

# 7 Laufzeit-Speicherverwaltung

## 7.1 Grundlagen

Bevor wir die Code-Generierung betrachten, müssen wir uns Gedanken über zur Laufzeit des zu generierenden Programms notwendige Aktivitäten zur Zuordnung und Freigabe von Speicherplatz machen.

Diese Manipulationen werden i.a. von einem *laufzeitunterstützenden Paket* (Runtime-Package) ausgeführt, das Routinen enthält, die zusammen mit dem generierten Code geladen werden. Die uns interessierenden *Datenobjekte*, um deren Speicherplatz es geht, sind

- *explizit* in Programmen vorkommende Größen (Variable, Arrays, Records),
- *intern* erzeugte Hilfsvariable (z.B. zur Aufnahme von Zwischenwerten bei der Auswertung von Ausdrücken),
- *Verwaltungsinformationen* für die Ablauforganisation: Unterprogrammaufrufe, Verwaltung von dynamischen Datenbereichen.

### Zentrale Begriffe der Speicherverwaltung

- *Lebensdauer* von Objekten
- *Gültigkeitsbereich* von Objekten
- *Statische* Speicherverwaltung  
Objekte existieren während der gesamten Laufzeit des Programms. Der Compiler kann ihnen direkt Adressen zuordnen
- *Dynamische* Speicherverwaltung  
Objekte werden während des Programmlaufs angelegt und auch wieder freigegeben, Zugriff erfordert i.a. Adreßrechnung
- *Programmeinheit*  
bezüglich der Speicherverwaltung nicht weiter zerlegbare Teile des Programms  
**Bsp.:** Prozedur, Funktion, Block
- *Datenraum*  
explizit im Programm verwendete, intern erzeugte Daten und Verwaltungsinformationen

**Bsp.:** FORTRAN besitzt eine *statische* Speicherverwaltung.

Pascal besitzt eine *dynamische* Speicherverwaltung (Lebensdauer von lokalen Prozedurvariablen ist identisch mit der *Aktivierungsdauer* einer Prozedur bei „Stack-dynamischen“ Variablen. Mit NEW und DISPOSE können zusätzlich zu beliebigen Zeitpunkten „Heap-dynamische“ Datenbereiche erzeugt und freigegeben werden).

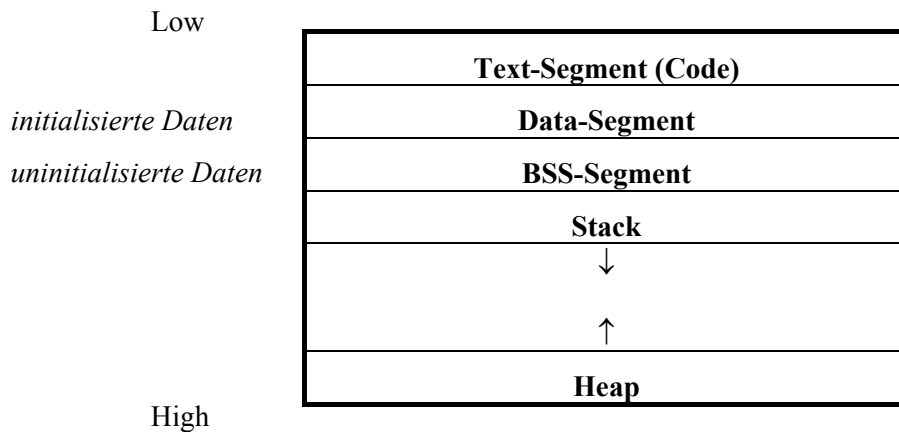
C ähnelt Pascal, es gibt keine geschachtelten Prozeduren, aber zusätzliche anonyme Anweisungs-Blöcke zwischen { und }.

## 7.2 Unterteilung des Laufzeitspeichers

Der vom Betriebssystem für die Ausführung eines Programmes zur Verfügung gestellte Speicherplatz muß vom Compiler geeignet unterteilt werden, um folgendes unterzubringen:

- den erzeugten Zielcode
- Datenobjekte
- Stack zum Speichern von Informationen über Prozeduraktivierungen
- eventuell Heap für spezielle, dynamisch erzeugte Datenobjekte

Der *Zielcode* kann in einem *statisch* festgelegten Bereich abgelegt werden, z.B. am unteren Ende des Speichers. Die Größe einiger Datenobjekte kann zur Übersetzungszeit bekannt sein, was eine statische Zuordnung zu einem Bereich möglich macht.



Sprachen wie Pascal oder C verwenden den Stack, um Datenobjekte zu speichern, deren Lebensdauer an Prozeduraktivierungen gebunden ist : Stack-dynamische Objekte.

Der *Heap* dient zur Ablage aller anderen Datenobjekte : Heap-dynamische Objekte.

## 7.3 Activation Records (AR)

Die Information, die man bei einer Aktivierung einer Prozedur benötigt, wird durch einen zusammenhängenden Speicherblock verwaltet, der Activation Record, auch Frame, DSA genannt wird.

### Prinzipieller Aufbau eines Activation Records

Rückgabewert	(häufig in Registern)
aktuelle Parameter (optional)	(häufig in Registern)
Dynamic Link (optional)	Zeiger auf den AR der aufrufenden Programmeinheit
Static Link (optional)	Zeiger auf den AR der statisch überge- ordneten Programmeinheit
geretteter Maschinenzustand (Rückkehradresse)	
lokale Daten	
interne Daten	

Weder alle Sprachen noch alle Compiler brauchen jedes Feld des AR.

Bei Sprachen wie Pascal oder C ist es üblich, den AR bei der Aktivierung einer Prozedur auf den Laufzeit-Stack zu *legen* und wieder zu entfernen, wenn die Aktivierung beendet ist.

### Weitere wichtige Begriffe

*Environment* (Umgebung) einer aktiven Prozedur: alle Datenräume, auf die sie Zugriff hat.

**Bsp.:** Pascal: lokale Variablen, Parameter, statisch übergeordnete Datenräume und Heap.  
FORTRAN: lokale Variablen, COMMON-Blöcke, Parameter.

*Runtime-Stack:* Bei der Aktivierung der Programmeinheit wird durch den auf dem Stack gepushten AR ein neuer Datenraum angelegt. Beim Verlassen der Einheit wird dieser freigegeben (Pop).

Wesentliche Vorkehrungen dabei:

- Vorbereitung für den Abbau des Stacks
- Maßnahmen zum Auffinden des Environments

**Bsp.:** (Pascal)

```
PROGRAM Main;

VAR i1,i2: integer;

  PROCEDURE P1;
  VAR j1,j2: integer;
  BEGIN
    ...
    IF j1 = 0 THEN P2;
    ...
  END;
```

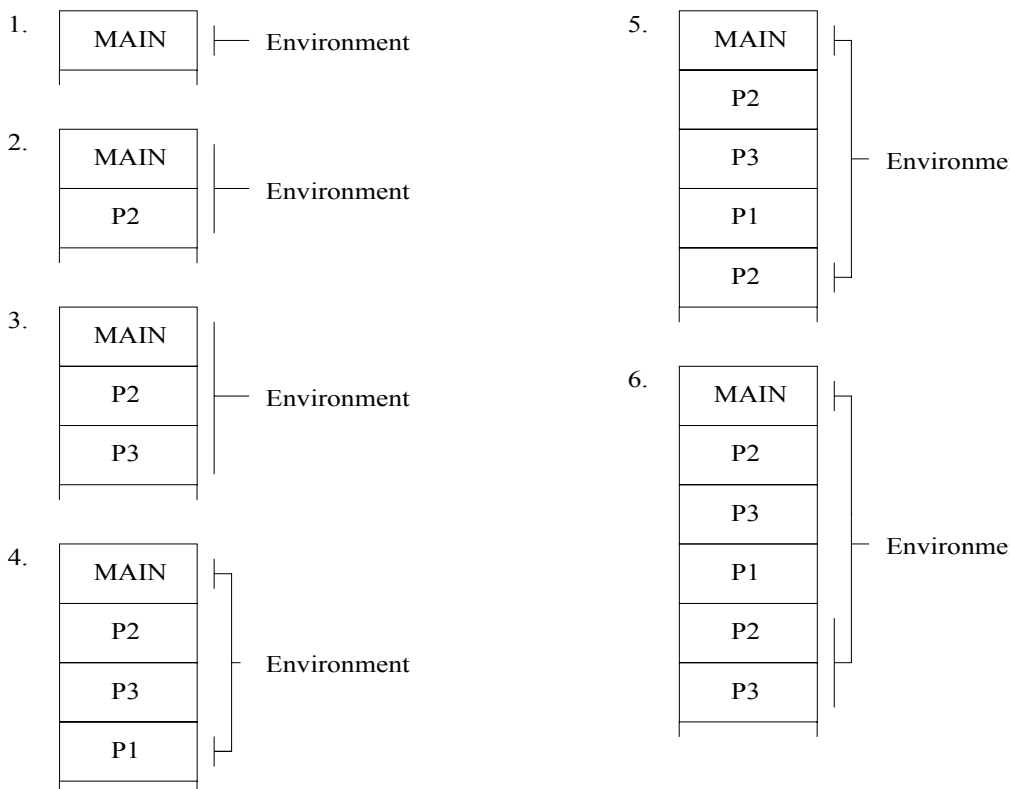
```

PROCEDURE P2;
VAR k1,k2 : integer;
  PROCEDURE P3;
  VAR l1,l2: integer;
  BEGIN
    ...
    IF l1 = 0 THEN P1;
  END;
BEGIN
  ...
  IF k1 = 0 THEN P3;
  ...
END

BEGIN
  ...
  IF i1 = 0 THEN P2;
  ...
END.

```

*Aufgabe:* Aufbau des Stacks betrachten bei einer Folge möglicher Prozeduraufrufe (immer Bedingung *true*)!



In C gibt es keine geschachtelten Funktionen. Wir müssen deshalb das Beispiel inhaltlich modifizieren.

**Bsp.: (C)**

```
void P1();
void P2();
void P3();

void P1() {
    int j1,j2;

    ...
    if (j1 == 0) P2();
    ...
}

void P2() {
    int k1,k2;

    ...
    if (k1 == 0) P3();
    ...
}

void P3() {
    int l1,l2;

    ...
    if (l1 == 0) P1();
    ...
}

int main(){
    int i1,i2;

    ...
    if (i1 == 0) P2();
    ...
}
```

Die Aufruffreihenfolge ist dieselbe wie im Pascal-Beispiel. Zum Environment gehören jedoch immer nur die unterste Funktion und etwaige globale Daten, nämlich externe Variablen.

## 7.4 Dynamic-Link- und Static-Link-Ketten

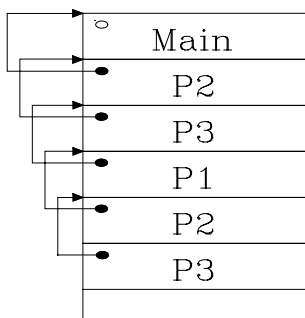
Der AR einer Programmeinheit enthält i.a. zwei Zeiger: Dynamic Link und Static Link. Dynamic-Link zeigt auf den AR der aufrufenden Prozedur bzw. Einheit. Static-Link zeigt auf den AR der statisch übergeordneten Einheit (Schachtelung im Text).

Die Dynamic-Link-Kette dient zum Abbau des Stacks bei der Rückkehr an die Aufrufstelle.  
Die Static-Link-Kette verkettet alle Datenräume, die zum Environment einer Prozedur gehören.

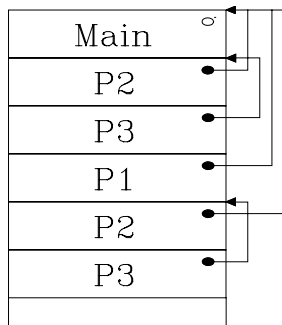
Objekte können adressiert werden über

- die Differenz zwischen dem Niveau ihrer Verwendung und dem Niveau ihrer Deklaration,
- die Distanz zur Basis des Datenraumes, dem sie angehören.

**Bsp.:** Dynamic-Link-Kette nach dem 2. Aufruf von P3



Static-Link-Kette für den gleichen Fall



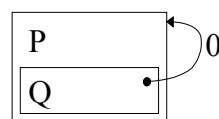
### Initialisierung der Static-Link-Kette beim Einrichten eines AR im Stack

Ist  $n$  das Niveau der aufrufenden Prozedur, und  $m$  das Niveau der aufgerufenen Prozedur, so geht man in der Static-Link-Kette der aufrufenden Prozedur um  $k = (n-m+1)$  Schritte zurück und setzt den Static-Link auf den so erreichten AR.

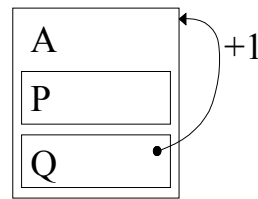
Wichtige Fälle sind:

$$m = n + 1 \quad \text{P ruft Q}$$

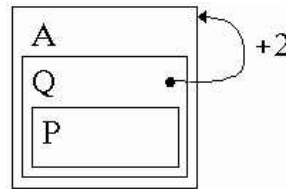
$$\Rightarrow k = n - n - 1 + 1 = 0$$



$m = n$             P ruft Q  
 $\Rightarrow k = n - n + 1 = 1$



$m = n - 1$         P ruft Q  
 $\Rightarrow k = n - n + 1 + 1 = 2$

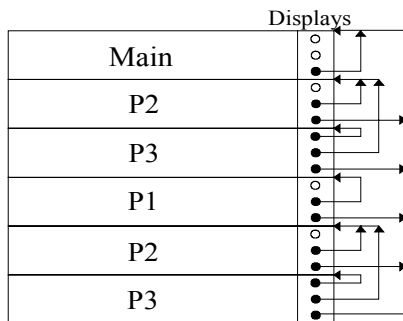


## Displays

Diese stellen eine Alternative zur Static-Link-Kette dar. Jeder AR enthält einen Vektor mit Zeigern auf alle zum Environment gehörenden Datenräume. Objekte können adressiert werden über

- das statische Niveau ihrer Deklaration,
- die Distanz zur Basis des Datenraumes, dem sie angehören.

**Bsp.:** von früher



### Initialisierung eines Displays bei Anlegen eines AR auf den Stack

1. Sei  $m$  das Niveau der gerufenen Prozedur.  
 Übernahme des Display-Vektors aus dem AR der rufenden Prozedur für die Indexposition  $0, 1, \dots, m-1$ .
2. Auf Position  $m$  wird die Adresse des neuen AR eingetragen.

## 7.5 Verwaltung dynamischer Arrays

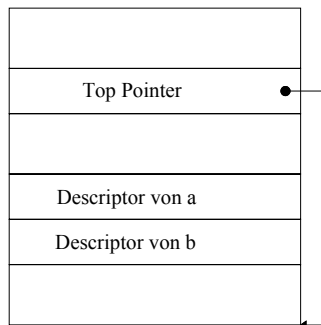
Wenn eine Programmiersprache nur Arrays mit konstanten Grenzen kennt (z.B. Pascal), so können Arrays wie gewöhnliche Variablen im Activation Record angelegt werden. Insbesondere sind bekannt: Gesamtlänge, Distanz jeder Komponente des Arrays zur Basis des Datenraums.

Wenn eine Sprache Arrays mit dynamischen Grenzen zuläßt (ALGOL 60, PL/I), so wird zur Übersetzungszeit nur ein Deskriptor für das Feld angelegt, der einen festen Platzbedarf hat. Mit Hilfe der Komponenten des Deskriptors kann dann der Code zum Zugriff auf die Komponenten des Arrays erzeugt werden. Zur Laufzeit wird dann der Activation Record erzeugt und dabei der Descriptor initialisiert und Platz für die Daten des Arrays reserviert.

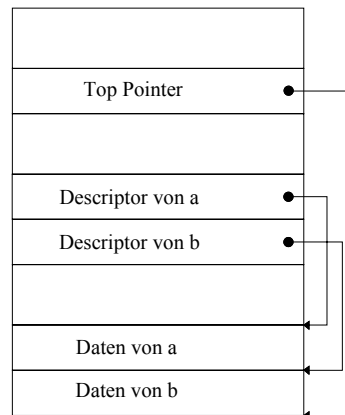
Diese dynamische Array-Verwaltung kann natürlich auch auf statische Arrays angewandt werden.

**Bsp.:** a: ARRAY [1..m] OF integer;  
 b: ARRAY [m..n] OF real;

1. Schritt



2. Schritt



### Array-Deskriptoren

Für ein Array mit n Dimensionen, etwa der Gestalt

a: ARRAY [u<sub>1</sub>..o<sub>1</sub>, u<sub>2</sub>..o<sub>2</sub>, ..., u<sub>n</sub>..o<sub>n</sub>] OF integer

in Pascal-Schreibweise, wird ein Deskriptor mit folgenden Aufbau angelegt:

u <sub>1</sub>	o <sub>1</sub>	s <sub>1</sub>
u <sub>2</sub>	o <sub>2</sub>	s <sub>2</sub>
.		
.		
.		
U <sub>n</sub>	o <sub>n</sub>	s <sub>n</sub>
a <sub>0</sub>		

s<sub>i</sub>: Größe einer Array-Scheibe der (n-i)-ten Dimension

s<sub>n</sub> = Größe einer Array-Komponenten

s<sub>n-1</sub> = Größe einer Zeile, etc.

a<sub>0</sub>: Adresse des fiktiven Array-Anfangs

**Bsp.:** a: ARRAY [1..4, 2..3] OF real; (4 Byte = real)

1	4	8
2	3	4
a <sub>0</sub>		

$$a_0 = \text{Adresse}(a[1,2]) - (2*4 + 1*8)$$

a[1, 2]
a[1, 3]
a[2, 2]
a[2, 3]
a[3, 2]
a[3, 3]
a[4, 2]
a[4, 3]

**Zugriff:**

$$\text{Adresse}(a[i, j]) = a_0 + i*8 + j*4$$