lehrmeister. Den Sinn und den Verwendungszweck eines Werkzeugs lernt man dann besonders gut, wenn man es vermisst. Der Schweiss ist es, der in Erinnerung bleibt ;-)))

## Namensräume

Im folgenden wird es um eine Variante zum GOSUB-Befehl gehen. Wir haben ja schon gelernt, dass es ein sehr wichtiges Prinzip ist, sein grosses Programmierproblem in lauter kleine, möglichst allgemeingültige Programmierprobleme zu zerlegen. Das ist eine der grössten Künste des Programmierens und noch dazu eine, die sehr Spass macht, wenn man sie beherrscht. Denn dann kann man all die kleinen Lösungen, die man in den Monaten und Jahren programmiert hat, immer wieder benutzen. Es entstehen immer mehr kleine Unterprogramme, die z.B. Schlüsselwörter in Strings suchen, sie suchen und ersetzen, sortieren, Teilstrings auf ganz bestimmte Art und Weisen in Zahlen umwandeln, Strings formatieren usw.. Genauso mit Grafik, Dateien u.v.m.

Wenn Sie versucht haben, so ausführlich Unterprogramme zu nutzen, werden Sie allerdings immer wieder auf ein grosses und ärgerliches Hindernis gestossen sein: Die Variablen. Wenn man sich angewöhnt, wie in der Informatik üblich, für Schleifen z.B. die Variablen I, J und K herzunehmen - in Unterprogrammen geht das nicht. Denn meist sind die gleichen Variablen ja schon im Hauptprogramm verwendet worden. Also muss man dann im Unterprogramm I1, J1 und K1 oder sowas nehmen. Aber dann ruft das Unterprogramm noch ein Unterprogramm auf. Man denkt dabei nicht daran, dass dieses ja auch I1, J1 usw. verwendet - und schon kracht's. Und man begreift erst mal gar nicht, warum. Denn es kommt ja keine Fehlermeldung.

In der Informatik nennt man so etwas eine "Name Space Pollution", eine Namensraumverschmutzung. Bisher wird das Problem der Namensraumverschmutzung so gross gewesen sein, dass eine sinnvolle Wiederverwendung von Unterprogrammen nur dann möglich war, wenn man bei jedem Programmierprojekt an alle Unterprogramme Hand anlegt und ihre Variablennamen mühsam so hinbiegt, dass sie sich in der aktuell verwendeten Kombination nicht überschneiden - was für ein Umstand!

Das wird jetzt besser. Und dazu gibt es in QBASIC SUB's. SUB's sind Unterprogramme, innerhalb der Variablennamen ihre eigene Bedeutung haben. Eine Variable I innerhalb des SUB hallo1 ist eine andere Variable, als das I innerhalb des SUB hallo2 und alle beide sind andere Variablen, als das I innerhalb des Hauptprogramms. Und das gilt für alle anderen Variablen genauso. Ein Variablenname ist also jetzt nicht mehr eins zu eins mit einem Speicherplatz verknüpft, sondern er verweist auf verschiedene Speicher, je nachdem, in welchen SUB oder Hauptprogramm wir uns befinden. Wozu das gut ist, brauche ich nun gewiss nicht mehr zu erklären! Wir sagen zu so etwas auch: Der Raum innerhalb eines SUB's ist ein Namensraum. Und wir lernen jetzt, dass QBASIC

offenbar mehrere Namensräume verwalten kann.

Allgemein nennen wir solche Unterprogramme mit eigenen Namensräumen *Funktionen* oder *Methoden* oder (wie schon früher gelernt) *Routinen*. Wir werden den Terminus "Funktion" beibehalten, für SUBs und auch für die gleich noch einzuführenden FUNCTIONS. "Funktionen" im informatischen Sinn haben also nicht unbedingt einen Rückgabewert wie die Sinusfunktion oder Wurzelfunktion.

## Navigationsprobleme

Schreiben Sie einmal in ein neues Programm folgende Anweisungen:

```
SUB hallo1
```

Sobald Sie die Zeile mit einem ENTER abschliessen, ergänzt QBASIC von selbst eine weitere Zeile:

```
SUB hallo1
END SUB
```

Überhaupt übernimmt QBASIC jetzt ziemlich energisch die Regie, was geschrieben werden darf und was nicht. Wenn Sie versuchen, vor das SUB eine Zeile einzufügen, macht QBASIC sofort daraus einen Kommentar. Wenn Sie versuchen, nach dem END SUB noch etwas einzufügen, dann protestiert QBASIC und verbietet Ihnen das glatt. Was ist denn da los?

Stillschweigend hat QBASIC hier ein neues Fenster innerhalb der Entwicklungsumgebung aufgemacht. Sie sehen das, wenn Sie F2 drücken oder auf Ansicht->SUBs gehen. Da sehen Sie eine Liste der Routinen, aus denen das Programm besteht. Und Sie sehen, dass das schon zwei sind: hallo1 und "Unbenannt", das Hauptprogramm, das den Dateinamen trägt. Wenn Sie "Unbenannt" auswählen, dann landen Sie wieder im Hauptprogramm und damit vor einem leeren Bildschirm. Hier ist alles so wie es schon immer war.

Das Konzept ist also, für jedes Unterprogramm, jedes SUB ein eigenes Fenster aufzumachen. Daher das Schreibverbot: Vor und nach dem SUB machen im SUB-Fenster Anweisungen keinen Sinn.

Übrigens ist diese Art, SUB's im eigenen Fenster zu verwalten, eine Spezialität von QBASIC. In anderen BASIC-Versionen und Entwicklungsumgebungen schreibt man die Funktionen ins selbe Fenster wie das Hauptprogramm.

## Wir testen SUB's

Wir füllen nun mal unser neues SUB hallo1 mit Inhalt. Und das Hauptprogramm auch:

```
Hauptprogramm:
-----
DIM i AS INTEGER

FOR i = 0 TO 3
  PRINT "main", i
  PRINT "-----"
  hallo1
NEXT i

Sub's:
-----
```

```

SUB hallo1

  DIM i AS INTEGER

  FOR i = 0 TO 3
    PRINT "hallo1", i
  NEXT i

END SUB

```

Hier sehen wir, wie ein SUB aufgerufen wird. Nämlich einfach mit seinem Namen. Ohne irgendein Schlüsselwort davor. Als wenn das SUB ein eigener Befehl von QBASIC wäre. Und genauso sind Funktionen (so nennt man SUB's im allgemeinen) auch gedacht: Sie sollen den Befehlssatz einer Programmiersprache erweitern. Durch den eigenen Namensraum ist das jetzt auch möglich: Man kann einen SUB aufrufen, ohne sich darum zu kümmern, was innerhalb des SUB's passiert. Aber ist das auch so? Schauen wir uns die Ausgabe an:

Die Ausgabe:

```

main          0
-----
hallo1        0
hallo1        1
hallo1        2
hallo1        3
main          1
-----
hallo1        0
hallo1        1
hallo1        2
hallo1        3
main          2
-----
hallo1        0
hallo1        1
hallo1        2
hallo1        3
main          3
-----
hallo1        0
hallo1        1
hallo1        2
hallo1        3

```

Hat funktioniert. Wir haben im Hauptprogramm die Schleifenvariable I verwendet, ohne uns darum zu kümmern, dass I in hallo1 auch verwendet wird. Aber das macht nichts, da in hallo1 I eine andere Variable darstellt. Der Wert vom I des Hauptprogramms bleibt dabei erhalten: Nach dem ersten Aufruf von hallo1 fährt das Hauptprogramm ganz brav mit I=1 fort.

## Übung:

Schreiben Sie das gleiche Testprogramm mit GOSUB's und vergleichen Sie das Ergebnis!

## Kein GOSUB mehr

Ab jetzt empfehle ich Ihnen, kein GOSUB mehr benutzen. Auch innerhalb von SUB's nicht. Wenn Sie sich das Problem von Namensraumkonflikten unbedingt weiter ans Bein binden wollen, können Sie natürlich weiter daran festhalten, aber da selbst das kleinste Unterprogramm meistens Variablen verwendet, werden Sie dann auch weiterhin viel Ärger haben...

## Argumente

Nun haben wir aber ein Problem. Innerhalb des SUB's können wir Variablen anderer SUB's oder des Hauptprogramms nicht sehen. Nehmen wir an, wir wollen dem SUB vom Hauptprogramm mitteilen: "Vertausche die Reihenfolge der Zeichen des folgenden Strings!" Wie "geben" wir dem SUB den String?

Dazu gibt es *Argumente*. Die schreibt man in Klammern hinter den SUB-Namen:

```
SUB vertausch (s AS STRING)
  DIM slen AS INTEGER, i AS INTEGER, s1 AS STRING

  slen = LEN(s)
  FOR i = 1 TO slen
    s1 = s1 + MID$(s, slen - i + 1, 1)
  NEXT i

  PRINT s1
END SUB
```

Diese Funktion nimmt den übergebenen String s, vertauscht die Reihenfolge der Zeichen und gibt das Resultat auf dem Bildschirm aus. Aufrufen können wir sie im Hauptprogramm mit

```
vertausch
```

also z.B.

```
vertausch "qbasic"
```

## Rückgabe von Werten

So ist unsere Funktion "vertausch" allerdings noch nicht schön. Wer braucht schon eine Funktion, die die Zeichenreihenfolge vertauscht und dann sofort das Resultat auf dem Bildschirm ausgibt? Meistens werden wir den invertierten String irgendwie weiterverwenden wollen. Aber dazu müsste das SUB in der Lage sein, das Resultat wieder an uns zurückzugeben.

Kein Problem. Dazu tauschen wir das Schlüsselwort SUB gegen FUNCTION aus. Und verpassen dem Funktionsnamen selbst einen Datentyp, wie wenn es eine normale Variable wäre. Da in diesem Fall die Funktion einen String zurückgeben soll, ist die Funktion vom Typ String:

```
FUNCTION vertausch(s AS STRING) as STRING
  (...)
END FUNCTION
```

Jetzt ist nur noch die Frage, wie der Rückgabewert definiert wird. Nun, logisch wäre es, das ganz genauso wie bei einer Variablen zu machen:

```
' So geht's in QBASIC leider nicht
FUNCTION vertausch(s AS STRING) as STRING
  (...)
vertausch=...
END FUNCTION
```

Hier ist QBASIC unkonsequent und lässt nur eine Typdeklaration mittels der Spezifizierer ("Spezialzeichen") zu:

```
' So geht's in QBASIC
```

```

FUNCTION vertausch$(s AS STRING)
(...)
vertausch$=...
END FUNCTION

```

In neueren BASIC-Versionen funktioniert aber auch die obere, logischere Syntax.

Im konkreten Beispiel:

```

FUNCTION vertausch$(s AS STRING)
  DIM slen AS INTEGER, i AS INTEGER, s1 AS STRING

  slen = LEN(s)
  FOR i = 1 TO slen
    s1 = s1 + MID$(s, slen - i + 1, 1)
  NEXT i

  vertausch$=s1
END FUNCTION

```

Im Hauptprogramm verwenden wir die Funktion genauso wie eine der eingebauten Funktionen (sin() oder asc()), allerdings ohne die Klammern um die Argumente. Das mag QBASIC nicht:

```

Hauptprogramm:
-----
DIM s as STRING

s = vertausch$("QBASIC")
PRINT s

```

Schauen Sie sich einmal das Hauptprogramm an! Ist das nicht prima lesbar und verständlich? Jeder weiss, was dieses Hauptprogramm macht! Und das ist das Geheimnis, warum es in der strukturierten Programmierung möglich wird, riesige Programme zu erstellen: Weil dank der Strukturierung in Funktionen und dank der Reduzierung der Menge an Variablennamen, mit der hantiert wird, die Programme übersichtlich und lesbar bleiben. In unserem Hauptprogramm hätten wir sogar auf das s verzichten können:

```

Hauptprogramm:
-----
PRINT vertausch$("QBASIC")

```

Allerdings will QBASIC bei FUNCTION's Klammern um die Argumente herum haben. Auch da ist kein so richtig einleuchtendes Prinzip erkennbar. QBASIC will, dass SUB's ohne Klammern um die Argumente aufgerufen werden und FUNCTION's mit Klammern, obwohl beides Funktionen sind, nur einmal mit und einmal ohne Rückgabewert. (Das ist ein typisches Beispiel für mangelnde informatische Stringenz in älteren BASIC-Implementierungen, die bei komplexeren Problemen die Gefahr hervorruft, dass die Programme "zerfasern", weil abstraktere, schönere Lösungen nicht möglich sind.)

## Rückgabe von mehreren Werten, gemeinsame Variablen

Was mache ich aber, wenn mehrere Werte zurückgegeben werden sollen? Meinetwegen erzeugt die Funktion zwei Koordinaten x und y. Wie gebe ich diese wieder zurück? Nun, da Funktionen (in QBASIC) nur einzelne Zahlen oder höchstens einen String zurückliefern können, müssen wir einen anderen Weg gehen. Wir schreiben einmal folgendes Testprogramm:

```

SUB testsub(x as integer)
  x=1

```



```

END SUB

'Hauptprogramm
x=0
CALL testsub(x)
PRINT x

```

Was wird ausgegeben? Eine Eins. Es ist also so, dass Argumente von der Funktion (dem SUB) nicht nur gelesen, sondern auch geschrieben werden können. Und Änderungen bleiben in der aufrufenden Funktion wirksam. Mit anderen Worten: Argumente sind Variablen, die Aufrufer und Funktion gemeinsam verwalten. Man nennt solche Argumente "By Reference"-Argumente.

Eigentlich sind "By Reference"-Argumente nicht so gut, da sie das Prinzip der getrennten Namensräume zwischen den Funktionen aufweichen und daher wieder Fehleranfälligkeit mit sich bringen. Da aber QBASIC einige Beschränkungen im Bereich "Strukturierte Programmierung" aufweist, sind wir in diesem Fall ganz froh um dieses "Feature".

**Noch ein Beispiel:** Sollen Hauptprogramm und das SUB testsub() die Variable n gemeinsam verwalten, dann übergibt man sie einfach testsub() mit: SUB testsub(x as integer, n as integer). Dabei ist x das eigentliche Argument und n eben nur die gemeinsame Variable, die z.B. einen Programmstatus oder ein Limit enthalten kann. Will man zudem noch einen Wert zurückgeben, dann kann man entweder daraus eine FUNCTION machen (bei nur einem Rückgabewert empfehlenswert) oder der SUB noch ein drittes Argument mit auf den Weg geben SUB testsub(x as integer, n as integer, returnvalue as integer).

## Goldene Kommentarregel:

Jedes SUB sollte gleich hinter der Kopfzeile einen Kommentar mit folgenden Angaben enthalten:

- Was tut dieses SUB?
- Was haben die Argumente für eine Bedeutung?
- Was gibt das SUB zurück?

## Deklaration von Funktionen (DECLARE)

Ihnen wird wahrscheinlich schon aufgefallen sein, dass QBASIC eigenmächtig auch im Hauptprogramm aktiv wird, sobald sie das Programm speichern: Es fügt Zeilen der Form

```
DECLARE Subname (Argumente)
```

hinzu. Warum macht es das? Nun, das hat technische Gründe. Der Interpreter muss von Anfang an, also schon bei der ersten interpretierten Programmzeile, wissen, welche Funktionen es im Programm gibt und wie sie aufgerufen werden. Dazu dienen die DECLARE's. Der Interpreter startet also mit den DECLARE's, erst dann geht er zur ersten Zeile des Hauptprogramms. Für Sie als Programmierer haben die DECLARE's eine kleine Falle parat: Falls die DECLARE's schon eingefügt wurden und Sie im Nachhinein einen Funktionskopf noch einmal ändern, dann meckert QBASIC: Falsche Anzahl der Argumente (oder etwas ähnliches). Sie müssen die Änderung immer an zwei Stellen durchführen, bei der *Deklaration*, also dem DECLARE, und bei der *Definition* im SUB-Fenster.

```
%%1
```

---

## Ein Anwendungsbeispiel: Animation in 3D

---

Jetzt haben wir das Grundlegende zur Funktionenprogrammierung gelernt. Es gibt in QBASIC noch ein paar andere Konzepte, wie z.B. SHARED-Variablen, die aber eher unelegante Krückstöcke für Konzepte der strukturierten Programmierung sind, die in moderneren BASIC-Implementationen (und erst recht in anderen Sprachen wie PASCAL, Java oder Python) besser umgesetzt wurden. Daher ist es sinnvoller, sich damit erst im nächsten Teil zu beschäftigen.

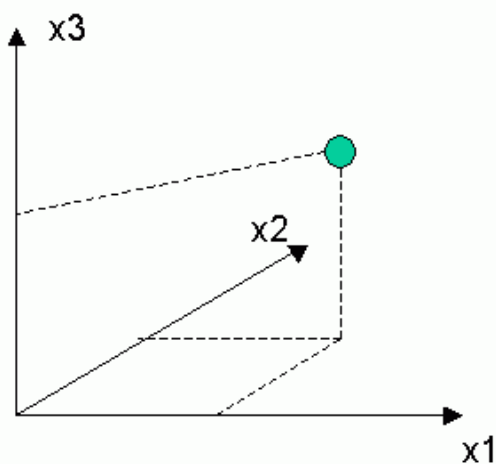
Damit Sie aber ein Gefühl dafür entwickeln, wie mächtig die Werkzeuge sind, die Sie in den letzten Kapiteln gelernt haben, sollen Sie abschliessend für diesen Teil ein Übungsprogramm erstellen, bei dem Sie vermutlich nicht annehmen, dass Sie das schaffen könnten: Eine 3D-Animation. Ein Körper soll im dreidimensionalen Raum um eine beliebige Achse gedreht werden können.

Die einzige grosse Kunst bei dieser Übung ist es, die grosse Aufgabe in kleine Aufgaben zu zerlegen, die wir jede in einer eigenständigen Funktion lösen können. Dann müssen wir die Funktionen nur noch richtig "zusammenstöpseln" - und das war's.

Es sei vorweggenommen: Natürlich gibt es heute für diese Aufgabe komfortable Ressourcen (DirectX, OpenGL). Aber erstens sind diese für QBASIC nur schwer zugänglich und zweitens macht es ja auch Spass, zu sehen, wie diese Ressourcen intern dem Prinzip nach funktionieren.

### Dreidimensional und zweidimensional

Fangen wir mal ganz einfach an: Ein Punkt im dreidimensionalen Raum - wie zeichne ich den, wenn meine Zeichnung doch eigentlich nur zweidimensional sein kann? Wir müssen hier zwischen der dreidimensionalen Welt "hinter" dem Bildschirm unterscheiden und dem 2-dimensionalen Abbild der Welt auf dem Bildschirm. Ein ganz zentraler Trick wird sein, ersteinmal alle Operationen im 3-dimensionalen Weltkoordinatensystem (Weltsystem) auszuführen und erst am Schluss uns darum zu kümmern, wie wir das nun ins 2-dimensionale Bildkoordinatensystem (Screen) überführen.



Jeder Punkt hat im Weltsystem Koordinaten  $(x_1, x_2, x_3)$ . Wir betrachten dieses Weltsystem aus einer bestimmten (fixen, isometrischen) Perspektive, in der die Achse  $x_2$  in die Screenebene hineinzeigt und damit in die Screenebene "übersetzt" werden muss. Die Koordinaten  $x_1$  und  $x_3$  entsprechen also zunächst denen des Screens. Aber dann müssen wir von dort im Winkel der  $x_2$ -Achse zur  $x_1$ -Achse weiterlaufen. Nennen wir diesen  $\alpha$ . Und zwar um die Strecke  $x_2 \cdot \cos(\alpha)$  in Richtung  $x_1$  und um die Strecke  $x_2 \cdot \sin(\alpha)$  in Richtung  $x_3$ . Damit ist die ganze Übersetzung fertig:

- $x(2)=x1(3)+x2(3)*\sin((\alpha)$
- $y(2)=x3(3)+x2(3)*\cos((\alpha)$

Die Zahl in Klammern gibt jeweils an, zu welchem System die Koordinate gehört, (2) ist das Bildschirmsystem, (3) ist das dreidimensionale Weltsystem.

$\alpha$  ist eine Systemkonstante, die unseren Blickwinkel angibt. (Dieser soll nicht veränderlich sein.) Ein Wert für Alpha, der gut aussieht, ist 66 Grad.

Die erste Funktion, die wir also programmieren können, ist eine, die zu einem 3D-Punkt die Koordinaten auf dem 2D-Schirm liefert. Nennen wir sie `coords3d(x1,x2,x3)`. Da zwei Koordinaten zurückgegeben werden müssen (x und y für den Screen), machen wir zwei Funktionen draus: `coords3dx%(x1,x2,x3)` und `coords3dy%(x1,x2,x3)`. (Das Prozentzeichen, weil die Funktionen INTEGER-Werte zurückgeben.) Jede Funktion entspricht einer der obigen Formeln.

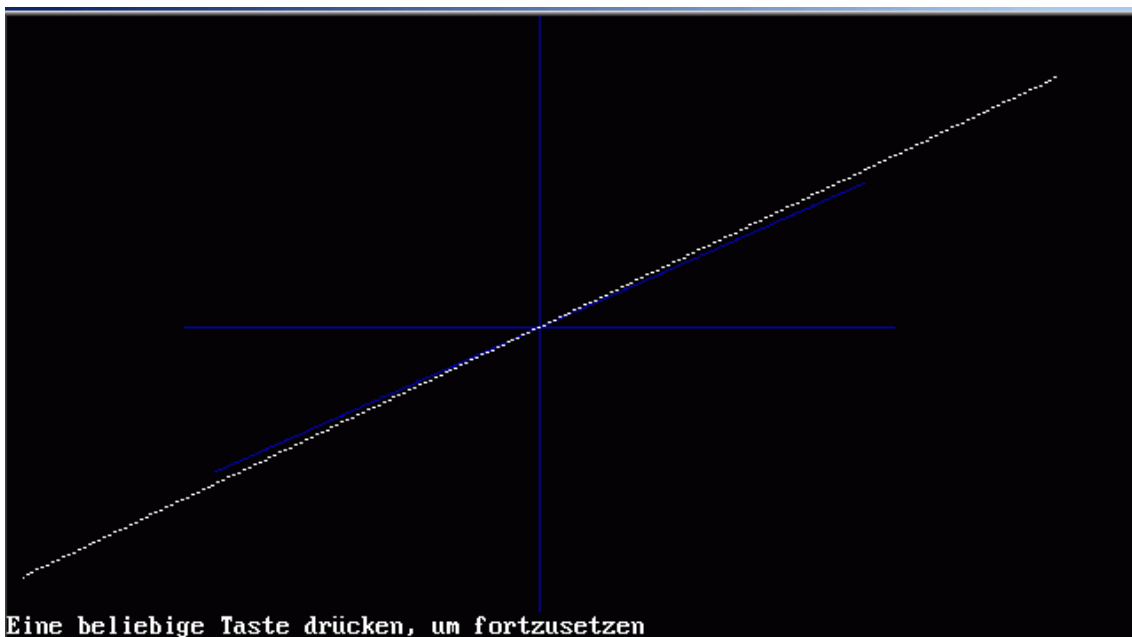
## 3D-Linien

...sind nun sehr, sehr einfach. Unser `line3D`-Befehl setzt einfach in den `LINE`-Befehl von QBASIC die `coords3dx%`- und `coord3dy%`-Funktion für Anfangs- und Endpunkt der Linie ein und fertig.

Ein Hinweis: Wenn Sie jetzt losschreiben (und das sollten Sie), dann benutzen Sie bitte Screen 9. Das hat seinen Grund weiter unten...

## 3D-Koordinatensystem

Mit Hilfe unseres `line3D`-Befehls können wir nun leicht ein 3-dimensionales Koordinatensystem zeichnen:



, hier gleich noch mit einer 3D-Linie darin.

Die Funktion, die dieses Koordinatensystem zeichnet, nennen wir "`coordsys`".

Man kann aber schon nettere Sachen machen. Z.B. eine dreidimensionale Spirale. Dazu lassen wir `x1` und `x2` einen Kreis beschreiben, während `x3` in die Höhe wandert:

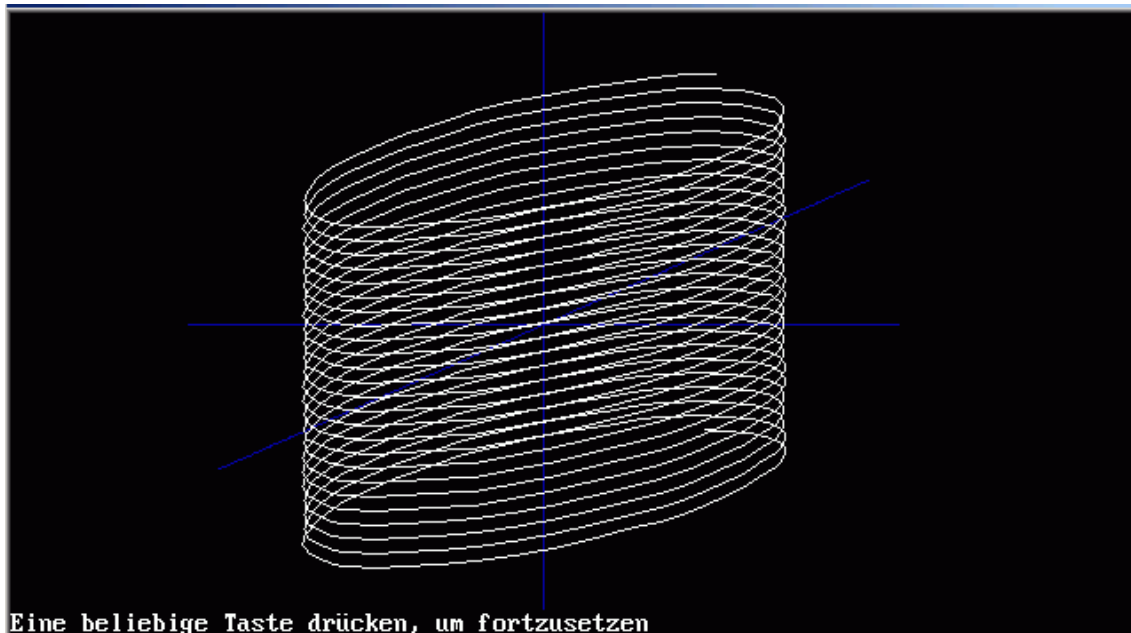
```
FOR i = 0 TO 1000
  y1 = INT(100 * SIN(i / 20 * 22 / 7) + .5)
  y2 = INT(100 * COS(i / 20 * 22 / 7) + .5)
  y3 = INT((i - 500) / 5 + .5)
  IF (i = 0) THEN
```

```

      x1 = y1: x2 = y2: x3 = y3
    END IF
    CALL line3d(x1, x2, x3, y1, y2, y3, 15)
  END

```

Das sieht dann so aus:



## 3D-Punkte

Allerdings werden wir merken, dass das Hantieren mit jeweils 3 Koordinaten für jeden 3D-Punkt mit der Zeit etwas umständlich wird. Daher führen wir einen neuen Datentyp ein:

```

TYPE point3dtyp
  x1 as DOUBLE
  x2 as DOUBLE
  x3 as DOUBLE
END TYPE

```

$x_1$ ,  $x_2$  und  $x_3$  haben wir bewusst als Fließkommazahl mit hoher Genauigkeit gewählt. Für die später auszuführenden Bewegungen und Rotationen werden viele Additionen und Multiplikationen erforderlich. Da kommt es leicht zu Rundungsungenauigkeiten, die sich dann anhäufen und schliesslich das Bild verzerren, wenn wir die innere Rechengenauigkeit zu niedrig wählen.

## Noch ne 3D-Linienfunktion

Diese nimmt nicht sechs Einzelkoordinaten, sondern zwei 3D-Punkte entgegen:

```
SUB line3dp (p1 AS point3dtyp, p2 AS point3dtyp, coll AS INTEGER)
```

Ist doch schick, oder?

## 3D-Klötze

Das Problem, eine Figur in 3D darzustellen, gehen wir so an, dass wir diese Figur aus lauter viereckigen Klötzen darstellen wollen. In diesem Beispiel beschränken wir uns auf die Darstellung eines Klotzes. Wie beschreiben wir den Klotz? Nun, ein solcher Klotz hat acht Eckpunkte. Und 3D-Punkte haben wir ja schon. Also nichts einfacher als das:

```

TYPE cubetyp
  p1 AS point3dtyp
  p2 AS point3dtyp

```

```

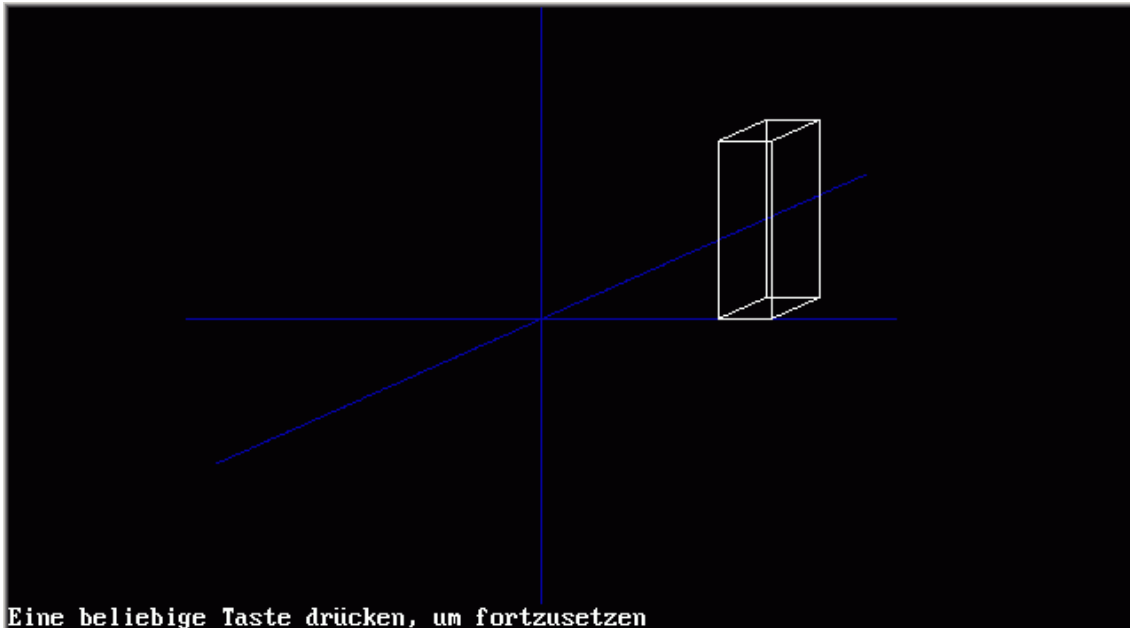
p3 AS point3dtyp
p4 AS point3dtyp
p5 AS point3dtyp
p6 AS point3dtyp
p7 AS point3dtyp
p8 AS point3dtyp
END TYPE

```

...und schon ist er fertig. Wollen wir einen solchen Klotz zeichnen, müssen wir nur zwölf Linien zeichnen: Mit jeweils vier Linien zeichnen wir "Boden" und "Deckel" des Klotzes, die z.B. von p1,p2,p3,p4 und p5,p6,p7,p8 dargestellt werden. Und dann verbinden wir jeweils eine Ecke des Bodens mit einer Ecke des Deckels, also p1 mit p5, p2 mit p6 usw. Macht zwölf Linien, die mit Hilfe von line3dp() schnell geschrieben sind.

Die Frage ist allerdings, wie wir die Eckpunktkoordinaten bestücken. Das sind doch immerhin jetzt 24 Koordinaten! Eine gute Idee ist, sich einen Stützpunkt auszusuchen. In meinem Kopf z.B. ist es immer der vordere untere linke Punkt. Dies sei p1. Und dessen Lage geben wir an. Z.B. legen wir ihn einfach in den Nullpunkt: p1.x1=0 : p1.x2=0 : p3.x3=0. Dann definieren wir eine Funktion, die nennen wir einfach "cubeinit". Also initialisiere den Klotz. Und zwar mit drei Angaben: Breite, Länge, Höhe. Anfangs soll der Klotz so liegen, dass die Breite in die x1-Richtung (nach rechts) geht, die Länge in die x2-Richtung und die Höhe natürlich in die Höhe (x3). cubeinit(breite, laenge, hoehe) nimmt uns dann die Initialisierung der restlichen sieben Punkte ab.

Und siehe da, wir können unseren ersten Klotz zeichnen!



## Verschiebung

Als nächstes wollen wir Bewegung ins Spiel bringen. Wir fangen einfach an und verschieben erst einmal den Klotz. In sechs mögliche Richtungen. Die Funktion nennen wir "movecube3d":

```
SUB movecube3d(cube as cubetype, dx as point3dtyp)
```

In dx sind die drei Schrittgrößen in x1, x2 und x3-Richtung gespeichert. In der Mathematik nennt man einen solchen "Raumschritt" auch einen "Vektor". Mit solchen Vektoren bekommen wir es bei der Rotation nochmal zu tun. Hier ist aber alles einfach: Einfach die jeweilige Koordinate von dx auf die jeweilige Koordinate aller Punkte des Klotzes draufaddieren und dann alles neu zeichnen. Fertig. Damit das richtig schön dynamisch wird, sollten wir unser movecube3d von einer Tastaturschleife aus aufrufen:

```

DIM cube as cubetype
DIM dp as point3dtyp

cube.p1.x1=0
cube.p1.x2=0

```

```

cube.pl.x3=0

CALL cubeinit(cube,20,20,100)

CALL cubedraw(cube) 'Das ist die Funktion, die die zwölf Linien malt.

a$ = ""
ascreen = 0
WHILE (a$ <> "q")
  a$ = ""
  WHILE (a$ = ""): a$ = INKEY$: WEND
  IF (a$ = CHR$(0) + "M") THEN dp.x1=3:dp.x2=0:dp.x3=0:CALL movefig3d(cube,dp)
  IF (a$ = CHR$(0) + "K") THEN dp.x1=-3:dp.x2=0:dp.x3=0:CALL movefig3d(cube,dp)
  IF (a$ = CHR$(0) + "P") THEN dp.x1=0:dp.x2=3:dp.x3=0:CALL movefig3d(cube,dp)
  IF (a$ = CHR$(0) + "H") THEN dp.x1=0:dp.x2=-3:dp.x3=0:CALL movefig3d(cube,dp)
  IF (a$ = "t") THEN dp.x1=0:dp.x2=0:dp.x3=3:CALL movefig3d(cube,dp)
  IF (a$ = "g") THEN dp.x1=0:dp.x2=0:dp.x3=-3:CALL movefig3d(cube,dp)
WEND

```

Damit können wir den Klotz in allen drei Dimensionen über den Bildschirm bewegen:

- Cursor links/rechts = x1-Richtung
- Cursor rauf/runter = x2-Richtung
- "t" und "g" = x3-Richtung

Bitte beachten: Innerhalb von movefig3d() muss nochmals cubedraw() aufgerufen werden und cubedraw() muss immer *alles* neu zeichnen. Also "CLS" und dann das Koordinatensystem (sofern man es haben möchte) und den ganzen Klotz.

Das können wir natürlich nur deswegen so machen - und auch nur deswegen ist die Aufgabe für uns jetzt in BASIC relativ einfach bewältigbar - weil unser Rechner so irre schnell ist, dass er das locker in Millisekunden hinkriegt.

## Double Buffer

Das mit den Verschieben funktioniert, solange wir nur einmal pro Sekunde eine Cursortaste drücken. Wenn wir sie aber gedrückt halten, dann bewegt sich die Figur zwar, aber sie wird nicht mehr richtig gezeichnet. Wir sehen sie kaum noch. Das liegt daran, dass die Figur die meiste Zeit gar nicht da oder im halb gezeichneten Zustand ist. In der Millisekunde, in der sie fertig ist, wird sie ja schon wieder gelöscht.

Was können wir da machen? Die entsprechende Lösung heisst "Double Buffering". Wie wir schon im ersten Teil bei der [Grafikprogrammierung](#) erfahren haben, benötigt der Computer zur Darstellung des Screens Speicher. Die Grafikkarte nimmt diesen Speicherinhalt und bildet ihn laufend (ca. 60 bis 100 Mal in der Sekunde) auf dem Monitor ab. Nun gibt es die Möglichkeit, einen kompletten zweiten solchen Speicher zu reservieren. Ob das technisch auch geht, hängt von der verwendeten Grafikkarte ab - bzw. vom Grafikkartenstandard, den QBASIC ansteuert. Dieser EGA/VGA-Standard, den QBASIC nur kennt, ist schon ein bisschen alt und daher gibt es diesen doppelten Speicher nicht bei allen Betriebsmodi, nicht z.B. bei SCREEN 12 oder SCREEN 13, sehr wohl aber bei SCREEN 9. Und den haben wir ja wohlweislich genommen.

Man nennt solche Speicher "Bildschirmseiten". Wir reservieren also zwei Bildschirmseiten. Eine wird dargestellt und eine ist versteckt im Hintergrund. Der Clou ist nun, dass wir auch auf diese versteckte Seite zeichnen können. Wir bauen also das Bild auf der versteckten Seite auf. Und erst wenn es fertig ist, schalten wir es sichtbar (und wechseln gleichzeitig mit dem Zeichnen auf die andere, nun unsichtbare Seite.) Das Zeichnen geschieht auf diese Art und Weise im Hintergrund und es wird im Vordergrund immer eine fertige Zeichnung angezeigt. Daher flackert das Bild nicht mehr.

Das Reservieren und Ansteuern der Bildschirmseiten geschieht alles mit dem SCREEN-Befehl:

Reservieren:	SCREEN 9. Das impliziert schon zwei Bildschirmseiten. By default wird Seite 0 angezeigt und auf Seite 0 gezeichnet
Zeichnen:	SCREEN 9,,,. Die aktive Seite ist die, auf die gezeichnet wird.

Das Zeichnen muss jetzt also so geschehen, dass in einer Variablen die dargestellte Seite gespeichert ist. Nennen wir sie `ascreen`. Sie hat den Wert null oder eins. Vor dem Zeichnen muss als aktive Seite `1-ascreen` eingestellt werden. Dann wird gezeichnet. Anschliessend wird diese Seite auch angezeigt: `SCREEN 9,,0,1` `9,,1-ascreen,1-ascreen`. Und `ascreen` selbst wird aktualisiert: `ascreen=1-ascreen`.

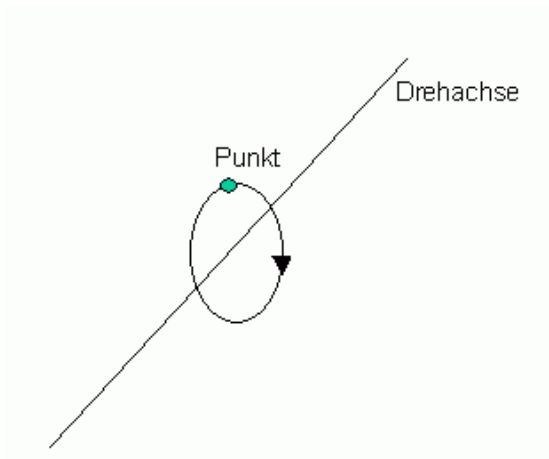
```
'Test fuer zwei Bildschirmseiten
'Seite 1 wird angezeigt, auf Seite 0 wird gezeichnet.
SCREEN 9, , 0, 1
FOR ix = 0 TO 31
  LINE (ix * 20, 0)-(ix * 20, 479)
NEXT ix
'Nach dem Tastendruck wird Seite 0 angezeigt:
WHILE (INKEY$ = ""): WEND
SCREEN 9, , 0, 0
```

Das Wechseln der Bildschirmseiten integriert man am Besten in `cubedraw()`.

Und? Nun sollte das Bewegen des Klotzes richtig flüssig von statten gehen!

## Drehung

Wir kennen in der Geometrie zwei Drehungen: Punktdrehungen und Achsdrehungen. Wir beschäftigen uns mit Achsdrehungen:



### Die Definition einer Achse

Eine Achse ist einfach eine Linie. Wir benötigen also nur zwei Punkte, durch die diese Linie läuft, nennen wir sie `x` und `y`. Zweckmässigerweise werden sie vom Typ `point3dtyp` sein. Für die Achse definieren wir also einfach einen weiteren Typ "`axetyp`":

```
TYPE axetyp
  x: point3dtyp
  y: point3dtyp
END TYP
```

Für den Anfang sollten wir `x` in den Nullpunkt unseres Klotzes legen, da unsere Drehung so einfacher zu rechnen ist.

Die Achse schliessen wir in die Definition unseres Klotzes mit ein.

```
TYPE cubetyp
  p1 AS point3dtyp
  p2 AS point3dtyp
  p3 AS point3dtyp
  p4 AS point3dtyp
  p5 AS point3dtyp
  p6 AS point3dtyp
  p7 AS point3dtyp
  p8 AS point3dtyp
```

```

    axe AS axetyp
END TYPE

```

Beispiel einer Initialisierung wäre z.B.

```

cube.axe.x.x1=cube.p1.x1
cube.axe.x.x2=cube.p1.x2
cube.axe.x.x3=cube.p1.x3
cube.axe.y.x1=cube.p7.x1
cube.axe.y.x2=cube.p7.x2
cube.axe.y.x3=cube.p7.x3

```

Dann verlief die Drehachse diagonal durch den Klotz. Aber das können Sie natürlich nach Lust und Laune wählen!

## Der Satz des Pythagoras

Nun kommen wir zu einem Stück Mathe, von dem Sie wahrscheinlich nicht dachten, dass Sie dem noch einmal begegnen werden: Zum Satz von Pythagoras. Und zwar in Verbindung mit der Vektordarstellung unserer Achse.

Die Zweipunktendarstellung ist sehr angenehm, wenn man die Achse sich vorstellen und definieren soll. Für die Rechnung der Drehung benötigen wir jedoch die Angabe der Linie in einer anderen Form. Wir haben weiter vorne ja schon gelernt, dass ein "Vektor" die Angabe eines "Raumschritts" und damit einer Richtung ist: "Soundsoviel Schritte in x1, soundsoviel Schritte in x2, soundsoviel Schritte in x3". Ein Vektor mit (1,2,2) würde z.B. bedeuten: "1 Schritt in Richtung x1, 2 Schritte in Richtung x2, 2 Schritte in Richtung x3". Es ist eine Richtungsangabe, von wo aus diese Schritte passieren, ist egal.

Diesen Vektor können wir aus unseren zwei Punkten leicht generieren: Wir ziehen einfach die einzelnen Koordinaten voneinander ab:  $v.x1=y.x1-x.x1$  usw.. Das grössere Problem ist, dass die Länge des Vektors genau eins sein muss. Das nennt man dann einen "Einheitsvektor". Wie machen wir das? Dazu brauchen wir nun Herrn Pythagoras. Er sagt uns, wenn wir x1 in Richtung 1 und x2 in die dazu senkrechte Richtung 2 marschieren, wie lange dann die Diagonale ist:

$$d = \sqrt{x_1^2 + x_2^2}$$

Die Länge unseres Vektors im Zweidimensionalen wäre also  $d$ . Und da wir im Dreidimensionalen arbeiten, heisst es halt für  $d$ :

$$d = \sqrt{x_1^2 + x_2^2 + x_3^2}$$

Wenn nun unser Vektor  $d$  lang ist, dann kriegen wir ihn auf die Länge eins, indem wir jede Koordinate um  $1/d$  kürzen. Unser kleiner Minibeweis:

$$\sqrt{\left(\frac{x_1}{d}\right)^2 + \left(\frac{x_2}{d}\right)^2 + \left(\frac{x_3}{d}\right)^2} = \frac{1}{d} \sqrt{x_1^2 + x_2^2 + x_3^2} = \frac{d}{d} = 1$$

Damit haben wir den "Baustoff" für unseren Einheitsvektor zusammen. Wir können das in einer SUB einheitsvektor(v as point3dtyp) zusammenschreiben. In v speichern wir den Vektor. Die Koordinaten von v werden dann auf Einheitslänge umgerechnet:

```

DIM d as double
d = sqrt(v.x1^2+v.x2^2+v.x3^2)
IF (d>0)
    v.x1=v.x1/d
    v.x2=v.x2/d
    v.x3=v.x3/d
ELSE
    d=-1
END IF

```



```
einheitsvektor=d
```

Fertig. Und schon haben wir unseren Achseneinheitsvektor.

## Die Drehung eines Punktes um eines Achse

Der nächste Teil ist einfach nur eine längliche Formel. Sie nennt sich "Drehmatrix" oder "Rotationsmatrix". Geben Sie diesen Begriff einmal in die [Wikipedia](#) ein und Sie erhalten die fertige Drehmatrix. Da wimmelt es von Cosinussen und Sinussen usw. Wir müssen nicht verstehen, warum sie so aussieht, wir müssen sie nur in BASIC umsetzen und wir müssen wissen, wie man sie benutzt. Da wir inzwischen unseren Achseneinheitsvektor haben, ist das auch nicht weiters schwierig. Um Ihnen die Tipparbeit zu erleichtern, gebe ich Ihnen hier die Umsetzung in QBASIC gleich fertig an:

```
FUNCTION rotmatr1! (x1 AS DOUBLE, x2 AS DOUBLE, x3 AS DOUBLE, alpha AS DOUBLE, adir AS DOUBLE,
    DIM r as double
    r = (COS(alpha) + a1 ^ 2 * (1 - COS(alpha))) * x1 + (a1 * a2 * (1 - COS(alpha)) - a3 * SIN(alpha)
    r = r + (a1 * a3 * (1 - COS(alpha)) + a2 * SIN(alpha)) * x3
    rotmatr1 = r
END FUNCTION

FUNCTION rotmatr2! (x1 AS DOUBLE, x2 AS DOUBLE, x3 AS DOUBLE, alpha AS DOUBLE, a1 AS DOUBLE, a2
    DIM r as double
    r = (a1 * a2 * (1 - COS(alpha)) + a3 * SIN(alpha)) * x1
    r = r + (COS(alpha) + a2 ^ 2 * (1 - COS(alpha))) * x2
    r = r + (a2 * a3 * (1 - COS(alpha)) - a1 * SIN(alpha)) * x3
    rotmatr2 = r
END FUNCTION

FUNCTION rotmatr3! (x1 AS DOUBLE, x2 AS DOUBLE, x3 AS DOUBLE, alpha AS DOUBLE, a1 AS DOUBLE, a2
    DIM r as double
    r = (a3 * a1 * (1 - COS(alpha)) - a2 * SIN(alpha)) * x1
    r = r + (a3 * a2 * (1 - COS(alpha)) + a1 * SIN(alpha)) * x2
    r = r + (COS(alpha) + a3 ^ 2 * (1 - COS(alpha))) * x3
    rotmatr3 = r
END FUNCTION
```

(Am Besten, Sie übernehmen das via Copy and Paste in Ihr Programm; das vermeidet Tippfehler!)

(a1,a2,a3) ist der Einheitsvektor, (x1,x2,x3) ist der zu drehende Punkt und alpha ist der Drehwinkel.

Es empfiehlt sich, eine weitere Routine "rot3dpoint(p as point3dtyp, adir as point3dtyp, alpha as double)" einzuführen, die die Anwendung der einzelnen Funktionen rotmatr!() auf die Koordinaten von p übernimmt. adir enthält den Achseneinheitsvektor.

## Die Drehung des Klotzes

Na, Überblick verloren? Was müssen wir nun machen, damit wir den ganzen Klotz drehen können? Ganz einfach: Wir müssen

1. aus cube.axe den Einheitsvektor adir berechnen.
2. rot3dpoint() auf alle acht Eckpunkte des Klotzes anwenden.

Nennen wir das rotfig3d(cube as cubetyp, alpha as double) und es ist das Äquivalent zu movefig3d().

Nun können Sie das Steuerungsmenü entsprechend erweitern, um Befehle zur Rotation und z.B. auch um Befehle zur Veränderung der Achslage.

```
WHILE (a$ <> "q")
    a$ = ""
    WHILE (a$ = ""): a$ = INKEY$: WEND
    IF (a$ = CHR$(0) + "M") THEN dp.x1=3:dp.x2=0:dp.x3=0:CALL movefig3d(cube,dp)
    IF (a$ = CHR$(0) + "K") THEN dp.x1=-3:dp.x2=0:dp.x3=0:CALL movefig3d(cube,dp)
```

```

IF (a$ = CHR$(0) + "P") THEN dp.x1=0:dp.x2=3:dp.x3=0:CALL movefig3d(cube,dp)
IF (a$ = CHR$(0) + "H") THEN dp.x1=0:dp.x2=-3:dp.x3=0:CALL movefig3d(cube,dp)
IF (a$ = "t") THEN dp.x1=0:dp.x2=0:dp.x3=3:CALL movefig3d(cube,dp)
IF (a$ = "g") THEN dp.x1=0:dp.x2=0:dp.x3=-3:CALL movefig3d(cube,dp)
IF (a$ = "n") THEN CALL rotfig3d(cube,-10)
IF (a$ = "m") THEN CALL dotfig3d(cube,+10)
WEND

```

## Das Bewegen der Drehachse

Wenn wir movefig3d() anwenden, dann verschieben wir den Klotz - aber nicht die Drehachse. Das Ergebnis sieht dann lustig aus: Der Klotz dreht sich nicht mehr in sich selbst, sondern in etwa um den Nullpunkt des Koordinatensystems. Was müssen wir tun, damit sich die Drehachse mitbewegt?

An sich ist die Antwort ersteinmal ganz einfach: Einfach die Achskoordinaten mitverschieben. Wir müssten also movefig3d() auch auf die Koordinaten von cube.axe anwenden. Die Sache hat nur einen Haken: Es nutzt nichts. Wenn wir aus den beiden Punkten der Achse wieder einen Einheitsvektor bilden, bleibt dieser Vektor immer derselbe, egal wie wir x und y verschoben haben. Und die Drehmatrix nimmt an, dass dieser Einheitsvektor durch den absoluten Nullpunkt geht.

Die Lösung des Problems heisst dieses Mal in der Fachsprache "Koordinatentransformation". Einfacher gesagt. Wir rechnen temporär die Koordinaten aus, die der Klotz hätte, wenn dessen Achse immer noch im Nullpunkt liegen würde. Dann drehen wir *diese* Koordinaten. Und rechnen sie anschliessend wieder zurück. Klingt komplizierter als es ist.

Sehr nützlich ist es, für diese Operation eine allgemeine addpoint() und subpoint()-Routine zu haben, die die Koordinaten zweier Punkte addiert und subtrahiert.

Dann hüllen wir unsere rot3dpoint()-Routine durch eine rot3dpoint1()-Routine ein. In dieser wird zuerst die Hintransformation ausgeführt, dann rot3dpoint() aufgerufen und dann die Rücktransformation ausgeführt. Das Ganze sieht dann so aus:

```

SUB rot3dpoint1(p AS point3dtyp, diff as point3dtyp, adir AS point3dtyp, alpha AS double)

    DIM p1 AS point3dtyp

    'p1=p-diff
    subpoint p1, p, diff

    CALL rot3dpoint(p1, adir, alpha)

    'p=p1+diff
    addpoint p, p1, diff

END SUB

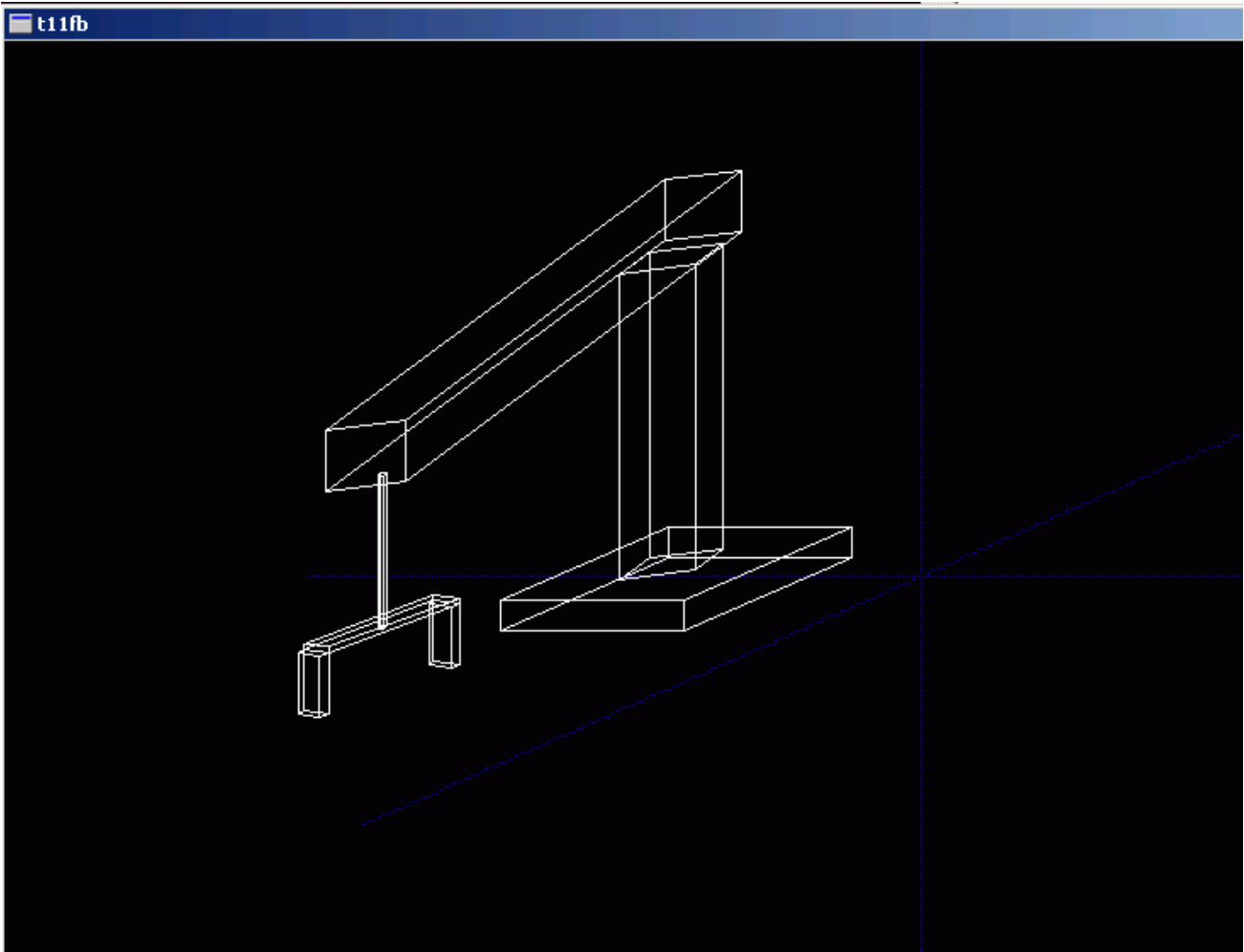
```

Und schon haben wir auch dieses Problem gelöst.

## Ergebnis und beyond: Ein 3D-Kran

Ich hoffe, es fiel Ihnen nicht schwer, aus den einzelnen Bausteinen nun Ihren rotierenden Klotz zusammenzubauen. Wenn Sie es geschafft haben, werden Sie hoffentlich ein bisschen begeistert sein, dass Sie das schaffen konnten. Alles zusammen klang ja höllisch kompliziert. Und wenn Sie im ersten Teil dieses Kurses ("zu Zeiten des C16") vor dieser Aufgabe gestanden wären, hätten Sie nicht eine Sekunde daran gedacht, einen ganzen Kubus in einem 3D-Raum um eine verschobene Drehachse rotieren und dann das Abbild ins Zweidimensionale projizieren zu lassen. Aber dank der Zerlegung in die einzelnen Aufgaben und die Möglichkeit, diese in getrennten Funktionen bearbeiten zu lassen, war es letztendlich nicht schwer, oder? Wahrscheinlich haben Sie sogar Lust auf mehr bekommen. Kann man nun aus mehreren Klötzen eine ganze Figur zusammenbauen? Kein Problem! Kann man einzelne dieser Klötze um eine Achse rotieren und dabei Nachbarklötze "mitnehmen" lassen, so dass "Gelenke" entstehen? Kein Problem! Schwieriger wird es, wenn man die Aussenflächen der Klötze "lackieren" möchte. (Man nennt das in der 3D-Fachsprache "Rendern"). Dann muss man nämlich wissen, welche Flächen aus Sicht des Betrachters vorne und hinten sind. Aber auch das ist ganz und gar kein unlösbares Problem.

Als kleines Ergebnisbeispiel habe ich einen kleinen Kran gebaut. In QBASIC. Genau mit den Routinen, die wir hier besprochen haben. Ohne Zuhilfenahme irgendwelcher weiterer Ressourcen. Um Sie nicht in Versuchung zu führen, liefere ich allerdings nicht den Quellcode, sondern nur ein Kompilat aus - sogar eines für Windows. Wie man das macht, wird Ihnen im folgenden "Freebasic"-Teil verraten.



Für diese Windows-Version sollten Sie allerdings einen einigermaßen leistungsfähigen Rechner haben (Pentium4 ab 2 GHz, Athlon XP ab Quantispeed 1500). Der Grund liegt darin, dass Windows zwei Grafiksysteme besitzt, ein einfaches, sehr langsames, vor allem, was Doublebuffer-Grafiken angeht. Und ein schnelles, das s.g. DirectX-System, das hier aber nicht benutzt wird.

#### Die Bedienung:

- Cursortasten: Bewegung in x1/x2-Ebene
- "v" und "b": Drehung des Turms nach rechts und links
- "g": Zugseil nach oben, "t": Zugseil nach unten.
- "d": Ausleger ausfahren, "ä": Ausleger einfahren.
- "n","m": Drehen des Greifers
- "x","c": Schliessen, Öffnen des Greifers
- "q": Schliessen

Viel Spass!