7.4.6 Embedded-Array – Eingebettetes Array

Dynamische Arrays sind echte Gambas-Objekte, während eingebettete (embedded) Arrays eine Möglichkeit darstellen, *auf einen Teil eines Objekts als Array zuzugreifen*. Dem Array-Inhalt ist in diesem Fall ein Speicherblock alloziert – genau dort, wo das eingebettete Array objekt-bezogen deklariert wurde. Die Quintessenz ist, dass in Gambas normalerweise alle Eigenschaften von einem Objekt über Referenzen erreichbar sind, also außerhalb dieses Objekts liegen. Bei eingebetteten Arrays (daher auch der Name) ist das nicht der Fall. Diese Arrays bekommen bei der Definition der Klasse eine feste Größe und befinden sich dann direkt als Speicherblock in der Beschreibung der Klasse, wie sie dem Interpreter vorliegt.

Syntax für ein Embedded-Array:

```
[ STATIC ] { PUBLIC | PRIVATE } Identifier [ Array dimensions ... ] AS NativeDatatype
```

- Ein eingebettetes Array ist ein Array, das direkt einem Objekt zugeordnet ist.
- Ein eingebettetes Array kann man nicht initialisieren.
- Ein eingebettetes Array kann nicht geteilt werden und wird zerstört, wenn das assoziierte Objekt zerstört wird.
- Die Größe eines eingebetteten Arrays muss bei der Kompilierung bekannt sein.
- Bei einem Embedded-Array bezieht sich "Embedded" auf die Ablage des Arrays im Speicher!
- In Gambas3 können eingebettete Arrays nicht als lokale Variablen verwendet werden – aber sie können öffentlich sein!

Da eingebettete Arrays keine Objekte sind, Gambas jedoch nur mit Objekten arbeiten kann, beschreitet man einen anderen Weg und erstellt ein temporäres Objekt, das dem Interpreter zeigt, wie man mit solchen Arrays arbeiten kann, ohne dass sie jedoch vollwertige Objekte sind. Eingebettete Arrays können wie bereits o.a. nicht als *lokale* Variablen deklariert werden, da jeder Wert auf dem Gambas-Stack (dort liegen die lokalen Variablen) nur einen bestimmten Speicherbereich verbrauchen darf (32 Bytes auf einem 64-Bit-System) und Arrays können größer sein!

Die besondere Stellung von Embedded-Arrays zeigt sich auch in einem Beitrag von Emil Lenngren. Er schreibt: "Eingebettete Arrays verbrauchen weniger Speicher, da ihre Daten direkt in der Klassenstruktur alloziert sind. Aber sie sind langsamer, weil für jeden Zugriff ein Wrapper-Objekt erstellt werden muss. Ich empfehle, dass Sie auf eingebettete Arrays verzichten. Sie sind nützlich, wenn Sie eine Struktur erstellen müssen, die ein Array enthält, das Sie später einer *externen C-Funktion* übergeben wollen – aber sonst nicht."

Ähnlich äußerte sich Benoît Minisini: "Eingebettete Arrays wurden eingeführt, um die Möglichkeit zu haben, eine Speicherstruktur zu imitieren, die in einer C-Bibliothek deklariert wurde (In C sind alle Strukturelemente als Block innerhalb der Struktur alloziert - wie es in Gambas normalerweise nicht ist – außer bei eingebetteten Arrays). Sonst gibt es keinen echten Grund, sie zu nutzen."

Eingebettete Arrays sind – wie oben von *Lenngren* und *Minisini* betont – Randerscheinungen.

Trotzdem soll mit einem Beispiel der (formale) Einsatz eines eingebetteten Arrays gezeigt werden:

```
PUBLIC embArray[3, 3, 3] As Integer
...
Public Sub btnShowEmbeddedArray_Click()
  Dim i, ii, iii As Integer
' Dim embArray[3, 3, 3] As Integer ' → FEHLER!

  For i = 0 To 2
    For ii = 0 To 2
      For iii = 0 To 2
        Print "[ "; i; " | "; ii; " | "; iii; " ]"; " = ";
        embArray[i, ii, iii] = i * 9 + ii * 3 + iii
        Print embArray[i, ii, iii]
      Next
    Next
  Next

End ' btnShowEmbeddedArray_Click()
```

Ausgabe in der IDE-Konsole:

```
[ 0 | 0 | 0 ] = 0
[ 0 | 0 | 1 ] = 1
[ 0 | 0 | 2 ] = 2
[ 0 | 1 | 0 ] = 3
...
[ 1 | 0 | 0 ] = 9
[ 1 | 0 | 1 ] = 10
[ 1 | 0 | 2 ] = 11
...
[ 2 | 2 | 0 ] = 24
[ 2 | 2 | 1 ] = 25
[ 2 | 2 | 2 ] = 26
```