

KAPITEL 10

Zeiger - Der Datentyp POINTER

Dieses Kapitel wird Sie mit einem weiteren in Component Pascal existierenden Datentyp bekannt machen, den Zeigern, bei dem es sich um einen der vielseitigsten und in der Programmierarbeit sehr häufig gebrauchten Datentyp handelt. Zum Verständnis der Zeiger ist es nötig, sich ein wenig intensiver als bisher mit den Einzelheiten der Datenorganisation in einem Computer zu beschäftigen.

Wie bereits früher dargestellt, werden beim Start eines Moduls seine Daten in den Arbeitsspeicher geladen und vom Prozessor entsprechend den Programmanweisungen verarbeitet. Im Kapitel 3.1 habe ich Ihnen den Arbeitsspeicher als eine Art Schrank vorgestellt, in dem die Daten nach ihrer Größe und Struktur in "Schubladen" (Speicherzellen) untergebracht sind. Damit der Prozessor diese Speicherzellen adressieren kann, müssen sie unabhängig von der jeweiligen Verwendung in einem Programm gegliedert werden. Dafür stellt man sich die einzelnen Speicherzellen übereinander angeordnet vor, wobei jede Zelle mit einer fortlaufenden Nummer (Adresse) versehen ist, angefangen mit der untersten Zelle, die die Adresse Null besitzt. Beim Einschalten eines Computers werden, bei der Adresse Null anfangend, der Reihe nach alle nötigen Teile des Betriebssystems geladen, danach alle nötigen Teile aus den Anwenderprogrammen. Jedes (Teil-)Programm bleibt solange im Speicher, bis es explizit entladen wird, erst danach werden alle von ihm belegten Speicherzellen freigegeben. Dieser während der Laufzeit eines Programms dauerhaft belegte Speicher wird wegen der Art, in der er belegt worden ist, Stapel (engl. stack) oder Kellerspeicher genannt.

Naturgemäß droht jedem Computer früher oder später ein Speicherüberlauf, wenn immer neue Programmteile geladen, aber keine entladen werden, es wird stets weiterer Speicher belegt und irgendwann kommt der Moment, in dem nichts mehr geht. Um dieses Risiko ein wenig kleiner zu machen, hat man sich ein Verfahren ausgedacht, das den Namen Haldenspeicher oder kurz Halde (engl. heap) trägt. Dieser Haldenspeicher wird in einem freien Bereich des vorhandenen Arbeitsspeichers eingerichtet und neue Programmteile werden in dem Moment dorthin geladen, in dem sie benötigt werden. Selbstverständlich wäre dies Verfahren sinnlos, wenn die Verwaltung beider Speicherarten in identischer Form durchgeführt würde, da in diesem Fall kein einziges Byte an Speicher eingespart werden könnte. Im Gegensatz zum Kellerspeicher, der während der Existenzdauer eines Programms bzw. einer Prozedur unverändert bleibt, kann der Haldenspeicher jedoch programmgesteuert sowohl wachsen als auch schrumpfen, Zeiger machen dies möglich.

10.1. Zeiger sind Datenadressen

Die Unterschiede zwischen einer Zeigervariablen und allen anderen bisher vorgestellten Variablen möchte ich Ihnen an dem Programm `Zeiger.odc` erklären. Sie finden im Modulrumpf zwei Typen deklariert, `Name = ARRAY 19 OF CHAR` und `Kundenliste = ARRAY 2 OF Name`, außerdem zwei Variablen, deren Typdeklarationen den Bestandteil `POINTER TO` enthalten, der im Deutschen gesprochen wird als "Zeiger auf ...". Die erste Variablendeklaration `Kunde: POINTER TO Kundenliste` lautet also: "Kunde vom Typ Zeiger auf Kundenliste", wobei mit `Kundenliste` ein expliziter Basistyp verwendet wird. Die folgende Deklaration `Liste: POINTER TO ARRAY 2 OF Name` zeigt Ihnen, daß wie bei den strukturierten Datentypen `ARRAY` und `RECORD` auch bei Zeigern in Variablendeklarationen anonyme Typen verwendet werden können.

Worin besteht nun der Unterschied zwischen der Variablendeklaration `VAR Kunde: POINTER TO Kundenliste` und einer (nicht im Programm erscheinenden) Deklaration `VAR Kunde: Kundenliste`? Bei der zweiten Deklaration handelt es sich um eine statische Variablendeklaration, der benötigte Speicher ist durch den Variablentyp (`Kundenliste`) gegeben und wird in dem Moment im Kellerspeicher angelegt, in dem das Modul bzw. der entsprechende Modulteil in den Arbeitsspeicher geladen wird, wo für die Variable genau die durch den Variablentyp vorgegebene Speichergröße reserviert und, sofern es sich um eine global deklarierte Variable handelt, erst beim Entladen des Moduls wieder freigegeben wird; in dieser Weise festgelegter Speicher wird statischer Speicher genannt. Das Modul "weiß" während der gesamten Existenzdauer der Variablen, wo es den ihr zugewiesenen Speicher findet, es kann also jederzeit darauf zugreifen, den in der Variablen gespeicherten Wert lesen oder einen neuen Wert hineinschreiben.

Sie sehen, daß es sich bei statischem Speicher um eine verschwenderische und wenig flexible Art der Speichernutzung handelt. Es ist durchaus nicht selten, daß eine Variable nur ein einziges Mal oder nur für einen kurzen Zeitraum innerhalb eines Moduls benötigt wird, während ein geladenes Modul solange im Speicher bleibt, bis es explizit entladen wird. Dies Problem läßt sich mit einer dynamischen Variablendeklaration wie `VAR Kunde: POINTER TO Kundenliste` umgehen. Bei einer derartigen Variablen kann der Programmierer individuell festlegen, an welcher Stelle des Programmablaufs und für welchen Zeitraum der Variablen der für ihren Basistyp (`Kundenliste`) benötigte Speicherplatz zur Verfügung gestellt wird.

Dies ist aber nicht der einzige Grund für die Existenz dynamischer Variablen. Ein zweiter, vielleicht sogar wesentlicher Grund, auf den ich bereits im Kapitel 9.3 hingewiesen habe, liegt darin, daß es Situationen gibt, in denen man bei der Deklaration einer Variablen noch nicht alle ihre Eigenschaften endgültig festlegen möchte, die Datenstruktur soll flexibel bleiben, damit sie in verschiedenen, zwar ähnlichen aber nicht identischen Fällen verwendbar ist. Zu diesem Zweck eignen sich dynamische (Zeiger-) Variablen, deren Eigenschaften im Sinn der objektorientierten Programmierung zur Laufzeit modifiziert werden können, ohne den Quelltext des Moduls zu ändern oder dieses neu zu kompilieren.

Dynamische Variablen sind allerdings nicht völlig kostenlos. Zwar muß beim Laden des Moduls kein Platz für den eigentlichen Variableninhalt im Kellerspeicher reserviert werden, aber es wird Speicher für eine zusätzliche Variable belegt, einen "Zeiger auf ..." den Nutzinhalt. Diese Zeigervariable dient dazu, die Speicheradresse des eigentlichen Variableninhalts in dem Moment aufzunehmen, in dem während der Laufzeit des Programms für diesen Inhalt Speicher angefordert wird, sie "zeigt" also auf den (künftigen) Nutzinhalt der Variablen. Der Nutzinhalt selbst ist anonym, er wird nicht über eine eigene Variablendeklaration zugänglich gemacht, sondern kann nur über den auf ihn weisenden Zeiger adressiert werden.

Jeder Zeiger belegt unabhängig von dem Inhalt, der sich hinter ihm verbirgt, stets vier Byte im Kellerspeicher, während der für den Inhalt benötigte Speicher im Haldenspeicher angelegt wird und in seiner Größe von der jeweiligen Deklaration des Basistyps abhängt. Der Gesamtspeicherbedarf einer Variablen wird also um die genannten vier Byte erhöht. Dem steht eine Reihe von Vorteilen gegenüber, die Sie im Folgenden kennenlernen werden.

Einer dieser Vorteile ist die erwähnte Tatsache, daß der von der Basisvariablen benötigte Speicher in den meisten Fällen nur für kurze Zeit belegt ist, der für Programme verfügbare Speicher wird, über die gesamte Programmlaufzeit gesehen, faktisch größer. Die Unterschiede in der Art und Größe des Speicherbedarfs werden Ihnen vielleicht am deutlichsten, wenn Sie die beiden oben erwähnten Deklarationen vergleichen. Deklarieren Sie `Kunde` als `VAR Kunde: Kundenliste`, so belegt `Kunde` für die gesamte Existenz des Moduls 76 Byte Speicher (zur Erinnerung: Eine Variable vom Typ `CHAR` belegt 2 Byte Speicherplatz), wobei diese Größe dauerhaft fixiert ist. Bei der dynamischen Deklaration `VAR Kunde: POINTER TO Kundenliste` sind 4 Byte statisch für die Zeigervariable `Kunde` reserviert, 76 Byte für die anonyme Basisvariable des Zeigers `Kunde` jedoch dynamisch nur für den vom Programmierer oder Benutzer festlegbaren Zeitraum ihrer Existenz. Es ist unmittelbar einsichtig, daß auf diese Weise Vorteile zu erwarten sind, weil der für die Basisvariable reservierte Haldenspeicher flexibel angefordert und freigegeben werden kann, sobald er nicht mehr benötigt wird.

Wie erfolgt nun die Bereitstellung von dynamischem Speicher für den Nutzinhalt? Aus dem dem Zeiger zugeordneten Basistyp (`Kundenliste` im Fall von `Kunde`) wird der benötigte Speicher (76 Byte) ermittelt und im Haldenspeicher an einer aktuell freien Position reserviert. In den Speicherplatz, den die Zeigervariable `Kunde` selbst im Kellerspeicher belegt, wird als Wert die im Haldenspeicher liegende Startadresse des der Zeigervariablen zugeordneten Nutzinhalts eingetragen, so daß dieser Inhalt von da an über die Zeigervariable zugänglich ist. Es ist jedoch äußerst wichtig, sich klarzumachen, daß der Inhalt des so reservierten Speicherplatzes dadurch noch nicht festgelegt ist; bevor er benutzt werden kann, muß er genau wie bei allen anderen Variablen zunächst zugewiesen werden.

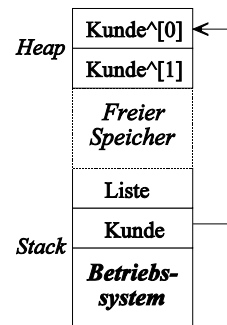


Abbildung 14
Speicherbelegung nach `NEW(Kunde)`

Im Anweisungsteil der Prozedur `Start` finden Sie dementsprechend als erstes die Speicherallokation für den Zeiger `Kunde` über die Standardprozedur `NEW` und in den anschließenden beiden Programmzeilen die Zuweisung von Werten an die Variablenfelder. Gleichzeitig zeigen Ihnen diese Zuweisungen, wie Sie an die Inhalte der Zeiger herankommen - man sagt auch, die Zeigerinhalte "dereferenzieren" - können. Dazu wird an die Zeigervariable der "Dereferenzierungsoperator" `^` (der Pfeil oder Zeiger) gehängt. Auf diese Weise wird das Programm angewiesen, als erstes zu der ihm bekannten Adresse des Zeigers im Kellerspeicher zu gehen, dort die Adresse des im Haldenspeicher befindlichen Zeigerinhalts zu lesen und diese Stelle aufzusuchen, um den eigentlichen Inhalt, die `ARRAY`-Variable `Kunde^` vom Typ `Kundenliste` zu bearbeiten. Die Programmzeile `Kunde^[0] := "Heinz Gramvoll"` speichert folglich die Zeichenkette "Heinz Gramvoll" in dem ersten der beiden Felder der der Zeigervariablen `Kunde` zugeordneten Basisvariablen `Kunde^`, wobei diese Speicherung nicht im Kellerspeicher, sondern im Haldenspeicher erfolgt. Entsprechend speichert die folgende Programmzeile `Kunde[1] := "Karla Sorgenfrei"` die Zeichenkette "Karla Sorgenfrei" im zweiten Feld von `Kunde^`.

Es sollte Ihnen auffallen, daß in dieser zweiten Zuweisung der Dereferenzierungsoperator `^` fehlt. Die Grundregel lautet, daß bei Zeigern zwischen der Zeigervariablen (z. B. `Kunde`) und der zugehörigen anonymen Basisvariablen (hier `Kunde^`) unterschieden werden muß. Grundsätzlich ist die Basisvariable über den Dereferenzierungsoperator `^` zu adressieren. In allen Fällen, in denen aus dem Zusammenhang eindeutig hervorgeht, ob es sich um Zeigervariablen oder Basisvariablen handelt, erlaubt Component Pascal die implizite Dereferenzierung, der Operator `^` darf weggelassen werden. Dies stellt für den Programmschreiber eine große Erleichterung dar, wie Sie bei Ihrer Arbeit mit Zeigern bald feststellen werden, allerdings muß man sich trotzdem (oder gerade deshalb) darüber klar sein, ob die jeweils benutzte Variable eine Zeigerva-

riable oder die von dem Zeiger referenzierte Basisvariable ist, es gibt Situationen, in denen beide Interpretationen sinnvoll sind. In diesen Fällen muß der Programmierer für die nötige Klarheit sorgen, denn der Compiler unterstellt bei fehlendem Dereferenzierungsoperator prinzipiell stets eine Zeigervariable. Sie werden diese Unterschiede weiter unten und im nächsten Programm `DatDyn.odc` genauer kennenlernen.

Der den Zuweisungen folgende Block des Anweisungsteils der Prozedur `Start` stellt keine Besonderheit dar, die gerade gespeicherten Inhalte der Variablen werden ausgelesen und auf dem Bildschirm ausgegeben. Zur Verdeutlichung der eben erläuterten impliziten Dereferenzierung finden Sie in den Ausgabeanweisungen für die Kundennamen noch einmal die Schreibweise mit und ohne Dereferenzierungsoperator.

Wie erfolgt nun der zweite Teil der Verwaltung von Haldenspeicher, die Freigabe nicht mehr benötigter Bereiche? Component Pascal ist, wie der Name ausdrückt, eine Sprache, mit der Software-Komponenten erstellt werden können. Solche Komponenten existieren in offenen Systemumgebungen, in denen anders als bei abgeschlossenen Programmen zur Kompilationszeit eines Programms nicht festgestellt werden kann, welche anderen Komponenten zur Laufzeit auf eine existierende Komponente (oder einen ihrer Teile) zugreifen werden. Daher ist eine explizite Freigabe von dynamischem Speicher nicht möglich. Vielmehr verwaltet das System alle dynamisch allozierten Speicherbereiche, indem es in den Leerlaufzeiten des Prozessors prüft, ob ein Speicherbereich noch referenziert ist, ob also noch wenigstens ein Zeiger auf diesen Speicherbereich existiert. Sofern dies nicht der Fall ist (und nur dann), wird der fragliche Speicher freigegeben und kann anschließend wiederverwendet werden. Diese Art der Speicherverwaltung sammelt also den "Speichermüll" ein, sie wird deshalb im Amerikanischen `garbage collection` genannt.

Eine weitere damit zusammenhängende Neuigkeit dieses Programms lernen Sie in den abschließenden Zeilen der Prozedur `Start` kennen. Mit der Zeile `Kunde := NIL` wird der Zeigervariablen `Kunde` der spezielle Wert `NIL` zugewiesen. Es handelt sich dabei um eine symbolische Speicheradresse, die auf "Nichts" weist. Nun werden Sie fragen, wozu der Wert `NIL` dient, eine Antwort finden Sie in den folgenden Zeilen. Wegen des Wertes `NIL` kann der Compiler den Versuch, auf die anonyme Basisvariable `Kunde^` zuzugreifen, als illegal erkennen und gegebenenfalls mit einer Fehlermeldung reagieren. Weiter kann der Programmschreiber eine Zeigervariable daraufhin prüfen, ob sie den Wert `NIL` enthält und geeignete Anweisungen zur Vermeidung einer Katastrophe in das Programm einfügen, Sie sehen ein Beispiel für diese im Umgang mit Zeigervariablen wichtige Möglichkeit in der `IF`-Abfrage von `Start`.

Abschließend möchte ich Sie an Hand der in Kommentarklammern stehenden `ASSERT`-Prozedur noch einmal auf den Unterschied aufmerksam machen, der zwischen einer Zeigervariablen (`Kunde`) und der zugeordneten Basisvariablen (`Kunde^`) besteht (zur `ASSERT`-Prozedur siehe das Programm `ProcTyp.odc` und den Anhang C). Entfernen Sie die Kommentarklammern um den Aufruf dieser Prozedur, lassen Sie das Modul neu kompilieren und ausführen, so werden Sie die daneben als Kommentar stehende Fehlermeldung erhalten, da der Compiler erkennt, daß es sich bei `Kunde^` nicht um eine Zeigervariable, sondern um die zugeordnete anonyme Variable des Zeigerbasistyps `Kundenliste`, also um eine `ARRAY`-Variable handelt,

weshalb die beiden Seiten des versuchten Vergleichs als inkompatibel beanstandet werden. Dieser Unterschied zwischen Zeigern und Zeigerbasisvariablen soll im folgenden Modul *TutDatDyn* ausführlich erläutert werden.

10.2. Daten und Zeiger - Speichernutzung dynamisch

In dem Programm *DatDyn.odc* finden Sie den "Zwilling" von *DatRec.odc*, dem Programm, das Sie mit der Verwendung von Verbunden bekannt gemacht hat. Verbunde sind, wie erwähnt, wegen ihrer Vielseitigkeit und der Möglichkeit, mit Ihnen Daten zu kapseln, ein oft genutzter Datentyp. Andererseits sind Verbunde speicherintensive Datentypen, ein Programm, das ausgiebig Gebrauch von Verbundvariablen macht, kann leicht die Kapazität eines Computers überfordern. Dies ist einer der Gründe, warum man insbesondere bei Verbundvariablen Gebrauch von der im vorigen Abschnitt beschriebenen Möglichkeit der flexiblen Speichernutzung durch Zeiger macht.

Wegen der inhaltlichen Ähnlichkeit von *DatDyn.odc* und *DatRec.odc* werden Sie keine Verständnisschwierigkeiten bei dem vorliegenden Programm haben, ich beschränke mich daher bei der Erläuterung auf die - allerdings wesentlichen - Unterschiede. Als erste Änderung gegenüber dem Programm *DatRec.odc* finden Sie die zusätzliche Deklaration des gesonderten Verbundtyps *Name*, der den anonymen Typ des Verbundfeldes *GanzerName* in *DatRec.odc* ersetzt, und als Neuigkeit darin die Änderung der Felddeklaration *Nachname: ARRAY 15 OF CHAR* in *Nachname: POINTER TO ARRAY OF CHAR*. Diese Deklaration stellt Ihnen eine andere Art von *ARRAY*-Typ vor, ein dynamisches *ARRAY*. Bei einem dynamischen *ARRAY* fehlt die Angabe der *ARRAY*-Länge, es handelt sich ähnlich wie bei offenen *ARRAY*-Parametern in Prozedursignaturen um *ARRAY*-Typen unbestimmter Länge, die tatsächlichen Längen der Variablen eines dynamischen *ARRAY*-Typs müssen erst zur Laufzeit des Programms festgelegt werden, ich gehe auf die Einzelheiten weiter unten im Zusammenhang mit der Prozedur *Start* ein.

Die zweite Änderung betrifft den Verbundtyp *Anschrift*, der in *AnschriftDescriptor* umbenannt worden ist. Die damit zusammenhängende Einführung des im Anschluss daran stehenden Zeigertyps *Anschrift = POINTER TO AnschriftDescriptor* zeigt Ihnen die oben erwähnte Möglichkeit, außer den bisherigen Deklarationen von Zeigertypen als *POINTER TO ARRAY ...* auch den Datentyp *RECORD* als Basistyp von Zeigertypen verwenden zu können. Durch die Umbenennung des Verbundtyps *Anschrift* in *AnschriftDescriptor* und die Einführung des Zeigertyps *Anschrift = POINTER TO AnschriftDescriptor* ist aus dem Verbundtyp *Anschrift* des Programms *DatRec.odc* ein gleichnamiger Zeigertyp geworden. Daraus ergeben sich zwei wesentliche Vorteile. Einerseits ermöglicht die - einheitliche - Verwendung des Zusatzes "Descriptor" (oder der Kurzform "Desc", die Sie in den folgenden Programmen finden werden) die schnelle Identifizierung eines Verbundtyps als Basistyp eines Zeigertyps, der das gleiche Präfix als Bezeichner hat;

auch in großen und deshalb leicht unübersichtlichen Programmen ist es dadurch möglich zu erkennen, daß es sich um zusammengehörende Partner handelt. Andererseits kommt es nicht selten vor, daß man sich als Programmierer bei der Weiterentwicklung von Programmen irgendwann entscheidet, einen ursprünglich als Verbund eingeführten Typ zu "dynamisieren", den Verbund also zum expliziten Basistyp eines Zeigertyps zu machen. In solchen Fällen kann man, mit wenigen sonstigen Änderungen, den Typbezeichner des Verbundes für den Zeiger und damit für alle Variablen dieses Typs beibehalten und lediglich den Basistyp umbenennen. Eine zweite, noch einfachere Weise solche Änderungen durchzuführen, werden Sie später kennenlernen, dabei nutzt man die im vorigen Abschnitt erwähnte Möglichkeit aus, in Component Pascal Zeigertypen auch anonym vereinbaren zu können (Beispiele dazu finden Sie in den Programmen 10.5 und 10.5a).

Die Prozedur *TutDatDyn.Schreib* ist mit *TutDatRec.Schreib* nahezu identisch, die erste Änderung finden Sie gleich zu Beginn in einem Kommentar erläutert. Da *Anschrift* als Zeigervariable nur wenig (4 Byte) Speicher benötigt, kann dieser Parameter wie Parameter einfacher Datentypen als VAL-Parameter vereinbart werden, während dies bei Verbundvariablen wegen des Speicherbedarfs nur in zwingenden Ausnahmefällen geschehen sollte.

Die zweite Veränderung sehen Sie ebenfalls gleich zu Beginn der Prozedur, für jede deklarierte Zeigervariable muß mit der Standardprozedur *NEW* im "heap" der benötigte Speicher angefordert werden, bevor die Zuweisung von Werten an die einzelnen Felder erfolgt. Dies gilt sowohl für die Variable *Er* vom Typ *Anschrift* als auch für das Variablenfeld *Er.GanzerName.Nachname*, die beide Zeigertypen haben. Während Sie die Verwendung von *NEW* für den Zeiger *Er* bereits aus dem vorigen Programm kennen, gibt es bei der Allokation von Speicher für dynamische *ARRAY*-Variablen eine Neuigkeit, die Standardprozedur *NEW* hat für solche Variablen einen zweiten Parameter. Mit diesem zweiten Parameter wird die Länge einer *ARRAY*-Variablen zur Laufzeit bei deren Allokation festgelegt, nicht wie bei einem *ARRAY*-Typ bereits zur Kompilationszeit bei der Deklaration der Variablen, als Programmierer sind Sie also ähnlich wie bei offenen *ARRAY*-Parametern in Prozeduren nicht gezwungen, dem Programmbenutzer "falsche" Vorschriften in Bezug auf die Größe der von ihm verwendeten Variablen machen zu müssen. Allerdings müssen Sie auch bei dynamischen *ARRAY*-Typen die bereits im Programm *String.odc* im Zusammenhang mit Zeichenketten erwähnten Risiken offener *ARRAY*-Deklarationen beachten.

Ein weiterer für die Programmierarbeit wichtiger Punkt betrifft die im vorigen Absatz erwähnte Notwendigkeit, zwischen einer Zeigervariablen und ihrer Basisvariablen zu unterscheiden. Sie sehen an der (in Kommentarklammern stehenden) vierten - und der fünften Anweisung der Prozedur *Start*, daß Sie eine Fehlermeldung erhalten, wenn Sie versuchen, nicht der String-Variablen *Er.GanzerName.Nachname*, sondern dem Zeiger *Er.GanzerName.Nachname* die Zeichenkette "*Brecht*" zuzuweisen. An dieser Stelle muß zwischen dem Zeiger und der Basisvariablen explizit unterschieden werden. Ein weiteres Mal sehen Sie

die Notwendigkeit der Dereferenzierung mit dem Operator \wedge in der Zeile $Du^\wedge := Er^\wedge$, mit der der Inhalt der Basisvariablen Er^\wedge in die *RECORD*-Variable Du^\wedge kopiert wird.

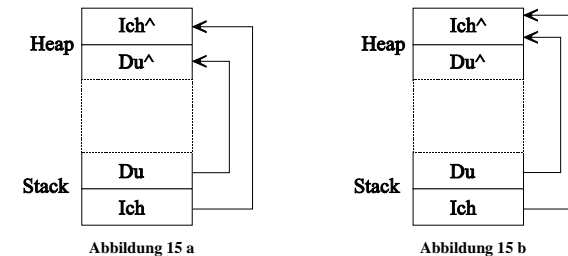
Die nächsten Programmzeilen erläutern einige wesentliche Punkte, die in diesem Zusammenhang zu beachten sind. Die Zeile `ASSERT(Du.GanzerName.Nachname = Er.GanzerName.Nachname, 60)` prüft die Einhaltung eines Teils des davor stehenden Programmschritts, die Gleichheit zweier Zeigerwerte. Diese beiden Werte sind gleich, da bei dem Kopiervorgang alle Datenfelder von Er^\wedge kopiert wurden, mit dem Feld *Nachname* also auch die Speicheradresse des *ARRAYs* $Nachname^\wedge$. Wenn Sie nach dem ersten Testen des Programms das "=" Zeichen durch ein "#" Zeichen ersetzen, das Programm neu kompilieren und ausführen lassen, werden Sie eine entsprechende Fehlermeldung des Systems (*TRAP 60 (postcondition violated)*) erhalten, die Ihnen die Gleichheit bestätigt.

Die folgende Zeile `NEW(Du.GanzerName.Nachname, 15)` alloziert neuen Speicher für das dynamische *ARRAY* $Du.GanzerName.Nachname^\wedge$, wodurch sich der Wert des Zeigers $Du.GanzerName.Nachname$ notwendigerweise von dem bisherigen Wert unterscheidet, diese Ungleichheit wird in der anschließenden Zeile `ASSERT(Du.GanzerName.Nachname # Er.GanzerName.Nachname, 61)` geprüft und bestätigt. Erst nach der Allokation kann mit der nächsten Zeile $Du.GanzerName.Nachname^\wedge := Er.GanzerName.Nachname^\wedge$ der Brecht'sche Name kopiert werden, und die folgende Zeile `ASSERT(Du.GanzerName.Nachname # Er.GanzerName.Nachname, 62)` bestätigt, daß die Zeiger weiterhin verschiedene Speicherbereiche referenzieren. Die beiden Aufrufe `NEW(Du)` und `NEW(Du.GanzerName.Nachname, 15)` erzeugen also zusammen mit den Zuweisungen $Du^\wedge := Er^\wedge$ und $Du.GanzerName.Nachname^\wedge := Er.GanzerName.Nachname^\wedge$ eine vollständige Kopie (engl. deep copy) des ursprünglich Er^\wedge zugewiesenen Speicherinhalts, dieser ist danach insgesamt zweimal vorhanden.

Die anschließenden beiden Anweisungen `NEW(Ich)` und `NEW(Ich.GanzerName.Nachname, 15)` allozieren in analoger Weise Speicher für die anonymen Variablen Ich^\wedge sowie $Ich.GanzerName.Nachname^\wedge$ und die folgenden Zusicherungen zeigen, daß die allozierten Speicherbereiche unterschiedliche Adressen besitzen. Die nächste Zuweisung $Ich^\wedge := Du^\wedge$ kopiert neben allen anderen Feldern auch das Feld $Du.GanzerName.Nachname$, daher sind die Adressen der anonymen *ARRAY*-Variablen $Du.GanzerName.Nachname^\wedge$ und $Ich.GanzerName.Nachname^\wedge$ anschließend ebenso identisch wie es die beiden Adressen $Er.GanzerName.Nachname$ und $Du.GanzerName.Nachname$ nach der Zuweisung $Du^\wedge := Er^\wedge$ gewesen sind. Der wesentliche Unterschied ergibt sich aus der vertauschten Reihenfolge der Anweisungen $Du^\wedge := Er^\wedge$ und `NEW(Du.GanzerName.Nachname, 15)` einerseits sowie andererseits der Anweisungen `NEW(Ich.GanzerName.Nachname, 15)` und $Ich^\wedge := Du^\wedge$. Die Zusicherungen des Programms zeigen Ihnen, daß im ersten Fall die Zeiger $Du.GanzerName.Nachname$ und $Er.GanzerName.Nachname$ verschiedene, im zweiten Fall aber die Zeiger $Ich.GanzerName.Nachname$ und $Du.GanzerName.Nachname$ identische Werte haben. Im ersten Fall bleiben nach der Zuweisung $Du.GanzerName.Nachname^\wedge := Er.GanzerName.Nachname^\wedge$ die beiden darauf verweisenden Zeiger $Du.GanzerName.Nachname$ und $Er.GanzerName.Nachname$

$Name.Nachname$ verschieden, die Inhalte sind an verschiedenen Speicherplätzen existierende, inhaltlich identische Kopien. Dagegen ist im zweiten Fall der mit der Anweisung `NEW(Ich.GanzerName.Nachname, 15)` allozierte Speicherbereich nicht mehr zugänglich, die Zeiger $Du.GanzerName.Nachname$ und $Ich.GanzerName.Nachname$ referenzieren den selben Speicher. Die weiter unten in *Start* erfolgende Zuweisung $Du.GanzerName.Nachname^\wedge := "Weill"$ hätte daher ebensogut $Ich.GanzerName.Nachname^\wedge := "Weill"$ lauten können, das Ergebnis wäre gleich gewesen. Die Ausgabeanweisung `Schreib(Ich)` liefert folglich den Nachnamen von Du , obwohl Ich und alle seine Felder inzwischen nicht geändert wurden.

Die Zuweisung $Du := Ich$, die Sie weiter unten im Programm finden, erzeugt eine analoge Situation, allerdings in noch krasserer Weise, sie kopiert den Wert der Zeigervariablen Ich in die Zeigervariable Du (engl. shallow copy), beide Zeiger referenzieren anschließend den selben Speicherbereich (s. Abb. 15). Der gesamte vorher über den Zeiger Du erreichbare Speicher und sein Inhalt sind jetzt unzugänglich, denn dieser kann nur über einen auf ihn weisenden Zeiger adressiert werden, den es nach dem "Umbiegen" von Du auf Ich^\wedge jedoch nicht mehr gibt.



$Du := Ich$
Umbiegen des Zeigers Du auf Ich^\wedge

In älteren Programmiersprachen hat eine solche Situation fatale Wirkungen. Selbst wenn der Programmierer das Umbiegen des Zeigers absichtsvoll durchführt, weil er den alten Speicherinhalt nicht mehr benötigt, ist der nicht länger referenzierte Speicher im System weiterhin als belegt eingetragen und kann nicht wieder verwendet werden, er ist "Speichermüll" geworden. Zur Vermeidung solchen Speichermülls muß in diesen Sprachen der nicht länger erforderliche, mittels *NEW* allozierte Speicher explizit an das System zurückgegeben werden, Vorgängersprachen von Component Pascal wie Modula-2 enthalten zu diesem Zweck eine zu *NEW* inverse Operation *DISPOSE*, mit der die Freigabe dynamischen Speichers vom Programmierer durchgeführt werden kann. Component Pascal benötigt (und kennt) wegen seiner dynamischen Speicherverwaltung (garbage collection) keinen *DISPOSE*-Operator, für den Programmierer entfällt damit eine Quelle häufiger und schwerwiegender Fehler.

Da global deklarierte Variablen während der gesamten "Lebensdauer" eines Moduls existieren, von ihnen beanspruchter Speicher also auch bei Zeigervariablen wie bei statischen Variablen belegt bleibt, sollte dieser Speicher, wenn er nicht mehr benötigt wird, explizit durch die Zuweisung von *NIL* für den garbage collector freigegeben werden. Wie Sie am Ende des Programms als Kommentar angemerkt finden, sollten Sie auch allen anderen nicht länger benötigten Zeigervariablen den Wert *NIL* zuweisen, Sie erleichtern dem garbage collector die Arbeit, die Stabilität Ihrer Programme wird es Ihnen danken.

Im Zusammenhang damit möchte ich Sie auf drei in Kommentarklammern stehende Zeilen aufmerksam machen, die Ihnen noch einmal den Unterschied zwischen Zeigern und ihren Basisvariablen verdeutlichen. In der ersten dieser Zeilen - *Schreib(Du[^])* - wird versucht, der Prozedur *Schreib* einen Verbund als Parameter zu übergeben. Obwohl die formalen Parameter von *Schreib* (bis auf den Zusatz *VAR*) in den beiden Modulen *TutDatRec* und *TutDatDyn* namentlich identisch sind, handelt es sich um inhaltlich unterschiedliche Parameter, da der Typ *Anschrift* im einen Fall einen Verbund, im anderen jedoch einen Zeiger darstellt. Das Gleiche gilt für die Zeile *Schreib(Temp)*, es wird der Versuch gemacht, einen Verbund an einen Zeigerparameter zu übergeben und auch bei der Zeile *Temp := Bekannte[^][1]* liegt eine ähnliche Situation vor. *Bekannte* ist ein Feld mit Zeigern (*Anschrift*) als Basisvariablen, während es sich bei *Temp* um einen Verbund handelt. Machen Sie sich bitte den Unterschied zwischen dieser Zeile und der folgenden Zuweisung *Temp := Bekannte[1][^]* deutlich, bei *Bekannte[1][^]* handelt es sich um eine anonyme Verbundvariable, auf die der Zeiger *Bekannte[1]* verweist, deren Wert kann folglich in die Verbundvariable *Temp* kopiert werden.

10.3. Statische und dynamische Typen

Im Kapitel 8 haben Sie im Zusammenhang mit den Programmen *ErbRec2.odc* und *ErbRec3.odc* einige Begriffe der objektorientierten Programmierung (OOP) kennengelernt, drei wesentlichen von ihnen - Klassen, Methoden und Objekte - begegnen Sie in diesem Abschnitt wieder. Wie Sie erinnern werden, ist in Component Pascal eine Klasse gegeben durch einen erweiterbaren Verbundtyp zusammen mit den auf den Objekten (Variablen) des Typs definierten Prozeduren, die in OOP-Terminologie Methoden genannt werden.

Während Ihnen das Programm *ErbRec3.odc* typgebundene Prozeduren als Methoden von Verbundvariablen vorgestellt hat, zeigt Ihnen das Programm *ErbRec4.odc*, in welcher Weise typgebundene Prozeduren im Zusammenhang mit Zeigern verwendet werden können. Als erste Deklaration finden Sie im Modul *TutErbRec4* den Ihnen aus *TutErbRec3* bekannten Verbundtyp *Rikscha* wieder. Auch die folgenden Deklarationen der Typen *Boot* und *BootZeit* sind in beiden Modulen inhaltlich im wesentlichen identisch,

allerdings sind im Modul *TutErbRec4* beide Typen in der aus dem Modul *TutDatDyn* bekannten Weise zu Zeigertypen erweitert worden.

Die unterschiedliche Deklarationsreihenfolge der Typen *BootDesc = EXTENSIBLE RECORD (TutErbRec2.Boot)* und *Boot = POINTER TO BootDesc* einerseits sowie *BootZeit = POINTER TO BootZeitDesc* und *BootZeitDesc = RECORD (BootDesc)* andererseits zeigt Ihnen eine Erweiterung des Prinzips, daß in Component Pascal nur verwendet werden kann, was zuvor erklärt wurde. Für Typdeklarationen gilt, daß benutzerdeklarierte Typen innerhalb ihres Gültigkeitsbereichs generell bekannt sind (implizite FORWARD-Deklaration), der Zeigertyp *BootZeit = POINTER TO BootZeitDesc* kann vor dem Typ *BootZeitDesc* deklariert werden.

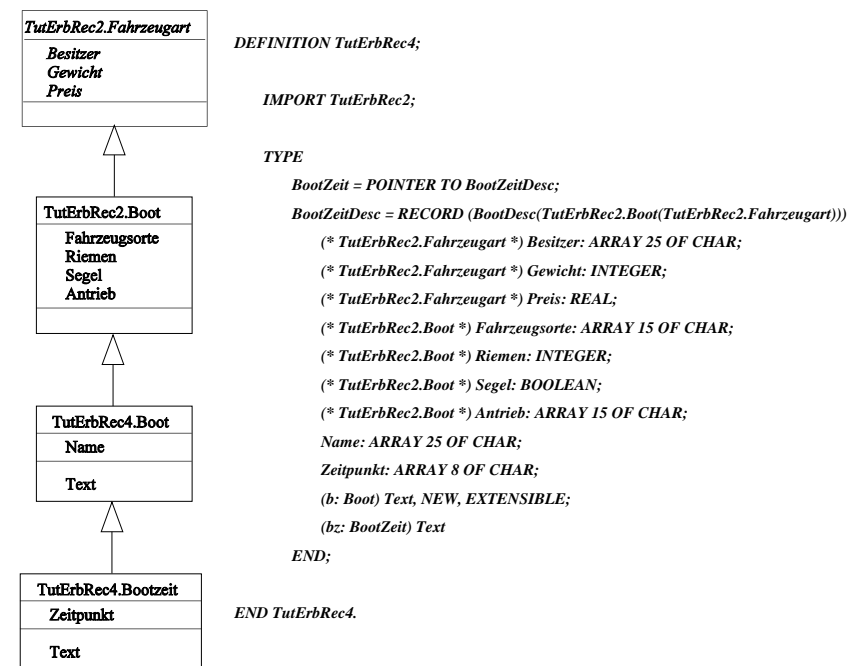


Abbildung 16
Vererbungshierarchie der Klasse *TutErbRec4.Bootzeit*
als UML Diagramm und als Schnittstellenauszug

Im Anschluss an die Typdeklarationen werden in *TutErbRec4* ebenso wie in *TutErbRec3* drei Methoden als typgebundene Prozeduren deklariert. Die erste, *PROCEDURE (IN r: Rikscha) Text, NEW*, ist durch den Empfängerparameter an die Klasse *Rikscha* gebunden, wobei das Schlüsselwort *IN* deutlich macht, daß es sich bei *Rikscha* um eine Klasse des Typs *RECORD* handelt. Im Gegensatz dazu haben die Empfängerparameter in den beiden folgenden Signaturen, *PROCEDURE (b: Boot) Text, NEW, EXTENSIBLE* und *PROCEDURE (bz: BootZeit) Text* keine Zusätze, sie sind *VAL*-Parameter. Auf diese Weise wird eine typgebundene Prozedur einer als Zeigertyp deklarierten Klasse zugeordnet, sie kann nur von einer (Zeiger-)Variablen des zugehörigen Typs aktiviert werden. Im übrigen haben typgebundene Prozeduren von Zeigerklassen die im 8. Kapitel bei dem Programm *ErbRec3.odc* erläuterten Eigenschaften, insbesondere können sie alle für typgebundene Prozeduren existierenden Attribute wie *NEW* oder *EXTENSIBLE* erhalten.

Abbildung 16 zeigt im Vergleich die Vererbungshierarchie der Klasse *TutErbRec4.Bootzeit* in UML Notation und in der Ihnen bekannten Form als Schnittstellenbeschreibung (die Klassen sind zur Vereinfachung mit den Namen der Zeiger identifiziert worden). UML Notation ist Ihnen im 9. Kapitel in Zusammenhang mit dem MVC Muster begegnet. In UML kann eine Klasse insgesamt drei Felder enthalten, der Name der Klasse steht im obersten Feld, im mittleren Feld sind die - in UML Attribute genannten - Datenfelder der Klasse zu finden und im untersten Feld ihre Methoden (s. a. Anhang D). Außerdem unterscheidet UML zwischen konkreten und abstrakten Klassen. Im 8. Kapitel habe ich erläutert, daß abstrakte Klassen nicht instantiiert werden können, d. h. es können keine Objekte solcher Klassen deklariert werden. Abstrakte Klassen werden in UML durch Kursivschrift gekennzeichnet, konkrete Klassen sind in Normalchrift dargestellt. UML Diagramme sind nützliche Mittel zur Visualisierung von Klassenbeziehungen, die Gegenüberstellung der Abbildung 16 zeigt, daß mit ihnen die wesentlichen Eigenschaften von Klassen übersichtlich und in einer programmiersprachenunabhängigen Weise dargestellt werden können.

Eine wichtige Neuigkeit des Moduls *TutErbRec4*, die aus der Verwendung von Zeigertypen resultiert, finden Sie in der Kommandoprozedur *Start*. Während in *TutErbRec3.Start* die Zuweisung *Nautilus := MolaMola* vom Compiler als unzulässig beanstandet wird, gibt es in *TutErbRec4.Start* dabei keine Probleme, anscheinend entgegen den im 8. Kapitel erläuterten Regeln, wonach die Zuweisung einer Verbundvariablen an eine zweite von anderem Typ nicht möglich ist. An dieser Stelle ist es wichtig sich klarzumachen, daß kein Verstoß gegen die Regeln vorliegt, da die Zuweisung *Nautilus := MolaMola* in *TutErbRec4.Start* etwas anderes als die (nicht vorhandene) Zuweisung *Nautilus^ := MolaMola^* bedeutet.

Eine solche Zuweisung stellte tatsächlich den Versuch dar, die Verbundvariable *MolaMola^* in die Verbundvariable *Nautilus^* zu kopieren, was in der konkreten Konstellation des Programms aus zwei Gründen nicht realisierbar wäre. Erstens ist die Verbundvariable *Nautilus^* nicht vom selben Typ *BootZeit* wie *MolaMola^*, das würde vom Compiler bemerkt und beanstandet. Aber selbst wenn *Nautilus* nicht wie im Programm als *Nautilus: Boot*, sondern als *Nautilus: BootZeit*, dem Typ von *MolaMola*, deklariert wäre,

gäbe es zwar bei der Zuweisung *Nautilus^ := MolaMola^* keine Beschwerden durch den Compiler, es gäbe jedoch bei der Ausführung des Programms einen Laufzeitfehler (*TRAP NIL dereference (read)*). Der versuchte Kopiervorgang ließe sich nicht durchführen, da das Objekt *Nautilus^* nicht alloziert ist (die entsprechende Anweisung *NEW(Nautilus)* steht zur Verdeutlichung in Kommentarklammern in der Prozedur *Start*). Dagegen ist die Zuweisung *Nautilus := MolaMola* möglich, denn mit ihr wird lediglich ein Zeiger (eine Speicheradresse) kopiert, der Zeiger *Nautilus* zeigt anschließend auf das Objekt *MolaMola^* (s. Abb. 17).

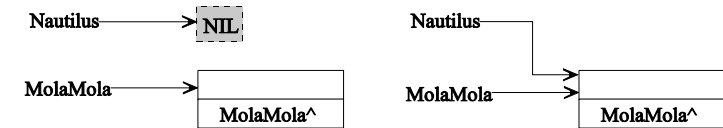


Abbildung 17
Die Zeiger *MolaMola* und *Nautilus* vor und nach der Zuweisung
Nautilus := MolaMola

Sie sehen, wie sich mit Zeigern die im 8. Kapitel erwähnte Möglichkeit ergibt, ein Objekt (eine Verbundvariable) mit einem anderen Namen zu versehen, die Einheit von Name, Typ und Inhalt eines Objekts wird aufgehoben, das Objekt *MolaMola^* ist jetzt sowohl über den Namen *MolaMola* als auch über den Namen *Nautilus* adressierbar. Für die Programmierarbeit ergeben sich daraus weitreichende Konsequenzen, die Sie in diesem und den folgenden Abschnitten des Kapitels und ausführlich im nächsten Kapitel unter dem Begriff Polymorphie kennenlernen werden.

Erste Konsequenz ist die Tatsache, daß durch die Zuweisung *Nautilus := MolaMola* das Objekt *Nautilus* seine Identität gewechselt hat, die Zeiger *Nautilus* und *MolaMola* weisen nach der Zuweisung auf den selben Verbund. Die im Anschluss an die Zuweisung zu findende Zeile *Nautilus.Name := "Nautilus"* ersetzt den bis dahin im Feld *Name* des Verbundes gespeicherten Wert "*MolaMola*" durch "*Nautilus*". Darin scheint nichts besonderes zu liegen, denn das Feld *Name* ist in der gemeinsamen Basisklasse *Boot* der beiden Variablen deklariert und damit Bestandteil jedes der (ursprünglich verschiedenen) Objekte *Nautilus* und *MolaMola*. Die inhaltliche Identität der beiden Objekte hat jedoch weitere Folgen. Die in den Zeilen *IF Nautilus.Name = "Nautilus" THEN Nautilus.Text END* zu findende Aktivierung der Methode *Nautilus.Text* gibt nicht, wie man vielleicht erwarten könnte, den Inhalt der zu *Boot*, dem statischen Typ von *Nautilus* gehörenden Methode *PROCEDURE (b: Boot) Text* auf dem Bildschirm aus, sondern den Inhalt der zum statischen Typ des Objekts *MolaMola* gehörenden Methode *PROCEDURE (bz: BootZeit) Text*.

Machen Sie sich bitte klar, daß die Anweisung ebensogut *IF MolaMola.Name = "Nautilus" THEN Nautilus.Text END* wie *IF MolaMola.Name = "Nautilus" THEN MolaMola.Text END* wie auch *IF Nautilus*

lus.Name = "Nautilus" THEN MolaMola.Text END lauten könnte, das Ergebnis wäre in allen vier Fällen gleich. Auf Grund der Zuweisung *Nautilus := MolaMola* hat das Objekt *Nautilus* seinen ursprünglichen, statischen Typ *Boot* verloren und den dynamischen Typ *BootZeit* bekommen.

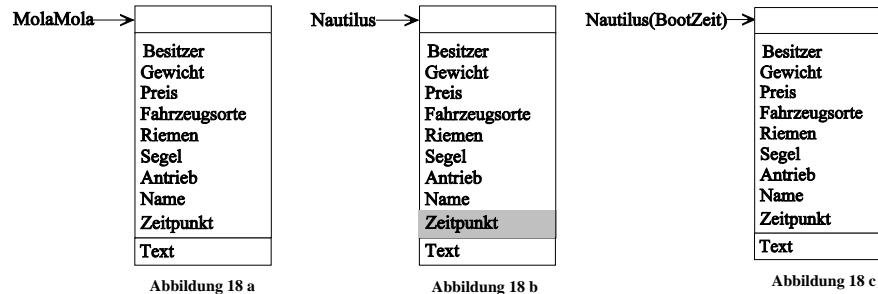


Abbildung 18 a

Abbildung 18 b

Abbildung 18 c

Maskiertes Feld *Zeitpunkt* im Objekt *Nautilus* nach der Zuweisung *Nautilus := MolaMola* und Objekt *Nautilus* mit Typzusicherung

Mit den Begriffen statischer Typ und dynamischer Typ eines Objekts verbindet sich die Tatsache, daß in einem Programm zur Kompilationszeit jedes Objekt einem festen Typ zugeordnet ist, dem Typ seiner Klasse, dieser Typ wird der statische Typ des Objekts genannt, die Klasse heißt Basisklasse des Objekts. Anders aber als beispielsweise bei Variablen des Typs *INTEGER*, deren Typ während ihrer gesamten Lebensdauer unverändert bleibt, kann der Typ eines (Zeiger-)Objekts zur Laufzeit eines Programms geändert werden, das Objekt kann, wie oben *Nautilus*, einen dynamischen Typ erhalten, der sich von seinem statischen Typ unterscheidet.

Allerdings kann diese Typänderung nicht völlig willkürlich erfolgen. Ein Objekt kann nur einen dynamischen Typ erhalten, der eine Erweiterung seines statischen Typs ist, der dynamische Typ kann also nur der Typ einer der Unterklassen der Basisklasse des Objekts sein. Dabei ist ein weiterer wichtiger Aspekt zu berücksichtigen. Bei einer dynamischen Typänderung verliert das Objekt nicht die gesamte "Erinnerung" an seinen statischen Typ, seine durch diesen Typ vorgegebene Grundstruktur bleibt unabhängig vom jeweiligen dynamischen Typ erhalten. Für das Objekt *Nautilus* bedeutet dies, daß nach der Zuweisung *Nautilus := MolaMola* zwar beide Zeiger auf das selbe Basisobjekt weisen, weshalb man annehmen könnte, daß das in *MolaMola* vorhandene Feld *Zeitpunkt* nicht nur über den Zeiger *MolaMola*, sondern auch über den Zeiger *Nautilus* adressierbar wäre. Wegen der "Erinnerung" von *Nautilus* an seinen statischen Typ *Boot*, der dies Feld nicht enthält, ist das jedoch nicht möglich. Entfernen Sie die Kommentarklammern in der Zeile *Nautilus.Zeitpunkt := "gestern"*, erhalten Sie beim Kompilieren die dahinter stehende Fehlermeldung, der Compiler akzeptiert unabhängig vom dynamischen Typ eines Objekts nur Zugriffe auf Felder, die im statischen Typ des Objekts deklariert sind, alle übrigen Felder werden maskiert.

Die Anweisung *MolaMola.Zeitpunkt := "jetzt"* wird dagegen vom Compiler anstandslos verarbeitet, denn *Zeitpunkt* ist ein Feld des statischen Typs von *MolaMola* (s. Abb. 18a und 18 b).

Es gibt allerdings Möglichkeiten, ein Objekt für begrenzte Zeit seinen statischen Typ vollständig "vergessen" zu lassen. In der Zeile *Nautilus(BootZeit).Zeitpunkt := "heute"* stellt der hinter dem Variablenamen *Nautilus* in Klammern stehende Zusatz (*BootZeit*) eine sogenannte lokale Typzusicherung (local type guard) dar. Mit einer derartigen Typzusicherung wird das Objekt *Nautilus* für die Dauer der aktuellen Anweisung vollständig dynamisiert, es erhält durch die Typzusicherung über seinen statischen Typ *Boot* hinaus Zugriff auf alle Felder und Methoden, die im Typ *BootZeit* deklariert sind (s. Abb. 18 c). (Weitere Einzelheiten über Typzusicherungen lernen Sie im 11. Kapitel kennen).

Die in der Prozedur *Start* in Kommentarklammern stehende Zeile *MolaMola := Nautilus* stellt umgekehrt den Versuch dar, ein Objekt der Basisklasse *Boot* einem Objekt der Erweiterungsklasse *BootZeit* zuzuweisen. Bei dieser Zuweisung ist nicht klar, welches der Wert des Feldes *Zeitpunkt* der Klasse *BootZeit* sein sollte, da dies Feld in der Basisklasse *Boot* des Objekts *Nautilus* nicht vorhanden ist. Zuweisungen von Objekten einer Basisklasse an (Zeiger-)Objekte einer Erweiterungsklasse sind generell nicht durchführbar, denn die Werte der in dieser Unterklasse eventuell zusätzlich vorhandenen Felder wären stets unbestimmt. Dynamische Typänderungen eines Objekts sind nur in der Form erlaubt, daß bei der Änderung zwar gegebenenfalls einzelne Felder des Objekts maskiert werden, jedoch keine hinzukommen können.

Bei einer Zuweisung, durch die sich der Typ eines Objekts dynamisch ändert, werden durch die Zuweisung die Werte aller Felder des Objekts, auch der maskierten, nicht geändert. Dies gilt nicht nur für die Werte der Datenfelder, sondern auch für die Werte der Methoden. Folglich ist der "Wert" der Methode *Nautilus.Text* nach der Zuweisung *Nautilus := MolaMola* identisch mit dem der Methode *MolaMola.Text*, der Aufruf von *Nautilus.Text* aktiviert die zur Klasse *BootZeit* gehörende Methode.

Wie Sie beim Vergleich der von den Modulen *TutErbrec3* und *TutErbrec4* erzeugten Bildschirmausgaben feststellen werden, liefern in *TutErbrec4* wegen der Änderung des (dynamischen) Typs von *Nautilus* und der daraus folgenden Änderung des Inhalts der Methode *Nautilus.Text* die beiden in Kommentarklammern stehenden Anweisungen *MolaMola.Fahrzeugsorte := "Fahrrad"* und *Nautilus.Fahrzeugsorte := "Geisterschiff"* der Prozedur *Start* (nach Entfernen der Klammern und Neukompilation) andere Ergebnisse als die gleichlautenden Anweisungen des Moduls *TutErbrec3*; die von den Namen *Nautilus* und *MolaMola* referenzierten Objekte sind im einen Fall identisch, im anderen verschieden. Sie sollten die auf der Verwendung von Zeigern beruhenden Gemeinsamkeiten und Unterschiede beider Module genau verstanden haben, der Umgang mit Zeigerklassen und von ihnen abgeleiteten Objekten wird in den restlichen Teilen des Tutoriums die zentrale Rolle spielen.

Einige weitere Besonderheiten von Zeigern und typgebundenen Prozeduren lernen Sie in den Programmen *Methode1.odc* und *Methode2.odc* kennen. In *Methode1.odc* werden in der Ihnen aus den vorigen

Programmen bekannten Weise ein Verbundtyp *Record = EXTENSIBLE RECORD ... END* und ein assoziierter Zeigertyp *Pointer = POINTER TO Record* deklariert sowie zwei typgebundene Prozeduren, *PROCEDURE (VAR r: Record) RText, NEW, EXTENSIBLE* und *PROCEDURE (p: Pointer) PText, NEW*, wobei aus den Empfängerparametern der Signaturen hervorgeht, daß die erste eine Methode der Klasse *Record*, die zweite eine Methode der Klasse *Pointer* ist.

Die Kommandoprozedur *TutMethode1.Start* ist einfach gebaut, zu jeder Klasse wird mit *RVar: Record* und *PVar: Pointer* ein Objekt deklariert. Anschließend erhält das Feld *RVar.Name* den Wert "RVAR-" und die Methode *RVar.RText* wird aktiviert. Bei der Erläuterung des Programms *ErbRec4.odc* habe ich darauf hingewiesen, daß an Zeigerklassen gebundene Methoden nur von einem Zeigerobjekt aktiviert werden können. Der in Kommentarklammern stehende Aufruf *RVar.PText* ist daher nicht möglich, das Objekt *RVar* hat nur Zugriff auf die seiner Klasse *Record* zugeordnete Methode *RText*. Dagegen können Sie den anschließenden Zeilen entnehmen, daß das Zeigerobjekt *PVar* nicht nur Zugriff auf die der Zeigerklasse *Pointer* zugeordnete Methode *PText*, sondern auch auf die zur Klasse *Record* gehörende Methode *RText* hat. Dies ist deshalb möglich, weil der Zeiger *Pointer* durch die Typdeklaration *POINTER TO Record* eindeutig mit dem Typ *Record* verbunden ist. Umgekehrt weiß die Klasse *Record* jedoch nichts von der Existenz des Typs *Pointer*, ein Objekt der Klasse *Record* kann folglich nicht auf die zur Klasse *Pointer* gehörende Methode *PText* zugreifen. Die Konsequenzen dieser Asymmetrie zeigt Ihnen das anschließende Programm *Methode2.odc*.

Das Modul *TutMethode2* ist dem Modul *TutMethode1* in seinem Aufbau sehr ähnlich, es deklariert mit *Record1 = EXTENSIBLE RECORD (TutMethode1.Record)*, *Record2 = EXTENSIBLE RECORD (Record1)* und *Record3 = RECORD (Record2)* drei Unterklassen der Klasse *TutMethode1.Record*, weshalb das Modul *TutMethode2* zusätzlich zum Modul *Out* das Modul *TutMethode1* importieren muß. Weiter werden drei Zeigerklassen *Pointer1 = POINTER TO Record1*, *Pointer2 = POINTER TO Record2* und *Pointer3 = POINTER TO Record3* deklariert. Zeigerklassen der Art *POINTER TO <benutzerdeklariertes Verbundtyp>* sind keine selbständigen Klassendeklarationen, sie benötigen keine gesonderten Angaben von Basisklassen, diese sind durch die jeweils referenzierten Verbundtypen gegeben.

Die im Anschluss deklarierten Methoden entsprechen im Aufbau und Inhalt weitgehend denen des Moduls *TutMethode1*, wobei jede mit Ausnahme der letzten, *PROCEDURE (IN pr: Record2) PRText, NEW, EXTENSIBLE*, die gleichnamige Methode ihrer Basisklasse überschreibt. Beachten Sie bitte, daß alle Methoden mit Ausnahme von *PROCEDURE (VAR r2: Record2) RText* und *PROCEDURE (p2: Pointer2) PText* das Attribut *EXTENSIBLE* haben und deshalb überschrieben werden können, während die beiden zuletzt erwähnten *final*, also nicht überschreibbar sind.

Jede der redefinierten Methoden, die praktisch selbsterklärend sind, ruft an ihrem Ende in der Ihnen aus dem Programm *ErbRec4.odc* bekannten Weise die gleichnamige Methode der Basisklasse auf. Im Zusammenhang damit gibt es einige erwähnenswerte Details. Die in der Methode *PROCEDURE (VAR r2:*

Record2) RText zu findende Zeile *r2.RText^* macht klar, daß Basismethoden nur rekursiv aufgerufen werden können, unmittelbare Zugriffe sind nur auf die Methode der direkten Basisklasse möglich, jedoch nicht auf überschriebene Methoden der in der Klassenhierarchie tiefer liegenden Klassen. Ebenso ist ein derartiger *super-call* nur von der überschreibenden Methode aus erlaubt, nicht aber von anderen Programmteilen aus, wie die in Kommentarklammern in der Prozedur *Start* zu findende Zeile *RVar.RText^* und die dahinter stehende Fehlermeldung zeigen.

Die Anweisungen der Prozedur *Start* sollten Ihnen keine Verständnisschwierigkeiten bereiten, es werden wie im Modul *TutMethode1* Objekte aller Klassen deklariert und danach die zu den Objektklassen gehörenden Methoden aktiviert. Zwei Details möchte ich dabei herausheben. Der im Anschluss an die Zeilen *P1Var := P2Var* und *P1Var.PText* stehende Kommentar weist Sie auf die im Zusammenhang mit dem Programm *ErbRec4.odc* erläuterte Tatsache hin, daß Objekte einer Zeigerklasse andere Eigenschaften als Objekte einer Verbundklasse haben. Die nur bei Zeigerobjekten mögliche Zuweisung *P1Var := P2Var* ändert den Wert des Zeigers *P1Var*, dieser zeigt anschließend auf das Objekt *P2Var^* mit dem Ergebnis, daß nach der Zuweisung in allen Programmschritten, die das Objekt *P1Var* enthalten, die aktuellen Werte der Felder des Objekts *P2Var* verwendet werden. Dies betrifft nicht nur die Datenfelder, sondern auch die Prozeduren, wobei selbstverständlich zu berücksichtigen ist, daß in *P2Var* neu deklarierte (Daten- und Prozedur-)Felder maskiert, d. h. für *P1Var* unsichtbar wären, was allerdings im hier vorliegenden Fall keine Auswirkung hat, da es zusätzliche Felder nicht gibt, sowohl das Datenfeld *Name* als auch die Zeigermethode *PText* und die Verbundmethode *RText* sind Bestandteile der beiden statischen Typen *Pointer1* bzw. *Pointer2*. Das Objekt *P1Var* hat nach der Zuweisung *P1Var := P2Var* den dynamischen Typ *Pointer2* und der Aufruf *P1Var.PText* aktiviert die zu dem Typ gehörende Methode *PROCEDURE (p2: Pointer2) PText*, ebenso aktiviert der anschließende Aufruf *P1Var.RText* die Methode *PROCEDURE (VAR r2: Record2) RText*.

Die Kommentare am Ende der Prozedur *Start* machen Sie darauf aufmerksam, daß jede Klasse die Methoden ihrer Basisklasse unverändert erbt, sofern diese in der Klassendeklaration nicht überschrieben werden. Mit Ausnahme der jeweiligen Objektamen liefern die Aufrufe *R3Var.RText* und *P3Var.PText* also die gleichen Ergebnisse wie die Aufrufe *R2Var.RText* und *P2Var.PText*.

Eine interessante Programmiermöglichkeit stellt Ihnen die letzte Methode des Moduls - *PROCEDURE (IN pr: Record2) PRText* - vor. Diese Methode wird nur von *P3Var*, einem Objekt des Typs *Pointer3* benutzt und wäre deshalb anscheinend einfacher als *PROCEDURE (pr: Pointer2) PRText* zu deklarieren gewesen. Allerdings dürften in diesem Fall innerhalb der Methode die Werte der Felder des aktuellen Empfängerparameters geändert werden, da der Empfängerparameter einer Zeigermethode ein *VAL*-Parameter sein muß. Sofern Sie diese Möglichkeit ausschließen wollen, können Sie den hier gezeigten "Umweg" über einen schreibgeschützten *IN*-Empfängerparameter benutzen, auch wenn die Methode nur von einer Zeigervariablen benutzt werden soll. Ein derartiger Schreibschutz kann sich als wichtig erweisen bei

exportierten Methoden, die in einer externen Unterklasse zwar wie *PRTtext* überschreibbar, deren Empfängerparameter aber gegen Veränderungen gesichert sein soll.

Die wesentlichen Aspekte der hier erläuterten Einzelheiten des Moduls sind in der Bildschirmausgabe der Prozedur *Start* zu finden, Sie sollten diese unbedingt vor der Ausführung der Prozedur selbst erarbeiten und Ihr Ergebnis anschließend mit dem Ergebnis des Rechners vergleichen, wobei Sie insbesondere auf die Abfolge der Aufrufe überschriebener Methoden achten sollten und auf die Unterschiede, die sich aus den unterschiedlichen Eigenschaften von Verbund- und Zeigerobjekten ergeben.

10.4. Stapel - Das FILO-Prinzip

Mit den Zeigern haben Sie den letzten der in Component Pascal vordefinierten Datentypen kennengelernt. Zeiger sind sehr flexible Programmierkonzepte, mit denen sich auf einfache Weise vielseitig verwendbare Datenstrukturen wie Stapel, Schlangen, geordnete Listen und Bäume erstellen lassen, die Sie in den weiteren Abschnitten dieses und der folgenden Kapitel kennenlernen werden.

Der ersten dieser Datenstrukturen, dem Stapel, begegnen Sie mit dem Programm *Stapel.odc*. Der Begriff des Stapels ist Ihnen bereits bei der Erläuterung der Speicherorganisation eines Computers begegnet, es handelt sich um ein Gebilde, bei dem jedes hinzukommende Objekt - wie neue Blätter bei einem Papierstapel - auf den bereits vorhandenen angeordnet wird. Umgekehrt läßt sich lediglich das oberste Objekt vom Stapel entfernen, das zuerst eingeordnete Objekt kann nur als letztes wieder entnommen werden. Aus diesem Grund werden Stapel mit den entsprechenden englischen Namen als FILO(First-In-Last-Out)-Listen oder auch LIFO(Last-In-First-Out)-Listen bezeichnet.

Anders als bei einem realen Stapel müssen die einzelnen Objekte im Arbeitsspeicher eines Computers nicht physikalisch, sondern lediglich logisch "aufeinander" angeordnet sein. Eine solche logische Anordnung bestimmt sich durch die Tatsache, daß in einem Stapel jedes Objekt - bis auf das unterste - stets ein Nachfolgeobjekt hat. Sie können sich diese Verknüpfung vorstellen wie eine Telefonkette, bei der jeder angerufene Teilnehmer lediglich die Nummer eines weiteren Teilnehmers kennt, den er anrufen soll. Es kommt also nicht darauf an, wo sich das einzelne Objekt befindet, sondern ausschließlich darauf, wodurch es von seinem Nachfolgeobjekt "weiß". Dieses "Wissen" - die Telefonnummer - besteht beim Arbeitsspeicher in der Kenntnis der Adresse des Nachfolgers und es läßt sich leicht mit Hilfe von Zeigern realisieren, da Zeigervariablen nur dazu da sind, Speicheradressen aufzunehmen.

Jedes Objekt einer logisch verketteten Liste muß also mindestens aus einem Zeiger bestehen, der auf ein weiteres Objekt - den Nachfolger - verweist, darüber hinaus wird ein derartiges Objekt irgendwelche "Nutzinhalte" aufnehmen sollen. Im Allgemeinen werden diese etwas anderes als Zeiger sein und deshalb andere Datentypen haben, daher verwendet man den Typ *RECORD* als Grunddatentyp für Listenstruk-

turen. Wegen der Tatsache, daß jedes Objekt seinen Nachfolger kennen muß, enthält der Verbund - neben anderen Feldern - ein Feld, das auf ein Objekt seines eigenen Typs verweist. Es handelt sich um eine rekursive Typdeklaration, ähnlich wie sie im Zusammenhang mit dem Programm *Verbund.odc* erwähnt wurde. Bei der Erläuterung dieses Programms habe ich Sie darauf aufmerksam gemacht, daß rekursive Typdeklarationen bei zusammengesetzten Datentypen syntaktisch zwar möglich, aber - mit einer Ausnahme - aus verständlichen Gründen nicht zugelassen seien. Die Ausnahme beruht darauf, daß man bei einem rekursiv deklarierten Verbund die Rekursion nicht statisch, sondern dynamisch durchführt. Während bei der statischen Rekursion jeder Verbund eine vollständige Ausgabe seiner selbst enthalten müßte, benötigt man bei der dynamischen Rekursion lediglich einen Zeiger, der auf "sich selbst" - genauer gesagt auf ein Objekt seines eigenen Grundtyps - verweist.

Die Deklaration eines in dieser Weise aufgebauten Verbundes finden Sie in dem Programm *Stapel.odc*. Sie besteht aus zwei Teilen, die Ihnen bereits bekannt sind, der Deklaration des Verbundes *StapelDesc* und der Deklaration eines Zeigers *Stapel*, der auf diesen Verbund verweist. Einerseits zeigt also der Zeiger *Stapel* auf den Verbund *StapelDesc*, andererseits enthält *StapelDesc* ein Feld *Folgezeiger*, dessen Typ *Stapel* ist. Es handelt sich um ein ähnliches Problem, wie es im Programm *Wechsel.odc* bei der FORWARD-Deklaration von Prozeduren auftrat, bei der ebenfalls zwei Strukturen wechselseitig verknüpft sind, eine der beiden muß unvermeidlich auf die ihr folgende Bezug nehmen. Auch in diesem Fall gilt die im Zusammenhang mit dem Modul *TutErbRec4* dargestellte Regel, daß benutzerdefinierte Typen innerhalb des selben Gültigkeitsbereichs als bekannt gelten, Sie können also bei einer rekursiven Typdeklaration einen Zeigertyp, der auf einen Verbundtyp verweist, in dem er selbst verwendet wird, wahlweise vor oder hinter diesem Typ deklarieren.

Für den externen Zugriff auf den Stapel wird eine Zeigervariable, oft *Wurzel* (engl. root) genannt, vom Typ des Stapelzeigers benötigt. Nach außen repräsentiert der Zeiger *Wurzel* den Stapel, er stellt die einzige globale Zugriffsmöglichkeit dar. Ein neu deklariertes Zeiger sollte, wie jede neu deklarierte Variable, als erstes einen Standardwert zugewiesen bekommen, bei Zeigern wäre das der Wert *NIL*. Dies geschieht - im Prinzip - im Modul *TutStapel* als Initialisierung unmittelbar beim Laden des Moduls. Allerdings steht die entsprechende Zuweisung in Kommentarklammern, die beigefügte Erläuterung weist darauf hin, daß in *BlackBox* Zeigervariablen aus Sicherheitsgründen stets mit *NIL* initialisiert sind.

Bei der Implementierung eines Stapels muß man sich überlegen, welche Aktionen mit ihm durchgeführt werden sollen. Die beiden wesentlichen sind das Hinzufügen eines neuen Objekts und das Entfernen des jeweils obersten der vorhandenen Objekte. Im Modul *TutStapel* sind dafür die Prozedur *Stapeln* und die Funktion *EntfernterEintrag* zuständig. Mit der Prozedur *Stapeln* wird ein neues Objekt auf dem Stapel abgelegt. Dieser Prozedur wird vom Benutzer - der Prozedur *Einlesen* - kein ganzes Stapelobjekt, sondern nur der Nutzinhalte in Form des zuvor eingelesenen Wertes der *INTEGER*-Variablen *Zahl* übergeben. Die

Prozedur *Stapeln* "weiß" selbst, wie sie daraus ein vollständiges Objekt macht und dies auf dem Stapel ablegt.

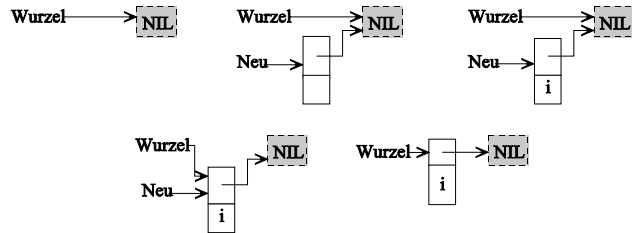


Abbildung 19 a
Zeigerverwaltung in der Prozedur *TutStapel.Stapeln*
Leere Liste

Zur Erzeugung des neuen Stapelobjekts wird in *Stapeln* eine Zeigervariable *Neu: Stapel* deklariert und für das Objekt mit *NEW(Neu)* die (anonyme) Verbundvariable *Neu^* erzeugt. Im ersten Schritt erhält deren Feld *Folgezeiger* den Wert des globalen Stapelzeigers *Wurzel* zugewiesen, der *Folgezeiger* zeigt damit auf das bisherige oberste Objekt des Stapels. Bei einer leeren Liste enthält *Wurzel* den Wert *NIL*, *Neu.Folgezeiger* enthält folglich ebenfalls den Wert *NIL*. Im zweiten Schritt wird der Wert des Parameters *i* (die von *Einlesen* übergebene Zahl) in das Feld *Element* von *Neu^* kopiert, das Objekt ist damit vollständig (Sie finden die Aktionen in den Abbildungen 19 a und 19 b dargestellt, jeder Anweisung der Prozedur entspricht ein Teilbild). Da das Objekt *Neu^* jetzt das oberste im Stapel sein soll, muß im nächsten Schritt der Zeiger *Wurzel* "umgebogen" werden, er zeigt anschließend nicht mehr auf das bisherige oberste Objekt, sondern auf *Neu^*.

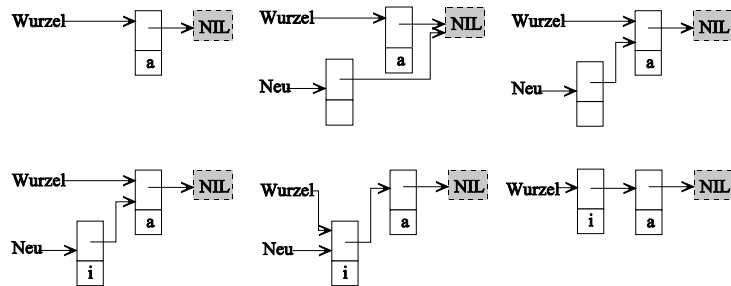


Abbildung 19 b
Zeigerverwaltung in der Prozedur *TutStapel.Stapeln*
Andere Listen

Beachten Sie, daß *Neu.Folgezeiger* sowohl bei einem leeren (s. Abb. 19 a) als auch bei einem bereits existierenden Stapel (s. Abb. 19 b) der Wert des Zeigers *Wurzel* zugewiesen wird, auf diese Weise ist die Prozedur *Stapeln* in beiden Fällen einsetzbar, der Zeiger *Wurzel* zeigt am Ende der Prozedur jeweils auf das zuletzt eingefügte Objekt. Beachten Sie bitte weiter, daß jeder Zeiger *Neu*, der in *Stapeln* benutzt wird, nur dort und nur während der Laufzeit der Prozedur existent ist, außerhalb ist der gesamte Stapel nur über den Zeiger *Wurzel* zugänglich. Und denken Sie bitte auch an die erwähnte Tatsache, daß bei einem Stapel über den auf ihn weisenden globalen Zeiger *Wurzel* immer nur auf das zuletzt eingefügte Objekt zugegriffen werden kann, vorausgesetzt, der Stapel ist nicht leer. In diesem Fall enthält *Wurzel* den Wert *NIL*, eine Situation, die bei Zeigeroperationen stets gesondert zu berücksichtigen ist.

Die zweite wesentliche Operation, das Abnehmen eines Objekts vom Stapel wird von der Funktion *EntfernterEintrag* ausgeführt, die an die aufrufende Stelle den Objekthalt zurückgibt. Sie werden in den folgenden Programmen sehen, daß sich in Hinblick auf eine mögliche Weiterverwendung nicht nur des Objekthalts, sondern des gesamten Objekts die Implementierung des Entfernens als Funktion anbietet obwohl sie nicht zwingend ist, das Entfernen eines Objekts läßt sich auch mit einer Prozedur verwirklichen, ein Beispiel dafür finden Sie im Programm *Schlangen1.odc* (Programm 10.5 a).

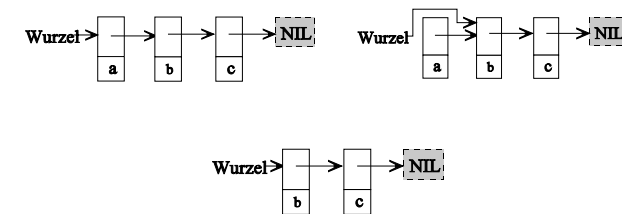


Abbildung 19 c
Zeigerverwaltung in der Prozedur *TutStapel.EntfernterEintrag*
Alle Listen

Die für das Entfernen der Stapelobjekte zuständige Funktion *EntfernterEintrag* arbeitet mit der lokalen *INTEGER*-Variablen *Eintrag*. Diese nimmt den Inhalt des zu entfernenden Objekts auf und stellt ihn am Ende der Prozedur durch *RETURN Eintrag* für die weitere Verarbeitung zur Verfügung. Als erstes wird die Variable *Eintrag* im Anweisungsteil mit einem festen Wert initialisiert, was stets geschehen sollte, da bei lokalen Variablen eine Initialisierung nicht selbsttätig vorgenommen wird. Als nächstes erfolgt die bei fast allen Zeigeroperationen notwendige Prüfung, ob der Zeiger *Wurzel* einen gültigen Inhalt *Wurzel^* hat, also einen anderen Wert als *NIL* enthält. Wenn dies der Fall ist, wird der Objekthalt *Wurzel.Element* der Variablen *Eintrag* zugewiesen und danach der Zeiger *Wurzel* auf das nachfolgende Objekt weitergerückt (s. Abb. 19 c), der Stapel enthält jetzt nicht mehr das bisherige oberste Objekt. Im letzten Schritt wird der

in *Eintrag* gespeicherte Objektkinhalt der aufrufenden Prozedur *Ausgeben* mittels *RETURN* zur Verfügung gestellt. Beachten Sie bitte, daß von der Prozedur *Ausgeben* die Zahlen tatsächlich, der Bezeichnung *Stapel* entsprechend, in umgekehrter Reihenfolge zum Einlesevorgang angezeigt werden.

10.5. Schlangen - Das FIFO-Prinzip

Die im vorigen Abschnitt dargestellte Stapelverarbeitung ist eine der Möglichkeiten, Aneinanderreihungen von Objekten im Arbeitsspeicher eines Computers zu verwalten, eine weitere Form solcher allgemein "lineare Listen" genannten Strukturen sind die sogenannten Schlangen.

Unter einer Schlange kann man sich wahrscheinlich am einfachsten etwas vorstellen, wenn man an eine Warteschlange vor dem Schalter einer Bank denkt. Bei solchen Warteschlangen wollen und sollen im Gegensatz zu einem Stapel diejenigen Kunden zuerst bedient werden, die am längsten gewartet haben, anders gesagt: Wer zuerst kommt, darf als Erster gehen, Schlangen erhalten deshalb auch den Namen FIFO (First-In-First-Out)-Listen oder auch LILO (Last-In-Last-Out)-Listen. Die Struktur einer Schlange ähnelt der eines Stapels, es handelt sich ebenfalls um eine verkettete Liste, bei der jedes Objekt neben anderen Daten eine Referenz (einen Zeiger) auf ein Objekt seines eigenen Typs enthält, die auf ein nachfolgendes Objekt verweist oder aber den Wert *NIL* hat.

Sie finden im Programm *Schlangen.odc* die Deklaration eines Typs *Schlange*, der dem Typ *Stapel* des vorangegangenen Programms prinzipiell gleicht, er unterscheidet sich von diesem Typ lediglich durch die Tatsache, daß die Typdeklaration - anders als in den bisherigen Programmen - nicht mehr aus einem Zeiger und einen gesonderten Verbund besteht, der Zeigerbasistyp ist in diesem Fall anonym. Auch die Art der "Nutzlast" hat sich geändert, sie besteht aus zwei Zeichenketten, *Vorname* und *Nachname*.

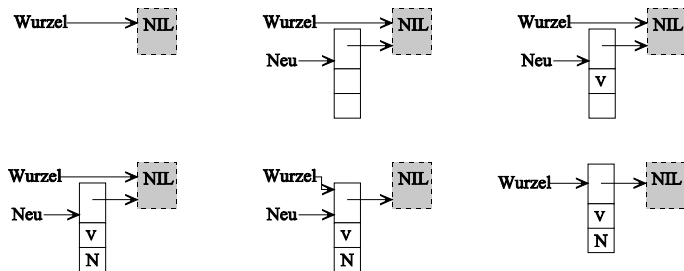


Abbildung 20 a
Zeigerverwaltung in der Prozedur *TutSchlangen.Anhängen*
Leere Liste

Die Prozedur *Anhängen* interpretiert deren Werte jeweils als ihre formalen Parameter *Vorname* und *Nachname*, die als VAL-Parameter deklariert sind. Wie Sie vielleicht noch vom Programm *String.odc* (Programm 7.5) erinnern, ist dies nötig, um aus den übergebenen konstanten Parameterwerten die von der Prozedur benötigten Variablenwerte zu erzeugen. Im übrigen ist *Anhängen* der Prozedur *Stapeln* aus dem Programm *Stapel.odc* ähnlich, die Prozedur *Anhängen* erzeugt ebenfalls aus den Parameterwerten (den "Rohdaten") ein vollständiges Listenobjekt *Neu*[^], dessen einzelne Felder die aktuellen Parameterwerte zugewiesen bekommen.

Für das Einfügen eines neuen Objekts in die Liste müssen bei der Schlange wie beim Stapel zwei Situationen unterschieden werden, die leere Liste ist anders zu behandeln als eine Liste, die bereits Objekte enthält. Da anders als bei einem Stapel bei der Schlange das neue Objekt an das bisherige letzte gehängt werden soll, der globale Zeiger *Wurzel* jedoch bei der Schlange ebenso wie beim Stapel stets auf das vordeste Objekt zeigt, wird für das Anhängen ein anderer Algorithmus als beim Stapel benötigt.

Im Fall einer leeren Liste (*Wurzel = NIL*) erhält der Zeiger *Wurzel* nach Eintragung der Daten in *Neu*[^] den Wert des Zeigers *Neu* zugewiesen, damit ist das Anhängen erledigt (s. Abb. 20 a). Gibt es dagegen bereits Objekte in der Schlange, ist das Anhängen komplizierter, es wird ein lokaler Zeiger *Local* benötigt. Dieser wird zu Beginn auf den Anfang der Schlange, das einzige zugängliche Objekt *Wurzel* gesetzt (eigentlich *Wurzel*[^], zur Vereinfachung werden im Folgenden wie bereits in einigen vorangegangenen Fällen die Zeiger mit den Objekten identifiziert, auf die sie verweisen, sofern keine Verwechslungen zu erwarten sind). Eine *WHILE*-Schleife rückt dann den Zeiger *Local* solange um je ein Objekt weiter, bis er auf das letzte Objekt (erkennlich an der Bedingung *Local.Folgezeiger = NIL*) zeigt. Dessen Folgezeiger wird im Anschluss umgebogen auf das bisher isolierte Objekt *Neu*, das damit an der richtigen Stelle in die Liste gefügt worden ist (s. Abb. 20 b).

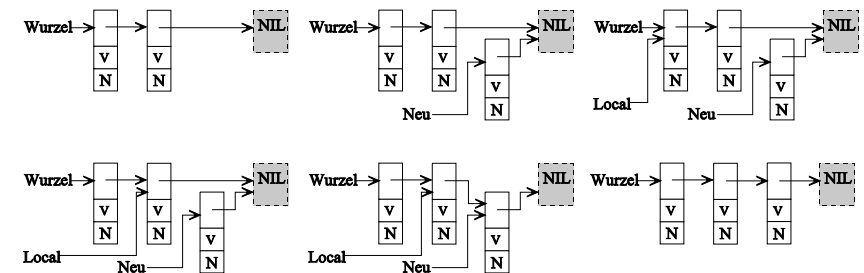


Abbildung 20 b
Zeigerverwaltung in der Prozedur *TutSchlangen.Anhängen*
Andere Listen

Das Entfernen eines Objekts aus der Schlange vollzieht sich wie bei dem Stapel, das erste Objekt wird bei beiden in prinzipiell identischer Weise aus der Liste entfernt (auf eine gesonderte Abbildung habe ich deshalb verzichtet). Die Funktion *EntfernterEintrag* unterscheidet sich von der gleichnamigen Funktion des vorigen Programms dennoch in einigen Einzelheiten. Es wird nicht der Inhalt des entfernten Objekts in Form mehrerer Einzelwerte - *Vorname, Nachname* - an die aufrufende Stelle zurückgegeben, sondern (ein Zeiger auf) das vollständige entfernte Objekt vom Typ *Schlange*. Dies macht sich bemerkbar an den Deklarationen der Funktion, es gibt außer dem Zeiger *Kopf* keine sonstigen lokalen Variablen. Als Folge ist eine wichtige Änderung zu beachten; da das entfernte Objekt vollständig zurückgegeben wird, muß dafür gesorgt werden, daß bei der Weiterverwendung dieses Objekts kein Einfluß auf die übrige Listenstruktur genommen werden kann, deshalb wird *Kopf.Folgezeiger* der Wert *NIL* zugewiesen, das Objekt wird auf diese Weise vollständig von der Liste getrennt. Machen Sie sich bitte klar, daß diese Zuweisung von *NIL*, wie im Kommentartext der Funktion dargelegt, nur in der gezeigten Reihenfolge stattfinden kann, nachdem also die Zeiger *Wurzel* und *Kopf* auf verschiedene Objekte zeigen, da anderenfalls die über *Wurzel* zugängliche Restliste lediglich noch das vorderste Objekt, den Kopf der Schlange, enthielte.

Während die Prozedur *Einlesen* keine Neuigkeiten bietet, gibt es in der Prozedur *Ausgeben* einen beachtenswerten Aspekt. Man könnte versucht sein, auf die lokale Variable *Kopf* zu verzichten, dies ist jedoch aus folgenden Gründen nicht möglich. Von jedem zurückgegebenen Objekt sollen zwei Einzelwerte, *Vorname* und *Nachname*, angezeigt werden, deshalb lassen sich direkte Ausgabeanweisungen, wie sie in den Kommentarklammern stehen, nicht verwenden, denn jeder Aufruf der Funktion *EntfernterEintrag* gibt ein neues Objekt aus der Liste zurück, die als Kommentar zu findenden Zeilen *Out.String(EntfernterEintrag().Vorname)* und *Out.String(EntfernterEintrag().Nachname)* zeigten daher nicht Vorname und Nachname des selben, sondern zweier verschiedener, aufeinander folgender Objekte an. Unabhängig davon ist es grundsätzlich "guter Programmierstil", weiterzuverarbeitende Variablen, wie hier *Kopf*, zur Vermeidung unerwarteter Nebeneffekte lokal zu machen, sofern nicht eindeutige Gründe entgegenstehen.

Die Programme *Stapel.odc* und *Schlangen.odc* haben Ihnen mit dem Stapel und der Schlange die beiden grundlegenden Formen der Organisation einer linearen Liste vorgestellt. Beide gehen, bildlich gesprochen, davon aus, daß die Liste aus einer Aneinanderreihung von Objekten besteht, von denen jedes aus einem Objektinhalt und einem Verweis auf das Folgeobjekt zusammengesetzt ist. Lineare Listen sind also in dieser Sichtweise rekursive Datenstrukturen, sie bestehen stets aus einem "Kopf", dem ersten Objekt, das über den globalen Listenzeiger *Wurzel* zugänglich ist und einer nachfolgenden Restliste, die wiederum aus einem Kopf und einer Restliste besteht. Dabei ist es unerheblich, daß die Restliste möglicherweise leer ist, an der grundsätzlichen Struktur der Liste ändert dies nichts.

Eine andere Auffassung von linearen Listen zeigen Ihnen die Typdeklarationen des Programms *Schlangen1.odc*. Die in diesem Programm vorgestellte Idee der linearen Liste - hier wieder als Schlange implementiert - geht ebenfalls aus von Objekten (des Typs *Element*), die aus einem Inhalt und einem Ver-

weis auf ein Nachfolgeobjekt bestehen. Anders als bei den beiden vorigen Programmen ist die Schlange selbst jedoch nur eine Art "Gehäuse", das keinen Nutzinhalt hat, vielmehr lediglich aus den beiden Zeigern *ErstesElement* und *LetztesElement* besteht, die auf das erste und das letzte Objekt der Liste verweisen.

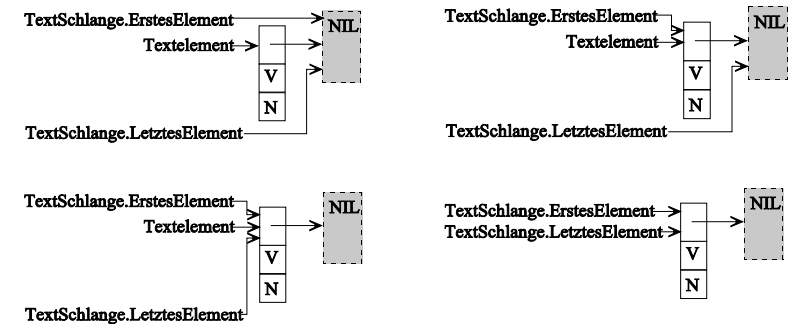


Abbildung 21 a
Zeigerverwaltung in der Prozedur *TutSchlangen1.Anhängen*
Leere Liste

Das scheint auf den ersten Blick eine Komplikation darzustellen, Sie werden bei der Analyse des Algorithmus' der Prozedur *Anhängen* jedoch merken, daß mit dieser Aufteilung eine spürbare Vereinfachung und damit Beschleunigung des Einfügens neuer Objekte in die Liste erreicht werden kann (s. Abb. 21 a und 21 b). Außerdem ergibt sich als nicht unerheblicher Nebeneffekt durch die Trennung eine klarere Programmgliederung, deren Wert für die Lesbarkeit und Wartungsfähigkeit umfangreicher Programmpakete Sie nicht unterschätzen sollten.

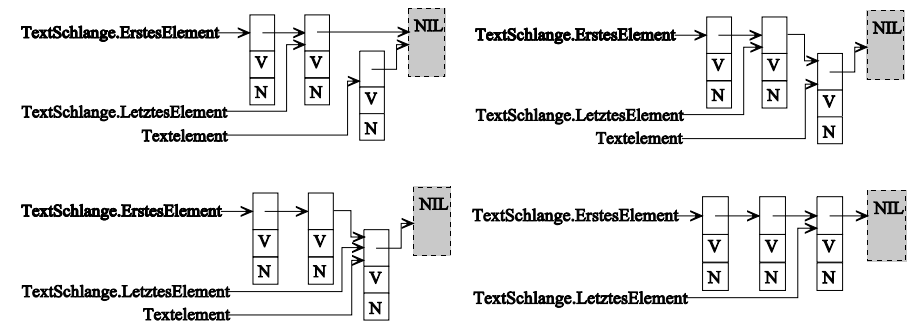


Abbildung 21 b
Zeigerverwaltung in der Prozedur *TutSchlangen1.Anhängen*
Andere Listen

Bei den Details von *Schlangen1.odc* werde ich mich auf einige wesentliche Punkte beschränken, der Aufbau des Programms und der Prozeduren sollte Ihnen keine Schwierigkeiten bereiten. Zum Einen sehen Sie, daß das Entfernen eines Objekts aus der Schlange hier nicht als Funktion ausgeführt ist, sondern als Prozedur mit einem für Sie neuen Parameter Typ, einem OUT-Parameter. OUT-Parameter sind ähnlich wie IN-Parameter spezielle VAR-Parameter, die von der aufrufenden Umgebung als Referenzen an eine Prozedur übergeben werden. Der Unterschied zwischen den Parameterarten besteht darin, daß der Wert eines VAR-Parameters zu Beginn und Ende der Prozedur bestimmt ist und Änderungen des Parameterwerts innerhalb der Prozedur anschließend im aktuellen Parameter zu finden sind (read-write parameter); die Werte von IN-Parametern sind bei Eintritt in die Prozedur ebenfalls bestimmt, können aber in der Prozedur nicht geändert werden (read-only parameter); dagegen sind die Werte von OUT-Parametern bei Eintritt in die Prozedur unbestimmt, sie können (und sollten) in der Prozedur geändert werden, ein in der Prozedur geschriebener Wert eines OUT-Parameters ist wie bei einem VAR-Parameter anschließend im aktuellen Parameter zu finden (write-only parameter).

Man kann Prozeduren mit OUT-Parametern als verkappte Funktionen ansehen, es handelt sich bei ihnen um einen beliebigen Programmiertrick, der oft eingesetzt wird, falls aus irgendeinem Grund eine richtige Funktion nicht verwendet werden kann oder nicht ausreicht, denn mit der Verwendung von OUT-Parametern in Prozeduren und Funktionen können Sie an die aufrufende Stelle nicht nur einen, sondern mehrere (im Prinzip unendlich viele) Werte zurückgeben.

Das geschilderte Programmierverfahren mit einem OUT-Parameter finden Sie außer in der Prozedur *Entfernen* ein zweites Mal in der Prozedur *NeueSchlange*. Man könnte versucht sein, diese Prozedur für überflüssig zu halten und im Initialisierungsteil des Moduls statt des Aufrufs *NeueSchlange(Textschlange)* einfach die direkten Anweisungen *NEW(Textschlange)*, *Textschlange.ErstesElement := NIL*, *Textschlange.LetztesElement := NIL* und *Textschlange.Elementzahl := 0* verwenden. Ich möchte Sie jedoch vor derartigen "Vereinfachungen" warnen. Die Prozedur mag Ihnen in diesem kurzen Programm übertrieben erscheinen, Sie sollten aber nicht vergessen, daß auch anfänglich kurze und übersichtliche Programme sehr häufig wachsen und Sie werden nur dann vor unliebsamen Überraschungen sicher sein können, wenn Sie für die Initialisierung der Felder einer mit *NEW* erzeugten globalen Variablen wie *Textschlange* grundsätzlich eine entsprechende Initialisierungsprozedur ähnlich der Prozedur *NeueSchlange* vorsehen.

Vergleichen Sie die analogen Prozeduren *Anhängen* der Module *TutSchlangen* und *TutSchlangen1* bzw. die zugehörigen Abbildungen (s. Abb. 20 und 21) im Einzelnen, werden Sie erkennen, daß sich in *TutSchlangen1* das Einfügen neuer Objekte in die Liste durch deren Aufteilung in Elemente (Typ *Element*) einerseits und Behälter (Typ *Schlange* mit den Feldern *ErstesElement* und *LetztesElement*) andererseits wegen des direkten Zugangs zum Einfügepunkt *LetztesElement* wesentlich beschleunigt. Darüber hinaus ergibt sich eine Vereinfachung der Listenverwaltung dadurch, daß zur Schlange nach ihrer erstmaligen Er-

zeugung stets nur Elemente hinzugefügt oder aus ihr entfernt werden, wodurch es zwar geschehen kann, daß die Schlange leer ist, sie selbst aber niemals (unabsichtlich) gelöscht wird.

Abschließend möchte ich Sie auf einen wesentlichen Punkt aufmerksam machen. Guter Programmierstil erfordert es, keine globalen Variablen zu verwenden, um das Risiko unerwünschter Nebenwirkungen klein zu halten. Im Modul *Schlangen1* finden Sie eine Abweichung von diesem Prinzip, das Modul deklariert eine globale Variable, *Textschlange*. Diese Variable stellt eine notwendige Ausnahme dar, sie muß global sein, denn einerseits sollen mehrere verschiedene Prozeduren unabhängig von einander auf sie zugreifen können, andererseits soll die Variable während der gesamten Laufzeit des Moduls den selben, eindeutig identifizierbaren Inhalt repräsentieren. Die gleiche Feststellung gilt übrigens für die in den beiden vorangegangenen Programmen *Stapel.odc* und *Schlangen.odc* deklarierten Zeiger *Wurzel*, die aus den selben Gründen global sein müssen.

Globale Variablen haben außer der Tatsache, daß sie innerhalb eines Moduls dauerhaft existieren und uneingeschränkt zugreifbar sind, eine zweite, darüber hinausgehende Bedeutung. In den folgenden Kapiteln lernen Sie mit der Aufteilung eines Programms auf mehrere, in sich abgeschlossene, aber miteinander verknüpfte Module eine wesentliche Erweiterung und Ergänzung des Prinzips der Datenkapselung kennen, die Modularisierung. Dem Prinzip der Datenkapselung sind Sie bereits früher bei Feldern und Verbunden begegnet, in denen Einzelvariablen unter einer gemeinsamen Bezeichnung zusammengefasst sind. Module sind eine weitere und darüber hinausgehende Möglichkeit der Datenkapselung, in ihnen lassen sich zusammengehörende Daten und Prozeduren unter einem gemeinsamen Bezeichner, dem Modulnamen, zusammenfassen.

Neben der Datenkapselung haben Module ein zweites wesentliches Merkmal, die Datenabstraktion. Im Verlauf des Tutoriums habe ich Sie zunehmend mit dem Gebrauch von Modulschnittstellen bekannt gemacht, Modulschnittstellen bilden die Verbindung zwischen dem Modulinneren, seiner Implementierung, und den Modulbenutzern, seinen Klienten. Schnittstellen zeigen die öffentlich verfügbaren Dienste eines Moduls, sie abstrahieren von den Einzelheiten der Implementierung.

Im Rahmen der Modularisierung spielen globale Variablen eine wesentliche Rolle, denn ähnlich, wie verschiedene Prozeduren innerhalb eines Moduls auf eine Variable nur direkt zugreifen können, wenn diese global ist, können externe Klienten eines Moduls auf dessen Daten (variablen) nur direkt zugreifen, wenn die Variablen exportiert werden. Diese müssen daher, wie alle Datenstrukturen, die exportiert werden sollen, innerhalb des exportierenden Moduls global sein.

Allerdings steht ein ungehinderter externer Zugang zu den Daten eines Moduls im Widerspruch zu den Prinzipien der Datenkapselung. Externe Klienten sollen auf Daten nur in kontrollierbarer Weise zugreifen dürfen, man exportiert deshalb im Allgemeinen nicht Datenvariablen und Datenfelder, sondern sieht für den Zugriff auf sie Prozeduren vor, dadurch lassen sich unerwünschte Veränderungen der Daten verhin-

dern. Wie das im Einzelnen geschieht und unter welchen Umständen auch Datenvariablen und Datenfelder ohne Risiko exportierbar sind, erfahren Sie in den Programmen der nächsten beiden Kapitel.