4. Einfache Datenstrukturen

4.1 Variable

Eine Variable in Programmiersprachen besteht üblicherweise aus:

Name **(individuelle Namensvergabe)**
Typ **(Wertebereich, Speicherbelegung)**
Wert **(entsprechend dem Typ)**
Speicherplatz **(entsprechend dem Wert)**

Beispiel:
Name: Anzahl
Typ: ganzzahlig (**int**)
Wert: 7

Beispiele von Typen (C, C++): short, int, unsigned int, long, unsigned long, float, double, char, unsigned char, bool

Typ	Größe	Wertebereich
int	4 (od. 2)	$-2^{31} \dots 2^{31}-1$
short int	2	$-2^{15} \dots 2^{15}-1$
long int	4	$-2^{31} \dots 2^{31}-1$
unsigned short int	2	$0 \dots 2^{16}-1$
unsigned long int	4	$0 \dots 2^{32}-1$
signed char	1	$-128 \dots 127$
unsigned char	1	$0 \dots 255$

ganzzahlige Wertebereiche

Variablen werden auf dem **Heap** gespeichert und belegen 1 Byte (char), 2 Byte (short int), 4 Byte (int, long, float) oder 8 Byte (long, double) oder eine noch größere Anzahl von Bytes.

Die Typen hängen von der Programmiersprache und vom Betriebssystem (32 Bit oder 64 Bit) ab. Beispielsweise erhält der Typ **long** auf einem 32-Bit-Betriebssystem 4 Byte und auf einem 64-Bit-Betriebssystem 8 Byte Speicherplatz. Bei den Zahlentypen unterscheidet man noch nach Festkommatypen (short, int, long, char) für die Aufnahme ganzer Zahlen und nach Gleitkommatypen (float, double) für die Aufnahme von reellen Zahlen. Für ganze Zahlen, die nicht negativ sind, können unsigned-Typen (unsigned short, unsigned int, unsignd long) verwendet werden. Hier entfällt das Vorzeichen-Bit, welches den maximalen Zahlenwert verdoppelt.

Variablen müssen im Programm vereinbart werden und sollten einen Anfangswert erhalten:
int x = 5, y = 10; unsigend long k = 1UL; char c = 'A'; char *name = "HTW";

Unter der **Adresse einer Variablen** versteht man einerseits die **Adresse des zugeordneten Speicherbereiches** im Hauptspeicher des Computers.

D.h. der **symbolische Name** der Variablen wird vom Computer in eine **reale Speicherplatzadresse** transformiert, der Nutzer braucht sich um diese Transformation nicht zu kümmern. Im Speicherbereich wird der **Wert** der Variablen untergebracht.

Andererseits kann man sich in Programmiersprachen wie C, C++, C# direkt auf die **Adressen von Variablen** (und anderen Objekten) beziehen. Es sind dann auch Operationen mit Adressen möglich.

Im folgenden Beispiel wird **px** als ein Zeiger auf einen **int**-Speicherplatz vereinbart. Da **px** mit der Adresse von **x** initialisiert wird, bewirkt die Zuweisung ***px = 8;** die Zuweisung des Wertes **8** an die Dereferenzierung ***px** der Adresse **px**, d.h. letztendlich **x = 8**;

Die **Dereferenzierung** von ***px** kann neben ***px** auch über ***(px + 0)** oder **px[0]** erfolgen:

Beispiel: int x = 5; int *px = &x; *px = 8; *(px + 0) = 8; px[0] = 8;

Adressen können auch auf neu mit **malloc** (**C**) oder **calloc (C)** bzw. **new** (**C++**) angelegte Speicherbereiche auf dem **Heap** verweisen (vgl. **pointer.pdf**):

```
C: int *py = (int *)malloc(sizeof(int));  *py=9; printf("py= %p\n", py);
                                                 printf("*py= %d\n", *py); free(py); py=0;

C++ :    int *py = new int(4); *py = 9; cout<<"py = "<<py<<endl;
                                        cout<<"*py = "<<*py<<endl; delete py; py = 0;
```

Mit **malloc**, **calloc** bzw. **new** auf dem Heap vergebener Speicher sollte mit **free** bzw. **delete** wieder freigegeben werden. In sogenannten "**managed**-Umgebungen", wie es in .NET C# bzw. JAVA der Fall ist, entfällt **free** bzw. **delete** und wird seitens eines "**garbage collectors**" automatisch, undeterminiert und zeitversetzt durchgeführt.

Adressen werden auch als Werte vom Datentyp **reference** bezeichnet. Zu beachten ist hierbei die Verwechslungsgefahr mit **Referenzen aus C++**, welche als "Zeiger für Arme" bezeichnet werden und wie folgt vereinbart werden: **int i = 0, &refi = i; ++refi; cout<<"i = "<<i<<endl;**

refi ist eine Referenz und hier ein **Alias-Name** für **i**, d.h. alle Operationen mit **refi** sind auch Operationen mit **i** und umgekehrt. Wichtig sind Referenzen insbesondere als Parameter, weil hierbei ein direkter Zugriff auf die als Referenz übergebene Variable von innerhalb nach außerhalb der Funktion erfolgt.

Variablen, die für die Adressierung von Speicherbereichen vorgesehen sind, werden häufig als **Zeigervariablen** bezeichnet. Zeigervariablen können einen ausgezeichneten Wert **0** bzw. **NULL** als Makrodefinition (C, C++) bzw **null** (C#) bzw. **nil** (LISP) besitzen, der aussagt, daß die Variable auf **keinen** Speicherplatz verweist.

Zeigervariablen besitzen in **32-Bit**-Betriebssystemen eine Länge von **4 Byte** und können damit maximal $\mathbf{2^{32} - 1}$ Byte , d.h. ca. **4 GByte** adressieren. In **64-Bit**-Betriebssystemen besitzen Zeigervariablen eine Länge von **8 Byte** und können damit $\mathbf{2^{64}-1}$ Byte addressieren.

Der Vorteil von Zeigervariablen und der dynamischen Speicherplatzanforderung über **new** ist, daß Datenstrukturen **während der Abarbeitung eines Programms** willkürlich aufgebaut, verändert und gelöscht werden können (**dynamische Datenstrukturen**). Damit sind **allgemeinere Algorithmen und Programme** formulierbar als es mit **festen Feldgrenzen** der Fall ist

Der Wert einer Variablen darf üblicherweise während der Laufzeit geändert werden.

Variablen können auch **const** sein. Eine Änderung dieser Variablen wird damit verhindert:
const x = 10; // Initialisierung
x = x+1; // **Übersetzungsfehler** des Compilers, da **const x**

Eine Anweisung **x = x + 1;** bedeutet in vielen Programmiersprachen, dass der Wert des Ausdrucks **x + 1** berechnet wird und das Ergebnis in **x** auf der linken Seite von = abgespeichert wird. Der alte Wert von **x** geht damit verloren.

4.2 Felder

Ein **Feld** ist eine Menge von Variablen mit **gleichem Typ**. Der **Name** eines Feldes ist die **Adresse** des Elementes mit **Index 0**. Einige ältere Sprachen (PASCAL) bzw. mathematische Sprachen (MATLAB) beginnen ein Feld mit Index 1. Felder können **Vektoren**, **Matrizen** oder **höherdimensionale** mathematische Konstrukte sein. Über **Indizes** kann auf die Feldelemente zugegriffen werden. Ein Zugriff **vector[i]** ist die Dereferenzierung der Adresse **vector + sizeof(int) * i** und kann auch als ***(vector + i)** geschrieben werden. Der Ausdruck **vector + i** ist die **Adresse** des Elementes mit Index **i** , d.h. **vector + i** impliziert die Länge des Basistyps des Feldes, hier der Typ **int**, dessen **Länge in Bytes** mit **sizeof(int)** bestimmt wird. Ein Vektor wird als eine Aufeinanderfolge von Bytes gespeichert, wobei die Adresse des Elementes mit Index **i** über **vector + sizeof(Basistyp) * i** berechnet wird, der Wert selbst wird über **vektor [i]** bzw. ***(vector + sizeof(Basistyp) * i)** erhalten (vgl. **vektor0.pdf**).

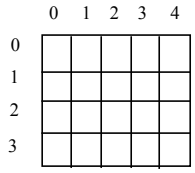
Beispiel:
```
const int n = 5;                    // Konstante
int vector [n];                     // Feld, hier Vektor auf Stack
for(int i=0; i<n; i++)              // alle Feldelemente durchlaufen
        vector[i] = i*i;            // Zuweisung i*i an vector[i]
```

Nachteil des Beispiels ist die **feste Feldgrenze n = 5**. Allgemeiner und vorteilhafter ist die Nutzung einer **dynamischen Feldgrenze (**vgl. **vektor.pdf)**:

Beispiel:
```
int n=1; cout<<"n = ";                  // C
do {  printf("\nZahl > 0 fuer die Dimension des Vektors: ");
      scanf("%d", &n); while(getchar()!='\n');
} while(n<=0);
int *vektor = (int *)malloc(n*sizeof(int)); // vektor[n]  dynamisch auf dem Heap
for(int i=0; i<n; i++)                  // alle Feldelemente durchlaufen
        vektor[i] = i*i;                // Zuweisung i*i an vector[i]
free(vektor); vektor = 0;               // Freigabe Speicherplatz auf Heap
```

Beispiel: Feld als Matrix mit festen Feldgrenzen

```
int matrix[4][5];
for(int i=0; i<4; i++)
   for(int k=0; k<5; k++)
          matrix[i][k] = i+k;
```



Die **Adresse** des Elementes **matrix[i][k]** lautet **matrix + i*5*sizeof(int) + j*sizeof(int)** , d.h. Matrizen werden **zeilenweise abgespeichert** (vgl. **matrix0.pdf**).

Matrizen und höherdimensionale Felder können ebenfalls **dynamisch** angelegt werden (vgl. **matrix.pdf**, **matrix2.pdf**).

4.3 Listen

Eine **Liste** ist eine **Folge von Variablen** (in der Regel gleichen Typs). Es muss zumindest eine Möglichkeit vorhanden sein, um vom Speicherplatz einer Variablen zum Speicherplatz der **nachfolgenden** Variablen zu gelangen. Es muss auch feststellbar sein, ob eine **nachfolgende Variable** existiert oder nicht. Desgleichen muss es eine Möglichkeit geben, die **erste Variable**, d.h. deren Speicherplatz, zu erreichen.
Man nennt die **Variablen einer Liste** auch **Listenelemente**.

Häufig benutzte Varianten:
(a) Dichte Abspeicherung: Die Speicherplätze der Listenelemente, d.h. der einzelnen Variablen, werden dicht entsprechend der Reihenfolge angeordnet. Der **Anfang** der Liste ist damit durch die **Adresse des ersten Elementes** gegeben. Die Adresse eines nachfolgenden Elementes findet man durch **Erhöhen der Adresse** um die Länge des Speicherplatzes für ein Listenelement. Die **Anzahl der Listenelemente** kann in einer gesonderten Variablen gespeichert sein.

Adresse des Listenelements	Listenelement
a	Element 1
a + b	Element 2
a + 2*b	Element 3
...	
a + (n - 1) * b	Element n
a - Adresse der Liste (Adresse des ersten Listenelementes)	
b - benötigter Speicherplatz je Listenelement (Byte)	

Man beachte, dass eine Liste in diesem Fall wie ein **eindimensionales Feld von Variablen** gleichen Typs abgespeichert ist und auch so verwendbar ist. Aus diesem Grund gibt es in vielen Programmiersprachen keine Standard-Datenstruktur vom **Typ Liste**, da **eindimensionale Felder** (**Vektoren**) genau diese Eigenschaft besitzen.

(b) Programmiersprache LISP
LISP ist eine Familie von Programmiersprachen, die 1958 erstmals spezifiziert wurde und am Massachusetts Institute of Technology (MIT) in Anlehnung an den Lambda-Kalkül entstand. Es ist nach Fortran die zweitälteste Programmiersprache, die noch verbreitet ist.

Auf Basis von **Lisp** entstanden zahlreiche Dialekte. Zu den bekanntesten zählen **Common Lisp** und **Scheme**. Daher bezieht sich der Begriff **Lisp** oft auf die Sprachfamilie und nicht auf einen

konkreten Dialekt oder eine konkrete Implementierung. **LISP** steht für **Lis**t **P**rocessing (Listen-Verarbeitung). Die Grunddatenstrukturen von **Lisp** sind Einzelwerte (Skalarwerte), die **Atome** genannt werden, und **Listen**. Die **Listen** können beliebig verschachtelt werden (**Listen von Listen**). Die Listen werden mit runden Klammern dargestellt, eine **einfache Liste** ist eine **lineare** Anordnung von **Atomen**, z.B.: **(A, B, C, D)**

(A, (B, C), D) ist eine Liste mit drei Elementen, von denen das zweite die Liste **(B, C)** ist.

((A, (B, C)), (D, E)) ist eine Liste mit zwei Elementen; das erste Element ist selbst eine Liste, bestehend aus dem Atom **A** und der Liste **(B, C)**.

Auch **((A))** ist eine Liste aus einem Element, welches eine Liste mit einem Element, dem Atom **A** ist.

Die Anordnung der Elemente liefert in der Regel mehr Informationen, als die Elemente selbst.

Grundlegende Funktionen für die Verarbeitung von Listenstrukturen sind folgende:

a.) **Kopf** einer Liste: **Car[z]** oder **Hd[z]** hat als Wert das **erste Element** der Liste **z**, also

z	Hd[z]
(A, B, C)	A
(A, (B, C))	A
((A, B), C)	(A, B)
((A))	(A)
(A)	A

b.) **Rumpf** einer Liste: **Cdr[z]** oder **Tl[z]** hat als Wert den **Rest** der Liste, den man erhält, wenn man das erste Element der Liste streicht. Ist kein Rest vorhanden, dann hat **Tl[z]** den Wert **NIL**. **NIL**, **T** (wahr) und **F** (false) sind universelle **Konstanten** des Systems.

z	Tl[z]
(A, B, C)	(B, C)
(A, (B, C))	((B, C))
((A, B), C)	(C)
((A))	NIL
(A)	NIL

c.) **Cons[x,y]** hat als Wert eine **neue Liste**, die **x als Kopf** und **y als Rumpf** hat.

x	y	Cons[x,y]
A	(B, C)	(A, B, C)
(A, B)	(C)	((A, B), C)
(A)	(B)	((A), B)
A	NIL	(A)
(A)	NIL	((A))

Man beachte, daß **Cons[Hd[x], Tl[x]] = x**

d.) **Atom [x]** ist dann und nur dann **wahr**, wenn **x ein Atom** ist.
Null [x] ist dann und nur dann **wahr**, wenn **x das Atom NIL** ist.
Eq [x, y] ist dann und nur dann **wahr**, wenn **x und y Atome** sind und **x = y** gilt.
Ist x oder y **kein Atom**, dann ist die **Funktion nicht definiert**.

Andere Funktionen werden mit Hilfe von **Hd**, **Tl** und **Cons** und bedingten Ausdrücken, die die Bedingungen **Atom**, **Null** und **Eq** verwenden, gebildet, z.B.:

(a) **Equal [x, y]** hat den Wert **wahr**, wenn entweder **x** und **y gleiche Atome** oder **gleiche Listenstrukturen** darstellen.

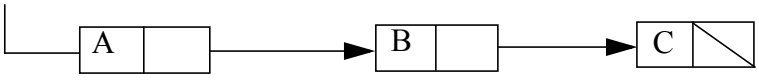
Equal [x, y] = [Atom[x] --> Atom[y] and Eq[x, y],
Atom[y] --> F,
Equal[Hd[x], Hd[y]] and Equal[Tl[x], Tl[y]]]

(b) **Rev[x]** hat als Wert eine Liste, die die Elemente von **x** in **umgekehrter Reihenfolge** enthält:

Rev[x] = Reva[x, NIL]
mit **Reva[a, b] = [Null[a] --> b, Reva[Tl[a], Cons[Hd[a], b]]]**

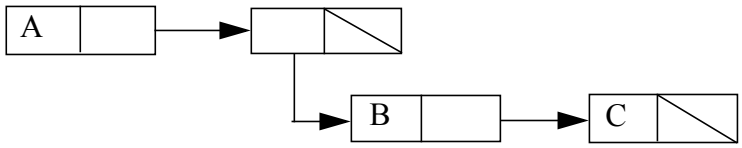
Beispiel: **Rev[(A, B, (C, D))] = ((C, D), B, A)**

Eine Listenstruktur wird in einer Rechenanlage für gewöhnlich durch eine Folge von Zellen dargestellt, die in zwei Hälften geteilt sind. Die eine Hälfte enthält das Listenelement, die andere Hälfte einen Zeiger auf das nächste Listenelement. D.h. Listen werden nicht in benachbarten Zellen gespeichert. Die Liste (A, B, C) würde folgendermaßen dargestellt werden:

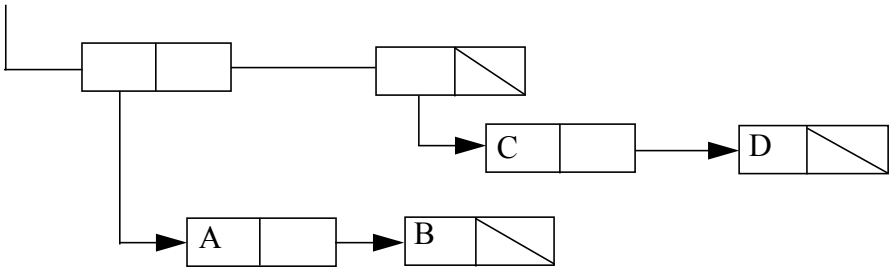


Die Pfeile zeigen an, daß die Zellen einen Zeiger zum nächsten Element enthalten. Die Diagonale in C zeigt das Ende der Liste an, es kann als das Atom **NIL** aufgefaßt werden.

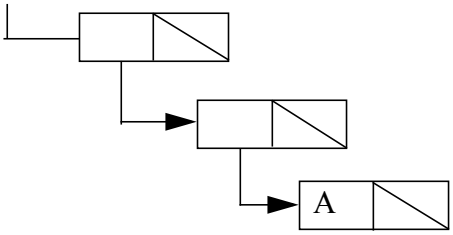
Teillisten, z.B. die Liste **(A, (B, C))** , wird folgendermaßen dargestellt:



Kompliziertere Strukturen sind: **((A, B), (C, D))**



(((A)))



Die Funktion **Cons[x, y]** nimmt eine neue Zelle und bringt **x** in den **Kopfteil** und **y** in den **Rumpfteil**.

Vollziehen Sie folgende Beispiele nach:
y = (B, C, D), **x = A**, **Cons[x, y] = (A, B, C, D)**

y = NIL, **x = A**, **Cons[x, y] = (A)**

x = (A, B, C), **Hd[x] = A**, **Tl[x] = (B, C)**

x = A, **Hd[x]** nicht definiert, **Tl[x]** nicht definiert

x = (A), **Hd[x] = A**, **Tl[x] = NIL**

x = (A, B, C), **Hd[Tl[Tl[x]]] = C**, **Tl[Tl[Tl[x]]] = NIL**

x = ((A, B), (C, D)), **Hd[x] = (A, B)**, **Tl[x] = ((C, D))**

x = (A, B), **y = (C, D, E)**, **Cons[Hd[x], Cons[Hd[Tl[x]], y]] = (A, B, C, D, E)**

x = NIL, **Atom[x] = F**, **Null[x] = F**, **Null[Hd[x]] = T**, **Null[Tl[x]] = T**

x = ((A, B), C, D), Eq[Hd[x], A] = nicht definiert, Eq[Hd[Hd[x]], A] = T

Wenn $\mathbf{B_i}$ logische Funktionen und $\mathbf{E_i}$ Aktionen sind, dann ist ein bedingter Ausdruck (auch als Mc Carthy - Funktion bezeichnet) der Art

$\mathbf{B_1 \text{ --> } E_1 , B_2 \text{ --> } E_2 , B_3 \text{ --> } E_3 , ... , B_n \text{ --> } E_n , E_{n+1}}$

äquivalent zu

if $\mathbf{B_1}$ then $\mathbf{E_1}$ else
if $\mathbf{B_2}$ then $\mathbf{E_2}$ else
...
if $\mathbf{B_n}$ then E_n else
$\mathbf{E_{n+1}}$

Logische Konjunktion: **And[a, b] = a --> b, F**
Logische Negation: **Not[a] = a --> F, T**
Logische Disjunktion: **Or[a, b] = a --> T, b**
Definition Null: **Null[x] = Atom[x] -->Eq[x, NIL], F**
Zusammenhängen Listen: **Append[x, y] = Null[x] --> y, Cons[Hd[x], Append[Tl[x], y]]**

Beispiele: **x = (A, B, C), y = (D, E)** , **Append[x, y]** = ... = **(A, B, C, D, E)**

Hello world Programm: **(print "Hello world")**

Berechnung von n! :
```
(defun factorial (n)
    (if (<= n 1)
        1
    (* n (factorial (- n 1)))))
```

Umkehrung einer Liste: **Rev[x] = Rev1[x, NIL]**
Rev1[a, b] = Null[a] --> b,
Rev1[Tl[a], Cons[Hd[a], b]]

x = (A, (B, C), D), Rev[x] = ... = (D, (B, C), A)

Übung:

1.) Man wähle aus der Liste **x = (A, (B, C), D)** das Atom **C** aus

2.) Für **x = (A, (B, C), D)** bestimme man die Werte der folgenden Ausdrücke:
Atom[x]
Atom[Hd[x]]
Null[Tl[Tl[Hd[Tl[x]]]]]
Eq[Hd[Tl[x]], B]

3.) Seien **x = A**, **y = B**, **z = (C, D)**. Welche Liste beschreibt **Cons[x, Cons[y, z]]** ?
Man baue die Liste **(A, B, (C, D))** aus **x**, **y** und **z** auf.

4.) Sei **x** eine Liste. Welche Liste beschreibt der Ausdruck
Cons[Cons[Hd[x], Tl[x]],Tl[Cons[Hd[x],NIL]]] ?

5.) Welche Liste erfüllt alle folgenden Bedingungen ?
Eq[Hd[Hd[x]],R] = T
Hd[Tl[Hd[x]]] = Hd[Tl[Tl[x]]]
Tl[Tl[Hd[x]]] = (C, H)
Eq[Hd[Tl[x]], T] = T
Tl[Tl [x]] = (I, G)

6.) **Ff[x]** gibt das erste Atom der Liste **x** an. Wie ist **Ff[x]** definiert ?
Beispiel: **x = (((A, B), C, D), E)** , **Ff[x] = A**

7.) **Ap[x, y]** setzt **y** hinter das letzte Element von **x** als das neue letzte Element ein.
Wie ist **Ap[x, y]** definiert ? Bsp: **x=(A, B, C)**, **y=(D, E)**, **Ap[x, y] = (A, B, C, (D, E))**

8.) Unter Verwendung der Funktion **Ap**, **Tl**, **Hd** und **Atom** definiere man eine Funktion **Revs[x]**, die die Reihenfolge aller Elemente von **x** und seiner Unterlisten umkehrt.
Beispiel: **x = (A, (B, C), D)** , **Revs[x] = (D, (C, B), A)**

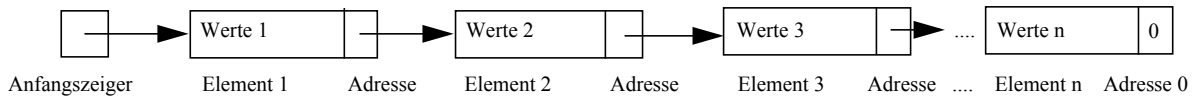
9.) **Flatten[x]** entfernt alle innere Klammern einer Liste **x**.
Beispiel: **x = (A, (B, (C, D)), E)** , **Flatten[x] = (A, B, C, D, E)**

Definition: **Flatten[x] = Flat[x, NIL]**
Flat[a, b] = Null[a] --> b,
Atom[a] --> Cons[a, b],
Flat[Hd[a], Flat[Tl[a], b]]
Für das Beispiel gebe man die einzelnen Schritte der Rekursion übersichtlich an.

10.) Die Funktion **Copy** dient zum Kopieren von Atomen und Listen.
Definition: **Copy[a] = Atom[a] --> a,**
Cons[Copy[Hd[a]], Copy[Tl[a]]]

Anhand der Liste **x = (((A), B))** in graphischer Darstellung ist die Liste **Copy[x]** entsprechend der Definition in Einzelschritten aufzubauen.

(c) Einfache Verweiskette:
Jede Variable erhält zusätzlich einen Speicherplatz zugeordnet, in dem die Adresse der nächsten Variablen untergebracht wird. Der Anfang der Liste ist durch die Adresse des ersten Elementes gegeben. Die letzte Variable erhält eine spezielle Adresse (z.B. Null) zugeordnet, die das Ende der Liste anzeigen soll.



Das folgende Beispiel zeigt den Aufbau einer **einfach verketteten Liste** in **C**.

Die Struktur **struct element** besitzt **int value** zum Speichern der Werte, **struct element *next** ist der Zeiger auf das Folgeelement der Liste. In der **main()**-Funktion ist **first** der Zeiger auf das erste Listenelement, am Anfang **0**. Dem Zeiger **eineE** wird dynamisch Speicherplatz für ein Listenelement zugewiesen und die Werte **value = 1** und **next = 0** belegt.

Das Einketten des Elementes **einE** geschieht mittels des Funktionsaufrufes **append(&first, einE)**. Da **first** in **append** neu belegt wird, muß die Adresse von **first** übergeben werden.

Für ein weiteres Listenelement müßte in **main** erneut mit **malloc** dynamisch Speicherplatz ausgefaßt werden, der wieder von **einE** referenziert werden kann. Im Beispiel fehlen Funktionen für das Löschen eines Listenelementes und das Löschen der gesamten Liste. Weiterhin könnten Funktionen zum Sortieren der Liste, zur Ermittlung der Anzahl der Elemente und zur Ausgabe der Liste sinnvoll sein.

```
#include <stdio.h>
#include <stdlib.h>
struct element {
       int value;             // der Wert des Elements
       struct element *next; // das nächste Element
};

void append(struct element **lst, struct element *new)
{
  struct element *lst_iter = *lst; // lokaler Zeiger zum Iterieren
  if ( lst_iter!= 0 ) {            // sind Elemente vorhanden ?
      while (lst_iter->next != 0 ) // suche das letzte Element
          lst_iter=lst_iter->next;
      lst_iter->next=new;          // Hänge das Element hinten an
  }
  else         // wenn die liste leer ist, bin ich das erste Element
      *lst=new;
}

void main(){
       struct element *first = 0; // init. die Liste mit 0 = leere liste
       struct element *einE = 0;  // Listenelement
       einE = malloc(sizeof(struct element)); // erzeuge ein neues Element
       einE->value = 1;       // Wert des Listenelementes
       einE->next = 0;        // Wichtig für das Erkennen des Listenendes
       append(&first, einE); // füge das Element in die Liste ein
}
```

Günstiger zu implementieren sind Listen in **C++**. Die Klasse **class liste** enthält mit **class element** den Typ für ein Listenelement, wobei der Typ **T** des Wertes **t** als Template-Parameter später festgelegt wird. Der Zeiger **pl** dient als Zeiger auf den Listenanfang. Die eingebettete Klasse **class element** dient als Typ für Listenelemente.

```
#include <iostream>
#include <iomanip>
using namespace std;

template <class T>class liste {
  class element {            // eingebettete Klasse als Typ für Listenelement
    public:
      T t;                   // Wert des Listenelements
      element *pe;           // Zeiger auf Nachfolger
      element(const T  &Item):t(Item),pe(0){}  // Konstruktor
  };
  element *pl;                      // Zeiger auf erstes Listenelement
 public:
  liste():pl(0){}                   // Konstruktor; Listenanfang, zu Beginn 0

  void InsFirst(const T &Item){     // Einfuegen an den Listenanfang
       element *p = new element(Item);
       p->pe = pl;
       pl = p;
  }

  void InsEnd(const T &Item){       // Einfügen an das Listenende
       element *p = pl, *pold = pl;
       unsigned long anz = 0UL;
       while(p){
         anz++;
         pold = p;
         p = p->pe;
       }
       pold->pe = new element(Item);
  }

  bool search(const T &Item){       // Listenelement Item in Liste ?
       element *p = pl;
       while(p){
         if(p->t == Item) return true;
         p = p->pe;
       }
       return false;
  }

  void RemFirst(){                  // Entferne erstes Listenelement
       element *p = pl;
       if(p){
         pl = p->pe;
         delete p;
       }
  }
```

```
  unsigned long anzahl(){            // Anzahl der Listenelemente
       element *p = pl;
       unsigned long anz = 0UL;
       while(p){
         anz++;
         p = p->pe;
       }
       return anz;
  }

  void show(){                        // Anzeige aller Listenelemente
       element *p = pl;
       unsigned long i = 0UL;
       cout<<"\nAusgabe der Liste:\n";
       while(p){
         cout<<"Element ["<<setw(4)<<i<<"] = "<<p->t<<endl;
         p = p->pe;
         i++;
       }
       cout<<endl;
  }
};

void main(){                          // Hauptprogramm
  int i = 0;
  liste<int> p;                       // Objekt p enthält Liste vom Typ int
  p.InsFirst(0);
  p.InsFirst(1);
  p.InsFirst(2);
  p.InsFirst(3);
  p.InsFirst(4);
  p.InsEnd(5);
  cout<<"p.anzahl = "<<p.anzahl()<<endl;
  p.show();
  cout<<"3 enthalten = "<<boolalpha<<p.search(3)<<endl;
  p.RemFirst();
  p.RemFirst();
  p.RemFirst();
  cout<<"p.anzahl = "<<p.anzahl()<<endl;
  p.show();
  cin.get();
}
/*
p.anzahl = 6
Ausgabe der Liste:
Element [   0] = 4
Element [   1] = 3
Element [   2] = 2
Element [   3] = 1
Element [   4] = 0
Element [   5] = 5
3 enthalten = true
p.anzahl = 3
Ausgabe der Liste:
Element [   0] = 1
Element [   1] = 0
Element [   2] = 5
*/
```

4.4 Tabellen

Eine Struktur (oder Zeile) sei eine endliche Menge von Variablen mit evtl. verschiedenen Typen.

Zwei Zeilen heißen **gleichartig**, wenn sie die gleichen Mengen von Variablen enthalten, und sich nur in den zugehörigen Speicherplätzen (und Werten) unterschieden.

Eine Tabelle ist eine Folge von gleichartigen Zeilen.

Beispiel.: Diese Tabelle enthält 4 gleichartige Zeilen von je 3 Variablen mit den Bezeichnungen *Name*, *Vorname* und *Telefonnummer*. Deren Typen sind verschieden.

Name	Vorname	Telefonnummer
Max	Meier	12345
Werner	Seifert	246815
Jan	Berger	7735442
Hella	Richter	85322
Gustav	Voigt	9222315

Tabellen werden vorrangig bei relationalen Datenbanken verwendet.