

PROGRAMMIEREN FÜR OMA

Eine Einführung ins Programmieren von Anfang an

Christof Schatz

Email: schatz@askos.de

www.askos.de

PROGRAMMIEREN FÜR OMA

Einleitung

Hintergrund dieses Tutorials ist es, dass sich in den letzten zehn Jahren zunehmend eine Lücke aufgetan hat zwischen "normalen" Usern und Programmierern. Während in den Achtzigerjahren Grundwissen in Programmierung noch bei mindestens 20% der Nutzer vorhanden war, ist dies heute fast nur noch bei Profi-Programmierern der Fall. Die Spezie des "Hobby-Programmierers" ist zwar nicht ausgestorben, aber sie schwindet dahin. Dies weitert nicht nur die Lücke zwischen "Wissenden" und "Unwissenden" bezüglich IT, dies bedeutet auch einen Verlust einer elementaren Nutzungsart von Computern. Als Statistiker z.B. kann ich heute nicht mehr einfach mal zu meinen Studenten sagen: "Lasst uns mal dazu kurz ein kleines Programm schreiben". Denn: Wer kann das schon?

Der Grund dafür ist nicht, dass die jüngeren Generationen weniger Talent mitbringen, als noch zehn oder zwanzig Jahre zuvor. Der Grund ist hauptsächlich die Fortentwicklung der PC's vom einfachen Homecomputer zur Hochleistungsworkstation. Das Programmieren ist dadurch nicht per se schwieriger geworden - nein, im Gegenteil. Aber der Einstieg ist *verwirrender* geworden. Ein Laie, der Programmieren lernen möchte, hat heute ein ähnliches Problem, wie ein Windows-Anfänger, der in der Online-Help vergeblich nach einem Hinweis sucht, wie man ein Programm startet. Dass dies (in Windows) über einen Doppelklick oder über das Start-Menü funktioniert, kann ihm höchstens mündlich jemand erklären. In der Online-Help "ertrinkt" der Anfänger in einer Flut von Funktionalität.

So ähnlich ergeht es jemandem heute, der einfach mal wissen will, was sich hinter diesem "Programmieren" verbirgt und der niemanden hat, der ihn persönlich die ersten Schritte voranbringt. In Einführungen zur Programmierung werden heute meist Übersichten über sämtliche Sprachen und Entwicklungssysteme gegeben, auf dass sich der Anfänger erst einmal entscheiden möge, *wo* er anfangen möchte. Doch genau das kann der Anfänger nicht leisten - dazu müsste er von diesen ganzen Sprachen und Systemen schon einen Begriff haben. Oft orientieren sich heutige (meist exzellente) Einführungen an den jeweils modernen Sprachen und Programmierwerkzeugen. Vor zwanzig Jahren war das kein Problem, da die Werkzeuge noch so primitiv waren, dass ein Anfänger es schnell begreifen konnte. Aber inzwischen haben sich die Werkzeuge zu komplexen Maschinenanlagen entwickelt, bei denen man als Anfänger eigentlich nur errahnen, aber nicht wirklich etwas begreifen kann.

Daher geht dieses Tutorial einen völlig anderen Weg. "Ganz vorne anfangen" ist das Ziel dieses Tutorials. So dass es auch noch Oma verstehen kann und könnte. Es beginnt mit einem sehr alten, sehr einfachen System, möglichst anschaulich und konkret und in sehr kleinen Schritten. Um dann in späteren Kapiteln auf die aktuellen Sprachen und Systeme zu sprechen zu kommen, wenn der

Anfänger schon keiner mehr ist und einen Begriff davon hat, was "Programmieren" überhaupt heisst. Allerdings: "Verständlich" heisst nicht "leichte Lektüre". Sie werden schon gefordert werden, keine Sorge...

Dieses Tutorial entsteht so nach und nach bei mir während der Arbeit und ist noch sehr unvollständig. Die ersten beiden von vier oder fünf Teilen sind einigermaßen fertig, der dritte ist in Arbeit (zwei Kapitel sind hier schon dabei).

[Weiter](#)

Teil 1: Einfaches Programmieren

<u>Der Commodore C16</u>	Seite 5
<u>Speicher</u>	Seite 11
<u>Programme besser editieren</u>	Seite 18
<u>Input und Sprünge</u>	Seite 20
<u>If-Anweisung</u>	Seite 25
<u>Mehr Komfort: CLEAR, RUN, END, STOP, LOAD, SAVE</u>	Seite 28
<u>Strings</u>	Seite 33
<u>Bildschirmpositionierung</u>	Seite 40
<u>Unterprogramme</u>	Seite 43
<u>Farbe und SOUND</u>	Seite 50
<u>Schleifen</u>	Seite 53
<u>Arrays, grössere Programme und Bugs</u>	Seite 56
<u>Die Millionärsshow und der DATA-Befehl</u>	Seite 62
<u>Bits, Bytes und BASIC</u>	Seite 68
<u>Ereignisse</u>	Seite 72
<u>Hochauflösende Grafik</u>	Seite 74

Der Commodore C16

Installation

Wie Sie Ihren PC einschalten und hochfahren, das wird hier nicht erklärt. Wenn Sie das nicht geschafft hätten, würden Sie diesen Text hier nicht lesen. Um aber mit Ihrem PC mal etwas ganz anderes machen zu können, nämlich ihn ganz einfach zu programmieren, müssen Sie sich ein kleines Programm herunterladen, das Ihnen diesen Zugang zu Ihrem PC verschafft. Dazu eine kleine Erklärung.

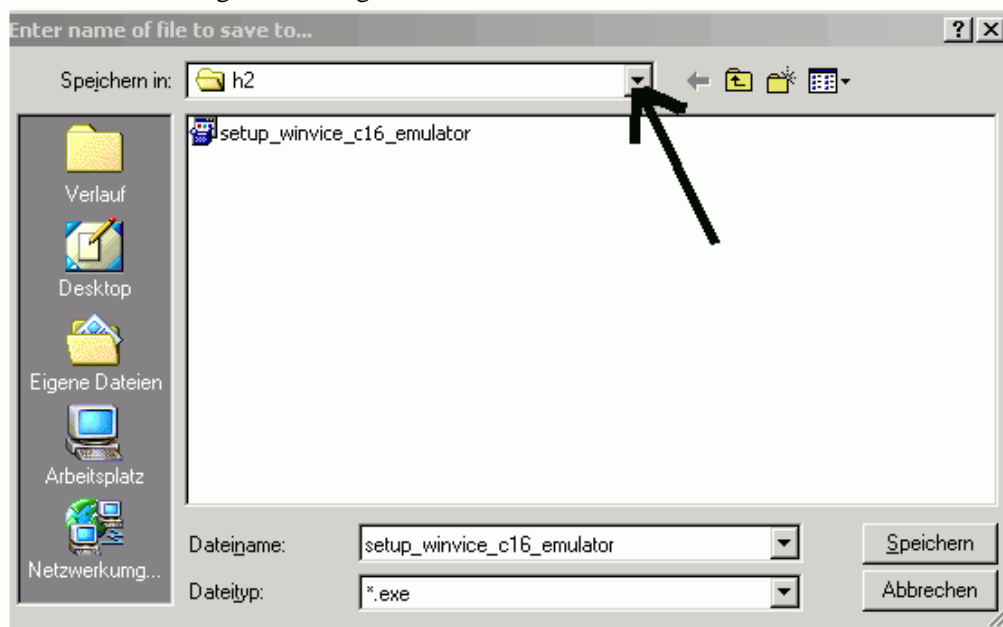
Vor etlichen Jahren gab es eine Firma namens Commodore, die Computer herstellte. Das waren ziemlich beliebte Computer. Sie waren noch viel einfacher als der PC, vor dem Sie heute sitzen. Und es machte sehr viel Spass, sie zu programmieren. Einer von diesen Commodore-Computern, ein Einsteiger-Computer, war der C16.

Weil Ihr PC sehr viel mehr kann, als der schickste Commodore jemals konnte, kann er natürlich erst recht alles, was ein C16 kann. Tolle Sache, nicht?

Was es braucht, ist ein kleines Programm, das dafür sorgt, dass Ihr PC sich genauso verhält wie ein C16, also die C16-Funktionen in PC-Funktionen umwandelt. So etwas nennt man einen *Emulator*. (Müssen Sie sich nicht merken.)

Diesen C16-Emulator müssen Sie sich erst installieren. [Klicken Sie hier.](#)

- Drücken Sie dann auf "Speichern als", bzw. "Save as" und OK
- Nun kommt der folgende Dialog:



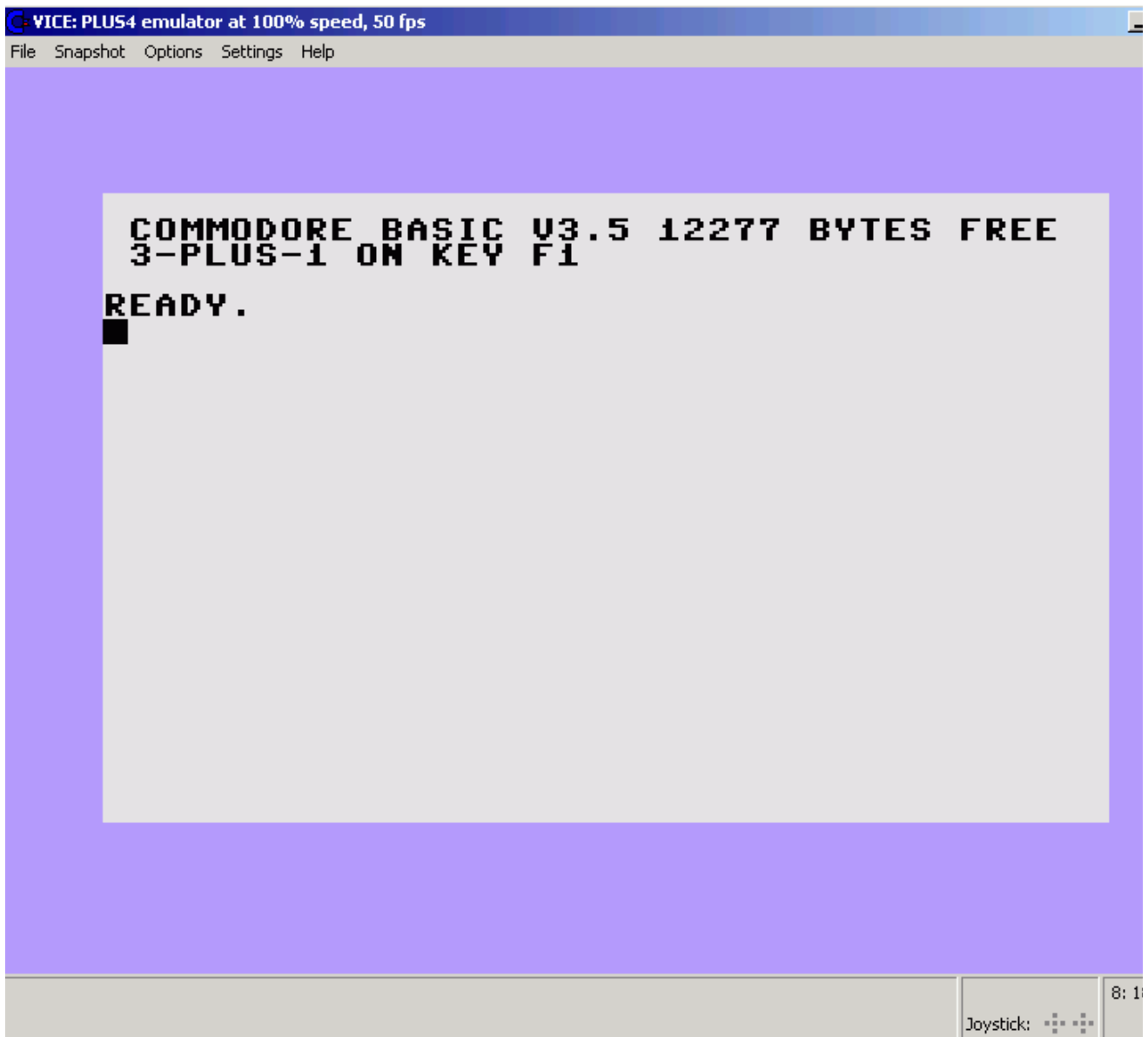
Klicken Sie auf den Pfeil nach unten, wie im Bild markiert

- Nun klappt eine Liste nach unten auf. Wählen Sie daraus die Zeile "Desktop", wie hier im Bild gezeigt:



- Dann klicken Sie unten im Dialog auf "OK".
- Je nach Geschwindigkeit Ihrer Internet-Verbindung dauert es nun einige Sekunden bis Minuten, bis die Programmdatei heruntergeladen ist.
- Wenn dies geschehen ist, ist auf Ihrem Desktop (die Bildschirmfläche im Hintergrund, auf der meist links viele kleine Bildchen angeordnet sind) ein neues Bildchen hinzugekommen namens "setup_winvice_c16_emulator". Klicken Sie darauf doppelt.
- Das Setup-Programm startet. Klicken Sie auf "Weiter".
- Dann kommt ein Hinweis bzgl. der Autoren dieses hervorragenden kleinen Programms. Ich bitte Sie, sich das kurz durchzulesen. Dann klicken Sie auf "Weiter".
- Und gleich nochmals auf zweimal auf "Weiter".
- Jetzt werden die Dateien entpackt und installiert.
- Dann klicken Sie auf "Installiertes Programm starten".
- Das war's.
- Ab jetzt finden Sie den C16-Emulator im Startmenü unter "C16-Emulator" und "C16-Emu".

Sie sollten folgendes Bild vor sich sehen:



Nun haben Sie den Commodore C16 gestartet. Wunderbar. Mal sehen, was wir damit machen können.

Erste Schritte auf dem C16

Das erste Wort

Klicken Sie das Fenster an, dann tippen Sie ein paar KLEINbuchstaben. Z.B. "anton". Es erscheint "ANTON". Aha. Kleinbuchstaben gibts nicht. Aber ansonsten ist erstmal alles wie in einem Textprogramm.

Nun drücken wir die RETURN (ENTER-)Taste für einen Zeilenwechsel. Es erscheint das folgende Bild:

```

COMMODE BASIC V3.5 12277 BYTES FREE
3-PLUS-1 ON KEY F1

READY.
ANTON
?SYNTAX ERROR
READY.

```

Das gefällt uns gar nicht. Da erscheint Text, den wir nicht getippt haben! Und den wir nicht verstehen!

Nun, "?SYNTAX ERROR" heisst in etwas übersetzt: "Das verstehe ich nicht". Der Computer sagt, dass er etwas nicht versteht. Nämlich das, was wir getippt haben: "ANTON". Wie auch. Das verstehen wir ja auch nicht. Das soll ja auch nichts heissen. "ANTON" war ja bloss irgendwas, Unsinn. Wir lernen also: Das ist anders als auf einer Schreibmaschine oder in einem Textprogramm: Der Computer versucht das, was wir schreiben, zu verstehen! Und wenn er es nicht versteht, schreibt er "?SYNTAX ERROR".

Lustige Zeichen

"READY" heisst, dass der Computer nun für ein neues Wort bereit ist. Aber welche Worte versteht denn der Computer? Bevor wir das ausprobieren, machen wir uns noch etwas mehr mit der Tastatur vertraut. Was gibt es denn sonst noch ausser Grossbuchstaben? Tippen wir auf der Tastatur mal einen Bindestrich. Dann kommt ein "?". Aha. Nun drücken wir die Löschaste rechts oben (BACKSPACE, die mit dem Pfeil nach links.) Das ist wie gewohnt.

Wenn wir nochmal die BACKSPACE-Taste drücken, kann es passieren, dass die Schreibmarke, der Cursor, an den rechten Bildschirmrand hüpf. Da können wir sie nicht gebrauchen. Aber das ist nicht schlimm. Wir geben einfach ein paar Leerzeichen ein, dann hüpf er schon wieder zurück.

Bei unserem Programmier-Computer C16 sind die Tastenfunktionen etwas anders, als gewohnt. Das merken wir besonders, wenn wir die Shift-Taste (Grosstellaste) und die Ziffern, bzw. Buchstaben betätigen. Da kommen dann lustige Zeichen zutage, Striche, Spielkartensymbole, Kreise.

```

●○-| | | | |
READY.
◆◆-| | | | |
?SYNTAX ERROR
READY.
◆◆-X| | | | |
?SYNTAX ERROR
READY.
!"#$%&'(>↑↓←
?SYNTAX ERROR
READY.

```

Die Zeichen "+", "*", "-", und "/" werden wir demnächst noch häufiger brauchen. Suchen Sie diese Zeichen einmal auf der Tastatur! Sie werden dabei sehen, dass nicht viel passieren kann, wenn Sie herumprobieren.

Haben Sie sie gefunden? Richtig:

- Plus-Taste des PC's: -
- ü-Taste des PC's: +
- #-Taste des PC's: *
- Bindestrich-Taste des PC's: /
- Und zur Vollständigkeit: Shift-Bindestrich ergibt das ? Das ist ganz besonders wichtig.

Der C16 als Taschenrechner

Nun geben wir einmal folgendes ein:

? 1+1

Und drücken dann die ENTER-Taste. Nicht vergessen: Für '+' müssen wir die ü-Taste drücken! Und für das Fragezeichen Shift-Bindestrich! Wenn's nicht geklappt hat: Einfach nochmal. Oder die BACKSPACE-Taste benutzen!

Das Ergebnis sollte so aussehen:

? 1+1
2

READY

Aha! Kein Syntax Error! Stattdessen eine "2"! Der Computer hat uns eine 2 zurückgegeben! Naheliegenderweise das Rechenergebnis. Offenbar hat er also "1+1" ausgerechnet.

Warum hat er das gemacht? Nun, dies hat das Fragezeichen am Anfang bewirkt. Der Computer rechnet, wenn zuerst ein Fragezeichen eingegeben wird und dann die Rechnung. So einfach ist das. Wollen wir ausrechnen, ob 227 durch 9 teilbar ist, dann rechnen wir

? 227/9

Das "/" steht für die Division. Am Ergebnis, 25.222222 sehen Sie, dass dies offenbar nicht der Fall ist.

Genauso können wir multiplizieren. Hier können wir gleich etwas Nützliches machen: Rechnen Sie doch mal alle Zweierpotenzen aus! Also 2, 2*2, 2*2*2, 2*2*2*2 usw. Die werden wir etwas später noch brauchen.

Die Rechenfunktionen des C16

Wenn Sie sonst mit Mathe nicht viel am Hut haben, werden Ihnen die Grundrechenarten ausreichen. Aber sind Sie Schüler, Student, Ingenieur o.ä., dann brauchen Sie vielleicht ab und zu auch einen Logarithmus oder einen Sinus. Welche Operatoren und Funktionen hat denn der C16 eingebaut? Viel können das ja nicht sein, schliesslich ist auf der Tastatur nichts von einem Sinus oder einem Logarithmus zu sehen.

Nun, geben wir mal folgendes ein:

?SQR(2)
1.41421356

READY

Das SQR schreiben Sie ganz normal als Wort aus. Aha! Und Sie erhalten die Wurzel aus 2! Beim C16 gibt es also im Gegensatz zum Taschenrechner mehr Funktionen als es Tasten gibt. Man schreibt die Funktionsnamen einfach aus. Daher ist beim Computer im allgemeinen die Schreibmaschinentastatur so wichtig: Es gibt einfach viel zu viel Funktionen, als dass man für jede eine eigene Taste einbauen könnte!

Hier eine Übersicht über die mathematischen Operatoren (wie + und -) und Funktionen (wie SQR) des C16:

+	Taste ü	Addition
-	Taste +	Subtraktion
*	Taste #	Multiplikation
/	Taste -	Division
Pfeil nach oben	Shift-Null	Potenz ("hoch")
SQR	Ausschreiben	Quadratwurzel
EXP	Ausschreiben	Exponentialfunktion
LOG	Ausschreiben	Natürlicher Logarithmus
SIN	Ausschreiben	Sinus
COS	Ausschreiben	Sinus
TAN	Ausschreiben	Sinus
ATN	Ausschreiben	Arcus Tangens
ABS	Ausschreiben	Betrag
SGN	Ausschreiben	Vorzeichen: 1, wenn Argument >0, -1, wenn Argument <0,0, wenn Argument=0
INT	Ausschreiben	Abrunden (INT(2.5)=2)

Die Funktionen Arcus Cosinus und Arcus Sinus sind nicht vorhanden, da sie relativ leicht mit Hilfe des Arcus Tangens berechnet werden können.

Speicher

Was ist ein Computer? Sie meinen, das sei schwer zu beantworten? Überhaupt nicht. "To compute" kommt aus dem Englischen und heisst "Rechnen". Genauer gesagt: "Rechnungen durchführen", die Tätigkeit des Rechnens also. Ein Computer ist ein Rechner. Um mit Zahlen zu rechnen.

Das sieht man ihm heute zugegebenermassen nicht mehr so an. Man muss schon ganz schön herumsuchen, bis man auf einem heutigen PC irgendwo etwas findet, mit dem man rechnen kann. Alles mögliche kann man: Surfen, Zeichnen, Malen, Texte schreiben, Musik hören. Aber rechnen?

Nun, man kann es so betrachten: Bei all diesen Tätigkeiten - Surfen, Zeichnen usw. - wird der Computer sozusagen "missbraucht". Was er im Inneren macht, wenn wir Texte schreiben, ist, in einer gewissen Form zu rechnen.

Bei den kleinen Computern wie dem C16 merkte man das schon eher. Die waren zu klein, um so ohne Weiteres andere Dinge wie Textverarbeitung oder Musik "rechnen" zu können. Man war mit dem technischen Gerät als solches stärker konfrontiert, als heute, wenn man es anschaltete. Das haben Sie gerade gesehen.

Da ein Computer ein Rechner ist, müsste der Taschenrechner ein Taschencomputer sein. Ist das so? Nein. Das heisst, nicht unbedingt. Denn in den letzten fünfzig Jahren (so lange existieren Computer ungefähr), hat man sich darauf geeinigt, nur die Rechner "Computer" zu nennen, die "programmierbar" sind. Ein Computer ist also ein **programmierbarer Rechner**. Und das war's auch schon. Was "programmierbar" heisst, werden wir noch sehen.

Manuelles Rechnen

Unser Leben hat viel mit Rechnen zu tun. In der Schule müssen Aufgaben in Mathe und Physik gelöst werden, im Haushalt müssen die Ausgaben ausgerechnet werden, mit Mieten und Aktien und Wohnungskauf muss rechnerisch umgegangen werden, in der Arbeit müssen Statistiken ausgerechnet und Rechnungen bearbeitet werden usw. An Anwendungen für unseren kleinen C16 sollte es also nicht fehlen.

Wir wollen uns aber eine andere Aufgabe stellen. Eine der berühmtesten Aufgaben der Mathematik. Wir wollen Primzahlen ausrechnen.

Was ist eine Primzahl? Das ist eine Zahl, die nur durch sich selbst und durch Eins ohne Rest teilbar ist. Und die Frage ist: Wieviel solche Primzahlen gibt es eigentlich?

Schauen wir uns mal die ersten Primzahlen an. 1 ist eine Primzahl. Denn 1 ist nur durch sich selbst (1) und durch 1 teilbar. 2 ist auch eine Primzahl. 3 auch: 3 ist durch 1 und durch 3 teilbar. 4 ist *keine* Primzahl. $4/2=2$. Das heisst, 4 ist ausser durch 4 und durch 1 auch durch 2 ohne Rest teilbar. 5 ist wieder eine Primzahl. 6 nicht. Keine gerade Zahl kann eine Primzahl sein, denn sie ist natürlich immer durch 2 teilbar. 7 ist wieder ein Primzahl. 9 ist keine Primzahl, denn sie ist auch durch 3 teilbar. 11 ist dafür wieder eine Primzahl.

Die ersten Primzahlen sind also:

1, 2, 3, 5, 7, 11

Fast jede ungerade Zahl. Aber eben nicht jede. Bei weitem nicht jede. Schauen wir uns 47 bis 53 an. Von 47 bis 53 sind nur 47 und 53 Primzahlen. Und 55 und 57 dann schon wieder nicht.

Gibt es eine Formel, die uns sagt, welche Zahl eine Primzahl und welche nicht? Das wissen wir nicht. Aber die Mathematik hat in ihrer viertausendjährigen Geschichte bisher keine gefunden. So gibt es nur einen Weg, herauszufinden, ob z.B. 221 eine Primzahl ist: Wir müssen es ausprobieren. Alle möglichen Teiler durchprobieren. Machen Sie das mal mit dem C16! Das ist eine gute Übung, um mit ihm warmzuwerden! Es ist nicht soviel Arbeit, wie es auf den ersten Blick aussieht.

Ob 221 eine Primzahl ist oder nicht, verrate ich erst am Schluss dieses Kapitels. Aber wenn Sie hier engagiert gerechnet haben, werden Sie gemerkt haben, dass Sie eine Zahl immer wieder eintippen müssen: 221. Ein bisschen blöd. Man sollte dem Rechner sagen können: "Merk dir mal die 221". Der C16 müsste dafür ein Gedächtnis haben, einen elektronischen Speicher. Manche Taschenrechner haben ja so etwas, eine Speichertaste. Hat der C16 auch so eine?

Hat er. Dazu brauchen wir neben dem "?" aber nun ein weiteres Wort, das er versteht. Dieses Wort ist LET. Mit LET kann man eine Zahl speichern:

```
LET A=221  
READY
```

Das =-Zeichen finden Sie auf der Akzent-Taste links neben BACKSPACE. Und: ENTER-Taste nicht vergessen!

Der Name des Speichers ist A. Man nennt das auch eine *Variable*. Aber was nun? Nichts wird angezeigt! Nein. Ist ja auch klar. Wir haben ja auch nichts gerechnet. Nur abgespeichert.

Schauen wir aber nun, ob sich der C16 die Zahl auch behalten hat. Tippen wir ein:

```
? A  
221  
  
READY
```

Tatsächlich! Hat er! Und "A" geht schneller als "221"! Nun können wir also viel leichter tippen:

```
? A/3  
73.6666667
```

```
READY  
? A/5  
44.2
```

```
READY
```

usw.

Nun wissen wir auch, warum der C16, anders als ein Taschenrechner, auch Buchstabentasten braucht. Wegen der Speicher.

Hat der C16 nur einen Speicher? Wir können ja versuchen, auch mal in B was abzuspeichern:

```
LET B=11
```

READY

? B

11

READY

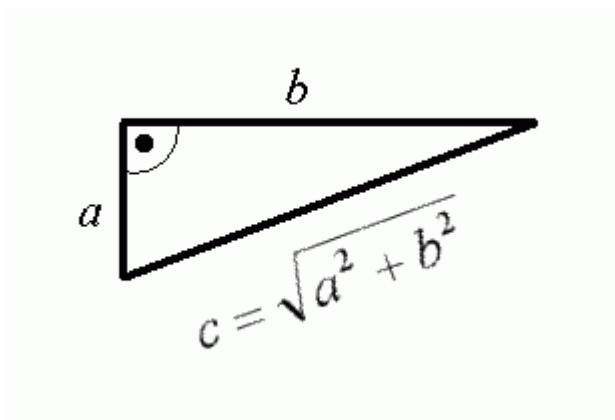
Prima. Klappt offenbar. Es ist sogar so, dass man den Speichern nicht die Buchstaben des Alphabets nach zuweisen muss, sondern ihnen eigene Namen nach Gutdünken geben kann. Z.B. kann

LET PI=3.14159

ganz nützlich sein.

Rechnungen speichern

In der Schule und später im Studium oder in technischen Berufen hat man es immer wieder mit *Formeln* zu tun. Formeln sind mathematische "Kochrezepte". Sie enthalten Buchstaben, für die wir bekannte Zahlen einsetzen und sie liefern uns dann die gesuchte Zahl. Wie lang ist die Grundseite eines rechtwinkligen Dreiecks? $c = \sqrt{a^2 + b^2}$, a und b sind die Längen der Schenkel. Das "^" steht für "hoch". Im C16 wird dafür allerdings ein Pfeil nach oben verwendet, der über der Null steht. Also: Shift und Null gleichzeitig liefert den Pfeil, der auf dem C16 "hoch" bezeichnet. Und wir schreiben hier für diesen Pfeil "^". Es gibt noch andere Formeln: Wie schnell fährt ein Auto (gemessen in km/h), das 320 m in 5 s zurückgelegt hat? $v = s / t * 3.6$, $s=320\text{m}$, $t=5\text{s}$.



Nehmen wir an, wir wollen eine Tabelle erstellen, die zu jeder Länge von a und b die Länge von c angibt, der Grundseite des Dreiecks:

a	b	c
10cm	10cm	?
20cm	10cm	?
30cm	10cm	?
40cm	10cm	?
50cm	10cm	?
10cm	20cm	?
20cm	20cm	?
30cm	20cm	?
40cm	20cm	?
50cm	20cm	?
10cm	30cm	?
20cm	30cm	?
30cm	30cm	?
40cm	30cm	?
50cm	30cm	?
10cm	40cm	?

20cm	40cm	?
30cm	40cm	?
40cm	40cm	?
50cm	40cm	?

usw.

Das gibt eine nette Rechnerei, selbst, wenn wir auf die Idee kommen, den Wert von b jeweils abzuspeichern:

```
LET B=10
```

```
READY
? SQR(10^2+B^2)
14.142136
```

```
READY
? SQR(20^2+B^2)
22.36067
```

```
READY
? SQR(30^2+B^2)
31.6227766
```

```
READY
```

```
...
...
...
```



Das ist schon ein nerviges Getippe! (Vor allem, weil wir jetzt am Anfang noch nicht genau wissen, wo die ganzen Sonderzeichen sind. Das Plus ist auf der β -Taste und das Fragezeichen über dem Bindestrich...) Achtung! Da kann's dann auch noch einen ganz üblen Vertipper geben: Wenn man Shift+ β eingibt, erscheint ebenfalls ein Plus. Es ist allerdings ein bisschen grösser und es "funktioniert" in der Formel nicht wie ein Plus. Der C16 meldet dann "Syntax Error". Und man sieht's einfach nicht, da ja das Plus eigentlich genau da steht wo es stehen soll - es ist nur das falsche.

Bald wünschen wir uns, dass wir diese elende Formel selbst irgendwie abspeichern könnten und nicht dauern immer wieder das "SQR(" und dann "^2" usw. eintippen müssten. Abgesehen davon, dass man sich doch dabei sehr leicht vertippen kann.

Wir tippen die Formel ein, unter Benutzung von A und B. Aber halt - vor das "?" schreiben wir eine Zahl, und zwar eine Eins:

```
1 ? SQR(A^2+B^2)
```

Achten Sie darauf, zwischen dem 1, ? und dem SQR jeweils ein Leerzeichen zu schreiben.

Die 1 ist also neu. Was ist passiert? Gar nichts, wie es scheint. Weniger als nichts. Der Computer hat kein Ergebnis ausgespuckt. Er hat nicht mal "READY" geschrieben. So was. Was soll denn das?

Nun: Der Computer hat die Zeile nur gespeichert. Und zwar im Speicher Nummer Eins. Das bedeutet die Eins vor der Anweisung. Er hat nicht nur eine Zahl, er hat die ganze Rechnung gespeichert. Wenn wir den Befehl LIST eingeben, dann können wir uns den Inhalt dieses Rechnungsspeichers wieder ansehen:

```
LIST
```

```
1 PRINT SQR(A^2+B^2)
```

```
READY
```

Schon wieder eine Überraschung. Da ist was anders. Wenn wir das, was hinter der 1 kommt, mit dem vergleichen, was wir eben oben eingetippt haben, dann ist das "?" durch ein ganzes Wort ausgetauscht. PRINT steht jetzt da. Das liegt daran, dass das "?" nur eine Abkürzung ist, für einen sehr oft gebrauchten Befehl (wir haben ihn ja schon oft gebraucht!), den PRINT-Befehl. Der eben sagt: "Drucke die folgende Rechnung auf dem Bildschirm aus!".

LIST bedeutet also, dass wir uns alle gespeicherten Rechnungen ausgeben lassen. Geben wir mal noch eine zweite Rechnung ein. Und zwar hinter eine "Zwei". Wir sagen: Wir speichern die Rechnung in "Zeile 2".

```
2 ? SQR((A+10)^2+B^2)
```

Wir erhöhen also A um 10 und rechnen dann wieder die Formel aus. Das ist die zweite Zeile. Und dann geben wir LIST ein. Jetzt sehen wir:

```
LIST
```

```
1 PRINT SQR(A^2+B^2)
```

```
2 PRINT SQR((A+10)^2+B^2)
```

```
READY
```

Aha. Jetzt sind schon zwei Rechnungen hintereinander gespeichert. Eine in Zeile 1, die andere in Zeile 2. Und was machen wir jetzt damit? Wir wollen die Rechnungen ausführen! Die Zahlen in A und B sind ja schon gespeichert. Wir geben nun den Befehl GOTO ein. Mit GOTO führt man die gespeicherten Rechnungen hintereinander aus. Wobei man hinter GOTO die erste auszuführende Rechnung angibt (deren Zeile), d.h hier die 1

```
GOTO 1
```

```
0
```

```
14.1421356
```

"GOTO 1" hiess: Führe nun diese beiden Rechnungen aus. Zuerst die hinter der "1" und dann die hinter der "2". Aber es kommt nicht das heraus, was wir eigentlich wollten. "Null" macht keinen Sinn. Aber seien wir mal ehrlich: Das kann ja auch gar nicht sein. Wir haben ja in A und B noch gar nichts abgespeichert. Denn, man merke sich: Sobald wir *Rechnungen* abspeichern, gehen die

Zahlenspeicher verloren. Das mag dumm sein, aber das ist nun mal so. Also, wir speichern also auch in A und B eine Zahl.

```
LET A=10
```

```
READY
```

```
LET B=10
```

```
READY
```



```
READY.
LET A=10
READY.
LET B=10
READY.
GOTO 1
14.1421356
22.3606798
READY.
█
```

Wow! So eine kurze Eingabe und soviel Ergebnis! Und so schnell! Wir haben nicht nur die Lösung für eine Rechnung erhalten, sondern für zwei gleichzeitig! Und so nebenbei: Wir haben gerade zum ersten Mal programmiert! Unsere beiden gespeicherten Rechnungen, das ist unser erstes (zweizeiliges) Programm!

Gespeichertes editieren

"Editieren" heisst "verändern". Wie verändern wir gespeicherte Programme? Das fängt immer mit der Eingabe von "LIST" an.

```
LIST
```

```
1 PRINT SQR(A^2+B^2)
2 PRINT SQR((A+10)^2+B^2)
```

```
READY
```

Ändern müssen wir das Programm vor allem dann, wenn es einen Fehler enthält. Dann können wir mit den Cursortasten einfach über den Bildschirm an die Stelle laufen, die wir verändern wollen. Allerdings überschreiben wir beim Tippen das Alte immer. Ändern wir z.B. einmal die Zeile 2 in "2 PRINT SQR((A+1X)^2+B^2)" um, d.h. eine "0" wird zu einem "X". Wir laufen einfach, bis der Cursor über der passenden Null steht und schreiben ein "X". Damit sind wir aber noch nicht fertig. Der C16 hat die neue Zeile nämlich noch nicht gespeichert. Dazu müssen wir immer noch ein ENTER eingeben. Das ist wichtig und das vergisst man leicht. Nach jeder Änderung noch ein ENTER eingeben! So, jetzt ist unsere Änderung gespeichert. Sie war natürlich Unsinn. Was soll der Computer mit der Zahl "1X" anfangen? Kontrollieren wir die Änderung zuerst, indem wir LIST eingeben:

```
LIST
```

```
1 PRINT SQR(A^2+B^2)
2 PRINT SQR((A+1X)^2+B^2)
```

```
READY
```

Nun schauen wir, was noch in unseren Speichern steht:


```
? A  
0
```

```
READY  
? B  
0
```

```
READY
```

Mist. Das müssen wir also nochmal neu eingeben. Also: LET A=10+ENTER, LET B=10+ENTER. Und nun: GOTO 1.

Das Ergebnis? Syntax Error! Der C16 hat entdeckt, dass da nur Blödsinn steht. Also gehen wir wieder mit den Pfeiltasten zurück und ändern das X wieder in eine Null. Danach ENTER nicht vergessen!

Die Frage ist: Können wir auch Zeichen einfügen? Können wir. Und zwar mittels Shift-Backspace. Die Backspace-Taste ist ja die Taste rechts oben mit dem Pfeil nach links. Wenn dazu nun auch noch die Shift-Taste drücken, rutscht der Inhalt der restlichen Zeile nach rechts. Wenn wir also drei Zeichen einfügen wollen, müssen wir dreimal Shift-Backspace drücken.

Übungsaufgabe1:

Starten Sie das Programm mit den anderen Beträgen der Tabelle. Also A=30 und B=10 und A=50 und B=10 und A=10 und B=20 usw. Denken Sie daran: Wenn Sie am Programm nichts verändern, bleiben die Inhalte der Zahlenspeicher erhalten.

Übungsaufgabe2:

Erweitern Sie das Programm so, dass es insgesamt fünf Werte ausgibt: Den Wert für A, A+10, A+20, A+30 und A+40. Dann müssen Sie das Programm nur noch fünfmal starten, um die Ergebnisse von 25 Rechnungen zu erhalten! Ein Tipp: Sie können eine ganze Zeile *kopieren*, indem Sie einfach ihr eine neue Zeilennummer geben und dann ENTER drücken. Gehen Sie z.B. in Zeile 2, machen aus der 2 eine 3 und drücken ENTER. Wenn Sie nun einen LIST-Befehl eingeben, werden Sie feststellen, dass Ihr Programm nun drei Zeilen hat!

Programme besser editieren

Vielleicht sind Sie schon bei den ersten Gehversuchen ein bisschen verzweifelt? Wie kriege ich eine Programmzeile wieder weg? Wie lösche ich das ganze Programm? Kann ich mehrere Programme gleichzeitig haben? Wie kopiere ich eine Zeile? Diesen Fragen wollen wir hier nachgehen.

Löschen von Zeilen

Einfach Zeilennummer+ENTER eingeben. Also eine Leerzeile sozusagen.

Löschen des ganzen Programms

NEW+ENTER eingeben.

Eine Zeile kopieren

An einem Beispiel.

```
LIST
1 PRINT SQR(A^2+B^2)
```

READY

Ich will eine ähnliche Zeile wie Zeile 1 schreiben. Also gehe ich zuerst mit dem Cursor in Zeile 1. Dann ändere ich die Zeilennummer in "2" und drücke ENTER. Anschliessend gebe ich ein LIST ein und schaue das Ergebnis an:

```
LIST
1 PRINT SQR(A^2+B^2)
2 PRINT SQR(A^2+B^2)
```

READY

Und nun kann ich Veränderungen an Zeile 2 vornehmen.

STOP und mehrere Programme

Sie müssen nicht immer genau die Zeilennummern benutzen, die hier in den Beispielen verwendet werden. Es ist eigentlich egal, wie die Zeilen numeriert sind, Hauptsache, die Nummern sind eindeutig. Folglich kann man auch mehrere Programme eingeben, indem man sie in verschiedene Zeilenbereiche schreibt:

```
10 LET A=1
20 LET B=2
30 PRINT A+B
40 STOP
100 INPUT A
110 INPUT B
```

```
120 PRINT SQR(A^2+B^2)
130 PRINT SQR((A+10)^2+B^2)
```

Hier sind zwei Programme. Das erste endet bei Zeile 30 und das zweite beginnt bei Zeile 100. Damit das erste nicht einfach weiterläuft, müssen wir einen STOP-Befehl dazwischen einfügen.

Wie starte ich die Programme? Nun, mit einem GOTO und der Anfangszeile des jeweiligen Programms. In diesem Beispiel: "GOTO 10" für das erste und "GOTO 100" für das zweite.

Komfortableres LIST

Es kann nun passieren, dass Sie mehr Programmzeilen im Speicher haben, als auf den Bildschirm passen. Was dann? Nun, Sie können das Listing begrenzen lassen:

- "LIST 100-" listet ab Zeile 100
- "LIST -100" listet bis Zeile 100
- "LIST 100" listet ausschliesslich Zeile 100
- "LIST 10-40" listet die Zeilen 10 bis 40

So, damit müssten Sie vorerst mal zurechtkommen. Auf geht's zu neuen Programmen!

Input und Sprünge

INPUT

Bis jetzt hat uns unser abgespeicherte Zweizeiler ja noch nicht soviel genutzt. Es ist zwar nett, mit "GOTO 1" auf einen Schlag zwei Ergebnisse zu bekommen, aber das Eintippen von "LET A=10, GOTO 1, LET A=20, GOTO 1, LET A=30, GOTO 1" ist immer noch ziemlich umständlich.

Etwas besser wird es, wenn wir die LET-Anweisungen mit ins Programm integrieren. Denn wir können auch andere Anweisungen, nicht nur die PRINT-Anweisungen abspeichern. Das sähe dann so aus:

```
1 LET A=10
2 LET B=10
3 PRINT SQR(A^2+B^2)
4 PRINT SQR((A+10)^2+B^2)
```

Zur Veränderung eines Werts müssen wir allerdings erst "LIST" eingeben, dann zur Stelle gehen, den Wert überschreiben, in die nächste Leerzeile gehen und dann GOTO eingeben - auch nicht gerade einfach.

Für die Aufgabe, aus einem Programm Werte vom Benutzer entgegenzunehmen, gibt es den INPUT-Befehl. Will man, dass das Programm den Benutzer zur Eingabe eines Werts der Variablen A auffordert, schreibt man "INPUT A". Wenn Sie noch nicht ganz verstanden haben, was das soll: Hier ein kleines Beispielprogramm, das zwei Werte vom Benutzer entgegennimmt und deren Summe berechnet - nicht sehr nützlich, aber verständlich.

```
10 INPUT A
20 INPUT B
30 PRINT A+B
```

Wenn Sie das ausprobieren, sehen Sie, dass das Programm an der Stelle des Inputs einfach anhält und ein Fragezeichen zeigt. Der Benutzer muss das Programm schon kennen, um zu wissen, dass er jetzt eine Zahl eingeben muss - und welche Zahl er eingeben muss. Aber auch da gibt's Verbesserungsmöglichkeiten. Die kommen später.

Ergänzen wir also unser Pythagoras-Programm, indem wir die LET-Zeilen durch Input-Zeilen ersetzen:

```
1 INPUT A
2 INPUT B
3 PRINT SQR(A^2+B^2)
4 PRINT SQR((A+10)^2+B^2)
```

Probieren Sie es aus. Sie werden feststellen: Jetzt geht's wirklich schneller. Einfach die beiden Eingangszahlen eingeben und schon hat man die Ergebnisse für die nächsten beiden Tabellenzeilen.

GOTO

Wir müssen ja für die Tabelle verschiedene Längen in sehr regelmässigen Abständen eingeben. Das Ganze läuft also ungefähr so: GOTO 1, 10 eingeben, 10 eingeben, Ablesen. GOTO 1, 30 eingeben, 10 eingeben, Ablesen. GOTO 1, 50 eingeben, 10 eingeben, Ablesen... Ginge das nicht praktischer? Könnte der Rechner nicht selbst das Einkommen immer um 10 oder 20 erhöhen? Rechnen kann er ja...

Die Erhöhung ist kein Problem. Aber wie bringen wir ihn dazu, quasi sich selbst immer wieder neu zu starten? Mit dem GOTO-Befehl. Mit GOTO sagen wir dem Rechner, er soll von dieser Zeile aus in eine ganz andere Zeile springen. Z.B. ganz an den Anfang. Der GOTO-Befehl kann mit ins Programm.

Bevor wir ihn mit reinnehmen, sollten wir jedoch noch etwas anderes verändern: Die Zeilennummern. Es ist einfach unpraktisch, dass wir jedes Mal das ganze Programm umnummerieren müssen, wenn wir nur eine Zeile einfügen. Es ist sehr viel praktischer, zwischen den Zeilen Platz zu lassen, also immer im 10-er-Abstand zu nummerieren:

```
10 INPUT A
20 INPUT B
30 PRINT SQR(A^2+B^2)
40 PRINT SQR((A+10)^2+B^2)
```

Dann kann man auch mal eine Zeile dazwischen einschieben, ohne gleich alle neu nummerieren zu müssen.

Zunächst wollen wir auf Zeile 40 verzichten. Wir müssen keinen zweiten Wert mehr als eigene PRINT-Anweisung haben, wenn wir sowieso vorhaben, Zeile 30 mehrmals zu benutzen.

Nach Zeile 30 soll A um 10 erhöht werden. Schreiben Sie eine entsprechende Anweisung!

Nun, wie sieht sie aus? Vielleicht so:

```
LET C=A+10
LET A=C
```

Wir speichern in einem Zwischenspeicher, C, den um 10 erhöhten Wert. Dann speichern wir wieder in A den Wert von C. Das ist logisch. Aber es geht noch einfacher:

```
LET A=A+10
```

Das klingt sinnlos. Aber wir dürfen nicht vergessen: Das "=" ist kein Gleichheitszeichen im mathematischen Sinne. Es ist eine Zuweisung. Es sagt: "Das ist der Wert, den du in A speichern sollst." Die ganze Zeile sagt also: "Nimm das, was rechts vom Gleichheitszeichen ist, werte es aus und speichere es im Speicher A." Das ist machbar. Und funktioniert. Damit lautet also unser Gesamtprogramm nun:

```
10 INPUT A
20 INPUT B
30 PRINT SQR(A^2+B^2)
40 LET A=A+10
```

Wenn wir nach Ausführung des Programms ein "? A" eingeben, sehen wir, dass es funktioniert hat.

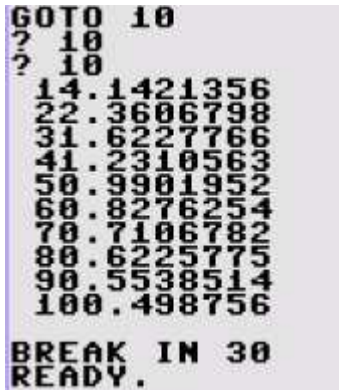
Nun kommt aber das Entscheidende: Die Einführung des GOTO-Befehls. Er sorgt dafür, dass der Rechner bei der automatischen Ausführung an eine ganz bestimmte Stelle des Programms springt. Wo soll er hinspringen? Nun, nachdem die neuen Werte von A und B feststehen, soll er wieder

rechnen. Also in Zeile 30, wo gerechnet wird:

```
10 INPUT A
20 INPUT B
30 PRINT SQR(A^2+B^2)
40 LET A=A+10
50 GOTO 30
```

Vorsicht! Dieses Programm läuft unendlich! Mit der ESC-Taste (Taste ganz links oben in der Ecke) können Sie es anhalten.

OK, nun kann's losgehen:



```
GOTO 10
? 10
? 10
14.1421356
22.3606798
31.6227766
41.2310563
50.9901952
60.8276254
70.7106782
80.6225775
90.5538514
100.498756
BREAK IN 30
READY.
```

Wuff! Auf einen Schlag haben wir zehn, zwanzig, dreissig Ergebnisse! Jetzt sehen wir, was die Macht des Programmierens ausmacht: Der Computer berechnet in Sekunden das, wozu wir mit Handtippen Minuten und Stunden gebraucht hätten.

Eine Anmerkung zu Variablennamen

Einen Speicher wie A und B nennen wir im Programmieren "Variable". Wir können so einer Variablen auch einen anderen Namen geben, z.B. KUH oder ZINS oder GUTHAB. LET GUTHAB=200 geht also genauso wie LET A=10. Allerdings muss man beim C16 höllisch aufpassen: Nur die ersten beiden Buchstaben sind für den verwendeten Speicher charakteristisch. Alle anderen "sieht" der C16 nicht. Folge: GUTHAB1 und GUTHAB2 bezeichnen genau denselben Speicher! Da sie beide mit "GU" anfangen! Daran muss man immer denken und prüfen, ob es vielleicht schon einen anderen Speicher mit diesen beiden Anfangsbuchstaben gibt.

Übung

Natürlich reisst es einen nicht vom Hocker, Seitenlängen von Dreiecken zu berechnen. Schreiben Sie nun einmal ein ganz eigenes Programm:

- Sie haben 200 EUR und wollen das Geld anlegen. Ihre Bank macht Ihnen ein Angebot: Sie zahlen das Geld ein und legen in jedem Monat 10 EUR dazu. Sie bekommen 3% Zins pro Jahr. Wenn der Betrag 1000 EUR übersteigt, bekommen Sie 4%. Wieviel haben Sie nach 10 Jahren? Sie müssen diese Frage so lösen, dass Sie das Programm zunächst mit 3% Zins ausführen. Wenn der Betrag 1000 EUR übersteigt, stoppen Sie, ändern den Zins auf 4%, tragen das bisher erzielte Guthaben ein und lassen weiterrechnen.
- Jetzt wird's schwieriger: Schreiben Sie Ihr erstes Spiel! Und zwar den Klassiker "Mondlandung". Der geht so: Sie haben TS Einheiten Treibstoff. Ihre Raumfähre befindet sich mit der Geschwindigkeit V0 im Anflug auf den Mond. Der Mond zieht die Fähre auch

noch an. Und zwar mit der Beschleunigung $GM=G_0^2/((R_0+S)^2)$. S ist die momentane Höhe der Fähre über der Oberfläche. Dieser Abstand wird in jeder Zeiteinheit DT vermindert um $S_1=V_0*DT$ und (lernt man in der Schule in Physik) um $S_2=1/2*GM*DT^2$. GM erhöht gleichzeitig die Geschwindigkeit um $GM*DT$. Anschliessend ist also $S=S-S_1-S_2$ und $V=V+GM*DT$. Mit dem Treibstoff können Sie die Fähre abbremesen. Und zwar vermindern Sie die Geschwindigkeit um DV pro Treibstoffeinheit. So. Und jetzt müssen Sie die Fähre so abbremesen, dass sie den Boden erreicht, aber dort mit einer Geschwindigkeit V

Mein Programm zur "Mondlandung":

Mein Mondlandeprogramm

```
10 LET TS=40
20 LET V0=30
30 LET G0=500
40 LET S0=200
50 LET R=100
60 LET UMAX=2
70 LET DT=1
80 LET T=0
100 INPUT TR
105 LET TS=TS-TR
110 LET GM=G0↑2/((R+S0)↑2)
120 LET V0=V0-TR*3+GM
130 LET S0=S0-.5*GM-V0
140 PRINT S0
150 PRINT V0
155 PRINT TS
160 GOTO 100
```

Das sind immerhin schon mehr als 10 Zeilen. Aber das Prinzip bleibt das Gleiche: Abgespeicherte Rechnungen. Und Eingabebefehle. Das war's.

[Fenster schliessen](#)

If-Anweisung

Bedingte Anweisungen

Das mit der Unterbrechung mit Hilfe der TAB-Taste geht schon, ist aber unbequem. Spätestens beim Mondlandungsspiel kommt die drängende Frage auf: "Wie kann ich es einrichten, dass der Computer anhält, wenn die Oberfläche erreicht ist?". Man sollte dem C16 sagen können: "Wenn der Wert XY negativ ist, dann höre auf". Oder allgemeiner: "Wenn die und die Bedingung eintritt, dann tue das und das." Auch das geht. Man kann einer Anweisung eine *Bedingung* voranstellen und sagen: "Führe die Anweisung nur dann aus, wenn die Bedingung erfüllt ist". Das ist die IF-Anweisung. Eigentlich ist IF keine Anweisung sondern der Zusatz zu einer anderen, beliebigen Anweisung. Wenn das Programm nur dann A ausgeben soll, wenn A grösser als 2 ist, dann schreibt man

```
IF A>2 THEN PRINT A
```

. (Das ">"-Zeichen finden Sie unter ":" auf dem PC, das "<"-Zeichen unter ";" .) Hinter IF kommt also die Bedingung und hinter THEN die Anweisung.

Als Beispiel unser Pythagoras-Programm:

```
10 INPUT A
20 INPUT B
30 PRINT SQR(A^2+B^2)
40 LET A=A+10
50 IF A<60 THEN GOTO 30
```

Hier werden also für A alle Werte von 10 bis 50 durchlaufen. Wenn A 60 erreicht hat, ist die Bedingung in Zeile 50 nicht mehr erfüllt und das Programm endet.

Übungsaufgabe:

Erweitern Sie das Pythagoras-Programm so, dass für das Wertepaar A,B alle Werte zwischen 10 und 50 (in 10er-Schritten) durchlaufen werden, wie in [der anfangs erstellten Tabelle](#) gezeigt.

Vergleichsoperatoren

Wir haben eben eine neue Verknüpfung (einen neuen Operator) verwendet: Nicht +, -, * oder /, sondern "<". Das gibt's auch auf keinem Taschenrechner. Weil das nur im Zusammenhang mit IF einen Sinn macht und weil es IF nur beim Programmieren gibt. $A < B$ und $A > B$ sind mögliche Bedingungen. Es gibt noch zwei andere:

Zeichen	Beispiel	Bedeutung
=	$A=B$	Ist gleich
<>	$A <> B$	Ist ungleich

Manchmal soll bei IF nicht nur eine Bedingung überprüft werden, sondern derer zwei oder mehr.

Beispiel: Es soll gesprungen werden, wenn $A < B$ und $A < C$. Dazu gibt's den Operator AND:

```
IF A < B AND A < C THEN GOTO 220
```

Manchmal soll auch dann gesprungen, wenn nur eine von zwei Bedingungen zutrifft. Dann verknüpft man die beiden Bedingungen mit einem OR-Operator:

```
IF A < B OR A < C THEN GOTO 220
```

Der dritte Operator schliesslich bezieht sich nur auf eine Bedingung. Er kehrt die Bedingung um. Die IF-Anweisung wird genau dann ausgeführt, wenn die Bedingungen *nicht* zutrifft. Der Operator heisst NOT

```
IF NOT (A < B) THEN GOTO 220
```

Sie sehen, dass hier die Bedingung $A < B$ in Klammern gesetzt wurde. An sich wäre das nicht notwendig gewesen. Aber Klammern sind nützlich, um sicherzustellen, was das Programm zuerst und was zuletzt prüfen soll. In diesem Fall prüft es zuerst $(A < B)$ und dann dreht es das Ergebnis davon herum: NOT. Wenn wir drei Bedingungen bilden, wird das Verwenden von Klammern unverzichtbar.

Bei $IF A < B OR A < C AND A < D THEN \dots$ weiss das Programm nicht, ob zuerst die vorderen zwei Vergleiche geprüft und das OR-Ergebnis dann mit dem dritten Vergleich verknüpft werden soll oder andersherum: Zuerst die letzten beiden Bedingungen auswerten und nur dann, wenn diese beiden zutreffen, das Ergebnis mit der ersten Bedingung OR-verknüpfen. Schauen wir uns dazu ein Beispiel an:

```
310 INPUT A
320 INPUT B
330 INPUT C
340 IF (A < B OR A < C) AND (A < 2) THEN PRINT A+B+C
350 IF (A < B) OR (A < C AND A < 2) THEN PRINT A-B-C
360 IF A < > 99 THEN GOTO 310
```

Dieses Programm nimmt 3 Zahlen entgegen. Wenn die Zahlen der Bedingung von Zeile 340 entsprechen, dann wird die Summe ausgegeben. Wenn die Zahlen der Bedingung von Zeile 350 entsprechen, dann soll $A-B-C$ ausgegeben werden. Im Normalfall werden dann die nächsten drei Zahlen entgegengenommen, es sei denn, in A ist die 99. Das erlaubt dem Benutzer, durch die Eingabe von 99 das Programm zu beenden.

Die beiden Bedingungen in 340 und 350 unterscheiden sich nur in der Klammersetzung.

Nun ein kleines Spiel: Finden Sie durch Probieren mal eine Zahlenkombination, für die nur EINE Zahl ausgegeben wird! Sie werden bald eine finden. Welche ist es, $A+B+C$ oder $A-B-C$? Versuchen Sie, herauszufinden, warum die eine Bedingung zutrifft und die andere nicht!

Lösung: Bei $A=4$, $B=7$ und $C=2$ wird nur $A-B-C$ ausgegeben.

Übung

Erweitern Sie das Mondlandungsspiel so, dass es die folgenden Bedingungen überprüft:

- Ist $S0 > 1000$? Dann ist die Fähre in den Weltraum abgehauen ==> Spiel zuende und verloren.
- Ist $S0 < 0$? Dann ist die Fähre auf dem Boden. Prüfe: Ist $V0 \leq VMAX$ ==> Gewonnen, ansonsten zerschellt.

- TR muss auf TS begrenzt werden. Wie kann man das mittels IF formulieren?

Mehr Komfort: CLEAR, RUN, END, STOP, LOAD, SAVE

In den vorangegangenen Kapiteln haben wir die Pfeiler des Programmierens kennengelernt. Mit PRINT, INPUT, LET, GOTO und IF können Sie schon fast alles programmieren. Aber es ist etwas mühsam. Der C16 bietet Ihnen nicht nur diese elementaren Befehle, sondern noch einiges drumherum. Dies soll in diesem und den folgenden Kapiteln besprochen werden.

CLEAR und RUN

Bis jetzt haben wir unsere Programme immer mit GOTO gestartet. Haben wir allerdings - wie im Moment - mehrere Programme im Speicher, kann es zu Problemen kommen, wenn das Programm Speicher benutzt, ohne sie vorher mit festdefinierten Werten zu belegen. Wir haben inzwischen sicher die Übersicht über all die Speicher und ihre Belegungen verloren. Wenn wir nun ein Programm starten, sind die Speicher garantiert nicht richtig gesetzt und das Programm macht nur Mist.

Daher gibt es ohnehin eine Grundregel beim Programmieren:

Weise jeder Variablen (jedem Speicher) *innerhalb* des Programms einen Anfangswert zu - entweder mit LET oder mit INPUT.

Man soll es also gerade *nicht* so machen, wie wir in den vorangegangenen Kapiteln (aus der Not heraus): Mit Variablen arbeiten, die im Programm selbst nicht gesetzt wurden.

Um hier Fehler leichter erkennen zu können, ist es möglich, vor dem Programmstart alle Variablen auf einmal zu *löschen*. Das macht man mit dem Kommando CLEAR. Man sollte also vor jedem Start des Programms ein CLEAR eingeben. Um sich das zu ersparen, gibt es das RUN-Kommando. Es ist eine Kombination aus CLEAR und GOTO: Zuerst alles löschen, dann loslegen. Es ist üblich, Programme mit RUN zu starten.

END und STOP

Wahrscheinlich ist es ihnen schon längst aufgefallen: Wenn Sie zwei Programme hintereinander im Speicher stehen haben, dann kann es leicht passieren, dass beim Start des ersten Programms das zweite mit ausgeführt wird. Wenn Sie anfangs das erste Programm in den Zeilen 1 bis 4 und das zweite ab Zeile 10 stehen hatten, dann wurde bei GOTO 1 natürlich alles ausgeführt, Zeile 1 bis 4 und dann weiter ab Zeile 10. Kann man das auch trennen? Kann man. Mittels dem END-Befehl. Kommt das Programm an ein END, dann beendet es das Programm.

```
10 INPUT A
20 IF A=1 THEN GOTO 110
30 IF A=2 THEN GOTO 150
110 INPUT A
120 INPUT B
130 PRINT A+B
```

```

140 END
150 INPUT A
160 INPUT B
170 PRINT A-B
180 END

```

Dieses Programm nimmt zuerst einmal eine Zahl entgegen. Mit dieser Zahl wird aber nicht gerechnet, sie wird als ein Schalter verarbeitet. Wurde 1 eingegeben, verzweigt das Programm nach 110 und summiert zwei Zahlen auf. Wurde 2 eingegeben, verzweigt das Programm nach 150 und bildet die Differenz aus zwei Zahlen. Der END-Befehl sorgt dafür, dass das Programm nicht vom Summe-Teil in den Differenzteil läuft.

Im Grunde der gleiche Befehl ist der STOP-Befehl. Er verursacht nur eine andere Antwort des Rechners. Nach einem END-Befehl antwortet er mit "READY", nach einem STOP-Befehl mit "BREAK IN LINE XXX" (XXX steht hier für eine Zeilennummer). Daher wird STOP zum Testen des Programms benutzt. Im entgeltig laufenden Programm wäre der STOP-Befehl nicht so gut, weil "BREAK" ja heisst: Das Programm wurde unterbrochen, eigentlich müsste es weiterlaufen. Da ist was schiefgelaufen. Und das kann dann den Nutzer (auch Sie selbst) irritieren.

SAVE und LOAD

Wahrscheinlich haben Sie sich schon ziemlich geärgert. Es ist ja ganz schön, das Programmieren, aber wenn man nach jedem Beenden des C16 (also von Yape), bzw. nach dem Herunterfahren des PC's alle Programme wieder eintippen muss - das kann doch nicht wahr sein. Kann sich der C16 die eingegebenen Programme nicht behalten?

Um sich noch ein bisschen mehr zu ärgern: Es war durchaus beabsichtigt, dass Sie das/die Programme an jedem Morgen neu eintippen. Das schafft Bewußtsein, lässt das Gefühl für "Speicher" vom Verstand ins Herz sickern. Der Speicher des C16 (und des PC's, der vom C16 angesteuert wird), ist elektronischer Speicher. Alles wird irgendwie in elektrischen Schaltkreisen gespeichert. Ist der Strom weg, ist auch der Speicherinhalt weg. Bei Taschenrechnern ist das auch so - nur ist da der Strom nie weg, da die Batterie den Speicher auch in ausgeschaltetem Zustand mit Strom versorgt.

Diesen elektronischen Speicher nennt man **Hauptspeicher**. Dort werden sowohl die Variablen als auch das Programm gespeichert. Wir werden an anderer Stelle noch näher darauf eingehen. Jetzt interessiert uns erstmal, welche Wege es gibt, den Hauptspeicherinhalt, besonders das Programm permanent zu sichern.

Erster Weg: Abschreiben

Sie lachen? Gut, zugegebenermassen ist das heute - wir werden es gleich lernen - keine gute Alternative mehr. In den 70er- und frühen 80er-Jahren stand genau dies am Anfang jeder Programmierer-Karriere: Das Programm vom Bildschirm auf Papier zu übertragen. Die anderen, weiter unten beschriebenen Möglichkeiten, waren zu teuer; das Geld hatte gerade für den nackten Homecomputer gereicht und ein Massenspeicher kostete mindestens 300 DM, ein guter (Diskettenlaufwerk) um die 1000 DM.

Zweiter Weg: Ausdrucken

Wenn man einen Drucker zur Verfügung hat, geht der Weg auf's Papier via Ausdrucken natürlich schneller. Für die umgekehrte Richtung, vom Papier in den Hauptspeicher zurück gibt es immer noch nur die Tastatur und den tippenden Menschen.

Dritter Weg: Kassettenrecorder

Der für den C16 damals übliche Weg war ein Kassettenrecorder. Den konnte man an den C16

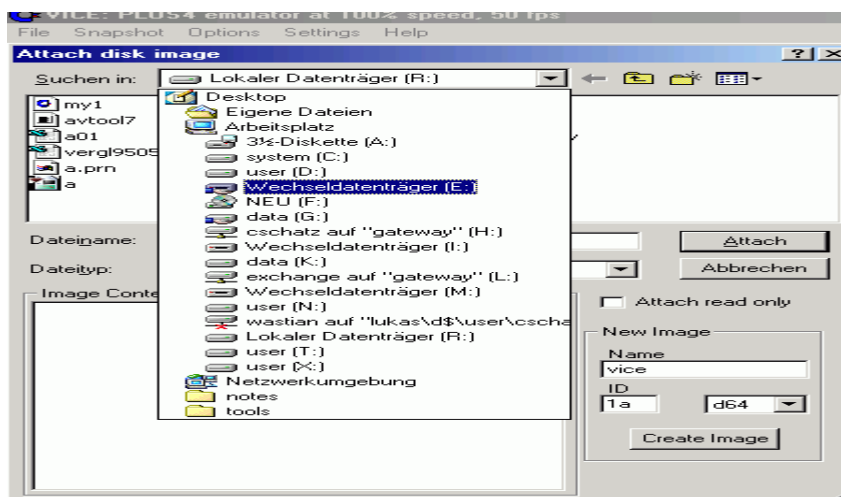
anschliessen. Dann nahm man eine handelsübliche Kassette (am Besten eine mit kurzer Länge, z.B. C30) und steckte sie in den Recorder. Nun gab man im C16 das Kommando SAVE ein (plus ENTER). Anschliessend kommt die Meldung "PRESS PLAY & RECORD ON TAPE". Dann musste man den Kassettenrecorder auf "Aufnahme" stellen. War das getan, fing ein grässliches Piepsen an, ziemlich ähnlich wie ein Faxgerät oder analoges Modem. Der Bildschirm verschwand und es dauerte - je nach Länge des Programms - 10 Sekunden bis 10 Minuten, bis er wieder kam. Anschliessend konnte man die Kassette anhalten und das Programm war auf Kassette (in Form des Piepsens) gespeichert. Nun konnte man den C16 abschalten. Am nächsten Morgen legte man die Kassette ein und spulte sie erstmal an den Anfang zurück. Dann gab man beim C16 den Befehl LOAD ein. Anschliessend verschwand der Screen (Bildschirm). Nun drückte man auf PLAY. Und nach 10 Sekunden bis 10 Minuten tauchte der Bildschirm wieder auf. In der Hälfte der Fälle mit einer ERROR-Meldung. Dann musste man sein Glück nochmal versuchen. In der anderen Hälfte mit einer OK-Meldung. Dann gab man den LIST-Befehl ein, sah das Programm und war sehr glücklich.

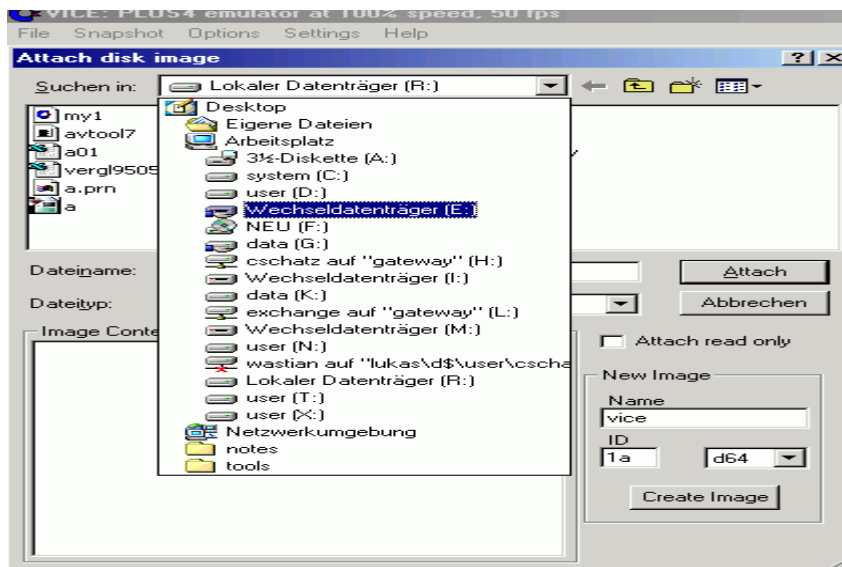
Königsweg: Diskette

Hier und da gab es auch hinreichend Reiche oder Fanatische, die 1000 DM für ein Diskettenlaufwerk ausgaben ;-). Die Technik ist im Grunde die gleiche, wie beim Kassettenrecorder: Der Speicherinhalt wird in der Magnetschicht einer beschichteten Folie festgehalten. Diese Art von Speicherung braucht keinen Strom. Im Gegensatz zum Tonband in der Kassette kann der Lese- und Schreibkopf des Diskettenlaufwerks wie der Tonkopf eines Schallplattenspielers die ganze Platte, d.h. die ganze scheibenförmige Folie erreichen. Der Unterschied liegt nicht nur in einer viel höheren Übertragungsgeschwindigkeit, er liegt vor allem in der Möglichkeit, die einzelnen Programme auf der Platte *gezielt* zu laden. Beim Tonband liegt unter dem Kopf immer nur ein Programm und man muss dieses komplett am Kopf vorbeispielen oder -spulen, um ans nächste zu gelangen. Bei der Diskette kann man den Kopf gezielt zum Programm bringen - innerhalb von Millisekunden.

Wir haben nun den Komfort, den Königsweg gehen zu können. Ihr PC hat (wahrscheinlich noch) ein Diskettenlaufwerk. Das wäre das praktischste. Ein USB-Stick oder eine Speicherkarte wären auch gut. Das ist natürlich nicht das gleiche wie damals das beim C16. Vor allem Speicherkarten fassen viel mehr Daten und sind schneller. Aber das schadet ja nichts. Zuerst muss das entsprechende Laufwerk (bei der Diskette wäre es A:) für den C16 zugänglich gemacht werden. Dazu schieben Sie erstmal eine Diskette/Speicherkarte ins Laufwerk, bzw. stöpseln einen Stick an. Dann gehen Sie mit der Maus in das Menü "File" des Emulators und dort zu "Attach Disk Image" und dann nach rechts zu "Drive 8".

Nun müssen Sie sich eine virtuelle Diskette erzeugen. Dazu klappen Sie ganz oben das Ordner-Menü auf und gehen auf "31/2-Diskette A:" oder auf "Wechseldatenträger".





Dann geben Sie der virtuellen Diskette einen Dateinamen, z.B. "mydisk". Nun erzeugen Sie die Diskette durch Klick auf "Create Image" ganz unten. Und dann klicken Sie weiter oben auf "Attach". Damit haben Sie Ihre virtuelle Diskette erzeugt und angeschlossen.

Ab jetzt können Sie Ihre Programme auf der Diskette speichern. Dazu benutzen Sie das Kommando SAVE wie bei Kassetten. Der Unterschied besteht darin, dass Sie noch zwei Dinge zusätzlich angeben müssen: Einen Programmnamen und die Zahl 8. Eine Speicherung geht also durch

```
SAVE "MEINPROG", 8
```

Das war's schon. Die Zahl 8 ist etwas mysteriös. Es handelt sich hier um eine Gerätenummer: Der Kassettenrecorder hat die 1 (was weggelassen werden kann) und das Diskettenlaufwerk die Nummer 8. Warum die 8 und nicht die 2, das weiss ich nicht.

Laden tut man mit dem LOAD-Kommando analog:

```
LOAD "MEINPROG", 8
```

Allerdings funktioniert das nur, falls - wie jetzt - nach dem Starten des Emulators auch eine virtuelle Diskette "attached" wurde. Wollen Sie an Ihre virtuelle Diskette nach einem Neustart des Emulators wieder herankommen, gehen Sie wie oben wieder in das Fenster "Attach disk image", wählen im Dateifenster die Datei "mydisk1", die die virtuelle Diskette verkörpert und gehen auf "Attach". Anschliessend ist die virtuelle Diskette wieder angeschlossen und das LOAD funktioniert.

Und fertig ist die Laube. Wir können nun nicht nur ein Programm abspeichern, sondern derer mehrere - soviel, wie eben auf der Diskette Platz haben (mehr zum Platz im Kapitel "Bits und Bytes"). Wir müssen ihnen nur unterschiedliche Namen geben.

Wenn man die Übersicht verloren hat, was da eigentlich so alles auf der Diskette drauf ist, kann man das Kommando DIRECTORY eingeben. Daraufhin listet der C16 den Inhalt der Diskette auf - alle Programme, die schon gespeichert wurden.

Das sieht dann vielleicht so aus:

```
1      "MEINPROG"      PRG
1      "PROG2"         PRG
1      "RECHNEN"       PRG
```

Die erste Spalte zeigt die Grösse des Programms in s.g. Blocks. Bis jetzt haben unsere Programme alle offensichtlich nur einen Block Grösse. (Die Einheit "Block" ist veraltet - das brauchen Sie sich nicht zu merken.) Dann kommt der Name. Und dann noch "PRG", was "Programm" bedeuten soll. Das ist der Dateityp. Es kann auch noch andere Dateitypen geben; damit werden wir uns noch beschäftigen.

Strings

Texte ausgeben

In diesem Kapitel lernen wir, dass der Computer noch mit anderem umgehen kann als mit Zahlen. Schauen Sie sich mal folgendes Programm an:

```
10 LET HALLO=1
20 PRINT HALLO
30 PRINT "HALLO"
```

Lassen Sie es laufen. Die Ausgabe ist:

```
RUN
1
HALLO

READY
```

Verstehen Sie, was da passiert? Bin Zeile 20 gibt das Programm den Inhalt der Variablen HALLO aus. Aber in Zeile 30 gibt das Programm gar keine Zahl aus. Es gibt einfach das *Wort* HALLO aus! Was ist denn da passiert?

Nun, der PRINT-Befehl kann nicht nur Gerechnetes und Speicherinhalte ausdrucken, er kann auch Zeichenketten ausdrucken. Texte zum Beispiel. Der Unterschied besteht in den Gänsefüßchen. Wird eine Zeichenkette im Programm von Gänsefüßchen eingeschlossen, wird sie als Zeichenkette und nicht als Variablenname interpretiert.

Und wozu soll das gut sein? Zunächst kann es ganz nützlich sein, die Zahlen, die man auf dem Bildschirm ausgibt, zu kommentieren. Hier eine verbesserte Variante des Mondlande-Programms:

```
1 PRINT "-----"
2 PRINT "MONDLANDUNG"
3 PRINT "-----"
10 LET TS=40
20 LET V0=30
30 LET G0=500
40 LET S0=200
50 LET R=100
60 LET VMAX=2
90 PRINT "HOEHE:"
91 PRINT S0
100 PRINT "GESCHWINDIGKEIT:"
101 PRINT V0
110 PRINT "TREIBSTOFF:"
111 PRINT TS
120 PRINT "WIEVIEL TREIBSTOFF ZUGEBEN?"
125 INPUT TR
130 IF TR>TS THEN TR=TS
140 LET TS=TS-TR
150 LET GM=G0^2/((R+S0)^2)
160 LET V0=V0-TR*3+GM
```

```

170 LET S0=S0-.5*GM-V0
180 IF S0>0 THEN GOTO 90
190 IF V>VMAX THEN GOTO 220
200 PRINT "SIE HABEN GEWONNEN!"
210 GOTO 230
220 PRINT "DIE FAEHRE IST ZERSCHELLT! OH JEH!"
230 PRINT "NOCHMAL? (1)"
240 INPUT A
250 IF A=1 THEN GOTO 10
260 END

```

Erweitertes PRINT und INPUT

Der PRINT-Befehl hat noch mehr Möglichkeiten als die, die wir bisher kennengelernt haben. Zunächst haben Sie ja sicher bemerkt, dass jedes neue PRINT in eine neue Zeile druckt. Das macht die Ausgabe von Tabellen unmöglich, wo man ja mehrere Werte in eine Zeile packen muss. Nun, da gibt's gleich zwei Lösungen. Sie können hinter den PRINT-Befehl *mehrere* Ausdrücke schreiben (also Rechnungen oder Strings), getrennt durch ein **Semikolon** oder durch ein **Komma**:

```

10 PRINT "BEISPIEL FUER KOMMA-"
20 PRINT "TRENNUNG"
30 PRINT "A","B","C"
40 PRINT "BEISPIEL FUER SEMIKOLON-"
50 PRINT "TRENNUNG"
60 PRINT "A";"B";"C"

```

```

RUN
A      B      C
ABC

```

```
READY
```

Das Komma sorgt für einen Tabulatoreffekt: Der nächste Ausdruck wird 8 Spalten weiter ausgegeben. Beim Semikolon wird der nächste Ausdruck unmittelbar hinter den vorhergehenden gehängt.

Eine PRINT-Zeile kann auch mit einem Komma oder einem Semikolon abgeschlossen werden:

```

10 PRINT "EINKOMMEN1";
20 INPUT A

```

```

RUN
EINKOMMEN1?30000

```

```
READY
```

Der Effekt ist, dass der Cursor für das nächste PRINT oder INPUT direkt an die Ausgabe anschliesst. Auf diese Weise können Sie Ihren PC zu einer bmbastischen Jubelhymne auf Sie anstimmen lassen:

```

10 PRINT "NAME IST DER GROESSTE! _ _ _";
20 GOTO 10

```

Die "_"-Zeichen stehen für ein Leerzeichen. Und statt NAME schreiben Sie natürlich Ihren Namen rein. Und das Semikolon am Ende von Zeile 10 dürfen Sie auf keinen Fall vergessen!

Auch der INPUT-Befehl kennt noch ein kleines Schmankerl. Statt INPUT A kann man auch INPUT "Text";A schreiben. Das ist eine Zusammenfassung von PRINT "Text"; und dann INPUT A: Es wird erst ein Text ausgegeben und dahinter kommt das ?, wo die Zahl eingegeben wird. Diese Form ist sehr nützlich, da man fast immer dem Nutzer vorher sagen möchte, welche Art Zahl er jetzt eingeben soll.

String-Variablen

Strings können auch in Variablen abgespeichert werden, wie Zahlen. Dazu gibt es einen besonderen *Variablentyp*. Bis jetzt kannten wir nur Zahlvariablen, jetzt lernen wir auch Stringvariablen kennen. LET A\$="ABCD" speichert den String "ABCD" in der Variablen A\$ ab. Wichtig ist also das \$-Zeichen hinter dem Variablennamen, das dem Programm sagt: Hier soll nicht eine Zahl, sondern ein String abgespeichert werden.

Stringvariablen können natürlich nicht mit Rechenoperationen verknüpft werden wie Zahlvariablen. Plus, Minus, Mal oder Geteilt hätten hier keinen Sinn. Trotzdem hat das + eine Bedeutung. Dazu ein Beispiel:

```
10 LET A$="123"
20 LET B$="456"
30 LET C$=A$+B$
40 PRINT C$
```

```
RUN
123456
```

```
READY
```

+ verknüpft die beiden Strings also im wörtlichen Sinne: Er hängt sie aneinander.

Übungsaufgabe: Schreiben Sie ein Programm, das folgenden Bildschirmausdruck erzeugt:

```
*****
      *
     *
    *
   *
  *
 *
*
*
*
*
*
*
*
*****
```

Sie sehen, dass Sie auf diese Weise mit dem C16 sogar sehr einfache Dinge malen können. Die erste Grafik!

String-Funktionen

LEN()

Auch für Strings gibt es eine ganze Reihe von Funktionen. Da wäre zuerst LEN(). Mit LEN

bestimmt man die Länge eines Strings (LEN=LENgth).

```
10 LET A$="ABCD"
20 PRINT LEN(A$)
```

```
RUN
4
```

```
READY
```

Die 4 sagt: Der String A\$ ist vier Zeichen lang.

LEFT\$(), RIGHT\$(), MID\$()

Dann wäre da LEFT\$() und RIGHT\$(). LEFT\$(A\$,3) gibt die linken 3 Zeichen von A\$ aus, im Beispiel A\$="ABCD" wäre LEFT\$(A\$,3)="ABC". Bei RIGHT\$(A\$,3)="BCD" sind's die rechten 3 Zeichen. Und MID\$() greift sich schliesslich aus der Mitte einen Unterstring heraus: MID\$(A\$,Position,Länge) ist die genaue Syntax und das sieht dann am Beispiel so aus: MID\$(A\$,2,2)="BC", MID\$(A\$,2,1)="B", MID\$(A\$,3,2)="CD".

ASC() und CHR\$()

Intern sind Strings und Zahlvariablen gar nicht so weit auseinander, wie Sie vielleicht meinen. Es zeigt sich doch, dass ich am Anfang recht gehabt hatte: Im Grunde ist ein Computer eine Rechenmaschine und kann nur mit Zahlen umgehen. Tatsächlich speichert der Computer Strings als eine Kette von Zahlen. Jede Zahl ist dabei eine Nummer. Und da jedes Zeichen des Computers eine solche Nummer trägt, kann dann jeder Nummer ein Zeichen zugeordnet werden. Doch welche Nummer gehört zu den verschiedenen Zeichen? Mit Hilfe der ASC()-Funktion ist das leicht herauszufinden:

```
10 INPUT "ZEICHEN";A$
20 LET A$=MID$(A$,1,1)
30 PRINT ASC(A$)
40 GOTO 10
```

Dieses kleine Programm printet Ihnen den Zeichencode für jedes eingegebene Zeichen. Sicherheitshalber sorgt Zeile 20 dafür, dass wirklich nur ein Zeichen in A\$ steht, d.h. LEN(A\$)=1 ist, sonst macht Zeile 30 nicht viel Sinn.

Die Umkehrfunktion zu ASC() ist CHR\$(). Damit geht's noch übersichtlicher. Argument von CHR\$ ist der Zeichencode und CHR\$ ergibt das entsprechende Zeichen.

```
10 PRINT "NR", "ZEICHEN"
20 I=0
30 PRINT I,CHR$(I)
40 LET I=I+1
50 IF (I<256) THEN GOTO 30
```

ASCII

Dieses Progrämmchen druckt Ihnen eine ganze Tabelle mit allen Zeichencodes auf den Bildschirm. Warum nur von 0 bis 255? Weil es nicht mehr Zeichencodes gibt. Nicht beim C16 und (standardmässig) nicht mehr beim PC. Warum, das wird Ihnen im Kapitel "Bits und Bytes" erklärt. Hier sei nur erwähnt: Die Zeichenkodierung könnte theoretisch bei jedem Computertyp anders sein. Aber das wäre sehr unpraktisch. Vor allem wäre so kaum Kommunikation möglich. Daher hat man

sich schon sehr früh in der Computergeschichte darauf geeinigt, einen bestimmten Vorrat an Zeichen einheitlich zu kodieren, nämlich den Bereich 32 bis 127. Das ist die s.g.

ASCII-Kodierung. Die Tabelle, die das Progrämmchen ausdrückt, zeigt also in diesem Bereich die ASCII-Tabelle. Oberhalb von 127 sind meist die ganzen länderspezifischen Sonderzeichen, die bis heute keine vereinheitlichte Kodierung besitzen. Weswegen man bis vor kurzem tunlichst keine deutschen Umlaute in Emails nutzen sollte, weil sonst beim Empfänger anstelle der Umlaute nur komische Sonderzeichen zu sehen waren. Und weswegen man in Web-Adressen und Emailadressen keine deutschen Umlaute benutzen darf, auch heute noch nicht.

Seit ein paar Jahren beginnt sich eine andere Kodierung langsam zu verbreiten: **UNICODE**. Hier hat man mehr als 65000 verschiedene Codes. Damit kann man alle Zeichen aller Sprachen der Welt, selbst japanisch und chinesisch locker einheitlich vercoden. Webbrowser arbeiten tw. schon mit UNICODE, während Email weiterhin auf ASCII beruht.

Am oberen Progrämmchen sehen wir noch was: Warum haben wir in Zeile 20 die Variable I verwendet? Und nicht A, B, C ...? Nun, in der Mathematik ist es üblich, für Zählvariablen, s.g. Indizes, die Buchstaben i,j,k und l herzunehmen. Und das hat die Informatik übernommen. Daher hat man es sich angewöhnt, Zählervariablen mit I,J,K usw. zu benennen. Das ist für die Lesbarkeit des Programms sehr gut: Man erkennt am Namen sofort, dass es sich um einen Zähler handelt.

VAL() und STR\$()

Nehmen wir an, Ihr Programm soll vom Benutzer eine Eingabe entgegennehmen. Dabei soll es ihm ermöglicht werden, entweder eine Zahl oder den Buchstaben Q (für "Quit") einzugeben. Wie machen wir das? Wir müssen uns doch entscheiden, ob wir den Benutzer eine Zahl oder einen String eingeben lassen. Oder? Nicht unbedingt. Wir können auch das eine zum andern machen. *Casting* nennt sich das in der Informatik. Wenn man Variablen oder Werte unterschiedlichen Typs ineinander umwandelt. Bei A=VAL(A\$) versucht der Computer, A\$ als Zahl zu interpretieren. Wenn A\$="2.3", dann besitzt nachher A den Wert 2.3. Wenn A\$="X3X", dann ergibt VAL(A\$) keinen Sinn. In diesem Fall wird einfach eine Null zurückgegeben. Also Vorsicht beim Ergebnis Null! Dieser Fall muss im Programm immer überprüft werden.

Übungsaufgabe: Schreiben Sie ein Programm, das einen String darauf untersucht, ob es sich um eine Zahl handelt!

STR\$ macht genau das Umgekehrte: Es wandelt eine Zahl in einen String um. Das geht natürlich immer. Wozu braucht man das? Nun, ein Beispiel: Für Tabellen o.ä. kann es nützlich sein, die Zahl rechtsbündig an einer Stelle des Bildschirms auszugeben. Das kann der C16 aber nicht von sich aus. Aber man kann es programmieren.

Übungsaufgabe: Schreiben Sie ein kleines Programm zur Honorarabrechnung. Es nimmt die Zahl der Stunden entgegen, multipliziert sie mit dem (im Programm fest gespeicherten) Stundensatz, gibt Stundenzahl und Nettohonorar aus, dann die MWSt (auf der Basis von 16%) und schliesslich das Brutto-Honorar. Soweit nicht schwer. Aber nun der Kick: Alle Ausgaben bitte rechtsbündig! Da werden Sie die STR\$() bemühen müssen!

Ein Spiel mit Strings: Mastermind

Eine gute Übung ist es, schon bekannte einfache Spiele nachzuprogrammieren. Die "Mondlandung" war unser erstes Beispiel. Hier kommt das nächste. Für die meisten Spiele fehlt uns aber noch eine für Spiele eminent wichtige Funktion:

Diese werden mit der Funktion RND() generiert. Es ist immer eine Zahl größer gleich null und kleiner 1. RND() braucht auch ein Argument (einen Übergabewert, den man zwischen die Klammer schreibt.). Der ist allerdings in diesem Fall sehr technischer Natur und soll hier erstmal nicht weiter erklärt werden. Mit RND(1) fahren Sie immer richtig. Das folgende Programmchen generiert 10 Zufallszahlen zwischen 0 und 1 und dann 10 Ganzzahlen zwischen 1 und 6, simuliert also in seinem zweiten Teil einen Würfel.

```
5 PRINT "REINE ZUFALLSZAHLEN"
10 LET I=0
20 PRINT I,RND(1)
30 LET I=I+1
40 IF I<10 GOTO 20
50 PRINT "WUERFELZAHLEN"
60 LET I=0
70 PRINT I,RND(1)*6+1
80 LET I=I+1
90 IF I<10 GOTO 20
```

Für das Testen von Programmen ist es unverzichtbar, dass die Reihenfolge der Zufallszahlen nachvollziehbar bleibt. Das heisst, wenn man das Programm zum zweiten Mal startet, kommt die gleiche Sequenz von Zahlen. Um das zu erreichen, ist es beim C16 notwendig, die RND-Funktion am Anfang einmal mit einem *negativen* Wert aufzurufen. Wenn Sie also die Zeile

```
7 LET A=RND(-1)
```

einschieben, dann kommt immer die gleiche Zahlenfolge.

Mastermind

Mastermind ist ein Spiel, bei dem ein Spieler 6 farbige Stecker so in einer Reihe anordnet, dass es der andere nicht sieht. Der andere muss nun durch Kombinatorik herausbekommen, welche Steckerkombination gewählt wurde. Anfangs wählt er eine zufällige Kombination. Der erste Spieler bewertet dann diese Kombination. Er sagt, wieviel Stecker die richtige Farbe haben und wieviel Stecker die richtige Position haben. Aber er sagt nicht, *welche* Stecker dies sind. Dies bleibt dann der Kombinationsgabe des aktiven Spielers überlassen.

Den Schwierigkeitsgrad von Mastermind kann man variieren: Je grösser die Anzahl der zu erratenden Stecker und je grösser die Farbauswahl, desto schwieriger.

Dieses Spiel wollen wir nun programmieren. Den passiven Part des ersten Spielers übernimmt der Computer. Es sieht schwerer aus, als es ist.

Zunächst müssen wir uns überlegen, wie das ganze für den Spieler aussehen muss. Die erste Frage ist: Was sind Stecker und was sind Farben? Nun, diese Rolle sollen Zeichen übernehmen. Nehmen wir z.B. die Buchstaben A bis F, dann sind 6 Farben im Spiel. Und die Steckeranordnung ist ein String. Hat die Anordnung 4 Stecker, ist der String eben 4 Zeichen lang. Gesucht wird also z.B. der String "AABF". Der ist aber für den Spieler unsichtbar. Der Benutzer wird nun aufgefordert, selbst einen String einzugeben. Z.B. gibt er dann "FAEF" ein. Und nun muss der C16 beurteilen, wieviel Buchstaben überhaupt richtig sind und wieviel an der richtigen Position sitzen. Im Beispiel müsste er ausgeben: 3 Buchstaben sind richtig (A und zweimal F) und zwei sitzen an der richtigen Position (2. und 4. Buchstabe). Und dann hat der Benutzer den nächsten Versuch.

Die Aufgabe gliedert sich also in

- Zufälligen String bilden.
- Eingabe auffordern und String entgegennehmen.
- Richtige Buchstaben und richtige Positionen zählen.
- Zählergebnisse ausgeben.
- GOTO Nächste Eingabe

Bei der Prüfung müssen Sie eifrig von MID\$() Gebrauch machen. Und bei der Stringbildung müssen Sie zunächst zufällige Zeichencodes in einem bestimmten Bereich (schauen Sie in der ASCII-Tabelle nach!) mit der RND()-Funktion bilden und diese dann mittels CHR\$() in die Zeichen umwandeln. So, und jetzt genug der Tipps. Jetzt sind Sie dran!

Bildschirmpositionierung

...wenn wir schon beim Spielen sind. Die meisten Spiele haben sehr ausgefeilte Bildschirm (Screen)-Ausgaben. Denken wir an solche Spieleklassiker wie "Tetris", "PacMan" oder "Space Invaders": Da fliegen Gegenstände, Monster, Raumschiffe über den Bildschirm. Das ist mit unserem PRINT, selbst wenn wir beliebige Zeichenketten ausgeben können, nicht so einfach zu machen. Ausserdem erfolgt die Steuerung des Bildschirmgeschehens nicht mit zeilenorientierten Eingaben, die alle mit einem ENTER abzuschliessen sind. Sondern man drückt eine Taste und sofort passiert etwas. Kann das der C16 aus? Klar kann er das.

Bildschirm löschen

Um interessante Bildschirm (Screen)-Ausgaben zu gestalten, müssen wir den Bildschirm zuerst komplett löschen können. Das machen wir mit dem Befehl **SCNCLR**, was für "Screen Clear" steht:

```
10 I=0
20 PRINT "HALLO! ";
30 LET I=I+1
40 IF I<30 THEN GOTO 20
50 PRINT "DRUECKEN SIE ENTER, DANN "
60 PRINT "WIRD DER BILDSCHIRM GELOESCHT";
70 INPUT A$
80 SCNCLR
```

Die Organisation des Bildschirms

Der Screen des C16 ist organisiert wie eine Tabelle: Er hat 40 Spalten, in die jeweils ein Buchstabe passt. Und 25 Zeilen:

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	
00																																									
01																																									
02																																									
03																																									
04																																									
05																																									
06	D	A	S					I	S	T						E	I	N																							
07																																									
08																																									
09																																									
10																																									
11																																									
12																																									
13																																									
14																																									
15																																									
16																																									
17																																									
18																																									
19																																									
20																																									
21																																									
22																																									
23																																									
24																																									

Normalerweise, wenn wir nichts weiter angeben, dann setzt der C16 von sich aus die Bildschirmposition des nächsten PRINT's. Aber wir haben schon zwei Methoden kennengelernt, diese Position zu beeinflussen: Ein Semikolon hinter der PRINT-Zeile schliesst die nächste Ausgabe hinter der ersten an, ein Komma versetzt sie in derselben Zeile an die nächste Tabulator-Position.

Darüber hinaus gibt es noch die **TAB**-Funktion, die es erlaubt, die Spalte direkt anzugeben. (Eigentlich handelt es sich dabei auch nicht um

eine Funktion, denn sie gibt nicht, wie eine Sinus-Funktion, einen Wert zurück. Es ist einfach ein Schlüsselwort innerhalb einer PRINT-Zeile.) `PRINT TAB(20);"Hallo!"` gibt das Wort "Hallo!" in Spalte 20 aus.

Das hilft ein bisschen, aber eine Zeilensteuerung erreichen wir dadurch auch noch nicht. Die bekommen wir nur mit Hilfe eines anderen Ausgabebefehls, des Befehls `CHAR`. Hier können wir beides angeben, Zeile und Spalte: **`CHAR ,10,10,"Hallo!"`** setzt den Text "Hallo!" in Zeile 10 und Spalte 10. Das können Sie zuerst auch ohne Programm, einfach auf der Kommandozeile versuchen. Vergessen Sie das führende Komma nicht!

`CHAR` ist allerdings mit `PRINT` nicht ganz identisch, da es keine Rechnungen, sondern *nur* Strings ausgeben kann. Aber das stört uns nicht weiters: Mit Hilfe von `STR$()` können wir auch das lösen. Übrigens können wir `CHAR` auch dazu benutzen, *nur* die Bildschirmposition für das nächste `PRINT` zu setzen:

```
10 SCNCLR
20 CHAR ,0,9,""
30 PRINT TAB(10);1+1:
```

schreibt die `PRINT`-Ausgabe in Zeile 10 und Spalte 10.

Mondlandung - die grafische Version

Mithilfe von `CHAR` können Sie nun einerseits professionelle Bildschirmeingabemasken erstellen, wie Sie es z.B. von Kassensystemen kennen. Und andererseits können Sie `CHAR` für Spiele benutzen, um Objekte zu zeichnen oder sogar zu bewegen. Programmieren wir an unserer Mondlandung weiter. Wir wollen die Fähre nun zeichnen. Dazu nehmen wir die Sonderzeichen der ASCII-Tabelle aus dem vorigen Kapitel zu Hilfe. Wie wär's mit der hier?

```
LIST 130-
130 PRINTCHR$(117);CHR$(101);CHR$(105)
135 PRINT CHR$(106);CHR$(102);CHR$(110)
140 PRINT CHR$(110);" ";CHR$(109)

READY.
RUN 130
  O
 / \
READY.
```

Als nächstes schreiben wir einen Programmteil, der die Fähre an eine bestimmte Position auf den Bildschirm schreibt. Die Spalte stehe im Speicher `X`, die Zeile im Speicher `Y` und dieses Zeichen beschreibe das oberste linke Zeichen der Fähre. Dann lautet dieser Programmteil (die Zeilennummern sind beliebig):

```
100 CHAR ,X,Y,CHR$(117);CHR$(101);CHR$(105)
110 CHAR ,X,Y+1,CHR$(106);CHR$(102);CHR$(110)
120 CHAR ,X,Y+2,CHR$(110);" ";CHR$(109)
```

Um die Fähre zu bewegen, müssen wir die Fähre an der alten Position löschen, dann die Position verändern und dann die Fähre neu zeichnen. Das folgende Programm lässt die Fähre über den ganzen Bildschirm huschen:

```
10 X=RND(1)*37
20 Y=RND(1)*23
70 CHAR ,X,Y," _ _ "
80 CHAR ,X,Y+1," _ _ "
90 CHAR ,X,Y+2," _ _ "
100 CHAR ,X,Y,CHR$(117)+CHR$(101)+CHR$(105)
110 CHAR ,X,Y+1,CHR$(106)+CHR$(102)+CHR$(110)
120 CHAR ,X,Y+2,CHR$(110)+" "+CHR$(109)
130 LET I=0
140 LET I=I+1
150 IF I<10 THEN GOTO 140
160 GOTO 10
```

Die `" _ "` stehen wieder für Leerzeichen. Wie Sie sehen, müssen wir bei `CHAR` aufpassen: Wir können die Sonderzeichen nicht mittels eines ; einanderhängen (";" gibt's nur beim `PRINT`-Befehl), sondern müssen aus den drei Sonderzeichen mittels `"+"` einen String zusammenkleben. Eine weitere Besonderheit ist der Teil 130 bis 150. Man kann ihn auch weglassen, aber dann huscht die Fähre so schnell, dass man sie gar nicht sieht. 130 bis 150 tut eigentlich gar nichts, es zählt sinnlos die Variable `I` hoch, braucht so Zeit und lässt damit die Fähre länger an ihrer Position auf dem Bildschirm stehen.

Wir brauchen natürlich auch noch eine Mondoberfläche. Das kann einfachheitshalber ein gerader Strich sein. Oder - etwas intuitiver - eine gezackte Linie, die die Unebenheiten und Krater darstellt.

Den Rest des Programms haben wir ja schon. Wir müssen nun nur noch die Variable `"S0"` in Bildschirmkoordinaten `Y`, also Werte zwischen 1 und 24 übersetzen. Wobei wir aufpassen müssen: Ein kleines `S0` entspricht einem grossen `Y`. So könnte es gehen:

```
10 LET Y=24-S0/250*24
```

Wenn `S0` auf dem Anfangswert 200 steht, dann ist `S0/250` etwa 0,8 und `Y=24-0,8*24` dürfte ziemlich klein, also bei 2 bis 4 so liegen. D.h.,

die Fähre schwebt ganz oben. Wenn $S_0=0$ ist, dann ist $Y=24-0=24$ und die Fähre ist ganz unten.

Vielleicht war Ihnen das im zweiten Kapitel dieses Tutorials mit dem vielen Rechnen ein bisschen lästig und Sie dachten: "Meine Güte, ich will doch programmieren lernen und keine Mathestunde". Nun ja, jetzt sehen Sie: Wir beschäftigen uns mit einem Computerspiel und müssen noch viel mehr rechnen! Aber so ist das mit dem Computer. Auch wenn der Benutzer "aussen" nur sich bewegende Raumfähren sieht - im Programm selbst ist ziemlich viel Mathematik und es muss ziemlich viel gerechnet werden. "Computed" eben.

Unterprogramme

Einführung: GOSUB und RETURN

In den ersten Kapiteln haben wir die grundlegenden Befehle PRINT, LET und GOTO kennengelernt. Prinzipiell können wir mit diesen Befehlen fast alles programmieren. Viele Dinge, die wir in den folgenden Kapiteln programmiert haben, wären aber allein mit diesen Befehlen aber sehr unkomfortabel, sehr, sehr unkomfortabel sogar, praktisch eigentlich doch unmöglich. Andere Dinge, wie die Sonderzeichen oder Input mit Prompt sind die Sahne auf dem Kaffee, aber im Notfall auch entbehrlich.

In diesem Kapitel kommen wir zu einem unscheinbar anmutenden Befehl, den wir bisher vermutlich noch gar nicht vermisst haben. Er heisst **GOSUB**. Aber er gehört keineswegs zur notfalls entbehrlichen Sahne, sondern eher zu den Dingen, ohne die die meisten Programmieraufgaben nur noch theoretisch, nicht mehr jedoch praktisch lösbar wären.

GOSUB funktioniert zunächst wie GOTO: GOSUB 200 springt in Zeile 200. Aber es ist etwas komfortabler als GOTO. Es merkt sich nämlich die *Absprungstelle*. Es gibt nun einen zweiten Befehl namens RETURN. Wenn das Programm an ein RETURN kommt, dann springt es wieder zurück zu dieser Absprungstelle. Ein Beispiel:

```
10 PRINT "Hier ist Zeile 10"  
20 GOSUB 100  
30 PRINT "Hier ist Zeile 30"  
40 END  
100 PRINT "Hier ist Zeile 100"  
110 RETURN
```

Das Programm startet bei Zeile 10, springt in Zeile 20 nach Zeile 100, in Zeile 110 dahin zurück, von wo es gekommen war, nämlich nach Zeile 20, führt dann Zeile 30 aus und endet in Zeile 40.

Diese Programm hat eine erkennbare *Struktur*: Es besteht aus einem *Hauptprogramm* in den Zeilen 10 bis 40 und einem *Unterprogramm* in den Zeilen 100 und 110. Mit GOSUB kann man *Unterprogramme* bauen.

Aber wozu braucht man Unterprogramme? Zum Erledigen von Unteraufgaben. Teilaufgaben. Der Witz besteht darin, dass man diese Teilaufgaben von *jeder Stelle des Hauptprogramms* aus erledigen lassen kann. Schauen wir uns wieder ein Beispiel an.

Wir begegneten im [String-Kapitel](#) dem Wunsch, einen String rechtsbündig ausgeben zu können. Dies ist so eine Teilaufgabe oder Unteraufgabe. Sie muss oft und an vielen Stellen eines entsprechenden Buchhaltungs- oder Tabellierungsprogramms ausgeführt werden. Deshalb nennt man solche Unterprogramme auch oft *Routinen*. Schauen wir zunächst, wie wir diese Rechtsbündig-Routine programmieren und dann, wie wir sie einsetzen können.

Eine Routine: Rechtsbündige Ausgabe von Strings

Nochmals zur Erklärung, was "rechtsbündig" eigentlich bedeutet: Man möchte eine Reihe von Zahlen untereinander so ausgeben, dass ihre Enden übereinander stehen.

```
123
 45
  6
```

ist eine rechtsbündige Ausgabe.

```
123
 45
  6
```

ist eine linksbündige Ausgabe. Für Zahlen sind natürlich meist nur rechtsbündige Ausgaben praktisch, da hier die zusammengehörigen Stellen untereinander stehen. Wenn wir aber mit dem PRINT-Befehl einfach untereinander Zahlen ausgeben, erscheinen sie linksbündig. Da müssen wir uns etwas einfallen lassen.

Man kann gedanklich die Aufgaben auf vielerlei Weise lösen.

1. Man nimmt einen zweiten String und kopiert den Inhalt des ersten rechtsbündig in den zweiten
2. Man verschiebt den Inhalt innerhalb des ersten Strings nach rechts.
3. Man schiebt am Anfang des Strings einen Leerstring geeigneter Länge ein.

Realisieren lässt sich auf dem C16 zunächst nur die dritte Variante, weil die ersten beiden verlangen, dass wir einzelne Zeichen im String verändern, was nicht so einfach geht. Das hatten wir schon beim Mastermind-Spiel.

```
10 LET S=LEN(A$)
20 LET IMAX=W-S //Stelle fest, wieviel Zeichen noch fehlen
30 LET I=0 //Zeile 30 bis 80: Baue den Verlängerungsstring B$ mit W-S Zeichen.
40 LET B$=""
50 IF IMAX<=0 THEN GOTO 100 //wenn keine Zeichen fehlen,
   dann überspringe den nächsten Teil
60 LET B$=B$+"_"
70 LET I=I+1
80 IF I<IMAX THEN GOTO 60
100 LET A$=B$+A$
```

Die Texte ab // sind Kommentare, die nicht abgetippt werden dürfen.

So, das war der harte Teil. Wenn Sie nicht jedes Detail verstanden haben: Macht überhaupt nichts. Der Witz an Routinen ist ja, dass man sie nicht verstehen muss, um sie benutzen zu können.

Bis jetzt ist das äusserlich ja ein ganz normales kleines Programm, ohne Ein- und Ausgabe. Wie wird es zu einem Unterprogramm? Ganz einfach: Man fügt am Ende noch ein RETURN ein:

```
...
110 RETURN
```

Aber nun ist das ja auch doof: Wir starten ja direkt ins Unterprogramm, ohne A\$ und W besetzt zu haben!

Auch kein Problem. Wir können ja eine Zeile 5 mit einem GOTO-Befehl einfügen, der uns ins Hauptprogramm befördert. Es steht nirgendwo geschrieben, dass Hauptprogramme immer am Anfang stehen müssen. Das Hauptprogramm selbst können wir ja z.B. ab Zeile 1000 laufen lassen.

Dann gibt es vorher sogar noch Platz für ein paar weitere Unterprogramme.

Im Hauptprogramm selbst setzen wir W, lassen den Benutzer einen String eingeben, machen ihn rechtsbündig, geben ihn aus und lassen den Benutzer den nächsten String eingeben.

```

5 GOTO 1000
10 LET S=LEN(A$)
20 LET IMAX=W-S //Stelle fest, wieviel Zeichen noch fehlen
30 LET I=0 //Zeile 30 bis 80: Baue den verlängerungsstring B$ mit W-S Zeichen.
40 LET B$=""
50 IF IMAX<=0 THEN GOTO 100 //wenn keine Zeichen fehlen,
   dann überspringe den nächsten Teil
60 LET B$=B$+"_"
70 LET I=I+1
80 IF I<IMAX THEN GOTO 60
100 LET A$=B$+A$
110 RETURN
1000 LET W=20
1010 SCNCLR
1020 INPUT "GEBEN SIE EINEN STRING EIN (Q=Quit)",A$
1030 GOSUB 10
1040 PRINT A$
1050 IF A$<>"Q" THEN GOTO 1020

```

Sie können das Programm ganz leicht erweitern, indem Sie die Eingabe am unteren Bildschirmende vornehmen lassen und die Tabelle mittels CHAR ,X,Y,A\$-Befehl in der oberen Bildschirmhälfte aufbauen. Und meinetwegen auch gleich noch Summe und Mittelwert ausgeben. Dann haben wir schon ein kleines Statistik-Programm beieinander!

Notwendige Ergänzung: REM, der Kommentar-Befehl

Es gibt einen Befehl, hätten wir ihn am Anfang erklärt, er wäre uns total widersinnig erschienen: REM. REM bewirkt, dass das, was dahinter folgt, einfach nur abgespeichert wird, aber ansonsten *nichts* bewirkt. Wirklich gar nichts. Kein Rechnen, keine Bildschirmausgabe, kein Speichern, nichts. Warum das? Nun, wenn wir ein Programm mit vielen Zeilen schreiben und die Programme komplexer werden, dann können wir sie nicht mehr rein aus den Programmzeilen heraus verstehen. Wir müssen wenigstens den einzelnen Programmteilen Namen geben können, besonders den Routinen. So dass wir als Programmierer sehen: "Ah ja! Zeile 10 bis 110 ist die Rechtsbündig-Routine"! Und das geht mit einem *Kommentar*. Der C16 und erst recht Ihr PC bietet genug Programmspeicher, um auch noch solche Kommentare in den Speicher mitaufzunehmen.

Eine Kommentarzeile sieht so aus:

```
10 REM ICH BIN EIN KOMMENTAR
```

Kommentieren wir also unser Programm ein bisschen:

```

4 REM SPRUNG INS HAUPTPROGRAMM
5 GOTO 1000
8 REM RECHTSBUENDIG-ROUTINE
9 REM -----
10 LET S=LEN(A$)
20 LET IMAX=W-S
30 LET I=0
40 LET B$=""
50 IF IMAX<=0 THEN GOTO 100
60 LET B$=B$+"_"

```

```

70 LET I=I+1
80 IF I<IMAX THEN GOTO 60
100 LET A$=B$+A$
110 RETURN
1000 REM HAUPTPROGRAMM

1005 LET W=20
1010 SCNCLR
1020 INPUT "GEBEN SIE EINEN STRING EIN (Q=Quit)",A$
1030 GOSUB 10
1040 PRINT A$
1050 IF A$<>"Q" THEN GOTO 1020

```

So sieht doch das Programm gleich viel lesbarer aus. Vielleicht werden Sie die Frage stellen: Ist das denn nötig, die Kommentare ins Programm selbst zu schreiben? Ich kann mir das Programm doch ausdrucken und dann die Kommentare von Hand dazufügen!

Können Sie machen. Aber wenn Sie das Programm verändern, werden Sie immer neue Ausdrücke anfertigen und da müssen Sie dann sämtliche Kommentare immer neu reinschreiben. Ausserdem: Lassen Sie das Programm ein paar Monate liegen. Wenn Sie es dann mal wieder in den Speicher laden, wissen Sie dann, wo der aktuelle Papierausdruck dafür noch ist? Nein, es hat schon ziemliche Vorteile, den Kommentar in elektronischer Form und untrennbar mit dem Programm zusammen abzuspeichern.

Anwendung: Das Honorarabrechnungsprogramm

Wir hatten schon einige Kapitel zuvor als Übungsaufgabe, ein Honorarabrechnungsprogramm zu schreiben. Und das geht nur dann vernünftig, wenn wir die Ergebnisse - Stundenzahl, Mehrwertsteuer, Nettobetrag, Bruttobetrag - rechtsbündig auf dem Bildschirm ausgeben. Das Programm müsste so aussehen:

```

1010 PRINT "HONORARABRECHNUNG"
1020 LET STUNDSATZ=30
1030 LET MWST=0.16
1040 INPUT "ANZAHL STUNDEN (ENDE=99)", NSTUND
1050 IF NSTUND=99 THEN END
1060 LET A$=STR$(NSTUND)
>>Mache A$ rechtsbündig
1070 LET NST$=A$
1080 LET A$=STR$(NSTUND*STUNDSATZ)
>>Mache A$ rechtsbündig
1090 LET NETTO$=A$
1100 LET A$=STR$(VAL(NETTO$)*MWST)
>> Mache A$ rechtsbündig
1110 LET MW$=A$
1120 LET A$=STR$(VAL(NETTO$)+VAL(MW$))
>> Mache A4 rechtsbündig
1130 LET BRUTT$=A$
1140 PRINT "ERGEBNIS"
1150 PRINT "-----"
1160 PRINT "STUNDEN: ";TAB(20);NST$
1170 PRINT "NETTO: ";TAB(20);NETTO$
1180 PRINT "MWST: " ; TAB(20);MW$
1190 PRINT "BRUTTO: ";TAB(20);BRUTT$
1200 GOTO 1040

```

Wie man sieht, habe ich in den Programmcode Textzeilen eingefügt, an den Stellen, an denen eine Unteraufgabe zu erledigen ist. Wie bauen wir die Unterprogramme nun oben ein? Nun, mit Ihren

bisherigen Kenntnissen (vor diesem Kapitel) hätten Sie das so gelöst: Sie hätten einfach den Code an der jeweiligen Stelle immer wieder eingefügt, wo er gebraucht wird. Den nachfolgenden Code hätten Sie nach hinten verschoben. In dieser Versuchung steht man als Programmierer immer. Dass man, statt die Teilaufgabe ordentlich und allgemeingültig zu lösen, eine spezifische Lösung mitten in das Hauptprogramm "reinbastelt". Ergebnis:

- Jede Menge Tipparbeit
- Fehlerträchtig: Sie können den Code dreimal richtig eintippen, beim vierten Mal machen Sie dann doch einen Fehler.
- Riesig umständlich zu behandeln, wenn das Programm getestet wird. Jedesmal, wenn ein Fehler im Unterprogramm auftaucht, müssen Sie es an vier Stellen korrigieren!
- Unflexibel: Weil es so viel Mühe kostet, jedes Mal das Unterprogramm in den Hauptprogrammtext einzuschleusen, werden Sie bald eine schlechtere, aber einfach zu tippendere Lösung bevorzugen.
- Total unübersichtlich zu lesen. Man wird im Programmtext von der Hauptaufgabe in das Bearbeiten irgendwelcher Unteraufgaben abgelenkt.
- Sie werden in Ihrer späteren Arbeit dem Problem immer wieder begegnen und sich daran erinnern, das Problem schon mal gelöst zu haben. Da der zugehörige Code aber in Ihren alten Programmen irgendwo reingebastelt ist, werden Sie ihn nicht mehr wiederfinden und - wenn doch - ihn nicht mehr verstehen. Sie werden also die gleiche Aufgabe immer und immer wieder neu lösen müssen. Welche Verschwendung!

Diese ganzen Probleme löst der kleine GOSUB-Befehl elegant und mit einem Schlag:

```

4 REM SPRUNG INS HAUPTPROGRAMM
5 GOTO 1000
8 REM RECHTSBUENDIG-ROUTINE
9 REM -----
10 LET S=LEN(A$)
20 LET IMAX=W-S
30 LET I=0
40 LET B$=""
50 IF IMAX<=0 THEN GOTO 100
60 LET B$=B$+"_"
70 LET I=I+1
80 IF I<IMAX THEN GOTO 60
100 LET A$=B$+A$
110 RETURN
1000 REM HAUPTPROGRAMM
1001 REM -----
1005 LET W=20
1010 PRINT "HONORARABRECHNUNG"
1020 LET STUNDSATZ=30
1030 LET MWST=0.16
1040 INPUT "ANZAHL STUNDEN (ENDE=99)", NSTUND
1050 IF NSTUND=99 THEN END
1060 LET A$=STR$(NSTUND)
1064 REM A$ RECHTSBUENDIG
1065 GOSUB 10
1070 LET NST$=A$
1080 LET A$=STR$(NSTUND*STUNDSATZ)
1084 REM A$ RECHTSBUENDIG
1085 GOSUB 10
1090 LET NETTO$=A$
1100 LET A$=STR$(VAL(NETTO$)*MWST)
1104 REM A$ RECHTSBUENDIG
1105 GOSUB 10
1110 LET MW$=A$

```

```

1120 LET A$=STR$(VAL(NETTO$)+VAL(MW$))
1124 REM A$ RECHTSBUENDIG
1125 GOSUB 10
1130 LET BRUTT$=A$
1140 PRINT "ERGEBNIS"
1150 PRINT "-----"
1160 PRINT "STUNDEN: ";TAB(20);NST$
1170 PRINT "NETTO: ";TAB(20);NETTO$
1180 PRINT "MWST:" ; TAB(20);MW$
1190 PRINT "BRUTTO:";TAB(20);BRUTT$
1200 GOTO 1040

```

Das Tolle daran: Jedesmal, wenn wir bei der Bearbeitung des Hauptprogramms mal wieder "Rechtsbündig machen" brauchen: Einfach GOSUB 10, das war's. Unsere Programme werden dadurch sehr mächtig. Und auch besser lesbar. Jemand, der sich für die Unterprogramme nicht interessiert, der braucht sie auch nicht zu lesen. Der kann sich auf das Lesen des Hauptprogramms beschränken und es dadurch viel schneller verstehen.

Mehranweisungszeilen

Der C16 bietet die Möglichkeit, mehrere Anweisungen in eine Zeile zu packen. Z.B.

```

10 INPUT "GEBEN SIE ZWEI ZAHLEN EIN",A:INPUT B:
IF A<B THEN PRINT "A IST KLEINER ALS B":IF A>B
THEN PRINT "A IST GROESSER ALS B":IF A=B THEN
PRINT "A IST GLEICH B":GOTO 10

```

Man fügt also die zweite Anweisung hinter die erste, getrennt durch einen Doppelpunkt.

Mit diesem Doppelpunkt kann man - wie im obigen Beispiel zu sehen - viel Unsinn anrichten. Normalerweise werden Programme total unübersichtlich und schwer zu editieren, wenn man mehrere Anweisungen in eine Zeile packt. Es gibt nur ganz wenige Ausnahme. Die eine sind mehrere gleichartige Zuweisungen. Z.B. 10 LET X=1:LET Y=10

Hier fördert es sogar die Übersicht, wenn man die beiden Zuweisungen in eine Zeile schreibt, da man sofort sieht, dass es sich um die gleiche Sache handelt: Z.B. Eigenschaften einer Position oder einer geometrischen Figur o.ä.

Die zweite sind Kommentare. Es kann sinnvoll sein, die Kommentare hinter die Anweisungen zu schreiben: 1010 GOSUB 10:REM RECHTSBUENDIG

Hier kann man ev. sogar noch die Zuweisung des Arguments mit in die Zeile packen:

```

1010 LET A$="ABC___":GOSUB 10:REM RECHTSBUENDIG

```

Zunächst sieht das ungewohnt aus, aber mit der Zeit wird man diese Trilogie vor dem inneren Auge als einen eigenen "Befehl" der Form RECHTSBUENDIG(A\$) lesen. Das macht Sinn.

Übungsaufgaben

Übungsaufgabe 1: Manchmal ist es notwendig, Strings in Zahlen umzuwandeln. Dazu haben wir die VAL()-Funktion. Diese hat aber den Nachteil, dass sie Null zurückgibt, egal, ob im Argumentstring etwas stand, was gar keine Zahl war oder tatsächlich eine Null. Das heisst, man kann eine Routine gut gebrauchen, die den String A\$ darauf prüft, ob es sich dabei um eine Zahl

handelt. Schreiben Sie so eine Routine und bauen Sie sie in das Honorarabrechnungsprogramm ein, indem Sie die Stundenzahl nicht als NSTUND-Zahl, sondern als String entgegennehmen.

Übungsaufgabe 2: Beim Mastermind-Spiel waren wir mit der Teilaufgabe konfrontiert, ein Zeichen an einer ganz bestimmten Stelle eines Strings zu verändern. Mit Hilfe der Befehle LEFT\$ und RIGHT\$ muss man dazu den String um das zu ersetzende Zeichen herum neu zusammensetzen. Ziemlich umständlich. Schreiben Sie ein Routine und bauen Sie diese in das Mastermind-Programm ein. Vergessen Sie die Möglichkeit von Mehrfachanweisungen nicht!

Übungsaufgabe 3: Im Mondlandespiel muss man immer wieder eine Raumfähre auf den Bildschirm zeichnen. Auch das ist eine Teilaufgabe für sich: Eine Routine, die eine Mondlandefähre an die Position X,Y auf den Bildschirm zeichnet. Und eine Routine, die die Mondlandefähre an der Stelle X,Y wieder löscht. Schreiben Sie diese Routinen und bauen Sie diese in das Spiel ein. Sie werden sehen, dass das Programm plötzlich viel einfacher und übersichtlicher wird!

Farbe und SOUND

Die Aufteilung des C16-Bildschirms

Obwohl es sich beim C16 um einen Einsteigercomputer handelte, kann er mit Farben umgehen. Und zwar kann er bis zu 128 Farben gleichzeitig am Bildschirm anzeigen. Das war mehr als fast alle anderen Homecomputer. (Sinclair ZX81: Max. 2, Sinclair Spectrum: Max. 4, Commodore C64: Max. 16, Amstrad-Schneider CPC: Max. 27, Atari 600/800XL: Max. 128.) Ihr PC hat keine Probleme, Tausende von Farben gleichzeitig darzustellen.

Die Darstellung von Farben ist nicht so etwas Zentrales für die Kunst des Programmierens und sie funktioniert überall, in jedem Programmiersystem anders, deshalb wird sie hier nicht so ausführlich erklärt.

Man muss unterscheiden, welchen Bereich des Bildschirms man ansprechen möchte:



- Den Rand (Border)
- Den Vordergrund (Foreground, die Farbe der Zeichen)
- Den Hintergrund (Background)

Von allen drei Bereichen wird die momentane Farbe in jeweils einem Farbspeicher festgehalten. Es gibt also den Hintergrund-Farbspeicher (Nr. 0), den Vordergrund(farb)-Speicher (Nr. 1) und den Rand-Speicher (Nr. 4). (Zusätzlich gibt es noch zwei weitere Speicher, die s.g. Multicolor-Speicher Nr. 2 und 3.)

Die Farben und der Befehl COLOR

Der Befehl COLOR (engl. "Farbe") dient dazu, den Inhalt der Farbspeicher zu ändern. Beim Rand funktioniert das am einfachsten und wir können das gleich gefahrlos auf der Kommandozeile

testweise ausführen:

COLOR 4, 5 + ENTER ändert die Randfarbe in Violett. 4 ist dabei der Farbspeicher (Rand) und 5 die Nummer der Farbe. Es gibt Farben 1 bis 16. Die zugehörigen Farben:

1	Schwarz
2	Weiss
3	Rosa
4	Türkis
5	Violett
6	Hellgrün
7	...hier dürfen Sie weitermachen
8	
9	
10	
11	
12	
13	
14	
15	
16	

Die Tabelle habe ich nicht ganz ausgefüllt. Sie können den COLOR-Befehl in ein kleines Programm einbinden, das Ihnen hintereinander jede Farbe als Randfarbe zeigt und die dazugehörige Nummer und dann wartet, bis Sie eine Taste drücken. Dann geht's weiter zur nächsten Farbe. Das sollte für Sie ein Kinderspiel sein.

Helligkeit

Man kann dem COLOR-Befehl noch ein drittes Argument übergeben, die Helligkeit, eine Ganzzahl zwischen 0 und 7: COLOR 4,3,0 setzt die Farbe Nr. 3 auf ganz dunkel, COLOR 4,3,7 auf ganz hell. Dadurch kommen (theoretisch) die 128 Farben zustande: 16 mal 8 Farben. In Wirklichkeit sind es weit weniger, da es bei Schwarz keine Abstufungen gibt und die 16 Farbnummern nur 11 faktisch verschiedenen Farben entsprechen. Aber für's erste reicht das auch.

Hintergrund

Die Änderung des Hintergrund funktioniert im Prinzip genauso wie die des Randes (nur mit Farbspeicher 0). Aber man muss aufpassen: In dem Moment, in dem man den Hintergrund auf die Vordergrundfarbe setzt (im Regelfall werden Sie Schwarz, Farbe Nr. 1, als Vordergrundfarbe haben), können Sie nichts mehr lesen! Keinen Programmtext, kein READY und kein Kommando! Sie müssen dann entweder blind schreiben, oder den C16 "ausschalten" (Reset im Menü "Files"). Es ist zu empfehlen, in die obige Farbtabelle zu schauen, bevor man hier Experimente macht.

Bezüglich der "Blindheits"gefahr gilt hier natürlich das Gleiche. Aber der Vordergrund funktioniert auch noch ein bisschen anders. Die Änderung wirkt sich zunächst überhaupt nicht aus, denn sie bezieht sich nicht auf den Vordergrund an sich, sondern auf die *aktuelle* Schriftfarbe. Man muss sich das vorstellen wie einen Kugelschreiber, bei dem man die Farbmine ausgetauscht hat. Das, was sich mit Rot bisher geschrieben habe, bleibt rot, aber das was ich ab jetzt schreibe, wird grün.

Auf diese Art und Weise kann nun die Vordergrundfarbe Zeichen für Zeichen ändern und alle 128 Farben gleichzeitig auf den Bildschirm bringen. Machen Sie das einmal!

SOUND

Es sei hier nebenbei bemerkt, dass der C16 auch über SOUND verfügt. Dabei handelt es sich allerdings um eine sehr einfache Tonausgabe, die mit einer Soundkarte des PC's nicht zu vergleichen ist. Aber für kleine Spiele kann sie ganz nett sein. Leider hat hier der Yape-Emulator, der die Funktionen des C16 in die des PC's umsetzt, einen Defekt, so dass der C16-SOUND am PC leider nicht funktioniert.

Schleifen

FOR-Schleifen

Wir haben während der vergangenen Kapitel immer wieder im Programm etwas *durchzählen* müssen. Das ist eine sehr häufige Aufgabe. Führe etwas dreimal oder zehnmal oder tausendmal durch. Bis jetzt haben wir uns immer mit einer GOTO-Konstruktion beholfen:

```
10 LET I=0
20 REM TUE ETWAS
30 LET I=I+1
40 IF I<99 THEN GOTO 20
```

Der C16 bietet hierfür eine elegantere Lösung, die Anweisung FOR. Man schreibt vor die Befehle, die man N-mal ausführen lassen möchte, die Anweisung FOR I=1 TO N. Und dahinter die Anweisung NEXT N:

```
10 FOR I=0 TO 99
20 REM TUE ETWAS
30 NEXT I
40 PRINT "FERTIG"
```

Den Rest übernimmt die FOR-Anweisung selbst: Die Initialisierung der Zählvariable I (es kann auch eine andere Variable sein, z.B. J oder BUMPF) und das Weiterzählen. LET I=0 und LET I=I+1 können also entfallen. Der Computer beginnt bei Zeile 10, I mit 0 zu besetzen. Dann führt er Zeile 20 aus. Bei Zeile 30 springt er zur Zeile 10 zurück (jeweils ein FOR und NEXT gehören immer zusammen!), erhöht I um 1, prüft, ob es kleiner gleich 99 ist. Wenn ja, geht er wieder zur Zeile 20, wenn Nein, dann macht er hinter dem FOR-NEXT-Block, also bei Zeile 40 weiter.

Wenn so ein mehrmaliges "im-Kreis-Gehen" in einem Programm vorkommt, ein Teil des Programms also mehrmals wiederholt wird, nennt man das eine "Schleife". Die FOR-Anweisung ermöglicht es, schnell und elegant Zählschleifen zu programmieren.

Beispiele brauchen wir hier eigentlich nicht mehr zu geben, es kamen schon genügend in der vorangegangenen Kapiteln vor. Hier noch das Beispiel aller ASCII-Zeichen mit FOR-Schleife:

```
10 FOR I=32 TO 127
20 PRINT I,CHR$(I)
30 NEXT I
```

Verschachtelte Schleifen

Man kann auch mehrere Schleifen ineinanderschachteln:

```
10 REM SUCH E PRIMZAHLEN BIS 1000
20 FOR I=1 TO 1000
30 FOR J=2 TO I/2
40 IF INT(I/J)=I/J THEN GOTO 70
50 NEXT J
```

```
60 PRINT I
70 NEXT I
```

Im obigen Beispiel lernen wir gleich auch noch etwas anderes: Schleifen abubrechen. Es können Situationen entstehen, in denen es nicht sinnvoll oder notwendig ist, alle Schleifen voll zu durchlaufen. In diesem Fall kann man die Schleife abbrechen, indem man einfach hinausspringt, wie es in Zeile 40 geschieht.

WHILE/UNTIL-Schleifen

Der C16 beherrscht noch eine weitaus mächtigere Schleifenart, die WHILE-, bzw. UNTIL-Schleifen. Das Abbruchkriterium ist hier nicht eine Zählervariable, sondern es kann eine beliebige Bedingung sein. Die WHILE-Schleifen beginnen mit DO WHILE und enden mit dem Schlüsselwort LOOP anstelle von NEXT.

```
10 LET A$="A"
20 DO WHILE A$<>"E"
30 INPUT "GEBEN SIE EINEN STRING EIN",A$
40 LOOP
```

Dieses Progrämmchen macht nichts anderes, als vom Benutzer andauernd eine Stringeingabe abzuverlangen. Es bricht ab, sobald der Benutzer "E" eingibt, da es nur weitermacht, solange A\$<>E ist. In Zeile 10 mussten wir A\$ auf einen Wert <>"E" initialisieren, damit die WHILE-Schleife überhaupt betreten wird.

Wir können das noch optimieren, indem wir aus der WHILE-Schleife eine UNTIL-Schleife machen. Beide sind sehr verwandt. Es ist ein bisschen die Frage des Programmierstils, ob man eher die eine oder die andere Variante einsetzt. Bei der UNTIL-Schleife kommt die Schleifenbedingung an den Schluss und bezeichnet nicht die Bedingung zur Fortsetzung der Schleife, sondern die Bedingung für den Abbruch.

```
10 DO
20 INPUT "GEBEN SIE EINEN STRING EIN",A$
30 LOOP UNTIL A$="E"
```

Randbemerkung: Der C16 lässt auch die Variante DO UNTIL ... LOOP und DO ...LOOP WHILE zu. Da diese aber keine Entsprechung in anderen Programmiersprachen haben und nicht notwendig für die Vollständigkeit einer Programmiersprache sind, werden sie hier weggelassen. In allen "klassischen" Programmiersprachen, von PASCAL über C++ bis Java gibt es die drei Schleifentypen FOR, WHILE und DO-UNTIL.

Übungsaufgabe: Schreiben Sie ein "Game of Life". Das ist die Simulation eines Biosystems, in dem Tiere einen Vorrat von Pflanzen auffressen. Gibt es keine Pflanzen mehr, pflanzen sich die Tiere nicht mehr fort und verhungern. Dann können sich die Pflanzen wieder vermehren. Dann gibt es wieder genug zu fressen und die Tiere können sich auch wieder vermehren. Wir realisieren diese Simulation, indem wir den Bildschirm des C16 in ein Biotop mit Pflanzen und Tieren verwandeln: Jede Zelle, in die ein Buchstabe kommt, ist ein Tier oder eine Pflanze. Tiere sind rot, Pflanzen sind grün. Wir malen also ein grünes oder rotes Viereck. Wir haben also 40 mal 25 Vierecke. Die äusserste Schleife ist die Zeitschleife. In jedem Zeitschritt geht der Computer alle Zeilen (Zeilenschleife) und alle Spalten (Spaltenschleife) durch und schaut nach jeder Zelle. Ist die Zelle rot, dann muss nach den Nachbarzellen geschaut werden, denn das Tier braucht was zum Fressen. Gibt es ein grünes Nachbarfeld, wird dieses rot gemacht (Tier frisst und pflanzt sich fort), ansonsten wird es grün. (Tier stirbt und eine neue Pflanze entsteht dort, da hier kein Pflanzenfresser mehr existiert.) Am Anfang müssen Tiere und Pflanzen zufällig gestreut werden.

Damit Sie wissen, welche Zelle grün und welche rot ist, sollten Sie folgendes Unterprogramm verwenden:

```

1000 REM GETCOLOR OF X,Y-POSITION
1010 REM COL=COLOR
1020 REM LUM=LUMISCENCE
1030 IF (X<0) THEN PRINT "ERROR IN GETCOLOR()":STOP
1040 IF (X>39) THEN PRINT "ERROR IN GETCOLOR()":STOP
1050 IF (Y<0) THEN PRINT "ERROR IN GETCOLOR()":STOP
1060 IF (Y>23) THEN PRINT "ERROR IN GETCOLOR()":STOP
1070 LET H=PEEK(2048+Y*40+X)
1080 LET LUM=INT(H/16)
1090 LET COL=H-LUM*16+1
1100 RETURN

```

Sie müssen also in X und Y Spalte und Zeile der gesuchten Zelle eingeben und erhalten in COL und LUM Farbe und Helligkeit zurück. Zeile 1070 können Sie nicht verstehen. Die Funktion "PEEK" ist für Ihre weitere Programmierkarriere aber auch nicht wichtig. Ansonsten finden Sie aber einige andere interessante "Figuren": Sie sehen, wie ich den Routinenkopf gestaltet habe:

Funktionsbeschreibung der Routine und die Beschreibung der Ausgabevariablen. Sie sehen dann vier Zeilen mit Prüfungen der Eingabevariablen X und Y. Diese könnten ja falsch sein. Wenn sie ausserhalb der gültigen Wertebereichs sind, gibt das Programm eine Fehlermeldung mit einer ungefähren Ortsangabe aus und stoppt. Da in der IF-Anweisung jeweils zwei Befehle stehen müssen und um sich grossartige GOTO-Orgien zu ersparen, habe ich den zweiten Befehl mit einem Doppelpunkt hinter den ersten gehängt und damit gebündelt.

So, nun haben Sie das Rüstzeug für Ihr "Game of Life"! Viel Spass damit!

Arrays, grössere Programme und Bugs

Was sind Arrays?

Arrays sind eine besondere Form von Speichern. Wir haben bisher zwei Typen von Speichern/Variablen kennengelernt: Zahlvariablen und Stringvariablen. Jetzt kommt ein dritter hinzu. Was fehlt denn?

Nun, jetzt, da unsere Programmieraufgaben langsam grösser werden, kann es passieren, dass wir nicht nur ein, zwei oder drei Variablen in unserem Programm brauchen, sondern vielleicht ein Dutzend oder mehr. Und manchmal wäre es gut, wenn diese Speicher nicht alle individuelle Namen tragen würden, sondern durchnummeriert wären. A1, A2, A3 usw. Zum Beispiel könnte es für manche Probleme nützlich sein, eine Grösse zu zehn verschiedenen Zeitpunkten abzuspeichern. Soweit so schön.

Aber wir können ja die Variablen dann A1, A2, A3 benennen. Das schon. Aber wenn wir nun zum Beispiel alle zehn Variablen auf null setzen wollen, müssen wir zehn LET-Anweisungen schreiben. Da hilft kein Weg drumrum:

```
10 LET A1=0
20 LET A2=0
30 LET A3=0
usw.
```

Bei zehn geht's ja noch, aber bei hundert wird das Nonsense. Man sollte die Variablen alle über ihre Nummer ansprechen können. Nehmen wir an, die Nummer steht im Zähler I, dann müsste man sagen können: LET AI=0. Statt LET A1=0.

Genau das kann der C16 auch. Nur heisst es nicht LET AI=0. Sondern, damit man Nummer und Variablennamen auseinanderhalten kann: LET A(I)=0. Zwischen den Klammern steht die Nummer, der s.g. Index. A besteht dann aus mehreren Einzelvariablen. Insgesamt heisst das dann ein *Array* oder - zu deutsch - ein Feld oder eine Feldvariable.

Wenn wir nun die FOR-Schleifen aus dem Vorkapitel verwenden, dann können wir mit Feldern ganz elegant umgehen:

```
10 FOR I=0 TO 99
20 LET A(I)=0
30 NEXT I
```

Auf diese Art und Weise haben wir auf einen Schlag hundert Variablen initialisiert. Aber warum zählt der Knirch von Null ab? Erstens zählen Programmierer immer von Null ab. Mich hat einmal jemand gefragt, warum sie das tun. Ich wollte ihm das ganz lässig mit einem Satz hinwerfen und habe mich dabei ziemlich blamiert. Man kann das schon erklären, aber es ist nicht so ganz einfach. Es hat hauptsächlich damit zu tun, dass man bei Zählern, die von Null ab zählen, besser den Zähler in 10er-Gruppen oder 4er-Gruppen o.ä. einteilen kann, und zwar mit dem Modulooperator. Aber das sei hier nur am Rande erwähnt. Es gibt auch noch ein Zweitens: Computer zählen meistens von Null ab. So auch der C16. Egal wie wir es machen und wollen, wenn wir mit einem Array arbeiten, gibt

es das Element Null.

Jetzt fehlt hier noch was. Im Gegensatz zu einzelnen Variablen muss der C16 wissen, wie gross das Array denn sein soll, dass wir im Programm erwähnen. Beim ersten Auftreten eines A()-Ausdrucks muss er also über die Grösse von A() informiert sein. Und das machen wir mit einem DIM-Befehl. Der DIM-Befehl muss vor dem ersten Auftreten von A() im Programm einmal erscheinen. DIM A(100) reserviert ein Array von 100 Zahlenspeichern. Man nennt dieses Erklären, dass man nun einen bestimmten Speicher haben will, *Deklarieren* von Variablen.

Und nun Vorsicht mit dem Nullzählen! Wollen wir die Elemente X(1) bis X(10), so dürfen wir nicht DIM X(10) deklarieren, sonst haben wir ein Element zu wenig! Denn dann ist X(0) bis X(9) deklariert - der C16 zählt immer von Null ab! Wir müssen also DIM X(11) deklarieren. Und lassen das nullte Element eben unbenutzt.

Beispiel für die Verwendung von Arrays: Ein Statistikrechner

Das nächste Programm ist ein bisschen grösser und stellt einen kleinen Statistikrechner dar. Es deklariert ein Feld X(), und lässt es durch den Benutzer füllen. Anschliessend rechnet es Minimum, Maximum, Mittelwert und Streuung der Zahlen aus, zeichnet eine Häufigkeitsverteilung als Balkendiagramm auf den Bildschirm und gibt daneben die vier Ergebnisse aus.

```

10 REM STATISTIKRECHNER
20 REM -----
30 DIM X(100)
35 DIM XG(10)
40 REM HAUPTPROGRAMM
50 REM .....
55 DO
60 GOSUB 500:REM FUELLE X()
70 IF (N>0) THEN GOSUB 800:REM WERTE X() AUS
80 REM WERTE STEHEN IN MEAN, STD, XMIN, XMAX
90 IF (N>0) THEN GOSUB 1500:REM ZEICHNE BALKENDIAGR
100 INPUT A$
110 LOOP UNTIL A$<>"J"
120 SCNCLR
130 END
500 :
510 REM X() FUELLEN
520 REM .....
530 SCNCLR
540 COLOR 4,1,0:COLOR 1,3,4:COLOR 2,1,0
550 CHAR ,0,0,""
560 PRINT "=====
570 PRINT "          STATISTIKRECHNER"
580 PRINT "=====
590 REM DIESE UEBERSCHRIFT KOENNEN SIE MIT HILFE DER
600 REM GRAFIKZEICHEN NOCH SCHOENER MACHEN.
610 PRINT:PRINT
620 COLOR 1,6,4
635 LET N=0:REM ANZAHL DER DATEN
640 LET A=0
650 DO WHILE (A<>-999)
640 GOSUB 2000:REM EINGABE AUFFORDERN
650 LET X(N)=A
660 LET N=N+1
670 IF (N>99) THEN GOSUB 2200:RETURN:REM
MELDUNG: MAX 100 ZAHLEN

```

```

680 LOOP
690 RETURN
799 :
800 REM AUSWERTUNG
810 REM .....
820 LET SUM=0
830 LET SQSUM=0
840 FOR I=0 TO N-1
850 SUM=SUM+X(I)
860 SQSUM=SQSUM+X(I)!2 ! steht für Pfeil nach oben (Potenzzeichen)
870 IF I=0 THEN IMIN=0:XMIN=X(I)
880 IF I=0 THEN IMAX=0:XMAX=X(I)
890 IF (X(I)XMAX) THEN XMAX=X(I):IMAX=I
910 NEXT I
920 MEAN=SUM/N
930 STD=-1
940 IF N>1 THEN STD=SQR(1/N*SQSUM-MEAN!2)
950 RETURN
1499 :
1500 REM BALKENDIAGRAMM
1510 REM .....
1520 SCNCLR
1530 COLOR 1,3,4
1540 CHAR ,10,0,"HAEUFIGKEITSDIAGRAMM"
1550 GOSUB 2100:REM GRUPPIERE DATEN NACH XG()
1560 GOSUB 2200:REM SUCHE MAXIMUM IN XG()
1570 LET DY=XGMAX/20
1580 FOR I=0 TO 9
1590 COLOR 1,4,4
1600 LET ICOL=I*2+5
1610 LET IROW2=25
1620 LET IROW1=24-INT(XG(I)/DY+0.5)
1630 FOR J=IROW1 TO IROW2
1640 CHAR ,ICOL,J,CHR$(166)
1650 NEXT J
1660 NEXT I
1670 REM AUSGABE ACHSENSCHRIFT
1680 CHAR ,5,25,STR$(XMIN)
1690 CHAR ,23,25,STR$(XMAX)
1695 LET XMID=(XMAX-XMIN)/2
1700 CHAR ,14,25,STR$(XMID)
1710 REM AUSGABE DER STATISTIK
1720 LET ICOL=30:LET IROW1=5
1730 CHAR ,ICOL,IROW1,"MEAN:"+STR$(MEAN)
1750 CHAR ,ICOL,IROW1+1,"STREU:"+STR$(STD)
1760 CHAR ,ICOL,IROW1+2,"MIN:"+STR$(IMIN)
1770 CHAR ,ICOL,IROW1+3,"MAX:"+STR$(IMAX)
1775 REM EINGABE: WEITERMACHEN?
1780 CHAR ,ICOL-10,IROW1+5,"NOCH EINE STATISTIK? (J/N)"
1790 RETURN
1999 :
2000 REM EINGABE AUFFORDERN
2010 CHAR ,0,5,"GEGEN SIE ZAHL NR "+STR$(N)+" EIN"
2020 CHAR ,0,6,"(ENDE: -999)"
2030 CHAR ,0,7,"":INPUT A
2040 REM EINGABEBEREICH LOESCHEN
2050 FOR I=5 TO 7:CHAR ,0,I,
"_____":NEXT I
2060 RETURN
2100 REM GRUPPIERE DATEN
2110 FOR I=0 TO 9
2120 LET XG(I)=0

```

```

2130 NEXT I
2140 LET DX=(XMAX-XMIN)/10
2150 FOR I=0 TO N-1
2160 LET IG=(X(I)-XMIN)/DX
2170 LET XG(IG)=XG(IG)+1
2180 NEXT I
2190 RETURN
2200 REM SUCHE MAXIMUM IN XG
2210 FOR I=0 TO 9
2220 IF I=0 THEN IMAX=I:XGMAX=XG(I)
2230 IF XG(I)>XGMAX THEN IMAX:=I:XGMAX=XG(I)
2240 NEXT I
2250 RETURN

```

Dieses Programm soll mehr als nur die Verwendung von Arrays zeigen. Weiteres gleich weiter unten. Aber zunächst mal zu den Arrays: Es werden zwei benutzt, X() und XG() und sie werden gleich am Anfang mittels eines DIM-Befehls deklariert. Man sollte die DIM-Befehle alle an den Anfang und beieinander schreiben, damit man den Überblick darüber behält, welche Felder man schon deklariert hat und welche nicht.

Die Benutzung ist nicht weiters schwierig. Sie sehen, dass die einzelnen Elementvariablen der Arrays nie anders als in einer Schleife angesprochen werden. Z.B. Zeile 650 oder 850. Spätestens jetzt müsste klar sein, dass die Aufgabe ohne Arrays gar nicht gelöst hätte werden können, denn erstens wäre es unmöglich gewesen, die ganzen Schleifen "aufzuwickeln" und explizit jeden Schleifendurchgang hinzuschreiben und zweitens ist beim Schreiben des Programms noch gar nicht klar, wie lang die Schleifen eigentlich sein werden, d.h. mit wieviel Elementen überhaupt gerechnet wird.

Ein paar Empfehlungen für's Programmieren

Nun noch ein paar Erklärungen zum Programm selbst. Ich vermute, hätten Sie das Programm als Anfänger selbst geschrieben, Sie hätten es etwas anders gemacht. Folgende Merkmale werden Sie entdecken:

- Viele Kommentare
- Gliederung des Programms in lauter Unterprogramme. Das Hauptprogramm ist sehr kurz.
- Verwendung vieler Schleifen. (Und aller drei Schleifentypen, die Sie im letzten Kapitel kennengelernt haben.)
- Dadurch erübrigt sich die Verwendung von GOTO's

Tatsächlich ist der GOTO-Befehl ein Merkmal sehr einfacher Programmierung, wie wir sie am Anfang kennengelernt haben. Bei längeren Programmen macht ein unüberlegtes Verwenden von GOTO's das Programm unkontrollierbar. Hier ein abschreckendes Beispiel, um deutlich zu machen, was ich meine:

```

10 PRINT "GEBEN SIE EINE ZAHL EIN (1 BIS 10)"
20 INPUT A
30 IF A<3 THEN GOTO 100
100 LET I=0
110 LET I=I+1
120 IF I<5 OR A>2 THEN GOTO 150
130 GOTO 110
140 LET I=I+A
150 IF I<10 THEN GOTO 100
160 GOTO 10

```

Wissen Sie, was dieses Programm macht? Ich nicht. (Und ich habe es geschrieben!) Und das Programm ist noch nicht einmal lang. Ergo: Solange man keine besseren Befehle wie GOSUB und Schleifen hat, ist GOTO für kurze Programme OK. Aber der C16 hat GOSUB und Schleifen und daher kann man bis auf ein paar wenige Ausnahmen darauf verzichten. Das sollte kein Dogma sein: Wenn Ihnen partout nicht einfällt, wie Sie ein Problem ohne die einmalige Verwendung eines GOTO's lösen können, dann nehmen Sie das GOTO. Drei GOTO's in einem Programm sind keine Katastrophe. Aber zehn können es schon sein.

Es gibt eine Situation, die GOTO's erforderlich zu machen scheint. Wenn man zwei Programmteile hat, bei denen in Abhängigkeit einer IF-Anweisung entweder der eine oder der andere ausgeführt werden soll. Schönes Beispiel sind die Zeilen 2220 und 2230. Falls die Bedingung zutrifft, werden die beiden durch Doppelpunkt getrennten Befehle hinter THEN ausgeführt, sonst nicht. Was ist, wenn es nun nicht nur zwei kurze Befehle sind, sondern zehn? Dann ist es wohl schlecht, sie in einer Riesenzeile alle per Doppelpunkt hintereinanderzuhängen. Bisher sind wir dann immer hinter dem IF mit GOTO in einen separaten Programmteil gesprungen und am Ende wieder zurück. Nun, das machen wir dann immer noch so, aber nun machen wir es nicht mehr mit GOTO, sondern viel übersichtlicher mit einem eigenen kleinen Unterprogramm und GOSUB.

Noch eine Bemerkung zur Maximumsuche Zeile 2200 bis 2250. Hier stehen zwei IF-Anweisungen hintereinander, die hinter dem THEN identische Anweisungsblöcke haben. Warum haben wir die beiden nicht zu einer zusammengefasst? IF I=0 OR XG(I)>XGMAX THEN ...? Das wäre ein richtiger Bug gewesen. Sie wissen nicht, was ein Bug ist? Dann lesen Sie gleich unten weiter. Jedenfalls hätte es nicht funktioniert und wir hätten den Fehler nicht so schnell gefunden. Das Problem ist, dass wir durch die erste IF-Anweisung XGMAX einen Anfangswert zuweisen. Wenn wir beide IF-Anweisungen zusammenfassen, hat XGMAX zum Zeitpunkt, zu dem der Computer die IF-Bedingung auswertet, gar keinen Anfangswert. Keinen definierten jedenfalls. Wir kennen ihn nicht. Und das kann dann ganz schön in die Hose gehen.

Bugs

Was sind Bugs? Das ist ein englisches Wort und heisst "Wanze". Diese kleinen Tierchen, die man ev. als Bettgenossen in einem nicht allzu reinlichen Hotel hat. Ich kann die Geschichte nicht mehr genau wiedergeben, aber so ungefähr. Die ersten Computer der Geschichte in den Jahren 1945 bis 1955, hatten als elektrische Einheiten zum Speichern von Zahlen (die Programme wurden auf Lochstreifen gestanzt) s.g. elektromechanische Relais. Das waren mechanische Schalter, die elektrisch umgeschaltet werden konnten. Vorausgesetzt, nichts Störendes kam in die Schaltwippe. Die Wanzen, die damals hier und da durch die riesigen Anlagen krabbelten, wussten davon aber nichts, krochen dazwischen, wurden vom Relais erschlagen, das Relais hängte und der Millionen-Dollar-Computer ging wegen der kleinen Wanze nicht mehr. Dann mussten die Techniker solange suchen, bis sie die Wanze in einem der Tausenden von Relais gefunden hatten.

Ich glaube, es war John v. Neumann, der eine recht smarte Mathematikerin eingestellt hatte, die ihm beim Betrieb einer dieser ersten Ungetüme, ENIAC oder EDSAC, half. Als er einmal zu ihr kam und sie mitten in der Maschine herumkroch und er fragte, was sie da tue, antwortete sie trocken: "I'm debugging the machine." (Ich entwanze die Anlage). Damit hatte sie ein neues Wort geboren. "Debugging". So, wie sich mit der Zeit das Wort "Bug" für Fehler im Computer und dann auch in den Programmen (in der "Software") verbreitete, so verbreitete sich für die Tätigkeit der Fehlersuche das Wort "Debugging". Jedesmal, wenn wir also in unserem Programm einen Bug suchen, (und das kann genauso mühsam sein, wie in Tausenden von Relais die geschichtsträchtige Wanze zu suchen), "debuggen" wir. Es gibt bei grösseren Programmiersystemen wie dem C16 sogar extra Funktionalitäten und Programme, um diese Suche zu erleichtern. Das sind dann "Debugger".

Man kann ja Arrays mit Tabellen vergleichen. Mit Tabellen, die mehrere Zeilen, aber nur eine Spalte haben. Jedes Element ist eine Zelle der Tabelle. Gibt es auch mehrspaltige Tabellen? Ja, die kann der C16 auch. `DIM X(10,5)` wäre eine solche mehrspaltige Tabelle mit 10 Zeilen und 5 Spalten. Der Zugriff ist ganz analog zum eindimensionalen Array. Es gibt sogar dreidimensionale Arrays, was dreidimensionalen Tabellen entsprechen würde: `DIM X(10,5,8)` z.B.. Wieviel Dimensionen der C16 beherrscht, weiss ich gar nicht. Aber es kommt selten vor in der Programmierung, dass man mehr als drei Dimensionen braucht.

Ein Anwendungsbeispiel bringe ich hier nun nicht, da im nächsten Kapitel ein ausführliches Programm unter Verwendung eines zweidimensionalen Arrays kommt.

Die Millionärsshow und der DATA-Befehl

Als ich mich ransetzte, um diese Einführung zu schreiben, da trieb mich nicht so sehr die Freude, zu zeigen, was es für komplizierte Programmieretechniken gibt. Vielmehr will ich zeigen, wie man mit Hilfe von Programmierung "denken" kann, Probleme analysieren und lösen und vielleicht auch mit ihnen spielen kann. Was man mit Programmierung machen kann, ist mir wichtig, nicht die Tätigkeit selbst.

Systematisches Jokern

Deshalb will ich hier in diesem Kapitel mal eine etwas ungewöhnliche Überlegung und ihre Bearbeitung einschieben. Ich nehme an, viele von Ihnen kennen/kannten die Show "Wer wird Millionär?" von Günter Jauch. Ein Trivial Pursuit für's Fernsehen. Ein Kandidat beantwortet Fragen von Jauch, indem er aus vier Antworten auswählt. Wählt er falsch, hat er verloren. Wählt er richtig, kommt die nächste Frage und sein möglicher Gewinn verdoppelt sich. Wenn er falsch rät, darf er den schon erreichten Gewinn mit nach Hause nehmen. Bei 50 Euro fängt's an und geht bis 1 Million Euro.

Es gibt während dem Spiel drei Joker. Einmal darf der Kandidat das Publikum befragen, einmal darf er sich 2 von 4 Antwortalternativen streichen lassen und einmal darf er jemand zuhause anrufen, den er kennt.

Wenn Sie die Show kennen und schon selbst angesehen haben, dann werden Sie sich sicher auch schon vorgestellt haben, dass Sie selbst im Kandidatenstuhl sitzen. Und jeder wird sich so seine Strategie zurechtgelegt haben, wie er mit den Jokern umgeht. Der eine sagt: "Sobald ich auch nur ein bisschen unsicher bin, verwende ich einen Joker. Schliesslich geht es von Anfang an um nicht wenig Geld und vor allem: Wenn ich raus bin, nutzen mir die übrigen Joker eh nichts mehr." Der andere sagt: "Ne, Blödsinn. Am Anfang sind die einfachen Fragen. Wenn ich da schon die Joker einsetze, bin ich bei den schwierigen Fragen verloren und erreiche nie die Million. Ich muss am Anfang eine kleine Unsicherheit riskieren, damit ich später überhaupt eine Chance habe, die die schwierigen Fragen zu beantworten!"

Was ist richtig davon? Sie meinen, dass man das nicht objektiv beantworten könnte? Na, schauen wir mal, was mit dem C16 und ein bisschen Nachdenken denn so geht.

Was uns interessiert, sind zwei Fragen:

- Nach welcher Strategie maximiert man seinen zu erwartenden Gewinn?
- Nach welcher Strategie maximiert man die Wahrscheinlichkeit, 1 Million zu holen?

Bei jeder Frage gibt es eine Wahrscheinlichkeit, dass der Kandidat die Frage richtig beantwortet. Die gross die Wahrscheinlichkeit bei jeder Frage sein soll, können wir in die folgende Tabelle eintragen.

Frage	Gewinn	Wahrscheinlichkeit
1	50	100
2	100	95
3	200	90
4	500	90
5	1000	85
6	2000	85

7	4000	80
8	8000	70
9	16000	60
10	32000	50
11	64000	40
12	125000	30
13	250000	25
14	500000	25
15	1000000	25

Ein Joker soll die Wahrscheinlichkeit (einfachheitshalber) auf 100% setzen.

Nun muss der Computer ran. Es soll einfach spielen. Eine Frage besteht darin, dass gemäss der Wahrscheinlichkeit der Fragennummer gewürfelt wird. Erinnern Sie sich an die RND()-Funktion! Und zwar nur zwischen 0 und 1. Kommt 0 heraus, ist das Spiel vorbei und der bisher erreichte Gewinn wird notiert. Kommt 1 heraus, kommt die nächste Frage.

Am Anfang jeder Runde werden drei Fragen (durch den Benutzer/zufällig/systematisch in einer Schleife: Wie Sie wollen) ausgewählt, bei denen der Joker eingesetzt wird. Z.B. Frage 2,5 und 7. Bei diesen wird die Wahrscheinlichkeit auf 100% gesetzt, d.h. die Frage wird sicher beantwortet (Sie müssen in diesem Fall die RND()-Funktion gar nicht mehr bemühen.)

Sie sollten jede Jokersetzung, also jedes Jokerfragentripel, nicht nur einmal durchspielen, sondern mindestens 20mal, besser 100mal oder mehr. Denn das Ergebnis hängt ja auch vom Zufall ab. Wenn man aber oft spielt, dann hängt der Anteil, mit dem die Million erreicht wird und der mittlere Gewinn in den Spielen nicht mehr so vom Zufall ab.

Der DATA-Befehl

Wir haben noch ein kleines Problem: Wie bekommen wir die Tabellenwerte ins Programm? Nun, mit Arrays ist das an sich kein Problem, aber nun 30 Zeilen mit LET X(0,0)=50, LET X(0,1)=100, LET X(1,0)=100,... zu schreiben, ist auch irgendwie blöd. Es gibt eine kleine Vereinfachung, den DATA/READ-Befehl.

Dazu schreibt man in eine Zeile hinter den DATA-Befehl (am besten ganz ans Programmende) die Zahlenwerte in der Reihenfolge, in der sie nachher in das Array geschrieben werden sollen. So kann man viele Zahlen sozusagen mitten in den Programmtext schreiben:

```
10 DIM X(I)
15 RESTORE 50
20 FOR I=0 TO 9
30 READ X(I)
40 NEXT I
50 DATA -19,31.5,22.3,34,-4.5,9.3,-298,877,-29,76.39
```

Im Beispielprogrammchen stehen die Zahlen in der DATA-Zeile 50. Wir sehen auch schon, wie wir dann an diese Zahlen rankommen. Zunächst teilen wir dem Computer mit, wo die Zahlen stehen, die er abholen soll: RESTORE 50. Das heisst: "Hole sie in Zeile 50". Da steht unser DATA-Befehl mit Zahlenliste. Mit jedem READ holen wir uns einen Wert. Beim ersten READ die -19, dann die 31.5 usw. Im Beispiel haben wir gleich ein Array benutzt, um es auf diese Art und Weise mit der hinter DATA angegebenen Liste bequem zu füllen.

So. Jetzt sind Sie dran! Machen Sie sich an Ihre Übungsaufgabe! Schreiben Sie ein Programm, dass Ihnen den mittleren Gewinn und die Millionengewinnwahrscheinlichkeit für eine bestimmte Jokersetzung ausgibt! Und stellen Sie damit fest, welche Jokerstrategie am effektivsten ist!

Zu schwierig?

Zugegeben: Obwohl das eigentliche Programm nicht kompliziert ist, wird es nicht einfach sein, zu verstehen, was der Knirch in diesem Kapitel eigentlich meint. Wie dieses "Frage und Antwort"-Geschehen in ein Programm umgemünzt werden soll. Und wie da auch noch der Joker irgendwo auftauchen soll. Deshalb hier noch ein paar Hilfestellungen, u.a. der Programmteil, der das eigentliche Spiel darstellt. Wenn Sie dann immer noch nicht zurechtkommen - keine Panik, schauen Sie sich einfach im nächsten Kapitel die fertige Lösung an. Sie sind auch schon sehr gut, wenn Sie die nachvollziehen und verstehen können!

Also: Was passiert, wenn Günter Jauch einem Kandidaten in der Sendung eine Frage stellt? Der Kandidat wählt aus vier Antworten eine aus. Tut dabei sein Bestes, die richtige Antwort zu finden. Klappt natürlich nicht immer. Manchmal wählen die Kandidaten ihre Antwort sogar auf gut Glück. Wenn der Kandidat nicht viel Ahnung hat, dann wählt er sogar immer auf gut Glück, weil er ja nichts weiss, was ihn der richtigen Antwort näher bringen würde als allen anderen. Das Ganze ist also eine Art Lotteriespiel. Wenn der Kandidat eine gute (Jauch-)Bildung hat, dann ist die Wahrscheinlichkeit, dass er die richtige Antwort zieht, mehr als ein Viertel, mehr als 25%. Wenn er es "hundertprozentig sicher" weiss (und damit recht hat), ist sie sogar 100%. Das heisst, wir können jedes Spiel darstellen als eine Lottoziehung mit dem Ausgang "Richtig" und "Falsch" und einer Wahrscheinlichkeit p für "Richtig", die zwischen 25% und 100% schwankt - je nachdem, wie gut der Kandidat ist.

So, nun sind wir der Sache schon näher. Wenn nun ein Joker ins Spiel kommt, gehen wir davon aus, dass der Kandidat nach Anwendung des Jokers die Antwort sicher weiss: p geht auf 100%. (Das stimmt nicht ganz. Aber für unsere vereinfachte Betrachtung hier reicht diese Annahme vollkommen aus.)

Wie sieht nun eine Frage aus? Wir nehmen eine Wahrscheinlichkeit p . p nehmen wir aus der Tabelle von oben. Wenn der Kandidat einen Joker anwendet, ist die Frage gelaufen und das Ergebnis der Frage ist "Richtig". Wenn nicht, dann lassen wir den Computer würfeln. Zunächst eine gleichverteilte Zufallszahl zwischen null und eins: $LET X=RND(1)$. Wenn X dann kleiner ist als p , dann ist die Antwort "Richtig". Ansonsten "Falsch".

Ein ganzes Spiel besteht darin, von Frage 1 ab Fragen zu spielen. Jedesmal mit dem p , das in der obigen Tabelle der entsprechenden Frage zugeordnet sind. Da die Fragen immer schwieriger werden, wird das p immer kleiner. Es kann aber natürlich nicht unter 25% sinken. Wenn eine Antwort "Falsch" lautet, ist das Spiel aus. Ergebnis ist der bisher erreichte Gewinn. Wenn alle Fragen "Richtig" ergeben, ist der letzte Gewinn eine Million.

Ziel ist es nun, den Computern viele solche Spiele spielen zu lassen und den mittleren Gewinn zu ermitteln und die Zahl der Spiele zu zählen, die 1 Million ergeben haben.

Jetzt klarer? Wenn immer noch nicht, dann schauen Sie sich einfach die Zeilen 2000 bis 2300 im Listing des folgenden Kapitels an. In 2200 bis 2260 wird eine Frage durchgespielt, in 2000 bis 2150 ein Spiel.

Ein Lösungsvorschlag


```

1 REM WER WIRD MILLIONAER?
2 REM =====
3 REM ANALYSE VON JOKERSTRATEGIEN
4 REM BY C. SCHATZ, 2002
10 DIM P0(16,2)
20 DIM JOK(3)
29 REM ANZAHL SPIELE PRO JOKERS:
30 LET N=30
40 LET MODUS=2:REM 1=MANUELL,2=AUTOMATIK
70 GOSUB 8900
77 IF MODUS=2 THEN GOTO 500
80 LET AS="J"
90 DO WHILE AS="J"
100 GOSUB 200:REM JOKER SETZEN
110 REM GOSUB 2000
120 GOSUB 1000:REM N SPIELE SPIELEN
130 GOSUB 1500:REM SHOWRESULT
140 LOOP
199 END
200 REM JOKERSETZEN
210 SCNCLR
220 COLOR 0,1,0
230 COLOR 4,1,0
240 COLOR 1,7,4
250 CHAR ,5,2,"BEI WELCHEN FRAGEN SOLLEN
260 CHAR ,5,3,"DIE JOKER EINGESETZT WERD
EN?"
270 CHAR ,10,5,"ERSTER JOKER?"
280 CHAR ,15,6,"":INPUT F
290 IF F<1 OR F>15 THEN GOTO 280
300 LET JOK(0)=INT(F)
310 CHAR ,10,9,"ZWEITER JOKER?"
320 CHAR ,15,10,"":INPUT F
330 IF F<1 OR F>15 THEN GOTO 320
340 LET JOK(1)=INT(F)
350 CHAR ,10,12,"DRITTER JOKER?"
360 CHAR ,15,13,"":INPUT F
370 IF F<1 OR F>15 THEN GOTO 350
380 LET JOK(2)=INT(F)
390 SCNCLR
400 REM AUTOMATIK-MODUS
410 SCNCLR
420 PRINT "AUTOMATIK-MODUS"
430 PRINT "WIEVIEL JOKERSETS SOLLEN GETE
TET WERDEN?"
440 INPUT NJOK
450 SCNCLR
460 LET JOKSUM=0
470 FOR K=0 TO NJOK-1
480 GOSUB 1700:REM JOKER ZUFALLSETZEN
490 GOSUB 1000:REM N SPIELE SPIELEN
500 LET SUM=0
510 FOR I=0 TO 2
520 LET SUM=SUM+JOK(I)
530 NEXT I
540 LET MEANJ=SUM/3
550 REM ERGEBNISSE LISTEN
560 PRINT JOK(0);TAB(5);JOK(1);TAB(10);J
OK(2);
5672 PRINT TAB(15);MEANJ;TAB(25);GEWME;TA
B(32);PMILL
5680 NEXT K
5690 END
5699 :
1000 REM VIELE SPIELE
1001 REM
1002 REM ZURÜCK '-' GEWME=MITTLERER GEWIN
N
1003 REM PMILLION=WAHRSCHEINLICHKEIT, EINE MILLION
ZU GEWINNEN
1010 LET GSUM=0
1020 LET NSUM=0
1030 FOR I=0 TO N-1
1040 GOSUB 2000:REM EIN SPIEL
1045 IF MODUS=1 THEN PRINT I,FRAGE,GEWIN
N
1050 LET GSUM=GSUM+GEWINN
1060 IF GEWINN>500000 THEN LET NSUM=NSUM
+1
1070 NEXT I
1080 LET GEWME=GSUM/N
1090 LET PMILL=NSUM/N
1100 RETURN
1500 REM SHOWRESULT
1510 SCNCLR
1520 COLOR 0,7,4
1530 COLOR 1,1,0
1540 CHAR ,5,3,"ERGEBNIS"
1550 COLOR 1,3,2
1560 CHAR ,10,5,"MITTLERER GEWINN:"
1570 CHAR ,10,6,STR$(GEWME)
1580 CHAR ,10,8,"WAHRSCHEINLICHKEIT,"
1590 CHAR ,10,9,"EINE MILLION ZU GEWINNE
N"
1600 CHAR ,10,10,STR$(PMILL)
1610 COLOR 1,4,2
1620 CHAR ,10,20,"NOCH EIN JOKERSET? (J/
N)"
1630 CHAR ,5,21,"":INPUT AS
1640 SCNCLR
1650 RETURN
1699 :
1700 REM JOKER ZUFALLSETZEN
1710 FOR I=0 TO 2
1720 LET JOK(I)=INT(RND(1)*15+1)
1730 NEXT I
1740 RETURN
2000 REM EIN SPIEL
2010 REM RUECKGABE GEWINN
2020 LET FRAGE=1
2030 LET GEWINN=1
2040 DO WHILE (FRAGE<=15) AND (GEWINN=1)
2050 LET GEWINN=0
2060 FOR J=0 TO 2
2070 IF JOK(J)=FRAGE THEN GEWINN=1
2075 NEXT J
2080 IF GEWINN=0 THEN GOSUB 2200
2085 REM ? FRAGE, GEWINN
2090 LET FRAGE=FRAGE+1
2100 LOOP

```

Aufbau des Programms:

2200 bis 2300	Durchführung einer Antwort (Frage)
2000 bis 2200	Ein Spiel
1000 bis 1500	Viele Spiele
200 bis 400	Benutzerdialog: Joker setzen
1500 bis 1650	Benutzerdialog: Ergebnisse anzeigen
500 bis 1000	Spezialmodus
0 bis 50	Deklarationen und Setzen der Systemgrößen
50 bis 200	Hauptprogramm

Im Hauptprogramm wird zunächst die Modus-Variable abgefragt, und es wird ggf. in einen speziellen

Programmmodus gesprungen. Ansonsten wird eine Schleife betreten, die der Benutzer am Ende der Programmbenutzung mit "N" wieder verlassen kann. Nun werden um Unterprogramm "Joker setzen" die Joker vom Benutzer auf die Fragen gesetzt, in denen sie angewandt werden sollen. Anschliessend "N Spiele spielen" aufgerufen. Das wiederum ruft N-mal "Spiel spielen" auf. Soweit der grobe Aufbau.

Der Spezialmodus wird bei den Ergebnissen im nächsten Abschnitt erläutert.

Die Ergebnisse

Haben Sie ein paar Jokersetzen ausprobiert? Sie werden feststellen, dass der Rechner für jedes Spiel ca. eine halbe Sekunde braucht (etwas schneller als in der echten Show also...). Wenn wir zur Sicherheit 1000 Spiele spielen lassen (N=1000), um einen genauen Mittelwert für Gewinn und Millionenwahrscheinlichkeit zu ermitteln, sind das 500 Sekunden, als ca. 8 Minuten pro Jokersetzung. Es gibt bei 3 Jokern und 15 Fragen eine Menge verschiedene Setzungen (dreitausend und noch was). Die können wir so also nicht alle durchprobieren.

Nun, man kann N auch kleiner wählen. N=100 z.B. oder N=30 oder N=10. Wieviel man mindestens braucht, können Sie dadurch testen, dass Sie diesselben N Spiele zweimal hintereinander durchspielen lassen und die Mittelwerte vergleichen. Weichen Sie zu stark voneinander ab, haben Sie N zu klein gewählt. Ich habe auf diese Weise herausgefunden, dass N=100 schon sein sollte.

Ich bin dann auf die Idee gekommen, den Computer die Joker einfach zufällig wählen zu lassen. Das ist der Spezialmodus. Sie brauchen sich das aber nicht weiters anzuschauen - das funktioniert nicht. Man kann am Ergebnis nichts erkennen, die Mittelwerte scheinen ziemlich willkürlich hin und her zu schwanken.

Danach bin ich auf den Gedanken gekommen, im manuellen Modus die Joker einfach im Block zu setzen, zuerst ganz am Anfang und dann immer weiter nach oben zur Frage 15 zu schieben. Das ergab etwas Sinnvolles für die Gewinne. Die Millionenwahrscheinlichkeiten waren immer null - die Million ist schon sehr unwahrscheinlich. Wir müssten sehr, sehr viele Spiele spielen, um diese Unwahrscheinlichkeiten so genau bestimmen zu können, damit man sie vergleichen kann. Daher habe ich Zeile 1060 geändert. Ich habe den Computer zählen lassen, wenn ein Gewinn grösser als 32000 ist. Da sind die Wahrscheinlichkeiten grösser. Schliesslich hat mich interessiert, was passiert, wenn man einen Joker aus dem Block rausnimmt. Daher habe ich dann noch Kombinationen wie "1 Joker am Anfang, 2 in der Mitte" oder "2 am Anfang, einer in der Mitte" getestet. Das Ergebnis sehen Sie unten.

Joker1	Joker2	Joker3	Gewinn	p
1	2	3	4648	0.02
4	5	6	7653	0.01
7	8	9	11016	0.05
10	11	12	10433	0.06
13	14	15	4543	0
3	8	9	10206	0.04
3	4	9	5981	0

Das Ergebnis ist eindeutig: Am Besten für Gewinn und 64000-Wahrscheinlichkeit ist es, wenn die Joker in der Mitte benutzt werden, zwischen Frage 7 und 12. Die vorgezogene oder nachgelagerte Benutzung eines Jokers bringt nichts.

Man sieht also, dass wir mit unserem kleinen Programm eine nette Frage beantworten konnten, bei der uns anfangs gar nicht eingefallen wäre, dass man sie überhaupt so klar beantworten kann.

Im Beispielprogramm kommen ein paar Kommentarzeilen vor, z.B. Zeile 110, die eigentlich gar keine Kommentarzeilen sind, sondern Befehlszeilen mit einem REM davor. Hier sind ehemalige Befehlszeilen durch Einfügen eines REM's deaktiviert worden. Man kann sie bei Bedarf so wieder schnell aktivieren und Sie können anhand dieser Zeilen sehen, wie die Programmentwicklung ursprünglich lief.

Ich habe zuerst die DATA-Zeilen und die Einleseroutine 8900 bis 9100 geschrieben und getestet. Dann bin ich vom Kern zur Schale vorgegangen, habe also zuerst mit der kleinsten Einheit, der einzelnen Frage in Zeile 2200 angefangen. Man sieht das noch an dem auskommentierten PRINT-Befehl in Zeile 2240. Hier habe ich mir zunächst beim Testen der Routine für jede einzelne Frage p und x, Wahrscheinlichkeit und Würfelergbnis ausgeben lassen. Als das funktionierte, konnte ich die Zeile auskommentieren.

Als Nächstes habe ich die Routine für ein Spiel aufgesetzt. In Zeile 2085 habe ich mir die das Ergebnis jeder Frage ausgeben lassen, um zu kontrollieren, wie das einzelne Spiel verläuft, welche Fragen "der Kandidat" noch richtig beantwortet. So konnte ich beurteilen, ob die letzte erreichte Frage auch richtig ermittelt wird.

Als auch die Spielroutine richtig funktionierte, ging's zum "Turnier", zu der Routine, die N Spiele durchspielt. Hier wird ohnehin in Zeile 1045 das Ergebnis jedes Spiels ausgegeben.

Was lernen wir also draus? Man muss ein Programm nicht von vorne nach hinten schreiben, sondern kann es ein bisschen chaotisch angehen: Man löst zuerst die Teilaufgäbchen, die einem leicht und beherrschbar erscheinen, in kleinen Routinen. Dann bastelt man diese Routinen zu anderen Routinen zusammen, die schon grössere Aufgaben lösen. Und so weiter. Beim Test der Routinen sollte man mit PRINT-Anweisungen (und ggf. auch Input's dahinter, um die Ausgabe auch lesen zu können), nicht sparen. Und man sollte die PRINT's nicht einfach löschen, wenn man sie nicht mehr braucht, sondern auskommentieren. Es kann ja sein, dass man sie später testweise nochmal braucht.

Bits, Bytes und BASIC

BASIC

Wir haben nun eine ganze Menge Befehle und Funktionen des C16 für die Programmierung kennengelernt. So eine Menge an Befehlen und Funktionen und die Regeln, wie daraus ein Programm wird, nennt man eine *Programmiersprache*. Wir haben also gerade unsere erste Programmiersprache erlernt. Die C16-Programmiersprache? Nicht nur. In der Zeit, in der es den C16 gab, war die eingebaute Programmiersprache praktisch immer diesselbe, bis auf kleinere Abweichungen im Bereich der Ein- und Ausgabebefehle, die ja stark von der Technik des Computers abhängen. Diese Programmiersprache war BASIC (Beginners All Symbolic Instruction Code). Sie haben also gerade BASIC gelernt. BASIC ist heute noch eine sehr beliebte Programmiersprache auf Computern.

Bits und Bytes

Wir haben ganz am Anfang gelernt, wie man Zahlen speichert. Später, dass man auch Programme in den Speicher schreiben kann. Jetzt gehen wir etwas in die Tiefe und fragen: Wie funktioniert das eigentlich, dass so unterschiedliche Dinge gespeichert werden können?

Die elektronischen Speicher eines Computers sind s.g. bistabile Transistoren oder Flipflops. Sie müssen sich drunter irgendetwas Elektronisches vorstellen. Diese "Dinger" können eine 0 oder eine 1 speichern. Man nennt das 1 *Bit*. Zur Speicherung von Zahlen oder Buchstaben ist das natürlich zu wenig. Also bündelt man die Bits zu Bit-Gruppen, s.g. "Worten". In der Frühzeit der Computer war die Anzahl der Bits pro Wort von Maschine zu Maschine verschieden. Je nachdem, was der Computerbauer hauptsächlich zu Speicherung gedacht hatte: Ganzzahlen, Gleitkommazahlen oder Buchstaben. Von 12 Bits pro Wort bis 50 Bits pro Wort kam alles vor. Anfang der 60er-Jahre führte IBM ein grosses Computersystem ein, das System /360. Und da dieses System sehr universell gedacht war, wurden hier die Bits zu Standardworten gebündelt. Diese nannte man und nennt man heute noch *Bytes*. Jedes Byte hat 8 Bit. Warum acht? Ein Byte kann damit 256 verschiedene Zustände annehmen, also eine Ganzzahl zwischen 0 und 255 speichern. Damit können Bytes schon mal ganz gut dazu genutzt werden, Ziffern und Buchstaben zu speichern - mittels ASCII-Codierung, die wir schon kennengelernt haben. Womit das Geheimnis gelüftet wäre, wie ein Computer ein Programm speichert. Schliesslich können mehrere Bytes wiederum gebündelt werden, z.B. 4. Das sind dann 32 Bits. Diese können eine Ganzzahl zwischen -2 Milliarden und +2 Milliarden aufnehmen. (Oder zwischen 0 und 4 Milliarden). Das reicht auch für eine Zahl. Bei Gleitkommazahlen ist es etwas komplizierter, hier werden 4, 5 oder sogar 8 Bytes dazu benutzt, die Zahl in s.g. Exponentialdarstellung zu speichern, bei der Mantisse und Exponent unterzubringen sind.

Kilo, Mega und jenseits

Mein allererster programmierbarer Taschenrechner hatte 38 Bytes Programmspeicher. Das war übersichtlich. Aber damit kann man natürlich nicht viel anfangen. Daher spricht man oft über Tausende oder Millionen von Bytes. 1024 Bytes sind 1 Kilobyte, abgekürzt 1 K. 1024, richtig, nicht 1000. Denn 1024 sind 2^{10} Bytes. Da der Computer alles binär (im 0/1-System) darstellt, ist das praktischer: Damit 1024 Bytes angesprochen werden können, braucht man eine Nummer mit genau 10 Bits.

1024 K sind 1 Megabyte, abgekürzt 1 MB. Also 2^{20} Bytes.

1024 MB sind 1 Gigabyte, abgekürzt 1 GB. also 2^{30} Bytes.

1024 GB sind 1 Terabyte, abgekürzt 1 TB. Also 2^{40} Bytes.

1024 TB sind 1 Petabyte, abgekürzt 1 PB. Also 2^{50} Bytes.

Grössere Einheiten werden Sie wahrscheinlich nicht brauchen - jedenfalls nicht in den nächsten 2 Jahren;-).

Jetzt können Sie die Meldung, die beim Starten des C16 kommt, verstehen: "12277 Bytes free". Der C16 hat nämlich 16 K Hauptspeicher. 16385 Bytes. Davon braucht er für das BASIC-System und den Bildschirm (siehe Kapitel "Grafik") etwas. Und es bleiben dann ca. 12K für den Programmierer übrig. Wieviel ist das? Nun, da diese Bytes sowohl für Daten als auch für das Programm erhalten muss und da Daten (Strings oder Zahlen) unterschiedlich viele Bytes brauchen und jede Programmzeile wieder unterschiedlich lang ist, kann man das schwer griffig sagen. Was die Programmlänge betrifft, kann man die Faustformel bilden: Ca. 20 Bytes pro Zeile. Bei 1K Zahlenspeicher (ca. 250 Zahlen) bleiben dann noch ca. 11000

Bytes und das sind dann ca. 550 Zeilen. Soviel haben wir bisher nicht gebraucht. Aber das kann schnell kommen, wenn man an einem Programm längere Zeit arbeitet, es pflegt und erweitert.

Wie kann ich beim C16 herausbekommen, wieviel Bytes ich schon verbraucht habe? Nun, geben Sie einmal folgendes Programm ein und starten es:

```
10 DIM A(4000)
```

Das ist alles. Anscheinend passiert hier noch gar nichts. Tatsächlich versucht Ihr PC in Zeile 10 Speicher für 4000 Zahlen zu reservieren. Jede Zahl braucht 4 Bytes. Er benötigt also 16000 Bytes. Das C16-System hat aber nur 12277 Byte frei. Folglich klappt das nicht:

```
?Out of memory error in 10
```

Diese Fehlermeldung werden Sie immer bekommen, wenn Sie entweder zuviel Speicher verbrauchen wollen oder ein zu langes Programm eingetippt haben (oder beides).

Natürlich können Sie sich auch schon vorher bei Gelegenheit informieren, wieviel Speicher noch frei ist. Das geht über die Funktion **fre(0)**. (Man kann auch noch andere Argumente als die Null übergeben, um weitere Speicherdetails zu erfahren, aber das soll uns hier nicht interessieren.). Laden wir z.B. unser Millionärsprogramm aus dem letzten Kapitel und geben **FRE(0)** ein, erhalten wir "9706". Das heisst, wir haben noch 9706 Bytes Speicher frei. Unser Programm ist also "? 12277 - FRE(0)", 2578 Bytes lang, rund 2,5K.

Wenn wir nun das Programm laufen lassen und es mitten im Betrieb mit der Tab-Taste unterbrechen, können wir das gleiche Kommando nochmal abschieken. Dieses Mal sind alle Speicher, die das Programm benötigt, schon reserviert und belegt. Nun müsste mehr Speicher verbraucht sein. Tatsächlich: Nun meldet der C16: "2958". Das heisst, es sind nun 2958 Bytes verbraucht. 380 Bytes werden durch Variablenspeicher belegt.

Speicher und ihre Grösse

3K Speicher, den wir für unser Millionärsprogramm brauchten: Ist das viel oder ist das wenig? Es ist nicht unwichtig als Programmierer, ein bisschen ein Gefühl für Speicherbedarf und Speichergrössen zu entwickeln. Theoretisch wird das immer weniger wichtig, weil die Computerspeicher immer grösser werden. Aber an irgendwelchen Stellen ist immer ein Flaschenhals, der es nahelegt, einen Blick auf den Speicherbedarf zu werfen. Sei es die zur Verfügung stehende Modem-Übertragungsrate, sei es, dass der Computer, für den man gerade programmiert, ein Mobiltelefon ist, sei es, dass man ein Programm geschrieben hat, dass, je länger es läuft, je mehr Speicher braucht.

Aus der Sicht des Taschenrechners, mit der wir unser Tutorial gestartet haben, sind 3K viel. Wir sehen ja, dass wir rund 200 Programmzeilen darin untergebracht haben und dazu noch rund 100 Zahlenvariablen. Aus der Sicht heute üblicher Hauptspeichergrössen in PC's - 256 MB sind eher wenig - sind 3K ein Brösel, den man dort kaum noch wiederfinden wird. Deshalb ist es vielleicht nicht ganz unwichtig, sich einmal eine Übersicht über Speichergrössen zu verschaffen.

Hauptspeichergrössen

Die folgende Tabelle ist auch als nette Übersicht für etwas fortgeschrittenere Programmierer gedacht, daher tauchen einige Fachbegriffe auf, die wir noch nicht kennengelernt haben, wie z.B. "Stackspeicher".

Art des Speichers	Grösse	Jahre, in denen er aktuell war
Univac, erster in Serie gebauter Grossrechner, Kosten in heutigen Preisen ca. 2 Mio. EUR	8 K	1952-1960
PDP8, beliebter Minicomputer, Kosten 100.000 US\$	4K	1965 bis 1975
Commodore C16, Kosten 150 US\$	16 K	1985
Professioneller Büro-Computer (CP/M oder Apple)	64 K	1982
Erster DOS-PC	16 bis 64 K	1981
Maximaler Stackspeicher unter DOS und 16-Bit-Windows	64 K	1981 bis 1990
Maximaler konventioneller Hauptspeicher unter DOS	640 K	1981 bis 1990
PC	8 bis 32 MB	1995

Maximaler Hauptspeicher unter Windows 9x/ME	512 MB	1995 bis 2002
PC	128 bis 512 MB	2003
Hauptspeicherbedarf von Windows XP	min. 40 MB	ab 2002
Hauptspeicherbedarf von Linux	6 MB bis 100 MB (je nach Ausbau)	ab 1994
Maximaler konventioneller Hauptspeicher unter 32-Bit-Linux/Windows	4 GB	1995 bis ?
Maximaler ansteuerbarer Hauptspeicher eines Pentium 4-PC's	64 GB	2001 bis ?

Massenspeichergrößen

Massenspeicher sind die Datenträger, die die Programme und Daten permanent speichern, auch in der Zeit, in der der Computer ausgeschaltet ist, wie z.B. eine Diskette (nicht mehr ganz aktuell) oder eine DVD, aber auch USB-Sticks u.a.

Art des Speichers	Grösse	Jahre, in denen er aktuell war
Lochkarte	640 Bytes	1940 bis 1985
Audiocassette C60	80 K	1965 bis 1985
Floppy Disk	1,44 MB	1985 bis 2003
Zip-Disk	100 MB bis 250 MB	1992 bis 2001
CD-ROM/CD-RW	700 MB	ab 1994
DVD	4,7 GB	ab 2000
DVD2 (blauer Laser)	27 GB (?)	ab 2006
Festplatte	10 MB	1982
Festplatte	100 MB	1990
Festplatte	1 GB	1995
Festplatte	10 GB	2000
Festplatte	100 GB	2003
Festplatte	1 TB (?)	2006

Hoch- und Niedrigsprachen

Es ist nicht notwendig für einen Programmierer, sehr viel über die innere Funktion eines Computers zu wissen, aber die wesentlichsten Grundsätze zu kennen, ist sehr wichtig. Dazu gehört es, ein *Executable* von einem *Quellcode* unterscheiden zu können.

Maschinensprache

Es ist nicht so, dass die Elektronik eines Computers in der Lage wäre, mit dem BASIC-Programm umzugehen, das wir ihr vorsetzen. Die Elektronik eines Computers ist eigentlich ziemlich einfach. Sie besteht im Wesentlichen aus zwei Bausteinen:

1. Dem Prozessor (CPU)
2. Dem Hauptspeicher

Nur beide zusammen machen einen Computer aus. Der Prozessor alleine ist eher so etwas wie die eigentliche Rechenmaschine im Computer. Es ist ein Chip, ein schwarzer Baustein, mit vielen Kontaktbeinchen. Je nachdem, welche Spannungen an diesen Beinchen anliegen, aktivieren wir an dieser CPU eine ihrer Funktionen. Die CPU hat auch einige eigene Speicher. Es sind allerdings wenige, höchstens ein Dutzend, und sie haben feste Namen, z.B. E1, E2 usw. oder a,b,c bis i. Man nennt sie "Register". (Das können Sie wieder vergessen.) Eine Funktion kann z.B. sein, den Inhalt eines Speicherbytes in ein Register zu verfrachten. "*ld E1,(8000)*" hiesse z.B., den Inhalt von Speicherzelle 8000 in das Register E1 zu laden. "*ld E1,(8000)*" ist dabei nur der Name, den wir dieser Funktion geben, die CPU kennt die Funktion nur unter einer bestimmten Bitkombination in Form von Spannungen an ihren Füßchen. "*add E1,E2*" wäre z.B. eine andere solche Funktion. Es ist eine Art Programmiersprache, aber es ist eine sehr einfach gebaute, bescheidene Programmiersprache, in

der es nicht einfach ist, ein "richtiges" Programm zu schreiben. Man nennt so eine Sprache *Maschinensprache*. Ein Programm in Maschinensprache nennt man ein *Executable*, weil es durch die CPU direkt ausführbar ist. Sie liegen als Dateien auf Ihrer Festplatte. Executables auf Windows-PCs enden mit .exe oder .com, auf Linux-PCs meist mit .bin (für "binary"). Bausteine von Executables auf Windows-PCs enden in der Regel mit der Endung ".dll".

Jede CPU "spricht" also ihre eigene Maschinensprache. Die Prozessoren aus der Familie der s.g. x86-Prozessoren (8088 bis Pentium 4) beherrschen alle 8088-Maschinensprache, selbst ein Pentium 4 aus dem Jahr 2002. Unter Windows 9x/NT/2000/XP sind allerdings alle Binaries in der originären Maschinensprache des 80386 geschrieben, so dass die 8088-Fähigkeit (s.g. "Real Mode") meist nicht mehr benötigt wird - ausser beim Starten des PC's. Der C16 hat als CPU eine 7502, ein Abkömmling des in den 80er-Jahren weit verbreiteten 6502. Dessen Maschinensprache hat gar nichts mit der der x86-CPU's zu tun. Ein Executable für x86-PC's läuft auch nicht auf einem Apple-Computer oder auf einer Sun-Workstation, da diese jeweils andere Prozessoren haben. Wenn eine CPU auf ein Programm in einer ihr fremden Maschinensprache trifft, haben wir eine *Inkompatibilität*: Executable und CPU sind nicht *kompatibel*, passen nicht zueinander.

Ab dem Jahr 2004 wird es neben Apple- und x86-PCs noch zwei weitere Prozessorfamilien geben: Itanium und Opteron (Athlon64). Beide beherrschen zwar die Maschinensprache des 80386 (das meint man, wenn man auch von "32-Bit-Modus" spricht), nicht mehr jedoch die des 8088. Und beide haben ihre eigene originäre (und unterschiedliche) Maschinensprache: Opteron-Binaries sind zum Itanium inkompatibel und umgekehrt.

Interpreter und Compiler

Wie kommt es aber, dass wir den C16 in BASIC programmieren können? Nun, einfach dadurch, dass im C16 ein bestimmtes Executable eingebaut ist. Dieses Maschinenprogramm geht den eingetippten BASIC-Text Zeile für Zeile durch und *übersetzt* ihn in 7502-Maschinensprache. Das hat zwei Vorteile:

1. Wir müssen uns nicht mit der primitiven und technischen Maschinensprache herumschlagen und können so schneller und besser unser Programm erstellen.
2. Der BASIC-Text, der s.g. *Sourcecode* oder zu deutsch *Quelltext* (Gegensatz zum Executable) kann auch auf anderen Computern verwendet werden. Dann übersetzt ein entsprechendes BASIC-Übersetzungsprogramm den Text in die dortige Maschinensprache.

Solch ein Maschinenprogramm, das einen Quelltext Zeile für Zeile in Maschinensprache übersetzt, nennt man einen *Interpreter*. Der C16 hat also einen eingebauten BASIC-Interpreter. Natürlich kann man auch für andere Sprachen Interpreter programmieren, so gibt es auch LISP-Interpreter, LOGO-Interpreter, Javascript-Interpreter usw.

Zu Interpretern gibt es auch eine Alternative: *Compiler*. Compiler übersetzen den Quelltext nicht Zeile für Zeile, sondern "auf einen Rutsch". Das führt dazu, dass man in einem Interpreter ein Quelltext-Programm einfach starten kann, als wäre es ein Executable. Bei der Arbeit mit einem Compiler muss man immer vorher den Übersetzungsvorgang starten, man muss den ganzen Quelltext *compilieren*. Der Lohn der Mühe ist, dass man dann ein fertiges Maschinenprogramm hat, das in der Regel einen Faktor 5 bis 10 schneller läuft, als interpretierte Programme.

Für den C16 wird es früher wohl keinen BASIC-Compiler gegeben haben. Der Grund lag in seinem zu kleinen Hauptspeicher. Ein Compiler braucht viel Hauptspeicher: Der Compiler selbst (er ist ja nicht fest eingebaut, wie der BASIC-Interpreter, sondern wird von Cassette oder Diskette geladen) muss gespeichert werden, der Quelltext und das Compilat, das fertige Maschinenprogramm. Dafür sind selbst 64K sehr knapp.

Ereignisse

Wahrscheinlich haben Sie während Ihrer Spieleprogrammierung in den letzten Kapiteln eine Funktion am C16 bitter vermisst: Die direkte Reaktion auf einen Tastendruck. Die einzige Möglichkeit, die Tastatur einzubeziehen, ist bisher ja der Input-Befehl. Für viele Fälle, insbesondere für Spiele, in der eine Spielfigur gesteuert werden soll, Pfeil nach rechts - Figur nach rechts usw., ist Input völlig ungeeignet: "In welche Richtung wollen Sie?" - "Rechts+ÉENTER" - "Vielen Dank, es geht jetzt nach rechts." Toll. Nein, so kann keine Spannung beim Formel-1-Fahren aufkommen.

Ein Tastendruck ist das Beispiel eines *Ereignisses*, auf das das Programm irgendwie reagieren soll. Der Vorteil des C16-Systems ist u.a., dass so etwas darauf einfach zu programmieren ist - ganz im Gegensatz zu den grösseren, heute üblichen Programmiersystemen auf dem PC.

GETKEY

GETKEY ist INPUT noch am ähnlichsten. Bei "GETKEY A\$" wartet der Computer einfach auf den nächsten Tastendruck. Und übergibt das Ergebnis - also das Zeichen, das der gedrückten Taste entspricht - in A\$. Auf ein ENTER wartet der Computer nicht, sondern er reagiert sofort. Dadurch ist es hier natürlich nicht möglich, auf einfache Weise mehr als ein Zeichen zu übergeben - Strings müssen mit INPUT übergeben werden.

Wieviel Programmzeilen brauchen Sie für einen einfachen Editor auf dem PC? GETKEY macht Ihnen die Arbeit sehr einfach:

```
10 REM EINFACHER EDITOR FUER DEN C16
20 LET A$=""
30 DO WHILE A$<>"&"
40 GETKEY A$
50 IF A$<>"&" THEN PRINT A$;
60 LOOP
```

Das war's schon. Probieren Sie es aus! Sie können nun ganz normal auf dem C16 tippen, Zeilenumbrüche eingeben, sogar mit Backspace (rückwärts löschen) und den Cursortasten arbeiten. Mit dem &-Zeichen verlassen Sie den Editor wieder. Alles kein Problem. Weil der C16 alle Steuerzeichen A\$ übergibt und der PRINT-Befehl die Steuerzeichen auch wieder korrekt umsetzt.

GET

Was aber machen, wenn der PC etwas tun soll, während er auf die nächste Tasteneingabe wartet? Nun, dann brauchen wir einen Befehl, der lediglich in der Zeit auf die Tastatur lauscht, in der der Befehl abgearbeitet wird. Der PC bleibt nicht am Befehl stehen, sondern schaut nur kurz auf die Tastatur und saust dann weiter. Wenn gerade nichts auf der Tastatur passiert ist, gibt der Befehl eben "Nichts" zurück. Das ist der GET-Befehl des C16.

```
10 REM TEST VON GET
20 LET A$=""
30 DO WHILE A$<>"&"
40 LET A$=""
```



```

50 DO WHILE A$=""
60 GET A$
70 IF A$="" THEN PRINT "X";
80 IF A$<>"" THEN PRINT A$
90 LOOP
100 LOOP

```

Auch dieses Testprogramm verlassen Sie mit "&" (oder mit einem BREAK mittels Tabtaste). Das Programm spuckt laufend Zeichen aus. Wenn keine Taste gedrückt ist, spuckt es ein "X" aus, ansonsten das Tastaturzeichen. Aus gutem Grund wurde in Zeile 80 das Semikolon weggelassen. So sieht man am Besten, was passiert, wenn eine Taste gedrückt wird.

Was wir als Ergebnis erwarten würden, wäre z.B.:

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXD
D
D
D
XXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

...wenn ein "D" gedrückt und wieder losgelassen wurde.

Wir erhalten aber:

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXD
XXD
XD
XD
XXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

Was passiert da? Nun, obwohl wir eine Taste gedrückt halten, wird offenbar nur ab und an von GET ein Zeichen "gesehen". Das liegt daran, dass die Dauerfunktion einer Taste mit einer bestimmten Rate Tastensignale abgibt ("feuert") und diese sehr viel langsamer ist als unser Programm! Deshalb dauert es zwei bis drei Schleifendurchläufe, bis GET wieder einem Signal begegnet. Was kann man tun? Man muss die Abfrage soweit verlangsamen, dass GET seltener nachschaut, als die Taste "feuert". **Übungsaufgabe!**

Übungsaufgabe: Räuber und Gendarm

Schreiben Sie ein kleines Spiel, in dem auf dem Bildschirm ein Auto herumfährt. Sie können das Auto nicht stoppen, Sie können es nur nach rechts und links steuern. Auf dem Bildschirm befindet sich auch ein zweites Auto. Das währt vom Computer gesteuert, d.h. zufällig in der Gegend herum. Und Sie müssen das zweite Auto fangen!

Hochauflösende Grafik

Text- und Pixeldarstellung

Oft besteht der Wunsch, etwas auf den Bildschirm zu malen. Für ein Spiel sollen es Auto's, Figuren oder Ufo's sein. Für die Arbeit Kurven, Pläne oder Charts.

An sich sind wir dazu schon in der Lage. Wir können schon Spiele mit einer Spielfigur erstellen. Und haben schon Balkendiagramme gezeichnet (im Statistikrechner). Aber es entsteht die typische "Klötzchengrafik" der frühen PC-Zeit: Wir müssen das Bild aus lauter viereckigen Klötzchen zusammensetzen, nämlich unseren Zeichen. Das Bild ist also aus einem 40 mal 25 - Raster zusammengesetzt. Kann man dieses Raster nicht feiner machen?

Man muss es irgendwie können. Denn die Zeichen auf dem Bildschirm sind in sich ja auch feiner gerastert. Wenn Sie genau hinsehen, erkennen Sie, dass jedes Zeichen aus 7 mal 7 Punkten zusammengesetzt ist. So ein Zeichen nennt man ein "Pixel". Wobei jeweils noch ein Punkt Abstand bleibt. Insgesamt sind es also 8 mal 8 Pixel pro Zeichen. Und $40 \times 8 = 320$ Pixel in der Horizontalen und $25 \times 8 = 200$ Pixel in der Vertikalen insgesamt. Können wir diese 320 mal 200 Pixel nicht separat ansteuern?

Bei einem leistungsfähigen Computer kann man das. Bei PC's heute sowieso kein Thema mehr. Dennoch sollte man die Unterscheidung im Kopf behalten: Es gibt bei der Bildschirmdarstellung zwei unterschiedliche Modi: Textdarstellung und Pixeldarstellung. Beim ersteren wird der Bildschirm aus Zeichen zusammengesetzt, beim letzteren aus einzelnen Pixeln.

Unsere erste Pixelgrafik

Den C16 stellt man mittels des Befehls GRAPHIC auf Pixeldarstellung um. Dabei sollten Sie allerdings vorsichtig sein. Wenn der C16 im HiRes (High Resolution Graphic- Modus) ist, dann kann er keinen Text mehr darstellen. Jedenfalls nicht mehr so einfach. Und damit auch keine Fehlermeldung, kein "READY" und keine Zeichen, die Sie mit der Tastatur eingeben: Im HiRes-Modus sind Sie "blind". Wenn Sie ein Programm schreiben, sollten Sie also

1. am Ende immer wieder nach GRAPHIC 0 zurückschalten lassen (Textmodus).
2. eine Programmunterbrechung möglichst vermeiden.
3. sich darauf einstellen, im Notfall GRAPHIC 0 + ENTER blind einzutippen.

Koordinatensysteme

Wie steuere ich ein Pixel mit dem C16 an? Es muss ja quasi einen "Namen" tragen. Der "Name" ist seine Position im Raster, also quasi "Zeile" und "Spalte", wie beim CHAR-Befehl. Man nennt Zeile und Spalte in diesem Zusammenhang *Koordinaten* und benennt sie meistens mit X (Spalte) und Y (Zeile). Aber Vorsicht! In der Mathematik zeigt die Y-Achse meist von unten nach oben, d.h. 0 ist unten und 199 oben. Beim Programmieren behält Y die Zeilenlogik bei: 0 ist oben und 199 unten. Es gibt also nun 320 "Spalten" und 200 "Zeilen". Der Befehl **DRAW 1,X,Y** zeichnet einen Punkt an

die Koordinate X,Y. Die Eins steht für die Farbe: Vordergrundfarbe. Die Null wäre die Hintergrundfarbe. DRAW funktioniert allerdings nur, falls der HiRes-Modus eingeschaltet ist. Das macht man mit **GRAPHIC 1,1**. Im folgenden ein kleines Beispielprogramm:

```
10 REM DER C16 BEKOMMT MASERN
20 GRAPHIC 1,1
30 FOR I=1 TO 100
40 LET X=INT(RND(1)*320)
50 LET Y=INT(RND(1)*200)
60 DRAW 1,X,Y
70 NEXT I
80 CHAR ,0,20,"DRUECKEN SIE EINE TASTE"
90 GETKEY A
100 GRAPHIC 0
```

Das ist natürlich noch nicht so beeindruckend. Aber es zeigt das Prinzip. Und es zeigt, dass das mit dem Zeichen-Ausgeben auf dem HiRes-Schirm doch funktioniert: Aber eben nur mit dem CHAR-Befehl. PRINT und INPUT gehen nicht. Hier kommt ein SYNTAX ERROR.

Eine Kurve zeichnen

In der Mathematik ist eine Kurve ein Bild in einem XY-Diagramm, in dem zu jedem X ein Y zugeordnet und der entsprechende Punkt eingezeichnet wird. Wir wollen einmal den PC die Kurve zeichnen, die sich ergibt, wenn $y=0.2*x^3+x^2$ (^ steht für den Pfeil nach oben) sein soll. Der interessante Wertebereich für x liegt zwischen -5 und +5.

Das Problem ist bei allen Computergrafiken, von den Computerkoordinaten (0 bis 319) auf die Weltkoordinaten (-5 bis +5) zu kommen und in Y wieder zurück auf Computerkoordinaten (0 bis 199). Dazu braucht es etwas Rechnerei, die im folgenden Programm in den Zeilen ??? und ??? vorgenommen wird. Dazwischen steht die eigentliche Formel, wobei für die Parameter 0.2 vor x^3 und 1 vor x^2 A und B eingesetzt wurde.

```
10 REM PUENKTCHENPARABEL
20 LET A=0.2
30 LET B=1
40 LET DX=10
50 LET XMIN=-5
60 LET DY=10
70 LET YMIN=-2
80 GRAPHIC 1,1
90 FOR I=0 TO 319
100 REM I=X-COMPUTER
110 LET X=XMIN+DX*I/319
120 LET Y=A*X^3+B*X^2
130 LET YC=INT(Y/DY*200+YMIN)
140 DRAW 1,I,YC
150 NEXT I
```

Hier noch eine Erklärung zu den Variablen:

- I: Computer-X-Koordinate
- X: Welt-X-Koordinate
- Y: Welt-Y-Koordinate
- YC: Computer-Y-Koordinate
- XMIN: Linker Rand der X-Achse (Weltsystem)
- DX: Breite X-Achse (weltsystem)

- YMIN: Unterer Rand der Y-Achse (Weltsystem)
- DY: Höhe Y-Achse (Weltsystem)

Linien zeichnen

Die Pünktchenkurve oben ist schon ganz hübsch. Noch hübscher wäre es, wenn wir die einzelnen Punkte mit Linien verbinden könnten. Natürlich können wir das mit lauter weiteren Pünktchen machen. Aber der C16 bietet uns mehr Komfort: Er hat einen Befehl zum Zeichnen von Linien. Und zwar ist das der gleiche Befehl DRAW, den wir schon können. Aber nun mit erweiterter Syntax:

DRAW 1,X1,Y1 TO X2,Y2.

Dieser Befehl zeichnet im Computer-Koordinatensystem eine Linie vom Punkt (X1/Y1) zum Punkt (x2/y2).

Übungsaufgabe: Erweitern Sie das obere Kurvenzeichen-Programm so, dass es die Punkte mit Linien verbindet. Denken Sie daran, dass sich der Computer immer an den vorhergehenden Y-Wert erinnern muss, wenn er die Linie zeichnen soll. Und dass am Anfang beim ersten Punkt es nichts zurückzuerinnern gibt. Elegant lösen Sie das so, dass Sie am Anfang den "Erinnerungspunkt" auf den ersten zu zeichnenden Punkt legen. Viel Spass!

Weitere Grafikbefehle

Der C16 kennt noch eine ganze Reihe weiterer Grafikbefehle:

CIRCLE

Einfache Syntax: CIRCLE COL,X,Y,R zeichnet einen Kreis um (X/Y) mit Radius R. COL=0 oder 1

Vollständige Syntax: CIRCLE COL,X,Y,XR,YR,W1,W2,ROT,DEG zeichnet ein Polygon entlang eines Ellipsenumfangs. XR=Radius der Halbachse in X-Richtung der Umfangsellipse, YR=Radius der Y-Halbachse, W1 und W2 Start- und Endwinkel, falls das Polygon nur unvollständig gezeichnet werden soll (Segment der Umfangsellipse), ROT=Rotationswinkel der X-Halbachse gegen die Horizontale, DEG=Winkel zwischen zwei Eckpunkten des Polygons. DEG=1 zeichnet faktisch eine Ellipse, DEG=45 ein Oktogon, DEG=90 ein Viereck und DEG=120 ein Dreieck.

BOX

Einfache Syntax: BOX COL,XLINKS,YOBEN,XRECHTS,YUNTEN zeichnet ein Viereck zwischen Punkt (XLINKS,YOBEN) und (XRECHTS/YUNTEN).

Vollständige Syntax: BOX COL,XLINKS,YOBEN,XRECHTS,YUNTEN,ROT,FILL. Rotiert das Viereck um ROT Grad gegen die Horizontale. FILL=1: Viereck wird gefüllt. FILL=0: Es wird nur der Umriss gezeichnet.

PAINT

PAINT COL,X,Y,MODE füllt eine Fläche, die durch einen Umriss definiert ist, mit der Farbe COL. (Also COL=1 im HiRes-Modus). MODE ist im HiRes-Modus ohne Belang, kann auf Null gesetzt sein.

SSHAPE

Mit SSHAPE kann man einen kleinen Bildschirmausschnitt abspeichern, um ihn dann mit GSHAPE an einer beliebigen anderen Stelle des Bildschirm wieder auszugeben. Auf diese Art und Weise kann man bewegte kleine Figuren programmieren.

Syntax: SSHAPE String,XLINKS,YOBEN,XRECHTS,YUNTEN. Die Koordinaten geben

dabei den Bildschirmausschnitt an, der gespeichert werden soll. STRING kann ein String sein, z.B. "UFO" oder eine Stringvariable wie A\$. Weiter unten ist ein kleines Beispielprogramm zu SSHAPE und GSHAPE angegeben, das ein kleines Ufo über den Bildschirm hoppeln lässt. Man nennt eine solche bewegte kleine Grafik auch eine "Sprite" oder "Bob".

GSHAPE

Syntax: GSHAPE STRING,X,Y,MODE. STRING enthält die Grafikdaten des Sprites, (X/Y) ist der Punkt der linken oberen Ecke des Sprites und MODE ist ein s.g. *Flag*, eine Zählvariable, die verschiedene Zustände beschreibt. MODE=0, also Modus null, bedeutet, dass das Sprite einfach auf den Bildschirm gezeichnet und alles Darunterliegende überdeckt wird.

- Mode 0: Überdeckend zeichnen.
- Mode 1: Invertiert zeichnen.
- Mode 2: Schwarze Pixel der Shape füllen weisse Pixel des Hintergrunds. (OR-Verknüpfung der Shape- und Hintergrundpixel.)
- Mode 3: Weisse Pixel des Hintergrunds "bohren" sich in die Shape. (AND-Verknüpfung). Das heisst, die "Löcher" der Shape sind durchsichtig.
- Mode 4: Shape bildet "Negativ" des Hintergrunds. (Exklusiv-OR (XOR-) Verknüpfung.)

Beispielprogramm zu SSHAPE/GSHAPE

Das folgende Programm zeichnet mit DRAW-Befehlen ein kleines Ufo, speichert es mit SSHAPE ab und lässt es dann mittels Anwendung der Befehle BOX (Löschen des Ufos an alter Position) und GSHAPE (Zeichnen an neuer Position) über den Bildschirm "fliegen". Da der C16 nun mal nicht der schnellste Computer war und er auf Ihrem PC in Originalgeschwindigkeit läuft, ist das "Fliegen" allerdings etwas holprig. Man sieht deutlich, wie die einzelnen Bilder gemalt und wieder gelöscht werden.

```

10 GRAPHIC 1,1
20 DRAW 1,101,100 TO 114,100
30 DRAW 1,100,99: DRAW 1,115,99
40 DRAW 1,101,98: DRAW 1,114,98
50 DRAW 1,102,97 TO 103,97
55 DRAW 1,112,97 TO 113,97
60 DRAW 1,104,96 TO 111,96
70 DRAW 1,107,95 TO 107,94
80 DRAW 1,107,94 TO 109,94
90 DRAW 1,109,94 TO 109,95
100 GOSUB 200
110 GOSUB 400
130 GETKEY AS$
140 GRAPHIC 0,1
150 END
200 REM ABSPEICHERN
210 SSHAPE AS$,100,94,115,100
220 SCNCLR
300 RETURN
400 REM GRAFIK BEWEGEN
405 X=0:Y=100
406 DX0=10:DY0=6
407 DX=DX0:DY=DY0
410 FOR I=1 TO 300
415 IF I>1 THEN BOX 0,X1,Y1,X1+15,Y1+10,
,1
420 GSHAPE AS$,X,Y,0
425 X1=X:Y1=Y
430 X=X+DX
440 Y=Y+DY
450 IF Y>200 OR Y<0 THEN DY=-DY
460 IF X>320 OR X<0 THEN DX=-DX
470 NEXT I
490 RETURN

```

Im hochauflösenden Grafikmodus gibt es nur zwei Farben: Hintergrunds- und Vordergrundsfarbe. Warum das beim C16 so ist und beim PC nicht, das erfahren Sie im nächsten Abschnitt. Aber auch der C16 hat noch Farbe im Köcher: Mit dem Farbgrafik-Modus (Multicolor Graphic Mode). Hier erhöht sich die Zahl der Farben auf 4, allerdings auf Kosten der Auflösung: Nun sind nur noch 160 statt 320 Pixel in der Horizontalen möglich.

GRAPHIC 3,1 schaltet den C16 in den Farbgrafikmodus. Ab da kann die Vordergrundfarbe die Werte 1 bis 3 annehmen, während 0 die Hintergrundsfarbe bleibt. **DRAW 3,0,0 TO 159,199** zeichnet also eine Linie von ganz links oben bis ganz rechts unten in der Farbe 3. Mit dem Befehl **COLOR** (den hatten wir schon), können Sie dann den Farbregistern 0 bis 4 (0=Hintergrund, 1 bis 3 = Vordergrund, 4=Rahmen) ihre Farb- und Helligkeitswerte zuweisen.

Übungsaufgabe: Passen Sie das Kurvenzeichnenprogramm an den Farbmodus an! Zeichnen Sie Koordinatenachsen in einer anderen Farbe ein! Lassen Sie das Programm das Minimum und Maximum der Kurve bestimmen und zeigen Sie es mit einem roten Viereck.

Grafik und Speicher

Sie müssen sich mit dem C16 und seinen Grafikmöglichkeiten nicht besonders ausführlich auseinandersetzen. Sie sind, verglichen mit einem heutigen PC, sehr beschränkt und dieses Kapitel soll nur ein paar Grundbegriffe und ein gewisses Gefühl für Pixelgrafik vermitteln.

In diesem Abschnitt werden wir die Grafik-Beschränkungen des C16 kurz anschauen, um zu verstehen, was Speicher und Grafik miteinander zu tun haben. Das ist nicht nur für Sie als Programmierer sehr nützlich zu wissen, sondern auch als Benutzer einer Digitalkamera, eines DV-Camcorders, von Programmen für Videoschnitt und -kompression, für den Download von Filmen aus dem Internet usw.

Der Bildschirmspeicher

Jeder Computer muss das, was er auf dem Bildschirm anzeigt, irgendwie zwischenspeichern. Die Bildschirmelektronik muss 50 bis 100 Mal in der Sekunde für jeden Fleck des Bildschirms wissen, was dort stehen soll und das kann sie nicht direkt vom Programm erfahren, sonst müssten sich die Programme mit nichts anderem beschäftigen als dem Bildschirmaufbau.

Um das zu verhindern, gibt es den Bildschirmspeicher, in dem das gesamte Bild in irgendeiner Form abgelegt ist und den die Videoelektronik auslesen kann. Bei einfachen Computern wie dem C16 handelt es sich dabei um einen Teil des Hauptspeichers.

Wie ist der Bildschirminhalt gespeichert? Im Textmodus in Form der einzelnen Zeichen. Das heisst, der ganze Bildschirmspeicher ist 40x25 Bytes = 1000 Bytes gross. Bei 16K Speicher kein Problem. Allerdings kommen nochmals 1000 Bytes für die Farbe dazu, macht also zusammen 2K. Unter anderem wegen dieser 2K zeigt der C16 am Anfang nur verfügbare 12K und nicht 16K an.

Im Grafikmodus wird jedes Pixel getrennt abgespeichert. Ein Bild, das Pixel für Pixel gespeichert ist, nennt man eine **Bitmap**. Das sind dann 320x200=64000 Pixel. Wenn jedes Pixel ein Byte

bräuchte, bräuchte man dazu 64K Bildschirmspeicher. Viel zu viel für den kleinen C16. Allerdings könnte dann jedes Pixel 256 verschiedene Zustände einnehmen, d.h. es wären 256 Farben pro Pixel möglich. Wir sehen also: Wir können deshalb nicht wie beim Text "beliebige" Farben auswählen, weil dann der Bildschirmspeicher nicht ausreichen würde. Und selbst wenn er ausreichen würde: Der C16 ist als Uralt-Computer nicht in der Lage, 64K 50 mal in der Sekunde an den Monitor zu übertragen. Daher musste man die Zahl der Bits pro Pixel von 8 auf eins senken. Dann haben wir 2 mögliche Farben (Bit=0 oder 1) und nur 8K Bildschirmspeicher. Das ist die s.g. 8K-Grafik, die damals bei Homecomputern üblich waren.

Jetzt wird aber auch klar, dass, wenn wir die HiRes-Grafik einschalten, nicht mehr so wahnsinnig viel Speicher für Programm und Daten haben: 12K-8K=4K Speicher bleiben übrig. Vielleicht haben Sie schon bemerkt, dass im HiRes-Modus die "Out of Memory"-Meldung ziemlich bald erscheint.

Im Farbgrafikmodus werden die 8K anders aufgeteilt: 160x200 Pixel x 2 Farbbits. Denn mit 2 hoch 2 = 4 Zuständen kann man 4 Farben darstellen. Nach dem gleichen Prinzip funktioniert das auch mit Ihrer Digitalkamera und Ihrem Camcorder: Wollen Sie die Zahl der Pixel erhöhen, brauchen Sie mehr Speicher, d.h. es geht nicht mehr soviel Foto oder Film auf Ihr Speichermedium. Die Farbauflösung können Sie nicht reduzieren, da in diesem Fall noch Kompressionsverfahren benutzt werden, weil ansonsten auf Ihre ganze Festplatte nur 10 Minuten Film gehen würden. Bei diesen Kompressionsverfahren (MPEG, DivX) würde es nichts bringen, die Farbauflösung zu reduzieren, im Gegenteil: Man bräuchte sogar mehr Speicher. Aber Sie begegnen der Farbauflösung bei den Einstellungen Ihrer Bildschirmanzeige, bei der reduzierten Farbauflösung von kleinen Internetgrafiken (GIFs: Beschränkt auf 256 Farben), usw.

Einige gängige Grafikmodi und ihr Speicherbedarf

Grafikmodus	Auflösung	Farben	Speicher pro Pixel	Speicherbedarf
C16 Multicolor	160x200	4	2 Bit	8K
C16 HiRes	320x200	2	1 Bit	8K
DOS: CGA1	320x200	4	2 Bit	16K
DOS: CGA2	640x200	2	1 Bit	16K
DOS: EGA64Lo	640x200	16	4 Bit	64K
DOS: EGA64Hi	640x350	4	2 Bit	64K
DOS: VGA	320x200	256	8 Bit	64K
DOS: VGA	640x480	16	4 Bit	256K
Windows: VGA	640x480	256	8 Bit	512K
Windows: SVGA	800x600	256	8 Bit	512K
Windows: SVGA	800x600	65000	16 Bit	1 MB
Windows: XGA	1024x768	65000	16 Bit	2 MB
Windows: XGA	1024x768	16 Mio.	24 Bit	4 MB
Windows: SXGA	1280x1024	16 Mio.	24 Bit	4 MB
Windows: SXGA	1280x1024	32 Mrd.	32 Bit	6 MB

Windows: UXGA	1600x1200	32 Mrd.	32 Bit	8 MB
---------------	-----------	---------	--------	------

Beim Lesen dieser Tabelle dürfte die Logik klar werden. Heute gängige Grafikkarten haben mindestens 32 MB Speicher und sind damit locker in der Lage, höhere Auflösungen in 4 Byte Farbtiefe ("Truecolor") zu liefern, als ein irgendwie bezahlbarer Monitor darstellen kann. In der Praxis sind die XGA-Auflösungen noch sehr beliebt, vor allem in Verbindung mit LCD-Displays.