

PureBasic Pointer Tutorial - Inhalt

- | **Vorwort**
- | **Kapitel 1: Grundlagen**
 - | Einleitung
 - | Was genau ist ein Pointer?
 - | Probleme bei der Verwendung von Pointern
 - | Pointer in PureBasic
 - | Zusammenfassung
- | **Kapitel 2: Pointer auslesen**
 - | Einleitung
 - | Nutzen
 - | Notation
 - | Pointer zu Variablen
 - | Pointer zu Strukturen
 - | Pointer zu Elementen in Strukturen
 - | Pointer zu Arrays
 - | Pointer zu Linked Lists
 - | Pointer zu Prozeduren
 - | Pointer zu Labels
 - | Zusammenfassung
- | **Kapitel 3: Pointer Setzen**
 - | Einleitung
 - | Grundprinzip
 - | Notation
 - | Pointer zu Strukturen
 - | Pointer zu einzelnen Variablen
 - | Pointer zu Speicherbereichen
 - | Strukturen als Parameter bei Prozeduren
 - | Strukturen als Rückgabewerte von Prozeduren
 - | Zusammenfassung

Vorwort:

Worum geht es hier?

Da das Thema Pointer (deutsch: Zeiger) zu sehr vielen Fragen im PureBasic Forum geführt hat, wurde ich gebeten, einmal ein Tutorial zu schreiben, dass dieses Thema ausführlich erklärt. Ich habe versucht, die Erklärungen möglichst einfach und verständlich zu gestalten, so dass man ihnen, auch mit etwas weniger Vorkenntnissen, was Computer und Programmierung angeht, leicht folgen kann.

Wer sich jetzt fragt, was eigentlich PureBasic überhaupt ist, der sollte mal hier vorbeischauchen:
<http://www.PureBasic.com>

Vorraussetzungen:

Zum Verständnis dieses Tutorials sollte man die grundsätzlichen Regeln der PureBasic-Programmierung verstanden haben. Man sollte also schon ein paar kleinere Programme mit PureBasic erstellt haben, und damit etwas vertraut sein.

Insbesondere über die verschiedenen Typen von Variablen, über die Verwendung von Strukturen und Prozeduren sollte man einige Kenntnisse haben.

Info: In den Beispielen arbeite ich oft mit den Debug-Befehlen, um einfach Werte und Ergebnisse darzustellen. Es ist deshalb wichtig, das die Beispiele mit eingeschaltetem Debugger ausgeführt werden.

Über das Tutorial:

Dieses Tutorial ist ausdrücklich dafür bestimmt, weiterverbreitet zu werden. Ich bin also jedem dankbar, der es in irgend einer Form weitergibt, oder auf seiner Homepage zum Download anbietet.

Wer noch Fragen zum Thema hat, oder Verbesserungsvorschläge, was dieses Tutorial betrifft, der kann diese per Email (freak@abwesend.de) oder im PureBasic Forum (<http://www.Pure-Board.de>) an mich (Freak) stellen.

[[Zum Kapitel 1](#)]

By Timo 'Freak' Harter - Freak@abwesend.de

PureBasic Pointer Tutorial - Kapitel 1: Grundlagen

Einleitung:

Den Arbeitsspeicher, in dem ein Programm abläuft, kann man sich wie eine Datei vorstellen. Es ist ein speziell für das Programm reservierter Bereich, in dem der Programmcode und alle Variablen geladen werden.

Die verschiedenen Teile dieses Speichers kann man über Adressen ansprechen. Die Adresse ist nichts anderes, als der Punkt, an dem sich die gesuchten Daten im Speicher befinden, gemessen, vom Anfang des für dieses Programm reservierten Speichers. Von dieser relativen Speicheradressierung merkt der Programmierer allerdings nichts. Windows simuliert für die einzelnen Programme einen mit 32Bit adressierbaren zusammenhängenden Speicher, selbst wenn dieser quer über den physikalischen Arbeitsspeicher verteilt ist. Wichtig ist nur, dass man nur auf den Arbeitsspeicher, der dem eigenen Programm zugeteilt ist direkten Zugriff hat.

Mit Hilfe der Poke() und Peek() Befehle kann man zum Beispiel diesen Speicher direkt manipulieren.

Was genau ist jetzt ein Pointer?

Nun, da jede Variable, und jede Linked List und auch alle anderen Speicherelemente immer eine Adresse besitzen, und es in einigen Fällen auch sehr Praktisch ist, diese zu kennen, hat man die Pointer eingeführt. Sie sind nichts anderes, als die Adresse zu irgend einem Element im Speicher des Programms. Man muss sich einfach im Klaren darüber sein, dass man, wenn man von einem Pointer spricht, eben nicht den Inhalt einer Variable, oder eines sonstigen Elementes spricht, sondern nur von seiner Adresse.

Man kann auch Pointer von Elementen verändern. Diese verweisen dann auf einen anderen Speicherbereich, als vorher.

Ein Pointer ist immer eine 32Bit Adresse, das heißt, man kann diese Adressen in LONG Variablen speichern.

Probleme bei der Verwendung von Pointern:

Ein häufiges Problem entsteht dadurch, dass man bei der Arbeit mit Pointern meist direkte Manipulationen im Arbeitsspeicher vornimmt. Bei einem Fehler im Programm kann dies schnell zu einem Absturz des Programms kommen. Der PureBasic Debugger erkennt diese Arten von Fehlern meist nicht, was die Suche nach einem derartigen Fehler erschwert.

Aus diesem Grund empfehle ich, Pointer nur da im Programm zu verwenden, wo sie sinnvoll und notwendig sind. Je weniger Pointer man verwendet, desto leichter lässt sich eine mögliche Fehlerquelle in Verbindung damit finden.

Die Tatsache, dass dies leicht zu Programmabstürzen führt sollte einen aber nicht davon abhalten, sich einmal mit Pointern zu beschäftigen, denn richtig eingesetzte Pointer geben dem Programmierer eine Menge neuer Möglichkeiten.

Pointer in PureBasic:

Viele Programmierer, die bereits in anderen Sprachen mit Pointern gearbeitet haben, haben Probleme damit, die Art, wie Pointer in PureBasic funktionieren zu verstehen. Das Problem ist, dass zum Beispiel in C die Pointer Handhabung noch um ein vielfaches flexibler ist, als in PureBasic. Ich versuche, hier in diesem Tutorial, alles zu beschreiben, was man in PureBasic mit Pointern machen kann. Was hier nicht beschrieben steht, ist einfach so nicht möglich.

Zusammenfassung:

- | Pointer sind Speicheradressen von Elementen im Arbeitsspeicher eines Programms
- | Pointer sind 32Bit Adressen, und können als LONG gespeichert werden.

- ▮ Programmieren mit Pointern macht die Fehlersuche etwas schwerer.

[\[Zum Inhalt \]](#)

[\[Zum Kapitel 2 \]](#)

By Timo 'Freak' Harter - Freak@abwesend.de

PureBasic Pointer Tutorial - Kapitel 2: Pointer auslesen

Einleitung:

Die einfachste Form der Verwendung von Pointern, ist das Auslesen der Speicheradresse von Elementen. Wichtig hierbei ist, dass das ganze statisch ist, das heißt man kann diese Adresse nur lesen, nicht aber verändern. Diese Art der Handhabung wird im **Kapitel 3** beschrieben.

Nutzen:

Hat man die Speicheradresse, von zum Beispiel einer LONG Variable, so kann man diese direkt mit PokeL() verändern. Dies ist zwar meistens nicht besonders nützlich, aber in manchen Fällen schon. Häufiger findet diese Art und Weise Nutzen bei der Programmierung mit der Windows API (=Application Programming Interface, das sind von Windows bereitgestellte Funktionen, die man mit PureBasic verwenden kann. Das sind all die Funktionen mit einem Unterstrich.) .

Die Funktionen der WinAPI verlangen oft nach Pointern zu STRINGS zum Beispiel. In PureBasic selber braucht man auch manchmal den Pointer zu einer Prozedur, zum Beispiel für die Funktion SetWindowCallback() oder PackerCallback(). Darauf gehe ich aber bei dem Punkt "**Pointer zu Prozeduren**" noch genauer ein.

Notation:

Um den Pointer zu einem Element zu erhalten, notiert man einfach ein @ davor (Ausnahme sind Labels, da muss man ein ? davor notieren). Wichtig ist, dass man bei Prozeduren und Linked Lists die Klammern () nicht vergisst, denn sonst sieht PureBasic dieses Element als eine Variable an.

Wichtig: Man kann den Pointer zu einem Element auch in einer Variable speichern, wichtig dabei ist, dass man vor dieser Variable nichts besonderes notiert (auch dann nicht, wenn man diesen Pointer wieder aus der Variable auslesen will.)

Beispiel:

```
Variable1 = 5           ; Beliebige Variable
Variable2 = @Variable1  ; Pointer in Variable2 speichern.

Debug Variable1         ; Vergleich
Debug Variable2
Debug @Variable1
Debug @Variable2
```

Dieses Beispiel soll verdeutlichen, wo viele Leute ein Problem beim Verständnis von Pointern haben. Es ist nämlich immer die Frage, ob man von der Adresse, oder dem Wert spricht.

Man könnte meinen, wenn man einen Pointer mit @ ausliest, und in einer Variablen Speichert, brauche man auch wieder ein @ vor dieser Variable, um den gespeicherten Pointer wieder auszulesen.

In Wirklichkeit hat man aber hier im Beispiel den Pointer zu Variable1 als Wert in Variable2 gespeichert. (Wie man ja bei dem Debug Output sieht, stimmt der Pointer zu Variable1 mit dem Wert von Variable2 überein.) Würde man jetzt, wie hier im Beispiel vor Variable2 auch ein @ schreiben, so erhält man den Pointer zu dieser Variable, also einen komplett anderen Wert. Diese Variable ist ja auch an irgend einer anderen Stelle im Arbeitsspeicher gespeichert.

Hier darf man in seinem Programm nichts durcheinander bringen, weil man dann, wie hier gezeigt komplett andere Werte oder Pointer bekommen kann, was dann zum Beispiel bei einer Poke() oder Peek () Anweisung direkt zu einem Programmabsturz führen kann.

Pointer zu Variablen:

Wie bereits gesagt, ist die wohl häufigste Verwendungsmöglichkeit hierfür die Programmierung mit der WinAPI.

Beispiel: Übergeben eines Strings an eine WinAPI Funktion.

```
Windows.s = Space(256)
GetWindowsDirectory_(@Windows.s, 256)
Debug Windows.s
```

Da Strings eine variable Länge haben, muss man vorher dem String eine Länge geben, die größer ist, als die des Strings, den man erhalten will.

Die Funktion verlangt als erstes Parameter den Pointer zu einem String, und als zweiten Parameter die Länge, die der String hat. (Diese Informationen kann man in jeder API Dokumentation nachlesen. Zum Beispiel hier: <http://msdn.microsoft.com/Library>)

Das Prinzip hierbei ist immer das selbe, wenn nach einem Pointer gefragt wird (Sieht man meist an einem "lp" vor dem Namen des Parameters) muss man in PureBasic ein @ vor die Variable schreiben. Übrigens werden bei der WinAPI Strings immer als Pointer übergeben.

Pointer zu Strukturen:

Hier gilt das gleiche, wie bei Strings. In Verwendung mit der WinAPI kann man Strukturen nur als Pointer übergeben. Unter Verwendung der in Kapitel 3 beschriebenen Pointer kann man diese Art auch verwenden, um Strukturen als Parameter an eine Prozedur zu übergeben.

Wichtig hierbei ist Folgendes: Man gibt nur den Namen der Variable, die die Struktur beinhaltet, mit einem @ davor an. Würde man danach noch ein Element dieser Struktur notieren, so erhält man den Pointer zu diesem Element, und eben nicht zu der Struktur.

Beispiel:

```
Structure TestStruktur
  Element1.l
  Element2.l
EndStructure

Variable.TestStruktur

Debug @Variable
Debug @Variable\Element1
Debug @Variable\Element2
```

Wie man sieht, stimmt der Pointer zum ersten Element der Struktur mit dem zur Variable, die die Struktur enthält überein. Dies hängt einfach damit zusammen, dass eine Struktur ja auch nichts anderes ist, als eine Aneinanderreihung von Datentypen im Arbeitsspeicher. Wichtig ist nur, dass man weiß, dass die Notation hier zu unterschiedlichen Ergebnissen führt, falls es sich eben nicht um das erste Element handelt.

Ein Beispiel zur Verwendung befindet sich im Kapitel 3 unter "**Strukturen als Parameter bei Prozeduren**"

Pointer zu Elementen in Strukturen:

Wie bereits gesagt, notiert man hierbei den Name der ein @, dann die Variable, die eine Struktur enthält mit dem Element, zu dem man den Pointer erhalten möchte.
Also zum Beispiel "@Variable\Element2".

Man kann dieses Element der Struktur eigentlich wie eine Einzelne Variable betrachten, deshalb gibt es eigentlich nicht mehr dazu zu sagen, als schon bei "Pointer zu Variablen" steht.

Pointer zu Arrays:

Um den Pointer zu einem Array zu erhalten, notiert man einfach den Namen des Arrays, mit einem @ davor, und einer leeren Klammer () danach. Man kann auch den Pointer zu einem Element des Arrays bekommen, indem man in der Klammer die Nummer des Elementes angibt. Auch hier gilt, wie bei den Strukturen, dass der Pointer zum ersten Element (Bei PB Arrays ist das die Nummer 0) mit dem Pointer zum Array an sich übereinstimmt.

Nützlich sind Pointer bei Arrays einerseits, da man so Variablen in einem Array speichern kann, und dann trotzdem die einzelnen Elemente einzeln an zum Beispiel eine WinAPI Funktion übergeben kann. Auch fordern manche WinAPI Funktionen den Pointer zu einem Array voller Daten. In diesem Fall gibt man dann kein Element in der Klammer an, und erhält somit den Pointer zum ganzen Array.

Beispiel:

```
Dim Array.l(10)

Debug @Array()
Debug @Array(0)
Debug @Array(5)
```

Pointer zu Linked Lists:

Um den Pointer zu einer Linked List zu erhalten, notiert man den Namen der Linked List mit einem @ davor. Wichtig ist, dass man die Klammern hinter dem Namen der Liste nicht vergisst.
Mit dieser Notation erhält man immer den Pointer zum aktuellen Element der Liste.

Ich will jetzt hier nur darauf eingehen, wie Pointer in Verwendung mit den normalen Linked List Befehlen nützlich sind. Die vielen weiteren Möglichkeiten, die man mit Pointern und Linked Lists hat, würden den Rahmen dieses Tutorials sprengen.

Man kann die Adresse eines Listenelementes in einer Variable abspeichern, dann Änderungen an der Liste vornehmen (z.B. die Liste durchsuchen) und dann anhand des Pointers die Liste wieder zurück auf das vorherige Element setzen.

Beispiel:

```
NewList Liste.l()
AddElement(Liste()): Liste() = 1
AddElement(Liste()): Liste() = 2
AddElement(Liste()): Liste() = 3

FirstElement(Liste())
; (das aktuelle Element ist jetzt 1)

Debug Liste()
```

```

Element.1 = @Liste()    ; Pointer speichern

ResetList(Liste())      ; Liste durchsuchen
While NextElement(Liste())
    Debug Liste()
Wend

; Element zurück auf 1 setzt (mit Pointer)
ChangeCurrentElement(Liste(), Element)

Debug Liste()

```

Pointer zu Prozeduren:

Um den Pointer zu einer Prozedur zu erhalten, notiert man einfach den Namen der Prozedur mit einem @ davor, und einer leeren Klammer () danach.

Den Pointer zu einer Prozedur braucht man zum Beispiel, um einen WindowCallback, einen PackerCallback, oder einen Thread zu erzeugen. Auch in der WinAPI Programmierung kann dies gebrauch werden, wenn für einen bestimmten Vorgang eine Callback Prozedur gebraucht wird. (Ein Callback ist eine Prozedur, deren Pointer man an Windows übergibt, und die dann von Windows aufgerufen werden kann)

Beispiel: (Erstellen eines neuen Threads)

```

Procedure Thread(Parameter.1)
    For i = 0 To 1000000
        Next i
    EndProcedure

CreateThread(@Thread(), 0)

```

Pointer zu Labels:

Da der Programmcode sich ja auch im Arbeitsspeicher befindet, liegt es nahe, dass man auch dazu Pointer auslesen kann. Man kann, indem man den Namen eines Labels (deutsch: Sprungmarke, die Marken, die man mit Gosub/Goto aufrufen kann) mit einem ? davor notiert, die Adresse von genau dieser Stelle des Codes im Arbeitsspeicher ermitteln.

Man wird sich jetzt die Frage stellen, wozu das gut sein soll. Nun, der Pointer zu einfachem PB Code ist auch relativ nutzlos. Interessant wird es allerdings, wenn bei dem Label eine Datei mit IncludeBinary eingebunden wurde. Man kann nämlich dann mit den Speicherbefehlen auf die in dieser Datei gespeicherten Daten zugreifen.

Beispiel 1: (Einbinden von Bitmaps in die Exe Datei)

```

; im Code (Bild wird aus dem Speicher geladen)

CatchImage(1, ?BildLabel)

; hier kommt der Rest vom Code...

End

BildLabel:
IncludeBinary "MeinBild.Bmp"

```


Wichtig hierbei ist, dass `End` vor dem eingebundenen Bild steht. Ansonsten werden die eingebundenen Bilddaten, als Programmcode angesehen und ausgeführt. Das wird garantiert zu einem Absturz führen.

Beispiel 2: (Einbinden beliebiger Dateien in die Exe Datei)

```
; im Code

If OpenFile(0, "Daten.dat")
  WriteData(?DateiLabel1, ?DateiLabel2 - ?DateiLabel1)
  CloseFile(0)
EndIf

; Rest vom Code

End

DateiLabel1:
IncludeBinary "Daten.dat"
DateiLabel2:
```

Hier wird die ganze Datei, die sich ja an der Stelle `?DateiLabel1` befindet mit Hilfe des `WriteData()` Befehls in eine Datei geschrieben. Da die Datei ja genau zwischen den zwei Labels liegt, kann man über die Differenz der beiden Pointer die Größe der dazwischenliegenden Datei ausrechnen.

Zusammenfassung:

- | Pointer zu Elementen erhält man, indem man ein `@` vor die Elementnamen setzt (? Bei Labels)
- | Handelt es sich um ein Array, eine Linked List oder eine Prozedur, so muss man danach eine leere Klammer notieren.
- | Diese Pointer kann man nur lesen, nicht verändern.

[\[Zum Kapitel 1 \]](#)

[\[Zum Kapitel 3 \]](#)

By Timo 'Freak' Harter - Freak@abwesend.de

PureBasic Pointer Tutorial - Kapitel 3: Pointer Setzen

Einleitung:

Die bisher beschriebenen Arten der Pointerhandhabung sind sozusagen alles "nur lesen" Operationen gewesen. Jetzt kommen wir zu der Art von Pointerhandhabung, bei der man den Pointer selber manipulieren kann. Dies ist eigentlich nicht schwerer, als die in Kapitel 2 beschriebenen Methoden, es gilt nur auch hier, dass man das Prinzip verstanden haben muss, um unnötige Fehler und Programmabstürze zu vermeiden.

Grundprinzip:

Den Gedanken dahinter möchte ich einmal Anhand von Strukturen und Pointern erklären. Man stellt sich vor, man hat eine Struktur, die sich irgendwo im Speicher befindet. Man kennt auch die Adresse, an der diese Struktur sich befindet. (So läuft das zum Beispiel, wenn WinAPI eine Struktur als Ergebnis zurückgibt.)

Wie würde man nun die Daten in dieser Struktur lesen?

Man kann ja dies ja nicht mit einer normalen Variable, die eine Struktur enthält lösen, denn diese ist ja dann auf ihren eigenen Speicherbereich begrenzt.

Hier kommen Pointer ins Spiel. Man kann eine sozusagen "virtuelle" Strukturierte Variable erstellen, bei der man dann festlegen kann, auf welchen Bereich des Arbeitsspeichers sie verweist. Also ein selbst definierter Pointer. Man kann dann mit Hilfe dieses Pointers den Inhalt der Struktur im Speicher lesen.

Dies ist zum Beispiel auch nützlich, wenn man einen Speicherbereich mit mehreren hintereinanderliegenden gleichen Strukturen hat. Man kann den Pointer dann sozusagen durch den gesamten Speicherbereich verschieben, und all diese Strukturen nacheinander auslesen, bzw. modifizieren.

Notation:

Notiert man einen Stern (*) vor einer Variable, oder einer Variable mit einer Struktur, so erstellt man damit anstatt einer Variable einen Pointer.

Wichtig: Man erstellt damit nur "virtuell" etwas. Man kann also in dem Pointer selbst nicht direkt etwas speichern, sondern man greift immer auf den Speicherbereich zu, auf den der Pointer gerade zeigt.

Folgendes Beispiel soll das Problem verdeutlichen:

```
Structure TestStruktur
  Wert1.l
  Wert2.l
EndStructure

Variable.TestStruktur
Variable\Wert1 = 5

*Pointer.TestStruktur
*Pointer\Wert1 = 5
```

Führt man diesen Code mit dem Debugger aus erhält man die Meldung: "Pointer is Null".

Zuerst wird eine Variable definiert. Es ist klar, dass man auf diese auch gleich zugreifen kann.

Der erstellte Pointer reagiert aber nicht, wie eine Variable! Er wurde zwar erstellt, zeigt aber noch auf keinen Bereich im Speicher, somit darf man da auch noch keine Werte hinschreiben. Man muss eben erst definieren, wohin der Pointer zeigen soll.

Um dies zu definieren macht man folgendes:

Wenn man den Pointer erstellt hat, notiert man den Namen des Pointers mit einem Stern davor, und weist ihm mit einem Gleichheitszeichen die Speicheradresse zu, auf die er verweisen soll.

Notiert man noch den Namen eines Elementes dahinter (mit einem Backslash), so greift man damit auf den Wert, der an dieser Stelle im Speicher gespeichert ist zu.

Man kann die Adresse, auf die der Pointer zeigt jederzeit ändern. Alle nachfolgenden Zugriffe auf die Elemente des Pointers beziehen sich dann auf die aktuell gültige Adresse.

Folgendes Beispiel zeigt, wie dies in der Praxis funktioniert:

```
Structure TestStruktur
  Wert1.l
  Wert2.l
EndStructure
```

```
Variable.TestStruktur
Variable\Wert1 = 5
```

```
*Pointer.TestStruktur
*Pointer = @Variable
```

```
Debug *Pointer\Wert1
```

```
PointerWert.l = *Pointer
Debug PointerWert
Debug @Variable
```

Ich setze hier den Pointer auf die Adresse der Variable. Somit kann ich über den Pointer auf die Elemente von Variable zugreifen. Was ich mit der Debug-Anweisung auch tue.

Wie am Ende dieses Beispiels gezeigt, kann man die Adresse, auf die ein Pointer verweist auch wieder auslesen. Dazu notiert man einfach wieder nur den Namen des Pointers mit einem Stern davor (keine Elemente). Wie man hier sieht ist die Adresse von Pointer und der Variable gleich. (Das war ja auch beabsichtigt.)

Man kann Pointer wie Variablen auch als Global, Shared oder Protected definieren. Dies hat wie bei Variablen auch die Auswirkung, dass sie nur für eine bestimmte, oder eben für alle Prozeduren gelten. Die Definition funktioniert wie bei Variablen auch:

```
Global *Pointer.Struktur
Shared *Pointer.Struktur
Protected *Pointer.Struktur
```

Es gilt bei der Arbeit mit diesen Pointern zu beachten, dass man sie nur auf Adressen setzen darf, für die das Programm auch Lese- oder Schreibzugriff hat. Ansonsten wird das Programm mit dem Hinweis auf eine Zugriffsverletzung abstürzen.

Pointer zu Strukturen:

Eigentlich bin ich beim Punkt "**Notation**" schon praktisch auf alles eingegangen, was bei Pointern zu Strukturen wichtig ist:

- Notiert man nur den Namen des Pointers mit einem Stern (ohne Elemente), so erhält/setzt man die

Adresse, auf die der Pointer zeigt.

- Notiert man das ganze mit einem Element dahinter, so kann man dieses manipulieren/auslesen.

Ein Beispiel zur Verwendung befindet sich beim Punkt "**Strukturen als Parameter bei Prozeduren**"

Pointer zu einzelnen Variablen:

Hier ergibt sich ein Problem (wobei die Frage ist, ob das als Problem anzusehen ist).

Wie bereits beschrieben erhält man bei einer Struktur den Zugriff auf den Speicherbereich, auf den der Pointer zeigt indem man das gewünschte Element dazu notiert. Wie aber erhält man den Inhalt des Speichers eines Pointers, der auf eine einzelne Variable zeigt?

Dazu gibt es in PureBasic keine direkte Möglichkeit.

Es stellt sich auch die Frage, ob man dies überhaupt braucht. Schließlich kann man die Adresse ja in jeder LONG Variable speichern, und wenn man Zugriff auf den Wert im Speicher will, kann man PeekL()/PokeL(), PeekW()/PokeW(), PeekB()/PokeB(), PeekF()/PokeF() oder PeekS()/PokeS() verwenden.

Will man es aber gerne mit Pointern direkt lösen empfehle ich folgende Lösung:

```
Structure Byte
  Wert.b
EndStructure

*Pointer.Byte

Variable.b = 5

*Pointer = @Variable

Debug *Pointer\Wert
```

Man definiert praktisch eine Struktur, die nur 1 Byte groß ist. Man kann dann wie bei den Strukturen die Adresse manipulieren, und über "***Pointer\Wert**" den Wert im Speicher lesen.

Dies funktioniert natürlich auch mit WORD und LONG.

Strings sind hier ein Sonderfall, auf den ich noch kurz eingehen möchte:

Speichert man einen String in einer Struktur, so speichert man immer nur den Pointer zum wirklichen String in der Struktur. Das kommt daher, dass Strings ja eine variable Länge haben, eine Struktur muss aber eine vordefinierte, feste Größe haben. Schreibt man also ein einer Struktur zum Beispiel "String.s", so wird dies intern durch einen 4Byte Pointer zum String ersetzt. (Normale String-Variablen sind intern übrigens auch Pointer.) Aus diesem Grund kann man mit dieser Struktur-Technik Strings nicht einfach einlesen. Hier würde ich zu PeekS() und PokeS() greifen.

Ihr werdet bestimmt schon ein mal in irgendeinem Codebeispiel, oder in der PB Hilfe die Verwendung solcher Pointer ohne eine Struktur gesehen haben:

```
*Fenster = OpenWindow(0, 100, 100, 200, 200, #PB_Window_SystemMenu, "Fenster")
CreateGadgetList(*Fenster)
```

Dies ist ohne Weiteres möglich. Allerdings manipuliert man hier immer nur eine Adresse, kann aber nie Lesen, was dort wirklich gespeichert ist. Da ein Pointer nichts anderes ist, als eine 4Byte Adresse, hat man hier also praktisch eine LONG Variable mit der Erinnerung, dass es sich hierbei um eine gespeicherte Adresse handelt, und mehr nicht.

Ich habe mir deshalb angewöhnt, den Stern für Pointer nur da zu verwenden, wo er auch Sinn macht, also eigentlich nur im Zusammenhang mit Strukturen. In Fällen wie diesen kann man nämlich genauso gut eine LONG Variable verwenden.

Aus den oben genannten Gründen funktioniert etwas wie *Pointer.s = "Hallo" überhaupt nicht. Man kann auf diese Weise nicht auf Strings im Speicher zugreifen.

Pointer zu Speicherbereichen:

In vielen Beispielen sieht man, dass Leute, wenn sie mit Speicherbereichen arbeiten, Pointer verwenden.

Beispiel:

```
*Speicher = AllocateMemory(0, 1024)
```

Hierzu möchte ich sagen, dass ich das nicht für besonders hilfreich halte. Hier ist nirgendwo eine Struktur im Spiel, und wie ich bereits gesagt habe, kann *Speicher also nichts anderes, als die Adresse speichern.

Für solche Dinge empfehle ich, normale LONG Variablen zu verwenden. Ich würde Pointer wirklich nur da verwenden, wo es notwendig und Sinnvoll ist, verwendet man sie zu oft, hat man es um so schwerer Fehler zu finden.

Da kann natürlich jeder anderer Meinung sein, als ich. Ich habe nur schon öfters gesehen, dass diese Art der Programmierung bei den Leuten zu einem Durcheinander, und damit zu Fehlern geführt hat.

Strukturen als Parameter bei Prozeduren:

Die folgenden zwei Punkte sollen ein praktischen Anwendungsbeispiel für die Verwendung von Pointern sein, dass zusätzlich auch ein anderes Problem der Pb Programmierung beschreibt und löst, nämlich die Übergabe von Strukturen an Prozeduren. Dieses Beispiel befindet sich auch in der Hilfe von PB, allerdings ist es dort so knapp beschrieben, dass man es (ohne Vorkenntnisse) kaum versteht.

Problem: Man kann keine Strukturen als Parameter an eine Prozedur übergeben. Nur die Standard Typen von PureBasic.

Lösung: Ein Pointer ist ja auch praktisch ein Standard Typ, also kann man den Pointer zu einer Struktur an eine Prozedur übergeben.

Hierzu ermittelt man beim Aufrufen der Prozedur die Adresse mit Hilfe des in Kapitel 2 beschriebenen @ Symbols, und nimmt in der Prozedur diese Adresse über einen mit einem Stern definierten Pointer entgegen.

Beispiel:

```
Structure TestStruktur
  Wert1.l
  Wert2.w
  Wert3.b
EndStructure

Procedure TestProzedur (*Pointer.TestStruktur)

  Debug *Pointer\Wert2

EndProcedure
```

```
Variable.TestStruktur  
Variable\Wert2 = 521  
  
TestProzedur(@Variable)
```

Der Pointer in der Prozedur bekommt beim Aufruf die Adresse der übergebenen Struktur zugewiesen. Man kann nun also über den Pointer auf die Struktur zugreifen.

Wichtig: Modifiziert man Werte der Struktur mit dem Pointer, so wird die Ursprungsstruktur des Hauptprogramms verändert, da der Pointer ja auf deren Speicherbereich verweist. Dies ist beim übergeben der Standard Typen ja anders. Die als Parameter erhaltenen Variablen bekommen dort einen anderen Speicherbereich zugewiesen als die, die als Parameter im Hauptprogramm übergeben wurden.

Man kann diesen Effekt auch zu nutze machen, um somit einfach mit Hilfe einer Prozedur den Inhalt einer Struktur zu verändern. Dies ist auch ein guter Weg, um die Rückgabe von Strukturen bei zu realisieren.

Beispiel:

```
Structure TestStruktur  
  Wert1.l  
  Wert2.l  
  Wert3.l  
EndStructure  
  
Procedure TestProzedur(*Pointer.TestStruktur)  
  
  *Pointer\Wert3 = *Pointer\Wert1 + *Pointer\Wert2  
  
EndProcedure  
  
Variable.TestStruktur  
Variable\Wert1 = 5  
Variable\Wert2 = 10  
  
TestProzedur(@Variable)  
  
Debug Variable\Wert3
```

Strukturen als Rückgabewerte von Prozeduren:

Man kann das nach genau dem selben Prinzip machen, wie im vorigen Abschnitt.

Man erstellt in der Prozedur eine Variable mit einer Struktur, und übergibt dann den Pointer (mit @) als Rückgabewert. Im Hauptprogramm erstellt man einen Pointer (mit *), der dann auf die Adresse des Rückgabewertes zeigt.

Beispiel:

```
Structure TestStruktur  
  Wert1.l  
  Wert2.l  
  Wert3.l  
EndStructure  
  
Procedure TestProzedur( )
```

```

Variable.TestStruktur
Variable\Wert2 = 521

ProcedureReturn @Variable
EndProcedure

*Pointer.TestStruktur
*Pointer = TestProzedur()

;Delay(10)

Debug *Pointer\Wert2

```

Auch falls das Beispiel so bei euch funktioniert, spätestens, wenn man den Kommentar vor dem Delay() wegmacht, kommt es nicht mehr zum gewünschten Ergebnis. Warum?

Hierzu muss man wissen, dass am Ende einer Prozedur alle Variablen freigegeben werden. Das heißt, nachdem man mit ProcedureReturn die Prozedur verlassen hat, hat man nur noch den Pointer zu einer nicht mehr existierenden Variable. Der Wert, den man dann lesen kann ist also nutzlos.

Um dies zu verhindern, muss man dafür sorgen, dass die Variable beim Verlassen der Prozedur nicht zerstört wird. Dies kann man erreichen, in dem man sie als Shared oder Global definiert.

Beispiel:

```

Structure TestStruktur
  Wert1.l
  Wert2.l
  Wert3.l
EndStructure

Procedure TestProzedur()

  Shared Variable.TestStruktur

  Variable\Wert2 = 521

  ProcedureReturn @Variable
EndProcedure

*Pointer.TestStruktur
*Pointer = TestProzedur()

Delay(10)

Debug *Pointer\Wert2

```

Diese Methode hat einige entscheidende Nachteile:

- | Man kann den Namen der Variable nicht mehr im Hauptprogramm verwenden, wie diese ja Global ist.
- | Ruft man mehrmals die Prozedur auf, so wird immer die gleiche Variable mit ihrer Struktur manipuliert. Das heißt, die Werte, die vorher darin gespeichert waren, sind dann verloren, bzw. werden verändert.

Die Bessere Methode ist hier in meinen Augen die, dass das Hauptprogramm die Variable erstellt, sie als Parameter an die Funktion übergeben wird, und die Funktion schreibt dann die passenden Werte hinein. Man macht sich damit den oben beschriebenen Effekt zu nutze.

Dies hat klare Vorteile gegenüber der anderen Möglichkeit:

- | Es werden keine Globale Variablen gebraucht.
- | Man kann die Funktion aufrufen sooft man will, weil man ja immer eine andere Variable übergeben kann.

Zusammenfassung:

- | Mit einem Stern vor der Variable definiert man einen veränderbaren Pointer.
- | Diese sind nur "virtuell", man muss ihnen zuerst eine Adresse zuweisen.
- | Man kann sie als Global, Shared oder Protected definieren.
- | Sie sind eigentlich nur im Zusammenhang mit Strukturen wirklich sinnvoll.

[[Zum Kapitel 2](#)]

By Timo 'Freak' Harter - Freak@abwesend.de