WINCULA

Makrosprache für Anwendungen

Gültig ab Version 11.3.0

19.09.2020

Inhaltsverzeichnis

1. Einführung

Die Makrosprache *Wincula* wurde zur Programmierung von Makros für Anwendungsprogramme entwickelt. Die Sprache verfügt über einen umfangreichen Befehlssatz. Sie ist modular aufgebaut und sehr praxisnah. Sie kann auch von ungeübten Anwendern leicht erlernt werden.

Die vorliegende Sprachbeschreibung versucht, die Syntax der Makrosprache *Wincula* verständlich zu beschreiben. Zum besseren Verständnis werden eine Reihe von Beispielen vorgestellt. Da jede Anwendung unterschiedliche Makrofunktionen bereitstellt, ist zu beachten, dass die in den Beispielen verwendeten Funktionen nicht unbedingt in jeder Anwendung zur Verfügung stehen. Ein Formularprogramm stellt verständlicherweise andere Funktionen bereit als ein Texteditor.

Bitte betrachten Sie also die Beispiele wirklich exemplarisch. Sie können diese Beispiele nicht unbedingt in jeder Anwendung eins zu eins verwenden.

Um das Verständnis der Makrosprache *Wincula* zu erleichtern, sollen zunächst folgende Fragen erörtert werden:

- was ist ein Makro
- wie ruft man ein Makro auf
- wie werden Makros abgespeichert

Die nächsten Kapitel werden diese Fragen beantworten.

1.1. Was ist ein Makro

Die einfachste Form eines Makros ist die nacheinander folgende Ausführung mehrerer verschiedener Funktionen eines Programms. Das folgende Beispiel zeigt ein einfaches Makro, welches das aktuelle Tagesdatum als Text ermittelt und diesen in ein dafür vorgesehenes Datenfeld schreibt.

- Ermitteln des aktuellen Tagesdatums in Textform
- Schreibe den Text in das Feld *Datenfeld*

Für alle in diesem Makro ausgeführten Funktionen stellt die Anwendung, für die das Makro geschrieben wird, einen entsprechenden Makrobefehl bereit.

Natürlich können Makros auch komplizierter sein. Es ist zum Beispiel möglich, Befehle nur unter bestimmten Bedingungen oder mehrfach auszuführen. Außerdem kann vorher eine Anfrage beim Benutzer mittels eines Dialogs gemacht werden. Auf die verschiedenen Möglichkeiten wird später genauer eingegangen.

1.2. Wie ruft man ein Makro auf

Makros der Makrosprache *Wincula* können auf verschiedene Weise aufgerufen werden. Der Aufruf kann auf Knopfdruck durch den Anwender erfolgen oder auch automatisch, falls bestimmte Ereignisse innerhalb der Anwendung eintreten.

Soll ein Makro per Knopfdruck ausgeführt werden, kann z.B. ein Knopf in der Buttonzeile der Anwendung eingefügt werden. Gleichzeitig wird der Name einer Datei festgelegt, in der sich die Makrobefehle befinden und die ausgeführt wird, falls der Knopf gedrückt wird.

Manche Anwendungen - z.B. Formularsoftware - stellen auch die Möglichkeit zu Verfügung, Knöpfe innerhalb des Dokumentes einzublenden. Hier können natürlich ebenfalls Makroaufrufe verankert werden.

Die automatische Ausführung eines Makros findet immer dann statt, wenn ein bestimmtes Ereignis eintritt - z.B. wenn eine Taste gedrückt wird oder sich der Wert eines Datenfeldes verändert. Hierzu ein Beispiel: Es soll immer dann ein Saldofeld aktualisiert werden, wenn in einem von mehreren Feldern, in denen Zahlen eingegeben werden, sich ein Wert ändert. Die automatische Ausführung des Makros ist also an das Ereignis *Feld ändert sich* gebunden.

1.3. Wie werden Makros gespeichert

Es gibt prinzipiell zwei Arten, Makros zu speichern. Sie können entweder in einer Makrodatei abgespeichert werden oder sie können in ein Dokument eingebettet werden. Makros, die von verschiedenen Stellen aus aufgerufen werden sollen, sollten in einer Datei gespeichert werden. Makros, die ein Dokument automatisieren sollen, werden in der Regel in das Dokument – z.B. ein Formular – eingebettet. Dadurch ist sichergestellt, dass die Makros auch dann wirksam sind, wenn das Dokument auf einen anderen Rechner übertragen wird. Die Makrobefehle bilden sozusagen mit dem Dokument eine Einheit.

Makros, die in einer Datei gespeichert sind, können jederzeit durch einen Makrobefehl aus einem anderen Makro heraus aufgerufen werden. Diese Methode wird bevorzugt angewendet, wenn ein zentraler Vorgang automatisiert werden soll. Hierzu ein Beispiel:

Ein Unternehmen will alle Formulare mit dem Logo der Firma versehen. Das Firmenlogo liegt als Bitmap vor. Aus Gründen des Speicherplatzes und der Performance soll das Logo nicht in jedes einzelne Formular eingebettet werden. Außerdem behält sich die Unternehmensleitung vor, das Logo jederzeit verändern zu können, ohne alle Formulare einzeln ändern zu müssen. In diesem Fall wird jedes Formular mit einem eingebetteten Makro versehen, welches bei dem Ereignis *neues Formular anlegen* aufgerufen wird. Das eingebettete Makro ruft ein globales Makro auf, welches sich zentral in einer Datei auf dem Netzwerk des Unternehmens befindet. Dieses Makro lädt das vorgegebene Logo in das dafür in dem Formular vorgesehene Bitmap-Objekt.

Das obige Beispiel zeigt, wie hoch der Grad der Automatisierung ist, der durch Makros erreicht werden kann.

Makros, die in Dateien abgespeichert werden, erhalten zusätzlich zu den Makrobefehlen einen Rahmen, der die Datei als Makrodatei erkennbar macht. Dieser Rahmen ist in einem gesonderten Kapitel beschrieben.

2. Syntax der Makrosprache

Makros, die in der Makrosprache *Wincula* verfasst werden, bestehen aus einem oder mehreren Makrobefehlen, die hintereinander angeordnet werden. Jeder Makrobefehl wird durch ein Semikolon abgeschlossen. Innerhalb einer Zeile können mehrere Befehle aufeinanderfolgen.

Zur besseren Lesbarkeit der Makros können zwischen zwei Makrobefehle Leerzeichen, Tabulatoren oder Zeilenschaltungen eingeschoben werden, ohne die Wirkungsweise des Makros zu beeinträchtigen. So kann beispielsweise für jeden Befehl eine neue Zeile begonnen werden. Es können leere Zeilen eingeschoben werden, oder die Zeilen können durch Tabulatoren eingerückt werden.

Ein einzelner Makrobefehl muss jedoch immer innerhalb einer Zeile stehen. Schlüsselwörter und Makrobefehle dürfen nicht durch Leerzeichen oder Zeilenschaltungen unterbrochen werden.

2.1. Einführendes Beispiel

Bevor die Syntax im Einzelnen strukturiert erläutert wird, wollen wir zunächst einen Blick auf ein kleines Beispiel werfen.

```
GetToday (Datumspuffer);
SetText ("Datum", Datumspuffer);
```

Das Makro führt das Beispiel aus Kapitel 1 aus. Die Funktion *GetToday* ermittelt das augenblickliche Systemdatum und speichert es in Textform auf der Zeichenfolgenvariablen *Datumspuffer* ab.

Danach wird dieser Text in das Feld *Datum* des bearbeiteten Dokumentes übertragen und wird in diesem Moment für den Benutzer sichtbar.

Dieses Makro vermittelt ganz intuitiv, wie Makros in *Wincula* formuliert werden. So sieht man z.B., dass *Makrofunktionen* aufgerufen werden - in diesem Fall *GetToday* und *SetText.* Diese Funktionen werden individuell von jeder Anwendung bereitgestellt und können die verschiedensten Leistungen erbringen. Sie werden mit sogenannten *Parametern* aufgerufen, die in Klammern und durch Kommata getrennt an die Funktion übergeben werden.

Schließlich haben wir sogenannte *Variablen* verwendet. In diesem Fall wurde eine Zeichenfolgenvariable mit dem Namen *Datumspuffer* verwendet. Alle Variablen eines Makros müssen vor ihrer Verwendung *deklariert* werden. Diese Deklaration haben wir in diesem Beispiel der Einfachheit halber weggelassen. Die Deklaration von Variablen erfolgt immer am Anfang des Makros.

Nach dem Verständnis dieses einführenden Beispiels werden im Folgenden alle möglichen syntaktischen Komponenten eines Makros erläutert.

Die folgenden Kapitel erläutern - jeweils anhand von Beispielen untermauert - die verschiedenen syntaktischen Elemente, die ein einem Makro verwendet werden können.

2.2. Funktionsaufrufe

Zentraler Kern eines jeden Makros ist, wie auch obiges Beispiel zeigt, der Aufruf sogenannter Makrofunktionen. Funktionen werden in *Wincula* aufgerufen, indem der Name der Funktion, gefolgt von einer Liste von Parametern, die durch Kommata getrennt in runden Klammern eingeschlossen werden.

Parameter von Funktionen

Die Art und Anzahl der Parameter einer Funktion hängen von der Funktion selbst ab. Es gibt auch Funktionen ohne Parameter. Diese werden mit einer leeren Parameterliste aufgerufen, die einfach durch ein leeres Klammerpaar () symbolisiert wird. Beispiel:

```
GetScreenHeight ();
```

Parameter können sowohl dazu verwendet werden, Informationen in die Funktion hinein-, als auch wieder heraus zu transportieren. Hierzu ein Beispiel.

Der folgende Makrobefehl ruft die Funktion *AddFields* auf. Diese Funktion wird z.B. in Formularanwendungen benötigt. Sie saldiert alle Felder eines Formulars, deren Name mit der Zeichenfolge übereinstimmt, die als erster Parameter übergeben wird. Die Zeichenfolge *"A*"* bedeutet, dass alle Felder saldiert werden, die mit dem Buchstaben *A* beginnen. Nach erfolgter Berechnung schreibt die Funktion den ermittelten Saldo auf die Gleitkommavariable zurück, die als zweiter Parameter angegeben ist. In diesem Fall ist dies die Variable *Ergebnis*. Das Makro lautet also:


```
AddFields ("A*", Ergebnis);
```

Man beachte, dass auch hier die Deklaration der Gleitkommavariablen *Ergebnis* der Einfachheit halber weggelassen wurde.

Man beachte auch, dass mittels des ersten Parameters eine Information in die Funktion übergeben wird, nämlich der Name der Felder und mittels des zweiten Parameters eine Information aus der Funktion zurück in das Makro gelangt, nämlich die ermittelte Summe.

Solche Parameter, auf denen Information in das Makro zurücktransportiert wird, müssen sogenannte *Variablen* sein. Dies sind benannte Speicherplätze, auf denen das Makro Zwischenergebnisse ablegen kann. Solche Variablen müssen innerhalb eines jeden Makros deklariert werden, bevor sie verwendet werden, damit die Makrosprache weiß, welchen Datentyp (*Zeichenfolge, Ganzzahl, Gleitkommazahl, etc*.) diese Variablen haben. Deklarationen werden in einem späteren Kapitel behandelt.

Verfügbare Funktionen

Die Liste der Funktionen, die aufgerufen werden können, ist reichhaltig. Es stehen grundsätzlich zwei Arten von Funktionen zur Verfügung:

- Allgemeine Funktionen (können in allen Anwendungen verwendet werden)
- Anwendungsspezifische Funktionen

Die allgemeinen Funktionen sind unabhängig von der Anwendung und können daher in allen Anwendungen verwendet werden. Sie werden von der Makrosprache *Wincula* bereitgestellt. Die anwendungsspezifischen Funktionen hängen von der augenblicklichen Anwendung ab. Ein Formularprogramm stellt andere Funktionen zur Verfügung als ein Texteditor. Bitte entnehmen Sie die verfügbaren Funktionen einer Anwendung den entsprechenden Funktionslisten, die für jede Anwendung, die *Wincula* unterstützt, veröffentlicht werden.

Funktionswert einer Funktion

Eine Funktion liefert grundsätzlich einen Funktionswert zurück. Dieser ist immer eine ganze Zahl. Die Bedeutung des Funktionswertes hängt von der verwendeten Funktion ab. Der Funktionswert kann z.B. ein Fehlerschlüssel sein, der zurückmeldet, ob die Funktion erfolgreich ausgeführt werden konnte. Manche Funktionen liefern nützliche Informationen als Funktionswert zurück - so auch die bereits in einem Beispiel zitierte Funktion *GetScreenHeight*, welche ohne Parameter aufgerufen wird und als Wert die Höhe des Bildschirms in Pixeln zurückliefert.

Jede Funktion kann daher auch überall dort verwendet werden, wo normalerweise eine ganze Zahl stehen darf. Dies ist z.B. in einer Formel der Fall, deren Ergebnis auf eine Variable zugewiesen wird. Folgendes Beispiel ermittelt die Höhe des Bildschirms, multipliziert diese mit 2 und weist das Ergebnis auf die Ganzzahlvariable *DoppelteHöhe* zu:

```
DoppelteHöhe = GetScreenHeight () * 2;
```

Man beachte, dass der Funktionsaufruf in diesem Fall nicht mit einem Semikolon abgeschlossen wird, da der Makrobefehl hier nicht beendet wird, sondern der Funktionsaufruf einen Ersatz für eine Zahl bildet.

Das folgende Kapitel erläutert die Verwendung von Zuweisungen genauer.

2.3. Zuweisungen

Neben den Funktionsaufrufen stellen Zuweisungen eine wichtige syntaktische Komponente der Makrosprache *Wincula* dar. Eine Zuweisung hat eine rechte und eine linke Seite. Dazwischen steht immer ein Gleichheitszeichen. Die linke Seite einer Zuweisung ist immer eine Variable. Auf dieser Variablen wird das Ergebnis der Zuweisung abgespeichert.

Es sei nochmals daran erinnert, dass Variablen vor ihrer Verwendung unter Angabe eines Datentyps deklariert werden müssen. Deklarationen werden in einem späteren Kapitel behandelt.

Auf der rechten Seite einer Zuweisung muss etwas stehen, das einen Wert darstellt und somit zugewiesen werden kann. Das kann eine Konstante sein, z.B. eine ganze Zahl, eine Gleitkommazahl oder eine Zeichenfolge. Es kann aber auch ein berechneter Wert sein, der z.B. durch eine Formel berechnet wird. Schließlich kann es auch der Funktionswert einer zuvor aufgerufenen Funktion sein. Die folgenden Zuweisungen sind alle gültig:


```
Wert = 3;
Wert = 3 * 5;
Wert = 3 * TageBisWeihnachten;
Wert = GetScreenHeight ();
Wert = GetScreenHeight () * 3;
Gleit = 3,14 * 25,80;
```

Voraussetzung ist, dass die Variable *Wert* als ganzzahlige Variable und die Variable *Gleit* als Gleitkommazahl deklariert worden sind. Näheres über die Verwendung von Formeln wird in einem späteren Kapitel erläutert.

Der Typ der Variablen, auf die zugewiesen wird, muss zu dem Typ des Wertes auf der rechten Seite der Zuweisung passen. Wenn dies nicht der Fall ist, wird der Wert automatisch angepasst. Z.B. werden ganze Zahlen beim Zuweisen automatisch in Gleitkommazahlen umgewandelt. Es können allerdings nicht alle Typen automatisch gewandelt werden. Insbesondere lassen sich Textkonstanten nicht in Zahlen wandeln.

Jede Zuweisung muss durch ein Semikolon abgeschlossen werden.

Man beachte, dass eine Variable sowohl auf der linken als auch auf der rechten Seite einer Zuweisung auftauchen darf. Die Zuweisung:

```
Wert = Wert + 1;
```

erhöht die Zahl, die in der Variablen *Wert* gespeichert ist, um eins. Es wird nämlich immer erst die rechte Seite einer Zuweisung ausgerechnet und danach das Ergebnis auf die Variable auf der linken Seite zugewiesen.

2.4. Bedingte Befehle

Häufig ist es notwendig, Befehle nur dann auszuführen, wenn bestimmte Bedingungen erfüllt sind. Hierzu verwendet man bedingte Anweisungen. Bedingte Anweisungen werden durch das Schlüsselwort **if** eingeleitet. Danach folgt in Klammern die Bedingung, unter der die nachstehende Anweisung ausgeführt wird. Die Syntax ist also:

```
If (Bedingung)
        Funktionsaufruf ();
```

Die *Bedingung* ist ein logischer Ausdruck, z.B. ein Vergleich zweier Zahlen oder Variablen, der nur den Wert *wahr* oder *falsch* annehmen kann. Logische Ausdrücke werden in einem späteren Kapitel genauer erläutert.

Das folgende Beispiel lässt ein Piepen im Lautsprecher ertönen, falls auf der Variablen mit dem Namen *Wert* die Zahl 15 steht. Ansonsten bleibt der Lautsprecher stumm. Es wird also der logische Ausdruck *Wert == 15* verwendet werden, der wahr ist, wenn die Variable *Wert* den Wert 15 hat und sonst falsch:

```
If (Wert == 15)
        Beep (-1);
```

Das doppelte Gleichheitszeichen ist kein Schreibfehler, sondern unterscheidet die Abfrage auf Gleichheit von der Zuweisung, die ja ebenfalls durch ein Gleichheitszeichen formuliert wird.

Bedingte Befehle mit Blockanweisung

Was nun, wenn aufgrund der Bedingung nicht nur ein Befehl, sondern mehrere ausgeführt werden sollen? Kein Problem. In diesem Fall folgt der *if*-Anweisung statt eines einzelnen Befehls ein Block von Befehlen. Damit der Block als solcher erkennbar ist, wird er in geschweifte Klammern {} eingeschlossen.

Das folgende Beispiel schaltet, lässt den Lautsprecher dreimal ertönen, wenn die Variable *Wert* den Wert 15 hat:

```
If (Wert == 15)
        {
        Beep (-1);
        Beep (-1);
        Beep (-1);
        }
```

Abfrage des Gegenteils einer Bedingung

Die Makrosprache *Wincula* verfügt über kein direktes Sprachmittel, dass in anderen Programmiersprachen dem Schlüsselwort *else* entspricht. Wenn sowohl in dem Fall, dass eine Bedingung *wahr* ist, Befehle ausgeführt werden sollen und in dem Fall, dass sie *falsch* ist, andere Befehle ausgeführt werden sollen, kann das in *Wincula* nur durch folgende Konstruktion erfolgen:


```
If (Wert == 15)
    FunktionWennWert15Ist ();
if (Wert != 15)
    FunktionWennWertNicht15Ist ();
```

Es wird also eine zweite *if*-Anweisung formuliert, die die gegenteilige Abfrage abhandelt. In diesem Fall also geprüft, ob der Wert der Variablen *Wert* ungleich 15 ist. Die Abfrage auf *ungleich* ist hier syntaktisch ebenfalls vorweggenommen. Logische Abfragen werden, wie bereits gesagt, in einem späteren Kapitel erörtert.

2.5. Schleifen

Alle bisherigen Erörterungen und Beispiele basieren auf der Grundlage, dass alle Befehle - auch die bedingten - der Reihe nach von oben nach unten abgearbeitet werden, ohne dass irgendwelche Wiederholungen stattfinden.

Auf Wunsch lassen sich jedoch einzelne Befehle oder Gruppen von Befehlen auch mehrfach durchführen. Die Frage nach der Häufigkeit einer Wiederholung hängt dabei ganz von der Aufgabenstellung des Makros ab.

Für die wiederholte Ausführung von Befehlen wird das Schlüsselwort *while* verwendet. Mittels einer *while*-Anweisung kann eine sogenannte *Schleife* programmiert werden. Die Syntax einer *while-Schleife* ist:

```
while (Bedingung)
    {
    WiederholtAusgeführteFunktion1 ();
    WiederholtAusgeführteFunktion1 ();
    }
```

Die Syntax ähnelt also sehr einer bedingten Anweisung mit *if*. Bei der Programmierung von Schleifen ist es natürlich sehr wichtig, dass sich während des Durchlaufens der Schleife die Bedingung ändert. Sollte dies nicht der Fall sein und die Bedingung ist auch nur ein einziges Mal *wahr*, so wird die Schleife bedauerlicherweise *unendlich oft* durchlaufen, wie z.B. die folgende Schleife, die nicht zur Nachahmung empfohlen wird:


```
Wert = 15;
while (Wert == 15)
    {
    Beep (-1);
    }
```

Der Effekt dieses Makros wird den Lautsprecher des Rechners ziemlich beanspruchen. Besser ist es also, eine Schleife zu *kontrollieren*. Dies geschieht in der Regel mit einer *Kontrollvariablen*. Dies ist eine ganzzahlige Variable, die zu Beginn der Schleife auf einen definierten *Anfangswert* gesetzt wird und die innerhalb der Schleife schrittweise verändert wird.

Das folgende Beispiel formuliert eine Schleife, die insgesamt 10 Pieptöne abgibt:
Wert den Wert 15 hat:

```
Wert = 1;
while (Wert <= 10)
    {
    Beep (-1);
    Wert = Wert + 1;
    }
```

In der *while*-Bedingung wird der bisher unbekannte logische Ausdruck *Wert <= 10* verwendet. Dieser ist wahr, wenn die Variable *Wert* einen Wert enthält, der kleiner oder gleich 10 ist. Logische Ausdrücke werden in einem späteren Kapitel erläutert.

Das Makro zeigt eindrucksvoll, wie Schleifen formuliert werden können und somit Befehle mehrfach ausgeführt werden können.

Abbrechen von Schleifen durch den Befehl *break*

Manchmal entsteht in der Praxis der Wunsch, eine Schleife während eines Durchlaufs abzubrechen, ohne abzuwarten, ob die Schleifenbedingung beim nächsten Durchlauf noch erfüllt ist oder nicht. Dies ist z.B. dann der Fall, wenn etwas Unvorhergesehenes eintritt. In diesem Fall kann eine Schleife durch das Schlüsselwort *break* sofort beendet werden. Das Schlüsselwort *break* wird wie ein Funktionsaufruf verwendet und durch ein Semikolon abgeschlossen. Es ist aber keine wirkliche Funktion und erhält daher auch keine Klammern.

Das folgende Beispiel öffnet in einer Schleife mittels der Funktion *MessageDialog* eine Dialogbox, die der Benutzer mit *OK* oder *Abbruch* beantworten kann. Wenn der Benutzer mit OK antwortet, liefert die Funktion den Funktionswert *OK*. Ansonsten wird der Wert

ABORT als Funktionswert zurückgeliefert. Wenn mit *Abbruch* geantwortet wird, wird auch die Schleife abgebrochen. Ansonsten werden weitere Befehle in der Schleife ausgeführt:

```
while (1)
    {
    if (MessageDialog ("Ich bin die Frage in der Box", INFO, OK) != OK)
                break;
    WeitereFunktion ();
    }
```

Es fällt auf, dass die Schleifenbedingung eine Konstante ist. Dies bedeutet, dass die Bedingung immer den Wert *wahr* hat. Die Schleife würde also theoretisch unendlich oft laufen. Dadurch, dass in dem Fall, dass die Funktion *MessageDialog* einen Wert liefert, der nicht *OK* ist, wenn der Dialog abgebrochen wird, kommt die Schleife dann durch die Ausführung des Befehls *break* doch zum Ende.

2.6. Abbrechen von Makros durch den Befehl *exit*

Bei der Behandlung von Schleifen im vorigen Kapitel wurde ein Schlüsselwort vorgestellt, mit dem eine Schleife jederzeit abgebrochen werden kann, falls etwas Unvorhergesehenes passiert.

Der gleiche Mechanismus ist auch für das Abbrechen des gesamten Makros vorgesehen. Dies geschieht durch das Schlüsselwort *exit***.** Der Befehl *exit* wird wie eine Funktion aufgerufen und durch ein Semikolon abgeschlossen. Da es sich nicht um eine wirkliche Funktion handelt, werden keine Klammern angegeben.

Wenn ein Makro mittels des Befehls *exit* abgebrochen wird, werden automatisch alle Dateien, die während dieses Makros durch Makrofunktionen (z.B. *OpenFile)* geöffnet wurden, geschlossen.

2.7. Deklaration von Variablen

Während der Erläuterung der Syntax der Makrosprache *Wincula* in den vorangegangenen Kapiteln wurde immer wieder von Variablen gesprochen. Variablen sind benannte Speicherplätze, auf denen innerhalb eines Makros Zahlen und Zeichenfolgen zwischengespeichert werden können.

Manche Konstruktionen erfordern die Verwendung von Variablen. Hierzu gehören z.B. die Zuweisungen, die in einem eigenen Kapitel behandelt werden. Auch werden Variablen

dann benötigt, wenn eine Funktion über einen Parameter einen Wert nach außen zurückliefert.

Bevor eine Variable verwendet werden kann, muss diese deklariert werden. Die Deklaration erfolgt durch ein Schlüsselwort, welches auch den Variablentyp festlegt. Nach dem Schlüsselwort folgt ein Leerzeichen oder Tabulator. Danach wird der Name der Variablen genannt. Die ganze Anweisung wird durch ein Semikolon abgeschlossen.

Der Name der Variablen darf weder mit einem Funktionsnamen oder Schlüsselwort identisch sein, noch darf derselbe Name in einer anderen Deklaration desselben Makros verwendet werden.

Wincula unterscheidet drei Basistypen von Variablen. Die Deklaration dieser Variablen wird in den folgenden Kapiteln erläutert.

Ganzzahlen

Die Deklaration einer *ganzzahligen* Variablen erfolgt durch das Schlüsselwort *int*. Eine ganzzahlige Variable kann Zahlenwerte zwischen -1.000.000.000 und 1.000.000.000 aufnehmen.

Gleitkommazahlen

Die Deklaration einer Gleitkommavariablen erfolgt durch das Schlüsselwort *float*. Auf Gleitkommavariablen können Zahlen bis zu einer Genauigkeit von 14 Stellen abgespeichert werden. Hierbei werden Vor- und Nachkommastellen gemeinsam betrachtet. Wenn die Zahlen größer werden, als diese Stellenzahl, geht Genauigkeit verloren. Dies dürfte in der Praxis kaum auftreten.

Zeichen

Die Deklaration einer Zeichenvariablen erfolgt durch den Befehl *char*. Eine Zeichenvariable kann ein beliebiges Zeichen, wie z.B. einen Buchstaben, eine Ziffer oder ein anderes Zeichen aufnehmen. Es können auch Zahlenwerte zwischen -127 und 127 auf einer Zeichenvariablen gespeichert werden.

Zeichenvariablen werden in der Regel indiziert, da in ihnen nicht nur ein einzelnes Zeichen, sondern eine ganze Zeichenfolge gespeichert werden soll. Das Indizieren einer Variablen bedeutet, dass die Variable nicht nur einmal vorhanden ist und genau ein Zeichen aufnehmen kann, sondern dass sie mehrfach vorhanden ist und mehrere Zeichen aufnehmen kann. Der Vorgang des Indizierens wird im nächsten Kapitel beschrieben.

Indizierte Variablen und Variablen für Zeichenfolgen

In der Makrosprache *Wincula* lassen sich neben den sogenannten einfachen Variablen auch indizierte Variablen anlegen. Einfache Variablen können jeweils nur einen Wert ihres Typs aufnehmen. Indizierte Variablen erlauben es, unter einem Namen mehrere Speicherzellen des entsprechenden Typs anzusprechen. Die Speicherzellen sind durchnummeriert und werden als Speicherzelle 0 bis ... angesprochen.

Die Deklaration von indizierten Variablen erfolgt wie bei einfachen Variablen unter Angabe des Variablennamens. Dahinter wird in eckigen Klammern die Anzahl Speicherzellen angegeben, die benötigt wird, z.B. in der Form:

```
int Name [100];
char Name [100];
```

Welche Bedeutung haben solche indizierten Variablen? Dazu sei zunächst gesagt, dass die Verwendung von indizierten numerischen Variablen recht selten ist. Viel häufiger werden hingegen die indizierten Zeichenvariablen verwendet, da diese als Variablen zur Aufnahme von Zeichenfolgen, also der Aufnahme mehrerer Zeichen, dienen.

Dazu sei das Beispielmakro noch einmal in Erinnerung gerufen, das wir ganz zu Anfang verwendet haben, um das aktuelle Tagesdatum in formatierter Form als Text in ein Feld mit dem Namen *Datum* zu transportieren. Im Gegensatz zu den früheren Beispielen ist jetzt auch die Variablendeklaration ergänzt:

```
char Datumspuffer [100];

GetToday (Datumspuffer);
SetText ("Datum", Datumspuffer);
```

Die Funktion *GetToday* ermittelt das augenblickliche Tagesdatum als formatierten Text und speichert dieses auf der indizierten Zeichenvariablen *Datumspuffer* als Text ab. Die Variable kann eine Zeichenfolge von bis zu 100 Zeichen aufnehmen, da sie entsprechend indiziert ist.

Anschließend wird die Variable benutzt, um den Text an die Funktion *SetText* zu übergeben. Hundert Zeichen sollten genug für ein Datum sein. Wenn es nötig wird, können aber auch Zeichenfolgen von einigen zehntausend Zeichen angelegt werden.

Es kann immer nur ein Zeichen weniger auf einer indizierten Zeichenfolgenvariablen abgespeichert werden, als bei der Deklaration angegeben ist. Wenn als Index also eine 100 angegeben ist, können bis zu 99 Zeichen gespeichert werden. Der Grund hierfür ist, dass

intern jede Zeichenfolge automatisch mit einem *Bremszeichen* versehen wird. Dieses Zeichen zeigt das Ende der Zeichenfolge an, falls die Zeichenfolge kürzer als die Maximallänge der Variablen ist. Das *Bremszeichen* ist immer eine Null.

2.8. Zugriff auf indizierte Variablen

In den meisten Fällen werden indizierte Variable nur als Ganzes angesprochen - also z.B. als Puffer für mehrere Zeichen, wie in den Beispielen der vorangegangenen Kapitel.

Für besonders knifflige Programmierungen kann jedoch auch jede Zelle (bzw. jedes Zeichen, wenn es sich um eine Zeichenvariable handelt) einer indizierten Variablen einzeln angesprochen und verändert werden. Wie genau, erläutert dieses Kapitel.

Es sei insbesondere in diesem Zusammenhang darauf hingewiesen, dass natürlich nicht nur Variablen vom Typ *char* indiziert werden können, sondern auch solche vom Typ *int* oder *float*.

Der Zugriff auf einzelne Speicherzelle einer indizierten Variablen erfolgt durch Angabe der Nummer der Speicherzelle in eckigen Klammern hinter dem Variablennamen. Die Speicherzellen werden ab Null nummeriert, so dass die Speicherzelle mit der Nummer Null die erste aller Zellen meint.

Durch das folgende Makro wird auf der indizierten Zeichenvariablen *Text* die Zeichenfolge "ABC" zusammengesetzt:

```
char Text [100];

Text [0] = 'A';
Text [1] = 'B';
Text [2] = 'C';
Text [3] = 0;
```

Es werden zunächst nacheinander die Buchstabenkonstanten 'A', 'B' und 'C' auf die ersten drei Speicherzellen der indizierten Zeichenvariablen *Text* zugewiesen. Dann wird das *Bremszeichen* Null auf die nächste Speicherzelle zugewiesen, damit das Ende des Textes angezeigt wird. Dies ist nötig, da die Variable ja 100 Zellen lang ist und *Wincula* wissen muss, wo der Text innerhalb dieser 100 Zellen endet.

Das folgende Beispielmakro besetzt die Zeichenfolgenvariable *Quelle* mit dem Text *"Test"* vor und kopiert diese Zeichenfolge dann auf die Zeichenfolgenvariable *Ziel:*


```
char Quelle [100];
char Ziel [100];

Quelle ="Test";

Index = 0;
while (Quelle [Index] != 0)
    {
    Ziel [Index] = Quelle [Index];
    Index = Index + 1;
    }
Ziel [Index] = 0;
```

Die Schleifenkontrollvariable *Index* läuft von Null bis zu der Stelle, an der in der Quellzeichenfolge das Zeichen Null, also die Bremse, auftritt. In der Schleife wird das in der Quellzeichenfolge gelesene Zeichen in die Zielzeichenfolge kopiert. Schließlich wird nach Durchlauf der Schleife auch die Zielzeichenfolge mit der notwendigen Bremse versehen.

Man beachte, dass das gesamte Makro nichts anderes tut, als die Zuweisung:

```
Ziel = Quelle;
```

Das Makro dient also vornehmlich dem Verständnis. Interessant ist auch die Zeile:

```
Quelle = "Test";
```

In der die Zeichenfolgenvariable *Quelle* mit dem Text *"Test"* vorbesetzt wird. Hier liegt eine Zuweisung vor, in der ausnahmsweise keine Zahl auf eine Zahlvariable, sondern eine konstante Zeichenfolge, also ein Text, auf eine Zeichenfolgenvariable zugewiesen wird.

2.9. Konstanten

In allen vorangegangenen Kapiteln wurden, ohne näher darauf einzugehen, immer wieder Konstanten aller Art verwendet. Bevor das Kapitel über die Syntax von *Wincula* abgeschlossen wird, soll aber doch noch genau definiert werden, was Konstanten sind, welche Arten von Konstanten es gibt und wie sie verwendet werden können.

Der wichtigste Unterschied von Konstanten zu Variablen ist, wie der Name bereits sagt, dass sie ihren Wert nicht verändert können. Der Wert einer Konstanten steht mit dem Hinschreiben derselben fest. Konstanten können daher nur überall dort eingesetzt werden, wo sich nichts verändert.

Z.B. darf eine Konstante nicht als Parameter an eine Funktion übergeben werden, auf den die Funktion etwas in das Makro zurückübergibt. Auf solchen Parametern muss unbedingt eine Variable angegeben werden. Hingegen können auf allen Parametern, die etwas an die Funktion übergeben, sowohl Konstanten, wie auch Variablen verwendet werden.

Wincula kennt vier Arten von Konstanten: *Ganzzahlen*, *Gleitkommazahlen*, *Zeichen* und *Zeichenfolgen*.

Ganzzahlen

Ganzzahlen werden als Ziffernfolge dargestellt. Wenn die Zahl negativ sein soll, geht diesen ein Minuszeichen voran.

Gleitkommazahlen

Gleitkommazahlen werden ebenfalls als Ziffernfolge dargestellt. Die Nachkommastellen werden mit einem *Punkt* als Dezimaltrenner abgetrennt. Selbstverständlich können auch Gleitkommazahlen durch ein vorangestelltes Minuszeichen negativ sein.

Zeichen

Zeichenkonstanten umfassen genau ein Zeichen oder einen Buchstaben. Das Zeichen muss zur syntaktischen Eindeutigkeit in einfache Apostrophe eingeschlossen werden. Anstelle der Zeichenkonstante kann auch der ASCII- oder ANSI-Wert des Zeichens als Ganzzahl verwendet werden kann, also z.B. 65 anstelle von 'A'. Für manche Zeichen muss aus Gründen der syntaktischen Eindeutigkeit sogar der Dezimalwert verwendet werden, wie z.B. für das Semikolon.

Zeichenfolgen (Texte)

Zeichenfolgenkonstanten - oder schlicht *Texte* - werden als Text hingeschrieben, der in doppelte Anführungszeichen eingeschlossen ist.

Für die Codierung von Steuerzeichen können sogenannte Ersatzdarstellungen in Texten verwendet werden. Immer wenn das Zeichen \ in einer Zeichenfolgenkonstanten auftritt, wird eine Ersatzdarstellung eingeleitet. Soll das Zeichen \ selbst abgelegt werden, so muss anstelle dessen seine Verdoppelung \\ hingeschrieben werden. Folgende Ersatzdarstellungen werden erkannt:

\n	Zeilenschaltung

\r Wagenrücklauf
\t Tabulator

Falls das Zeichen \ häufig vorkommt, kann dieses mittels der Makrofunktion *SetStringEscape* auch umdefiniert und auf ein anderes Zeichen, welches weniger häufig vorkommt, abgebildet werden. Dann übernimmt ein anderes Zeichen die Funktion des \.

Beispiele von Konstanten

Folgende Zuweisungen enthalten gültige Konstanten aller vier Typen:

```
int Ganzzahl;
float Gleitkommazahl;
char Zeichen;
char Zeichenfolge [100];

Ganzzahl = 100;
Gleitkommazahl = 34.50;
Zeichen = 'A';
Zeichenfolge = "TEXT";
```

Systemkonstanten

Neben den oben genannten Konstanten stellt *Wincula* eine Reihe von benannten Konstanten zur Verfügung, die in verschiedenen Funktionen benötigt werden. Die wichtigsten dieser Konstanten sind:

Dialoge und Dialogfunktionen

```
OK
ABORT
INFO
NO
QUEST
EXCLAIM
EXCL
YES
STOP
```


Funktionen für den Zugriff auf die Registry

```
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
```

Dateifunktionen

```
READ
WRITE
```

2.10. Formeln

Wincula verfügt über einen eigenen Formelinterpreter. Dieser gestattet es, überall dort, wo eine Zahlkonstante angegeben werden kann, auch eine Formel hinzuschreiben.

Eine Formel ist ein arithmetischer Ausdruck - z.B. eine Multiplikation. Es können die vier Operatoren der Grundrechenarten Plus (+), Minus (-), Mal (*) und Dividieren durch (/) verwendet werden.

Bei den arithmetischen Operationen gelten die üblichen Vorrangregeln, wie z.B. das die Multiplikation und Division Vorrang vor der Addition und Multiplikation haben. Es können jedoch beliebige Klammern gesetzt werden, um die Reihenfolge der Berechnung zu definieren.

Beispiele für Formeln, die in Zuweisungen auf der rechten Seite stehen:

```
Wert = 3 + 5;
Wert = 3 * 8;
Wert = 3 - 7 + 2;
Wert = 3 * Anzahl + 5;
Wert = (3 * (8 + Warenwert)) / Einkaufspreis
```

Wenn eine Formel Gleitkommazahlen enthält, ist das Ergebnis der Formel auch eine Gleitkommazahl. Wenn hingegen die Formel nur ganze Zahlen enthält, dann ist das Ergebnis der Formel eine ganze Zahl.

Bitte beachten Sie, dass bei Zuweisungen auf Variablen eines anderen Typs automatisch eine Typkonvertierung durchgeführt wird. Beim Zuweisen einer Gleitkommazahl auf eine ganze Zahl wird der Nachkommaanteil abgeschnitten.

2.11. Bedingungen und logische Ausdrücke

Wincula verfügt, wie in den vergangenen Kapiteln beschrieben, über die Möglichkeit, durch die **if**-Anweisung bedingte Befehle auszuführen und durch die *while*-Anweisung Schleifen zu formulieren. Diese beiden Anweisungen erfordern jeweils eine sogenannte *Bedingung*.

Bedingungen sind logische Ausdrücke, die entweder den Wert *wahr* oder *falsch* annehmen können. Dieses Kapitel widmet sich den Möglichkeiten, Bedingungen und damit logische Ausdrücke zu formulieren.

Die am häufigsten verwendeten logischen Ausdrücke sind Vergleichsoperationen zwischen zwei Zahlen. Diese wurden in den vorangegangenen Beispielen bereits mehrfach verwendet. Eine Vergleichsoperation besteht aus einem Vergleichsoperator und zwei Operanden. Die Operanden sind die beiden zu vergleichenden Zahlen. Sie stehen links und rechts des Operators. Der Operator bestimmt, welcher Art der Vergleich ist. Folgende Vergleichsoperatoren können verwendet werden:

==	prüft, ob der linke und rechte Operand gleich sind
!=	prüft, ob der linke und rechte Operand ungleich sind
<=	prüft, ob der linke Operand kleiner oder gleich dem rechten ist
>=	prüft, ob der linke Operand größer oder gleich dem rechten ist
<	prüft, ob der linke Operand kleiner als der rechte ist
>	prüft, ob der linke Operand größer als der rechte ist

Wenn eine der beiden zu vergleichenden Größen eine Gleitkommazahl ist, so wird der Vergleich auf Gleitkommabasis durchgeführt.

Jede der Operationen liefert als Ergebnis entweder den Wert *wahr* oder *falsch*. Beispiele von Operationen sind:

3 == 5	liefert den Wert *falsch*
3 != 5	liefert den Wert *wahr*
3 < 5	liefert den Wert *wahr*
5 >= 5	liefert den Wert *wahr*

Darüber hinaus gibt es auch Operationen die es ermöglichen, mehrere Vergleichsoperationen mit einer *und*- bzw. mit einer *oder*-Anweisung zu verbinden bzw. mit einer *nicht*-Anweisung zu verneinen. Die *und*- bzw. *oder*-Verbindung hat als linken bzw. rechten Operanden eine Vergleichsoperation. Die zugehörigen Operatoren heißen:

&& Die Bedingung ist wahr, wenn der linke *und* der rechte Operand wahr sind.
|| Die Bedingung ist wahr, wenn der linke *oder* der rechte Operand wahr sind.

Die *nicht*-Anweisung bezieht sich immer auf die nachfolgende Vergleichsoperation und verneint diese. Sie wird durch das Ausrufezeichen **!** dargestellt. Folgende Beispiele erläutern die logischen Operationen in Verbindung mit den Vergleichsoperationen:

x == 3 || y == 4 Die Bedingung ist wahr, wenn die Variable *x* den Wert *3* oder die Variable *y* den Wert *4* hat.

x > 5 && x < 7 Die Bedingung ist wahr, wenn die Variable zwischen den Werten *5* und *7* liegt.

Es können, wie bei arithmetischen Operationen, auch beliebige Klammern gesetzt werden, um die Reihenfolge zu definieren, in der Operationen durchgeführt werden. Insbesondere bei der *nicht*-Anweisung ist das Klammern immer anzuraten:

! (a == 3) Die Variable *a* soll nicht den Wert *3* haben.

Diese Bedingung ist selbstverständlich identisch mit der Bedingung *(a != 3).*

Die genannten logischen Operationen und Vergleichsoperationen ermöglichen eine umfangreiche Formulierung aller möglichen Abfragen und machen die Verwendung der Befehle *if* und *while* überschaubar und verständlich.

2.12. Rahmen für Makros in Dateien

In der Einleitung zu dieser Dokumentation wird darauf hingewiesen, dass Makros sowohl in eingebetteter Form als auch in Dateien abgelegt werden können. Wenn Makros in Dateien abgelegt werden, sollten die eigentlichen Befehle des Makros in einen Standardrahmen eingebettet werden. Dieser lautet:

```
Macro "Name"

    {
    eigentliche Befehle ...
    }
```

Dieser Rahmen dient einerseits dazu, Dateien, die Makros beinhalten, an ihrem Inhalt als solche zu erkennen und die Ausführung von Dateien, die eigentlich keine Makros sind, zu verweigern.

Andererseits erhält das Makro durch diesen Rahmen einen Namen, der von der Anwendung gegebenenfalls ausgewertet werden kann.

Der Rahmen muss bei eingebetteten Makros weggelassen werden. Er wird nur bei Makros benötigt, die in Dateien abgelegt werden.

