C-Programmierkurs für Fortgeschrittene

Clelia Albrecht, Felix Boes, Johannes Holke

6. April 2017

Für Mama und Papa

Für Mamma und Papa

Für Mama und Papa

Hätte ich doch nur schon früher den fortgeschrittenen Programmierkurs besucht.

Dennis MacAlistair Ritchie

Habt ihr was miteinander? Falls nein, geht doch mal nen Film zusammen schauen (perks of being a wallflower, the fault in our stars zb).

Ein Teilnehmer des Kurses 2015

Inhaltsverzeichnis

1 Vorwort 2

2 Wiederholung 3
- 2.1 Funktionenpointer . . . 3
- 2.2 Strukturdefinitionen . . . 5
- 2.3 Datenstrukturen . . . 5
- 2.4 Pointer auf Strukturen . . . 7
- 2.5 Typdefinitionen . . . 7
- 2.6 Anwendung: Verkettete Listen . . . 9
- 2.7 Anwendung: Hashtabellen . . . 12

3 Externe Bibliotheken 16
- 3.1 Aber das habe ich doch ganz schnell selber geschrieben! . . . 16
- 3.2 Installieren und Linken . . . 17
- 3.3 Beispiel: die joelixblas Bibliothek . . . 18
  - 3.3.1 Installation . . . 18
  - 3.3.2 Kompilieren eines Testprogramms . . . 19
  - 3.3.3 Das CSR-Format für Matrizen . . . 20
  - 3.3.4 Anwendung: Das Poissonproblem . . . 22

4 Multithreading mit OpenMP 27
- 4.1 Forking und Joining . . . 28
- 4.2 Sektionen . . . 29
- 4.3 Schleifen . . . 29
- 4.4 Barrieren . . . 30
- 4.5 Shared Memory . . . 31
- 4.6 single, master und critical . . . 33
- 4.7 Locks . . . 34

5 Ausblick: Parallelisierung mit MPI 36
- 5.1 Warum „Message Passing“? . . . 36
- 5.2 Hello World und Kompilierung . . . 37
- 5.3 rank und size . . . 38
- 5.4 Send und Recv . . . 38
- 5.5 Broadcast und Reduce . . . 41

6 Debugging 43
- 6.1 gdb . . . 44
- 6.2 Valgrind . . . 46

7 Weiteres nützliche Sprachkonstrukte **50**
7.1 Kommandozeilenargumente 50
7.2 Bedingte Auswertung 50
7.3 Konstante Variablen 51

A Referenzen **54**
A.1 Referenz `<math.h>` 54
A.2 Referenz `<time.h>` 55
A.3 Referenz `<stdlib.h>` 57
A.4 Referenz `<limits.h>` 58
A.5 Referenz `<float.h>` 58

1 Vorwort

Dieses Skript ist Begleitmaterial zu dem von Clelia Albrecht, Felix Boes und Johannes Holke an der Uni Bonn gehaltenen **C**-Programmierkurs für Fortgeschrittene.

Es enthält Teile des von Jesko Hüttenhain und Lars Wallenborn geschriebenen Skriptes zum Anfängerprogrammierkurs **C** [4]. Inhaltlich baut dieses Skript auf dem von Clelia Albrecht und Johannes Holke in den Wintersemestern 2015/2016 und 2016/2017 an der Universität Bonn gehaltenen **C**-Programmierkurs für Erstsemester auf [3].

Ziel dieser Vorlesung ist die Vermittlung einiger fortgeschrittener Programmierkonzepte, die bei wissenschaftlicher Programmierung häufig auftauchen oder sehr hilfreich sein können. Dazu gehören abstrakte Datentypen wie Listen, Hashtabellen und speichersparende Matrixformate genau so wie die Nutzung externer Bibliotheken und Hilfsprogrammen zum debuggen wie `valgrind` und dem `gdb`.

Das Skript fasst die wichtigsten Themen aus der Vorlesung zusammen und dient als zusätzliche Sammlung von Referenzen. Es ersetzt nicht den Besuch der Vorlesung und erst recht nicht die Bearbeitung der zugehörigen Übungszettel, die wir dem Leser besonders ans Herz legen.

Das Skript ist wie folgt aufgebaut: In **Kapitel 2** werden einige Sprachkonstrukte, die zum Ende des Anfängerkurses hin besprochen wurden, noch einmal wiederholt, insbesondere werden `structs` und Funktionenpointer noch einmal aufgegriffen. Als Anwendung nutzen wir diese Sprachkonstrukte für Listen und Hashtabellen.

In **Kapitel 3** beschäftigen wir uns mit der Verwendung externer Bibliotheken, da viele Programme, die man in der wissenschaftlichen Programmierung schreibt, durch gute externe Bibliotheken schneller, übersichtlicher und besser werden. Wir erläutern den Umgang mit externen Bibliotheken anhand einer von uns geschriebenen, kleinen Bibliothek, die den Einstieg erleichtert.

Kapitel 4 behandelt die **OpenMP**-Bibliothek, die das Parallelisieren des Programms durch Multithreading ermöglicht. Dazu erläutern wir die gebräuchlichsten Funktionen und diskutieren, in welchen Fällen Multithreading nützlich und in welchen vielleicht eher problematisch ist.

Ein sehr großer Teil des Programmierens ist das Debuggen. Im Anfängerprogrammierkurs haben wir bereits das Debuggen mit Hilfe der `printf`-Funktion diskutiert, häufig kommt man jedoch mit Hilfsprogrammen zum Debuggen schneller auf den Fehler. Zwei dieser Debugging-Hilfen, nämlich `valgrind` und den `gdb`, stellen wir in **Kapitel 6** vor.

Weitere sehr nützliche, aber kleinere Sprachkonstrukte und -konzepte haben wir in **Kapitel 7** gesammelt.

2 Wiederholung

In diesem Kapitel fassen wir einige Sprachkonzepte zusammen, welche schon im vorangegangenen Einführungskurs [3] behandelt wurden und für den Inhalt dieses Kurses wichtig sind. Insbesondere werden wir Funktionenpointer wiederholen, sowie `structs` und abstrakte Datentypen. Als neues Beispiel für abstrakte Datentypen werden wir Hashtabellen einführen.

2.1 Funktionenpointer

Funktionen werden vom Compiler in Maschinencode übersetzt, der letzten Endes bei der Ausführung auch im Speicher liegt, und dort von der CPU als Befehle interpretiert wird. Daher haben auch Funktionen bei der Ausführung eine Adresse im Speicher. In C ist es möglich, die Adresse einer Funktion zu ermitteln und diese in einer Variablen zu speichern – in einem sogenannten *Funktionenpointer*. Einen Funktionenpointer deklariert man wie folgt:

```
RÜCKGABETYP (*NAME)(PARAMETERTYP1, ..., PARAMETERTYPn);
```

Als Beispiel würde die variable `function` im folgenden Programmcode eine Variable sein, die Pointer auf Funktionen speichert, welche zwei `double`-Variablen als Argumente erwarten und ein `double` als Rückgabewert liefern. Der Hit: Wir können diesen Pointer danach wieder dereferenzieren und die referenzierte Funktion aufrufen!

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main() {
5    /* Deklaration des Funktionenpointers: */
6    double (*function)(double,double);
7    /* Wir weisen function die Adresse von pow zu: */
8    function = &pow;
9    /* Nun dereferenzieren wir function, erhalten
10      damit pow zurück und rufen es auf: */
11   printf("%f\n", (*function)(2.0,3.0));
12   return 0;
13 }
```

Damit nicht genug, wir wollen Funktionenpointer an andere Funktionen übergeben. Im folgenden Beispiel beschreiben wir eine Funktion, welche die Trapezsumme einer Funktion $f : \mathbb{R} \to \mathbb{R}$ im Intervall $[a, b]$ berechnet:

```
1  #include <stdio.h>
2  #include <math.h>
3
4  /* Der Einfachheit halber definieren wir einen neuen
```

```
 5     Typ fuer Funktionen, die ein double erwarten und
 6     ein double liefern: */
 7 typedef double (*REALFUNCTION)(double);
 8
 9 double Trapez(REALFUNCTION f, double a, double b) {
10   if (a > b) return Trapez(f,b,a);
11   return (b-a) * ( (*f)(b) + (*f)(a) ) / 2;
12 }
13
14 int main() {
15   printf("%f\n", Trapez(&exp,1,3) );
16   printf("%f\n", Trapez(&log,1,3) );
17   printf("%f\n", Trapez(&sin,1,3) );
18   return 0;
19 }
```

Mit Hilfe von Funktionenpointern können wir die `qsort`-Funktion aus `<stdlib.h>` verwenden:

```
1 void qsort(
2   void          *array,
3   unsigned long  count,
4   unsigned long  size,
5   int (*compare) (const void *a, const void *b)
6 );
```

Diese Funktion erwartet an der Adresse `array` genau `count` Elemente, die jeweils `size` viele Speicherzellen in Anspruch nehmen. Der Funktionenpointer `compare` verweist auf eine Funktion, mit der zwei solche Elemente verglichen werden können. Dabei wird die Rückgabe von `compare` wie folgt interpretiert:

Rückgabe	**Bedeutung**
`0`	Die Elemente gelten als “gleich”.
`1`	Das Element bei `a` ist “größer” als das bei `b`.
`-1`	Das Element bei `a` ist “kleiner” als das bei `b`.

Mit Hilfe dieser Informationen sortiert `qsort` das Array bei `array` aufsteigend. Ein einfaches Beispiel, um ein `int`-array absteigend zu sortieren:

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3
 4 int absteigen(const void *a, const void *b) {
 5   int da = *((int const *)a);
 6   int db = *((int const *)b);
 7   if (da < db) return 1;
 8   else if (db < da) return -1;
 9   else return 0;
10 }
```

```
11
12 int main() {
13   int i;
14   int a[10] = { 23, 1, 23, 576, 3, 97, 7, 743, 2, 98 };
15   qsort( a, 10, sizeof(*a), &absteigen );
16   for (i=0; i<10; i++) printf("%i ", a[i]);
17   return 0;
18 }
```

2.2 Strukturdefinitionen

Strukturen werden verwendet, um mehrere, im Speicher hintereinander angeordnete Variablen zusammenzufassen. Mit folgender Syntax definiert man eine Strukturtyp und damit einen ganz *neuen Datentyp*:

```
struct STRUKTURNAME {
  DATENTYP 1 FELDNAME 1;
  DATENTYP 2 FELDNAME 2;
    ...
  DATENTYP n FELDNAME n;
};
```

Eine solche Strukturdefinition steht außerhalb von Funktionscode meist am Anfang des Quellcodes (bzw. in der Headerdatei eines Moduls). Man kann im folgenden Code nun `struct STRUKTURNAME` als Datentyp verwenden – Variablen dieses Typs heißen *Strukturen* und enthalten mehrere "Untervariablen", welche auch als *Felder* bezeichnet werden. Um auf die Felder einer Struktur zuzugreifen, verwendet man den *Punktoperator* :

```
STRUKTURNAME.FELDNAME
```

Enthält eine Struktur `X` ein Feld von Typ `T` mit Namen `y`, so ist `X.y` die in `X` enthaltene Variable vom Typ `T`.

2.3 Datenstrukturen

Der Begriff "Datenstruktur" bezieht sich nicht allein auf die Definition einer Struktur im obigen Sinne. Um neue Datentypen zu erstellen, braucht man auch Operatoren, um auf diesen Datentypen zu arbeiten. Ein Beispiel hier:

```
1 #ifndef _COMPLEX_H
2 #define _COMPLEX_H
3
4 /* Der neue Datentyp struct COMPLEX repräsentiert eine komplexe
     Zahl, indem wir Real- und Imaginärteil separat abspeichern. */
```

```
5  struct COMPLEX {
6    double real;
7    double imag;
8  };
9
10 /* multipliziere zwei komplexe Zahlen: */
11 struct COMPLEX mul( struct COMPLEX alpha, struct COMPLEX beta );
12
13 /* addiere zwei komplexe Zahlen: */
14 struct COMPLEX add( struct COMPLEX alpha, struct COMPLEX beta );
15
16 /* dividiere zwei komplexe Zahlen durcheinander: */
17 struct COMPLEX div( struct COMPLEX alpha, struct COMPLEX beta );
18
19 /* bilde das negative einer komplexen Zahl: */
20 struct COMPLEX neg( struct COMPLEX alpha );
21
22 /* potenziere eine komplexe Zahl mit einer ganzen Zahl: */
23 struct COMPLEX pot( struct COMPLEX alpha, int n );
24
25 #endif
```

Listing 1: Die komplexen Zahlen mit Rechengesetzen (complex.h)

```
1  #include "complex.h"
2
3  /* wir benötigen die Strukturdefinition von struct COMPLEX. Dies ist
      ein weiterer Grund, warum die Quellcodedatei ihre zugehörige
     Headerdatei für gewöhnlich einbindet. */
4
5  struct COMPLEX mul ( struct COMPLEX alpha, struct COMPLEX beta ) {
6    struct COMPLEX gamma;
7    gamma.real = alpha.real * beta.real - alpha.imag * beta.imag;
8    gamma.imag = alpha.real * beta.imag + alpha.imag * beta.real;
9    return gamma;
10 }
11
12 /* ... weitere Definitionen */
```

Listing 2: Die komplexen Zahlen mit Rechengesetzen (complex.c)

Man implementiert solche Datenstrukturen mit ihren Operatoren nun zusammen in ein Modul, damit die Verwendung des neuen Datentyps in vielen verschiedenen Programmen möglich wird. Ein Programm könnte das Modul nun in etwa so verwenden:

```
1  #include <stdio.h>
2  #include "complex.h"
3
4  int main() {
```

```
5   struct COMPLEX c = {0.6,-0.8}, d = c;
6     /* Berechne das Quadrat des Betrags von c: */
7   d.imag = -d.imag;
8   d = multiply(d,c);
9   printf("%5.3lf%+5.3lfi\n",d.real, d.imag);
10  return 0;
11 }
```

Listing 3: Anwendung komplexer Zahlen

Man sollte sich zum Zeitpunkt der Programmierung einer Datenstruktur vollkommen klar darüber sein, wie ein Programm später mit dieser Datenstruktur arbeiten soll. Manchmal ist es sogar am günstigste zuerst ein kleines Programm zu schreiben, dass die Datenstruktur verwendet (welches man ohnehin braucht, um sie zu testen) und dann erst mit Programmierung der Datenstruktur selbst zu beginnen.

2.4 Pointer auf Strukturen

Genau wie Arrays können Strukturen unter Umständen sehr viel Speicherplatz beanspruchen. Häufig kann man sich dann darauf beschränken, nur einen Pointer zu übergeben. Möchte man nun auf ein Feld `y` einer Struktur zugreifen, die durch einen Pointer `p` gegeben ist, so muss man einfach den Pointer dereferenzieren und auf die dadurch zurückgewonnene Strukturvariable durch den Punktoperator zugreifen:

```
(*p).y
```

Der Punktoperator bindet allerdings stärker als der Dereferenzierungsoperator (man sagt auch, der Punktoperator hat eine höhere *Operatorpräzedenz*, genau wie man auch sagt "Punkt- vor Strichrechnung"). Das bedeutet, dass die gerade gesetzten Klammern notwendig sind. Die Schreibweise `*p.y` entspräche `*(p.y)`, was, da der Punktoperator nicht auf Pointer definiert ist, syntaktisch falsch wäre. Da dies etwas umständlich ist, wurde in C dafür die abkürzende Schreibweise `p->y` eingeführt. Um unser Beispiel zu erweitern:

```
1 double Im(struct COMPLEX *c) { return c->imag; }
2 double Re(struct COMPLEX *c) { return c->real; }
```

Listing 4: Real- bzw. Imanginärteil einer komplexen Zahl zurück zu geben

Man bezeichnet diesen Operator auch als den *Pfeiloperator*.

2.5 Typdefinitionen

In C ist es nicht nur möglich, sich neue Datentypen auf die oben genannte Weise zu definieren, man kann auch jedem bestehenden Datentyp einen weiteren Namen zuweisen. Dies ist möglich durch die Verwendung von `typedef`:

```
typedef SCHABLONE;
```

Die SCHABLONE hat die Syntax einer Variablendeklaration (ohne Initialisierung). Der Name der Variablen, den wir bei dieser "Deklaration" angeben ist der so entstandene Datentyp. Wird er vom Compiler in einer Variablendeklaration gefunden so wird er in der SCHABLONE durch den deklarierten Variablennamen ersetzt und an dieser Stelle eingefügt. Hier ein Beispiel:

```
1 typedef double POINT[2];
2 POINT p = { 2, 4 };
```

Der definierte Datentyp heißt POINT. Bei der Variablendeklaration wird der Name des Typs in der Schablone durch den Namen der deklarierten Variable (hier p) ersetzt. Damit ist obiger Code äquivalent zu

```
1 double p[2] = { 2, 4 };
```

Es ist gegebenenfalls wichtig, sich klarzumachen, dass typedef sich bei Deklaration mehrerer Variablen in einem Statement freundlich verhält:

```
1 POINT p = { 2, 4 }, q = { 1, 7 };
```

wird zu

```
1 double q[2] = { 1, 7 };
2 double p[2] = { 2, 4 };
```

Zunächst werden also alle Kommata expandiert und einzelne Variablendeklarationen aus dem Statement gemacht und *danach* die Schablone angewandt. Beachte:

```
typedef char *STRING;    --wird zu-->    char *s1;
STRING s1, s2;                           char *s2;
```

Während andererseits:

```
char *s1,s2;    --wird zu-->    char *s1;
                                char  s2;
```

Besonders im Zusammenhang mit Strukturdefinitionen stellen Typdefinitionen eine ernsthafte Erleichterung dar. Man betrachte etwa das wie folgt modifizierte Beispiel 1:

```c
#ifndef _COMPLEX_H
#define _COMPLEX_H

struct COMPLEX {
  double real;
  double imag;
};

typedef struct COMPLEX COMPLEX;

COMPLEX mul(COMPLEX alpha, COMPLEX beta);
COMPLEX add(COMPLEX alpha, COMPLEX beta);
COMPLEX div(COMPLEX alpha, COMPLEX beta);
COMPLEX neg(COMPLEX alpha );
COMPLEX pot(COMPLEX alpha, int n );

#endif
```

Man kann Typdefinitionen mit Strukturdefinitionen direkt verbinden, indem man schreibt

```
typedef struct {
  DATENTYP 1 FELDNAME 1
  DATENTYP 2 FELDNAME 2
  ...
  DATENTYP n FELDNAME n
} NAME;
```

Danach kann man dann neue Variablen vom Typ NAME deklarieren, welche der Struktur entsprechen. Beispiel:

```c
typedef struct {
  double real;
  double imag;
} COMPLEX;
```

2.6 Anwendung: Verkettete Listen

Wir wollen nun das Erlernte verwenden, um eine bekannte und wichtige Datenstruktur in C zu implementieren. Als Datenstruktur bezeichnen wir ein Schema, nach dem Daten einer gewissen Form im Computer gespeichert werden. Dazu gehören auch eine Reihe von Operationen auf der Datenstruktur, um diese zu modifizieren. Ein nahe liegendes Schema, um Daten im Computer zu Speichern, ist uns bereits bekannt: Arrays. Auch werden alle notwendigen Operationen auf Arrays von der Sprache C bereits in Form von elementarer Pointerarithmetik zur Verfügung gestellt.

Es gibt jedoch eine weiteres Schema, um einen sortierten Satz von mehreren Variablen zu speichern. Dieses Schema wird als (doppelt) verkettete Liste bezeichnet. In einer verketteten Liste wird jede Variable x_i in einen sogenannten Knoten (engl. "node") eingebettet, der neben x_i auch einen Rückwärtspointer p_i (previous) auf x_{i-1} und einen Nachfolgerpointer $\texttt{next}_i$ (next) auf x_{i+1} enthält. Der erste Rückwärtspointer und der letzte Nachfolgerpointer sollen der Nullpointer sein. Diese Knoten modellieren wir als Strukturen:

```
1 typedef struct NODE {
2   struct NODE  *next;
3   struct NODE  *prev;
4   double        data;
5 } NODE;
```

Wir können also in der Definition der Struktur bereits Pointer auf die Struktur verwenden, die gerade im Begriff ist, definiert zu werden[1]. Jeder Knoten ist nun eine Struktur, die einen Pointer auf den nächsten Knoten und die gespeicherte Variable enthält. Die Liste selbst repräsentieren wir auch durch eine Struktur, welche Pointer auf das erste und letzte Element speichert:

```
1 typedef struct {
2   NODE *first;
3   NODE *last;
4 } LIST;
```

Eine Liste soll (graphisch dargestellt) immer folgenden Aufbau haben:

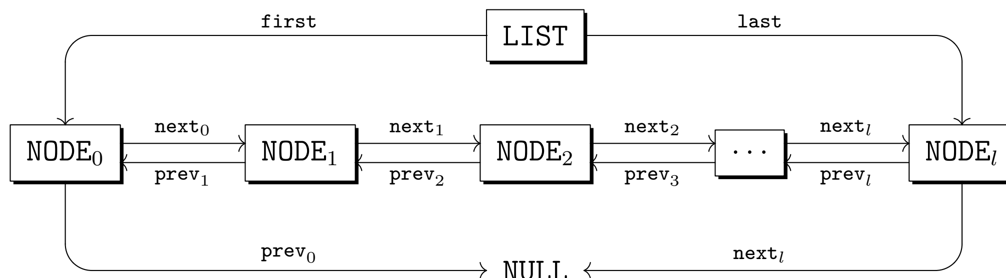
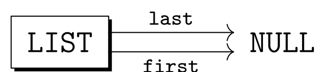


Abbildung 1: Darstellung einer doppelt verketteten Liste

Wir erlauben weiterhin, dass sowohl `first` als auch `last` der Nullpointer sind und wollen in diesem Falle sagen, die Liste sei leer:



[1] Dies liegt daran, dass Pointer ohnehin immer die gleiche Größe haben und lediglich Adressen speichern.

Nun haben wir die Datentypen definiert, doch wir benötigen noch Funktionen, um mit ihnen sinnvoll arbeiten zu können. Die gesamte Headerdatei für das Modul, dass eine Listenstruktur bereitstellt, könnte etwa wie folgt aussehen:

```
1  #ifndef LIST_H
2  #define LIST_H
3
4  typedef struct NODE {
5    struct NODE  *next;
6    struct NODE  *prev;
7    double        data;
8  } NODE;
9
10 typedef struct {
11   NODE *first;
12   NODE *last;
13 } LIST;
14
15 LIST *list_create();        /* Liste erstellen */
16 void  list_free(LIST *L);   /* Speicher freigeben */
17
18 /* Füge hinter cursor einen neuen Knoten in die Liste ein. Falls
19    cursor gleich NULL ist, füge am Anfang der Liste ein. Gibt
20    einen Pointer auf  das neu eingefügte Element zurück, oder NULL
21    im Fehlerfall. */
22 NODE *list_insert(LIST *L, NODE *cursor, double data);
23
24 /* Lösche einen Knoten aus der Liste. */
25 void list_delete(LIST *L, NODE *del);
26
27 #endif
```

Es ist eine hilfreiche Übung, dieses Modul selbst zu implementieren. Wir wollen zum Schluss noch zeigen, wie man alle Elemente einer Liste fachgerecht mit einer Schleife durchläuft:

```
1 LIST *L;
2 NODE *p;
3 /* Liste initialisieren, Einträge erzeugen, etc. */
4
5 for (p=L->first; p; p = p->next) {
6   /* arbeite mit Knoten p */
7 }
```

Der Pointer `p` zeigt zu Beginn der `for`–Schleife auf das erste Element der Liste und springt in jedem Schritt zum nächsten. Sobald man am Ende angekommen ist, wird `p` der Wert `NULL` zugewiesen, da der `next`–Pointer des letzten Elements dorthin verweist: Die Schleife bricht ab.

2.7 Anwendung: Hashtabellen

Verkettete Listen haben unter anderem den Vorteil, dass Dateneinträge beliebig hinzugefügt oder gelöscht werden können. Ein beliebiger Zugriff auf einen Dateneintrag hat bei einer verketteten Liste der Länge n jedoch Zugriffszeit $\mathcal{O}(n)$. Wenn man in einer Situation eine Speicherstruktur benötigt in der häufig auf einzelne Einträge zugegriffen werden muss, so bietet sich als Alternative eine Hashtabelle an, mit dieser ist der Zugriff auf einen Eintrag (bei gut gewähler Hashfunktion, siehe unten) in konstanter Zeit möglich. Im Gegensatz zu Listen können die Einträge in einer Hashtabellen jedoch nicht ohne weiteres sortiert werden.

Hashtabellen kann man zum Beispiel gut zur Verwaltung von Datenbanken einsetzen, beispielsweise könnte man die Nutzerprofile eines sozialen Netzwerkes in Form einer Hashtabelle speichert. Weiterhin kommen Hashtabellen in vielen verschiedenen Algortithmen zum Einsatz.

Wir besprechen nun im Detail, wie Hashtabellen funktionieren. Ähnlich wie bei Listen sind die Dateneinträge sogenannte Knoten. Diese enthalten die zu speichernden Daten sowie einen sogenannten „Schlüssel", zumeist ein Integer, der den Knoten eindeutig identifiziert. Ein Knoten für eine Tabelle, in der wir `double`-Werte speichern wollen sieht also so aus:

```
1 typedef struct {
2   int key;      /* Der Schluessel */
3   double data; /* Die Daten */
4 } HASH_NODE;
```

Listing 5: Die Knoten einer Hastabelle

Eine einfache Version der Hashtabelle besteht aus einem Array `H` der Länge n von verketteten Listen `H[i]`. Die Einträge der Listen sind `HASH_NODE`s.

Desweitern gehört zu der Definition einer Hashtabelle die sogenannte Hashfunktion f, dies soll eine Funktion sein, welche zu jedem Schlüssel einen Integerwert zwischen 0 und $n-1$ liefert. Der Wert $f(s)$ für einen Schlüssel s wird auch „Hashwert" von s genannt. Der im Allgemeinen der Raum der möglichen Schlüssel deutlich größer ist als n, ist f für gewöhnlich eine surjektive Funktion. Desweiteren sollte f in $\mathcal{O}(1)$ laufen, also nur konstante Zeit benötigen, unabhängig vom Wert des Schlüssels.

Das Einfügen eines neuen `HASH_NODE` T funktioniert nun wie folgt:

1. Berechne $i = f(\texttt{T.key})$.

2. Hänge T an den Anfang der Liste `H[i]` an.

Und das Suchen nach dem `HASH_NODE` mit dem gegebenen Schlüssel s:

1. Berechne $i = f(\texttt{s})$.

2. Suche in der Liste `H[i]` nach dem Knoten mit Schlüssel s.

Wie man sieht ist die Laufzeit des Einfügens konstant. Das Suchen nach einem Eintrag ist linear in der Länge von `H[i]`. Es kommt also darauf an, die Hashfunktion f und die größe von n geschickt zu wählen, so dass die Länge der einzelnen Listen nicht großwird. Ist die Länge nach oben beschrnkt (unabhängig von der Anzahl der Elemente in der Hashtabelle), dann hat das Suchen Laufzeit $\mathcal{O}(1)$.

Beispiel 1. Sind die Schlüssel Integer und kommen ungefähr gleich verteilt in den Daten vor, so ist eine einfache Wahl für f die Rechnung modulo n:

```
1 int hashfunktion (int s, int n) {
2  return s % n;
3 }
```

Beispiel 2. Hashtabellen eignen sich auch besonders gut für Schlüssel die keine Integer sind. Stellen wir uns ein soziales Netzwerk wie `knuddels.de` vor. Ein Dateneintrag für einen Account besteht dann aus dem Usernamen sowie allen gespeicherten Daten (Freunde, Gruppen, Nachrichten, etc.).

```
1 typedef struct {
2   char *    username; /* Der Username */
3   struct    account;  /* Enthaelt die wichtigen
4                           Accountinformationen */
5 } knuddelseintrag;
```

Eine einfache Hashfunktion für ist zum Beispiel alle Zeichen als `int` aufzuaddieren. Da der Wert zwischen 0 und n sein muss, rechnen wir am Ende natürlich modulo n.

```
1 int hashfunktion (char * username, int n) {
2  int pos, value = 0;
3
4  for (pos = 0;pos < strlen (username);pos++) {
5    value += (int) username[pos];
6  }
7  return value % n;
8 }
```

Mit Hashtabellen kann man also die sogenannten assoziativen Arrays implementieren. Ein „Index“ ist hierbei ein String und kein normaler Integer. Als würde man für den Usernamen `"Joelix"` schreiben `array["Joelix"]` [2]

Es gibt einige Möglichkeiten Hastabellen effizienter zu machen. Eine Möglichkeit ist, die Länge des Arrays n variabel zu gestalten und zum Beispiel immer bei ungefähr der Hälfte der Anzahl der Einträge in der Tabelle zu halten. Weitere

[2] Dies ist kein legaler `C`-Code und dient nur der Veranschaulichung.

Möglichkeiten beschäftigen sich damit anders mit Kollisionen umzugehen, also dem Fall, wenn zwei Schlüssel den Selben Hashwert liefern.

Zum Schluss möchten wir noch zeigen, wie eine Headerdatei zu einem Modul aussehen könnte, welches Hashtabellen mit `double` Daten implementiert.

```
1  #ifndef HASHTABLE_H
2  #define HASHTABLE_H
3
4  #include "list.h" /* Wir benutzen die verketteten
5                       Listen von weiter oben. */
6
7  struct HASH_TABLE;  /* Ist weiter unten definiert. */
8
9  /* Ein Knoten der Hashtabelle. */
10 typedef struct {
11   int key;     /* Der Schluessel */
12   double data; /* Die Daten */
13 } HASH_NODE;
14
15 /* Die vom User definierte Hashfunktion, neben einem Schluessel
16  * ist noch die Hashtabelle selbst der Input. */
17 typedef int (*hashfunc) (int key, struct HASH_TABLE * H);
18
19 typedef struct HASH_TABLE {
20   LIST     **buckets;      /* Das Array von Listen H[i] */
21   int        num_buckets;  /* Die Laenge des Arrays */
22   hashfunc   hashfunktion; /* Die Hashfunktion f */
23 } HASH_TABLE;
24
25 /* Erstelle eine Hashtabelle mit vorgegebener Anzahl an Buckets
26  * und Hashfunktion. */
27 HASH_TABLE * hashtable_create (int num_buckets, hashfunc f);
28
29 /* Gebe den Speicher einer Hashtabelle wieder frei. */
30 void         hashtable_free (HASH_TABLE *H);
31
32 /* Erstelle einen neuen Knoten und fuege ihn in eine Hashtabelle
33  * ein. Gibt einen Pointer auf den neu eingefuegten Knoten zurueck,
34  * oder NULL bei einem Fehler. */
35 HASH_NODE * hashtable_insert (HASH_TABLE *H, int Schluessel,
36                               double data);
37
38 /* Suche den Knoten mit vorgegebenem Schluessel in einer
39  * Hashtabelle. Gibt einen Pointer auf den Knoten zurueck, wenn
40  * dieser in der Tabelle gespeichert war, NULL sonst. */
41 HASH_NODE * hashtable_get (HASH_TABLE *H, int Schluessel);
42
43 /* Loesche den Knoten mit vorgegebenem Schluessel aus der Tabelle.
       */
44 void hashtable_delete (HASH_TABLE *H, int Schluessel);
45
```

```
46 #endif
```

3 Externe Bibliotheken

Schreibt man ein Programm oder arbeitet and einem größeren Projekt, so kommt es immer wieder vor, dass man sich vor der Aufgabe sieht ein „Standardproblem“ zu lösen. Damit meinen wir an dieser Stelle ein Problem, mit welchem sich vermutlich schon einmal ein anderer Programmierer auseinander gesetzt hat.

Klassische Beispiele für „Standardprobleme“ sind zum Beispiel grundlegende Operationen der linearen Algebra, wie beispielsweise Matrixmultiplikationen, Skalarprodukte und die Berechnung von Matrix- und Vektornormen. In vielen Anwendungen hat man es mit unterschiedlich dicht besetzten Matritzen zu tun, für die man – für eine möglichst effiziente Implementierung – jeweils ein eigenes Datenformat schreiben sollte.

Befindet man sich in so einer Situation, so lohnt es, zu schauen, ob es eine Lösung des Problems nicht in einer externen Bibliothek gibt.

Eine Bibliothek ist im Grunde genommen nichts anderes als eine Sammlung vorkompilierter Objektdateien, welche man in seinem Programm einbinden und Nutzen kann. Diese Sammlung kommt im Regelfall in einer Archivdatei mit Endung '.a', '.so' oder '.la'.

Wir möchten in diesem Kapitel näher auf die Nutzung von externen Bibliotheken eingehen und den Umgang mit ihnen am Beispiel der `joelixblas` Bibliothek lernen.

3.1 Aber das habe ich doch ganz schnell selber geschrieben!

Gerade wenn man bisher noch keine größeren Projekte implementiert hat, stellt sich bei der Nutzung externer Bibliotheken oft die Frage: Wozu der Aufwand? Das kann ich doch auch selber programmieren?

Tatsächlich erscheint die Einarbeitung in eine externe Bibliothek häufig als überproportional aufwändig. Obwohl es durchaus viele sehr gut dokumentierte Bibliotheken mit hervorragenden Benutzerhandbüchern gibt, kann man auch Pech haben und auf eine eher unübersichtliche dokumentierte Bibliothek treffen, bei der die beabsichtigte Verwendung nicht schnell ersichtlich ist.

Das kann frustrierend sein und dafür sorgen, dass die Aussicht, alles selber zu schreiben und dann wenigstens das Interface genau zu kennen sehr verlockend wird.

Natürlich kann man dieser Versuchung auch nachgeben, aber es lohnt sich kurz über folgende Argumente nachzudenken:

1. Erstens spart man sich die Zeit für die tatsächliche Implementation der benötigten Routinen. Und diese Zeit kann deutlich mehr sein als der Aufwand, sich in ein fremdes Interface einzulesen (zumindest wenn dies hinreichend gut dokumentiert ist). Anders gesagt: auch wenn es nicht sonderlich schön

ist, sich zwei Tage in eine fremde Bibliothek einzulesen um dann einen Algorithmus in wenigen Tagen zu implementieren, kann sich dies zeitlich lohnen, wenn die Alternative ist, die benötigten Routinen in mehreren Wochen zu schreiben.

2. Zweitens könnte die Verwendung einer bereits existierenden Bibliothek das Programm deutlich effizienter machen. Es gibt sehr viele Bibliotheken (open source oder auch komerziell), an denen ganze Entwicklerteams über Jahre arbeiten und die immer weiter auf genau die zu lösende Aufgabe optimiert werden. Die Wahrscheinlichkeit, dass man die gleiche Effizienz in einer annehmbaren Zeit erreicht, ist verschwindend klein.

3. Drittens lohnt es sich sehr darüber nachzudenken, was man in Zukunft mit seinem Programm vorhat. Als Beispiel: man schreibt ein serielles Programm, dass eine Aufgabe möglichst effizient lösen soll. Möchte man es in Zukunft eventuell parallelisieren oder auf andere Art anpassen? Wenn man alles selbst implementiert hat, muss man dann auch alle Algorithmen selbst anpassen, was je nach Aufgabe sehr aufwändig werden kann. Nutzt man von Anfang an eine Bibliothek, die diese Anpassungen bereits implementiert hat, ist der Aufwand ungleich kleiner.

Es kann also einige Situationen geben, in denen sich die Verwendung von Bibliotheken lohnt. Im folgenden widmen wir uns zunächst allgemeinen Prinzipien der Bibiliotheksnutzung, bevor wir diese anhand eines Beispiels näher erläutern.

3.2 Installieren und Linken

Um die eigenen Programme mit einer externen Bibliothek zu verbinden muss man dem Kompiler mitteilen, wo die Bibliothek und wo die Headerdateien zu finden sind. Eventuell muss vor dem Nutzen einer Bibliothek diese erst installiert werden.

Das Allererste, was man in diesem Fall tun sollte, ist sich durchzulesen, was die Ersteller der Biblithek vorgeben. Man wirft also einen Blick ins Manual der Bibliothek. Dieses findet man meistens direkt auf der Homepage der Bibliothek oder zusammen mit der Bibliotheksdatei. Im Manual steht in den meisten Fällen gut und genau beschrieben, wie man die entsprechende Bibliothek vernünftig einrichtet und Programme kompiliert, die sie benutzen.

Findet man kein Manual, so könnte die Information auch in einer `README` oder `INSTALL` Datei stehen, welche mit der Bibliothek ausgeliefert wurde.

Sollte im Manual nichts dazu stehen, so gibt es folgenden Standardweg: Lokalisiere den Ordner, in dem sich die '.a', '.so', bzw. '.la' Datei befindet. Wir nehmen an, dieser Ordner ist `/path/to/lib/`. Lokalisiere den Ordner, in dem sich die Headerdateien befinden, die man zur Benutzung der Bibliothek benötigt. Nor-

malerweise heißt dieser Ordner `include`. Wit nehmen also `/path/to/include` an.

Man teilt dem Compiler nun mit Hilfe des Flags `-lBIBNAME` mit, gegen welche Bibliothek gelinkt werden soll. Der Leser erinnert sich an dieser Stelle vielleicht an die `math`-Bibliothek und dass man hier `-lm` benutzt hat. `BIBNAME` ist der erste Teil, des Namens der Archivdatei. Also entweder `BIBNAME.a`, `BIBNAME.so` oder `BIBNAME.la`.

Zweitens muss man dem Compiler noch mitteilen, wo sich die Archivdatei befindet, die geschieht mit dem Flag `-L/path/to/lib/`.

Zuletzt benötigt der Compiler noch die Information, an welcher Stelle, die Headerdateien zu finden sind. Dies geschieht mit dem Flag `-I/path/to/include`.

In Kapitel 3.3.2 diskutieren wir dies noch einmal an einem Beispiel.

3.3 Beispiel: die joelixblas Bibliothek

Als Beispiel für eine externe Bibliothek möchten wir die von den Autoren implementierte `joelixblas` Bibliothek [2] benutzen. Diese Bibliothek implementiert grundlegende lineare Algebra[3] Funktionen für Matrizen und Vektoren im $\mathbb{R}^n$.

Wir weisen hier ausdrücklich darauf hin, dass `joelixblas` nur für Lernzwecke geschrieben und optimiert wurde und auch so benutzt werden sollte. Für „ernsthafte“ Programme, die BLAS Routinen brauchen, empfehlen wir zum Beipiel die `LAPACK` [1] oder `ATLAS` [5] Bibliothek zu verwenden.

3.3.1 Installation

Lade dir Datei `joelixblas-1.1.tar.gz` von der Seite http://github.com/joelix/joelixblas/archive/v1.1.tar.gz herunter. Entpacke die Datei in den Ordner `joelixblas-1.1`. Navigiere im Terminal in den Ordner und führe `make` aus:

```
1 $ cd joelixblas-1.1
2 $ make
```

Danach sollten sich 2 neue Unterordner in dem Ordner befinden: `build` und `lib`.

```
1 $ ls
2 AUTHORS  COPYING  download  joelixblas  Makefile
3 build    doc      INSTALL   lib         README.md
```

Die Bibliothek wurde erfolgreich kompiliert, wenn der Inhalt des Ordners so aussieht. Im Ordner `doc` befindet sich das Benutzerhandbuch mit den Beschreibungen aller Funktionen.

[3] „blas“ steht tatsächlich im Fachjargon für „basic linear algebra subprograms“.

Öffne das Handbuch und mache dich damit vertraut. Bei der Arbeit mit externen Bibliotheken ist es immer essenziell sich an das Handbuch oder (falls dieses nicht vorhanden ist) die Dokumentation der Headerfiles zu halten. Hier steht beschrieben, welche Funktionen und Datentypen die Bibliohtek zur Verfügung stellt und wie man diese benutzt.

3.3.2 Kompilieren eines Testprogramms

Wir beschreiben an dieser Stelle, wie ein Programm vernünftig gegen die Bibliothek gelinkt wird und kompiliert werden kann. Unser Testprogramm sei wie folgt, quasi ein „Hello World" für die `joelixblas` Bibliothek.

```
1  #include <vektor.h>
2  #include <joelix_error.h>
3
4  int main () {
5    Joelix_Vektor V;
6
7    /* Initialisiere V als Vektor der Laenge 4 */
8    joelix_vektor_init (&V, 4);
9    if (joelix_fehler_code == F_ERFOLG) {
10     /* Falls das geklappt hat, gebe Vektor aus und
11      * gebe dann den Speicher wieder frei. */
12     joelix_vektor_print (V);
13     joelix_vektor_loeschen (&V);
14   }
15   return 0;
16 }
```

Wir nehmen an, dass die Quelldatei den Namen `joelix_hwelt.c` hat und in dem Ordner `source` liegt, welcher im selben Ordner wie der Ordner `joelixblas-1.1` liegt. Du kannst das Programm natürlich an einem beliebigen anderen Ordner abspeichern und kompilieren, dann musst du nur daran denken, die Dateipfade im Folgenden auch anzupassen.

Um dieses Quelldatei zu kompilieren müssen wir `gcc` mitteilen, dass gegen die `joelixblas` Bibliothek gelinkt werden soll. Dies geschieht mit `-ljoelixblas`. Dazu benötigt `gcc` noch die Information, wo diese Bibliothek zu finden ist. Dies geschieht, wie in Kapitel 3.2 beschrieben, mit dem Befehl `-Llibpfad`. `libpfad` ist der Pfad zu dem Ordner mit der installierten Bibliothek, in unserem Fall also `../joelixblas-1.1/lib/`. Als letztes benötigt `gcc` die Information, an welcher Stelle nach dem Header `<vektor.h>` gesucht werden soll[4]. Dies geschieht mit dem Befehl `-Iincpfad`, wobei `incpfad` der Pfad zu dem Ordner mit den Headerdateien ist. In unserem Fall ist dies `../joelixblas-1.1/joelixblas/include`.

Der komplete Kompilierbefehl sieht demnach so aus:

[4] Dem Aufmerksamen Leser fällt auf, dass wir hier Spitze Klammern „<" und „>" verwenden.

```
1 $ cd source
2 $ gcc -Wall -pedantic -o joelix_hwelt joelix_hwelt.c \
3       -L../joelixblas-1.1/lib/ \
4       -I../joelixblas-1.1/joelixblas/include/ \
5       -ljoelixblas
```

Die Backslashes „\“ am Ende der Zeilen gebe dabei an, dass der Befehl in der nächsten Zeile fortgesetzt wird. Wenn du alles in eine Zeile schreibst, kannst sie also weglassen.

Das Programm sollte nun kompiliert sein und der Ouput sollte wie folgt aussehen:

```
1 $ ./joelix_hwelt
2 (0.000000, 0.000000, 0.000000, 0.000000)
```

3.3.3 Das CSR-Format für Matrizen

Oft hat man es mit sogenannten dünn besetzten Matrizen zu tuen, d.h. Matrizen von denen man von vornherein weiß, dass sie nicht viele von 0 verschiedene Einträge haben. Zum Beispiel die Adjazenmatrix eines Graphen mit einer großen Knotenanzahl N, in dem jeder Knoten mit höchstens $n < N$ anderen Knoten verbunden ist. Ein anderes Beispiel sind Bandmatrizen, diese tauchen oft in der numerischen Mathematik auf. Bandmatrizen sind Matrizen in denen nur einigen Diagonalen von 0 verschiedenen Werte haben. Wir werden später in Kapitel 3.3.4 eine solche Matrix kennenlernen. Ein weiteres Beispiel sind die Randmatrizen eines simplizialen Komplex, die in der algebraischen Topologie eine fundamentale Bedeutung spielen.

Nun wäre es sehr ineffektiv, wenn man sich für eine dünn besetzte Matrix jeden 0 Eintrag speichern würde. Um diese zu umgehen gibt es eine Reihe an Datenformaten, dünn besetzten Matrizen zu speichern. Wir stellen hier an dieser Stelle das CSR-Format (Compressed Sparse Row) vor. Damit lässt sich der Speicheraufwand für eine $n \times m$ Bandmatrix von $\mathcal{O}(nm)$ auf $\mathcal{O}(n)$ reduzieren.

Sei im folgenden M eine $n \times m$ Matrix mit genau N von 0 verschiedenen Einträgen. Um M im CSR-Format zu speichern, definieren wir die drei Arrays `werte`, `spalten` und `zeilen`. Die Arrays `werte` und `spalten` haben Länge N und `zeilen` hat Länge $n + 1$.

In dem Array `werte` speichern wir die Werte der nicht-null Einträge in lexikographischer Ordnung, d.h. Eintrag a_{ij} kommt vor $b_{i'j'}$ genau dann, wenn $i < i'$ erfüllt ist oder aber $i = i'$ und $j < j'$ gleichzeitig gilt. Insbesondere speichern wir also zuerst die Werte aus Zeile 1, dann die aus Zeile 2 und so weiter.

In dem Array `spalten` speichern wir in Index i den Spaltenindex des i-ten nicht-null Eintrags.

In dem Array `zeilen` speichern wir in Index i die Gesamtzahl an nicht-null Einträgen in allen Spalten vor Spalte i.

Beispiel 3. Wir betrachten die 5×5 Matrix M:

$$M = \begin{pmatrix} 2 & 0 & 0 & 1 & 0 \\ 0 & -7/10 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 42 & 17 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{1}$$

Für M sehen die obigen Arrays des CSR-Formates wie folgt aus.

```
1 werte = {2, 1, -0.7, 1, 3, 42, 17, -1}
2 spalten = {0, 3, 1, 2, 4, 1, 2, 0}
3 zeilen = {0, 2, 5, 5, 7, 8}
```

Listing 6: Das CSR-Format für M

Wir sehen zum Beispiel, dass der dritte nicht-null Eintrag $\texttt{werte}[2] = -0.7$ ist und in Spalte $\texttt{spalten}[2] = 1$ zu finden ist.

Da `zeilen` an der Stelle i die Anzahl an nicht-null Werten in allen Zeilen vor Zeile i speichert, ist `werte[zeilen[i]]` der Wert des ersten nicht-null Eintrags in Zeile i. Der zugehörige Spaltenindex ist `spalten[zeilen[i]]`. Will man also bei einer im CSR-Format gegebenen Matrix den Eintrag a_{ij} auslesen, so startet man mit $k = \texttt{zeilen}[i]$ und erhöht k bis $\texttt{spalten}[k] \geq j$. Falls nun $k > j$, dann ist $a_{ij} = 0$, falls $k = j$, dann ist $a_{ij} = \texttt{werte}[k]$.

Beispiel 4. Wir beschreiben hier kurz die Matrix-Vektor-Multiplikation mit Hilfe des CSR-Formates. Gegeben sei ein $n \times n$ Matrix M im CSR-Format und ein Vektor x als n-dimensionales Array. Wir berechen $Mx = b$.

```
1   int i, k;
2   for (i = 0;i < n;i++) {
3     /* Schleife ueber alle Zeilen der Matrix */
4     b[i] = 0;
5     for (k = M->zeilen[i];k < M->zeilen[i+1];k++) {
6       /* Schleife ueber alle nicht-null Eintrage dieser Zeile */
7       /* Der Wert an Stelle k steht in Spalte spalten[k] und muss
8          dehalb  mit dem Wert in Zeile spalten[k] von x multi-
9          pliziert werden. */
10      b[i] += M->werte[k] * x[M->spalten[k]];
11    }
12  }
```

Listing 7: Matrix-Vektor Multiplikation

Wir sehen in Bespiel 4, dass die Matrix-Vektor-Multiplikation Laufzeit $\mathcal{O}(N)$ hat, im Gegensatz zu $\mathcal{O}(nm)$ bei der Matrix-Vektor-Multiplikation mit einer als $n \times m$ Array gespeicherten Matrix.

3.3.4 Anwendung: Das Poissonproblem

Wir möchten an dieser Stelle eine Beispielanwendung diskutieren. Motiviert wird diese durch das Poissonproblem.

Wir stellen uns die Temperaturverteilung in einem (ein-dimensionalen) Stab vor. Da wir Mathematiker sind und es uns einfach machen können, sei der Stab o.B.d.A. durch das Einheitsinterval $[0,1]$ modelliert. Nehmen wir weiterhin an, der Stab hätte eine Ausgangstemperatur T_0 und ein nicht näher spezifiziertes Gerät (z.Bsp. Kühlung, Luft, Ofen, etc.) sorgt dafür, dass die Endpunkte 0 und 1 auch immer diese Temperatur haben.

Nun erhitzen (oder kühlen) wir den Stab im Bereich $(0,1)$ mit Hilfe einer Wärmequelle, zum Besipiel ein Bunsenbrenner oder Eiswürfel o.Ä., und beobachten, wie sich die Temperatur innerhalb des Stabes verändert. Unsere Wärmequelle liefere am Punkt x die Temperatur $g(x)$ für eine stetige Funktion g. Die Temperatur $u(x,t)$ am Punkt x zum Zeitpunkt t wird nach dem Gesetz der Wärmeleitung durch die Gleichung[5]

$$\frac{\partial}{\partial t}u(x,t) - au''(x,t) = g(x), \tag{2a}$$

$$u(0,t) = T_0, \tag{2b}$$

$$u(1,t) = T_0, \tag{2c}$$

beschrieben. Hierbei bezeichnet u'' die zweifache Ableitung nach x und $\frac{\partial}{\partial t}u(x,t)$ die Ableitung nach der Zeit t; die Materialkonstante a ist ein Parameter, der von der Beschaffenheit des Stabes abhängt und angibt wie gut das Material Wärme leitet.

Nach einiger (in der Theorie unendlicher) Zeit wird sich ein Gleichgewicht einstellen bei dem sich die Temperatur nicht mehr ändert. D.h. im Gleichgewicht gilt $\frac{\partial}{\partial t} = 0$ und wir erhalten die sogenannte Poisson-Gleichung

$$-u''(x) = g(x), \tag{3a}$$

$$u(0) = T_0, \tag{3b}$$

$$u(1) = T_0. \tag{3c}$$

Die Konstante a hat im Gleichgewicht nur noch den Einfluss eines Faktors und ist deshalb mathematisch nicht mehr interessant für die Bestimmung einer Lösung. Der Einfachheit halber, nimmt man in der Poisson-Gleichung deshalt $a = 1$.

Zusammengefasst bedeutet dies, dass eine Funktion $u\colon [0,1] \to \mathbb{R}$, die Gleichung (3) erfüllt, für jeden Punkt x die Temperatur des Stabes im Gleichgewicht angibt.

[5] Falls du noch nicht weißt, was die partiellen Ableitungen $\partial/\partial t$ genau dedeuten ist das kein Problem. Wir brauchen es hier nur für die Herleitung, aber die finale Gleichung (3), die wir lösen, benötigt keine partiellen Ableitungen.

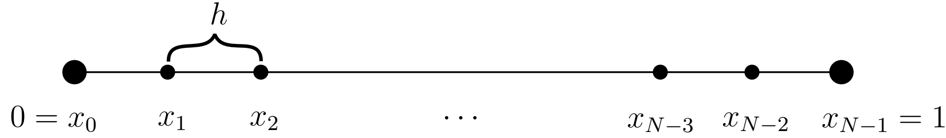


Abbildung 2: Der stilisierte eindimensionale Stab und die von uns verwendeten Stützstellen x_i.

Wir wollen nun diese Gleichung Modellieren und ein Programm schreiben, welches eine (approximative) Lösung berechnet. Dafür verwenden wir das sogenannte „finite Differenzen“ Verfahren. Da wir nicht den Funktionswert der Lösung an den unendlich vielen Zahlen in $[0, 1]$ berechnen können, legen wir N Stützpunkte $x_0, \ldots, x_{N-1}$ im Intervall fest und bestimmen Werte $\hat{u}(x_i)$, die $u(x_i)$ annähern für jedes i. Als Abstand zwischen den Stützpunkten wählen wir $h = 1/(N-1)$, sodass gilt: $x_i = ih$ und $x_{i+1} - x_i = h$. Wir haben dies mal in Figur 2 graphisch dargestellt.

Da für die Ableitung von u and der Stelle x_i gilt, dass

$$u'(x_i) = \lim_{\epsilon \to 0} \frac{u(x_i) - u(x_i + \epsilon)}{\epsilon} \tag{4}$$

gilt weiterhin

$$u''(x_i) = \lim_{\epsilon \to 0} \frac{-u(x_i - \epsilon) + 2u(x_i) - u(x_i + \epsilon)}{\epsilon^2}. \tag{5}$$

Zusammen mit $-u''(x_i) = g(x_i)$ machen wir uns dies zu Nutze und modellieren

$$g(x_i) = \frac{-\hat{u}(x_i - h) + 2\hat{u}(x_i) - \hat{u}(x_i + h)}{h^2} \tag{6a}$$

$$= \frac{-\hat{u}(x_{i-1}) + 2\hat{u}(x_i) - \hat{u}(x_{i+1})}{h^2} \tag{6b}$$

Hierbei gilt Gleichung (6b) natürlich nur für die Indizes $i \neq 0$ und $i \neq N-1$.

Diese Randwerte bei x_0 und x_{N-1} nehmen eine Sonderstellung ein, da die Werte von $\hat{u}$ hier schon durch T_0 vorgegeben sind. Dies hat Einfluss auf die Werte $\hat{u}(x_1)$ und $\hat{u}(x_{N-2})$, wobei wir ab nun annehmen wollen, dass $N > 3$ gilt. Wir erhalten für $\hat{u}(x_1)$

$$g(x_1) = \frac{-\hat{u}(x_0) + 2\hat{u}(x_1) - \hat{u}(x_2)}{h^2} \tag{7a}$$

$$= \frac{-T_0 + 2\hat{u}(x_1) - \hat{u}(x_2)}{h^2} \tag{7b}$$

$$\Leftrightarrow g(x_1) + \frac{T_0}{h^2} = \frac{2\hat{u}(x_1) - \hat{u}(x_2)}{h^2} \tag{7c}$$

und analog für $\hat{u}(x_{N-1})$

$$g(x_{N-2}) + \frac{T_0}{h^2} = \frac{-\hat{u}(x_{N-3}) + 2\hat{u}(x_{N-2})}{h^2}. \tag{8}$$

Wir fassen Gleichungen (6), (7) und (8) in eine Matrix-Vektor Gleichung zusammen:

$$\underbrace{\frac{1}{h^2}\begin{pmatrix} 1 & & & & & & & \\ & 2 & -1 & & & & & \\ & -1 & 2 & -1 & & & & \\ & & -1 & 2 & -1 & & & \\ & & & \vdots & & & & \\ & & & -1 & 2 & -1 & & \\ & & & & -1 & 2 & & \\ & & & & & & & 1 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} \hat{u}(x_0) \\ \hat{u}(x_1) \\ \hat{u}(x_2) \\ \hat{u}(x_3) \\ \vdots \\ \hat{u}(x_{N-3}) \\ \hat{u}(x_{N-2}) \\ \hat{u}(x_{N-1}) \end{pmatrix}}_{\hat{u}} = \underbrace{\begin{pmatrix} \frac{T_0}{h^2} \\ \frac{T_0}{h^2} + \quad g(x_1) \\ g(x_2) \\ g(x_3) \\ \vdots \\ g(x_{N-3}) \\ \frac{T_0}{h^2} + g(x_{N-2}) \\ \frac{T_0}{h^2} \end{pmatrix}}_{b} \tag{9}$$

Das bedeutet, wir sind an der Lösung des Gleichungssstems

$$A\hat{u} = b \tag{10}$$

interessiert sind. Dazu werden wir (selbstverständlich) die Funktionen aus der `joelixblas` Bibliothek verwenden um das sogenannte CG-Verfahren zu implementieren. Das CG-Verfahren ist ein effizientes numerisches Verfahren zum iterativen Lösen von linearen Gleichungssystemen. Da wir durchaus an großen Werten für N interessiert sind, ist ein direktes Lösen durch z.Bsp. den Gaußalgorithmus wegen zu großer Laufzeit nicht mehr praktikabel.

Das CG-Verfahren funktioniert nur, wenn die Matrix A symmetrisch und positiv definit ist, diese Bedingungen sind aber praktischerweise erfüllt[6].

Der genau Algorithmus steht in Pseudocode in Algorithmus 3.1 oder auf Wikipedia[7].

Als kurzes Beispiel zeigen wir in Abbildung 3 das Ergebnis einer Simulation mit 50 Stützstellen und rechter Seite $g(x) = 850e^{-\frac{1}{2}(x-0.5)^2}$, also einer Wärmequelle am Mittelpunkt, deren Temperatur zu den Rändern hin exponentiell abfällt.

Aufgabe. Implementiere das CG-Verfahren und benutze es, um das Poissonproblem zu Lösen. Nutze dazu die `joelixblas` Bibliothek.

Aufgabe*. Löse das 2D Poissonproblem. Die zu lösende Gleichung lautet

$$-\Delta u = g \text{ in } [0,1]^2 \tag{11a}$$

$$u = f \text{ auf } \partial[0,1]^2 \tag{11b}$$

[6] Das kannst du gerne selber nachrechnen, wenn dir langweilig ist.

[7] https://de.wikipedia.org/wiki/CG-Verfahren

Algorithmus 3.1 : CG-Verfahren (Matrix A, Vektor x, Vektor b, double ϵ)

```
1  x_0 ← 0;
2  r_0 ← b;
3  d_0 ← r_0;
4  res ← 1;
5  maxit ← 1000;
6  k ← 0;
7  while res > ε and k < maxit do
8      z ← A d_k;
9      α_k ← (r_k · r_k) / (d_k · z);
10     x_{k+1} = x_k + α_k d_k;
11     r_{k+1} = r_k − α_k z;
12     β_k = (r_{k+1} · r_{k+1}) / (r_k · r_k);
13     d_{k+1} ← r_{k+1} + β_k d_k;
14     res ← ‖r_{k+1}‖;
15     k++;
16 x ← x_k;
```

wobei hier ∂ den Rand des Gebiets bezeichnet und Δ den Laplace-Operator. Natürlich verwendest du auch hierfür die Funktionen aus der `joelixblas` Bibliothek.

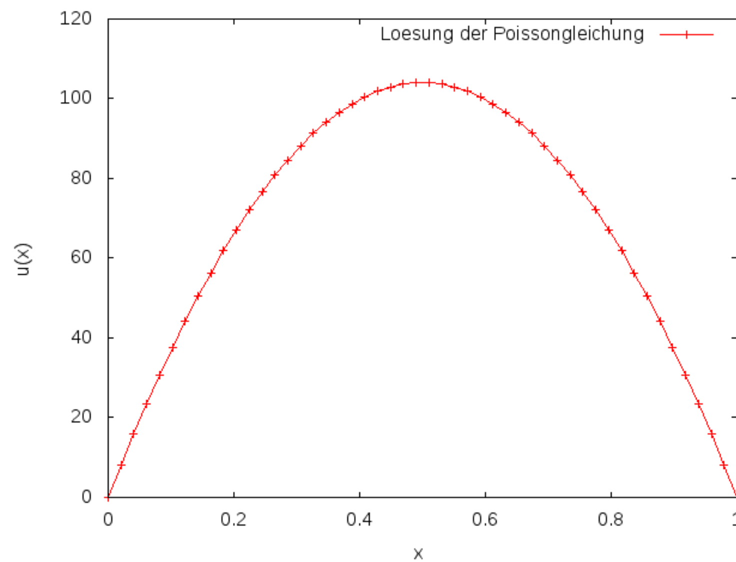


Abbildung 3: Die approximierte Lösung des 1D Poissonproblems mit rechter Seite $g(x) = 850e^{-\frac{1}{2}(x-0.5)^2}$ und $N = 50$. Die Randbedingungen sind $T_0 = 0$. Man kann sich das simulierte Experiment vorstellen als würden wir eine ca. 1400 Grad heiße Flamme in die Mitte das Stabes halten, deren Temperatur zu den Rändern hin exponentiell abfällt. Gleichzeitig kühlen wir die Randpunkte permanent auf 0 Grad Celsius. Dieses Bild wurde mit `gnuplot` [6] erzeugt.

4 Multithreading mit OpenMP

Diese Kapitel soll den Einstieg in Thread-Programmierung erleichtern, kann dabei natürlich weder die Vollständigkeit einer Dokumentation bieten noch die Erfahrung ersetzen, die man für Thread-Programmierung benötigt.

In diesem Kapitel möchten wir eine Bibliothek vorstellen mit der in C sogenanntes "Threads" realisiert sind. Threads ermöglichen es, mehrere Dinge gleichzeitig zu berechnen. In alten Heimcomputern ist dies nicht möglich – das höchste der Gefühle ist hier, zwei Berechnungen sehr schnell abwechselnd durchzuführen. In modernen Computern oder sogar in Computersystemen, die für paralleles Rechnen ausgelegt sind, stehen mehrere CPUs (oder CPUs mit mehreren Kernen) zur Verfügung. Alle C-Programme, die wir bisher geschrieben haben, nutzen maximal eine CPU bzw. maximal einen CPU-Kern. Die Idee beim Threading ist, neben dem Hauptprogramm (ab jetzt *Masterthread* genannt) weitere Nebenprogramme zu starten (ab jetzt *Workerthreads* genannt). Den Masterthread und die Workerthreads zusammen nennt man das *Threadteam*.

Um die technische Realisierung wird sich die Bibliothek "OpenMP" kümmern, wir wollen anhand des folgenden Beispiels zeigen, wie sie verwendet werden kann:

```
1 int main() {
2     int i;
3     double a[10000];
4     for(i=0; i<10000; i++) {
5         a[i] = i*i / 2.0;
6     }
7     return 0;
8 }
```

In diesem einfachen Programm wird ein Array mit den halben Quadratzahlen gefüllt. Zu beachten ist, dass die Berechnung der nächsten Quadratzahl nicht von der vorhergehenden abhängt, also können wir durch das Hinzufügen von nur zwei Zeilen die Berechnung parallelisieren:

```
1 #include <omp.h>
2 int main() {
3     int i;
4     double a[10000];
5     #pragma omp parallel for
6     for(i=0; i<10000; i++) {
7         a[i] = i*i / 2.0;
8     }
9     return 0;
10 }
```

Verwendet man gcc, muss beim Kompilieren das Compiler-Flag `-fopenmp` angegeben werden. Es weißt den gcc dazu an, das "Compiler-Plugin" `openmp` zu verwenden. Führt man das Programm danach aus, erzeugt der Masterthread in

Zeile 5 einige Workerthreads und die Berechnung in der `for`-Schleife wird auf sie aufgeteilt – diesen Vorgang nennt man *Fork*. In Zeile 8 ist die Schleife zuwende und die Workerthreads verschwinden wieder – dies nennt man *Join*.

Die Anweisung `#pragma omp parallel for` ist eine Kurzschreibweise für eine andere Anweisung, diese werden wir zunächst kennen lernen. Solche Anweisungen von OpenMP haben die folgende Form:

```
#pragma omp DIRECTIVE CLAUSES
```

Das weitere Ziel ist nun, Worte zu lernen, die als DIRECTIVE und CLAUSES angegeben werden können (mehrer Klauseln werden durch Leerzeichen oder Kommata getrennt).

4.1 Forking und Joining

Um an irgendeiner Stelle zu Forken – also den Masterthread anzuweisen Workerthreads zu erzeugen – verwendet man die Direktive `parallel`. Danach kann ein C-Block angegeben werden, der dann von allen Threads gleichermaßen ausgeführt wird:

```
1 #pragma omp parallel
2 {
3     do_work();
4 }
```

In diesem Beispiel wird die Funktion `do_work` von allen Threads gleichermaßen aufgerufen. Innerhalb des `parallel`-Block können nun verschiedene weitere Direktiven angegeben werden um dafür zu sorgen, dass nicht alle Threads exakt das gleiche machen. Ein naiver Ansatz wäre etwa die Verwendung der *Laufzeitfunktion* `omp_get_thread_num`, die in `<omp.h>` deklariert ist und die eindeutige Nummer des gerade aktiven Threads zurück gibt.

```
1 #pragma omp parallel thread_num(2)
2 {
3     if (omp_get_thread_num() == 0) do_work();
4     else do_other_work();
5 }
```

Listing 8: Verwendung der Parallel-Direktive

In diesem Beispiel wurde ausserdem die Klausel `thread_num(ANZAHL)` verwendet mit der man die Anzahl der Threads steuern kann. Alternativ kann man auch die Umgebungsvariable `OMP_NUM_THREADS` setzen um die Anzahl der Threads fest zu legen. Wird beides nicht gemacht, entscheided OpenMP über die beste Anzahl von Threads (liegt nur eine CPU mit nur einem Kern vor, beträgt diese wahrscheinlich 1).

Eine weitere Klausel ist `if`, sie nimmt eine Expression als Argument entgegen. Wenn die Expression zu wahr auswertet, wird der Fork tatsächlich ausgeführt, sonst wird einfach mit einem einzigen Thread, dem Masterthread fortgefahren. Dies ist z.B. dann nützlich, wenn sich herausgestellt hat, dass sich Parallelisierung erst ab einer bestimmten Datengröße lohnt und das Programm vorher, wegen des Verwaltungsaufwand, der durch Threads versacht wird, sogar langsamer wird.

4.2 Sektionen

Möchte man, wie im Listing 8 aus dem letzten Kapitel, zwei Aufgaben an zwei Threads verteilen, kann die Direktive `sections` verwendet werden. Im darauffolgenden Block, kann mehrmals die Direktive `section` angegeben werden. Erreicht das Programm den `sections` Block, werden die eingeschlossenen `section`s unter den bestehenden Threads aufgeteilt. Gibt es mehr `section`s als Threads, wird ein Thread mehrere `section`s nacheinander ausführen, gibt es mehr Threads als `section`s, haben einige Threads nichts zu tun.

```
1  #pragma omp sections
2  {
3  #pragma omp section
4      {
5          do_work();
6      }
7  #pragma omp section
8      {
9          do_other_work();
10     }
11 }
```

Die Direktive `sections` sollte natürlich nur innerhalb eines `parallel`-Blocks angegeben werden, sonst werden die `section`s nur unter dem Masterthread "aufgeteilt", der dann einfach die gesamte Arbeit macht.

4.3 Schleifen

Zum paralellisieren einer For-Schleife, verwendet man die Direktive `for`

```
1  #pragma omp for
2  for(i=0; i<10000; i++) {
3      a[i] = i*i / 2.0;
4  }
```

Wir möchten an dieser Stell erneut darauf hinweisen, dass eine wichtige Eigenschaft des Codes innerhalb der For-Schleife darin besteht, dass kein Durchlauf von einem vorhergehenden abhängt: Erreicht das Threadteam nämlich die `for`-Direktive, steht nicht fest, mit welchem Schleifendurchlauf begonnen wird. Es

könnte sehr gut sein, dass der Durchlauf für `i=1` *vor* dem Durchlauf von `i=0` ausgeführt wird. Folgender Code würde ohne Paralellisierung korrekt funktionieren, *mit* aber nicht:

```
1 a[0]=1
2 #pragma omp for
3 for(i=1; i<n; i++) {
4     a[i] = a[i-1] * 2.0; /* DO NOT TRY THIS AT HOME */
5 }
```

Ein großes Problem an solchem Code ist ausserdem, dass er sehr wohl wie erwartet funktionieren *kann* (z.B. wenn man ihn Zuhause testet, nur eine CPU mit nur einem Kern hat und die Sterne richtig stehen). Es kann aber ganz plötzlich dazu kommen, dass er unerwartete Resultate liefert, weil beispielsweise der Durchlauf von `i=2` vor dem Durchlauf von `i=1` ausgeführt wird und `a[1]` darum noch undefiniert ist. Die oben stehende Schleife läßt sich also einfach *nicht prallelisieren.*

Es sind auch nicht alle Formen von For-Schleifen erlaubt, beispielsweise muss zu Beginn der Schleife bereits feststehen, wie oft sie ausgeführt wird, damit es überhaupt möglich ist, die verschiedenen Schleifendurchlaufe auf die Threads aufzuteilen.

4.4 Barrieren

Erreicht ein Thread des Teams eine mit der `barrier`-Direktive definierte Barriere, wartet er darauf, bis alle anderen Threads auch diese Stelle erreicht haben:

```
1 #pragma omp parallel
2 {
3     do_work1();
4 #pragma omp barrier
5     do_work_that_depends_on_work1();
6 }
```

Listing 9: Barriere

Sowohl hinter der `sections`-Direktive als auch hinter der `for`-Direktive befindet sich eine sogenannte *impliziete Barriere*, das bedeutet, dass nach Beenden des entsprechenden Blockes mit der weiteren Ausführung auf das gesamte Threadteam gewartet wird. Mit der Klausel `nowait` kann dieses Verhalten geändert und die impliziete Barriere entfernt werden:

```
1 #pragma omp parallel
2     f = 2.0;
3 #pragma omp for nowait
4     for(i=0; i<n; i++) {
5         z[i] = x[i] + y[i] * f;
```

```
 6      }
 7  #pragma omp for nowait
 8      for(i=0; i<n; i++) {
 9          a[i] = b[i] + c[i];
10      }
11  #pragma omp barrier
12      sum = 0;
13      for(i=0; i<n; i++) {
14          sum += z[i] * a[i];
15      }
16      do_something_with(sum);
17  }
```

Listing 10: Expliziete Barriere

In diesem Beispiel wurde in Zeile 11 eine expliziete Barriere eingefügt, da sonst nicht sichergestellt ist, dass die Arrays `z` und `a` schon vollständig berechnet wurden. Hier sei darauf hingewiesen, dass die Berechnung von `sum` von jedem Thread einzeln ausgeführt wird. Es wäre nun naheliegend diese Berechnung auch zu parallelisieren, dabei ist allerdings Vorsicht geboten! Mehr dazu in 4.5.

4.5 Shared Memory

Bisher haben wir noch kein Wort dazu verloren, was beim Erreichen eines `parallel`-Blocks mit den bis dahin deklarierten Variablen geschieht. Es gibt zwei naheliegende Verhaltensweisen:

private Variable Man übergibt der Klausel `private` eine Komma-getrennte Liste von Variablen, die als "privat" ausgewiesen werden sollen. Jeder Thread erhält dann eine Variable von gleichem Namen und Typ. Ändert der Thread sie, sind diese Änderung nur für ihn selbst gültig. Standardmäßig sind private Variablen für jeden Thread uninitialisiert.

Möchte man sie mit dem Wert der Variablen vor dem fork initialisieren, muss man sie an die Klausel `firstprivate` übergeben (man kann sie dann auch aus der Liste von `private` weg lassen).

Bei der `for`- bzw. `sections`-Direktive gibt es eine klar definierte "letzte Aufgabe", nämlich den sequentiell letzten Schleifendurchlauf bzw. die lexikographisch letzte `section`. Möchte man den Wert einer privaten Variable dieser letzten Aufgabe beim Join übernehmen, muss man sie in der Variablenliste, die man an `lastprivate` übergibt, aufführen (auch hier kann die Variable aus der Liste von `private` weg gelassen werden).

geteilte Variable / shared Variable alle Threads teilen sich die gleiche Variable. Auch `shared` nimmt eine Liste von Variablen entgegen, die danach

von allen Threads des Teams geteilt werden. Das Schreiben in geteilte Variablen verursacht sehr viele Probleme bei Programmierung der mit Threads und sollte so oft wie möglich vermieden werden.

Die `default`-Klausel der `parallel`-Direktive nimmt ein Argument entgegen, es ist entweder `none` oder `shared` (ist die Klausel nicht angegeben ist dies der Standard). Die Angabe von `default(shared)` bedeutet, dass Variablen, die weder expliziet als private oder geteilt ausgewiesen werden als shared Variables betrachtet werden. Gibt man hingegen `default(none)` an, so müssen Variablen die innerhalb des `parallel`-Blocks verwendet werden expliziet als "privat" oder "geteilt" ausgewiesen werden. Um versehentliches Schreiben in eine geteilte Variable zu vermeiden empfehlen wir darum immer die Angabe von `default(none)`.

Um nun die Berechnung der Summe aus Listing 10 zu parallelisieren, könnte man folgenden naiven Ansatz verfolgen:

```
1 sum = 0;
2 #pragma omp parallel shared(sum)
3     #pragma omp for
4     for(i=0; i<n; i++) {
5         sum += z[i] * a[i]; /* BAAAD! */
6     }
7 }
```

Listing 11: Falsche Art eine Summenberechnung zu parallelisieren

Das Problem tritt beim Schreiben in die geteilte Variable `sum` auf: Es kann sein, dass Thread A den Wert von `sum` aus dem Speicher ließt, dann Thread B das gleiche macht. Jetzt erst addiert Thread A den Wert `z[i] * a[i]` zu `sum` und speichert ihn wieder im Speicher. Danach macht Thread B das gleiche und überschreibt den Wert, den Thread A gespeichert hat, wir haben also einen Summanden verloren. Solche Probleme lößt man im Allgemeinen mit Locks (siehe 4.7) oder der `ciritcal`-Direktive (siehe 4.6), in diesem speziellen Fall stellt OpenMP aber eine sehr ellegante Methode zur Verfügung: die `reduction`-Klausel. Sie nimmt einen Operatoren und, mit einem Doppelpunkt davon getrennt, eine Liste von Variablen entgegen. Alle übergebenen Variablen werden als privat ausgewiesen und, je nach Operator, initialisiert.

<table>
<tr><th>Operator</th><th>Wert</th></tr>
<tr><td>+</td><td>0</td></tr>
<tr><td>*</td><td>1</td></tr>
<tr><td>-</td><td>0</td></tr>
<tr><td>&</td><td>˜0</td></tr>
<tr><td>|</td><td>0</td></tr>
<tr><td>^</td><td>0</td></tr>
<tr><td>&&</td><td>1</td></tr>
<tr><td>||</td><td>0</td></tr>
</table>

Tabelle 1: Erlaubte Operatoren und ihre Initialisierungswerte

Jeder Thread kann nun auf seine eigene Version der Variablen zugreifen und problemlos beschreiben. Nach Beendigung des entsprechenden Blockes werden dann die privaten Variablen aller Threads mit dem angegebenen Operator kombiniert. Eine notwendige Voraussetzung an den Operator muss es also sein, dass er kommutativ und assoziativ ist.

```
sum = 0;
#pragma omp parallel reduction(+:sum)
    #pragma omp for
    for(i=0; i<n; i++) {
        sum += z[i] * a[i];
    }
}
```

Listing 12: Parallelisierung einer Summenberechnung

4.6 single, master und critical

Ein hinter der `single`-Direktive angegebener Block wird nur von einem Thread ausgeführt. Wird die `nowait`-Klausel nicht angegeben, gibt es danach eine implizite Barriere.

```
do_work1(); /* wird von jedem Thread ausgeführ */
#pragma omp single
{
    do_work2(); /* wird nur ein einziges Mal ausgeführt */
}
do_work3(); /* wird wieder von jedem Thread ausgeführ */
```

Listing 13: Ausführung von nur einem Thread

Die Direktive `master` verhält sich wie die `single`-Direktive mit folgenden Unterschieden:

- Der Thread, der die Aufgabe ausführen soll, ist immer der Master-Thread. Dies ermöglicht es beispielsweise auf den Wert einer privaten Variable über mehrere `master`-Blöcke hinweg zuzgreifen.
- Nach dem `master`-Block gibt es keine impliziete Barriere.

Schlussendlich ermöglicht es die `critical`-Direktive einen Block nur von exakt einem Thread *gleichzeitig* auszuführen. Im Gegensatz zu `single` und `master` wird er allerdings von jedem Thread ausgeführt. Die `ciritcal`-Direktive erhält ein optionale Argument – ihren Namen. Gibt es an unterschiedlichen Stellen `critical`-Blöcke mit dem gleichen Namen, kann immer nur eine gleichzeitig ausgeführt werden. Erreicht ein Thread den Beginn eines `critical`-Blocks, während ein Block mit dem gleichen Namen bereits ausgeführt wird, wartet er.

```
1 #pragma omp critical(writeinfile)
2 {
3     write_data_to_file();
4 }
5 do_something();
6 #pragma omp critical(writeinfile)
7 {
8     write_data_to_file();
9 }
```

Listing 14: Ausführung von nur einem Thread

Das Argument für den Namen kann weg gelassen werden, der Block erhält dann einen Standard-Namen. Es darf also auch nur ein `critical`-Block mit dem Standard-Namen ausgeführt werden.

Code-Bereiche, die immer nur von einem Thread ausgeführt werden dürfen (wie das Schreiben in eine Datei beispielweise) nennt man *kritische Bereiche*, daher auch der Name der `critical`-Direktive. Manchmal ist die Definition von kritischen Bereichen über diese Direktive allerdings nicht flexibel genug: Immerhin darf über das gesamte Programm hinweg immer nur ein einziger `critical`-Block mit dem gleichen Namen ausgeführt werden. Es ist aber auch denkbar, dass zwei Threadteams, die die gleiche Funktion aufrufen sich gegenseitig nicht behindern sollen. In solchen Situation verwendet man *Locks*.

4.7 Locks

Ein Lock ist eine Variable vom Typ `omp_lock_t`, welcher in `omp.h` definiert ist. Ein Lock hat zwei Zustände: entweder "offen" oder "geschlossen", was dafür verwendet werden kann, sehr flexible kritische Sektionen zu definieren. in `omp.h` werden dazu noch folgende Funktionen definiert:

omp_init_lock(*omp_lock_t) Initialisiert eine Lock-Variable, muss vor Benutzung eines Locks aufgerufen werden. Nach Initialisierung ist eine Lock-Variable offen.

omp_destroy_lock(*omp_lock_t) Zerstört eine Lock-Variable wieder, diese Funktion muss mit jeder Lock-Variablen aufgerufen werden. Vor der Zerstörung muss die Variable offen sein.

omp_set_lock(*omp_lock_t) versucht eine Lock-Variable zu schließen, ist sie bereits geschlossen, so wartet der aktuelle Thread mit der Ausführung (diese Verhalten nennt man auch *blocking*) bis die Variable wieder offen ist und wiederholt den Vorgang.

omp_unset_lock(*omp_lock_t) öffnet die Lock-Variable. Die Lock-Variable sollte vom gleichen Thread wieder geöffnet werden, der sie auch geschlossen hat.

omp_test_lock(*omp_lock_t) versucht eine Lock-Variable zu schließen. Ist sie bereits geschlossen, gibt die Funktion 0 zurück. Ansonsten schließt sie das Lock und gibt 1 zurück. (Diese Verhalten nennt man auch *non-blocking*).

Eine kritische Sektion läßt sich nun beispielsweise so realisieren:

```
1  omp_lock_t write_to_file_lock;
2  omp_init_lock(&write_to_file_lock);
3
4  #pragma omp parallel
5  {
6      do_something1();
7
8      omp_set_lock(&write_to_file_lock);
9      write_to_file();
10     omp_unset_lock(&write_to_file_lock);
11
12     do_something2();
13
14     while(!omp_test_lock(&write_to_file_lock)) {
15         do_something_else();
16     }
17     write_to_file();
18     omp_unset_lock(&write_to_file_lock);
19
20     do_something3();
21 }
22 omp_destroy_lock(&write_to_file_lock);
```

Listing 15: Kritische Sektion mit Locks

5 Ausblick: Parallelisierung mit MPI

In Kapitel 4 haben wir das Parallelisieren mit Threads besprochen. Hier liegt die Idee von geteilten Speicher (shared memory) zu Grunde, dies bedeutet, dass sich alle Threads den selben Arbeitsspeicher teilen. Außerdem wird das Programm nur von einem Thread, dem Masterthread, gestartet und dieser erstellt zur Laufzeit neue Threads.

Allerdings sind diesem Prinzip Grenzen gesetzt, denn es kann sich nur eine gewisse Anzahl an CPUs den selben Speicher teilen. Derzeit sind übliche Zahlen bei 8 oder 16 CPUs. Große Workstations kommen auch schonmal auf 24 aber danach ist im Grunde Schluss. Möchte man sein Programm noch weiter parallelisieren (man spricht auch von Skalieren), so sollte man sich mit dem Modell des verteilten Speichers (distributed memory) beschäftigen. Bei diesem Modell geht man davon aus, dass man mehrere voneinander separierte CPU+Speicher Elemente hat. Dass bedeutet, dass eine CPU Zugriff auf ihren eigenen Speicher hat, aber den von anderen CPUs nicht sehen/auslesen kann.

Der Anzahl an möglichen CPUs ist bei diesem Prinzip keine Grenze gesetzt. Beispielsweise ist der derzeit in Deutschland größte Rechner der Supercomputer JUQUEEN am Forschungszentrum Jülich mit 458.752 CPUs[8]. Um sich die volle Rechenkraft von so vielen CPUs zu Nutze machen zu können muss man allerdings auch besonders gute parallele Programme schreiben.

Wir möchten an dieser Stelle einen kleinen Ausblick in die Programmierung auf verteilten System geben. Hierzu nutzt man für gewöhnlich `MPI`. `MPI` steht für „Message Passing Interface“ und ist ein vom MPI-Forum festgelegter Standard für das Empfangen und Verschicken von Nachrichten zwischen verschiedenen Prozessen. Dieser Standard ist von verschiedenen Leuten implementiert worden, das Interface unterscheidet sich dabei aber nicht. Zwei wichtige Open Source Implementationen sind `openmpi` und `MPICH`. Wir empfehlen `MPICH` zu nutzen.

5.1 Warum „Message Passing“?

Im Gegensatz zu Multithreading mit OpenMP wo das Programm ein einziges Mal aufgerufen wird und Threads erstellen kann, wird beim Programmieren mit `MPI` das Programm direkt n Mal von n verschiedenen sogenannten Prozessen aufgerufen. Dies bedeutet, dass jeder Prozess seine eigene Kopie des Programms ausführt und insbesondere seinen komplett eigenen Bereich im Arbeitsspeicher besitzt in dem er Programmvariablen abspeichert und bearbeitet.

Ein Prozess A kann also nicht ohne weiteres auf Werte zugreifen die ein Prozess B verändert hat.

Die wenigsten Probleme lassen sich aber so parallelisieren, dass jeder Prozess komplett für sich alleine arbeiten kann und am Ende ein Ergebnis steht. Statt-

[8]siehe `https://www.top500.org/lists/2016/11/`

dessen ist es zwischendurch immer wieder notwendig, dass Zwischenergebnisse ausgetauscht werden oder auf ander Weise miteinander kommuniziert werden muss. Dieses kommunizieren zwischen den Prozessen geschieht mit Nachrichten. Eine Nachricht ist eigentlich nichts anderes als ein Array von Werte, welche ein Prozess an einen anderen schickt. Der empfangende Prozess muss ein gleichgroßes Array angelegt haben, in welchem bei Empfangen dann die Werte des anderen Prozesses abgelegt werden. Wir werden weiter unten Besprechen, wie genau dies im Detail funktioniert.

5.2 Hello World und Kompilierung

Das einfachste `MPI`-Programm gibt auf jedem Prozess ein kurzes Nachricht aus und sieht wie folgt aus.

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main (int argc, char *argv[]) {
5    MPI_Init (&argc, &argv);
6    printf ("Hallo\n");
7    MPI_Finalize ();
8
9    return 0;
10 }
```

Listing 16: Die Datei hallo_welt.c

Am Anfang und am Ende eines `MPI`-Programmes stehen immer die Funktionen `MPI_Init` und `MPI_Finalize`, wobei erstere Pointer auf die Kommandozeilenparameter erhält. Siehe dazu auch Kapitel 7.1.

Kompiliert werden `MPI`-Programme mit einem eigenen Kompiler, `mpicc`. Dieser wird mit installiert, wenn man sich `openmpi` oder `mpich` installiert und wird genauso benutzt wie `gcc`. Wir benutzen nun noch zusätzlich das Flag `-std=c99`.

```
1  $  mpicc -std=c99 -Wall -pedantic -o hallo_welt hallo_welt.c
```

Ausgeführt werden `MPI`-Programme mit dem Befehl `mpirun -np n ./PROGRAMMNAME`, hierbei steht n für die Anzahl an Prozessen, die eine Kopie des Programms starten sollen.

```
1  $ mpirun -np 4 ./hallo_welt
2  Hallo
3  Hallo
4  Hallo
5  Hallo
```

5.3 rank und size

Innerhalb der Ausführung bekommt jeder Prozess eine eindeutige Nummer zwischen 0 und $n-1$ zugewisen, den Rang. Dies ist wie die `thread_num` bei OpenMP. Diesen Rang kann man mit der Funktion `MPI_Comm_rank` auslesen lassen. Die Anzahl n an Prozessen, welche beim Aufruf von `mpirun` gestartet wurde kann man mit der Funktion `MPI_Comm_size` auslesen.

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main (int argc, char *argv[]) {
5    int rank, size;
6
7    MPI_Init (&argc, &argv);
8
9    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
10   MPI_Comm_size (MPI_COMM_WORLD, &size);
11
12   printf ("Hallo von %i. Gesamtzahl %i\n", rank, size);
13
14   MPI_Finalize ();
15   return 0;
16 }
```

Listing 17: hwelt_rang.c

Wenn wir dieses Programm kompilieren und mit 4 Prozessen ausführen erhalten wir folgende Ausgabe:

```
1 $  mpicc -std=c99 -Wall -pedantic -o hwelt_rang hwelt_rang.c
2 $  mpirun -np 4 ./hwelt_rang
3 Hallo von 0. Gesamtzahl 4
4 Hallo von 1. Gesamtzahl 4
5 Hallo von 2. Gesamtzahl 4
6 Hallo von 3. Gesamtzahl 4
```

Das Argument `MPI_COMM_WORLD` ist ein sogenannter Kommunikator, mit dem wir uns in diesem Skript nicht weiter beschäftigen wollen. Grob gesagt sagt `MPI_COMM_WORLD` aus, dass wir alle Prozesse betrachten wollen, die bei `mpirun` gestartet wurden. Es ist mit dem Kommunikator möglich Untergruppen von Prozessen zu bilden, dies wird allerdings erst in sehr fortgeschrittenen Anwendungen benötigt.

5.4 Send und Recv

Wir beschreiben nur wie das versenden von einfachen Nachrichten in `MPI` umgesetzt ist. Dafür benutzt man die Funktionen `MPI_Send` und `MPI_Recv`. Stellen

wir uns vor der Prozess mit Rang i möchte Daten an den Prozess mit Rang j schicken. Diese Daten müssen dann alle von gleichen Datentyp sein und in einem Array der Länge m gespeichert sein. Der Empfangende Prozess j muss auch ein Array der Länge m allozieren, um die Daten dort empfangen zu können. Nun ruft Prozess i die Funktion `MPI_Send` auf und Prozess j die Funktion `MPI_Recv`.

Prozess i kann erst mit der Ausführung des Programms fortfahren, wenn Prozess j die Nachricht empfangen hat. Genauso kann Prozess j erst fortfahren, wenn die Nachricht dort empfangen wurde[9].

Die Funktionen `MPI_Send` und `MPI_Recv` sind folgendermaßen deklariert:

```
1  int MPI_Send(const void *buf, int count, MPI_Datatype datatype,
2               int dest, int tag, MPI_Comm comm);
3
4  int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
5               int source, int tag, MPI_Comm comm,
6               MPI_Status *status);
```

Die Argumente von `MPI_Send` bedeuten

- `buf` - Die Adresse des Arrays der zu sendenden Daten.
- `count` - Die Anzahl der Einträge des Arrays.
- `datatype` - Ein enum, welcher den Datentyp spezifiziert, z.Bsp. `MPI_INT`, `MPI_DOUBLE`, `MPI_CHAR`.
- `dest` - Der Rang des Prozesse, an den die Nachricht geschickt wird.
- `tag` - Eine Art Label, welches man der Nachricht geben kann. Schickt man nur eine Nachricht and `dest`, so kann man `tag` auf 0 setzen.
- `comm` - Der Kommunikator. Wir benutzen wieder `MPI_COMM_WORLD`.

Die Argumente von `MPI_Recv` bedeuten

- `buf` - Die Adresse des Arrays, in dem die empfangenen Daten gespeichert werden sollen.
- `count` - Die Anzahl zu empfangender Einträge des Arrays.
- `datatype` - Ein enum, welcher den Datentyp spezifiziert, z.Bsp. `MPI_INT`, `MPI_DOUBLE`, `MPI_CHAR`.
- `source` - Der Rang des Prozesse, vom dem die Nachricht empfangen wird.

[9]Dieses Blockieren kann man auch umgehen mit nichblockierender Kommunikation. Diese ist uns aber für diese Einführung hier zu kompliziert.

- `tag` - Eine Art Label, welches man der Nachricht geben kann. Schickt man nur eine Nachricht and `dest`, so kann man `tag` auf 0 setzen.
- `comm` - Der Kommunikator. Wir benutzen wieder `MPI_COMM_WORLD`.
- `status` - Man kann sich noch mehr Informationen über die Nachricht speichern lassen. Wir machen von dieser Möglichkeit keinen Gebrauch und benutzen `MPI_STATUS_IGNORE`.

Hier ist ein kurzes Beispiel, welches das Prinzip noch einmal erläutern soll. Der Prozess mit Rang 0 schickt eine Zahl an den Prozess mit Rang 1, welcher diese ausgibt.

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

int main (int argc, char *argv[]) {
  int rank, size;

  MPI_Init (&argc, &argv);

  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  if (size > 1) {
    if (rank == 0) {
      int nachricht = 42; /* wird verschickt */
      /* schicke einen Integer an Prozess 1 */
      MPI_Send (&nachricht, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
    if (rank == 1) {
      int empfange;
      /* empfange einen Integer von Prozess 0 */
      MPI_Recv (&empfange, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
      printf ("Prozess %i hat Nachricht %i empfangen.\n",
              rank, empfange);
    }
  }
  MPI_Finalize ();
  return 0;
}
```

Listing 18: test_send.c

Wir kompilieren und führen das Programm aus.

```
$ mpirun -np 4 ./test_mpi_send
Prozess 1 hat Nachricht 42 empfangen.
```

5.5 Broadcast und Reduce

Es gibt auch Fälle, in denen es notwendig ist, Information von einem Prozess and alle anderen zu verschicken, oder das alle Prozesse Daten haben, welche auf eine bestimmte Art und Weise kombiniert und an einen Prozess geschickt werden sollen.

Ein Beispiel für die erste Operation, das Broadcasten, ist zum Beispiel, wenn ein Prozess Informationen aus einer Datei ausliest, welche für alle Prozesse wichtig sind. Dies ist deutlich effizienter als jeden Prozess die Datei öffnen zu lassen.

Ein Beispiel für die zweite Operation, das Reduce, ist bespielsweise das Berechnen des Maximums eines Vektors, dessen Einträge über alle Prozesse verteilt sind. Jeder Prozess berechnet dann lokal das Maximum unter allen seinen Einträgen und unter diesen wird dann mit Hilfe von `MPI_Reduce` das globale Maximum bestimmt und an einen Prozess gesendet

Ein Beispiel für die zweite Operation, das Reduce, ist bespielsweise das Berechnen des Maximums eines Vektors, dessen Einträge über alle Prozesse verteilt sind. Jeder Prozess berechnet dann lokal das Maximum unter allen seinen Einträgen und unter diesen wird dann mit Hilfe von `MPI_Reduce` das globale Maximum bestimmt und an einen Prozess gesendet.

Hier sind die Signaturen der Funktionen `MPI_Bcast` und `MPI_Reduce`.

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm);

int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm);
```

Die Argumente von `MPI_Bcast` bedeuten dabei folgendes.

- `buffer` - Die Adresse des Arrays der zu verschickenden/zu empfangenden Daten.
- `count` - Die Anzahl der Einträge in `buffer`.
- `datatype` - Der Datentyp der Einträge, siehe oben.
- `root` - Der Prozess von dem aus an alle gesendet wird.
- `comm` - Ein Kommunikator.

Die Argumente von `MPI_Reduce` bedeuten dabei folgendes.

- `sendbuf` - Auf allen Prozessen die Adresse des Arrays der Daten, die reduziert werden sollen.
- `recvbuf` - Auf dem `root`-Prozess die Adresse des Arrays in welchem die Daten empfangen werden sollen.

- `count` - Die Anzahl der Einträge in `sendbuf` und `recvbuf`.
- `datatype` - Der Datentyp der Einträge, siehe oben.
- `op` - Die Operation, die auf die Daten angewendet werden soll. z.Bsp. `MPI_SUM`, `MPI_PROD`, `MPI_MAX`, `MPI_MIN`.
- `root` - Der Prozess an den die reduzierten Daten gesendet werden.
- `comm` - Ein Kommunikator.

Diese Funktionen sind sogenannte kollektive Funktionen und müssen von allen Prozessen aufgerufen werden.

Als erstes Beispiel verwenden wir `MPI_Bcast` um ein `double`-Array an alle Prozesse zu senden.

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main (int argc, char *argv[]) {
5    double daten[10];
6    int rank;
7
8    MPI_Init (&argc, &argv);
9
10   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
11   if (rank == 0) {
12     /* fuelle das Array */
13     int i;
14     for (i = 0;i < 10;i++) daten[i] = 3.14 * i;
15   }
16   /* Verschicke das Array von Rang 0 aus an alle */
17   MPI_Bcast (daten, 10, MPI_DOUBLE, 0, MPI_COMM_WORLD);
18
19   if (rank == 1) {
20     printf ("5. Eintrag im Array ist %f\n", daten[5]);
21   }
22
23   MPI_Finalize ();
24   return 0;
25 }
```

Listing 19: test_bcast.c

Die Ausgabe dieses Programms, wenn es mit mehr als 2 Prozessen aufgerufen wird ist:

```
1 $ mpirun -np 3 ./test_bcast
2 5. Eintrag im Array ist 15.700000
```

Denn jeder Prozess hat jetzt die von Rang 0 berechneten Daten.

Als Beispiel für `MPI_Reduce` geben wir ein Programm an, welches die Summer aller Ränge berechnet.

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

int main (int argc, char *argv[]) {
  int rank, summe;

  MPI_Init (&argc, &argv);

  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  /* Verschicke das Array von Rang 0 aus an alle */
  MPI_Reduce (&rank, &summe, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
      ;

  if (rank == 0) {
    printf ("Habe Summe bekommen: %i\n", summe);
  }

  MPI_Finalize ();
  return 0;
}
```

Listing 20: test_reduce.c

Die Ausgabe ist nun abhängig von der Anzahl der Prozesse.

```
$ mpirun -np 2 ./test_reduce
Habe Summe bekommen: 1
$ mpirun -np 10 ./test_reduce
Habe Summe bekommen: 45
```

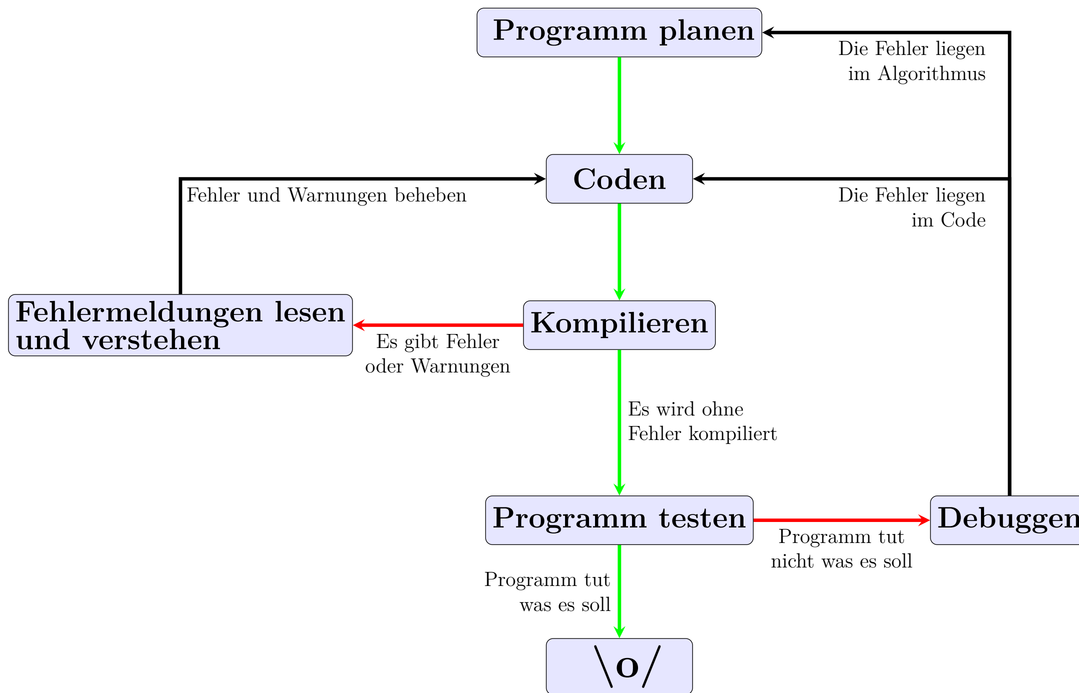


Abbildung 4: Der Programmierprozess (mega-awesome)

6 Debugging

Es passiert selten, dass ein geschriebenes Programm direkt so funktioniert, wie man es sich am Anfang vorgestellt hat. Sind einmal alle Kompiliermeldungen behoben, testet man sein Programm ausführlich. Das Debugging setzt an, wenn das Programm an diesem Punkt nicht korrekt arbeitet.

Debugging bezeichnet also den Prozess zum systematischen Finden und Beheben von Programmierfehlern.[10]

Es gibt verschiedenste Techniken und Hilfsmittel zum Debuggen. Eine leichte und schnelle Art ist sicherlich das Debuggen mit Hilfe von `printf`. Wir werden an dieser Stelle zwei Programme zum Debuggen vorstellen, zum einen `gdb`, ein Debugger, der es ermöglicht das Programm Zeile für Zeile auszuführen und `valgrind`, welches sehr gut geeignet ist, um Speicherlecks und Ähnliches festzustellen.

[10] Funfact: Der Begriff Debuggen geht tatsächlich auf ein echtes Insekt zurück. Grace Hopper (unter anderem Entwicklerin des ersten Compilers und Erfinderin von COBOL, einer der ersten Programmiersprachen) fand 1947 während ihrer Arbeit an Mark II eine Motto im Rechner, welche ihr Ende in einem Relais gefunden hatte und so zu einer Fehlfunktion des Rechners beitrug.

6.1 gdb

Der GNU Debugger (gdb) dient dazu, ein kompiliertes Programm Schritt für Schritt auszuführen und währenddessen, den Inhalt des Speichers zu überprüfen und eventuell weitere Funktionen aufzurufen.

Damit ein `C`-Programm mit gdb gedebuggt werden kann, muss es vorher mit dem Compilerflag `-g` kompiliert worden sein. Wir demonstrieren in Listing 20 wie eine Datei `hallo_welt.c` zum Debuggen kompiliert und das resultierende Programm `hwelt` mit gdb ausgeführt wird.

```
1 gcc -Wall -pedantic -o hwelt hallo_welt.c -g
2 gdb hwelt
```

Listing 21: gdb

Nach dem Aufruf `gdb PROGRAMM` startet der Debugger und sollte nach einigen Zeile informativem Konsolenoutput die Zeilen

```
1 Reading symbols from PROGRAMM...done.
2 (gdb)
```

ausgeben. Dann ist der gdb zum Debuggen bereit. Steht dort stattdessen

```
1 Reading symbols from PROGRAMM...(no debugging symbols found)...done.
2 (gdb)
```

so kann der gdb nicht verwendet werden. Dies liegt in den meisten Fällen daran, dass man das Kompilerflag `-g` vergessen hat.

Wenn alles korrekt geladen wurde, kann nun das Debugging mit gdb beginnen. Wie in einer Konsole gibt es einen interaktiven Prompt, welcher Usereingaben erwartet. Der gdb kann mit verschiedenen Befehlen gesteuert werden. Wir sammeln hier die Wichtigsten:

- **start** - Startet mit der Schritt-für-Schritt Ausführung des Programms.
- **next** - Führt die nächste Zeile des Programms aus. Funktionsaufrufe werden dabei im Ganzen ausgeführt.
- **step** - Wie **next**, aber falls die nächste Zeile einen Funktionsaufruf enthält, so wird an den Anfang der Funktion gesprungen und mit der ersten Zeile in dieser Funktion fortgefahren.
- **list** - Zeige die aktuelle Zeile im Programm an, sowie ein paar Zeilen vorher und nachher.
- **break ZEILE** oder **break FUNKTIONSNAME** oder **break DATEINAME:ZEILE** - Setze einen Breakpoint, das heißt einen Punkt im Programm, bei dem die automatische Ausführung (siehe **continue**) stoppt.

Nach **break** kann entweder eine Zeilennummer, ein Funktionsname, oder ein Dateiname mit Zeilennummer (getrennt durch `":"`) angegeben werden. Die automatische Ausführung stoppt dann in der angegebenen Zeile der aktuellen Quelldatei, bzw. beim Aufruf der genannten Funktion, bzw. in der angegebenen Zeile der angegebenen Quelldatei.

- **info break** - Zeige alle gesetzten Breakpoints an.
- **delete NUM** - Jeder Breakpoint besitzt eine Nummer, diese wird bei **info break** mit angezeigt. Der **delete** Befehl löscht den Breakpoint mit der gegebenen Nummer.
- **continue** - Führe alle folgenden Zeilen automatisch aus bis das Programmende oder ein Breakpoint erreicht ist.
- **run** - Verhält sich wie die Abfolge **start**, **continue**.
- **print AUSDRUCK** - Gebe AUSDRUCK auf der Konsole aus. AUSDRUCK kann zum Beispiel ein Variablename sein, dann wird der Inhalt der Variable ausgegeben. Casting und Dereferenzierungen sind auch möglich. Hat man zum Beispiel eine Variable `p` vom Typ `int *`, kann man mit `print *p` den Wert des Integer anzeigen lassen.
- **watch VARIABLE** - Setzt einen Watchpoint auf eine Variable. Dies bedeutet, dass jedes Mal, wenn sich der Wert der Variable verändert, dieser Wert ausgegeben wird.
- **call FUNKTION(PARAMETER)** - Ruft die Funktion FUNKTION auf und druckt das Ergebnis.
- **where** - Zeigt die sogenannte Backtrace an. Insbesondere enthält diese Informationen darüber, wo im Programm man sich gerade befindet und welche Funktionsaufrufe einen dorthin geführt haben. Dies ist sehr nützlich, wenn man herausfinden möchte, an welcher Stelle ein Programm zum Beispiel einen Speicherfehler (Segmentation Fault) verursacht. Es reicht dann oft die Kombination **run**, **where** um die exakte Zeile zu finden, in der der Speicherfehler geschieht.
- **quit** - Beendet `gdb`.

Tip: Meistens genügt schon die Abfolge **run**, **where**, um die Zeile zu identifizieren, in der ein Programm durch einen Speicherfehler oder Ähnliches terminiert. Wird das Programm nicht unerwartet terminiert, sondern es kommt zu falschen Ergebnissen, dann ist es tatsächlich notwendig Schritt für Schritt die Ausführung zu überwachen und den Status von Variablen zu überprüfen.

Um Schreibarbeit zu ersparen unterstützt `gdb` Abkürzungen dieser Befehle. Die Abkürzung ist für gewöhnlich der erste Buchstabe des Befehls. D.h. **n** für **next**, **s** für **step**, **b** für **break**, etc.. Gibt man eine leere Zeile ein, so wird automatisch der letzte Befehl wiederholt.

Möchte man dem zu debuggenden Programm Kommandozeilenparameter (siege Kapitel 7.1) übergeben, so muss man vor dem Programmnamen noch das Flag `--args` an `gdb` übergeben.

```
1 gdb --args PROGRAMM ARG1 ARG2 ... ARGn
```

6.2 Valgrind

In der nordischen Mythologie ist Valgrind das Haupttor nach Walhalla. Nur die von den Walküren ausgewählten und damit edelsten und ruhmreichsten Krieger, die für würdig befunden wurden, konnten Valgrind unbeschadet passieren.

Das Programm (oder besser die Programmensammlung) `valgrind` spürt Speicherlecks und allgemein Fehler im Speichermanagement eines Programms auf. Nur die Programme, die `valgrind` unbeschadet, d.h. ohne besondere Meldungen, passieren, erweisen sich daher als würdig auch echte Programme genannt zu werden und in die weite Welt entlassen zu werden.

Jedes Programm sollte unabhängig davon, ob man einen Fehler vermutet oder nicht, mindestens einmal mit `valgrind` getestet werden. Dies geschieht recht simpel mit dem Aufruf

```
1 valgrind PROGRAMM
```

Auch für `valgrind` sollte das Programm vorher mit `-g` kompiliert werden.

Bei obigem Aufruf läuft das Programm einmal unter Aufsicht von `valgrind` und alle dabei auftretenden Speicherungereihmtheiten werden auf der Konsole ausgegeben[11].

Wir besprechen das Benutzen von `valgrind` an einem Beispiel, siehe die Datei `valgrind_test.c` in Listing 21.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 void speicherleck (int n) {
 5   int *x;
 6   x = malloc (n * sizeof (int));
 7   x[10] = 4;
 8 }
 9
10 int main () {
11   printf ("Dieses Programm verursacht Speicherfehler.\n");
12   fflush(stdout); /* Druckt den Puffer der Standardausgabe aus. */
```

[11]Achtung: Hierbei läuft das Programm wesentlich langsamer als ohne `valgrind`.

```
13   speicherleck (5);
14   return 0;
15 }
```

Listing 22: valgrind_test.c

Dieser Code hat zwei Speicherprobleme: erstens wird in Zeile 7 auf einen Speicherbereich zugegriffen für den kein Speicher alloziert wurde, und zweitens wird der Speicher, der in Zeile 6 alloziiert wird, nicht wieder freigegeben. Der erste Fehler muss nicht zwangsläufig zu einem Segfault führen, sondern es kann sein, dass dieses Programm scheinbar fehlerfrei läuft. Gerade aus diesem Grund sollte man `valgrind` immer laufen lassen, nicht nur, wenn es offensichtliche Probleme gibt.

Führen wir nun `valgrind` mit dem kompilierten Code aus (wir nennen das Programm hier `valgrind_test`), so sieht die Ausgabe wie folgt aus:

```
1 valgrind ./valgrind_test
```

```
1 ==4077== Memcheck, a memory error detector
2 ==4077== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et
     al.
3 ==4077== Using Valgrind-3.10.1 and LibVEX; rerun with -h for
     copyright info
4 ==4077== Command: ./valgrind_test
5 ==4077==
6 Dieses Programm verursacht Speicherfehler.
7 ==4077== Invalid write of size 4
8 ==4077==    at 0x400625: speicherleck (valgrind_test.c:7)
9 ==4077==    by 0x400653: main (valgrind_test.c:13)
10 ==4077==  Address 0x51fc068 is 20 bytes after a block of size 20
     alloc'd
11 ==4077==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/
     vgpreload_memcheck-amd64-linux.so)
12 ==4077==    by 0x400618: speicherleck (valgrind_test.c:6)
13 ==4077==    by 0x400653: main (valgrind_test.c:13)
14 ==4077==
15 ==4077==
16 ==4077== HEAP SUMMARY:
17 ==4077==     in use at exit: 20 bytes in 1 blocks
18 ==4077==   total heap usage: 1 allocs, 0 frees, 20 bytes allocated
19 ==4077==
20 ==4077== LEAK SUMMARY:
21 ==4077==    definitely lost: 20 bytes in 1 blocks
22 ==4077==    indirectly lost: 0 bytes in 0 blocks
23 ==4077==      possibly lost: 0 bytes in 0 blocks
24 ==4077==    still reachable: 0 bytes in 0 blocks
25 ==4077==         suppressed: 0 bytes in 0 blocks
26 ==4077== Rerun with --leak-check=full to see details of leaked
     memory
27 ==4077==
```

```
28 ==4077== For counts of detected and suppressed errors, rerun with: -
     v
29 ==4077== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from
     0)
```

In Zeile 6 sehen wir den Output unseres Programms, der Rest ist Output von `valgrind`. In Zeile 7 listet `valgrind` einen Speicherzugriffsfehler auf, den unser Programm an dieser Stelle erzeugt. In diesem Fall bedeutet dies ein `Invalid write of size 4`, also einen ungültigen Schreibzugriff von 4 Bytes. Wir erfahren in der nächsten Zeile auch genau an welcher Stelle, nämlich in `valgrind_test.c:7`, also in der Datei `valgrind_test.c` in Zeile 7. Die Zeile 8 und 9 im Listing sind übrigens eine Backtrace (siehe **where** im `gdb` Kapitel).

Nach der Zeile `HEAP SUMMARY` listet `valgrind` auf, wie viel Speicher bei Programmende noch in Gebrauch (d.h. alloziert und noch nicht freigegeben) ist. In diesem Fall sind dies 20 Bytes (5 mal `sizeof(int)`). Danach wird die Anzahl an Allozierungen und Speicherfreigaben insgesamt angegeben.

Unter `LEAK SUMMARY` wird noch einmal zusammengefasst wie viel Speicher tatsächlich durck Lecks verloren gegangen ist. In diesem Fall 20 Bytes.

Diese Informationen reichen meistens aus, um zu identifizieren, welchen Speicher man vergessen hat freizugeben oder wo man auf Speicher zugreift der einem nicht gehört. Falls man jedoch mehr Informationen über die Speicherlecks benötigt, so ruft man (wie in Zeile 26 vorgeschlagen) `valgrind` mit dem Flag `--leak-check=full` auf.

```
1 valgrind --leak-check=full ./valgrind_test
```

```
1  ==4189== Memcheck, a memory error detector
2  ==4189== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et
      al.
3  ==4189== Using Valgrind-3.10.1 and LibVEX; rerun with -h for
      copyright info
4  ==4189== Command: ./valgrind_test
5  ==4189==
6  Dieses Programm verursacht Speicherfehler.
7  ==4189== Invalid write of size 4
8  ==4189==    at 0x400625: speicherleck (valgrind_test.c:7)
9  ==4189==    by 0x400653: main (valgrind_test.c:13)
10 ==4189==  Address 0x51fc068 is 20 bytes after a block of size 20
      alloc'd
11 ==4189==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/
      vgpreload_memcheck-amd64-linux.so)
12 ==4189==    by 0x400618: speicherleck (valgrind_test.c:6)
13 ==4189==    by 0x400653: main (valgrind_test.c:13)
14 ==4189==
15 ==4189==
16 ==4189== HEAP SUMMARY:
17 ==4189==     in use at exit: 20 bytes in 1 blocks
18 ==4189==   total heap usage: 1 allocs, 0 frees, 20 bytes allocated
```

```
19 ==4189==
20 ==4189== 20 bytes in 1 blocks are definitely lost in loss record 1
     of 1
21 ==4189==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/
     vgpreload_memcheck-amd64-linux.so)
22 ==4189==    by 0x400618: speicherleck (valgrind_test.c:6)
23 ==4189==    by 0x400653: main (valgrind_test.c:13)
24 ==4189==
25 ==4189== LEAK SUMMARY:
26 ==4189==    definitely lost: 20 bytes in 1 blocks
27 ==4189==    indirectly lost: 0 bytes in 0 blocks
28 ==4189==      possibly lost: 0 bytes in 0 blocks
29 ==4189==    still reachable: 0 bytes in 0 blocks
30 ==4189==         suppressed: 0 bytes in 0 blocks
31 ==4189==
32 ==4189== For counts of detected and suppressed errors, rerun with: -
     v
33 ==4189== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from
      0)
```

Nun ist zum Output noch der Bereich nach Zeile 21 hinzugekommen. Hier steht zu jedem Speicherblock, der am Ende noch nicht freigegeben ist, wo genau er alloziiert wurde. Wir erfahren an dieser Stelle, dass unser Speicherblock in `valgrind.c` Zeile 6 durch einen Aufruf von `malloc` alloziiert wurde.

Möchte man dem zu testenden Programm noch Kommandozeilenparameter (siehe Kapitel 7.1) übergeben, so kann dies einfach nach dem Programmnamen geschehen, wie man es sonst auch tuen würde.

```
1 valgrind PROGRAMM ARG1 ARG2 ... ARGn
```

7 Weiteres nützliche Sprachkonstrukte

7.1 Kommandozeilenargumente

Ein Programm, kann bei seinem Aufruf in der Kommandozeile zusätzliche Parameter als Zeichenfolgen übergeben bekommen. Um auf diese Parameter zugreifen zu können, ändern wir die Definition der `main`–Funktion zu

```
1 int main( int count, char **args );
```

Das Betriebssystem übergibt dann bei Ausführung des Programms einen Array von Strings in `args` und dessen Länge in `count`. Die Strings `args[0]` bis `args[count-1]` enthalten dann die Kommandozeilenargumente. Auf diese Art und Weise kann einem Programm etwa der Dateiname einer Datei übergeben werden, mit der es arbeiten soll. Betrachte etwa folgendes Programm `arg_print.c`:

```
1 #include <stdio.h>
2
3 int main( int count, char **args ) {
4   int i;
5   for (i = 0; i < count; i++)
6     printf("Argument %i: %s\n",i,args[i]);
7   return 0;
8 }
```

Auf der Kommandozeile würde diese Code das folgende Ergebnis haben:

```
$ gcc -Wall -Wpedantic -o arg_print arg_print.c
$ ./arg_print Dies ist ein Test
Argument 0: ./arg_print
Argument 1: Dies
Argument 2: ist
Argument 3: ein
Argument 4: Test
$ ./arg_print "Dies ist ein Test"
Argument 0: ./arg_print
Argument 1: Dies ist ein Test
```

Der Standard schreibt vor, dass `arg[0]` stets das laufende Programm beschreiben soll. Die meisten (oder vielleicht sogar alle) Betriebssysteme beschreiben das laufende Programm durch seinen Dateinamen.

7.2 Bedingte Auswertung

Es gibt in C einen einzigen eingebauten *ternären* Operator, also einen Operator mit drei Argumenten: Die bedingte Auswertung. Er hat die Syntax

BEDINGUNG ? EXPRESSION1 : EXPRESSION2

Der Wert dieser Expression ist gleich dem Wert von EXPRESSION1, falls BEDINGUNG nicht zu 0 auswertet und andernfalls gleich dem Wert von EXPRESSION2. Als Beispiel eine recht unübersichtlich kurze Signumsfunktion:

```
int sgn(int a) { /* Signumsfunktion */
    return (a<0) ? -1 : (a?1:0);
}
```

7.3 Konstante Variablen

Es gibt in C die Möglichkeit, manche Variablen als "konstant" zu markieren. Dies heißt, dass der Wert dieser Variablen nicht verändert werden *soll*. Es doch zu tun führt zu undefiniertem Verhalten. Dazu schreibt man den Modifikator `const` vor den Variablennamen:

```
double const pi = 3.141592;
```

Genauer: Wenn eine konstante Variable durch eine Zuweisung verändert wird, so gibt der Compiler eine Warnung oder einen Fehler aus. Wenn man also unhöflicherweise die Warnung des Compilers ignoriert, so ist es trotzdem möglich, eine konstante Variable zu verändern. Da dies aber laut Standard zu undefiniertem Verhalten führt, kann man sich nicht darauf verlassen, dass auch das passiert, was man denkt. Der Modifikator `const` ist demnach eine Programmierhilfe, um sicherzustellen, dass eine Variable ihren Wert nicht ändert.

Wir wollen `const` nun etwas allgemeiner verstehen: Wenn eine Variable `v` vom Dereferenzierungslevel n erstellt wird, so können wir hinter das k-te Sternchen bei der Variablendeklaration ein `const` schreiben, damit die $(n-k)$-fache Dereferenzierung von `v` eine konstante Variable ist. Betrachten wir die folgenden Deklarationen:

```
  int                  e =  23;
  int   const          f =  42;
  int         *const   pe =  &e;
  int   const *const   pf =  &f;
  int         *const *ppe = &pe;
  int   const *const *ppf = &pf;
```

Hier werden deklariert:

1. Eine nicht-konstante Variable `e`.

2. Eine konstante Variable `f`.

3. Ein konstanter Pointer `pe` auf eine nicht-konstante Variable.
4. Ein konstanter Pointer `pf` auf eine konstante Variable.
5. Ein nicht-konstanter Pointer `ppe` auf einen konstanten Pointer auf eine nicht-konstante Variable.
6. Ein nicht-konstanter Pointer `ppf` auf einen konstanten Pointer auf eine konstante Variable.

Ohne Warnung kompilieren also die folgenden Statements:

```
1 ppf++;
2 (**ppe)++;
3 e++;
4 (*pe)++;
```

Während die folgenden eine Warnung erzeugen:

```
1 pf++;
2 (**ppf)++;
3 f++;
```

Literatur

[1] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[2] Clelia Albrecht und Felix Boes Johannes Holke. `joelixblas`, eine Implementierung von Matrix-Vektorrechnung zu Lehrzwecken. http://github.com/joelix/joelixblas, März 2017.

[3] Clelia Albrecht und Johannes Holke. *C-Programmierkurs für Mathematikersties*. selfpublished, Oktober 2016. http://fsmath.uni-bonn.de/old/?q=de/node/817.

[4] Jesko Hüttenhain und Lars Wallenborn. C-Programmierkurs, 2016. https://github.com/ThorProgKurs/skript/releases.

[5] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27:3 – 35, 2001. New Trends in High Performance Computing.

[6] Thomas Williams, Colin Kelley, and many others. Gnuplot 4.6: an interactive plotting program. http://gnuplot.sourceforge.net/, April 2013.

A Referenzen

Es folgen Referenzen für einige wichtige Systemmodule, welche C bereits zur Verfügung stellt.

A.1 Referenz <math.h>

Durch einbinden der Systemheaderdatei <math.h> stehen die folgenden mathematischen Funktionen zur Verfügung:

Deklaration	Rückgabewert
Âdouble acos(double x);Â	Arcuscosinus von x
Âdouble asin(double x);Â	Arcussinus von x
Âdouble atan(double x);Â	Arcustangenz von x
Âdouble atan2(double y, double x);Â	Arcustangenz von x/y
Âdouble ceil(double x);Â	Kleinste ganze Zahl, welche $\geq$ x ist
Âdouble cos(double x);Â	Cosinus von x
Âdouble cosh(double x);Â	Cosinus hyperbolicus von x
Âdouble exp(double x);Â	e^x, wobei $e = 2.718281\ldots$
Âdouble fabs(double x);Â	$\|x\|$ (Betrag von x)
Âdouble floor(double x);Â	Größte ganze Zahl, welche $\leq$ x ist
Âdouble ldexp(double x, double y)Â	$x \cdot 2^y$
Âdouble log(double x);Â	Natürlicher Logarithmus von x
Âdouble log10(double x);Â	Zehnerlogarithmus von x
Âdouble pow(double x, double y);Â	x^y
Âdouble sin(double x);Â	Sinus von x
Âdouble sinh(double x);Â	Sinus hyperbolicus von x
Âdouble sqrt(double x);Â	$\sqrt{x}$ (Quadratwurzel aus x)
Âdouble tan(double x);Â	Tangens von x
Âdouble tanh(double x);Â	Tangens hyperbolicus von x

Tabelle 2: Von math.h exportierte Funktionen

Außerdem ist in <math.h> eine Konstante HUGE_VAL definiert: Dies ist der größte Wert, den eine Variable vom Typ double annehmen kann. Fast immer entspricht dies einem symbolischen Wert "∞".

A.2 Referenz <time.h>

Zunächst sind in der <time.h> einige neue Datentypen definiert, welche in der Regel nur Umbenennungen (siehe 2.5) von gewöhnlichen, ganzzahligen Datentypen sind:

Datentyp	Bedeutung
clock_t	Speichert *CPU-Zeiten* (siehe unten)
size_t	Speichert ganzzahlige Größenangaben
time_t	Datentyp für Zeitangaben (meistens eine Anzahl von Sekunden)

Tabelle 3: Einfache Datentypen aus time.h

Eine CPU braucht für jeden Maschinenbefehl die gleiche Zeit. Dieses Zeitintervall wird auch als *Takt* bezeichnet, und eine CPU-Zeit entspricht einer gewissen Anzahl Takte.

Weiterhin definiert ist die Struktur `struct tm`, welche alle Komponenten einer Kalenderzeit im Gregorianischen Kalender enthält. Ihre Definition muss mindestens die folgenden Felder enthalten:

```
 1 struct tm {      /* Feldbeschreibung          Intervall */
 2                  /* ------------------------------------ */
 3   int tm_sec;    /* Sekunden nach der Minute     [0,59] */
 4   int tm_min;    /* Minuten nach der Stunde      [0,59] */
 5   int tm_hour;   /* Stunden seit Mitternacht     [0,23] */
 6   int tm_mday;   /* Tag des Monats               [1,31] */
 7   int tm_mon;    /* Monat seit Januar            [0,11] */
 8   int tm_year;   /* Jahreszahl                 [1900,[ */
 9   int tm_wday;   /* Tag seit Sonntag              [0,6] */
10   int tm_yday;   /* Tag seit dem 1. Januar     [0,365] */
11                  /* ------------------------------------ */
12   int tm_isdst; /* Zeigt an, ob es sich um Sommerzeit */
13                  /* (daylight saving time) handelt.    */
14                  /* 0 steht hierbei für "Nein", -1 für */
15                  /* "unbekannt", alles andere für "Ja" */
16 };
```

Tabelle 4 gibt Aufschluss über die Funktionen zum Berechnen der Zeit, welche in <time.h> definiert sind:

Deklaration	**Effekt**
Â`time_t time(time_t *p);`Â	Gibt bei Misserfolg `-1` zurück, andernfalls die momentane Zeit. Ist `p` $\neq$ `NULL`, so wird dieses Ergebnis in `*p` zusätzlich noch abgespeichert.
Â`clock_t clock();`Â	Gibt bei Misserfolg `-1` zurück, andernfalls die seit Programmstart vergangene CPU-Zeit.

Tabelle 4: Funktionen zum Ermitteln der Zeit

Die folgenden Umrechnungsfunktionen geben stets den gleichen Pointer auf einen internen `struct tm` zurück, welcher *auf keinen Fall* vom Programmierer freigegeben werden sollte. Des weiteren Überschreibt jeder Aufruf einer Umrechnungsfunktion die Werte dieser Struktur:

Deklaration	**Umrechnung in**
Â`struct tm *gmtime(time_t*);`Â	UTC-Kalenderzeit
Â`struct tm *localtime(time_t*);`Â	Lokale Kalenderzeit

Tabelle 5: Umrechnungsfunktionen für Zeiten

Um die Funktion `clock()` sinnvoll zu verwenden, ist in der `<time.h>` noch die Konstante `CLOCKS_PER_SEC` definiert, welche die Anzahl CPU-Takte pro Sekunde angibt.

A.3 Referenz <stdlib.h>

Zunächst ist in der <stdlib.h> der Datentyp size_t (siehe dazu auch Tabelle 3 in Anhang A.2) und die Konstante NULL definiert. Weiterhin stehen durch einbinden von <stdlib.h> die folgenden Funktionen zur Verfügung:

Deklaration	Funktionsweise
Âvoid *malloc(size_t c);Â	Siehe [4].
Âvoid *realloc(void *p, size_t c);Â	Siehe [4].
Âvoid free(void *p);Â	Siehe [4].
Âint system(char *s);Â	Hat den gleichen Effekt, als wäre die Zeichenfolge s auf der Kommandozeile eingegeben worden.
Âint abs(int i);Â	Berechnet den Absolutwert von i.
Âlong labs(long i);Â	Berechnet den Absolutwert von i.
Âint atoi(char *s);Â	Gibt die Ganzzahl zurück, die in Dezimalschreibweise in s steht.
Âdouble atof(char *s);Â	Gibt die Gleitkommazahl zurück, die in s steht.

Tabelle 6: Funktionen aus stdlib.h

Des Weiteren stehen zur Verfügung:

Deklaration	Effekt
Âvoid srand(unsigned s);Â	Initialisiert den internen Zufallsgenerator mit dem Wert s.
Âint rand();Â	Liefert eine Pseudo–Zufallszahl aus dem Bereich von 0 bis RAND_MAX (wobei RAND_MAX eine Konstante ist, die garantiert ≥ 32767 ist).

Tabelle 7: Funktionen für Zufallszahlen aus stdlib.h

Der Zufallszahlengenerator muss per srand initialisiert werden. Wird rand ohne vorherige Initialisierung aufgerufen, sind die zurückgegebenen Zufallszahlen bei jedem Programmablauf die gleichen (und damit nicht wirklich zufällig). Für gewöhnlich verwendet man die Funktion time (siehe Anhang A.2), um den Zufallszahlengenerator mittels

```
1 srand( time( NULL ) );
```

zu initialisieren.

A.4 Referenz <limits.h>

In der `<limits.h>` sind die folgenden Konstanten definiert, welche maximale und minimale Größe ganzzahliger Datentypen enthalten.

Konstante	**Mindestwert**	**Bedeutung**
`CHAR_BIT`	8	Anzahl Bits in einem Byte
`SHRT_MIN`	-32767	Minimalwert für `signed short`
`SHRT_MAX`	+32767	Maximalwert für `signed short`
`USHRT_MAX`	65535	Maximalwert für `unsigned short`
`INT_MIN`	-32767	Minimalwert für `signed int`
`INT_MAX`	+32767	Maximalwert für `signed int`
`UINT_MAX`	65535	Maximalwert für `unsigned int`
`LONG_MIN`	-2147483647	Minimalwert für `long int`
`LONG_MAX`	+2147483647	Maximalwert für `long int`
`ULONG_MAX`	4294967295	Maximalwert für `unsigned long int`

Tabelle 8: Limits für Ganzzahlen

In der Realität sind die Werte häufig betragsmäßig größer als der Mindestwert, insbesondere für `int` und `long int`.

A.5 Referenz <float.h>

In der `<float.h>` sind die folgenden Konstanten definiert, welche im Zusammenhang mit Fließkommazahlen häufig von Bedeutung sind:

Konstante	**Bedeutung**
`FLT_MAX`	Größter, endlicher Wert für ein `float`
`FLT_MIN`	Kleinster positiver, endlicher Wert für ein `float`
`FLT_EPSILON`	Kleinster positiver `float`–Wert ε, für den $1.0 + \varepsilon \neq 1.0$ gilt
`DBL_MAX`	Größter, endlicher Wert für ein `double`
`DBL_MIN`	Kleinster positiver, endlicher Wert für ein `double`
`DBL_EPSILON`	Kleinster positiver `double`–Wert ε, für den $1.0 + \varepsilon \neq 1.0$ gilt

Tabelle 9: Limits und Konstanten für Fließkommazahlen

Die Konstante `DBL_EPSILON` eignet sich hervorragend als Fehlertoleranz bei numerischen Algorithmen.