Speicherverwaltung

Bezüglich der Speicherverwaltung wurden die Verhältnisse gegenüber den Anfangsjahren der Computerei immer komplizierter. Anfangs befasste sich der ganze Rechner jeweils nur mit einem einzigen Programm, und der Programmierer hatte den gesamten realen Arbeitsspeicher in seinem Zugriff. Falls er dynamische Datenstrukturen realisieren wollte, musste er die Speicherverwaltung selbst programmieren. Später kamen Programmiersprachen hinzu, die dem Anwendungsprogrammierer die Programmierung der Speicherverwaltung abnahmen. Wenn man heute mit einer Anweisung der Form

```
pointerzelle:= NEW(zellentyp)
```

Speicherplatz für eine Zelle bestimmten Typs anfordert, muss man sich als Anwendungsprogrammierer nicht mehr darum kümmern, wo dieser Speicherplatz real im Arbeitsspeicher reserviert wird. Dennoch sollten auch die Programmierer, die nur in höheren Sprachen programmieren, noch eine grobe Vorstellung davon haben, wie es in der Hardware des Computers aussieht.

Jedem isolierbaren Programmsystem wird heute ein sog. virtueller Arbeitsspeicheradressraum zugeordnet. Für die Speicherverwaltung innerhalb dieses virtuellen Adressraums unterscheidet man 3 große Bereiche: die program section, den Stack und die sogenannte Heap oder Halde. In der program section wird der abwickelbare Programmcode untergebracht. Der Stack, zu dem der Stackpointer im Prozessor gehört, bildet einen eigenen Speicherbereich, der keine besondere Speicherverwaltung benötigt, weil hier durch die Operationen `PUSH` und `POP` immer ganz klar ist, wo neuer Speicher hinzugenommen wird und wo Speicher freigegeben wird. Der Bereich, der eine besondere Speicherverwaltung braucht, ist die Halde. In diesem Speicherbereich liegen alle Zellen, die durch `NEW` angefordert werden. Dieser Bereich sieht aus wie die Liegewiese im Strandbad, die mehr oder weniger zufällig belegte und unbelegte Teile enthält. Das Erfassen von freigewordenen Speicherbereichen läuft unter dem Begriff „Garbage Collection“, also Einsammeln von Müll.

In Bild 53 ist gezeigt, wie man sich die Aufteilung des virtuellen Arbeitsspeicheradressraums vorstellen muss. Der Stack und die Heap wachsen gegeneinander, und die Speicherverwaltung, die als Teil des Betriebssystems programmiert wird, muss die Programmabwicklung abbrechen, wenn der Stack oder die Heap noch weiter wachsen wollen, obwohl der ganze freie Platz zwischen Heap und Stack schon verbraucht ist.

Der Stack ist nur bezüglich der Speicherverwaltung wirklich ein Stack, bezüglich des Zugriffs auf die im Stack liegenden Zellen bei der Ausführung von Anweisungen im Programm ist dieser Speicherbereich aber ein sog. „Random Access“-Speicher (Speicher mit wahlfreiem Zugriff), an dessen Zellen man direkt ungehindert herankommt. Dass man hier dennoch von einem Stack spricht, rührt von der Tatsache her, dass dieser Speicherbereich nur durch PUSH-Operationen der Speicherverwaltung wächst und nur durch POP-Operationen der Speicherverwaltung schrumpft.

Im Stack liegen ausschließlich Zellen, denen der Programmierer in seinem Programm einen Namen gegeben hat. Es gibt nur zwei Arten von Zellen, die der Programmierer benennt, nämlich die deklarierten Variablen und die formalen Variablen.

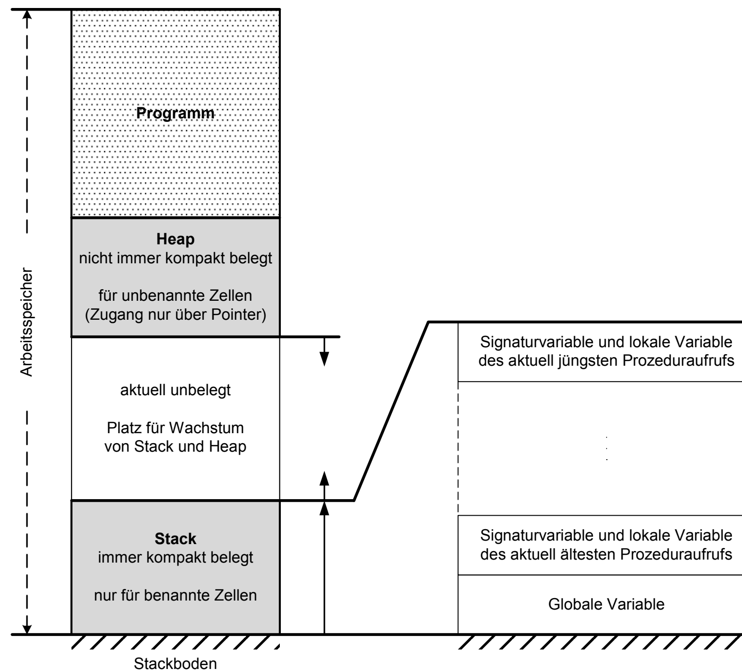

Bild 53 Adressraumaufteilung für die Speicherverwaltung

Da im voranstehenden Satz die Wörter „Zelle“ und „Variable“ nebeneinander vorkommen, erscheint es angebracht, einige Betrachtungen zum Variablenbegriff anzustellen.

Der Begriff „Variable“

Der Begriff Variable gehört in die Formelwelt der Mathematik. Die Mathematik ist die Welt der statischen Sachverhalte, in der die Dinge so sind wie sie sind, in der aber nichts entsteht und nichts verschwindet. So kann man in der Mathematik nicht sagen: $2+5$ wird 7. Man kann nur sagen: $2+5$ ist 7. In der Mathematik gibt es die Funktionen, aber es gibt nicht den Vorgang der Berechung eines Funktionswerts zu einem gegebenen Argument. Die Berechnung ist ein Geschehen, und in der Mathematik gibt es kein Geschehen. Die Vorstellung einer Zeitachse und des Paares vorher/nachher gehört nicht in die Mathematik, sondern in die Systemtechnik.

Formeln sind sprachliche Gebilde, in denen irgendwelche mathematischen bzw. logischen Sachverhalte ausgedrückt werden. Diese Formeln sind Strukturen aus Symbolen. Anstelle des Wortes Symbol können wir auch das Wort Bezeichner benutzen. Symbole oder Bezeichner sind optisch oder akustisch wahrnehmbare Muster, denen eine Mensch oder eine Gruppe von Menschen in einem willkürlichen definitorischen Akt jeweils eine Bedeutung zugeordnet haben.

Ein Bezeichner bezeichnet entweder eine Konstante oder eine Variable. Eine Konstante oder ein Wert darf keinesfalls mit dem Begriff Zahl gleichgesetzt werden. Wenn wir in einer For-

mel, also in einem sprachlichen Ausdruck eine Konstante oder einen Wert bezeichnen, heißt das lediglich, dass wir uns festgelegt haben bezüglich des Gegenstandes, über den wir reden wollen. Beispielsweise könnten wir über die Zahl 5 oder über Johann Wolfgang von Goethe oder über die Sinusfunktion reden wollen. Alle drei genannten Gegenstandsbeispiele sind Konstante bzw. Werte in unserem Sprachgebrauch.

Wenn wir dagegen Aussagen machen wollen, in denen der Gegenstand, über den wir reden wollen, noch in gewisser Weise offen gelassen ist, dann verwenden wir Variable. Mit einer Variable muss immer ein Wertebereich verbunden sein. Der Wertebereich einer Variablen ist die Menge der Konstanten bzw. Werte, die als mögliche Belegung der Variablen in Betracht kommen. So könnte ich beispielsweise sagen: Wir betrachten eine Funktion f(x). Hier habe ich mich nicht auf eine ganz bestimmte Funktion festgelegt, sondern habe offen gelassen, welche Funktion gemeint sein könnte. Später könnte ich dann sagen: Jetzt nehmen wir an, f sei die Sinusfunktion. Dann habe ich die Variable f mit der Konstanten „Sinusfunktion“ belegt.

Es ist wichtig zu erkennen, dass ein Wertebereich selbst in der Rolle eines Wertes vorkommen darf. So könnte ich beispielsweise sagen: Wir betrachten zwei Variable x und y, die beide den Wertebereich W haben sollen. In diesem Fall ist W eine Variable, deren Wertebereich die Menge der Wertebereiche ist. Später könnte ich dann sagen: Wir betrachten jetzt den Fall, dass W der Wertebereich `INTEGER` ist. In diesem Fall habe ich die Variable W mit dem Wert `INTEGER`, also mit einem ganz bestimmten endlichen Ganzzahlenbereich belegt. Ich habe W nicht mit einer konstanten ganzen Zahl belegt, sondern mit einer Menge ganzer Zahlen, die eindeutig definiert ist und deshalb als Konstante bezeichnet werden darf.

Nun kommen wir zur Betrachtung des Begriffs „Variable“ im Bereich der Systemtechnik. Im Bereich der Systemtechnik reden wir immer über dynamische Systeme, die einen Aufbau haben und worin sich ein Geschehen abspielt. Wir finden in diesem System Komponenten, die wir als Akteure, also als Verursacher des Geschehens betrachten, und Orte, die wir als Aktionsfelder bezeichnen, an denen wir das Geschehen beobachten können. Es hat sich eingebürgert, im Bereich der Systemtechnik das Wort „Variable“ für etwas zu benutzen, was eigentlich ein Zwitter aus einer Variablen und einer Konstanten ist. In der Systemtechnik brauchen wir Bezeichner zur Identifikation von Speicherzellen. Diese Speicherzellen sind Behälter, in denen sich zeitlich nacheinander unterschiedliche Werte befinden können. Der Bezeichner zur Identifikation einer Speicherzelle im System bezeichnet zweifellos eine Konstante, denn wir haben uns ja mit dieser Bezeichnung festgelegt, über welche Speicherzelle wir reden wollen. Nun wollen wir aber nicht nur über die Speicherzelle reden, sondern auch über die Werte, die in dieser Speicherzelle liegen können. Wenn wir nun eine Aussage über diese Werte machen wollen, ohne dass diese Aussage einen ganz bestimmten Wert betrifft, müssen wir eine Variable benutzen, deren Wertebereich die Menge aller möglichen Inhalte der betrachteten Speicherzelle ist. Für diese Variable nimmt man nun den gleichen Bezeichner, den man zur Identifikation der Speicherzelle benutzt hat. Dieser Bezeichner bezeichnet also eine Konstante, wenn wir ihn als Identifikator der Speicherzelle betrachten, und er bezeichnet eine Variable, wenn wir ihn mit Elementen aus dem Wertebereich der möglichen Inhalte der Speicherzelle belegen können.

Im folgenden wird der Begriff Variable im systemtechnischen Sinne benutzt.

Globale und lokale Varaible

Die nun folgenden Überlegungen stelle ich an, um Sie zum Begriff des abstrakten Datentyps zu führen. Voraussetzung hierfür ist das Wissen um den Unterschied zwischen globalen und lokalen Variablen. Eine globale Variable steht für eine Speicherzelle, die während des gesamten Programmlaufs existiert und die überall im Programmtext angesprochen und benutzt werden kann. Im Stack liegen die globalen Variablen ganz unten (siehe Bild 53). Globale Variable müssen ganz am Anfang des Programmtextes eingeführt werden, damit sie weiter hinten bekannt sind (siehe Bild 54).

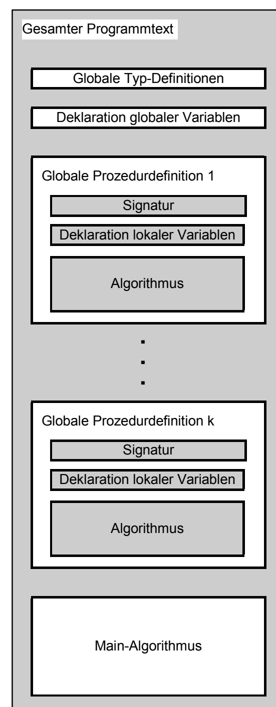

Bild 54 Programmstruktur mit globalen und lokalen Variablen

Lokale Variable dagegen werden in Prozedurdefinitionen eingeführt. Sie stehen für Speicherplätze, die erst mit dem Aufruf der jeweiligen Prozedur geschaffen werden und die nach Ablauf der Prozedur wieder verschwinden. Außerhalb der Prozedur können solche lokalen Variablen also nicht angesprochen und verwendet werden. Man kann sich anschaulich vorstellen, der Ablauf der Prozedur entspreche der Zubereitung eines Griesbreis, wofür man einen Kochtopf benötigt. Dieser Kochtopf entspricht dem lokalen Speicherplatz. Er wird zu Beginn der Kochprozedur bereitgestellt, kann während der Kochprozedur benutzt werden und wird am Ende der Kochprozedur wieder weggenommen.

Die lokalen Variablen liegen im Stack über den globalen Variablen. Pro Prozeduraufruf, zu dem noch kein Return erfolgt ist, liegen im Stack sowohl die lokalen Variablen als auch die aktuell belegten formalen Variablen, die der Programmierer in der Signatur benannt hat. Es kann sehr wohl sein, dass es im Stack zu einer Bezeichnung gleichzeitig unterschiedliche Zellen gibt. Beispielsweise könnte der für die globalen Variablen zuständige Programmierer eine mit i bezeichnete lokale Variable deklariert haben:

```
INTEGER i;
```

Gleichzeitig aber könnte ein Prozedurprogrammierer die gleiche Deklaration in sein Programm schreiben und damit eine lokale Variable einführen. Die Prozedur könnte außerdem noch rekursiv sein, so dass mehrere Aufrufe dieser Prozedur erfolgen können, bevor der erste Return auftritt. Dann liegen im Stack zu einem bestimmten Zeitpunkt mehrere Zellen, die alle die Bezeichnung i tragen. Wenn nun eine Anweisung ausgeführt werden soll, welche eine Veränderung des Inhalts einer Zelle i verlangt, beispielsweise

```
i:=i+2;
```

muss wohldefiniert sein, welche der verschiedenen Zellen i im Stack denn nun betroffen sein soll. Von bestimmten – eher exotischen – Fällen in bestimmten Programmiersprachen abgesehen gilt die Regel, dass immer die Zelle betroffen ist, die am nächsten beim TOP liegt. Der Programmabwickler muss also immer vom TOP ausgehend in Richtung zum Boden hin nach einer Zelle suchen, die den gegebenen Namen trägt.

Abstrakte Datentypen

Wir werden nun eine Aufgabenstellung betrachten, bei der wir gerne die Eigenschaften einer globalen Variablen mit den Eigenschaften einer lokalen Variablen verbinden würden.
Wir betrachten die Programmierung eines sogenannten Zufallszahlengenerators. Wir stellen uns vor, wir wollten im Hauptprogramm die Ausgabe einer 50-elementigen Folge von Zufallszahlen aus dem Ganzzahlenbereich 1 bis10 programmieren. Dies könnte wie folgt aussehen:

```
FOR i=1 STEP 1 UNTIL 50 DO PRINT( zufallszahl() );
```

Der Algorithmus zur Zufallszahlengenerierung soll hier nicht im Detail behandelt werden. Ich will Ihnen lediglich das Grundprinzip vorstellen:

Wir führen eine globale Integervariable `zustand` ein, die als 32-stelliges Binärwort den Wertebereich

$$-2^{31} \leq \texttt{zustand} \leq 2^{31}-1$$

hat und die wir bereits bei ihrer Deklaration mit einem Initialwert belegen.
Wir stellen uns nun eine Überführungsfunktion `folgezustand(z)` vor, die jeder Zahl aus diesem Wertebereich eine andere Zahl zuordnet, wobei erst nach 2^{32} Wiederholungen der Anweisung `zustand:= folgezustand(zustand)` ein bereits einmal vorgekommener Zustandswert das nächste Mal vorkommt. Der Ergebniswertebereich dieser Funktion ist also gleich ihrem Argumentwertebereich.
Daneben stellen wir uns eine zweite Funktion `ergebnisextraktion(z)` vor, die jeder Zahl aus dem großen Wertebereich eine Zahl aus dem Wertebereich 1 bis10 zuordnet. Mit diesen beiden Funktionen können wir die Funktionsprozedur `zufallszahl()` wie folgt schreiben:

```
INTEGER  zufallszahl();
{
 zustand:= folgezustand(zustand);
 RETURN ergebnisextraktion(zustand);
}
```

Man erhält auf diese Weise selbstverständlich keinen echten Zufallsprozess. Man spricht von einer „Pseudozufallsfolge“.

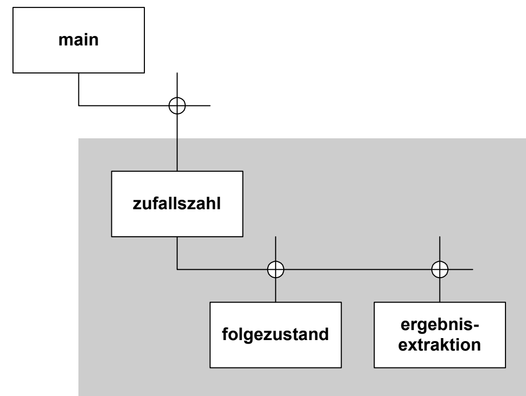

Bild 55 Aufrufschichtung im betrachteten Beispiel

Wenn wir die Variable `zustand` nicht als globale Variable, sondern als lokale Variable der Zufallsprozedur eingeführt hätten, könnte sich kein Zufallseffekt ergeben, denn dann würde die Funktionsprozedur `zufallszahl()` bei jedem Aufruf den gleichen Wert zurückliefern.

Die Notwendigkeit, die Variable `zustand` als globale Variable einführen zu müssen, sollte uns aber nicht gefallen, denn diese Variable gehört ja ganz fest zu der Prozedur `zufalls-zahl()` und sollte nur innerhalb dieser Prozedur verwendet werden. An Hand dieses Beispiels sind wir also auf einen Fall gestoßen, wo es wünschenswert ist, Prozeduren mit Variablen zu einer höheren Einheit zu verbinden, ohne dass diese Variablen lokale Variablen der Prozedur werden. Wir benötigen also in den Programmiersprachen zur Formulierung einer solchen Kapselung eine neue Art von Klammern, in die hinein wir die Variablen und die Prozeduren schreiben können. Eine derart geklammerte programmiersprachliche Einheit wird als abstrakter Datentyp (Englisch: abstract data type, abgekürzt ADT) bezeichnet.

Stack als Exemplar eines Abstrakten Datentyps

Das Standardbeispiel in den Lehrbüchern zur Einführung abstrakter Datentypen ist der Stack. Wir betrachten den abstrakten Datentyp `STACK OF INTEGER`.
Und wir stellen uns vor, dass wir in unserem System drei Exemplare solcher Stacks benötigen, die wir wie folgt deklarieren:

```
s1, s2, s3 : STACK OF INTEGER;
```

Jeder der drei Stacks `s1`, `s2` und `s3` ist eine Speicherstruktur, die im Programmtext außerhalb der ADT-Definition nur unter Verwendung der fünf Prozeduren `CLEAR`, `PUSH`, `POP`, `TOP` und `HIGHT` angesprochen werden kann. Für den Programmierer, der eine solche Prozedur verwendet, ist es völlig irrelevant, wie die Speicherstruktur zur Realisierung des konkreten Stacks aussieht. Das Wissen um die konkrete Speicherstruktur für den Stack wird nur von dem Programmierer benötigt, der die fünf Stack-Prozeduren innerhalb der ADT-Definition schreiben muss. Wir betrachten hierzu zwei alternative Datenstrukturen zur Realisierung eines `STACK OF INTEGER`.

Ein `STACK OF INTEGER` könnte dadurch realisiert werden, dass man ein `ARRAY OF INTEGER` anlegt und in einer zusätzlichen `INTEGER`-Variablen den jeweils aktuellen Index des TOP-Elements speichert (siehe Bild 56).

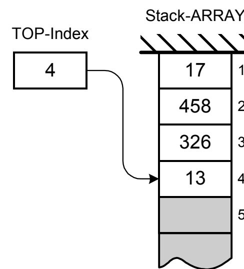

Bild 56 Implementierung eines Stacks als ARRAY

Bei dieser Realisierung eines Stacks im Arbeitsspeicher liegen beide Teile des Stacks, also sowohl die Zelle für den TOP-Index als auch das Array für die Stack-Elemente im Speicherbereich „Stack“ des Bildes 53, weil es sich bei diesen beiden Teilen des Stacks um benannte Speicherbereiche handelt.
Die Programme für die ADT-Prozeduren sind bei dieser Stackrealisierung recht einfach:

```
void CLEAR();
{
 topindex:=0;
}
```

```
void PUSH(INTEGER i);
{
 topindex:= topindex+1;
 stackarray[topindex]:= i;
}
```

```
void POP();
{
 topindex:= topindex-1;
}
```

```
INTEGER TOP();
{
 RETURN stackarray[topindex];
}
```

```
INTEGER HEIGHT();
{
 RETURN topindex;
}
```

Bild 57 Zugriffsprozeduren für die Stackimplementierung gemäß Bild 56

Es wurde hier darauf verzichtet, die unbedingt zu beachtenden Ausnahmefälle in die Programme einzubringen. Diese Ausnahmefälle bestehen darin, dass die Prozeduren POP und TOP bei leerem Stack nicht aufgerufen werden sollten. Da dies aber nicht verhindert werden kann, muss man innerhalb der Prozeduren diesen Sonderfall abfangen. Eine weitere Ausnahmesituation besteht darin, dass man ja nicht unendlich viel Speicherplatz zur Verfügung hat und man deshalb mit der Möglichkeit rechnen muss, dass ein PUSH-Aufruf in einer Situation erfolgt, wo das vorhandene Array bereits voll ist. Auch diesen Ausnahmefall muss man in der Prozedur berücksichtigen, wenn sie robust sein soll.

In Bild 58 ist eine völlig andere Möglichkeit zur Realisierung eines Stacks als Speicherstruktur gezeigt. In diesem Fall wird mit Pointern gearbeitet, und jedes Stackelement ist eine zweiteilige Struktur aus einem Integer und einem Pointer. Der Pointer zeigt jeweils auf das darunter liegende Stackelement. Deshalb ist die Pointerzelle des zuunterst liegenden Stackelements leer, was sich in der Beschriftung NIL äußert.

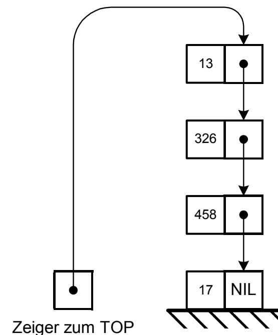

Bild 58 Stackimplementierung als Pointerstruktur

Bei dieser Stackrealisierung ist nur die Zelle für den Zeiger zum TOP eine benannte Zelle und liegt im Speicherbereich „Stack“ des Bildes 53. Die Zellen für die Stackelemente sind dagegen unbenannt und liegen deshalb in der Heap.

In diesem Fall sehen die Prozedurprogramme ganz anders aus als im Falle der Stackrealisierung nach Bild 56. Als Beispiel sei die PUSH-Prozedur betrachtet.

```
void PUSH(INTEGER i);
{STACKELEMENT *zeigerpuffer;
 {zeigerpuffer:= NEW(STACKELEMENT);
  zeigerpuffer.zahlenfeld:= i;
  zeigerpuffer.pointerfeld:= topzeiger;
  topzeiger:= zeigerpuffer;}
}
```

Der Programmierer, der die Prozeduren außerhalb der ADT-Definition benutzt, braucht nicht zu wissen, ob eine Struktur nach Bild 56 oder nach Bild 58 oder noch eine ganz andere Struktur zur Stackrealisierung gewählt wurde. Für ihn genügt es zu wissen, dass durch die PUSH-Operation ein Element auf den Stack gelegt wird und durch die POP-Operation wieder eins herunter genommen wird. Weil also hier von der konkreten Speicherstrukturierung abstrahiert wird, ist die Bezeichnung abstrakter Datentyp sehr zutreffend. Durch diese Abstraktion wird die Portabilität der Programme gefördert und es wird die Arbeitsteilung bei der Programmierung erleichtert. Portabilität bedeutet, dass die verwendenden Programmteile unverändert beibehalten können, auch wenn die verwendeten abstrakten Datentypen durch andere Speicherstrukturen realisiert werden.

Portabilität

Mit der Portabilität kann man folgende Anschauung verbinden: Lokomotiven fahren auf Schienen, d.h. die Lokomotiven verwenden den Gleisunterbau, den man in diesem Fall als Plattform bezeichnen könnte. Damit die Verwendung einwandfrei möglich ist, müssen sowohl dem Konstrukteur der Lokomotive als auch dem Konstrukteur des Gleisunterbaus gewisse Normen bekannt sein, an die sie sich halten müssen. In diesem Fall ist es der Schienenabstand und die Form der Laufflächen. Dagegen ist die Frage, wie die Schienen auf dem Gleisbett montiert sind, für den Konstrukteur der Lokomotive irrelevant oder sie sollte zumindest irrelevant sein. Wenn der Konstrukteur der Lokomotive auf die Idee käme, den Geschwindig-

keitsmesser dadurch zu realisieren, dass er einen bestimmten Schwellenabstand als gegeben voraussetzt, den er dann zur Geschwindigkeitsbestimmung nutzt, würde er die Portabilität seiner Lokomotive verhindern. Denn wenn er die Geschwindigkeit dadurch bestimmt, dass er feststellt, wie viele Schwellen pro Zeiteinheit überfahren werden, setzt er voraus, dass es überhaupt Schwellen gibt und dass diese einen ganz bestimmten Abstand haben. Damit bindet er das Funktionieren seines Geschwindigkeitsmessers an bestimmte Eigenschaften der Plattform. Entsprechend sollte ein Programmierer immer darauf bedacht sein, keine Informationen zu benutzen, die der Plattformkonstrukteur in freier Entscheidung ändern dürfen sollte.

“Eingebaute” Abstrakte Datentypen

Es ist durchaus eine sinnvolle Frage, ob `INTEGER` ein in die Sprache eingebauter abstrakter Datentyp ist. Es gibt höhere Sprachen, wo dies der Fall ist, und andere, wo dies nicht der Fall ist. Woran erkennt man das? Ein abstrakter Datentyp ist daran erkennbar, dass der Programmierer, der Exemplare eines solchen Typs benutzt, diese Exemplare nur wesensgerecht benutzen kann und kein Codierungswissen bei der Benutzung auswerten kann. Die in dieser Aussage verwendeten Begriffe „wesensgerecht“ und „Codierungswissen“ werden im folgenden erklärt.

Am Beispiel des Typs `INTEGER` kann leicht erklärt werden, was eine wesensgerechte Benutzung darstellt. `INTEGER` kennzeichnet einen Wertebereich, der ausschließlich ganze Zahlen innerhalb eines bestimmten Intervalls umfasst. Alles, was man mit ganzen Zahlen machen kann, stellt eine wesensgerechte Benutzung dar. Man kann also die arithmetischen Operationen ausführen, man kann fragen, ob die Zahl gerade oder ungerade sei, man kann fragen, ob es sich um eine Primzahl handelt, und so weiter. In der Maschine aber sind die Exemplare vom Typ `INTEGER` bereits Ergebnisse einer Interpretationskette, die von bestimmten physikalischen Mustern an bestimmten Orten ausgeht. So wird beispielsweise einem bestimmten Verlauf der magnetischen Feldstärke auf einem magnetischen Medium eine Folge von Binärsymbolen zugeordnet, die in bestimmten Umfange abgepackt Binärwörter darstellen, denen man dann durch eine bestimmte Interpretation eine ganze Zahl oder eine Gleitkommazahl oder sonst etwas zuordnen kann. Während die Interpretationsschritte vom physikalischen Muster wegführen zu immer höheren Interpretationsergebnissen, stellt die Codierung den umgekehrten Weg dar. So ist beispielsweise der Schritt von der Integerzahl zum zugehörigen Binärwort ein Codierungsschritt.

Wenn man nun die Codierungsregeln kennt, kann man ein gegebenes `INTEGER`-Exemplar auch als Binärwort betrachten, auf das man die Operationen anwenden kann, die für Binärwörter gelten. So könnte man beispielsweise schreiben

$$5 \mathbin{\&} 6 = 4,$$

wobei man mit dem &-Operator nicht die Addition meint, denn die hätte ja das Ergebnis 11 geliefert. Man hat hier vielmehr die Vorstellung von zwei Binärwörtern, die stellenweise logisch UND-verknüpft werden:

```
 00101 = 5
 00110 = 6
----------
 00100 = 4
```

Es gibt Programmiersprachen, da kann der Programmierer kein Codierungswissen bezüglich der Integerzahlen nutzen, so dass er in diesen Fällen keine Integerexemplare logisch UND-verknüpfen könnte. Hier ist `INTEGER` ein abstrakter Datentyp. Es gibt aber auch Programmier-

sprachen, in denen kann der Programmierer logische Verknüpfungen auf Integerzahlen verlangen. In diesem Fall ist `INTEGER` kein abstrakter Datentyp.

Bei der Computerkonstruktion werden nur verhältnismäßig wenige Interpretationsregeln bzw. Codierungsvorschriften festgelegt und eingebaut. Es sind dies die Interpretationsregeln bzw. Codierungsvorschriften für die `INTEGER` -Zahlen, die `REAL` -Zahlen und die alphanumerischen Zeichen. Ein wesentlicher Anteil bei der Gestaltung von Software besteht darin, Codierungsvorschriften bzw. Interpretationsregeln festzulegen, die von den in der Programmiersprache bzw. in der Computerhardware vorgegebenen Datentypen zu beliebigen informationellen Wertebereichen führen. So ist es beispielsweise keine Festlegung des Programmiersprachengestalters, dass Integerzahlen als Personalnummern verwendet werden. Dies entscheidet der Programmierer, der Personalnummern benötigt. In diesem Beispiel gibt es also eine Interpretation, die von einer `INTEGER-Zahl` zu einer Personenidentifikation führt. `PERSON` als abstrakter Datentyp ist nur dann gegeben, wenn man auf die Personen-identifikation nicht die Operationen anwenden kann, die für Integerzahlen bekannt sind. So ergibt es beispielsweise keinen Sinn zu verlangen, zu einer Personenidentifikation eine bestimmte ganze Zahl hinzu zu addieren, denn mit dem Ausdruck „Herr Albert Krause + 17“ kann man keine sinnvolle Vorstellung verbinden.

Ein abstrakter Datentyp ist nur dann zweckmäßig konzipiert, wenn es leicht fällt, das Wesen der Exemplare zu beschreiben. Man muss sich also fragen: Was könnte ich alles über ein Exemplar wissen wollen, von dem ich weiß, dass es von diesem Typ ist? Und was könnte ich alles mit diesem Exemplar machen wollen? Man denke wieder an den abstrakten Datentyp `STACK`. Hier könnte man bezüglich eines aktuellen Exemplars wissen wollen, wie viele Elemente im Stack liegen, welche Elemente dies sind und wie sie von unten nach oben geordnet sind. Zur Stackveränderung stehen nur drei Möglichkeiten zur Verfügung. Man kann verlangen, dass der ganze Stack geleert wird, oder dass ein neues Element oben drauf gelegt wird oder dass das oberste Element weggenommen wird. Damit ist alles gesagt, was zum Wesen des Stacks gehört.

Sie erinnern sich sicher daran, dass beim Herannahen des Übergangs vom Jahr 1999 zum Jahr 2000 viel über das Softwareproblem der Datumsdarstellung berichtet wurde. Dieses Problem bestand darin, dass noch sehr viel Software im Einsatz war, bei der die Jahresangaben in den Datumsstrukturen nur mit zwei Dezimalstellen codiert waren, weil man glaubte, die Jahreszahl immer dadurch gewinnen zu können, dass man implizit die Ziffernfolge 19 voranstellt. Es wäre sinnvoll gewesen, den Datentyp `DATUM` als abstrakten Datentyp einzuführen, denn dann wäre die Codierungsentscheidung gekapselt gewesen und hätte sich an keiner Stelle außerhalb der Definition des abstrakten Datentyps ausgewirkt.

Es gilt ganz allgemein, dass man Codierungsentscheidungen möglichst kapseln sollte, damit sich eine Nutzung des Codierungswissens nicht über die ganze Software verteilt.