Prozedurale Programmierung

Informationen und Zusatzmaterial
zur Vorlesung und Übung

Prof. Dr. Siegfried M. Rump

Institut für Zuverlässiges Rechnen (E-19)

Wintersemester 2018/2019

Version: 8. Oktober 2018

Ansprechpartner für die Übung:

Florian Kerkhoff
E-Mail: `Florian.Kerkhoff@tuhh.de`
Betreuter Pool: Montags, 9:45 - 11:15 Uhr in E2.042P5b

Inhaltsverzeichnis

Zeit	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
08:00					
08:15					
08:30					
08:45					
09:00					
09:15					
09:30					
09:45	**Praktikum: Prozedurale Programmierung** Siegfried Rump E – SBC3 Raum E2.042P5b 09:45 – 11:15				
10:00					
10:15					
10:30					
10:45					
11:00					
11:15					
11:30			**Praktikum: Prozedurale Programmierung** Siegfried Rump L – DE17 Raum 3038P1 11:30 – 13:00		
11:45					
12:00	**Testat: Prozedurale Programmierung** Siegfried Rump E – SBC3 Raum E2.024P3c 12:00 – 16:00				
12:15					
12:30					
12:45					
13:00					
13:15					
13:30					
13:45					
14:00					
14:15				**Praktikum: Prozedurale Programmierung** Siegfried Rump E – SBC3 Raum E2.054P4b 14:15 – 15:45	
14:30					
14:45					
15:00					
15:15					
15:30					
15:45					
16:00					
16:15					
16:30					
16:45	**Praktikum: Prozedurale Programmierung** Siegfried Rump E – SBC3 Raum E2.009P3a 16:45 – 19:45				
17:00					
17:15					
17:30					
17:45				**Prozedurale Programmieru...** Siegfried Rump H – SBC5 Raum Audimax1 17:45 – 18:30	
18:00					
18:15					
18:30				**HÜ: Prozedurale Programm...** Siegfried Rump H – SBC5 Raum Audimax1 18:30 – 19:15	
18:45					
19:00					
19:15					
19:30					
19:45					

Abbildung 1: Überblick der Veranstaltungen (`https://intranet.tuhh.de`).

Organisatorisches

- Der **erste Termin am 25.10.2017** ist die reguläre **Vorlesung** (jeweils Do. 17:45-18:30 Uhr, Audimax 1).
- Im Anschluss an die Vorlesung findet die **Hörsaalübung** (jeweils Do. 18:30-19:15 Uhr, Audimax 1) statt. In der Hörsaalübung werden Hinweise zur Bearbeitung der Übungsaufgaben gegeben. → Keine Anwesenheitspflicht
- Das **Praktikum** sind Zeiten, in denen Rechnerpools für ihre Arbeit am PC reserviert sind, mit Ausnahme des Montags-Termins 9:45 - 11:15 Uhr im Pool E2.042P5b. Hier findet jede Woche eine offene **Sprechstunde** statt, während welcher ein Tutor für Fragen oder Probleme zur Verfügung steht. Die anderen Pool-Termine sind nicht betreut.
  → Keine Anwesenheitspflicht

  $\Longrightarrow$ Jedoch können zu diesen Zeiten ihre **Gruppentestate** stattfinden, Ort und Zeit Ihrer jeweiligen Gruppe sind im StudIP angegeben.
  → **Anwesenheitspflicht** zum Erhalt des Klausurbonus

Klausurbonus durch Gruppentestate

- **Klausur:** Am Ende des Semesters, 5 Aufgaben à 12 Punkte, bestanden ab 30 Punkten.

  → Grundlage: Vorlesung und Übungen.

- **Übungsschein als Klausurbonus:** Ab Note 4.0 werden auf jede der 5 Aufgaben jeweils 2 Punkte (maximal 12 Punkte) addiert.

  Bedingungen:

  - Gilt nur für den ersten Klausurversuch nach Erhalt des Übungsscheins.
  - Erhalt von min. 84 von 120 möglichen Punkten **(70%)** aller Übungsblätter (inkl. Programmierprojekt).
  - Erhalt von mindestens 6 von 10 Punkten pro Übungsblatt (maximal eine Ausnahme).
  - Zwei kleine Programmieraufgaben (**Präsenzaufgaben**) muss jedes Gruppenmitglied alleine lösen. Alle möglichen Präsenzaufgaben sind bereits im StudIP einsehbar und können / sollten zuvor geübt werden.

Ablauf der Gruppentestate

- **Dreiergruppen** stellen wöchentlich beim Tutor die eigene Lösung vor (25 - 30 Minuten).
- StudIP `https://e-learning.tu-harburg.de/studip/` Anmeldeverfahren zu den Gruppentestaten:
  - Unter **TeilnehmerInnen** → **Funktionen/Gruppen** in Dreiergruppe eintragen
  - Viele Gruppen sind bereits mit einem Termin versehen. Die Terminvereinbarung der übrigen Gruppen erfolgt per E-Mail direkt mit dem Tutor.
- Einzelne Teilnehmer einer Gruppe können eine **unterschiedliche Punktezahl** bekommen.
- **Täuschungsversuche** führen zum Verlust des Scheins und es erfolgt eine Meldung an das Prüfungsamt!

Vorlesung 1

- Ein C-Programm
- Rechneraufbau
- Ordnungsprinzipien (Bytes, Takt, ...)
- Syntax/Semantik
- Syntaxdiagramme
- Variable, Zuweisungen, Basis-Operationen
- Zweierkomplement, „wrap around“
- Syntax: **#include**, **void**, **int**, **long int**, printf("%d\n", x), + − ∗ /, **return**

Einfaches C-Programm

```
#include <stdio.h>    /* Präprozessordirektive */

int main(void) {      /* Beginn Hauptprogramm */
  long int a, b, c;   /* Vereinbarungen */
  int x, y;           /*        "       */
  a = b = 5;          /* Anweisungen */
  x = 7 * a - b;      /*      "      */
  printf("%d\n", x);  /*      "      */
  return (0);         /* Ergebnis "0" entspricht
                         "Kein Fehler" */
} /* main */          /* Ende */
```

Vorlesung 2

- Ausdrücke, Klammern
- Zuweisungen
- Bedingungen/Vergleiche, if-Abfragen
- `while`-Schleife
- Gleitpunkt-Datentypen
- IEEE 754-Darstellung und -Arithmetik
- Syntax: <, <=, >, >=, ==, !=, &&, ||, **if**, **else**, **while**, **float**, **double**

Vorlesung 3

- Gleitkommaarithmetik
- Typkonvertierung int, float, double
- `do-while`-Schleife
- Syntax: %f, %e, %lf, %le, %c, **short int**, **char**, **do while**

Gleitkommaarithmetik

Eine 4-Byte-Gleitkommazahl (binary32 oder der C-Datentyp **float**) wird nach dem IEEE 754 Standard im Speicher folgendermaßen dargestellt:

sign	exponent			significant			
s	e_1	...	e_8	m_1	m_2	...	m_{23}

Gespeichert werden folgende drei Komponenten:

- sign: Vorzeichenbit s
- exponent: "biased" (um $B = 127$ geshifteter) Exponent

$$e := \sum_{i=1}^{8} 2^{8-i} e_i = 128e_1 + 64e_2 + \cdots + 2e_7 + e_8$$

- fraction (gespeicherte Anteil vom significant)

$$f := \sum_{i=1}^{23} 2^{-i} m_i = \frac{m_1}{2} + \frac{m_2}{4} + \frac{m_3}{8} \cdots + \frac{m_{23}}{2^{23}}$$

Die **normalisierte** (Fall 3) Gleitkommazahl x ergibt sich aus den gespeicherten Komponenten:

$$x = \underbrace{(-1)^s}_{sign} \cdot \underbrace{(\underline{1} + f)}_{significant} \cdot \underbrace{2^{e-B}}_{exponent} \tag{1}$$

Die **denormalisierte** (Fall 4) Gleitkommazahl x ergibt sich aus den gespeicherten Komponenten (ohne die **implizite Eins**):

$$x = \underbrace{(-1)^s}_{sign} \cdot \underbrace{(\underline{0} + f)}_{significant} \cdot \underbrace{2^{e-B}}_{exponent} \tag{2}$$

Für die Interpretation der dargestellten Gleitkommazahl x unterscheidet man folgende 5 Fälle:

Fall	exponent	fraction	Resultierende Gleichpunktzahl
1	$e = 255$ $(E = 128)$	$f \neq 0$	$x = nan$ (not a number)
2	$e = 255$ $(E = 128)$	$f = 0$	$x = (-1)^s \cdot inf$ (infinite ∞)
3	$0 < e < 255$ $(-127 < E < 128)$	$f \neq 0$	Gleichung (1)
4	$e = 0$ $(E = -127)$	$f \neq 0$	Gleichung (2)
5	$e = 0$ $(E = -127)$	$f = 0$	$x = 0$

Exponentialfunktion

Beispiel: $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$ sehr gute Konvergenz für $|x| \leq 1$.

$$e = 1 + 1 + 0.5 + 0.16\overline{6} + 0.041\overline{6} + 0.008\overline{3} + \ldots$$

$$\frac{1}{10!} \sim 2.7e-7$$

$$\frac{1}{20!} \sim 4.1e-19$$

```c
#include <stdio.h>

int main(void) {
  float y = 1.0,  /* y float => Error<=1e-7 */
        t = 1.0, x = 1.0;
  int i = 0;
  while (t > 1e-7) {
    i = i + 1;
    t = t * x / i;  /* Konversion int i -> float */
    y = y + t;
  }
  printf("e ist (naeherungsweise): %f", y);
  return (0);
} /* main */
```

Problem: 1e−7 ist eine absolute und keine relative Genauigkeit. Darum besser:

```c
float y = 1.0, t = 1.0, x = 1.0, yold = 0.0;
int i = 0;
while (y != yold) {
  i = i + 1;
  t = t * x / i;
  yold = y;
  y = y + t;
}
```

Summieren, bis sich der Summenwert (gleitpunktmäßig) nicht mehr verändert.
Vorteil: Programm identisch für **double** und **float**.

Vorlesung 4

- Funktionen (nicht rekursiv)
- **for**-Schleife
- Syntax: +=, −=, ∗=, /=, x++, x−−, #include <math.h>, **for**

Funktionen und lokale Variablen

Lokale Vereinbarung innerhalb der Funktion fabs sind unabhängig von Vereinbarungen in der Funktion main.

```
#include <stdio.h>

double fabs(double x){
  if (x < 0)
    x = -x;
  return (x);
} /* fabs */

int main(void){
  double x = -3.14;
  printf("%f\n", fabs(x));
  return (0);
} /* main */
```

Die Variable x taucht in beiden Funktionen auf, jedoch beide haben nichts miteinander zu tun! Nur der Wert von x aus der main-Funktion wird in die Variable x in fabs kopiert (→ **Call-by-Value**).

Vorlesung 5

- Arrays
- Adressen
- Zeiger
- Makros
- Syntax: A [..], A+..., ∗(A+...), ∗x, &x, **sizeof**(), #define, A [..][..]

Vorlesung 6

- Zeiger
- Makros
- Adressübergabe
- Dynamische Felder
- Syntax: exp(), scanf("%d", &j), (**int** ∗) malloc(n), free ()

Skalarprodukt (dot product)

Das Skalarprodukt zweier Vektoren $x, y \in \mathbb{R}^n$ ist

$$x^T y = \sum_{i=1}^{n} x_i y_i \in \mathbb{R}.$$

```c
#include <stdio.h>
#define N 10

double dot(int n, double *x, double *y) {
  double res = 0.0;
  while (n) {
    res += (*x++)*(*y++);
    n--;
  }
  return (res);
} /* dot */

int main(void) {
  int i;
  double p[N], q[N];
  for (i=0; i<N; i++)
    scanf("%lf %lf", &p[i], q+i);
  printf("x^T*y= %f\n", dot(N, p, q));
  return (0);
} /* main */
```

Vorlesung 7

- Rekursion
- Sortieren (Mergesort)
- `char`-Datentyp
- Syntax: **#include** <stdlib.h>, (... *) malloc(**sizeof** (...)), exit (), NULL

Dynamische Speicherallokation mit malloc

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

int main(void) {
  int n;
  double* A;
  scanf("%d", &n);
  A = (double*) malloc(n * sizeof(double));
  if (A == NULL) {
    printf("Error: no memory");
    exit(1);
  }
  return (0);
} /* main */
```

Beispiel für void*

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

void* allocate(int n) {
  void* p;
  p = malloc(n);
  if (p == NULL) {
    printf("not enough memory");
    exit(1);
  }
  return(p);
} /* allocate */

int main (void){
  int *x, n ,i;
  double *B;
  scanf("%d", &n);
  x = (int *) allocate(n * sizeof(int));
  B = (double *) allocate(2 * n * sizeof(*B));
  for (i = 0; i < n; i++) {
    *x++ = 0;
    scanf("%lf", B++);
  }
  x -= n; B -= n; /* Zeiger auf Anfang zurücksetzen */
  return (0);
} /* main */
```

Vorlesung 8

- Analyse einfacher Ausdrücke
- Makefile
- Einfacher Taschenrechner
- **switch**-Anweisung
- Strings
- Syntax: **unsigned char**, #include "myheader.h", getchar(), getc(stdin), **char** str[] = "mystring", printf("%s", str), **switch**(...) {**case** ... : ...; **default** : ...;}

IEEE-754 PrintBits

```c
#include <stdio.h>

void PrintBits (double d) {
  char *ch, i;
  unsigned char k;
  ch = ((char *)(&d))+7;
  /* ch ist ein Zeiger auf das 8-te und letzte byte
     der IEEE-754-Darstellung von d
     | |  11 bit Exponent    |  52 bit Mantisse  |
     |V|e|e|e|e|e|e|e|e|e|e|M|M|M|M|.....|M|M|M|
     63          ch->56     52|51                0| */
  for (i = 0; i < 8; i++, ch--) {
    /* alle 8 bytes absteigend (ch--) durchgehen */
    for (k = 128; k > 0; k >>= 1) {
      /* k hat zu Beginn der Schleife das bit-Muster
         0...010000000, d.h. im 8-ten bit eine 1 und
         sonst Nullen. Nach jedem Schleifendurchlauf
         wird die 1 durch die bit-Operation k>>=1 um
         ein eine Position nach rechts geschoben. */
      printf("%d", (k & (*ch)) != 0);
      /* Das & ist der bit-weise logische UND-Operator,
         d.h. k und (*ch) werden bit-weise verglichen.
         Das Ergebnis ist genau dann ungleich 0, wenn
         (*ch) an der Position, an der die 1 in k steht
         ebenfalls eine 1 hat, d.h. im j-ten Schleifen-
         durchlauf wird das j-te bit von oben ausgegeben. */
    }
    printf(" ");
  }
} /* PrintBits */

int main (void) {
  int i;
  double d = 0;
  printf("Bitte Wert eingeben: ");
  for (i = 10; i > 0; i--) {
    scanf("%lf", &d);
```

```
    PrintBits(d);
    printf("\n\n");
  }
  return 0;
} /* main */
```

Taschenrechner

```
#include <stdio.h>

double expression(char *ch);
double term(char *ch);
double factor(char *ch);
double number(char *ch);

double expression(char *ch) {
  double d;
  switch (*ch) {
    case '-':  *ch = getchar(); d = -term(ch); break;
    case '+':  *ch = getchar(); d = term(ch); break;
    default :  d = term(ch); break;
  }
  while (1)
    switch (*ch) {
      case '+':  *ch = getchar(); d = d + term(ch); break;
      case '-':  *ch = getchar(); d = d - term(ch); break;
      default :  return(d);
    }
} /* expression */

double term(char *ch) {
  double d = factor(ch);
  while (1)
    switch (*ch) {
      case '*':  *ch = getchar(); d = d * factor(ch); break;
      case '/':  *ch = getchar(); d = d / factor(ch); break;
      default :  return(d);
    }
} /* term */

double factor(char *ch) {
  double d;
  switch (*ch) {
    case '(':  *ch = getchar(); d = expression(ch); *ch = getchar(); break;
    default :  d = number(ch);
  }
  return(d);
} /* factor */

double number(char *ch) {
  double s=1, d=0;
  switch (*ch) {
    case '+':  *ch = getchar(); break;
    case '-':  *ch = getchar(); s = -1; break;
  }
  while ( ('0'<=*ch) && (*ch<='9') ) {
```

```
    d = 10*d + (*ch-'0');
    *ch = getchar();
  }
  return(s*d);
} /* number */

int main(void) {
  char ch;
  double d;
  printf("Geben Sie einen Ausdruck ein:\n");
  ch = getchar();
  while (ch!='q') {
    d = expression(&ch);
    printf("value = %f\n",d);
    ch = getchar() ;
  }
  return 0;
} /* main */
```

Vorlesung 9

- Strings
- Dateiein-/ausgabe
- Strukturierte Datentypen
- Kommandozeilenargumente
- Syntax: **sizeof**(str), sprintf, sscanf, stdin, stdout, tolower, fscanf, fprintf, fclose, File* fopen(**char** *filename, **char** *mode), (mode: "r", "w", "a"), **struct** ...{...; ... ;};

Datei Ein- und Ausgabe (file I/O)

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

int main(void){
  FILE *in, *out;
  char ch;
  double x;
  printf("I/O from/to file (y/n)? ");
  ch = getchar();                          /* <=> fgetc(stdin); */
  if (tolower(ch) == 'y') {
    in = fopen("Matrix.txt", "r");
    out = fopen("Results.txt", "w");
  } else {
    in = stdin;
    out = stdout;
  }
  fscanf(in, "%lf", &x);
  fprintf(out, "x=%f\n", x);
  fclose(in);
  fclose(out);                  /* auch, um Daten zu sichern! */
  return (0);
} /* main */
```

Vorlesung 10

- strukturierte Datentypen
- Lineare Listen
- Hash-Tabellen
- Datentyp Matrix
- Syntax: (*mypointer).mycomponent, mypointer->mycomponent, **typedef**

Strukturierte Datentypen

```
#include <stdio.h>
#include <stdlib.h> /* Umwandlung von Zahlen, Speicherverwaltung, etc */
#include <string.h> /* String-Operationen, z.B. strcpy */

/*****************************************************************
  Strukturdefinitionen
*****************************************************************/
struct sAdresse {
  char* Ort;
  int PLZ;
  char* Strasse;
  int Hausnummer;
};

struct sPerson {
  char Name[20];
  char Vorname[20];
  char Geschlecht;
  int  Alter;
  struct sAdresse Anschrift;
};

/*****************************************************************
  Initialisierungsfunktionen
*****************************************************************/
/**
 * Stellt Speicher für eine Variable vom Typ struct sPerson bereit.
 * Gibt einen Zeiger auf den Speicher zurück.
 */
struct sPerson *PersonAlloc(void) {
  struct sPerson *P; /* Zeiger auf lokale Variable vom Typ sPerson */
  /* Instantiierung von P: Mit malloc wird der Speicherplatz für
     eine Variable vom Typ struct sPerson bereitgestellt.
     Der Teil (struct sPerson*) ist eine explizite Typkonvertierung
     (type cast) auf den Datentyp von P. */
  P = (struct sPerson *) malloc(sizeof(struct sPerson));
  return P;
} /* PersonAlloc */

/**
```

```
 * Die Funktion PersonInit dient zum setzen der Attribute einer
 * Variablen vom Typ struct sPerson
 *
 * P - Zeiger auf Variable vom Typ struct sPerson, für welche
 *     die Ausprägung der Attribute geändert werden soll.
 * Name - zu setzender Name
 * Vorname - zu setzender Vorname
 * sex - zu setzendes Geschlecht
 * age - zu setzendes Alter
 * adr - Anschrift
 */
void PersonInit(struct sPerson *P, char *Name, char *Vorname, char sex,
                int age, struct sAdresse adr) {
  strcpy(P->Name, Name);
  strcpy(P->Vorname, Vorname);
  P->Geschlecht = sex;
  P->Alter = age;
  P->Anschrift = adr;
} /* PersonInit */

int main(void) {
  /* Definition einer Variablen vom Typ "struct sAdresse". Die Attribute
     werden hier bereits mit den Werten Ort = "Hamburg", PLZ = 21075,
     Strasse = "Schwarzenbergstr." und Hausnummer = 70 belegt. */
  struct sAdresse Adresse = {"Hamburg",21075,"Schwarzenbergstr.",70};
  /* Definition einer Variablen vom Typ "struct sPerson". Nur die Attribute
     Name = "Meyer", Vorname = "Max", Geschlecht = 'm' und Alter = 23
     werden gesetzt, nicht aber das Attribut Anschrift. */
  struct sPerson P1 = {"Meyer", "Max", 'm', 23};
  /* Definition eines Zeigers auf eine Variable vom Typ "struct sPerson". */
  struct sPerson *custodian;

  /* Die folgenden printf-Anweisungen zeigt, wie man auf einzelne
     Attributeder Struktur-Variablen P1 zugreift. */
  printf("einige Attribute von P1\n");
  printf("Name: %s\n", P1.Name);
  printf("Vorname: %s\n", P1.Vorname);
  printf("Geschlecht: %c\n", P1.Geschlecht);
  printf("Alter: %d\n", P1.Alter);
  /* Leere Ausgabe */
  printf("PLZ Wohnort: %d %s\n", P1.Anschrift.PLZ, P1.Anschrift.Ort);
  getchar();
  /* Setzen der Anschrift */
  P1.Anschrift = Adresse;
  printf("PLZ Wohnort: %d %s\n", P1.Anschrift.PLZ, P1.Anschrift.Ort);
  getchar();

  /* Arbeiten mit Zeigern auf Strukturvariablen */

  custodian = PersonAlloc(); /* Speicher bereitstellen */
  /* Attribute setzen */
  PersonInit(custodian, "Wehner", "Willi", 'm', 52, Adresse);
  /* Zugriff auf Attribute */
  printf("Geschlecht von custodian: %c\n", (*custodian).Geschlecht);
  /* Einfacher */
```

```
  printf("Geschlecht von custodian: %c\n", custodian->Geschlecht);
  /* Direkte Änderung der Ausprägung einzelner Attribute */
  strcpy(custodian->Name, "Djerassi");
  printf("Name von Custodian = %s\n", custodian->Name);
  return 0;
} /* main */
```

Listenelement einfügen

```
#include <stdio.h>
#include <stdlib.h> /* Umwandlung von Zahlen, Speicherverwaltung, etc */
#include <string.h> /* String-Operationen, z.B. strcpy */

/* Strukturdefinition */
struct sNameList {
  char Name[20];           /* Name einer Person in der Liste */
  struct sNameList *succ; /* Zeiger auf den Nachfolger */
};

void ListPrint(struct sNameList *z) {
  struct sNameList *buffer;
  buffer = z;
  printf("\n");
  while(z != NULL){
    printf("%s -> ",z->Name);
    z = z->succ;
  }
  printf("NULL");
  z=buffer;
} /* ListPrint */

int main(void) {
  /* Deklaration eines Zeigers pn auf eine Lineare Liste */
  struct sNameList *pn, *buffer, *z, *Elt;

  /* Ersten Namen in Liste eintragen */
  pn = (struct sNameList*) malloc(sizeof(struct sNameList));
  strcpy(pn->Name, "Lodge");
  pn->succ = NULL;

  /* Zweiten Namen in Liste eintragen */
  buffer = (struct sNameList*) malloc(sizeof(struct sNameList));
  strcpy(buffer->Name, "Hannah");
  buffer->succ = NULL;
  pn->succ = buffer;

  /* Dritten Namen in Liste eintragen */
  buffer = (struct sNameList*) malloc(sizeof(struct sNameList));
  strcpy(buffer->Name, "Wehner");
  buffer->succ = NULL;
  pn->succ->succ = buffer;

  /* Liste von vorne ausgeben */
  ListPrint(pn);
  getchar(); /* Pause */
```

```
  /* Listenelement "Carus" nach "Hannah" einfügen */

  z = pn; /* z zeigt auf erstes Listenelement mit Namen "Lodge" */
  pn = z->succ; /* pn zeigt auf zweites Listenelement mit Namen "Hannah" */
  /* Neues Listenelement erzeugen */
  Elt =(struct sNameList*) malloc(sizeof(*Elt));
  /* Neuen Namen "Carus" eintragen */
  strcpy(Elt->Name, "Carus");
  Elt->succ = pn->succ; /* Nachfolger von "Carrus" wird "Wehner" */
  pn->succ =Elt;        /* Nachfolger von "Hannah" wird "Carus" */

  /*Liste von vorne ausgeben*/
  ListPrint(z);
  getchar(); /* Pause */

  /* Zweites Listenelement "Hannah" löschen */

  pn = z->succ; /* pn zeigt auf zweites Listenelement mit Namen "Hannah" */
  z->succ = pn->succ; /* Der Nachfolger von "Lodge" ist jetzt "Carus" */
  free(pn); /* Speicher von pn freigeben */

  /* Liste von vorne ausgeben */
  ListPrint(z);
  getchar(); /* Pause */
  return 0;
} /* main */
```

Hash-Tabelle

```
#include <stdio.h>
#include <ctype.h>  /* Klassifizierung und Umwandlung von Zeichen */
#include <stdlib.h> /* Umwandlung von Zahlen, Speicherverwaltung, etc. */
#include <string.h> /* String-Operationen, z.B. strcpy */

#define HASHSIZE 256

/* Strukturdefinition */
typedef struct NameList {
  char Name[20];          /* Name einer Person in der Liste */
  struct NameList *succ; /* Zeiger auf den Nachfolger */
} sNameList;

/* Die eigentliche Hash-Tabelle,
   ein globales Array von Zeigern auf sNameList. */
sNameList *aHashTable[HASHSIZE];

/**
 * Berechnet den Hashwert zu einem String und gibt diesen zurück.
 *   str - String dessen Hashwert berechnet werden soll
 */
int hash(char *str) {
  int hashval;
  for (hashval = 0; *str; hashval += *str++);
  return (hashval % HASHSIZE);
} /* hash */
```

```c
/**
 * Fügt ein neues Element am Anfang der Liste ein und gibt einen Zeiger
 * auf den neuen Listenanfang zurück.
 *   list - Zeiger vom Typ sNameList auf Listenanfang
 *   p - Zeiger auf das neue Element
 */
sNameList* insert_front(sNameList *list, sNameList *p) {
  p->succ = list; /* Der alte Listenanfang wird zum Nachfolger des
                     neuen Elements. */
  return p; /* Das neue Element wird neuer Listenanfang. */
} /* insert_front */

/**
 * Fügt einen neues Element p mit Namen in die Hash-Tabelle ein
 */
void insert_name(sNameList *p) {
  /* Berechne den Hashwert des Namens des neuen Elements,
     um die Position in der Hash-Tabelle zu ermitteln,
     an welcher das neue Element eingefügt werden soll. */
  int hashvalue = hash(p->Name);
  /* Füge den neuen Namen in die lineare Liste an der berechneten
     Stelle in der Hash-Tabelle ein. */
  aHashTable[hashvalue] = insert_front(aHashTable[hashvalue],p);
} /* insert_name */

/**
 * Suche in der Hash-Tabelle nach einem Element mit dem Namen str.
 */
sNameList* lookup(char *str) {
  /* Analog zu insert_name: berechne den Hashwert von str,
     um die Position in der Hash-Tabelle (aHashTable[hashvalue]) zu ermitteln,
     an welcher sich ein Element mit dem Namen str nur befinden kann. */
  int hashvalue = hash(str);
  /* Suche in der ermittelten linearen Liste nach einem Element mit dem
     Namen str. */
  sNameList *pn;
  for (pn = aHashTable[hashvalue]; pn != NULL; pn = pn->succ) {
    if (strcmp(pn->Name, str) == 0) {
      return pn; /* Name str gefunden, gib Zeiger auf das Element zurück */
    }
  }
  return NULL; /* Name nicht gefunden, gibe NULL zurück */
} /* lookup */

/**
 * Gibt die Elemente einer linearen Liste in der Konsole aus,
 * inklusive abschließendem NULL.
 */
void print_list(sNameList *z) {
  while(z != NULL){
    printf("%s -> ", z->Name);
    z = z->succ;
  }
  printf("NULL\n");
```

```
} /* print_list */

/*
 * Gibt die gesamte Hash-Tabelle aus
 */
void print_table(void) {
  int i;
  printf("\n");
  printf("-----------------------------------------\n");
  printf(" Aktueller Inhalt der Hash-Tabelle: \n");
  printf("#########################################\n");
  for (i = 0; i < HASHSIZE; i++) {
    printf("%3d ", i + 1);
    print_list(aHashTable[i]);
    if (i%25 == 0 && i != 0) {
      printf("Mehr ... bitte Enter drücken");
      getchar();
    }
  }
  printf("#########################################\n");
  printf("     Ende der Tabelle erreicht     \n");
  printf("-----------------------------------------\n");
}

int main(void) {
  char str[20]; /* Speichert temporär eingelesene Namen */
  FILE *file; /* Textdatei aus der die Namen gelesen werden */
  sNameList *newElement; /* Zeiger für neues Element */
  int i;
  char jn;

  /* Initialisiere die Hash-Tabelle */
  for (i = 0; i < HASHSIZE; i++) {
    aHashTable[i] = NULL;
  }

  /* Öffne Textdatei */
  file = fopen("NamesList.txt", "r");
  if (file == NULL) {
    printf("Datei konnte nicht geoeffnet werden.\n");
    return -1;
  }

  printf("\nLese Namen ein und fuege diese in die Hash-Tabelle ein.\n");
  while (fscanf(file, "%s\n", str) != EOF) {
    /* Reserviere Speicher für neues Element */
    newElement = malloc(sizeof(*newElement));
    /* Setze den eingelesen Namen ein */
    strcpy(newElement->Name, str);
    /* Füge das neue Element in die Tabelle ein */
    insert_name(newElement);
  }

  fclose(file); /* Schließe Datei, da nicht mehr gebraucht. */
```

```
  printf("\nNamen komplett eingelesen, druecke Enter...\n");
  getchar();
  print_table();

  /* Jetzt versuchen wir, einige Namen in der Tabelle zu finden */
  while (1) {
    printf("\n\n");
    printf("Nach wem suchen Sie? Bitte Namen eingeben:\n");
    printf("(Die Eingabe \"niemand\" beendet das Programm)\n");
    scanf("\n%s",str);
    if (strcmp(str, "niemand") == 0) {
      break;
    }
    printf("Suche nach %s, Ergebnis: ",str);
    newElement = lookup(str);
    if (newElement != NULL) {
      printf("%s wurde gefunden\n", newElement->Name);
    } else {
      printf("%s wurde nicht gefunden!\n",str);
      do {
        printf("Soll der Name hinzugenommen werden? (j/n) \n");
      } while ((jn = getchar()) && (jn != 'j') && (jn != 'n'));
      if (jn == 'j') {
        /* Reserviere Speicher für neues Element */
        newElement = malloc(sizeof(sNameList));
        /* Setze den Namen des neuen Elements */
        strcpy(newElement->Name, str);
        /* Füge das Element in die Tabelle ein */
        insert_name(newElement);
        printf("Name wurde hinzugefügt\n");
        print_table(); /* Gebe neue Tabelle aus */
      }
    }
  }
  return 0;
} /* main */
```

Vorlesung 11

- Datentyp Matrix
- Zweidimensionale Felder
- Syntax: A [...][...]

Matrix als eindimensionales Array double*

```
#include <stdio.h>
#include <stdlib.h> /* Speicherverwaltung */
#include <time.h>   /* Zufallszahlen */

/* Matrix wird als Zeiger auf double realisiert */
typedef double* Matrix;

/**
 * Alloziert den benötigten Speicher für eine m mal n Matrix und gibt
 * einen Zeiger auf die Matrix zurück.
 */
Matrix matAlloc(int m, int n) {
  Matrix A;
  A = (Matrix) malloc(m * n * sizeof(double));
  if (A == NULL) {
    printf("error: not enough memeory\n");
    return NULL;
  }
  return A;
}

/**
 * Gibt den für eine Matrix allokierten Speicher wieder frei.
 */
void matFree(Matrix A) {
  free(A);
}

/**
 * Gibt die Matrix in der Konsole aus.
 */
void printMat(Matrix A, int m, int n) {
  int i, j;
  printf("\n");
  for (i = 0; i < m; i++){
    for (j = 0; j < n; j++){
      printf("%-1.3e ", A[i*n + j]);
    }
    printf("\n");
  }
}

int main(void) {
  int i, j, m = 5, n = 10;
```

```
  Matrix matA = matAlloc(m, n); /* Speicher allozieren */

  srand(time(NULL)); /* Zufallszahlengenerator initialisieren */

  /* Matrix mit Nullen initialisieren */
  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
      matA[i*n + j] = 0.0;
    }
  }
  printf("Matrix wurde initialisiert\n");
  printMat(matA, m, n); /* Matrix ausgeben */
  getchar(); /* Pause */

  /* Matrix mit Zufallswerten füllen */
  for (i = 0; i < m; i++){
    for (j = 0; j < n; j++){
      matA[i*n + j] = rand();
    }
  }
  printf("Matrix mit Zufallswerten ueberschrieben\n");
  printMat(matA, m, n); /* Matrix ausgeben */
  getchar(); /* Pause */

  matFree(matA); /* Speicher freigeben */
  return 0;
}
```

Matrix als zweidimensionales Array double**

```c
#include <stdio.h>
#include <stdlib.h> /* Speicherverwaltung */
#include <time.h>   /* Zufallszahlen */

/* Matrix wird als Zeiger auf Zeiger auf double realisiert */
typedef double* Vector; /* Zeiger auf double */
typedef Vector* Matrix; /* Zeiger auf Zeiger auf double */

/**
 * Alloziert den benötigten Speicher für eine m mal n Matrix und gibt
 * einen Zeiger auf die Matrix zurück.
 */
Matrix matAlloc(int m, int n) {
  int i;
  Matrix B;
  /* Zuerst wird Speicher für m Zeilen (Vektoren == Zeiger auf double-Arrays)
     reserviert */
  B = (Matrix) malloc(m * sizeof(Vector));
  if (B == NULL) {
    printf("error: not enough memeory\n");
    return NULL;
  }
  /* Danach wird der Speicher für die m Vektoren selbst reserviert,
     also pro Vektor wird Speicher für n double-Werte reserviert. */
  for (i = 0; i < m; i++) {
    B[i] = (Vector) malloc(n * sizeof(double));
    if (B[i] == NULL) {
      printf("error: not enough memory\n");
      return NULL;
    }
  }
  return B;
}

/**
 * Gibt den für eine Matrix allokierten Speicher wieder frei.
 *   B - Matrix, deren Speicher freigegeben wird
 *   m - Anzahl der Zeilen (Vektoren)
 */
void matFree(Matrix B, int m) {
  m--; /* Jetzt ist m höchster Zeilen-Index */
  /* Genau umgekehrt zu matAlloc, wird jetzt erst der Speicher
     der m Zeilen (Vektoren) selbst wieder frei gegeben... */
  for( ; m >= 0; m--){
    free(B[m]); /* Gebe Speicher des Vektors (n double-Werte) frei */
  }
  /* ... danach der Speicher für die m Zeilen (Vektoren). */
  free(B);
}

/**
 * Gibt die Matrix in der Konsole aus.
 */
```

```
void printMat(Matrix B, int m, int n) {
  int i, j;
  printf("\n");
  for (i = 0; i < m; i++){
    for (j = 0; j < n; j++){
      printf("%-1.3e ", B[i][j]);
    }
    printf("\n");
  }
}

int main(void) {
  int i, j, m = 5, n = 10;
  Matrix matA = matAlloc(m, n); /* Speicher allozieren */

  srand(time(NULL)); /* Zufallszahlengenerator initialisieren */

  /* Matrix mit Nullen initialisieren */
  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
      matA[i][j] = 0.0;
    }
  }
  printf("Matrix wurde initialisiert\n");
  printMat(matA, m, n); /* Matrix ausgeben */
  getchar(); /* Pause */

  /* Matrix mit Zufallswerten füllen */
  for (i = 0; i < m; i++){
    for (j = 0; j < n; j++){
      matA[i][j] = rand();
    }
  }
  printf("Matrix mit Zufallswerten ueberschrieben\n");
  printMat(matA, m, n); /* Matrix ausgeben */
  getchar(); /* Pause */

  matFree(matA, m); /* Speicher freigeben */
  return 0;
}
```

Vorlesung 12

- Schleifenabbruch
- **static**-Variablen
- Zeiger auf Funktionen
- Syntax: **break**, **continue**, **goto** mymark;, mymark: ..., **static**, **int** (∗ f)(**double** x)