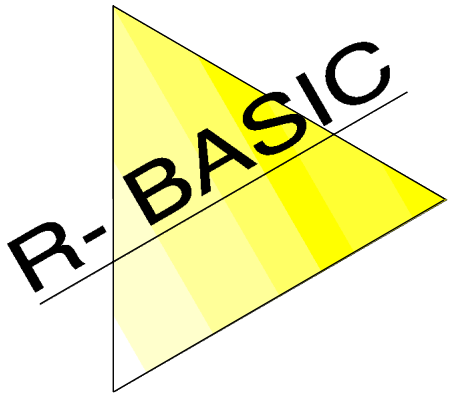
R-BASIC

Einfach unter PC/GEOS programmieren



Programmierhandbuch

Volume 1
Tutorial, Konzepte, Variablen und Typen

Version 1.0

(Leerseite)

Inhaltsverzeichnis

(Leerseite)

1. Ein Anfänger-Tutorial

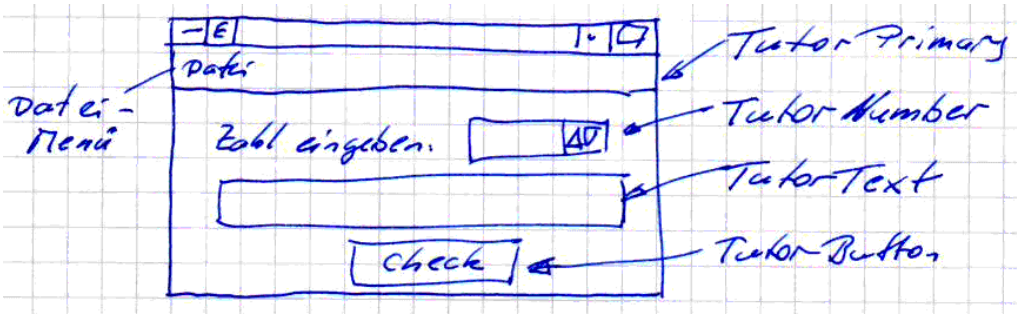
Dieses Tutorial beschreibt an einem einfachen Beispiel, wie man ein R-BASIC Programm erstellt. Dabei werden viele Fakten und Zusammenhänge einfach benutzt, ohne sie ausführlich zu erklären. Stattdessen gibt es jeweils Verweise auf die Kapitel im Handbuch, wo Sie weiterführende Erklärungen finden. Einige Begriffe sind **fett** markiert. Diese Begriffe kommen im Handbuch immer wieder vor. Eine exakte Erklärung dieser Begriffe finden Sie im nächsten Kapitel.
Die Schrittfolge, die hier dargestellt wird, können Sie prinzipiell auf jedes Programm anwenden.

1. Der Plan ...

Als Einstieg wollen wir uns ansehen, wie man das folgende Programm erstellt. Der Nutzer soll eine Zahl eingeben und das Programm soll prüfen, ob die Zahl gerade ist oder nicht.

2. Die UI (das User-Interface)

Das Erstellen eines Programms beginnt in den allermeisten Fällen mit der Programmierung der Benutzeroberfläche, der UI. Das heißt, Sie legen fest welche Objekte es gibt und wie sie angeordnet sind.
Jedes Objekt hat einen Typ, der Programmierer sagt, es gehört einer **Klasse** an. Die Klasse bestimmt die Eigenschaften und Fähigkeiten eines Objekts. Die in R-BASIC verfügbaren Klassen und deren Eigenschaften sind im "Objekt-Handbuch" beschrieben. Es ist durchaus vernünftig, sich zunächst eine Zeichnung anzufertigen. So kann man sich überlegen, wie die Programmoberfläche ungefähr aussehen könnte und wie viele und welche Objekte man benötigt. Außerdem kann man den wichtigsten Objekten jetzt schon einen aussagekräftigen **Namen** geben.



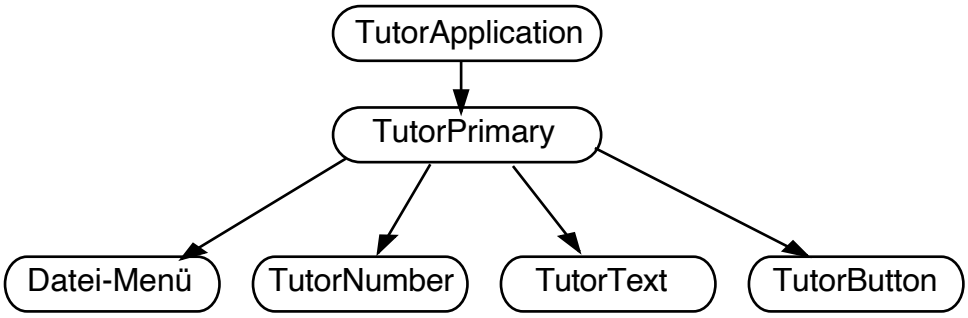
Prinzipiell sind Sie in der Wahl des Namens frei, es hat sich jedoch bewährt, den Namen aus zwei Teilen zusammenzusetzen. Der erste Teil beschreibt das Programm oder den Programmteil, zu dem das Objekt gehört. In unserem Fall wählen wir "Tutor". Der zweite Teil bezeichnet die Klasse, zu der das Objekt gehört. Das Primary Objekt heißt daher TutorPrimary, der Button heißt TutorButton usw. Das Datei-Menü ist auch ein Objekt. Es wird aber vom Primary-Objekt automatisch erzeugt, so dass wir uns darum nicht kümmern müssen.

Für unser Programm benötigen also wir 5 **Objekte**:

1. Ein Objekt der Klasse "Number". Hier kann der Nutzer die Zahl eingeben.

2. Ein Objekt der Klasse "Memo". Das ist ein Text-Objekt, mit dem wir das Ergebnis unserer Überprüfung anzeigen wollen.
3. Ein Objekt der Klasse "Button". Das ist der Schalter, mit dem wir die Überprüfung starten wollen.
4. Das eigentliche Programmfenster ist auch ein Objekt. Es gehört der Klasse "Primary" an.
5. Zusätzlich benötigt jedes Programm ein unsichtbares Objekt der Klasse "Application". Dieses stellt die Verbindung zum GEOS-System her.

Alle Objekte sind miteinander verbunden. Man sagt, ein Objekt hat "**children**" (Kinder). Details zu Children und die Organisation von Objekten in Bäumen (Trees) finden Sie im Kapitel 2.1 (Objekte und Objekt-Bäume) des Objekt-Handbuchs.
In unserem Fall sieht die Verbindung, der sogenannte Objekt-Tree (Objektbaum), wie folgt aus. Zum Beispiel ist das Objekt TutorNumber ein child (Kind) des Objekts TutorPrimary. Im Gegenzug ist das Objekt TutorPrimary das parent (Eltern) des Objekts TutorNumber.



Es ist ganz wichtig, dass alle Objekte in den Objekttree eingebunden sind. Dazu gibt es die Anweisung "Children". **Vergessen** Sie das Einbinden eines Objekts in den Tree, so wird das Objekt **nicht auf dem Bildschirm** erscheinen. Es gibt darüber **keine Fehlermeldung**, da Sie nichts Verbotenes getan haben.

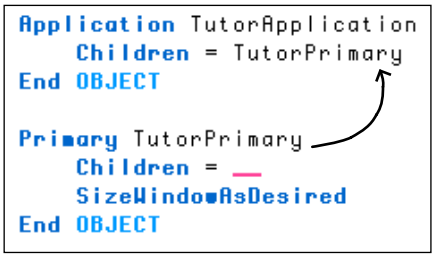
Alle Objekte werden im UI-Codefenster vereinbart. Das Application-Objekt hat genau ein Child, des Primary-Objekt.

```
Application TutorApplication
  Children = TutorPrimary
End OBJECT
```

Um ein Objekt anzulegen müssen Sie nicht jede Zeile einzeln tippen. Öffnen Sie stattdessen das Menü "Extras" -> "Code-Bausteine" -> "Neues Objekt". Dort finden Sie das Application-Objekt und auch alle anderen von uns benötigten Objekte. Sie brauchen nur noch die programmspezifischen Dinge ergänzen. Das Application-Objekt wird von uns nicht als Child eines anderen Objekts gesetzt, darum kümmert sich das System.

Nun legen wir das Primary-Objekt an und binden es in den Tree ein. Haben Sie daran gedacht, das Menü "Extras" -> "Code-Bausteine" -> "Neues Objekt" zu verwenden, um das Primaryobjekt anzulegen? Nachdem Sie das Objekt angelegt

haben erscheint automatisch ein Dialogbox, die Sie beim Einbinden des Objekts in den Tree unterstützt.



Unser Primary-Objekt soll drei Children haben: das Number-Objekt, den Text und den Button. Es ist unter GEOS nicht üblich die Position der Objekte explizit anzugeben. Stattdessen geben wir an, wie die Objekte angeordnet werden sollen. Details zum zu diesem Thema finden Sie im Kapitel 3.3 (Geometriemanagement) des Objekt-Handbuchs. Wir wollen, dass die Children horizontal zentriert sind (justifyChildren = J_CENTER). JustifyChildren ist ein **Instancevariable** des Primaryobjekts. Instancevariablen können unterschiedliche Werte annehmen. In unserem Fall weisen wir ihr den Wert J_CENTER zu. J_CENTER ist eine numerische **Konstante**, d.h. J_Center steht symbolisch für eine Zahl. Die **UI-Anweisung** SizeWindowAsDesired bewirkt, dass das Primary nur so groß ist, wie unbedingt nötig. Probieren Sie ruhig aus, was passiert, wenn Sie eine oder mehrere dieser Zeilen auskommentieren (Ausrufezeichen davor schreiben).

```
Primary TutorPrimary
  Children = TutorNumber, TutorText, TutorButton
  justifyChildren = J_CENTER
  SizeWindowAsDesired
End OBJECT
```

Für das Number-Objekt setzen wir zur Demonstration ein paar Instancevariablen. Die Instancevariable Caption$ enthält Text, der das Number-Objekt näher beschreiben soll. Er erscheint direkt neben oder auf dem Objekt. Außerdem legen wir einen Startwert (value) sowie einen Minimal- und einen Maximalwert (minVal, maxVal) fest. Unser Numberobjekt stellt nur ganze Zahlen dar. Im Kapitel 4.7.2 (Display-Format) des Objekt-Handbuchs ist beschrieben, wie man Numberobjekte konfiguriert, damit sie Dezimalstellen darstellen können.

```
Number TutorNumber
  Caption$ = "Zahl eingeben:"
  value = 12
  minVal = -100
  maxVal = 100
End OBJECT
```

Das Ergebnis unserer Prüfung wollen wir in ein Textobjekt schreiben. Da der Nutzer dort nichts eingeben soll setzen wir das Textobjekt auf "nur Lesen", d.h. wir weisen der Instancevariablen readOnly den Wert TRUE zu. TRUE ist wieder eine numerische Konstante. Die UI-Anweisung TextFrame erzeugt einen Rahmen um das Objekt, der sonst bei read-only Objekten fehlt. Da wir noch keine Prüfung vorgenommen haben setzen wir als Anfangswert für den darzustellenden Text das

Wort "ungetestet". Schließlich bewirkt die UI-Anweisung justifyText = J_CENTER, dass der Text zentriert dargestellt wird. Eine ausführliche Beschreibung der Textobjekte finden Sie im Kapitel 4.10 des Objekt-Handbuchs.

```
Memo TutorText
  text$ = "ungetestet"
  readOnly = TRUE
  TextFrame
  justifyText = J_CENTER
End OBJECT
```

Schließlich brauchen wir noch einen Button, mit dem wir die Prüfung der Zahl starten. Wenn wir den Button aktivieren wird eine spezielle **Routine**, der **Actionhandler** des Buttons, gestartet. Die Anweisung ActionHandler = Check-Number legt fest, dass die Routine "CheckNumber", die wir noch schreiben müssen, gestartet werden soll, wenn wir den Button aktivieren. Button-Objekte sind im Kapitel 4.3 des Objekt-Handbuchs beschrieben.
Die Null in der Caption$ - Zeile legt fest, dass der Buchstabe an Position Null (hier das ’P’) unterstrichen und zur Tastaturnavigation benutzt werden soll.

```
Button TutorButton
  Caption$ = "Prüfen" , 0
  ActionHandler = CheckNumber
End OBJECT
```

3. Der Actionhandler

Nun müssen wir noch den Actionhandler des Buttons schreiben. Actionhandler sind spezielle Routinen, die direkt von einem Objekt aufgerufen werden. Action-handler, die von einem Button aufgerufen werden, müssen als BUTTONACTION vereinbart werden. Auch hier benutzen wir das Menü "Extras" -> "Code-Bausteine" um den Actionhandler anzulegen (Unterpunkt "Action-Handler").

```
BUTTONACTION CheckNumber
DIM z        ’ z: zahl
  z = TutorNumber.value
  IF z/2 = Int(z/2) THEN
    TutorText.text$ = "gerade"
  ELSE
    TutorText.text$ = "ungerade"
  END IF
END ACTION
```

Die Zeile "z = TutorNumber.value" ist eine **Zuweisung**. Sie liest den Zahlenwert (value) des Numberobjekts aus und speichert ihn in der Variablen z. Man sagt, der Variablen z wird ein Wert zugewiesen. Z muss vorher mit DIM vereinbart worden sein.

Mit der IF-Anweisung können wir Entscheidungen treffen. Ist der zwischen IF und THEN stehende Ausdruck wahr, so werden die Anweisungen hinter THEN

ausgeführt (in unserem Fall ist es nur eine Anweisung), andernfalls die hinter ELSE. End IF schließt den Entscheidungsteil. Das Kapitel 2.5.1 (Verzweigungen) des Programmierhandbuchs beschreibt die Möglichkeiten von R-BASIC, Entscheidungen zu treffen.

Um festzustellen, ob die Zahl gerade ist vergleichen wir die Hälfte der Zahl (z/2) mit dem ganzzahligen Anteil (**Funktion** Int(..)) der halben Zahl. Ist zum Beispiel z = 5, so ist z/2 = 2,5 und Int(z/2) = 2. Die Werte sind nicht gleich, z ist ungerade. Einen Überblick über die in R-BASIC verfügbaren mathematischen Funktionen finden Sie im Kapitel 2.3.1 (Überblick über numerische Funktionen) des Programmierhandbuchs.

Die **Anweisung** TutorText.text$ = "...." weist dem Textobjekt einen neuen Text zu, der sofort angezeigt wird. Zeichenketten (genannt Strings) werden in " " eingeschlossen, Stringvariablen werden üblicherweise durch ein angehängtes $ gekennzeichnet. Das ist Ihnen bestimmt schon bei Caption$ aufgefallen.

Nachdem Sie das Programm eingetippt und gestartet haben sollten Sie damit herumspielen. Ändern Sie einzelne Codezeilen oder kommentieren Sie sie aus. Überlegen Sie vorher, was passieren wird und vergleichen Sie Ihre Vorhersagen mit dem erreichten Ergebnis.

4. Erste Verbesserungen

Wir wollen nun unser Programm etwas verändern, indem wir die Eingabe der Zahl nicht über ein Numberobjekt, sondern über ein Textobjekt machen. Das sieht besser aus und "Drag'n Drop" funktioniert auch.

Dazu verwenden wir die andere Textobjektklasse, über die R-BASIC verfügt. Das Objekt InputLine stellt einen einzeiligen Text bereit. Mit der Anweisung "justifyCaption = J_TOP" verschieben wir die Aufschrift über das Objekt. Captions sind sehr vielseitig. Sie können nicht nur Text sondern auch Grafiken enthalten. Details dazu finden Sie im Kapitel 3.1 (Caption: Die Objekt-Beschriftung) des Objekt Handbuchs.

```
InputLine TutorInputText
  Caption$ = "Zahl eingeben:"
  justifyCaption = J_TOP
  text$ = "12"
  backColor = WHITE
  textFilter = TF_SIGNED_NUMERIC + TF_NO_SPACES
End OBJECT
```

Der Startwert (text$ = "12") ist natürlich keine Zahl, sondern ein String. Mit "backColor = WHITE" machen wir den Texthintergrund weiß. WHITE ist eine numerische Konstante (ein Symbol für eine Zahl). R-BASIC hat für alle 16 Standardfarben eine entsprechende Konstante. Näheres zur Beschreibung von Farben finden Sie im Kapitel 2.8.2 (Farben) des Programmierhandbuchs.

Von besonderer Bedeutung ist für unsere Anwendung die Zeile "textFilter = TF_NUMERIC + TF_NO_SPACES". Textfilter sind eigentlich etwas für Fortgeschrittene, aber an dieser Stelle sehr hilfreich. Ein Textfilter weist das Objekt an, nur bestimmte Zeichen zu akzeptieren. In unserem Fall sind die Werte TF_SIGNED_NUMERIC (nur Ziffern, Minus und Leerzeichen) kombiniert mit TF_NO_SPACES (keine Leerzeichen) angebracht. Das Setzen eines Textfilters erspart uns bei der Auswertung die Aufwändige Prüfung, ob der Nutzer überhaupt eine Zahl und nicht etwa "Paul" oder "eins" eingegeben hat. Textfilter sind im Kapitel 4.10.5 (Textfilter) des Objekthandbuchs beschrieben.

Bitte vergessen Sie nicht, das neue Objekt anstelle des Number-Objekts als Child des Primary-Objekts einzutragen.

Natürlich müssen wir jetzt unsere Auswerteroutine anpassen. Insbesondere müssen wir den vom Nutzer eingegebenen Text in eine Zahl umwandeln. Diese Aufgabe erledigt die Funktion Val(...). Val steht für das englische Wort value (Wert). "t$ = TutorInputText.text$" speichert zuvor den vom Nutzer eingegeben Text in der Stringvariablen t$. Die DIM-Anweisung erkennt an dem angehängten $-Zeichen, dass die Variable einen Text und keine Zahl aufnehmen soll.

```
BUTTONACTION CheckNumber
DIM z, t$    ' zahl, text
  t$ = TutorInputText.text$
  z = Val(t$)
  IF z/2 = int(z/2) THEN
    TutorText.text$ = "gerade"
  ELSE
    TutorText.text$ = "ungerade"
  END IF
END ACTION
```

Tipp: Da die Variable t$ nirgends weiter gebraucht wird kann man sie auch einsparen. Die ersten beiden Zeilen werden dann zu:

```
  z = Val( TutorInputText.text$ )
```

Details zu Variablen und den zugehörigen Datentypen finden Sie weiter unten, im Kapitel 2.2 (Variablen und Typen). Der leichte Umgang mit Zeichenketten (Strings) gehört zu den großen Stärken der BASIC-Syntax. Einen Überblick über die in R-BASIC zur Verfügung stehenden Stringfunktionen finden Sie im Abschnitt 2.4.1 des Programmierhandbuchs.

5. Weitere Änderungen

Nun möchten wir die am Anfang eingestellte Zahl durch eine zufällig ausgewählte Zahl ersetzen. Dazu benötigen wir eine Routine, beim Programmstart automatisch ausgeführt wird. Das ist eine sehr häufig auftretende Situation. In R-BASIC ist das so gelöst, dass wir dem Application-Objekt einen "OnStartup" Handler geben. Wir

nennen ihn "AppStartup". Wir hätten ihn auch "Paul" nennen können, aber die Namenswahl AppStartup enthält gleichzeitig einen Hinweis darauf, was der Handler tun soll. Das ist prinzipiell immer eine gute Idee. Der UI-Code des Application-Objekts sieht jetzt also so aus:

```
Application TutorApplication
  Children = TutorPrimary
  OnStartup = AppStartup
End OBJECT
```

Das Application-Objekt ist im Kapitel 4.1 des Objekthandbuchs beschrieben. Es bietet zum Beispiel die Möglichkeit einem Programm ein eigenes Token (Icon) zu geben und vieles mehr.

Unser AppStartup Handler muss als SYSTEMACTION vereinbart werden. Je nachdem, ob wir das Number-Objekt TutorNumber oder das InputLine-Objekt TutorInputText zur Eingabe der Zahl verwenden fällt der Handler geringfügig anders aus.
Wir schauen uns zunächst die Variante mit dem Number-Objekt an.

```
' Variante  mit Number-Objekt
SYSTEMACTION AppStartup
DIM z        ' zahl
  Randomize
  z = Int( 100 * Rnd() )
  TutorNumber.value = z
END ACTION
```

Als erstes initialisieren wir mit der Anweisung Randomize den Zufallsgenerator. Das sollte jedes Programm, das mit Zufallszahlen arbeitet, tun.

Die Funktion Rnd() liefert eine Zufallszahl x im Bereich 0 <= x < 1. Wenn wir unsere Zufallszahl z im Bereiche 0 <= z < 100 haben wollen müssen wir diesen Wert mit 100 multiplizieren. Die Funktion Int() (Int steht für Integer, ganzzahlig) schneidet die Nachkommastellen ab. Die Zeile z = Int(100 * Rnd()) weist also der Variablen z eine Ganze Zahl im Bereich zwischen 0 und 99 zu.

Schließlich weisen wir mit der Anweisung TutorNumber.value = z dem Objekt TutorNumber den Wert der Variablen z zu. Das Objekt zeigt diesen Wert sofort an.

Wenn wir das Textobjekt zur Eingabe der Zahl verwenden sieht der Handler so aus:

```
SYSTEMACTION AppStartup
DIM z        ' zahl
  Randomize
  z = Int( 100 * Rnd() )
  TutorInputText.text$ = Str$(z)
END ACTION
```

Wir sehen, dass sich nur die letzte Zeile unterscheidet. Zunächst heißt die Instancevariable, die wir belegen müssen, bei einem Textobjekt "text$". Diese enthält natürlich eine Zeichenkette (einen String) und keine Zahl. Deswegen müssen wir die Zahl z mit der Funktion Str$() in eine Zeichenkette umwandeln.
Das ist schon alles.

Zusammenfassung

- Jedes BASIC Programm besteht aus den Objekten und dem eigentlichen BASIC Code.
- Die Objekte sind in einer Parent-Child-Struktur miteinander verbunden.
- Man benötigt auf jeden Fall ein Application-Objekt und ein Primary-Objekt.
- Der Programmcode wird in Form von Actionhandlern realisiert.
- Actionhandler werden aufgerufen, wenn der Nutzer eine Aktion auslöst.

Tipps

- Programmieren lernt man nicht durch Lesen, sondern durch Programmieren. Spielen Sie ruhig an den Programmen herum. Ändern Sie etwas, lassen Sie Anweisungen weg und sehen Sie, welche Auswirkungen das hat.

- Wenn Sie sich mit einem neuen Problem beschäftigen, schauen Sie in die Beispiele. R-BASIC liefert zu nahezu allen Befehlen, Instancevariablen und Objekten Programmbeispiele mit.

- Es ist völlig normal, dass Sie beim Lesen der Dokumentation vieles nicht auf den ersten Blick verstehen. PC/GEOS Objekte sind wunderbar intelligent und vielseitig einsetzbar. Entsprechend kompliziert sind einige Stellen der "Bedienungsanleitung" (Dokumentation). Sie brauchen fürs Erste nur das zu verstehen, was sie gerade benutzen wollen.

- Suchen Sie sich für den Anfang einfache, überschaubare Projekte. Viele Programmierer haben mit einem Lottozahlengenerator oder etwas Ähnlichem angefangen.

Beherzigen Sie nach Möglichkeit die folgenden Ratschläge:

- Machen Sie sich im Vorfeld eine Plan, was das Programm genau können soll und was es nicht können soll.
- Verwenden Sie aussagekräftige Namen für Variablen, Objekte und Routinen. Ein Textobjekt, in das der Nutzer etwas eingeben soll, könnte "EingabeText" heißen. Nennt man es nur "Eingabe" könnte man später in Zweifel kommen, ob man das Textobjekt oder den eigegebenen Text meint.
- Verwenden Sie großzügig Kommentare. Kommentieren Sie vor allem die Ideen hinter einem Programmabschnitt. Das Notieren einer Idee hilft Ihnen, das Problem gut zu durchdenken und verringert so die Fehlerquote.

2 Die BASIC Programmiersprache

Die R-BASIC Programmiersprache baut auf einer einfachen BASIC Syntax auf. Diese ist, wenn man den grundlegenden Elementen der englischen Sprache mächtig ist, weitgehend selbsterklärend. Dabei geht R-BASIC weit über klassische BASIC Programmiersprachen hinaus. Es unterstützt nicht nur das GEOS Objektsystem sondern auch die typischen GEOS-Eigenschaften und System-dienste wie lange Dateinamen, Tokens (Icons), die Zwischenablage, Hilfedateien, Timer und mehr.

Beachten Sie, dass die Groß- und Kleinschreibung von Kommandos, Variablen, Objekten usw. unter R-BASIC keine Rolle spielt.

Ziel des Programmierhandbuchs - dem Handbuch, dass Sie gerade lesen - ist es, die Elemente der BASIC-Programmiersprache darzustellen. Dabei wird nicht jedes Mal auf die notwendigen Objekte eingegangen. Diese werden im Objekthandbuch beschrieben. Um die Beispiele und Codefragmente aus den folgenden Kapiteln auszuprobieren haben Sie zwei Möglichkeiten.

Möglichkeit 1 (empfohlen):
Öffnen Sie das Beispielprogramm "Hallo 1" aus dem Ordner "R-BASIC\ Beispiele\Erste Schritte" und speichern Sie es unter einem neuen Namen. Die Beispielprogramme enthalten bereits alle notwendigen Objekte. Ersetzen Sie dann im Fenster "BASIC Code" den Code des Beispiels durch den Code, den Sie ausprobieren möchten.

```
BUTTONACTION DemoHandler
  ... hier neuen Code einfügen
End ACTION
```

DemoAction ist der Name der Routine (des sogenannten Actionhandlers), die aufgerufen wird, wenn Sie auf den Button "Beispiel starten" klicken. Sie können den Zusammenhang zwischen dem Button und dem Actionhandler im Fenster "UI-Code" nachvollziehen, wenn Sie ganz nach unten scrollen.

Möglichkeit 2:
Öffnen Sie ein neues, leeres BASIC Programm. Schreiben Sie im Fenster "BASIC Code" die Anweisung **ClassicCode** und dann den Code, den Sie ausprobieren möchten.

```
ClassicCode
  ... hier neuen Code einfügen
```

Die Anweisung ClassicCode versetzt R-BASIC in den sogenannten "klassischen" Modus. In diesem Modus stellt R-BASIC ein paar Objekte bereit, die sich so verhalten, als hätten Sie einen altmodischen Homecomputer vor sich. Sie können direkt Text und Grafik auf den Bildschirm ausgeben, ohne sich um den Objekthintergrund kümmern zu müssen. Details dazu finden Sie im Kapitel 2.13.1 (Der klassische BASIC Modus).

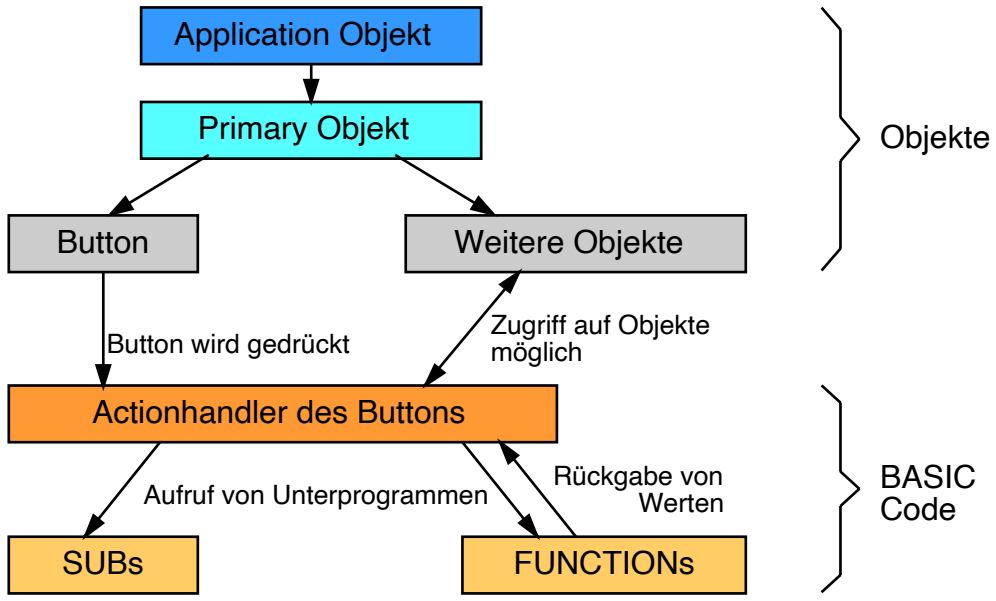
2.1 Grundlegende Konzepte

Hier finden Sie eine Zusammenstellung wissenswerter und grundlegender Fakten zur BASIC-Sprache. Eine ausführliche Beschreibung der Zusammenhänge finden Sie in den folgenden Kapiteln.

2.1.1 Die Struktur eines BASIC Programms

Jedes R-BASIC-Programm besteht im Wesentlichen aus zwei Teilen: den Objekten und dem eigentlichen Programmcode.

Die Objekte sind sie sichtbaren Elemente des Programms. Sie werden im UI-Code Fenster vereinbart. UI steht für User-Interface. Die Objekte heißen daher auch UI-Objekte. Alle Objekte sind miteinander verbunden. Ein "Parent" Objekt hat ein oder mehrere "Children". Jedes der Children kann wieder Parent für weitere Objekte sein. Diese Struktur nennt man einen Objekt-Baum (object tree).
Das Top-Objekt jedes Programms ist ein Application-Objekt. Dessen Child ist ein Primary-Objekt, das Hauptfenster des Programms. Dieses enthält wiederum die anderen sichtbaren Objekte des Programms: Buttons, Listen, Objekte zur Grafikausgabe und so weiter.
R-BASIC stellt die allermeisten der unter PC/GEOS verfügbaren Objekte bereit. Die Anwendung der Objekte wird dem Programmierer dabei so einfach wie möglich gemacht. Eine ausführliche Beschreibung der Objekte finden Sie im Objekt-Handbuch.



Solange nichts passiert ist das Programm im Wartezustand. Wenn der Nutzer nun zum Beispiel einen Button anklickt, so ruft dieser seinen Actionhandler auf. Dieser Handler kann wiederum andere Subs oder Functions aufrufen. Außerdem kann er auf die Daten der anderen Objekte des Programms zugreifen, diese lesen oder verändern. Natürlich kann er auch Grafiken ausgeben oder auf Dateien zugreifen.

Auf diese Weise wird die Funktionalität des Programms implementiert. Nachdem der Code des Actionhandlers abgearbeitet ist geht das Programm wieder in den Wartezustand über. ... Bis der Nutzer die nächste Aktion macht und der nächste Actionhandler aufgerufen wird.

Zahlen

Zahlen können in folgender Weise dargestellt werden:

- Einfache Zahlen, z.B. 12 oder 4.8. Dezimaltrennzeichen ist immer der Punkt.
- Darstellung mit Zehnerpotenzen z.B. 1.8E4 (= $1{,}8 \cdot 10^4$ = 18000)
- Binärdarstellung, z.B. &B1001 (= 9)
- Hexadezimale Darstellung z.B. &HFF (= 255)
- Zahlen dürfen keine Leerzeichen enthalten.

Eine ausführliche Behandlung des Themas finden Sie im Kapitel 2.3 (Arbeit mit numerischen Ausdrücken)

Strings (Zeichenketten)

- Zeichenketten werden in BASIC als "Strings" bezeichnet
- Strings werden immer in Anführungszeichen gesetzt, z.B. "Im Haus des Donners"
- Innerhalb von Strings können Sonderzeichen vorkommen. Sie werden mit einem Backslash "\" eingeleitet. Zum Beispiel eröffnet "\r" eine neue Zeile und "\200" fügt das Zeichen mit dem ASCII-Code 200 ein.

Eine ausführliche Behandlung des Themas finden Sie im Kapitel 2.5 (Arbeit mit Strings)

Trennzeichen

- Erfordert ein Befehl mehrere Parameter, müssen diese durch Komma ’,’ getrennt werden.
- Hinter jedem Bezeichner (Variablennamen und dgl.) muss ein Zeichen folgen, dass nicht Teil eines Bezeichners sein kann. Üblicher Weise sind das ein Leerzeichen, ein Komma ’,’ oder eine öffnende Klammer ’(’. R-BASIC kann sonst nicht erkennen dass DIMA eigentlich DIM A heißen soll.
- Leerzeichen oder Tabulatoren sind überall erlaubt, außer innerhalb von Bezeichnern und innerhalb von Zahlen.

Kommentare

Kommentare und Leerzeilen dienen der optischen Strukturierung und der Erläuterung des Programmcodes. Kommentare und Leerzeilen werden beim Compilieren ignoriert, d.h. sie verlängern das Programm nicht, es wird dadurch auch nicht langsamer. Weiter unten finden Sie einen Exkurs zur Verwendung von Kommentaren und zur optischen Strukturierung von Programmen.

Klammern

Klammern haben in R-BASIC zwei Funktionen:
1. Strukturieren von mathematischen Ausdrücken. Ausdrücke, die in Klammern gesetzt sind, haben immer Vorrang.
2. Einschließen von Funktionsargumenten.
   Einige BASIC-Befehle erfordern Klammern, andere nicht. Das Konzept dahinter ist sehr einfach:

- Befehle, die am Anfang einer Zeile stehen, erfordern keine Klammern.
  Es ist aber erlaubt, die Befehlsparameter in Klammern zu setzen
  Beispiel:

```
LINE  20, 30, 100, 200
LINE ( 20, 30, 100, 200 )     ' Beides ist gleichwertig
```

- Befehle (genauer: Funktionen), die auf der rechten Seite einer Zuweisung stehen (können) erfordern in jedem Fall Klammern, damit R-BASIC weiß, wie es die Parameter zu behandeln hat.

Tipp für Umsteiger: In R-BASIC kann man statt der BASIC-üblichen runden Klammern ( ) auch die eckigen Klammern [ ] verwenden. Damit kann man, wie in anderen Programmiersprachen üblich, Feldindizes in eckige Klammern setzen, während man Funktionsargumente in runde Klammern setzt. R-BASIC unterscheidet jedoch beide Klammertypen nicht.

Exkurs: Setzen von Klammern

Sicher ist Ihnen aufgefallen, dass einige BASIC-Befehle Klammern erfordern, andere nicht. Das Konzept dahinter ist sehr einfach:
Befehle, die am Anfang einer Zeile stehen, erfordern keine Klammern. z.B.

```
PRINT A, B, C$
```

Befehle (genauer: Funktionen), die auf der rechten Seite einer Zuweisung stehen (können), erfordern in jedem Fall Klammern, damit R-BASIC weiß, wie es die Parameter zu behandeln hat. Außerdem erhöht das die Lesbarkeit ungemein. Nehmen wir als Beispiel (SQR berechnet die Quadratwurzel):

```
DIM A, X
  A = 16
  X = SQR A • 4   ' <-- Falsch, Klammern fehlen
  PRINT X
```

Es ist etwas anderes, ob wir die Klammern so setzen:

```
X = SQR (A • 4)
```

oder so

```
X = SQR(A) • 4
```

Im ersten Fall erhalten wir 8, im zweiten 16.

Außerdem sollte Sie bei logischen Operatoren nicht mit Klammern sparen. Die Operatoren folgen einer bestimmten Hierarchie, so dass es schnell passiert, dass der Compiler etwas anderes versteht, als Sie ihm sagen wollten. Beispiel:

```
X = (7 OR 3) AND 1      ' liefert 1
X = 7 OR (3 AND 1)      ' liefert 7
```

Exkurs: Optische Strukturierung des Programms

REM

REM (Remark - Anmerkung) leitet einen Kommentar ein. Der Kommentartext erstreckt sich bis zum Ende der Zeile und kann beliebige Zeichen enthalten. Sie sollten Ihr Programm immer ausführlich kommentieren. Das erleichtert das Verständnis des eigenen Programms, wenn Sie es später noch einmal anschauen oder überarbeiten.

- REM kann durch eine Apostroph ’ oder einem Ausrufezeichen ! abgekürzt werden.
- Vor REM (bzw. ’ oder !) kann man den Doppelpunkt weglassen.
- Kommentare verlangsamen den Ablauf des Programms nicht! Der Compiler ignoriert alle Kommentare, während er das Programm übersetzt.

Der Doppelpunkt :

Der Doppelpunkt ’:’ trennt mehrere Anweisungen in einer Zeile.

Syntax: Anweisung1 : Anweisung2

Beispiel:

```
COLOR 7,0 : CLS
```

Tipps:

- Sie sollten eine häufige Verwendung des Doppelpunktes vermeiden, da er die Übersichtlichkeit des Programms negativ beeinflussen kann.
- Anweisungsfolgen, die mit einem Doppelpunkt getrennt wurden, laufen geringfügig schneller ab, als wenn sie jede in einer eigenen Zeile stehen. Der Unterschied ist jedoch sehr gering.
- Vor einem Kommentar ist kein Doppelpunkt erforderlich.
- Ein Doppelpunkt am Zeilenende ist zulässig und wird von R-BASIC ignoriert.

Leerzeichen und Tabulatoren

Es ist dringend zu empfehlen, ein Programm optisch zu strukturieren. Dazu eignen sich insbesondere Leerzeilen und Einrückungen. Die optische Strukturierung verbessert die Lesbarkeit und signalisiert die Struktur des Programms. Die Abarbeitungsgeschwindigkeit wird nicht beeinflusst.

Das folgende Beispiel gibt die ersten Quadratzahlen aus. Dabei wird jede durch 3 teilbare Zahl rot dargestellt.

```
' Unstrukturiert
DIM n, z

COLOR 15, 0
CLS
Print "Ausgabe der Quadratzahlen"
FOR n = 1 TO 15
z = n*n
IF z/3 = Int (z/3) THEN
Ink 12
ELSE
Ink 15
End IF
Print n, z
NEXT n
PRINT "Fertig"
```

```
' Strukturiert mit Einrückungen, Kommentaren und Leerzeilen
DIM n, z

  COLOR 15, 0                 ' Weiß auf Schwarz
  CLS
  Print "Ausgabe der Quadratzahlen"

  FOR n = 1 TO 15
    z = n*n
    IF z/3 = Int (z/3) THEN   ' Wenn durch 3 teilbar
        Ink 12                ' Vordergrund rot
    ELSE
        Ink 15                ' ansonsten: Vordergrund weiß
    End IF

    Print n, z                ' Ausgabe, tabuliert (wegen Komma)
  NEXT n

  PRINT "Fertig"
```

Verwendung der Code-Fenster

Wenn Sie ein Programm schreiben stehen Ihnen bis zu 6 Code-Fenster zur Verfügung.

UI-Objekte
: In diesem Fenster müssen die UI-Objekte des Programms vereinbart werden. Das Schreiben von Code in diesem Fenster ist nicht möglich.

DIM & DATA
: Bei umfangreichen Projekten sollten hier global Deklarationen untergebracht werden. Das Schreiben von Code ist möglich, aber ganz schlechter Stil.

Exports
: Wenn Sie eine Library schreiben werden hier Deklarationen untergebracht, die von der Library "exportiert" werden. Libraries sind im Kapitel 2.12 beschrieben.

BASIC-Code, Tools, Init-Code
: Diese Fenster sind für den eigentlichen Programmcode vorgesehen. Aus Sicht von R-BASIC sind diese Codefenster alle gleichwertig. Einem erfahrenen Programmierer ermöglichen sie, seinen Code übersichtlicher zu gestalten. Programmieranfängern wird empfohlen zunächst nur das Fenster BASIC-Code zu benutzen.

Die Fenster werden in der Reihenfolge "Exports" -> "DIM & DATA" -> "UI-Objekte" -> "Tools" -> "BASIC-Code" -> "Init" compiliert. Dadurch stehen den drei eigentlichen Codefenstern sowohl die Vereinbarungen aus Exports und DIM & DATA als auch die Namen aller UI-Objekte zur Verfügung.

Tipps:

- Die Codefenster können auch mit den Tastenkombinationen Strg-1 bis Strg-6 angewählt werden.
- Im Menü "Optionen" -> "Editor-Einstellungen" finden Sie den Menüpunkt "Code-Windows umbenennen". Dort können Sie den Windows "DIM & DATA", "BASIC-Code", "Tools", und "Init" andere, an ihr aktuelles Projekt angepasste, Namen geben.

2.1.2 Begriffe und Fakten

Im Folgenden werden ein paar Begriffe erläutert, die in den Handbüchern immer wieder vorkommen. Sie müssen diese Definitionen nicht auswendig lernen, aber um die Handbücher zu verstehen und sich mit anderen Programmierern verständigen zu können sollten Sie in etwa wissen, was sie bedeuten.

Ausdruck
- Alle Berechnungen oder Formeln zur Ermittlung eines Wertes werden als Ausdruck bezeichnet.
- Am wichtigsten sind numerische Ausdrücke (das Ergebnis ist eine Zahl), es gibt aber auch String-Ausdrücke (das Ergebnis ist ein Text), Objekt-Ausdrücke, Handle-Ausdrücke usw.
- Ausdrücke stehen häufig - aber nicht ausschließlich - auf der rechten Seite einer Zuweisung. Beispiel dafür:

```
y = 7
z = Int(y) + 5
st$ = "Hallo Welt"
```

- Siehe auch: Funktion, Parameter, Zuweisung

Actionhandler
- ActionHandler sind spezielle Unterprogramme, die automatisch aufgerufen werden, wenn der Nutzer ein "Ereignis" auslöst, z.B. auf einen Button klickt.
- Siehe auch: Funktion, Parameter, Routine, Sub

Anweisung
- Als Anweisung wird eine einzelne Codezeile bezeichnet. Eine Anweisung geht bis zum Zeilenende oder bis zu einem Doppelpunkt. Dann stehen mehrere Anweisungen in einer Zeile.
- Der Begriff Anweisung wird häufig verwendet, wenn man nicht explizit angeben kann oder will, ob es sich um eine Deklaration, eine Zuweisung oder eine UI-Anweisung handelt.
- Kommentar- und Leerzeilen werden nicht als Anweisungen bezeichnet.
- Kontroll-Anweisungen (z.B. FOR-TO-NEXT) stellen einen Spezialfall dar. Sie beeinflussen den Programmablauf.
- Vergleiche auch: Befehl, Deklaration, Parameter, UI-Anweisung, Zuweisung

Befehl
- Ein Befehl wird im Programm aufgerufen um eine bestimmte Aufgabe zu erledigen, z.B. LINE, Print, CLS, EXIT oder FontFind.
- Befehle werden oft von Parametern gefolgt. Beispiel:

```
      LINE 10, 20, 100, 200
```

- Sie auch: Anweisung, Deklaration, Zuweisung, Parameter

Bezeichner

- Alle Namen für Variablen, Objekte, Strukturen usw. heißen "Bezeichner". Sie dürfen bis zu 32 Zeichen lang sein.
- Am Anfang eines Bezeichners steht immer ein Buchstabe. Zulässige Zeichen sind weiterhin: die Ziffern (0..9), der Unterstrich '_' und das Dollar-Zeichen '$'.
- R-BASIC unterscheidet nicht zwischen Groß- und Kleinschreibung. Es ist egal, ob sie CLS, cls oder Cls schreiben. Der Editor erkennt R-BASIC Befehle und hebt sie hervor. Dabei wird, wie in den Handbüchern auch, oftmals eine kombinierte Groß-Kleinschreibung verwendet, z.B. FillEllipse statt FILLELLIPSE oder fillellipse. BASIC Schlüsselworte wie DIM, IF, THEN oder FOR werden groß geschrieben.
- Bezeichner dürfen keine Leerzeichen enthalten.

Deklaration

- Deklarationen sind "Vereinbarungen". Zum Beispiel vereinbart

```
DIM A as Real
```

  eine Variable mit dem Namen A, die eine Real-Zahl speichern kann.
- Weitere Beispiele für Deklarationen sind

```
DECL   SUB   MaleBild ( )
CONST  y_0 = 12
```

- Eine Deklaration erzeugt noch keinen ausführbaren Code. Sie zeigt nur dem Compiler an, dass der vereinbarte Bezeichner existiert und welche Eigenschaften er hat.
- Es wird empfohlen, für globale Deklarationen nur das Dim&Data-Fenster zu nutzen.
- Vergleiche auch: Anweisung, UI-Anweisung, Zuweisung

Ereignis

- Jeder Vorgang, der eine Reaktion des Programms erfordert, wird als Ereignis bezeichnet.
- Ereignisse sind zum Beispiel:
  * Das Betätigen einer Taste auf der Tastatur
  * Das Anklicken eines Buttons
  * Das Auswählen eines Eintrags aus einer Liste
- Ereignisse werden vom GEOS-System an das zuständige Objekt weitergeleitet. Dieses behandelt das Ereignis dann intern oder es ruft den passenden BASIC-Handler auf, damit das R-BASIC Programm das Ereignis behandeln kann.

Funktion

- Funktionen sind Unterprogramme, die einen Wert zurückgeben.
- Es gibt BASIC-interne Funktionen (z.B. Int( )) und selbst definierte Funktionen (Schlüsselwort FUNCTION).
- Beim Aufruf einer Funktion müssen Klammern angegeben werden, auch wenn die Funktion keine Parameter hat.
- Siehe auch: Actionhandler, Parameter, Routine, Sub

Hint
- Hints (=Hilfen) sind spezielle Instancevariablen, die einem Objekt mitteilen, wie es sich zu verhalten oder darzustellen hat.
- Hints können von Objekten ignoriert werden, wenn es die Situation erfordert.

Instancevariable
- Die Instancevariablen enthalten die Eigenschaften der einzelnen Objekte. So hat jeder Button z.B. eine Aufschrift, die aber von Button zu Button verschieden ist.
- Instancevariablen werden im Objekt-Handbuch bei den zugehörigen Objekten besprochen.
- Vergleiche auch: Methode, Objekt, UI-Anweisung

Kommando
- Siehe Befehl, Anweisung

Konstante
- Eine Konstante ist ein symbolischer Name für einen festen, d.h. während des Programmablaufs nicht veränderbaren, Wert.
- In R-BASIC sind über einhundert numerischen Konstanten definiert (d.h. sie stehen für eine Zahl). Besonders wichtig sind die Konstanten TRUE (Wert: –1) und FALSE (Wert: 0)
- Eigene Konstanten der Typen Real und String kann man mit der Anweisung CONST definieren.

Methode
- Methoden sind "Anweisungen an ein Objekt". Der Aufruf einer Methode führt dazu, dass das Objekt eine bestimmte Operation ausführt.
- Beispiel: Die Methode "Open" bringt eine Dialogbox auf den Schirm:

```
MyDialog.Open
```

- Vergleiche auch: Instancevariable, Objekt, UI-Anweisung

Objekt
- Die sichtbaren Elemente der grafischen Oberfläche werden in R-BASIC als Objekte bezeichnet. Das sind zum Beispiel ein Button, eine Liste oder eine Dialogbox.
- Die in R-BASIC verfügbaren Objekte werden im Objekt-Handbuch besprochen.
- Vergleiche auch: Instancevariable, Methode, UI-Anweisung

Parameter
- Parameter sind Werte, die an eine Routine oder einen Befehl übergeben werden. Felder können nicht als Parameter übergeben werden.
- Für Parameter sind alle in R-BASIC verfügbaren Typen zulässig.
- Siehe auch: Actionhandler, Funktion, Routine, Sub

Routine
- Die Begriffe Routine und Unterprogramm werden zusammenfassend für Sub, Function und ActionHandler verwendet. Man benutzt sie, wenn man nicht näher spezifizieren will oder kann, ob man ein Sub, eine Function oder einen ActionHandler meint.
- Siehe auch: Actionhandler, Funktion, Parameter, Sub

Sub
- Eine SUB (englisch für Subroutine, Unterprogramm) ist ein in sich geschlossener Programmteil, der eine bestimmte Aufgabe zu erledigen hat.
- SUB's können mehrfach aufgerufen werden und sie dienen der Strukturierung des Programms.
- Siehe auch: Actionhandler, Funktion, Parameter, Routine

UI-Anweisung
- UI-Anweisungen sind spezielle Anweisungen, mit denen Objekte vereinbart oder die Startwerte für Instancevariablen belegt werden.
- UI-Anweisungen können nur im UI-Code Fenster stehen.
- Vergleiche auch: Anweisung, Deklaration, Objekt, Instancevariable, Zuweisung

Unterprogramm
- Siehe: Routine

Variable
- Variablen dienen zum Speichern von Zahlen, Texten und anderen Daten. Ihr Inhalt kann verändert werde, d.h. er ist variabel. Daher kommt auch der Name.
- Auf lokale Variablen kann nur innerhalb der Routine, in der sie deklariert wurden, zugegriffen werden.
- Globale Variablen sind für alle Programmteile sichtbar.
- Variablen sind eines der grundlegenden Konzepte in einer Programmiersprache. Sie werden ausführlich im nächsten Kapitel (Variablen und Typen) erläutert.

Zuweisung
- Zuweisungen bestehen aus einer Variablen (das kann auch eine Objekt-Instancevariable sein), einem Gleichheitszeichen und auf der rechen Seite einem Ausdruck, dessen Wert der Variablen zugewiesen werden soll.

```
NAME$ = "Paul"
```

  Dadurch wird der Wert "Paul" in der Variablen NAME$ gespeichert.
- Der Ausdruck auf der rechten Seite muss kompatibel zum Variablentyp sein, d.h. Sie dürfen z.B. einer String-Variablen keine Zahl zuweisen. Natürlich gibt es entsprechende Umwandlungs-(Konvertierungs)-Funktionen.
- Vergleiche auch: Anweisung, Deklaration, UI-Anweisung

(Leerseite)

2.2 Variablen und Typen

2.2.1 Was sind Variablen?

Wenn Sie eine Zahl oder eine Text abspeichern und anschließend wieder darauf zugreifen wollen, benötigen Sie mindestens zwei Dinge: genug Speicherplatz um die Daten abzulegen und einen Namen, unter dem Sie wieder auf die Daten zugreifen können. Um den Speicherplatz kümmert sich R-BASIC automatisch, nur den Namen müssen Sie selbst vergeben. Unter diesem Namen können Sie später auch eine andere Zahl oder einen anderen Text in diesem Speicherplatz ablegen. Der Inhalt des Speicherplatzes ist also veränderlich, d.h. variabel. Man sagt daher, dass man es mit einer **Variablen** zu tun hat. Im Sprachgebrauch wird der Begriff "Variable" gleichbedeutend mit dem **Variablennamen** verwendet. Statt "Die Variable mit dem Namen X" sagt man "Die Variable X". Gemeint ist in beiden Fällen jedoch, dass sich hinter dem Namen X ein Speicherbereich verbirgt, der einen **Wert**, z.B. eine Zahl oder einen Text, enthält. Variablen, die eine Zahl enthalten, bezeichnet man als "numerische" Variablen. Variablen, die einen Text enthalten, bezeichnet man als "String" Variablen. Der englische Begriff "String" steht für Faden, Schnur bzw. Kette, in unserem Fall für eine Abfolge von Zeichen (= Buchstaben), eine "Zeichenkette".
Um eine Variable zu vereinbaren verwenden wir das Schlüsselwort **DIM**. Wir müssen außer dem Namen der Variablen auch angeben, welche Art von Daten sie enthalten soll, d.h. wir müssen ihren **Typ** (auch Datentyp genannt) angeben. Dazu dient das Schlüsselwort **AS**. Der Typ der Variablen beschreibt außer der Art der Daten auch den benötigten Speicherplatz, die "Größe" der Variablen. Das ist nicht zu verwechseln mit dem Wert der Variablen. Der "Wert" gibt den Inhalt der Variablen an (z.B. "Hallo"), die Größe den bereitgestellten Speicherplatz, z.B. 10 Byte.

Ein einfaches Beispiel:

```
DIM X AS REAL
  x = 12          ' Weise der Variablen X den Wert 12 zu
  Print x         ' 12 erscheint
  x = -7.9        ' Weise der Variablen X den Wert -7.9 zu
  Print x         ' -7.9 erscheint
```

DIM

Mit dem Schlüsselwort **DIM** werden Variablen vereinbart. DIM ist sehr mächtig und kann in verschiedenen Varianten verwendet werden, die im Folgenden erklärt werden:

Beispiel 1: Einfache Variablendefinition, automatische Typerkennung

```
DIM A, B
a = 5
b = 3.7
Print A*B
```

Es wurden zwei numerische Variablen vom Datentyp "Real" vereinbart. Der Variablen A wird der Wert 5, der Variablen B der Wert 3,7 zugewiesen. Beachten Sie, dass als Dezimaltrennzeichen **immer** ein Punkt zu schreiben ist. Außerdem sehen Sie, dass die Groß/Kleinschreibung der Variablennamen egal ist. Anschließend wird das Produkt (in unserem Fall 18.5) ausgegeben.

Beispiel 2: Einfache Variablendefinition, automatische Typerkennung

```
DIM A$, B$
A$ = "Hallo "
B$ = "Welt"
Print A$; B$
```

Hier wurden zwei Stringvariablen vereinbart. R-BASIC erkennt an dem nachgestellten Dollarzeichen ($), dass es sich um eine Zeichenkettenvariable (Stringvariable) handeln soll. Das Dollarzeichen ist Teil des Namens der Variablen und darf nicht weggelassen werden. R-BASIC unterscheidet die Variablen A$ (sprich: "A-String") von der Variablen A.
Der Variablen A$ wird der Wert "Hallo ", der Variablen B$ der Wert "Welt" zugeordnet. Die Print-Anweisung gibt den Text "Hallo Welt" aus.

Beispiel 3: Einfache Variablendefinition, automatische Typerkennung

```
DIM Auto, Bus$
```

Diese Anweisung vereinbart die Realvariable Auto und die Stringvariable Bus$.

Wie für alle Bezeichner in R-BASIC gilt auch für Variablen: Sie müssen mit einem Buchstaben beginnen, dürfen Buchstaben, Ziffern und die Sonderzeichen '$' (Dollar) und '_' (Unterstrich) enthalten und können bis zu 32 Zeichen lang sein. Groß- und Kleinschreibung wird nicht unterschieden.

DIM .... AS

Das Schlüsselwort AS legt den Typ der Variablen fest.

Beispiel 4: Numerische Variablen (siehe auch Kapitel 2.2.2)

```
DIM A, B     AS   WORD
DIM Hilf     AS   Longint
DIM ups      AS   REAL
```

Beispiel 5: Stringvariablen (siehe auch Kapitel 2.2.3)

```
DIM U$, V$   AS String
DIM Name$    As String(20)
```

Es ist in BASIC üblich, alle String Variablen durch ein Dollarzeichen am Ende zu kennzeichnen. Erzwungen wird dies in R-BASIC jedoch nicht.

Beispiel 6: Feldvariablen

```
DIM  X(10)   AS REAL
```

Ein Feld fasst Variablen gleichen Typs zusammen. Sie werden über den Feldindex angesprochen. Das Beispiel vereinbart 11 REAL variablen, X(0) (Sprich X-von-Null) bis X-von-10). Felder werden im Abschnitt 2.2.5 ausführlich besprochen.

Hier ist die komplette Liste der Syntaxvarianten von DIM:

Syntax: **DIM** **<VariablenListe>**
Es werden ausschließlich REAL und STRING Variablen vereinbart. Endet der Variablenname auf '$' (Dollar) wird eine Stringvariable vereinbart, ansonsten eine REAL Variable.
Beispiel: **DIM** A, B(12), C$, D$(10,10)

Syntax: **DIM** **<VariablenListe>** **AS** **<VariablenTyp>**
Es werden Variablen vom angegebenen Typ vereinbart.
Beispiel: **DIM** X, Y, Z, P(12), Q(4,4) AS Integer

Syntax: **DIM** **<VariablenListe>** **AS** **HUGE** **<VariablenTyp>**
alternativ: **HUGE** **<VariablenListe>** **AS** **<VariablenTyp>**
Das Schlüsselwort HUGE (=riesig) ist nur im Zusammenhang mit Feldern zulässig. Es zeigt an, dass die Felder sehr groß sind und daher nicht vollständig im Speicher gehalten werden können.
Beispiel: **DIM** F(999, 999) AS **HUGE** REAL
Es wird ein Feld mit 1000 x 1000 = 1 000 000 Elementen vereinbart. Jedes Element erfordert 10 Byte, das Feld insgesamt 10 Megabyte.

Dabei bedeuten:

<VariablenListe> Namen der zu vereinbarenden Variablen.
Einfache Variablen und Feldvariablen sind erlaubt.

<VariablenTyp> Typ der vereinbarten Variablen. Es sind alle R-BASIC bekannten Datentypen erlaubt: Die Standarddatentypen, R-BASIC Strukturtypen und mit dem Schlüsselwort STRUCT selbst definierte Datentypen.

2.2.2 Numerische Datentypen und numerische Ausdrücke

Die Verarbeitung von Zahlen und mathematischen Funktionen gehört zu den Kernaufgaben einer Programmiersprache. Numerische Variablen speichern Zahlen. In R-BASIC stehen die folgenden Typen zur Verfügung:

Typ	Speicherbedarf	Inhalt
REAL	10 Byte	Reelle Zahl, mit Vorzeichen $\pm 3.9999 \cdot 10^{4931}$, 19 Stellen
BYTE	1 Byte	vorzeichenlose Zahl, 0 .. 255
WORD	2 Byte	vorzeichenlose Zahl, 0 .. 65535
INTEGER	2 Byte	Ganze Zahl, vorzeichenbehaftet – 32768 ... 32767
DWORD	4 Byte	vorzeichenlose Zahl 0 ... 4 294 967 295
LONGINT	4 Byte	Ganze Zahl, vorzeichenbehaftet – 2 147 483 648 ... 2 147 483 647
WWFIXED	4 Byte	Dezimalzahl, vorzeichenbehaftet maximal 4 Nachkommastellen – 32768.0 ... 32767.9999

Realvariablen

Realvariablen (real = reelle Zahlen) speichern Zahlen im Bereich von $\pm 3.9999 \cdot 10^{4931}$ mit einer Genauigkeit von 19 Stellen. Zahlen mit negativen Exponenten sind bis $\pm 10^{-4931}$ erlaubt.

Byte, Word, Integer, DWord und Longint Variablen

Für Einsteiger in die Programmierung ist der Zahlentyp REAL völlig ausreichend. Fortgeschrittene Programmierer würden jedoch bald die in der Computertechnik üblichen "kleinen" Zahlentypen vermissen. R-BASIC unterstützt daher zusätzlich die in der Tabelle angegebenen Typen. Sie benötigen auch wesentlich weniger Speicherplatz als Realvariablen.

WWFixed Variablen

Berechnungen mit dem Datentyp WWFixed werden merklich schneller ausgeführt als mit den anderen numerischen Datentypen. Die begrenzte Genauigkeit von 4 Nachkommastellen ist für viele Anwendungszwecke, insbesondere bei der Ausgabe von Grafik, völlig ausreichend. Damit R-BASIC mit WWFixed-Werten rechnet sind besondere Regeln einzuhalten. Diese sind im Kapitel 2.3.6 (Schnelle Mathematik mit WWFixed) beschrieben. Sie sollten dieses Kapitel unbedingt lesen, bevor Sie den Datentyp WWFixed verwenden!

Beispiele:

```
DIM     a, b, c  AS  REAL
DIM     p, q, r  AS  Word
DIM     u, v, w  AS  Integer
```

Hinweise:

- Mit der Ausnahme von WWFixed rechnet R-BASIC grundsätzlich mit REAL-Zahlen. Die anderen numerischen Typen werden daher vor jeder Benutzung in REAL umgerechnet. Es ist in R-BASIC ein Irrtum anzunehmen, dass die Verwendung der Datentypen Word oder Integer die Rechengeschwindigkeit vergrößert.
- Als Dezimaltrennzeichen muss immer ein Punkt verwendet werden.
- Die "kleinen" Zahlentypen können grundsätzlich überall dort verwendet werden, wo auch Realvariablen zulässig sind.
- Bei Zuweisung von Werten außerhalb des Wertebereichs werden die überzähligen Bits einfach ignoriert. Das bedeutet:
  - Bei vorzeichenbehaftete Typen (INTEGER, LONGINT sowie WWFixed) werden zu große Zahlen zu negativen Zhalen und umgekehrt.
  - Vorzeichenlose Typen (BYTE, WORD, DWord) arbeiten mit einem Übertrag. Das entspricht einer Modulo-Operation.
  - Real-Zahlen speichern einen speziellen Fehlerwert.

Beispiele für Zuweisungen außerhalb des Wertebereichs:

```
DIM x, y     ' REAL Variablen, weil kein Typ vorgegeben
DIM b   AS   Byte
DIM i   AS   Integer
DIM w   AS   Word

   b = 12.3      ' Der Wert 12 wird gespeichert
   b = 300       ' Der Wert 44 wird gespeichert (300 = 256 + 44
                 ' bzw. 300 MOD 256 = 44
   i = 98766     ' Gespeichert wird 32767
                 ' Das ist der größtmögliche Integerwert
   w = 98766     ' Gespeichert wird 33230
                 ' weil 98566 = 65536 + 33230
                 ' bzw. 98766 MOD 65536 = 33230
```

Für Zahlen gelten die folgenden Regeln

- Einfache Zahlen sind z.B. 12 oder 4.89
  Als Dezimaltrenner wird immer der Punkt '.' verwendet, egal was Sie in den PC/GEOS Voreinstellungen festgelegt haben. Dadurch kann man BASIC-Programme auf allen PC/GEOS-Rechnern sofort laufen lassen.

- Vor jede Zahl darf ein Vorzeichen (+ oder –) gesetzt werden.

- Für Zahlen mit 10er-Potenzen wird das E (oder e) verwendet.
  $-3{,}78 \cdot 10^{12}$ wird also so geschrieben: **–3.78E12**
  $6{,}673 \cdot 10^{-11}$ sieht so aus: **6.673E-11**
  Entsprechendes gilt auch für die Ausgabe von Zahlen durch R-BASIC.

- Leerzeichen innerhalb von Zahlen sind unzulässig.

- Zahlen können auch in binärer Schreibweise (Vorsatz **&B**, z.B. 5 als **&B101**) und in hexadezimaler Schreibweise (Vorsatz **&H**, z.B. 243 als **&HF3**) dargestellt werden. In diesen Fällen sind 32 Bit oder 8 Hexadezimalstellen zulässig (Zahlenbereich DWord). Zahlen in dieser Schreibweise werden grundsätzlich als positive Zahlen behandelt.

Tabelle der Rechenoperationen

Operator	Bedeutung
+, -, *, /	Grundrechenarten: Addition, Subtraktion, Multiplikation, Division
^	Exponentialoperator, z.B. 3^2 = 9
MOD	Modulo-Division z.B. 8 MOD 3 = 2
AND, OR, XOR, NOT	Bitweise logische Operationen

Mathematische Funktionen

- R-BASIC verfügt über eine Vielzahl von mathematischen Funktionen. Diese können beliebig verknüpft werden, wobei R-BASIC die üblichen Vorrangregeln (Punktrechnung vor Strichrechnung, Klammern gehen vor usw.) beachtet.

Beispiele:

```
y = 4*sin(5*x) + 7
y = sqr( 1 + tan(z))
```

- Überall dort, wo in den Beispielen Zahlen oder numerische Variablen verwendet wurden, können auch komplexe numerische Ausdrücke stehen.

2.2.3 Stringtypen und Stringausdrücke

Zeichenketten wie z.B. "Hallo Welt" werden in BASIC als Strings bezeichnet. Stringvariablen speichern Zeichenketten (Texte) und sind in BASIC üblicherweise durch ein angehängtes $-Zeichen markiert. Das $-Zeichen ist Teil des Namens und darf nicht weggelassen werden. Es gibt zwei Zeichenkettentypen: STRING und STRING(n).

Variablen, die als **STRING** vereinbart wurden, können bis zu 128 Zeichen aufnehmen. Variablen vom Typ **STRING(n)** können bis zu **n** Zeichen lang sein. Dabei gilt: n <= 1024.

Beispiele:

```
DIM T$        ' STRING Variablen, weil der Name auf $ endet
              ' Max. 128 Zeichen
DIM  R$, S$    AS STRING         ' Max. 128 Zeichen
DIM  P$, Q$    AS STRING(100)    ' Max. 100 Zeichen
DIM  U$, V$    AS STRING(1024)   ' Max. 1024 Zeichen
                                 ' 1024 ist die obere Grenze
DIM  A, B      AS STRING         ' Auch Stringvariablen, obwohl
                                 ' der Name nicht auf $ endet
```

Für Strings gelten die folgenden Regeln:

- Stringkonstanten werden immer in Anführungszeichen gesetzt. Umlaute und Sonderzeichen sind erlaubt.

```
A$ = "Im Haus des Donners wird mir übel."
```

- Stringvariablen sollten immer durch ein $ (Dollarzeichen) am Ende gekennzeichnet sein. z.B. A$ (sprich A-String). Verstöße gegen diese Regel verschlechtern die Lesbarkeit des Programms.

- Strings können durch den Operator + zusammengefügt werden. Klammern sind erlaubt, aber nicht nötig.

```
A$ = "Im Haus des Donners "
A$ = A$ + "wird mir übel."
```

- Ein Anführungszeichen oder andere besondere Zeichen innerhalb einer Zeichenkette werden mit einem "Rückwärtsstrich" (Backslash) '\' eingeleitet. Beispiel: Die Anweisung

```
A$ = "Im \"Haus des Donners\"!"
```

Speichert den Text: **Im "Haus des Donners"!** in der Variablen A$

- Weitere Zeichen, die mit einem Backslash eingeleitet werden:
  \t Tabulator
  \r oder \n Zeilenschaltung
  \\ Backslash

- Ein Backslash gefolgt von bis zu drei Ziffern (Dezimalzahl) kann verwendet werden, um ASCII-Zeichen in Strings einzufügen, die nicht direkt auf der Tastatur verfügbar sind.

```
A$ = "a \195\180 b"
```

Speichert den Text **a √¥ b** in der Variablen A$
ASCII-Codes unterhalb von 32 (nicht druckbare Codes) sind ebenso zulässig. Die Null beendet eine String, darauffolgende Zeichen werden ignoriert.

- Mit der Anweisung CONST können Stringkonstanten definiert werden. Die Stringkennung $ am Ende ist dabei optional.
  Beispiel:

```
CONST NAME$ = "Müller"
CONST FULL_NAME$ = "Paul " + NAME$
Print "Mein Name ist "; NAME$; ", "; FULL_NAME$
```

2.2.4 Weitere Datentypen

In diesem Abschnitt werden Datentypen beschrieben, die für spezielle Zwecke, in denen Zahlen oder Strings nicht geeignet sind, eingesetzt werden. Die Beispiele enthalten daher gelegentlich Befehle, die erst in den zugehörigen thematischen Kapiteln beschrieben werden. Als Einsteiger werden Sie dieses Kapitel sicher nur überfliegen.

Tabelle der hier beschriebenen Standarddatentypen

Typ	Speicherbedarf	Inhalt
File	6 Byte	Referenz auf eine Datei
Object	8 Byte	Referenz auf ein UI-Objekt
Handle	6 Byte	Referenz auf eine R-BASIC interne Datenstruktur, die nicht bereits von FILE oder OBJECT abgedeckt ist.

Variablen der hier beschriebenen Typen können behandelt werden wie alle anderen Variablen in R-BASIC auch. Man kann z.B. Felder anlegen (z.B. DIM Dateien(10) AS FILE), sie als Elemente von Strukturen verwenden, als Parameter an SUB's oder FUNCTION's übergeben oder als Rückgabetyp von FUNCTION's benutzen.

2.2.4.1 Der Datentyp FILE

Wenn Sie mit einer Datei arbeiten wollen, benötigen Sie eine **Referenz** auf diese Datei. Für diesen Zweck gibt es Variablen vom Typ **FILE** (Dateivariablen). Alle Dateibefehle erwarten eine solche Variable, um zu wissen, welche Datei gemeint ist. Um die Details, z.B. wo die Datei auf der Platte zu finden ist und wie groß sie aktuell ist, kümmert sich dabei das GEOS-System. Eine ausführliche Beschreibung der Arbeit mit Dateien finden Sie im Kapitel 6 des Handbuchs "Spezielle Themen".

Dateivariablen müssen explizit **AS FILE** deklariert werden:

```
DIM   fh   AS  FILE
```

Beim Öffnen / Anlegen einer Datei wird der Variablen ein Wert zugewiesen. Die Struktur dieses Wertes ist BASIC-intern und besteht aus mehreren Zahlen.

```
fh =  FileOpen "info.txt"
```

Zum Arbeiten mit der Datei verwenden Sie die Variable, z.B.

```
x = FileRead ( fh )
FileWrite  fh, x
```

Wenn Sie fertig sind, müssen Sie die Datei schließen:

```
FileClose  fh
```

Danach enthält die Variable immer noch einen Wert, er ist jetzt jedoch ungültig. Deswegen sollten Sie, wenn es nicht zu sehr auf Geschwindigkeit ankommt, die Variable mit Hilfe der Funktion NullFile() einen "Null-Wert" zuweisen, sie also löschen. Dieser Schritt ist jedoch nicht zwingend erforderlich.

```
fh = NullFile()
```

NullFile

NullFile( ) ist eine Funktion, die eine "leere" Dateivariable liefert, d.h. sie dient zum Löschen einer Filevariablen. **Achtung! NullFile( )** überschreibt die Referenz in der Filevariablen mit Nullen, ohne die Datei zu schließen! Das muss vorher passiert sein. Sie können NullFile() auch benutzen, um zu prüfen, ob eine Dateivariable leer ist.

Syntax: **<fVar>** = **NullFile**( )
Die Klammern sind erforderlich, weil NullFile eine Funktion ist.
<fVar>: Variable vom Typ FILE

Hinweise:

- Es ist nicht zwingend erforderlich eine freigegebene Dateivariable auch mit **NullFile( )** zu löschen. R-BASIC kann jedoch eine gelöschte Filevariable erkennen und so eventuell einen Systemabsturz verhindern.

FileInfo$

Die Funktion **FileInfo$** liefert einen Text, der interne Informationen über eine Dateivariable liefert. Sie können diese Funktion zur Fehlersuche einsetzen.

Syntax: **<stringVar>** = **FileInfo$**(**<fileExpr>**)
<fileExpr>: Variable oder Ausdruck vom Typ FILE
<stringVar>: Stringvariable

2.2.4.2 Der Datentyp HANDLE

Ein "Handle" (= Anfasser) enthält eine **Referenz** auf eine Datenstruktur, deren Inhalt von R-BASIC verwaltet wird. Wo die Daten abgelegt sind, wie sie intern organisiert sind usw. braucht den R-BASIC Programmierer nicht zu kümmern, er greift immer **indirekt** über das **Handle** auf die Daten zu.

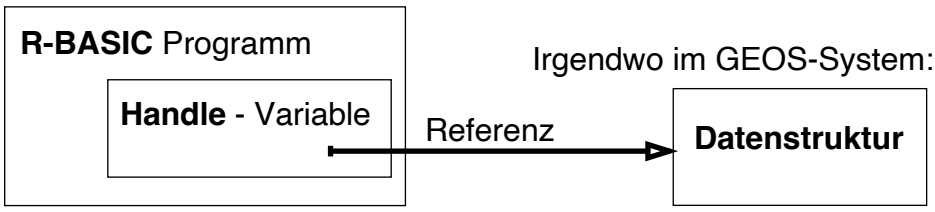


Bild: Eine Handlevariable enthält eine Referenz auf eine Datenstruktur

Üblicherweise gibt es einen BASIC-Befehl, der ein Handle **anlegt** (d.h. die dahinterstehenden Datenstrukturen initialisiert), einen oder mehrere Befehle, die das Handle **benutzen** (d.h. auf die dahinter stehenden Datenstrukturen zugreifen) und einen Befehl, der das Handle wieder **freigibt**. Der letzte Schritt ist sehr wichtig und sollte nicht vergessen werden, da hinter einem Handle oft Speicherblöcke stehen - und Speicher ist bekanntlich knapp unter GEOS.

*Der Datentyp **HANDLE** ermöglicht es R-BASIC auf sehr einfache Weise mit extrem komplexen Datenstrukturen umzugehen. Der Programmierer muss sich nicht mit den Interna dieser Datenstrukturen herumschlagen.*

Beispielsweise legt der Befehl **FileFindFirst$** (suche die erste Datei und liefere ihren Namen) ein Handle an. Dahinter steht ein Speicherblock, in dem eine Liste der im Ordner vorhandenen Dateien und weitere Verwaltungsdaten abgelegt werden. Der direkte Zugriff auf diese Liste wäre für den BASIC Programmierer sehr aufwändig oder gar nicht machbar. Deswegen benutzt **FileFindNext$** das Handle um nacheinander die einzelnen Dateinamen aus der Liste auszulesen. **FileFindDone** sorgt schließlich dafür, dass der von **FileFindFirst$** angeforderte Speicher wieder freigegeben wird.

```
DIM  han   AS   HANDLE
DIM  name$                    ! Stringvariable
...
name$ = FileFindFirst$ ( han, "*" )   ! Handle initialisieren.
                              ! "*" heißt: alles finden
...
name$ = FileFindNext$ ( han ) ! Handle benutzen
...
FileFindDone ( han )          !   Handle freigeben, d.h. die
                              ' Datenstrukturen werden
                              ' freigegeben. Die in han
                              ' gespeicherten Werte sind
                              ' jetzt ungültig.
```

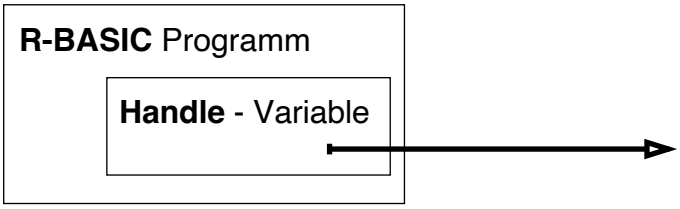


Bild: Das Handle wurde freigegeben, die Referenz ist noch vorhanden, aber ungültig.

Nachdem das Handle freigegeben wurde (im Beispiel mit **FileFindDone**) kann die Variable sofort für andere Zwecke wieder verwendet werden.

Das Konzept der Handles ist dem des Datentyps **FILE** analog. Auch hier werden R-BASIC-intern Daten angelegt (**FileOpen**), verwendet (**FileRead** bzw. **FileWrite**) und wieder freigegeben (**FileClose**).

NullHandle

NullHandle( ) ist eine Funktion, die ein "leeres" Handle liefert, d.h. sie dient zum Löschen einer Handlevariablen. **Achtung! NullHandle( )** überschreibt die Referenz in der Handlevariablen mit Nullen, ohne die dahinter stehenden Datenstrukturen zu löschen! Das muss vorher passiert sein.

Syntax: **<han>** = **NullHandle**( )
Die Klammern sind erforderlich, weil NullHandle eine Funktion ist.
<han>: Variable vom Typ HANDLE

Hinweise:
- Es ist nicht zwingend erforderlich ein freigegebenes Handle auch mit **NullHandle( )** zu löschen. Die irrtümliche Verwendung eines bereits ungültigen Handles kann aber zu einem Systemabsturz führen. R-BASIC kann jedoch ein mit **NullHandle()** gelöschtes Handle erkennen und so eventuell einen Systemabsturz verhindern.
- Nachdem ein Handle freigegeben wurde (d.h. die dahinter stehenden Datenstrukturen freigegeben wurden) kann die Variable sofort für andere Zwecke wiederverwendet werden. Ein vorheriges Löschen mit **NullHandle()** ist nicht nötig.

HandleInfo$

Die Funktion **HandleInfo$** liefert einen Text, der interne Informationen über jede Art von Handle liefert. Sie können diese Funktion zur Fehlersuche einsetzen.

Syntax: **<stringVar>** = **HandleInfo$**(**<han>**)
<han>: Variable oder Ausdruck vom Typ HANDLE
<stringVar>: Stringvariable

2.2.4.3 OBJECT Variablen

Eine Variable vom Type "Object" enthält eine **Referenz** auf ein GEOS-Objekt, z.B. einen Button, eine Liste oder ein anders Objekt des sogenannten "User-Interface". Eine ausführliche Beschreibung der Arbeit mit Objekten finden Sie im Objekt Handbuch.

Objektvariablen müssen explizit **AS OBJECT** deklariert werden. Danach kann man ihnen eine Wert (ein Objekt) zuweisen und sie wie jedes explizit aufgeführte Objekt verwenden.

```
DIM   ob   AS  OBJECT

  ob = DemoPrimary              ' Ein Objekt aus dem UI-Code
                                ' Fenster
  ob.Caption$ = "Neue Titelzeile"
```

NullObj

NullObj( ) ist eine Funktion, die ein "leeres" Objekt liefert, d.h. sie dient zum Löschen einer Objektvariable, oder zum Prüfen, ob sie leer ist.

Syntax: **<oVar>** = **NullObj**( )
Die Klammern sind erforderlich, weil NullObj eine Funktion ist.
<oVar>: Variable vom Typ OBJECT

ObjInfo$

Die Funktion **ObjInfo$** liefert einen Text, der interne Informationen über eine Objektvariable liefert. Sie können diese Funktion zur Fehlersuche einsetzen.

Syntax: **<stringVar>** = **ObjInfo$**(**<objExpr>**)
<objExpr>: Variable oder Ausdruck vom Typ OBJECT
<stringVar>: Stringvariable

2.2.4.4 R-BASIC Strukturtypen

Die Strukturtypen sind hier der Vollständigkeit halber aufgelistet. Sie werden im Kontext der Kapitel erklärt, in dem sie gebraucht werden. Im R-BASIC Anhang finden Sie eine Zusammenfassung der Elemente dieser Strukturen und ihrer Bedeutung.

Typ	Speicherbedarf	Inhalt
GeodeToken	7 Byte	Ein "Token" eines Programms oder Dokuments
DateAndTime	12 Byte	Datum und Uhrzeit
PrintFontStruct	14 Byte	Typ der Systemvariablen **printFont**. Enthält Informationen zur Steuerung der Textausgabe mit PRINT.
NumberFormatStruct	32 Byte	Typ der Systemvariablen **numberFormat**. Enthält Informationen zur Formatierung von Zahlen bei der Ausgabe mit PRINT.
GraphicDrawStruct	48 Byte	Typ der Systemvariablen **graphic**. Enthält Informationen zur Grafikausgabe wie Linienbreite, Flächenfarbe usw.
TransMatrix	60 Byte	Transformationsmatrix des Bildschirms wie Skalierung oder Rotation
PaletteEntry	3 Byte	Ein einzelner Eintrag in einer Farbpalette für Bitmaps
FullPalette	768 Byte	Eine vollständige Bitmap-Farbpalette mit 256 Einträgen
AnyStruct	3500 Byte	Strukturtyp der zu allen anderen Strukturen zuweisungskompatibel ist.
GraphicInfo	8 Byte	Informationen über eine Grafik (Bitmap oder GString)
DocumentConfigStruct	160 Byte	Daten zur Konfiguration des DocumentGuardian-Objekts
PointList	134 Byte	Liste von Punkten um Polygone und verbundene Linien zu zeichnen.
ReleaseNumber	8 Byte	Release- oder Protokollnummer einer Datei
RectDWord	16 Byte	Koordinaten eines Rechtecks

2.2.5 Felder

Felder (auch Arrays genannt) sind Zusammenfassungen von Variablen gleichen Typs, die über einen Index abgesprochen werden.

Anfänger haben erfahrungsgemäß Schwierigkeiten, sich diesem Thema zu nähern, aber Felder sind ein sehr leistungsfähiges Konzept, dass in keiner Programmiersprache fehlen darf.

Häufig gibt es nämlich das Problem, dass eine Liste von Werten, z.B. Namen und die dazugehörigen Telefonnummern verarbeitet werden sollen. Natürlich könnte man folgendermaßen vorgehen:

```
DIM NA1$, NA2$, NA3$, NA4$        ' usw.
DIM TEL1, TEL2, TEL3, Tel4        ' usw.

  NA1$ = "Müller" : TEL1 = 1234567     ' usw.

  PRINT NA1$, TEL1
  PRINT NA2$, TEL2
  PRINT NA3$, TEL3
```

Das ist nicht nur sehr aufwändig, sondern auch sehr fehleranfällig und unflexibel. Die Lösung für das Problem heißt Felder.

Ein Feld fasst Variablen gleichen Typs (und i.A. gleicher Bedeutung) so zusammen, dass die einzelnen Variablen des Feldes über eine Nummer (den Index) angesprochen werden. Dieser Index geht immer von Null bis zu einem vereinbarten Maximalwert. Die Anweisung

```
  DIM   NA$(10),   TEL(10)
```

vereinbart 11 Stringvariablen und 11 Realvariablen:

NA$(0), NA$(1), .. usw. bis NA$(10) für die Namen
TEL(0) bis TEL(10) für die Telefonnummern.

Damit haben wir auf einen Schlag Platz für 11 Personen in der Liste, die außerdem über ihren Index (Nummer) eindeutig identifiziert werden können.

Diese Feldvariablen können genau wie ganz normale Variablen verwendet werden:

```
  NA$(1) = "Müller"
  TEL(1) = 1234567        ' usw.
```

Feldvariablen sind wie geschaffen für die Zusammenarbeit mit FOR-Schleifen. Die Ausgabe der Namensliste gestaltet sich extrem einfach:

```
  FOR   N = 0   TO   10
    PRINT NA$(N),  TEL(N)
    NEXT  N
```

Eine Berechnung des Feldindex ist zulässig und wird oft benutzt. Man kann hierfür auch sehr komplexe Formeln verwenden, wenn man will. Sollte das Ergebnis der Indexberechnungen einmal nicht ganzzahlig sein, so schneidet R-BASIC den Wert ab.

Achtung: Wird eine Formel zur Berechnung eines Feldindexes verwendet, kann es passieren, dass das Ergebnis dieser Formel außerhalb des zulässigen Bereichs für den Index liegt. In diesem Fall kommt es zu einem Laufzeitfehler und die Programmabarbeitung wird beendet.

Beispiele:

```
DIM A(10)    AS  WORD    ' A(0) bis A(10)
DIM C$(10,2)             ' C$(0,0) bis  C$(10, 2)
DIM Z(4,2,2,1)           ' 4 Dimensionen – sehr exotisch
```

- Für Felder sind alle in R-BASIC vorhandenen Typen zulässig:
  - numerische Typen (Real, Byte, Word, Integer, DWord und Longint)
  - Zeichenketten-Typen String und String(n)
  - Handle, Object und File
  - BASIC interne Strukturtypen
  - selbst definierte Strukturen

- Der zulässige Index beginnt immer mit Null und endet bei dem in der Vereinbarung angegebenen Wert. Im Beispiel oben: A hat 11 Elemente: A(0) bis A(10) (sprich: A-von-Null bis A-von-10).

- Die Größe einer Felddimension ist auf 32767 (positive Integerwerte) beschränkt. Folgende Vereinbarung führt zu einem Compilerfehler:

```
DIM F(35000)             ' DAS GEHT NICHT
```

- Felder können bis zu 16 Dimensionen (Indizes) haben. Im Beispiel oben hat C$ zwei und Z hat 4 Dimensionen.

- Nichtganzzahlige Indizes werden von R-BASIC gerundet

- Liegt ein Feldindex außerhalb des vereinbarten Bereichs (i.A. durch Berechnung) kommt es zu einem Laufzeitfehler und das Programm wird beendet.

- Bei der Vereinbarung von Feldern können für die Indizes Konstanten verwendet werden. Erlaubt sind außerdem die Grundrechenarten, Klammern sowie die folgenden Operatoren und Funktionen:

  die Operationen ^ (Exponent) und MOD (Modulo-Division),
  logische Operatoren (OR, AND, NOT, XOR)
  die Funktionen INT(), ASC() und SizeOf().

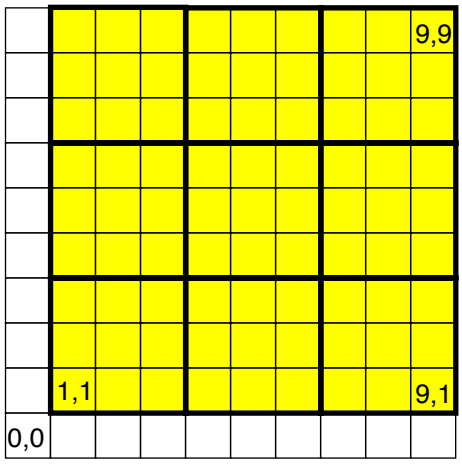
```
CONST   NUM_WORKERS       20
CONST   NUM_PEOPLE        NUM_WORKERS +  5

DIM   A(NUM_WORKERS), B(NUM_PEOPLE)    AS WORD
DIM   C(2*NUM_WORKERS - 7)
DIM   W( Int(NUM_WORKERS/3) )
DIM   Z( 4*sizeof(GeodeToken) + 3 )
```

Felder mit mehreren Dimensionen:

Obwohl Felder mit einem Index (einer Dimension) oft ausreichen, kann man auch Felder mit mehreren Dimensionen vereinbaren. Zweidimensionale Felder kommen noch recht häufig vor. Die beiden Indizes werden dann häufig als x- und y-Koordinate bezeichnet.

Beispiel 1:
Ein Sudoku-Programm könnte zur Verwaltung der 9 x 9 Felder des Sudokubrettes ein zweidimensionales REAL-Feld verwenden:

```
 DIM  BRETT ( 9, 9) AS REAL
```



Die linke untere Ecke könnte dem Element BRETT(1,1), rechts unten BRETT(9, 1) und rechts oben BRETT(9, 9) entsprechen. Die Elemente mit dem Index Null sind natürlich auch vorhanden, aber in diesem konkreten Beispiel unbenutzt oder für Spezialaufgaben verwendbar.
Der folgende Code prüft, ob in der ganz rechten Spalte die 7 schon vergeben ist:

```
 FOR   N = 1 TO 9
   IF brett(9, N) = 7 THEN PRINT "7 ist vergeben"
   NEXT N
```

Felder mit mehr als einer Dimension erfordern sehr schnell sehr viel Platz. Das Feld BRETT im obigen Beispiel vereinbart bereits 100 Realvariablen (10 in jeder Dimension). Das kostet schon 1000 Byte Speicherplatz. Als WORD-Feld wären es

nur 200 Byte - der Preis dafür ist eine geringfügig erhöhte Zugriffszeit, da die WORD-Werte von R-BASIC vor ihrer Benutzung jedes Mal in eine Realzahl konvertiert werden.

Beispiel 2:
Die Punkte einer Bitmap kann man ebenso als Feld mit zwei Dimensionen (x- und y-Koordinate) auffassen.

```
DIM   Bild(800, 600)   AS  HUGE DWORD
```

Da das Feld sehr groß ist (801x601x4 Byte = 1,9 MB) haben wir es als HUGE vereinbart (siehe Abschnitt 2.2.7).

R-BASIC unterstützt bis zu 16 Dimensionen und der HUGE Speicher für Felder kann bis 2 GB groß werden - damit sollten selbst die ausgefallensten Wünsche erfüllbar sein.

2.2.6 Globale und Lokale Variablen

Neben dem Datentyp einer Variablen ist es ebenso bedeutsam, welcher Programmteil auf eine Variable zugreifen kann. R-BASIC kennt grundsätzlich zwei Gültigkeitsbereiche von Variablen: **globale** und **lokale** Variablen.

Globale Variablen werden außerhalb von Unterprogrammen (SUB's, FUCNTION's und ActionHandlern) vereinbart. Die Vereinbarung (mit **DIM**) erfolgt üblicherweise am Anfang des Programms. Bei größeren Projekten sollten Sie alle globalen Variablen im "Dim & DATA" Fenster vereinbaren. Auf globale Variablen kann von jedem Programmteil aus zugegriffen werden. Für globale Variablen stehen in R-BASIC drei Speicherbereiche bereit: der eigentliche "**globalen Variablenspeicher**" (bis zu 12 kByte), der "**globalen Stringspeicher**" (bis zu bis zu einigen hundert kByte) und der "**HUGE Speicher**" (bis zu 2 GigaByte). Details dazu finden Sie im nächsten Kapitel.

Sie sollten globale Variablen nur dann einsetzten, wenn es wirklich nötig ist. Besonders Anfänger verwenden gern ausschließlich globale Variablen. Die Verwendung einer globalen Variablen in verschiedenen Unterprogrammen zu verschiedenen Zwecken kann jedoch schnell zu "unerklärlichen" Wechselwirkungen und Fehlen führen.

Lokale Variablen werden innerhalb von Unterprogrammen definiert. Sie sind deswegen nur diesem Unterprogramm bekannt. Dabei gelten die im Folgenden genannten Spielregeln.

- R-BASIC legt (wie jede andere Programmiersprache auch) den Speicherplatz für diese Variablen erst beim Aufruf des Unterprogramms an und gibt ihn nach Ausführung des Unterprogramms wieder frei.
- Parameter, die an Unterprogramme übergeben werden, werden intern genau wie lokale Variablen behandelt. R-BASIC kopiert beim Aufruf eines Unterprogramms die Übergabeparameter in den lokalen Variablenbereich des Unterprogramms. Nach Beendigung des Unterprogramms werden die Werte jedoch nicht zurückkopiert.
- Die Benennung der lokalen Variablen und Parametern kann völlig unabhängig von anderen Programmteilen erfolgen. Namensdopplungen mit globalen Variablen und lokalen Variablen bzw. Parametern anderer Unterprogramme sind daher kein Problem. Bei der Verwendung eines Variablennamens prüft R-BASIC zunächst, ob eine lokale Variable (oder ein Parameter) dieses Namens existiert. Wenn ja, wird diese verwendet. Erst dann wird geprüft, ob eine globale Variable dieses Namens existiert und diese ggf. verwendet. Auf lokale Variablen anderer Unterprogramme kann grundsätzlich nicht zugegriffen werden.

Für lokale Variablen stehen für jedes Unterprogramm bis zu 8 Kilobyte zur Verfügung. Dort werden alle lokalen Variablen, auch die Stringvariablen, abgelegt.

Im folgenden Beispiel gibt es drei globale Variablen (A, B, und X). Die Sub namens DemoSub hat zwei lokale Variablen: Den Parameter X und die Integer-Variable A. Deswegen kann sie auf die globalen Variablen A und X nicht

zugreifen. Die Zeile "A = 2 * X - 3" belegt die lokale Variable A mit dem doppelten des Wertes des Parameters X, abzüglich 3. Die Zeile "B = SQR(A)" belegt die **globale** Variable B mit der Quadratwurzel aus der lokalen Variablen A.
Nach dem Aufruf der Sub mit "DemoSub 12" enthält die globale Variable B den Wert 4,5826 (= Sqr(2*12-3)). Die globalen Variablen A und X sind nicht verändert.

```
DIM A, B, X       ' Globale Variablen vom Typ REAL

SUB DemoSub (X AS word)
DIM A as Integer
  A = 2 * X - 3
  B = SQR(A)
END SUB

' Aufruf:
DemoSub 12
```

Die Belegung einer globalen Variablen in einer SUB ist prinzipiell ein schlechter Stil und sollte vermieden werden (was leider nicht immer geht). Sauberer wäre im obigen Beispiel die Verwendung einer Funktion, die einen Wert zurückliefert:

```
DIM A, B, X       ' Globale Variablen vom Typ REAL

FUNCTION DemoFunc (X AS word) AS REAL
DIM A as Integer
  A = 2 * X - 3
  RETRUN SQR(A)
END FUNCTION

' Aufruf:
B = DemoFunc ( 12 )
```

2.2.7 Interne Verwaltung der Variablen, HUGE Variablen

Lokale Variablen werden **alle** in einem Speicherblock abgelegt, der "lokaler Variablenspeicher" genannt wird. Dadurch ist ein sehr effizienter Zugriff auf die lokalen Variablen möglich. Jedes Unterprogramm hat seinen eigenen lokalen Variablenspeicher. Dieser Speicherblock kann für jedes Unterprogramm bis zu 8 Kilobyte groß sein. Das ist meist mehr als genug, PC/GEOS-SDK-Programmierer haben z.B. deutlich weniger zur Verfügung.

Ein häufiger auftretendes Problem bei lokalen Variablen sind jedoch Variablenfelder vom Typ STRING, da jedes Feldelement 129 Byte benötigt. Die (lokale) Anweisung

```
DIM   SF$(120)  AS  STRING
```

erzeugt ein Feld mit 121 Elementen (Index von Null bis 120) und fordert bereits 121 Elemente a 129 Byte = 15609 Byte an. Es kommt zu einem Compilerfehler.
Sie können dann entweder den Datentyp STRING(N) verwenden, z.B.

```
DIM SF$(120)  AS  STRING(64)
```

womit nur 121*65 = 7865 Byte angefordert werden oder auf globale Stringvariablen (STRING oder HUGE STRING) ausweichen.

Für **globale Variablen** verwendet R-BASIC drei verschiedene Speicherbereiche. Im normalen "**globalen Variablenspeicher**" werden alle Variablen und Felder abgelegt, die nicht vom Datentyp STRING oder STRING(N) sind und die nicht mit dem Schlüsselwort HUGE vereinbart wurden. Für den globalen Variablenspeicher stellt R-BASIC bis zu 12 Kilobyte Speicher bereit. Auf Variablen im globalen Variablenspeicher kann sehr effizient zugegriffen werden. Die globale Vereinbarung

```
DIM RF(200)    AS REAL
```

erzeugt ein Feld mit 201 Elementen (Index von Null bis 200) und belegt 201*10 = 2010 Byte im globalen Variablenspeicher. Hingegen belegt die globale Vereinbarung

```
DIM SF$(200)   AS STRING
```

keinen Speicher im globalen Variablenspeicher. STRING Variablen (und Variablen vom Typ STRING(N)) werden **immer** im **globalen Stringspeicher** angelegt. Dieser Speicher wird von R-BASIC dynamisch und effektiv verwaltet, da der wirklich von einer Stringvariablen benötigte Platz davon abhängt, ob sie einen kurzen oder einen langen Text enthält. Je nach erforderlicher Situation legt R-BASIC Stringvariablen im RAM oder in einer Datei ab. Sie können insgesamt 16383 (&H3FFF) globale Stringvariablen vereinbaren, wobei jedes Feldelement

als eigene Variable zu zählen ist. Die Länge der Strings ist dabei unerheblich. Auch ein Stringfeld mit 16000 Elementen ist daher kein Problem:

```
DIM   SF$(16000)   AS String(128)
```

R-BASIC stellt die erforderlichen 2 Megabyte bereit.

Wollen Sie noch größere Felder verwalten können Sie das Schlüsselwort **HUGE** (= riesig) verwenden. Der **Huge Speicher** ist eine von R-BASIC bereitgestellte Datei auf der Festplatte. Alle HUGE Feldelemente haben eine feste Größe und eine feste Position in der Datei (d.h. eine dynamische Stringverwaltung findet nicht statt). Sie können auf die HUGE Variablen mit der ganz normalen BASIC Syntax zugreifen, die Dateiverwaltung von GEOS sorgt dafür, dass der Geschwindigkeitsverlust durch die häufigen Festplattenzugriffe nicht allzu groß ausfällt. Der Huge Speicher kann insgesamt bis zu 2 Gigabyte groß sein. Jedoch darf auch bei HUGE Feldern ein einzelner Feldindex den Wert 32767 nicht überschreiten. Verwenden Sie für größere Datenmengen bitte mehrdimensionale Felder. Das folgende Beispiel vereinbart drei Felder vom Typ und ein Feld vom Typ Word der Gesamtgröße von ca. 42 Megabyte.

```
DIM  ImageRed(1280, 1024)   AS  HUGE REAL
DIM  ImageGreen(1280, 1024) AS  HUGE REAL
DIM  ImageBlue(1280, 1024)  AS  HUGE REAL
DIM  ImageMask(1280, 1024)  AS  HUGE WORD
```

Hinweis: Globale Variablen einer Library (egal ob exportiert oder nicht) werden im globalen Variablenspeicher des aufrufenden Programm abgelegt. Wenn mehrere Programme eine Library gleichzeitig verwenden, so wird für jedes Programm ein eigener Satz dieser Variablen angelegt. Library und Programm verwenden diese Variablen so, als ob sie allein im System sind. Eine gegenseitige Beeinflussung verschiedener Programme ist ausgeschlossen.

2.2.8 Strukturen

Mit Hilfe der STRUCT-Anweisung kann man Variablen unterschiedlichen Typs zusammenfassen. Anfänger haben erfahrungsgemäß Schwierigkeiten, sich diesem Thema zu nähern, aber Strukturen sind ein sehr leistungsfähiges Konzept, dass in keiner Programmiersprache fehlen darf.

2.2.8.1 Grundlagen

Häufig besteht das Problem, dass zum Verwalten von Daten sehr viele Informationen für ein einzelnes Objekt gespeichert werden müssen. Beispielsweise benötigt man für eine Kontaktliste neben Namen und Vornamen auch Telefon, Email, Fax, Geburtsdatum und einiges mehr.

Strukturen bieten die Möglichkeit, alle diese Informationen "im Block" zu speichern und zu verwalten. Dazu muss man zunächst einen neuen Struktur-Typ vereinbaren, der alle nötigen Informationen enthält. Die Syntax sieht so aus:

```
STRUCT  Person        ' Person ist der Name des neuen
                      ' Strukturtyps
  Name$, Vorname$     AS String(30)
  Tel$                AS String(15)
  persNummer          AS Word          ' max 65000 - das reicht
  END  STRUCT         ' Ende der Definition
```

Anmerkung 1: Strukturen müssen bei ihrer Vereinbarung mit STRUCT eine feste Größe haben. Daher muss der Datentyp STRING(N) verwendet werden, der Platz für einen String der maximalen Länge N fest reserviert.

Anmerkung 2: Strukturen sind auf eine Größe von 3500 Byte begrenzt. Man sollte daher, besonders bei größeren Strukturen, darauf achten, welche Datentypen man einsetzt. REAL kostet z.B. 10 Byte, WORD aber nur 2. String(n) reserviert N+1 Byte (Max. N Zeichen plus 1 Byte Ende-Kennung). Eine einfache Variable der oben definierten Struktur "Person" benötigt z.B. 80 Byte.

Will man den neuen Typ verwenden, vereinbart man einfach mit DIM entsprechende Variablen:

```
DIM  Chef, Knecht,  Magd  AS Person
```

Verwendet werden Strukturvariablen genau wie alle anderen Variablen, allerdings muss man sowohl die Variable (z.B. Chef) als auch das Strukturelement (z.B. Name$), getrennt durch einen einfachen Punkt '.' angeben.

```
Chef.Name$ = "Schneider"
Chef.Vorname$ = "Wilhelm"
Chef.Tel$ = "030 456 897 5654"
Chef.persNummer = 1               ' Der Boss ist wichtig
Magd.Name$ = Knecht.Name$         ' Sie haben geheiratet
```

Man kann aber auch eine komplette Strukturvariable einer anderen zuweisen, vorausgesetzt der Typ stimmt überein. Dadurch werden die Daten kopiert (Zeile 1). Zeile 2 und 3 dienen der Illustration.

```
Chef = Knecht             ' Er wurde befördert
Chef.persNummer = 1       ' persNummer wurde auch kopiert:
                          ' neu setzen
Knecht.persNummer = 0     ' Die Stelle ist verfügbar.
```

Ein weiterer Vorteil: Wenn man später feststellt, dass man eine wichtige Information, z.B. den Geburtsort, hinzufügen will, ist das kein Problem. Man ergänzt einfach die Strukturdefinition:

```
STRUCT  Person
  Name$, Vorname$     AS String(30)
  Tel$                AS String(15)
  gebOrt$             AS String(20)     ' das ist neu.
  persNummer          AS Word
  END  STRUCT
```

Alle bisher geschriebenen Programmteile arbeiten weiter wie gewohnt, man muss nur den Code zum Verwalten des Geburtsortes hinzufügen.

Für die Elemente einer Struktur sind - mit Ausnahme des Typs STRING - alle in R-BASIC vorhandenen Typen zulässig:

- numerische Typen (Real, Byte, Word, Integer, DWord und Longint)
- Zeichenketten-Typ: nur String(n). (String ohne (n) ist unzulässig)
- Object, File und Handle
- andere Strukturen: R-BASIC intern oder selbst definiert
- Felder der oben genannten Typen

NullStruct

Die Funktion NullStruct() liefert eine "leere" Struktur zurück, dient also zum Löschen einer Struktur-Variablen. NullStruct ist auf Strukturen jeden Typs anwendbar, ebenso auf Struktur-Elemente, die selbst Strukturen sind.

Syntax: strukturVariable = NullStruct()
Parameter: Keine. Die Klammern sind aber erforderlich.

Beispiel:

```
STRUCT       Stru1
  A, B   AS REAL
  END STRUCT

STRUCT  Stru2
    A        AS REAL
    S, T     AS Stru1
  END STRUCT

DIM stVar AS Stru2

  stVar = NullStruct()    ' Löscht die gesamte Struktur
  stVar.s = NullStruct()  ' Löscht nur stVar.s
                ' stVar.a  und  stVar.t  bleiben erhalten
```

2.2.8.2 Verschachtelung von Strukturen

Es ist zulässig Strukturen zu definieren, die andere Strukturen als Elemente enthalten. Dies bezeichnet man als Verschachtelung von Strukturen.
Die Verschachtelung von Strukturen untereinander ist prinzipiell unbegrenzt. Eine gute Planung der Strukturen ist aber Voraussetzung, sonst werden diese Verschachtelungen schnell unübersichtlich und damit fehleranfällig.

Beispiel

```
STRUCT Firmenleitung
  TheBoss         AS  Person
  Tippse          AS  Person
  Buchhalter      As   Person
  END STRUCT

DIM DieChefs AS Firmenleitung
  DieChefs.TheBoss.Name$ = "Setag"
  DieChefs.TheBoss.Vorname$ = "Llib"
  DieChefs.Tippse.Name$ = "Clausen"
  DieChefs.Tippse.Vorname$ = "Ella"
```

Die Gesamtgröße einer Struktur ergibt sich als Summe der Größe der einzelnen Strukturelemente. Die Struktur "Firmenleitung" belegt z.B. 303 Byte. Bei Verwendung von Felder innerhalb von Strukturen ist zu beachten, dass der Feldindex immer bei Null beginnt. R-BASIC unterstützt Strukturen bis zu einer Größe von 3500 Byte.

2.2.8.3 Strukturen und Felder

Strukturen können beliebig mit Feldern kombiniert werden. Hier sehen Sie an einigen Beispielen, wie das geht. Beachten Sie jeweils die Position der Feldindizes (die in den Klammern)

Felder von Strukturen

```
STRUCT Point              ' Punkt mit 2 Koordinaten
  px,  py        AS Word
  END STRUCT

DIM pointList(10) AS Point    ' 11 Punkte

pointList(0).px = 0       ' Erster Punkt, Koordinaten (0,0)
pointList(0).py = 0
pointList(1).px = 100     ' Zweiter Punkt, Koordinaten (100,50)
pointList(1).py = 50
```

Felder als Strukturelemente

```
STRUCT  ByteFeld
  anzahl     AS  Integer
  value(2000)    AS Byte      ' 2000 Byte. value heißt "Wert"
  END   STRUCT

DIM   puffer  AS ByteFeld

puffer.value(10) = 27
puffer.value(11) = 15
PRINT  puffer.value(10) + 256 * puffer.value(11)
```

Die Struktur "Person" aus dem Abschnitt 2.2.8.1 wird vorausgesetzt:

```
STRUCT  Firma
  Name$      AS String(100)    ' z.B. der Firmenname
  Worker(16) AS Person
  END STRUCT ' Insgesamt 1818 Byte.

DIM  Metzgerei   AS Firma
  Metzgerei.Name$ = "Metzger Hackbeil"
  Metzgerei.Worker(0).Name$ = "Malocher"
  Metzgerei.Worker(0).Vorname$ = "Max"
  Metzgerei.Worker(1).Name$ = "Schufter"
  Metzgerei.Worker(1).Vorname$ = "Siegfried"
```

2.2.8.4 Strukturen und Unterprogramme

Strukturen können als Parameter und Rückgabewerte von Unterprogrammen (SUB's, FUNCTON's) verwendet werden. Das folgende Beispiel illustriert das. Die Struktur "Person" aus dem Abschnitt 2.2.8.1 wird vorausgesetzt:

```
SUB PrintPerson( p AS Person)
  Print p.Vorname$;" ";p.Name$;" Tel.";p.Tel$
END SUB

FUNCTION  NewPerson (n$, v$, geb$ as string) AS Person
DIM P AS Person
  p.Name$ = n$
  p.Vorname$ = v$
  p.GebOrt$ = geb$
  RETURN P
END FUNCTION

DIM Jemand  as Person
Jemand = NewPerson("Panther", "Paulchen", "Farbtopf")
PrintPerson(Jemand)           ' Die Klammern sind optional
```

2.2.8.5 Formale Syntax

Aus Kompatibilitätsgründen werden je zwei Syntaxvarianten für Strukturen und Strukturvariablen unterstützt. Funktionell unterscheiden sich die Varianten aber nicht.

Vereinbarung von Strukturen

Standard Syntaxvariante

```
STRUCT <StructName>
        <elementListe> AS Type          ' z.B.  a, b AS  Real
        ...                             ' weitere Strukturelemente
        END STRUCT
```

Syntaxvariante mit DIM

```
STRUCT <StructName>
        DIM <elementListe> AS Type          ' z.B.  DIM a, b AS  Real
        ...
        END STRUCT
```

Vereinbarung von Strukturvariablen

Standard Syntaxvariante

```
DIM <varListe> AS StructName
```

Syntaxvariante mit STRUCT

```
DIM <varListe> AS STRUCT StructName
```

Beispiele Standardsyntax:

```
STRUCT BspStruct
    a, b     AS real
    c$       AS String(20)
    END STRUCT

DIM P1, P2 As BspStruct
```

Beispiele alternative Syntax:

```
STRUCT BspStruct
    DIM   a, b   AS real
    DIM   c$     AS String(20)
    END STRUCT

DIM P1, P2 As STRUCT BspStruct
```

2.2.8.6 Namenskonventionen

Für die Bezeichnung von Strukturen und Strukturelementen gelten genau zwei einfache Regeln:

1. Der Name des Strukturtyps (z.B. Point s. unten) muss innerhalb des Programms eindeutig sein. Es gelten die gleichen Konventionen wie für globale Variablen oder Unterprogramme.

2. Der Name von Strukturelementen muss nur innerhalb der Struktur eindeutig sein. Namensdopplungen mit globalen Variablen, Elementen anderer Strukturen, ja sogar mit BASIC Befehlen, sind erlaubt. Sie haben bei Strukturelementen also wesentliche mehr Freiheiten als bei einfachen Variablen.

Am besten verwenden Sie einfache, selbsterklärende Bezeichnungen. Falls R-BASIC ein Problem findet, gibt es einen Compilerfehler, d.h. ein Programm, dass sich compilieren lässt, ist namenstechnisch in Ordnung.

Die folgenden Beispiele zeigen einige Möglichkeiten auf.

```
STRUCT Point          ' eine einfache Struktur
  px, py AS Word
  END  STRUCT

STRUCT Demo
  a, b   AS Real      ' nichts besonderes
  pi     AS Byte      ' kein Konflikt mit 3.1415...
  px, py AS Word      ' kein Konflikt mit der
                      ' Struktur Point
  color  AS Longint   ' selbst BASIC-Befehle sind zulässig
  p      AS Point     ' Struktur innerhalb der Struktur
  p2          AS Point ' Struktur innerhalb der Struktur
  END  STRUCT

' Variablen-Definitionen
DIM d  AS Demo
DIM A  AS Demo   ' kein Problem mit A in der Struktur Demo
DIM px AS REAL   ' kein Problem mit px in der Struktur Point

  px = 12        ' Zuweisung zur Realvariablen

  d.pi  = 12
  d.px = 45
  d.p.px = 59    ' d.px wird nicht geändert
  d.p2.px = 79   ' d.p.px und d.px bleiben erhalten
  d.a = pi       ' weist 3.1415... zu.
                 ' d.pi bleibt erhalten (=12, siehe oben)
```

Das folgende Beispiel zeigt zwar zulässige Vereinbarungen die aber namenstechnisch sehr unübersichtlich gewählt sind:

```
STRUCT Point          ' eine einfache Struktur
  px, py AS Word
  END  STRUCT

STRUCT Komisch
  point  AS Point     ' selbst dieser Name ist zulässig
  komisch    AS Real  ' Kein Problem mit dem Namen der
                      ' Struktur, aber sehr unübersichtlich
  END  STRUCT

DIM py AS Point       ' py ist hier erlaubt, auch wenn es schon
                      ' in Point enthalten ist
                      ' das ist aber nicht mehr übersichtlich
DIM k AS Komisch

  py.px = 28          ' Zuweisung zum Strukturelement px der
                      ' Strukturvariablen py
  py.py = 234         ' R-BASIC verwechselt das nicht
  k.komisch = 12
  k.point.px = 17
```

Beispiel für eine unzulässige Vereinbarung

```
  STRUCT  Line        ' Schlecht. LINE ist ein BASIC-Befehl
    x0, y0, x1, y1 AS Word
    END  STRUCT
```

2.2.8.7 Ein Anwendungsbeispiel

Mit Strukturen kann man sehr einfach externe Daten abbilden. Das Beispiel zeigt die Struktur der ersten Bytes einer PCX-Datei, des sogenannten Headers. Der Code am Ende des Beispiels liest den PCX-Header komplett ein und gibt die Abmessungen des Bildes aus.

```
STRUCT   PCXPalette          ' ein Farb-Paletteneintrag
  rt, gn, bl     AS Byte
  END   STRUCT

STRUCT   PCXFileHeader               ' steht am Dateianfang
  id, version            AS Byte
  compressed         AS Byte
  bitsPerPlane           AS Byte
  xMin, yMin         AS Word
  xMax, yMax         AS Word
  xRes, yRes         AS Word
  colorMap(16)       AS PCXPalette    ' ein Struktur-Feld
  reserved           AS Byte
  colorPlanes        AS Byte
  bytesPerLine       AS Word
  paletteInformation AS Word
  screenSizeX            AS Word
  screenSizeY            AS Word
  fill (53)          AS Byte          ' ein Byte Feld,
                                      ' 54 Byte
  END  STRUCT

DIM header AS PCXFileHeader
DIM f AS FILE

  f = FileOpen("FLOWER.PCX")
  header = FileRead ( f, SizeOf(PCXFileHeader))     ' alles
                                                    ' einlesen
  FileClose (f)
  Print "Abmessungen:"; header.xMax; "x"; header.yMax; "Pixel"
```

2.2.8.8 AnyStruct

Dieser Abschnitt richtet sich an fortgeschrittene Programmierer. Eine Übersicht über die verschiedenen Strukturtypen, die in R-BASIC definiert sind, können Sie im Abschnitt 2.2.4.4 finden.

In sehr seltenen Spezialfällen sind Routinen sinnvoll, denen in verschiedenen Situationen Strukturen verschiedener Typen übergeben werden sollen oder die in unterschiedlichen Situationen Strukturen verschiedener Typen zurückgeben sollen. Dafür dient der Typ AnyStruct. Variablen oder Funktionen von diesem Typ sind zuweisungskompatibel zu jedem beliebigen Strukturtyp.
Die Library "VMFiles" macht davon Gebrauch.

Definition:

```
STRUCT AnyStruct
  any_struct_dummy_byte_array_ (3499) as BYTE
  End struct
```

Hinweise:
- Eine Variable oder ein Parameter des Typs AnyStruct belegt 3500 Byte im Variablenspeicher.
- Es ist im Allgemeinen nicht nötig, auf die Elemente dieser Struktur zuzugreifen.
- Bei einer Zuweisung oder Parameterübergabe werden stets nur so viele Bytes kopiert, wie der kleinere der beteiligten Strukturtypen fassen kann.

Beispiel 1: Einfache Zuweisungen

```
DIM g AS GeodeToken
DIM t AS DateAndTime
DIM a AS AnyStruct

! Erlaubt ist z.B. folgendes
g = a
a = t
```

Beispiel 2: Übergabe verschiedener Strukturtypen an eine SUB

```
DIM g AS GeodeToken
DIM t AS DateAndTime

DECL SUB AnyTest1(a as AnyStruct, x as WORD)

AnyTest1(g, 0)       ' Übergabe GeodeToken
AnyTest1(t, 1)       ' Übergabe DateAndTime
```

Beispiel 3: Rückgabe verschiedener Strukturtypen

```
DIM g AS GeodeToken
DIM t AS DateAndTime

DECL Function AnyTest2(x as WORD) AS AnyStruct

g = AnyTest2(0)      ' Zuweisung an eine GeodeToken Struktur
t = AnyTest2(1)      ' Zuweisung an eine DateAndTime Struktur
```

Implementationen der Routinen aus den Beispielen

```
SUB AnyTest1(a as AnyStruct, x as WORD)
DIM gt as GeodeToken
DIM dat as DateAndTime
  IF x = 0 THEN
    gt = a
    <hier mit gt arbeiten>
  ELSE
    dat = a
    <hier mit dat arbeiten>
  END IF
END SUB

Function AnyTest2(x as WORD) AS AnyStruct
DIM gt as GeodeToken
DIM dat as DateAndTime
  IF x = 0 THEN
    <hier gt belegen>
    return gt
  ELSE
    <hier dat belegen>
    return dat
  END IF
END FUNCTION
```

2.2.9 Die Funktionen SizeOf und Swap

SizeOf

Die Funktion **SizeOf** (Größe von) liefert den Speicherbedarf einer Variablen oder eines Datentyps. Diese Funktion ist sehr hilfreich bei der Fehlersuche und im Zusammenhang mit Strukturen. Außerdem können Sie Ihr Programm besser lesbar gestalten. Die Formulierung "10 * **SizeOf**(WORD)" macht klar, dass der Speicherbedarf von 10 Word-Variablen gemeint ist, während die Zahl 20 alles Mögliche bedeuten kann. Die Verwendung von **SizeOf** für Strukturen erspart Ihnen das mühsame und fehleranfällige Zusammenzählen der Größen der einzelnen Elemente und liefert automatisch wieder den korrekten Wert, wenn Sie die Definition der Struktur später ändern.

Syntax:	**<numVar>** = SizeOf(**type**)
type:	Beschreibt den Datentyp, dessen Größe ermittelt werden soll.

Wichtig:

SizeOf() wird schon vom Compiler in eine Zahl übersetzt, d.h. es ist von der Ablaufgeschwindigkeit egal ob Sie schreiben:
y = 10 oder y = SizeOf (REAL)

Zulässig für **type** sind:

- R-BASIC Datentypen (Real, Byte, Word, File usw.)
- von R-BASIC definierte Strukturdatentypen (GeodeToken, DateAndTime, NumberFormatStruct usw.)
- mit STRUCT selbst definierte Datentypen
- Variablen beliebigen Typs:
  - Einfache Variablen
  - Feldvariablen
  - Strukturvariablen
  - Strukturelemente

Nicht zulässig sind einfache Systemvariablen (z.B. MaxX, tabWidth, fileError usw.) und Strukturelemente von Systemvariablen (z.B. graphic.areaColor).

Beispiel 1:

```
DIM y, a, c$
DIM s$   AS   STRING(30)

y = SizeOf(REAL)        ' liefert 10
y = SizeOf(A)       ' liefert 10
y = SizeOf(C$)      ' liefert 129
y = SizeOf(S$)      ' liefert 31
```

Beispiel 2 (komplexes Beispiel):

```
' Es sei eine Struktur MyStruct definiert:
STRUCT MyStruct
  a      AS Real
  b(12) AS word
  END STRUCT

DIM  msVar  AS  MyStruct
  y = SizeOf(MyStruct)             ' in diesem Fall: 28
  y = SizeOf(msVar)           ' in diesem Fall: 28
  y = SizeOf(msVar.a)         ' in diesem Fall: 10
  y = SizeOf(msVar.b(0))      ' in diesem Fall: 2

DIM  msFeld(4)  AS MyStruct
  y = SizeOf(msFeld(0))       ' in diesem Fall: 28
  y = SizeOf(msFeld(0).a)     ' in diesem Fall: 10
```

Swap

Das Kommando Swap vertauscht die Werte zweier Variablen. Das ist übersichtlicher und deutlich schneller als der Umweg über eine temporäre Variable.

Syntax:	Swap **var1** , **var2**
var1, var2:	Variablen, deren Werte vertauscht werden sollen.

Die Anweisung

```
  Swap a, b
```

verhält sich so, als würden Sie den folgenden Code schreiben, wobei die Variable tmp vom gleichen Datentyp ist, wie die Variablen a und b.

```
  tmp = a
  a = b
  b = tmp
```

- Zulässig für die Variablen von Swap sind, analog zu SizeOf, Variablen aller in R-BASIC verfügbaren Datentypen, auch selbst definierte Strukturen, Strukturelemente und Feldvariablen.
- Nicht zulässig sind Systemvariablen (z.B. MaxX, tabWidth, FileError usw.) und Strukturelemente von Systemvariablen (z.B. graphic.areaColor).
- Der Compiler führt keinen strengen Typvergleich, sondern nur einen Größenvergleich aus. Damit können Sie z.B. auch Word- und Integer-Variablen vertauschen. Der Programmierer ist selbst dafür verantwortlich, keine inkompatiblen Typen zu vertauschen.
- Das Vertauschen zuweisungskompatibler Variablen mit verschiedener Größe (z.B. Integer und Real) ist nicht möglich.

2.2.10 Die CONST Anweisung

Mit der Anweisung **CONST** kann man symbolische Namen für numerische Werte oder Strings festlegen. Diese werden vom Compiler durch ihren Wert ersetzt.

Syntax: CONST **name** = **<wert>**

Der Typ der Konstante wird durch den Typ des Wertes bestimmt. Pro Anweisung kann nur eine Konstante definiert werden.

Beispiele:

```
CONST anzahl_werte = 12     ' eine Real-Konstante
CONST testwert = -3.786     ' jede Zahl ist zulässig
CONST author$ = "Mein Name"
```

Vorteile:

- Das Programm wird übersichtlicher und besser lesbar
- Die Fehlerwahrscheinlichkeit sinkt drastisch
  Ändert man den Konstanten-Wert (eine Stelle im Programm), so wirkt sich die Änderung an alles Stellen aus, an denen die Konstante verwendet wurde.

Im folgenden Beispiel braucht man beim Ändern der Konstanten (anzahl oder startwert) den Code nicht anzupassen:

```
CONST   anzahl = 31
CONST   startwert = 115.7
DIM         N, Feld(anzahl)

  FOR N = 0 TO anzahl
  Feld(N) = startwert
  NEXT N
```

Numerische Konstantendefinitionen können sich auf andere Konstanten beziehen und diese mit Zahlen, den Grundrechenarten und Klammern verknüpfen. Außerdem sind zugelassen: Die Operationen ^ (Exponent) und MOD (Modulo-Division), logische Operatoren (OR, AND, NOT, XOR) und die Funktionen INT(), ASC(), SQR(), FRAC(), TRUC(), SIN(), COS(), TAN(), EXP(), LN(), LOG(), LG() und SizeOf(). Für Stringkonstanten ist nur das Pluszeichen erlaubt.

```
CONST   MAX_X        = 600
CONST   DELTA_X      = SizeOf(MyStruct) + 10
CONST   MAX_Y        = 2* MAX_X + DELTA_X
CONST   MASK_X       = INT((MAX_X+7)/8)
CONST   K2           = (MAX_X AND &hF) OR 3
CONST   NAME_SIMPEL  =   "Meier"
CONST   NAME_FULL    =   "Hans " + NAME_SIMPEL
```

Analog darf man auch bei Felddefinitionen vorgehen:

```
DIM     A(MAX_X), B(MAX_X + DELTA_X / 2)   AS WORD
```