

## Das Mysterium „String“

# Das Mysterium “String” in Delphi

```
{ TObject class }  
type  
  TObject = class  
    constructor Create;  
    class function Free;  
    procedure AttachInstance(instance: Pointer): TObject;  
    function ClassUpToDate;  
    class function ClassName: ShortString;  
    class function ClassParent: TClass;  
    class function ClassInfo: TClassInfo;  
    class function InstanceSize: Integer;  
    class function InstanceTable: TClassTable;  
    class function GetInterfaceEntry(const IID: TGUID; out Obj): Pointer;  
    function SafeCallExceptionTable(const IID: TGUID; out Obj): Pointer;  
    procedure AfterConstruction; virtual;  
    procedure BeforeDestruction; virtual;  
    class function Dispatch(var Message; virtual;  
    procedure FreeInstance(var Message); virtual;  
    destructor Destroy; virtual;  
  end;
```

# **Inhaltsverzeichnis**

<b>EINLEITUNG</b>	<b>1</b>
<b>1 STRING, SHORTSTRING UND PCHAR</b>	<b>2</b>
1.1 Der ShortString	2
1.2 Der AnsiString	2
<b>2 PCHARS UND ARRAYS OF CHAR</b>	<b>4</b>
<b>3 STRINGS ALS PARAMETER</b>	<b>5</b>
3.1 var-Parameter und Pointer	5
3.2 “normaler”- und const-Parameter	6
<b>4 REFERENZZÄHLER UND LÄNGENZÄHLER</b>	<b>8</b>
4.1 Der Längenzähler	9
4.2 Referenzzähler	10
4.3 Direkte „String-Manipulation“	11
4.4 Indirekte „String-Manipulation“	13
4.5 Konstante Strings und Objekt-Properties	15
<b>5 STRINGS ↔ PCHARS</b>	<b>17</b>
5.1 Methode 1 – PChar(String)	17
5.2 Methode 2 - @String[1]	18
5.3 Methode 3 – Pointer(String)	19
<b>6 STRINGS UND APIs</b>	<b>20</b>
6.1 Das Programm „APICall“	20
6.2 Ergebnisse und Konsequenzen	22
<b>DANKSAGUNG</b>	<b>23</b>

**Einleitung**

Auch wenn der Titel dieses Tutorials aus einem einfachen, unscheinbaren Wort besteht, das sogar schon einem Anfänger bestens bekannt sein sollte, ist dieses Tutorial alles andere als simpel, da es in vielen Teilen gute Kenntnisse von Pointern (wem diese fehlen, der kann sich auch mal mein Pointer-Tutorial zu Gemüte führen) und Assembler-Grundkenntnisse voraussetzt. In den Grundzügen mag es vielleicht auch für ein bisschen fortgeschrittenere Anfänger interessant und auch wichtig sein, um das Tutorial jedoch in seiner Gesamtheit zu verstehen, muss man sich bereits recht gut mit der Materie auskennen!

Strings sind einer der elementarsten und wichtigsten Typen in der Programmierung, daher stellt Delphi ein äußerst leistungsfähiges und flexibles String-System zur Verfügung. Aber obwohl jeder diese Strings einsetzt, wissen nur die wenigsten, was intern wirklich abläuft.

**Kontakt**

Bei Fragen, Anregungen, Kritiken oder sonstigem einfach bei mir melden!

E-Mail: [motzi@manuel-poeter.de](mailto:motzi@manuel-poeter.de)

Homepage: [www.mmanuel-poeter.de](http://www.mmanuel-poeter.de)

## 1 String, ShortString und Pchar

### 1.1 Der ShortString

Mit der ständigen Weiterentwicklung von Delphi waren die Strings einem gewissen Fortschritt unterworfen. Wer noch von Turbo Pascal kommt, kann sich bestimmt noch an die alten Strings mit einer maximal Länge von 255 Zeichen erinnern. Diese Strings sind folgendermaßen aufgebaut:

16	D	a	s		i	s	t		e	i	n		T	e	x	t
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Das Zeichen an der Stelle 0 enthält die Länge des Strings, erst danach kommt dann der eigentliche Text. Der Nachteil dabei ist, dass der String aufgrund des Längenbytes auf 255 Zeichen begrenzt ist (da das ja der höchstmögliche Wert eines Bytes ist).

Diese String-Art wurde bis Delphi 2 verwendet und existiert auch heute noch.

### 1.2 Der AnsiString

Seit Delphi 2 jedoch wurde eine neue, viel leistungsfähigere String-Art eingeführt. Diese neuen String sind den aus den APIs bekannten PChars sehr ähnlich – beide sind nullterminiert, das heißt der String gilt beim ersten Auftauchen eines Nullzeichens (das Zeichen mit dem ANSI-Wert #0) als abgeschlossen. Es gibt also kein Längenbyte mehr, sondern der String kann (theoretisch) beliebig lang werden, vorausgesetzt, es kommt dazwischen kein Nullzeichen vor. Die neuen AnsiStrings sind nun folgendermaßen aufgebaut:

1	0	0	0	8	0	0	0	T	e	s	t	t	e	x	t	#0
-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9
Referenzzähler				Längenzähler				Text								

Dem Referenz- und Längenzähler habe ich mit Kapitel 4 einen eigenen großen Abschnitt in diesem Tutorial gewidmet, in dem diese beiden Besonderheiten gesondert behandelt werden.

Eine String-Variable wird nun folgendermaßen deklariert<sup>1</sup>:

```
var
  sNewString: String;
  sOldString: ShortString;
```

sNewString stellt also einen neuen dynamischen String dar, während sOldString einem alten statischen String aus Turbo Pascal entspricht. Wieso „dynamisch“ und „statisch“?

Ein statischer String aus Turbo Pascal belegt immer 256 Zeichen. Das entspricht dem Längenbyte plus 255 Zeichen, die von Text belegt werden können. Diese 256 Zeichen sind immer reserviert, egal, ob sie benutzt werden oder nicht! Das Längenbyte gibt nur an, wie viele der 255 Zeichen Text enthalten. Die restlichen Bytes bleiben ungenutzt, aber dennoch reserviert!

<sup>1</sup> Die Compiler-Schalter \$H bzw. \$LONGSTRING bestimmen ob der Typ „String“ einen neuen dynamischen String oder einen alten statischen String darstellt. Standardmäßig sind diese Schalter immer aktiviert!

Es gibt auch noch eine „Unterart“ dieser ShortStrings, deren Länge irgendwo zwischen 0 und 255 liegen muss. Diese Strings sind ebenfalls statisch, werden jedoch durch das Wort „String“ und einer darauf folgenden eckigen Klammer mit der maximalen Länge definiert:

```
var  
  sStaticString: String[40]
```

Der belegte Speicher eines solchen Strings beträgt immer die angegebene Länge plus 1 (das Längenbyte).

Die neuen (dynamischen) Delphi-Strings reservieren den benötigten Speicher zur Laufzeit dynamisch, belegen also immer nur soviel Platz, wie sie auch wirklich benötigen. Diese Strings sind intern eigentlich nur Pointer, die implizit dereferenziert werden – alles komplett transparent für den Entwickler (daher ergibt ein `SizeOf(String)` auch immer nur 4). Damit sind Strings und PChars eigentlich fast identisch, denn ein PChar ist auch nur ein Pointer auf das erste Zeichen einer Zeichenkette. Nicht umsonst ist der Typ PChar folgendermaßen deklariert:

```
type  
  PChar = ^Char;
```

Das einzige, was Strings und PChars letztlich doch noch unterscheidet, sind die internen Stringbehandlungsroutinen und die bei den Strings intern mitgeführten Referenz und Längenzähler. Und genau dieses interne String-Handling, das komplett transparent abläuft, ist es, was das Arbeiten mit diesen Strings so komfortabel und gleichzeitig so unanfällig für Fehler macht. Z.B. wird intern immer das abschließende Nullzeichen mitgeführt, d.h., um das muss man sich überhaupt nicht kümmern, wenn man jedoch bei einem PChar das Nullzeichen vergisst, kann dieser auf einmal überraschenderweise ungeahnte Dimensionen annehmen (bis zum nächsten auftretenden Nullzeichen eben). Auf die Referenz- und Längenzähler komme ich später in Kapitel 4 noch einmal zurück.

## 2 PChars und Arrays of Char

Es gibt noch eine weitere Möglichkeit, eine Zeichenkette zu definieren – ein array of Char. Diese Methode liefert einen statischen Speicherblock, in dem man eine Zeichenkette ablegen kann (ähnlich den ShortStrings), jedoch ohne das führende Längenbyte. Außerdem sind diese array of Char nicht wie die ShortStrings auf 255 Zeichen begrenzt. Solche array of Char werden gerne als statischer Puffer für APIs verwendet, die einen PChar erwarten und Daten in diesem ablegen.

```
var:
  szBuffer: array [0..MAX_PATH] of Char;

GetTempPath(NumberOf(szBuffer), szBuffer);
```

Die meisten werden sich jetzt wundern, warum ich die Variable szBuffer direkt in die API-Funktion übergeben kann, obwohl doch eigentlich ein PChar erwartet wird. Das liegt daran, dass PChar und array of Char zuweisungskompatibel sind, jedoch **nur**, wenn das array nullindiziert ist!

```
var:
  szCorrect   : array [0..10] of Char;
  szIncorrect : array [1..10] of Char;
  p: PChar;

p := szCorrect;    // -> funktioniert
p := szIncorrect;  // -> funktioniert nicht (Incompatible types: 'Array'
                  // and 'PChar')
```

Genau genommen kümmert sich Delphi um die korrekte Zuweisung. D.h. Delphi erkennt, was wir vorhaben, ermittelt die Adresse des ersten Zeichens des Arrays und weist diese dem PChar zu. Insofern könnte man auch einen nicht nullindizierten Array nehmen und einem PChar zuweisen, wenn wir uns selbst um diese Zwischenschritte kümmern:

```
var:
  szSelfmade: array [1..10] of Char;
  p: PChar;

p := @szSelfmade[1];    // -> funktioniert
```

Leider sind Strings weder zu PChars noch zu array of Char zuweisungskompatibel, um also einen String in einen PChar/array of Char zu bekommen, benötigt man die Funktion StrPCopy:

```
function StrPCopy(Dest: PChar; const Source: string): PChar;
```

Diese Funktion kopiert den mit Source angegebenen String in den mit Dest angegebenen Buffer, wobei Dest mindestens Platz für die Länge von Source plus 1 (das Nullzeichen) bieten muss. Eine andere Möglichkeit ist außerdem noch das direkte Typcasting, aber auf das komme ich später noch in Kapitel 5 zurück.

Umgekehrt ist ein PChar sehr wohl zu einem String zuweisungskompatibel. Es gibt zwar noch eine Funktion StrPas, die einen PChar in einen String konvertiert, allerdings fristet diese nur mehr ein Dummy-Dasein, da die Implementierung folgendermaßen aussieht:

```
function StrPas(const Str: PChar): string;
begin
  Result := Str;
end;
```

Eine einfache Zuweisung des PChars an den String reicht also bereits aus. Der Compiler erkennt die Situation und leitet alle nötigen Zwischenschritte automatisch ein – alles komplett transparent für den Entwickler!

### 3 Strings als Parameter

Es gibt 4 verschiedene Möglichkeiten, einen String an eine Funktion/Prozedur zu übergeben, 2 dieser Methoden sind im Prinzip jedoch komplett identisch (sie werden vom Compiler zumindest identisch umgesetzt). Was das heißt, möchte ich hier erklären. Folgende Möglichkeiten gibt es, um einen String als Parameter an eine Funktion bzw. Prozedur zu übergeben:

1. "normaler" Parameter
2. const-Parameter
3. var-Parameter
4. als Pointer

Die Methoden 3 und 4 unterscheiden sich zwar dadurch, wie sie implementiert werden, allerdings wird ein var-Parameter intern auch nur als Pointer übergeben, wodurch der Compiler bei beiden Methoden identischen Code erzeugt. Dass diese Aussage stimmt, zeigt das nächste Kapitel.

#### 3.1 var-Parameter und Pointer

Deklaration und Aufruf einer Prozedur mit einem var-Parameter:

```
procedure foo_var(var s: String);  
  
var  
  s: String;  
  
  s := 'Test';  
  foo_var(s);
```

Deklaration und Aufruf einer Prozedur mit einem Pointer-Parameter:

```
procedure foo_pointer(p: PAnsiString);  
  
var  
  s: String;  
  
  s := 'Test';  
  foo_pointer(@s);
```

Von der Deklaration und vom Aufruf her schauen beide Methoden anders aus, aber dennoch erzeugen der Compiler aus beiden Aufrufen identischen Assembler-Code:

Assembler-Code des Aufrufs der Funktion mit var-Parameter:

```
lea    eax, [ebp-$04]  
call   foo_var
```

Assembler-Code des Aufrufs der Funktion mit Pointer-Parameter:

```
lea    eax, [ebp-$04]  
call   foo_pointer
```

In beiden Fällen wird die Adresse der String-Variable ermittelt und über das eax-Register an die Funktion übergeben. Damit sollte bewiesen sein, dass diese beiden Methoden im Prinzip identisch sind. Außerdem sollte klar sein, dass der erzeugte Assembler-Code mit nur zwei Zeilen sehr gering ausfällt und diese Methoden daher sehr schnell sind.

### 3.2 “normaler”- und const-Parameter

Deklaration und Aufruf einer Prozedur mit einem “normalen” Parameter:

```
procedure foo_normal(s: String);  
  
var  
  s: String;  
  
  s := 'Test';  
  foo_normal(s);
```

Deklaration und Aufruf einer Prozedur mit einem const-Parameter:

```
procedure foo_const(const s: String);  
  
var  
  s: String;  
  
  s := 'Test';  
  foo_const(s);
```

Auch diese beiden Methoden scheinen auf den ersten Blick identisch, wenn man sich den erzeugten Assembler-Code für den Aufruf ansieht:

Assembler-Code des Aufrufs der Funktion mit „normalem“ Parameter:

```
mov     eax, [ebp-$04]  
call    foo_normal
```

Assembler-Code des Aufrufs der Funktion mit const-Parameter:

```
mov     eax, [ebp-$04]  
call    foo_const
```

Der eigentliche Unterschied wird erst in der aufgerufenen Funktion sichtbar. Während bei einem const-Parameter dort nichts weiter wichtiges geschieht, findet man bei der Funktion mit normalem Parameter folgende Code-Blöcke am Anfang:

```
push    ebp  
mov     ebp, esp  
push    ecx  
mov     [ebp-$04], eax  
mov     eax, [ebp-$04]  
call    @LstrAddRef  
xor     eax, eax  
push    ebp  
push    $0046c81e  
push    dword ptr fs:[eax]  
mov     fs:[eax], esp
```

bzw. am Ende der Funktion:

```
xor     eax, eax  
pop     edx  
pop     ecx  
pop     ecx  
mov     fs:[eax], edx  
push    $0046c825  
lea     eax, [ebp-$04]  
call    @LStrClr  
ret  
jmp     @HandleFinally  
jmp     -$10
```

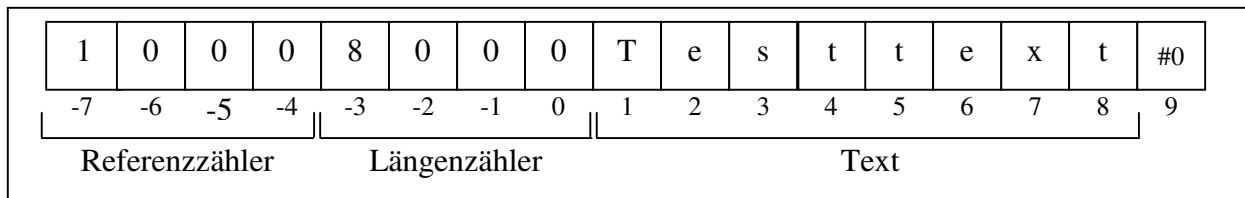
Durch diesen Code wird eine lokale Kopie des Strings angelegt, welche dann innerhalb der Funktion benutzt wird, wodurch der übergebene String davor geschützt wird, verändert zu werden. Außerdem wird noch extra ein SEH-Frame (SEH = Structured Exception Handling) angelegt, damit die lokale Kopie des Strings auf jeden Fall am Ende wieder korrekt freigegeben wird.

Dass das ganze dann natürlich wieder eine Menge Arbeit bedeutet, dürfte jedem einleuchten, und dass sich diese Mehrarbeit eher schlecht auf das Laufzeitverhalten auswirkt, sollte auch klar sein. Daher sollten Strings, wenn möglich, immer nur als const-Parameter bzw. wenn sie von der aufgerufenen Funktion verändert werden sollen, als var-Parameter übergeben werden.

## 4 Referenzzähler und Längenzähler

Wie bereits erwähnt, führen Delphi-Strings (die ShortStrings haben wir schon hinter uns gelassen, wenn ich hier Delphi-Strings sage, meine ich nur mehr AnsiStrings) intern einen Referenzzähler (32Bit) und einen Längenzähler (ebenfalls 32Bit) mit. Diese beiden Zähler werden nur intern verwendet und tauchen nirgends auf, aber dennoch sollte man ungefähr wissen, wie Delphi diese intern verwendet und was für Auswirkungen/Nebeneffekte dabei auftreten können.

AnsiStrings sind folgendermaßen aufgebaut:



Bekanntlich kann man ja per Offset auf die einzelnen Zeichen eines Strings zugreifen:

```
var
  s: String;

ShowMessage('Das erste Zeichen des Strings ist ein: ' + s[1]);
```

Das erste Zeichen eines Strings hat also immer das Offset 1 und das letzte Zeichen das Offset „Length(String)“. Der Referenzzähler und der Längenzähler liegen nun sozusagen im negativen Offset-Bereich. Nämlich der Referenzzähler im Bereich von -7 bis -4 und der Längenzähler im Bereich von -3 bis 0. In hardgecodetem Code erkennt der Compiler die Versuche auf diese negativen Offsets zuzugreifen und gibt einen entsprechenden Fehler aus:

```
var
  s: String;

ShowMessage(s[0]); // Element 0 inaccessible - use 'Length' or 'SetLength'
ShowMessage(s[-1]); // Constant expression violates subrange bounds
```

Mit ein paar kleinen Tricks lassen sich aber auch diese negativen Offsets auslesen. Ich habe mir dazu einen kleinen Record gebastelt:

```
type
  PStringInfo = ^TStringInfo;
  TStringInfo = packed record
    iRefCount      : Integer;
    iLenCount      : Integer;
    pStringAddr    : Pointer;
  end;
```

Mit Hilfe dieses Records kann man nun ganz einfach den Referenzzähler und den Längenzähler auslesen:

```
var
  s: String;
  p: PStringInfo;

s := 'Test';
p := Pointer(s); // der String ist intern nur ein Pointer
                  // -> dem Pointer auf den StringInfo-Record
                  // den String-Pointer zuweisen

Dec(Integer(p), 8); // den Pointer um 8 dekrementieren (4Bytes Referenz-
                  // und 4Bytes Längenzähler)
```

```
ShowMessage('References: ' + IntToStr(p^.iRefCount) + #13#10 +  
            'Length: '      + IntToStr(p^.iLenCount) + #13#10 +  
            'String: '      + String(@p^.pStringAddr));
```

Wichtig: das Feld `pStringAddr` liegt an der Stelle, an der der String beginnt! D.h., es kann nicht direkt als `PChar` oder `String` gesehen werden, über das man direkt an den String-Inhalt kommt, sonst würden die ersten 4 Zeichen des Strings als Adresse interpretiert und dereferenziert werden, was aller Wahrscheinlichkeit nach in einer `EAccessViolation-Exception` enden würde. Stattdessen muss man, wie gezeigt, die Adresse des Feldes nehmen, welche nun intern dereferenziert wird.

Wir wissen nun zwar, dass es so etwas wie einen Referenz- und Längenzähler gibt, aber bis jetzt noch nicht, wofür diese da sind.

#### 4.1 Der Längenzähler

Der Längenzähler enthält (wie der Name bereits vermuten lässt) die Länge des Strings. Die meisten werden sich jetzt denken, warum eigentlich die Länge des Strings gespeichert wird, wenn er doch nullterminiert ist. Der Grund ist einfach der, dass die Länge des Strings nicht immer der Anzahl der Zeichen bis zum ersten Nullzeichen entspricht. z.B.:

```
var  
  s: String;  
  
s := 'Test';  
s[2] := #0;
```

In der zweiten Zeile wird einfach das zweite Zeichen des Strings auf ein Nullzeichen gesetzt. Der String ist also an dieser Stelle eigentlich abgeschlossen! Eine Ausgabe von `s` würde also nur „T“ ergeben. Dennoch ergibt „`Length(s)`“ 4! Wieso?

Auch bei den neuen dynamischen Strings muss es nicht immer der Fall sein, dass nur genau so viel Speicher reserviert ist, wie vom Text belegt wird. Das hat unter anderem Optimierungsgründe. Nicht umsonst gibt es die Funktion `SetLength`. Diese Funktion setzt die Länge eines Strings, aber nicht dessen Inhalt! Der String hat also die festgelegte Länge, aber undefinierten Inhalt! Was hat das ganze nun mit Optimierung zu tun? Dazu braucht man nur diese beiden Codes vergleichen:

```
var  
  s: String;  
  i: Integer;  
  
s := '';  
for i := 1 to 100 do  
  s := s + 'x';
```

```
var  
  s: String;  
  i: Integer;  
  
SetLength(s, 100)  
for i := 1 to 100 do  
  s[i] := 'x';
```

Kein gravierender Unterschied, oder? Doch, allerdings wird dieser erst bei einer Laufzeitmessung sichtbar. Im ersten Fall muss der Speicher für `s` immer wieder realloziert werden, da der Speicherverbrauch immer wieder langsam um 1 angehoben wird, das bedeutet es wird immer wieder neuer Speicher reserviert und dann der ganze Block aus dem alten

Speicher in den neuen geschoben – eine „Vergewaltigung“ des Speichermanagers! Im zweiten Fall hingegen reserviert „SetLength()“ gleich unsere 100 Bytes, und wir schreiben unseren String-Inhalt einfach Zeichen für Zeichen in unseren bereits reservierten Speicherblock.

Wir wissen nun also – „Length()“ gibt die Länge des Strings zurück, und zwar nicht die Länge des Textes, sondern die Anzahl der tatsächlich reservierten Bytes in denen wir Text ablegen können.<sup>2</sup>

## 4.2 Referenzzähler

Und was ist nun der Referenzzähler? Der Referenzzähler zählt die Anzahl der Referenzen auf den String. Seit Delphi 2 wurde die Stringverwaltung dahingehend optimiert, dass 2 identische Strings nicht 2 mal im Arbeitsspeicher liegen, sondern beide auf denselben Speicher verweisen. Der Referenzzähler enthält nun die Anzahl der Referenzen, also die Anzahl der Strings mit demselben Inhalt. Wird nun einer der Strings verändert, so bekommt er einen neuen (seinen eigenen) Speicherplatz zugewiesen, und der Referenzzähler des Strings auf den er vorhin verwiesen hat, wird um 1 dekrementiert. Ich habe dazu ein kleines Test-Programm namens „Reference1“ geschrieben, das diesen Sachverhalt etwas verdeutlicht:

```
program Reference1;

{$APPTYPE CONSOLE}

uses
  SysUtils;

type
  PStringInfo = ^TStringInfo;
  TStringInfo = packed record
    iRefCount   : Integer;
    iLenCount   : Integer;
    pStringAddr : Pointer;
  end;

procedure ShowStringInfo(const s: String);
var
  p: PStringInfo;
begin
  p := Pointer(s);
  Dec(Integer(p), 8);

  WriteLn(Format('  String:      %s', [s]));
  WriteLn(Format('  p.RefCount:  %d', [p^.iRefCount]));
  WriteLn(Format('  p.LenCount:  %d', [p^.iLenCount]));
end;

var
  s, s2: String;

begin
  WriteLn('Geben sie einen Text ein:');
  ReadLn(s);
  WriteLn;

  WriteLn('Der eingegebene Text wird einer zweiten Variable zugewiesen');
  WriteLn('  s2 := s;');
```

<sup>2</sup> Um die Länge des tatsächlichen Textes (bis zum ersten Auftreten eines Nullzeichens) zu ermitteln kann man die Funktion „strlen()“ benutzen.

```

s2 := s;
WriteLn;

WriteLn('Inhalt von s:');
ShowStringInfo(s);
WriteLn(Format('  Pointer(s):    0x%p', [Pointer(s)]));
WriteLn(Format('  @s:          0x%p', [@s]));
WriteLn;

WriteLn('Inhalt von s2:');
ShowStringInfo(s2);
WriteLn(Format('  Pointer(s2):   0x%p', [Pointer(s2)]));
WriteLn(Format('  @s2:         0x%p', [@s2]));

ReadLn;
end.

```

Ein Testlauf dieses Programms zeigt folgendes Ergebnis:

```

Geben sie einen Text ein:
Test

Der eingegebene Text wird einer zweiten Variable zugewiesen
s2 := s;

Inhalt von s:
String:      Test
p.RefCount:  2
p.LenCount:  4
Pointer(s):   0x00B90838
@s:          0x0040A848

Inhalt von s2:
String:      Test
p.RefCount:  2
p.LenCount:  4
Pointer(s2):  0x00B90838
@s2:         0x0040A84C

```

Wie man erkennen kann, sind alle Felder identisch, bis auf die Adressen der beiden Variablen (@s <> @s2), was ja auch klar ist, da es 2 verschiedene Variablen sind. Aber beide enthalten denselben Zeiger auf dieselbe Adresse und damit auf denselben String, und daher enthält der Referenzzähler (p.RefCount) eben auch den Wert 2.

### 4.3 Direkte „String-Manipulation“

Das ist ja ganz nett, allerdings gibt es dadurch auch einige kleine Fallen. Delphi erkennt direkte verändernde String-Zugriffe und kann entsprechend darauf reagieren. Hierzu habe ich das kleine Test-Programm „Reference2“ geschrieben:

```

program Reference2;

{$APPTYPE CONSOLE}

uses
  SysUtils;

type
  PStringInfo = ^TStringInfo;
  TStringInfo = packed record
    iRefCount    : Integer;
    iLenCount    : Integer;

```

```
pStringAddr : Pointer;
end;

procedure ShowStringInfo(const s: String);
var
  p: PStringInfo;
begin
  p := Pointer(s);
  Dec(Integer(p), 8);

  WriteLn(Format('  String:      %s', [s]));
  WriteLn(Format('  p.RefCount:   %d', [p^.iRefCount]));
  WriteLn(Format('  p.LenCount:   %d', [p^.iLenCount]));
end;

var
  s, s2: String;

begin
  WriteLn('Geben sie einen Text ein (mind. 2 Zeichen):');
  ReadLn(s);
  WriteLn;

  WriteLn('Der eingegebene Text wird einer zweiten Variable zugewiesen');
  WriteLn('  s2 := s;');
  s2 := s;
  WriteLn;

  WriteLn('Inhalt von s:');
  ShowStringInfo(s);
  WriteLn(Format('  Pointer(s):    0x%p', [Pointer(s)]));
  WriteLn(Format('  @s:          0x%p', [@s]));
  WriteLn;

  WriteLn('Inhalt von s2:');
  ShowStringInfo(s2);
  WriteLn(Format('  Pointer(s2):   0x%p', [Pointer(s2)]));
  WriteLn(Format('  @s2:         0x%p', [@s2]));
  WriteLn;

  WriteLn('Variable s wird nun direkt an Position 2 verändert');
  WriteLn('  s[2] := 'X''');
  s[2] := 'X';
  WriteLn;

  WriteLn('Inhalt von s:');
  ShowStringInfo(s);
  WriteLn(Format('  Pointer(s):    0x%p', [Pointer(s)]));
  WriteLn(Format('  @s:          0x%p', [@s]));
  WriteLn;

  WriteLn('Inhalt von s2:');
  ShowStringInfo(s2);
  WriteLn(Format('  Pointer(s2):   0x%p', [Pointer(s2)]));
  WriteLn(Format('  @s2:         0x%p', [@s2]));
  ReadLn;
end.
```

Ein Durchlauf mit der Eingabe von „Test“ hat folgendes Ergebnis:

```
Geben sie einen Text ein (mind. 2 Zeichen):
Test

Der eingegebene Text wird einer zweiten Variable zugewiesen
  s2 := s;

Inhalt von s:
  String:      Test
  p.RefCount:  2
  p.LenCount:  4
  Pointer(s):  0x00B90838
  @s:          0x0040A848

Inhalt von s2:
  String:      Test
  p.RefCount:  2
  p.LenCount:  4
  Pointer(s2): 0x00B90838
  @s2:         0x0040A84C

Variable s wird nun direkt an Position 2 verändert
  s[2] := 'X'

Inhalt von s:
  String:      TXst
  p.RefCount:  1
  p.LenCount:  4
  Pointer(s):  0x00B908EC
  @s:          0x0040A848

Inhalt von s2:
  String:      Test
  p.RefCount:  1
  p.LenCount:  4
  Pointer(s2): 0x00B90838
  @s2:         0x0040A84C
```

Die Variable `s` wurde direkt verändert. Delphi hat dies erkannt und dem String einen eigenen, neuen Speicherblock mit dem neuen veränderten String zugewiesen (die beiden Strings verweisen nicht mehr auf dieselbe Adresse – `Pointer(s) <> Pointer(s2)`), den Referenzzähler des neuen Strings auf 1 gesetzt und den Referenzzähler des Strings, auf den `s` vorher gezeigt hat, um 1 dekrementiert. Daher sind die Referenzzähler von `s` und `s2` am Ende beide 1.

#### 4.4 Indirekte „String-Manipulation“

Wie schaut das ganze jedoch bei einer indirekten Veränderung (z.B. über Pointer) aus? In diesem Fall erkennt Delphi die Gefahr nicht, wodurch nicht nur ein String, sondern alle Strings, die den veränderten Speicherblock referenzieren, verändert werden. Auch hier gibt es wieder ein kleines Test-Programm namens „Reference3“ dazu:

```
program Reference3;

{$APPTYPE CONSOLE}

uses
  SysUtils;

type
```

```

PStringInfo = ^TStringInfo;
TStringInfo = packed record
  iRefCount    : Integer;
  iLenCount    : Integer;
  pStringAddr  : Pointer;
end;

procedure ShowStringInfo(const s: String);
var
  p: PStringInfo;
begin
  p := Pointer(s);
  Dec(Integer(p), 8);

  WriteLn(Format('  String:          %s', [s]));
  WriteLn(Format('  p.RefCount:      %d', [p^.iRefCount]));
  WriteLn(Format('  p.LenCount:       %d', [p^.iLenCount]));
end;

var
  s, s2: String;

begin
  WriteLn('Geben sie einen Text ein (mind. 2 Zeichen):');
  ReadLn(s);
  WriteLn;

  WriteLn('Der eingegebene Text wird einer zweiten Variable zugewiesen');
  WriteLn('  s2 := s;');
  s2 := s;
  WriteLn;

  WriteLn('Inhalt von s:');
  ShowStringInfo(s);
  WriteLn(Format('  Pointer(s):      0x%p', [Pointer(s)]));
  WriteLn(Format('  @s:          0x%p', [@s]));
  WriteLn;

  WriteLn('Inhalt von s2:');
  ShowStringInfo(s2);
  WriteLn(Format('  Pointer(s2):    0x%p', [Pointer(s2)]));
  WriteLn(Format('  @s2:          0x%p', [@s2]));
  WriteLn;

  WriteLn('Variable s wird nun INdirekt an Position 2 verändert');
  WriteLn('  PChar(Cardinal(Pointer(s)) + 1)^ := 'X'');
  PChar(Cardinal(Pointer(s)) + 1)^ := 'X';
  WriteLn;

  WriteLn('Inhalt von s:');
  ShowStringInfo(s);
  WriteLn(Format('  Pointer(s):      0x%p', [Pointer(s)]));
  WriteLn(Format('  @s:          0x%p', [@s]));
  WriteLn;

  WriteLn('Inhalt von s2:');
  ShowStringInfo(s2);
  WriteLn(Format('  Pointer(s2):    0x%p', [Pointer(s2)]));
  WriteLn(Format('  @s2:          0x%p', [@s2]));
  ReadLn;
end.

```

Ein Durchlauf mit der Eingabe von „Test“ zeigt folgendes Ergebnis:

```
Geben sie einen Text ein (mind. 2 Zeichen):
Test

Der eingegebene Text wird einer zweiten Variable zugewiesen
  s2 := s;

Inhalt von s:
  String:      Test
  p.RefCount:  2
  p.LenCount:  4
  Pointer(s):   0x00B90838
  @s:          0x0040A848

Inhalt von s2:
  String:      Test
  p.RefCount:  2
  p.LenCount:  4
  Pointer(s2): 0x00B90838
  @s2:         0x0040A84C

Variable s wird nun INdirekt an Position 2 verändert
  PChar(Cardinal(Pointer(s)) + 1)^ := 'X'

Inhalt von s:
  String:      TXst
  p.RefCount:  2
  p.LenCount:  4
  Pointer(s):   0x00B90838
  @s:          0x0040A848

Inhalt von s2:
  String:      TXst
  p.RefCount:  2
  p.LenCount:  4
  Pointer(s2): 0x00B90838
  @s2:         0x0040A84C
```

Man sieht, dass alle Werte von s als auch von s2 unverändert bleiben (Referenzzähler und Adresse), aber dass sowohl s als auch s2 den neuen veränderten Text „TXst“ enthalten.

Auch wenn man wohl eher selten derartig absurde String-Veränderungen über Pointer realisiert, spätestens bei API-Funktionen, die ein Typcasting nach PChar erfordern, sollte man sich diese Tatsache wieder bewusst machen (wie man im nächsten Kapitel sieht)!

#### 4.5 Konstante Strings und Objekt-Properties

String-Konstanten (sowohl solche, die als „const“ deklariert sind, als auch solche, die direkt im Code per Hochkommas stehen) haben immer einen Referenzzähler von „-1“! Auf solche Strings wird keine Referenz gesetzt, sondern sie werden immer in einen neuen String mit einem Referenzzähler von 1 kopiert.

Ähnlich verhält es sich bei String-Objekteigenschaften, die Lesezugriffe nicht direkt an die private String-Variable weiterleiten, sondern eine Get-Methode haben. Bei solchen Get-Methoden wird bei jedem Aufruf ein neuer String für die Rückgabe erstellt, und ein solcher hat dann selbstverständlich auch immer einen Referenzzähler von 1!

Bei einer simplen Zuweisung von einem String zu einem anderen fügt der Compiler intern einen Aufruf der Funktion „\_LStrAsg()“ aus der Unit System.pas ein. Diese überprüft den Referenzzähler, und falls dieser den Wert -1 hat, wird über die Funktion

---

„\_NewAnsiString()“ (ebenfalls aus der Unit System.pas) ein neuer String erstellt und die Zeichenkette kopiert. Andernfalls wird einfach der Referenzzähler erhöht. Eine Zuweisung eines Strings zu einem anderen besteht also in den meisten Fällen aus dem kopieren von 4 Bytes anstatt der gesamten Zeichenkette!

## 5 Strings ↔ PChars

Vorhin hab ich ja bereits gesagt, dass PChars zuweisungskompatibel zu Strings sind, aber wie verhält sich das jetzt mit Strings zu PChars? Es gibt 3 Möglichkeiten, einen String in einen PChar zu casten, wobei der Compiler bei jedem dieser Casts anderen Code erzeugt.

Methode 1:

```
PChar(String)
```

Methode 2:

```
@String[1]
```

Methode 3:

```
Pointer(String)
```

Gehen wir die Methoden der Reihe nach durch und betrachten wir mal den vom Compiler generierten Code.

### 5.1 Methode 1 – PChar(String)

Dazu habe ich das simple Programm „PChar1“ geschrieben, das folgendermaßen aussieht:

```
program PChar1;

{$APPTYPE CONSOLE}

uses
  Windows,
  SysUtils;

var
  s: String;
begin
  WriteLn('Geben sie einen Text ein:');
  Readln(s);
  MessageBox(0, PChar(s), 'String-Konstante', MB_OK);
end.
```

Der Compiler generiert dabei für den MessageBox-Aufruf folgenden Assembler-Code:

```
push    $00
push    $004087D8
mov     eax, [s]
call    @LStrToPChar
push    eax
push    $00
call    MessageBox
```

Wir haben in diesem Aufruf neben dem direkten Cast auch gleich den Sonderfall einer direkten Übergabe einer String-Konstante. Solche Konstanten werden vom Compiler, je nachdem wie sie eingesetzt werden, unterschiedlich gehandhabt. In diesem Fall übergibt der Compiler lediglich die Adresse, an der die Zeichenkette liegt, nämlich \$00408300. (Wer sich mit den Aufruf-Konventionen auskennt, weiß, dass bei der stdcall-Konvention, die von den APIs verwendet wird, die Parameter von rechts nach links über den Stack übergeben werden.) Für den Typcast nach PChar schiebt der Compiler einen Aufruf von „LStrToPChar()“ (aus der Unit System.pas) ein.

```
function _LStrToPChar(const s: AnsiString): PChar;
```

Was macht diese Funktion nun? Im Prinzip nichts anderes, als testen, ob s einen Wert enthält oder nicht. Ist dies der Fall, springt die Funktion sofort wieder zurück, und es wird der in eax befindliche Wert auf den Stack gepusht – also die Adresse, auf die unser String zeigt – die Adresse des ersten Zeichens unserer Zeichenkette. Enthält der String jedoch keinen Wert (ein Nullstring), so ist die Adresse, auf die unser String verweist, *nil*. Anstatt nun jedoch nil zurückzugeben, gibt die Funktion die Adresse eines Bytes zurück, das den Wert 0 hat, also im Prinzip einen String darstellt, der nur aus dem abschließenden Nullzeichen besteht.

## 5.2 Methode 2 - @String[1]

Wir wissen ja, dass ein String (genauso wie ein PChar) nur ein Zeiger auf eine Zeichenkette ist. Diese Methode basiert nun ganz einfach darauf, dass man die Adresse des ersten Zeichens des Strings hernimmt und als „PChar-Adresse“ weitergibt.

Auch hier gibt es wieder ein kleines Programm „PChar2“, welches im Prinzip genauso aussieht wie „PChar1“, nur eben mit einem anderen Typecast.

```
program PChar2;

{$APPTYPE CONSOLE}

uses
  Windows,
  SysUtils;

var
  s: String;
begin
  WriteLn('Geben sie einen Text ein:');
  ReadLn(s);
  MessageBox(0, @s[1], 'String-Konstante', MB_OK);
end.
```

Der hier vom Compiler generierte Code schaut nun folgendermaßen aus:

```
push    $00
push    $00408830
mov     eax, $0040A848
call    @UniqueStringA
push    eax
push    $00
call    MessageBox
```

In diesem Fall wird also die Funktion „UniqueString()“ aus der Unit System.pas aufgerufen. Die Hilfe sagt zu dieser Funktion folgendes:

```
procedure UniqueString(var str: string); overload;
procedure UniqueString(var str: WideString); overload;
“Following a call to SetString, str is guaranteed to reference a unique string—
that is, a string with a reference count of one. For normal string handling, there is
no need to call UniqueString. UniqueString is used only in cases where an
application casts a string to a PChar or PWideChar and then modifies the
contents of the string.”
```

UniqueString erzeugt also einen neuen String mit demselben Inhalt an einer neuen Speicherstelle, und setzt den Referenzzähler auf 1. Der Referenzzähler des Strings, auf den der String vorher verwiesen hat, wird dementsprechend um 1 dekrementiert (schließlich geht ja eine Referenz „verloren“).

Das ist natürlich ein Mehraufwand zum simplen LStrToPChar-Aufruf von Methode 1 und kostet dementsprechend auch mehr Rechenzeit. Allerdings ist dieser Mehraufwand in gewissen Situationen durchaus gerechtfertigt bzw. sogar äußerst wichtig (das wieso sehen wir später)!

### 5.3 Methode 3 – Pointer(String)

Die „simpelste“ der 3 Methoden besteht einfach aus einem direkten Typecast auf einen Pointer. Und da ein PChar schließlich auch nur ein Zeiger ist, wird diese Vorgehensweise vom Compiler auch gar nicht beanstandet.

Und auch hier das ganze wieder in Form eines Programms „PChar3“, mit demselben Aufbau wie schon bei den 2 vorhergehenden:

```
program PChar3;

{$APPTYPE CONSOLE}

uses
  Windows,
  SysUtils;

var
  s: String;
begin
  WriteLn('Geben sie einen Text ein:');
  Readln(s);
  MessageBox(0, Pointer(s), 'String-Konstante', MB_OK);
end.
```

Ebenfalls wieder der vom Compiler erzeugte Code für den MessageBox-Aufruf:

```
push    $00
push    $004087E0
mov     eax, [s]
push    eax
push    $00
call    MessageBox
```

Einem geschulten Auge fällt sofort auf, was hier passiert, aber selbst einem nicht so geschulten Auge müsste auffallen, dass dieser Code hier kürzer ist als die der beiden vorherigen Methoden, und dass zusätzliche Aufrufe des internen String-Handlings gänzlich fehlen. Diese Art von Typecasting gibt also wirklich nur die Adresse der Zeichenkette weiter, die im String gespeichert ist, und führt keine internen Überprüfungen oder ähnliches durch. Diese Methode ist folglich auch die schnellste (wenn auch nicht wesentlich schneller als die Methode 1).

## 6 Strings und APIs

Solange Strings nur als Eingabe-Parameter an API-Funktionen übergeben werden, kann man dies auch bedenkenlos tun. Man muss sie eben nur entsprechend mit einer der 3 Methoden aus Kapitel 5 casten, damit der Compiler diese auch akzeptiert. Gefährlich wird es allerdings, wenn Delphi-Strings an API-Funktionen übergeben werden, die diese als Buffer verwenden und Daten in diesem ablegen! Denn dann tritt genau der Fall ein, der in Kapitel 4.4 beschrieben wird. Der Compiler hat natürlich keine Ahnung, was die Funktion mit dem String macht, und leitet daher keine entsprechenden Maßnahmen ein, um andere Referenzen auf diesen String vor Veränderungen zu „schützen“. Ich habe hierfür ein kleines Beispiel-Programm namens „APICall“ geschrieben, das die API-Funktion „GetWindowsDirectory“ verwendet, um diesen Sachverhalt darzustellen. Der Code dieses Programms fällt etwas länger aus als die Codes der bisherigen Programme.

### 6.1 Das Programm „APICall“

```

program APICall;

{$APPTYPE CONSOLE}

uses
    Windows,
    SysUtils;

type
    PStringInfo = ^TStringInfo;
    TStringInfo = packed record
        iRefCount      : Integer;
        iLenCount      : Integer;
        pStringAddr    : Pointer;
    end;

procedure ShowStringInfo(const s: String);
var
    p: PStringInfo;
begin
    p := Pointer(s);
    Dec(Integer(p), 8);

    WriteLn(Format('  String:          %s', [s]));
    WriteLn(Format('  p.RefCount:      %d', [p^.iRefCount]));
    WriteLn(Format('  p.LenCount:      %d', [p^.iLenCount]));
end;

var
    bMethod: Byte;
    s, s2: String;
begin
    WriteLn('Welche Methode soll für den Typecast verwendet werden (1-3)?');
    ReadLn(bMethod);
    if not (bMethod in [1..3]) then
    begin
        WriteLn('Ungültige Eingabe');
        ReadLn;
        Exit;
    end;

    WriteLn;
    WriteLn('Der Variable "s" wird ein Wert zugewiesen');
    WriteLn('  s := 'Das ist ein Buffer für GetWindowsDirectory'');

```

```

s := 'Das ist ein Buffer für GetWindowsDirectory';
WriteLn;

WriteLn('Der eingegebene Text wird einer zweiten Variable zugewiesen');
WriteLn('  s2 := s;');
s2 := s;

WriteLn;
WriteLn('Inhalt von s:');
ShowStringInfo(s);
WriteLn(Format('  Pointer(s):    0x%p', [Pointer(s)]));
WriteLn(Format('  @s:          0x%p', [@s]));
WriteLn;

WriteLn('Inhalt von s2:');
ShowStringInfo(s2);
WriteLn(Format('  Pointer(s2):   0x%p', [Pointer(s2)]));
WriteLn(Format('  @s2:         0x%p', [@s2]));
WriteLn;

WriteLn('Das Windows-Verzeichnis wird nun in die Variable "s"
eingelesen...');
case bMethod of
  1 :
    begin
      WriteLn('  GetWindowsDirectory(PChar(s), Length(s));');
      GetWindowsDirectory(PChar(s), Length(s));
    end;

  2 :
    begin
      WriteLn('  GetWindowsDirectory(@s[1], Length(s));');
      GetWindowsDirectory(@s[1], Length(s));
    end;

  3 :
    begin
      WriteLn('  GetWindowsDirectory(Pointer(s), Length(s));');
      GetWindowsDirectory(Pointer(s), Length(s));
    end;
end;

WriteLn;
WriteLn('Inhalt von s:');
ShowStringInfo(s);
WriteLn(Format('  Pointer(s):    0x%p', [Pointer(s)]));
WriteLn(Format('  @s:          0x%p', [@s]));
WriteLn;

WriteLn('Inhalt von s2:');
ShowStringInfo(s2);
WriteLn(Format('  Pointer(s2):   0x%p', [Pointer(s2)]));
WriteLn(Format('  @s2:         0x%p', [@s2]));
WriteLn;

ReadLn;
end.

```

Man hat am Anfang die Möglichkeit anzugeben, welche Typecast-Methode verwendet werden soll, und kann dann je nachdem die Ergebnisse vergleichen. Die 3 Methoden sind in derselben Reihenfolge wie sie in Kapitel 5 aufgeführt sind:

- Methode 1: `GetWindowsDirectory(PChar(s), Length(s));`
- Methode 2: `GetWindowsDirectory(@s[1], Length(s));`
- Methode 3: `GetWindowsDirectory(Pointer(s), Length(s));`

## 6.2 Ergebnisse und Konsequenzen

Je nachdem, welche Methode gewählt wird, schaut das Ergebnis anders aus (wobei die Methoden 1 und 3 identische Ergebnisse liefern, da ja beide den direkten Zeiger auf den Speicherbereich an die API-Funktion weitergeben). Hier der Vergleich, wie die beiden Variablen `s` und `s2` nach den Aufrufen von `GetWindowsDirectory()` ausschauen.

Methode 1 bzw. 3:

```
Inhalt von s:
String:      C:\WINNT ein Buffer für GetWindowsDirectory'
p.RefCount:  2
p.LenCount:  44
Pointer(s):  0x00BA0838
@s:          0x0040B84C

Inhalt von s2:
String:      C:\WINNT ein Buffer für GetWindowsDirectory'
p.RefCount:  2
p.LenCount:  44
Pointer(s2): 0x00BA0838
@s2:         0x0040B850
```

Methode 2:

```
Inhalt von s:
String:      C:\WINNT ein Buffer für GetWindowsDirectory'
p.RefCount:  1
p.LenCount:  44
Pointer(s):  0x00BA0914
@s:          0x0040B84C

Inhalt von s2:
String:      'Das ist ein Buffer für GetWindowsDirectory'
p.RefCount:  1
p.LenCount:  44
Pointer(s2): 0x00BA0838
@s2:         0x0040B850
```

Wie man sieht, sind die beiden Strings im ersten Fall identisch, obwohl der Windows-Pfad eigentlich nur in „s“ eingelesen wurde – ein kleiner Nebeneffekt der internen String-Handlings, der zur gemeinen Falle werden kann, wenn man sich dessen nicht bewusst ist.

Im zweiten Fall hingegen enthalten die beiden Strings, so wie es sein soll, unterschiedliche Zeichenketten – das Ergebnis des intern automatisch aufgerufenen „`UniqueString()`“. Womit auch klar sein sollte, mit welcher Typecast-Methode man einen String an eine API-Funktion (die Daten in diesen String schreibt) übergeben sollte, wenn man sich nicht sicher ist, dass dieser String ein „Unikat“ (mit einem Referenzzähler von 1) ist.

**Danksagung**

Ich möchte mich an dieser Stelle kurz bei all jenen bedanken, die bei der Erstellung dieses Tutorials mitgeholfen haben. Das sind zum einen all jene, die sich das Tutorial bereits im Vorfeld durchgelesen und dabei viele gute Vorschläge und Kritikpunkte eingebracht haben. Nur durch sie konnte das Tutorial zu dem werden, was es jetzt ist. Und zum anderen möchte ich mich noch bei meiner Freundin bedanken, die sich die Mühe gemacht hat, das ganze Korrektur zu lesen, obwohl sie wahrscheinlich kein Wort von all dem verstanden hat.