Zahlentypen in Computern

In unseren ersten Übungsbeispielen geht es um arithmetische Algorithmen. Deswegen soll nun kurz auf die unterschiedlichen Zahlentypen in Computern eingegangen werden. In der Mathematik ist es zulässig, die ganzen Zahlen als echte Teilmenge der reellen Zahlen zu betrachten; in der Softwarewelt ist diese Betrachtung unzulässig. In der Softwarewelt sind die ganzen Zahlen keine Teilmenge der reellen Zahlen. In der Softwarewelt kommen die reellen Zahlen gar nicht vor, obwohl es den numerischen Datentyp `REAL` gibt. Jede Zahl vom Typ `REAL` kann zwangsläufig nur mit einer endlichen Anzahl von Bits kodiert sein, und deshalb ist jede Zahl vom Typ `REAL` eine rationale Zahl, d.h. eine Zahl, die als Bruch zweier ganzer Zahlen darstellbar ist. Die ganzen Zahlen, deren Typbezeichnung in den Programmiersprachen `INTEGER` ist, werden im Rechner durch andere Binärfolgen dargestellt als die `REAL`-Zahlen gleichen Werts. Die `INTEGER`-Zahl 15 sieht also rechnerintern anders aus als die `REAL`-Zahl 15.0. Deshalb sollte man auch nie eine `REAL`-Zahl schreiben, ohne einen Punkt hinter den ganzzahligen Anteil zu setzen, auch wenn hinter dem ganzzahligen Anteil nur noch Nullen folgen.

Heutzutage gilt allgemein, dass `INTEGER`-Zahlen rechnerintern als Binärwörter codiert werden, die als zweierkomplementcodierte ganze Zahlen zu interpretieren sind. In Bild 16 wird eine Speicherzelle für m-stellige Binärwörter betrachtet.

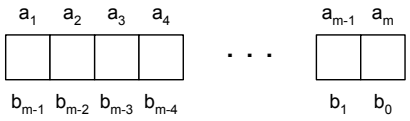

Bild 16 Speicherzelle für ein m-stelliges Binärwort

Die m Binärstellen sind auf zwei unterschiedliche Arten nummeriert. Die naheliegendste Nummerierung entspricht der Zählung der Komponenten eines Vektors. Es wird von links nach rechts von 1 bis m gezählt. Es kommt aber auch vor, dass von rechts nach links gezählt wird, wobei die Zählung bei 0 beginnt und bei m-1 endet. Diese von der gewohnten Vektorzählung abweichende Zählung kann nur begründet werden, wenn das Binärwort als codierte ganze Zahl interpretiert werden soll. In diesem Fall hat die rechtsaußenstehende Binärstelle das Stellengewicht 2^0. Wenn das m-stellige Binärwort nur mit nichtnegativen ganzen Zahlen belegt werden soll, erhält die linksaußenstehende Binärstelle das Gewicht 2^{m-1}. Dagegen wird das Gewicht dieser Stelle im Falle von zweierkomplementcodierten ganzen Zahlen auf -2^{m-1} festgelegt. Aus dem Binärwort, welches als zweierkomplementcodierte ganze Zahl zu interpretieren ist, erhält man dann den Zahlenwert durch die Formel

$$- b_{m-1} * 2^{m-1} + \sum_{i=0}^{i=m-2} b_i * 2^i$$

Die linksaußenliegende Binärstelle ist in diesem Fall die einzige mit einem negativen Gewicht. Daraus kann man folgen, dass diese Stelle als Vorzeichen der Zahl betrachtet werden darf, wobei eine Eins ein negatives Vorzeichen bedeutet.

Beispielsweise nehmen wir an, das Binärwort sei 10-stellig, d.h. es gelte m=10. Wenn wir nun die Zahl -4 als 10-stelliges Binärwort codieren wollen, dann erhalten wir von der linksaußenstehenden Stelle einen Beitrag von $-2^9 = -512$. Dies muss ergänzt werden durch einen positiven Wert von +508, der sich als Summe der gewichteten restlichen 9 Binärstellen ergeben muss. Wir erhalten

$$-4 = -512 + 508 = -512 + (256 + 128 + 64 + 32 + 16 + 8 + 4)$$

-4 entspricht also dem Binärwort 1111111100.

Allgemein gilt, dass man in m-stelligen Binärwörtern, die man als zweierkomplementcodierte ganze Zahlen interpretiert, den Zahlenbereich

$$-2^{m-1} \le i \le +2^{m-1} - 1$$

unterbringen kann. Im Falle des Bytes, welches ein 8-stelliges Binärwort ist, ergibt sich also der Wertebereich zu $-128 \le i \le +127$.

Man muss sich bewusst sein, dass man in einem Programm, worin man eine Speicherzelle nur durch die Angabe `INTEGER` typisiert hat, keine vollständige Information über den Wertebereich dieser Speicherzelle festgelegt hat. Denn es hängt von der aktuellen Entscheidung bei der Programmübersetzung ab, mit wie viel Stellen die Speicherzelle realisiert wird. Die vollständige Angabe würde neben der Festlegung `INTEGER` auch noch erfordern, dass man die Länge des verwendeten Binärwortes mit angibt. In machen Programmiersprachen ist festgelegt, dass mit `SHORT_INTEGER` eine Binärwortlänge von 16 gemeint ist, mit `INTEGER` eine Binärwortlänge von 32 und mit `LONG_INTEGER` eine Binärwortlänge von 64.

Neben dem Typ `INTEGER` gibt es auch den Zahlentyp `REAL`, der in manchen Programmiersprachen auch `FLOAT` heißt. Es wurde schon gesagt, dass dieser Zahlentyp auf rationale Zahlen beschränkt ist, die als Bruch mit einem ganzzahligen Zähler und mit einem ganzzähligem Nenner gedacht werden können. Das Wort Gleitkommazahl ist die deutsche Bezeichnung für die `REAL`- bzw. `FLOAT`-Zahlen. Wenn ein Binärwort als Gleitkommazahl interpretiert wird, wird dieses Binärwort in zwei Abschnitte zerlegt, wobei jeder der beiden Abschnitte als zweierkomplementcodierte ganze Zahl betrachtet wird. Der eine Abschnitt wird Mantisse m genannt, der andere Exponent e. Als Wert, der dem Binärwort zuzuordnen ist, gilt

$$\text{Gleitkommazahlenwert} = \text{Mantisse} * 2^{\text{Exponent}}$$

Die Tatsache, dass heute in den Rechensystemen die Zahlen nicht mehr durch binärcodierte Dezimalziffern dargestellt werden, sondern als Zahlen im Dualsystem, führt dazu, dass eine Zahl nicht unbedingt exakt so wieder ausgegeben werden kann wie sie eingegeben wurde. Als Beispiel wird gezeigt, wie die Dezimalzahl 1,3 als Gleitkommazahl codiert wird, wobei hier der Einfachheit halber sehr kurze Wortlängen für die Mantisse (10) und den Exponenten (5) gewählt werden. Der Benutzer wird bei der Eingabe tatsächlich nur die Tastenfolge 1,3 erzeugen. Aber aus dieser Tastenfolge wird intern der Bruch 333:256 erzeugt, der sich schreiben lässt als $333 * 2^{-8}$. Die Mantisse hat also den Wert 333 und der Exponent hat den Wert –8. Hierzu gehört das 15-stellige Binärwort

0101001101 11000

Wenn nun anschließend eine Ausgabe dieser gespeicherten Zahl verlangt wird, dann wird wieder eine Dezimalziffernfolge berechnet, aber diese heißt nicht 1,3, sondern 1,30078125 .

Speicherzellen und Zuweisungsoperationen

Die Speicherzellen bilden die Grundlage jeglicher imperativen Programmierung. Die Zuweisungsanweisungen sind also der Kern jedes imperativen Programms. Eine Zuweisungsanweisung verlangt, dass in die Speicherzelle, die durch den links vom Zuweisungsoperators stehenden Ausdruck identifiziert wird, ein Inhalt eingebracht wird, der durch den rechts vom Zuweisungsoperator stehenden sprachlichen Ausdruck identifiziert wird.

Beispiele:

```
i := 5;
x := 16.3 - SQRT(18.3);
y := x - 4.2;
```

Wenn in einer Zuweisungsanweisung im sprachlichen Ausdruck, der den einzubringenden Inhalt beschreibt, Bezeichner von Speicherzellen vorkommen – wie beispielsweise x auf der rechten Seite der dritten Beispielsanweisung - , dann ist dies zu lesen als „Aktueller Inhalt der Speicherzelle x“.

Da in den beiden ersten Übungen der gleiche Algorithmus vorgegeben ist, und der Unterschied der vorgegebenen Programme nur darin besteht, dass in der Übung 1 das Programm in der Sprache FORTRAN und in der Übung 2 das Programm in der Sprache C formuliert ist, kann man diese beiden Programmtexte nebeneinanderlegen und vergleichen. Dabei stellt man fest, dass die Unterschiede verhältnismäßig gering sind. Insbesondere sehen die algorithmischen Teile für die arithmetischen Formeln fast identisch aus.

Sowohl in der Sprache FORTRAN als auch in der Sprache C gibt es Anweisungen der Form:

```
Anzahl = Anzahl + 1
```

Wer nicht mit den Besonderheiten von Programmiersprachen vertraut ist, wird diese Programmzeile gar nicht als Anweisung deuten können, sondern er wird sie als unsinnig einstufen. In mathematischer Sicht könnte diese Zeile nur dann als sinnvoll gedeutet werden, wenn `Anzahl` eine unendliche Zahl ist, denn nur unendliche Zahlen bleiben unverändert, wenn man eins darauf addiert.

In diesem Sinne, dass `Anzahl` eine unendliche Größe sei, ist die Zeile aber gar nicht gemeint. Vielmehr wird hier das symmetrische Gleichheitszeichen verwendet, obwohl eine unsymmetrische Bedeutung damit verbunden werden soll. Die Programmzeile soll nämlich gedeutet werden als

```
Anzahl(nachher) = Anzahl(vorher) + 1
```

In dieser vervollständigten Schreibweise hat man überhaupt keine Probleme mehr zu erkennen, was verlangt wird: Es soll der bisherige Wert der Variablen `Anzahl` um 1 erhöht werden. Das Wort `Anzahl` ist in diesem Fall die Bezeichnung einer Speicherzelle, in der eine ganze Zahl gespeichert werden kann. Und die Anweisung verlangt, dass man zuerst einmal in die Speicherzelle hineinschaut, den dortigen Wert nimmt, im Kopf eine Eins darauf addiert und das Ergebnis an Stelle der bisher dort gespeicherten Zahl in diese Speicherzelle schreibt.

Da man nicht immer die zusätzlichen Angaben `vorher` bzw. `nachher` zu den Variablenbezeichnern hinzufügen will, ist es sinnvoller, anstelle des Gleichheitszeichens ein unsymmetrisches Symbol zu benutzen. In etlichen Programmiersprachen wurde als sogenannter Zuweisungsoperator die Zeichenfolge := festgelegt.

Nun gehe ich noch einmal auf die Zuweisungen ein, die in den beiden Programmtexten praktisch gleich aussehen. Und ich möchte noch einmal betonen, wie irreführend die Verwendung des symmetrischen Gleichheitszeichens in einer Zuweisung ist. Stellen Sie sich vor, ich würde in einem Text den Satz schreiben: „Wendt ist einen Meter größer als Wendt.“ Dieser Satz könnte von niemandem als sinnvoller Satz akzeptiert werden. Wenn ich dagegen schreiben würde: „Wendt im Jahr 2001 ist einen Meter größer als Wendt im Jahre 1944“, dann hätte überhaupt niemand Schwierigkeiten beim Verständnis dieses Satzes.

Die historische Entwicklung bezüglich des Zuweisungsoperators war wie folgt: Die erste Programmiersprache war FORTRAN. Sie wurde im Jahre 1957 eingeführt, und als Zuweisungsoperator wurde = festgelegt. Im Jahre 1960 kam die Programmiersprache ALGOL; hier wurde die Unsymmetrie des Zuweisungsoperators explizit zum Ausdruck gebracht, indem man zu schreiben hatte : = . Später kamen dann wieder Programmiersprachen, die als Zuweisungsoperator das Gleichheitszeichen benutzten.

Man muss vermuten, dass die Programmiersprachenerfinder den Programmierern das Drücken von Tasten ersparen wollten. Nun kommt es aber nicht darauf an, dem einzelnen Programmierer die Zahl der zu drückenden Tasten zu minimieren, sondern es kommt darauf an, dafür zu sorgen, dass ein einmal geschriebenes Programm auch nach verhältnismäßig langer Zeit noch für den Leser verständlich ist, egal ob dieser Leser der Programmierer selbst ist oder eine andere Person. Programmiersprachen sollten nicht für Programmierer entwickelt werden, die für sich alleine irgendwelche Programme schreiben, von denen sie sicher sein können, dass sie nie von jemand anderem gelesen werden, und bei denen es ihnen auch nicht wichtig ist, ob sie selbst ihre Programme nach ein oder zwei Jahren noch verstehen. Programmiersprachen sollten im Hinblick auf die Arbeitsteilung erfunden werden, die es erfordert, dass man durch schnelles Hinschauen die einzelnen Programmieraussagen versteht.

Die Verwendung des Gleichheitszeichens als Zuweisungsoperator hat noch einen anderen Nachteil. Wenn man nämlich die Bedingung

```
IF x = 17
```

schreiben will, dann kann man dies in dieser Form nicht mehr tun, weil das Gleichheitszeichen ja schon für andere Zwecke verbraucht ist. In der Sprache C muss man deshalb die Bedingung in der Form

```
IF x == 17
```

schreiben. Auch dieses ist ein Unfug, aber man kann sich leider nicht dagegen wehren, weil man eben manchmal doch gezwungen ist, in solchen Programmiersprachen zu schreiben. Man sollte sich aber dann wenigstens ärgern, wenn man so etwas schreiben muss.

Die drei Arten von Identifikationen

Der Begriff der Zuweisung eignet sich sehr gut für eine Behandlung des Begriffes der Identifikation. Identifikation muss bei jeder Kommunikation stattfinden, denn es muss ja erreicht werden, dass der Kommunikationspartner die gleichen Dinge denkt wie der, von dem die Kommunikation ausgeht. Bei einer Zuweisung muss links ein Ort identifiziert werden und rechts ein Wert. Denn jede Zuweisung verlangt, dass ein bestimmter Wert an einen bestimmten Ort gebracht wird, d.h. dass ein Behälter einen bestimmten Inhalt bekommen soll.

Nicht nur im Zusammenhang mit Zuweisungen, sondern grundsätzlich, gibt es nur drei unterschiedliche Möglichkeiten, etwas zu identifizieren. Die eine Möglichkeit ist das Zeigen. Damit man auf etwas zeigen kann, muss es wahrnehmbar und aktuell im Zeigebereich des Zeigenden sein. Die zweite Möglichkeit zur Identifikation ist das Zeigen auf ein wahrnehmbares stellvertretendes Symbol. Wenn ich beispielsweise einen Menschen identifizieren will, dieser aber aktuell nicht in meinem Zeigebereich liegt, kann ich seinen Namen aussprechen. Der Name ist ein Symbol, welches willkürlich als Stellvertreter der Person zugeordnet wurde. Dadurch, dass ich den Namen ausspreche, mache ich ihn wahrnehmbar, und das ist eine Art des Zeigens.

Bei der Programmierung müssen wir im allgemeinen Dinge identifizieren, die grundsätzlich nicht zeigbar sind. Entweder, weil sie abstrakt sind, wie beispielsweise die Zahl 5,2 , oder weil sie im Inneren des Rechners verborgen sind, was insbesondere für die Speicherzellen gilt.

Die dritte Form der Identifikation ist die Umschreibung. Diese wenden wir an, wenn wir weder auf das Gemeinte noch auf ein stellvertretendes Symbol des Gemeinten zeigen können.

Wir betrachten nun die folgenden drei Zuweisungen:

```
        m := 5;
        j := k;
    w[7] := 5.2 + v;
```

Die Speicherorte, die jeweils links vom Zuweisungsoperator identifiziert werden müssen, sind hier alle drei durch Angabe eines stellvertretenden Symbols identifiziert. Die Bezeichnungen m, k und w müssen im Programm als Namen für Speicherzellen vereinbart worden sein.

Der jeweilige Wert, der bei Ausführung der Zuweisung in die identifizierte Speicherzelle gebracht werden soll, wird rechts vom Zuweisungsoperator identifiziert. Im ersten Statement wird dieser Wert durch Angabe eines stellvertretenden Symbols identifiziert. Im zweiten und im dritten Statement dagegen wird der Wert jeweils umschrieben. Im zweiten Statement wird der Wert dadurch umschrieben, dass ein Ort k identifiziert wird, und der aktuell an diesem Ort k befindliche Wert ist der gemeinte. Im dritten Statement wird der gemeinte Wert durch eine arithmetische Formel umschrieben, welche verlangt, dass zwei Werte addiert werden, wovon der eine Summand in Form eines Symbols und der andere in Form einer Ortsangabe identifiziert wird.

Die in den Zuweisungen vorkommenden Ortsbezeichner stehen immer stellvertretend für Speicherorte innerhalb des Computers. Man muss sich diese Orte als interne Aktionsfelder in Bild 9 vorstellen. Man braucht sich nicht dafür zu interessieren, in welcher Form die Informa-

tionen in diesen Speicherzellen liegen – man braucht ja auch nicht zu wissen, wie eine gedachte „Fünf“ im Hirn biochemisch erscheint, wenn man arithmetische Operationen ausführt.

Nun muss es aber im Programm neben den Zuweisungen auch noch Ein-/ Ausgabeanweisungen geben, sonst könnte man als Rechnerverhalten nur ein Autistenverhalten beschreiben. Autisten sind Personen, in deren Kopf sich ein reiches Gedankenleben abspielt, die aber nicht in der Lage sind, mit der Außenwelt zu kommunizieren.

Ein/Ausgabeanweisungen

Ein-/ Ausgabeanweisungen sind als Anweisungen an den im Bild 9 schattiert dargestellten Akteur zu betrachten. Die Teile der Programmiersprachen, die sich ausschließlich auf die internen Aktionsfelder beziehen, sind durch Mathematik und Logik ausreichend zu erfassen. Sie sind in allen Programmiersprachen in ähnlicher Weise definiert. Dagegen geht es bei den Ein-/ Ausgabeanweisungen um die Wandlung informationeller Inhalte in materiell-energetisch Wahrnehmbares und umgekehrt. Da es hier eine offene Vielfalt gibt, unterscheiden sich hier die Programmiersprachen verhältnismäßig stark, und wer eine neue Programmiersprache lernt, muss gerade im Bereich der Ein-/ Ausgabe meistens ziemlich viel Neues hinzulernen.

Man sieht diese Problematik schnell ein, indem man ein einfaches Beispiel betrachtet. In einer internen Speicherzelle sei irgendwie die Zahl 0.00025 gespeichert, selbstverständlich in einer Form, die uns nicht zu interessieren braucht. Diese Zahl kann nun geschrieben werden als 0.00025 oder $25*10^{-5}$ oder $2.5E-4$, und selbstverständlich gibt es noch sehr viele andere mögliche Formen. Wie soll man dem Rechner mitteilen, welche Form man für die Eingabe oder die Ausgabe wünscht? Wie soll man dem Rechner mitteilen, an welche Stelle der Ausgabefläche die Zahl geschrieben werden soll, in welcher Schriftart, in welcher Größe und in welcher Farbe?

Es gibt Programmiersprachen, bei denen kann man diese unterschiedlichen Wünsche gar nicht äußern; in diesen Fällen ist die Schrifttype und die Schriftgröße einfach per Konstruktion festgelegt. Andere Programmiersprachen erlauben hier eine starke Parametrisierung. Die diesbezüglichen Anweisungen werden im allgemeinen als Formatanweisungen bezeichnet.

Vergleich unserer beiden Programmversionen FORTRAN und C

In den ersten beiden Übungen haben wir ein und denselben Algorithmus in zwei unterschiedlichen Sprachen formuliert, zum einen in Fortran, einer Sprache aus dem Jahre 1957, und zum anderen in C, einer Sprache aus dem Jahre 1985. Man darf nicht sagen, das Programm sei der Algorithmus, sondern man darf nur sagen, dem Programm liegt der Algorithmus zu Grunde. Die Idee des Algorithmus hat mit der Möglichkeit der Programmierung grundsätzlich erst einmal nichts zu tun.

Wir legen nun die beiden Programmtexte nebeneinander und versuchen, durch Vergleich herauszufinden, welche Konzepte in beiden Texten vorkommen, die sich nur in der Formulierung unterscheiden, und wo es einseitig vorkommende Konzepte gibt, die auf der anderen Seite fehlen.
Zuallererst stellen wir fest, dass es auf beiden Seiten das Konzept des Kommentars gibt. Dies ist kein Konzept, welches zum Begriff des Algorithmus gehört, sondern es ist ein Konzept, welches ausschließlich in den Bereich der Programmtexte gehört. Für die Ausführung des

Programms sind Kommentare völlig irrelevant; sie werden nur für den menschlichen Leser eingefügt, um ihm das Verständnis des Programmtextes zu erleichtern.

In beiden Programmtexten gibt es die Klammerung des Programms, d.h. also die sprachlichen Teile, die das Programm eröffnen bzw. beenden. Die eigentliche Eröffnung des Programms geschieht auf der C-Seite durch den Ausdruck

```
main()
```

der nur sagt: „Hier beginnt das Programm.“ Davor finden wir aber im C-Text noch die Anweisung

```
#include <stdio.h>
```

zu der es im FORTRAN-Text keine Entsprechung gibt. Diese Unsymmetrie ist begründet in dem großen zeitlichen Abstand, der die Entwicklung dieser beiden Programmiersprachen trennt. FORTRAN wurde im Jahre 1957 entwickelt, und damals gab es als periphere Eingabegeräte nur die Lochkartenleser und als periphere Ausgabegeräte nur die Lochkartenstanzer. Diese Begrenzung auf sehr einfache klar definierte Ein-/ Ausgabegeräte erlaubte es, die Ein-/Ausgabeanweisungen als Teil der Sprache zu definieren. Die Sprache C wurde ungefähr 25 Jahre nach FORTRAN entwickelt. Zu dieser Zeit gab es schon eine riesige Vielfalt an peripheren Geräten. Außerdem war man sicher, dass in Zukunft weitere Formen der Ein-/Ausgabe erfunden und realisiert werden würden. Deshalb hätte es keinen Sinn gemacht, sich bei der Definition der Sprache C auf ein bestimmtes Repertoire von Ein-/Ausgabegeräten festzulegen. Deshalb muss man zu jedem C-Programm mindestens eine Datei dazunehmen, in der die Ein-/Ausgabeprozeduren stehen, die sich auf die aktuell verwendeten Ein-/Ausgabegeräte beziehen. Die Datei `stdio.h` enthält die Ein-/Ausgabeprozeduren zum Anschluss von Terminals und Druckern, also zum Anschluss einer Peripherie, die der allgemeine interaktive Computernutzer standardmäßig vorfindet. Der Dateiname ist eine Abkürzung von „Standard-Input-Output“.

Typisierung von Variablen

Es gibt Programmiersprachen, bei denen die Speicherzellen typisiert sind und andere Programmiersprachen, bei denen die Speicherzellen nicht typisiert sind. Typisierung einer Speicherzelle bedeutet, dass bei der Einrichtung der Speicherzelle festgelegt wird, dass die Speicherzelle nicht für beliebige Informationen verwendet werden darf, sondern nur für Elemente von einem bestimmten Typ. Beispielsweise könnte eine Speicherzelle i eingeführt werden, in die man nur ganze Zahlen, also Zahlen vom Typ `INTEGER` einbringen darf.

Es besteht hierbei eine einfache Analogie zu Behältern aus dem täglichen Leben. Man denke an Sprudelflaschen, in die man nicht nur trinkbare Flüssigkeiten einspeichern kann, sondern auch giftige Flüssigkeiten oder Sand oder Salz. Es ist aber sinnvoll, die Benutzung von Sprudelflaschen auf die Füllung mit trinkbaren Flüssigkeiten zu beschränken. Es sind schon kleine Kinder zu Tode gekommen, weil ihre Eltern in Sprudelflaschen giftige Flüssigkeiten aufbewahrt haben. Entsprechend problematisch ist es, wenn bei der Programmierung in typisierte Speicherzellen Inhalte eingebracht werden, die nicht den geforderten Typ haben. Es ist insbesondere Aufgabe des Compilers, sicher zu stellen, dass solche Fehlbelegungen nicht vorkommen.

Wenn eine Speicherzelle im Programm als erforderlich eingeführt wird, bedeutet dies, dass Speicherplatz für diese Zelle reserviert wird. Man kann bei dieser Reservierung gleich mit verlangen, dass die Speicherzelle einen ganz bestimmten initialen Inhalt bekommt. Man kann

aber auch auf die Angabe eines initialen Inhalts verzichten. Allerdings hat die Zelle auch dann, wenn man keinen initialen Inhalt vorgegeben hat, zwangsläufig einen initialen Inhalt, den man nur nicht kennt. Denn die Speicherzellen können nichts anderes enthalten als Folgen endlicher Länge von Nullen und Einsen. Den Inhalt „leer“, der bei einer Sprudelflasche sehr wohl von einem nichtleeren Inhalt unterschieden werden kann, gibt es bei den Speicherzellen für Binärwörter nicht.

Es gibt Beschreibungen von Compilern, denen man entnehmen kann, mit welchen Initialinhalten sie die Speicherzellen belegen, für die im Programm kein Initialinhalt verlangt wird. Man sollte aber diese vom Compiler eingebrachten Initialbelegungen nicht als garantiert betrachten, d.h. man sollte im Programm immer davon ausgehen, dass eine Speicherzelle, die man nicht selbst explizit mit einem Wert belegt hat, keinen verwendbaren Wert hat, denn sonst wird die Ausführbarkeit des Programms von dem verwendeten Compiler abhängig, und das sollte natürlich auf jeden Fall vermieden werden.