19 Dynamische Datenstrukturen, Zeiger
Eine Einführung

Gliederung

19.1 Zur Speicherbelegung eines Turbo-Pascal-Programms 2

19.2 Eigenschaften der dynamischen Variablen. Routinen.....3

19.3 Demo-Programm 1: Eigenschaften dynamischer Variablen 7

19.4 Demo-Programm 2: Anwendung bei großen Arrays.....9

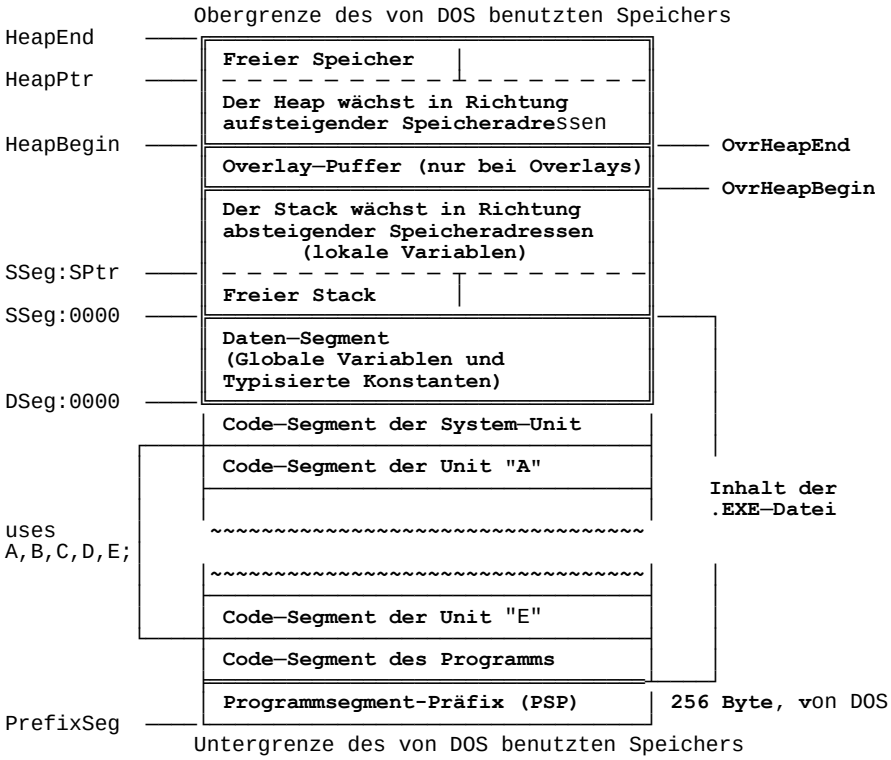
19.5 Demo-Programm 3: Anwendung bei verketteten Listen.....10

19.6 Demo-Programm 4: Einfügen in verkettete Liste.....13

19.1 Zur Speicherbelegung eines Turbo-Pascal-Programms

Bei allen bisherigen Beispielen waren die Variablen "statisch", d.h. sie hatten einen festen und reservierten Speicherplatz während des gesamten Programmlaufs. Variablen, die z.B. nur einmal zu Beginn des Programmlaufs gebraucht wurden, haben somit genauso Speicherplatz blockiert wie die Variablen, die im gesamten Programm benutzt wurden. Es war nicht möglich, den Speicherplatz der nicht mehr benötigten Variablen freizugeben und zu einem späteren Zeitpunkt neue Variablen einzuführen. Der gesamte Speicherplatz der statischen Variablen beträgt in Turbo-Pascal max. ca. 64 KByte. Die Beschränkung ist aber durch die in PCs verwendeten Intel-Mikroprozessoren und dem darauf abgestimmten Betriebssystem MS-DOS bestimmt (Datensegment max. 64 KByte).

Die folgende Graphik zeigt die Speicherbelegung eines Turbo-Pascal-Programms in schematischer Form (nach Borland-Handbuch, geringfügig modifiziert):



Das folgende Programm demonstriert die Beschränkung des Speicherplatzes für statische Variablen.

```
program Pas19011;           { Beschränkung des Speicherbedarfs aller }
        { globalen statischen Variablen auf max. ca. 64 KByte.       }
        { Technisch bedingt durch Intel-Mikroprozessor bzw. DOS.     }
        { Die lokalen Variablen der Routinen werden auf den Stack    }
        { gelegt. Die Stackgröße kann mit dem Compilerbefehl $M oder }
        { mit dem Menüpunkt Options/Memory_sizes auf Werte zwischen  }
```

```
        { 1024 Byte und 65520 Byte eingestellt werden. Die Standard- }
        { einstellung beträgt 16384 Byte. Beim Verlassen der Routine }
        { wird der Stackspeicher wieder frei.                        }
uses
  CRT;

var
  s:    array[1..253] of string;   { 253 * 256 = 64 768 }
  x, y: Real;                      {   2 *   6 =     12 }
  b:    array[1..42]  of Byte;     {  42 *   1 =     42 }
                                   { Summe:    = 64 806 bei TP 7.0 }
                                   { Summe:    = 64 822 bei TP-6.0 }
  (*  oder nur:                    { Summe:    = 64 836 bei TP-5.0 }
  b:  array[1..64822] of Byte;
  *)
  { Bei Überschreitung des Grenzwertes (im vorliegenden Programm bei }
  { 64 806) wird die Kompilation mit der Fehlermeldung "Error 49:    }
  { Data segment too large" abgebrochen. Der Grenzwert hängt etwas   }
  { von den verwendeten Units und von der Pascal-Version ab, beträgt }
  { aber immer max. 64 KByte. }
begin
  ClrScr;
  Write('Tastendruck ... ');
  repeat
  until KeyPressed;
end.
```

Dynamische Variablen (allgemeiner dynamische Datenstrukturen) können dagegen zu jedem beliebigen Zeitpunkt des Programmlaufs eingeführt werden. Der von ihnen belegte Speicherplatz kann bei Bedarf auch wieder freigegeben werden. Dynamische Datenstrukturen bestehen aus einem Zeiger (engl. Pointer), der im allgemeinen im statischen Variablenspeicher abgespeichert wird und dem zugehörigen Datenobjekt, das im freien Speicher (Heapspeicher, Haufenspeicher) gespeichert wird. **Bei komplexeren Datenstrukturen (verkettete Listen, Ringe, Bäume usw.) werden Zeiger auch im Heapspeicher abgelegt.** Der Zeiger enthält lediglich die Speicheradresse des Datenobjekts im Intel-Format *Segment:Offset*. Der sonst ungenutzte Heapspeicher, der häufig wesentlich größer als 64 KByte ist, kann somit sinnvoll genutzt werden. Bestimmte Datenstrukturen wie z.B. verkettete Listen sind nur mit Zeigern realisierbar.

19.2 Eigenschaften der dynamischen Variablen. Standard-Prozeduren und -Funktionen

Zeiger sind 4-Byte-Speicheradressen im Intel-Format *Segment:Offset* (ss:oo). Sie werden mit wählbaren Bezeichnern in der üblichen Form gekennzeichnet, z.B. mit

```
x, s1, s2, i, p
```

und in der Variablendeklaration mit »*zeigerbezeichner*`: ^`*datentyp*« deklariert.

Beispiel:

```
type
  Str10 = string[10];
  Ptr   = ^Integer;
var
  x:   ^Real;
  s1:  ^Str10;   { Nicht:  s1: ^string[10]; }
  s2:  ^Str10;
  i:   Ptr;      { indirekt }
```

In der Deklaration muß also der **Datentyp eines Zeigers** mit einem **vorgesetzten Hochpfeil** gekennzeichnet werden.

Mit `Pointer` steht ein vordefinierter nichttypisierter Zeigertyp zur Verfügung, der zu allen Zeigertypen kompatibel ist. Anwendung z.B. bei den Prozeduren Mark und Release.

Beispiel:
```
var
  p: Pointer;
```

Die Zeiger zeigen auf die zugehörigen **Objekte**, die mit dem Zeigerbezeichner und einem **nachgesetzten Hochpfeil** gekennzeichnet werden, z.B.

`x^, s1^, s2^ und i^.`

Die Objekte werden im freien Speicher (Heapspeicher, Haufenspeicher) abgelegt. Bei Bedarf kann der durch die Objekte belegte Speicherbereich wieder freigegeben werden (dynamische Speicherverwaltung). Die Objekte werden nicht deklariert.

Mit typkompatiblen Zeigern sind folgende Operationen möglich:

- Zuweisungen Beispiel: `s1 := s2;`
- Vergleich »=« Beispiel: `if s1 = s2 then ...;`
- Vergleich »<>« Beispiel: `if s1 <> s2 then ...;`

Eingaben und Ausgaben von Zeigern sind im Gegensatz zu den Objekten auf die sie zeigen, nicht zulässig.

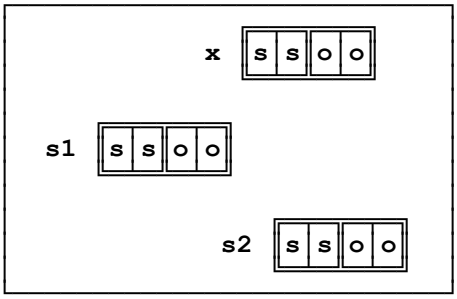
Somit **nicht** zulässig: `ReadLn(s1);` oder `WriteLn(s1);`

Mit den Objekten sind dagegen alle typgerechten Operationen zulässig, z.B:

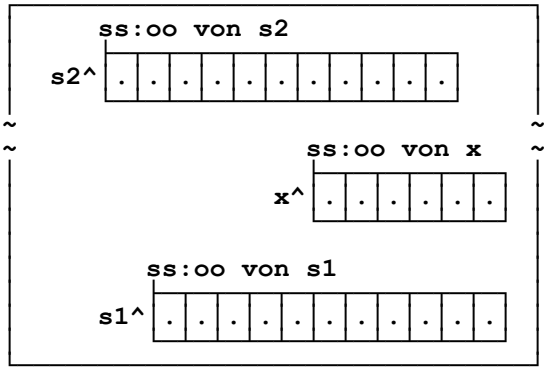
`ReadLn(s1^);` oder `WriteLn(s1^);`
`y := x^;` oder `x^ := Sin(y);` (deklarierte Real-Variable `y` vorausgesetzt)

Das Objekt kann mit Ausnahme von *»file«* jeder beliebige Datentyp sein, also z.B. auch strukturierte Typen wie Arrays, Strings, Sets oder Records.

Statischer Variablenspeicher
max. 64 KByte, mit Zeigern



Heapspeicher
mit den zugehörigen Objekten



Die Standard-Routinen:

- Die Prozedur `New(`*`zeigervariable`*`)`
  initialisiert den Zeiger und richtet Speicherplatz im Heap für neue Objekte ein. Damit wird ist zwar der Zeiger initialisiert, aber noch nicht das zugehörige Objekt. Der Zugriff über einen nichtinitialisierten Zeiger führt zu schweren Programmfehlern, da der Zeiger zufällig auch auf einen Speicherbereich zeigen kann, der vom Betriebssystem belegt ist. Der Zugriff auf das nichtinitialisierte Objekt eines initialisierten Zeigers liefert lediglich nichtdefinierten Daten-Müll.
- Die Prozedur `Dispose(`*`zeigervariable`*`)`
  gibt den (Heap-) Speicherplatz der Objekte wieder frei. Der Zeiger ist danach nicht mehr definiert und das Objekt ist nicht mehr zugänglich. Ein Zugriff stellt einen Fehler dar.
- Die spezielle Zeigerkonstante **nil**
  (reserviertes Wort, steht für *not in list*) zeigt "nirgendwo hin". Dieser Zeiger ist zu allen Zeigertypen kompatibel. Anwendung siehe spätere Demo-Programme.
- Die Prozedur `Mark(`*`zeigervariable`*`)`
  markiert die momentane Spitze des Heaps und speichert den Wert in der Zeigervariablen (die einen beliebigen Typ haben kann, z.B. auch "Pointer"). Die Mark-Prozedur (markiere Heap-Spitze) wird für die Release-Prozedur benötigt.
- Die Prozedur `Release(`*`zeigervariable`*`)`
  gibt den Bereich des Heap-Speichers zwischen der mit Mark markierten Stelle und der momentanen Spitze wieder frei. Somit werden *alle* dynamischen Variablen, die seit Mark erzeugt wurden, gelöscht. Achtung: Bei Verwendung der Unit GRAPH werden der verwendete Graphiktreiber und die verwendeten Zeichensätze dynamisch, also im Heap, gespeichert. Ein versehentliches Freigeben der von ihnen benutzten Speicherbereiche mit Mark/Release führt zum Programm-Absturz.

Hinweis: Mark/Release dürfen nicht in Verbindung mit den Prozeduren GetMem und FreeMem benutzt werden; siehe später.

- Die Funktion `MemAvail`
  liefert die Größe des gesamten freien Heap-Speichers mit dem Ergebnistyp LongInt.
- Die Funktion `MaxAvail`
  liefert die Größe des größten freien zusammenhängenden Blocks im Heap-Speichers mit dem Ergebnistyp LongInt.
- Die Funktion `SizeOf(`*`variable`*`)`
  `SizeOf(`*`datentyp`*`)`
  ist allgemein verwendbar und liefert die Größe einer Variablen oder eines Datentyps.

  Beispiel: `Write(SizeOf(s1^));`

  In Verbindung mit dynamischen Variablen ist es wichtig zu wissen, daß ab Turbo-Pascal Version 6.0 der belegte Speicher im Heap-Speicher immer auf volle 8-Byte-Gruppen aufgefüllt wird. In der Regel wird also mehr Speicher reserviert, als tatsächlich benötigt wird. Besteht z.B. ein dynamischer Rekord aus zwei Real-Feldern (je 6 Byte), so liefert SizeOf 12, reserviert werden aber 16 Byte im Heap-Speicher. Auf diesen Umstand ist bei der Ermittlung der maximalen Anzahl der Daten im Heap durch eigene programmtechnische Maßnahmen einzugehen, da SizeOf alleine im allgemeinen ein falsches Ergebnis liefert. Nur im Sonderfall "`SizeOf(...)` **`mod`** `8 = 0`" ist das Ergebnis richtig. Siehe späteres Beispiel.
- Die Prozedur `GetMem(`*`zeigervariable, groesse`*`)`
  `groesse`: Word-Ausdruck
  reserviert einen Bereich bestimmter Größe auf dem Heap-Speicher. Diese Prozedur darf nicht in Verbindung mit Mark/Release benutzt werden.
- Die Prozedur `FreeMem(`*`zeigervariable, groesse`*`)`
  `groesse`: Word-Ausdruck
  gibt einen Bereich bestimmter Größe auf dem Heap-Speicher wieder frei. Diese Prozedur darf nicht in Verbindung mit Mark/Release benutzt werden.
- In der Entwicklungsumgebung kann der akutelle Wert des Zeigers über das Dialogfenster »*Auswerten und Ändern*« (Aufruf mit *Strg+F4*) nach Eingabe des Zeiger-Bezeichners (z.B. »*s1*«) abgefragt werden. Die Speicheradresse erscheint dann im Fenster »*Ergebnis*« in der Hex-Schreibweise »*PTR(*$*segment,* $*offset)*«.

  Beispiel: `PTR($74F5,$8)`
  `$74F5` Segment-Adresse in hex
  `$8` Offset-Adresse in hex
  Übliche Intel-Darstellung in hex: `74F5:0008`
  Die physikalische Adresse des Objekts ist dann:
  `16 * (7 * 4096 + 4 * 256 + 15 * 16 + 5) + 8 = 479 064`

19.3 Demo-Programm 1: Eigenschaften dynamischer Variablen

```
program Pas19031; { Kap. 19: Dynamische Datenstrukturen      }
                  { Turbo-Pascal, kha                        }
                  { Demo: Eigenschaften der Zeiger (Pointer) }
uses
  CRT;

type
  Str10 = string[10];

var
  x:      ^Real;   { x ist ein Zeiger auf ein  Objekt  vom Typ Real  }
  i:      ^Word;   { i ist ein Zeiger auf ein  Objekt  vom Typ Word  }
  s1, s2: ^Str10;  { s1 und s2 sind Zeiger auf Objekte vom Typ Str10 }
                   { Nicht zulässig: s1, s2: ^string[10];            }
  HeapSpeicher: LongInt;

procedure WarteAufTastendruck;
var
  Ch: Char;
  Sp, Ze: Byte;
begin
  Sp := WhereX;
  Ze := WhereY;
  GotoXY(10, 25); Write('Weiter mit Tastendruck ... ');
  while KeyPressed do
    Ch := ReadKey;
  repeat
  until ReadKey <> '';
  GotoXY(1, 25); ClrEol;
  GotoXY(Sp, Ze);
end;

begin
  TextBackground(Blue); TextColor(Yellow); ClrScr;

  { Die Standardprozedur »New« belegt bei den folgenden drei Aufrufen
    für jedes Objekt einen Bereich auf dem Heap-Speicher entsprechend
    dem Datentyp des Objektes, bei »x^« 6 Byte, bei »s1^« und »s2^«
    je 11 Byte (10 Zeichen und das Längenbyte), und setzt die Zeiger
    »x«, »s1« und »s2« auf die Anfangsadresse des jeweiligen
    Bereiches.

    Der benötigte Speicherplatz des Objektes wird aber ab Turbo-
    Pascal 6.0 immer auf volle 8-Byte-Gruppen aufgerundet (bei Arrays
    ist für die Aufrundung die Gesamtgröße und nicht die Größe eines
    Elementes maßgebend). Im vorliegenden Fall werden 48 Byte (und
    nicht 30 Byte) auf dem Heap-Speicher belegt.

    Für s1^: 11 Byte -----> 16 Byte
    Für s2^: 11 Byte -----> 16 Byte
    Für x^:   6 Byte ----->  8 Byte
    Für i^:   2 Byte ----->  8 Byte
                      Summe: 48 Byte auf Heap-Speicher belegt
  }

  HeapSpeicher := MemAvail;  { Funktion »MemAvail« liefert   }
```

```
                { die Gesamtgröße des freien Heap-Speiches }
                { Datentyp LongInt.                        }
  WriteLn('Der freie Heapspeicher vorher:     ', HeapSpeicher);
  WriteLn('Der größte zusammenhängende Block: ', MaxAvail, ' vor');
  WarteAufTastendruck;
  New(s1);          { Zeiger s1 }
  New(s2);          { Zeiger s2 }
  New(x);           { Zeiger x  }
  WriteLn('Der größte zusammenhängende Block: ', MaxAvail, ' nach');
  WriteLn('Durch s1, s2 und x belegt:     ',
          HeapSpeicher - MemAvail);
  WriteLn;
  s1^ := 'Julia';   { Objekt s1^ belegen }
  s2^ := 'Konrad';  { Objekt s2^ belegen }
  x^  := 47.11;     { Objekt x^  belegen }
  Writeln('01: Objekt s1^: ', s1^);   {¦01: Objekt s1^: Julia       }
  Writeln('02: Objekt s2^: ', s2^);   {¦02: Objekt s2^: Konrad      }
  Writeln('03: Objekt x^ : ', x^:5:2);{¦03: Objekt x^ : 47.11       }
  if s1 = s2
     then Writeln('04: Zeiger s1 =  Zeiger s2')
     else Writeln('04: Zeiger s1 <> Zeiger s2');
                                      {¦04: Zeiger s1 <> Zeiger s2  }
  WarteAufTastendruck;
  s1 := s2;
  { Der Zeiger s1 wird mit dem Zeiger s2 belegt. Das alte Objekt
    s1^ ist nicht mehr erreichbar. Es steht aber als Müll noch im
    Heap-Speicher. }
  Writeln('05: Objekt s1^: ', s1^);   {¦05: Objekt s1^: Konrad      }
  Writeln('06: Objekt s2^: ', s2^);   {¦06: Objekt s2^: Konrad      }
  if s1 = s2
     then Writeln('07: Zeiger s1 =  Zeiger s2')
     else Writeln('07: Zeiger s1 <> Zeiger s2');
                                      {¦07: Zeiger s1 = Zeiger s2   }
  s1^ := 'Anton';
  Writeln('08: Objekt s1^: ', s1^);   {¦08: Objekt s1^: Anton       }
  Writeln('09: Objekt s2^: ', s2^);   {¦09: Objekt s2^: Anton       }
  s2^ := 'Huber';
  Writeln('10: Objekt s1^: ', s1^);   {¦10: Objekt s1^: Huber       }
  Writeln('11: Objekt s2^: ', s2^);   {¦11: Objekt s2^: Huber       }
  if s1^ = s2^
     then Writeln('12: Objekt s1^ =  Objekt s2^')
     else Writeln('12: Objekt s1^ <> Objekt s2^');
                                      {¦12: Objekt s1^ = Objekt s2^ }
  WarteAufTastendruck;
  Dispose(s1); { »Dispose« gibt den Speicherbereich frei, auf den   }
               { der Zeiger »s1« zeigt. »s1« ist nach »Dispose«     }
               { nicht mehr definiert. Das Objekt »s1^« ist nicht   }
               { mehr zugänglich. Ein Zugriff darauf stellt einen   }
               { schweren Fehler dar, der aber nicht gemeldet wird. }
```

```
  WriteLn('13: Zeiger s1 mit Dispose frei. Fehlerhafter Zugriff ',
          'auf Objekt s1^: ', s1^);
  WarteAufTastendruck;

  New(s1);

  s1^ := 'Meier';
  WriteLn('14: Objekt s1^: ', s1^);   {¦14: Objekt s1^: Meier      }

  New(s1);
              { Nachfolgend Müll, da Objekt s1^ nicht initialisiert }
  WriteLn('15: Objekt s1^: ', s1^);   {¦15: Objekt s1^: ... Müll .. }

  WriteLn('16: "SizeOf": Größe Zeiger s2 und Objekt s2^: ',
               SizeOf(s2), ', ', SizeOf(s2^));
               { Größe des Zeigers immer 4 Byte! }
                                      {¦16: "SizeOf": ..... : 4, 11 }
  WarteAufTastendruck;

  GetMem(i, 2);  { Syntax: "GetMem(zeigervariable, groesse)" }
                 {          Entspricht "New(i)"              }
  i^ := 4711;    {          wobei "groesse" Word-Ausdruck    }
  WriteLn('17: Objekt i^:  ', i^);    {¦17: Objekt  i^:  4711      }

  FreeMem(i, 2); { Syntax: "FreeMem(zeigervariable, groesse)" }
                 {          wobei: "groesse" Word-Ausdruck    }

  Dispose(s1);
  Dispose(s2);
  Dispose(x);

  WarteAufTastendruck;
end.
```

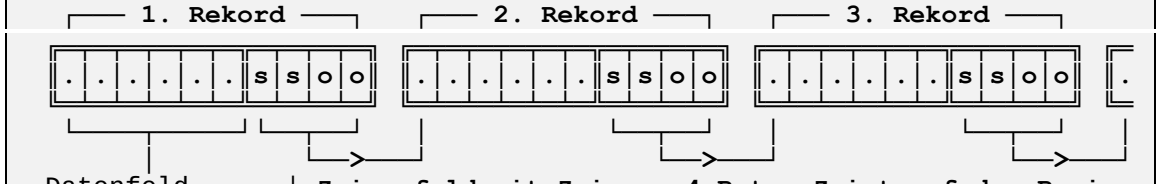
19.4 Demo-Programm 2: Anwendung bei großen Arrays

```
program Pas19041; { Kap. 19: Dynamische Datenstrukturen             }
        { Demo: Verwendung des Heap-Speichers für große Datenmengen. }
        { Zwei Real-Arrays mit zusammen 120 000 Byte.               }
uses
  CRT;

const
  iMax = 10000;  { Array-Größe }

type
  RealArray = array[1..iMax] of Real;        { Datentyp Real: 6 Byte }

var
  i:       Word;
  Array1,              { Bei statischen Arrays käme die Fehler-      }
  Array2:  ^RealArray; { meldung »Error 96: Too many variables«,     }
                       { da die beiden Arrays zusammen 120 000 Byte  }
                       { und das Datensegment max. 64 KByte für      }
                       { globale Variablen und typisierte Konstanten }
                       { zur Verfügung stellt.                       }
begin
  TextBackground(Blue); TextColor(Yellow); ClrScr;
  GotoXY(15, 2); WriteLn('Demonstration: Dynamische Variablen');
  TextColor(White);
```

```
  GotoXY(15, 4); WriteLn('Der Heap-Speicher vorher:  ', MemAvail:7);

  New(Array1);    { Reservieren des Speicher- }
  New(Array2);    { platzes auf dem Heap      }

  GotoXY(15, 5); WriteLn('Der Heap-Speicher nachher: ', MemAvail:7);
  GotoXY(15, 7);
  WriteLn('Initialisieren der beiden Real-Arrays bis ', iMax);

  for i := 1 to iMax do
    begin
      Array1^[i] :=  Sqrt(i);   { Demo-  1.00  1.41  1.73 ... 100.00 }
      Array2^[i] := -Sqrt(i);   { Daten -1.00 -1.41 -1.73 ...-100.00 }
    end;                        {        ----- ----- -----     ----- }
            { Später Summenberechnung:  0.00  0.00  0.00 ...   0.00 }
  GotoXY(15, 12);
  WriteLn('Es folgt Addition der Elemente der beiden Arrays ... ');

  for i := 1 to iMax do
    begin
      Array1^[i]  :=  Array1^[i] + Array2^[i];
      GotoXY(15, 13);
      WriteLn('i = ', i:5, ',  Summe = ', Array1^[i]:12:9);
    end;
  Dispose(Array1);  { Am Programmende }
  Dispose(Array2);  { nicht notwendig }

  GotoXY(15, 15); Write('Fertig ... ');
  repeat
  until KeyPressed;
end.
```

19.5 Demo-Programm 3: Anwendung bei verketteten Listen

```
program Pas19051;  { Kap. 19: Dynamische Datenstrukturen      }
  { Turbo-Pascal,    Demo: Einfach verkettete Liste mit Reals }
  { In diesem Programm wird eine verkettete Liste demonstriert. Mit
    ihm kann im Rahmen des Heap-Speichergröße eine beliebige Anzahl
    von Real-Daten verarbeitet werden. Zur Demonstration wird eine
    wählbare Anzahl von zufälligen Real-Daten generiert und in die
    verkettete Liste geschrieben. Die Daten werden anschließend wieder
    ausgelesen. Dabei wird (nur zur Demo) der Mittelwert dieser Daten
    berechnet.

    Das Programm kann als Modell für verkettete Listen anderer Daten-
    typen dienen.
```




```
  Datenfeld       | Zeigerfeld mit Zeiger: 4 Byte. Zeigt auf den Beginn
  Real x: 6 Byte  | des nächsten Rekords. Beim letzten gültigen Rekord
                  | muß "nil"  in das Zeigerfeld geschrieben werden.
  }
uses
```

```
  CRT;

type
  ZgrTyp = ^Rekord; { Zeigt auf Records des Typs "Rekord". Vorwärts- }
  Rekord = record   { bezug in diesem Fall bei Zeigern zulässig.     }
             DatenFeld:  Real;     { Real x: 6 Byte                  }
             ZeigerFeld: ZgrTyp;   { Zeiger: 4 Byte, Summe = 10 Byte }
           end;     { Turbo-Pascal: 16 Byte Speicherbedarf und       }
                    { nicht 10, wegen Auffüllen auf 8-Byte-Gruppen   }
var
  p,
  pStart,
  pLetzt,
  pNeu:      ZgrTyp;
  Zeiger:    Pointer; { "Pointer": Vordefin. untypisierter Zeigertyp }
  i, n,
  nMax1,
  nMax2:     LongInt;
  x,
  xMittel:   Real;
  Ch:        Char;
  Bildschirmanzeige: Boolean;

begin
  repeat
    TextBackGround(Blue); TextColor(Yellow); ClrScr;
    Writeln('Demo: Einfach verkettete Liste mit Zufalls-' +
            'Real-Zahlen im Datenfeld', #13#10);
    TextColor(White);

    nMax1 := MemAvail div (8 * (SizeOf(p^) div 8 +  1)) - 1;
        { nMax1 so für TP ab 6.0, ".. auf Achtergruppen auffüllen"   }
        { "SizeOf(p^)" selbst liefert den (Netto-) Wert 10           }

        { oder in Primitivform durch "Abzählen" so: }
    nMax2 := MemAvail div 16 - 1;   { zwar nur 10 Byte, aber Auf- }
        { TP-6.0: ... div 16 - 1 }  { füllen auf Achtergruppen .. }
        { TP-5.0: ... div 10 - 1 }

    WriteLn('Der Heap-Speicher: ', MemAvail,
            ', somit nMax1 = ', nMax1, ', bzw. nMax2 = ', nMax2);

    Write('Eingabe Anzahl n, Ende mit 0: ');
    ReadLn(n);

    if n <= 0 then EXIT;   { >>>>>>>>>>>>>> }

    Write('Mit Bildschirmanzeige (j/n):  j');
    GotoXY(WhereX - 1, WhereY);
    repeat
      Ch := ReadKey;
      if Ch = #13
         then Ch := 'j';
    until UpCase(Ch) in ['J', 'N'];
    WriteLn(Ch); WriteLn;

    if UpCase(Ch) = 'J'
       then Bildschirmanzeige := True
       else Bildschirmanzeige := False;

    Mark(Zeiger);     { Hier: Heap-Anfang "markieren"             }
```

```
    New(p);           { Erzeugt den ersten dynamischen Rekord     }
    pStart := p;      { Zeiger pStart wird ebenfalls auf diesen   }
                      { Rekord gesetzt (Sichern des Startzeigers) }
    WriteLn('--- Das Einlesen ----':31);
    for i := 1 to n do
      begin
        x := Random;  { Für Demo: Zufalls-Realzahlen  0.0 <= x < 1.0 }
        if Bildschirmanzeige     { Bildschirmanzeige nur für Demo    }
           then Writeln('i = ':14, i:2, x:15:6);
        p^.DatenFeld   := x;     { In Datenfeld des Rekords p^ den   }
                                 { Real-Wert x schreiben             }
        pLetzt         := p;     { Zeiger sichern, damit später ...  }
        New(pNeu);               { Neuen Rekord anlegen              }
        p^.ZeigerFeld  := pNeu;  { In Zeigerfeld des alten Rekords   }
                                 { neuen Zeiger schreiben            }
        p              := pNeu;  { Aktuellen Zeiger p auf neuen Wert }
      end;
    pLetzt^.ZeigerFeld := nil;   { In das Zeigerfeld des letzten     }
                                 { Rekords wird "nil" geschrieben    }
    WriteLn;
    WriteLn('--- Das Auslesen ----':31);
    p       := pStart;  { Zeiger p zurück auf den ersten Rekord      }
    xMittel := 0.0;
    i       := 0;       { Die Anzahl braucht nicht bekannt zu sein ! }
    while p <> nil do
      begin
        Inc(i);          { Der Zähler }
        if Bildschirmanzeige     { Bildschirmanzeige nur für Demo    }
           then Writeln('i = ':14, i:2, p^.Datenfeld:15:6);
        xMittel  := xMittel + p^.Datenfeld;
        p := p^.ZeigerFeld;   { nächster Record }
      end;
    xMittel := xMittel / i;
    WriteLn;
    WriteLn('Anzahl der Daten: ', i, '   Mittelwert: ', xMittel:9:6);
    Write(#13#10, 'Weiter mit Tastendruck ... ');
    while KeyPressed do
      Ch := ReadKey;
    Ch := ReadKey;
    Release(Zeiger);      { Heap ab markierter Stelle wieder }
                          { freigeben. Hier: gesamten Heap.  }
  until n = 0;
end.
```

19.6 Demo-Programm 4: Einfügen in verkettete Liste

```
program Pas19061;  { Kap. 19: Dynamische Datenstrukturen     }
  { Turbo-Pascal, Demo: Einfügen in einfach verkettete Liste }
  {
    ┌── 1. Rekord ──┐    ┌── 2. Rekord ──┐    ┌── 3. Rekord ──┐
   ┌─┬─┬─┐ ┌─┬─┬─┬─┬─┐ ┌─┬─┬─┐ ┌─┬─┬─┬─┬─┐ ┌─┬─┬─┐ ┌─┬─┬─┬─┬─┐ ┌
   │.│.│.│ │.│.│s│s│o│o│ │.│.│.│ │.│.│s│s│o│o│ │.│.│.│ │.│.│s│s│o│o│ │.
   └─┴─┴─┘ └─┴─┴─┴─┴─┘ └─┴─┴─┘ └─┴─┴─┴─┴─┘ └─┴─┴─┘ └─┴─┴─┴─┴─┘ └
     └──────┐  └──────┐  └──────┐  └──────┐  └──────┐  └──────┐
            ▼         ▼         ▼         ▼         ▼         ▼
  Datenfeld       | Zeigerfeld mit Zeiger: 4 Byte. Zeigt auf den Beginn
  String s:       | des nächsten Rekords. Beim letzten gültigen Rekord
  string[80]      | muß "nil"  in das Zeigerfeld geschrieben werden.
  81 Byte  }

uses
  CRT;

const
  Dummy = 'Dummy';

type
  Str80  = string[80];
  ZgrTyp = ^Rekord; { Zeigt auf Records des Typs »Rekord«. Vorwärts- }
  Rekord = record   { bezug in diesem Fall bei Zeigern zulässig.     }
             DatenFeld:  Str80;   { String: 81 Byte                  }
             ZeigerFeld: ZgrTyp;  { Zeiger:  4 Byte, Summe = 85 Byte }
           end;     { Turbo-Pascal: 88 Byte Speicherbedarf und       }
                    { nicht 10, wegen Auffüllen auf 8-Byte-Gruppen   }
var
  p,
  pStart,
  pLetzt,
  pNeu:      ZgrTyp;
  Zeiger:    Pointer; { "Pointer": Vordefin. untypisierter Zeigertyp }
  i, n,
  nMax1,
  nMax2:     LongInt;
  s:         Str80;
  Ch:        Char;
  Gefunden:  Boolean;
  Bildschirmanzeige: Boolean;

procedure Tastendruck;
begin
  TextColor(Yellow);
  Write(#13#10, 'Weiter mit Tastendruck ... ');
  while KeyPressed do
    Ch := ReadKey;
  Ch := ReadKey;
  WriteLn;
  TextColor(White);
end;

begin
```

```
repeat
  TextBackground(Blue); TextColor(Yellow); ClrScr;
  Writeln('Demo: Einfach verkettete Liste mit Strings im ' +
          'Datenfeld', #13#10);
  TextColor(White);

  nMax1 := MemAvail div (8 * (SizeOf(p^) div 8 +  1)) - 1;
      { nMax1 so für TP 6.0, ".. auf Achtergruppen auffüllen"      }
      { "SizeOf(p^)" selbst liefert den (Netto-) Wert 10           }

      { oder in Primitivform durch "Abzählen" so: }
  nMax2 := MemAvail div 88 - 1;   { zwar nur 85 Byte, aber ab }
      { TP-6.0: ... div 88 - 1 }  { TP 6.0 auf Achtergruppen .}
      { TP-5.0: ... div 85 - 1 }

  WriteLn('Der Heap-Speicher: ', MemAvail,
          ', somit nMax1 = ', nMax1, ', bzw. nMax2 = ', nMax2);

  Write('Eingabe Anzahl n, Ende mit 0: ');
  ReadLn(n);

  if n <= 0 then EXIT;   { >>>>>>>>>>>>>> }

  Write('Mit Bildschirmanzeige (j/n):  j');
  GotoXY(WhereX - 1, WhereY);
  repeat
    Ch := ReadKey;
    if Ch = #13
       then Ch := 'j';
  until UpCase(Ch) in ['J', 'N'];
  WriteLn(Ch); WriteLn;

  if UpCase(Ch) = 'J'
     then Bildschirmanzeige := True
     else Bildschirmanzeige := False;

  Mark(Zeiger);     { Hier: Heap-Anfang "markieren"             }

  New(p);           { Erzeugt den ersten dynamischen Rekord     }
  pStart := p;      { Zeiger pStart wird ebenfalls auf diesen   }
                    { Rekord gesetzt (Sichern des Startzeigers) }

  WriteLn('--- Das Einlesen ----':31);

  for i := 1 to n do
    begin
      Str(i, s);
      s := Dummy + ' ' + s;    { Nur für Demo:  }
      if Bildschirmanzeige     { Bildschirmanzeige nur für Demo    }
         then Writeln('i = ':14, i:6, ':     ', s);
      p^.DatenFeld   := s;     { Den String s in das Datenfeld des }
                               { Rekords p^ schreiben              }
      pLetzt         := p;     { Zeiger sichern, damit später ...  }
      New(pNeu);               { Neuen Rekord anlegen              }
      p^.ZeigerFeld  := pNeu;  { In Zeigerfeld des alten Rekords   }
                               { neuen Zeiger schreiben            }
      p              := pNeu;  { Aktuellen Zeiger p auf neuen Wert }
    end;

  pLetzt^.ZeigerFeld := nil;   { In das Zeigerfeld des letzten     }
                               { Rekords wird »nil« geschrieben    }

  Tastendruck;
```

```
    WriteLn('--- Das Auslesen ----':31);

    p  := pStart;  { Zeiger p zurück auf den ersten Rekord      }
    i  := 0;       { Die Anzahl braucht nicht bekannt zu sein ! }

    while p <> nil do
      begin
        Inc(i);                { Der Zähler }
        if Bildschirmanzeige   { Bildschirmanzeige nur für Demo      }
           then Writeln('i = ':14, i:6, ':     ', p^.Datenfeld);
        p := p^.ZeigerFeld;    { nächster Record }
      end;

    WriteLn('Anzahl der Daten: ':28, i);

    Tastendruck;

    WriteLn('--- Listenelement einfügen ---':40);

    p        := pStart;  { Zeiger p zurück auf den ersten Rekord     }
    Gefunden := False;

    s        := 'Dummy 3';  { Danach soll eingefügt werden }

    while (p <> nil) and (not Gefunden) do
      begin
        if Bildschirmanzeige     { Bildschirmanzeige nur für Demo   }
           then Writeln(p^.Datenfeld:33);
        if p^.Datenfeld = s
           then begin
                  Gefunden := True;
                  New(pNeu);
                  pNeu^.Datenfeld  := 'Einfügezeile';
                  pNeu^.Zeigerfeld := p^.Zeigerfeld;
                  p^.Zeigerfeld    := pNeu;
                end
           else p := p^.ZeigerFeld;   { nächster Record }
      end;

    Tastendruck;

    WriteLn('--- Das Auslesen nach der Erweiterung ----':52);
    p       := pStart;  { Zeiger p zurück auf den ersten Rekord      }
    i       := 0;       { Die Anzahl braucht nicht bekannt zu sein ! }

    while p <> nil do
      begin
        Inc(i);                  { Der Zähler }
        if Bildschirmanzeige     { Bildschirmanzeige nur für Demo    }
           then WriteLn('i = ':14, i:6, ':     ', p^.Datenfeld);
        p := p^.ZeigerFeld;      { nächster Record }
      end;
    WriteLn('Anzahl der Daten: ':28, i);

    Tastendruck;

    Release(Zeiger);      { Heap ab markierter Stelle wieder }
                          { freigeben. Hier: gesamten Heap.  }
  until n = 0;
end.
```

80250404 Dr. K. Haller