Programmieren in C

Notizen zur Vorlesung, Wintersemester 2021/2022

PD Dr. Andreas Paffenholz

10. Februar 2022

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Version vom 10. Februar 2022

Inhaltsverzeichnis

Version vom 10. Februar 2022

Version vom 10. Februar 2022

Version vom 10. Februar 2022

Vorbereitung

Programmieren kann man nur durch Üben und Ausprobieren lernen. Sie sollten also von Anfang an versuchen, die Codebeispiele aus diesem Skript auf Ihrem eigenen PC zu übersetzen, Variationen auszuprobieren und eigene Programme zu schreiben. Dabei können Sie am Anfang natürlich auch Code aus anderen Programmen in Ihr eigenes kopieren, wenn Sie zwar wissen (oder wenigstens glauben zu wissen), was die Abfolge von Anweisungen macht, aber sie vielleicht nicht hätten selbst schreiben können. Je mehr Erfahrung Sie später erwerben, je mehr davon werden Sie verstehen.

Um selbst Programme zu schreiben, brauchen Sie zwei Dinge:

- Einen `C` -*Compiler* wie zum Beispiel `gcc`, und
- einen *Texteditor* wie z.B. `Visual Studio Code`.

Im moodle der Vorlesung finden Sie für die verschiedenen Betriebssysteme `Windows`, `MacOS`, und `Linux` eine Installationsanleitung und einige einfache Tests, ob Ihre Installation erfolgreich war.

Die meisten Beispiele aus diesem Skript finden Sie hier. Die Sammlung wird im Lauf des Semesters ergänzt. Sie können sich die Programme auf Ihren PC kopieren. Dazu öffnen Sie ein Terminal, und geben dort

```
git clone https://git.rwth-aachen.de/programmieren_in_c/2021_22_codebeispiele
```

ein. Damit erhalten Sie ein neues Unterverzeichnis `2020_21_codebeispiele`, in dem alle bisher veröffentlichten Beispiele liegen. Die Beispiele des Skripts finden Sie im Unterordner `skript`. Wenn Sie im Ordner `2021_22_codebeispiele` sind, können Sie mit

```
git pull
```

die Sammlung aktualisieren. Sie sollten alle Programme ausprobieren und versuchen, sie zu verändern (andere Rechnungen, andere Ausgaben, neue Parameter, Teile in Funktionen auslagern, etc.) sobald wir diese Themen in der Vorlesung behandelt haben. Dafür sollten Sie allerdings *immer* eine Kopie der Originaldatei machen, zum Beispiel mit

```
cp collatz.c collatz_variante_1.c
```

da sonst `git pull` möglicherweise nicht mehr funktioniert. Falls Sie doch mal eine Originaldatei bearbeitet haben, dann können Sie sie in der Regel mit

```
git checkout -- collatz.c
```

wieder auf den Ausgangszustand zurücksetzen (den Dateinamen müssen Sie natürlich auf die Datei anpassen, die Sie zurücksetzen wollen).

Version vom 10. Februar 2022

`git` ist ein *Versionskontrollsystem*. Mit einem solchen System können Sie Änderungen an Dateien nachvollziehbar machen und Änderungen, die andere an einer Kopie der Dateien vorgenommen haben, in Ihre Version aufnehmen. Das kann lokal oder dezentral zwischen mehreren Benutzern erfolgen, in den meisten Fällen benutzt man dafür jedoch eine zentrale Stelle, an der die Hauptversion der Dateien liegt (das *repository)*, und die sich alle Bearbeiter:innen kopieren können und nach Bearbeitung wieder mit der Hauptversion vereinigen können. Anschließend können alle anderen Bearbeiter:innen ihre lokale Kopie mit den Änderungen aktualisieren. `git` eignet sich also seh gut dafür, gemeinsam an einem Projekt zu arbeiten und alle Änderungen regelmäßig zwischen den Beteiligten auszutauschen. So ist auch immer klar, was die aktuelle Version ist, und Sie können sehen, was die anderen gemacht haben.

In unserem Fall ist die zentrale Stelle an der Adresse

`https://git.rwth-aachen.de/programmieren_in_c/2021_22_codebeispiele`.

Mit `git clone` erstellen Sie eine Kopie der Dateien, mit `git pull` aktualisieren Sie Ihre Kopie. Mit `git push` könnten Sie, wenn Sie entsprechenden Zugang haben, Ihre Änderungen in die Hauptversion übertragen. Dieser Zugang fehlt Ihnen in diesem Fall, aber Sie können selbst unter `git-ce.rwth-aachen.de` eigene Projekte anlegen und mit anderen zusammen bearbeiten. Mit Ihrer TU-ID können Sie sich dort anmelden. Wir schauen uns das im Lauf der Vorlesung genauer an.

Bei allen Programmen, die in diesem Skript auftauchen und in der Sammlung enthalten sind, steht oberhalb, wo sie das Programm unterhalb des Ordners `skript` finden. Zum Beispiel stünde

Programm 1: `kapitel_01/approx_sin.c`

```
1 // Hier steht der Programmcode
```

im Unterverzeichnis `kapitel_01` und hat den Namen `approx_sin.c`.

Ein auf dem PC ausführbares *Programm* kann aus mehreren Dateien mit `C`-Code bestehen, dabei gibt es immer eine *Haupt*datei, in der der Code steht, mit dem die Ausführung des Programms beginnt (erkennbar daran, dass in der Datei eine Zeile, die mit `int main` beginnt, der *Haupt*funktion, steht). An Anfang dieser Datei steht ein Hinweis, wie das Programm übersetzt und benutzt werden kann. Das kann zum Beispiel so aussehen:

```
1  /************************************
2  * Beispielprogramm zur Vorlesung
3  * Einfuehrung in die Programmierung I
4  * Andreas Paffenholz
5  * TU Darmstadt, Wintersemester 2021/22
6  * (c) 2021-
7  *
8  * Laenge der Collatzfolge
9  *
10 * Uebersetzen mit
11 * gcc collatz.c -o collatz
12 * Aufruf mit
13 * ./collatz
14 ************************************/
```

Um das Programm zu übersetzen und auszuführen, wechseln Sie in das Verzeichnis, in der die Datei liegt (wenn Sie es ausprobieren wollen, dann finden Sie das Beispiel

unter `2021_22_codebeispiele/skript/vorbereitung`) und rufen dort den Befehl auf, der unter `Uebersetzen mit` steht, also

```
gcc collatz.c -o collatz
```

Anschließend können Sie es mit der Anweisung, die unter `Aufruf mit` steht, ausführen, in diesem Fall also mit

```
./collatz
```

In dem Verzeichnis finden Sie auch eine `Makefile` für alle Programme im Verzeichnis. Diese können Sie mit dem Aufruf

```
make
```

ausführen und so alle Programme des Verzeichnisses auf einmal übersetzen. In einer `Makefile` kann man Regeln festlegen, nach denen der Code eines Programms übersetzt werden kann. Das ist sehr nützlich, weil es oft sinnvoll und notwendig ist, den Code eines Programms auf mehr als eine Datei zu verteilen, die dann separat übersetzt und miteinander in der richtigen Reihenfolge verbunden werden müssen. Wir werden das schon in den ersten Vorlesungen machen, um unseren Code zu strukturieren und Tests für einzelne Teile des Programms schreiben zu können, die wir benutzen werden, um Ihren Code zu überprüfen. Später können Sie solche Tests auch selbst schreiben.

Den dann komplizierteren Ablauf für das Übersetzen eines Programms muss man in einer `Makefile` nur einmal festlegen und kann ihn dann mit einem einzigen Aufruf von `make` immer wieder ablaufen lassen. `make` kann noch einiges mehr und hat daher, wie wir sehen werden, eine manchmal gewöhnungsbedürftige Syntax. Wir gehen später genauer auf einige einfache Varianten ein.

Dieses Skript basiert auf einer ersten Version aus dem Wintersemester 2020/21. Ich werde es parallel zur Vorlesung aber an die Inhalte dieses Semesters anpassen, Fehler korrigieren und Anregungen von Studierenden aus dem letzten Kurz einarbeiten. Das Skript wird daher am Anfang sicherlich wieder neue Tippfehler und vielleicht auch neue inhaltliche Fehler enthalten und es werden Teile fehlen oder nicht ausreichend erklärt sein.

Hinweise darauf können Sie mir gerne per Mail oder Discord schicken, oder einem der Tutoren in den Tutorien oder mir nach der Vorlesung mitteilen. Wenn Sie Fragen zum Inhalt haben können Sie die ebenfalls in Discord oder nach der Vorlesung stellen. In Discord finden Sie wahrscheinlich auch oft einen der Tutoren, der bei Fragen direkt weiterhelfen kann.

Darmstadt, Oktober 2021

Version vom 10. Februar 2022

Version vom 10. Februar 2022

1 Einführung

Die Benutzung von Computern ist in der modernen Mathematik kaum noch wegzudenken und kann auf vielfältige Weise erfolgen. Das beginnt mit dem Einsatz für Literaturrecherche und die Vorbereitung neuer Aufsätze oder Bücher, den Einsatz von Datenbanken oder mathematischer Software zum Testen von Hypothesen und Vermutungen über Software zur Klassifikation (endlicher) Familien von mathematischen Objekten für die weitere Betrachtung bis zur Erstellung spezieller Software für mathematische Probleme.

Mathematische Software kann dabei als allgemeines Werkzeug für mathematische Betrachtungen (ggf. in einem eingegrenzten Gebiet der Mathematik, z.B. Geometrie oder Algebra) sein, oder individuell für eine konkrete mathematische Frage (z.B. um systematisch Beispiele für eine konkrete Vermutung zu testen) geschrieben werden.

1.1 Inhalte und Ziele der Vorlesung

In dieser Vorlesung sollen Sie anhand einer konkreten Programmiersprache alle grundlegenden Aufgaben bei der Programmierung erlernen. Wir werden dabei immer auf die speziellen Anforderungen in der Mathematik achten.

Wie beim Lernen von Fremdsprachen wie Englisch oder Französisch gehört auch zum *Programmieren* mehr als die Kenntnis der Wörter und Grammatik der Sprache. Wir werden uns zum Beispiel auch typische Formulierungen in der Sprache ansehen und uns mit sinnvollen Strukturierungen eines Programms, also eines Textes in der Sprache, beschäftigen.

Programmieren beginnt in der Regel lange vor der ersten konkreten Zeile des Programmcodes, die wir tatsächlich in einen Editor schreiben, mit der Überlegung, was unser Programm machen soll und welche Voraussetzungen wir annehmen, mit der Planung, wie wir unser Programm aufbauen wollen, mit der Überlegung, welche Daten uns zur Verfügung stehen und wie wir unsere Daten effizient so speichern, dass wir schnell an die wesentlichen Informationen kommen, und oft auch der mit der Auswahl oder der Entwicklung geeigneter Algorithmen für unsere Problemstellung. Das ist ähnlich zu der Herangehensweise, die Sie auch schon für Aufsätze oder Erörterungen kennen, wo Sie sich vor dem eigentlichen Schreiben mit der Fragestellung und den Quellen beschäftigen und anschließend eine Gliederung entwerfen müssen (sollten), die sie dann mit Stichpunkten und einzelnen ausgearbeiteten Entwürfen füllen, bevor Sie den endgültigen Text formulieren. Wie bei Aufsätzen sollten Sie auch bei Programmen sich nach Fertigstellung das Ergebnis erneut durchlesen, Ihre Strukturierung überprüfen, und nach Verbesserungen suchen.

Jenseits von sehr einfachen Programmen ist es unwahrscheinlich, dass Ihr Programm von Anfang an fehlerfrei ist (wie bei Textaufsätzen auch). Sie werden es mehrfach

Version vom 10. Februar 2022

durchgehen und von anderen ausprobieren lassen, bis es zuverlässig funktioniert. Wir werden uns daher auch ansehen, mit welchen Methoden man auf Fehlersuche gehen kann, und wie man durch begleitende Tests des Programms einzelne Teile kontinuierlich überprüfen kann, um bestehenden Code fehlerfrei zu halten.

Auch wenn man sich an Anfang beim Entwurf des Programms schon Algorithmen für alle auftretenden Teilprobleme überlegt hat, sollte man die Effizienz seines Programms regelmäßig betrachten und bei Bedarf Anpassungen vornehmen. Meistens kann man Teile des Programms auf viele unterschiedliche Weisen in konkreten Programmcode umsetzen, und erst später fällt auf, wie sich manche Teile kürzer, effizienter und leichter verständlich schreiben lassen. Oft bemerkt man auch erst während des Programmierens, an welchen Stellen ein Programm (zu) viel Zeit oder Speicher braucht, oder wo man nicht sinnvoll und ausreichend schnell auf die richtigen Daten zugreifen kann. In solchen Fällen wird man Teile des Programms neu entwickeln oder die verwendeten Datenstrukturen anpassen. Um in einem solchen Fall nicht das komplette Programm umschreiben zu müssen, ist es sinnvoll, einzelne Teile des Programms modular anzulegen und mit einfachen Schnittstellen zwischen den Teilen zu arbeiten.

Wie sie schon von Betriebssystemen und Apps auf Ihrem Mobiltelefon kennen, besteht in der Regel der Bedarf, später verbesserte oder korrigierte Versionen der Software bereitzustellen, oder sie an neue Rahmenbedingungen anzupassen. Sie sollten daher bereits beim ersten Schreiben darauf achten, dass Sie später noch alle Ihre Entscheidungen für Algorithmen, Datenstrukturen und Programmentwurf nachvollziehen können und entsprechend dokumentiert haben. Auch hierfür werden wir uns einige Strategien im Lauf des Semesters ansehen.

Viele Mathematiker wechseln nach dem Studium in Berufe, die Programmierkenntnisse erfordern, und Sie sollten früh anfangen, hierin Erfahrungen zu sammeln. Sie können diese Kenntnisse aber auch schon im Studium einsetzen, z.B. um für manche Berechnungen bei den Hausaufgaben, für Praktika, oder bei der Benutzung anderer mathematischer Software. In einigen Arbeitsgruppen gibt es auch interessante Themen für Bachelor- oder Masterarbeiten, die die Entwicklung neuer Software oder Fortführung bzw. Ergänzung bestehender Software beinhalten.

1.2 Programmiersprachen

Aktuelle übliche Prozessoren in Computern beherrschen einen inzwischen relativ großen Befehlssatz. Diese *Maschinenbefehle* sind an die Architektur des Prozessors und die direkt dazugehörenden Komponenten wie Register und verschiedene Datenzwischenspeicher (Caches) angepasst. Für Menschen ist solcher Maschinencode weder gut lesbar, noch kann man darin sinnvoll und problemangepasst ein Programm, also eine Abfolge von Befehlen, die eine vorgebende Aufgabe lösen sollen, schreiben.

Die Liste der Instruktionen eines Prozessors ist zudem typischerweise auf genau diesen Prozessor zugeschnitten. Ein Programm, das mit diesen Instruktionen geschrieben wurde, wird nicht auf anderen Prozessoren laufen.

Programme werden daher selten direkt in Maschinensprache geschrieben, sondern in einer *Programmiersprache* (einer sogenannten *Hochsprache* im Gegensatz zur *Maschinensprache*) wie z.B. `Java`, `python` oder `C/C++`. Diese Sprachen sind einfacher zu

verstehen, bieten komplexere Programmstrukturen und sind dadurch deutlich kürzer, sind intuitiver Vorstellung von Programmabläufen angepasst und können einen auf spezielle Probleme angepassten Satz von Methoden zur Verfügung stellen. Um diese Programme auszuführen, braucht man dann ein spezielles Programm, das die Abfolge der Instruktionen in einer Hochsprache in die Maschinensprache des Prozessors übersetzt, auf dem das Programm ausgeführt werden soll.

Inzwischen gibt es eine große Anzahl unterschiedlicher Programmiersprachen, die an unterschiedliche Anforderungen der Aufgaben, die damit gelöst werden sollen, angepasst sind. Das können z.B.

(i) die Effizienz des resultierenden Programms,

(ii) die Möglichkeit, das Programm schnell und unkompliziert entwickeln zu können,

(iii) eine einfache Anpassung an veränderte Anforderungen, oder

(iv) die Bereitstellung von Funktionen für spezielle Gegebenheiten (z.B Webdienste, die auf dem Rechner, der eine Webseite aufruft, ein Programm ausführen wollen)

sein. Mit zunehmender Abstraktion und Effizienz der Prozessoren entstehen (und verschwinden) neue Sprachen in immer schnellerer Folge. Trotz aller Unterschiede bauen jedoch viele Programmiersprachen auf den gleichen Grundstrukturen und den gleichen algorithmischen Ideen auf. Es ist daher meistens deutlich leichter, in einer neuen Sprache zu programmieren, wenn man eine erste gründlich verstanden hat.

In dieser Vorlesung betrachten wir die Sprache `C`. Da viele Sprachen, wie z.B. `Java` oder auch Skriptsprachen wie `Python`, ähnliche Strukturen aufweisen, schränken wir uns damit nicht wesentlich ein. Auch wenn Sie noch nicht programmieren können, sehen Sie wahrscheinlich die Ähnlichkeit zwischen dem Code in den Beispielen Programme 1.1 bis 1.5. In allen Programmen wird $\sin(x)$ über die Summation der ersten n Terme der Reihenentwicklung

$$\sin(x) \;=\; \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{(2k+1)!}\,,$$

approximiert, wobei n eine natürliche Zahl ist, die als Eingabe zusammen mit x abgefragt wird.

Insbesondere können Sie wahrscheinlich die Zeilen in der Mitte der Programme, in denen die Summenterme in einer Variable `summand` ausgerechnet werden und die Schleife, die (mit dem *Schlüsselwort* `for`) diese Berechnung für die ersten n Terme wiederholt, identifizieren. Sie werden anschließend der Summe in der Variable `approx_sin` hinzugefügt. Dabei wird das x, für das wir $\sin(x)$ approximieren wollen, in der Variablen `arg` übergeben, und die Anzahl der Terme der Summe in `n`. Der erste Summand der Summe (für $k = 0$) ist x, wir initialisieren also `summand` und `approx_sin` damit. Wenn nun s_k der Summand zu k ist, dann erhalten wir s_{k+1} durch

$$s_{k+1} \;:=\; s_k \cdot \frac{-x \cdot x}{2 \cdot k \cdot (2 \cdot k + 1)}\,.$$

Wir ersetzen also in jedem Schritt `summand` mit diesem Ausdruck, und haben so in `summand` immer s_{k+1} stehen. Das addieren wir in jedem Schritt zu `approx_sin`, bis wir $k = n - 1$ erreicht haben.

Version vom 10. Februar 2022

Programm 1.1: `kapitel_01/approx_sin.c`

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

int main( int argc, char** argv ) {

   float arg, approx_sin, summand;
   int n, k;
   // Argument fuer Sinus
   arg = atof(argv[1]);
   // Anzahl ausgewerteter Summanden
   n = atoi(argv[2]);

   // Summand fuer i=0
   approx_sin = arg;
   summand = arg;
   for ( k=1; k<n; k=k+1 ) {
      summand = summand
          * (-1)*arg*arg/(2*k*(2*k+1));
      approx_sin = approx_sin + summand;
   }

   printf("Der Sinus von %f ist naeherungsweise %f\n", arg, approx_sin);

   return 0;
}
```

Die Teile am Anfang und Ende, in denen das Programm initialisiert wird, die Daten eingelesen und das Ergebnis ausgegeben werden, sind oft unterschiedlicher zwischen den Sprachen und für sie wahrscheinlich noch unverständlich. Strukturell sollten sie Ihnen aber trotzdem ähnlich erscheinen. Wenn wir einmal wissen, was in Programm 1.1 passiert, sollte uns daher die Formulierung in den anderen Sprachen leichter fallen. Dann kennen wir schon den prinzipiellen Ablauf eines Programms, und müssen uns nur noch um die konkrete Formulierung in dieser Sprache kümmern.

1.2.1 Interpretierte und kompilierte Sprachen

Wir haben oben schon festgehalten, dass wir zu einer Programmiersprache ein spezielles Übersetzungsprogramm brauchen, um ein Programm ausführen zu können. Das Übersetzungsprogramm muss unsere Programmiersprache in eine Liste von Anweisungen übersetzen, die der Prozessor versteht.

Das kann zu unterschiedlichen Zeitpunkten erfolgen. Bei den sogenannten *kompilierten* Sprachen wird die Übersetzung einmal nach der Programmerstellung (und jeder Änderung) vorgenommen (mit einem *Compiler)*. Das Programm liegt anschließend auch in *Maschinencode* vor, der unabhängig von dem von uns erstellten Programmcode ausgeführt oder verteilt werden kann. Die Sprache `C`, die wir uns ansehen, gehört zu dieser Familie von Sprachen.

Bei den *interpretierten* Sprachen erfolgt die Übersetzung erst bei der Ausführung des Programms (mit einem *Interpreter)*. Dazu muss zum Zeitpunkt der Ausführung auf dem

Programm 1.2: `kapitel_01/ApproxSin.java`

```java
public class ApproxSin
{
   public static void main(String args[])
   {
     float arg = Float.parseFloat(args[0]);
     int n = Integer.parseInt(args[1]);

      float approx_sin = arg;
      float summand = approx_sin;
      for (int i = 1; i < n; i++) {
         summand = summand
                  *(-1)*arg*arg/(2*i*(2*i+1));
         approx_sin += summand;
      }
      System.out.printf("Der Sinus von %.10f" +
                     " ist naeherungsweise %.10f\n",
                     arg, approx_sin);
   }
}
```

Programm 1.3: `kapitel_01/approx_sin.py`

```python
import sys

# Argument fuer Sinus
arg = float(sys.argv[1])
# Anzahl ausgewerteter Summanden
n = int(sys.argv[2])

# Summand fuer i=0
approx_sin = arg
summand = approx_sin

for k in range(1,n):
  summand = summand * (-1)*arg*arg/(2*k*(2*k+1));
  approx_sin = approx_sin + summand

print("Der Sinus von %f ist naeherungsweise %f \n" \
     %(arg,approx_sin))
```

PC daher sowohl der Programmcode als auch ein passendes Übersetzungsprogramm vorhanden sein. Beispiele für interpretierte Sprachen sind `Python`, `Perl`, oder `PHP`.

Beide Ansätze haben Vor- und Nachteile. Da bei kompilierten Sprachen zur Laufzeit des Programms die Zeit für die Übersetzung wegfällt, sind diese in der Regel schneller. Zudem hat das Übersetzungsprogramm, das sich vor der Ausführung den kompletten Code ansehen kann, mehr Möglichkeiten zur Optimierung, sodass hier in der Regel auch der effizientere Maschinencode entsteht. Interpretierte Sprachen haben hingegen den Vorteil, dass es oft leichter ist, Programme in der Sprache zu erstellen und auch Teile des Programms vor der Fertigstellung schon auszuführen und zu testen. Auch die Fehlersuche und das Testen von Varianten ist leichter. Zudem können interpretierte Sprachen oft auch *interaktiv* ausgeführt werden, d.h. wir können Anweisungen einzeln an den Interpreter übergeben, und das Ergebnis ansehen und danach entscheiden, wie wir weitermachen wollen.

Da Maschinencode immer speziell für einen Prozessor (oder eine Prozessorfamilie)

Programm 1.4: kapitel_01/approx_sin.pl

```
my ($arg,$n) = @ARGV;

my $approx_sin = $arg;
my $summand = $approx_sin;
foreach my $k (1..$n-1) {
   $summand = $summand
            *(-1)*$arg*$arg/(2*$k*(2*$k+1));
   $approx_sin += $summand;
}

printf "Der Sinus von %.10f ist naeherungsweise %.5f\n",
      ($arg, $approx_sin);
```

Programm 1.5: kapitel_01/approx_sin.jl

```
function approx_sin(arg::Float32, n::Int)
   approx_sin = arg
   summand = arg
   for k in 1:n
      summand = summand *(-1)*arg*arg/(2*k*(2*k+1));
      approx_sin += summand;
   end
   return approx_sin
end

arg = parse(Float32, ARGS[1])
n = parse(Int, ARGS[2])

sin = approx_sin(arg,n)
println("Der Sinus von $arg ist naeherungsweise $sin")
```

erstellt werden muss, muss bei der Verteilung von kompiliertem Code auch eine Version des Maschinencodes für jede Prozessorfamilie erstellt werden, auf dem das Programm benutzt werden soll. Bei interpretierten Sprachen hingegen kann der gleiche Code auf allen Prozessoren ausgeführt werden, für die ein Übersetzungsprogramm zur Verfügung steht.

Um diese Flexibilität mit der höheren Effizienz kompilierter Sprachen zu verbinden gehen manche Sprachen einen Mittelweg. Dabei wird das Programm mit einem Compiler in einen maschinencodenahen, aber prozessorunabhängigen, Bytecode übersetzt, für den es dann zu vielen Prozessorfamilien einen Interpreter gibt. Das bekannteste Beispiel hier ist die Sprache `Java`.

Prinzipiell ist es natürlich auch möglich, ein Programm direkt in Maschinencode (bzw. in *Assembler*, einer auf den Befehlssatz des Prozessors ausgerichteten Sprache) zu schreiben. Befehlssätze von Prozessoren, die an den Möglichkeiten der Chiparchitektur ausgerichtet sind, und menschliches Denken unterscheiden sich allerdings erheblich, und viele Abläufe, die sich für uns mit wenigen Worten beschreiben lassen (z.B. die Anweisung, bestimmte Teile eines Prozesses zu wiederholen), erfordern lange Codesequenzen in Maschinencode, die sich leicht automatisiert erzeugen lassen. Es hat sich daher als sinnvoll erwiesen, Sprachen zu entwickeln, die sich mit ihren bereitgestellten Datenstrukturen und Methoden an den Problemen und Aufgaben orientieren, die wir damit lösen wollen. Diese Entwicklung schreitet mit den wachsenden Anforderungen an Programme und den komplexeren Prozessorarchitekturen auch immer weiter voran.

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

Tabelle 1.1: Alle C-Befehle

Nur im Bereich von wirklich zeitkritischen Anwendungen, oder in Bereichen, wo Stromversorgung, Speicherverfügbarkeit oder Prozessorarchitektur beschränkt sind, werden noch maschinennahe Sprachen verwendet.

1.2.2 Die Sprache C

Die Sprache C ist eine der ersten großen Programmiersprachen, die entwickelt wurden. Sie existiert seit ca. 1970 und ist damit auch eine der ältesten noch aktiv genutzten Programmiersprachen, und gleichzeitig die Grundlage und Inspiration für viele weitere Programmiersprachen.

C hat einen sehr kleinen Satz von Wörtern in seiner Sprache. Insgesamt sind es nur die 32 Wörter, die in Table 1.1 stehen. Auch wenn Sie noch kein C können, können Sie vermutlich bei manchen erraten, wofür sie gut sind. Zu diesen 32 Wörtern kommen allerdings noch einige weitere, die in den sogenannten *Standardbibliotheken* stehen. Das sind in C geschriebene Programmteile, die Funktionen bereitstellen, die in dem kleinen Befehlssatz von C nicht vorkommen, aber häufig benutzt werden (z.B. Funktionen für die Ein- und Ausgabe auf dem Bildschirm).

Der kleine Befehlssatz macht es einerseits leicht, in C zu programmieren, da Sie nur 32 Vokabeln (und die zugehörige Grammatik) lernen müssen, und andererseits schwierig, da Sie viele Aufgaben, für die es in anderen Sprachen eigene Kommandos gibt, in C selbst schreiben müssen.

Da C zu einer Zeit entstanden ist, als Prozessoren noch nicht so komplex und leistungsstark waren, ist die Sprache an vielen Stellen auch näher am direkten Maschinencode als das bei moderneren Sprachen der Fall ist, die mehr Abstraktion bieten und neuere Prozessorarchitekturen anders ausnutzen können. Trotzdem gehört C immer noch zu den effizientesten allgemeinen Programmiersprachen, und viele Programme, bei denen kurze Rechenzeit oder geringer Speicherverbrauch wichtig sind, sind in C geschrieben.

Erstes Anfänge der Sprache sind schon 1972 veröffentlicht worden, und 1978 haben Brian Kernighan und Dennis Ritchie den ersten Entwurf eines Standards für die Sprache vorgelegt. Der erste vollständige Sprachstandard wurde mit C 89 in den Jahren 1989/90 veröffentlicht, und es gibt immer noch Projekte, die in C entwickeln und diesen Standard benutzen. In den Jahren 1995, 1999, 2011 und 2018 wurden Ergänzungen und Verbesserungen veröffentlicht, die zum Teil auch neuere Entwicklungen und Anforderungen in der Softwareentwicklung aufnahmen. Dabei ist C 99 der wichtigste und am weitesten verbreitete Standard. Alle anderen haben nur kleinere Änderungen gebracht, die in

vielen Projekten nicht verwendet werden. Auch wir werden uns in der Vorlesung an C 99 orientieren.

2 Ein Steilkurs

In diesem Abschnitt wollen wir uns die Gesamtstruktur eines C-Programms und alle wesentlichen Strukturen des Programms anschauen. Die einzelnen Punkte werden wir dann in späteren Kapiteln vertiefen und ausbauen.

Es ist nicht notwendig und auch nicht so gedacht, dass Sie in diesem Abschnitt alle Teile des vorgestellten Programms und alle C-Sprachkonstruktionen, die in dem Programm vorkommen, vollständig verstehen. Unser Ziel ist es, die Struktur des Programms nachvollziehen, den Programmablauf zu verstehen und die wesentlichen Sprachkonstruktionen zu erkennen. Die Syntax der einzelnen Befehle, ihre Anforderungen und Variationsmöglichkeiten schauen wir uns später an.

In den ersten Tutorien werden Sie dann auf der Basis des hier vorgestellten Programms eigene Programme schreiben, indem sie den Code für die Aufgabenstellung anzupassen. Dabei können Sie am Anfang die meisten Teile aus dem schon vorhandenen übernehmen, und kompliziertere Teile und Konstruktionen, die wir noch nicht besprochen haben, stellen wir Ihnen zur Verfügung.

2.1 Die Collatzfolge

Für unser Beispiel eines C-Programms nehmen wir eine bekannte mathematische Folge, die Collatz- oder Hailstonefolge.

Zu einer vorgegebenen Startzahl $c_0 := s \in \mathbb{Z}_{\geq 0}$ ist das die Folge $c := (c_k)_{k \in \mathbb{Z}_{\geq 0}}$ mit

$$c_{k+1} := \begin{cases} c_k/2 & \text{wenn } c_k \text{ gerade ist} \\ 3c_k + 1 & \text{sonst.} \end{cases}$$

Für den Startwert $c_0 = 24$ ergibt sich die Folge

$$c = (24,\ 12,\ 6,\ 3,\ 10,\ 5,\ 16,\ 8,\ 4,\ 2,\ 1,\ 4,\ 2,\ 1,\ 4, \ldots)\,.$$

Wir sehen, dass die Folge für diesen Startwert periodisch wird und immer wieder die Teilfolge $4,\ 2,\ 1$ wiederholt. Für den Startwert $c_0 = 57$ ergibt sich

$$\begin{aligned} c = (57,\ 172,\ 86,\ 43,\ 130,\ 65,\ 196,\ 98,\ 44,\ 22,\ 11,& \\ 34,\ 17,\ 42,\ 21,\ 64,\ 32,\ 16,\ 8,\ 4,\ 2,\ 1, \ldots)\,.& \end{aligned}$$

Auch dieses Mal wird die Folge periodisch mit der gleichen Teilfolge $4,\ 2,\ 1$ wie zuvor. Wenn das passiert, brechen wir die Folge an der 1 ab und bezeichnen den kleinsten Index k, sodass $c_k = 1$ ist, als die *Länge* $l_s := k$ der Collatzfolge zum Startwert $c_0 = s$.

Version vom 10. Februar 2022

Wenn es kein solches k gibt setzen wir $l_s = \infty$. Die Länge der Folge zum Startwert 24 ist also 10, die zum Startwert 57 ist 21.

Es ist unbekannt, ob der zweite Fall, also der eines Startwerts s mit unendlicher Länge l_s, überhaupt vorkommt. Ebenso ist es unbekannt, ob es andere periodische Teilfolgen als die schon entdeckte Folge $4, 2, 1$ gibt. Es wird aber allgemein vermutet, dass das nicht der Fall ist und $l_s \in \mathbb{Z}_{\geq 0}$ für alle $s \in \mathbb{Z}_{\geq 0}$ ist (das ist die sogenannte *Collatzvermutung*).

Wir wollen ein Programm schreiben, das uns zu einem von uns vorgebenden Startwert die Länge der Folge berechnet. Wir nehmen dabei vorerst an, dass die Folge immer nach endlich vielen Schritten die 1 erreicht und periodisch wird. Das ist für nicht allzu große Startwerte bekannt[1]. Mit Stand von 2020 wissen wir tatsächlich für alle Startwerte $s \leq 2^{68}$, dass die Folge nach endlich vielen Schritten die 1 erreicht[2]. Die Größe von ganzen Zahlen, die Sie in Ihrem Programm verarbeiten können, ist beschränkt, und diese Schranke ist mit $2^64 - 1$ kleiner als 2^{68}, so dass es für unser Programm keine Folge gibt, die nicht abbricht. Sie können das Programm später, wenn Sie mehr über `C` wissen, so verbessern, dass es abbricht, wenn die Folge zu lang wird.[3]

2.2 Collatzfolge als Computerprogramm

Aktuelle handelsübliche Computer, bzw. die darin verbauten Prozessoren, sind in der Lage, einen sehr begrenzten Satz an Rechenoperationen auszuführen und damit gegebene Daten in neue Daten zu transformieren. Dieser Prozess ist deterministisch, bei gleichen Eingangsdaten erhalten Sie immer das gleiche Ergebnis[4]. Um komplexere Berechnungen zu machen, können wir eine ganze Liste solcher Befehle an den Prozessor geben.

Hier ist eine (sehr vage) umgangssprachliche Definition eines Computerprogramms.

> Ein *Computerprogramm* ist eine *deterministische* Abfolge von Steuerungsbefehlen zur Manipulation von zulässigen Daten, die nach endlicher Zeit mit einem Ergebnis abbricht.

Bevor wir unser erstes Programm in `C` schreiben, wollen wir uns ansehen, wie die Berechnung der Collatzfolgen als deterministische Abfolge von Steuerungsbefehlen aussehen könnte. Eine solche Abfolge wird oft auch als *Algorithmus*[5] bezeichnet und in *Pseudocode* notiert. Für die Collatzfolge steht eine Formulierung in Pseudocode in Algorithmus 2.1.

[1] Später zählen wir es als Fehler, das Programm hier nicht abzusichern, z.B. durch einen Zähler, der nach einer festen Zahl von Iterationen abbricht. Aber am Anfang wollen wir das Programm nicht zu kompliziert machen.

[2] David Barina. "Convergence verification of the Collatz problem". In: *The Journal of Supercomputing* (2020). DOI: 10.1007/s11227-020-03368-x.

[3] Wir werden später sehen, dass wir doch in der Lage sind, größere Zahlen zu verarbeiten, dann wird eine solche Absicherung notwendig. Allerdings wird dann auch die Rechenzeit schon sehr groß sein, und Sie müssen damit rechnen, dass die Größe der Folgenglieder gegenüber dem Anfangswert noch deutlich steigen kann, bevor sie zu 1 abfallen.

[4] Wenn Sie aktuellere Entwicklungen der Prozessor- bzw. Computerentwicklung verfolgen, werden Sie sehen, dass das eine Vereinfachung ist und inzwischen neue Varianten entwickelt wurden, die andere Eigenschaften haben. Für uns reicht aber dieses einfache Modell aus.

[5] Genauer muss ein Algorithmus etwas mehr erfüllen: Er hat eine genau festgelegte Menge möglicher Eingaben und Ausgaben, und sollte nach endlich vielen Schritten fertig werden.

Algorithmus 2.1: Collatzfolge

Eingabe : Startwert s
Ausgabe : Länge l_s der Collatzfolge zum Startwert s

```
1 c ← s
2 l ← 1
3 solange c ≠ 1 tue
4 |   wenn c gerade dann
5 |   |   c ← c/2
6 |   sonst
7 |   |   c ← 3c + 1
8 |   l ← l + 1
9 zurück l
```

Die Notation dürfte größtenteils selbsterklärend sein, aber wir sollten auf einige Teile eingehen:

(i) Am Anfang legen wir fest, welche Eingabedaten wir erwarten, und welche Ausgabe der Algorithmus haben soll. Von einem Algorithmus erwarten wir, dass er auf *allen* zulässigen Eingabedaten das entsprechend der Problemstellung *korrekte* Ergebnis nach einer endlichen Anzahl von Rechenschritten zurückgibt.

(ii) Zuweisungen auf Variablen (was sie aus der Schule in der Form $x = 4$ und in Ihrer Analysisvorlesung als $x := 4$ gesehen haben) schreibt man in Algorithmen oft mit $\leftarrow$. Das Symbol „=“ hat in der Mathematik zu viele verschiedene Bedeutungen, die vom Kontext abhängen und für einen Computer nicht immer entscheidbar sind. In der Sprache `C` wird die Zuweisung dann doch mit `=` geschrieben, dafür erfolgt z.B. der Vergleich zweiter Variablen mit `==`.

(iii) In Zeile 3 legen wir fest, dass eine Abfolge von Befehlen ausgeführt werden soll, solange eine Bedingung erfüllt ist. Die Bedingung steht zwischen **solange** und **tue** (hier $c \neq 1$). Um zu kennzeichnen, welche Steuerungsbefehle wir wiederholt ausführen wollen und welche erst nach Ende dieser Wiederholungen kommen sollen, rücken wir die Befehle ein und ziehen eine senkrechte Linie bis zum letzten Befehl in der Liste.

(iv) In der Definition der Collatzfolge treten verschiedene Fälle auf, je nachdem, ob die vorangegangene Folgenglied gerade oder ungerade war. In unserem Pseudocode schreiben wir das in der Form

wenn ... **dann** ... **sonst** ....

Die Bedingung für die Entscheidung, was gemacht wird, steht zwischen **wenn** und **dann** (hier die Entscheidung ob c gerade ist). Die Befehle, die im einen oder anderen Fall ausgeführt werden, werden wieder eingerückt.

(v) Am Ende geben wir mit **zurück** das Ergebnis (hier die Länge) zurück.

Wir haben mit unserem Pseudocode einen vollkommen deterministischen Ablaufplan für unser Ziel, die Längen von Collatzfolgen zu bestimmen, bekommen. In dieser Form kann er zwar noch nicht von einem Computer verarbeitet werden, aber wir haben einen klaren Ablaufplan bekommen, den wir jetzt in eine Sprache *übertragen* müssen, die ein Computer versteht.

An dieser Stelle kommen bei der Umsetzung dann noch einige weitere Aufgaben hinzu, die in der mathematischen Beschreibung nicht vorkommen und dafür auch keine Rolle spielen. Wir müssen uns zum Beispiel überlegen, wie wir unseren konkreten Startwert an das Programm übermitteln können, wie wir also mit dem Programm *kommunizieren* können. Ebenso soll uns das Programm über das Ergebnis informieren, wir müssen also wissen, wie wir das Ergebnis auf den Bildschirm (oder in eine Datei, oder, mit Hilfe eines Druckers, auf ein Blatt Papier) schreiben können.

Variablen, z.B. den Folgenwert c selbst, können wir in unserem Pseudocode einfach definieren. Wenn Sie den Ablauf konkret nachvollziehen, schreiben sie sich die Werte für c vielleicht auf ein Blatt Papier, oder merken sich die Zwischenwerte einfach. Computer sind hier nicht so flexibel. Wir müssen explizit festlegen, welche Variablen wir haben und wie viel Platz wir dafür im Speicher des Computers benötigen. In der konkreten Umsetzung als Programm kommen daher neben der Formulierung des eigentlichen Algorithmus noch einige technische Teile hinzu, die die Kommunikation mit dem System koordinieren.

Im letzten Kapitel haben wir schon besprochen, dass der Befehlssatz des Prozessors zwar vorgegeben, und damit im Prinzip auch die Übersetzung festgelegt ist, aber man nimmt bei der Erstellung eines auf einem Prozessor lauffähigen Programms in der Regel einen Umweg und formuliert das Programm in einer *Programmiersprache*, für die es dann ein schon vorhandenes Programm gibt, das unseren Code in Maschinenbefehle übersetzt. Damit werden wir zum einen vom konkreten Prozessor unabhängig, zum anderen ist es sehr mühsam, ein Programm direkt in Maschinensprache zu schreiben.

Wie schon mehrfach erläutert, wählen wir dafür die Sprache `C`. Wir haben schon diskutiert, dass es andere Optionen gibt. Falls wir uns später für eine andere Sprache entscheiden, können wir aber wieder unseren Pseudocode nehmen und müssen diesen nur in der konkreten Programmiersprache neu formulieren. Die grundlegenden Sprachkonstruktionen sind in allen Sprachen ähnlich, so dass der Transfer einfach sein sollte.

Eine Möglichkeit, unseren Algorithmus in `C` zu formulieren, finden Sie in Programm 2.1. Gegenüber dem Algorithmus in Pseudocode tauchen hier deutlich mehr Zeilen auf, in denen Befehle stehen, deren Sinn Ihnen vermutlich noch unklar ist, und die Sie im Moment noch nicht hätten selbst schreiben können. Das liegt, wie wir schon gesehen habe, unter anderem daran, dass wir dem Computer einige zusätzliche Informationen zur Ein- und Ausgabe sowie zur Ausführung des Programms vorgeben müssen.

Zwischen Zeile 4 und Zeile 14 erkennen Sie aber wahrscheinlich die Anweisungen aus dem Pseudocode wieder.

(i) Die Anweisung **solange** ... **tue** ist zu `while ( ... ) { ... }` geworden, die Bedingung steht in Klammern hinter `while` und statt einer vertikalen Linie schließen wir die Befehle, die wiederholt werden sollen, in geschweifte Klammern ein. Der Text, also die Anweisungen, zwischen den geschweiften Klammern, dürfen wir über mehrere Zeilen verteilen. Hier haben wir viel Freiheit, die Anweisungen so anzuordnen, wie wir wollen. Es haben sich aber einige bewährte Regeln durchgesetzt, an die wir uns halten wollen. Das besprechen wir später.

(ii) Aus **wenn** ... **dann** ... **sonst** ... ist ein `if ( ... ) { ... } else { ... }` geworden, die Bedingung steht in Klammern hinter `if` und wieder werden die

Programm 2.1: `kapitel_02/collatz.c`

```c
#include <stdio.h>

int collatz_laenge(int startwert) {
    int c = startwert;
    int laenge = 1;
    while ( c != 1 ) {
        if ( c % 2 == 0 ) {
            c = c/2;
        }  else {
            c = 3*c+1;
        }
        laenge = laenge + 1;
    }
    return laenge;
}


int main() {
    int startwert = 0;
    printf("Geben Sie eine positive ganze Zahl ein: ");
    scanf("%d",&startwert);

    int laenge = collatz_laenge(startwert);
    printf("Die Collatz-Folge mit Start %d hat Laenge %d\n",startwert,laenge);

    return 0;
}
```

alternativen Befehle in geschweifte Klammern eingefasst. Wieder haben wir die Anweisung in eine neue Zeile geschrieben.

(iii) Für die Zuweisung auf eine Variable benutzt `C` „=“, und bei der ersten Verwendung einer Variablen müssen wir sie als eine Variable für eine ganze Zahl *deklarieren*. Das geschieht durch ein vorangestelltes `int`, um anzuzeigen, dass die Variable eine ganze Zahl enthalten soll. Dabei steht `int` für *integer*, das englische Wort für eine *ganze Zahl*.[6]

(iv) Ein Test auf Gleichheit bzw. Ungleichheit geht in `C` mit `==` bzw. `!=`. Das sehen Sie in Zeile 7 und Zeile 6.

(v) In Zeile 7 müssen wir auch feststellen, ob eine Zahl gerade ist. Das überprüft man am schnellsten mit dem *Modulo-Operator* `%`, der den Rest bei ganzzahliger Division zurückgibt. Das Ergebnis von `c%2` ist also entweder 0 oder 1, je nachdem, ob `c` gerade oder ungerade ist.

(vi) Alle abgeschlossenen Programmbefehle enden mit einem Semikolon. Das ist notwendig, da `C` Leerzeichen, Tabs und Zeilenumbrüche nur zur Trennung von Variablen- und Befehlsnamen nutzt und ansonsten ignoriert. Vergessen des Semikolons ist am Anfang eine der häufigsten Fehlerquellen beim Programmieren (da man meistens jede Anweisung in eine neue Zeile schreibt und daher *sieht*, was eine

[6] Dabei hat es aber nicht ganz die umgangssprachliche Bedeutung, da ein `int` nur einen endlichen Ausschnitt der ganzen Zahlen darstellen kann, $\mathbb{Z}$ aber natürlich unendlich viele Elemente hat.

einzelne Anweisung ist. Der Compiler sieht es nicht, da er die Zeilenumbrüche nicht sieht. Andere Programmiersprachen machen das anders.)

Schauen wir uns nun auch den Rest des Programms an. Ein einfaches C-Programm wie dieses hat drei wesentliche Teile, die nacheinander im Programm stehen[7].

Am Beginn des Programms stehen die *includes*.

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

Die Sprache C selbst umfasst nur sehr wenige Befehle (wir haben sie schon in Table 1.1 gesehen). Um trotzdem einige wiederkehrende Funktionen (wie z.B. die Funktion **printf**, die später im Code auftaucht und eine Ausgabe auf den Bildschirm schreibt) nicht immer wieder neu programmieren zu müssen, oder aus Programmen, die wir früher mal geschrieben haben, in unseres kopieren müssen, gibt es sogenannte *Bibliotheken* (*libraries)* mit fertigem C-Code, die wir in unser Programm einbinden können. Das passiert an dieser Stelle, wo wir eine Standardbibliothek laden. Sie können später solche Bibliotheken auch selbst schreiben, um Code, der in vielen Programmen benutzt wird, nur einmal bereitstellen zu müssen. Wir besprechen das später im Detail, bis dahin sollten Sie einfach diese Zeile an den Anfang aller Ihrer Programme schreiben.

Der zweite wesentliche Teil eines C-Programms ist der Code

```
18 int main() {
19     int startwert = 0;
20     printf("Geben Sie eine positive ganze Zahl ein: ");
21     scanf("%d",&startwert);
22 
23     int laenge = collatz_laenge(startwert);
24     printf("Die Collatz-Folge mit Start %d hat Laenge %d\n",startwert,laenge);
25 
26     return 0;
27 }
```

in dem eine sogenannte **main**-Funktion steht. An dieser Stelle fängt das Programm mit seiner Arbeit an. Der Teil

```
1   int main() {
2     // Anweisungen
3     return ⟨Zahl⟩
4   }
```

ist dabei festgelegt (es gibt eine zweite Variante, die wir später sehen)[8]. Wir müssen am Ende mit `return` eine ganze Zahl zurückgeben, die dem System mitteilt, ob das Programm erfolgreich gelaufen ist, oder ob ein Fehler aufgetreten ist. Im Beispiel geben wir 0 zurück, was dem System mitteilt, dass das Programm erfolgreich war[9].

Zwischen `int main()` und `return` kann beliebiger C-Code stehen. Das schauen wir uns gleich an. Vorher betrachten wir aber noch den dritten Teil eines Programms, der bei uns aus den folgenden Zeilen besteht.

[7] Man kann manches vertauschen, aber aus Gründen der Lesbarkeit werden wir uns meistens an diese Reihenfolge halten.

[8] Der doppelte Schrägstrich `//` leitet einen Kommentar im Code ein, der bis zum Ende der Zeile reicht und bei der Übersetzung in Maschinencode nicht beachtet wird. In diesem Skript benutzen wir das oft, um bei den Erläuterungen einzelne unwesentliche Teile auszulassen, aber die Stelle zu markieren, wo sie stehen müssten.

[9] Haben Sie eine Idee, warum 0 für Erfolg steht, und nicht z.B. 1?

```
3  int collatz_laenge(int startwert) {
4      int c = startwert;
5      int laenge = 1;
6      while ( c != 1 ) {
7          if ( c % 2 == 0 ) {
8              c = c/2;
9          }  else {
10             c = 3*c+1;
11         }
12         laenge = laenge + 1;
13     }
14     return laenge;
15 }
```

Hier können sogenannte *Funktionen* definiert werden, die einzelne Teile unserer Berechnung zusammenfassen. In unserem Fall genau eine Funktion `collatz_laenge`. Wie schon bei `while` werden alle Befehle, die zur Funktion gehören, in geschweifte Klammern eingeschlossen. Funktionen können einen Wert zurückgeben. Wie bei Variablen müssen wir den Typ benennen. In unserem Fall geben wir eine ganze Zahl zurück, was wir mit `int` vor dem Funktionsnamen kennzeichnen. Außerdem können wir Parameter an die Funktion übergeben. Das passiert zwischen den beiden Klammern nach dem Funktionsnamen. Auch diese brauchen einen Typ und einen Namen. Im Beispiel übergeben wir eine ganze Zahl in den Parameter `startwert`. Mit diesem Parameter können wir dann innerhalb der Funktion arbeiten (wir kopieren ihn in unserem Beispiel nur einmal in die Variable `c`).

Da wir angegeben haben, dass die Funktion eine ganze Zahl zurückgibt, steht am Ende ein `return`, das festlegt, welche Zahl zurückgegeben wird[10].

Wenn Sie `collatz_laenge` mit **`main`** vergleichen, dann sehen Sie, dass die Struktur ähnlich ist. Das ist kein Zufall, denn auch **`main`** ist einfach eine Funktion, allerdings mit einem speziellen Namen und einer speziellen Bedeutung.

Unsere Funktion `collatz_laenge` wird ein einziges Mal aufgerufen, in Zeile 23. Dort wird eine Zahl an die Funktion übergeben und die Länge einer neuen Variablen zugewiesen.

Die weiteren Zeilen der Funktion **`main`** geben nur noch Text auf dem Bildschirm aus und lesen eine Zahl ein. Mit der Struktur von **`scanf`** beschäftigen wir uns später. Wenn Sie eine ganze Zahl einlesen wollen, können Sie diese Zeile einfach kopieren (aber passen sie `startwert` an, wenn Ihre Variable anders heißt, und behalten sie das & davor).

`printf` gibt eine Zeile auf dem Bildschirm aus. Im Prinzip schreibt es die Zeile zwischen den Anführungszeichen (beide oben!) auf den Bildschirm. In der Zeichenfolge können *Platzhalter* und *Steuerzeichen* vorkommen, mit denen wir den Inhalt von Variablen in die Ausgabe einfügen und die Ausgabe formatieren können. Die Zeichenfolge `\n` ist ein *Steuerzeichen* und erzeugt einen Zeilenumbruch. Es gibt viele weitere, aber für uns spielt nur noch ein weiteres (das Steuerzeichen `\0`) eine Rolle. Wir besprechen das später, vorerst brauchen wir nur den Zeilenumbruch, um unsere Ausgabe zu strukturieren.

Für jeden *Platzhalter* `%d` in der Zeile muss anschließend eine Variable kommen, die

[10] Ein `return` kann auch an anderen Stellen stehen und mehrfach auftauchen, das sehen wir später.

eine ganze Zahl enthält. Im Beispiel also in der zweiten Zeile mit `printf` zweimal. Es gibt viele weitere Platzhalter, die alle am vorangestellten `%` erkennbar sind. Einige weitere tauchen später auf[11].

Jetzt haben wir unser erstes Programm in C analysiert. Sie haben eine ganze Reihe von Programmkonstruktionen kennengelernt, und Sie sollten auf der Basis dieses Programms auch schon in der Lage sein, durch Anpassung des Codes einige einfache eigene Programme zu schreiben.

Bis jetzt haben wir uns aber nur angesehen, wie ein Programm in C auszusehen hat. Um wirklich ein Programm auf einem Computer ausführen zu können, müssen wir uns noch überlegen, wie wir den Code im Computer schreiben, speichern, und anschließend in ein ausführbares Programm umwandeln.

2.3 Programme schreiben und übersetzen

In diesem Abschnitt nehmen wir an, dass Sie auf Ihrem PC oder Laptop ein Terminal in einer Linux- oder Unix-Umgebung zur Verfügung haben. Eine Anleitung, wie Sie das unter Windows, Mac und Linux einrichten können, finden Sie in den Begleitmaterialien zur Vorlesung. Wir setzen hier auch einfache Kenntnisse im Umgang mit einem Terminal voraus. Auch hierzu finden Sie eine Anleitung in den Begleitmaterialien.

Um Programmcode zu schreiben, brauchen sie ein Programm, das unformatierten Text abspeichern kann. Texteditoren wie LibreOffice sind dafür nicht geeignet. Es gibt eine ganze Reihe einfacher Editoren für linuxähnliche Umgebungen, wie z.B. *vim*, *emacs*, oder *gedit*. Etwas umfangreichere geeignete Editoren, die auf allen Plattformen zur Verfügung stehen, sind z.B. *Atom* oder *Visual Studio Code*. In den Begleitmaterialien finden Sie weitere Informationen.

Im Beispiel ist der Programmcode zeilenweise so geschrieben, dass in jeder Zeile nur eine Anweisung steht und Anweisungen, die innerhalb unserer Schleife mehrfach ausgeführt werden oder innerhalb einer Verzweigung stehen, eingerückt sind. Diese Art der Formatierung von Programmcode entspricht der üblichen Konvention. Für C ist allerdings jede Form von Leerzeichen oder Zeilenumbruch gleich, mehrfache Leerzeichen, Zeilenwechsel oder Einrückungen mit Tab sind gleichwertig. Leerzeichen sind auch nur notwendig, um Anweisungen der Sprache C oder Variablen zu trennen[12].

Wir hätten unsere Bedingung ab Zeile Zeile 7 auch in einer der folgenden Formen

```
1 if ( c % 2 == 0 ) { c = c/2; }  else { c = 3*c+1; }
```

oder

```
1 if(c%2==0){c=c/2;} else{c=3*c+1;}
```

oder

```
1 if(c%2
2 ==0){c
3 =c/2;} else{c=3                                                    *c
```

[11] Das Prozentzeichen selbst müssen Sie deswegen als `%%` schreiben.

[12] Das ist z.B. in `python` anders, wo Zeilenumbrüche und Einrückungen tatsächlich eine semantische Bedeutung haben.

```
4  +1
5  ;}
```

schreiben können. Auch wenn Sie bisher noch wenig von C kennen, sehen Sie vielleicht schon den Vorteil der ursprünglichen Schreibweise, in der zusammengehörende Teile auch in einer gemeinsamen Zeile oder einem gemeinsamen Block strukturiert sind. C muss trotzdem erkennen können, wo eine Anweisung endet, daher schließen Anweisungen immer mit einem Semikolon ab.

In diesem Kurs sollten Sie sich direkt angewöhnen, ihre Programme sinnvoll strukturiert aufzuschreiben, also in der Regel nur eine Anweisung pro Zeile zu schreiben und zusammengehörende Teile in Schleifen oder Verzweigungen einzurücken. Bei Ihren ersten Abgaben wird Ihr Tutor Ihren Code mit Ihnen durchgehen und Hinweise zur Formatierung geben. In späteren Abgaben könnte schlecht formatierter Code auch zu Abzügen führen.

Sie sollten alle Teile Ihres Programms und wesentliche Schritte sowie Algorithmen und implizite Annahmen im Code dokumentieren. Das haben wir hier unterlassen, um den Code kurz zu halten und auf einer Seite abdrucken zu können. Mit Kommentaren könnte Ihr Programm z.B. wie in Programm 2.2 aussehen. In diesem Skript verzichten wir fast überall auf Kommentare. Einerseits würde sonst der Text deutlich länger, andererseits ist es hier auch verzichtbar, da wir hier nur Code drucken, der im umgebenden Text erklärt wird. Bei Programmen ohne begleitenden Text muss innerhalb des Codes jeder Schritt mit einem Kommentar ausreichend erklärt werden.

Speichern Sie die Datei anschließend in ein Verzeichnis auf Ihrem Computer, öffnen Sie ein Terminal und wechseln Sie in das Verzeichnis mit der Code-Datei. üblicherweise werden Dateien mit C-Code mit der Endung `.c` gespeichert. Es ist sinnvoll, sich an diese Konvention zu halten, da viele Editoren und Compiler, die mit C-Code umgehen können, solche Dateien als Dateien mit C-Code interpretieren und z.B. spezielle Menüpunkte zum Arbeiten mit C einbinden oder in Ihrem Code Schlüsselwörter der Sprache in einer anderen Farbe schreiben (wie das auch hier im Skript der Fall ist). Das erleichtert die Arbeit mit Code oft erheblich.

Um Ihren C-Code nun in ein Programm zu übersetzen, das der Computer ausführen kann, brauchen wir ein Hilfsprogramm, einen sogenannten *Compiler*, das C-Code in Maschinencode übersetzen kann. Es gibt eine ganze Reihe solcher Compiler. Unter Linux sind die üblicherweise verwendeten Compiler `gcc` und `clang`. Wir nehmen in der Vorlesung an, dass Sie `gcc` mindestens in Version `4.9` benutzen, aber die Unterschiede zu `clang` sind so gering, dass sie für die meisten Programme in der Vorlesung keine Rolle spielen. Die Version Ihres Compiler überprüfen Sie mit

```
$ gcc --version

gcc (Debian 4.9.2-10+deb8u2) 4.9.2
Copyright (C) 2014 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Andere Compiler für C ergeben natürlich auch korrekten und ausführbaren Maschinencode, aber die Bedienung des Programms und die Fehlermeldungen, falls die Übersetzung fehlschlägt, können anders aussehen.

Wir nehmen nun an, dass Sie Ihr Programm unter dem Namen `collatz.c` abgespei-

Programm 2.2: `kapitel_02/collatz.c`

```c
// Standardbibliotheken
#include <stdio.h>

/* Funktion zur Bestimmung der Laenge der Collatzfolge zu einem Startwert
   Eingabe: startwert als ganze Zahl
   Ausgabe: Laenge der Folge
   Annahmen: - startwert ist positiv, kein Check
             - Folge erreicht nach endlich vielen Schritten die 1
 */
int collatz_laenge(int startwert) {
  int c = startwert;         // Initialisierung
  int laenge = 0;
  while ( c != 1 ) {         // naechstes Folgenglied, falls 1 nicht erreicht
    if ( c % 2 == 0 ) {
      c = c/2;
    }  else {
      c = 3*c+1;
    }
    laenge = laenge + 1;     // laenge aktualisieren
  }
  return laenge;
}

// main: Einstiegspunkt in das Programm
int main() {
  int startwert = 0;         // Eingabevariable initialisieren
  printf("Geben Sie eine positive ganze Zahl ein: ");
  scanf("%d",&startwert);    // Startwert vom Terminal einlesen

  // Aufruf der Funktion zur Bestimmung der Laenge
  int laenge = collatz_laenge(startwert);

  // Ergebnis ausgeben
  printf("Die Collatz-Folge mit Start %d hat Laenge %d\n",startwert,laenge);

  // Programm ohne Fehlermeldung beenden
  return 0;
}
```

chert haben. Dann können Sie das Programm mit

```
$ gcc collatz.c -o collatz
```

übersetzen. Hierbei steht $ stellvertretend für die von Ihrem Terminal angezeigte *Eingabeaufforderung*. Oft ist das auch ein >, oder es steht noch der Name des Verzeichnisses oder das Datum davor. Wenn Sie das Terminal zum ersten Mal öffnen, wird Ihnen eine Eingabeaufforderung angezeigt, dann sehen Sie die Form bei Ihnen.

Wenn Ihr Code korrekt war, erhalten Sie im gleichen Verzeichnis eine ausführbare Datei mit dem Namen `collatz`. Sie können Ihr Programm dann z.B. die Folgenlänge für den Startwert 24 berechnen lassen.

```
$ ./collatz

Geben Sie eine positive ganze Zahl ein: 24
```

```
Die Collatz-Folge mit Start 24 hat Laenge 10
```

Hierbei ist `collatz` der Name des Programms, den wir beim übersetzen mit `gcc` gewählt haben, und das vorangestellte `./` zeigt Ihrem System an, wo es das Programm findet. Solange sie immer im gleichen Verzeichnis arbeiten, können Sie die Syntax einfach immer übernehmen.

Wenn Sie in Ihrem Code einen syntaktischen Fehler gemacht haben, wird `gcc` eine Fehlermeldung ausgeben.

```
gcc collatz.c -o collatz

collatz.c:21:27: error: expected ';' after expression
scanf("%d",&startwert)
                           ^
                             ;
1 error generated.
```

Im Beispiel ist die Meldung ziemlich eindeutig, aber gerade am Anfang werden Ihnen manche dieser Meldungen sehr unverständlich vorkommen. Wir gehen später darauf ein, im Moment ist es in diesem Fall wahrscheinlich einfacher, Sie gehen zu Ihrem Editor zurück und überprüfen Ihr Programm noch einmal Zeile für Zeile. `gcc` gibt Ihnen zwar einen Hinweis auf die Zeile (im Beispiel steht `21:27` für die Zeile 21 und das Zeichen 27 in dieser Zeile (inklusive der Leerzeichen!), aber das muss nicht immer die Stelle des eigentlichen Fehlers sein, sondern ist die Stelle, wo dem Compiler aufgefallen ist, dass etwas nicht stimmen kann. Das kann manchmal viel später im Code sein, oder sogar in einer anderen Datei. Mit etwas Übung lernt man allerdings recht schnell, anhand der Fehlermeldung trotzdem die richtige Stelle im Code zu finden.

Typische Fehler, nach denen Sie Ausschau halten sollten, bevor Sie einzelne Zeilen analysieren, sind vergessene Semikola am Ende einer Zeile oder fehlende geschweifte Klammern (oder nur eine der beiden).

Der Compiler kann Ihr Programm natürlich nur auf *syntaktische* Fehler überprüfen, und dort auch nur auf solche, die tatsächlich zu syntaktisch falschen Code führen (das sehen Sie auch in normaler Sprache, z.B. können Sie in dem Satz *Sie will er nicht* in der Mitte ein Komma setzen, beides ist syntaktisch korrekt). Der Compiler kann keine inhaltlichen Fehler finden (z.B. ungerade Folgenglieder in unserem Programm mit 4 statt 3 zu multiplizieren, oder die Klammer in Zeile 11 mit nachfolgenden Zeile 12 zu vertauschen).

Mit diesem Aufruf von `gcc` haben Sie schon den ersten *optionalen Parameter* von `gcc` kennengelernt. Mit der Option `-o` können Sie einen Namen für Ihr Programm angeben. Wenn Sie den Parameter weglassen, bekommt Ihr Programm den Namen `a.out`. Es gibt viele weitere solche Parameter zu `gcc`, von denen wir einige später noch kennenlernen werden. Optionale Parameter zu Programmen unter Linux beginnen immer mit einem „`-`“, oder „`--`“ falls der Parameter mehr als einen Buchstaben hat, der Parameter `-h` gibt oft eine Liste aller Parameter aus.

Sie wissen nun, wie Sie einfachen Programmcode in ein ausführbares Programm übersetzen können. Auch wenn wir das später noch im Detail aufgreifen werden, wollen wir uns kurz überlegen, was `gcc` gemacht hat, um das ausführbare Programm zu erzeugen. Der Compiler fasst hier mehrere Schritte zusammen. Bei komplizierteren Programmen kann es sinnvoll sein, diese Schritte einzeln auszuführen. Das, und die

einzelnen Schritte, schauen wir uns später genauer an.

(i) Im ersten Schritt löst der Compiler alle `#include`-Befehle auf, indem er Informationen über die in der Bibliothek enthaltenen Funktionen an dieser Stelle in unseren Code kopiert. Dieser Schritt ist Teil des *preprocessing*, der Vorbereitung auf die eigentliche Übersetzung in Maschinencode. Später sehen wir, dass in diesem Schritt noch weitere wichtige Aufgaben erledigt werden können.

(ii) Anschließend kann der Compiler überprüfen, ob der Code syntaktisch korrekt ist. Dazu braucht er die Bibliotheksinformationen, denn in unserem Code zur Collatzfolge ist nicht festgelegt, wie zum Beispiel der Aufruf von `printf` zur Bildschirmausgabe aussehen soll.

(iii) Nach dieser Prüfung erzeugt der Compiler sogenannten `Objektcode`. Das ist im Prinzip schon ausführbarer Code, aber es fehlt noch der Objektcode der Bibliotheken, und Informationen an das Betriebssystem, wie der Code ausgeführt werden soll.

(iv) Diese Verbindung schafft `gcc` im letzten Schritt, dem sogenannten *linking*. Hier werden alle relevanten Teile des Maschinencodes zusammengefügt und so abgespeichert, dass das System weiß, dass es sich um ein ausführbares Programm handelt.

Nach diesem letzten Schritt haben Sie ein vollständiges Programm in Maschinencode.

3 Erste Grundlagen

Nachdem wir nun unser erstes Programm erfolgreich geschafft haben und die prinzipiellen Teile und Strukturen eines Programms erkannt haben, wollen wir in diesem Abschnitt genauer auf die einzelnen Teile eingehen und ihre Syntax erklären. Wir beginnen dabei mit der Definition von Variablen, bevor wir uns die zwei wesentlichen Programmstrukturen, die *Schleifen* und die *Verzweigungen* im Detail anschauen.

3.1 Variablen

In unseren Programmen wollen wir mit Variablen arbeiten können wie in der Mathematik auch (z.B. mit der Variable x bei Polynomfunktionen $x \to x^2 - 3x + 2$ bzw. $f(x) = x^2 - 3x + 2$), um unser Programm mit unterschiedlichen Werten laufen zu lassen, oder Berechnungen, die sich nur im Wert einer Variablen unterscheiden, nicht mehrfach aufschreiben zu müssen, sondern die gleiche Abfolge von Anweisungen mehrfach ausführen zu können. Im Beispiel des Anfangskapitels haben wir das auch schon z.B. in Form unserer Variable `c` für Folgenglieder gesehen.

In der Mathematik treffen wir, wenn wir eine Variable benutzen, meistens implizit oder explizit eine Annahme darüber, aus welchem Wertebereich die Zahlen kommen dürfen, die wir für die Variable einsetzen können. Das könnten z.B. reelle Zahlen für das x einer Polynomfunktion sein, oder ganze Zahlen bei der Collatzfolge.

In `C` müssen wir explizit festlegen, aus welchem Wertebereich die Werte einer Variablen kommen dürfe.[1] Dabei gibt es vordefinierte Möglichkeiten, die wir benutzen können. `C` erwartet eine solche Information nicht nur für Zahlen, sondern z.B. auch für Variablen, die einen Buchstaben enthalten (was für den Computer am Ende doch wieder nur eine Zahl ist, der bei der Ausgabe eine besondere Bedeutung zugewiesen wird). Diese Festlegung wird in `C` (und anderen Sprachen) als *Typ* der Variablen bezeichnet. `C` kennt im wesentlichen nur drei mögliche Typen

- `int` für ganze Zahlen
- `float` und `double` für Dezimalzahlen (Fließkommazahlen)
- `char` für Zeichen.

Die Zuordnung ist in der Form auch nicht präzise. Der Rechner muss die Belegung der Variablen im Speicher des Computers ablegen. Da dieser endlich ist, steht dafür nicht beliebig viel Platz zur Verfügung und es kann nicht wirklich *jede* ganze Zahl abgespeichert werden. Es steht nur ein kleiner Ausschnitt des Zahlraums zur Verfügung, ebenso bei den Dezimalzahlen. Der genaue Bereich hängt vom Compiler und dem

[1] Es gibt auch Programmiersprachen, die diese Festlegung nicht erwarten, sondern versuchen, den Wertebereich der Variablen aus unserer Verwendung oder Belegung zu erraten

Version vom 10. Februar 2022

verwendeten Rechner ab, aber in der Regel ist es

- -2^{15} bis $2^{15}-1$ oder -2^{31} bis $2^{31}-1$ für `int`,
- -2^{7} bis $2^{7}-1$ oder $[0, 2^{8}-1]$ für `char`,
- $1.2 \cdot 10^{-38}$ bis $3.4 \cdot 10^{38}$ mit sechs gültigen Nachkommastellen für `float`.
- $2.2 \cdot 10^{-308}$ bis $1.8 \cdot 10^{308}$ mit 19 gültigen Nachkommastellen für `double`.

Beachten Sie, dass bei dem Typ `char` nicht festgelegt ist, ob der Zahlbereich symmetrisch um 0 liegt oder positiv ist. Mit vorangestellten Schlüsselwörtern können wir manche dieser Typen variieren. So können wir `int` eines der Schlüsselwörter `short`, `long` oder `long long` voranstellen. Damit ergeben sich die Bereiche

- -2^{15} bis $2^{15}-1$ für `short int`,
- -2^{31} bis $2^{31}-1$ oder -2^{63} bis $2^{63}-1$ für `long int`,
- -2^{63} bis $2^{63}-1$ für `long long int`,

Bei diesen Typen kann `int` am Ende auch weggelassen werden, also z.B. `long` statt `long int`. Mit dem Zusatz `signed` oder `unsigned` können sie den Zahlbereich der `int`-Typen und von `char` symmetrisch um 0 machen oder in den positiven Bereich verschieben. So deckt `unsigned short` den Bereich von 0 bis $2^{16}-1$ ab, und `signed char` den Bereich von -128 bis 127.

Auch dem Typ `double` kann ein `long` vorangestellt werden, es hängt aber von Prozessor und Compiler ab, ob der Typ einen über `double` hinausgehenden Bereich abdeckt. Die Dezimalzahltypen gibt es nicht mit `unsigned`.

Nun kennen wir einige mögliche Typen. Eine Variable mit dem Namen `x` vom Typ `int` definieren wir nun durch

```
int x;
```

oder, wenn wir ihr gleich den Wert 4 geben wollen, mit

```
int x = 4;
```

Den Typ einer Variablen dürfen und müssen wir nur genau einmal festlegen, und zwar vor ihrer ersten Verwendung. Ein Variablenname darf auch nur genau einmal vorkommen.[2] Nach ihrer Deklaration können wir immer wieder auf die Variable zugreifen und auch ihren Wert verändern, solange sie tatsächlich deklariert ist. Spätestens mit dem Ende des Programms werden alle Deklarationen gelöscht und der Speicherbereich wieder freigegeben. Das kann allerdings auch früher erfolgen. Wir haben unsere Variablen in unserem Beispiel Programm 2.1 innerhalb der Funktion `main` oder `collatz_laenge` deklariert. Ihre Deklaration steht damit innerhalb des durch die geschweiften Klammern `{...}` als zu der Funktion gehörenden Codeblocks, und sie sind tatsächlich auch nur dort deklariert. Am Ende der Funktion (mit dem Aufruf von `return`) wird die Deklaration gelöscht. Allgemeiner gilt diese Regel für jeden von `{...}` eingeschlossenen Teil des Codes, und eine Variable ist nur bis zur nächsten schließenden Klammer gültig[3].

Die Wertebereiche der einzelnen Typen sind nicht vollständig festgelegt und können sich je nach Prozessor unterscheiden. Um die konkreten Werte auf dem eigenen Computer oder Laptop herauszufinden, können wir in einer speziellen Bibliothek nachsehen,

[2] genau genommen einmal in einem Block oder Modul, aber das betrachten wir später, vorerst bleiben wir dabei, dass jeder Name nur einmal vorkommt

[3] Wir vereinfachen hier etwas, bevor wir das später noch einmal aufgreifen, aber für die nächste Zeit, und für die Beispiele in diesem Kapitel ist das so ausreichend.

die, wie die schon vorgestellte Bibliothek `stdio.h`, mit jeder Installation von `gcc` oder `clang` mitkommen.

Wir brauchen hierfür die Bibliotheken `limits.h` für die Grenzen der Ganzzahltypen und `float.h` für die Dezimaltypen, in denen die korrekten Werte hinterlegt sind. Programm 3.1 zeigt einige Konstanten, die in den Bibliotheken definiert sind.

Programm 3.1: `programs/kapitel_03/show_limits.c`

```c
#include <stdio.h>
#include <limits.h>
#include <float.h>

int main() {
  unsigned int x = UINT_MAX; // groesster Wert fuer unsigend int
  long l = LONG_MIN; // kleinester Wert fuer long
  float f = FLT_MAX;         // groesster Wert fuer float
  float eps = FLT_EPSILON;   // Abstand zwischen 0 und naechstem float

  printf("Die groesste darstellbare Zahl fuer unsigned int ist %u\n", x);
  printf("Die kleinste darstellbare Zahl fuer long int ist %ld\n", l);
  printf("Die groesste darstellbare Zahl fuer float ist %e\n", f);
  printf("Die kleinste positive Dezimalzahl ist %e\n", eps);
  return 0;
}
```

Die Ausgabe könnte z.B so aussehen:

```
Die groesste darstellbare Zahl fuer unsigned int ist 4294967295
Die kleinste darstellbare Zahl fuer long int ist -9223372036854775808
Die groesste darstellbare Zahl fuer float ist 3.402823e+38
Die kleinste positive Dezimalzahl ist 1.192093e-07
```

Im Allgemeinen ist es aber sinnvoll, bei der Programmierung immer nur vom über alle Varianten hinweg kleinsten Bereich auszugehen. Andernfalls kann Ihr Programm nicht mehr ohne Anpassungen auf anderen Computern übersetzt und ausgeführt werden, oder die Ausführung führt zu oft schwer nachzuvollziehenden Fehlern.

Die ganzzahligen Typen sind ringförmig angeordnet, wenn wir zur die größten darstellbaren Zahl eins addieren, erhalten wir die kleinste darstellbare Zahl. Wir sehen das z.B. mit Programm 3.2.

Programm 3.2: `programs/kapitel_03/zahlbereiche_int.c`

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

int main () {

    int au = INT_MAX;
    int al = INT_MIN;
    long bu = LONG_MAX;
    long bl = LONG_MIN;

    printf("die groesste ganze Zahl vom Typ int ist %d, und die kleinste %d\n", au, al);

    printf("die naechstkleinere Zahl ist %d\n", al-1);
    printf("die groesste ganze Zahl vom Typ long ist %ld, und die kleinste %ld\n", bu,
    bl);
```

```
14      printf("die naechstkleinere Zahl ist %ld\n", bl-1);
15
16      return 0;
17  }
```

Das Programm gibt

```
die groesste ganze Zahl vom Typ int ist 2147483647, und die kleinste -2147483648
die naechstkleinere Zahl ist 2147483647
die groesste ganze Zahl vom Typ long ist 9223372036854775807, und die kleinste
    -9223372036854775808
die naechstkleinere Zahl ist 9223372036854775807
```

aus. Diese Eigenschaft liegt an der Art, in der ganze Zahlen meistens in Computern gespeichert werden. Alle Zahlen werden binär, also zur Basis 2 gespeichert. In dieser Speicherung gibt die erste Ziffer das Vorzeichen an, Zahlen die mit 0 starten, sind größer oder gleich Null, Zahlen, die mit 1 starten sind negativ. Wie bei den Dezimalzahlen stehen die Ziffern zu höheren Zweierpotenzen vorne[4]. Die Binärzahl 01010011[5] steht also für die Zahl 83. Um zur Darstellung der entsprechenden negativen Zahl zu kommen, wird nun an allen Stellen 0 mit 1 oder umgekehrt vertauscht, und dann 1 addiert. Die Zahl -83 hat also die Darstellung 10101101. Die gleiche Vorschrift führt dann auch wieder zur 83 zurück. Wenn wir nur die 0 an der ersten Stelle durch eine 1 ersetzt hätten, hätte die Zahl 0 keine eindeutige Darstellung.

Die größte ganze Zahl, die mit einem Byte darstellbar ist, ist daher 01111111 oder 255. -255 ist dann 10000001. Wenn wir 1 addieren, erhalten wir 10000000, eine negative Zahl. Wenn wir darauf erneut 1 addieren, kommen wir zu 10000001, was wir schon als -255 identifiziert haben. Die kleinste darstellbare Zahl ist daher -256. Diese Art der Darstellung ganzer Zahlen heißt *Two's Complement*.

Bei den Typen `float` und `double` können die Abstände zwischen aufeinanderfolgenden in `C` darstellbaren Zahlen sehr groß werden. Der Typ `float` kann zwar Zahlen bis zu $3.4 \cdot 10^{38}$ darstellen, aber da nur die ersten sechs Ziffern der Zahl gespeichert werden, haben aufeinanderfolgende darstellbare Zahlen einen Abstand von bis zu 10^{32}!

Wie wir schon mit dem vorangegangenen Programm gesehen haben, fehlen aber auch alle Zahlen zwischen 0 und `FLT_EPSILON`. Wir dürfen aus mathematischer Sicht also keine allzu hohen Genauigkeitsansprüche an die Ergebnisse stellen, die mit Variablen vom Typ `float` oder `double` berechnet wurden, oder wir müssen eine gute Abschätzung des möglichen Fehlers machen, um die Qualität und Aussagekraft des Ergebnisses bewerten zu können.

Es ist auch nicht sinnvoll, zwei Zahlen dieser Typen auf *Gleichheit* oder *Ungleichheit* zu testen. Zwei Rechenwege, die bei mathematisch exakter Berechnungen zum gleichen Ergebnis führen, können hier durch Rundung in Zwischenergebnissen ungleiche Resultate erzielen, während zwei Berechnungen, die zu unterschiedlichen Ergebnissen führen, gleich aussehen können, wenn ihre ersten sechs Ziffern übereinstimmen (wie gesehen, könnten sie um 10^{32} auseinanderliegen!). Programm 3.3 verdeutlicht das mit einem Beispiel.

[4] Das ist technisch nicht ganz richtig, die Zahlen werden in Blöcken zu je acht Ziffern gespeichert. Innerhalb eines Blocks fällt die Zweierpotenz von links nach rechts, aber die Blöcke kommen in umgekehrter Reihenfolge. Das hat technische Gründe.

[5] die Zahl der Ziffern in aktuellen Prozessoren ist immer ein Vielfaches von 8, also von einem Byte, jede einzelne Ziffer ist dann ein Bit in diesem Byte

Programm 3.3: `programs/kapitel_03/genauigkeit.c`

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

void vergleiche(float f1, float f2 ) {
    if ( f1 == f2 ) {
        printf("Die eingegebenen Zahlen sind gleich\n");
    }  else {
        printf("Die eingegebenen Zahlen sind %f und %f\n",f1,f2);
    }
}

int main () {

    long max = LONG_MAX;

    float f1 = 1.03;
    float f2 = 1.14;

    vergleiche(f1,f2);
    vergleiche(f1*max,f1*max-1000);

    return 0;
}
```

Das Programm gibt

```
Die eingegebenen Zahlen sind 1.030000 und 1.140000
Die eingegebenen Zahlen sind gleich
```

aus.

3.1.1 Listen

Von den Grundtypen können wir einen weiteren Datentyp ableiten, der eine Liste von Werten des gleichen Typs speichern kann. Dazu setzen wir die Länge unserer Liste (die Anzahl der Elemente, die sie aufnehmen soll), in eckigen Klammern hinter den Variablennamen.

```c
int liste[20];
```

Die Länge der Liste kann nach der Definition nicht mehr verändert werden, man muss sich also von Anfang an überlegen, wie viele Einträge die Liste maximal haben wird. Wenn das nicht klar ist, muss man überschätzen und eine größere Liste definieren und sich parallel merken, bis zu welchem Eintrag der Inhalt der Liste relevant ist. Wie bei den Grundtypen kann die Liste auch bei der Definition schon mit Werten initialisiert werden.

```c
int liste[5] = {1,3,5,7,9} ;
```

Dabei muss nicht jeder Eintrag initialisiert werden,

```c
int liste[10] = {1,3,5,7,9} ;
```

legt nur die ersten fünf Einträge fest. Auf diese Weise können wir aber nur am Ende Werte offen lassen, es ist nicht möglich, zwischendrin eine Lücke zu lassen.

Auf die einzelnen Elemente der Liste kann man anschließend ebenfalls mit eckigen Klammern zugreifen. Dabei sind die Elemente der Liste von 0 an aufsteigend numeriert, und wir können natürlich auch neue Werte auf einzelne Stellen in der Liste zuweisen.

```
int liste[5] = { 1,3,5,7,9} ;
int prod = liste[2] * liste[3]; // prod ist 35
liste[0] = 2;
```

Beim Zugriff auf die Liste gibt es keine Kontrolle, ob die Liste so lang ist, dass das Listenelement überhaupt existiert.

```
int liste[5] = {2,4,6,8,10} ;
int wert = liste[10];   // Ergebnis undefiniert, Fehler wird nicht erkannt
```

Es ist also unsere Aufgabe als Programmierer sicherzustellen, dass alle Listenzugriffe nur auf Elemente erfolgen, die tatsächlich definiert sind. Solche Indexfehler sind eine recht häufige, und oft schwer zu entdeckende Fehlerquelle, insbesondere, wenn wir nicht auf ein von vornherein feststehendes Element zugreifen, sondern den Index in einer Variablen haben.

```
int liste[5] = {2,4,6,8,10} ;
int i = 0;
while ( i <= 5 ) { // Fehler: wir greifen als letztes auf liste[5] zu,
                   // das ist nicht definiert
  printf("An der Stelle %d steht %d\n",i,liste[i]);
}
```

Wenn wir die Liste direkt mit Werten versehen, können wir auch den Compiler zählen lassen und brauchen die Länge der Liste nicht angeben.

```
int liste[] = {2,4,6,8,10} ;
```

Die eckigen Klammern sind aber trotzdem notwendig. Wir können natürlich auch die anderen Typen in eine Liste packen.

```
float fliste[3] = {1.5, 2.5, 3.5} ;
char zeichen[10] = {'H', 'a', 'l', 'l', 'o' }
char zeichenkette[100] = "Eine Zeichenkette";
```

Wir wir an dem Beispiel sehen, gibt es bei Zeichenketten auch die Möglichkeit, alle Einträge der Liste direkt hintereinander und insgesamt in " eingeschlossen zu übergeben. Diese Vereinfachung der Syntax kommt dem üblichen Gebrauch von Listen vom Typ `char` zur Speicherung von Text entgegen. In dieser Form verhalten sie sich aber etwas anders als in der Form davor. Das schauen wir uns im nächsten Abschnitt genauer an.

Listen haben eine feste Länge, die wir nicht mehr verändern können. Eine übliche Vorgehensweise um dieses Problem zu umgehen ist es, die Größe von Listen so zu wählen, dass alle vorgesehenen Fälle darin Platz haben. Wenn wir z.B. statistische Daten verarbeiten, die aus bis zu 100 Kategorien bestehen können, legen wir eine Liste der Länge 100 an und merken uns die tatsächliche Anzahl der Kategorien in einer zweiten Variablen. Es ist allerdings oft ungünstig, eine solche Wahl einer maximalen Größe explizit im Code festzuhalten. Falls wir die Daten anpassen und nun mehr Kategorien vorsehen, müssen wir alle Stellen im Code finden, wo wir die maximale Länge von 100 benutzt haben.

Für solche Fälle ist es üblich, sogenannte *Präprozessormakros* einzusetzen. Damit können wir einmal am Anfang des Codes eine Variable definieren, und der Präprozessor,

der auch die Bibliotheksreferenzen, die wir mit `include` am Anfang gemacht haben, ersetzt, ersetzt diese Variable an allen Stellen, an denen sie vorkommt, durch den anfangs definierten Wert. Der `C`-Compiler sieht dann nur die Datei mit den ersetzten Werten, nicht mehr die Variable. Konkret definieren wir eine solche Variable durch

```
#define MAX_LAENGE 100
```

Beachten Sie, dass hier kein Semikolon ans Ende kommt, da es keine Anweisung der Sprache `C` ist. Wir können damit unsere Liste durch

```
int liste[MAX_LAENGE];
```

definieren, oder diese Variable auch an anderen Stellen einsetzen, z.B.

```
int i = 0;
int sum = 0;
while ( i < MAX_LAENGE ) {
  sum = sum + liste[i];
  i = i+1;
}
```

3.1.2 Zeichenketten

Für die Ein- und Ausgabe von Daten müssen wir immer wieder mit Zeichenketten (*Strings*) umgehen, und diese abspeichern, zusammensetzen oder modifizieren. Daher können wir Zeichenketten auch direkt, eingeschlossen in obenstehende Anführungszeichen, angeben. Wir haben das im Beispiel schon bei den Zeichenketten, die an **`printf`** und **`scanf`** übergeben wurden, gesehen. Wir können eine solche Zeichenkette direkt in einer Liste von `char` speichern und wieder ausgeben.

```
  char zeichenkette[100] = "eine Zeichenkette";
  printf("Wir betrachten %s in diesem Abschnitt\n", zeichenkette);
```

Dabei sehen wir einen neuen Platzhalter `%s` für Zeichenketten. An der Stelle scheinen wir allerdings noch ein kleines Problem zu haben. Die Bildschirmausgabe ist wie erwartet

```
Wir betrachten eine Zeichenkette in diesem Abschnitt.
```

Unsere Liste hat aber Platz für 100 Zeichen. Um zu wissen, wie lang die Zeichenkette wirklich ist, kommt ein neues Steuerzeichen `\0` ins Spiel. `C` fügt es automatisch am Ende als Markierung an und weiß auf diese Weise, wann die relevanten Zeichen enden. Es wird bei der Bildschirmausgabe nicht mit ausgegeben. Allerdings braucht `C` ausreichend Platz für dieses zusätzliche Zeichen in der Liste, weshalb eine Liste von Zeichen, die auf diese Weise verwendet werden soll, immer mindestens Platz für ein weiteres Zeichen nach den sichtbaren haben muss. Für das Wort *Darmstadt* brauchen wir also eine Liste der Länge 10 (oder länger).

In den Zeichenketten können wir auch Steuerzeichen unterbringen. Das, neben `\0`, einzige für uns relevante Steuerzeichen ist `\n`, das einen Zeilenumbruch macht.

```
  char zeichenkette[100] = "Ein Wiesel\nsass auf einem Kiesel\n";
  printf("%s", zeichenkette);
```

schreibt

```
Ein Wiesel
sass auf einem Kiesel
```

auf den Bildschirm.

Intern ist `char` eigentlich, wie wir schon gesehen haben, ein Zahltyp. Er wird jedoch, wenn er in einem Kontext, in dem Zeichen erwartet werden, als Zeichen interpretiert. Die Übersetzung erfolgt dabei über die ASCII-Tabelle, die Buchstaben, Zahlen, Sonderzeichen und einige Steuerzeichen einen Wert zwischen 0 und 127 zuweist. Dabei entspricht z.B. A der 65 und Z der 90, die weiteren Großbuchstaben liegen dazwischen. Die Kleinbuchstaben sind der Bereich von 97 bis 122, und die Zahlen von 0 bis 9 entsprechen dem Bereich 48 bis 57. Dazwischen finden sich z.B. Satzzeichen und Klammern. Unter 32 finden sich Steuerzeichen.

Eine häufige Aufgabe ist die Manipulation solcher Zeichenketten, also Kopieren, Aneinanderhängen, Vergleichen, oder die Länge bestimmen. Dafür stellt die Bibliothek `string.h` einige Funktionen bereit. Programm 3.4 stellt einige davon vor.

Programm 3.4: `kapitel_03/zeichenketten_funktionen.c`

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

int main() {
  char d[10] = "Darmstadt";
  char f[30] = "Frankfurt";
  char s[12] = " liegt bei ";
  char t[30];

  printf("%s hat %lu Buchstaben\n",d,strlen(d));

  strcat(t,d);
  strcat(t,s);
  strcat(t,f);
  printf("%s\n", t);

  printf("%s", d);
  if ( strcmp(d,f) <= 0 ) {
    printf(" kommt vor ");
  }  else {
    printf(" kommt nach ");
  }
  printf("%s\n", f);

  strcpy(f,d); if ( strcmp(f,d) == 0 ) {
    printf("f und d sind gleich\n");
  }

  return 0;
}
```

Wir gehen sie hier einmal durch.

- `strlen` bestimmt die Länge der Zeichenkette (also die Zahl der Zeichen bis zum ersten `\0`).
- `strcmp` vergleicht zwei Zeichenketten lexikographisch (also in der gleichen Ordnung, nach dem Lexika sortiert werden). Dabei wird eine ganze Zahl kleiner als

Null (meistens -1) zurückgegeben, wenn die erste vor der zweiten Zeichenkette kommt, eine Zahl größer als Null (meistens 1), wenn es umgekehrt ist, und 0 wenn sie gleich sind.

- ▷ `strcat` braucht zwei Zeichenketten als Argumente und hängt die zweite an die erste an. Dabei muss die Liste `char[]` der ersten Zeichenkette lange genug sein um beide Zeichenketten und das Steuerzeichen `\0` aufzunehmen.

3.2 Verzweigungen

Je nach Eingabe oder Zwischenergebnissen kann man sein Programm unterschiedlich fortfahren lassen. Wir haben das im vorangegangen Kapitel schon einmal anhand der Collatz-Folgen studiert.

Wir haben zwei Möglichkeiten, den Programmablauf über eine Verzweigungsbedingung zu steuern. Zum einen den Befehl `if(Bedingung)`, dessen nachfolgender Codeblock nur ausgeführt wird, wenn die in den Klammern nach `if` stehende Bedingung erfüllt ist. Zum anderen haben wir das Kommando `switch(value)`, das abhängig vom Wert von `value` einen Codeblock auswählt.

3.2.1 Verzweigungen mit `if`

Die häufigste Form der Verzweigung, die in `C` verwendet wird, ist die Aufteilung des Programmlaufs mit `if`. Die Grundform ist

```
if(Bedingung) { ... }
if(Bedingung) { ... }  else { ... }
```

In der zweiten Form kann nach `else` eine Alternative angegeben werden, die ausgeführt wird, wenn die Bedingung nicht erfüllt ist. Sämtlicher Code, der von der Bedingung gesteuert werden soll, muss in geschweiften Klammern eingeschlossen sein. Die Bedingung selbst muss in runden Klammern stehen.

Welche Form die Bedingung haben kann, sehen wir uns in einem späteren Abschnitt ausführlich an. Bis dahin beschränken wir uns auf einfache alphanumerische Tests, die Gleichheit oder eine Ordnungsrelation überprüfen. Damit könnten Bedingungen z.B. die Form

```
1  x == 4       // Gleichheit, die Variable x muss 4 enthalten
2  x < 7        // Relation, x muss eine Zahl enthalten, die kleiner 7 ist
3  x != y       // Ungleichheit, die Variablen x und y muessen verschieden sein
4  x % 3 == 0   // x muss ohne Rest durch 3 teilbar sein
```

haben. Ein Beispiel einer Verzweigung mit Alternative ist dann

```
1 if ( c % 2 == 0 ) {
2   c = c/2;
3 }  else {
4   c = 3*c+1;
5 }
```

Weitere einfache Operatoren für Vergleiche sind <= und >=. Sie können natürlich auch zwei Variablen vergleichen, jedoch sollten dann beide vom gleichen Typ sein (also beide

`int` oder `float`). Der Vergleich von `float`-Variablen ist allerdings problematisch und sollte vermieden werden. Wir diskutieren später, warum das so ist.

Ebenso können einfache Rechenoperationen in der Bedingung vorkommen, z.B. könnte man die Summe der Werte zweier Variablen vom Typ `int` durch

```
x + y <= 5
```

beschränken.

Es ist auch erlaubt, mehrfach zu verzweigen, indem nach dem `else` statt einer Folge von Kommandos ein weiteres `if` angefügt wird.

Programm 3.5: `kapitel_03/mehrfache_verzweigung.c`

```
1  unsigned int x = 7;
2  if ( x == 0 ) {
3    printf("x ist 1\n");
4  }  else if ( x == 1 ) {
5    printf("x ist 1\n");
6  }  else if ( x == 1 ) {
7    printf("x ist 1\n");
8  }  else if ( x == 2 ) {
9    printf("x ist 2\n");
10 }  else if ( x == 3 ) {
11   printf("x ist 3\n");
12 }  else {
13   printf("x ist groesser als 3\n");
14 }
```

Hierbei ist das letzte `else` wieder optional.

3.2.2 Verzweigungen mit `switch`

Es gibt eine weitere Möglichkeit, im C-Code zu verzweigen. Die `switch`-Anweisung ermöglicht es, in Abhängigkeit des Werts einer Variablen unterschiedliche Fälle auszuführen. Dabei darf die Variable nur einen Typ haben, der als ganze Zahl interpretiert werden kann, also alle Varianten von `int` oder `char`. Die Variable wird `switch` in der Form

```
switch(⟨Variable⟩) {  ... }
```

vorgegeben. Innerhalb des Blocks der geschweiften Klammern können dann einer oder mehrere Abschnitte der Form

```
case ⟨Wert⟩:
  // Anweisungen
  break;
```

stehen, und als letztes ggf. noch

```
default:
  // Anweisungen
  break;
```

kommen. Der letzte Befehl jedes Falls muss `break` sein, damit die Bearbeitung an dieser Stelle abbricht. Andernfalls werden nach dem ersten Fall, der auf die Variable zutrifft, auch alle nachfolgenden Fälle ausgeführt. Der mit `default` eingeleitete Fall ist optional,

und wird nur ausgeführt, wenn keiner der vorangegangenen Fälle zutrifft. Hier ist die gleiche Fallunterscheidung, die wir oben mit einer Folge von `if ... else` formuliert haben, mit `switch ... case` realisiert.

Programm 3.5: `kapitel_03/mehrfache_verzweigung.c`

```
1   switch ( x ) {
2   case 0:
3     printf("x ist 1\n");
4     break;
5   case 1:
6     printf("x ist 1\n");
7     break;
8   case 2:
9     printf("x ist 2\n");
10    break;
11  case 3:
12    printf("x ist 3\n");
13    break;
14  default:
15    printf("x ist groesser als 3\n");
16    break;
17  }
```

Anders als bei einer Verzweigung mit `if` können wir in einer `switch`-Anweisung nur auf Gleicheit testen.

3.3 Schleifen

Mit *Schleifenstrukturen* können wir eine Abfolge von Programmbefehlen mehrfach ausführen. Dabei haben wir die Möglichkeit, die Werte von Variablen in jedem Durchlauf zu verändern. Neben Verzweigungen ist das eine der häufigsten Programmkonstruktionen, die in der Programmierung auftaucht.

Es gibt in `C` drei verschiedene Möglichkeiten, eine solche Schleife zu programmieren, die sich in ihrer Syntax an umgangssprachlichen Formulierungen von Wiederholungen orientieren. Insbesondere können wir unterscheiden, ob wir eine bestimmte Abfolge von Befehlen

- ▷ für eine feste Anzahl von Iterationen ausführen wollen, oder ob wir die Befehle
- ▷ ausführen wollen, bis eine vorgegebene Bedingung erfüllt ist.

Alle Optionen können prinzipiell durch geeignete Umformulierung gegeneinander ausgetauscht werden, mit der richtigen Wahl erleichtern wir aber oft das Lesen und Verstehen des Codes.

3.3.1 `while`

Die `while`-Schleife haben wir schon im Einführungskapitel bei der Collatz-Folge kennengelernt. Ihre Grundform ist

```
while(Bedingung) { ... }
```

und in unserem Beispiel haben wir die Folgenvorschrift innerhalb des äußeren Blocks geschweifter Klammern so lange iteriert, bis wir bei der 1 angekommen waren.

Programm 2.1: kapitel_02/collatz.c

```
1     while ( c != 1 ) {
2         if ( c % 2 == 0 ) {
3             c = c/2;
4         }  else {
5             c = 3*c+1;
6         }
7         laenge = laenge + 1;
8     }
```

Der Test der Bedingung erfolgt dabei immer am Anfang jedes Durchlaufs. Wenn während der Bearbeitung der Abfolge von Befehlen also die Variable `c` auf 1 gesetzt wird, wird der Block trotzdem bis zum Ende ausgeführt.

Wir sind an dieser Stelle als Programmierer dafür verantwortlich, dass die Bedingung auch tatsächlich irgendwann *nicht mehr erfüllt* ist, und zwar in allen Situationen und mit allen Parametern, mit denen das Programm aufgerufen wird. C bewahrt uns nicht davor, ein Programm zu schreiben, das nicht abbricht, sondern den Block von Befehlen beliebig oft wiederholt.

3.3.2 `for`

Wenn wir stattdessen eine feste Anzahl von Wiederholungen für unsere Abfolge von Befehlen vorgeben (die durchaus von einer vor dem Start der Schleife definierten Variablen abhängen kann), oder wenn wir während der Schleife wissen möchten, in welcher Wiederholung wir gerade sind, dann eignet sich die Konstruktion der Schleife mit `for` in der Regel besser dafür. Hier ist die Grundform

```
for(Initialisierung; Bedingung; Fortsetzung) { ...}
```

Wobei *Initialisierung*, *Bedingung* und *Fortsetzung* eine Abfolge von Befehlen sein können. Dabei wird

- die Anfangsinitialisierung genau einmal am Start der Schleife ausgeführt,
- die Fortsetzung am Ende jedes Durchlaufs ausgeführt, und
- die Bedingung vor jedem Durchlauf, aber nach der Initialisierung, getestet und die Schleife abgebrochen, wenn die Bedingung nicht erfüllt ist.

Die drei Teile sind durch ein Semikolon getrennt. Prinzipiell können alle drei Teile leer sein oder kompliziertere Befehlsfolgen enthalten. Meistens sind `for`-Schleifen aber von der folgenden eher einfachen Form

```
1 for ( int i = 0; i < 10; i = i+1) {
2   // Zu wiederholender Code, kann die Variable i benutzen
3 }
```

Hier wird am Anfang eine ganzzahlige Variable mit dem Namen `i` auf 0 initialisiert, vor jedem weiteren Durchlauf um 1 erhöht und die Schleife abgebrochen wird, wenn `i` einen Wert von 10 oder größer hat. Die Schleife wird also (falls `i` nicht in der Schleife verändert wird), zehnmal durchlaufen.

Wie schon erwähnt, sind die Schleifenkonstruktionen austauschbar. Unser Beispiel einer `while`-Schleife hätten wir auch als

```
for ( ; c != 1; ) {
  if ( c % 2 == 0 ) {
    c = c/2;
  }  else {
    c = 3*c+1;
  }
  laenge = laenge + 1;
}
```

schreiben können. Hier wird nichts initialisiert, weder am Anfang noch vor den weiteren Durchläufen. Wir haben nur eine Abbruchbedingung. Die beiden Semikola müssen wir natürlich trotzdem machen, um dem Compiler anzuzeigen, welche der drei Teile leer ist.

3.3.3 `do while`

Die letzte Möglichkeit, eine Schleife zu schreiben ist eine einfache Variation der ersten. in der `do-while`-Schleife wird nur der Test der Bedingung ans Ende des Durchlaufs verlegt statt am Anfang überprüft zu werden. Ansonsten verhält sie sich genauso wie eine `while`-Schleife. Die Schleife wird mindestens einmal durchlaufen, und so lange wiederholt, bis die Bedingung nicht mehr erfüllt ist.

```
  int i = 0;
  do {
    i = i+1;
  }  while ( i < 10 );
```

Auch diese Schleifenvariante lässt sich durch eine der anderen ausdrücken.

Wir schauen uns diese Variationsmöglichkeiten bei einem konkreten Problem an. Gesucht ist zu einer vorgegebenen Zahl die Anzahl aller geordneten Paare nichtnegativer Zahlen (a, b), die $a^2 + b^2 \leq n$ erfüllen. Diese Anzahl lässt sich leicht mit zwei verschachtelten Schleifen lösen. Wenn wir dafür Schleifen mit `for` benutzen wollen, könnte das wei folgt aussehen.

Programm 3.6: `kapitel_03/schleifenvariationen.c`

```
  counter=0;
  for( x=0; x*x <= n; x=x+1 ) {
    for( y=0; x*x + y*y <= n; y=y+1 ) {
      counter = counter+1;
    }
    y=0;
  }
```

Bei der Verwendung einer Schleife mit `while` sähe die gleiche Schleife wie folgt aus. Hier müssen wir natürlich vorher daran denken, die Variablen `x` und `y` anfangs auf 0 zu setzen.

Programm 3.6: `kapitel_03/schleifenvariationen.c`

```
  counter=0;
  x=0;
```

```
  y=0;
  while ( x*x <= n ) {
    while ( x*x + y*y <= n ) {
      y=y+1;
      counter=counter+1;
    }
    y=0;
    x=x+1;
  }
```

Einer Schleife mit `do ... while()` könnte so aussehen, wobei wieder `x` und `y` zu Beginn 0 sein sollten.

Programm 3.6: kapitel_03/schleifenvariationen.c

```
  counter = 0;
  x = 0;
  y = 0;
  do {
    do {
      counter=counter+1;
      y=y+1;
    }  while ( x*x + y*y <=n );
    y=0;
    x=x+1;
  }  while ( x*x <= n);
```

Natürlich können wir die Schleifenvarianten auch mischen, z.B. eine `while`-Schleife mit einer `for`-Schleife.

Programm 3.6: kapitel_03/schleifenvariationen.c

```
  counter=0;
  x=0;
  while ( x*x <= n ) {
    for ( y = 0; x*x + y*y <= n; y = y+1 ) {
      counter=counter+1;
    }
    x=x+1;
  }
```

Alle anderen acht Kombinationen lassen sich genauso benutzen.

3.3.4 Schleifenmodifikationen

Manchmal ist es wünschenswert, eine Schleife (in der Regel abhängig von einer Bedingung) während des Durchlaufs abzubrechen und entweder mit dem nächsten Durchlauf zu beginnen, oder die Schleife komplett abzubrechen und mit dem Code nach der Schleife fortzufahren. Dafür gibt es die beiden Befehle `continue` und `break`. Beide brechen den Durchlauf ab. `continue` geht an den Anfang der Schleife, `break` springt hinter die Schleife.

```
  for ( int i = 0; i < 10; i = i+1 ) {
    if ( i == 5 ) {
      continue;
    }
    printf("%d ",i);
```

```
  }
  printf("\n");
```

gibt daher

```
0 1 2 3 4 6 7 8 9
```

aus, und

```
  for ( int i = 0; i < 10; i = i+1 ) {
    if ( i > 5 ) {
      break
    }
    printf("%d ",i);
  }
  printf("\n");
```

gibt

```
0 1 2 3 5
```

aus. Die zweite Schleife lässt sich natürlich schöner schreiben, indem man schon in der Schleifenbedingung statt `i < 10` direkt `i < 5` testet. In der Form ist die Schleife deutlich besser zu lesen.

In anderen Fällen ist `break` jedoch durchaus nützlich. Wir können z.B. eine Schleife abbrechen, wenn ein Fehler auftritt. Falls der Abbruch von der Rückgabe einer Funktion abhängt, die in der Schleife aufgerufen wird, und deren Eingabeparameter erst in der Schleife zusammengestellt werden, ist `break` ebenfalls eine einfache Möglichkeit, die Schleife zu beenden.

3.4 Funktionen

Wir haben schon zwei Varianten von *Funktionen* in unserem Beispielprogramm gesehen. Zum einen haben wir die Funktion `printf` benutzt, die uns von der Sprache über die Bibliothek `stdio.h` zur Verfügung gestellt worden ist, benutzt. Zum anderen haben wir uns mit `collatz_laenge` eine eigene Funktion definiert:

Programm 2.1: `kapitel_02/collatz.c`

```
int collatz_laenge(int startwert) {
    int c = startwert;
    int laenge = 1;
    while ( c != 1 ) {
        if ( c % 2 == 0 ) {
            c = c/2;
        }  else {
            c = 3*c+1;
        }
        laenge = laenge + 1;
    }
    return laenge;
}
```

Die Grundform einer Funktion ist

```
⟨Rückgabetyp⟩ ⟨Funktionsname⟩ ( ⟨Parameter⟩, ... ) {
  // Anweisungen
}
```

Jede Funktion braucht einen eindeutigen Namen[6]. In unserem Fall ist der Name `collatz_laenge`. Eine Funktion kann maximal einen Wert zurückgeben (analog zu den Polynomfunktionen aus der Analysis, wenn Sie $f(x)$ bestimmen, bekommen Sie den y-Wert des zugehörigen Punkts (x, y) des Funktionsgraphen zurück). Wir müssen bei einer Funktion festlegen, welchen Typ die Rückgabe hat. Das erfolgt durch Voranstellen eines Typbezeichners (in unserem Beispiel der Collatzfolge ist das `int`) vor den Funktionsnamen. Wenn nichts zurückgegeben werden soll (wie z.B. bei der Funktion `printf`), dann muss `void` vorangestellt werden, z.B.

```
void ausgabe(int ergebnis) {
  printf("Die gesuchte Zahl ist %d.\n",ergebnis);
}
```

Nach dem Funktionsnamen können Parameter an die Funktion übergeben werden. Diese stehen zwischen normalen Klammern (wieder wie bei Polynomfunktionen der Form $f(x) = x^2$ oder $f(x, y) = x^2 + y^2$, in denen x bzw. x und y die an die Funktion übergebenen Parameter sind). Auch diese Parameter brauchen einen Typ (in unserem Beispiel haben wir genau einen Parameter `int startwert`. Wenn wir mehr als einen Parameter vorsehen, werden diese durch Kommata getrennt. Die Reihenfolge, in der die Parameter später beim Aufruf übergeben werden, muss mit der Reihenfolge übereinstimmen, in denen die Parameter in diesem Funktionskopf aufgeführt werden. Die Klammern müssen auch vorhanden sein, wenn die Funktion keinen Parameter annimmt, z.B.

```
float euler() {
  return 2.71828;
}
```

Die Anweisungen, die beim Aufruf der Funktion ausgeführt werden sollen, stehen anschließend in einem von geschweiften Klammern eingeschlossenen Codeblock. Die Parameter können innerhalb der Funktion natürlich benutzt werden.

Mit der Anweisung `return` können wir die Funktion verlassen und ggf. einen Wert an die aufrufende Stelle zurückgeben. Wir können `return` an beliebigen Stellen im Code und auch mehrfach verwenden (z.B. innerhalb einer Verzweigung, wenn die Rückgabe von der Verzweigungsbedingung abhängen soll). Funktionen, die nach Ihrer Deklaration einen Wert zurückgeben, müssen dies auch tun. Es ist die Aufgabe der Programmiererin sicherzustellen, dass immer ein `return` erreicht wird.

Funktionen, die keinen Wert zurückgeben, können `return` (ohne Argument) verwenden, z.B. um in einer Verzweigung die Funktion vorzeitig zu verlassen, müssen das aber nicht.

Wir können die übergebenen Parameter in einer Funktion auch verändern. Unsere Beispielfunktion hätte auch die Form

```
int collatz_laenge(int c) {
int laenge = 1;
```

[6] Es reicht, wenn der Name innerhalb jeder Einheit eindeutig ist, das schauen wir uns später an. Vorerst sollte ein Funktionsname im gesamten Programm eindeutig sein

```
  while ( c != 1 ) {
    if ( c % 2 == 0 ) {
      c = c/2;
    }  else {
      c = 3*c+1;
    }
    laenge = laenge + 1;
  }
  return laenge;
}
```

haben können.

Hier ergibt sich eine Frage, die Ihnen vielleicht nicht sofort offensichtlich ist. Betrachten wir das folgende Programm.

```
void aenderung(int x) {
  x = 5;       // wir aendern den Parameter zu 5
}

int main() {
  int a = 4;    // a ist 4
  aenderung(a);
  // welchen Wert hat a an dieser Stelle?
return 0;
}
```

Tatsächlich werden die Werte von Variablen bei der Übergabe an eine Funktion *kopiert*, im Hauptprogramm **main** hat die Variable `a` daher auch nach dem Aufruf der Funktion `aenderung()` den Wert 4. Die Übergabe einer Kopie an Funktionen ist im Prinzip bei allen an Funktionen übergebenen Daten so, an manchen Stellen ist das allerdings etwas verdeckt und wir lernen erst später, wie wir das erkennen.

Zum Beispiel scheinen Listen auf den ersten Blick eine Ausnahme zu sein:

```
void aenderung(int x[2]) {
  x[0] = 5;       // wir aendern den Parameter zu 5
}

int main() {
  int a[2] = {4,5} ;    // a ist eine Liste mit zwei Eintraegen 4 und 5
  aenderung(a);
  // welchen Wert hat a[0] an dieser Stelle?
return 0;
}
```

Hier hat `a[0]` nach Aufruf der Funktion den Wert 5. Die Einträge von Listen lassen sich also innerhalb einer Funktion verändern. Das können wir ausnutzen, z.B. wenn wir mehr als einen Wert aus einer Funktion zurückgeben wollen. Wir können dafür vor dem Aufruf eine Liste passender Länge deklarieren, diese an die Funktion übergeben und die Rückgabewerte darin speichern, und sie dann in der aufrufenden Funktion anschließend auslesen und verwenden.

Wir sehen später, dass auch hier die in `a` gespeicherte Information *kopiert* wird, allerdings steht in `a` nur ein Verweis (ein *Zeiger)* auf einen Bereich im Speicher, in dem die eigentlichen Listeninhalte abgelegt sind. Wir haben in der Funktion also einen zweiten Verweis auf die gleiche Stelle im Speicher, die wir dann mit der Zuweisung von 5 auf $x[0]$ verändern. Das greifen wir später noch einmal im Detail auf.

Version vom 10. Februar 2022

Manchmal ist es wünschenswert, in Funktionen einige Statusinformationen oder Einstellungen über alle Aufrufe der Funktion hinweg beizubehalten. Dafür können wir innerhalb von Funktionen einer Variablendeklaration das Schlüsselwort `static` voranstellen. Diese Variable wird dann nur beim ersten Aufruf der Funktion deklariert, und ihr Zustand bleibt für alle weiteren Aufrufe der Funktion erhalten. Eine typische Anwendung ist eine Variable, die zählt, wie oft eine Funktion aufgerufen wurde. Hier ist ein Beispiel.

Programm 3.7: `kapitel_03/static.c`

```c
#include <stdio.h>

void counter () {
  static int a = 1;
  printf("Das ist der %d. Aufruf\n",a++);
}

int main() {

  counter();
  counter();
  counter();

  return 0;
}
```

Das Programm gibt

```
Das ist der 1. Aufruf
Das ist der 2. Aufruf
Das ist der 3. Aufruf
```

auf dem Bildschirm aus.

3.5 Zufall

Die Funktion `rand()` erzeugt eine (Pseudo-)Zufallszahl im Bereich `[0...RAND_MAX]`. Diese Funktion ist in der Bibliothek `stdlib.h` definiert, die daher am Anfang mit

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

eingebunden werden muss.

Wirklich zufällige Zahlen können auf einem Computer nicht erzeugt werden, wir sprechen daher eigentlich besser von Pseudozufallszahlen. Die Folge der Zahlen, die von der Funktion `rand()` erzeugt wird, ist tatsächlich deterministisch. Ohne geeignete Initialisierung erhalten wir immer die gleiche Folge von Zahlen. Die Folge ist auch periodisch, d.h. wenn wir ausreichend viele Zufallszahlen erzeugen, wiederholt sich die Folge. Bei moderneren Implementierungen müsste man allerdings sehr lange auf eine Wiederholung warten, und wenn man eine statistische Auswertung auf die erhaltenen Zahlen macht, dauert es eine Weile, bis man erkennt, dass die Folge deterministisch ist.

Zur Initialisierung gibt es die Funktion `srand(seed)`, wobei `seed` vom Typ `unsigned int` sein muss. Damit bleibt die Folge zwar deterministisch, aber durch die Eingabe

eines `seed` wird der Eintrittspunkt in die Folge variiert. Der Zufallsgenerator nimmt diese Zahl zur Bestimmung der ersten Zufallszahl.

Diese Wahl sollte sinnvollerweise mit jedem Aufruf des Programms wechseln. Eine typische Wahl für `seed` ist `time(0)`, was die aktuelle Zeit (in Sekunden seit 1970) angibt. Diese Funktion erfordert

```
#include <time.h>
```

am Anfang des Programms. Eine mögliche Anwendung zeigt das folgende Programm, das eine Folge von 100 Zufallszahlen ausgibt.

Programm 3.8: `kapitel_03/zufallszahlen.c`

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

int main() {
  srand(time(0));
  printf("100 Zufallszahlen im Bereich von 0 bis %d.\n",RAND_MAX);
  for ( int i = 0; i < 100; ++i ) {
    printf("%d ",rand());
  }

  printf("\n");
  return 0;
}
```

Zur Fehlersuche kann es sinnvoll sein, `srand` mit einer festen Zahl zu initialisieren, damit jeder Programmlauf identisch wird.

Version vom 10. Februar 2022

4 Programmstrukturierung

Wir wollen uns in diesem Kapitel einige Möglichkeiten ansehen, wie längerer Code sinnvoll strukturiert werden kann. Dazu schauen wir uns zuerst den Unterschied zwischen einer Funktions*deklaration* und ihrer -*definition* an, bevor wir sehen, wie wir Code auf mehrere Dateien aufteilen können und erst beim Übersetzen zusammenfügen. Dafür führen wir *Header* ein, die wir benutzen können, um Funktionsdeklarationen in anderen Dateien bekannt zu machen, damit diese Funktionen dann zur Verfügung stehen. Dieser Ansatz wird von vielen Programmbibliotheken verfolgt, und insbesondere auch von der C-Standardbibliothek, der wir schon mit der Einbindung von `stdio.h` begegnet sind.

4.1 Deklaration und Definition

Bisher haben wir eine Funktion an genau einer Stelle im Code notiert. Jede weitere Definition führt beim Übersetzen des Codes zu einer Fehlermeldung. Es kann jedoch manchmal sinnvoll oder sogar notwendig sein, den Namen einer Funktion sowie ihren Typ und den ihrer Parameter an verschiedenen Stellen im Code bekannt zu machen, da sich der Compiler sonst über eine *unbekannte* oder *implizit definierte* Funktion beschwert. Um dieses Problem zu umgehen gibt es die Möglichkeit, eine Funktion nur zu *deklarieren* statt direkt zu definieren. Dafür schauen wir uns die folgenden beiden Beispiele an.

```
1   int func(int var1, int var2);
2
```

```
1   int func(int var1, int var2) {
2     return var1 + var2;
3   }
4
```

Im linken Beispiel wird die Funktion nur *deklariert*. Auf diese Weise machen wir den Namen der Funktion bekannt und legen fest, welchen Typ die Rückgabe hat, wieviele Parameter die Funktion hat und von welchem Typ diese sind.

Auf der rechten Seite steht dann eine vollständige *Definition* der Funktion. Hier führen wir aus, was die Funktion mit den Parametern machen soll und wie sie ihre Rückgabe bestimmt. Bisher haben wir Funktionen immer diesen Weg gewählt und unsere Funktionen direkt vollständig definiert.

Für den Compiler ist es zum Zeitpunkt des Übersetzens des Programms in Maschinencode an allen Stellen, wo die Funktion benutzt wird, ausreichend, wenn der Datentyp der Rückgabe und die Parameter bekannt sind, um den Code, in dem sie aufgerufen wird, zu übersetzen. An der Stelle, an der die Funktion aufgerufen wird, wird beim Übersetzen eine *Sprungmarke* im Maschinencode eingefügt, die später auf die Funktion, deren Maschinencode ja ebenfalls im Speicher liegt, verweist. Erst wenn alle Programmteile am Ende zu einem einzigen Gesamtprogramm zusammengefügt werden, muss

Version vom 10. Februar 2022

irgendwo auch eine Definition der Funktion vorhanden sein, für die dann tatsächlich Maschinencode erzeugt wird, und auf die dann die Sprungmarken zeigen können.[1] Diese *Zusammenführung* der Codeteile wird nach dem Übersetzen in einem zweiten Schritt gemacht.

Der Maschinencode der Funktion darf nur ein einziges Mal erzeugt werden, sonst ist für den Compiler nicht ersichtlich, auf welche Kopie verwiesen werden soll. Es darf also (insgesamt in allen Dateien unseres Projekts) nur eine einzige Definition der Funktion geben. Wir dürfen eine Funktion jedoch beliebig oft deklarieren, solange alle Deklarationen identisch sind.

Diese Möglichkeit, eine Funktion nur zu deklarieren, kann aus verschiedenen Gründen sinnvoll sein. Es ermöglicht uns zum Beispiel, unseren Code anders zu strukturieren. Um eine Übersicht über alle vorhandenen Funktionen zu haben, könnten wir zum Beispiel alle Funktionen am Anfang deklarieren, und die Definitionen erst im Anschluss, oder sogar erst nach der Funktion `main`, schreiben[2]. Dann sehen wir am Anfang des Programms, welche Methoden uns zur Verfügung stehen und welche Parameter sie benötigen. Im nächsten Abschnitt sehen wir auch, dass es sinnvoll sein kann, Deklaration und Definition sogar auf unterschiedliche Dateien zu verteilen.

Insbesondere bei der Aufteilung von Funktionen über mehrere Dateien kann es aber auch zwingend notwendig sein, eine Funktion zu deklarieren, da eine Funktion nur einmal definiert werden kann, ihre Deklaration aber in allen Dateien zur Verfügung stehen muss.

Aber auch innerhalb einer einzigen Datei kann es notwendig sein. Das nachfolgende Beispiel, das die Collatzfolge zu einem Startwert auf den Bildschirm schreibt, ist zu kurz, um eine solche Aufteilung der Funktionen wirklich sinnvoll zu machen, aber es zeigt, dass es Lösungen des Problems geben kann, bei denen eine Deklaration zwingend notwendig ist.

Programm 4.1: `kapitel_04/collatz_declaration.c`

```
1  /************************************
2  * Beispielprogramm zur Vorlesung
3  * Einfuehrung in die Programmierung I
4  * Andreas Paffenholz
5  * TU Darmstadt, Wintersemester 2021/22
6  * (c) 2021-
7  *
8  * Collatz mit Funktionsdeklarationen
9  *
10 * Uebersetzen mit
11 * gcc collatz_declarations.c -o collatz_declarations
12 * Aufruf mit
13 * ./collatz_declarations
14 ************************************/
15
16 #include <stdio.h>
```

[1] Selbst das muss nicht unbedingt sein, wenn wir mit dynamischen Bibliotheken arbeiten, die erst zur Laufzeit des Programms geladen werden. Dann fällt das Fehlen einer Definition zu einer Funktionsdeklaration erst beim Aufruf des Programms auf. Wir schreiben aber keine Programme, die Bibliotheken dynamisch laden.

[2] Manche Programmierer*innen ziehen auf diese Weise die Funktion `main` an den Anfang des Programms. Wir werden diesem Schema in diesem Skript nicht folgen.

```c
#include <stdlib.h>

void collatz_multiply ( int c );

void collatz_divide (int c) {
  do {
    printf("%d ",c);
    c = c/2;
  }  while ( c % 2 == 0 );

  if ( c != 1 ) {
    collatz_multiply(c);
  }
}

void collatz_multiply ( int c ) {
  printf("%d ",c);
  collatz_divide(3*c+1);
}

int main(int argc, char** argv ) {
  int s = atoi(argv[1]);

  if ( s != 1 ) {
    s % 2 == 0 ? collatz_divide(s) : collatz_multiply(s);
  }
  printf("1\n");

  return 0;
}
```

Wie schon diskutiert, sollten wir immer versuchen, kleine abgeschlossene Teile unseres Programms, die eine spezifische Aufgabe lösen, in eine Funktion auszulagern. Dadurch können wir den Code besser strukturieren und später leichter Veränderungen vornehmen. Ebenso können wir die gleiche Funktion mehrfach benutzen, wenn eine Aufgabe an verschiedenen Stellen im Programm gelöst werden muss. Wenn Sie diesem Ansatz folgen, können Sie bei etwas komplizierteren Programmen dazu kommen, dass eine Aufteilung in solche Einheiten eine Deklaration am Anfang erzwingt.

Bei der Deklaration einer Funktion können wir auch die Namen der Parameter weglassen (aber nicht den Bezeichner der Funktion). Die Deklaration vom Anfang könnte auch so aussehen:

```c
  int func(int, int);
```

Das ist oft auch sinnvoll. Wenn wir in einer Funktionsdefinition den Parameternamen ändern, sollten wir sonst daran denken, das auch in allen Deklarationen anzupassen. Das ist zwar nicht notwendig, ist aber oft schwer zu lesen, wenn die Parameternamen abweichen. Es kann sinnvoll sein, Namen zu vergeben, wenn die Parameternamen Ihre Bedeutung beschreiben und so die Verwendung der Funktion schon an ihrer Deklaration ersichtlich ist. Allerdings sollte die Verwendung einer Funktion, ihre Eingabe und Ausgabe, und damit auch die Anzahl und Reihenfolge ihrer Parameter, immer auch in der begleitenden Dokumentation beschrieben werden. Dabei bietet es sich oft an, Funktionen oberhalb ihrer Deklaration oder Definition zu dokumentieren.

Es gibt auch die Möglichkeit, Variablen nur zu deklarieren, statt sie zu definieren. Bei

ihrer Definition wird auch ein passender Speicherbereich reserviert und der Variablen zugeordnet. Das sollte sinnvollerweise nur ein einziges Mal passieren, damit alle Referenzen auf diese Variable auch den richtigen Speicherbereich auslesen. Wir werden sehen, dass es, sobald wir unseren Code auf mehr als eine Datei verteilen, es manchmal aber nötig sein kann, den Namen einer Variablen an mehreren Stellen bekannt zu machen. Dafür brauchen wir ein weiteres Schlüsselwort `extern`, denn die Varianten

```
1    int x;
2
```

```
1    int x = 0;
2
```

reservieren beide Speicher für `x`. Im ersten Fall wird nur bei der Belegung des Speichers kein Wert in den Bereich geschrieben, wenn wir also auf `x` zugreifen, bevor auch ein Wert zugewiesen wurde, erhalten wir keinen sinnvollen Wert zurückgeliefert. Mit

```
1  extern int x;
```

wird eine Variable nur deklariert.

4.2 Header

Bisher haben wir alle Programme vollständig in eine einzige Datei geschrieben. Sobald unser Code jedoch länger wird, wird das unübersichtlich. Daher haben wir die Möglichkeit, Code auf mehrere Dateien zu verteilen und erst beim Übersetzen des Programms[3] wieder zusammenzuführen.

Wir haben das auch schon verwendet, indem wir in der ersten Zeile unserer Programme

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

eingefügt haben. Die `#include`-Anweisung[4] kopiert den Code der Datei `stdio.h` an diese Stelle bevor der Compiler mit der Übersetzung des Programms anfängt. Dabei wird an festgelegten Stellen auf dem System nach der Datei gesucht, daher muss hier nicht der volle Pfad im Dateisystem angegeben werden.

Wir können die gleiche Methode auch für unsere eigenen Dateien benutzen. Wenn wir Code in einer Datei `zusatz.h` im gleichen Ordner, in dem unser eigentliches Programm liegt, abspeichern, können wir den Code mit

```
1  #include "zusatz.h"
```

beim Übersetzen an diese Stelle kopieren. Beachten Sie, dass wir hier den Dateinamen in Anführungszeichen statt spitze Klammern gesetzt haben. Dadurch wird die Datei zuerst im Verzeichnis der Datei, in die sie eingebunden ist, gesucht. Ein Beispiel für eine Bibliothek und ein Hauptprogramm, in das sie eingebunden wird, steht in Programm 4.2. Es ist üblich, aber nicht zwingend, Dateinamen für Bibliotheken die Endung `.h` zu geben. Die meisten Editoren erkennen solche Dateien dann als `C`-Bibliothek und können den Code entsprechend darstellen und auch nach weiteren zugehörigen Dateien suchen.

[3] eigentlich erst beim *Linken* des Programms, das sehen wir später im Detail.

[4] eigentlich eine Anweisung an den Präprozessor, nicht an den Compiler selbst. Der Präprozessor läuft vor dem Compiler und bereitet den Code auf. Das betrachten wir später genauer.

Programm 4.2: kapitel_04/collatz_main.c

```c
#include <stdio.h>
#include "collatz.h"

int main() {
    int startwert = 0;
    printf("Geben Sie eine positive ganze Zahl ein: ");
    scanf("%d",&startwert);

    int laenge = collatz_laenge(startwert);
    printf("Die Collatz-Folge mit Start %d hat Laenge %d\n",startwert,laenge);

    return 0;
}
```

Programm 4.2: kapitel_04/collatz.h

```c
int collatz_laenge(int startwert) {
    int c = startwert;
    int laenge = 1;
    while ( c != 1 ) {
        if ( c % 2 == 0 ) {
            c = c/2;
        }  else {
            c = 3*c+1;
        }
        laenge = laenge + 1;
    }
    return laenge;
}
```

An dem Aufruf des Compilers ändert sich dabei nichts, der nachfolgende Befehl erzeugt ein Programm mit dem Namen `collatz`.

```
& gcc collatz_main.c -o collatz
```

Wie wir uns schon weiter oben überlegt haben, kann es sinnvoll sein, alle Funktionsdeklarationen an den Anfang der Datei zu ziehen, um schneller einen Überblick über die Verfügbaren Methoden zu bekommen. Das ist bei der Verteilung von Code auf mehrere Dateien noch wichtiger, um eine gesuchte Funktion leichter finden zu können. Wir können sogar noch einen Schritt weitergehen und die Deklarationen in einer eigenen Datei zusammenzufassen und die (meisten der) Definitionen in eine eigene Datei zu schreiben. Dateien, die die Deklarationen enthalten, werden üblicherweise *Header* genannt (daher kommt auch die Dateiendung). Es hat sich eingebürgert, und wird von manchen Editoren auch so erwartet, dem Header mit den Deklarationen und der Datei mit den zugehörigen Definitionen den gleichen Namen zu geben, und für den Header die Endung `.h` zu verwenden und der Datei mit den Definitionen die Endung `.c` zu geben. Das erleichtert die Zuordnung der Dateien. Damit könnte unser Programm für die Collatzfolgen wie in Programm 4.3 aussehen.

Programm 4.3: kapitel_04/collatz_main_v2.c

```c
#include <stdio.h>
#include "collatz_v2.h"

int main() {
```

```
5       int startwert = 0;
6       printf("Geben Sie eine positive ganze Zahl ein: ");
7       scanf("%d",&startwert);
8
9       int laenge = collatz_laenge(startwert);
10      printf("Die Collatz-Folge mit Start %d hat Laenge %d\n",startwert,laenge);
11
12      return 0;
13  }
```

Programm 4.3: kapitel_04/collatz_v2.h

```
1   int collatz_laenge(int);
```

Programm 4.3: kapitel_04/collatz̆.c

```
1   #include "collatz_v2.h"
2
3   int collatz_laenge(int startwert) {
4       int c = startwert;
5       int laenge = 1;
6       while ( c != 1 ) {
7           if ( c % 2 == 0 ) {
8               c = c/2;
9           }  else {
10              c = 3*c+1;
11          }
12          laenge = laenge + 1;
13      }
14      return laenge;
15  }
```

Beachten Sie, dass wir hier die Datei `collatz_v2.h` über den Befehl `#include` in `collatz_v2.c` geladen haben. Das ist hier noch nicht zwingend notwendig, aber man möchte oft kleine Codeteile oder Variablendefinitionen im Header stehen lassen, und dann müssen diese bei Übersetzung der Funktionsdefinitionen bekannt sein.

Bei dieser Aufteilung ändert sich der Aufruf des Compilers. Wir müssen alle Dateien mit Funktionsdefinitionen an den Compiler übergeben, also z.B. `gcc` wie folgt aufrufen.

```
& gcc collatz_main_v2.c collatz_v2.c -o collatz
```

Der Header `collatz_v2.h` muss weiterhin in dem Aufruf nicht explizit genannt werden, da der Inhalt dieser Datei über `#include` in die beiden anderen Dateien kopiert wird.

Sobald auch Definitionen in Headern enthalten sind, bekommen wir jedoch mit unserem Ansatz ein Problem, wenn wir einen Header in mehr als eine Datei einbinden wollen. Für Funktionen können wir das Problem in der Regel umgehen, nicht jedoch für Variablen oder für die komplexeren Datenstrukturen, die wir später selbst definieren wollen. Im nächsten Abschnitt werden wir zwar diskutieren, dass in den meisten Fällen nicht sinnvoll ist, eine Variable außerhalb einer Funktion zu definieren, aber manchmal ist doch nützlich. Später lernen wir auch, wie wir komplexere Datentypen für unsere Anwendungen selbst definieren können.

Da eine Definition im Unterschied zur Deklaration nur genau einmal auftauchen darf, kann ein Header, der eine Definition enthält, nicht ohne weiteres mehrfach eingebunden werden.

Wir könnten versucht sein, solche Definitionen in einer eigenen Datei zu sammeln, die wir am Anfang einmal einbinden. Zum einen ist allerdings oft nicht klar, in welcher Reihenfolge Header eingebunden werden, oder wir haben überhaupt keine Kontrolle darüber, weil wir Code verwenden, der nicht von uns geschrieben wurde, zum anderen werden alle Codedateien (mit der Endung `.c`), die wir an `gcc` übergeben, zuerst einzeln übersetzt, bevor sie zu einem einzigen Programm zusammengesetzt werden. Eine Variablendefinition muss jedoch in allen Teilen, in der die Variable verwendet wird, schon zum Zeitpunt des Übersetzens dieses einzelnen Teils verfügbar sein.

Eine Möglichkeit, dieses Problem aufzulösen, ist die Verwendung von Steuerungsanweisungen an den Präprozessor, der die einzelnen Dateien zusammenbaut. Wir können diese Präprozessoranweisungen, die vor jeder Interpretation des Codes und Überprüfungen der Syntax und Konsistenz ausgeführt werden, benutzen, um zu kontrollieren, ob eine Datei mit Deklarationen und ggf. auch Definitionen schon eingebunden wurde und die darin enthaltenen Funktions- und Variablennamen schon bekannt sind.

Konkret können wir mit der Anweisung `#define`, die wir schon für globale Konstanten kennengelernt haben, im Header eine Variable definieren, anhand derer wir testen können, ob der Header schon eingebunden ist. Diesen Test können wir mit `#ifndef ... #endif` machen, wobei `#ifndef` testet, ob die nachfolgende Variable *nicht* definiert ist. Unser Header könnte so aussehen:

```
1  #ifndef COLLATZ_V2_H
2  #define COLLATZ_V2_H
3
4  int collatz_laenge(int);
5
6  #endif
```

Hier wird in der ersten Zeile getestet, ob die Variable `COLLATZ_V2_H` definiert ist. Wenn das der Fall ist, wird alles bis `#endif` ignoriert. Andernfalls wird die Variable definiert und der Code des Headers eingebunden. Das wird genau ein einziges Mal passieren.

Wir schauen uns an einer Reihe von Beispielen noch einmal an, dass die Aufteilung von Bibliotheken in Deklarationen und Definitionen sowie die Absicherung gegen mehrfaches Einbinden der gleichen Bibliothek in manchen Fällen notwendig ist. Dazu betrachten wir zuerst das Beispiel aus Programm 4.4. Mit

```
gcc main_v1.c -o main_v1
```

können Sie das Programm erfolgreich übersetzen.

Wir können an diesem Vorgehen aber gleich mehrere Probleme erkennen. In der Bibliothek B wird eine Funktion von A aufgerufen. Wir wissen schon, dass diese Funktion dann mindestens schon deklariert sein muss, damit der Übersetzer weiß, welche Parameter übergeben werden und was zurückkommt. Die Definition ist hier schon vorhanden, da eine `#include`-Anweisung nur dazu führt, dass vor dem Übersetzen der Code aus der eingebundenen Datei an die Stelle des `#include` kopiert wird. Da wir die Bibliothek A vor B einbinden, kommt die Definition von `f_A_call_from_B()` zuerst. Wenn wir jetzt allerdings nicht wüssten, dass eine Funktion aus A in B verwendet wird, könnten wir die Bibliotheken auch in umgekehrter Reihenfolge (B vor A) im Hauptprogramm einbinden wollen. Dann lässt sich jedoch das Hauptprogramm nicht mehr übersetzen (probieren Sie es!).

Sie können sich den Code, der nach Bearbeitung aller Präprozessor-Anweisungen

Programm 4.4: kapitel_04/A_v1.h

```
1  #include <stdio.h>
2
3  //#include "B_v1.h"
4
5  void f_A_call_from_main() {
6    printf("[f_A_1] aufgerufen\n");
7    //  f_B_call_from_A();
8  }
9
10 void f_A_call_from_B() {
11   printf("[f_A_2] aufgerufen\n");
12 }
```

Programm 4.4: kapitel_04/B_v1.h

```
1  #include <stdio.h>
2
3  //#include "A_v1.h"
4
5  void f_B_call_from_main() {
6    printf("[f_B_1] aufgerufen\n");
7    f_A_call_from_B();
8  }
9
10 void f_B_call_from_A() {
11   printf("[f_B_2] aufgerufen\n");
12 }
```

Programm 4.4: kapitel_04/main_v1.c

```
1  #include "A_v1.h"
2  #include "B_v1.h"
3
4  int main() {
5
6      f_A_call_from_main();
7      f_B_call_from_main();
8
9      return 0;
10 }
```

(also unter anderem aller `#include`s und `#define`s) an den eigentlichen Übersetzer übergeben wird, anzeigen lassen, wenn Sie den Parameter `-E` an `gcc` übergeben.

```
gcc -E main_v1.c
```

schreibt eine lange Folge von Code auf den Bildschirm. Am Ende können Sie die beiden Bibliotheken und das Hauptprogramm wiederfinden.

Hier ist ein weiteres Problem mit diesem Ansatz. Möglicherweise brauchen wir den Aufruf von `f_A_call_from_main()` im Hauptprogramm nicht. Dann ist beim Schreiben des Hauptprogramms nicht mehr ersichtlich, dass wir die Bibliothek A überhaupt einbinden müssen. Wenn wir das zugehörige `#include` allerdings herausnehmen, dann fehlt die Definition dieser Funktion. Vernünftigerweise binden wir dann die Bibliothek A in die Bibliothek B ein, indem wir die Kommentierung in Zeile 2 entfernen.

Dies ist auch grundsätzlich ein sinnvolles Vorgehen. In einer Bibliothek sollten alle Bibliotheken eingebunden werden, deren Funktionen verwendet werden. Auf diese Weise müssen Sie, wenn Sie die Bibliothek verwenden wollen, nicht daran denken, und es auch nicht unbedingt wissen, dass weitere Bibliotheken eingebunden werden müssen.

Wenn wir jetzt allerdings doch beide Funktionsaufrufe im Hauptprogramm lassen, muss auch dort die Bibliothek A eingebunden werden, und nach Auflösen der `#includes` steht die Funktion doppelt im Code (schauen Sie sich das mit `gcc -E` an). Hier können wir uns mit dem oben schon beschriebenen Trick helfen, bei der ersten Einbindung einer Bibliothek eine Kontrollvariable zu setzen, die wir vor dem Einbinden abfragen. Damit könnte unser Code wie in Programm 4.5 aussehen.

Programm 4.5: kapitel_04/A_v2.h

```
1  #ifndef HEADER_A_DEF_H
2  #define HEADER_A_DEF_H
3
4  #include <stdio.h>
5
6  //#include "B_v2.h"
7
8  void f_A_call_from_main() {
9    printf("[f_A_1] aufgerufen\n");
10   //  f_B_call_from_A();
11 }
12
13 void f_A_call_from_B() {
14   printf("[f_A_2] aufgerufen\n");
15 }
16
17 #endif
```

Programm 4.5: kapitel_04/B_v2.h

```
1  #ifndef HEADER_B_DEF_H
2  #define HEADER_B_DEF_H
3
4  #include <stdio.h>
5
6  #include "A_v2.h"
7
8  void f_B_call_from_main() {
9    printf("[f_B_1] aufgerufen\n");
10   f_A_call_from_B();
11 }
12
13 void f_B_call_from_A() {
14   printf("[f_B_2] aufgerufen\n");
15 }
16
17 #endif
```

Programm 4.5: kapitel_04/main_v2.c

```
1  #include "A_v2.h"
2  #include "B_v2.h"
3
4  int main() {
5
6      f_A_call_from_main();
7      f_B_call_from_main();
8
9      return 0;
10 }
```

In dieser Form lässt sich das Programm wieder übersetzen, obwohl Bibliothek A an zwei Stellen eingebunden wird.

Wir sind jedoch noch nicht alle Probleme los. Falls wir in der Bibliothek A eine Funktion aus B verwenden wollen, z.B. indem wir die Kommentierung in Zeile 9 entfernen (und dann in Zeile 5 auch die Bibliothek einbinden), können wir das Programm nicht mehr übersetzen, obwohl es prinzipiell möglich sein sollte. Durch unsere Aufteilung der Funktionen muss nun einerseits A vor B eingebunden werden, um die Deklaration von `f_A_call_from_B()` zu kennen, für A muss aber B vorhanden sein, um die Deklaration von `f_B_call_from_A()` zu kennen.

Die wechselseitigen Aufrufe der Funktionen sind jedoch nur in der Definition enthalten, nicht in der Deklaration. Letztere wäre jedoch ausreichend gewesen, damit `C` weiß, wie es die Funktionen behandeln muss. Die einzige Option, diesen Konflikt aufzulösen ist nun, wie oben beschrieben, die Definitionen und Deklarationen in unterschiedliche Dateien zu schreiben, und jeweils nur die Datei mit den Deklarationen über `#include` einzubinden. Damit sieht unser Code wie in Programm 4.6 aus. In dieser Form sind alle Teile in sich unabhängig verwendbar und die wechselseitige Einbindung der Funktionen ist aufgelöst. Dafür müssen jetzt alle Teile mit Funktionsdefinitionen auch übersetzt werden. Der Aufruf von `gcc` sieht daher jetzt so aus:

Programm 4.6: kapitel_04/A_v3.h

```
#ifndef HEADER_A_DEF_H
#define HEADER_A_DEF_H

void f_A_call_from_main();
void f_A_call_from_B();

#endif
```

Programm 4.6: kapitel_04/B_v3.h

```
#ifndef HEADER_B_DEF_H
#define HEADER_B_DEF_H

void f_B_call_from_main();
void f_B_call_from_A();

#endif
```

Programm 4.6: kapitel_04/A_v3.c

```
#include <stdio.h>

#include "A_v3.h"
#include "B_v3.h"

void f_A_call_from_main() {
  printf("[f_A_1] aufgerufen\n");
  f_B_call_from_A();
}

void f_A_call_from_B() {
  printf("[f_A_2] aufgerufen\n");
}
```

Programm 4.6: kapitel_04/B_v3.c

```
#include <stdio.h>

#include "B_v3.h"
#include "A_v3.h"

void f_B_call_from_main() {
  printf("[f_B_1] aufgerufen\n");
  f_A_call_from_B();
}

void f_B_call_from_A() {
  printf("[f_B_2] aufgerufen\n");
}
```

Programm 4.6: kapitel_04/main_v3.c

```
#include "A_v3.h"
#include "B_v3.h"

int main() {

    f_A_call_from_main();
    f_B_call_from_main();

    return 0;
}
```

```
gcc main_v3.c header_A_v3.c header_B_v3.c -o main_v3
```

Hier werden alle drei Dateien erst einzeln von `gcc` in Maschinencode übersetzt, bevor sie zu einem einzigen Programm `main_dec` zusammengepackt werden. Diese Schritte könnten wir auch einzeln ausführen und auf diese Weise nur Teile übersetzen, die sich auch wirklich verändert haben, aber das sehen wir uns erst später an, wenn wir Bibliotheken betrachten, mit denen wir fremden Code in unsere Programme übernehmen können.

Diese Form der Aufteilung ist auch die übliche Form, sobald Code in Bibliotheken ausgelagert wird. In dieser Vorlesung werden wir in den Beispielen dieser Konvention nicht immer folgen, um bei den Codebeispielen nicht immer mehrere Dateien durchgehen zu müssen. Ihrem eigenen Code sollten Sie von Anfang an über diese Struktur in

einzelne angeschlossene Einheiten zerlegen.

4.3 Bereiche und Dateien

Wir schon haben bisher verlangt, dass Code, der zu einer Schleife oder Verzweigung gehört, in geschweiften Klammern eingeschlossen wird. Diese Klammern gehören eigentlich nicht direkt zu den Schleifen- und Verzweigunsanweisungen, sondern fassen die Anweisungen innerhalb der Schleife oder Verzweigung zu einer einzigen Anweisung zusammen. Wenn nur genau eine Anweisung in einer Schleife wiederholt werden soll, sind diese Klammern tatsächlich auch nicht notwendig. Die folgenden Programmausschnitte addieren beide die Zahlen von 0 bis 9.

```
1   int summe = 0;
2   for ( int i = 0; i < 10; ++i ) {
3     summe += i;
4   }
```

```
1   int summe = 0;
2   for ( int i = 0; i < 10; ++i )
3     summe += i;
```

Nach ihrer vorgegebenen Syntax erwarten Schleifen und Verzweigungen genau eine Anweisung, auf die sie sich beziehen. Nun wollen wir in der Regel mehr als eine Anweisung wiederholen lassen, und an dieser Stelle kommen die Klammern ins Spiel. Mit den geschweiften Klammern können wir mehrere Anweisungen so zusammenfassen, dass sie *nach außen* als eine einzige Anweisung angesehen werden[5].

Sie können die Klammern auch unabhängig von Schleifen oder Verzweigungen verwenden, um eine Reihe von Anweisungen zu einem einzigen Anweisungsblock zusammenzufassen.

```
1   int main() {
2
3     int x = 7;
4     {
5       int y = x+4;
6       {
7         int i = 0;
8         {
9           for ( i = 0; i < 3; ++i ) {
10            y += i*y;
11          }
12        }
13      }
14    }
15    return 0;
16  }
```

Allerdings hat das hier keine weiteren Auswirkungen auf das übersetzte Programm. Wir sehen aber schon, dass die Anweisungsblöcke auch geschachtelt sein können.

Die geschweiften Klammern machen jedoch einen neuen *Bereich* auf, der die Gültigkeit von Variablendeklarationen beschränkt. Alle Variablen, die innerhalb eines Bereichs definiert werden, sind auch nur innerhalb dieses Bereichs bekannt. Sobald der Bereich

[5] Bei Funktionsdefinitionen sind die geschweiften Klammern allerdings notwendig, auch wenn nur ein `return` in der Funktion stünde, hier gehören sie zur Syntax dazu.

endet, wird die Variable gelöscht und der dafür reservierte Speicherbereich wieder freigegeben. Solche Variablen werden auch als *lokale Variablen* bezeichnet.

Wir haben das schon bei Funktionen gesehen, wo wir alle Werte, die wir in der Funktion benutzen wollten, auch als Parameter übergeben mussten, damit wir auf Ihre Werte zugreifen konnten. Auch die Schleifenvariable `i`, die wir bei der Definition einer Schleife in der Form

```
1  for ( int i = 0; i < 3; ++i ) {
2    // Anweisungen
3  }
4  i++; // Fehler, i ist unbekannt
```

bekommen, ist nur innerhalb der Schleife verfügbar.

Wir können sogar innerhalb eines Bereichs eine schon außerhalb definierte Variable neu definieren und damit die ursprüngliche Variable verdecken. Das Programm

Programm 4.7: `kapitel_04/bereiche.c`

```
1  #include <stdio.h>
2
3  int main() {
4    int x = 0;
5    {
6      printf("%d ",x);
7      int x = 1;
8      printf("%d ",x);
9      {
10       int x = 2;
11       printf(" %d ",x);
12     }
13     printf(" %d ",x);
14   }
15   printf(" %d\n",x);
16   return 0;
17 }
```

schreibt

```
0 1 2 1 0
```

auf den Bildschirm. In den meisten Situationen sollte man von dieser Möglichkeit allerdings eher keinen Gebrauch machen, da die wechselnde Bedeutung von `x` leicht zu übersehen ist und daher zu schwer erkennbaren Fehlern führt. Wichtig wird diese Eigenschaft, sobald sie Code aus mehreren Quellen zusammenfügen und keinen Einfluss auf die Wahl der Variablennamen haben. Dann können wir über die Lokalität der Variablendefinition sicherstellen, dass wir immer auf unsere eigene Variable zugreifen, auch wenn es, zum Beispiel in einem Header, noch eine andere Variable mit dem gleichen Namen gibt (die dann notwendigerweise eine *globale Variable* sein müsste, was wir uns gleich ansehen werden).

Eine Variante einer lokalen Variable sind solche, die mit dem Schlüsselwort `static` definiert worden sind. Wir haben dieses Schlüsselwort schon im vorherigen Abschnitt eingeführt. Solche Variablen sind zwar lokal, da sie außerhalb des Bereichs, in dem sie definiert wurden, nicht sichtbar sind, aber ihre Definition bleibt über alle Durchläufe des Bereichs erhalten, während normale Variablen bei jedem neuen Durchlauf neu

initialisiert werden. Das wirkt sich natürlich nur aus, wenn ein Bereich auch tatsächlich mehrfach erreicht wird, also zum Beispiel bei Schleifen und Funktionen.

Alle Variablen, die nicht mit `static` definiert sind, könnten wir auch mit dem Schlüsselwort `auto` zum Beispiel in der Form

```
auto unsigned int x = 3;
```

definieren, aber da `auto` der Standard ist, wenn nichts anderes angegeben ist, wird das in der Regel weggelassen.

Schleifen mit `for` erlauben Anweisungen im ersten Block. Wir haben das benutzt, um dort die Variable `i` zu definieren. Tatsächlich wird die ganze Schleife als eigener Bereich angesehen, der den Bereich der Anweisungen, die wiederholt werden sollen, noch umschließt. Daher ist der folgende Code gültig, allerdings nicht sonderlich sinnvoll, und es ist sicherlich auch schwer nachzuvollziehen, was der Code macht, sobald der Anweisungsblock in der Schleife umfangreicher wird und die Neudefinition von `i` nicht mehr so offensichtlich ist.

Programm 4.8: `kapitel_04/for_opens_block.c`

```c
#include <stdio.h>

int main() {

  int x = 5;
  for ( int i = 0; i < 3; ++i ) {
    printf("%d ",i);
    int i = 5;
    printf("%d ",i);
  }

  printf("\n");
  return 0;
}
```

Seine Ausgabe ist

```
0 5 1 5 2 5
```

Alle Variablen, die innerhalb eines Bereichs definiert werden, sind *lokale Variablen*. Wir können eine Variable auch außerhalb aller Bereiche definieren. Dadurch erhalten wir eine *globale Variable*. Eine solche Variable ist in allen Funktionen sichtbar, die in der gleichen Datei definiert werden (sofern wir die Variable nicht durch die Definition einer lokalen Variable gleichen Namens überdecken).

Programm 4.9: `kapitel_04/globale_variable.c`

```c
#include <stdio.h>

int g = 4;

inf func() {
  g++;
}

int main() {
  printf("%d ", g);
```

```
11    func();
12    printf("%d ", g);
13    func();
14    printf("%d\n", g);
15
16    return 0;
17  }
```

gibt

```
4 5 6
```

aus. Da es in größeren Programmen schwierig ist, globale Variablendefinitionen zu erkennen, sollten solche globalen Definitionen vermieden werden. Insbesondere muss ihre Definition, wenn unser Programm aus mehreren Dateien besteht, nicht unbedingt in der Datei definiert sein, die wir gerade betrachten, oder die von uns geschrieben wurde.

Globale Variablen können sinnvoll sein, wenn es eine zentrale Datenstruktur gibt, auf die alle (oder viele) Funktionen zurückgreifen müssen, und deren Zustand für das gesamte Programm gültig ist. Das könnte bei graphischen Programmen z.B. die Datenstruktur zur Verwaltung des Inhalts der graphischen Oberfläche sein. Darauf gehen wir in diesem Kurs aber nicht ein und werden daher auch in der Regel globalen Variablen verwenden.

Der Bereich einer globalen Variable ist die Datei, in der sie definiert ist. Wenn wir eine solche Variable über die Grenzen einer Datei hinweg in mehreren Dateien benutzen wollen[6], dann müssen wir die Variable an allen weiteren Stellen *deklarieren*, da eine Definition nur genau einmal erfolgen darf, der Typ aber beim übersetzen bekannt sein muss. Das geht mit `extern` in der Form

```
1   extern int x;
```

Auch diese Deklaration muss natürlich wieder vor jeder Verwendung in der Datei erfolgen.

4.4 Anweisungen und Ausdrücke

Am Ende dieses Abschnitts wollen wir noch zwei Begriffe einführen, die bei einer Diskussion der Syntax von `C` immer wieder auftauchen. Wir werden immer wieder von *Anweisungen* und von *Ausdrücken* in `C` sprechen, und haben die Begriffe auch schon benutzt, ohne sie präzise zu beschreiben.

Alles, was `C`, bzw. dann den Prozessor, dazu bringt, tatsächlich eine Operation durchzuführen, ist eine *Anweisung*. Anweisungen in `C` werden durch ein Semikolon abgeschlossen, oder sind eine Abfolge von Anweisungen, die in geschweifte Klammern eingeschlossen sind. In unserer Notation haben wir bisher immer nur genau eine Anweisung in eine Zeile geschrieben, später sehen wir aber auch Anweisungen, die man sinnvoller über mehrere Zeilen hinweg notiert[7]. Alle folgenden Zeilen sind Anweisungen.

[6] Beachten Sie, dass Code in Headern, die mit `#include` eingebunden werden, vor dem Übersetzen in die Datei kopiert werden, eine Definition im Header steht also dort zur Verfügung.

[7] Dem Compiler wäre das aber egal, wie wir schon diskutiert haben. Für diesen ist nur das Semikolon relevant.

```c
int x = 4+5;
x = x+3;
printf("x ist %d\n",x);
{ int y = 1; int z = 2; int yz = y*z; }
```

wobei wir die letzte Zeile eher als

```c
{
  int y = 1;
  int z = 2;
  int yz = y*z;
}
```

notiert werden sollte.

Ein *Ausdruck* in C ist alles, was einen Wert enthält, der in Anweisungen verwendet werden kann, und den wir einer Variablen zuweisen könnten (aber nicht müssen). Ausdrücke können einfache Konstanten sein oder komplexe und lange Ketten von Operationen. Da Ausdrücke einen Wert zurückgeben, können wir Ausdrücke auch als Bausteine in weiteren Ausdrücken benutzen. Alle folgenden Zeilen sind Ausdrücke.

```c
10
x
10 + x
1.2 * ( x + 3 ) * ( y + 5)
y = x = 3
```

Operatoren, die wir im nächsten Kapitel betrachten, erwarten immer Ausdrücke als Operanden. Ausdrücke sind in der Regel Teil von Anweisungen.

Version vom 10. Februar 2022

5 Operatoren

Ein *Operator* modifiziert oder verbindet Werte oder Variablen in einer Programmiersprache. Sie existieren auch in der Mathematik, werden aber oft nicht so benannt oder so deutlich herausgestellt. Zu den bekannten Operatoren zählen die *arithmetischen Operatoren* `+`, `-`, `*`, `/`, wir haben aber schon weitere gesehen, wie zum Beispiel den *Modulo*-Operator `%`, oder die *Vergleichsoperatoren* `==`, `<` und `>`. Wir haben sie bisher auf Variablen oder Konstanten vom Typ `int` angewendet.

5.1 Arithmetische Operatoren

Die *arithmetischen Operatoren* sind die üblichen mathematischen Rechenoperationen wie Addition, Subtraktion, Multiplikation und Division, aber auch der Rest bei ganzzahliger Division. Die Operatoren verknüpfen zwei Werte und geben das Ergebnis der Operation zurück. Sie sind damit eigentlich nur Funktionen, die zwei Parameter annehmen und einen Wert zurückgeben.

Im Unterschied zu Funktionen werden Operatoren aber notiert wie in der Mathematik auch, indem das Symbol des Operators zwischen die Argumente gesetzt wird, also in der Form

Programm 5.1: `kapitel_05/op01.c`

```
1  int summand1 = 3;
2  int summand2 = 4;
3  int summe = summand1 + summand2;
```

statt wie bei Funktionen zum Beispiel in der Form

```
1  int summe = +(summand1,summand2);
```

Die korrekte Schreibweise ist deutlich besser lesbar. Wir wollen den Operator auch auf verschiedene Typen anwenden und zum Beispiel sowohl zwei ganze Zahlen also auch zwei Dezimalzahlen addieren können. Das wäre in `C` mit einer Definition als Funktion nicht möglich, da ein Funktionsname (unabhängig von seinen Argumenten und seines Rückgabetyps) eindeutig sein muss und daher nur entweder für `int` oder `float` definiert werden kann[1].

Operatoren in `C` sind für mehrere Typen definiert und erkennen anhand ihrer Operatoren, welche Variante ausgeführt werden soll. Wir können auch Operanden mit

[1] In vielen anderen Sprachen, unter anderem auch in der Sprache `C++`, die Sie sich im nächsten Semester anschauen, gehören die Parameter und der Rückgabetyp zum Namen der Funktion. Dann können wir den gleichen Funktionsbezeichner mehrfach verwenden, solange sich die Typen oder die Zahl der Parameter unterscheidet.

Version vom 10. Februar 2022

unterschiedlichem Zahltyp mit einem Operator verknüpfen. In diesem Fall konvertiert der Compiler einen oder beide Typen vorher zu einem gemeinsamen Typ und wendet dann den Operator an.

Die genauen Konversionsregeln sind komplizierter, gehen aber prinzipiell von den kleineren zu den größeren Typen. Für die Ganzzahltypen ist die Reihenfolge der Umwandlungen

`char` $\Longrightarrow$ `short` $\Longrightarrow$ `int` $\Longrightarrow$ `long` $\Longrightarrow$ `long long`

und wenn der Typ `unsigned` ist, wird zum nächsten Typ mit Vorzeichen konvertiert, der den Zahlbereich vollständig abdeckt. Wenn Konversion bis zu `long long` nicht ausreicht, oder der andere beteiligte Operand den Typ `float` oder `double` hat, wird in dieser Reihenfolge weiter konvertiert. Bei großen Zahlen, oder wenn eine Konversion zu einem Fließkommatyp notwendig ist, kann hier Information verloren gehen. Daher sollte implizite Konversion von Typen nach Möglichkeit vermieden werden.

Wenn der Typ der Variablen, auf die zugewiesen wird, nicht den Typ der Rückgabe des Operators hat, wird auch hier konvertiert. Je nach den beteiligten Typen kann hier ebenfalls wieder Information verloren gehen oder auch ein völlig falscher Wert herauskommen (z.B. wenn das Ergebnis vom Typ `long` und $> 2^{32}$ ist, aber einer Variablen vom Typ `int` zugewiesen wird). Das kann zu Fehlern führen, die meistens sehr schwer zu finden sind, und sollte daher von Anfang an vermieden werden. Der Compiler gibt hier nicht immer eine Warnung aus, da eine solche Umwandlung nicht immer erkennbar ist.

Bei der Division hängt nicht nur der Typ, sondern auch das Resultat von den Operatoren ab. Wenn beide ganzzahlig sind, wird auch eine ganzzahlige Division mit Rest ausgeführt und der Quotient als ganze Zahl zurückgegeben.

Programm 5.2: `kapitel_05/op02.c`

```
1   int n = 17;
2   int m = 5;
3   int ierg = n/m;  // erg ist 3
4   int rest = n%m; // rest ist 2
5
6   //Achtung: auch bei expliziten Zahlwerten
7   ierg = 17/3; // ergibt 5
```

Wenn jedoch mindestens ein Operand einen Fließkommatyp hat, wird auch das Ergebnis als Dezimalzahl berechnet.

Programm 5.2: `kapitel_05/op02.c`

```
1   float f = 17.; // eine 0 hinter dem Dezimalpunkt kann weggelassen werden
2   int k = 5;
3   float g = 5.;
4
5   float ferg = f/k; // ergibt 3.4
6   ferg = f/g;       // ebenfalls 3.4
7
8   // Achtung: Zahlen ohne Dezimalpunkt sind ganze Zahlen
9   ferg = 17/5;      // ergibt 3.0
10  ferg = 17./5;     // ergibt 3.4
```

Eine Besonderheit ist der Operator `-`, der je nach seiner Stellung die Differenz bildet, oder den Wert der Variablen negiert.

Programm 5.2: kapitel_05/op02.c

```
1  int a = 1;
2  int b = 3;
3  int c = -a;   // Negation: c ist -1
4  int d = a-b;  // Subtraktion: d ist -2
5  int e = 5*-3; // e ist  -15
```

Eine Reihung wie in dem letzten Ausdruck sollte jedoch vermieden werden und die Reihenfolge durch Klammerung deutlich gemacht werden.

5.1.1 Inkrement und Dekrement

Da es sehr oft vorkommt, dass zum Beispiel eine Zählervariable in Schleifen genau um 1 erhöht oder erniedrigt werden muss, gibt es für diese Operation zwei eigene Operatoren, das *Dekrement* `--` und *Inkrement* `++`.

Programm 5.3: kapitel_05/op03.c

```
1  int i = 10;
2  ++i; // entspricht i = i+1
3  --i; // entspricht i = i-1
```

Beide Operatoren geben ihren Wert auch zurück, können also einer Variablen zugewiesen werden.

Programm 5.4: kapitel_05/op04.c

```
1  int i = 10;
2  int j = ++i; // i und j sind 11
```

Der Operator kann auch nachgestellt werden. Das macht nur einen Unterschied, wenn auch zugewiesen wird[2]. Der nachgestellte Operator (Postinkrement oder -dekrement) gibt den Wert vor der Ausführung zurück, der vorangestellte Operator (Prädekrement oder -inkrement) den Wert nach der Ausführung.

Programm 5.5: kapitel_05/op05.c

```
1  int i = 10;
2  int j = ++i; // i und j sind 11
3  int k = i++; // i ist 11 und k ist 10
```

Typischen Anwendungen sind in Schleifen:

Programm 5.6: kapitel_05/op06.c

```
1  for ( int i = 0; i < 10; ++i ) {
2    // i steigt von 0 bis 9
3  }
4
5  int a[5] = {0, 1, 2, 3, 4} ;
6  int i = 0;
7  while ( i < 5 ) { // gibt die Elemente von a der Reihe nach aus
8    printf("%d\n", a[i++]);
9  }
```

[2] Fast. Der vorangestellte Operator ist etwas schneller in der Ausführung.

In der zweiten Schleife wird erst der Wert von `i` an die Auswertung von `a[i]` übergeben, und dann der Wert von `i` um 1 erhöht. In einer Anweisung dürfen Inkrement und Dekrement insgesamt nur einmal auf eine Variable angewendet werden, sonst ist das Ergebnis undefiniert.

Programm 5.7: `kapitel_05/op07.c`

```c
int b = 2;
int s = b++ * b++; // kann 4 oder 6 sein
int t = b-- * b++; // kann 2, 4, oder 6 sein
```

5.2 Zuweisung

Es gibt nur einen Zuweisungsoperator in `C`, den Operator `=`, der der Variablen auf der linken Seite das Ergebnis des Ausdrucks auf der rechten Seite zuweist.

```c
int x = 5*6/3; // x ist 15
```

Zur Vereinfachung des Codes gibt es aber eine Reihe von Verbindungen arithmetischer Operationen mit der Zuweisung. Diese bilden sich aus dem arithmetischen Operator, gefolgt von `=`, also zum Beispiel `+=` um erst den Wert der Variablen links und rechts des Operators zu addieren, und dann der Variablen auf der linken Seite zuzuweisen.

Bei diesem Operator darf, wie auch beim Zuweisungsoperator selbst, nur eine Variable stehen, deren Wert verändert werden kann (insbesondere darf die Variable nicht `const` sein).

```c
int x = 5;
int y = 2;

x += 8; // x ist 13
x *= y; // x ist 26
y /= 1; // y ist 1
x %= 3; // x ist 2
```

5.3 Vergleichsoperatoren

In `C` stehen uns auch die üblichen Vergleichsoperatoren wie $<$ oder $\geq$ zur Verfügung. Da es keinen Variablentyp in `C` gibt, der einen Wahrheitswert speichert, geben alle Vergleichsoperatoren einen Wert vom Typ `int` zurück. Alle Anweisungen, die eine Bedingung auswerten, wie z.B. `while(...)`, interpretieren 0 als *falsch* und alle von 0 verschiedenen Werte als *wahr*. Wir haben die folgenden Vergleichsmöglichkeiten:

```c
int x = 7;
int y = 14;

x == y; // Gleichheit, gibt einen von 0 verschiedenen Wert zurueck, meistens 1
x != y; // Ungleichheit, gibt 0 zurueck
x < y;  // kleiner als
x <= y; // kleiner als oder gleich
x > y;  // groesser als
x >= y; // groesser als oder gleich
```

5.4 Logische Verknüpfungen

Mit `&&` für *und*, `||` für *oder* und `!` für die Negation stehen uns drei logische Operatoren zur Verfügung, mit denen wir die Ergebnisse von Vergleichen verbinden können.

```
int x = 5;

if ( x >= 2 && x <= 7 ) {
  // Anweisungen werden nur ausgefuehrt, wenn x zwischen 2 und 7 liegt
}

int y = 2;

if ( !(x < y || x > 2*y) ) {
  // Anweisungen, nur wenn y < x < 2*y
}
```

5.5 Ein ternärer Operator

Bisher waren alle Operatoren *unär*, hatten also genau einen Operanden, wie z.B. `-` als Negation oder `!`, oder *binär*. Es gibt einen einzigen *ternären* Operator, der manchmal nützlich ist:

```
int y = 7;

int x = y == 7 ? 2 : 4; // x wird 2
int z = y != 7 ? 2 : 4; // z wird 4
```

Der Operator `?:` wertet seinen ersten Operator als Bedingung aus und führt entweder Ausdruck des zweiten Operators aus, wenn die Bedingung wahr (also nicht 0) ist, und sonst den Ausdruck im dritten Operator. Ein ähnliches Ergebnis erzielen wir mit

```
int x;
if ( y == 7 ) {
  x = 4;
}  else {
  x = 2;
}
```

Den ternären Operator können wir allerdings direkt in einer Zuweisung verwenden, während wir bei der Konstruktion mit `if` erst die Variable `x` deklarieren mussten.

5.6 Kommaoperator

Wir können auch eine Reihe von Ausdrücken mit Kommata aneinanderreihen. Dabei wird der Wert des letzten Ausdrucks zurückgegeben. Meistens wird dieser jedoch nicht verwendet, da es beim Durchlesen des Programms leicht zu übersehen ist, wo die Kommata sind (und so möglicherweise Fehler übersehen werden).

Eine Stelle, bei der der Kommaoperator häufig eingesetzt wird, sind Stellen, an denen ein einzelner Ausdruck erlaubt ist, wir aber eine Kette von solchen auszuführen. Ein

Version vom 10. Februar 2022

typisches Beispiel ist die Schleife mit `for`, wenn wir am Ende eines Durchlaufs mehr als eine Variable für den nächsten Durchlauf anpassen wollen. So schreibt

```
int j = 5:
for ( int i = 0; i < 5; ++i, --j ) {
  printf("%d - %d\n",i,j);
}
```

die Zahlen

```
0 - 5
1 - 4
2 - 3
3 - 4
4 - 3
```

auf den Bildschirm.

5.7 Reihenfolge

Da Operatoren immer einen Wert zurückgeben, können wir auch das Ergebnis von Operatoren wieder in solche einsetzen. Bei den arithmetischen Operatoren hätten wir darüber wahrscheinlich nicht nachgedacht, und sofort angenommen, dass ein Ausdruck der Form

```
int x = (4 + 4*5 + 4*3) / 6;
```

korrekt zu 5 auswertet. Das funktioniert allerdings nur, da die Operatoren `*`, `+` und `/` ihr Ergebnis zurückgeben. Außerdem beachtet `C` an dieser Stelle die üblichen Vorschriften, welche Operation zuerst ausgeführt werden soll, und geht nicht der Reihe nach vor und gibt 22 zurück.

Aber auch alle anderen Operatoren können verknüpft werden. Das geschieht neben den arithmetischen Operatoren besonders häufig noch mit Vergleichen und logischen Operatoren.

```
if ( (x == y) && ( a == b) ) {
  // Anweisungen, falls x und y sowie a und b gleich sind
}
```

Hier muss allerdings, wie auch schon bei den arithmetischen Operatoren, die Reihenfolge der Ausführung beachtet werden. Schon in dem Beispiel sind um `x==y` und `a==b` Klammern gesetzt, da sonst zuerst `y && a` ausgewertet würde (Beachten Sie, dass Wahrheitswerte nur ganze Zahlen sind, `y&&a` gibt also 1 zurück, wenn sowohl `y` als auch `a` nicht 0 sind, und 0 sonst).

Die Ausführungsreihenfolge entspricht im wesentlichen dem, was man aus der Mathematik kennt und auch intuitiv erwarten würde. Die genaue Reihenfolge steht in Table 5.1. Sie finden dort auch einige Operatoren, die wir erst in späteren Kapiteln kennenlernen werden. Operatoren mit der gleichen Priorität in der Tabelle sind gleichwertig, und wenn mehrere davon hintereinander kommen, gibt die Auswertungsreihenfolge an, ob die Kette von rechts oder von links abgearbeitet wird. In den meisten Fällen entspricht das der Reihenfolge, in der wir auch intuitiv die Ausdrücke interpretieren würden. Das folgende Programm

Priorität	Operator	Beschreibung	Auswertung

1	`()`
	`[]`
	`++`
	`--`
	`.`
	`->`
		Funktionsaufruf
		Listenelementzugriff
		Postinkrement
		Postdekrement
		Elemente von `struct`
		Elemente von `struct` über Zeiger
			von links nach rechts

2	`+,-`
	`!`
	`++`
	`--`
	`sizeof`
	`(type)`
	`*`
	`&`
		Vorzeichen
		logische Negation
		Präinkrement
		Prädekrement
		Speichergröße
		Typumwandlung
		Dereferenzierung
		Adressoperator
			von rechts nach links

3	`*, /, %`	Multiplikation, Division, Rest	von links nach rechts
4	`+,-`	Addition, Subtraktion	von links nach rechts
5	`<, <=, >, >=`	Vergleiche	von links nach rechts
6	`==, !=`	Gleichheit	von links nach rechts
7	`&&`	logisches und	von links nach rechts
8	`||`	logisches oder	von links nach rechts
9	`?:`	Konditionaloperator	von rechts nach links

10	`=`
	`+=, -=`
	`*=, /=, %=`
		Zuweisung	von rechts nach links

11	`,`	Komma	von links nach rechts

Tabelle 5.1: Auswertungsreihenfolge der Operatoren. Die Operatoren zur Typumwandlung, zur Dereferenzierung, für den Elementzugriff in Strukturen und den Adressoperator lernen wir erst später kennen.

Programm 5.8: `kapitel_05/op08.c`

```
1  #include <stdio.h>
2
3  int main() {
4
5    int x;
6    int y = x = 2+3-4;
7    int z = 4;
8    if ( !--y ) {
9      printf("y ist %d\n",y);
10   }
11
```

```
12    if ( x == 1 || y != 1 && z == 4 || x*y == 1 ) {
13      printf("Die Bedingung ist wahr\n");
14    }
15
16    if ( z = --x == y ) {
17      printf("z ist %d\n",z);
18    }
19
20    return 0;
21  }
```

schreibt

```
y ist 0
Die Bedingung ist wahr
z ist 1
```

auf den Bildschirm. Durch Klammerung von Ausdrücken können wir die Reihenfolge beeinflussen. Wie in der Mathematik werden Ausdrücke in Klammern zuerst ausgewertet. Im Zweifel sollten sie immer Klammern setzen um die von Ihnen gewünschte Reihenfolge festzulegen. Das erleichtert in der Regel das Lesen komplizierterer Ausdrücke erheblich. Die meisten Compiler werden auch bei dem Beispielprogramm Warnungen anzeigen, die vorschlagen, die Reihenfolge durch Klammerung explizit zu machen und auf die Zuweisung innerhalb der Bedingung hinweisen, da die Verwechslung von = und == eine der häufigsten Fehler weniger geübter Programmier*innen sind.

Es gibt noch einige weitere Operatoren, wie den *Adressoperator* und den *Dereferenzierungsoperator*, sowie die *Verschiebeoperatoren*. Diese schauen wir uns an, wenn wir sie brauchen.

6 Zeiger

Der Inhalt jeder Variablen muss vom Programm im Speicher des Computers abgelegt werden. Mit jeder neuen Variablen, die Sie in Ihrem Programm definieren, fordert das Programm einen neuen Bereich passender Größe im Speicher an und weist diesen der Variablen zu.[1] Auf diesen Speicherort einer Variablen können wir in unseren Programmen zugreifen und ihn auch verändern. Von dieser Möglichkeit wird in C sehr oft Gebrauch gemacht, da es zum einen zu sehr effizienten Programmen führen kann und es zum anderen auch mache Dinge gibt, die sich ohne diesen Zugriff auf den Speicher nicht sinnvoll programmieren lassen.

6.1 Zeiger und Adressen

Aus der Sicht eines C-Programms ist der Speicher des Computers linear organisiert und in Abschnitte eingeteilt, die jeweils 1 Byte groß sind. Diese Abschnitte sind (auf- oder absteigend) durchnummeriert, und werden in der Regel der Reihe nach vom System an das Programm ausgegeben. Die an die einzelnen Abschnitte vergebene Nummer wird als *Adresse* des Speichers bezeichnet.

In C können wir uns diese Adresse anzeigen lassen, die Adresse in einer Variablen speichern und auf den Inhalt einer Speicheradresse direkt zugreifen. Wir werden sehen, dass dies eine sehr nützliche Methode ist, um auf Variablen zuzugreifen, Werte an Funktionen zu übergeben und eigene Datenstrukturen anzulegen.

Dafür gibt es zwei neue Operatoren, den *Adressoperator* & und den *Referenzoperator* ∗. Beide sind unär. Der erste gibt die Adresse der Variablen zurück, auf den er angewandt wird, der zweite den Inhalt der Adresse, auf die er angewandt wird. Um mit Adressen arbeiten zu können, müssen wir sie in einer Variablen speichern können.

Der Platzbedarf der verschiedenen Datentypen im Speicher ist unterschiedlich und meistens größer als ein einzelner Speicherabschnitt, der ein *Byte* oder acht *Bit* umfasst. Eine Variable vom Typ `char` benötigt genau ein Byte, aber alle weiteren Typen wie `short`, `int`, `long` und `double` benötigen schon vier oder acht Byte. Wir können uns das mit dem Operator `sizeof` anzeigen lassen, der sowohl einen Datentyp als auch eine konkrete Variable als Argument akzeptiert und die Speichergröße in Byte ausgibt. Hier ist ein Beispiel.

Programm 6.1: `kapitel_06/sizeof.c`

```
1 #include <stdio.h>
2
```

[1] Später sehen wir auch, wie wir selbst im Programm Speicher anfordern können um komplexere Datenstrukturen nach unseren Wünschen im Speicher organisieren zu können.

Version vom 10. Februar 2022

```c
int main() {

    printf("Die Speichergroesse von int ist %lu.\n",sizeof(int) );
    printf("Die Speichergroesse von char ist %lu.\n",sizeof(char) );
    printf("Die Speichergroesse von double ist %lu.\n",sizeof(double) );

    long x = 4;
    printf("Die Speichergroesse von x ist %lu.\n",sizeof(x) );

    return 0;
}
```

Die Ausgabe könnte wie folgt aussehen, die genauen Größen hängen aber vom Prozessor und Compiler ab.

```
Die Speichergroesse von int ist 4 Byte.
Die Speichergroesse von char ist 1 Byte.
Die Speichergroesse von double ist 8 Byte.
Die Speichergroesse von x ist 8 Byte.
```

Da wir beim Auslesen des Speichers in de Regel die ganze Variable und nicht den Inhalt einer einzigen Speicherzelle sehen wollen, müssen wir beim Auslesen des Speichers wissen, wie viele Speicherzellen nach der, auf die unser Adressvariable zeigt, noch relevant sind. Daher gibt es für die Speicherung einer Adresse zu jedem Datentyp einen eigenen *Adresstyp*, den wir durch Nachstellen eines `*` erzeugen. Variablen, die eine Adresse enthalten, werden meistens *Zeiger* (oder *pointer*) genannt. Hier ist ein Beispiel, in dem ein Zeiger auf eine Variable vom Typ `int` erzeugt wird.

Programm 6.2: `kapitel_06/zeiger_01.c`

```c
#include <stdio.h>

  int main() {

    int x = 5;
    int* p = &x;

    printf("An der Adresse %p ist der Wert %d gespeichert.\n", p, *p);

    return 0;
  }
```

Die Ausgabe sieht ungefähr aus wie

```
An der Adresse 0x7ffedfd61438 ist der Wert 5 gespeichert.
```

Die Adresse wird in der Regel bei jedem Aufruf aber eine andere sein. In dem Programm haben wir einen neuen Platzhalter `%p` für Zeigervariablen verwendet. Die Adressen sind hexadezimal nummeriert, was wir an dem führenden `0x` erkennen, als Dezimalzahl ist die Adresse `140.732.653.769.784`. Je nach Prozessorarchitektur sind Adressen eine Zahl im Bereich zwischen 0 und 2^{32} oder 2^{64}, wobei die niedrigen Adressen in der Regel vom System belegt sind.

Der Stern nach `int` muss nicht direkt neben dem Typ stehen, und kann an die Variable gezogen werden oder frei dazwischen stehen. Die Definitionen

```c
  int * x;
  int* y;
```

```
3   int *z;
```

ergeben also alle eine Zeigervariable auf einen `int`-Wert. Hier können Sie sich aussuchen, was Ihnen sinnvoller erscheint. Auch in der Literatur oder verschiedenen Projekten hat sich an der Stelle kein allgemeiner Standard durchgesetzt. Allerdings erzeugt

```
1   int * x, y;
```

nicht zwei Zeiger, sondern einen Zeiger `x` und eine Variable `y` vom Typ `int`, der `*` bindet also zur Variablen, nicht zum Typ.

Wir können über eine Zeigervariable auch den an der Adresse gespeicherten Inhalt ändern.

Programm 6.3: `kapitel_06/zeiger_02.c`

```
1  #include <stdio.h>
2
3  int main() {
4    int a = 5;
5    int *x = &a;
6
7    *x = 7;
8
9    printf("a ist %d\n",a);
10
11   return 0;
12 }
```

Zudem können wir Zeiger benutzen, um Variablen so an Funktionen zu übergeben, dass wir den Wert der Variablen auch in der Funktion verändern können.

Programm 6.4: `kapitel_06/zeiger_referenz.c`

```
1  #include <stdio.h>
2
3  void change(int* p) {
4    p++;
5  }
6
7  int main() {
8
9    int a = 1;
10
11   change(&a);             // wir uebergeben die Adressse von a
12   printf("a ist %d\n",a); // gibt "a ist 2" aus
13
14   return 0;
15 }
```

Diese Form der Parameterübergabe wird oft als *Übergabe einer Referenz* (*pass by reference*) bezeichnet, im Gegensatz zur *Übergabe des Werts* (*pass by value*), was wir bisher gemacht haben. Mit dieser Methode können wir auch mehr als eine Rückgabe aus einer Funktion erhalten, in dem wir ihr Zeiger auf Adressbereiche übergeben, die sie mit den Rückgabewerten füllen kann. Wir haben das schon in der Funktion **`scanf`** verwendet, die so viele Variablenadressen in ihrer Parameterliste übernimmt, wie die Zeichenkette im ersten Argument vorgibt. An dieser Stelle verstehen wir dann auch, warum wir im

Aufruf von `scanf` alle Variablennamen mit einem `&` versehen mussten, damit `scanf` in der Lage ist, die Variablen tatsächlich mit den eingegebenen Werten zu belegen.

Auch Zeiger sind natürlich Variablen, die ihren Wert, die Adresse, irgendwo im Speicher ablegen müssen. Wir können uns auch davon die Adresse geben lassen, und in einer Zeigervariablen speichern. Wenn wir von einen `int` ausgehen, hat dieser Zeiger den Typ `int **`. Eine einfache Dereferenzierung ergibt dann eine Zeigervariable auf einen Wert, und erst eine zweite Dereferenzierung ergibt den Wert selbst.

Programm 6.5: `kapitel_06/zeiger_auf_zeiger.c`

```c
#include <stdio.h>

int main() {
  int x = 3;

  int *  p = &x;
  int ** q = &p;

  printf("x ist %d\n",**q);

  int y = x* **q;
  printf("y ist %d\n",y);

  return 0;
}
```

Wir können diesen Prozess natürlich auch weiter iterieren, aber während Zeiger auf Zeigervariablen noch häufig Anwendung finden, kommen weitere Iterationen dieses Prozesses eher selten vor.

Eine typische Anwendung ist die Übergabe der Adresse eines Zeigers an eine Funktion, in der wir den Ort, auf den eine Zeigervariable zeigt, verändern wollen. Hier ist ein Beispiel, in dem wir einen Zeiger auf die größere von zwei Zahlen setzen[2].

Programm 6.6: `kapitel_06/zeiger_aendern.c`

```c
#include <stdio.h>

void max(int ** p, int * a, int * b) {
  if ( *a < *b ) {
    *p = b;
  }  else {
    *p = a;
  }
}

int main() {
  int x = 3;
  int y = 5;
  int * p;

  max(&p,&x,&y);
  printf("Das Maximum von %d und %d ist %d\n", x, y, *p);

```

[2] Dieses Problem könnte man leichter auf andere Weise lösen, wir sehen später Beispiele, wo die Übergabe der Adresse eines Zeigers tatsächlich die eleganteste Lösung ist.

```
19    return 0;
20  }
```

Der Multiplikationsoperator $*$ hat das gleiche Symbol wie die Dereferenzierung. Auf den ersten Blick könnte man annehmen, dass das zu Problemen führen könnte. Allerdings ist der Multiplikationsoperator binär und erwartet vor und hinter sich Variablen oder Konstanten von einem kompatiblen Typ, während die Dereferenzierung unär ist. Zudem lassen sich nur Werte von einem der Zahltypen miteinander multiplizieren. Da ein Zeigertyp niemals kompatibel ist zu einem Zahltyp, kann der Compiler immer entscheiden, ob ein $*$ dazu dient, den nachfolgenden Ausdruck zu dereferenzieren oder ob es eine Multiplikation mit dem vorangegangenen Ausdruck sein soll. Die folgenden drei Definitionen von `z1`, `z2` und `z3` sind also gleichwertig und weisen der Variablen `9` zu.

Programm 6.7: `kapitel_06/deref_mult.c`

```
1  int main() {
2    int x = 3;
3    int *  p = &x;
4    int ** q = &p;
5
6    int z1 = x * **q;
7    int z2 = x * * *q;
8    int z3 = x * * * q;
9
10   return 0;
11 }
```

Leichter verständlich wird der Code aber, wenn wir trotzdem Klammern setzen und das Produkt in der Form

Programm 6.7: `kapitel_06/deref_mult.c`

```
1    int z4 = x * (**q);
```

schreiben.

6.2 Arrays und Zeigerarithmetik

Jetzt können wir auch auflösen, warum es bei der Parameterübergabe an Funktionen so aussieht, als würden Listen nicht kopiert, sondern stünden in der Funktion direkt zur Verfügung. Aus der Sicht von `C` sind Listenvariablen ohne nachfolgendes Klammerpaar `[]` tatsächlich im wesentlichen Zeigervariablen auf das erste Element in der Liste. Das Programm

Programm 6.8: `kapitel_06/liste_zeiger.c`

```
1  #include <stdio.h>
2
3  int main() {
4
5    int liste[5] = {1,2,3,4,5} ;
6    printf("Das erste Element der Liste ist %d\n",liste[0]);
7    printf("Das erste Element der Liste ist %d\n",*liste);
```

```
8
9
10    return 0;
11  }
```

gibt

```
Das erste Element der Liste ist 1
Das erste Element der Liste ist 1
```

auf dem Bildschirm aus. Die Unterschiede zwischen Listen- und Zeigervariablen betrachten wir später, und schauen uns zuerst die Möglichkeiten an, die sich aus diesem Zusammenhang ergeben.

Wenn wir über

```
1    int liste[5];
```

eine Listenvariable definieren, wird ein ausreichend großer Bereich im Speicher dafür reserviert. Dieser Speicher ist immer zusammenhängend, die Elemente der Liste liegen also nacheinander im Speicher.

Zeigervariablen kennen den Typ der Variablen, auf die sie zeigen. Dadurch wissen sie auch, wie viel Speicherplatz sie belegen, und der Compiler kann bestimmen, wo die Adresse des nächsten Bereichs im Speicher ist. Auf diese Weise funktioniert der Operator `[]`, der ein Element der Liste zurückgibt. Die Listenvariable zeigt auf die Adresse des ersten Elements, und der Zeiger springt so viele Adressen weiter, wie der Index, multipliziert mit der Größe eines einzelnen Elements, angibt. Wir können das auch selbst ausnutzen:

Programm 6.9: `kapitel_06/zeigerarithmetik.c`

```
1  int main() {
2
3    int liste[5] = {1,2,3,4,5} ;
4    int x = liste[3]; // x ist 4
5    int y = *(liste+3); // ebenfalls 4
6
7    int * p = liste+3; // zeigt auf das vierte Element der Liste
8
9    return 0;
10 }
```

Diese Möglichkeit, Zeiger durch Addition (oder Subtraktion) auf einen anderen Speicherbereich zeigen zu lassen, nennt man *Zeigerarithmetik*.

Zeiger und Listen sind auch bei der Übergabe als Parameter an eine Funktion austauschbar, und wir können als Typ des übergebenen Parameters auch einen Zeiger vorsehen.

Programm 6.10: `kapitel_06/liste_referenz.c`

```
1  int func(int * l, int n) {
2    int s = 0;
3    for(int i = 0; i < n; ++i ) {
4      s += *(l+i);
5    }
6    return s;
7  }
```

```
8
9  int main() {
10   int liste[] = {1,2,3,4,5} ;
11
12   int s = func(liste,5); // s ist 15
13
14   return 0;
15 }
```

Die interne Betrachtung einer Listenvariablen als Zeiger auf das erste Element erklärt nun auch, warum es auf den ersten Blick so aussieht, als ob Listenvariablen anders als andere Typen bei der Übergabe an eine Funktion nicht kopiert wird. Tatsächlich wird auch hier kopiert, aber nur die Variable, die die Adresse des ersten Elements enthält. Original und Kopie zeigen daher auf den gleichen Speicherbereich, und mit Dereferenzierung lesen oder ändern wir den gleichen Wert im Speicher.

Wir können uns das über die Anzeige der Speicheradresse ansehen. Das Programm

Programm 6.11: `kapitel_06/liste_referenz_02.c`

```
1  #include <stdio.h>
2
3  void func(int * l, int n) {
4    for ( int i = 0; i < n; ++i ) {
5      printf("[func] Listenelement %d ist an der Adresse %p\n",i,l+i);
6    }
7  }
8
9  int main() {
10   int liste[3] = {1,2,3} ;
11
12   for ( int i = 0; i < 3; ++i ) {
13     printf("[main] Listenelement %d ist an der Adresse %p\n",i,liste[2]+i);
14   }
15
16   func(liste,3);
17
18   return 0;
19 }
```

schreibt

```
[main] Listenelement 0 ist an der Adresse 0x7ffeea2b945c
[main] Listenelement 1 ist an der Adresse 0x7ffeea2b9460
[main] Listenelement 2 ist an der Adresse 0x7ffeea2b9464
[func] Listenelement 0 ist an der Adresse 0x7ffeea2b945c
[func] Listenelement 1 ist an der Adresse 0x7ffeea2b9460
[func] Listenelement 2 ist an der Adresse 0x7ffeea2b9464
```

auf den Bildschirm (die konkreten Adressen könnten bei Ihnen natürlich andere sein).

Bei allen Gemeinsamkeiten von Zeigern und Listen gibt es allerdings auch zwei Unterschiede in der Behandlung von Listen- und Zeigervariablen. Eine Listenvariable kennt neben der Adresse des ersten Elements auch die Länge des ihr zugewiesenen Speicherbereichs. Daher kann zwar einer Zeigervariablen eine Listenvariable zugewiesen werden (und verliert dabei die Längeninformation), aber nicht umgekehrt, da die Längeninformation fehlt. Der Operator `sizeof` liefert bei einer Listenvariablen nicht die Speichergröße des Zeigers, sonder der Liste zurück. Das können wir benutzen, um die

Länge einer Liste zu bestimmen, indem wir den für sie reservierten Speicher durch die Speicherlänge eines einzelnen Elements teilen.

Programm 6.12: kapitel_06/liste_sizeof.c

```c
#include <stdio.h>

int main() {
  int liste[3] = {1,2,322} ;
  int * l = liste;

  printf("liste zeigt auf einen Speicherbereich der Laenge %lu, l nur auf %lu\n",sizeof
    (liste), sizeof(l));
  printf("Die Liste hat %lu Elemente\n",sizeof(liste)/sizeof(liste[0]));

  return 0;
}
```

Bei Zeigern müssen wir an manchen Stellen auf die Auswertungsreihenfolge von Operatoren achten. Wie wir in Table 5.1 sehen, bindet z.B. das Postinkrement ++ stärker als der Dereferenzierungsoperator $*$. Das müssen wir beachten, wenn wir den Wert einer Variablen über einen Zeiger erhöhen wollen.

Programm 6.13: kapitel_06/liste_dereference.c

```c
  int x = 1;
  int *p = &x;

  // *p++; // verschiebt erst den Adresszeiger um die Laenge
           // eines int und gibt den Inhalt dieses Speicherbereichs aus
           // dieser Bereich muss nicht initialisiert sein!

  (*p)++;  // x ist 2
```

Da auch der Listenzugriff mit [] Vorrang vor der Dereferenzierung hat, ergibt aber

Programm 6.13: kapitel_06/liste_dereference.c

```c
  int x = 1;
  int y = 2;
  int * liste[2] = {&x, &y} ;
  int z = *liste[1]; // z ist 2
  printf("z ist %d\n",z);
```

das erwartete Ergebnis.

Bei der Übergabe einer Liste an eine Funktion sind die folgenden drei Definitionen des übergebenen Parameters gleichwertig.

```c
  // liste ist nur ein Zeiger auf das erste Element
  void f1(int * l, int laenge) {
    //Anweisungen
  }

  void f2(int liste[5], int laenge) {
    //Anweisungen
  }

  void f3(int liste[], int laenge) {
    //Anweisungen
```

```
12    }
13
14  int main() {
15    int liste[] = {1,2,3,4,5} ;
16
17    f1(liste,5);
18    f2(liste,5);
19    f3(liste,5);
20
21    return 0;
22  }
```

Allerdings könnten wir bei den letzten beiden Funktionsdefinitionen die Größe der Liste noch über `sizeof` bestimmen. Es ist in der Regel aber einfacher und führt zu lesbarerem Code, wenn wir die Länge explizit übergeben (falls sie nicht durch ein Präprozessormakro festgelegt ist).

6.3 Initialisierung und die Adresse `NULL`

Eine Zeigervariable kann eine Speicheradresse enthalten. Bei ihrer Definition wird aber kein Speicherbereich reserviert, auf den sie zeigen könnte, oder wo sie ihr zugewiesene Werte ablegen könnte. Der folgende Code führt daher zu einem Fehler.

```
1    int * p;
2    *p = 5;
```

Wir sehen später, wie wir zum Zeitpunkt der Definition einer Zeigervariablen auch einen Speicherbereich reservieren können, auf den sie zeigt. Bis dahin können wir in Zeigervariablen nur Adressen von schon existierenden Variablen speichern.

```
1    int * p;
2    int x;
3
4    p = &x;
5    *p = 5;
```

Wenn wir, wie wir das in der Regel bei Variablen vom Typ `int` oder `float` gemacht haben, Zeigervariablen direkt bei ihrer Definition einen Wert zuweisen wollen, haben wir zwei Optionen. Wir können natürlich direkt die Adresse einer bestehenden Variablen zuweisen. Wenn diese noch nicht definiert ist, oder wenn wir später dynamisch entscheiden wollen, welche Adresse zugewiesen wird, können wir die spezielle Adresse `NULL` verwenden. Dort kann nichts gespeichert werden. Allerdings können wir `NULL` in Vergleichen verwenden um zu testen, ob eine Zeigervariable auf eine sinnvolle Adresse zeigt. Die Adresse `NULL` ist in der Bibliothek `stdio.h` definiert, die daher eingebunden sein muss.

Programm 6.14: `kapitel_06/zeiger_null.c`

```
1  #include <stdio.h>
2
3  int main() {
4
5    int * p = NULL;
6    printf("p zeigt auf %p\n",p);
7
```

```
8     int x = 4;
9     if ( p == NULL ) {
10      p = &x;
11    }
12
13    printf("p zeigt jetzt auf %p\n",p);
14
15    return 0;
16  }
```

Diese Art der Initialisierung oder Zuweisung einer Zeigervariablen auf `NULL` wird oft verwendet, um bei der Rückgabe aus einer Funktion anzuzeigen, dass eine Aktion innerhalb der Funktion nicht erfolgreich war. Das könnte zum Beispiel der Zugriff auf eine Datei sein, oder die Abfrage eines Werts vom Benutzer, der nicht gültig war.

6.4 `const`-Zeiger und Zeiger auf `const`

Wir haben schon gesehen, dass wir Variablen mit dem Vorsatz `const` vor den Typ unveränderlich machen können. Auf diese Weise können wir auch einen Zeiger auf einen unveränderlichen Wert zeigen lassen.

```
1   int main() {
2     const int x = 4;
3     const int * p =&x;
4
5     *p = 5; // Fehler, p kann den Wert, auf den es zeigt, nicht veraendern
6
7     return 0;
8   }
```

Wir könnten anstatt des Werts aber auch verhindern wollen, dass p auf eine andere Adresse zeigen kann, dass also die Zeigervariable selbst unveränderlich wird. Dafür müssen wir das Schlüsselwort `const` nach dem $*$ in der Definition einfügen.

Programm 6.15: `kapitel_06/zeiger_const_01.c`

```
1  int main() {
2
3    int x = 4;
4    int y = 5;
5    int * const p1 = &x;
6    *p1 = 6;  // jetzt ist x == 6
7    p1 = &y;  // Fehler, p kann nicht auf eine andere Adresse zeigen
8
9    const int z = 7;
10   const int * const p2 = &z;
11   p2 = &x; // Fehler, p kann nicht auf eine andere Adresse zeigen
12   *p2 = 8; // Fehler, Wert an der Adresse p2 unveraenderlich
13
14   return 0;
15 }
```

Beachten Sie jedoch, dass das Schlüsselwort `const` nur die Veränderung durch die Variable, bei der es angegeben wird, verhindert wird. Wenn wir einen Zeiger auf die

gleiche Adresse haben, der veränderlich ist (oder umgekehrt), können wir den Wert an der Adresse doch verändern.

Programm 6.16: `kapitel_06/zeiger_const_02.c`

```c
int main() {
  int x = 5;
  const int * p1 = &x;
  *p1 = 6; // Fehler, *p1 unveraenderlich
  x = 7; // jetzt ergibt *p1 ebenfalls 7

  const int y = 8;
  int * p2 = &y; // erzeugt eine Warnung, keinen Fehler
  *p2 = 9; // jetzt ist y == 9

  return 0;
}
```

6.5 Mehrdimensionale Listen

Wir können auch Listen vo Listen definieren, indem wir bei der Definition mehrere Längen in eckigen Klammern angeben.

```c
  int matrix[3][2];
  matrix[1][1] = 4;
```

Eine solche mehrdimensionale Liste wird im Speicher zeilenweise hintereinander abgelegt. Bei unserem Beispiel steht also das Element an der Position $(1, 1)$ in der Matrix an der vierten Stelle des Speicherbereichs. Auch hier können wir wieder mit Zeigern und Zeigerarithmetik arbeiten, der Code

```c
  int matrix[3][2];
  *(matrix+3) = 4;
```

ist identisch zum vorangegangen. Allgemein berechnet man einen Index im Speicher mit

```c
  int n = 3;
  int m = 2;
  int matrix[n][m];
  int i = 1;
  int j = 1;

  *(matrix+i*m+j) = 4; // aendert wieder das Element an Position (1,1)
```

Bei der Übergabe an eine Funktion können wir jedoch nicht mehr alle Dimensionen in der Parameterliste weglassen, da ein mehrdimensionaler Array linear im Speicher liegt und `C` daher wissen muss, wann die nächste Zeile beginnt.

Programm 6.17: `kapitel_06/mehrdimensionale_listen.c`

```c
void f1(int n, int m, int (*mat)[m]) {
    // Anweisungen
  }

  void f2(int n, int m, int mat[n][m]) {
```

```
6       // Anweisungen
7     }
8
9     void f3(int n, int m, int mat[][m]) {
10      // Anweisungen
11    }
12
13
14  int sum(int a, int b) {
15    return a+b;
16  }
17
18    int main() {
19      int n = 3;
20      int m = 2;
21      int mat[n][m];
22
23      int (*fp)(int,int) = &sum;
24
25      int x = (*fp)(2,3);
26
27      f1(n,m,mat);
28      f2(n,m,mat);
29      f3(n,m,mat);
30
31      return 0;
32  }
```

Eine spezielle Form einer solchen mehrdimensionalen Liste haben wir schon gesehen, als wir uns die Variante der Funktionsdefinition von **main** angesehen haben, die zwei Argumente akzeptiert. Hier war das zweite Argument vom Typ `char**`, was wir jetzt als eine Listenvariable für eine Liste von Zeichenketten identifizieren können. Wir hätten an der Stelle daher die zweite Variable als `char*[]` definieren können.

6.6 Funktionszeiger und Callbacks

Wir können nicht nur Zeiger auf Variablen bzw. den Speicherbereich von Variablen definieren, sondern auch Zeiger, die auf Funktionen zeigen. Diese Zeiger können wir dann, wie andere Zeiger auch, kopieren oder auf diese Weise an andere Funktionen übergeben.

Die letztgenannte Anwendungsmöglichkeit dürfte auch die häufigste sein, und wird uns in dieser Vorlesung öfter begegnen. Die Definition eines solchen *Funktionszeigers* ist wahrscheinlich gewöhnungsbedürftig. Wir müssen in der Typbezeichnung mehrere Informationen unterbringen, da zur Definition einer Funktion der Rückgabetyp und die Reihenfolge und Typen aller Parameter gehört. Die Definition eines Funktionszeigers imitiert daher die Deklaration eines Funktionskopfes, bei der aber statt des Funktionsnamens ein Variablenname mit einem vorangestellten `*` steht, und nur die Typen der Parameter, aber keine Namen auftauchen. Formal sieht eine solche Definition daher so aus:

```
1   <Rueckgabetyp> (*<Variablenname>) ( Parametertyp1, ...)
```

Das folgende Beispiel sollte die Definition verdeutlichen.

Programm 6.18: `kapitel_06/funktionszeiger.c`

```
1  int sum(int a, int b) {
2    return a+b;
3  }
4
5  int main() {
6    int (*op) (int, int);
7    op = &sum;
8
9    int summe = (*op)(4,5); // summe ist 9
10
11   return 0;
12 }
```

Beachten Sie dabei, dass die dereferenzierte Zeigervariable `*op` in Klammern stehen muss, da die nachfolgenden Klammern um die Parameter sonst sowohl in der Definition als auch in der Anwendung zuerst ausgewertet würden. In der Definition führt das direkt zu einem Syntaxfehler, in der Anwendung würde `C` nach einer Funktion `op` suchen, die zwei `int`s als Parameter nimmt und eine Adresse eines `int` zuückgibt, die dann mit `*` dereferenziert wird.

Wir können solche Zeiger auch an Funktionen übergeben. Auf diese Weise können wir Programmteile, die sich zum Beispiel nur in der Berechnung eines Teilergebnisses unterscheiden, zusammenfassen.

Hier ist ein Beispiel, in dem wir einer Funktion `calc` die arithmetische Operation, die sie ausführen soll, übergeben.

Programm 6.19: `kapitel_06/funktionszeigeruebergabe.c`

```
1  #include <stdio.h>
2
3  int sum(int a, int b) {
4    return a+b;
5  }
6
7  int mul(int a, int b) {
8    return a*b;
9  }
10
11 void calc(int a, int b, int (*op)(int,int) ) {
12   printf("Die Berechnung ergibt %d\n", (*op)(a,b));
13 }
14
15 int main() {
16
17   calc(2,3,&sum);
18   calc(2,3,&mul);
19
20   return 0;
21 }
```

Seine Bildschirmausgabe ist

```
Die Berechnung ergibt 5
Die Berechnung ergibt 6
```

Eine interessante Anwendung, die sehr häufig vorkommt, ist die Übergabe von Vergleichsfunktionen an eine Funktion, die eine Liste sortiert oder andere Operationen auf einer Liste ausführt. Eine Funktion, die eine Liste sortiert, braucht ein Kriterium um für zwei Werte zu entscheiden, welcher davon in der sortierten Liste zuerst kommen soll. Schon für Listen mit ganzen Zahlen ist es je nach Anwendung wünschenswert, die Liste auf- oder absteigend zu sortieren. Wenn wir direkt in der Sortierfunktion den Operator < für den Vergleich benutzen, dann kann unsere Funktion nur aufsteigend sortieren, und wir müssen eine zweite Funktion schreiben, die absteigend sortiert. Wir können diese Verdopplung vermeiden, indem wir, statt < oder > direkt zu verwenden, die Übergabe einer Funktion vorsehen, die das Ergebnis des Vergleichs zurückgibt, und dann je nach Anwendung eine passende Vergleichsfunktion übergeben.

Noch interessanter wird das, wenn wir komplexere Daten haben. Sein Adressbuch könnte man zum Beispiel nach Vornamen, Nachnamen oder Wohnorten sortieren wollen.

In einem späteren Kapitel schauen wir uns solche Vergleichsfunktionen und Algorithmen für die Sortierung von Daten genauer an.

7 Testen

Software ist selten fehlerfrei, auch nach mehreren Verbesserungszyklen. Bei nachfolgenden Verbesserungen, zum Beispiel durch effizientere Algorithmen oder Modularisierung, können auch neue Fehler entstehen, oder alte erst sichtbar werden, weil Funktionen auf andere Weise, an anderer Stelle oder mit anderen Parametern benutzt werden. Ein größerer Anteil der Zeit, die für die Entwicklung eines neues Programms oder Weiterentwicklung eines bestehenden Programms verwendet wird, wird daher Fehlersuche und die Absicherung eines einmal erreichten korrekten Zustands der einzelnen Teile des Programms sein.

Die Frage, wie man effizient und koordiniert nach Fehlern sucht und verhindert, bei Anpassungen des Codes neue Fehler einzubauen, wird uns in diesem Kurs mehrfach, und Sie, wenn Sie weiterhin programmieren, immer wieder, beschäftigen. Wir wollen in diesem Kapitel drei erste Ideen vorstellen, wie wir hier systematisch vorgehen können.

Dabei geht es im ersten Abschnitt um eine einfache Idee zur Fehlersuche. Unsere Möglichkeiten hierzu werden wir in einem späteren Kapitel, unter anderem mit der Betrachtung von *Debuggern*, sehr viel eingehender diskutieren.

In den weiteren Abschnitten wollen wir uns damit beschäftigen, wie wir einen einmal erreichten Zustand absichern können. Dafür betrachten wir zum einen die Funktion `assert`, mit der wir im Code überprüfen können, ob eine Bedingung erfüllt ist, und andernfalls das Programm mit einer Fehlermeldung sofort beenden. Das ist oft nützlich, wenn wir in einem Abschnitt unseres Codes, zum Beispiel einer Funktion, Annahmen an den Zustand des Programms, zum Beispiel an die Werte von Variablen, machen, die wir im normalen Programmlauf nicht überprüfen können oder wollen. Es könnte zum Beispiel sein, dass die Überprüfung zeitaufwendig ist, und wir sie daher nur durchführen wollen, wenn wir unser Programm testen, oder wenn wir einen Fehler vermuten.

Im letzten Abschnitt betrachten wir dann eine Möglichkeit, von unserem Programm unabhängige sogenannte *unit tests* zu schreiben, mit denen wir einzelne Teile unseres Programms immer wieder überprüfen können. Jeder einzelne *unit test* überprüft dabei eine Funktion oder eine Familie zusammengehöriger Funktionen. In dem Test legen wir für eine Reihe von Eingaben für die Funktion die von uns erwartete Ausgabe fest, und wenn wir den Test aufrufen, wird die Funktion für diese Eingaben nacheinander ausgeführt und die Übereinstimmung mit unserer Erwartung überprüft und das Ergebnis ausgegeben. Auf diese Weise können wir nach jeder Änderung an einer Funktion testen, ob sie auch nach der Änderung noch die von uns erwarteten Ergebnisse liefert. Bei sinnvoller Auswahl der zu testenden Werte erhalten wir auf diese Weise eine gute Kontrolle, ob wir durch unsere Änderungen neue Fehler im Code gemacht haben oder die Funktion immer noch korrekt arbeitet.

Wir betrachten dabei hier nur eine sehr einfache Version solcher *unit tests*. In großen Softwareprojekten werden oft viele und komplexe Testszenarien entwickelt, die regel-

Version vom 10. Februar 2022

mäßig, meistens nach jeder einzelnen Änderung, aufgerufen werden, um Abweichungen möglichst sofort zu erkennen und dadurch leicht eingrenzen zu können, wo ein möglicher Fehler im Code zu suchen ist. Moderne Versionskontrollsysteme machen dies besonders einfach, da man zum Beispiel beim Einchecken jeder Codeänderung automatisch einen Durchlauf der *unit tests* auslösen kann. Je nach Einstellung kann man auf diese Weise eine Änderung im Code auch direkt verwerfen und den Autor informieren, dass seine Codeänderung noch Fehler enthalten muss.

7.1 Einfache Tests mit `printf`

Eine der einfachsten Möglichkeiten, nach Fehlern zu suchen oder den Programmablauf zu verfolgen, ist die Idee, an relevanten Stellen im Code eine Statusmeldung mit **`printf`** zu machen. Dabei können auch Parameterwerte oder Zwischenergebnisse ausgegeben werden, die wir dann mit unserer Erwartung vergleichen können.

Wir können das natürlich an allen Stellen machen, wo wir einen Fehler vermuten oder wo wir den Programmlauf kontrollieren wollen. Wenn wir das gesamte Programm überprüfen wollen, ist es oft sinnvoll, solche Ausgaben am Anfang und Ende jeder Funktion zu machen, um so nachvollziehen zu können, wie der Programmablauf in unserem Code war.

Damit wir die Ausgaben nur während des Testens sehen und nicht für eine Veröffentlichung des Programms alle Ausgaben aus dem Code löschen und später ggf. wieder einfügen müssen, bietet es sich an, solche Ausgaben über eine Anweisung an den Präprozessor ein- und ausschalten zu können. Dafür kann man sich mit

```
1 #define DEBUG
```

am Anfang der Datei, die die **`main`**-Funktion enthält, eine Variable definieren, und anschließend jede Ausgabe mit

```
1 #ifdef DEBUG
2 printf("Eine Kontrollausgabe im Programm\n");
3 #endif
```

zu umgeben. Wenn wir dann die Definition der Variablen `DEBUG` löschen oder auskommentieren, wird `gcc` (bzw. genauer der Präprozessor) alle in `#ifdef DEBUG ... #endif` eingeschlossenen Zeilen vor dem Übersetzen in Maschinencode vollständig löschen. Wir haben also in diesem Fall auch keinen möglichen Effizienzverlust durch Berechnungen, die wir innerhalb eines solchen Blocks für die Kontrollausgaben gemacht haben.

Wenn wir unsere Kontrollausgaben über eine solche Präprozessorvariable ausschalten, müssen wir natürlich darauf achten, dass wir innerhalb der Ausgaben den Zustand unseres Programms nicht verändern. Wir dürfen die Werte von Variablen nicht verändern, und keine neuen Variablen oder Funktionen definieren, die auch ausserhalb der Kontrollausgaben zur Verfügung stehen sollen.

Um leicht den Namen der Funktion, in der wir gerade sind, ausgeben zu können, gibt es in `C` die automatisch definierte Variable `__func__`, die den aktuellen Funktionsnamen enthält. Wir können unsere Ausgabe damit zum Beispiel auf die folgende Weise schreiben.

```
1 #ifdef DEBUG
```

```
printf("[%s] Eine Kontrollausgabe im Programm\n",__func__);
#endif
```

Damit wäre, wenn diese Zeile in einer Funktion `my_function` steht, die Ausgabe

```
[my_function] Eine Kontrollausgabe im Programm
```

Wenn wir das für unser übliches Programm zu den Collatzfolgen machen, könnte das dann wie in Programm 7.1 aussehen.

Programm 7.1: `kapitel_07/collatz_v1.h`

```
unsigned int collatz_laenge(unsigned int c) {
#ifdef DEBUG
    printf("[%s] Parameter c ist %u\n",__func__,c);
#endif
    unsigned int laenge = 0;
    while ( c != 1 ) {
        if ( c % 2 == 0 ) {
#ifdef DEBUG
            printf("[%s] c=%u ist gerade\n",__func__,c);
#endif
             c = c/2;
        }  else {
#ifdef DEBUG
            printf("[%s] c=%u ist ungerade\n",__func__,c);
#endif
             c = 3*c+1;
        }
#ifdef DEBUG
        printf("[%s] Lange wird erhoeht\n",__func__);
#endif
        laenge = laenge + 1;
    }
    return laenge;
}
```

Programm 7.1:`kapitel_07/collatz_main_v1.c`

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

int main() {
    unsigned int startwert = 0;
    printf("Geben Sie eine positive ganze Zahl ein: ");
    scanf("%d",&startwert);

    if ( startwert < 1 ) {
        printf("Die Eingabe %d muss >= 1 sein\n",startwert);
        return 0;
    }

    int laenge = collatz_laenge(startwert);
    printf("Die Collatz-Folge mit Start %u hat Laenge %u\n",startwert,laenge);

    return 0;
}
```

Wenn wir nicht auf der Suche nach Fehlern sind, sondern vorerst vermuten oder annehmen, dass unser Code korrekt arbeitet und wir uns davor schützen wollen, durch

Anpassungen, Erweiterungen, oder (dann nicht erfolgreiche) Verbesserungen neue Fehler im Code zu machen, können wir stattdessen unsere Funktion regelmäßig auf einer Reihe von Werten ausprobieren, für die wir die korrekte Antwort kennen. Um das umzusetzen, könnten wir uns zum Beispiel ein zweites, nur für den Test gedachtes Programm schreiben, das die zu testenden Funktionen über einen Header einbindet und dann für einige Werte aufruft. Für unsere Collatzfolgen wäre Programm 7.2 eine konkrete Möglichkeit für eine Umsetzung.

Programm 7.2: `kapitel_07/collatz_v2.h`

```
1  unsigned int collatz_laenge(unsigned int c) {
2      unsigned int laenge = 0;
3      while ( c != 1 ) {
4          if ( c % 2 == 0 ) {
5              c = c/2;
6          }  else {
7              c = 3*c+1;
8          }
9          laenge = laenge + 1;
10     }
11     return laenge;
12 }
```

Programm 7.2:`kapitel_07/collatz_main_v2.c`

```
1  #include <stdio.h>
2  #include "collatz.h"
3
4  void test(int param, int expected) {
5      if ( collatz_laenge(param) != expected ) {
6          printf("Test failed: collatz_laenge(%d) sollte %d sein, %d erhalten\n", param,
       expected, collatz_laenge(param) );
7      }  else {
8          printf("Test passed: collatz_laenge(%d)\n",param);
9      }
10 }
11
12 int main() {
13
14     test(5,5);
15     test(23,15);
16     test(7,16);
17
18     return 0;
19 }
```

In dem Testprogramm haben wir eine Funktion `void test(int param, int expected )` implementiert, die unsere Funktion `collatz_laenge` mit dem Parameter `param` aufruft und das Ergebnis mit dem Wert, der in `expected` übergeben wurde, vergleicht. Wir rufen diese Funktion dann für drei Werte auf, bei denen wir die zugehörige Collatzlänge kennen. Diese dürfen wir dann natürlich nicht vorher mit unserem Programm berechnet haben, denn wenn dieses schon fehlerhaft ist, verglichen wir an dieser Stelle mit dem falschen Wert und nähmen bei erfolgreichen zukünftigen Tests zu unrecht an, dass unser Programm korrekt ist.

Dieser Ansatz ist schon sehr nahe an dem heute oft benutzten Ansatz über *unit tests*. In unserer Testfunktion mussten wir hier noch einigen Code selbst und auf die konkret

zu testende Funktion angepasst schreiben. Wir werden im folgenden sehen, dass es Hilfsmittel gibt, die die Erstellung solcher Tests vereinfachen.

7.2 `assert`

Eine Möglichkeit, im Programmablauf Annahmen oder Erwartungen an Variablen oder Funktionen zu prüfen, bietet die Einbindung des Headers `assert.h` aus der C-Standardbibliothek. Hierin wird im wesentlichen nur die Funktion `assert(...)` definiert, die eine Bedingung überprüft und das Programm abbricht, wenn die Bedingung nicht erfüllt ist. Wenn die Bedingung erfüllt ist, läuft das Programm ohne eine Ausgabe durch `assert` weiter. Programm 7.3 zeigt ein einfaches Beispiel.

Programm 7.3: `kapitel_07/simple_assert.c`

```c
#include <assert.h>

int main() {
    int x = 2;
    assert(x == 2);

    return 0;
}
```

Da die Bedingung erfüllt ist, erhalten wir von dem Programm keine Ausgabe. Wenn wir jedoch den Aufruf von `assert(x==2)` durch `assert(x==5)` ersetzen, bricht das Programm mit einer Meldung ähnlich zu der folgenden ab.

```
Assertion failed: (x == 5), function main, file simple_assert.c, line 20.
[1]    87532 abort      ./simple_assert
```

Wir können solche Überprüfungen an beliebigen Stellen im Code unterbringen. Für unsere Collatzfolge könnte das also wie in Programm 7.4 aussehen.

Programm 7.4: `kapitel_07/collatz_v3.h`

```c
#include <assert.h>

unsigned int collatz_laenge(unsigned int c) {

    assert(c > 0);

    unsigned int laenge = 0;
    while ( c != 1 ) {

        assert( c > 1 );

        if ( c % 2 == 0 ) {

            assert(c % 2 == 0 );

            c = c/2;
        }  else {

            assert(c % 2 == 1 );

```

```
21              c = 3*c+1;
22          }
23          laenge = laenge + 1;
24      }
25      return laenge;
26  }
```

Programm 7.4:kapitel_07/collatz_main_v3.c

```
1  #include <stdio.h>
2  #include "collatz_assert.h"
3
4  int main() {
5      unsigned int startwert = 0;
6      printf("Geben Sie eine positive ganze Zahl ein: ");
7      scanf("%d",&startwert);
8
9      if ( startwert < 1 ) {
10         printf("Die Eingabe %d muss >= 1 sein\n",startwert);
11         return 0;
12     }
13
14     int laenge = collatz_laenge(startwert);
15     printf("Die Collatz-Folge mit Start %u hat Laenge %u\n",startwert,laenge);
16
17     return 0;
18 }
```

Wenn wir das Programm veröffentlichen oder in einer Anwendung einsetzen wollen, wollen wir möglicherweise die Überprüfungen mit `assert()` ausschalten, da die Auswertung der Bedingung Rechenzeit kostet. Das können wir seht leicht erreichen, indem wir in unserer Quelldatei mit

```
1  #define NDEBUG
```

die Variable `NDEBUG` definieren. Dann wird `gcc`, bzw. genauer der Präprozessor, alle Aufrufe von `assert()` vor dem Übersetzen des Codes entfernen.

Solche Aufrufe von `assert()` werden in Programmen oft dann eingesetzt, wenn wir bei der Implementierung implizite Annahmen an die Parameter eine Funktion machen, diese aber nicht testen wollen. Das ist dann besonders sinnvoll, wenn eine Methode oder Algorithmus theoretisch für eine größere Menge von Eingaben korrekt funktionierte, wir ihn aber nur für eine Teilmenge der möglichen Eingaben benötigen und wir, in dem wir dieses Wissen ausnutzen, in der Umsetzung Zeit sparen können. Wenn wir später doch die Menge der möglichen Eingaben vergrößern wollen und uns nicht mehr an unsere Annahme erinnern, dann wird das Programm möglicherweise falsch. Mit einem `assert()` könnten wir unsere Einschränkungen in Tests überprüfen, ohne in der finalen Version Laufzeitverluste zu haben.

Wir sollten immer darauf achten, dass unsere Bedingungen keine Nebeneffekte haben und zum Beispiel die Werte von Variablen verändern. Code der Form

```
1  int x = 2;
2  assert( 2 == x++ );
```

ist zulässig. Da aber der Wert von `x` hier verändert wird, wird unserer Programm je nachdem, ob die Variable `NDEBUG` gesetzt ist oder nicht, unterschiedliche Ergebnisse

liefern. Aber auch ohne den Einsatz von `NDEBUG` ist es schlechter Stil, in einem Aufruf von `assert()` den Zustand des Programms zu ändern. Es sollte immer möglich sein, alle Aufrufe von `assert()` aus dem Code zu entfernen.

7.3 Unit Tests

Mit Programm 7.2 haben wir eine Möglichkeit kennengelernt, unsere Funktionen regelmäßig daraufhin zu überprüfen, ob sie zu einer Auswahl vorgegebener Werte noch die erwarteten Ergebnisse liefern. Allerdings ist es eine eher aufwendige Methode, da wir für jeden Test ein neues Programm schreiben und entsprechende Checks und passende Kontrollausgaben programmieren müssen. Die Tests mit `assert()` waren einfacher, decken aber nicht die gleiche Anwendung ab.

Es sind daher einige Projekte in `C` entstanden, die diese Lücke füllen und es ermöglichen wollen, sehr einfach eine umfangreiche Sammlung von Tests zu erstellen, die alle Teile eines Programms abdecken können. Dabei soll jeder Test nur eine kleine Einheit, zum Beispiel eine einzelne Funktion, abdecken und unabhängig von den anderen Funktionen im Programm oder den anderen Tests funktionieren. Solche Tests werden oft *unit tests* genannt.

Wir wollen hier mit `Unity` ein Projekt vorstellen, mit der wir leicht einfache Tests für unsere Programme erstellen können, dass aber mit größerem Wissen auch für komplexere Programme einsetzbar ist. Sie können den `C`-Code des Projekts unter www.throwtheswitch.org/unity zum Beispiel als `zip`-gepackte Datei herunterladen. Sie können es an beliebiger Stelle entpacken. Wir müssen später einen Header und eine Quelldatei aus dem Projekt in unsere Tests einbinden.

Für einen Test mit `Unity` sollten wir alle Funktionen, die wir in einer gemeinsamen Einheit (der *unit*) testen wollen, in einem gemeinsamen Header mit zugehöriger Quelldatei zusammenfassen. Für unseren Test schreiben wir dann ein neues Programm, dass diesen Header einbindet und die Tests ausführt. Wir schauen uns mit Programm 7.5 direkt ein Beispiel für unsere Collatzfolgen an. Dabei haben wir den gleichen Header wir schon in Programm 4.2. Wir wollen für den Test nicht den Code unseres Programms anpassen müssen oder dort Anweisungen unterbringen, die nur für den Test relevant sind. Der eigentliche Test steht nur in der zugehörigen Quelldatei, die diesen Header einbindet, und dann die Tests definiert. Dafür müssen wir hier zusätzlich den Header `unity.h` aus `Unity` einbinden.

Die Tests folgen anschließend alle dem gleichen Schema. In der `main`-Funktion rufen wir zwischen den Funktionsaufrufen `UNITY_BEGIN()` und `UNITY_END` mit der Funktion `RUN_TEST` der Reihe nach unsere Tests auf. Diese bestehen aus normalen Funktionen, die weder einen Parameter annehmen noch zurückgeben dürfen. Hier können wir beliebigen Code zur Vorbereitung unterbringen. Die eigentlichen Tests bestehen dann aus Aufrufen von speziellen, von `Unity` bereitgestellten Funktionen, die alle mit `TEST_ASSERT_` beginnen, und dann, je nach Variante der Funktion, einen geeigneten Test durchführen.

Programm 7.5: `kapitel_07/collatz.h`

```
1 unsigned int collatz_laenge(unsigned int c) {
2     unsigned int laenge = 0;
3     while ( c != 1 ) {
```

```
4           if ( c % 2 == 0 ) {
5               c = c/2;
6           }  else {
7               c = 3*c+1;
8           }
9           laenge = laenge + 1;
10      }
11      return laenge;
12  }
```

Programm 7.5:kapitel_07/collatz_test.c

```
1   #include "collatz.h"
2   #include "../unity/src/unity.h"
3
4   void setUp () {}
5   void tearDown () {}
6
7   void test_collatz_laenge_2() {
8       TEST_ASSERT_EQUAL_INT( 5, collatz_laenge(6) ); // der Test wird fehlschlagen
9   }
10
11  void test_collatz_laenge() {
12      TEST_ASSERT_EQUAL_INT( 5, collatz_laenge(5) );
13      TEST_ASSERT_EQUAL_INT( 15, collatz_laenge(23) );
14  }
15
16  int main () {
17      UNITY_BEGIN();
18      RUN_TEST(test_collatz_laenge);
19      //RUN_TEST(test_collatz_laenge_2);
20      return UNITY_END();
21  }
```

Im Beispiel haben wir die Funktion `TEST_ASSERT_EQUAL_INT` benutzt, die zwei Argumente annimmt und ihre Gleichheit als ganze Zahlen überprüft. Das ist ähnlich zu `assert()`, allerdings können die Funktionen aus `Unity` eine besser organisierte Übersicht über erfolgreiche und fehlgeschlagene Tests abgeben.

Wir können unseren Test mit

```
gcc collatz_test.c ../unity/src/unity.c -o collatz_test
```

übersetzen. Dabei müssen Sie natürlich den Pfad zu `unity.c` daran anpassen, wo Sie die Dateien von `Unity` entpackt haben. Ein Programmlauf ergibt dann zum Beispiel

```
collatz_test.c:29:test_collatz_laenge:PASS

-----------------------
1 Tests 0 Failures 0 Ignored
OK
```

Wenn wir jedoch die auskommentierte Zeile hinzunehmen, wird auch der zweite Test ausgeführt, der fehlschlagen sollte, da die Länge der Collatzfolge zu 6 die Länge 8 hat. Und tatsächlich erhalten wir die Ausgabe

```
collatz_test.c:33:test_collatz_laenge:PASS
collatz_test.c:23:test_collatz_laenge_2:FAIL: Expected 5 Was 8
```

```
-----------------------
2 Tests 1 Failures 0 Ignored
FAIL
```

Die einfachste Form eines Tests sind die beiden Funktionen

```
TEST_ASSERT_TRUE(<Bedingung>)
TEST_ASSERT_FALSE(<Bedingung>)
```

die ähnlich wie `assert()` eine Bedingung überprüfen. Den Test

```
TEST_ASSERT_EQUAL_INT(<Erwartung>, <Ergebnis>)
```

haben wir oben schon gesehen. Mit

```
TEST_ASSERT_INT_WITHIN(<Bereich>,<Ergebnis>)
TEST_ASSERT_GREATER_THAN(<Schranke>,<Ergebnis>)
TEST_ASSERT_LESS_THAN(<Schranke>,<Ergebnis>)
```

können Sie einen Bereich oder obere bzw. untere Schranken testen. Wenn Sie an den Funktionsnamen am Ende noch `_MESSAGE` anhängen, also zum Beispiel die Funktion `TEST_ASSERT_EQUAL_INT_MESSAGE` verwenden, dann können Sie als letztes Argument zusätzlich eine Zeichenkette übergeben, die bei Scheitern des Tests ausgegeben wird. Ganze Listen können wir leicht mit

```
TEST_ASSERT_EQUAL_INT_ARRAY(<Erwartung>, <Ergebnis>, <Leange der Liste>)
```

überprüfen. Diese FUnktionen funktionieren, anders als ihr Name vielleicht nahelegt, auch für Variablen vom Typ `long` und `short`. Wenn die Variablen mit `unsigned` definiert sind, dann müssen wir in den Name der Tests `_INT_` durch `_UINT_` ersetzen. Wenn uns die Kontrolle der Länge der Variablen wichtig ist, gibt es Funktionen, die statt `_INT_` zum Beispiel `_INT32_` heissen. Das wird für uns keine Rolle spielen. Zum Vergleichen von `char` können sie statt `_INT_` die entsprechenden Funktionen mit `_CHAR_` verwenden.

Neben den eigentlichen Testfunktionen erwartet `Unity` immer, dass zwei Funktionen

```
void setUp () {}
void tearDown () {}
```

definiert sind, die wie die Testfunktionen keine Parameter annehmen dürfen und keinen Wert zurückgeben. Sie können hier jedoch Code in die Funktion schreiben. Die erste wir vor allen Tests, die zweite nach den Tests aufgerufen.

Mit unseren bisherigen Kenntnissen werden Sie wahrscheinlich keine Anwendung für diese Funktionen haben. Das wird sich ändern, wenn wir später komplexe Datentypen definieren und dafür die Speicherverwaltung implementieren müssen. Bis dahin müssen Sie nur daran denken, die beiden leeren Funktionsdefinitionen in alle Testprogramme zu übernehmen.

Wenn wir in unseren Tests Fließkommazahlen verwenden, ist es, wie wir schon gesehen haben, problematisch, die *Gleichheit* von solchen zu überprüfen, da sich durch Rundungsfehler und die begrenze Anzahl von Nachkommastellen Abweichungen vom exakten Wert ergeben können. Bei unseren Tests sollten wir also nicht die Gleichheit testen, sondern prüfen, ob die Abweichung von der Erwartung innerhalb eines von uns vorgegebenen sinnvollen Bereichs liegt. Das können wir zum Beispiel mit der Testfuntkion

```
TEST_ASSERT_FLOAT_WITHIN(<Toleranz>, <Erwartung>, <Ergebnis>)
```

machen. In Programm 7.6 sehen Sie ein Beispiel für einen solchen Test. Dabei wird in der Funktion `approx_sin` der Sinus von `arg` über die Reihendarstellung

$$\sin(x) = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

approximiert, indem wir die unendliche Summe nach den ersten `n` Summanden abbrechen. Sie wissen aus der Verlesung zur Analysis, dass diese endliche Reihe für wachsendes n gegen $\sin(x)$ konvergiert und die Größe des Restterms

$$\delta(n) = \sum_{i=n+1}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

gegen 0 konvergiert. Wir bekommen also tatsächlich eine Näherung für $\sin(x)$, die mit wachsendem n besser wird.

Programm 7.6: `kapitel_07/approx_sin.h`

```
1  #include <stdlib.h>
2
3  float approx_sin( float arg, int n ) {
4
5     float approx_sin, summand;
6     int k;
7
8     // Summand fuer i=0
9     approx_sin = arg;
10    summand = arg;
11    for ( k=1; k<n; k=k+1 ) {
12       summand = summand
13           * (-1)*arg*arg/(2*k*(2*k+1));
14       approx_sin = approx_sin + summand;
15    }
16
17    return approx_sin;
18 }
```

Programm 7.6:`kapitel_07/approx_sin_test.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "approx_sin.h"
4
5  #include "../unity/src/unity.h"
6
7  float delta = 0.0001;
8
9  void setUp () {
10 //    delta = .01;
11 }
12
13 void tearDown () {}
14
15 void test_approx_sin() {
16     TEST_ASSERT_FLOAT_WITHIN(delta, 0, approx_sin(3.14,10));
17
```

```
}

int main () {
    UNITY_BEGIN();
    RUN_TEST(test_approx_sin);
    return UNITY_END();
}
```

Weitere Testfunktionen finden Sie in der Dokumentation zu `Unity`, zum Beispiel unter der Adresse github.com/ThrowTheSwitch/Unity#readme oder github.com/ThrowTheSwitch/Unity/blob/master/docs/UnityAssertionsReference.md.

Bei dem Entwurf von geeigneten Tests gibt es einige Fallstricke, die Sie nach und nach kennenlernen werden. Auf ein mögliches Problem wollen wir hier noch eingehen. Dafür betrachten wir Programm 7.7, das zwei Header mit den zugehörigen Quelldateien enthält. Wir übersetzen es mit

```
gcc fake_example.c fake_example_1.c fake_example_2.c -o fake_example
```

Programm 7.7: `kapitel_07/fake_example.c`

```
#include <stdio.h>
#include "fake_example_1.h"
#include "fake_example_2.h"

int main() {
    printf("Das Ergebnis ist %d\n", f2(5) );
    return 0;
}
```

Programm 7.7:`kapitel_07/fake_example_1.h`

```
#ifndef FAKE_EXAMPLE_1_H
#define FAKE_EXAMPLE_1_H

int f1 (int x);

#endif
```

Programm 7.7:`kapitel_07/fake_example_1.c`

```
int f1 (int x) {
    return 2*x;
}
```

Programm 7.7:`kapitel_07/fake_example_2.h`

```
#ifndef FAKE_EXAMPLE_2_H
#define FAKE_EXAMPLE_2_H

#include "fake_example_1.h"

int f2 ( int x);

#endif
```

Programm 7.7:`kapitel_07/fake_example_2.c`

```c
#include "fake_example_2.h"
#include "fake_example_1.h"

int f2 ( int x) {
    return 3*f1(x);
}
```

Wir wollen die Funktionen in den beiden Headern getrennt testen. Das können wir prinzipiell mit den Programm 7.8 und Programm 7.9 erreichen, die wir mit

```
gcc fake_example_test_1.c fake_example_1.c ../unity/src/unity.c -o
    fake_example_test_1
gcc fake_example_test_2.c fake_example_2.c fake_example_1.c ../unity/src/unity.c -o
     fake_example_test_2
```

übersetzen können.

Programm 7.8: `kapitel_07/fake_example_test_1.c`

```c
#include <stdio.h>
#include "fake_example_1.h"

#include "../unity/src/unity.h"

void setUp () {}
void tearDown () {}

void test_f1() {
    TEST_ASSERT_EQUAL_INT( 10, f1(5) );
}

int main () {
    UNITY_BEGIN();
    RUN_TEST(test_f1);
    return UNITY_END();
}
```

Programm 7.9: `kapitel_07/fake_example_test_2.c`

```c
#include <stdio.h>
#include "fake_example_2.h"

#include "../unity/src/unity.h"

void setUp () {}
void tearDown () {}

void test_f2() {
    TEST_ASSERT_EQUAL_INT( 30, f2(5) );
}

int main () {
    UNITY_BEGIN();
    RUN_TEST(test_f2);
    return UNITY_END();
}
```

Bei dem zweiten Test haben wir jedoch das Problem, dass er vom ersten Header abhängt, und es daher sein kann, dass ein Fehlschlagen des zweiten Tests keinen Fehler

in der Funktion im zweiten Header, sondern einen Fehler der Funktionen im ersten Header anzeigt. Unser Ziel war es jedoch, für alle Teile *unabhängige* Tests zu schreiben. Um das zu erreichen, können wir als Ersatz für die Funktion aus dem ersten Header eine spezielle Variante für unsere Tests schreiben, die ohne Rechnung nur genau die Ergebnisse zurückliefert, die wir für den Test brauchen. Das könnte in unserem Fall wie in Programm 7.10 aussehen. Wir übersetzen dann mit

```
gcc fake_example_test_2.c fake_example_2.c fake_example_1_fake.c ../unity/src/unity.
    c -o fake_example_test_2
```

Programm 7.10: `kapitel_07/fake_example_1_fake.c`

```
1 #include <stdio.h>
2
3 int f1(int x) {
4     if ( x == 5 ) {
5         printf("[%s] returning fake value\n",__func__);
6         return 10;
7     }
8     return 0;
9 }
```

Hier ist der Test der zweiten Funktion nun unabhängig geworden von der konkreten Ausgestaltung der ersten Funktion. Natürlich werden solche Techniken erst richtig sinnvoll, wenn die Funktionen, die wir testen wollen komplizierter sind, und ggf. auch noch weitere Abhängigkeiten haben, so dass es ohne eine solche spezielle Funktion nicht mehr leicht ersichtlich ist, wo nun ein Fehler im Code vorliegen muss.

Header, bzw. die Implementierung der Funktionen als vereinfachte Version, die nur die im Test erwarteten Ergebnisse liefert, nennt man *Mock*- oder *Fake*-Funktionen, oder *Stubs*. In komplexeren Testszenarios wird zwischen diesen drei Begriffen noch feinere Unterschiede gemacht, je nachdem wie genau hier der restliche Code, der in diesem Test nicht getestet werden soll, von dem Code, der getestet werden soll, abgekapselt wird. Wir gehen aber hier nicht weiter auf diese Unterschiede ein und wollen unter allen drei Begriffen das gleiche verstehen.

Version vom 10. Februar 2022

8 Ein- und Ausgabe

Die Ein- und Ausgabe ist wesentlicher Bestandteil fast jedes Programms, da wir meistens nicht nicht auf einem fest vorgegebenen Satz von Daten rechnen wollen, sondern unser Programm für variable Eingaben benutzen wollen. Hier wollen wir uns die Möglichkeiten dafür ansehen. Wir fangen mit der Ein- und Ausgabe auf den Bildschirm an, bevor wir unsere Möglichkeiten auf die Ein- und Ausgabe in Dateien erweitern. Dabei werden wir auch sehen, dass die Ein- und Ausgabe über den Bildschirm nur ein Spezialfall ist, denn Das Betriebssystem sieht Lesen von Terminaleingaben und Schreiben ins Terminal als Dateioperationen auf zwei speziellen Dateien *stdout* und *stdin*, die vom System in jedem Programm automatisch eingebunden werden.

Bei Dateien müssen wir noch zusätzlich danach unterscheiden, ob wir eine Datei *lesen* oder *schreiben* wollen, und im zweiten Fall auch, ob wir sie *überschreiben* wollen. Gleichzeitiges Lesen und Schreiben sind zwar prinzipiell möglich, aber schwierig zu realisieren und sollte in der Regel vermieden werden. Ein Zugriff auf eine Datei erfolgt in `C` über einen Zeiger, der *sequentiell* den Dateiinhalt lesen kann. Wir können den Inhalt einer Datei immer nur an der Stelle lesen oder schreiben, wo dieser Zeiger aktuell hinzeigt.

8.1 Bildschirmaus- und -eingabe

In unseren Programmen haben wir schon fast immer Eingabe von Daten über das Terminal und Ausgabe von Daten auf dem Bildschirm vorgesehen, über die Funktionen **`scanf`** und **`printf`**. In diesem Abschnitt wollen wir uns das noch einmal systematischer ansehen. Außerdem lernen wir mit **`getchar`** noch eine Funktion kennen, die einzelne Zeichen vom Bildschirm liest.

8.1.1 Ausgabe

Wir haben schon gesehen, dass wir mit der Anweisung **`printf`** aus der Standardbibliothek `stdio.h` Text auf dem Bildschirm ausgeben können. Die Funktion nimmt als erstes Argument eine Zeichenkette, die auch *Platzhalter* und *Steuerzeichen* enthalten kann. Alle nachfolgenden Argumente werden der Reihe nach in die Platzhalter eingesetzt. Dabei müssen Platzhalter und Datentyp zusammenpassen.

Die Funktion gibt, wenn die Schreiboperation erfolgreich war, die Anzahl der ausgegebenen Zeichen (als `int`) zurück, und eine negative Zahl, wenn die Funktion nicht erfolgreich war. Diese Rückgabe wird aber in den meisten Programmen nicht aufgefangen und ausgewertet.

Version vom 10. Februar 2022

Typkürzel	Typ
d oder i	`signed int`
u	`unsigned int`
o	`unsigned int` oktal
x	`unsigned int` hexadezimal
X	`unsigned int` hexadezimal mit Großbuchstaben
f/F	`float`
e	`float` oder `double` in Exponentenschreibweise mit kleinem `e`
E	`float` oder `double` in Exponentenschreibweise mit großem `e`
g/G	`float` oder `double` in `e` oder `f`, je nachdem, was kürzer ist, mit kleinem/großem e
a/A	`float` oder `double` hexadezimal, mit kleinem/großem e
c	`char`
s	`char*` Zeichenkette
p	Adressen
n	speichert die bisher geschriebene Anzahl in der entsprechenden Variable (vom Typ `unsigned int`)
%	`%%` schreibt ein `%`

Tabelle 8.1: Typangaben im Platzhalter

	d oder i	u, o, x, X
hh	`signed char`	`unsigned char`
h	`short`	`unsigned short`
l	`long`	`unsigned short`
ll	`long long`	`unsigned long long`
z	**`size_t`**	**`size_t`**
t	`ptrdiff_t`	`ptrdiff_t`

Tabelle 8.2: Spezifizierung der Länge ganzzahliger Typen

Über Platzhalter in der Zeichenkette können wir Daten ausgeben und die Ausgabe formatieren. Alle Platzhalter beginnen mit `%` und enthalten einen Buchstaben, der den Typ der auszugebenden Variablen angibt. Table 8.1 gibt eine Übersicht. Für die abgeleiteten Typen wie `long int` brauchen wir zwischen `%` und dem Buchstaben noch eine Spezifizierung der Länge. Für die ganzzahligen Typen stehen diese in Table 8.2. Damit ergibt sich für die Ausgabe eines `long` z.B. `%ld` als Platzhalter. Beachten Sie, dass hier auch `char` wieder auftaucht. mit dem Platzhalter `%c` wird das zugehörige Zeichen ausgegeben, mit `%hhd` die Zahl. Für Dezimalzahlen gibt es noch `%Ld` zur Ausgabe eines `long double`. Zwischen `%` und Typ (ggf. mit Länge) können noch weitere Angaben stehen. Es gibt insgesamt noch drei Angaben, die möglich sind, aber nicht alle müssen auftauchen. Wenn mehrere angegeben werden, müssen sie aber in der Reihenfolge kommen, in der wir sie hier auflisten.

- An der ersten Stelle können ein oder mehrere der folgenden Angaben stehen
  - Mit einem `-` wird die Ausgabe linksbündig statt rechtsbündig gemacht. Das hat nur eine Auswirkung, wenn in nächsten Bereich auch eine Breite für die Ausgabe angegeben wird.

  - Mit einem `+` wird vor nichtnegativen Zahlen ein + ausgegeben. Ein - vor negativen Zahlen wird natürlich immer ausgegeben.
  - Mit einem Leerzeichen wird bei nichtnegativen Zahlen ein Leerzeichen gemacht.
  - Bei den ganzzahligen Typen `o`, `x` und `X` wird durch Angabe eines `#` der Zahl eine `0`, ein `0x` bzw. ein `0X` vorangestellt. Bei den Dezimaltypen erzeugt ein `#` einen Dezimalpunkt, auch wenn er für die Ausgabe der Zahl nicht notwendig wäre (weil sie ganzzahlig ist).
  - Mit einer `0` wird bis zur Breite der Ausgabe von vorne mit Nullen aufgefüllt. Das hat nur eine Auswirkung, wenn in nächsten Bereich auch eine Breite für die Ausgabe angegeben wird.

- ▷ An der Zweiten Stelle kann eine Zahl oder ein `*` stehen. Eine Zahl gibt die minimale Breite der Ausgabe an, bei einem Stern muss eine weitere Variable vor der auszugebenden Variablen an printf übergeben werden, die die Breite der Ausgabe angibt.
- ▷ An der letzten Stelle kann noch ein `.` mit nachfolgender Zahl oder einem `*` stehen. Im Falle eines Sterns muss eine weitere Zahl vor der auszugebenden Variablen an **`printf`** übergeben werden.

  Bei den ganzzahligen Typen gibt das die minimale Anzahl ausgegebener Stellen an, bei Bedarf wird von vorne mit Nullen aufgefüllt. Für die Dezimaltypen gibt es die Zahl der Nachkommastellen oder die Zahl der gültigen Stellen an. Für Zeichenketten gibt es die maximale Anzahl Zeichen an, die ausgegeben werden. Wenn vorher ein `\0` kommt, wird schon dort die Ausgabe abgebrochen.

  Bei den Zahltypen wird 0 angenommen, wenn keine Zahl hinter dem Punkt kommt, und bei einer Null hinter dem Punkt wird nichts ausgegeben, wenn die auszugebende Zahl 0 ist.

Es gibt noch einige weitere Typen und Spezifizierungen, die nur in Fällen gebraucht werden, die wir in dieser Vorlesung nicht betrachten.

Mit dem Steuerzeichen `\n` können wir in der Zeichenkette auch einen Zeilenumbruch einbauen. Es gibt noch eine Reihe weiterer Steuerzeichen (zum Beispiel `\t` für einen Tabulator), und auf diese Weise können auch alle Zeichen des *Unicode*-Zeichensatzes eingegeben werden.

Hier ist ein Beispiel für die Verwendung von **`printf`**.

Programm 8.1: `kapitel_08/formatierung.c`

```c
#include <stdio.h>

int main() {

  int c = printf("Einfuehrung in die Programmierung 1\n");
  printf("Die vorangegangene Zeile hat %d Zeichen\n\n", c);

  signed char ch = 65;
  printf ("Zeichen (char) %c und %c \n", 'a', ch);
  printf ("Zeichen als Zahl (char): %hhd\n\n", ch);

  printf ("ganze Zahlen als int: %d als long %ld\n", 2020, 4080400L);
  printf ("ganze Zahlen mit fuehrenden Nullen: %010d\n", 2020);
```

```
14    printf ("ganze Zahlen mit fuehrenden Leerzeichen: %10d\n", 2020);
15    printf ("ganze Zahlen als int hexadezimal und oktal: %#x und %#o\n",
16                                                        4080400, 4080400);
17    printf ("ganze Zahlen mit Vorzeichen:\n %+15d und % 5d\n %+15d und % 5d\n\n",
18                                                        2020, 2020, -2020, -2020);
19
20    printf ("Dezimalzahlen: %4.2f %+.0e %E \n\n", 3.14159, 3.14159, 3.14159);
21
22    printf ("Variable Breite: %*d \n\n", 10, 2020);
23
24    printf ("%s \n", "Zeichenketten");
25
26    return 0;
27 }
```

mit der Ausgabe

```
Einfuehrung in die Programmierung 1
Die vorangegangene Zeile hat 36 Zeichen

Zeichen (char) a und A
Zeichen als Zahl (char): 65

ganze Zahlen als int: 2020 als long 4080400
ganze Zahlen mit fuehrenden Nullen: 0000002020
ganze Zahlen mit fuehrenden Leerzeichen:       2020
ganze Zahlen als int hexadezimal und oktal: 0x3e4310 und 017441420
ganze Zahlen mit Vorzeichen:
           +2020 und  2020
           -2020 und -2020

Dezimalzahlen: 3.14 +3e+00 3.141590E+00

Variable Breite:       2020

Zeichenketten
```

8.1.2 Lesen einer Eingabe vom Terminal

Die einfachste Möglichkeit, eine Tastatureingabe vom Bildschirm einzulesen, folgt ähnlichen Regeln wie die Ausgabe, nur mit der Funktion `scanf` statt `printf`. Spezifikationen an den Typ sind hier allerdings nicht so sinnvoll, so dass nur die Typen aus Table 8.1 wichtig sind.

Im Unterschied zu `printf` müssen wir hier nicht eine Variable für jeden Platzhalter übergeben, sondern die *Adresse* einer Variablen, und `scanf` schreibt die eingelesenen Werte dann in der richtigen Reihenfolge in die gegebenen Speicherbereiche. Dabei muss der angegebene Typ zur Eingabe passen, sonst bricht `scanf` ab und lässt alle restlichen Eingaben im Eingabepuffer. Falls weitere Daten eingelesen werden, kann das dazu führen, dass zuerst diese verbliebenen Daten verwendet werden, bevor eventuelle weitere Eingaben berücksichtigt werden. Auch `scanf` gibt ein `int` zurück, das die Anzahl der erfolgreich belegten Variablen anzeigt. Anders als bei `printf` ist es hier durchaus sinnvoll, diese Rückgabe im Programm anzuschauen und zu reagieren, wenn nicht die erwartete Anzahl an Variablen belegt wurde, da hier Eingaben in einem bestimmten

Format erwartet werden, und der Programmbenutzer sich vielleicht nicht korrekt an das Format gehalten hat. Nicht korrekt initialisierte Variablen könnten dann im weiteren Verlauf des Programms zu falschen Berechnungen oder sogar zum Absturz des Programms führen.

Das, und die Notwendigkeit, dass die Eingabe exakt zum erwarteten Format passen muss, machen die Verwendung von **scanf** und Überprüfung der Eingabe schwierig. Das führt daher oft zu Fehlern in einem Programm, die nicht immer leicht zu finden sind.

Programm 8.2: `kapitel_08/eingabe_02.c`

```c
#include <stdio.h>

  int main() {

    int n;
    float f;
    printf("Geben Sie eine ganze und eine Dezimalzahl ein: ");
    int c = scanf("%d %f",&n,&f);
    if ( c != 2 ) {
      printf("Falsche Anzahl an Werten einglesen. Eingabe pruefen.\n");
      exit(1);
    }
    printf("%d Eingaben: Es wurden %d und %f eingelesen\n",c,n,f);

    return 0;
  }
```

Dabei werden Leerzeichen, Tabulator und Zeilenumbruch als Leerzeichen gewertet. Wenn daher Zahlen erwartet werden, werden diese Zeichen ignoriert bis wieder ein anderes Zeichen in der Eingabe erkannt wird. Im Programm oben also nur eine Zahl einzugeben und mit der Eingabetaste zu bestätigen führt nicht zu einem Fehler, sondern die Funktion **scanf** wartet auf weitere Zeichen, die es für den zweiten einzulesenden Parameter berücksichtigen möchte.

Das ist anders, wenn wir Zeichen statt Zahlen einlesen wollen. Da auch ein Zeilenumbruch, wie er von der Eingabetaste erzeugt wird, ein Zeichen ist, ist die mehrfache Eingabe einzelner Zeichen schwierig, da zwischendrin auch der Zeilenumbruch als Zeichen in der Eingabe steht und eingelesen oder aus der Eingabe gelöscht werden muss. Wir können das zum Beispiel in folgendem Programm sehen.

Programm 8.3: `kapitel_08/eingabe_03.c`

```c
#include <stdio.h>

int main() {

  for ( int i = 0; i < 4; ++i) {
    char c;
    printf("Zeichen: ");
    scanf("%c",&c);
    printf("Es wurde %c eingegeben\n",c);
  }

  return 0;
}
```

ergibt

```
Zeichen: a
Es wurde a eingegeben
Zeichen: Es wurde
 eingegeben
Zeichen: b
Es wurde b eingegeben
Zeichen: Es wurde
 eingegeben
```

Wenn sicher ist, dass nach der Eingabe ein Zeilenumbruch kommt, kann in der Schleife ein weiteres Zeichen eingelesen werden, das den Zeilenumbruch aufnimmt. Sicherer ist es, Zeichen mit

```
scanf(" %c",&c);
```

also einem Leerzeichen vor `%c` einzulesen. Das Leerzeichen passt auf Null oder mehr führende Leerzeichen, Tabulatoren und Zeilenumbrüche, sodass ein evtl. im Eingabepuffer verbliebener Zeilenumbruch hier eingelesen wird.

Da **scanf** auch eine komplette Zeichenkette nimmt, könnte man hoffen, dass man hier eine Aufforderung zur Eingabe formulieren könnte. Das ist leider nicht so. Wenn Sie in **scanf** außer Platzhaltern eine Zeichenkette mit Text angeben, wird auch diese Zeichenkette in der Eingabe erwartet, während die Aufforderung nicht auf den Bildschirm geschrieben wird. Im Programm

Programm 8.4: `kapitel_08/eingabe_01.c`

```c
#include <stdio.h>

int main() {

  int n;
  scanf("Geben Sie eine Zahl ein: %d",&n);
  printf("Es wurde %d eingegeben\",n);

  return 0;
}
```

muss daher für die Eingabe von 15 die Zeichenkette

Geben Sie eine Zahl ein: 15

getippt werden. Das ist in dieser Form nur nützlich, wenn einzelne Daten aus schon formatierten Dokumenten ausgelesen werden sollen, die wir in einer Datei oder einem Eingabestream vorliegen haben, und die präzisen Formatierungsrichtlinien folgen (also in der Regel von einem Programm erzeugt und nicht von Menschen eingegeben wurden).

Es gibt eine weitere Funktion, **getchar**, die eine Eingabe zeichenweise liest. Hier wird immer eine Zeichenkette eingelesen, sodass Zahlen vor ihrer Verwendung noch umgewandelt werden müssen (z.B. mit **atoi**).

Programm 8.5: `kapitel_08/eingabe_04.c`

```c
#include <stdio.h>

int main () {
  printf("Ein Zeichen eingeben: ");
  char c = getchar();

```

```
7     printf("Eingegeben: %c\n",c);
8
9   //  c = getchar();
10  //  printf("Eingegeben: %c\n",c);
11
12    return(0);
13  }
```

Auch hier müssen wir beachten, dass der Zeilenumbruch, den wir mit der Eingabetaste erzeugen, als Zeichen zählt und beim nächsten Einlesevorgang eingelesen wird. Sie sehen das, wenn Sie die beiden auskommentierten Zeilen ins Programm nehmen.

Wenn Sie mit **getchar** mehr als ein Zeichen einlesen wollen, müssen Sie eine Zeichenkette buchstabenweise aufbauen. Im folgenden Programm werden solange weitere Zeichen eingelesen und an die vorhandene Zeichenkette angehängt, bis ein Zeilenwechsel erscheint. Das führt dazu, dass, wie wir es wahrscheinlich erwarten, vom Terminal eine Zeichenkette eingelesen wird, die mit der Eingabetaste abgeschlossen wurde.

Programm 8.6: `kapitel_08/eingabe_05.c`

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main () {
5     char s[100];
6     int i = 0;
7     char c;
8
9     printf("Ein Zeichen eingeben: ");
10    while ( c != '\n' ) {
11      c = getchar();
12      s[i++] = c;
13    }
14    s[i] = '\0';
15    printf("Eingegeben: %s\n",s);
16
17    int a = atoi(s);
18    printf("Eingegeben: %d\n",a);
19
20    return(0);
21  }
```

Für das System ist die Ein- und Ausgabe auf den Bildschirm identisch zum Einlesen aus einer Datei und zum Schreiben in eine Datei. Dabei werden Ein- und Ausgabe intern als zwei Dateien **stdin** und **stdout** angesehen, wobei aus der ersten Datei Daten gelesen und in die zweite Datei Daten geschrieben werden. Damit können wir auch auf die Bildschirmein- und -ausgabe alle Methoden anwenden, die für Dateioperationen vorgesehen sind. Als Datei geben wir dann eine der beiden speziellen Dateien an.

Ein- und Ausgabe in eine Datei schauen wir uns im übernächsten Abschnitt an und lernen neue Methoden dafür kennen, die dann auch für Ein- und Ausgabe auf den Bildschirm genutzt werden können. Zuerst wollen wir uns jedoch einer typischen Aufgaben am Anfang eines Programms zuwenden, dem Einlesen und Interpretieren von Kommandozeilenoptionen.

8.2 Parsen von Optionen

Viele Programme, die aus einem Terminal heraus aufgerufen werden, nehmen bei Ihrem Aufruf Parameter an, die den Programmlauf steuern oder Eingabedaten sind. Optionen an solche Kommandozeilenprogramme werden in der Regel mit einem Bindestrich und einem nachfolgenden Buchstaben eingeleitet. Sie kennen das schon vom Kompilieren, wo sie mit der Option `-o` und einer nachfolgenden Zeichenkette den Namen der Ausgabedatei festlegen. Wenn Sie ein Programm schreiben wollen, das optionale Parameter annimmt, müssen Sie am Anfang des Programms die an **main** übergebene Liste von Zeichenketten durchgehen und Variablen entsprechend der Optionen setzen. In Programm 8.7 finden Sie ein Beispiel.

Programm 8.7: `kapitel_08/optionsparsen.c`

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

void printHelp() {
    printf("Optionen einlesen\n(c) Einfuehrung in die Programmierung 1\n");
    printf("\nOptionen:\n");
    char* fmtString = " %-7s -- %s\n";
    printf(fmtString, "-a","Option a");
    printf(fmtString, "-b <n>", "Option b mit Argument");
    printf(fmtString, "-c","Option c");
    printf(fmtString, "-h","diese Hilfe :)");
}

int main(int argc, char** argv) {
    int option_a = 0;
    int option_c = 0;
    int param = -1;

    for ( int i = 1; i < argc; ++i ) {
        if (strcmp(argv[i],"-h") == 0) {
            printHelp();
            return 0;
        }
        if (strcmp(argv[i],"-a") == 0) {
            printf("Option -a gefunden\n");
            option_a = 1;
        }
        if (strcmp(argv[i],"-b") == 0 && argc > i) {
            printf("Option -b gefunden\n");
            param = atoi(argv[++i]);
        }
        if (strcmp(argv[i],"-c") == 0) {
            printf("Option -c gefunden\n");
            option_c = 1;
        }
    }

    printf("Option a: %d, Parameter zu Option b: %d, Option c: %d\n",option_a, param,
    option_c);
    return 0;
}
```

Es ist üblich, dass Programme bei der Option `-h` einen kurzen Text ausgeben, der den Programmaufruf und seine Optionen erklärt. Ebenso sollte diese Hilfe angezeigt werden, wenn Angaben fehlen, die beim Aufruf immer angegeben werden müssen, oder wenn sich angegebene Optionen gegenseitig ausschließen.

8.3 Dateiein- und -ausgabe

Sobald größere Datenmengen ins Spiel kommen, wird es mühsam, diese bei jedem Programmaufruf neu vom Bildschirm einzulesen. In solchen Fällen wollen wir Daten in weiteren Dateien oder in einer Datenbank ablegen und beim Programmstart, oder wenn sie gebraucht werden, von dort einzulesen.

In diesem Abschnitt schauen wir uns an, wie wir Daten aus einer Datei einlesen können und Daten wieder in eine Datei zurückschreiben können. Dafür gibt es jeweils zwei mögliche Wege, die wir in den nächsten beiden Abschnitten betrachten.

8.3.1 Formatierte Ein- und Ausgabe

Wir können analog zur Bildschirmein- und -ausgabe mit den Funktionen **`fprintf`** und **`fscanf`** arbeiten. Beide Funktionen nehmen als zusätzliches erstes Argument einen *Dateideskriptor* (*file handle*), eine Variable, die den Zugang zu einer Datei des Systems speichert. Ansonsten verhalten Sie sich genauso wie **`printf`** und **`scanf`**.

Diese beiden letzteren Funktionen sind auch eigentlich nur spezielle Varianten der Dateifunktionen, die intern als Dateideskriptor eine der beiden speziellen Dateien **`stdin`** für die Eingabe aus einem Terminal bzw. **`stdout`** für die Ausgabe in ein Terminal einsetzen. Es gibt eine dritte solche spezielle Datei, die Datei **`stderr`**, in die Fehlerausgaben geschrieben werden können.

Bevor wir mit einer Datei arbeiten können müssen wir einen Dateideskriptor auf die Datei definieren. Dafür gibt es die Funktion `fopen`, die zwei Argumente nimmt. Das erste Argument ist der Dateiname und der Pfad dorthin. Das kann entweder ein absoluter Pfad sein, oder ein Pfad relativ zu dem Verzeichnis, von dem aus das Programm aufgerufen wurde. Das zweite Argument gibt an, wie wir auf die Datei zugreifen wollen. Hier haben müssen wir angeben, ob wir aus der Datei lesen, oder in die Datei schreiben wollen. Die folgende Anweisung öffnet die Datei `liste.txt` im gleichen Verzeichnis wie der spätere Programmaufruf zum Schreiben (*w*rite).

```
1 FILE * fh = fopen("liste.txt", w);
```

Dabei wird der Typ `FILE` in der Bibliothek `stdio.h` definiert[1]. Diese Variable `fh` können wir jetzt in **`fprintf`** als erstes Argument benutzten, und die ganze Ausgabe landet in der Datei `liste.txt` statt auf dem Bildschirm.

Unsere Optionen, wie eine Datei geöffnet werden soll, sind

- `r` öffnet eine Datei zum *Lesen*. Ein Versuch, Daten in die Datei zu schreiben, führt zu einem Fehler. Die Datei muss dafür schon existieren, sonst erhalten wir ebenfalls einen Fehler.

[1] Genauer ist `FILE` eigentlich kein Typ, sondern ein Makro, dass je nach System vor dem Kompilieren vom Präprozessor durch den auf dem aktuellen System gültigen Typ ersetzt.

- `w` öffnet eine Datei zum *Schreiben*. Eine schon existierende Datei gleichen Namens wird dabei gelöscht.
- `a` öffnet eine Datei zum *Anhängen*. Wenn die Datei schon existiert, werden neue Daten angehängt, die alte Datei verliert aber nicht ihren Inhalt. Das entspricht dem Schreiben, der Dateizeiger wird aber direkt ans Ende der Datei gesetzt.
- r+ öffnet eine Datei zum Lesen und Schreiben. Dabei wird eine alte Version der Datei beibehalten.
- w+ öffnet eine Datei zum Lesen und Schreiben. Eine schon existierende Datei gleichen Namens wird dabei gelöscht.
- a+ öffnet eine Datei zum Lesen und Schreiben, bzw. legt die Datei an, wenn Sie noch nicht existiert. Dabei werden neue Daten immer am Ende der Datei angehängt, selbst wenn zwischendurch der Dateizeiger an eine andere Stelle verschoben wurde.

Bei allen Operationen, die Interaktion mit dem System erfordern, sollten wir den Erfolg überprüfen und auf mögliche Fehler reagieren, da die Kontrolle über den Ablauf der Lese- und Schreiboperationen nicht innerhalb des Programms, bzw. innerhalb einer der C-Standardbibliotheken abläuft, sondern vom Betriebssystem organisiert wird. Das Betriebssystem meldet dann, ähnlich wie unser `return 0;` am Ende der `main`-Funktion, über einen Fehlercode zurück, ob die Ausführung erfolgreich war. Wie bei C steht hier `0` für Erfolg und alles andere für einen Fehler.[2]

Bei Dateioperationen könnte z.B. die Datei nicht vorhanden sein, der angegebene Dateipfad nicht existieren, oder nicht mehr ausreichend Speicherplatz auf der Festplatte vorhanden sein, um alle Daten in die Datei zu schreiben. Eine typische Erfolgsabfrage beim Öffnen einer Datei könnte wie in dem nachfolgenden Programmausschnitt aussehen.

```
1  FILE * fh = fopen("liste.txt", r);
2
3  if ( fh == NULL ) {
4    printf("Datei nicht lesbar\n");
5    exit(1);
6  }
```

Hier wird überprüft, ob wir wirklich einen Dateideskriptor auf die Datei `liste.txt` bekommen haben, und wenn das nicht der Fall ist, wird das Programm mit einer Bildschirmausgabe und der Anweisung `exit` beendet. Vom System wird Eingabe in die und Ausgabe aus der Datei so lange für andere Programme gesperrt, wie wir die Datei in unserem Programm geöffnet haben. Nach der Verwendung sollten wir eine Datei daher wieder schließen um den Zugriff für andere Programme freizugeben.

Wenn wir in eine Datei schreiben, wird die Schreiboperation in der Regel auch nicht direkt in die Datei, sondern in einen Zwischenspeicher geschrieben, da Schreiboperationen auf die Festplatte sehr lange dauern. Dieser Prozess wird ebenfalls vom Betriebssystem gesteuert und nicht von einer Funktion in unserem Programm. Erst wenn dieser Puffer voll ist, wird der gesamte Inhalt des Puffers in einem Durchgang in die Datei geschrieben.

[2] Sie können diese Information auch im Terminal abfragen. Bei der am häufigsten verwendeten *Shell*, der `bash`-Shell, steht diese Information nach Programmende in der Variablen `$?`, die sie z.B. mit `echo $?` ausgeben können.

Schließen der Datei erzwingt nun ebenfalls, dass die Daten geschrieben werden. Daher sichern wir uns durch Schließen der Datei auch gegen Datenverlust ab, falls das Programm nach der Schreiboperation abstürzt oder vom System beendet wird. Wir schließen eine Datei mit der Anweisung **fclose**.

```
fclose(fh);
```

Mit dem folgenden Programm schreiben wir zwei Zeilen in eine Datei `daten.txt` im gleichen Verzeichnis wie der Programmaufruf.

Programm 8.8: `kapitel_08/file_write.c`

```
#include <stdio.h>

int main() {

  FILE * fh = fopen("daten.txt","w");
  fprintf(fh,"Ein Wiesel\nsass auf einem Kiesel\n");
  fclose(fh);

  return 0;
}
```

Wir können uns die Datei im Terminal zum Beispiel mit der Anweisung `cat` ansehen.

```
$ cat daten.txt
Ein Wiesel
sass auf einem Kiesel
```

Mit **fscanf** können wir aus einer Datei einlesen. Die Rolle der Eingabetaste, mit der wir beim lesen vom Bildschirm die Verarbeitung der Eingabe gestartet haben, übernimmt hier das Zeilenende in der Datei[3]. Wie bei der Bildschirmeingabe muss der in **fscanf** Formatstring zur Eingabe in der Datei passen. Das Einlesen wird abgebrochen, sobald die Eingabe nicht mehr zum Format passt. Das passiert auch, wenn die Zeile länger ist als von **fscanf** erwartet. In dem Fall werden alle restlichen Eingaben bis zum Zeilenende ignoriert.

Hier ist ein Beispiel, wie mit **fscanf** Daten eingelesen werden können. Die Daten stehen dafür in einer Datei, die in der ersten Zeile die Anzahl der einzulesenden Daten angibt, und danach in jeder Zeile einen Datensatz aufführt. Das könnte dann so aussehen:

`kapitel_08/nachbarlaender.txt`

```
9
Belgien Bruessel 11429336
Daenemark Kopenhagen 5733551
Frankreich Paris 64979548
Luxemburg Luxemburg 583455
Niederlande Amsterdam 17035938
Oesterreich Wien 8823054
Polen Warschau 38170712
Schweiz Bern 8476005
Tschechien Prag 10618303
```

[3] Wenn eine Zeile länger als der Eingabepuffer ist, dann wird auch vorher eine Verarbeitung ausgelöst. Eingabedaten dieser Länge (wie sie z.B. bei Textverarbeitungsprogrammen nötig sind) wollen wir uns hier nicht ansehen, und sie werden auch besser mit einer der anderen Methoden verarbeitet.

Mit dem folgenden Programm lesen wir dann diese Datei ein und geben die Daten wieder auf dem Bildschirm aus.

Programm 8.9: `kapitel_08/einlesen_fscanf_01.c`

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

int main() {

  FILE * fh = fopen("nachbarlaender.txt","r");

  int anzahl;
  fscanf(fh,"%d",&anzahl);

  printf("Es werden %d Laender eingelesen\n",anzahl);

  char laender[anzahl][100];
  char staedte[anzahl][100];
  int einwohner[anzahl];

  for ( int i = 0; i < anzahl; ++i ) {
    fscanf(fh,"%s %s %d", laender[i], staedte[i], &einwohner[i]);
    //fscanf(fh,"%s %s %d", laender[i], staedte[i], einwohner+i);
  }

  fclose(fh);

  printf("%-11s %-11s %14s\n","Land", "Hauptstadt", "Einwohnerzahl");
  for ( int i = 0; i < anzahl; ++i ) {
    printf("%-12s %-12s %12d\n", laender[i], staedte[i], einwohner[i]);
  }

  return 0;
}
```

Beachten Sie hierbei, dass `C` in der Grundform nur ein Leerzeichen (oder Tabulator bzw. Zeilenumbruch) als Ende einer Zeichenkette erkennt. Die übliche Trennung durch Kommata in Tabellen erfordert etwas mehr Aufwand beim Einlesen, da wir `C` mitteilen müssen, dass auch ein Komma Werte auf der Zeile trennt.

Auf den ersten Blick könnten Sie auch annehmen, dass wir beim Einlesen der Zeichenketten `laender[i]` und `staedte[i]` über **`fscanf`** den Adressoperator vergessen haben. Diese Variablen sind jedoch schon Zeiger auf einen Speicherbereich, da sie Listenvariablen sind, deren Grundform auf die erste Adresse ihres Speicherblocks zeigt. Die dritte Liste ist nur eindimensional, so dass wir hier die Adresse übergeben müssen. Unter Ausnutzung der Zeigerarithmetik hätten wir statt `&einwohner[i]` allerdings auch `einwohner+i` übergeben können, da `einwohner` ein Zeiger auf das erste Element ist.

Die Funktion **`fscanf`** hat auch einen Rückgabewert, den wir im Beispiel nicht verwendet haben. Die Funktion gibt entweder die Anzahl der eingelesenen Zeichen oder den speziellen Wert **`EOF`** (für *end of file*) zurück, der angibt, dass keine weiteren Daten aus der Datei gelesen werden können.

Da es neben `char` und ganzen bzw. Dezimalzahlen keine Typen in `C` gibt, ist **`EOF`** allerdings intern ebenfalls nur eine Zahlkonstante (meistens `-1`), die verschieden von allen echten Zeichencodes ist, die **`getchar`** zurückgeben kann (also aus dem Bereich

`unsigned char` ist).

Damit hätten wir die Schleife zum Einlesen der Daten auch in der Form

```
int a = 0;
while ( fscanf(fh,"%s %s %d", laender[a], staedte[a], &einwohner[a]) != EOF ) {
  a++;
}
```

schreiben können. In diesem Fall brauchen wir die Angabe der Anzahl von Zeilen der Datei am Anfang nicht mehr.

8.3.2 Zeichenweise Ein- und Ausgabe

Die zweite Methode, Daten einzulesen, benutzt die Funktion `fgets`, die eine festgelegte Anzahl Zeichen einer Zeile einliest, maximal aber bis zum Zeilenende. Die eingelesenen Daten sind immer vom Typ `char`, wir müssen daher ggf. noch mit einer der bekannten Funktionen in den gewünschten Datentyp umwandeln. In der einfachsten Form könnte Einlesen so aussehen:

Programm 8.10: `kapitel_08/einlesen_fgets.c`

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

#define MAX_LINE 100

int main(int argc, char** argv) {

  FILE *fh;
  char* fname = argv[1];

  fh = fopen(fname,"r");
  if ( file == NULL ) {
    printf("Datei nicht lesbar\n");
    exit(1);
  }

  char line[MAX_LINE] = "";

  while (fgets(line, MAX_LINE, fh)) {
    printf("%s",line);
  }

  fclose(fh);

  return 0;
}
```

Das Programm liest eine Datei zeilenweise ein, und gibt jede Zeile direkt auf dem Bildschirm aus. `fgets` erwartet drei Argumente. Das erste ist eine Zeichenkette, in die die eingelesene Zeile geschrieben wird. Das zweite Argument gibt an, wie viele Zeichen maximal gelesen werden sollen, während das dritte die Datei angibt, aus der gelesen werden soll.

Die erste Variable sollte mindestens so viele Zeichen fassen, wie eingelesen werden können, sonst überschreibt `fgets` möglicherweise Speicherbereiche, die nicht dafür

vorgesehen waren. Wie wir schon gesehen haben, überprüft C die Größe von Listen nicht, so dass hier keine Fehlermeldung beim Übersetzen des Programms ausgegeben wird. **fgets** gibt wie **fscanf** die Anzahl der eingelesenen Zeichen oder **EOF** zurück. Daher können wir es als Bedingung in der while-Schleife einsetzen und auf diese Weise bis zum Dateiende lesen.

Für die Umwandlung der eingelesenen Zeichenkette in einzelne Daten betrachten wir zwei Optionen. Zum einen gibt es die Funktion **sscanf**, die ähnlich wie **fscanf** funktioniert, aber statt einer Datei eine Zeichenkette als erstes Argument erwartet, die dann entsprechend des angegebenen Formats Variablen zuweist. Unsere Länderliste könnten wir auch mit dem folgenden Programm einlesen.

Programm 8.11: kapitel_08/einlesen_fgets_01.c

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

#define MAX_LINE 100

int main() {

  FILE * fh = fopen("nachbarlaender.txt","r");

  int anzahl;
  char line[MAX_LINE];

  fgets(line, MAX_LINE, fh);
  sscanf(line,"%d",&anzahl);

  printf("Es werden %d Laender eingelesen\n",anzahl);

  char laender[anzahl][100];
  char staedte[anzahl][100];
  int einwohner[anzahl];

  int a = 0;
  while ( fgets(line, MAX_LINE, fh) ) {
    sscanf(line, "%s %s %d", laender[a], staedte[a], &einwohner[a]);
    a++;
  }

  fclose(fh);

  printf("%-11s %-11s %14s\n","Land", "Hauptstadt", "Einwohnerzahl");
  for ( int i = 0; i < anzahl; ++i ) {
    printf("%-12s %-12s %12d\n", laender[i], staedte[i], einwohner[i]);
  }

  return 0;
}
```

Damit haben wir allerdings immer noch das gleiche Problem wie beim Einlesen mit **fscanf**. Wir müssen auch hier vorher genau wissen, wie viele Daten wir in welcher Form in einer Zeile einlesen müssen. Damit können wir z.B. keine Matrix aus einer Datei einlesen, deren Spaltenzahl wir nicht schon zum Zeitpunkt des Übersetzens des Programms kennen.

Wenn die Anzahl der einzulesenden Daten in den Zeilen variabel ist, ist es oft besser, die Zeichenkette mit passenden Funktionen für Zeichenketten zu zerlegen und die Teile einzeln umzuwandeln. Insbesondere eignet sich hier die Funktion `strtok`, die eine gegebene Zeichenkette nacheinander entlang von uns gewählter Trennzeichen in Teile zerlegt. Die Funktion sieht so aus:

```
1   char * strtok(char *s, const char *begrenzer)
```

Dabei ist `s` die Zeichenkette, die wir zerlegen wollen und `begrenzer` enthält eine Liste von Zeichen, an denen getrennt werden soll. Die Funktion gibt den Anteil von `s` bis zum ersten Begrenzer zurück. Wenn wir weitere Teile von `s` haben wollen, rufen wir die Funktion mit `NULL` statt `s` auf, `strtok` liest dann solange weiter Teile von `s` ein, bis wir eine neue Zeichenkette übergeben. Wenn keine Zeichen in `s` mehr vorhanden sind, gibt die Funktion `NULL` zurück.

Hier ist ein Beispiel für das Einlesen einer Matrix. Die Daten könnten in der Form

kapitel_08/matrix.dat

```
3 4
12 34 56 78
2 4 6 8
3 6 9 12
```

in einer Datei vorliegen, wobei in der ersten Zeile die Dimension der nachfolgenden Matrix steht, zuerst die Anzahl der Zeilen, dann die Anzahl der Spalten. Mit dem folgenden Programm lesen wir die Daten ein und geben sie anschließend auf dem Bildschirm wieder aus.

Programm 8.12: kapitel_08/einlesen_fgets_02.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define LINE_LENGTH 100
6
7  int main(int argc, char** argv) {
8
9    char* fname = argv[1];
10   FILE * fh = fopen(fname,"r");
11   if ( fh == NULL ) {
12     printf("Datei nicht lesbar\n");
13     exit(1);
14   }
15
16   char line[LINE_LENGTH] = "";
17
18   int zeilen = 0;
19   int spalten = 0;
20   fscanf(fh,"%d %d\n",&zeilen,&spalten);
21   int mat[zeilen][spalten];
22
23     char* teilkette;
24     char begrenzer[2] = " "; // an Leerzeichen trennen, Laenge 2 wegen \0
25
26     int s = 0;
27     int z = 0;
```

```
28      while( fgets(line,LINE_LENGTH,fh) ) {
29        s = 0;
30        teilkette = strtok(line,begrenzer);
31        while ( teilkette != NULL ) {
32          mat[z][s++] = atoi(teilkette);
33      teilkette = strtok(NULL,begrenzer);
34        }
35        z++;
36      }
37
38      fclose(fh);
39
40      printf("Matrix mit %d Zeilen und %d Spalten eingelesen: \n", z, s);
41      for (int i = 0; i < z; ++i ) {
42        for (int j = 0; j < s; ++j ) {
43          printf("%d ",mat[i][j]);
44        }
45        printf("\n");
46      }
47
48      return 0;
49    }
```

Bei Daten variabler Länge in der Eingabe müssen wir, wie bisher, die Größe überschätzen und Variablen definieren, die Daten bis zu einem, in der Regel mit einem `#define` am Anfang des Programms festgelegten, Maximum aufnehmen können. Also so wie wir das im vorangegangenen Programm mit der Länge einer Zeile gemacht haben, die wir auf 100 Zeichen begrenzt haben.

Wir werden später sehen, dass wir auch selbst Speicher in einer von uns zur Laufzeit gewählten Größe dynamisch reservieren können und einen Zeiger auf den Anfang des Speicherblocks bekommen, und das daher passend zur Eingabe machen können. Auch in diesem Fall sollten wir allerdings bei größeren Datenmengen darauf achten, dass wir nicht für jeden Einlesevorgang neu Speicher anfordern, da diese Operation in der Regel sehr langsam ist.

In vielen Fällen ist es daher sinnvoll, die Größe der Daten am Anfang der Eingabe festzuhalten, so wie wir unsere Matrixdimension an den Anfang geschrieben haben, oder in einem ersten Durchgang durch die Daten keine Daten einzulesen sondern nur die Größe zu bestimmen, und erst in einem zweiten Durchgang dann die Daten in Datenstrukturen passender Größe einzulesen, die wir anhand der uns nun bekannten Größe der Daten deklariert haben. Eine weitere Möglichkeit für Listen, bei der wir dynamisch bei Bedarf gleich größere Blöcke von Speicher hinzufügen, sehen wir ebenfalls später.

Mit **`fgets`** haben wir zeichenweise eingelesen. Umgekehrt können wir mit

```
1   int fputs(const char *s, FILE *fh)
```

eine Zeichenkette `s` in die Datei, auf die `fh` zeigt, abspeichern. Die Zeichenkette muss dabei korrekt mit `\0` beendet sein. Falls der Schreibvorgang erfolgreich war, gibt die Funktionen die Anzahl der geschriebenen Zeichen, und damit eine positive Zahl zurück, andernfalls (z.B. wenn nicht genug Speicherplatz zur Verfügung steht), gibt sie **`EOF`** (für *end of file*) zurück, was, wie wir schon diskutiert haben, einfach eine negative Konstante ist, und in der Regel -1.

Um die oben eingelesene Matrix wieder in eine Datei zu schreiben, können wir z.B. den folgenden Codeausschnitt benutzen.

Programm 8.13: `kapitel_08/ausgabe_fputs_01.c`

```
   fh = fopen(fname_out,"w"); // zum Schreiben oeffnen
   if ( fh == NULL ) {
      printf("Datei nicht schreibbar\n");
      exit(1);
   }

   char outstring[LINE_LENGTH] = "";
   sprintf(outstring,"%d %d\n", zeilen, spalten);
   fputs(outstring, fh);

   for (int i = 0; i < z; ++i ) {
      outstring[0] = '\0';

      for (int j = 0; j < s; ++j ) {
         char n[MAX_DIGITS+2];  // fuer Leerzeichen zwischen Eintraegen und \0
         sprintf(n,"%d ", mat[i][j]);
         strcat(outstring, n);
      }

      strcat(outstring, "\n");
      fputs(outstring, fh);
```

Wie wir oben schon gesehen haben, ist eine Ein- oder Ausgabe auf den Bildschirm (in ein Terminal) für das Programm identisch zu einer Eingabe von oder Ausgabe in eine Datei. Dafür sind die speziellen Dateizeiger **`stdin`** für die Eingabe, **`stdout`** für die Ausgabe und **`stderr`** für eine Fehlerausgabe definiert. Diese drei (Pseudo-)Dateien werden immer am Anfang des Programms geöffnet und am Ende wieder geschlossen. Sie können diese drei Zeiger daher an allen Stellen einsetzen, an denen eine Funktion einen Dateizeiger erwartet.

Das ist insbesondere nützlich, wenn wir variabel auf den Bildschirm oder in eine Datei schreiben wollen. Wir könnten dafür eine Option in unserem Programm auswerten, und in Abhängigkeit ihres Werts wird entweder eine richtige Datei geöffnet, oder der Dateizeiger auf **`stdout`** gesetzt. Das könnte zum Beispiel wie in dem folgenden Programm gelöst werden.

Programm 8.14: `kapitel_08/ausgabe_fputs_02.c`

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

void printHelp() {
    printf("Variable Ausgabe\n(c) Einfuehrung in die Programmierung 1\n");
    printf("\nOptionen:\n");
    char* fmtString = " %-10s -- %s\n";
    printf(fmtString, "-o <Datei>","Ausgabe in Datei");
    printf(fmtString, "-h","diese Hilfe :)");
}

int main(int argc, char** argv) {
    int option_o = 0;

```

```
16      FILE * fh = stdout;
17
18      for ( int i = 1; i < argc; ++i ) {
19          if (strcmp(argv[i],"-h") == 0) {
20              printHelp();
21              return 0;
22          }
23          if (strcmp(argv[i],"-o") == 0) {
24              option_o = 1;
25              fh = fopen(argv[++i],"w"); // zum Schreiben oeffnen
26              if ( fh == NULL ) {
27                  printf("Datei nicht schreibbar\n");
28                  exit(1);
29              }
30          }
31      }
32
33      fprintf(fh,"EInfuehrung in die Programmierung 1\n");
34
35      if ( option_o ) {
36          fclose(fh);
37      }
38
39      return 0;
40  }
```

Beachten Sie dabei auch, dass Sie am Ende `fclose` aufrufen sollten, wenn Sie vorher eine richtige Datei geöffnet haben statt die Daten nach `stdout` zu schreiben.

Die Ausgabe auf `stdout` und `stderr` erfolgt in der Regel ununterscheidbar im Terminal. Mit Umleitungen im Terminal können Sie aber eine der Ausgaben zum Beispiel in eine Datei umleiten. Die passende Syntax für die *Shell* in Ihrem Terminal können Sie in der Dokumentation nachlesen. In der am häufigsten verwendeten *Bash* können Sie die Ausgabe aus einem Programm mit `>` umleiten. Der Aufruf eines Programms `prog` mit

```
./prog > ausgabedatei
```

leitet also die komplette Ausgabe auf `stdout` des Programms in die Datei `ausgabedatei` um.

Wenn Sie nur die Fehlerausgabe in eine Datei umleiten wollen, dann können sie das mit

```
./prog 2> logdatei
```

erreichen. Beides in die gleiche Datei zu schicken geht mit

```
./prog &> ausgabedatei
```

und beides in getrennte Dateien zu schicken mit

```
./prog > ausgabedatei 2> logdatei
```

Die Ausgabe beider Dateien, `stdout` und `stderr`, in eine einzige Datei zu schreiben, geht auch, indem man zum einen `stderr` nach `stdout` leitet, und zudem `stdout` in eine Datei. Das kann man mit

```
./prog > ausgabedatei 2>&1
```

erreichen. In älteren Versionen von *bash* ist das auch die einzige Variante, mit der man beide Ausgabe in eine Datei schicken kann. Wie beim Schreiben in Dateien in `C` werden auch hier existierende Dateien gelöscht. Wenn wir stattdessen anhängen wollen, dann können wir >> verwenden.

Umgekehrt können Sie mit < Daten nach **`stdin`** schreiben, also z.B. eine Datei auslesen und als Kommandozeileneingabe für Ihr Programm benutzen.

Version vom 10. Februar 2022

9 Algorithmen und Rekursion

In diesem Abschnitt wollen wir uns mit *Algorithmen* beschäftigen. Algorithmen sind im wesentlichen eine endliche Abfolge von Anweisungen zur (korrekten) Lösung eines Problems. Aus der Sicht des Programmierers brauchen wir solche Algorithmen, um überhaupt etwas programmieren zu können. Wenn wir zum Beispiel alle Teiler einer Zahl bestimmen wollen, brauchen wir, bevor wir anfangen können zu programmieren, eine Methode, wie wir diese Teiler bestimmen können, d.h. wir brauchen einen sogenannten *Algorithmus* dafür. Aus mathematischer Sicht kann man solche Algorithmen, also Abfolgen von Anweisungen, unabhängig von einer konkreten Programmiersprache betrachten und zum Beispiel untersuchen, für welche Eingaben unser Algorithmus ein im Sinne der Problemstellung korrektes Ergebnis liefert, und ob unsere spezielle Abfolge von Anweisungen für die Lösung des Problems *effizient* ist. Dabei müssen wir natürlich noch präzise festlegen, was wir an dieser Stelle eigentlich mit *effizient* meinen wollen. In der Regel interessiert man sich hier für die Anzahl der Einzelschritte, die man bei der Abarbeitung der Anweisungen machen muss, und wir vergleichen verschiedene Methoden, das gleiche Problem zu lösen, anhand dieser Anzahl. Dabei vergleichen wir diese Anzahlen für *große* Eingaben, um Ein- und Auslesen sowie Details der Implementierung nicht überzubewerten.

Algorithmen kann man unabhängig von einer konkreten Programmiersprache betrachten, und ihre Untersuchung ist inzwischen auch ein eigenes, sehr dynamisches, Gebiet innerhalb der Mathematik und Informatik. Sie werden Algorithmen aus mathematischer Sicht spätestens im vierten Semester erneut in der Vorlesung *Algorithmic Discrete Mathematics* begegnen. In dieser Vorlesung wird es vor allem um algorithmen auf Graphen gehen, also endlichen Strukturen, die aus einer endlichen Menge V von sogenannten Knoten und einer ausgezeichneten Teilmenge der Paare von Elementen von V bestehen. Ein prominentes Beispiel sind (vereinfachte) Strassennetzwerke, wo die Knoten die Kreuzungen sind und wird diejenigen Paare von Kreuzungen auszeichnen, zwischen denen eine direkte Strassenvewbindung ohne weitere Kreuzungen besteht.

Wir werden uns in dieser Einführung ins Programmieren nur kurz und überblicksweise mit der Theorie von Algorithmen und Komplexität beschäftigen und uns mehr mit konkreten Algorithmen beschäftigen, die wir dann natürlich in `C` in ein Programm umsetzen. Für mehr Theorie zu Algorithmen ist es hilfreich, wenn Sie sich vorher mit anderen Themen in der Mathematik vertraut machen, inbesondere der *Linearen Algebra* und der *Analysis*.

Im einem der nächsten Kapitel werden wir uns dann mit einem der am häufigsten auftretenden algorithmischen Probleme in der Programmierung beschäftigen, dem Sortieren einer Liste von Daten anhand eines vorgegebenen Sortierkriteriums (aus mathematischer Sicht einer *Totalordnung* auf der Menge aller möglichen Elemente der Liste). Dabei werden wir nur einige einfache Algorithmen vorstellen, und das

Version vom 10. Februar 2022

Thema später erneut aufgreifen um effizientere Algorithmen zu betrachten. Weitere Algorithmen zu anderen Aufgaben kommen dann später, und vor allem in den Tutorien.

9.1 Algorithmen

Algorithmen sind deterministische Abfolgen von Anweisungen, die zu jeder Instanz aus einer vorgegebenen zulässigen Menge von Eingaben nach einer endlichen Anzahl von Schritten ein Ergebnis zurückliefert.

Über Algorithmen spricht man in der Regel nur im Zusammenhang mit einem konkreten *Problem* Π, also einer (mathematischen) Fragestellung über alle Elemente einer vorgegebenen Eingabemenge $\mathcal{I}$ von Instanzen. Ein Algorithmus $\mathcal{A}$ *löst* ein Problem Π, wenn er zu jeder Instanz $I \in \mathcal{I}$ in endlich vielen Schritten eine (korrekte) Lösung aus einer Lösungsmenge S_I zurückgibt.

Hier sind einige Beispiele von Problemen:

- Alle Teiler einer Zahl. Hier ist die Menge der möglichen Eingaben (die Instanzen) die Menge der natürlichen oder ganzen Zahlen, und jede Lösung ist aus der Menge der endlichen Folgen natürlicher Zahlen.
- Der größte gemeinsame Teiler zweiter Zahlen. Hier sind die zulässigen Eingaben Paare natürlicher Zahlen, und wir wollen eine natürliche Zahl zurückbekommen.
- Ein Weg zwischen zwei Adressen in Darmstadt. Hier könnte die Eingabemenge aus Paaren von Häusern (Straße und Hausnummer) bestehen, und wir suchen eine Verbindung im Netzwerk der Straßen.
- Eine Verbindung zwischen zwei Haltestellen öffentlicher Verkehrsmittel.
- Einen Ablaufplan, um ein Gebäude zu evakuieren.

Wir wollen uns als konkretes Beispiel zuerst die Frage nach dem größten gemeinsamen Teiler zweier natürlicher Zahlen stellen und nach Algorithmen für dieses Problem suchen und sie vergleichen.

Die Methode in Algorithmus 9.1 ist eine erste algorithmische Möglichkeit, den größten gemeinsamen Teiler zweier positiver ganzer Zahlen $a, b \in \mathbb{Z}_{\geq 1}$ zu bestimmen. Die Notation in *Pseudocode* haben wir schon ganz am Anfang der Vorlesung kennengelernt, als wir uns zum ersten Mal mit dem Prinzip eines Programms vertraut gemacht haben. Diese Notation hat sich auch allgemein für Algorithmen eingebürgert, da wir damit unabhängig von einer konkreten Programmiersprache die wesentlichen Schritte eines Ablaufplans aufschreiben können.

Ein solcher Ablaufplan alleine macht allerdings noch keinen Algorithmus. Wir müssen dafür auch klären, ob diese Abfolge von Anweisungen für alle Eingaben, die wir zugelassen haben, auch nach endliche vielen Schritten abbricht und auf allen erlaubten Eingaben ein korrektes Ergebnis (im Sinne der Problemstellung, die der Algorithmus lösen soll) zurückgibt.

In dem Algorithmus für den größten gemeinsamen Teiler nehmen wir an, dass bei der Eingabe $a \geq b$ ist (wir vertauschen andernfalls), und testen dann der Reihe nach von b absteigend alle Zahlen, bis wir einen gemeinsamen Teiler gefunden haben. Hier ist eine Umsetzung in `C`. Wir sehen direkt, dass die konkrete Umsetzung länger (und

Algorithmus 9.1: Größter gemeinsamer Teiler, Variante 1

Eingabe : Natürliche Zahlen a und b
Ausgabe : Der größte gemeinsame Teiler von a und b

1 **wenn** $a \leq b$ **dann**
2 | vertausche a und b;
3 ggT = 0;
4 **für** *i von b bis 1* **tue**
5 | **wenn** *(a und b sind durch i teilbar)* **dann**
6 | | ggT = i;
7 | | break;
8 **zurück** ggT;

wahrscheinlich auch schwerer zu lesen) ist, da wir uns hier natürlich auch um Aufgaben kümmern müssen, die nicht direkt mit dem Algorithmus verbunden sind, wie zum Beispiel die Ein- und Ausgabe oder die korrekte Konvertierung der Daten.

Programm 9.1: `kapitel_09/ggt_v1.c`

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

int main(int argc, char** argv) {

   long long a = atoll(argv[1]);
   long long b = atoll(argv[2]);
   if ( a < b ) {
     long long c = a;
     a = b;
     b = c;
   }

   long long ggt = 0;
   for ( long long i = b; i > 0; --i ) {
     if ( a % i == 0 && b % i == 0 ) {
       ggt = i;
       break;
     }
   }

   printf("Der groesste gemeinsame Teiler von %lld und %lld ist %lld\n",a,b,ggt);
   return 0;
}
```

Wir sollten uns bei der Analyse von solchen Ablaufplänen bzw. Algorithmen immer (mindestens) die folgenden zwei Fragen dazu stellen.

- ▷ Ist der Ablaufplan korrekt im Sinne der Problemstellung, also ein Algorithmus der unser Problem löst?
- ▷ Ist der Algorithmus effizient? Oder können wir eine schnellere oder einfachere Methode angeben, die ebenfalls das Problem löst.

Die erste Frage ist für unser konkretes Beispiel leicht zu beantworten. Wir werden immer nach endlich vielen Schritten den größten gemeinsamen Teiler bekommen, wenn

wir alle Zahlen zwischen b und 1 absteigend durchprobieren, da 1 auf jeden Fall ein Teiler ist. Es kann also nicht passieren, dass wir am Ende des Ablaufplans ankommen und keinen Teiler gefunden haben, und wir sind nach höchstens b Durchläufen der Schleife bei diesem Teiler angekommen.

Die zweite Frage ist erheblich schwieriger. Wir können dafür versuchen, eine schnellere Methode zur Berechnung des größten gemeinsamen Teilers zu finden. Wenn wir damit erfolgreich sind, haben wir eine effizientere Möglichkeit erhalten. Im anderen Fall wissen wir allerdings nicht viel, denn es könnte ja sein, dass es trotzdem einen besseren Weg gibt, wir ihn nur nicht erkannt haben.

In Algorithmus 9.2 finden Sie für unser konkretes Problem eine weitere Methode. Aus der Analysis kennen Sie (oder Sie leiten es sich selbst her, wenn Sie sich nicht erinnern) die Gleichung

$$\mathrm{ggT}(a, b) \;=\; \mathrm{ggT}(b, a - b)$$

für natürliche Zahlen a, b mit $a \geq b$. Daraus lässt sich leicht ein weiterer Algorithmus für das gleiche Problem ableiten.

Algorithmus 9.2: Größter gemeinsamer Teiler, Variante 2

Eingabe : Natürliche Zahlen a und b
Ausgabe : Der größte gemeinsame Teiler von a und b

```
1  wenn a ≤ b dann
2  |  vertausche a und b;
3  solange b ≠ 0 tue
4  |  r = a - b;
5  |  wenn b > r dann
6  |  |  a = b;
7  |  |  b = r;
8  |  sonst
9  |  |  a = r;
10 zurück a;
```

Seine Umsetzung in `C` ist, wie beim vorherigen Programm, wieder etwas länger, da wir uns um Ein- und Ausgabe, Initialisierung und ähnliches kümmern müssen.

Programm 9.2: `kapitel_09/ggt_v2.c`

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

int main(int argc, char** argv) {

   long long a = atoll(argv[1]);
   long long b = atoll(argv[2]);
   if ( a < b ) {
     long long c = a;
     a = b;
     b = c;
   }

```

```
14      long long at = a;
15      long long bt = b;
16      while ( bt != 0 ) {
17         long long r = at - bt;
18         if ( bt > r ) {
19            at = bt;
20            bt = r;
21         }  else {
22            at = r;
23         }
24      }
25
26      printf("Der groesste gemeinsame Teiler von %lld und %lld ist %lld\n",a,b,at);
27      return 0;
28   }
```

Ist dieses Programm nun *effizienter* als das erste? Wenn wir beide Programme auf $a = 60$ und $b = 40$ anwenden, braucht das erste Programm 20 Iterationen um den korrekten größten gemeinsamen Teiler 20 der beiden Zahlen zu finden, während das zweite nur 3 Iterationen benötigt. In diesem Fall ist das zweite also deutlich schneller.

In diesem Beispiel haben wir vielleicht aber nur Glück gehabt mit unseren Wahlen. Wenn wir einen Algorithmus bewerten wollen, sollten wir alle möglichen Eingaben berücksichtigen und die Laufzeit unabhängig von einer konkreten Eingabe betrachten. Um das zu erreichen, ist es üblich, als Maßstab für die Effizienz eines Algorithmus die Zahl der Rechenschritte bis zum Ergebnis (die *Laufzeit*) zu nehmen, die der Algorithmus im *ungünstigsten Fall* benötigt. Das müssen wir allerdings im Verhältnis zur Größe der Eingabe betrachten, denn wenn wir zum Beispiel im ersten Algorithmus die eingegebenen Zahlen verdoppeln, verdoppelt sich auch die Zahl der Iterationen. Die eigentliche Komplexität des Algorithmus hat sich jedoch nicht verschlechtert.

Einen zweiten Effekt wollen wir ebenfalls nicht in unserer Abschätzung sehen. Wenn wir unseren Algorithmus auf sehr kleine oder einfache Werte anwenden (zum Beispiel auf $a = 2$ und $b = 1$ oder $a = b = 100$), dann wird die Zeit, die der Algorithmus benötigt, um seine Hilfsvariablen und die Schleife zu initialisieren, die Zeit, die wir für die eigentliche Berechnung brauchen, deutlich überwiegen. Bei anderen, komplizierteren, Algorithmen müssen vielleicht auch noch am Anfang einige Datenstrukturen aufgebaut und vorbereitende Rechnungen gemacht werden, um dann die eigentliche Berechnung durchzuführen, auf die es uns ankommt. Um diesen Einfluss bei der Bewertung eines Algorithmus nicht zu überschätzen, betrachtet man in der Regel eine asymptotische Laufzeit in Bezug auf die Größe der Eingabe, d.h. wir schauen uns den Grenzwert für wachsende Eingabegrößen der mit dieser Eingabegröße normierten Anzahl von Rechenschritten an.

Um das nun präziser zu fassen, müssen wir uns überlegen, wie wir die Zahl der Rechenschritte bestimmen wollen, und wie wir die Eingabegröße definieren. Dabei sollte unser Maß natürlich fein genug sein, um eine Abschätzung zu geben, die mit der praktischen Beobachtung übereinstimmt. Es muss andererseits aber auch ausreichend allgemein und einfach sein, um eine Aussage zu ermöglichen, die einen Vergleich mit anderen Algorithmen zulässt. Unser Maß sollte also keine Eigenheiten einer konkreten Prozessorarchitektur oder ähnliches berücksichtigen, um eine allgemeingültige Aussage über einen Algorithmus treffen zu können.

Es gibt daher keine universell gültige Antwort auf die beiden Fragen, wie Zahl der Rechenschritte oder Eingabegröße gemessen werden sollen. Je nach Anwendung und Art des Problems werden in der Mathematik und Informatik dafür eine ganze Reihe von Modellen verwendet. Wir betrachten hier nur ein sehr einfaches Modell, und werden auch keine mathematisch präzisen Beweise für unsere Laufzeitabschätzungen geben. Sie werden dieses Thema in späteren Vorlesungen noch einmal aufgreifen und vertiefen, z.B. in der *Algorithmischen Diskreten Mathematik*.

Für unser einfaches Modell nehmen wir an, dass alle Rechenoperationen die gleiche, konstante, Zeit benötigen, und ebenso alle Zuweisungen, Vergleiche und andere Operationen[1]. Um die Anzahl der Schritte zu bestimmen, zählen wir also tatsächlich die Anzahl der Operationen, die unser Algorithmus oder unser Programm für eine gegebene Eingabe ausführt. Wenn wir nun zu jeder festen EIngabegröße die Anzahl der Operationen betrachten, die wir im schlechtesten Fall (*worst case*) benötigen, dann können wir die konkrete Eingabe durch ihre *Eingabegröße* ersetzen. Das können wir dann als Funktion $f(n)$ in Abhängigkeit dieser Eingabegröße n schreiben.

Wir müssen allerdings noch sagen, was die *Eingabegröße* sein soll. Auch hier ist es oft von der Problemstellung abhängig, was eine gute Wahl dafür ist. In unserem Beispiel ist es vernünftig, zum Beispiel die Größe der beiden Eingabezahlen zu nehmen. Konkret hängt die Zahl der Iterationen eigentlich nur von der Größe der kleineren Zahl b ab, so dass wir diese als Eingabegröße nehmen wollen. In anderen Problemen können andere Wahlen sinnvoll sein. Wenn wir zum Beispiel nach der kürzesten Verbindung zwischen zwei Haltestellen eines Nahverkehrsnetzes suchen, ist eher die Gesamtzahl der Haltestellen relevant. Bei den Sortierverfahren, die wir später betrachten, könnten wir die Anzahl der zu sortierenden Elemente nehmen.

Die Laufzeitfunktion $f(n)$ für einen konkreten Algorithmus wird in der Regel eine eher komplizierte Funktion sein, die sich in der Regel auch nicht in einer geschlossenen Formel hinschreiben lässt. Für Vergleiche verschiedener Algorithmen wäre das also noch nicht richtig hilfreich. Nun sind wir aber nur am asymptotischen Verhalten der Funktion für große Eingabegrößen interessiert. Eine mögliche Lösung dafür ist es daher, statt der Funktion $f(n)$ selbst eine möglichst einfache Funktion $g(n)$ anzugeben, die sich asymptotisch ausreichend ähnlich verhält. Diese Funktion nehmen wir dann als Bewertung des Algorithmus, und vergleichen diese vereinfachten asymptotischen Funktionen, wenn wir Algorithmen vergleichen wollen.

Mathematisch unterteilen wir dazu die Menge aller Funktionen in Familien von Funktionen, die sich für ausreichend große Funktionswerte ähnlich zu unserer Vergleichsfunktion verhalten. Diese Unterteilung wird nicht disjunkt sein. Wir setzen dafür für eine Funktion $g : \mathbb{N} \to \mathbb{N}$

$$\mathcal{O}(g) := \{f : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}_{>0}, N \in \mathbb{N} : f(n) \leq c \cdot g(n)\ \forall n \geq N\}$$

$\mathcal{O}(g)$ ist die Familie aller Funktionen, die *asymptotisch von g nach oben beschränkt* sind.

[1] Das ist durchaus eine deutliche Vereinfachung, denn die Multiplikation großer Zahlen ist sicher aufwendiger (was Sie schon beim Kopfrechnen feststellen können). Solange alle beteiligten Zahlen ausreichend klein sind, um in wenige Speicherzellen zu passen (und zum Beispiel vom Typ `long` sind), ist es für Prozessoren allerdings tatsächlich unerheblich, wie groß die Zahlen in diesem Bereich wirklich sind, er braucht immer in etwa die gleiche Zeit. Das ändert sich natürlich, wenn die Zahlen diesen beschränkten Bereich überschreiten.

Analog setzen wir

$$\Omega(g) \;:=\; \{f : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}_{>0}, N \in \mathbb{N} : f(n) \geq c \cdot g(n)\ \forall n \geq N\}$$

für die Familie aller Funktionen, die *asymptotisch von g nach unten beschränkt* sind. Sie können sich leicht überlegen, dass zum Beispiel für jedes Polynom f vom Grad d und die Funktion $g(n^k)$ gilt

$$\begin{aligned} f \;&\in \mathcal{O}(g) && \text{für } k \geq d \\ f \;&\in \Omega(g) && \text{für } k \leq d \end{aligned}$$

Wir nehmen dann g als die relevante Funktion für eine Bewertung des Algorithmus.

Wenn Sie nun anfangen, Rechenschritte in den Algorithmen zu zählen, werden Sie zudem feststellen, dass, sobald wir für die erhaltene Funktion f eine gute Vergleichsfunktion g suchen, es keine Rolle mehr spielt, wie viele Schritte in den Schleifen nun konkret ausgeführt werden, solange diese Zahl in jedem Durchlauf konstant ist. Diese konstante Anzahl von Schritten innerhalb der Schleife wirkt sich nur auf die Konstante c aus, die wir in der Definition von $\mathcal{O}(g)$ wählen dürfen.

Beide Algorithmen brauchen nun im ungünstigsten Fall b Schritte bis zum Ergebnis (in beiden Fällen zeigt das die Wahl $a = b + 1$ für jedes b). Wenn $f_1(n)$ und $f_2(n)$ die zugehörigen Laufzeitfunktionen sind, können wir als Vergleichsfunktion daher in beiden Fällen $g(n) = n$ nehmen, und es gilt

$$f_1,\ f_2\ \in \mathcal{O}(g)\,.$$

Beide Algorithmen sind *asymptotisch* und im *ungünstigsten Fall* ähnlich effizient. Man will die Funktion g oft nicht benennen, und schreibt daher verkürzend oft $f_1, f_2 \in \mathcal{O}(n)$ und sagt, die beiden Algorithmen sind in $\mathcal{O}(n)$.

Ist das nun schon die effizienteste Möglichkeit, den größten gemeinsamen Teiler zweier Zahlen zu bestimmen? Neben der Identität

$$\mathrm{ggT}(a, b) \;=\; \mathrm{ggT}(a, a - b)$$

für $a \geq b$, die wir oben schon bemerkt haben, gibt es noch eine weitere. Wenn wir a in der Form

$$a = q \cdot b \;+\; r$$

für positive ganze Zahlen q und r darstellen können, dann gilt auch

$$\mathrm{ggT}(a, b) \;=\; \mathrm{ggT}(b, r)\,.$$

Die kleinste Zahl r, die wir hier verwenden können, ist der Rest der ganzahligen Division von a und b. Sie erfüllt $0 \leq r < b$. In `C` gibt es, wie wir gesehen haben, dafür den Modulooperator `%`, mit dem wir diesen Rest bei der Division leicht ausrechnen können, und wir erhalten `r=a%b`. Wenn wir diese Identität nun verwenden, bekommen wir das folgende Programm.

Programm 9.3: `kapitel_09/ggt_v3.c`

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

int main(int argc, char** argv) {

   long long a = atoll(argv[1]);
   long long b = atoll(argv[2]);
   if ( a < b ) {
     long long c = a;
     a = b;
     b = c;
   }

   long long at = a;
   long long bt = b;
   while ( bt != 0 ) {
      int r = at % bt;
      at = bt;
      bt = r;
   }

   printf("Der groesste gemeinsame Teiler von %lld und %lld ist %lld\n",a,b,at);
   return 0;
}
```

Ist das effizienter als die beiden vorangegangenen? Wir können dafür die folgende Überlegung anstellen. Wir nehmen dafür $a \geq b$ an, was nach den ersten Programmzeilen erfüllt ist.

Wenn nun $b \leq a/2$ ist, dann ist in der nächsten Iteration, in der wir a durch b und b durch den Rest bei der Division ersetzt haben, die dann größere Zahl höchstens $a//2$, ihre Größe hat sich also nach einer Iteration halbiert.

Wenn andererseits $b > a/2$ ist, dann ist der Rest der ganzzahligen Division von a durch b gerade $r := a - b \leq a/2$ und in der nächsten Iteration (in der wieder a durch b und b durch r ersetzt wird) gilt $b \leq a/2$. Nach dem vorangegangenen Fall ist also in der nächsten Iteration die Größe von a wieder halbiert.

In beiden Fällen wissen wir also, dass nach spätestens zwei Schritten die Größe von a halbiert wird. Das Programm bricht ab, wenn $b = 0$ ist. Das passiert, wenn in der vorangegangenen Iteration a durch b teilbar war. Da $a \geq b \geq 0$ ist das spätestens der Fall, wenn $a = 1$ ist. Das ist nach spätestens $2 \cdot \log_2 a$ Iterationen erreicht. Wir haben die Eingabegröße n von b statt a abhängig gemacht, aber da nach einer Iteration a durch b ersetzt wird, bricht der Algorithmus auch nach höchstens $1 + 2 \cdot \log_2 b$ Iterationen ab. Die Laufzeit ist also $\mathcal{O}(\log_2 n)$, und damit erheblich schneller als unsere beiden ersten Varianten!

Bei größeren Zahlen macht sich dieser Unterschied sehr deutlich bemerkbar. Zur Erstellung eines guten Programms gehört daher immer, sich über die verwendeten Methoden Gedanken zu machen und zu überlegen, wie ein Problem effizient gelöst werden kann.

Wir können eine Zeitmessung zum Vergleich auch konkret in unser Programm einbauen, um ein Gefühl für den Unterschied in der Laufzeit zu bekommen. Hier ist die dritte Variante mit Zeitmessung.

Programm 9.4: `kapitel_09/ggt_v3_zeitmessung.c`

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

int main(int argc, char** argv) {

    long long a = atoll(argv[1]);
    long long b = atoll(argv[2]);
    if ( a < b ) {
        long long c = a;
        a = b;
        b = c;
    }

    long long at = a;
    long long bt = b;

    clock_t start, end;
    start = clock();    // Start der Zeitmessung

    while ( bt != 0 ) {
        long long r = at % bt;
        at = bt;
        bt = r;
    }

    end = clock();     // Ende der Zeitmessung
    // Gemessen werden hier Prozessortakte
    // Zur Umwandlung muessen wir also noch durch die Taktrate dividieren
    double time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    printf("Der groesste gemeinsame Teiler von %lld und %lld ist %lld\n",a,b,at);
    printf("Die Berechnung hat %f Sekunden benoetigt\n", time_used);
    return 0;
}
```

Wenn wir analog auch `ggt_v2.c` zu einem Programm `ggt_v2_zeitmessung.c` umschreiben, können wir einen Vergleich machen.

```
./ggt_v2_zeitmessung 123212321232 12321232123212
Der groesste gemeinsame Teiler von 12321232123212 und 123212321232 ist 12
Die Berechnung hat 11.560545 Sekunden benoetigt

./ggt_v3_zeitmessung 123212321232 12321232123212
Der groesste gemeinsame Teiler von 12321232123212 und 123212321232 ist 12
Die Berechnung hat 0.000003 Sekunden benoetigt
```

Die konkrete Zeit hängt dabei natürlich vom Rechner ab, auf dem wir die Programme ausführen. Die gemessene Differenz hängt auch von den konkreten Zahlen ab. Bei der Laufzeit beziehen wir uns auf den *ungünstigsten* Fall, es kann also durchaus sein, dass die Programme auf manchen Eingaben ähnlich schnell sind oder das im Sinne unserer Bewertung schlechtere sogar schneller ist.

Die letzte Variante der Berechnung des größten gemeinsamen Teilers über den Rest bei ganzzahliger Division ist der sogenannte *Euklidische Algorithmus*. Dieser Algorithmus hat eine erweiterte Form, die zusätzlich eine Darstellung des größten gemeinsamen

Teilers als Linearkombination der beiden Zahlen berechnet. Zu jedem Paar ganzer Zahlen a und b gibt es zwei weitere Zahlen x und y, so dass

$$\mathrm{ggT}(a,b) \;=\; x \cdot a \;+\; y \cdot b\,.$$

Diese Darstellung lässt sich leicht aus der umgekehrten Folge der Quotienten q und Reste r der Darstellung $a = q \cdot b + r$ bestimmen, die wir in unserem Programm immer wieder bestimmen. Da wir die Berechnung mit dem Paar (q, r) aus der letzten Iteration beginnen müssten, müssen wir dafür aber alle Paar zwischenspeichern. Das geht leichter, wenn wir unser Programm *rekursiv* schreiben. Was das heißt und wie es geht ist das Thema des nächsten Abschnitts.

Zur Laufzeit sollten wir am Schluss noch eine kleine Anmerkung machen. Prinzipiell ist es eigentlich sinnvoller, die als Eingabegröße nicht die Zahl b sondern die Zahl ihrer Ziffern, also ihre *Kodierungslänge* h, zu betrachten. Da Computer im Binärsystem rechnen, ist das gerade $\log_2 b$. Damit hat die dritte Variante eine Laufzeit, die linear in der Kodierungslänge ist, während die ersten beiden Varianten *exponentielle* Laufzeit haben.

9.2 Rekursion

Wir können am Beispiel des euklidischen Algorithmus auch noch ein grundlegendes algorithmisches Prinzip kennenlernen. Wir haben in der letzten Variante die Identität

$$\mathrm{ggT}(a,b) \;=\; \mathrm{ggT}(b,r)$$

ausgenutzt, wobei r der Rest bei der ganzzahligen Division von a durch b ist. Erst wenn $b = 0$ war, haben wir abgebrochen, denn danach steht rechts und links das gleiche. In unserem Programm haben wir also die Berechnung des größten gemeinsamen Teilers zweier Zahlen immer wieder durch die Berechnung eines anderen größten gemeinsamen Teilers ersetzt, wobei die Größe der beteiligten Zahlen in jedem Schritt gefallen ist, bis auf beiden Seiten das gleiche stand. Diese Formulierung der Berechnung können wir auch direkt in unserem Programm ausnutzen. Dafür packen wir die Berechnung des größten gemeinsamen Teilers in eine eigenen Funktion mit zwei Parametern a und b, die entweder a zurückgibt, wenn $b = 0$ ist, oder sich selbst mit den neuen Parametern b und r aufruft und dann die Rückgabe dieses Aufrufs zurückgibt. Das folgende Programm führt diese Idee aus.

Programm 9.5: `kapitel_09/ggt_rekursiv.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  long long gcd ( long long a, long long b) {
5      return b ? gcd(b,a%b) : a;
6  }
7
8  int main(int argc, char** argv) {
9
10     long long a = atoll(argv[1]);
11     long long b = atoll(argv[2]);
```

```
12      if ( a < b ) {
13        long long c = a;
14        a = b;
15        b = c;
16      }
17
18      long long g = gcd(a,b);
19
20      printf("Der groesste gemeinsame Teiler von %lld und %lld ist %lld\n",a,b,g);
21      return 0;
22  }
```

Programme, oder Algorithmen, bei denen wir eine Funktion aus sich heraus erneut aufrufen, nennt man *rekursiv*. Eine rekursive Formulierung ist für uns oft leichter zu finden und zu verstehen, weil sie der Intuition der meisten Menschen näher ist.

Die Umsetzung des Programms als Maschinencode auf einem Computer durch den Compiler ist allerdings aufwendiger als unsere vorangegangen Ansätze, und das Programm wird oft in dieser Form auch langsamer sein (was bei diesem einfachen Problem des größten gemeinsamen Teilers kaum messbar sein wird).

Ein Grund, warum rekursive Algorithmen länger brauchen, kann man sich recht leicht überlegen. An allen Stellen, an denen wir die Funktion für den größten gemeinsamen Teiler erneut aufrufen, müssen alle Variablenbelegungen, die in dem Moment aktuell sind, zwischengespeichert werden, denn mit dem neuen Aufruf werden auch die Variablen der Funktion mit den neuen Parametern initialisiert. Wenn das Programm jedoch von diesem Aufruf zurückkehrt, erwarten wir natürlich, das auch die Variablenbelegungen wieder so sind, wie sie vor dem Aufruf waren. Für den Compiler heißt das, dass er das Programm an dieser Stelle anweisen muss, vor dem Aufruf einen Bereich im Speicher zu reservieren, dort die Variablenbelegung zu speichern, und nach Rückkehr diesen Speicher wieder zu lesen und die Werte wiederherzustellen. Im alten Ansatz, dem sogenannten *iterativen Ansatz*, brauchen wir diesen zusätzlichen Schritt nicht.

Über den rekursiven Ansatz können wir nun sehr leicht den *erweiterten* Euklidischen Algorithmus implementieren. Dazu müssen wir uns nur überlegen, wie wir bei der Betrachtung der Gleichung

$$\mathrm{ggT}(a,b) \;=\; \mathrm{ggT}(b,r)$$

für den Rest r der ganzzahligen Division von a durch b von der Darstellung des größten gemeinsamen Teilers von b und r als Linearkombination dieser beiden Zahlen zu einer Darstellung von des größten gemeinsamen Teilers von a und b kommen. Nehmen wir an, wir kennen x und y so dass

$$\mathrm{ggT}(b,r) \;=\; x\cdot b \;+\; y\cdot r\,.$$

und die Darstellung $a = q\cdot b + r$ für nichtnegative Zahlen q, r mit $0 \le r < b$. Dann können wir wie folgt umformen.

$$\mathrm{ggT}(a,b) \;=\; \mathrm{ggT}(b,r) \;=\; x\cdot b \;+\; y\cdot r \;=\; x\cdot b \;+\; y\cdot(a - q\cdot b) \;=\; y\cdot a \;+\; (x-q)\cdot b\,.$$

Damit haben wir eine Darstellung in a und b gefunden. Im Fall, dass $b = 0$ ist, ist eine solche Darstellung direkt zu finden, denn $\mathrm{ggT}(a,0) = a = 1\cdot a + 0\cdot b$. Damit können

wir eine rekursive Berechnung des erweiterten Euklidischen Algorithmus schreiben. Das folgende Programm gibt eine Möglichkeit, das in C umzusetzen.

Programm 9.6: kapitel_09/ggt_erweitert.c

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

long long gcd(long long a, long long b, long long* x, long long* y) {
   if (b == 0) {
      return a;
   }

   long long g = gcd(b, a%b, x, y);

   long long temp = *x;
   *x = *y;
   *y = temp - (a/b) * *y;

   return g;
}

int main(int argc, char** argv) {

   long long a = atoll(argv[1]);
   long long b = atoll(argv[2]);
   if ( a < b ) {
     long long c = a;
     a = b;
     b = c;
   }

   long long x = 1,y = 0;
   long long g = gcd(a, b, &x, &y);
   printf("gcd(%lld, %lld) = %lld = %lld*%lld + %lld*%lld\n", a, b, g, x, a, y, b);
   return 0;
}
```

Der erweiterte Euklidische Algorithmus ist ein Beispiel für eine Problemlösung, die sich in rekursiver Form erheblich leichter als Programm schreiben lässt als in iterativer Form. Es ist sicherlich trotzdem eine gute Übung zu versuchen, den Algorithmus ohne Rekursion als Programm in C zu schreiben.

Wir können hier das Programm sogar noch weiter verkürzen und uns die Vertauschung von x und y in jeder Iteration sparen. Das ändert natürlich nicht die Laufzeit, sondern nur die Konstante in der Abschätzung.

Programm 9.7: kapitel_09/ggt_erweitert_v2.c

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

long long ggt(long long a, long long b, long long* x, long long* y) {
   if (b == 0) {
      *x = 0;
      *y = 1;
      return a;
   }

```

```
11      long long g = ggt(b, a%b, y, x);
12      *x = *x - (a/b) * *y;
13
14      return g;
15  }
16
17  int main(int argc, char** argv) {
18
19      long long a = atoll(argv[1]);
20      long long b = atoll(argv[2]);
21      if ( a < b ) {
22          long long c = a;
23          a = b;
24          b = c;
25       }
26
27      long long x, y;
28      long long g = ggt(a, b, &y, &x);
29      printf("ggt(%lld, %lld) = %lld = %lld*%lld + %lld*%lld\n", a, b, g, x, a, y, b);
30      return 0;
31  }
```

In unserem Beispiel rufen die Funktion (höchstens) einmal in ihrem Ablauf sich selbst erneut auf. Das ist nicht zwingend, wir könnten in einem rekursiv formulierten Programm eine Funktion auch mehrfach aus sich heraus wieder aufrufen. Das kann allerdings, wenn wir bei der Umsetzung nicht aufpassen, zu sehr ineffizienten Programmen führen, obwohl der Code sehr klar, kurz und übersichtlich erscheint.

Das wollen wir uns am Beispiel der *Fibonacci-Folgen* ansehen. Die Fibonacci-Folge ist eine Folge $(f_k)_{k\geq 0}$ natürlicher Zahlen, die sich durch $f_0 = f_1 = 1$ und $f_k = f_{k-1} + f_{k-2}$ für $k \geq 2$ ergeben. Die ersten Folgenglieder sind also

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots$$

Die Berechnung eines Folgenglieds erfordert hier die Kenntnis von *zwei* vorangegangenen Folgengliedern.

Diese einfache Rekursionsformel $f_k = f_{k-1} + f_{k-2}$ suggeriert die folgende auf den ersten Blick einfache Umsetzung in ein rekursives Programm.

Programm 9.8: `kapitel_09/fibonacci.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  long fibonacci(int n) {
5    //printf("Aufruf mit %d\n",n);
6    if ( n <= 1) {
7      return 1;
8    }  else {
9      return fibonacci(n-1)+fibonacci(n-2);
10   }
11 }
12
13 int main(int argc, char** argv) {
14
15   int n = atoi(argv[1]);
16
```

```
17    printf("Die %d-te Fibnaccizahl ist %ld\n",n,fibonacci(n));
18    return 0;
19  }
```

Für kleine Wahlen von n funktioniert diese Programm gut. Aber schon für die 50ste Fibonaccizahl, die mit 20365011074 noch nicht sonderlich groß ist, braucht das Programm ungefähr eine Minute.

Wir können das Problem erkennen, wenn wir die auskommentierte Ausgabe des Funktionsparameters bei jedem Aufruf der Funktion `fibonacci` ins Programm aufnehmen. Die Ausgabe für $n = 5$ ist dann

```
Aufruf mit 5
Aufruf mit 4
Aufruf mit 3
Aufruf mit 2
Aufruf mit 1
Aufruf mit 0
Aufruf mit 1
Aufruf mit 2
Aufruf mit 1
Aufruf mit 0
Aufruf mit 3
Aufruf mit 2
Aufruf mit 1
Aufruf mit 0
Aufruf mit 1
Die 5-te Fibonaccizahl ist 8
```

Die Funktion wird also mehrfach für jeden Parameter aufgerufen, und zwar umso öfter, je kleiner dieser ist. Wenn wir ins Programm schauen, sollte auch klar werden, warum das so ist.

Immer, wenn wir die Berechnung der k-ten Fibonaccizahl f_k starten, berechnen wir die $(k-1)$-ste und $(k-2)$-te. Diese Berechnungen wissen nichts voneinander. Auch wenn wir eigentlich bei der Bestimmung von f_{k-1} die $(k-2)$-te Zahl f_{k-2} schon bestimmt haben, verwerfen wir dieses Ergebnis bei der Rückgabe an den übergeordneten Aufruf, und starten die Berechnung danach von vorne! Zur Berechnung von f_n brauchen wir also ungefähr 2^n Aufrufe der Funktion `fibonacci(k)`.

Bei der iterativen Berechnung der ersten Folgenglieder oben ist das nicht aufgefallen, denn wir haben uns, ohne darauf explizit hinzuweisen, die vorangegangenen Folgenglieder gemerkt und haben auf dieses Wissen bei der Berechnung des nächsten Folgenglieds zurückgegriffen. Das können wir natürlich auch in unserem Programm machen, indem wir in der Funktion `fibonacci(k)` eine Liste der bisher berechneten Werte vorhalten. Dafür können wir zum Beispiel eine Variable als `static` deklarieren, so dass ihre Werte über verschiedene Programmaufrufe hinweg erhalten bleibt.

Das könnte zum Beispiel wie im nächsten Programm aussehen. Hier geben wir der Funktion `fibonacci(n)` eine zusätzliche Liste `fib` und eine Variable, die die höchste bisher bestimmte Fibonaccizahl angibt, mit, die beide als `static` definiert sind. Um die Größe der Liste anzugeben, führen wir einen Parameter `MAX_FIB` ein, der die größte Fibonaccizahl angibt, die unser Programm bestimmen kann (tatsächlich können wir nicht bis $n = 1000$ gehen, da f_n größer ist als die größte Zahl, die sich mit dem Typ `long` darstellen lässt).

Programm 9.9: `kapitel_09/fibonacci_mit_dict.c`

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

#define MAX_FIB 1000

long fibonacci(int n) {
  //printf("Aufruf mit %d\n",n);

  static long fib[MAX_FIB] = { 1, 1} ;
  static int max_index = 1;

  if (n > max_index ) {
    fib[n] = fibonacci(n-1)+fibonacci(n-2);
    max_index = n;
  }

  return fib[n];
}

int main(int argc, char** argv) {

  int n = atoi(argv[1]);

  printf("Die %d-te Fibnaccizahl ist %ld\n",n,fibonacci(n));
  return 0;
}
```

Es gibt viele weitere Möglichkeiten, mit der *rekursiven Explosion* bei mehrfachem Aufruf einer Funktion aus sich heraus umzugehen. Sie sind eingeladen, weitere Varianten zu finden, die f_n auch für große n effizient berechnen. Wir werden das Problem erneut in einem anderen Kontext aufgreifen, wenn wir uns damit beschäftigen, wie wir Programme schreiben, die auch mit ganzen Zahlen umgehen können, die größer sind als der maximale Wert für `long long`. Solche Zahlen wollen wir als Mathematiker, wenn wir exakt rechnen wollen, in der Regel nicht als `double` abspeichern. Wir brauchen also eine andere Idee für die Darstellung großer Zahlen. Und dann brauchen wir, nebenbei, auch eine andere Idee, wie wir die vorher berechneten Werte der Fibonacci-Folge vorhalten, ohne zu viel Speicher zu verbrauchen.

Version vom 10. Februar 2022

10 Dynamische Speicherverwaltung

An zwei Stellen haben wir schon festgestellt, dass die feste Zuweisung von Speicher bei der Variablendeklaration schon zum Zeitpunkt des Kompilierens unsere Flexibilität beim Programmieren einschränkt:

- Wir können nur Listen definieren, deren Länge zum Zeitpunkt der Definition feststeht. Daher mussten wir oft die Länge überschätzen und als Größe eine Länge wählen, die für alle Anwendungsfälle ausreicht. Dadurch haben wir oft Speicher *verschwendet*, wenn die gewählte Länge nicht erreicht wurde, oder mussten abbrechen und nachjustieren, wenn mehr Daten in der Liste gespeichert werden sollten als wir vorgesehen haben.
- Zeigervariablen können nur auf den Speicherbereich existierender Variablen zeigen, wir können bisher keinen Speicherbereich für eine Zeigervariable reservieren, wo sie dann ihren Wert ablegen kann.

Um diese beiden Einschränkungen zu umgehen, wollen wir uns in diesem Abschnitt damit beschäftigen, wie wir selbst einen Bereich im Speicher reservieren, verwalten, vergrößern und verrkleinern, und am Ende, wenn wir den Speicher nicht mehr benötigen, wieder freigeben können.

10.1 Speicher reservieren

Die zwei wesentlichen Anweisungen, die wir dafür benötigen, sind die Funktionen **`malloc`** (für *memory allocation*) und **`free`**, wobei die erste einen Speicherbereich reserviert und uns einen Zeiger auf diesen Bereich zurückgibt, und der zweite einen reservierten Bereich wieder freigibt. Beide Anweisungen operieren direkt auf dem Speicher und wissen nichts über den Speicherbedarf der einzelnen Variablentypen. Es ist unsere Aufgabe als Programmierer*in, einen ausreichend großen Bereich für unsere Daten zu reservieren. Die beiden Funktionen sind im Standardheader `stdlib.h` definiert, den wir daher von jetzt an in der Regel in unsere Programme über

```
1 #include <stdlib.h>;
```

einbinden müssen.

Bisher wissen alle Zeiger, die wir definiert haben, auf welchen Typ von Daten sie zeigen (zum Beispiel speichert ein `int *` die Adresse des Anfangs eines Speicherbereichs, der ausreichend groß ist um ein `int` darin zu speichern, und alle nachfolgenden Adressen, die noch in diesen Bereich zeigen, werden nicht mehr vergeben). Wenn wir davon unabhängig sein wollen, brauchen wir einen Zeigertyp, der keinem konkreten Datentyp zugeordnet ist. Dafür gibt es den Typ `void *`, der eine Adresse ohne zugehörige Typinformation speichern kann.

Version vom 10. Februar 2022

Damit sieht die Funktion **malloc** so aus:

```
void * malloc(size_t groesse)
```

wobei **size_t** auf jedem System ein vorzeichenfreier ganzzahliger Typ ist, der mindestens eine Länge von 16 Bit hat, also Zahlen im Bereich von 0 bis $2^{32}-1$ darstellen kann. In der Regel wird das ein `unsigned int` sein. Er wurde eingeführt um auf allen Systemen einen ausreichend großen Typ für Speichergrößen zu haben ohne die systemspezifische Größe von `int` kennen zu müssen.

Um die Rückgabe von **malloc** einem Typzeiger zuzuweisen, müssen wir den zurückgegebenen Zeiger *umwandeln* (*casten*). Das passiert in der Regel implizit, also ohne dass wir den Compiler explizit dazu anweisen, wenn wir den neuen Speicher auf eine typisierte Zeigervariable zuweisen. Damit können wir einfach

```
char * char_zeiger = malloc(1);
int * int_zeiger = malloc(4);
```

schreiben, um einen Zeiger `char_zeiger` auf einen Speicherplatz für ein `char` und einen Zeiger `int_zeiger` auf einen Speicherplatz für ein `int` zu definieren. Wir können dann auf diesen Speicherbereich zuweisen:

```
*char_zeiger = 'a';
*int_zeiger = 42;
```

Wir können die Typumwandlung aber auch epxlizit vom Compiler fordern. Das geht, indem wir den Zieltyp in runden Klammern vor die Funktion stellen. Im Fall unseres `char`-Zeigers geht das auf die folgende Weise.

```
char * char_zeiger = (char *)malloc(1);
```

Diese *Typumwandlung* (*cast*) ist ein neuer Operator. Wir haben ihn schon ohne Erklärung in die Tabelle Table 5.1 aufgenommen. Er fällt in die Prioritätsstufe 2.

Prinzipiell kann man mit einer solchen Typumwandlung beliebige Typen ändern. In den meisten Fällen wird man aber kein sinnvolles Ergebnis erhalten und der Übersetzer wird uns (und kann uns auch nicht) in allen Fällen vor einer unzulässigen Umwandlung waren. Eine solche Umwandlung verändert nur die im Programm vorgehaltene Information, wie der Speicherinhalt interpretiert werden soll. Der Speicherbereich wird dabei nicht neu angeordnet, sondern, sinnbildlich, nur mit einem neuen Label versehen, was er angeblich enthält, auch wenn es vielleicht durch eine Umordnung eine sinnvolle Interpretation der Typumwandlung gäbe. Falls es zwischen zwei Datentypen eine solche gibt, die jedoch eine neue Anordnung der Daten erfordert, müssen wir diese Umwandlung selbst explizit schreiben.

Der Compiler akzeptiert (fast) jede Typumwandlung ohne Fehlermeldung. Wir sollten eine solche Umwandlung also nur vornehmen, wenn wir sicher sind, dass die neue Interpretation des Speichers zulässig ist und es keinen anderen Weg der Umwandlung gibt. Eine solche Stelle ist die Funktion **malloc**, die uns einen Speicherbereich ohne jedes Label für ihren Inhalt reserviert, und wir ihn mit einem Label versehen müssen, wenn wir ihn verwenden wollen.

Im Beispiel oben war die explizite Umwandlung unnötig, da der Compiler dies implizit für uns erledigt, wenn wir den `void`-Zeiger auf einen `char`-Zeiger zuweisen. Wir bräuchten diesen expliziten *cast* jedoch zum Beispiel, wenn wir die Rückgabe von **malloc** nicht

auf einen typisierten Zeiger zuweisen wollen (oder können), aber an dem Sepicherort eine Variable ablegen wollen. Dann könnten wir einen Wert vom Typ `int` auf die folgende Weise speichern und anschließend ausgeben.

```
void * m = malloc(4);
*(int *)m = 4;
printf("%d\n", *(int *)m);
```

Den expliziten *cast* für die Rückgabe von **`malloc`**, wenn wir auf einen typisierten Zeiger zuweisen, wollen wir in der Regel nicht machen, sondern ausnutzen, dass der Compiler das implizit für uns erledigt. Das ist in der Regel besser lesbar und führt zu weniger Fehlern.

Auch die epxlizite Deklaration von `void`-Zeigern und anschliessendes *casten*, wenn darauf zugegriffen werden soll, wie im Beispiel oben, sollte man vermeiden, solange es einen anderen sinnvollen Weg gibt. Hier an einer Stelle einen *cast* zu vergessen führt in der Regel zu schwierig zu findenden Fehlern.

10.2 Speichergröße

Insbesondere, wenn wir selbst Speicher reservieren wollen und Typumwandlung auf Zeiger anwenden, müssen wir darauf achten, dass der zugehörige Speicherbereich die passende Größe hat. Um herauszufinden, wie viel Speicher wir für unsere Daten reservieren müssen, können wir den `sizeof`-Operator benutzen. Er gibt uns für alle Grundtypen (und, wie wir später sehen, auch für komplexere Datentypen) die notwendige Speichergröße zurück. Mit dem folgenden Code können wir einige Beispiele ausgeben:

Programm 10.1: `kapitel_10/speichergroesse.c`

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

int main() {
   int    i1 = 0 ,  i2 = 0 ,  i3 = 0;
   char   c1 = 1,   c2 = 1 ,  c3 = 1;
   double f1 = 1.0, f2 = 2.0, f3 = 2.0;
   int arr[3] = {0} ;
   long long l = 0;

   printf("Adresse von a: %p\n",(void *)&i1);
   printf("Adresse von b: %p\n",(void *)&i2);
   printf("Adresse von c: %p\n",(void *)&i3);
   puts("--------------------");
   printf("Adresse von c1: %p\n",(void *)&c1);
   printf("Adresse von c2: %p\n",(void *)&c2);
   printf("Adresse von c3: %p\n",(void *)&c3);
   puts("--------------------");
   printf("Adresse von f1: %p\n",(void *)&f1);
   printf("Adresse von f2: %p\n",(void *)&f2);
   printf("Adresse von f3: %p\n",(void *)&f3);
   puts("--------------------");
   for ( int i = 0; i < 3; ++i ) {
      printf("Adresse von arr[%d]: %p\n",i,(void *)&arr[i]);
```

```
25      }
26      puts("-------------------");
27      printf("Adresse von l: %p\n",(void *)&l);
28
29      puts("-------------------");
30      printf("Speichergroesse von int:        %lu\n", sizeof(int));
31      printf("Speichergroesse von char:       %lu\n", sizeof(char));
32      printf("Speichergroesse von double:     %lu\n", sizeof(double));
33      printf("Speichergroesse von long long:  %lu\n", sizeof(long long));
34      printf("Speichergroesse von int*:       %lu\n", sizeof(int*));
35      printf("Speichergroesse von char*:      %lu\n", sizeof(char*));
36      printf("Speichergroesse von double*:    %lu\n", sizeof(double*));
37      printf("Speichergroesse von long long*: %lu\n", sizeof(long long*));
38
39      return 0;
40  }
```

Seine Ausgabe könnte in etwa so aussehen:

```
Adresse von a: 0x7ffeed9e7394
Adresse von b: 0x7ffeed9e7390
Adresse von c: 0x7ffeed9e738c
-------------------
Adresse von c1: 0x7ffeed9e738b
Adresse von c2: 0x7ffeed9e738a
Adresse von c3: 0x7ffeed9e7389
-------------------
Adresse von f1: 0x7ffeed9e7380
Adresse von f2: 0x7ffeed9e7378
Adresse von f3: 0x7ffeed9e7370
-------------------
Adresse von arr[0]: 0x7ffeed9e739c
Adresse von arr[1]: 0x7ffeed9e73a0
Adresse von arr[2]: 0x7ffeed9e73a4
-------------------
Adresse von l: 0x7ffeed9e7368
-------------------
Speichergroesse von int:        4
Speichergroesse von char:       1
Speichergroesse von double:     8
Speichergroesse von long long:  8
Speichergroesse von int*:       8
Speichergroesse von char*:      8
Speichergroesse von double*:    8
Speichergroesse von long long*: 8
```

Wir sehen anhand der Adressen, dass der Speicher für aufeinanderfolgend definierte Variablen nacheinander vergeben worden ist, in diesem Fall absteigend in der Adresse. Denken Sie daran, dass die Adressen als hexadezimale Zahlen geschrieben sind, mit den Buchstaben a bis `f` für die Stellenwerte `10` bis `15`, zwischen den Adressen *0x7ffeed9e738c* und `0x7ffeed9e7390` liegen also die Adressen `0x7ffeed9e738d`, `0x7ffeed9e738e` und `0x7ffeed9e738f`.

Die Adressen für die drei Variablen vom Typ `char` sind direkt anschließend an die vom Typ `int`, während zu den drei Variablen vom Typ `float` eine Adresse ausgelassen wurde. Speicherzugriff auf durch acht teilbare Adressen ist aufgrund der Prozessor- und Speicherarchitektur in den zur Zeit allgemein üblichen Computern meistens schneller,

daher wird manchmal ein Speicherbereich freigelassen, um eine solche Adresse zu wählen. Das System hätte die Möglichkeit, diese freien Teile später an weitere Variablen zu vergeben, die dort noch hineinpassen. In unserem Fall könnte also ein weiteres `char` die Adresse `0x7ffeed9e7388` erhalten. Wir können uns nicht darauf verlassen, dass Speicher bis auf solche Lücken zur Ausrichtung an Blöcken aufeinanderfolgend vergeben wird.

Anhand der Speichergrößen sehen wir, dass auf dem System, auf dem das Programm ausgeführt wurde, ein `char` nur 1 Byte, ein `int` 4 und ein `long long` 8 Byte belegen. Alle Adressen benötigen 8 Byte, um den gesamten Adressraum abzudecken. Diese Speichergröße einer Adresse ist unabhängig davon, was an der Adresse gespeichert werden soll. Das ist insbesondere nützlich, um Listen mit Einträgen variabler Länge zu erzeugen. Hier speichern wir dann nur einen Zeiger auf den Inhalt in der Liste. Dadurch haben alle Einträge der Liste erst einmal die gleiche Länge (damit zum Beispiel Zeigerarithmetik funktionieren kann), und der eigentliche Inhalt kann an einer anderen Stelle im Speicher abgelegt werden.

Mit dieser Funktion können wir nun einen passenden Speicherbereich für eine Variable des gewünschten Typs reservieren.

```
1 int * int_zeiger = malloc(sizeof(int));
```

Für eine Liste müssen wir noch mit der Zahl der Elemente multiplizieren. Mit

```
1 int * liste = malloc(20*sizeof(int));
```

erhalten wir eine Liste für 20 Einträge vom Typ `int`. Der Speicherbereich ist dadurch nicht initialisiert. Wenn wir also eine Anfangsbelegung haben wollen, müssen wir diese anschliessend zuweisen (zum Beispiel mit `*int_zeiger=10;` für unseren Zeiger aus dem ersten Beispiel). Falls nicht mehr genug Speicher zur Verfügung steht um unsere Anfrage zu erfüllen, gibt die Funktion `NULL` zurück. Wie bei Dateioperationen sollte überprüft werden, dass die Ausführung von `malloc` erfolgreich war. Solche Speicherfehler sind sonst sehr schwer zu finden.

```
1 int * liste = malloc(20*sizeof(int));
2 if ( liste == NULL ) {
3   printf("Nicht genug Speicher\n");
4   exit(1);
5 }
```

Für Listen gibt es noch eine Variante `calloc` von `malloc`, die statt einer Gesamtgröße die Anzahl der Elemente und die Größe eines einzelnen Elements nimmt, und jeden Eintrag in der Liste mit Null initialisiert.

```
1 int * liste1 = malloc(10*sizeof(int));
2 for ( int i = 0; i < 10; ++i) {
3   liste1[i] = 0;
4 }
5
6 int * liste2 = (int *)calloc(10, sizeof(int)); // liste1 und liste2
7                                                 // sind (inhaltlich) gleich
```

10.3 Speicher freigeben

Speicherbereiche, die wir selbst mit **malloc** oder **calloc** angefordert haben, müssen wir auch selbst wieder freigeben. Das geht mit der Funktion **free**.

```
int * liste = (int *)malloc(10*sizeof(int));
// liste verwenden
free(liste);
```

Das muss erfolgen, solange wir noch einen Zeiger haben, der auf den Speicherbereich zeigt, in der Regel also innerhalb des gleichen Bereichs. Andernfalls wird beim Verlassen des Bereichs der Zeiger gelöscht, und wir haben keine Möglichkeit mehr, auf den Speicherbereich zuzugreifen oder den Speicherbereich noch freizugeben. Wenn Programmen Speicher vom System zugeordnet ist, auf den das Programm nicht mehr zugreifen kann, spricht man von einem *Speicherleck* (*memory leak*).

Insbesondere bei Programmen, die einen hohen Speicherbedarf haben, kann das problematisch sein und zum Abbruch des Programms wegen Speichermangels führen, obwohl genügend Speicher vorhanden wäre, wenn unbenutzter Speicher zurückgegeben worden wäre.

Erst am Ende des Programms wird (dann vom Betriebssystem, nicht vom Programm), der für unser Programm reservierte und noch nicht wieder freigegebene Speicher automatisch wieder freigegeben.[1] Es ist aber sinnvoll und übliche Praxis, trotzdem am Ende allen reservierten Speicher in unserem Programm freizugeben und das nicht dem System zu überlassen.

Die einzige Möglichkeit, den Zugriff auf einen Speicherbereich über die Gültigkeit des Bereichs, in dem er angefordert wurde, hinaus zu behalten ist, rechtzeitig einen außerhalb des Bereichs definierten Zeiger auf diesen Bereich zeigen zu lassen. Das kann neben Variablen aus umschließenden Bereichen (oder globalen Variablen) bei Funktionen auch durch Rückgabe eines Zeigers auf den Bereich erfolgen. In diesem Fall müssen wir als Programmierer*innen daran denken, dass in der Funktion Speicher reserviert und uns ein Zeiger darauf übergeben wurde, mit dem wir später diesen Speicher wieder freigeben müssen.

```
int * platz_fuer_int() {
  int * p = (int *)malloc(sizeof(int));
  return p;
}

int main() {
  int * p = platz_fuer_int();
  *p = 10;

  free(p);
```

Es gibt viele Anwendungen, in denen es sinnvoll ist, einen Zeiger auf einen initialisierten Speicherbereich zurückzugeben. Auch viele Funktionen der Standardbibliothek machen das. Insbesondere in diesem Fall, oder wenn wir Bibliotheken benutzen, die

[1] Wobei das in der Regel nur heißt, dass das System registriert, dass der Speicher wieder für andere Prozesse verfügbar ist, in der Regel wird der *Inhalt* des Speichers nicht verändert. Programme, die diesen Speicher also im Anschluss zugewiesen bekommen, können möglicherweise die Daten aus unserem Programm auslesen.

nicht von uns geschrieben sind, ist es aber schwierig zu erkennen, ob wir nach Verwendung der Funktion Speicher freigeben müssen. Dem Funktionsaufruf können wir das nicht ansehen. Daher sollte diese Notwendigkeit beim Schreiben solcher Funktionen in die Dokumentation geschrieben werden, und es ist Aufgabe der Anwenderin oder des Anwenders, daran zu denken, bei Funktionen, die einen Zeiger zurückgeben, in der Dokumentation nachzusehen, ob anschließend Speicher freigegeben werden muss. Viele Bibliotheken bieten dafür spezielle Funktionen an, weil, wie wir später auch selbst sehen, es oft vorkommt, dass vor einem `free(p)` auf den uns übergebenen Zeiger erst noch andere Speicherbereiche freigegeben werden müssen. Das ist zum Beispiel der Fall, wenn `p` ein Zeiger auf eine Liste von Zeigern ist, also zum Beispiel vom Typ `char * argv[n]`. Wenn wir nun `p` zu früh freigeben, dann können wir den Speicher an `p[0]` bis `p[n-1]` nicht mehr erreichen. Diese Bereiche müssen vor `p` freigegeben werden.

Auf der anderen Seite haben wir hier auch eine häufige Quelle von Fehlern. Da wir Adressen beliebig kopieren können, ist es möglich, mehr als einen Zeiger auf den gleichen Speicherbereich zu haben. Sie werden später merken, dass diese Möglichkeit auch sehr nützlich ist und häufig vorkommt. Wir können nun jeden der Zeiger verwenden, um den Speicherbereich wieder ans System zurückzugeben, wir dürfen das allerdings nur genau ein einziges Mal machen.

```
1  int * p1 = malloc(sizeof(int));
2  int * p2 = p1;
3  *p2 = 10;
4  printf("%d\n",*p1); // schreibt 10 ins Terminal
5
6  free(p2);
7  // free(p1) // Fehler, der Speicher wurde schon freigegeben
```

In einfachen Programmen ist es beim Lesen leicht ersichtlich, ob ein Speicherbereich schon zurückgegeben wurde. Wenn wir die Reservierung, und vielleicht auch die Freigabe, an eine Funktion delegieren, kann es aber sein, dass wir beim Schreiben oder Lesen eines Programms nicht mehr erkennen, ob noch Speicher freigegeben werden muss. Hier hilft oft nur der Blick in die Dokumentation. Insbesondere, wenn Speicherreservierung in Verzweigungen oder Schleifen erfolgt, kann es sein, dass solche Freigabefehler erst spät auffallen, weil sie vielleicht nur in einem bestimmten Zweig des Programmablaufs auftreten.

10.4 Speicher vergrößern oder verkleinerm

Mit der Anweisung `realloc` können wir einen reservierten Speicherbereich verkleinern oder vergrößern,

```
1  void *realloc(void *zeiger, size_t groesse)
```

wobei `zeiger` auf den Bereich zeigt, der verändert werden soll, und `groesse` die neue Größe ist. Die Funktion gibt einen neuen Zeiger zurück, der üblicherweise direkt dem übergebenen Zeiger zugewiesen wird. Wenn wir auf einen anderen Zeiger zuweisen, dürfen wir den übergebenen Zeiger nicht mehr dereferenzieren, da wir nicht wissen, und auch nicht überprüfen können, ob der Zeiger noch auf einen unserem Programm zugeordneten Speicherbereich zeigt.

Version vom 10. Februar 2022

```c
int * liste = malloc(10*sizeof(int));
liste = realloc(liste, 20*sizeof(int) );
```

Beim Verkleinern wird der Anfang des Speicherbereichs beibehalten und der nicht mehr benötigte Bereich am Ende freigegeben. Insbesondere wird nichts kopiert. Speicherbereiche, die mit **malloc** oder **realloc** reserviert werden, sind allerdings immer zusammenhängend (sonst könnte z.B. die Zeigerarithmetik nicht funktionieren). Daher kann es, wenn der Bereich vergrößert wird, notwendig sein, einen vollständig neuen Bereich zu reservieren, die alten Daten dorthin zu kopieren, und den alten Bereich freizugeben, wenn am Ende des alten Bereichs nicht ausreichend freier Speicher vorhanden ist. Solange der alte Bereich aber erweitert werden kann, wird auch auf diese Weise erweitert und es muss nichts kopiert werden.

Daher ist es meistens effizienter, Speicher mit **realloc** zu verändern als selbst mit **malloc** einen passenden größeren Bereich anzufordern, die Daten zu kopieren, und den alten Bereich freizugeben. Falls es doch erforderlich sein sollte, einen Speicherbereich zu kopieren, steht uns die Funktion `memcpy` zur Verfügung, die einen angegebenen Speicherbereich an eine andere Stelle kopiert, ohne sich um die Struktur des Inhalts oder den Typ zu kümmern.

```c
  void * memcpy(void * ziel, const void * quelle, size_t n)
```

Die Funktion kopiert `n` Bytes des Speichers ohne Information über die dort gespeicherten Datentypen.

Eine typische Anwendung von **realloc** werden wir sehen, wenn wir Stapel und Warteschlangen, also Listen variabler Länge, bei denen Elemente nur am Anfang oder Ende entfernt oder eingefügt werden, betrachten.

Die Möglichkeit, dynamisch Speicher anzufordern ist insbesondere auch beim Einlesen der Argumente bei Programmstart interessant. Wenn hier eine variable Anzahl von Parametern (z.B. eine Liste von Zahlen, die wir sortieren wollen) übergeben wird, können wir zuerst das Argument **argc** an **main** auslesen und anschließend eine Liste passender Größe mit **malloc** anfordern. Hier ist ein Beispiel.

Programm 10.2: `kapitel_10/allocate_input.c`

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

int main(int argc, char** argv) {

  int n = argc-1;
  int* a = malloc(n*sizeof(int));

  for(int i = 0; i < n; ++i ) {
    a[i] = atoi(argv[i+1]);
  }

  for(int i = 0; i < n; ++i ) {
    printf("%d ",a[i]);
  }
  printf("\n");

  free(a);

```

```
20    return 0;
21  }
```

Damit ergibt sich

```
gcc allocate_input.c -o allocate_input
./allocate_input 1 2 3 4 5 6
1 2 3 4 5 6
```

In diesem Beispiel hätten wir natürlich auch einfach

```
1    int a[n];
```

deklarieren können, um unsere Liste zu erhalten. Wenn wir die Daten allerdings in eine Liste von Zeigern speichern wollen, dann geht dieser Ansatz nicht mehr. Im folgenden Beispiel nehmen wir zwar wieder eine Liste von `int`, für die es einfachere Lösungen gibt, aber das Programm zeigt trotzdem das Prinzip des Ansatzes. Wenn wir später eigene zusammengesetzte Datentypen definieren können, kommen wir um diese oder ähnliche Methoden nicht mehr herum.

Programm 10.3: `kapitel_10/allocate_input_zeiger.c`

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

int main(int argc, char** argv) {

  int n = argc-1;
  int** a = malloc(n*sizeof(int *));

  for(int i = 0; i < n; ++i ) {
    a[i] = malloc(sizeof(int));
    *a[i] = atoi(argv[i+1]);
  }

  for(int i = 0; i < n; ++i ) {
    printf("%d ",*a[i]);
  }
  printf("\n");

  for ( int i = 0; i < n; ++i ) {
    free(a[i]);
  }
  free(a);

  return 0;
}
```

In diesem Programm müssen wir daran denken, dass wir, bevor wir den Speicher an der Adresse der Variablen `a` (die Liste von Zeigern auf `int`) freigeben, die Speicherbereiche, auf die die einzelnen Einträge der Liste zeigen, freigeben müssen. Nach der Freigabe von `a` hätten wir darauf keinen Zugriff mehr.

10.5 Speicherverwaltung über Funktionen

Die Reservierung von Speicher und eine möglicherweise notwendige Initialisierung wird oft in einer Funktion zusammengefasst, da das zum einen häufig gebraucht wird, und zum anderen sonst die Lesbarkeit des Codes einschränkt, wenn diese Schritte immer wieder auftauchen. Dabei kann eine solche Funktion entweder einen Zeiger auf den neuen Speicherbereich zurückgeben, oder einen Zeiger als Argument annehmen und ihm den Speicherbereich zuweisen.

In der zweiten Variante müssen wir allerdings daran denken, die *Adressse* des Zeigers an die Funktion zu übergeben. Da bei Übergabe eine Variable kopiert wird, würden wir sonst nur die Kopie des Zeigers auf einen freien Speicherbereich zeigen lassen. Diese Kopie würde beim Verlassen der Funktion gelöscht, und der Speicher wird unerreichbar. Hier ist ein Beispiel für beide möglichen Varianten.

Programm 10.4: `kapitel_10/function_alloc_int.c`

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

int* get_int_return() {
    int * a = malloc(sizeof(int));
    *a = 438;
    return a;
}
void get_int_reference(int ** a) {
  *a = malloc(sizeof(int));
  **a = 439;
}

void delete_int(int * a) {
    free(a);
}

int main() {

  int *ptr = NULL;

  ptr = get_int_return();
  printf("ptr zeigt auf %p mit Wert %d\n", ptr, *ptr);

  delete_int(ptr);

  get_int_reference(&ptr);
  printf("ptr zeigt auf %p mit Wert %d\n", ptr, *ptr);

  delete_int(ptr);
  return 0;
}
```

Die Ausgabe könnte dann so aussehen:

```
ptr zeigt auf 0x7ff032c058f0 mit Wert 438
ptr zeigt auf 0x7ff032c05900 mit Wert 439
```

Wir haben in dem Programm auch schon die Freigabe des Speichers in eine Funktion ausgelagert. Das ist im Falle eines `int` eher nicht nötig, aber bei manchen komplexeren

Datenstrukturen, die wir später sehen, könnte es notwendig sein, vor der Freigabe des Speichers noch weitere Schritte zu unternehmen. Dann ist es sinnvoll, das in einer eigenen Funktion abzutrennen.

Wir sollten nicht vergessen, auch wirklich allen Speicher, den wir reserviert haben, wieder freizugeben. Daher wird die Funktion `delete_int(ptr)` auch vor der zweiten Zuweisung aufgerufen. Falls der mit dem ersten Aufruf reservierte Speicher an dieser Stelle noch nicht freigegeben ist, verlieren wir jeden Zugriff darauf und können ihn nicht mehr freigeben. Diese Mehrfachverwendung von Zeigern ohne vorherige Freigabe des Bereichs, auf den sie zeigen, ist eine häufige, und schwer zu findende Ursache von Speicherlecks.

Version vom 10. Februar 2022

11 Sortieren

Eine der häufigsten algorithmischen Aufgaben, denen man beim Programmieren begegnet, ist das Sortieren einer Liste von Daten anhand eines gegebenen Sortierkriteriums. Im einfachsten Fall ist das eine Liste ganzer Zahlen, die wir aufsteigend oder absteigend sortieren wollen. Mit den gleichen Methoden könnten wir jedoch auch eine Liste von Kontaktdaten zum Beispiel alphabetisch nach den Nachnamen sortieren wollen.

Sortieren tritt oft auch als vorbereitender Schritt für andere algorithmische Aufgaben auf. Wenn Sie zum Beispiel für eine gegebenen Liste ganzer Zahlen wissen wollen, ob eine bestimmte Zahl schon in der Liste enthalten ist, können Sie das wesentlich schneller feststellen, wenn diese Liste sortiert ist (nämlich in einer Laufzeit der Ordnung $\mathcal{O}(log\, n)$ statt $\mathcal{O}(n)$, wenn n die Länge der Liste ist).

Für das Sortieren von Listen gibt es eine ganze Reihe von Algorithmen, aus denen wir für unsere Anwendung einen jeweils geeigneten auswählen können. In nächsten Abschnitt wollen wir zwei einfache Sortieralgorithmen vorstellen, die Algorithmen *InsertionSort* und *BucketSort*. Vom ersten Algorithmus, *InsertionSort*, existieren viele Varianten, die sich je nach Problemstellung anbieten, und auch einige nah verwandte Algorithmen, die die algorithmische Idee von *InsertionSort* abwandeln, wie zum Beispiel *SelectionSort* oder *BubbleSort*.

Im zweiten Abschnitt diskutieren wir dann mit *MergeSort* ein komplizierteres, rekursives Verfahren, das allerdings für große Listen dann auch deutlich effizienter ist.

11.1 Einfaches Sortieren

Die Idee des ersten Algorithmus kennen Sie vielleicht von Kartenspielen. Nach dem Austeilen der Karten haben Sie einen ungeordneten, verdeckten Kartenstapel vor sich liegen. Von diesem Stapel nehmen Sie nur der Reihe nach eine Karte nach der anderen auf und sortieren Sie in den Fächer der Karten, die Sie schon in der Hand halten, und die in sich sortiert sind, an der Stelle ein, den die Karte in Ihrer Sortierung unter den bisher aufgenommenen Karten einnimmt. Die erste Karte, die Sie dabei aufnehmen, ist, als Liste mit nur einem Element, immer sortiert. Damit haben wir eine Anfangssituation mit einem in sich sortierten Kartensatz auf der Hand. Dieser Ansatz führt zum Algorithmus *InsertionSort*.

In unserem Programm können wir die zu sortierenden Daten natürlich nicht auf einen Haufen vor uns legen. Wir bekommen hier die Eingabe in Form einer unsortierten Liste, und wollen diese Liste sortiert zurückgeben. Dabei wollen wir keine neue Liste anlegen, sondern die Sortierung durch Vertauschen von Elementen in der Liste erreichen.

Dazu gehen wir die Liste von vorne nach hinten durch, und nehmen an, dass die

Version vom 10. Februar 2022

Elemente bis zu einem gegebenen Index k in sich sortiert sind, während der Rest der Liste unserem unsortierten Haufen entspricht. Wir starten dabei mit dem Index $k = 0$ (denken Sie daran, dass Listen in C mit dem Index 0 starten). Dabei sind die Elemente am Anfang nicht zwingend die in der gegebenen Ordnung kleinsten k Elemente der gesamten Liste, sondern sie sind nur in sich, also innerhalb dieser k ersten Elemente in der Liste , sortiert. Im hinteren Teil können noch kleinere Elemente stehen. Für die Liste

```
4 3 17 6 4 2 35 8 1 5 12
```

wäre also

```
3 4 6 17 4 2 35 8 1 5 12
```

ein Zwischenstand in der Sortierung für den Index $k = 3$.

Um den sortierten Teil nun um ein Element zu verlängern, nehmen wir das Element a an der Stelle $k+1$ aus der Liste, und verschieben nun so lange Elemente des ersten Teils um eine Position nach hinten, bis wir die Stelle gefunden haben, an der das Element a stehen muss und fügen es dort ein. Im Beispiel nehmen wir also die 4 an der fünften Stelle heraus und schieben 17 und 6 um eine Position nach rechts, bevor wir 4 wieder einfügen und die Liste

```
3 4 4 6 17 2 35 8 1 5 12
```

erhalten.

Hier ist der Algorithmus in Pseudocode.

Algorithmus: InsertionSort

Eingabe : Liste $A = (a_1, \ldots, a_n)$
Ausgabe : A aufsteigend sortiert

```
1
2 für i ← 2,...,n tue
3 |   b ← a_i
4 |   k ← i
5 |   solange k ≥ 2 und a_{k-1} > b tue
6 |   |   a_k ← a_{k-1}
7 |   |   k ← k − 1
8 |   a_k ← b
```

Für die erste Betrachtung ist es sicherlich sinnvoll, bei der Liste A anzunehmen, dass es eine Liste ganzer Zahlen ist, die wir mit der Ordnung $<$ in der Bedingung der inneren (**while**-)Schleife aufsteigend sortieren. Sie können sich aber schon hier überlegen, dass dieser Algorithmus nicht auf diesen Fall eingeschränkt ist. Die einzige Stelle, wo wir auf den ersten Blick die Ordnungsstruktur auf den ganzen Zahlen verwendet haben, ist in der Schleifenbedindung. Wenn wir Daten haben, die wir nach einem anderen Sortierkriterium sortieren wollen, müssen wir nur an dieser Stelle statt $<$ eine Funktion mit zwei Parametern einsetzen, die die Bedingung erfüllt sein lässt, wenn wir den ersten Parameter nach unserem Kriterium vor den zweiten sortieren wollen. Dafür können wir auch gut einen allgemeinen Funktionszeiger einsetzen, und so eine vom Sortierkriterium unabhängige Sortierfunktion schreiben.

Eine mögliche Umsetzung in C-Code, vorerst nur für ganze Zahlen mit aufsteigender Sortierung, könnte so aussehen.

kapitel_11/insertion_sort.c

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

void insertion_sort(int A[], int n) {

  for ( int k = 1; k < n; ++k ) {
    int b = A[k];
    int j = k-1;
    while ( j>= 0 && A[j] > b ) {
      A[j+1] = A[j];
      --j;
    }
    A[j+1] = b;
  }
}

int main (int argc, char** argv) {

    int size = argc-1;
    int A[size];

    for ( int i = 1; i < argc; i=i+1 ) {
        A[i-1] = atoi(argv[i]);
    }

    insertion_sort(A,size);

    for ( int i=0; i < size; i=i+1 ) {
        printf("%d ", A[i]);
    }
    printf("\n");

    return 0;
}
```

Ein sinnvolles Maß für die Größe der Eingabe ist in diesem Fall die Länge n der Liste (jedenfalls so lange, wie die Daten wieder in einen der einfachen Typen, z.B. `long` passt und daher eine Vergleichsoperation nicht wesentlich von der Größe der Daten abhängt).

Wir können leicht den ungünstigsten Fall für unseren Algorithmus angeben. Wenn die Liste in umgekehrter Reihenfolge an die Funktion übergeben wird, dann bricht die innere Schleife erst bei der Bedingung $j \geq 0$ ab und wir führen die Verschiebung eines Elements um eine Position insgesamt $n(n-1)/2$ mal aus. Damit haben wir einen Algorithmus gefunden, der eine Liste in der Zeit $\mathcal{O}(n^2)$ sortieren kann und auch im schlechtesten Fall tatsächlich eine Zeit proportional zu n^2 benötigt.

Es gibt weitere Sortierverfahren, die auf einer ähnlichen Idee beruhen und auch eine ähnliche Laufzeit haben, wie zum Beispiel *SelectionSort* oder *BubbleSort*. Der zweite dürfte eines der bekanntesten einfachen Sortierverfahren sein. Der Algorithmus geht für eine Liste der Länge n die gesamte Liste $n-1$ mal von $k=0$ bis $k=n-1$ durch und vertauscht das Element an der Stelle $k+1$ mit dem an der Stelle k, wenn das Paar

in der falschen Reihenfolge steht. Hier ist seine Formulierung in Pseudocode.

Algorithmus: BubbleSort

Eingabe : Liste $A = (a_1, \ldots, a_n)$
Ausgabe : A aufsteigend sortiert

```
1
2 für i ← 0, ..., n − 1 tue
3 |   für k ← 0, ..., n − 1 tue
4 |   |   wenn a_{k+1} < a_k dann
5 |   |   |   b ← a_k
6 |   |   |   a_k ← a_{k+1}
7 |   |   |   a_{k+1} ← b
```

Wir können uns überlegen, dass nach der ersten Iteration in der Liste das größte Element an der letzten Stelle, und damit an der korrekten Stelle für die sortierte Liste, steht, und iterativ weiter, dass nach der i-ten Iteration die letzten i Elemente an der richtigen Stelle stehen. Damit erhalten wir nach $n-1$ Iterationen eine Liste, bei der die letzten $n-1$ Elemente an der richtigen Stelle stehen. Damit muss auch das letzte verbliebene Element an der richtigen Stelle stehen, da es keine weiteren Möglichkeiten mehr dafür gibt. Damit erhalten wir nach $n-1$ Durchläufen der äußeren und der inneren Schleife eine sortierte Liste und haben erneut einen Sortieralgorithmus, der eine Laufzeit von $\mathcal{O}(n^2)$ hat. Eine Liste in umgekehrter Reihenfolge braucht auch zwingend alle diese Schritte, so dass wir im ungünstigsten Fall auch nicht schneller sein können.

Wir sehen im nächsten Abschnitt, dass wir diese Zeit verbessern können und geben einen Algorithmus an, der eine Laufzeit in $\mathcal{O}(n \log, n)$ hat. Man kann zeigen, dass unter der Annahme, dass wir während des Sortierens immer nur Paare von Elementen vergleichen und entscheiden können, welches Element das kleinere in unserer Ordnung ist (wie wir das im Algorithmus in der Bedingung der inneren Schleife machen), und wir keine weitere Information über die Elemente der Liste haben, keinen schnelleren Algorithmus geben kann.

Wenn wir zusätzliche Information über die Einträge der Liste haben, dann können wir aber durchaus schneller sortieren. Ein einfaches Beispiel ist *BucketSort*. Hier nehmen wir an, dass wir von vornherein ein Intervall $[l, u]$ kennen, in dem alle Elemente der Liste liegen. Wir können dann sortieren, in dem wir eine zweite Liste S der Länge $u - l + 1$ definieren, anfangs alle Einträge auf 0 setzen, dann einmal jedes Element a der eingegebenen Liste A hernehmen und den Eintrag an der Stelle a der Liste S um 1 erhöhen. Am Ende lesen wir einmal s aus und schreiben aufsteigend jedes Element a so oft in A, wie $S[a]$ angibt. Eine mögliche Implementierung, bei der wir annehmen, dass alle Einträge der Eingabe zwischen 0 und 1000 liegen, könnte wie folgt aussehen.

kapitel_11/bucket_sort.c

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

#define MAXRANGE 1000

int main(int argc, char** argv) {

  int S[MAXRANGE] = {0} ;
```

```
  for ( int i = 1; i < argc; ++i ) {
    int n = atoi(argv[i]);
    if ( n > MAXRANGE ) {
      printf("Eingabe zu gross\n");
      return 1;
    }  else {
      S[n]++;
    }
  }
  for ( int i = 0; i < MAXRANGE; ++i )
    for ( int j = 0; j < S[i]; ++j )
      printf("%d ", i);

  printf("\n");
  return 0;
}
```

Wenn in diesem Ansatz der Bereich $[l, u]$ wesentlich größer ist als die Länge von A, dann haben wir nicht viel gewonnen, denn wir müssen einmal über die Liste S iterieren. Wenn allerdings n größer ist als $u - l$, dann haben wir einen Algorithmus, der in $\mathcal{O}(n)$ liegt.

11.2 Merge Sort

Das Sortierverfahren *InsertionSort*, das wir im letzten Abschnitt kennengelernt haben, sortiert die Liste, indem sie ein neues Element mit allen schon vorhandenen Elementen vergleicht. Wir können uns Varianten davon, wie zum Beispiel *BubbleSort*, überlegen, aber jedes Verfahren, dass auf dieser oder einer ähnlichen Idee aufbaut, wird im wesentlichen in jeder Iteration ein Element mit allen anderen vergleichen (bei der Variante *InsertionSort* am Anfang weniger, aber sobald die Hälfte der Elemente schon sortiert ist, vergleichen Sie im schlechtesten Fall jedes Element mit mindestens der Hälfte der Elemente, und selbst wenn wir den Anfang nicht berücksichtigen, vergleichen Sie daher noch $n/2$ Elemente mit jeweils $n/2$ Elementen, und der Faktor $1/4$ spielt asymptotisch keine Rolle). Um die Laufzeit von $\mathcal{O}(n^2)$ also wirklich zu verbessern, brauchen wir eine ganz neue Idee.

Wir wollen für unseren nächsten Algorithmus die folgende, eigentlich sehr einfache, Idee verwenden. Wenn wir statt einer vollkommen unsortierten Liste L von n Elementen zwei Listen L_1 und L_2 mit jeweils $n/2$ Elementen bekommen, die schon jeweils in sich sortiert sind, dann könnten wir erheblich schneller eine Liste aus diesen beiden Teilen erzeugen, die vollständig sortiert ist. Statt nämlich beide Listen aneinanderzuhängen und dann *InsertionSort* anzuwenden, können wir beide Listen hernehmen, und das jeweils erste (und damit kleinste) Element jeder Liste vergleichen. Das kleinere der beiden (oder ein beliebiges, wenn sie gleich sind), schreiben wir in eine neue Liste S und löschen es von L_1 oder L_2. Wenn wir diesen Schritt wiederholen, bekommen wir immer das kleinste der noch in den Listen L_1 und L_2 verbliebenen Elemente. Da die Summe der Längen von L_1 und L_2 in jedem Schritt um 1 kleiner wird, sind wir hier nach n Schritten fertig, also in *linearer* Zeit $\mathcal{O}(n)$. wenn eine der Listen schon vor der anderen leer ist, dann können Sie die restlichen Element natürlich ohne Vergleich hinten an S anfügen. Trotzdem haben Sie bis dahin mindestens $n//2$ Vergleiche durchgeführt, was

Algorithmus 11.3: merge_sort

Eingabe : array $A = (a_1, \ldots, a_n)$
Ausgabe : A aufsteigend sortiert

```
1  wenn n > 1 dann
2      m ← ⌈n/2⌉                                          // Liste teilen
3      L ← A_{1,...,m}                                    // untere Hälfte
4      U ← A_{m+1,...,n}                                  // obere Hälfte
5      L ← Merge_Sort(L)                                  // Rekursion
6      U ← Merge_Sort(U)
7      i ← 1, j ← 1
8      für k ← 1, ..., n tue                              // Teillisten vereinigen
9          wenn j > n − m oder (i ≤ m und L_i ≤ U_j) dann // aus L wenn erstes
10             A_k ← L_i                                  // Element kleiner
11             i ← i + 1                                  // oder U leer
12         sonst
13             A_k ← U_j
14             j ← j + 1
```

wieder in $\mathcal{O}(n)$ liegt.

Diese Methode ist so natürlich noch nicht hilfreich, denn wir brauchen dafür zwei sortierte Teillisten von (ungefähr) gleicher Länge. Wenn wir jetzt also zu lange brauchen, diese zu erzeugen, dann hilft uns die lineare Laufzeit für das Zusammenfügen auch nicht. Wir sollten also also möglichst effizient zwei sortierte Teillisten erzeugen. Dafür hilft die folgende Beobachtung. Nachdem wir unsere Ausgangsliste L willkürlich in zwei Teile L_1 und L_2 zerlegt haben, haben wir zwei vollkommen unabhängige Sortierprobleme auf den beiden Listen. Wenn wir also hoffen, mit unserer Idee sortierter Teillisten erfolgreich zu sein, können wir den gleichen Ansatz auf die Listen L_1 und L_2 anwenden.

Wenn wir es schaffen, L_1 in zwei Teile von (ungefähr) gleicher Größe zu zerlegen und diese zu sortieren, können wir mit $n/2$ weiteren Vergleichen L_1 sortieren. Ebenso geht das mit L_2. Wenn wir also insgesamt vier sortierte Teillisten mit je (ungefähr) $n/4$ Elementen (zwei für L_1 und zwei für L_2) erzeugen können, dann können wir L_1 *und* L_2 mit höchstens $n/2 + n/2 = n$ weiteren Vergleichen sortieren.

Das können wir natürlich weiterdrehen. Die vier sortierten Teillisten können wir erzeugen, in dem wir jede der Liste willkürlich in zwei Teile teilen, diese sortieren, und dann mit weiteren insgesamt $4 \cdot n/4 = n$ vergleichen zusammenbauen. Nach k Schritten haben wir auf diese Weise 2^k Listen mit je ungefähr $n/2^k$ Elementen, die wir mit höchstens $2^k \cdot n/2^k = n$ Vergleichen zusammenbauen können.

Diese Iteration endet offensichtlich, wenn unsere Teilliste, die wir sortieren wollen, nur noch ein einziges Element enthält, denn eine solche Liste ist *immer* sortiert. Wir bekommen ein neues Sortierverfahren, den sogenannten *merge sort*, da seine zentrale Funktion das Zusammenfügen (*merge*) sortierter Teillisten ist.

Die Umsetzung ist etwas aufwendiger als bei den Verfahren im letzten Abschnitt. Bei unserem neuen Ansatz müssen wir uns die einzelnen Teillisten und ihre Stellung in der Gesamtliste merken. Zudem ist der Ansatz in der hier gegebenen Formulierung rekursiv,

so dass wir die Sortierung in eine Funktion auslagern müssen, die wir immer wieder mit neuen Parametern aufrufen können und dann ihre Rückgabe korrekt in die Liste einfügen müssen. In Algorithmus 11.3 haben wir die Idee noch einmal mit Pseudocode in formalisierter Notation aufgeschrieben. Da es wesentlich einfacher aufzuschreiben ist, haben wir in dieser Formulierung die jeweils zu sortierenden Teillisten aus der Gesamtliste A in neue Listen L und U kopiert. Auch wenn das die asymptotische Laufzeit nicht verändert, ist das ein spürbarer Mehraufwand an Operationen, so dass wir in der eigentlich Implementierung in `C` nachher darauf verzichten und stattdessen die Grenzen des zu sortierenden Bereichs übergeben.[1]

Wir sollten uns noch überlegen, ob wir mit diesem Ansatz wirklich schneller geworden sind. Da sich in jeder Iteration die Zahl der Elemente einer Liste ungefähr halbiert (wenn die Ausgangslänge ungerade ist, dann unterscheiden sich die Längen der Teile um 1), erreichen wir Listen der Länge 1, wenn die Zahl der Iterationen k die Ungleichung $n/2^k \leq 1$ oder $n \leq 2^k$ oder $\log_2 n \leq k$ erfüllt. Wir brauchen also $k := \lceil \log_2 n \rceil$ Iterationen. In jeder dieser Iterationen machen wir n Vergleiche (und ggf. eine lineare Anzahl weiterer Operationen um zum Beispiel die Elemente aus den Teillisten in eine Gesamtliste zu kopieren). Wir kommen also Insgesamt mit $n \log_2 n$ Vergleichen aus! Damit haben wir einen Algorithmus mit einer Laufzeit von $\mathcal{O}(n \log n)$[2] gefunden.

Die Bibliothek in Programm 11.1 zeigt eine mögliche Implementierung dieser Idee, die wir zum Beispiel mit Programm 11.2 auf eine eingelesene Liste von Zahlen anwenden können.

Programm 11.1: `kapitel_11/mergesort.h`

```
#ifndef MERGESORT_H
#define MERGESORT_H

void merge_sort(int* , int );

#endif
```

Nach außen sichtbar definieren wir nur eine einzige Funktion `merge_sort`, die eine Liste und ihre Länge übernimmt. Da die Liste über einen Array ankommt, wird jede Änderung des Arrays auch an die aufrufende Stelle zurückgegeben.

Programm 11.1: `kapitel_11/mergesort.c`

```
#include <stdlib.h>

#include "mergesort.h"

void merge(int* a, int l, int m, int u) {

   int aux[u-l];
   int l_counter=l, u_counter=m, aux_counter=0;

   while ( l_counter < m && u_counter < u) {
      if ( a[l_counter] < a[u_counter] )
```

[1] Auch die Rekursion sollte man in einer effizienten Implementierung auflösen und den Algorithmus iterativ schreiben.

[2] Die Basis des Logarithmus spielt hier keine Rolle, da $\log_a n = \log_a b \log_b n$ und $\log_a b$ ein konstanter Faktor ist. Man lässt diese Angabe daher in der Regel bei der Angabe von asymptotischen Laufzeiten in unserer Definition mit der Funktionsklasse $\mathcal{O}(f)$ weg.

```
12            aux[aux_counter++] = a[l_counter++];
13         else
14            aux[aux_counter++] = a[u_counter++];
15      }
16
17      while ( l_counter < m )
18         aux[aux_counter++] = a[l_counter++];
19
20      while ( u_counter < u )
21         aux[aux_counter++] = a[u_counter++];
22
23      for ( int i = l, aux_counter = 0; i < u; ++i, ++aux_counter )
24         a[i] = aux[aux_counter];
25   }
26
27   void sort(int* a, int l, int u) {
28      if ( u-l > 1 ) {
29         int m = l+(u-l)/2;
30         sort(a,l,m);
31         sort(a,m,u);
32         merge(a,l,m,u);
33      }
34   }
35
36   void merge_sort(int* a, int size) {
37      sort(a,0,size);
38   }
```

In der Implementierung unserer Bibliothek definieren wir dann die zwei wesentlichen Funktionen, die den beiden Teilen der Sortierung entsprechen. Die Funktion `merge` fügt zwei schon sortierte Teillisten zu einer Gesamtliste zusammen. Die Funktion `sort` testet, ob in der übergebenen Liste noch etwas zu tun ist (also ob die Liste noch mehr als ein Element enthält). Wenn das nicht der Fall ist, gibt sie die erhaltene Liste unverändert zurück. Andernfalls teilt sie die Liste in zwei (ungefähr) gleich große Teile auf, ruft sich dann rekursiv auf beiden Teilen selbst auf und ruft anschließend auf der Rückgabe beider Teile dann `merge` auf. Wir wollen, wie schon oben ausgeführt, so weit wie möglich das Kopieren von Listen vermeiden. Daher Teilen wir die Liste nicht auf, in dem wir die Elemente in zwei neue Listen schreiben, sondern wir geben nur einen unteren und oberen Index in unserer Liste an, die den Bereich markieren, den unsere Teilliste in der Gesamtliste einnimmt.

Diese beiden Funktionen müssen für den Benutzer nicht sichtbar sein. Die einzige über den Header sichtbare Funktion `merge_sort` übernimmt die Aufgabe, die von außen kommenden Daten so aufzubereiten, dass sie für einen ersten Aufruf von `sort` ausreichen. Das ist hier nur die Aufgabe, aus der Gesamtlänge der Liste einen innerhalb von `sort` zu sortierenden Bereich der Liste auszuwählen, was am Anfang alles zwischen den Elementen mit Index 0 und der Gesamtlänge ist.

Programm 11.2: `kapitel_11/mergesort_main.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "mergesort.h"
4
5  void print_array(int A[], int size) {
6    for ( int i=0; i < size; i=i+1 ) {
```

```
7       printf("%d ", A[i]);
8     }
9     printf("\n");
10  }
11
12  int main(int argc, char** argv) {
13
14    int size = argc-1;
15    int A[size];
16
17    for ( int i = 1; i < argc; i=i+1 ) {
18      A[i-1] = atoi(argv[i]);
19    }
20
21    print_array(A,size);
22    merge_sort(A,size);
23    print_array(A,size);
24
25    return 0;
26  }
```

In unserem Beispiel Programm 11.2 eines Programms, das die Sortiermethode aufruft, haben wir neben dem Einlesen der Liste von der Kommandozeile nur eine Funktion implementiert, die eine an sie übergebene Liste auf dem Bildschirm ausgeben kann. Hier können Sie je nach Anwendung dann Ihre eigenen weiteren Funktionen ansetzen.

Version vom 10. Februar 2022

Version vom 10. Februar 2022

12 Dynamische Listen

Die Form einer Liste in C, die wir bisher kennen, also in der Form

```
1 int liste[laenge];
```

ist ziemlich eingeschränkt. Zum einen können wir nur Listen von einfachen Datentypen bilden und müssen daher, wenn wir mehrere Informationen zusammenfassen wollen, mehrere parallele Listen pflegen, zum anderen müssen wir die Länge der Liste bei Ihrer Deklaration festlegen, wir können also nicht variabel auf Eingaben reagieren, die wir erst während des Programmlaufs erhalten (aus Berechnungen, die wir speichern wollen, oder aus Eingaben von außen, durch den Benutzer, das System, oder andere Prozesse). Zudem ist es schwierig, in diesen Listen Elemente einzufügen oder zu löschen, wenn das nicht am Ende der Liste passiert.

In diesem Kapitel wollen wir uns ansehen, wie wir zum einen aus den grundlegenden Datentypen neue, eigene, Datentypen zusammenbauen können, und wie wir Listen definieren können, deren Länge wir auch zur Laufzeit des Programms variieren können, und bei der wir an beliebiger Stelle Elemente einfügen und löschen können. Wir fangen mit der Deklaration eigener Datentypen an.

12.1 `struct` und `typedef`

Bisher sind wir mit den einfachen Datentypen `char`, `int` und `float` sowie Listen solcher Typen ausgekommen und haben gesehen, dass man damit schon viele nützliche Programme schreiben kann.

Prinzipiell könnte man damit auch weiterhin auskommen. Für komplexere Programme wäre es jedoch hilfreich, und auch der Lesbarkeit zuträglich, zusammengehörende Daten auch in einer gemeinsamen Struktur zusammenfassen, in einem Durchgang initialisieren oder aktualisieren und als ganzes an eine Funktion übergeben zu können. Wenn eigentlich zusammengehörende Daten in verschiedenen Listen abgelegt sind, passiert es schneller, eine Veränderung nicht in allen davon korrekt vorzunehmen, oder nicht alle an Funktionen zu übergeben.

12.1.1 `struct`

Um mehrere Daten in einer gemeinsamen Struktur zusammenzufassen, gibt es in C die Möglichkeit, einen neuen Datentyp über das Schlüsselwort `struct` zu deklarieren. Auf diese Weise können wir mehrere Variablen auch unterschiedlicher Typen zusammenfassen. Das können auch schon von uns deklarierte neue Typen sein, wir können also schrittweise immer umfangreichere Datentypen erzeugen.

Version vom 10. Februar 2022

Hier ist eine erste einfache Variante davon.

Programm 12.1: `kapitel_12/struct_def.c`

```
1  struct {
2    int i;
3    float f;
4  }  s;
5
6  s.i=7;
7  s.f=3.14;
```

Ein `struct` wird immer mit dem Schlüsselwort `struct` eingeleitet, und die darin enthaltenden Variablen werden in `{...}` eingeschlossen. In unserem Beispiel ist das eine Variable `i` vom Typ `int` und `f` vom Typ `float`. Diese Variablen sind die *Elemente* des `struct`. Damit haben wir einen neuen Datentyp geschaffen. Wir können dem Typ einen eigenen Namen geben und dann Variablen von diesem Typ erzeugen. Das sehen wir gleich. Im Beispiel haben wir direkt eine einzige Variable `s` dieses Typs definiert.

Um auf die Elemente unseres `struct` zuzugreifen, brauchen wir den Elementzugriffsoperator `.`, den wir ohne Erklärung schon in Table 5.1 gesehen haben. An den Namen der `struct`-Variablen wird der Name des Elements angehängt, auf das wir zugreifen wollen. Im Beispiel belegen wir einfach beide Variablen mit einem Wert.

Mit der Anweisung

```
1  struct {
2    int i;
3    float f;
4  } ;
```

haben wir einen anonymen neuen Datentyp geschaffen. Um diesen Typ später wie z.B. `int` immer wieder verwenden zu können, können wir einen *Bezeichner* (*tag*) vergeben, der diese Struktur eindeutig benennt. Der Bezeichner kommt direkt nach dem Schlüsselwort `struct`.

```
1  struct mein_typ {
2    int i;
3    float f;
4  } ;
```

Anschließend können wir mit

```
1  struct mein_typ s;
```

Variablen unseres neuen Typs definieren. Beachten Sie, dass das Schlüsselwort `struct` auch in der Variablendefinition auftaucht. Der neue Typ steht, wie eigentlich alle Definitionen in `C`, nur in dem Bereich zur Verfügung, in dem er definiert (bzw., wie wir sehen werden, zumindest deklariert) wurde. Da wir in der Regel einen neuen Datentyp im gesamten Programm nutzen wollen, steht die Definition meistens außerhalb jeder Funktion in der Datei, wie im nachfolgenden Beispiel.

Programm 12.2: `kapitel_12/simple_struct.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct st {
```

```
  int i;
  float f;
} ;

void show ( struct st s) {
  printf("Elemente i: %d und f: %f\n",s.i, s.f);
}

int main() {

  struct st s,t;

  s.i=7;
  s.f=3.14;

  t.i=8;
  t.f=9.81;

  show(t);

  return 0;
}
```

Oft wird die Definition auch direkt in einen Header ausgelagert, zusammen mit speziell auf diesen Typ zugeschnittenen Funktionen. Dies könnten unter anderem Funktionen sein, die dynamisch Speicher für alle im `struct` enthaltenen Elemente anfordern oder wieder freigeben, oder alle enthaltenen Elemente mit Standardwerten initialisieren. Dieser Header wird dann in allen Dateien eingebunden, in denen der neue Typ benutzt werden soll.

Wie alle Variablen wird auch ein `struct` bei der Übergabe an eine Funktion kopiert. Wenn wir die Elemente eines `struct` also in der Funktion verändern wollen, müssen wir, wie bei anderen Variablen auch, einen Zeiger auf den `struct` übergeben. Hier ist ein einfaches Beispiel.

```
  void aendern ( struct st *s ) {
    (*s).i = 2;
    s->f = 2.71;
}
```

Wir haben zwei Möglichkeiten, mit einem Zeiger auf einen `struct` auf seine Elemente zuzugreifen. Zum einen, und das ist wohl die naheliegende Variante, können wir unseren `struct` dereferenzieren und dann den Elementzugriffsoperator benutzen. Dabei müssen wir aber beachten, dass der Zugriffsoperator bei der Auswertung eine höhere Priorität besitzt als die Dereferenzierung (wie in Table 5.1 notiert). Um tatsächlich unseren Zeiger zu dereferenzieren bevor wir auf eine Element zugreifen, müssen wir also wie im Beispiel Klammern setzen.

Andernfalls geht der Compiler davon aus, dass wir eine Variable vom Typ des `struct`s haben, die ein Element enthält, dass ein Zeiger ist, den wir dereferenzieren wollen. Das könnte dann wie folgt aussehen.

```
  struct {
    int * ip;
  }  s;

  int j = 4;
```

```
6    s.ip = &j;
7    *s.ip = 5;  // j ist jetzt 5
```

Da diese Notation zum Dereferenzieren eines Zeigers auf einen `struct` mühsam und schwer zu lesen ist, gibt es einen speziellen Operator `->` für Elementzugriff über einen Zeiger auf einen `struct`. Das ist die übliche Variante um auf Elemente zuzugreifen.

Wir können natürlich auch dynamisch Speicher für unseren `struct` anfordern. Die Größe des benötigten Speichers können wir wieder mit `sizeof` bestimmen.

```
1    struct mein_typ * zeiger = (struct mein_typ *)malloc(sizeof(mein_typ));
2
3    zeiger->i = 3;
4    zeiger->f = 3.14;
```

Aufpassen müssen wir allerdings, wenn unsere Struktur selbst Zeiger enthält.

```
1    struct data {
2      int * int_zeiger;
3      float f;
4    } ;
5
6    struct data * data_zeiger = malloc(sizeof(struct data));
7
8    data_zeiger->int_zeiger = malloc(sizeof(int));
9    *data_zeiger->int_zeiger = 4;
```

Bei der Speicheranforderung für `data_zeiger` wird nur Speicher für ein `int *` (und ein `float`), also für das Speichern einer *Adresse* reserviert, nicht für den Wert, auf den `int_zeiger` zeigen könnte.

Das wäre auch nicht sinnvoll, denn es könnte zum einen schon eine Variable im Speicher geben, deren Adresse wir in `int_zeiger` speichern wollen, zum anderen können wir mit einem Zeiger sowohl auf eine einzelne Variable des Typs als auch auf eine ganze Liste zeigen wollen. Diesen Unterschied kann der Compiler nicht erkennen. Trotzdem wird diese Initialisierung oft vergessen.

Am Ende müssen dann auch alle reservierten Speicherbereiche wieder freigegeben werden.

```
1    free(data_zeiger->int_zeiger);
2    free(data_zeiger);
```

Die Reihenfolge ist hier wichtig. Wenn wir den Speicher von `data_zeiger` freigeben, wird der Speicher der Adresse von `int_zeiger` gelöscht, und wir haben keine Möglichkeit mehr, den Speicherbereich freizugeben, auf den `int_zeiger` zeigte.

12.1.2 `typedef`

Mit dem Schlüsselwort `typedef` können wir beliebigen Typen einen neuen Namen geben. Das ist im Zusammenhang mit den `struct`s aus dem vorangegangenen Abschnitt nützlich, aber auch für andere Typen. Hier sind einige Beispiele.

```
1    typedef int ganze_zahl;
2    typedef char* zeichenkette;
3    typedef int liste[5];
4
```

```
ganze_zahl a = 5;
zeichenkette z = "Mein Typ";
liste l = {1,2,3,4,5} ;
```

Mit einem `typedef` können wir auch den Wechsel des Datentyps leichter machen, solange sich beide Typen nicht in der Benutzung unterscheiden (z.B. die Wahl von `int` oder `long long`). Wenn wir am Anfang einen eigenen Namen vergeben, und für weitere Deklarationen diesen Namen verwenden, muss nur eine einzige Stelle angepasst werden. Im folgenden Programm reicht eine Änderung in der ersten Zeile, um die Summenfunktion z.B. die Summe zweier Variablen vom Typ `long long` zu berechnen.

```
typedef int ganze_zahl;

ganze_zahl summe(ganze_zahl a, ganze_zahl b) {
  return a+b;
}

int main() {
  ganze_zahl x = 4;
  ganze_zahl y = 5;

  ganze_zahl s = summe(x,y);

  return 0;
}
```

Mit `typedef` können wir natürlich auch für mit `struct` definierte Typen einen eigenen Namen vergeben. Nach der Definition

```
typedef struct {
  int i;
  float f;
}  MeinTyp;
```

können wir mit

```
MeinTyp s;
s.i = 1;
s.f = 3.14;
```

Variablen dieses Typs definieren. Sie können hier auch einen Bezeichner vergeben. Das ist zwar nur in wenigen Fällen notwendig, es ist aber im Allgemeinen üblich, beides zu vergeben.

```
typedef struct mein_typ {
  int i;
  float f;
}  MeinTyp;
```

Bezeichner und Typname dürfen auch identisch sein.

```
typedef struct Point {
  float x, y;
}  Point;

Point mid_point ( Point a, Point b) {
  Point m;
  m.x = (a.x+b.x)/2;
  m.y = (a.y+b.y)/2;
```

```
  return m;
}

struct Point mid_point_2 ( struct Point a, struct Point b) {
  struct Point m;
  m.x = (a.x+b.x)/2;
  m.y = (a.y+b.y)/2;
  return m;
}
```

An fast allen Stellen sind dann bei der Angabe des Typs die Varianten `struct mein_typ` und `MeinTyp` austauschbar. Die einzige für uns wesentliche Stelle, an der wir nur den Bezeichner, aber nicht den Typ benutzen können, sind (Vorwärts-)deklarationen der Struktur. Das passiert in der Regel, wenn wir innerhalb der Struktur einen Zeiger auf eine weitere Instanz der gleichen Struktur setzen wollen.

```
struct listen_element {
  int eintrag;
  struct listen_element * nachfolger;
}  ListenElement;
```

In dieser Definition erzeugen wir eine Struktur, die ein einzelnes Element einer Liste (hier nur ein `int`) speichern kann, und uns einen Zeiger auf die Speicheradresse des nächsten Elements in der Liste gibt. Da in der Definition an der Stelle, an der wir mit `struct listenelement *` diesen Zeiger definieren, der Typname noch nicht bekannt ist, können wir ihn hier noch nicht benutzten, sondern müssen auf den Bezeichner zurückgreifen.

Beachten Sie auch, dass wir hier nur einen *Zeiger* auf unsere Struktur aufnehmen können. Die Definition eine Variable des Typs der Struktur innerhalb der Struktur führte zu einer unendlichen Rekursion in der Definition.

Bei dem Bezeichner der Struktur reicht es `C` auch schon aus, wenn bei der Verwendung der Bezeichner nur deklariert ist, solange wir nicht auf die Elemente der Struktur zugreifen wollen. Wir könnten daher auch einen `typedef` auf den `struct` machen, bevor wir den `struct` definieren.

```
typedef struct listen_element ListenElement;

struct listen_element {
  int element;
  ListenElement * nachfolger;
} ;
```

Wenn wir mehrere Strukturen verschachteln wollen, können wir neben der Möglichkeit eines `typedef` auch nur die Struktur Vorwärtsdeklarieren, z.B. in dieser Form.

```
struct T;

struct S {
  int * i;
  struct T * t;
} ;

struct T {
  int j;
  struct S s;
} ;
```

Auch hier können wir nur einen Zeiger auf `struct T` aufnehmen.

Wenn wir ein Element aufnehmen wollen, müssen wir zuerst `struct T` vollständig definieren, also die Definition vor der von `struct S` schreiben. Dann darf aber `struct T` kein Element vom Typ `struct S` enthalten, da wir sonst wieder zu einer unendlichen Rekursion in der Definition kommen. Alle Varianten, die keine solche unendliche Rekursion enthalten, lassen sich also definieren.

Im nächsten Abschnitt über verkettete Listen werden wir genau solche Datenstrukturen genauer untersuchen.

12.2 Verkettete Listen

Für eine flexible Listenstruktur brauchen wir eine neue Idee, wie wir auf Listen schauen wollen. Die zentrale Idee hinter den *verketteten Listen* (*linked lists*) ist es, die einzelnen Datenpunkte der Liste als einzelne Datenstruktur anzusehen, und die Liste nur implizit dadurch aufzubauen, dass jeder Eintrag in der Liste weiss, welcher Eintrag nach ihm (oder vor ihm, oder beides) in der Liste steht. Dafür statten wir jeden Eintrag in der Liste neben seinen eigentlichen Daten mit einem Zeiger aus, der auf das nächste (oder das vorangegangene, oder mit zwei Zeigern, je einem, der auf das vorangegangene und ein einem, der auf das nächste) Element zeigt.

Konkret wollen wir also eine Datenstruktur in der folgenden Form definieren (dabei schränken wir uns vorerst darauf ein, dass jedes Element nur das nächste Element der Liste kennt).

```
1  struct data {
2    int n;
3    char name;
4  } ;
5
6  struct elem {
7    struct data * data;
8    struct elem * next;
9  } ;
```

Zur Vereinfachung nehmen wir im folgenden an, dass die Information, die wir in der Liste speichern wollen, nur vom Typ `int` ist. Dadurch können wir die Information direkt in `struct elem` unterbringen und kommen auf diese Weise vorerst mit einer einzigen Struktur der Form

```
1  struct elem {
2    int data;
3    struct elem * next;
4  } ;
```

aus, für die wir mit

```
1  struct elem * e = malloc(sizeof(struct elem));
```

einen neuen Speicherbereich mit einem zugehörigen Zeiger definieren können. Wenn auch die Daten in einer Struktur liegen und wir nur einen Zeiger auf die Daten in der Liste halten, dann müssen wir in einem zweiten Schritt natürlich auch dafür einen Speicherbereich reservieren.

Damit könnte ein erster Versuch für eine Liste so aussehen.

Programm 12.3: `kapitel_12/linked_list_01.c`

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

  struct elem {
    int data;
    struct elem * next;
  } ;

int main() {

  struct elem *e1 = (struct elem *)malloc(sizeof(struct elem));
  struct elem *e2 = (struct elem *)malloc(sizeof(struct elem));
  struct elem *e3 = (struct elem *)malloc(sizeof(struct elem));

  e1->data = 1;
  e2->data = 2;
  e3->data = 3;

  e1->next = e2;
  e2->next = e3;

  struct elem * e = e1;
  for ( int i = 0; i < 3; ++i ) {
    printf("Das %d. Element der Liste ist %d\n", i+1, e->data);
    e = e->next;
  }

  free(e1);
  free(e2);
  free(e3);

  return 0;
}
```

Hier haben wir eine Liste mit drei Elementen 1, 2 und 3 definiert. Wir sehen auch, wie wir auf die Elemente der Liste zugreifen können. Dafür haben wir uns einen Zeiger `e` auf das erste Element definiert, und ersetzen diesen Zeiger, nachdem wir die Information an der Stelle der Liste, auf die `e` zeigt, ausgelesen haben, durch den in `e->next` gespeicherten Zeiger. Wir dürfen natürlich nicht vergessen am Ende den reservierten Speicher auch ans System zurückzugeben.

Im Beispiel haben wir die ganze Liste ausgegeben. Wenn wir nur am k-ten Element interessiert sind, können wir stattdessen k-mal `e` durch `e->next` ersetzen und lesen erst dann die in `e->data` gespeicherte Information aus.

Wir haben auf diese Weise erfolgreiche eine Liste zusammengebaut, allerdings hat diese Form noch einige offensichtliche Schwächen.

- Der in `e3` enthaltene Zeiger `e3->next` ist nicht initialisiert. Wir dürfen ihn also nicht verwenden, und müssen daher wissen, dass wir unsere Schleife nach drei Durchläufen beenden müssen.
- Um die Liste aufzubauen, haben wir für jedes Element der Liste einen eigenen Zeiger `e1`, `e2`, `e3` definiert. Damit haben wir zum einen einen direkten Zugriff auf

jedes Element der Liste, und wir haben die Zeiger alle doppelt gespeichert, da sie (bis auf den ersten) auch als `e->next` auftauchen.

Das erste Problem können wir elegant dadurch lösen, dass wir den letzten Zeiger explizit auf die spezielle Adresse **NULL** zeigen lassen. Dadurch brauchen wir uns die Länge der Liste nicht zu merken. Wir sind genau dann beim letzten Element, wenn `next` auf **NULL** zeigt. Damit könnte unsere Schleife so aussehen.

Programm 12.4: `kapitel_12/linked_list_02.c`

```
  e3->next = NULL;

  struct elem * e = e1;
  int i = 1;
  while ( e != NULL )  {
    printf("Das %d. Element der Liste ist %d\n", i++, e->data);
    e = e->next;
  }
```

Um nicht für jedes Element einen zusätzlichen Zeiger definieren zu müssen, können wir unsere Liste nacheinander aufbauen und statt einer neuen Variablen den reservierten Speicher direkt dem `next`-Zeiger zuweisen. Hier ist ein Beispiel.

Programm 12.5: `kapitel_12/linked_list_03.c`

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

  struct elem {
    int data;
    struct elem * next;
  } ;

int main() {

  struct elem *e = (struct elem *)malloc(sizeof(struct elem));
  e->data = 1;
  struct elem * head = e;

  e->next = (struct elem *)malloc(sizeof(struct elem));
  e = e->next;
  e->data = 2;

  e->next = (struct elem *)malloc(sizeof(struct elem));
  e = e->next;
  e->data = 3;

  e->next = NULL;

  e = head;
  int i = 1;
  while ( e != NULL )  {
    printf("Das %d. Element der Liste ist %d\n", i++, e->data);
    e = e->next;
  }

  e = head;
  while ( e != NULL )  {
    struct elem * t = e->next;
```

```
35      free(e);
36      e = t;
37    }
38
39    return 0;
40  }
```

Das Beispiel enthält vermutlich einige Stellen, auf die wir eingehen sollten. Am Anfang initialisieren wir in Zeile 11 und der nachfolgenden Zeile einen Zeiger `e` mit einem ersten Element der Liste. In Zeile 15 reservieren wir für den `next`-Zeiger einen neuen Speicherbereich und lassen in Zeile 16 unseren Zeiger `e` darauf zeigen, um das nächste Element abzulegen.

Mit der Zuweisung in Zeile 16 haben wir aber die Information, wo das erste Element liegt, in `e` überschrieben! Daher müssen wir diese Information vorher sichern. Dafür wählen wir einen Zeiger `head` in Zeile 13, in den wir die Adresse von `e` übernehmen. Ohne diesen Zeiger hätten wir keine Möglichkeit mehr, an den Anfang der Liste zu kommen.

Nun können wir weitere Elemente an die Liste anfügen, indem wir in `e->next` einen neuen Speicherbereich initialisieren und dann `e` dorthin zeigen lassen. Im Beispiel machen wir das noch ein Mal, bevor wir `e->next = NULL` setzen und so das Ende der Liste markieren.

Für einen Durchlauf durch die Liste machen wir, wie vorher auch, eine Kopie des Zeigers auf das erste Element und nutzen dieses um der Reihe nach die Listenelemente anzusteuern.

Da wir nun keine Zeiger mehr auf jedes einzelne Element haben, müssen wir den Speicher hier auch über eine Schleife freigeben, die einmal durch die Liste läuft. Dabei müssen wir aufpassen, dass wir erst die Adresse in `e->next` sichern, bevor wir den Speicherbereich freigeben, da wir sonst die Information verlieren, wo der nächste Speicherbereich liegt, den wir freigeben müssen. Daher brauchen wir hier eine temporäre Variable `t`, die diese Adresse kurzzeitig aufnimmt.

Wir sehen hier, dass wir bei der Initialisierung neuer Listenelemente immer wieder den gleichen Code wiederholen. Das ist ein klarer Hinweis, dass wir diesen Code in eine Funktion auslagern sollten. Das ist auch die übliche Vorgehensweise bei der Initalisierung neuer Elemente. Auch die Speicherfreigabe wird überlicherweise in eine Funktion ausgelagert. Das ist in unserem Fall nicht wirklich notwendig, da wir nur ein `int` als Eintrag haben, wird allerdings sinnvoll, wenn hier zum Beispiel ein Zeiger auf einen `struct` enthalten ist, wie wir das am Anfang betrachtet haben. Daher machen wir das direkt auch im folgenden Beispiel.

Programm 12.6: `kapitel_12/linked_list_04.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct elem {
5    int data;
6    struct elem * next;
7  } ;
8
9  struct elem * new_elem ( int data ) {
10   struct elem * e = malloc(sizeof(struct elem));
```

```
  e->data = data;
  e->next = NULL;
  return e;
}

void free_elem(struct elem * e ) {
  free(e);
}

int main() {

  struct elem * e = new_elem(1);
  struct elem * head = e;

  for ( int i = 2; i <= 12; ++i ) {
    e->next = new_elem(i);
    e = e->next;
  }

  e = head;
  int i = 1;
  while ( e != NULL )  {
    printf("Das %d. Element der Liste ist %d\n", i++, e->data);
    e = e->next;
  }

  e = head;
  while ( e != NULL )  {
    struct elem * t = e->next;
    free_elem(e);
    e = t;
  }

  return 0;
}
```

Wir sollten weitere Aufgaben in Funktionen auslagern, um mit unserer Liste arbeiten zu können. Mit der Funktion

```
int get_elem ( struct elemt * e, int index ) {
  int i = 0;
  while ( i < index ) {
    e = e->next;
  }
  return e->data;
}
```

können wir zum Beispiel das Element an der Stelle `index` der Liste erhalten. Alternativ kann es manchmal sinnvoll sein, statt nur der im Element gespeicherten Information einen Zeiger auf das Element zurückzugeben. Das ist eine kleine Änderung an unserer Funktion.

```
struct elem * get_elem ( struct elemt * e, int index ) {
  int i = 0;
  while ( i < index ) {
    e = e->next;
  }
  return e;
}
```

Bei beiden Funktionen nehmen wir an, dass die Liste auch ausreichend viele Elemente enthält, so dass an der Stelle `index` ein Element steht. Wenn Sie das beim Aufruf der Funktion nicht wissen, könnten Sie die Funktion so modifizieren, dass sie im ersten Fall einen Wert ausserhalb des vorgesehenen Datenbereichs zurückgibt (zum Beispiel -1, wenn wir nur nichtneagtive Zahlen in unserer Liste haben), oder im zweiten Fall `NULL`. In beiden Fällen müssten Sie dann die Rückgabe überprüfen, bevor Sie das Ergebnis verwenden.

Auch das Löschen der Liste sollten wir abtrennen, und zum Beispiel mit dieser Funktion machen:

```
1 void free_list(struct elem * e ) {
2   while ( e != NULL )  {
3     struct elem * t = e->next;
4     free_elem(e);
5     e = t;
6   }
7 }
```

In eine solche dynamische Liste lässt sich sehr leicht ein neues Element an einer beliebigen Stelle einfügen.

```
1 void insert_after(struct elem * pos, struct elem * new) {
2   new->next = pos->next;
3   pos->next = new;
4 }
```

Hierfür brauchen wir einen Zeiger auf das Element hinter dem das neue Element eingefügt werden soll. Genauso lässt sich ein Element aus der Liste entfernen, wenn wir einen Zeiger auf das vorangehende Element haben.

```
1 void delete_after(struct elem * pos) {
2   struct elem * t = pos->next;
3   pos->next = pos->next->next;
4   free_elem(t);
5 }
```

Hier haben wir angenommen, dass hinter `pos` tatsächlich noch ein Element steht. Wenn das unklar ist, müssen Sie vorher testen, ob `pos->next != NULL` ist. In der Funktion brauchen wir wieder einen temporären Zeiger auf das zu löschende Element, denn in der zweiten Zeile setzen wir den `next`-Zeiger von `pos` auf den Nachfolger des zu löschenden Elements. Damit ist das Element aus der Liste verschwunden, aber es wäre auch nicht mehr erreichbar, um den Speicher freizugeben, wenn wir den Zeiger `t` nicht vorher definiert hätten.

Schwieriger wird es, wenn wir statt eines Zeigers auf ein Element einen Index angeben wollen. Wir könnten versucht sein, die folgende Funktion zu verwenden (wir nehmen wieder an, dass die Liste ausreichend lang ist, dass wir an der Stelle `index` einfügen können. Sie sollten das in der Regel allerdings in der Funktion überprüfen).

```
1 // ein unvollstaendiger Versuch, ein Element an der Stelle index einzufuegen
2 void insert_at_index(struct elem * head, int index, struct elem * new ) {
3   struct elem * e = head;
4   int i = 0;
5   while ( i < index-1 ) {
6     e = e->next;
7   }
```

```
  new->next = e->next;
  e->next = new;
}
```

Das Problem tritt hier auf, wenn wir versuchen, ein Element ganz am Anfang einzufügen. Wenn `index == 0` ist, wird `new` trotzdem nach dem ersten Element eingefügt. Das könnten wir versuchen, mit

```
// ein weitere unvollstaendiger Versuch, ein Element an der Stelle index einzufuegen
void insert_at_index(struct elem * head, int index, struct elem * new ) {
  if ( index == 0 ) {
    new->next = head; // new wird ganz vorne eingefuegt
    head = new;       // new das neue erste Element, und head sollte dorthin zeigen
  }  else {
    struct elem * e = head;
    int i = 0;
    while ( i < index-1 ) {
      e = e->next;
    }
    new->next = e->next;
    e->next = new;
  }
}
```

zu lösen. Aber auch das wird nicht funktionieren, denn beim Aufruf der Funktion wird der Zeiger in `head` *kopiert*, und nach der Rückkehr an die aufrufende Stelle wird `head` wieder seinen alten Wert haben und auf das zweite Element der Liste zeigen, während das erste nur solange erreichbar ist, wie der an die Funktion übergebene Zeiger `new` gültig ist.

Für dieses Problem gibt es zwei allgemein übliche Lösungen.

- Statt des Zeigers `head` übergeben wir einen *Zeiger* auf `head`. Damit können wir verändern, wohin `head` zeigt.
- Wir geben in dieser Funktion den Zeiger `head` zurück und ersetzen an der aufrufenden Stelle `head` mit dieser Rückgabe.

Hier sind Beispiele, wie eine solche Funktion dann aussehen könnte. Wir nehmen immer noch an, dass die Liste eine ausreichende Länge hat um tatsächlich am angegebenen Index ein Element einfügen zu können. Zuerst betrachten wir die Möglichkeit, einen Zeiger auf den Zeiger auf das erste Element zu übergeben.

```
void insert_at_index(struct elem ** head_ptr, int index, struct elem * new ) {
  if ( index == 0 ) {
    new->next = (*head_ptr)->next;
    head_ptr = &new;
  }  else {
    struct elem * e = *head_ptr;
    int i = 0;
    while ( i < index-1 ) {
      e = e->next;
    }
    new->next = e->next;
    e->next = new;
  }
}

int main() {
```

```
17    struct elem * head = new_elem(data);
18    struct elem * new = new_elem(data);
19    // [...]
20    insert_at_index(&head, index, new); // hier muss die Adresse
21                                        // von head uebergeben werden
22    // [...]
23    return 0;
24  }
```

Falls wir uns entscheiden, stattdessen immer dein Zeiger auf das erste Element zurückzugeben, könnte die Funktion so aussehen.

```
1  struct elem * insert_at_index(struct elem * head, int index, struct elem * new ) {
2    if ( index == 0 ) {
3      new->next = head->next;
4      head = new;
5    }  else {
6      struct elem * e = head;
7      int i = 0;
8      while ( i < index-1 ) {
9        e = e->next;
10     }
11     new->next = e->next;
12     e->next = new;
13   }
14   return head;
15 }
16
17 int main() {
18   struct elem * head = new_elem(data);
19   struct elem * new = new_elem(data);
20   // [...]
21   head = insert_at_index(head, index, new);
22   // [...]
23   return 0;
24 }
```

Das gleiche Problem haben wir natürlich, wenn wir Elemente an einem vorgegebenen Index aus der Liste löschen wollen. Sobald wir das erste Element der Liste entfernen wollen, müssen wir den Zeiger auf das erste Element verändern. Damit das von der Funktion an die aufrufende Stelle weitergegeben werden kann, müssen wir entweder wieder eine Adresse übergeben, oder wir geben den Zeiger auf das erste Element in der Funktion zurück.

12.3 Stapel und Warteschlangen

Die Datenstrukturen *Stapel* (*stacks*) und *Warteschlangen* (*queues*) sind spezielle, eingeschränkte Varianten von Listen. Strukturen dieser Art werden in vielen Algorithmen benötigt. Da oft sehr viele Zugriffe auf die Listen erfolgen oder diese sehr groß werden können, ist es wichtig, diese Datenstrukturen effizient zu implementieren um eine gute Laufzeit des Programms zu erhalten.

Durch die Einschränkung der möglichen Veränderungen auf der und Anfragen an die Liste haben wir die Chance, die zugehörigen Methoden effizienter zu programmieren,

da wir manche Fälle, die im allgemeinen Fall auftreten können, bei diesen Strukturen nicht überprüfen müssen.

12.3.1 Stapel

Ein *Stapel* ist eine Liste, bei der wir uns darauf einschränken, dass Elemente nur am Anfang der Liste eingefügt oder entfernt werden (alternativ am Ende). Das entspricht einem Papier- oder Aktenstapel, wo wir neue Akten oben auf den Stapel legen können, und auch nur von oben Akten wieder wegnehmen können.

Die zugehörigen Methoden werden oft `push` oder `push_front` und `pop` oder `pop_front` genannt. Zudem gibt es oft Funktionen, die das erste Element aus der Liste zurückgibt, ohne es aus der Liste zu entfernen, und die anzeigt, ob noch Elemente auf dem Stapel liegen Diese Funktionen heißen oft `peek` und `is_empty`. Hier ist eine einfache Implementierung dieser Ideen in einer Bibliothek `stack`. Dabei benutzen wir Zeiger auf den Head-Zeiger, um die Funktionen `push` und `pop` zu realisieren. Die Möglichkeit, den Zeiger auf den Listenanfang in diesen Funktionen immer zurückzugeben, betrachten wir weiter unten.

Programm 12.7: kapitel_11/stack.h

```
1
2  #ifndef STACK_H
3  #define STACK_H
4
5  #ifndef NODE_STRUCT
6  #define NODE_STRUCT
7
8  // Element
9  struct node {
10   int value;
11   struct node * next;
12 } ;
13
14 #endif
15
16 /* Initialisieren und Löschen des Stapels */
17 void free_stack(struct node *);
18
19 /* Standardfunktionen */
20 void push(struct node **, int);
21 int pop(struct node **);
22 int stack_is_empty(struct node **);
23 int peek_stack(struct node **);
24
25 /* I/O */
26 void print_stack(const struct node *);
27
28 #endif
```

Programm 12.7: kapitel_11/stack.c

```
1
2  #include<stdio.h>
3  #include<stdlib.h>
```

```c
#include <limits.h>

#include "stack.h"

/*
 * Initialisieren und Löschen des Stapels
 */

static struct node* init_node(int value) {
  struct node * node = malloc(sizeof(struct node));
  node->value = value;
  node->next = NULL;
  return node;
}

static void free_node(struct node * node) {
  free(node);
}

void free_stack(struct node * node ) {
  struct node* keep = NULL;
  while( node != NULL ) {
    keep = node;
    node = node->next;
    free_node(keep);
  }
}

/*
 * Standardfunktionen
 */

void push(struct node **head_ptr, int value) {
  struct node* new_node = init_node(value);
  new_node->next = *head_ptr;
  *head_ptr = new_node;
}

int pop(struct node **head_ptr) {
  int value = INT_MIN;
  if( *head_ptr ) {
    struct node * node = *head_ptr;
    value = node->value;
    *head_ptr = node->next;
    free(node);
  }
  return value;
}

int stack_is_empty(struct node **head_ptr) {
    return ( *head_ptr == NULL );
}

int peek_stack(struct node **head_ptr) {
  int value = INT_MIN;
  if( *head_ptr ) {
    value = (*head_ptr)->value;
  }
```

```
62
63    return value;
64  }
65
66
67  /*
68   * I/O
69   */
70
71  void print_stack(const struct node * node) {
72    static int count = 1;
73    printf("Aufruf %d, die Liste enthaelt: ", count++);
74    while (node != NULL ) {
75      printf("%d ", node->value);
76      node = node->next;
77    }
78    printf("\n");
79  }
```

Den in unserer Bibliothek definierten Stapel können wir nun in unser Programm einbinden. Hier ist ein einfaches Beispiel, in dem die Bibliothek eingebunden wird und wir einige Elemente auf den Stapel legen und wieder entfernen.

Programm 12.7: `kapitel_11/stack_main.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "stack.h"
5
6  int main() {
7
8    // Initialisierung
9    struct node * head = NULL;
10
11   push(&head, 17);
12   push(&head, 22);
13   push(&head, 30);
14   print_stack(head);
15
16   int value = pop(&head);
17   printf("Wert %d zurueckbekommen\n",value);
18   print_stack(head);
19
20   printf("Liste ist leer: %s\n", stack_is_empty(&head)?"ja":"nein" );
21   printf("Das naechste Element ist %d\n", peek_stack(&head));
22   pop(&head);
23   value = pop(&head);
24   printf("Wert %d zurueckbekommen\n",value);
25   printf("Liste ist leer: %s\n", stack_is_empty(&head)?"ja":"nein" );
26
27   free_stack(head);
28
29   return 0;
30 }
```

Wie wir schon weiter oben diskutiert haben, können wir auch den Zeiger auf den Listenanfang immer wieder zurückgeben.

Programm 12.8: kapitel_11/stack_v2.h

```
1
2  #ifndef STACK_V2_H
3  #define STACK_V2_H
4
5  // Element
6  struct node {
7    int value;
8    struct node * next;
9  } ;
10
11 /* Initialisieren und Löschen des Stapels */
12 struct node* init_node(int);
13 void free_node(struct node *);
14 void free_stack(struct node *);
15
16 /* Standardfunktionen */
17 struct node * push(struct node *, int);
18 struct node * pop(struct node *, int*);
19 int is_empty(struct node *);
20 int peek(struct node *);
21
22 /* I/O */
23 void print_stack(const struct node *);
24
25 #endif
```

Programm 12.8: kapitel_11/stack_v2.c

```
1
2  #include<stdio.h>
3  #include<stdlib.h>
4  #include <limits.h>
5
6  #include "stack_v2.h"
7
8  /*
9   * Initialisieren und Löschen des Stapels
10  */
11
12 struct node* init_node(int value) {
13   struct node * node = malloc(sizeof(struct node));
14   node->value = value;
15   node->next = NULL;
16   return node;
17 }
18
19 void free_node(struct node * node) {
20   free(node);
21 }
22
23 void free_stack(struct node * node ) {
24   struct node* keep = NULL;
25   while( node != NULL ) {
26     keep = node;
27     node = node->next;
28     free_node(keep);
29   }
```

```
}

/*
 * Standardfunktionen
 */

struct node * push(struct node *head, int value) {
  struct node* new_node = init_node(value);
  new_node->next = head;
  return new_node;
}

struct node * pop(struct node *head, int * value) {
  *value = INT_MIN;
  if( head ) {
    struct node * node = head;
    *value = node->value;
    head = node->next;
    free(node);
  }
  return head;
}

int is_empty(struct node *head) {
    return ( head == NULL );
}

int peek(struct node *head) {
  int value = INT_MIN;
  if( head ) {
    value = head->value;
  }

  return value;
}


/*
 * I/O
 */

void print_stack(const struct node * node) {
  static int count = 1;
  printf("Aufruf %d, die Liste enthaelt: ", count++);
  while (node != NULL ) {
    printf("%d ", node->value);
    node = node->next;
  }
  printf("\n");
}
```

Die Verwendung müssen wir dann auch entsprechend anpassen.

Programm 12.8: `kapitel_11/stack_main_v2.c`

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

#include "stack_v2.h"
```

```c

int main() {

  // Initialisierung
  struct node * head = NULL;

  head = push(head, 17);
  head = push(head, 22);
  head = push(head, 30);
  print_stack(head);

  int value = 0;
  head = pop(head, &value);
  printf("Wert %d zurueckbekommen\n",value);
  print_stack(head);

  printf("Liste ist leer: %s\n", is_empty(head)?"ja":"nein" );
  printf("Das naechste Element ist %d\n", peek(head));
  head = pop(head, &value);
  head = pop(head, &value);
  printf("Wert %d zurueckbekommen\n",value);
  printf("Liste ist leer: %s\n", is_empty(head)?"ja":"nein" );

  free_stack(head);

  return 0;
}
```

Beachten Sie, dass in dieser Version die Rückgabe der in dem Listenelement gespeicherten Information über eine an die Funktion übergebene Adresse erfolgt, da wir die Rückgabe über `return` schon für den Zeiger auf den Listenanfang brauchen. Da dies nicht besonders elegant ist, wird in vielen Implementierungen, die diesen Zugang verwenden, von der Funktion `pop` nur das erste Listenelement entfernt, aber das Element nicht zurückgegeben. Dafür muss dann vorher die Funktion `peek` aufgerufen werden.

12.3.2 Warteschlangen

Eine *Warteschlange* ist eine Liste, bei der wir uns darauf einschränken, dass Elemente nur am Ende der Liste eingefügt und nur am Anfang der Liste entfernt werden können. Der Ursprung des Namens kommt zum Beispiel von der Kassenschlange, wo Sie sich meistens hinten anstellen und langsam nach vorne vorrücken, wo ein Kassierer oder einen Kassiererin der Reihe nach Wartende aus der Warteschlange nimmt und den Einkauf abrechnet.

Auch bei Warteschlangen gibt es einige Funktionsnamen, die üblicherweise für die notwendigen Funktionen verwendet werden. Entweder gibt es in Analogie zu den Stapeln neben den Funktionen `peek` und `is_empty` noch die Funktionen `push` oder `push_back` zum Einfügen und `pop` oder `pop_front` zum Entfernen, oder diese Funktionspaar heißt `enqueue` (Einfügen) und `dequeue` (Entfernen). Hier ist eine einfache Implementierung.

Programm 12.9: kapitel_12/queue.h

```c

#ifndef QUEUE_H
#define QUEUE_H
```

```
4
5  #ifndef NODE_STRUCT
6  #define NODE_STRUCT
7  struct node {
8    int value;
9    struct node * next;
10 } ;
11
12 #endif
13
14 void print_queue(const struct node *);
15 void free_queue(struct node *);
16
17 void enqueue(struct node **, int);
18 int dequeue(struct node **);
19 int queue_is_empty(struct node *);
20 int peek_queue(struct node *);
21
22 #endif
```

Programm 12.9: kapitel_12/queue.c

```
1
2  #include<stdio.h>
3  #include<stdlib.h>
4  #include <limits.h>
5
6  #include "queue.h"
7
8  static struct node* init_node(int value) {
9    struct node * node = malloc(sizeof(struct node));
10   node->value = value;
11   node->next = NULL;
12   return node;
13 }
14
15 static void free_node(struct node * node) {
16   free(node);
17 }
18
19 void print_queue(const struct node * node) {
20   static int count = 1;
21   printf("Aufruf %d , die Liste enthaelt: ", count++);
22   while (node != NULL ) {
23     printf("%d ", node->value);
24     node = node->next;
25   }
26   printf("\n");
27 }
28
29 void free_queue(struct node * node ) {
30   struct node* keep = NULL;
31   while( node != NULL ) {
32     keep = node;
33     node = node->next;
34     free_node(keep);
35   }
36 }
```

```
37
38 void enqueue(struct node **head_ptr, int value) {
39   struct node* new_node = init_node(value);
40
41   if ( *head_ptr == NULL ) {
42     *head_ptr = new_node;
43   }  else {
44     struct node* node = *head_ptr;
45     while ( node->next != NULL ) {
46       node = node->next;
47     }
48     node->next = new_node;
49   }
50 }
51
52 int dequeue(struct node **head_ptr) {
53   int value = INT_MIN;
54   if( *head_ptr ) {
55     struct node * node = *head_ptr;
56     value = node->value;
57     *head_ptr = node->next;
58     free(node);
59   }
60
61   return value;
62 }
63
64 int queue_is_empty(struct node * head) {
65     return ( head == NULL );
66 }
67
68 int peek_queue(struct node * head) {
69   int value = INT_MIN;
70   if( head ) {
71     value = head->value;
72   }
73
74   return value;
75 }
```

Ein kleines Beispielprogramm wie das folgende zeigt die Verwendung unserer Warteschlange.

Programm 12.9: `kapitel_12/queue_main.c`

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  #include "queue.h"
5
6  int main() {
7
8    // Initialisierung
9    struct node * head = NULL;
10
11   print_queue(head);
12
13   enqueue(&head, 17);
14   enqueue(&head, 22);
```

```
15    enqueue(&head, 30);
16    print_queue(head);
17
18    int value = dequeue(&head);
19    printf("Wert %d zurueckbekommen\n",value);
20    print_queue(head);
21
22    value = dequeue(&head);
23    printf("Wert %d zurueckbekommen\n",value);
24    print_queue(head);
25    printf("Liste ist leer: %s\n", queue_is_empty(head)?"ja":"nein" );
26    printf("Das naechste Element ist %d\n", peek_queue(head));
27
28    value = dequeue(&head);
29    printf("%d vom Anfang der Liste genommen\n",value);
30    print_queue(head);
31    dequeue(&head);
32    print_queue(head);
33
34    printf("Liste ist leer: %s\n", queue_is_empty(head)?"ja":"nein" );
35    printf("Das naechste Element ist %d\n", peek_queue(head));
36
37    free_queue(head);
38
39    return 0;
40  }
```

Viele Programmierer nehmen umgekehrt auch an, dass eine Datenstruktur, die sie, zum Beispiel über eine Bibliothek, in ihr Programm einbinden und das diese Funktionsnamen bereitstellt, sich wie ein Stapel oder eine Warteschlange verhält, und die notwendigen drei Funktionen in der üblichen Bedeutung vorhanden sind. Wenn sie in einer Gruppe programmieren vermeiden Sie Überraschungen bei den anderen, wenn Sie sich an diese Konvention halten.

12.3.3 Laufzeit

Wir wollen uns noch überlegen, wie effizient unsere Implementierung eigentlich ist. Dafür wollen wir die Laufzeit der vier Operationen `push` bzw. `enqueue`, `pop` bzw. `dequeue` , `peek` und `is_empty` in Abängigkeit von der Länge des Stapels bzw. der Warteschlange bestimmen.

Für die Funktionen `peek` und `is_empty` ist das einfach. Die Funktion `peek` muss genau eine Referenz auflösen, während `is_empty` einen Vergleich durchführt. Beide Funktionen brauchen also konstante Zeit $\mathcal{O}(1)$. Das gleiche trifft auch auf `pop` und `dequeue` zu. Die Funktionen benötigen zwar mehr als eine Operation, aber Ihre Anzahl hängt nicht von der Länge der Liste ab. Auch das ist also eine konstante Operation.

Anders sieht es aus, wenn wir die beiden Einfügeoperationen betrachten. `push` muss ebenfalls nur eine konstante und von der Listenlänge unabhängige Anzahl Operationen machen. Die Funktion `enqueue` hingegen muss eine Kopie des Zeigers auf das erste Element erst bis ans Ende der Warteschlange bewegen. Dafür müssen bei einer Liste der Länge n auch n Referenzen auf den `next`-Zeiger aufgelöst werden. Die Funktion benötigt daher n Schritte und läuft also nur in Zeit $\mathcal{O}(n)$. Wir können das schon sehr deutlich mit dem folgenden Programm erkennen.

Programm 12.10: `kapitel_12/runtime.c`

```c
#include<stdio.h>
#include<stdlib.h>
#include <limits.h>
#include <time.h>

#include "stack.h"
#include "queue.h"

#define LAENGE 50000

int main() {

  // Initialisierung
  struct node * head = NULL;

  srand(time(0));
  clock_t anfang = clock();
  for ( int i = 0; i < LAENGE; ++i ) {
      push(&head,rand()%1000);
  }

  clock_t ende = clock();
  double zeit = ((double) (ende - anfang)) / CLOCKS_PER_SEC;

  printf("%d Elemente in %f Sekunden auf den Stapel gelegt\n",LAENGE,zeit);

  free_stack(head);
  head = NULL;

  srand(time(0));
  anfang = clock();

  for ( int i = 0; i < 50000; ++i ) {
      enqueue(&head,rand()%1000);
  }

  ende = clock();
  zeit = ((double) (ende - anfang)) / CLOCKS_PER_SEC;

  printf("%d Elemente in %f Sekunden an die Warteschlange angehaengt\n",LAENGE,zeit);

  free_queue(head);

  return 0;
}
```

Ein Aufruf des Programms gibt die ungefähr die folgende Ausgabe.

```
50000 Elemente in 0.002347 Sekunden auf den Stapel gelegt
50000 Elemente in 2.081194 Sekunden an die Warteschlange angehaengt
```

50000 Elemente an die Warteschlange anhängen dauert also rund eintausend Mal so lange wie die Elemente auf einen Stapel zu legen.

Wir können diesem Problem begegnen, indem wir bei Warteschlangen einen zweiten Zeiger mitführen, der immer auf das letzte Element der Schlange zeigt. Dafür müssen wir unsere Funktionen `enqueue` und `dequeue` zum Beispiel auf die folgende Weise

modifizieren.

Programm 12.11: kapitel_12/queue_tailptr.c

```c
void enqueue(struct node **head_ptr, struct node** tail_ptr, int value) {
  struct node* new_node = init_node(value);

  if ( *head_ptr == NULL ) {
    *head_ptr = new_node;
    *tail_ptr = new_node;
  }  else {
    (*tail_ptr)->next = new_node;
    *tail_ptr = new_node;
  }
}

int dequeue(struct node **head_ptr, struct node ** tail_ptr) {
  int value = INT_MIN;
  if( *head_ptr != NULL ) {
    struct node * node = *head_ptr;
    value = node->value;
    *head_ptr = node->next;
    free(node);
  }
  if ( *head_ptr == NULL ) {
      *tail_ptr = NULL;
  }
  return value;
}
```

Damit ändern sich natürlich auch die Aufrufe von `enqueue` und `dequeue`. Das folgende Programm zeigt das, und gibt eine Laufzeitmessung für die modifizierte Version.

Programm 12.12: kapitel_12/queue_tailptr_runtime.c

```c
#include<stdio.h>
#include<stdlib.h>
#include <limits.h>
#include <time.h>

#include "queue_tailptr.h"

#define LAENGE 50000

int main() {

  // Initialisierung
  struct node * head = NULL;
  struct node * tail = NULL;

  srand(time(0));
  clock_t anfang = clock();
  for ( int i = 0; i < LAENGE; ++i ) {
      enqueue(&head, &tail, rand()%1000);
  }

  clock_t ende = clock();
  double zeit = ((double) (ende - anfang)) / CLOCKS_PER_SEC;

```

```
25    free_queue(head);
26
27    printf("%d Elemente in %f Sekunden an die Warteschlange angehaengt\n",LAENGE,zeit);
28
29    return 0;
30  }
```

Damit erhalten wir

```
50000 Elemente in 0.002373 Sekunden an die Warteschlange angehaengt
```

Die neue Funktion `enqueue` hat wieder konstante Laufzeit $\mathcal{O}(1)$.

12.4 Dynamische Listen über Arrays

Die Implementierung von Listen über solche dynamischen Arrays ist zwar sehr flexibel und dieses Prinzip kommt an vielen Stellen zum Einsatz, durch die vielen Speicheroperationen ist es allerdings auch langsam im Vergleich zu den Listen fester Länge, die wir z.B. mit

```
1  int liste[30];
```

definieren können. Daher wird in vielen Implementierungen von Listen versucht, die Effizienz dieser Listen mit der Flexibilität der dynamischen Listen zu verbinden. Dafür speichern wir unsere Liste, also zum Beispiel einen Stapel oder eine Warteschlange, doch in einer Liste fester Länge und legen, wenn Elemente angefügt werden, eine neue, längere, Liste an und kopieren die alte Liste dort hinein.

In dieser Form ist das natürlich noch nicht effizienter, da wir nun beim Einfügen, statt für einzelnes Element Speicher zu reservieren und das Element dort abzuspeichern, Speicher für eine ganze Liste reservieren, die Liste dorthin kopieren und die alte Liste freigeben müssen. Daher gehen wir hier ähnlich vor wie bisher bei unseren Listen fester Länge. Am Anfang versuchen wir, eine Abschätzung über die benötigte Gesamtlänge zu treffen und legen eine Liste an, die, zumindest für die ersten Operationen, Platz ungenutzt lässt. Erst wenn wir keine Elemente mehr in der Liste speichern können, erweitern wir den Platz, indem wir eine neue Liste reservieren und die alte Liste kopieren. Dabei vergrößern wir die Liste aber nicht nur um die Plätze, die wir benötigen, um die aktuelle Anfrage zu bedienen, sondern legen gleich Platz für weitere Einfügungen an. Dabei können wir entweder jedes mal eine konstante Anzahl n an Plätzen hinzunehmen, oder die Länge der Liste um einen Faktor f verlängern. Meistens wird ein Faktor benutzt, der dann zwischen $3/2$ und 2 liegt, aber die richtige Wahl hängt auch von der Anwendung ab (davon, wie viele Elemente wir in welcher Rate erwarten).

Eine solche Konstruktion ist natürlich nur für Listen sinnvoll, bei denen wir nur am Anfang oder Ende einfügen wollen (wobei schon Einfügen am Anfang etwas Überlegung braucht), also nur für Warteschlangen und Stapel. Ein Element in der Mitte einfügen können wir nur, wenn wir den vorderen Teil um einen Platz nach links oder den hinteren Teil um einen Platz nach rechts kopieren. Damit müssten wir in jedem Schritt potetiell die gesamte Liste kopieren und verlieren somit den Vorteil gegenüber dynamischen verketteten Listen.

Wir schauen uns das einmal für den einfachsten Fall, also einen Stapel, genauer an. Die Anforderung neuen Speichers und das Kopieren können wir in C sehr effizient mit den Funktionen

```
void * realloc(void *zeiger, size_t neue_groesse);
void * memcpy( void *ziel, const void *quelle, size_t anzahl_char );
```

machen, wobei wir die zweite für unsere kleine Anwendung hier nicht benötigen. Die erste Anweisung **realloc** erweitert oder verkleinert den Speicherbereich, auf den `zeiger` zeigt, auf die Größe `neue_groesse`, kopiert, falls notwendig, den Inhalt, und gibt einen Zeiger auf den neuen Bereich zurück. Wenn der neue Bereich kleiner sein soll, wird nur der bestehende Bereich verkleinert und es muss nichts kopiert werden. Wenn der neue Bereich größer sein soll, wird zuerst versucht, den bestehenden Bereich zu erweitern. Wenn das nicht möglich ist, wird ein neuer Bereich belegt und der alte freigegeben. Falls kein ausreichender Speicherblock mehr belegt werden kann, wird nichts verändert und **NULL** zurückgegeben.

`memcpy` kopiert `anzahl_char` Zeichen von `quelle` nach `ziel`. Dabei muss der Speicherbereich, auf den `ziel` zeigt, ausreichen, um so viele Zeichen aufzunehmen, und `quelle` muss mindestens `anzahl_char` Zeichen enthalten. Andernfalls ist das Verhalten von `memcpy` undefiniert. Der Compiler kann eine Verletzung dieser Einschränkungen in der Regel nicht entdecken, es ist also Ihre Aufgabe als Programmierer*in, die Einhaltung der Bedingung sicherzustellen.

Da wir jetzt mit Listen arbeiten, deren maximale Länge die Länge der eigentlichen Liste übersteigt, müssen wir die Länge der Liste speichern. Ebenso brauchen wir die aktuelle maximale Länge, da wir sie nicht aus der Listenvariable auslesen können. Diese Information fasst man sinnvollerweise in einem `struct` zusammen, der zum Beispiel folgende Form haben kann.

```
struct stack {
  struct data ** elems;
  int mem_size;
  int n_elems;
} ;
```

Dabei ist `data` dann ein Zeiger auf die Liste, die wiederum vom Typ

```
  struct data liste[mem\_size]
```

ist. Wenn wir nun ein Element einfügen wollen, müssen wir zuerst testen, ob wir noch freien Speicher haben, also ob `mem_size` $>$ `n_elems`. Wenn das der Fall ist, fügen wir das neue Element hinten an und erhöhen `n_elems` um 1. Andernfalls erzeugen wir mit **realloc** eine größere Liste und lassen `elems` dorthin zeigen, bevor wir das Element anfügen.

Im folgenden Beispielprogramm führen wir das für eine Liste von Punkten in der Ebene aus. Jeder Punkt wird dabei durch seine x- und y-Koordinate gegeben, die in einem `struct data` zusammengefasst sind. Die wesentliche Funktion ist dabei die Funktion `push`, die bei Bedarf auch den Speicher vergrößert. Im Beispiel wird einmal angeforderter Speicher erst am Ende wieder freigegeben. Je nach Anwendung und Verfügbarkeit oder Verbrauch von Speicher kann es natürlich sinnvoll sein, auch zwischendrin wieder Speicher freizugeben, wenn die Länge des Stapels gefallen ist. Hierfür muss zwar niemals kopiert werden, aber da auch die Rückgabe des Speichers Zeit kostet, sollte

das nicht bei jedem entnommenen Element geschehen. Zudem sollten wir uns vor der Rückgabe von Speicher überlegen, wie wahrscheinlich es ist, dass wir kurz darauf die Liste wieder verlängern und dann neuen Speicher anfordern müssten.

Programm 12.13: `kapitel_12/stack_array.c`

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

struct data {
  int x;
  int y;
} ;

struct stack {
  struct data ** elems;
  int mem_size;
  int n_elems;
} ;


struct data* init_data(int x, int y) {
  struct data * data = malloc(sizeof(struct data));
  data->x = x;
  data->y = y;
  return data;
}

struct stack * init_stack(int n) {
  struct stack * s = malloc(sizeof(struct stack));
  s->elems = malloc(n*sizeof(struct data *));
  s->mem_size = n;
  s->n_elems = 0;
  return s;
}

void push(struct stack * s, struct data * d) {
  if ( s->n_elems == s->mem_size ) {
    printf("Resizing array\n");
    int new_size = s->mem_size ? s->mem_size * 2 : 4;
    s->elems = realloc(s->elems, (new_size+1) * sizeof(struct data *) );
    s->mem_size = new_size;
  }

  s->elems[s->n_elems++] = d;
}

struct data * pop(struct stack * s) {
  s->n_elems--;
  struct data * data = s->elems[s->n_elems];
  return data;
}

void print_data(const struct data * const data) {
  printf("(%d,%d) ", data->x, data->y);
}

```

```
53 void print_stack(const struct stack* const s) {
54   static int count = 1;
55   printf("Aufruf %d , der Stapel enthaelt: ", count++);
56   for ( int i = 0; i < s->n_elems; ++i ) {
57     print_data(s->elems[i]);
58   }
59   printf("\n");
60 }
61
62 void free_data(struct data * data) {
63   free(data);
64 }
65
66 void free_stack(struct stack * s ) {
67   for ( int i = 0; i < s->n_elems; ++i ) {
68     free_data(s->elems[i]);
69   }
70   free(s->elems);
71   free(s);
72 }
73
74 int main(int argc, char** argv) {
75
76   struct stack * s = init_stack(2);
77   for ( int i = 1; i < argc; i += 2 ) {
78     struct data * d = init_data(atoi(argv[i]), atoi(argv[i+1]));
79     push(s,d);
80   }
81
82   print_stack(s);
83
84   struct data * d = pop(s);
85   printf("(%d,%d) vom Stapel genommen\n", d->x, d->y);
86   free(d);
87
88   print_stack(s);
89
90   free_stack(s);
91
92   return 0;
93 }
```

Version vom 10. Februar 2022

13 Bibliotheken

In vielen Programmen brauchen wir immer wieder die gleichen oder ähnliche Methoden, um mit Teilaufgaben der Problemstellung umzugehen. Es ist in der Regel nicht sinnvoll, diese Aufgaben jedes Mal neu zu analysieren und eine passende Implementierung zu schreiben. Wir wollen hier auf vorbereitete Lösungen zurückgreifen, die wir oder andere vorher schon erstellt haben. Eine Möglichkeit dafür ist natürlich, den passenden Code jedes mal in unser Programm zu kopieren. Damit wächst allerdings unser eigener Code je nach Komplexität des kopierten Codes, und es wird erheblich schwieriger, einen Überblick über die Teile zu behalten, die wir selbst geschrieben haben und die wir daher pflegen, kontrollieren und möglicherweise von Fehlern bereinigen müssen. Wenn wir auf diese Weise fremden mit eigenem Code vermischen, entsteht aber ein noch deutlich größeres Problem. Auch der kopierte Code könnte noch fehlerhaft sein, oder der Autor fügt, vielleicht auch auf Ihren Wunsch hin, neue Methoden in seinen Code ein. Wenn Sie nun alles in Ihren Code kopiert haben, müssen Sie dies jedes mal wiederholen, wenn sich der fremde Code ändert, und dabei aufpassen, dass Sie keinen alten Code in Ihrem Programm lassen, den neuen Code komplett übernehmen, und in Ihrem Code darauf achten, dass Sie alle Anpassungen, die Sie vielleicht gemacht haben, auch korrekt übernehmen. Um diese Probleme zu vermeiden, sieht `C` von Anfang an die Möglichkeit vor, fremden Code über klar definierte Schnittstellen einzubinden, ohne dass wir den vollständigen Code übernehmen und kopieren müssen.

13.1 Einbinden fremder Header mit Quelldateien

Eine erste Möglichkeit der Trennung von Code in verschiedene Einheiten haben wir schon in Abschnitt 4.2 kennengelernt. Dort haben wir unseren eigenen Code in mehrere Teile zerlegt und auf verschiedene Dateien aufgeteilt. Dabei ist es sinnvoll, zusammengehörende Teile des Codes jeweils in einem Paar aus zwei Dateien zu verteilen, in dem die erste Datei (der *Header*) in der Regel nur die Deklarationen enthält, und die zweite Datei (die *Quelldatei)* die zugehörigen Definitionen. Auf diese Weise ist es im restlichen Code ausreichend, den Header über `#include` einzubinden. Außerdem brauchen wir nur Funktionen im Header deklarieren, die wir auch wirklich zur Verwendung in anderen Teilen des Programms benötigen. Für Funktionen, die nur innerhalb dieser Einheit aufgerufen werden, reicht es, sie in der Quelldatei zu definieren. Wir können solche Funktionen auch mit dem Schlüsselwort `static` definieren, damit sie außerhalb dieser Datei nicht sichtbar sind. Das gibt uns die Möglichkeit, Veränderungen im Code zu machen, die sich nicht auf des restliche Programm auswirken, solange wir die Deklaration der Funktionen im Header nicht verändern (und die Funktionen noch machen, was wir in der Dokumentation dafür angegeben haben).

Version vom 10. Februar 2022

Auf diese Weise können wir natürlich auch fremden Code in unser Programm einbinden. Wir übernehmen dafür Header und Quelldatei(en), und binden den Header über `#include` in unser Programm ein. Bei der Verwendung müssen wir auch nur in diesem Header nachsehen, welche Funktionen der fremde Code bereitstellt und (in der hoffentlich vorhandenen) Dokumentation nachsehen, was diese Funktionen machen, welche Eingaben sie erwarten und welche Annahmen daran sie möglicherweise treffen. Diese im Header deklarierten Funktionen nennt man oft *Schnittstelle* oder *API* (für *application programming interface*) der Bibliothek. Durch Austausch der Quelldatei können wir Änderungen und Verbesserungen der Bibliothek in unser Programm übernehmen ohne daran Anpassungen vornehmen zu müssen, solange sich die Schnittstelle nicht ändert.

Diese einfache Möglichkeit fremde Bibliotheken einzubinden, schauen wir uns an einem Beispiel an. Wir benutzen dafür die Bibliothek `tinyexpr` von Lewis Van Winkle. Wir können sie unter github.com/codeplea/tinyexpr finden. Sie ist unter der *zlib-Lizenz* veröffentlicht und darf im Wesentlichen frei verwendet werden[1]. Mit der Bibliothek können wir mathematische Ausdrücke in mehreren Variablen parsen und auswerten.

Ein Blick in den Header zeigt uns, dass die Bibliothek Strukturen

- `te_expr` für einen mathematischen Ausdruck und
- `te_variable` für eine Variable

definiert und und fünf Funktionen deklariert:

- `te_interp`, die einen Ausdruck liest und an einer Stelle auswertet,
- `te_compile`, die einen Ausdruck liest und in einem `te_expr` speichert,
- `te_eval`, die einen Ausdruck vom Typ `te_expr` an einer Stelle auswertet,
- `te_print`, die einen Ausdruck als Zeichenkette zurückgibt, und
- `te_free`, die den Speicher eines `te_expr` wieder freigibt.

Programm 13.1: `tinyexpr.h` (github.com/codeplea/tinyexpr)

```
1  typedef struct te_expr {
2      int type;
3      union {double value; const double *bound; const void *function;} ;
4      void *parameters[1];
5  }  te_expr;
6
7  typedef struct te_variable {
8      const char *name;
9      const void *address;
10     int type;
11     void *context;
12 }  te_variable;
13
14 /* Parses the input expression, evaluates it, and frees it. */
15 /* Returns NaN on error. */
16 double te_interp(const char *expression, int *error);
17
18 /* Parses the input expression and binds variables. */
19 /* Returns NULL on error. */
```

[1] die genauen Bedingungen stehen hier: opensource.org/licenses/Zlib, eine Kopie ist auch im Repository der Software enthalten.

```
20 te_expr *te_compile(const char *expression, const te_variable *variables, int var_count,
       int *error);
21
22 /* Evaluates the expression. */
23 double te_eval(const te_expr *n);
24
25 /* Prints debugging information on the syntax tree. */
26 void te_print(const te_expr *n);
27
28 /* Frees the expression. */
29 /* This is safe to call on NULL pointers. */
30 void te_free(te_expr *n);
```

Mit diesen Informationen und einem Blick in die Dokumentation können wir direkt ein kleines Programm schreiben, das einen Ausdruck mit der Funktion `te_interp` auswertet. Hier berechnen wir eine Approximation des goldenen Schnitts.

Programm 13.2: `kapitel_13/tinyexpr_bsp_01.c`

```
1  #include <stdio.h>
2
3  #include "tinyexpr.h"
4
5  int main(int argc, char ** argv) {
6
7      const char *ausdruck = "(1+sqrt(5))/2";
8      double wert = te_interp(ausdruck, 0);
9
10     printf("%s = %f\n", ausdruck, wert);
11
12     return 0;
13 }
```

Für die Übersetzung müssen wir bei der Wahl von `gcc` als Übersetzer noch die Option `-lm` angeben. Das lädt eine Bibliothek mit mathematischen Funktionen, die aus eher historischen Gründen nicht in der C-Standardbibliothek integriert ist[2]. Der Übersetzer `clang` lädt diese Bibliothek automatisch und die Option muss nicht angegeben werden. Damit sehen wir im Terminal die folgende Ausgabe.

```
\$ gcc tinyexpr.c tinyexpr_bsp_01.c -o tinyexpr_bsp_01 -lm
\$ ./tinyexpr_bsp_01
(1+sqrt(5))/2 = 1.618034
```

Mit der Funktion `te_compile` können wir auch einen Ausdruck mit Parametern einlesen und über `te_eval` dann immer wieder auswerten. Als Beispiel nehmen wir die Aufgabe, eine Funktion in einer Variablen x sowie eine Reihe von Werten für x einzulesen, den Ausdruck an diesen Werten auszuwerten und die Werte sowie die Auswertung in einer Tabelle darzustellen. Ein Programm, das diese Aufgabe löst, könnte zum Beispiel wie das folgende Programm aussehen.

[2] Eigentlich zerfällt diese Option in zwei Teile: Die Option selbst heißt `-l`, und sie nimmt einen Bibliotheksnamen als Parameter. Die Bibliothek, die wir hier laden wollen, heißt `m` oder `libm` für mathematische Funktionen. Da wir für unser Programm nichts darüber wissen müssen (das passiert alles innerhalb von `tinyexpr`), können Sie diese Bibliothek hier einfach einsetzen. Im nächsten Abschnitt lernen wir die Option `-l` für eine Bibliothek kennen, aus der wir Funktionen direkt in unserem Programm verwenden wollen.

Programm 13.3: kapitel_13/tinyexpr_bsp_02.c

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

#include "tinyexpr.h"

int main(int argc, char** argv) {

    if (argc < 2) {
        printf("Aufruf: %s ausdruck x1 x2 ... \n",argv[0]);
        return 0;
    }

    double x, y;
    te_variable varnames[] = {{"x", &x} } ;

    int fehler;
    te_expr *ausdruck = te_compile(argv[1], varnames, 1, &fehler);

    if ( !ausdruck ) {
        printf("\t%s\n", argv[1]);
        printf("\t%*s^\nFehler im Ausdruck an dieser Stelle\n", fehler-1, "");
        return 1;
    }

    int count = 2;
    printf("\n%5c | %s\n---------------\n",'x',argv[1]);
    while ( count < argc ) {
        x = atof(argv[count++]);
        y = te_eval(ausdruck);
        printf("%5.2f | %.2f\n", x,y);
    }

    te_free(ausdruck);

    return 0;
}
```

Wir betrachten als Beispiel die logistische Funktion $x \mapsto \frac{1}{1+e^{-x+\frac{1}{2}}}$ und werten diese Funktion an allen ganzen Zahlen zwischen -5 und 5 aus.

```
\$ gcc tinyexpr.c tinyexpr_bsp_02.c -o tinyexpr_bsp_02
\$ ./tinyexpr_bsp_02 "1/(1+e^(-x+1/2))" -5 -4 -3 -2 -1 0 1 2 3 4 5

    x | 1/(1+e^(-x+1/2))
---------------
-5.00 | 0.00
-4.00 | 0.01
-3.00 | 0.03
-2.00 | 0.08
-1.00 | 0.18
 0.00 | 0.38
 1.00 | 0.62
 2.00 | 0.82
 3.00 | 0.92
 4.00 | 0.97
 5.00 | 0.99
```

13.1.1 Exkurs: gnuplot

Bei der Arbeit mit Polynomen $x \mapsto p(x)$ möchten wir die Funktion auch gerne als Graph zu $(x, p(x))$ zeichnen können. Mit Hilfe graphischer Bibliotheken (wir nennen einige am Ende dieses Kapitels) können wir das prinzipiell auch innerhalb von `C` programmieren. Dazu müssen wir allerdings zuerst eine graphische Oberfläche initialisieren und dann unseren Graph darin darstellen. Das ist sehr aufwendig und geht über die Ziele dieser Vorlesung weit hinaus. `C` ist als Sprache auch nur bedingt dafür geeignet und wird nicht so oft für die Implementierung graphischer Oberflächen eingesetzt.

Für unsere Zwecke bietet es sich eher an, auch schon bestehende Programme zurückzugreifen, die anhand vorgegebener Daten eine Visualisierung erstellen und anzeigen können. Dann besteht der Gesamtprozess für die Anzeige unseres Funktionsgraphen aus zwei Schritten. Für den ersten Schritt schreiben wir ein Programm in `C`, dass die Daten, die wir anzeigen wollen, erzeugt und z.B. in eine Datei schreibt. In einem zweiten Schritt rufen wir dann mit diesen Daten ein Programm zur Visualisierung auf.

Wir wollen hier exemplarisch das Programm `gnuplot`[3] vorstellen. Das Programm ist frei verfügbar und kostenlos, allerdings nicht unter einer freien Lizenz lizenziert. `gnuplot` hat sehr viele und komplexe Möglichkeiten, Daten in 2D und 3D zu visualisieren. Wir wollen hier nur eine sehr einfach Option nutzen. Bei Interesse können Sie in der Anleitung zu gnuplot weitere Möglichkeiten lernen. Das Programm kann interaktiv benutzt werden. Nach dem Start im Terminal erhalten Sie den prompt von `gnuplot`.

```
> gnuplot

        G N U P L O T
        Version 5.4 patchlevel 2    last modified 2021-06-01

        Copyright (C) 1986-1993, 1998, 2004, 2007-2021
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')

Terminal type is now 'qt'
gnuplot>
```

Je nach Betriebssystem müssen Sie ggf. spezifizieren, mit welchem *window manager* Sie Ihre Daten anzeigen lassen wollen. Bei Linux ist das in der Regel

```
set terminal 'x11'
```

und bei MacOS

```
set terminal 'qt'
```

Danach können Sie Zeichenkommandos in `gnuplot` eingeben, die dann in einem Fenster angezeigt werden, das sich beim ersten Aufruf eines Zeichenbefehls öffnet.

Wenn wir in einer Datei `graph.dat` zeilenweise Paare `x y` von Daten $(x, y = p(x))$ unseres Polynoms haben, also z.B. in der Form

```
[...]
```

[3] gnuplot.info/

```
0.83664 0.69997
0.83666 0.70000
0.83668 0.70004
0.83670 0.70007
0.83672 0.70010
0.83674 0.70014
0.83676 0.70017
0.83678 0.70020
0.83680 0.70024
0.83682 0.70027
0.83684 0.70030
[...]
```

für das Polynom $p : x \mapsto x^2$, dann können wir uns das in `gnuplot` mit

```
plot 'graph.dat' with lines
```

anzeigen lassen. `gnuplot` wählt die Achsenskalen so, dass die Fensterfläche gut ausgenutzt wird. Das führt in der Regel dazu, dass die Abstände auf der x- und y-Achse nicht gleich sind. Das ist bei der Anzeige von Daten mit Einheiten (z.B. statistische oder physikalische Daten) sicher sinnvoll, bei unserem Graphen aus mathematischer Sicht nicht. Sie können mit

```
gnuplot> set size ratio -1
```

gleiche Skalen in x- und y-Richtung erzwingen.

Nun brauchen wir auch noch ein Programm, mit dem wir die Daten erzeugen, die wir anzeigen lassen wollen. Mit `tinyexpr` können unser Beispiel oben leicht modifizieren. Unser Programm könnte z.B. wie folgt aussehen.

Programm 13.4: `kapitel_13/tinyexpr_bsp_03.c`

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

#include "tinyexpr.h"

int main(int argc, char** argv) {

    if (argc < 6) {
        printf("Aufruf: %s <Polynom> <Anfang> <Ende> <Schrittweite> <Datei> \n",argv[0])
    ;
        return 0;
    }

    double x, y;
    te_variable varnames[] = {{"x", &x} } ;

    int fehler;
    te_expr *ausdruck = te_compile(argv[1], varnames, 1, &fehler);

    if ( !ausdruck ) {
        printf("\t%s\n", argv[1]);
        printf("\t%*s^\nFehler im Ausdruck an dieser Stelle\n", fehler-1, "");
        return 1;
    }

    int count = 2;
```

```
26      float lower = atof(argv[2]);
27      float upper = atof(argv[3]);
28      int steps = atoi(argv[4]);
29      float dist = (upper-lower)/steps;
30      float val = lower;
31
32      FILE *outfile;
33      outfile = fopen (argv[5], "w");
34     if (outfile == NULL) {
35       fprintf(stderr, "\nDatei kann nicht geoeffnet werden\n");
36       exit (1);
37     }
38      while ( val < upper ) {
39          x = val;
40          y = te_eval(ausdruck);
41          fprintf(outfile, "%.5f %.5f\n", x,y);
42          val += dist;
43      }
44
45      te_free(ausdruck);
46
47      return 0;
48  }
```

Wir können das Programm nach dem Übersetzen z.B. mit

```
./tinyepr_bsb_03 "1/10*x^3-1/2*x^2+1/8*x" -2 5 100000 graph.dat
```

aufrufen, und dann in `gnuplot`

```
        G N U P L O T
        Version 5.4 patchlevel 2    last modified 2021-06-01

        Copyright (C) 1986-1993, 1998, 2004, 2007-2021
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')

    gnuplot> set terminal 'qt'
    Terminal type is now 'qt'
    gnuplot> set size ratio -1
    plot 'graph.dat' with lines
```

aufrufen. Die Ausgabe könnte dann wie in Abbildung 13.1 aussehen.

Das Programm `gnuplot` bietet noch viele andere Möglichkeiten, Daten graphisch darzustellen. Sie können je nach Anwendung ausprobieren. Die prinzipielle Methode bleibt allerdings in allen Fällen die gleiche. Sie müssen zuerst ein Programm schreiben, dass Ihre Daten erzeugt und dann in einer Form, die `gnuplot` interpretieren kann, in eine Datei schreibt. Im Anschluss rufen Sie dann `gnuplot` mit diesen Daten auf. Wie Dateien je nach gewünschter Form der Anzeige aussehen müssen, damit `gnuplot` sie interpretieren kann, können Sie der Anleitung von `gnuplot` entnehmen.

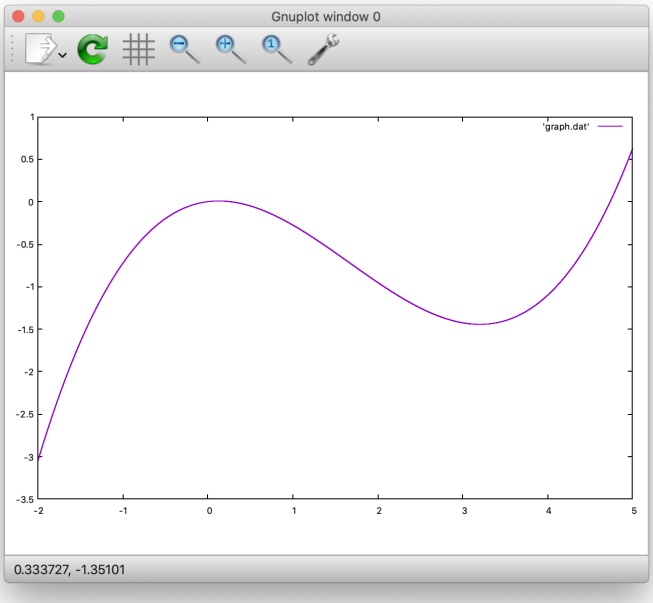


Abbildung 13.1: Die Ausgabe von gnuplot

13.2 Bibliotheken

Vollständigen Quellcode in das eigene Projekt zu kopieren ist, wie wir im vorangegangenen Abschnitt gesehen haben, möglich und bei kleinen Projekten vielleicht auch sinnvoll. Wenn wir größere Projekte einbinden wollen, wird unser Code dadurch jedoch unnötig aufgebläht und es wird schwierig werden, die eigenen Teile darin zu finden. Bei einem Update des fremden Codes müssen wir dann auch nachvollziehen, welche Dateien sich geändert haben, ob neue hinzugekommen sind oder alte weggefallen sind.

Da wir in unserem Programm eigentlich nur die Informationen aus dem Header benötigen, brauchen wir den Code aus den Quelldateien auch nicht im Original. Wir haben schon gesehen, dass wir beim Übersetzen von Code, der aus einem Hauptprogramm und Bibliotheken besteht, alle Quelldateien an `gcc` übergeben muss. Tatsächlich ist es dann auch so, dass `gcc` alle diese Dateien einzeln und der Reihe nach in Maschinencode übersetzt, bevor diese Teile in einem zweiten Schritt zu einem Programm zusammengefügt werden. Wir können diese zwei Schritte auch einzeln ausführen und erst nacheinander alle Teile in Maschinencode übersetzen und dann selbst die Teile (mit dem sogenannten *Linker*) zusammenfügen. Wie das praktisch geht, schauen wir uns später an. Hier wollen wir ausnutzen, dass es uns ausreicht, wenn die fremden Bibliotheken nur in übersetzter Form vorliegen, solange wir den Header im Original haben. Das hat (unter anderen) drei Vorteile:

- Wir verändern den eingebundenen Code nicht. Ihn also jedes mal, wenn wir unser Programm übersetzen, ebenfalls mitzuübersetzen, kostet unnötig Zeit, da dafür jedes Mal der gleiche Maschinencode erzeugt wird.
- Wenn wir nur den Header kennen, aber nicht den zugehörigen Quellcode mit

den Definitionen, dann können wir von der Bibliothek auch nur die im Header bekanntgegebenen Funktionen nutzen. Wir können also sicher sein, dass wir nur Funktionen benutzen, die offiziell Teil der Schnittstelle sind und nicht aus Versehen auch andere Funktionen verwenden, die vielleicht in einer neuen Version nicht mehr (in dieser Form) zur Verfügung stehen.

- ▷ Als Anbieter solcher Bibliotheken möchten wir vielleicht den von uns geschriebenen Code davor schützen, dass Anwender Teile oder den gesamten Code kopieren und in eigenen Programmen verwenden. Insbesondere, wenn wir neue Algorithmen entwickelt oder viel Zeit in eine effiziente Umsetzung verwendet haben und das Ergebnis verkaufen möchten, möchten wir uns gegen Kopieren schützen. Wenn wir nur den Maschinencode verteilen, machen wir eine Übernahme in andere Programm zumindest erheblich schwieriger.

Diese Methode ist die übliche Methode, Bibliotheken einzubinden, und zwar auch dann, wenn wir den kompletten Quellcode zur Verfügung hätten. Dazu wird der fremde Code zu einer sogenannten *statischen Bibliothek* (*static library*) übersetzt[4]. Wir müssen dann `gcc` beim Übersetzen unseres Programms mitteilen, dass diese Bibliothek eingebunden werden soll. Das passiert mit der Option `-l` an gcc, die den Namen der Bibliothek übernimmt[5]. Oft liegen solche Bibliotheken, wenn es keine Standardbibliotheken des Systems sind oder im Paketmanagement der Distribution enthalten sind und dann an einer Standardstelle installiert werden, nicht in Verzeichnissen, in denen `gcc` ohne Hinweis sucht. Daher gibt es noch die zwei Optionen `-I` und `-L`, mit denen wir einen Pfad zu den Headern bzw. einen Pfad zu den übersetzten Bibliotheken angeben können.

13.2.1 gmp

Wir haben schon ganz am Anfang gesehen, dass die ganzzahligen Typen `int`, auch mit Modifikationen, einen eher eingeschränkten Zahlraum haben. Für Mathematiker noch unbefriedigender ist es, dass wir nicht sinnvoll und effizient mit rationalen Zahlen rechnen können. Wir können rationale Zahlen als Paar ganzer Zahlen darstellen, aber dafür funktionieren natürlich die vorhandenen arithmetischen Operatoren nicht mehr und wir müssen eigene Funktionen dafür bereitstellen.

Für die Darstellung beliebig großer ganzer Zahlen und für das exakte Rechnen mit rationalen Zahlen gibt es zum Glück eine leistungsfähige und gut dokumentierte Bibliothek, die `gmp` (für *GNU Multiple Precision Arithmetic Library*). Die Bibliothek ist unter einen freien Lizenz (GPL v3) lizenziert. Sie können sie auf gmplib.org finden, aber fast alle Distributionen haben ein eigenes Paket dafür, das oft auch schon installiert ist.

Auch die Bibliothek `gmp` kann (wenigstens in der Sprache `C`, in der Version für `C++` ist das anders) nicht die vorhandenen arithmetischen Operatoren neu definieren, daher

[4] es gibt auch *dynamische Bibliotheken* (*dynamic libraries*), die wir hier nicht betrachten. Ihre Anbindung ist aber nicht wesentlich anders. Im Unterschied zu den statischen Bibliotheken, bei denen, wie bei unseren bisherigen Programmen auch, alle Funktionsdefinitionen der Bibliothek zusammen mit den von uns geschriebenen Funktionen am Ende in eine einzige ausführbare Datei zusammengepackt werden, werden bei dynamischen Bibliotheken nur Referenzen auf die Funktionen abgelegt, und erst zum Zeitpunkt der Ausführung wird der Maschinencode der Bibliothek mit den Definitionen geladen.

[5] Üblicherweise sind statische Bibliotheken im Dateisystem mit der Endung `.a` abgelegt und der Dateiname setzt sich aus `lib` und dem Bibliotheksnamen zusammen. Zum Beispiel heißt die Datei der Bibliothek `gmp`, die wir gleich betrachten, `libgmp.a`.

stellt sie eigenen Funktionen für alle notwendigen Operationen bereit. Sie kann auch nicht mit den Standarddatentypen wie `int` arbeiten, so dass sie, wie wir für die verketteten Listen, auf die Deklaration von `struct`s zurückgreift, die alle Informationen zu einem bestimmten Zahltyp bündelt. Für Operationen auf diesen Zahltypen, wie z.B. die Addition, benötigen wir dann spezielle Funktionen, die auf diesen Zahltypen arbeiten können.

Für die Liste der zur Verfügung stehenden Funktionen können Sie zwar auch hier wieder im Header nachsehen[6], aber die Bibliothek stellt so viele Funktionen und auf spezielle Systemgegebenheiten abgestimmte Optimierungen davon bereit, dass eine Dokumentation alleine über den Header nicht mehr sinnvoll ist. Sie sollten die Dokumentation daher auf der Webseite gmplib.org/manual/ oder, wenn installiert, in der lokalen Dokumentation nachsehen[7]. Dort finden wir zur Initialisierung einer ganzen Zahl zum Beispiel die Information, dass diese mit dem Typ `mpz_t` deklariert werden, also in der Form

```
mpz_t x;
```

in Analogie zu `int x`, und unter anderen die Funktionen

```
// Eine ganze Zahl initialisieren
void mpz_init (mpz_t x);

// Den Speicher einer ganzen Zahl wieder freigeben
void mpz_clear (mpz_t x);

// eine ganze Zahl auf eine andere zuweisen
void mpz_set (mpz_t rop, const mpz_t op);

// eine Zahl vom Typ unsigend int in eine ganze Zahl zuweisen
void mpz_set_ui (mpz_t rop, unsigned long int op);

// Addition
void mpz_add (mpz_t rop, const mpz_t op1, const mpz_t op2);

// Addition einer Zahl mit einem unsigned long
void mpz_add_ui (mpz_t rop, const mpz_t op1, unsigned long int op2)

// Ausgabe auf den Bildschirm oder in eine Datei
size_t mpz_out_str (FILE *stream, int base, const mpz_t op);
```

Es gibt noch viele weitere, aber anhand dieser können wir das Prinzip der Funktionen in der `gmp` schon erkennen. Da die Zahlen in einem `struct` gespeichert werden und variable Größe haben können, muss die Bibliothek mit dynamisch reserviertem Speicher arbeiten, den sie bei Bedarf erweitert. Wir brauchen also eine Funktion, die diese Speicherreservierung vornimmt, und eine weitere, die den Speicher wieder freigibt, wenn wir die Variable nicht mehr benötigen. Das geschieht mit den ersten beiden Funktionen in der Liste oben. `mpz_init` reserviert den Speicher für `x` und `mpz_fclear` gibt ihn wieder frei.

Wir können nicht die üblichen Operatoren auf den einfachen Datentypen benutzen und müssen daher alle Operatoren über Funktionen organisieren. Die Zuweisung erfolgt

[6] auf Linuxsystemen liegt diese Datei oft unter `/usr/include/x86_64-linux-gnu/gmp.h`, wenn sie installiert ist, aber ggf. müssten Sie danach suchen.

[7] Diese liegt oft unter `/usr/share/doc/gmp-doc/`.

in der gmp dabei mit mpz_set, die zwei Variablen als Argument nimmt und den Inhalt der zweiten in die erste kopiert. Wir erkennen aber schnell, dass das nicht ausreicht, da wir zuerst eine mit einer Zahl belegten Variable haben müssen, bevor wir damit auf eine andere Variable zuweisen können. Daher gibt es weitere Methoden, die an der zweiten Stelle einen der Standarddatentypen erwarten. Oben ist die Funktion mpz_set_ui aufgeführt, die ein unsigned long akzeptiert (was dann auch eine konstante Zahl sein kann). Damit wir auch Zahlen zuweisen können, die größer sind als unsigned long, gibt es auch eine solche Funktion, die eine Zeichenkette übernimmt, so dass wir unsere *große* Zahl als Zeichenkette einlesen und dann in eine Zahl im gmp-Format umwandeln können. Es gibt natürlich auch die umgekehrten Funktionen, was bei einer Zuweisung auf z.B. int natürlich nur sinnvoll ist, wenn die Zahl klein genug ist um mit int dargestellt werden zu können.

Auch die Addition, sowie alle restlichen arithmetischen Operatoren, müssen wir dann über solche Funktionen lösen. Angegeben ist die Funktion mpz_add, die drei Argumente übernimmt, die zwei Summanden, und eine Variable, in der das Ergebnis gespeichert werden soll. Da mpz_t intern Zeiger in den Speicher bereitstellt, könnten Sie bei Betrachtung der Funktion ein mögliches Problem bei der Verwendung sehen. Für die Standarddatentypen war es erlaubt, auf eine Variable, die wir in einer arithmetischen Operation verwendet haben, direkt wieder zuzuweisen, zum Beispiel in der Form

```
int i = 10;
i = i+1;
```

Die entsprechende Version für Zahlen vom Typ mpz_t sieht dann zum Beispiel wie im folgenden Ausschnitt aus

```
mpz_t i;
mpz_init(i);
mpz_set_ui(i,10);
mpz_add_ui(i,i,1);
```

Die Funktion mpz_add darf aber nun den Inhalt des ersten an sie übergebenen Arguments verändern. Als Zeiger verweist es auf den gleichen Speicherbereich wie die zweite Variable. Wenn nun die Funktion diesen Speicherbereich verändert, bevor die Addition ausgeführt wird (zum Beispiel, in dem es ihn mit 0 initialisiert), dann ist der Wert von i verloren, bevor die Addition ausgeführt wird. Die gmp garantiert für alle ihre Funktionen, dass das nicht passiert und der Inhalt von i erst überschrieben wird, wenn der Speicherinhalt nicht mehr für die Berechnung des Ergebnisses benötigt wird.

Es kann allerdings in solchen Situationen, wo wir Berechnungen mit den an einer Speicheradresse vorhandenen Informationen durchführen wollen und das Ergebnis eigentlich an der gleichen Stelle speichern wollen, für die Implementierung effizient sein, den Speicherbereich für das Ergebnis von Anfang an benutzen zu können. Es gibt daher auch viele Bibliotheken, bei denen die Verwendung der gleichen Variable für Ein- und Rückgabe nicht erlaubt ist. Der Compiler kann nicht erkennen, ob eine Funktion in diesem Sinn falsch verwendet wird, wir müssen immer in der Dokumentation nachsehen, wenn wir unsicher sind, ob eine Verwendung wie bei der gmp zulässig ist.

Die konkrete Benutzung der gmp wollen wir uns nun an zwei Beispielen ansehen, die hoffentlich die Verwendung der Funktionen anschaulich erklären. Zuerst betrachten wir ein Beispiel, in dem wir große ganze Zahlen benötigen. Dafür schauen wir uns die Folge

der *Fibonaccizahlen* $(f_k)_{k\in\mathbb{Z}_{\geq 0}}$ an. Diese Folge ist über die folgende Vorschrift definiert.

$$f_0 = 0 \qquad f_1 = 1 \qquad f_{k+1} = f_k + f_{k-1} \quad \text{für } k \geq 1$$

Der Anfang der Folge ist damit

$$1\,,\ 1\,,\ 2\,,\ 3\,,\ 5\,,\ 8\,,\ 13\,,\ 21\,,\ 34\,,\ 45\,,\ 89\,,\ 134\,,\ \ldots$$

Die Größe der Folgenglieder steigt recht schnell an. Hier ist eine mögliche Implementierung.

Programm 13.5: `kapitel_13/fibonacci.c`

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

#include "gmp.h"

void fibonacci(mpz_t f, int n) {

    mpz_init(f);                // Initialisieren (zur Sicherheit)
    mpz_set_ui(f,0);            // Initialisieren (zur Sicherheit)
    if ( n == 0 ) {
        return;
    }
    if ( n == 1 ) {
        mpz_set_ui(f,1);
        return;
    }

    mpz_t g;                    // wir brauchen zwei vorangegangene Folgenglieder
    mpz_init(g);
    mpz_set_ui(g,1);

    short even = 1;             // wir speichern das neueste Folgenglied
                                // abwechselnd in f und g, je nachdem
                                // in welchem das vorletzte steht
                                // even=1: g, even=0: f
    for ( int i = 2; i <= n; ++i, even = 1-even ) {
        even ? mpz_add(f,f,g) : mpz_add(g,f,g);     // Addition
    }

    if ( even )                 // in dem Fall steht das letzte Folgenglied in g
        mpz_set(f,g);

    mpz_clear(g);               // Speicher freigeben
}

int main(int argc, char** argv) {

    int k = atoi(argv[1]);                  // ein Argument wird eingelesen

    if ( k > 1000000 ) {
        printf("number too large\n");
        return 0;
    }

    mpz_t f;                                // Ein gmp-Integer fuer das Ergebnis
```

```
46      mpz_init(f);                            // Initialisieren
47      mpz_set_ui(f,0);                        // --- '' ---
48      fibonacci(f,k);
49      printf("Die %d-te Fibonaccizahl ist ",k);
50      mpz_out_str(stdout,10,f);
51      printf("\n");
52
53      mpz_clear(f);                            // Speicher freigeben
54
55      return 0;
56  }
```

Bei der Übersetzung müssen wir jetzt daran denken, dass wir die Bibliothek gmp mit der Option `-l` einbinden müssen.

```
\$ gcc fibonacci.c -o fibonacci -l gmp
```

Danach können wir unser Programm benutzen, zum Beispiel wie folgt.

```
\$ ./fibonacci 10
Die 10-te Fibonaccizahl ist 55
\$ ./fibonacci 40
Die 40-te Fibonaccizahl ist 102334155
\$ ./fibonacci 160
Die 160-te Fibonaccizahl ist 1226132595394188293000174702095995
\$ ./fibonacci 640
Die 640-te Fibonaccizahl ist 252699718319263320894925711359827010929017154482394239
801161275085293506253962890752693293024106582769745280308753764470944896883433 55
\$ ./fibonacci 5000
Die 5000-te Fibonaccizahl ist 387896845438832563370191630832590531208212771464624 51
061605972148955501390440370970108229164622106694792934528588829738134831020089549 82
940361430156911478938364216563944106910214505634133706558656238254656700712525929 90
385493381392883637834751890876297071203333705292310769300851809384980180384781399 67
488817655546537882916442689129803846137789690215022930824756663462249230718833248 03
280375039130352903304505842701147635242270210934637699104006714174883298422891491 27
310405432875329804427367682297724498774987455569190770388063704683279481135897373 99
931101062193081490185708153978543791953056175107610530756887837660336673554452588 44
886241619210553457493675897849027988234351023599844663934853256411952221859563060 47
536464547076033090242080638258492915645287629157575914234380914230291749108898415 52
098544324865940797935713168416928680395453095453886981146650820668628974206393234 38
488465240988742395873801976993820317174208932265468879364002630797780058759129671 38
963421425257911687275560036031137054775472460463998758804698517840867438286312 5
```

Spätestens bei der letzten Berechnung ist offensichtlich, dass wir den für `unsigned int` vorgesehenen Bereich verlassen haben (und auch den Bereich für `double`, so dass wir auch keine Approximation auf diese Weise speichern könnten).

Wenn Sie das Programm selbst geschrieben hätten, hätten Sie vielleicht versucht, das Programm *rekursiv* zu schreiben, also mit einer Funktion `fibonacci(int k)`, die für $k = 0$ und $k = 1$ die Werte 0 bzw. 1 zurückgibt, und sonst zweimal sich selbst für $k - 1$ und $k - 2$ aufruft und die Summe zurückgibt. Das ist möglich, aber wenn Sie das Programm auf diese Weise schreiben, werden Sie feststellen, dass es sehr langsam ist. Sie sollten sich überlegen, warum das so ist und wie Sie den Ansatz vielleicht trotzdem retten können. Das Problem hat nichts mit unserer Verwendung der `gmp` zu tun, Sie können das schon untersuchen, wenn Sie `int` als Datentyp nehmen und im Zahlbereich bleiben, den dieser Typ darstellen kann.

Im zweiten Programm wollen wir uns noch die Verwendung des von der gmp bereitgestellten Datentyps für *rationale Zahlen* ansehen. Das wollen wir anhand eines Programms für die Multiplikation zweier Matrizen aus $\mathbb{Q}^{m\times n}$ und $\mathbb{Q}^{n\times k}$ machen. Wie für die ganzen Zahlen finden wir in der Dokumentation der gmp die passenden Funktionen für den in der gmp definierten Datentyp `mpq_t` für rationale Zahlen.

```
1  // Eine rationale Zahl initialisieren
2  void mpq_init (mpq_t x);
3
4  // Den Speicher einer rationalen Zahl wieder freigeben
5  void mpq_clear (mpq_t x);
6
7  // eine rationale Zahl auf eine andere zuweisen
8  void mpq_set (mpq_t rop, const mpq_t op);
9
10 // eine Zahl vom Typ unsigend int in eine rationale Zahl zuweisen
11 void mpq_set_ui (mpq_t rop, unsigned long int op);
12
13 // eine als Zeichenkette der Form a/b gegebene Zahl zuweisen, base ist Basis des
       Zahlsystems, idR 10
14 int mpq_set_str (mpq_t rop, const char *str, int base);
15
16 // Zaehler und Nenner kuerzen, ggf. bei Eingabe notwendig
17 void mpq_canonicalize (mpq_t op);
18
19 // Addition
20 void mpq_add (mpq_t sum, const mpq_t addend1, const mpq_t addend2);
21
22 // Multiplikation
23 void mpq_mul (mpq_t product, const mpq_t multiplier, const mpq_t multiplicand);
24
25 // Ausgabe auf den Bildschirm oder in eine Datei
26 size_t mpq_out_str (FILE *stream, int base, const mpq_t op);
```

Wir sehen wieder die erwarteten Funktionen `mpq_init` und `mpq_clear` zur Speicherreservierung und -freigabe einer Zahl vom Typ `mpq_t`, ebenso wie einige Vertreter der Familie von Funktionen, die mit `mpq_set` beginnen und einer Zahl vom Typ `mpq_t` einen Wert zuweisen.

Bei der Eingabe von rationalen Zahlen müssen wir allerdings noch auf eine kleine Schwierigkeit achten. Die Darstellung einer rationalen Zahl ist nicht eindeutig, denn der Bruch $\frac{3}{6}$ und $\frac{1}{2}$ bezeichnen die gleiche Zahl, ebenso wie $\frac{-1}{2}$ und $\frac{1}{-2}$. Erst wenn wir zusätzlich verlangen, dass der Bruch vollständig gekürzt ist und der Nenner positiv ist, erhalten wir eine eindeutige Darstellung. Die Bibliothek `gmp` stellt in ihren Funktionen sicher, dass das Ergebnis in dieser Form ist. Bei der Eingabe wissen wir das allerdings nicht unbedingt. Daher gibt es die Funktion `mpq_canonicalize`, die einen Bruch kürzt. Man könnte das auch als Teil der `mpq_set`-Funktionen machen. Wenn wir als Programmierer*innen allerdings wissen, dass unsere Eingabe schon im korrekten Format ist, möchten wir gerne die Zeit zur Überprüfung sparen (immerhin muss dafür im Prinzip eine Primfaktorzerlegung bestimmt werden). Daher gibt es eine spezielle Funktion dafür, die wir aufrufen können, wenn wir von unserer Eingabe nicht sicher wissen, dass sie gekürzt ist.

Umgekehrt heißt das allerdings auch, dass die gmp in ihren Funktionen annimmt, dass Brüche gekürzt sind und dies an keiner Stelle überprüft. Wenn wir aus Versehen eine

Zahl vom Typ `mpq_t` nicht gekürzt initialisieren und dann vor der weiteren Verwendung der Zahl nicht `mpq_canonicalize` aufrufen, haben wir keine Garantie mehr für ein korrektes Ergebnis. Möglicherweise stürzt das Programm auch ab.

Nun wollen wir die Bibliothek an einer konkreten Aufgabe ausprobieren. Das folgende Programm liest zwei Matrizen aus zwei an das Programm übergebenen Dateien aus und schreibt das Produkt auf dem Bildschirm. Dabei haben wir die Funktionen für die Ein- und Ausgabe der Matrizen in einen Header ausgelagert, um das Hauptprogramm kürzer zu halten.

Programm 13.6: kapitel_13/matrix_io.h

```
#include <gmp.h>

#define MAX_N 10
#define MAX_M 10

void read_matrix(char* fname, mpq_t M[MAX_N][MAX_M], int* mx, int*my);

void print_matrix ( mpq_t M[MAX_N][MAX_M], int mx, int my);

void clear_matrix(mpq_t M[MAX_N][MAX_M], int mx, int my );

#endif
```

Programm 13.6: kapitel_13/matrix_io.c

```
#include "matrix_io.h"

void read_matrix(char* fname, mpq_t M[MAX_N][MAX_M], int* mx, int*my) {

    FILE *file;
    file = fopen(fname,"r");
    if ( file == NULL ) {
        printf("Datei nicht lesbar\n"); exit(1);
    }

    fscanf(file, "%d %d", mx, my);

    char c[] = " %s";
    char num[1000];

    for ( int i = 0; i < *my; i++) {
        for ( int j = 0; j < *my; j++) {
            fscanf(file,c,&num);
            mpq_init(M[i][j]);
            mpq_set_str(M[i][j],num,10);
        }
    }

    fclose(file);
}

void print_matrix ( mpq_t M[MAX_N][MAX_M], int mx, int my) {
    for ( int i = 0; i < mx; i++) {
        for ( int j = 0; j < my; j++) {
            mpq_out_str(stdout,10,M[i][j]);
            printf(" ");
```

```
32          }
33          printf("\n");
34      }
35      printf("\n");
36  }
37
38  void clear_matrix(mpq_t M[MAX_N][MAX_M], int mx, int my ) {
39      for ( int i = 0; i < mx; i++) {
40          for ( int j = 0; j < my; j++) {
41              mpq_clear(M[i][j]);
42          }
43      }
44  }
```

Programm 13.6: kapitel_13/matrix_multiplication.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #include <gmp.h>
5   #include "matrix_io.h"
6
7   void multiply (mpq_t R[MAX_N][MAX_M], int*  rx, int* ry, mpq_t M[MAX_N][MAX_M], mpq_t N
        [MAX_N][MAX_M], int mx, int my, int nx, int ny ) {
8
9       if ( my != nx ) {
10          printf("Matrix dimensions mismatch\n");
11          exit(1);
12      }
13
14      mpq_t q;
15      mpq_init(q);
16
17      for ( int i = 0; i < mx; ++i ) {
18          for ( int j = 0; j < ny; ++j ) {
19              mpq_init(R[i][j]);
20              mpq_set_str(R[i][j],"0",10);
21              for ( int a = 0; a < my; ++a ) {
22                  mpq_mul(q, M[i][a], N[a][j]);
23                  mpq_add(R[i][j], R[i][j], q);
24              }
25          }
26      }
27
28      mpq_clear(q);
29      *rx = mx;
30      *ry = ny;
31  }
32
33  int main(int argc, char** argv) {
34
35      int mx, my, nx, ny;
36      mpq_t M[MAX_N][MAX_M];
37      mpq_t N[MAX_N][MAX_M];
38
39      read_matrix(argv[1],M,&mx,&my);
40      read_matrix(argv[2],N,&nx,&ny);
41
```

```
42      print_matrix(M,mx,my);
43      print_matrix(N,nx,ny);
44
45      int rx, ry;
46      mpq_t R[MAX_N][MAX_M];
47      multiply(R,&rx,&ry,M,N,mx,my,nx,ny);
48      print_matrix(R,rx,ry);
49
50      clear_matrix(M,mx,my);
51      clear_matrix(N,nx,ny);
52      clear_matrix(R,rx,ry);
53
54      return 0;
55  }
```

Eine mögliche Eingabe könnte dann so aussehen.

```
\$ cat mat1.dat
4 4
1 3/4 3 5/2
3 3/7 4/5 2/13
1/2 4/3 2 17/4
3 4/5 7/2 2/7%


\$ cat mat2.dat
4 4
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

Diese Matrizen können wir nach Übersetzen des Programms als Argumente übergeben.

```
\$ gcc matrix_io.c matrix_multiplication.c -o matrix_multiplication -l gmp
\$ ./matrix_multiplication mat1.dat mat2.dat
1 3/4 3 5/2
3 3/7 4/5 2/13
1/2 4/3 2 17/4
3 4/5 7/2 2/7

1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1

1 3/4 3 5/2
3 3/7 4/5 2/13
1/2 4/3 2 17/4
3 4/5 7/2 2/7
```

Das Programm schreibt dabei die beiden Eingaben und das Produkt auf den Bildschirm. Beachten Sie, dass wir wieder mit der Option `-l` angeben mussten, dass die `gmp` eingebunden werden muss.

13.2.2 Matrizen und Speicher

In unserem Beispiel haben wir, um das Programm einfach zu halten, unsere Matrix wieder in einem einfachen zweidimensionalen Array gespeichert, und haben am Anfang des Programms über zwei Präprozessorvariablen eine maximale Größe für unsere Matrizen festgelegt. In praktischen Anwendungen wird man aus zwei Gründen anders vorgehen wollen.

Zum einen ist unser Ansatz bei großen Matrizen nicht mehr praktikabel. In der Numerischen Linearen Algebra sind Matrixdimensionen von mehreren tausend Zeilen und Spalten und noch deutlich größer durchaus üblich, ebenso eine deutliche Schwankung in der Größe. Mögliche Anwendungen wären hier z.B. Diskretisierungen für die Lösung von Differentialgleichungen, wobei lokal und je nach gewünschter Genauigkeit das Diskretisierungsgitter verfeinert werden muss und dadurch zu größeren Matrizen führt. Anwendungen dieser Art werden sie in der Vorlesung *Einführung in die Numerik* kennenlernen.

Ein weiterer Grund, warum wir Matrizen nicht in einem zweidimensionalen Array fester Größe speichern sollten liegt darin, dass die Matrizen, die wir typischerweise betrachten, oft weitere Struktur haben. Sie könnten z.B. *symmetrisch* sein, d.h., wenn $A = (a_{ij})_{1 \leq i,j \leq n}$ unsere Matrix ist, dann gilt $a_{ij} = a_{ji}$ für alle $1 \leq i, j \leq n$. Wir brauchen also nur die Hälfte aller Matrixeinträge zu speichern. Ebenso sind Matrizen, die in praktischen Problemen auftreten, oft *dünn besetzt*, d.h. $a_{ij} \neq 0$ nur für wenige Paare (i, j). Hier kann es sinnvoll sein, nur die von Null verschiedenen Einträge zu speichern. Schließlich kommt es auch oft vor, dass Matrizen aus praktischen Anwendungen eine *Blockstruktur* haben, also von der Form

$$M = \begin{bmatrix} \mathrm{P} & 0 \\ 0 & \mathrm{Q} \end{bmatrix}$$

für zwei Matrizen P und Q passender Dimension sind. Hier reicht es, P und Q zu speichern.

Für den allgemeinen Fall gibt es neben der oben verwendeten Form als zweidimensionale Liste

```
int n, m; // Zeilen- und Spaltenzahl
// n, m belegen
int mat[n][m];
```

auch die (eigentlich äquivalente) Definition als doppelten Zeiger in der Form

```
int n, m; // Zeilen- und Spaltenzahl
// n, m belegen
int ** mat;
mat = malloc(m * sizeof(int *));
for ( int i = 0; i < m; ++i ) {
  mat[i] = malloc(n * sizeof(int));
}
```

wobei Sie ggf. bei der Anforderung von Speicher anfangen sollten, ob Ihnen überhaupt Speicher vom System zurückgegeben wurde. Das könnte dann z.B. so aussehen:

```
// m, n: Zeilen- und Spaltenzahl
// die Funktion gibt NULL zurück, wenn nicht genug Speicher verfügbar ist
```

```
int ** allocate_mat (int m, int n) {
int ** mat;
mat = malloc(m * sizeof(int *));
if ( mat == NULL ) {
  return NULL;
}
for ( int i = 0; i < m; ++i ) {
  mat[i] = malloc(n * sizeof(int));
  if ( mat[i] = NULL ) {
    for ( int j = 0; j < i; ++j ) {
      free(mat[j]);
    }
    free(mat);
    return NULL;
  }
}
```

Wie wir gesehen haben, sind auch Listenvariablen eigentlich Zeiger, sodass sie auch auf eine mit

```
int mat[n][m];
```

definierte Matrix über `*(mat+i*n+j)` auf das Element an der Stelle (i, j) zugreifen können.

Eine häufig vorkommende Operation, die auf Matrizen ausgeführt werden muss, ist die Vertauschung von Zeilen, oder, allgemeiner, die *Permutation* der Zeilen. Einen Algorithmus, bei dem dies vorkommt, und den Sie schon kennen, ist die Gauß-Elimination zur Bestimmung der Lösung eines linearen Gleichungssystems. Hier müssen Sie in jedem Schritt eine Pivotzeile wählen und an die richtige Stelle tauschen. Diese Operation lässt sich in unserer Implementation einer Matrix über einen doppelten Zeiger sehr einfach realisieren, da hier nur zwei Zeiger vertauscht werden müssen. Wenn wir eine Matrix als zweidimensionale Liste vorliegen haben und häufig Zeilen vertauschen müssen, dann kann es effizient sein, statt eine echte Vertauschung der Zeilen vorzunehmen, eine zweite Liste

```
int perm[m];
```

mitzuführen, die die Reihenfolge der Zeilenindizes in der Matrix gegenüber der Ausgangsmatrix A, wenn wir permutieren. Auf den Index (i, j) greifen wir dann mit

```
e = a[perm[i]][j]
```

zu. Ebenso können wir natürlich dann eine Permutation der Spalten mitführen.

Wenn unsere Matrix A symmetrisch ist, dann reicht es, (etwas mehr als) die Hälfte der Einträge zu speichern. Eine solche Matrix können wir in der Form

$$A = L + D + L^T$$

schreiben, wobei D eine Diagonalmatrix ist, und in $L = (l_{ij})$ alle l_{ij} mit $i > j$ Null sind. Für eine solche Matrix kann man leicht z.B. die Definition über einen doppelten Zeiger wie oben so modifizieren, dass kein Speicherplatz verschenkt wird. Natürlich müssen Sie dann auch alle Funktionen für Matrixoperationen (wie Addition, Multiplikation etc.) so anpassen, dass sie mit der neuen Form der Speicherung zurechtkommen.

Bei dünn besetzten Matrizen hängt eine effiziente Struktur oft von der Anwendung ab, und davon, ob sich daraus noch weitere strukturelle Einschränkungen an die Form der Matrix ergeben. Wenn wir keine weitere Struktur kennen, dann nimmt man für dünn besetzte Matrizen A oft eine zeilenweise Speicherung, in der jede Zeile i Paare (j, a_{ij}) enthält, wobei wir nur solche Paare speichern, für die $a_{ij} \neq 0$.

In vielen Anwendungen kommen zum Beispiel sogenannte *Rang-1*-Matrizen vor, also quadratische Matrizen, deren Zeilen- oder Spaltenrang 1 ist (da Zeilen- und Spaltenrang gleich sind, ist es egal, welchen sie kontrollieren). Man kann sich überlegen, dass sich solche Matrizen A als Produkt von zwei Vektoren u und v in der Form

$$A \;=\; u \cdot v^t$$

schreiben lassen. Wir müssen also nur die beiden Vektoren u und v speichern, und kommen daher für eine n-dimensionale Rang-1-Matrix mit $2n$ Speicherplätzen aus. Auch das Produkt mit einem Vektor lässt sich hier effizient ausrechnen, denn es gilt

$$Ax \;=\; (u \cdot v^t)x \;=\; u(v^t \cdot x) \;=\; \langle v, x\rangle u\,,$$

das Produkt ist also ein skalares Vielfaches von u. Neben der Speicherung, die nur Platz linear in n beansprucht, haben wir hier also auch eine Multiplikation, die statt Cn^2 Operationen im allgemeinen Fall mit Cn Operationen, also einer in n linearen Anzahl, auskommt.

Eine weitere häufig auftretende Form von dünn besetzen Matrizen mit zusätzlicher Struktur sind die sogenannten *Bandmatrizen*. Eine Matrix $A = (a_{ij})_{1\leq i,j\leq n}$ ist eine *Bandmatrix*, wenn es $m_1, m_2 \in \mathbb{Z}_{\geq 0}$ gibt, so dass

$$a_{ij} \;=\; 0 \qquad \text{für} \qquad j \;<\; i - m_1 \quad \text{oder} \quad j \;>\; i + m_2$$

gilt für alle $1 \leq i \leq n$. Eine Bandmatrix lässt sich offensichtlich effizient mit einem passenden zweidimensionalen Array oder einem doppelten Zeiger speichern. Die Matrixoperationen müssen dann passend geschrieben werden.

Zum Abschluss betrachten wir ein einfaches Beispiel aus der numerischen linearen Algebra, in der solche Bandmatrizen vorkommen. Dafür betrachten wir eine zweimal stetig differenzierbare (uns unbekannte) Funktion

$$f \;:\; [0,1] \;\longrightarrow\; \mathbb{R}$$

von der wir wissen, dass $f(0) = f(1) = 0$ ist und wir kennen die Werte $f(x_i) = y_i$ an $N + 1$ *Stützstellen*

$$x_i \;=\; ih \qquad\qquad h \;=\; \frac{1}{N} \qquad\qquad 0 \;\leq i \;\leq\; n\,.$$

Wir möchten eine Approximation an die zweite Ableitung f'' von f berechnen. Dazu

können wir den Differenzenquotienten von f und f' betrachten:

$$\begin{aligned} z_i := f''(x_i) &\approx \frac{f'(x_{i+1}) - f'(x_i)}{h} \\ &\frac{1}{h}\left(\frac{f(x_{i+1}) - f(x_i)}{h} - \frac{f(x_i) - f(x_{i-1})}{h}\right) \\ &= \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} \end{aligned}$$

für $1 \leq i \leq N-1$. Damit erhalten wir unsere Approximationen z_i an den Stützstellen x_i als Matrixprodukt der Matrix A mit dem Vektor y, wobei

$$A := \frac{1}{h^2}\begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ldots & 0 \\ 0 & \cdots & 0 & 1 & -2 & 1 \\ 0 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix} \qquad y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-2} \\ y_{N-1} \end{bmatrix}.$$

Hier ist A eine Bandmatrix, bei der nur die Diagonale und die beiden Nebendiagonalen besetzt sind. Wir brauchen also nur $3n-2$ statt n^2 Speicherplätze (bzw. sogar nur $2n-1$, da die Matrix auch noch symmetrisch ist). Solche Bandmatrizen heißen auch *Tridiagonalmatrizen*.

Im nächsten Semester in der Fortsetzung zu dieser Vorlesung, in der *Einführung in die Numerik* und in Vorlesungen zur gewöhnlichen und partiellen Differentialgleichungen werden Ihnen solche und ähnliche Matrizen bei den Lösungsverfahren immer wieder begegnen. Dort werden Sie dann auch weitere Zerlegungen von solchen Matrizen und effiziente Algorithmen für Matrixoperationen kennenlernen.

Ein Punkt, auf den wir in diesem Abschnitt *nicht* eingegangen sind, der aber ein wesentliches Problem in der numerischen Behandlung von Matrizen im Computer darstellt, ist die Frage der *Rechengenauigkeit*, der *Fehlerfortpflanzung* und der *Fehlerkontrolle*. Wir haben oben unsere Matrizen mit Hilfe der exakten Arithmetik der `gmp` implementiert. In diesem Fall brauchen wir uns über Genauigkeit keine Gedanken machen, da wir immer *exakt* rechnen. In vielen Anwendungen können Sie aber nur mit den effizienteren Zahltypen `float` oder `double` rechnen, oder Ihre Ausgangsdaten sind von vornherein nur numerisch gegeben und mit Fehlern behaftet (z.B. weil sie aus Messungen stammen, die nur bis zu einer gewissen Genauigkeit möglich sind). In diesen Fällen kommt noch ein weiterer Aspekt zur Zerlegung und Speicherung einer Matrix sowie zur Implementierung der Matrixoperationen hinzu. Mathematisch äquivalente Formulierungen können sich numerisch sehr unterschiedlich verhalten. Insbesondere bei Produkten von Matrizen hat man oft das Problem, dass sich der Fehler in der Rechnung nur sehr schwer kontrollieren lässt und man sehr sorgfältig sein muss bei der Wahl der Methoden und ihrer Umsetzung. Auch hierzu lernen Sie mehr in den schon oben genannten Vorlesungen.

13.2.3 `sqlite3`

Als zweites Beispiel wollen wir uns ansehen, wie wir eine Datenbank anlegen und verwalten können. Dafür benutzen wir die Bibliothek `sqlite3`, die Methoden bereitstellt, mit denen wir eine einfache Datenbank erzeugen können und mit den Daten in strukturierter Weise interagieren können. Sie implementiert alle notwendigen Funktionen für eine Programmiersprache für Datenbankanfragen, die Sprache `SQL` (für *structured query language*).

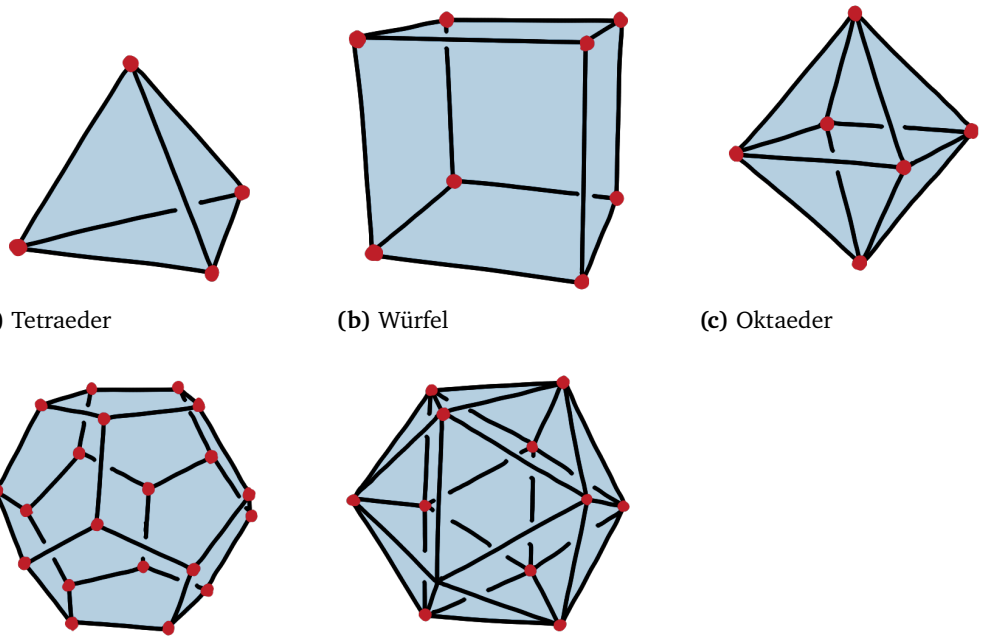
Die von `sqlite3` angelegte Datenbank ist sehr einfach aufgebaut und in einer auf dem System angelegten Datei enthalten. Es gibt erheblich mächtigere und schnellere Datenbanken, wie zum Beispiel `MariaDB`. Die Sprache `SQL` zum Anlegen von Tabellen in dieser Datenbank und für die Anfrage der Daten ist allerdings die gleiche, so dass wir, ohne viel an unserem Programm ändern zu müssen, eine andere zugrundeliegende Datenbank verwenden könnten, wenn der Umfang und die Anforderungen unserer Anwendung wachsen. Für uns liegt der Vorteil von `sqlite3` darin, dass wir keine spezielle weitere Software auf dem PC, auf dem wir unser Programm laufen lassen, installieren müssen, da alle Informationen der Datenbank in einer einzigen Datei abgelegt werden, deren Speicherort wir am Anfang angeben müssen. Diese Datei kann auch nur im Speicher liegen, und die Datenbank damit nur während des Programmlaufs existieren. Diesen Weg wählen wir im Beispiel unten.

`SQL` ist eine weitere Programmiersprache, die speziell für die Interaktion mit Datenbanken entwickelt wurde und dafür geeignete Methoden bereitstellt, wie zum Beispiel Suchanfragen und Filter auf solchen Anfragen, um nur eine Auswahl der Ergebnisse zu sehen oder diese in der gewünschten Reihenfolge zu sortieren. Da es hier nicht darum gehen soll, eine weitere Programmiersprache zu erlernen, werden wir uns in unseren Beispielen auf einige wenige, sehr einfache, Datenbankanfragen beschränken.

Als Anwendung wollen wir eine Datenbank erzeugen, die einige Informationen über die fünf *platonischen Körper*, also die fünf regulären dreidimensionalen Polytope, enthält. Die fünf Polytope sind das *Tetraeder*, das *Oktaeder*, der *Würfel*, das *Ikosaeder* und das *Dodekaeder*. Jedes dieser fünf Polytope ist die konvexe Hülle seiner Ecken, also der endlich vielen Punkte im Polytop, für die es eine affine Ebene gibt, deren Schnitt mit dem Polytop nur aus diesem Punkt besteht. Die fünf Polytope haben 4, 6, 8, 12 bzw. 20 Ecken. Zwischen (manchen Paaren von) Ecken haben wir dann Kanten (nämlich 6, 12, 12, 30 bzw. 30 Kanten in den einzelnen Polytopen), und der ganze Körper ist von endlich vielen Flächen (nämlich 4, 8, 6, 20 bzw. 12) begrenzt. In Abbildung 13.2 sind die fünf regulären Polytope abgebildet.

Unter allen weiteren Mengen im $\mathbb{R}^3$, die die konvexe Hülle endlich vieler Punkte sind, zeichnen sich diese fünf dadurch aus, dass sie genug Symmetrien haben, dass wir für je zwei Fahnen (das ist ein Tripel aus einer Ecke, Kante und Fläche, die ineinander enthalten sind), eine Symmetrie finden, die die eine Fahne auf die andere abbildet.

Wir wollen nun die Information über die Anzahlen von Ecken, Kanten und Flächen in einer Datenbank speichern und anschließend sortiert nach der Anzahl der Ecken wieder aus der Datenbank auslesen. Zudem wollen wir, in absteigender Sortierung, wissen, was die Differenz zwischen der Summe der Ecken und Flächen und den Kanten ist. Hier ist unser Beispielprogramm. Nach dem Programm gehen wir auf einige Teile genauer ein. Eine vollständige Dokumentation aller Methoden der Bibliothek `sqlite3` finden



(a) Tetraeder **(b)** Würfel **(c)** Oktaeder

(d) Dodekaeder **(e)** Dodekaeder

Abbildung 13.2: Die fünf regulären dreidimensionalen Polytope

Sie unter der Adresse sqlite.org/c3ref/intro.html.

Die Anweisungen in der Sprache SQL, die wir in dem Programm benutzen, sollten in ihrer Funktion beim Lesen einigermaßen verständlich sein, auch wenn Sie sie selbst nicht hätten schreiben können. Mit diesen Vorlagen sollten aber kleiner Modifikationen zur Anpassung in Ihren eigenen Programmen möglich sein. Auf die genaue Syntax können wir hier nicht eingehen, aber diese ist zum Beispiel auf der Seite sqlite.org/lang.html erklärt.

Programm 13.7: `kapitel_13/sqlite_example.c`

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

#include <sqlite3.h>


int callback_listing(void *data, int n_cols, char **col_values, char **col_names) {
    char* art = col_values[1];
    art[0] = toupper(art[0]);

    printf("%s %-10s hat %3d Ecken, %3d Kanten, and %3d Seiten\n",
            art,
            col_values[0], atoi(col_values[2]),  atoi(col_values[3]), atoi(col_values
    [4])
          );

    return 0;
}
```

```
19
20  int callback_euler(void *data, int n_cols, char **col_values, char **col_names) {
21      char* art = col_values[1];
22      art[0] = toupper(art[0]);
23
24      printf("%s %-10s hat Euler-Characteristik %2d\n",
25          art, col_values[0], atoi(col_values[2]));
26
27      return 0;
28  }
29
30  void cleanup( sqlite3 *db, char* sql_error, char* text ) {
31      fprintf(stderr, "%s\n", text);
32      fprintf(stderr, "SQL error: %s\n", sql_error);
33
34      sqlite3_free(sql_error);
35      sqlite3_close(db);
36  }
37
38  int main() {
39
40      // Die Objekte
41      char* names[5] = {"Tetraeder","Wuerfel","Oktaeder","Dodekaeder","Ikosaeder"} ;
42      char* art[5]   = {"das","der","das","das","das"} ;
43
44      // und ihre f-Vektoren,
45      // also die Tripel (v,e,f) von Ecken, Kanten und zweidimensionalen Seiten
46      int fv[5][3] = {
47          {  4,  6,  4} ,
48          {  8, 12,  6} ,
49          {  6, 12,  8} ,
50          { 20, 30, 12} ,
51          { 12, 30, 20}
52      } ;
53
54      // Der Zeiger auf unsere Datenbank
55      sqlite3 *db;
56      // und eine Variable, um Fehlermeldungen aufzunehmen
57      // da sqlite mit Betriebssystemfunktionen arbeitet,
58      // deren Erfolg wir nicht direkt kontrollieren können
59      // ist es wichtig, die Fehlermeldungen abzufragen und ggf. zu reagieren
60      char *error = 0;
61
62      // Datenbank oeffen oder erzeugen
63      int ret = sqlite3_open(":memory:", &db);
64      if (ret != SQLITE_OK) {
65          char msg[100];
66          sprintf(msg,"Datenbank konnte nicht geoeffnet werden: %s", sqlite3_errmsg(db));
67          cleanup(db, error, msg);
68          exit(1);
69      }
70
71      // Tabelle erzeugen
72      // hier mit sqlite3_exec
73      char *sql_table =
74          "CREATE TABLE regular(id INTEGER PRIMARY KEY, name TEXT, art TEXT, vertices
        INTEGER, edges INTEGER, facets INTEGER);";
75      ret = sqlite3_exec(db, sql_table, 0, 0, &error);
```

```
76      if (ret != SQLITE_OK ) {
77          cleanup(db, error, "Tabelle konnte nicht erzeugt werden" );
78          exit(1);
79      }  else {
80          printf("Tabelle erzeugt\n");
81      }
82
83      // Einlesestatement der Daten vorbereiten
84      // Da wir mehr als eine Zeile auf die gleiche Weise einfuegen wollen
85      // arbeiten wir hier mit prepare, step, finalize
86      // dann muessen prepare und finalize nur einmal ausgefuehrt werden
87      // waehrend wir step dazwischen oefter aufrufen koennen
88      char *sql_insert =
89          "INSERT INTO regular(name,art,vertices,edges,facets) VALUES (?, ?, ?, ?, ?)";
90      ret = sqlite3_prepare_v2(db, sql_insert, -1, &statement, 0);
91      if ( ret != SQLITE_OK ) {
92          cleanup(db, error, "SQL prepare fehlgeschlagen" );
93          exit(1);
94      }
95
96      // Der Reihe nach mit unserer vorbereiteten Einleseanweisung die Daten
97      // in die Tabelle schreiben
98      // Dafuer muessen wir mit den bind-Anweisungen die Parameter unseres
99      // SQL-Statements mit den konkreten Daten belegen
100
101     // Zeiger auf eine Variable, die eine vorbereitete SQL-Anweisung aufnimmt
102     sqlite3_stmt *statement;
103
104     for ( int i = 0; i < 5; ++i ) {
105         // Vorbereitung der Datenbankoperation
106         // Belegung der einzelnen Variablen
107         sqlite3_bind_text(statement, 1, names[i], -1, NULL);
108         sqlite3_bind_text(statement, 2, art[i],   -1, NULL);
109         sqlite3_bind_int(statement,  3, fv[i][0]);
110         sqlite3_bind_int(statement,  4, fv[i][1]);
111         sqlite3_bind_int(statement,  5, fv[i][2]);
112
113         // schreiben
114         sqlite3_step(statement);
115         int ret = sqlite3_reset(statement);
116         if ( ret != SQLITE_OK ) {
117             cleanup(db, error, "Daten konnten nicht eingefuegt werden" );
118             exit(1);
119         }
120     }
121
122     // Cache leeren, Anweisung loeschen
123     sqlite3_finalize(statement);
124     printf("Alle Daten eingefuegt\n");
125
126     // Daten wieder auslesen, zuerst die f-Vektoren
127     // Hier wieder mit sqlite3_exec, dem wir einen Zeiger auf callback_listing
        übergeben,
128     // was die Daten auf den Bildschirm schreibt
129     printf("f-Vektoren:\n");
130     char *sql_query =
131         "SELECT name, art, vertices, edges, facets FROM regular ORDER BY vertices";
132     ret = sqlite3_exec(db, sql_query, callback_listing, 0, &error);
```

```
133     if ( ret != SQLITE_OK ) {
134         cleanup(db, error, "Daten konnten nicht gelesen werden" );
135         exit(1);
136     }
137     printf("----\n");
138 
139     // und dann die Euler-Charakteristik
140     printf("Euler-Characteristik:\n");
141     sql_query =
142         "SELECT name, art, vertices-edges+facets-1 as e FROM regular ORDER BY e";
143     ret = sqlite3_exec(db, sql_query, callback_euler, 0, &error);
144     if ( ret != SQLITE_OK ) {
145         cleanup(db, error, "Daten konnten nicht gelesen werden" );
146         exit(1);
147     }
148 
149     // Datenbank schliessen
150     // Das ist insbesondere wichtig, wenn die Datenbank in eine Datei geschrieben wird
151     // und auch nach Programmende vorhanden sein soll
152     // ggf. werden hier noch bisher nur zwischengespeicherte Anweisungen ausgefuehrt!
153     sqlite3_close(db);
154 
155     return 0;
156 }
```

Die Bibliothek `sqlite3.h` stellt alle notwendigen Typen und Funktionen bereit. Der Zugriff auf eine Datenbank wird über den zentralen Typ `sqlite3` ermöglicht. Wir brauchen in unserem Programm einen Zeiger auf eine Variable von diesem Typ. Die Funktion `sqlite3_open` reserviert dann den notwendigen Speicher und initialisiert alle relevanten Daten in dem Datentyp. Alle Funktionen der Bibliothek `sqlite3` geben eine Statusinformation zurück. Da wir hier mit Daten außerhalb unseres Programms arbeiten, meistens Dateisystemzugriffe notwendig sind und mit der `SQL` eine weitere Programmiersprache beteiligt ist, sollten wir diese Rückmeldung immer überprüfen, da hier viele Fehlerquellen möglich sind, auf die wir mit unserem Programm keinen Einfluss haben (wie zum Beispiel zu wenig Speicherplatz für die Datenbank auf der Festplatte, oder fehlende Leserechte auf der Datei, in der die Datenbank liegt).

Die Funktion `sqlite3_open` nimmt zwei Parameter, den Speicherort der Datenbank und unseren Zugriffszeiger. Mit `:memory:` können wir eine Datenbank in den Hauptspeicher des PC legen. Bevor wir das Programm beenden, müssen wir mit `sqlite3_close` die Datenbank wieder schließen.

Wir haben danach zwei Möglichkeiten, über `SQL`-Befehle mit der Datenbank zu kommunizieren. Die einfache Variante ist die Funktion `sqlite3_exec`, die eine Anweisung in `SQL` auf der Datenbank ausführt und das Ergebnis zurückliefert. Da diese Funktion mit allen Ein- und Ausgaben der Datenbank zurechtkommen soll, und wir daher keine Annahmen über die Typen der Aus- und Eingabe treffen können, arbeitet diese Funktion, wie wir das schon bei den Funktionszeigern für unsere Sortierfunktionen gesehen haben, mit Zeigern vom Typ `void *`, die wir dann in unserem Programm geeignet umwandeln müssen.

Wenn wir eine Anfrage an eine Datenbank stellen, kann es auch sein, dass wir mehr als einen Datensatz als Ergebnis bekommen (zum Beispiel wenn wir eine Datenbank aller Studierenden dieser Vorlesung haben und die Datensätze aller Studierenden

erhalten wollen, die die Studienleistung bestanden haben). In der Regel wollen wir mit jedem Datensatz dann auch etwas machen, zum Beispiel auf dem Bildschirm ausgeben. Bei kleinen Datenbanken könnte es noch akzeptabel sein, alle Datensätze als Liste zurückzugeben, aber bei größeren Datenbanken könnte ein solches Vorgehen zu viel Speicher verbrauchen, und es könnte lange dauern, bis alle Daten aus der Datenbank ausgelesen und in einen Datentyp in `C` konvertiert sind. Daher möchte man in der Regel immer nur einen Datensatz lesen und verarbeiten, bevor der nächste ausgelesen wird. Hierfür gibt es zwei Optionen, das über eine Datenbankschnittstelle zu ermöglichen. In `sqlite3` nimmt die Funktion `sqlite3_exec` dafür einen Funktionszeiger als Argument an, dem wir eine Funktion übergeben können, die dann auf jedem Ergebnis ausgeführt wird[8].

Damit sieht die Deklaration von `sqlite3_exec` so aus:

```
int sqlite3_exec(
sqlite3*,
const char *sql,
int (*callback)(void*,int,char**,char**),
void *,
char **errmsg
);
```

Dabei wird im ersten Argument der Zeiger auf die Datenbank und im zweiten die Datenbankabfrage übergeben. Das dritte Argument ist die Deklaration des Funktionszeigers, den wir übergeben können. Dabei erhalten wir im ersten Argument dieser Funktion den Zeiger `void *`, der als viertes Argument an `sqlite3_exec` übergeben wird (damit haben wir die Möglichkeit, Daten an die Funktion zu übergeben, die nicht aus der Datenbank kommen), im zweiten bekommen wir die Anzahl der Einträge im Ergebnis, im dritten die Werte, und im vierten die Spaltentitel der Tabelle. Im letzten Argument von `sqlite3_exec` übergeben wir die Adresse einer Zeichenkette, in die die Bibliothek `sqlite3` eine Fehlermeldung schreiben kann, wenn die Datenbankabfrage fehlschlägt. Für diese Zeichenkette müssen wir eine Variable vom Typ `char*` definieren, und `sqlite3` reserviert bei Bedarf Speicher dafür. Wir müssen daher am Ende des Programms daran denken, diesen Speicher wieder mit `sqlite3_free` wieder freizugeben.

Die Ergebnisse der Datenbankabfrage bekommen wir immer als Liste von Zeichenketten (ähnlich wie die Argumente der Kommandozeile über `argv` in `main`) und müssen selbst geeignet konvertieren.

Wenn wir keine Daten von unserer Datenbankabfrage als Rückgabe erwarten (weil wir zum Beispiel Daten schreiben statt welche zu lesen), dann können wir statt einer Funktion auch 0 übergeben. Das gleiche gilt, wenn wir keine eigenen Daten über das vierte Argument an die Funktion übergeben wollen. Im Beispiel unten sehen sie das bei den Anweisungen zum Anlegen der Datenbanktabelle und beim Schreiben der Daten.

Die Anweisung `sqlite3_exec` ist eigentlich nur eine Zusammenfassung von drei Einzelfunktionen `sqlite3_prepare_v2`, `sqlite3_step` und `sqlite3_finialize`. Die erste dieser drei Funktionen initialisiert die Abfrage, mit der zweiten können wir die Datenbankabfrage immer wieder mit neuen Daten ausführen, und die dritte löscht die Abfrage

[8] Die andere Variante, dieses Problem zu lösen, besteht darin, das Auslesen auf zwei Schritte aufzuteilen, und mit dem ersten die Abfrage auszulösen und ein erstes Ergebnis zurückzugeben, und mit jedem Aufruf ein weiteres Ergebnis zu holen (wie es zum Beispiel `strtok` macht).

wieder[9]. Die Funktionsweise der drei Funktionen können Sie sich im Beispiel ansehen, wo wir auf diese Weise der Reihe nach die Informationen zu unseren platonischen Körpern in die Datenbank schreiben.

Die verwendeten Funktionen stellen natürlich nur einen kleinen Ausschnitt der für eine effektive Interaktion mit einer Datenbank notwendigen Funktionen dar. Wir können beim Schreiben und Lesen der Daten wesentlich mehr Informationen weitergeben und so den Prozess steuern und möglicherweise effizienter gestalten. Ebenso haben wir keine Funktionen verwendet, die die Struktur der Tabelle modifizieren (zum Beispiel neue Spalten anfügen), oder die Einträge oder gleich die ganze Tabelle löschen. Wir können auch Informationen aus mehreren Tabellen nach von uns vorgegeben Kriterien verbinden (um zum Beispiel aus einer Tabelle mit Namen und Geburtstagen und einer zweiten mit Namen und Adressen die Adressen aller Personen zu finden, die heute Geburtstag haben). Manche solcher Aktionen müssen in `SQL` formuliert werden, und für manche stellt `sqlite3` Methoden bereit.

13.3 Weitere Beispiele

Es gibt viele weitere nützliche Bibliotheken, die in `C` geschrieben sind oder eine passende Schnittstelle bieten, und die wir in unseren Programmen einbinden können. Wir wollen einige kurz vorstellen.

Aus dem Bereich der Mathematik gibt es zum Beispiel noch die folgenden Bibliotheken, die in vielen Anwendungen nützlich sind.

- Die `MPFR` (www.mpfr.org) ist eine Bibliothek für exakte Rechnungen mit Dezimalzahlen und korrekter Rundung.
- Die Bibliothek `BLAS` (www.netlib.org/blas/) bietet viele Funktionen aus dem Bereich der Linearen Algebra.
- `GSL: Scientific Library` (www.gnu.org/software/gsl/) ist eine Sammlung vieler Methoden aus dem naturwissenschaftlichen Bereich.
- `SCIP` (scipopt.org) ist eine Sammlung effizienter Methoden für lineare und ganzzahlige Optimierung.

Es gibt natürlich auch aus anderen Anwendungsbereichen schon fertige und gut gepflegte Bibliotheken. Wir wollen nur zwei Bibliotheken rausgreifen, mit denen wir eine graphische Ausgabe auf dem Bildschirm machen können.

- Mit `ncurses` (www.gnu.org/software/ncurses/ncurses.html) können wir den Cursor frei in einem Terminalfenster bewegen und auf diese Weise Stellen wieder überschreiben oder einfache zeichenbasierte Grafik im Terminalfenster ausgeben. Viele ältere Computerspiele benutzen diese Bibliothek, aber auch Anwendungen wir `htop` oder `mc`.
- Wesentlich ausgefeilter ist `cairo` (www.cairographics.org), mit der wir eigene

[9] Eigentlich macht diese letzte Funktion noch mehr. Da Datenbankzugriffe im Vergleich zum Programmlauf langsam sein können, werden die Anfragen aus `sqlite3_step` möglicherweise nicht sofort ausgeführt, sondern zwischengespeichert und erst später, wenn mehrere Anfragen zusammengekommen sind, ausgeführt. `sqlite3_finalize` sorgt dafür, dass anschließend alle Anfragen auch wirklich ausgeführt sind.

Fenster auf dem Bildschirm erzeugen können und eine eigene graphische Oberfläche für unsere Programme entwerfen können. Da *cairo* auf einer weiteren Bibliothek beruht, die es für alle Betriebssysteme gibt, ist das sogar unabhängig vom verwendeten Betriebssystem. Allerdings ist dann auch die Anwendung entsprechend komplizierter, da wir dann auch alle Aktionen in unseren Fenstern festlegen müssen (Klicken von Menüs und Schaltflächen, Eingaben, Ausgaben, Mausaktionen, ...).

Version vom 10. Februar 2022

14 Fehlersuche

Außer bei sehr einfachen Programmen ist es unwahrscheinlich, dass Ihr Code von Anfang an fehlerfrei ist. Und selbst wenn es auf den ersten Blick so aussieht als ob ein Programm zu allen Eingaben ein korrektes Ergebnis zurückliefert, fällt später auf, dass seltenere oder unwahrscheinlichere Eingaben nicht bedacht wurden und das Programm dann nicht wie erwartet funktioniert.

Das sehen sie auch bei professionell entwickelter Software, die sie auf Ihrem Computer installiert haben, und für die regelmäßig ein Update angeboten wird. In manchen Fällen dient das zwar auch dazu, neue Funktionen in ein Programm einzubauen, aber meistens werden vor allem zwischenzeitlich bekanntgewordene Fehler korrigiert.

Da wir Fehler erfahrungsgemäß nicht vermeiden können, ist es daher sinnvoll, sich gute Strategien zu überlegen, wie man sein Programm testen kann, und wie man Fehler im Code lokalisieren kann. Möglichkeiten, geeignete Tests für unsere Programme zu schreiben, haben wir uns schon in Kapitel 7 angesehen. Wir wollen uns hier mit der zweiten Frage beschäftigen und dazu zwei Strategien diskutieren.

Die erste und sehr einfache Möglichkeit, den Programmablauf zu verfolgen ist es, sich mit `printf` Statusausgaben an Stellen, die wir im Verdacht haben, falsch zu sein, auf den Bildschirm schreiben zu lassen. Diese Methode haben Sie wahrscheinlich schon selbst gefunden, oder Ihr Tutor hat Sie darauf hingewiesen.

Die zweite Variante verwendet ein spezielles Programm, einen sogenannten *debugger*, dessen einzige Aufgabe es ist, den Ablauf eines Programms transparent zu machen, in dem man an allen Stellen des Programm anhalten und sich die Belegung von Variablen ansehen kann. Es gibt eine Reihe solcher Programme, von denen wir eines, nämlich das Programm `gdb`, ansehen werden.

14.1 Bildschirmausgaben

Bevor man eine Fehlersuche mit einem Debugger versucht, ist es meistens sinnvoll, erst zu versuchen, den Fehler über Bildschirmausgaben, die Funktionsaufrufe oder Variablenbelegung ausgeben und die man mit dem erwarteten Programmablauf oder Belegung vergleicht. Das könnte zum Beispiel wie in dem folgenden Programm aussehen.

Programm 14.1: `kapitel_14/debugging_printf_01.c`

```
1 #include <stdio.h>
2
3 void func_A(int p) {
4   printf("[func_A] wurde mit Parameter %d aufgerufen\n",p);
5 }
6
```

```c
void func_B(int * p) {
  printf("[func_B] wurde mit Zeiger %p auf Parameter %d aufgerufen\n",p,*p);
  func_A(*p);
}

int main() {

  int s = 0;
  for ( int i = 0; i < 10; ++i ) {
    s += i;
    printf("[main] Iteration %d der Schleife, s ist %d\n",i,s);
    func_B(&s);
  }

  return 0;
}
```

In dem Programm lassen wir uns die Funktionsaufrufe und die Parameterbelegung sowie die Schleifendurchläufe und die in der Schleife veränderten Variablen ausgeben.

Manchmal kann es auch interessant sein zu wissen, in welchem Aufruf einer Funktion das Programm ist, oder wie oft eine Funktion aufgerufen wird. Hierfür bieten sich Variablen vom Typ `static` an.

```c
  void func(int p) {
    static int counter c = 1;
    printf("[func] wurde im %d-ten Aufruf mit Parameter %d aufgerufen\n",counter++,p);
  }
```

Während der Entwicklung eines Programms schreibt man solche Statusmeldungen oft direkt von Anfang an in den Code, um den Ablauf und die korrekte Funktionsweise von Anfang an immer wieder kontrollieren und nachvollziehen zu können. Trotzdem möchte man vielleicht zwischendrin eine erste Version des Programms veröffentlichen oder an Kollegen weitergeben. Diese Version sollte natürlich besser ohne unsere Statusmeldungen laufen. Dafür könnte man sie vorher alle aus dem Code entfernen und kanschließend wieder einfügen. Leichter wird es, wenn wir alle Statusmeldungen über Präprozessoranweisungen aus dem Code entfernen bzw. aufnehmen können. Das könnte wie folgt aussehen:

Programm 14.2: `kapitel_14/debugging_printf_02.c`

```c
#include <stdio.h>

void func_A(int p) {
  #ifdef DEBUG_OUTPUT
  printf("[func_A] wurde mit Parameter %d aufgerufen\n",p);
  #endif
}

void func_B(int * p) {
  #ifdef DEBUG_OUTPUT
  printf("[func_B] wurde mit Zeiger %p auf Parameter %d aufgerufen\n",p,*p);
  #endif
  func_A(*p);
}

int main() {
```

```
17
18    int s = 0;
19    for ( int i = 0; i < 10; ++i ) {
20      s += i;
21      #ifdef DEBUG_OUTPUT
22      printf("[main] Iteration %d der Schleife, s ist %d\n",i,s);
23      #endif
24      func_B(&s);
25    }
26
27    return 0;
28  }
```

Wenn wir dieses Programm in dieser Form übersetzen, werden keine Meldungen auf dem Bildschirm ausgegeben. Wenn wir hingegen am Anfang die Zeile

```
1   #define DEBUG_OUTPUT
```

in des Programm einfügen und wieder übersetzen, werden alle Meldungen ausgegeben. Wir können diese Präprozessorvariable auch als Option an den Compiler übergeben. Dafür brauchen wir die Option `-D`, hinter der die zu definierende Variable angehängt wird.

```
$ gcc debugging_printf_02.c -DDEBUG_OUTPUT -o debugging_printf_02
```

Diese Option setzt die Definition von `DEBUG_OUTPUT` an den Anfang jeder zu übersetzenden Datei. Sie können auch ein Leerzeichen nach `-D` machen, die Zusammenschreibung sieht man häufiger.

14.2 Der Debugger `gdb`

Bei längeren Programmen, oder wenn wir die Informationen, die wir kontrollieren wollen, nicht mehr vernünftig mit **`printf`** ausgeben können,. müssen wir unsere Fehlersuche auf andere Weise fortsetzen. Hierfür bieten sich *Debugger* an, die in der Lage sind, ein Programm im Prinzip Zeile für Zeile ablaufen zu lassen, und an jeder Stelle den genauen Zustand aller Variablen, Register und Stacks auszugeben. Das bei der Sprache `C` dafür am häufigsten verwendete Programm ist `gdb`[1], oft in Kombination mit einer graphischen Oberfläche, die die Ausgaben von `gdb` visuell darstellen kann. Wir wollen uns zunächst das Programm `gdb` selbst ansehen, bevor wir auf eine sehr einfache Visualisierung seiner Ausgaben eingehen.

Der Präprozessor fügt vor dem Übersetzen den Code aus allen an den Compiler übergebenen Dateien zu einem einzigen Programm zusammen, und der Compiler darf, und wird, anschließend beim Übersetzen des Codes in Maschinensprache den Codeablauf verändern um das Programm gut an die vom Prozessor verstandenen Befehle anzupassen. Der Prozessor ist, anders als wir (in den meisten Fällen), nicht darauf angewiesen, dass Variablen und Funktionen sinnvoll, also in der Regel entsprechend ihrer Verwendung, benannt sind, um und das Zurechtfinden im Programm zu erleichtern. Alle Namen werden intern einfach durch die Adressen ersetzt. Daher ist es nicht ohne weiteres möglich, von einer Stelle im Ablauf des Programms in Maschinensprache darauf

[1] www.gnu.org/software/gdb/

zurückzuschließen, welcher Stelle im C-Code diese entspricht. Um diese Zuordnung doch hinzubekommen, sehen Compiler die Möglichkeit vor, diese Referenzinformation (sogenannte *debugging symbols*) beim Übersetzen des Programms in den Maschinencode zu schreiben. Bei `gcc` dient dazu die Option `-g`. Natürlich ist es nicht sinnvoll, diese Kommentare im Maschinencode auch in der veröffentlichten Version einzufügen, da das Programm dadurch (erheblich) mehr Speicherplatz verbraucht. Betrachten wir dazu noch einmal das Programm `collatz` aus Kapitel 2.

Programm 14.3: `kapitel_02/collatz.c`

```
1  // Standardbibliotheken
2  #include <stdio.h>
3
4  /* Funktion zur Bestimmung der Laenge der Collatzfolge zu einem Startwert
5     Eingabe: startwert als ganze Zahl
6     Ausgabe: Laenge der Folge
7     Annahmen: - startwert ist positiv, kein Check
8               - Folge erreicht nach endlich vielen Schritten die 1
9   */
10 int collatz_laenge(int startwert) {
11   int c = startwert;          // Initialisierung
12   int laenge = 0;
13   while ( c != 1 ) {          // naechstes Folgenglied, falls 1 nicht erreicht
14     if ( c % 2 == 0 ) {
15       c = c/2;
16     }  else {
17       c = 3*c+1;
18     }
19     laenge = laenge + 1;      // laenge aktualisieren
20   }
21   return laenge;
22 }
23
24 // main: Einstiegspunkt in das Programm
25 int main() {
26   int startwert = 0;          // Eingabevariable initialisieren
27   printf("Geben Sie eine positive ganze Zahl ein: ");
28   scanf("%d",&startwert);     // Startwert vom Terminal einlesen
29
30   // Aufruf der Funktion zur Bestimmung der Laenge
31   int laenge = collatz_laenge(startwert);
32
33   // Ergebnis ausgeben
34   printf("Die Collatz-Folge mit Start %d hat Laenge %d\n",startwert,laenge);
35
36   // Programm ohne Fehlermeldung beenden
37   return 0;
38 }
```

Wir übersetzen es mit

```
$ gcc collatz.c -g -o collatz
```

Jetzt können wir den Debugger starten und darin das Programm laden.

```
$ gdb ./collatz
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
```

```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from collatz...done.
>>>(gdb)
```

Wir erhalten mit

```
>>>(gdb)
```

die Eingabeaufforderung des Debuggers (je nach System kann diese auch etwas anders aussehen). Mit einfachen Steuerbefehlen können wir nun das Programm starten, zeilenweise ablaufen lassen, einzelne Zeilen markieren und das Programm an diesen Zeilen anhalten lassen, und uns an allen Stellen die Variablenbelegung ansehen. Es gibt eine ganze Reihe von Steuerbefehlen für `gdb`. Für den Anfang sind die wichtigsten

- `break`, um eine Markierung (*breakpoint*) zu setzen
- `run` zum Starten des Programms,
- `continue`, um den Code bis zur nächsten von uns gesetzten Markierung oder zum Programmende laufen zu lassen,
- `next`, um eine Anweisung auszuführen und zur nächsten zu springen,
- `list`, um sich die aktuelle Zeile zusammen mit einigen Zeilen davor und danach anzeigen zu lassen,
- `print`, um sich den Wert einer Variablen anzeigen zu lassen.

Wir müssen vor dem Start des Programms mit `run` mindestens eine Markierung setzen, sonst läuft das Programm einfach bis zum Ende durch (oder, wenn es fehlerhaft ist, evtl. auch nur bis zum Absturz). Bei allen Anweisungen an den Debugger müssen Sie vom Namen der Anweisung so viel schreiben, dass diese eindeutig ist. Bei allen Anweisungen oben reicht dafür schon der erste Buchstabe.

Damit könnte ein Anfang eines Programmdurchlaufs mit dem Debugger zum Beispiel so aussehen.

```
(gdb) break 34
Breakpoint 1 at 0x400636: file collatz.c, line 34.
(gdb) run
Starting program: /home/paffenholz/temp/collatz
Geben Sie eine positive ganze Zahl ein: 24

Breakpoint 1, collatz_laenge (startwert=24) at collatz.c:34
34      laenge = laenge + 1;     // laenge aktualisieren
(gdb) list
29      if ( c % 2 == 0 ) {
30        c = c/2;
31      } else {
32        c = 3*c+1;
```

```
33      }
34      laenge = laenge + 1;     // laenge aktualisieren
35    }
36    return laenge;
37  }
38
(gdb) print laenge
$1 = 0
(gdb) continue
Continuing.

Breakpoint 1, collatz_laenge (startwert=24) at collatz.c:34
34      laenge = laenge + 1;     // laenge aktualisieren
(gdb) print laenge
$2 = 1
(gdb) continue
Continuing.

Breakpoint 1, collatz_laenge (startwert=24) at collatz.c:34
34      laenge = laenge + 1;     // laenge aktualisieren
(gdb) print laenge
$3 = 2
(gdb) next
28    while ( c != 1 ) {         // naechstes Folgenglied, falls 1 nicht erreicht
(gdb) print c
$4 = 3
(gdb) next
29      if ( c % 2 == 0 ) {
(gdb) next
32        c = 3*c+1;
(gdb) print c
$5 = 3
(gdb) next

Breakpoint 1, collatz_laenge (startwert=24) at collatz.c:34
34      laenge = laenge + 1;     // laenge aktualisieren
(gdb) print c
$6 = 10
(gdb) print laenge
$7 = 3
(gdb)
```

Hier haben wir eine Markierung auf die Zeile 34 des Programms gesetzt, und dann den Programmablauf gestartet. Wie bei einem normalen Programmlauf werden wir nach der Eingabe einer Zahl gefragt. Wir geben 24 ein. Danach läuft das Programm weiter, bis die Markierung erreicht ist. Wir lassen uns mit `list` den Kontext ausgeben und mit `print laenge` den Wert der Variablen `laenge`. `gdb` numeriert die abgerufenen Variablen durch, damit wir sie später verwenden könnten. Das geht mit `$` und einer fortlaufenden Nummer.

Wir wiederholen die Fortsetzung des Programms bis zur Markierung noch zweimal und schauen uns den Wert von `laenge` an. Danach lassen wir ein paar Schritte mit `next` nacheinander ausführen, und sehen uns den Wert von `c` an. Mit `continue` setzen wir das Programm wieder bis zur Markierung fort.

Die Anweisung `print` nimmt auch komplexe Ausdrücke in `C` an und führt die entsprechende Berechnung aus (soweit das möglich ist, also zum Beispiel Funktionen definiert

sind).

```
(gdb) print 2*c+1
$2 = 25
```

Wenn der Typ der Rückgabe an dieser Stelle allerdings nicht eindeutig aus dem Ausdruck geschlossen werden kann, müssen Sie das Ergebnis mit einem *cast* umwandeln. Wir können auch in die Variablenbelegung eingreifen und mit `set var` den Wert verändern. Zum Beispiel setzt

```
(gdb) set var c=2
```

Die Variable `c` auf 2. Wir können die Änderung überprüfen und unseren Durchlauf fortsetzen.

```
(gdb) print c
$8 = 2
(gdb) c
Continuing.

Breakpoint 1, collatz_laenge (startwert=24) at collatz.c:34
34      laenge = laenge + 1;     // laenge aktualisieren
(gdb) print c
$9 = 1
(gdb) c
Continuing.
Die Collatz-Folge mit Start 24 hat Laenge 5
[Inferior 1 (process 10390) exited normally]
```

Das Ergebnis ist in diesem Fall durch unsere Änderung natürlich (im mathematischen Sinn) falsch. Sie können diese Methode allerdings benutzen, um zum Beispiel Variablen, die nicht enthalten, was Sie erwarten, zu korrigieren, um zu testen, ob ab diesem Punkt dann Ihr Programm korrekt weiterläuft.

Wenn Sie noch einmal auf den Anfang des Beispiels zurückschauen, dann sehen Sie, dass unsere Markierung eine Nummer bekommen hat, die 1. Alle Markierungen, die wir setzen, werden fortlaufend durchnummeriert, und wir können Sie über diese Nummer identifizieren. Insbesondere können wir sie mit `delete` wieder löschen, oder mit `disable` und `enable` deaktivieren und wieder aktivieren. Bei einer Markierung können wir statt einer Zeile auch den Namen einer Funktion angeben, und wenn mehrere Dateien in unserem Projekt sind, müssen wir den Dateinamen vor die Zeilennummer schreiben, getrennt durch einen Doppelpunkt.

Statt einer Stelle im Code können wir mit einem Beobachter (*watch point*) den Programmlauf auch an allen Stellen anhalten, an denen sich der Speicher an einer Stelle ändert (überlicherweise also der Wert einer Variablen). Das geht mit der Anweisung `watch`. Der Debugger gibt, wie Ihnen vielleicht schon oben aufgefallen ist, immer die Codezeile aus, an der er den Programmlauf angehalten hat. Bei den Beobachtern ist das, wenn auf die beobachtete Variable zugewiesen wird, oft die Zeile *nach* der Zeile mit der Zuweisung, da erst am Ende der Anweisung die Speicherzelle verändert wird. Wir sehen das im folgenden Beispiel.

```
(gdb) b 27
Breakpoint 1 at 0x400603: file collatz.c, line 27.
(gdb) r
Starting program: /home/paffenholz/temp/collatz
```

```
Geben Sie eine positive ganze Zahl ein: 34

Breakpoint 1, collatz_laenge (startwert=34) at collatz.c:27
27    int laenge = 0;
(gdb) watch c
Hardware watchpoint 2: c
(gdb) c
Continuing.

Hardware watchpoint 2: c

Old value = 34
New value = 17
0x0000000000400625 in collatz_laenge (startwert=34) at collatz.c:30
30        c = c/2;
(gdb) c
Continuing.

Hardware watchpoint 2: c

Old value = 17
New value = 52
collatz_laenge (startwert=34) at collatz.c:34
34      laenge = laenge + 1;
(gdb) c
Continuing.

Hardware watchpoint 2: c

Old value = 52
New value = 26
0x0000000000400625 in collatz_laenge (startwert=34) at collatz.c:30
30        c = c/2;
(gdb)
```

Wir können einer Markierung oder einem Beobachter auch eine Bedingung mitgeben, die erfüllt sein muss, damit er auslöst. Das kann im wesentlichen jeder Ausdruck in `C` sein, der zu wahr oder falsch auswertet.

```
(gdb) break 29 if c % 5 == 0
Breakpoint 1 at 0x40060c: file collatz.c, line 29.
(gdb) run
Starting program: /home/paffenholz/temp/collatz
Geben Sie eine positive ganze Zahl ein: 12

Breakpoint 1, collatz_laenge (startwert=12) at collatz.c:29
29      if ( c % 2 == 0 ) {
(gdb) print c
$1 = 10
(gdb) c
Continuing.

Breakpoint 1, collatz_laenge (startwert=12) at collatz.c:29
29      if ( c % 2 == 0 ) {
(gdb) print c
$2 = 5
(gdb) c
Continuing.
```

```
Die Collatz-Folge mit Start 12 hat Laenge 9
[Inferior 1 (process 31871) exited normally]
(gdb)
```

Hier sehen wir, dass nach $c = 5$ der Programmablauf nicht mehr an der Markierung anhält, da danach die Folge mit $16, 8, 4, 2, 1$ zu Ende geht. Mit `condition` können wir die Bedingung einer Markierung verändern.

Mit `next` springen wir zur nächsten Anweisung. Falls das ein Funktionsaufruf ist, wird dabei die Funktion vollständig abgearbeitet und an der nächsten Anweisung in der Funktion, in der wir uns befinden, angehalten.

```
(gdb) break 44
Breakpoint 1 at 0x400688: file collatz.c, line 44.
(gdb) run
Starting program: /home/paffenholz/temp/collatz
Geben Sie eine positive ganze Zahl ein: 12

Breakpoint 1, main () at collatz.c:46
46    int laenge = collatz_laenge(startwert);
(gdb) next
49    printf("Die Collatz-Folge mit Start %d hat Laenge %d\n",startwert,laenge);
(gdb) next
Die Collatz-Folge mit Start 12 hat Laenge 9
52    return 0;
```

Wenn wir an dieser Stelle in die Funktion `collatz_laenge` wechseln wollen, können wir statt `next` die Anweisung `step` benutzen, die einen Schritt in der Ausführung weitergeht und dabei auch in die Funktion springt, wenn die Ausführung dort weitergeht.

```
(gdb) run
Starting program: /home/paffenholz/temp/collatz
Geben Sie eine positive ganze Zahl ein: 12

Breakpoint 1, main () at collatz.c:46
46    int laenge = collatz_laenge(startwert);
(gdb) step
collatz_laenge (startwert=12) at collatz.c:26
26    int c = startwert;         // Initialisierung
```

Falls Ihr Programm Argumente über die Kommandozeile einliest, haben Sie zwei Möglichkeiten, diese auch im Debugger zu setzen. In der ersten Variante können Sie diese mit der Option `-args` direkt beim Aufruf des Debuggers angeben. Zudem können Sie innerhalb des Debuggers die Anweisung `set args` benutzen.

Manchmal ist es auch interessant herauszufinden, von wo, und in welcher Reihenfolge Funktionen aus anderen Funktionen heraus aufgerufen wurden. Immer wenn wir eine Funktion aufrufen, muss das System die Funktion, aus der heraus wir die Funktion aufrufen (die auch **`main`** sein kann), mit ihrem aktuellen Zwischenstand an Variablenbelegungen aus dem aktuellen Speicher herausnehmen und auf einem Stapel zwischenparken. Diesen Verlauf können wir uns mit dem Befehlt `bt` (für *backtrace*) anzeigen lassen. Das machen wir beispielhaft mit dem rekursiven Programm für den größten gemeinsamen Teiler, bei dem wir von vornherein wissen, dass die Funktion zur Berechnung des `ggT` mehrfach aus sich heraus aufgerufen wird. Hier ist noch einmal das Programm.

Programm 14.4: kapitel_09/ggt_rekursiv.c

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

long long gcd ( long long a, long long b) {
    return b ? gcd(b,a%b) : a;
}

int main(int argc, char** argv) {

   long long a = atoll(argv[1]);
   long long b = atoll(argv[2]);
   if ( a < b ) {
     long long c = a;
     a = b;
     b = c;
   }

   long long g = gcd(a,b);

   printf("Der groesste gemeinsame Teiler von %lld und %lld ist %lld\n",a,b,g);
   return 0;
}
```

Wir rufen darauf den Debugger auf, setzen als Argumente die Zahlen 106 und 28, und lassen das Programm fünfmal an dem Aufruf der Funktion `gcd` vorbeilaufen, bevor wir unterbrechen.

```
gdb ggt_rekursiv
>>> set args 106 28
>>> b 20
Breakpoint 1 at 0x400576: file ggt_rekursiv.c, line 20.
>>> ignore 1 4
Will ignore next 4 crossings of breakpoint 1.
>>> r
Starting program: /home/paffenholz/temp/ggt_rekursiv 106 28

Breakpoint 1, gcd (a=4, b=2) at ggt_rekursiv.c:20
20      return b ? gcd(b,a%b) : a;
>>> bt
#0  gcd (a=4, b=2) at ggt_rekursiv.c:20
#1  0x0000000000400596 in gcd (a=6, b=4) at ggt_rekursiv.c:20
#2  0x0000000000400596 in gcd (a=22, b=6) at ggt_rekursiv.c:20
#3  0x0000000000400596 in gcd (a=28, b=22) at ggt_rekursiv.c:20
#4  0x0000000000400596 in gcd (a=106, b=28) at ggt_rekursiv.c:20
#5  0x0000000000400610 in main (argc=3, argv=0x7fffffffde98) at ggt_rekursiv.c:33
>>>
```

Hier sehen wir den Funktionsstapel, auf dem ganz unten die **main**-Funktion liegt, und darüber vier Aufrufe von `gcd`, zusammen mit den an sie übergebenen Parametern. Wenn wir das Programm eine Runde weiterlaufen lassen, sehen wir, dass ein neuer Aufruf auf den Stapel gelegt wird.

```
>>> c
Continuing.

Breakpoint 1, gcd (a=2, b=0) at ggt_rekursiv.c:20
20      return b ? gcd(b,a%b) : a;
```

```
>>> bt
#0  gcd (a=2, b=0) at ggt_rekursiv.c:20
#1  0x0000000000400596 in gcd (a=4, b=2) at ggt_rekursiv.c:20
#2  0x0000000000400596 in gcd (a=6, b=4) at ggt_rekursiv.c:20
#3  0x0000000000400596 in gcd (a=22, b=6) at ggt_rekursiv.c:20
#4  0x0000000000400596 in gcd (a=28, b=22) at ggt_rekursiv.c:20
#5  0x0000000000400596 in gcd (a=106, b=28) at ggt_rekursiv.c:20
#6  0x0000000000400610 in main (argc=3, argv=0x7fffffffde98) at ggt_rekursiv.c:33
>>>
```

Die an den aktuellen Aufruf übergebenen Parameter sind jetzt $a = 2$ und $b = 0$, hier wird also die rekursive Kette von Funktionsaufrufen enden. Wir setzen noch eine Markierung auf die Zeile 21 und setzen zweimal fort.

```
>>> b 21
Breakpoint 2 at 0x40059c: file ggt_rekursiv.c, line 21.
>>> c
Continuing.

Breakpoint 2, gcd (a=2, b=0) at ggt_rekursiv.c:21
21  }
>>> c
Continuing.

Breakpoint 2, gcd (a=4, b=2) at ggt_rekursiv.c:21
21  }
>>> bt
#0  gcd (a=4, b=2) at ggt_rekursiv.c:21
#1  0x0000000000400596 in gcd (a=6, b=4) at ggt_rekursiv.c:20
#2  0x0000000000400596 in gcd (a=22, b=6) at ggt_rekursiv.c:20
#3  0x0000000000400596 in gcd (a=28, b=22) at ggt_rekursiv.c:20
#4  0x0000000000400596 in gcd (a=106, b=28) at ggt_rekursiv.c:20
#5  0x0000000000400610 in main (argc=3, argv=0x7fffffffde98) at ggt_rekursiv.c:33
>>>
```

Ein Aufruf von `gcd` wurde hier wieder vom Stapel genommen, und es sind noch vier weitere vorhanden. Nach zwei weiteren Schritten sehen wir also

```
>>> c
Continuing.

Breakpoint 2, gcd (a=6, b=4) at ggt_rekursiv.c:21
21  }
>>> c
Continuing.

Breakpoint 2, gcd (a=22, b=6) at ggt_rekursiv.c:21
21  }
>>> bt
#0  gcd (a=22, b=6) at ggt_rekursiv.c:21
#1  0x0000000000400596 in gcd (a=28, b=22) at ggt_rekursiv.c:20
#2  0x0000000000400596 in gcd (a=106, b=28) at ggt_rekursiv.c:20
#3  0x0000000000400610 in main (argc=3, argv=0x7fffffffde98) at ggt_rekursiv.c:33
>>>
```

Die Funktionen werden also in der Reihenfolge wieder vom Stapel genommen, in der sie auf den Stapel gelegt wurden, so wie wir das anhand der rekursiven Beschreibung des Programms auch erwartet hätten. Mit `fin` können wir das Programm zu Ende laufen

lassen.

```
>>> fin
Run till exit from #0  gcd (a=22, b=6) at ggt_rekursiv.c:21
0x0000000000400596 in gcd (a=28, b=22) at ggt_rekursiv.c:20
20      return b ? gcd(b,a%b) : a;
Value returned is $1 = 2
```

Interessanter als bei rekursiven Programmen ist diese Funktion natürlich in Programmen, wo der Ursprung eines Funktionsaufrufs nicht so eindeutig schon anhand des Quellcodes nachvollzogen werden kann, zum Beispiel, wenn eine Funktion von vielen verschiedenen Stellen aufgerufen wird.

Es gibt noch eine ganze Reihe weiterer Möglichkeiten, mit `gdb` den Ablauf des Programms zu untersuchen. Mit

```
$ man gdb
```

bekommen Sie eine Zusammenfassung.

Wenn Sie das Programm länger benutzen, dann werden Sie feststellen, dass sich manche der Konfigurationsmöglichkeiten für gdb immer wieder eingeben, insbesondere wenn Sie von der Möglichkeit von `gdb` Gebrauch machen eigene kleine Unterfunktionen oder Abkürzungen für häufige Befehle zu definieren. Für solche Einstellungen hat `gdb` auch eine Initialisierungsdatei, in der solche Anweisungen hinterlegt und dann bei jedem Start automatisch ausgeführt werden. Genaugenommen gibt es zwei solche. In Ihrem `HOME`-Verzeichnis und im Verzeichnis, in dem sie `gdb` ausführen, können Sie eine Datei mit dem Namen `.gdbinit` ablegen (beachten Sie den Punkt am Anfang!). Die erste wirkt sich auf jeden Aufruf von `gdb` aus, die zweite nur auf Aufrufe im Verzeichnis, in dem sie liegt.

Außerdem wird es bei der Fehlersuche in komplizierteren Programmen auch vorkommen, dass Sie immer wieder die gleichen Markierungen und Bedingungen setzen werden, oder die Kommandozeilenparameter setzen müssen, wenn Sie nach einem Durchlauf einen Fehler gefunden, dieses im Code verbessert und das Programm neu übersetzt haben, es aber immer noch nicht so funktioniert, wie Sie sich das vorstellen. Um diese wiederholten Aufrufe, die spezifisch für ein Programm sind, nicht immer neu tippen zu müssen, können Sie diese in eine Textdatei schreiben, und diese mit der Option `-x` an `gdb` übergeben. Im letzten Beispiel hätten wir zum Beispiel

```
set args 106 28
b 20
ignore 1 4
```

in einer solchen Datei, zum Beispiel `ggt_gdb.init` ablegen können. Mit dem Aufruf

```
$ gdb ggt_rekursiv -x ggt_gdb.init
Reading symbols from ggt_rekursiv...done.
Breakpoint 1 at 0x400576: file ggt_rekursiv.c, line 20.
>>> r
Starting program: /home/paffenholz/temp/ggt_rekursiv 106 28

Breakpoint 1, gcd (a=4, b=2) at ggt_rekursiv.c:20
20      return b ? gcd(b,a%b) : a;
>>> bt
#0  gcd (a=4, b=2) at ggt_rekursiv.c:20
#1  0x0000000000400596 in gcd (a=6, b=4) at ggt_rekursiv.c:20
```

```
#2  0x0000000000400596 in gcd (a=22, b=6) at ggt_rekursiv.c:20
#3  0x0000000000400596 in gcd (a=28, b=22) at ggt_rekursiv.c:20
#4  0x0000000000400596 in gcd (a=106, b=28) at ggt_rekursiv.c:20
#5  0x0000000000400610 in main (argc=3, argv=0x7fffffffde98) at ggt_rekursiv.c:33
>>>
```

können wir gleich den Programmlauf starten.

14.3 Dashboard

Alleine mit gdb ist es mühsam, den Programmablauf mit der Lage der Markierungen zu überblicken und sich immer wieder die Werte der Variablen ausgeben zu lassen. Daher gibt es eine ganze Reihe von weiteren Programmen, die auf gdb aufbauen und die Ausgaben von gdb graphisch darstellen. Wir wollen hier ein sehr einfaches solches Programm kurz vorstellen, das gdb-dashboard[2].

Das Dashboard ist nicht wirklich ein eigenständiges Programm, sondern eine Reihe von Anweisungen an gdb, die es ermöglichen, die Ausgaben von gdb in einem zweiten Terminal darzustellen. Für unser Beispiel erhalten wir damit eine Ausgabe in der Form

```
--- Source --------------------------------------------------------------------
 24   */
 25  int collatz_laenge(int startwert) {
 26    int c = startwert;         // Initialisierung
 27    int laenge = 0;
!28    while ( c != 1 ) {         // naechstes Folgenglied, falls 1 nicht erreicht
29      if ( c % 2 == 0 ) {
 30        c = c/2;
 31      } else {
 32        c = 3*c+1;
 33      }
 34      laenge = laenge + 1;
--- Variables -----------------------------------------------------------------
arg startwert = 12
loc c = 12, laenge = 0
--- Breakpoints ---------------------------------------------------------------
[1] break at 0x0000000000400603 in collatz.c:28 for /h/ap/collatz.c:28 hit 1 time
--- Stack ---------------------------------------------------------------------
[0] from 0x0000000000400603 in collatz_laenge+22 at collatz.c:29
loc c = 12
loc laenge = 0
[1] from 0x0000000000400692 in main+77 at collatz.c:46
loc startwert = 12
loc laenge = 32767
```

In dieser Konfiguration und diesem Aufruf des gdb haben wir eine Markierung auf Zeile 28 gesetzt und dann einen weiteren Schritt im Programm gemacht. Im ersten Abschnitt der Ausgabe sehen wir einen Codeausschnitt der Stelle, an der wir sind. Darunter folgt eine Liste der aktuell definierten Variablen. Dann kommt unsere Markierung mit seiner Nummer und der Zeile im Code (sowie dem Dateinamen, was wichtig ist, wenn wir mehr als eine Datei mit code haben). Davor steht auch die Adresse dieser Programmzeile im Speicher, das ist in der Regel nicht interessant. Im letzten Abschnitt

[2] github.com/cyrus-and/gdb-dashboard

steht der Funktionsaufrufstapel (*stack*). In unserem Beispiel sehen wir hier, dass wir in der Funktion `collatz_laenge` in Zeile 29 sind und in diese Funktion zwei lokale Variablen `c` und `laenge` definiert sind. Ihr Wert ist angegeben. In der nachfolgenden Zeile sehen wir, dass die Funktion aus der Funktion **`main`** heraus aufgerufen wurde, und wir sehen wieder die in dieser Funktion lokalen Variablen. Bei komplizierten Programmen kann dieser Stapel länger sein, wenn wir weitere Funktionen aufrufen. Interessant ist dieser Stapel auch bei rekursiven Funktionen, da Sie hier sehen können, dass für jeden neuen Aufruf hier ein weiterer Eintrag im Stapel erscheint.

Zur Installation legen Sie die Datei `.gdbinit`, die auf der angegebenen Webseite steht, in ihr `$HOME`-Verzeichnis. Danach können Sie, nachdem Sie im Debugger das Programm gestartet haben, mit `help dashboard` die Hilfe zu `dashboard` aufrufen.

Die Ausgabe von `dashboard` erfolgt in einem anderen Terminal. Öffnen Sie dazu ein neues Terminal und lassen Sie sich den Namen anzeigen mit

```
$ tty
/dev/pts/1
```

Im Debugger können Sie nun mit

```
>>> dashboard -output /dev/pts/1
```

dieses Terminal für die Ausgabe festlegen. Danach können Sie das Programm wie im letzten Abschnitt mit `gdb` schrittweise oder bis zu Markierungen laufen lassen. Im `dashboard`-Terminal werden bei jeder Unterbrechung des Programms einige Parameter angezeigt. Sie können konfigurieren, was hier angezeigt wird (dazu wieder bei `help dashboard`) nachsehen). Mit

```
>>> dashboard expressions watch
```

können Sie zum Beispiel einen eigenen Ausdruck definieren, dessen Wert im `dashboard` dann angezeigt wird (das können Speicheradressen, Variablen, Berechnungen, Listenelemente, usw. sein).

Im vorangegangenen Abschnitt haben wir die Möglichkeit diskutiert, Konfigurationsbefehle an `gdb` in Dateien abzulegen, die am Anfang eingelesen werden. Die Konfiguration von `dashboard` bietet sich hierfür insbesondere an. In `.gdbinit` können wir unsere präferierte Konfiguration ablegen (wenn Sie das in die globale Konfiguration eintragen, dann müssen Sie daran denken, dass auch das `dashboard` in dieser Datei initialisiert wird, und diese nicht ersetzen, sondern am Ende ergänzen sollten), während wir in einer mit der Option `-x` eingelesenen Datei spezifische Konfigurationen für unser Programm einlesen, wie zum Beispiel Ausdrücke, die wir verfolgen wollen.

15 Objektcode und Linker

Bisher haben wir aus den Quelldateien mit `gcc` direkt eine ausführbare Datei erzeugt. Dafür haben wir `gcc` alle Quelldateien als Parameter übergeben, in der Form (siehe Programm 12.7)

```
\$ gcc stack.c stack_main.c -o stack_main
```

und haben dann unser Programm als `stack_main` erhalten. Dabei fasst `gcc` für uns zwei eigenständige Schritte der Erzeugung eines ausführbaren Programms zusammen,

- ▷ das Übersetzen der einzelnen Programmteile in Maschinencode (kompilieren) und
- ▷ das Verbinden (linken) der einzelnen Maschinencodeteile zu einem einzelnen Programm.

Der Ausdruck *übersetzen* taucht hier dann in einer zweiten Bedeutung auf, bisher haben wir ihn für den gesamten Prozess bis zum fertigen Programm benutzt. Aus diesem Grund benutzen manche für den Gesamtprozess lieber *bauen* statt *übersetzen*. Wir bleiben hier aber bei letzterem und weisen explizit darauf hin, welcher Prozess gemeint ist, wenn das nicht im Kontext ersichtlich ist.

Wir können diese Schritte der Programmerzeugung auch getrennt durchführen. Das hat einige Vorteile. Insbesondere ist es nicht immer nötig, alle Quellcodedateien erneut in Maschinencode zu übersetzen, wenn wir nur einige davon verändert haben. Es reicht dann, die veränderten Dateien neu zu übersetzen und mit den anderen anschließend zu verbinden (linken). Wir werden das im nächsten Kapitel auch bei der Erstellung von `makefiles` ausnutzen, die uns die Arbeit des Erstellen eines Programms aus dem Quellcode deutlich erleichtern, sobald unser Projekt umfangreicher wird.

Im ersten Schritt werden beim Übersetzen alle Dateien einzeln und unabhängig voneinander in Maschinencode übersetzt. Falls es zwischen den Dateien Querverbindungen gibt (Funktionsaufrufe, Variablendefinitionen, ...), nimmt `gcc` an dieser Stelle an, dass eine Definition, die sich nicht in der aktuellen Datei finden lässt, in einer anderen existieren muss und setzt nur einen Verweis auf die fehlende Definition. Erst der Linker versucht im zweiten Schritt dann diese Referenzen aufzulösen, und erst hier würden fehlende Funktions- oder Variablendefinitionen auffallen. Sie bekommen dann eine Fehlermeldung der Form *symbol(s) not found* oder *undefined reference*. Diese Meldung kommt vom Linker, anders als die Meldungen, dass zum Beispiel ein Semikolon fehlt.

Die beiden Schritte können wir mit Optionen an `gcc` steuern. Zur Erzeugung des Maschinencodes brauchen wir die Option `-c`. Für solche Dateien nimmt man in der Regel die Endung `.o`. Wenn wir keinen Namen für die Ausgabedatei angeben (mit der Option `-o`), dann erzeugt `gcc` von alleine für uns den Dateinamen, in dem die Endung `.c` durch `.o` ersetzt ist.

```
\$ gcc -c stack.h
```

erzeugt eine Datei `stack.o`. Diese Datei ist in weiten Teilen nicht mehr für uns lesbar. Sie können sie sich trotzdem einmal anzeigen lassen, zum Beispiel mit

```
\$ cat stack.o
```

Bei den meisten Systemen erscheint dann eine in weiten Teilen eher sinnlose Zeichenfolge, aber sie können die in `stack.c` enthaltenen Zeichenketten wie *die Liste enthält:* erkennen, und am Ende die Namen der Funktionen, die wir in der Datei definiert haben (die in der Datei enthaltenen *Symbole*). Eine etwas strukturiertere Ausgabe der in dieser Datei verwendeten Funktionsnamen, die noch aus anderen Dateien aufgelöst oder für andere Dateien zur Verfügung gestellt werden, können Sie sich mit

```
\$ nm stack.o
                 U _free
0000000000000050 t _free_node
0000000000000000 T _free_stack
00000000000000b0 t _init_node
                 U _malloc
0000000000000170 T _peek_stack
00000000000000f0 T _pop
00000000000001a0 T _print_stack
0000000000000214 d _print_stack.count
                 U _printf
0000000000000070 T _push
0000000000000150 T _stack_is_empty
```

ansehen. Die Ausgabe bei Ihnen kann etwas variieren. Wir wollen hier nicht näher auf die Ausgabe, ihre Bedeutung und die Information, die Sie daraus gewinnen könnten, eingehen. Sie erkennen aber alle in der Datei definierten Funktionsnamen (zum Beispiel `peek_stack`), und die Namen aller Funktionen, die verwendet wurden, aber nicht in der Datei definiert worden sind (die also vom Linker noch aufgelöst werden müssen), zum Beispiel `malloc`. Der Buchstabe `U` vor diesen Funktionen steht tatsächlich für *undefined*. Bei allen Funktionen, die in der Datei wirklich definiert sind (meistens am `T` oder `t` erkennbar) steht am Anfang der Zeile die Adresse der Funktion, relativ zum Anfang der Datei.

Wir müssen noch die zweite Quelldatei `stack_main.c` in Maschinencode übersetzen, mit

```
\$ gcc -c stack_main.c
```

bevor wir dann zum zweiten Schritt übergehen und die beiden Teile zusammenlinken. Das geht ebenfalls mit `gcc`, dem wir nun einfach die Objektdateien übergeben.

```
\$ gcc -o stack_main stack_main.o stack.o
```

Damit erhalten wir wie bisher auch eine ausführbare Datei `stack_main`. Der Linker baut den Maschinencode zu einer einzigen Datei zusammen, und versucht dabei, alle bisher offenen Symbole aus den einzelnen Dateien aufzulösen und mit der richtigen Definition aus einer anderen Datei zu verbinden. An dieser Stelle werden auch die Funktionen aus der Standardbibliothek, wie zum Beispiel `printf` oder `malloc` hinzugefügt, die als Maschinencode auf Ihrem PC vorliegen und von `gcc` zum Programm hinzugefügt werden.

Die verschiedenen Dateiarten können wir uns mit `file` auch ansehen:

```
file stack.c
stack.c: C source, UTF-8 Unicode text
\$ file stack.o
stack.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
\$ file stack_main
stack_main: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
     interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1
    ]=879af1a14f2af4d4cb4cda8b4302328d5f6f7bf4, not stripped
```

Auf eine nähere Erklärung der Ausgabe wollen wir auch an dieser Stelle verzichten, aber Sie sehen, dass sie anhand der Ausgabe von `file` zwischen den Dateiarten unterscheiden können.

Diese Aufteilung in zwei unabhängige Schritte hat einige Vorteile. Aus Sicht der Programmierer von `gcc` ist es sicherlich sinnvoll, wenn immer möglich, die Aufgabe in kleinere Teile zu zerlegen, die sich unabhängig erledigen lassen und dann auch unabhängig auf Funktion und Fehler überprüft werden können. Für uns hat es zum Beispiel den Vorteil, dass wir, wenn wir die Datei `stack_main.c` verändern, nur diese wieder in Maschinencode übersetzen müssen und sie dann mit der schon vorhandenen `stack.o` linken können um das Programm zu aktualisieren. Bei größeren Dateien kann das eine Menge Zeit einsparen. Aus diesem Grund ist es auch sinnvoll, sich mit dieser Aufteilung zu beschäftigen und diese bei größeren Projekten zur Beschleunigung des Übersetzens und der Vereinfachung des Prozesses der Programmerstellung auszunutzen. Da es bei vielen Dateien manchmal mühsam sein kann sich zu merken, welche Dateien man verändert hat, gibt es dafür wiederum Hilfsprogramme. Eines davon, dass auch noch den gesamten Prozess des Kompilierens und Linkens für uns strukturiert und organisiert, lernen wir im nächsten Abschnitt kennen. Dort tauchen dann auch die Option `-c` an `gcc` und die Objektdateien wieder auf.

Ein weiterer Grund der Aufteilung, auf den wir in dieser Vorlesung aber nicht weiter eingehen werden, ist die Erstellung von statischen oder dynamischen Bibliotheken. Wir sind solchen Bibliotheken schon begegnet. Etwas verdeckt schon der Standardbibliothek (zum Beispiel mit `stdio.h`, deren Funktionen an irgendeiner Stelle auch in Maschinencode vorliegen müssen, aber auch in fremden Bibliotheken wie der `gmp`, die wir mit `-l gmp` eingebunden haben. Auch diese liegt in Maschinencode auf Ihrem PC vor.

Bibliotheken sind, sehr vereinfacht, Dateien mit Maschinencode, die auf eine etwas andere Weise zusammengebaut sind als unsere Objektdateien, und die der Linker beim Verbinden unseres Programms zu einer einzigen Datei in unser ausführbares Programm einbindet. Bei der Standardbibliothek geschieht das für uns unsichtbar. Bei den anderen haben wir mit `-l` den Namen angegeben, und `gcc` weiß, wo es die Bibliothek suchen muss. Auf den meisten Systemen ist das das Verzeichnis `/usr/lib` und seine Unterverzeichnisse.

Die Bibliothek `gmp` liegt zum Beispiel oft an der Stelle

```
/usr/lib/x86_64-linux-gnu/libgmp.so
```

bei Linux-Systemen und

```
/usr/local/lib/libgmp.dylib
```

auf MacOS, wenn Sie die `gmp` mit `homebrew` installiert haben. Der Pfad kann aber abweichen. Bibliotheken gibt es in zwei Varianten, *statisch* und *dynamisch*. Statische

Bibliotheken werden beim Linken in das Programm kopiert, bei dynamischen Bibliotheken wird nur ein Verweis auf die Bibliothek aufgenommen und erst zur Laufzeit die Bibliothek wirklich geladen. Die Option `-l` an `gcc` wird also erst vom Linker verwendet, nicht von dem Teil, der den Maschinencode unserer Quelldateien erzeugt. Sie können mit `gcc` auch solche Bibliotheken erzeugen. Wie das geht, können Sie in der Dokumentation von `gcc` nachlesen.

Beide Versionen, statische und dynamische Bibliotheken, haben Vor- und Nachteile. In der Regel nimmt man aber die dynamische Version (das macht auch `gcc`, wenn Sie nichts anderes vorgeben), da dann das Programm kleiner wird, und wir nicht so oft den gleichen Code auf der Festplatte speichern müssen. Zudem profitieren Sie auf diese Weise auch davon, wenn die Autor*inn*en der Bibliothek Fehler verbessern und Sie die Bibliothek bei sich aktualisieren. Beim statischen Linken müssten Sie Ihr Programm neu übersetzen, während beim dynamischen Linken beim nächsten Aufruf automatisch die aktuelle Version benutzt wird. Auf der anderen Seite kann es aber auch passieren, dass dann Ihr Programm nicht mehr läuft, weil sich in der Bibliothek etwas verändert hat, was nicht zu Ihrem Programm passt.

`gcc` nimmt normalerweise an, dass Sie *dynamisch* linken wollen. Wenn Sie *statisch* linken wollen, können Sie `gcc` die Option `-static` mitgeben (eigentlich braucht nur der Linker diese Information). Wenn Sie das an unserem Beispiel ausprobieren, können Sie an der Ausgabe von

```
nm static_main
```

erkennen, dass beim dynamischen Linken viel weniger Symbole aufgeführt werden und einige noch als undefiniert gelistet werden (mit einem `U` in der zweiten Spalte). Beim statischen Linken ist die Liste erheblich länger, und es sollten auch keine undefinierten Symbole mehr vorhanden sein.

Mit der Option `-v` gibt Ihnen, neben einigen anderen Informationen auch die einzelnen Schritte aus, die beim Übersetzen erfolgen. Wenn Sie

```
gcc -v -o stack_main stack.c stack_main.c
```

ausführen, können Sie erst einige Konfigurationsoptionen und Suchpfade für Bibliotheken sehen, bevor, relativ weit unten, meistens ein Aufruf des Assemblerkompilers `as` erkennbar ist, und anschließend ein Aufruf des Linkers. Dieser kann verschiedene Namen haben, typisch sind `ld` oder `collect2`.

Zum Abschluss schauen wir uns noch kurz an, dass auch die Übersetzung in Machinencode selbst in zwei Teile zerfällt. In einem ersten Schritt wird hier nach `Assembler` übersetzt, einer etwas allgemeineren Maschinensprache, die prinzipiell auch noch für Menschen lesbar ist. In einem zweiten Schritt wird dieser Code dann in echten Maschinencode übersetzt, wie wir das oben gesehen haben. Die Übersetzung in Assembler erfolgt mit der Option `-S`. Dateien mit Assembler haben oft die Endung `.s`, und `gcc` wählt diese automatisch für uns, wenn wir nichts anderes angeben.

```
\$ gcc -S stack.c
```

erzeugt also eine Datei `stack.s`. Die Datei ist recht lang. Hier ist ein kleiner Ausschnitt.

```
peek_stack:
.LFB8:
        .cfi_startproc
```

```
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movq    %rdi, -24(%rbp)
        movl    $-2147483648, -4(%rbp)
        movq    -24(%rbp), %rax
        movq    (%rax), %rax
        testq   %rax, %rax
        je      .L14
        movq    -24(%rbp), %rax
        movq    (%rax), %rax
        movl    (%rax), %eax
        movl    %eax, -4(%rbp)
.L14:
        movl    -4(%rbp), %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE8:
        .size   peek_stack, .-peek_stack
        .section        .rodata
        .align 8
```

Hier wird die Funktion `peek_stack` definiert (wobei die Teile, die wir hier direkt identifizieren können, meistens Kommentare sind). Assemblercode arbeitet mit einfachen Kommandos, die die Einträge an einzelnen Registern des Speichers manipulieren können. Dazu kommt in der ersten Spalte die entsprechende Anweisung, und dahinter das oder die Register, die sie betrifft. Zum Beispiel verschiebt `movq` einen 64-bit Wert von einer Adresse an eine andere, und `popq` holt einen Wert vom Stack.

Version vom 10. Februar 2022

16 makefiles

Bei größeren Projekten kann es mühsam werden, bei dem Aufruf von `gcc` an alle Dateien mit Quellcode zu denken, die an den Compiler übergeben werden müssen. Außerdem ist es nicht unbedingt notwendig, nach jeder Änderung alle Dateien neu in Maschinencode zu übersetzen. Wenn wir Übersetzen in Maschinencode und anschließendes Linken der einzelnen Dateien zu unserem Programm einzeln ausführen, dann reicht es, nur die Dateien, in denen wir etwas verändert haben, erneut in Maschinencode zu übersetzen und dann alle Teile neu zu linken. Auf diese Weise können wir oft auch die Zeit bis zum übersetzten und ausführbaren Programm verkürzen, weil nur ein kleiner Teil der Dateien wirklich neu übersetzt werden muss.

Auf der anderen Seite kann es mühsam sein, bei diesem Prozess den Überblick zu behalten und wirklich auch alle Dateien, die sich verändert haben, neu zu übersetzen. Hierfür gibt es mit dem Programm `make`, das auf allen Linux- und Unix-basierten Betriebssystemen (also auch in MacOS, aber nicht in Windows, außer im WSL) in der Regel installiert ist, ein sehr nützliches Werkzeug, dass diese Aufgabe für uns übernimmt.

Dafür müssen wir einmal eine Konfigurationsdatei anlegen, die die Abhängigkeiten zwischen den Dateien unseres Projekts beschreibt, die Ziele definiert (also in der Regel die Erzeugung eines ausführbaren Programms) und die dafür notwendigen Anweisungen (also zum Beispiel den für unser Projekt notwendigen Aufruf von `gcc`) enthält. Die dafür notwendige Syntax mag am Anfang etwas gewöhnungsbedürftig sein, aber es reicht bei einfacheren Projekten in der Regel, sich eine schon bestehende Konfigurationsdatei zu nehmen und an die speziellen Anforderungen anzupassen. Da bei Projekten in `C` der Ablauf des Übersetzens in ein Programm meistens sehr ähnlich ist und einem kanonischen Schema folgt, müssen oft nur die konkreten Programmnamen angepasst werden. Hier ist ein (sehr einfaches) Beispiel.

Programm 16.1: `kapitel_16/collatz-1/makefile`

```
.PHONY: clean

collatz: collatz.c
    gcc -o collatz collatz.c

clean:
    rm collatz
```

Die Konfigurationsdatei kann einen beliebigen Namen haben und an einer beliebigen Stelle im Dateisystem liegen. Es ist allerdings vorteilhaft, sie `makefile` oder `Makefile` zu nennen und sie in das Verzeichnis zu legen, in dem auch die Dateien für unser Programm liegen. Wenn wir `make` ohne weitere Parameter aufrufen, sucht das Programm `make` nämlich im aktuellen Verzeichnis nach einer Datei mit einem dieser beiden Namen und nimmt diese als Konfigurationsdatei. Wenn wir die Datei anders nennen oder an einer

Version vom 10. Februar 2022

anderen Stelle ablegen, müssen wir den Namen (ggf. mit dem Pfad zur Datei) mit der Option `-f` an `make` übergeben.

Das Beispiel ist eine Konfiguration zum Übersetzen des Programms Programm 2.1 aus Kapitel 2. wir haben in der Konfiguration zwei *Ziele* definiert: `collatz` und `clean`. Ein Ziel hat immer die folgende Form.

```
<Ziel> : <Voraussetzungen>
   <Anweisungen>
```

In der ersten Zeile steht der Name des Ziels vor einem Doppelpunkt. In den nachfolgenden Zeilen stehen dann Anweisungen, die ausgeführt werden sollen, um dieses Ziel zu erreichen (also zum Beispiel der Aufruf von `gcc`). Die Einrückung vor den Anweisungen muss *zwingend* mit einem einzigen `Tab` erfolgen, nicht durch Leerzeichen. Andernfalls scheitert `make`. Das ist eine der häufigsten Fehlerquellen in `makefiles`, unter anderem auch, weil es in den meisten Editoren nicht sofort ersichtlich ist, ob die Einrückung durch `Tab` oder Leerzeichen erfolgt ist.

Ziele sind in der Regel Namen von Dateien, die erzeugt oder erneuert werden sollen, wie in unserem Fall bei der ersten Regel, die `collatz` aus `collact.c` erzeugt. Wir können auch Ziele angeben, die keine Datei erzeugen, sondern nur eine Aktion ausführen. In unserem Beispiel ist das das Ziel `clean`. Diese Ziele müssen wir in der `makefile` markieren, damit `make` weiß, dass es nicht auch einer Datei mit dem Namen des Ziels suchen soll (also in unserem Fall nach einer Datei mit dem Namen `clean`)[1]. Wir benennen solche nicht zu einem Dateinamen gehörenden Ziele (sogenannte *phony targets*) mit dem speziellen Ziel `.Phony` (beachten Sie den `.` am Anfang!), und wir können mehrere Ziele davon in einer Reihe dahinter angeben. Typische weitere Beispiele solcher Ziele, die zu keiner Datei gehören, und oft in `makefiles` auftauchen, sind `all`, `install`, `check` oder `test`.

Wenn es Voraussetzungen gibt, die erfüllt sein müssen, bevor wir die Anweisungen ausführen können, dann können wir diese in der ersten Zeile nach dem Doppelpunkt aufführen. Das sind wieder Namen von Zielen, also in der Regel Dateinamen von Dateien, die vorher vorhanden sein müssen. In unserem Fall wollen wir, dass für die Ausführung des Ziels `collatz` die Datei `collatz.c` schon vorhanden ist. In dieser Relation zwischen Ziel und Voraussetzung wird noch mehr überprüft, aber wir testen unsere `makefile` erst, bevor wir die Syntax weiter analysieren.

```
$ make collatz
gcc -o collatz collatz.c
$ make clean
rm collatz
$ make collatz
gcc -o collatz collatz.c
$ make collatz
make: 'collatz' is up to date.
```

An dieser kurzen Sequenz können wir schon einiges über `make` erkennen. Mit dem Ziel `collatz` wurde unsere Datei `collatz` erzeugt, und mit dem nachfolgenden Aufruf von `clean` wieder gelöscht. Wir haben sie dann wieder erzeugt, und direkt anschließend

[1] Das ist etwas zu strikt formuliert. Tatsächlich nimmt `make` an, dass ein Ziel nicht zu einer Datei gehört, wenn es keine solche gibt. Wenn wir allerdings ein Ziel `clean` definieren wollen, dass wie im Beispiel die erzeugte Programmdatei löscht, und in unserem Projekt kommt außerdem eine Datei mit dem Namen `clean` vor, dann wird `make` nicht machen, was wir erwarten.

noch einmal das Ziel `collatz` aufgerufen. Beim zweiten Aufruf hat `make` erkannt, dass die Datei `collatz` schon existiert, und ein erneuter Aufruf von `gcc` diese Datei nicht änderte.

Hier vergleicht `make` die Zeitstempel der Dateien. Wenn alle Änderungszeiten der Dateien, die in den Voraussetzungen aufgeführt sind, älter sind als die letzte Änderung der Datei, die erzeugt werden soll, dann nimmt `make` an, dass sich durch eine Ausführung seiner Anweisungen nicht mehr ändern kann, da sie schon mit den bestehenden Voraussetzungen ausgeführt wurden. Wenn nur eine der Dateien neuer ist, werden die Anweisungen ausgeführt. Das können wir zum Beispiel überprüfen, indem wir die Datei `collatz.c` bearbeiten, oder einfach ihren Zeitstempel mit der Anweisung `touch` aktualisieren.

```
$ make clean
rm collatz
$ make collatz
gcc -o collatz collatz.c
$ touch collatz.c
$ make collatz
gcc -o collatz collatz.c
```

Wenn wir `make` ohne Argumente aufrufen, wird das erste in der Datei gefundene Ziel ausgeführt, wir hätten also statt `make collatz` auch nur `make` aufrufen können[2]. In größeren Projekten ist es üblich, als ersten ein *phony target* `all` anzugeben, dass alles auslöst, was zur Erzeugung des vollständigen Projekts notwendig ist[3]. Damit könnte unsere Konfiguration so aussehen.

Programm 16.2: `anhang_B/collatz-2/makefile`

```
.PHONY: clean all

all: collatz

collatz: collatz.c
    gcc -o collatz collatz.c

clean:
    rm collatz
```

Das Ziel `all` enthält in der Regel keine Anweisungen, sondern nur Voraussetzungen. In unserem Fall erwarten wir für das Ziel `all`, dass die Datei `collatz` existiert. Bei einem *phony target* gibt es keine Datei, deren Zeitstempel wir für einen Vergleich heranziehen können. Es wird daher von `make` angenommen, dass es *immer* ausgeführt werden muss. Trotzdem wird in unserem Beispiel nicht mit jedem Aufruf das Programm neu übersetzt:

```
$ make
gcc -o collatz collatz.c
$ make
make: Nothing to be done for ‘all’.
```

[2] Genauer wird das Ziel ausgeführt, dass in der Variablen `.DEFAULT_GOAL := <Ziel>` angegeben ist, jedenfalls solange Sie die GNU-Version von make verwenden. Wenn diese Variable nicht explizit definiert ist, zeigt sie auf das erste Ziel in der Konfiguration.

[3] Falls das Projekt auch eine Dokumentation oder eine Anleitung enthält, die erzeugt werden soll, zum Beispiel in `Latex`, dann ist das meistens nicht im Ziel `all` enthalten, sondern wird üblicherweise durch ein *phony target* `doc` ausgelöst.

Hier evaluiert make zwar auch das Ziel collatz, erkennt aber an den Voraussetzungen dieses Ziels, dass die Anweisungen nicht ausgeführt werden müssen.

Wir nutzen an dieser Stelle aber eigentlich die Vorteile von *makefiles* noch nicht wirklich aus. Diese kommen, bei den meisten Projekten in C, erst zum tragen, wenn wir, wie wir es im letzten Kapitel gesehen haben, die Übersetzung des Programms in die zwei Schritte der Übersetzung in eine Objektdatei und das anschließende Linken zerlegen. Da nur das Linken alle Programmteile braucht, während die Übersetzung in eine Objektdatei nur eine Datei betrifft, hat make hier die Möglichkeit, durch Vergleich der Zeitstempel nur die Quelldateien neu zu übersetzen, die sich auch wirklich verändert haben, und am Ende alles mit den unveränderten Objektdateien zusammenzulinken. Hier ist ein Beispiel, für das wir den Stapel aus Programm 12.7 verwenden.

Programm 16.3: anhang_B/stack-1/makefile

```
.Phony: all clean

all: stack_main

stack_main: stack.o stack_main.o
    gcc -o stack_main stack.o stack_main.o

stack.o: stack.c stack.h
    gcc -c stack.c

stack_main.o: stack_main.c stack.h
    gcc -c stack_main.c

clean:
    rm stack_main.o stack.o stack_main
```

Hier haben wir mehrere Ziele definiert und die Übersetzung des Programms mit gcc in seine Schritte zerlegt. Wir können auch ausprobieren, dass make hier wirklich nur veränderte Dateien neu erzeugt.

```
$ make
gcc -c stack.c
gcc -c stack_main.c
gcc -o stack_main stack.o stack_main.o
$ make
make: Nothing to be done for ‘all’.
$ touch stack_main.c
$ make
gcc -c stack_main.c
gcc -o stack_main stack.o stack_main.o
```

Nachdem wir den Zeitstempel von stack_main.c verändert haben, wird die zugehörige Objektdatei neu erzeugt und wieder alle Teile gelinkt. Die Objektdatei stack.o wird hingegen nicht erneut erzeugt.

Beachten Sie, dass wir in der makefile bei den Abhängigkeiten der beiden Ziele stack.o und stack_main.o auch den Header stack.h aufgeführt haben. Da dieser in den Quelldateien eingebunden wird, wirkt sich natürlich eine Änderung darin auch auf die beiden Objektdateien aus, auch wenn es kein direktes Ziel gibt, in dem der Header involviert ist.

Bei größeren Projekten kann diese Liste der Abhängigkeiten von Headern (oder ande-

ren Dateien), die wir bei jeder Regel angeben müssen, schnell unübersichtlich werden, und wenn sich diese Liste verändert, weil zum Beispiel ein neuer Header hinzukommt, müssen alle diese Listen ergänzt werden. Um das zu vereinfachen, sind Variablen hilfreich, die wir in einem Header definieren können. Variablennamen werden meistens in Großbuchstaben geschrieben, aber das ist nur eine Konvention, und nicht verpflichtend. Auf Variablen wird mit =. Wenn wir auf eine Variable zugreifen wollen, müssen wir sie in runde Klammern setzen und ein $ davorschreiben[4]. Im nachfolgenden Beispiel haben wir drei Variablen definiert. Eine für eine Liste von Abhängigkeiten, die wir in den Voraussetzungen einer Regel verwenden, eine für den Namen des verwendeten Compilers (damit wir zum Beispiel leicht zu `clang` wechseln könnten, und eine letzte, in der wir Optionen an den Compiler sammeln (im Beispiel die Option, mit Debugsymbolen zu übersetzen).

Programm 16.4: `anhang_B/stack-2/makefile`

```
DEPS = stack.h
CC = gcc
CFLAGS = -g

.Phony: all clean

all: stack_main

stack_main: stack.o stack_main.o
    $(CC) $(CFLAGS) -o stack_main stack.o stack_main.o

stack.o: stack.c $(DEPS)
    $(CC) $(CFLAGS) -c stack.c

stack_main.o: stack_main.c $(DEPS)
    $(CC) $(CFLAGS) -c stack_main.c

clean:
    rm stack_main.o stack.o stack_main
```

Wir können auch den Platzhalter $*$ verwenden, der den gleichen Effekt hat im Terminal. Eine Regel der Form

```
prog : *.c
```

nimmt alle Quelldateien mit der Endung `.c` als Voraussetzung für die Erzeugung von `prog`. Eine Variable wird erst dann aufgelöst, wenn wir sie wirklich verwenden. Wenn eine Variable also von einer weiteren Variablen abhängt, dann kann sich der Wert einer Variablen bei mehrfacher Verwendung unterscheiden, wenn sich die Variable, von der sie abhängt, dazwischen geändert hat, In

```
LIBS = -lgmp
LDFLAGS = $(LIBS)
LIBS = -lsqlite3
```

ergibt `$(LDFLAGS)` am Ende also `-lsqlite3`. Das ist meistens nützlich in dieser Form, wir müssen aber aufpassen, dass dadurch keine inkrementelle Definition einer Variablen möglich ist, eine Definition der Form

[4] Alternativ in geschweifte Klammern, das macht keinen Unterschied, sollte aber innerhalb einer Konfigurationsdatei einheitlich sein. Bei Variablen, die nur aus einem Zeichen oder Buchstaben bestehen, können wir die Klammern auch ganz weglassen.

```
LDFLAGS = -lgmp
LDFLAGS = $(LDFLAGS) -lsqlite3
```

ergibt in der zweiten Zeile eine unendliche Schleife, statt der vielleicht erwarteten Definition `LDFLAGS = -lgmp -lsqlite3`[5]. Mit dem Platzhalter `%` können wir auch *Musterregeln* definieren. Da ist schon in unserem Beispiel nützlich, wo sich die Regel zur Erzeugung der beiden Objektdateien nur im Dateinamen unterscheidet. Hier können wir auch eine Regel der Form

```
%.o : %.c $(DEPS)
    gcc -c %.o
```

einführen. `make` wird an dieser Stelle für jede Quelldatei mit der Endung `.c`, die es findet, eine Regel erzeugen, dessen Ziel dann eine Datei ist, die den gleichen Stammnamen wir die Quelldatei hat, aber mit der Endung `.o`. Das Symbol `%` wird hierbei vor und hinter dem Doppelpunkt und in den Anweisungen durch den gleichen Namen ersetzt. Wir sehen gleich, dass wir durch geeignete Variablen hier auch Variationen in den Regeln ermöglichen können.

Es gibt eine Reihe von nützlichen Variablen, die in einer Regel einer Konfiguration von `make` automatisch definiert werden. Das sind

- `$@`: Der Name des Ziels (also zum Beispiel `stack_main` im zweiten Ziel)
- `$*`: Der Name des Ziels ohne Dateiendung (wenn es eine hat)
- `$<`: Der Name der ersten Datei in den Voraussetzungen
- `$^`: Alle Dateinamen in den Voraussetzungen (Duplikate werden ausgelassen, die Namen sind durch ein Leerzeichen getrennt)
- `$+`: wie zuvor, aber ohne dass Duplikate gelöscht werden
- `$?`: Die Namen aller Voraussetzungen, die einen neueren Zeitstempel als das Ziel haben.

Damit können wir die `makefile` für unseren Stapel auf die folgende Weise umschreiben.

Programm 16.5: `anhang_B/stack-3/makefile`

```
DEPS = *.h
OBJ = stack.o stack_main.o
CC = gcc
CFLAGS = -g

.Phony: all clean

all: stack_main

stack_main: $(OBJ)
    $(CC) $(CFLAGS) -o $@ $^

%.o: %.c $(DEPS)
    $(CC) $(CFLAGS) -c $<

clean:
    rm $(OBJ) stack_main
```

[5] Die Version von make aus dem GNU-Paket hat daher noch eine zweite Möglichkeit der Zuweisung, mit `:=`, bei der zum Zeitpunkt der Zuweisung alle Variablen aufgelöst werden. Damit wäre eine solche inkrementelle Definition möglich.

Sie können natürlich mehr als eine Anweisung unter einer Regel haben (beachten Sie aber, dass die Zeile mit einem `Tab` anfangen muss und nicht durch Leerzeichen eingerückt sein darf). Es muss auch keine Anweisung mit `gcc` sein, jedes Programm, das sie von der Kommandozeile aus aufrufen können, können Sie auch hier aufrufen. Bei der Ausführung von `make` wurden die einzelnen ausgeführten Anweisungen ins Terminal geschrieben. Das ist oft nützlich, um den Ablauf nachzuvollziehen und Fehler zu entdecken, aber sobald die Anweisungsketten komplizierter werden, möchte man diese Ausgabe vielleicht unterdrücken. Das kann man mit einem `@` am Anfang erreichen. Wenn sie die Regeln für die Ausführbare Datei und die Objektdateien durch

```
stack_main: $(OBJ)
        @$(CC) $(CFLAGS) -o $@ $^

%.o: %.c $(DEPS)
        @$(CC) $(CFLAGS) -c $<
```

ersetzen, wird `make` nichts mehr auf den Bildschirm schreiben.

Ein weiterer Vorteil von Variablen in einer `makefile` anstelle einer expliziten Auflistung des Werts an allen Stellen ist die Möglichkeit, diese Variable beim Aufruf von `make` zu überschreiben. Wenn wir in unserem Beispiel also mit `clang` statt `gcc` kompilieren wollen, können wir

```
$ make CC=clang
clang -g -c stack.c
clang -g -c stack_main.c
clang -g -o stack_main stack.o stack_main.o
```

ausführen. Damit werden alle Definitionen der Variable `CC` innerhalb der `makefile` ignoriert. Auf die gleiche Weise können wir auch neue Variablen definieren, die innerhalb der `makefile` dann zur Verfügung stehen. Allerdings müssen wir dann sicherstellen, dass wir die Variable entweder immer übergeben, oder nicht auf die Variable zugegriffen wird, wenn wir sie nicht definieren. Sonst scheitert `make` mit einer Fehlermeldung.

Eine solche Variablenübergabe können wir auch benutzen, um in unserem Programm Variablen an den Präprozessor von `gcc` zu übergeben (um zum Beispiel eine Variable `DEBUG`, die normalerweise auf 0 steht, auf 1 zu setzen und Kontrollausgaben in unserem Programm zu aktivieren). Diese müssen wir dann natürlich an den Compiler weitergeben, zum Beispiel über die Variable `CFLAGS`, die eine Option `-DDEBUG=1` angehängt bekommt.

Es gibt noch viele weitere Möglichkeiten, den Prozess der Programmerzeugung über `make` zu steuern. Wir haben hier nur einen sehr kleinen Ausschnitt der Möglichkeiten dargestellt. Mit dem, was wir hier gesehen haben, sollten Sie allerdings schon für die meisten Projekte kleinerer und mittlerer Größe eine effiziente und leicht aktuell zu haltende `makefile` schreiben können.

Version vom 10. Februar 2022

17 Binäre Bäume

In diesem Kapitel wollen wir uns Datenstrukturen anschauen, die auf *Bäumen* basieren. Bäume sind eine Verallgemeinerung unserer Listenstrukturen aus Kapitel 12. Bei den *verketteten Listen*, die wir uns dort angesehen haben, hat jeder Knoten in der Liste (höchstens) einen Nachfolger und einen Vorgänger in der Liste. In einer Baumstruktur kann nun jeder Knoten mehr als einen Nachfolger haben (aber immer noch nur höchstens einen Vorgänger). Solche Strukturen haben sich auch für die Speicherung von Listen (die ja eigentlich linear sind) als sehr effizient erwiesen, da wir in Bäumen zum Beispiel schneller suchen und an der richtigen Stelle einfügen können, wenn wir die Relation zwischen einem Konten und seinen Nachfolgern geeignet organisieren. Wir werden zwei solcher Strukturen genauer ansehen, die *balancierten Bäume* (konkret als sogenannte *AVL-Bäume*) und *Heaps*. Über die Heap-Struktur erhalten wir auch einen weiteren, in seiner Laufzeit mir `MergeSort` vergleichbaren Sortieralgorithmus, den `HeapSort`.

17.1 Bäume

Ein *Baum* ist eine verkettete Datenstruktur, in der jeder Knoten höchstens einen Vorgänger und mehrere Nachfolger haben kann. Ein Beispiel ist in Abbildung 17.1. In der Abbildung haben wir den *Knoten* des Baumes jeweils eigene Variablennamen gegeben (von a bis t und w), in unseren Implementierungen sind die Knoten dann natürlich wieder nur über die in der Struktur des Knotens enthaltenen Zeiger auf die Nachfolger (und eventuell auch auf den Vorgänger) erreichbar, und wir schreiben die *in dem Knoten gespeicherte Information* an diese Stelle. Diese Zeiger sind im Bild durch die auf die Nachfolger zeigenden Pfeile gekennzeichnet.

Es gibt genau einen Knoten ohne Vorgänger (wie auch in bei den verketteten Listen). Diesen Knoten nennt man meistens *Wurzel* des Baums (und anders als in der Natur ist in den meisten Darstellungen eines Baums in der Mathematik und Informatik die Wurzel oben). Im Beispiel ist der Knoten w die Wurzel.

Die Nachfolger eines Knotens sind seine *Kinder*, während der Vorgänger oft als *Elternknoten* bezeichnet wird. Die Kinder des Knotens g sind also o, p und q, während g der Elternknoten von o, p und q ist. Kinder des gleichen Elternknotens heißen *Geschwister*. Ein Knoten ohne Kinder heißt *Blatt*, alle anderen Knoten heißen *innere Knoten*. Im Beispiel sind also unter anderen i, j, n und s Blätter, w, b und g sind innere Knoten.

Beachten Sie, dass jeder Knoten höchstens einen Vorgänger hat, und genau einen, wenn der Knoten nicht die Wurzel des Baums ist. Jeder Baum hat eine Wurzel (die Entsprechung des Kopfes bei unseren Listen), und es gibt keine Möglichkeit, entlang der durch die Pfeile gegebenen Verbindungen im Kreis zu laufen ohne eine Verbindung mehrfach zu nutzen (auch dann nicht, wenn wir erlauben entgegen der Pfeilrichtung

Version vom 10. Februar 2022

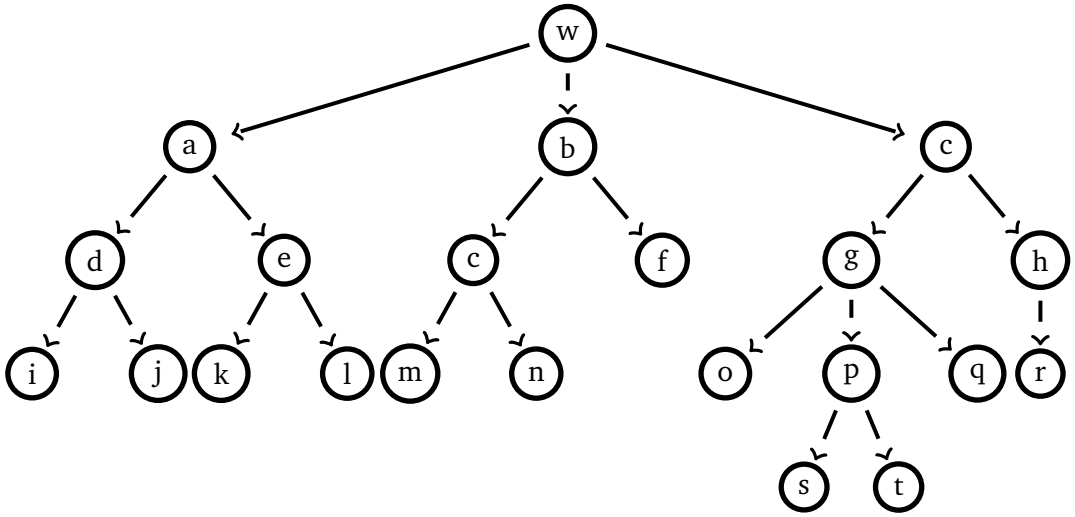


Abbildung 17.1: Ein Baum

zu laufen)[1].

Von jedem Knoten gibt es daher einen eindeutigen Pfad zur Wurzel, in dem man vom Knoten aus immer zum Elternknoten geht, bis man die Wurzel erreicht hat. Für den Knoten m ist das zum Beispiel der Pfad $m - c - b - w$. Die Länge dieses Pfads ist die Anzahl der Kanten auf dem Weg, im Beispiel also 3. Diesen Abstand eines Knotens von der Wurzel nennt man auch die *Höhe* des Knotens, und ein *Level* im Baum sind alle Knoten auf gleicher Höhe. Auf dem ersten Level liegen also die Knoten a, b und c. Auch zwischen beliebigen zwei Knoten im Baum gibt es genau einen Pfad über Kanten, der die beiden Knoten verbindet. Dabei gehen wir von einem Knoten solange zu seinem Elternknoten weiter, bis wir einen gemeinsamen Vorfahren gefunden haben, und gehen dann wieder über die Kinder nach unten zurück. Für die Konten o und r ist das der Pfad $o - g - c - h - r$. Ein Knoten ist *Nachfahre* eines anderen Knotens, wenn dieser Knoten auf dem Weg zur Wurzel liegt. Die Nachfahren von g sind also o, p, q, s, und t.

Die *Höhe* eines Baums ist die maximale Höhe eines seiner Knoten. Offensichtlich brauchen wir uns dafür nur die Höhen der Blätter ansehen. Seltener wird die *Höhe* eines Baums auch mit *Tiefe* oder *Baumtiefe* bezeichnet, auch wenn diese Bezeichnungen der Darstellung in Bildern mit der Wurzel an der Spitze entgegenkäme.

Ein *Teilbaum* eines Baumes ist ein Knoten zusammen mit allen seinen Nachfahren. Ein Teilbaum ist auf diese Weise selbst wieder ein Baum, mit dem ersten Knoten als Wurzel. Alle Begriffe übertragen sich auf diesen Baum, und seine Höhe ist die Gesamthöhe minus die Höhe seiner Wurzel.

Der *Grad* eines Knotens ist die Anzahl seiner Kinder. Im Beispiel haben alle Knoten einen Grad zwischen 0 und 3. Wir nennen einen Baum *binär*, wenn alle Knoten höchstens Grad 2 haben, also höchstens zwei Kinder haben. Meistens geben wir dann auch noch eine Reihenfolge der Kinder vor, wir sprechen dann vom *linken* bzw. *rechten* Kind eines Knotens. Genauso können wir dann vom *rechten* und *linken Teilbaum* unterhalb eines Knotens sprechen, den Teilbäumen, die das rechte oder linke Kind als Wurzel haben.

[1] Das ergibt sich schon aus den restlichen Anforderungen, wie Sie im vierten Semester lernen werden.

In C können wir einen Knoten in einem (binären) Baum zum Beispiel so definieren:

```
struct node {
  struct node * parent;
  struct node * left_child;
  struct node * right_child;
  struct data * elem;
} :
```

wobei `struct data` dann ein Datentyp ist, der die an diesem Knoten im Baum gespeicherte Information enthält. Zur Vereinfachung werden wir im Folgenden meistens stattdessen einfach ein `int` speichern. Wir arbeiten also mit der folgenden Struktur.

```
struct node {
  struct node * parent;
  struct node * left_child;
  struct node * right_child;
  int data;
} :
```

Es sollte Ihnen aber inzwischen keine Probleme mehr bereiten, den Code bei Bedarf entsprechend zu erweitern. Ein kritischer Punkt dabei ist dann natürlich, auch daran zu denken, beim Erzeugen von Knoten passenden Speicherplatz zu reservieren (das geschieht nicht, wenn sie Speicherplatz für eine Instanz von `struct node` reservieren, dort wird nur Platz für den Zeiger auf die Daten reserviert!) und diesen wieder freigeben, wenn Sie den Knoten löschen.

In Knoten, die keinen Elternknoten (*parent*) haben oder nur eines oder kein Kind (*child*), lassen wir den entsprechenden Zeiger, wie schon beim letzten Knoten in verketteten Listen, auf `NULL` zeigen. Nicht alle Implementierungen einer Baumstruktur haben oder brauchen den Zeiger zum Elternknoten. In dem Fall muss man immer über den Zeiger auf die Wurzel bis zum gewünschten Knoten absteigen (ähnlich wie wir in verketteten Listen immer vom Zeiger auf den Kopf der Liste bis zum gesuchten Element weitergegangen sind). Auch in unseren Beispielen werden wir darauf verzichten, da die korrekte Mitführung dieses Zeigers in den einfachen Beispielen mehr Code erfordert als wir an anderer Stelle sparen können. Sie sind aber eingeladen, die Struktur in den Beispielen entsprechend zu erweitern.

In der Regel möchte man bei der Speicherung in einer Baumstruktur Daten haben, die sich vollständig entsprechend einer Vergleichsoperation (einer Totalordnung) anordnen lassen, wie zum Beispiel eine Liste von Zahlen. Dieses Sortierkriterium kann natürlich auch ein Eintrag innerhalb der Struktur *struct data* sein, und vielleicht auch nur speziell zu diesem Zweck dort aufgenommen sein (also zum Beispiel ein *Index* für einen Eintrag in einer Liste). Bei Daten, die sich nicht sinnvoll anordnen lassen, oder bei denen das Sortierkriterium für eine schnelle Sortierung zu aufwendig ist, behilft man sich an dieser Stelle oft mit *Hashes*, einer Zahl, die man aus den Daten schnell ausrechnen kann und die (auf der erwarteten Menge möglicher Eingaben) nach Möglichkeit eindeutig ist.

Um nun sinnvoll Daten in einer solchen Struktur speicher zu können, müssen wir noch weitere Annahmen treffen. Wenn wir zum Beispiel eine Liste von Zahlen in einer solchen Baumstruktur speichern, und wir anschließend feststellen wollen, ob eine vorgegebene Zahl in der Liste ist, wollen wir nicht den gesamten Baum durchsuchen müssen. Wenn wir jedoch zum Beispiel festlegen, dass der Index eines Knotens (der Wert in `struct data`, nach dem wir zwei Elemente vergleichen können, im folgenden also einfach

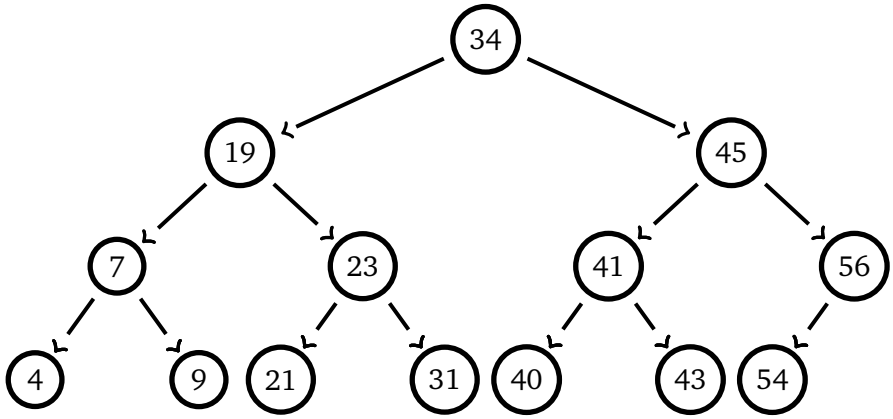


Abbildung 17.2: Die Liste $\{4, 7, 9, 19, 21, 23, 31, 34, 40, 41, 43, 54, 56\}$ in einer Baumstruktur

ein `int`) eines Elternknotens immer größer als der seines linken Kinds und kleiner als der seines rechten Kinds sein soll, können wir oft schneller feststellen, ob unsere Liste ein Element mit einem bestimmten Index enthält. Wenn wir beim Vergleich mit der Wurzel nämlich feststellen, dass unser Index kleiner (größer) als der Index der Wurzel ist dann brauchen wir unseren Index nur noch mit Elementen des rechten (linken) Teilbaums vergleichen, da die Elemente des anderen Teilbaums zu groß (klein) sind. Diesen Prozess können wir natürlich im Teilbaum wiederholen, so dass wir auf jedem Level im Baum mit höchstens einem Element vergleichen müssen. Wir brauchen also maximal so viele Vergleiche wie der Baum hoch ist.

Durch die Anforderung, dass alle Elemente links *kleiner* und *rechts* größer sind, haben wir ausgeschlossen, dass es in unserer Liste gleiche Einträge (zumindest gleich bzgl. ihres Index) gibt. Das ist für den Anfang eine sinnvolle Annahme, da es die Diskussion erheblich vereinfacht. Wir wollen den allgemeineren Fall hier auch nicht wirklich betrachten. Da bei vielen Anwendungen aber durchaus gleiche Werte in Listen vorkommen können, diskutieren wir am Ende des Abschnitts kurz, welche Möglichkeiten wir haben, damit umzugehen. Sie können das dann selbst in den Code unserer Beispiele integrieren.

Wir wollen das einmal an einem konkreten Beispiel betrachten. Der Baum in Abbildung 17.2 hat die Eigenschaft, dass links immer kleinere und rechts größere Einträge stehen (wobei wir als einzigen Dateneintrag den Index als ganze Zahl haben). Wenn wir jetzt feststellen wollen, ob die 31 Teil unserer Liste ist, vergleichen wir zuerst mit dem Element in der Wurzel, also 34. Unser Element ist kleiner, also suchen wir im linken Teilbaum weiter und vergleichen mit der 19. Diesmal ist unser Element größer, und wir suchen im rechten Teilbaum weiter. Wir vergleichen also mit 23, und da das kleiner ist als 31, suchen wir wieder rechts weiter. Dort finden wir die 31. Nach vier Vergleichen sind wir also am Ziel.

Wir können auch sehr einfach in eine solche Struktur neue Daten einfügen oder vorhandene Daten löschen und dabei die Vorgabe, dass links die kleineren Zahlen stehen und rechts die größeren, erhalten. Wenn wir im vorangegangenen Beispiel die 32 einfügen wollten, wären wir auf die gleiche Weise bis zur 31 im Baum abgestiegen. Dann hätten wir die 32 als rechtes Kind an die 31 angehängt. Der neue Baum sähe dann wie in Abbildung 17.3 aus.

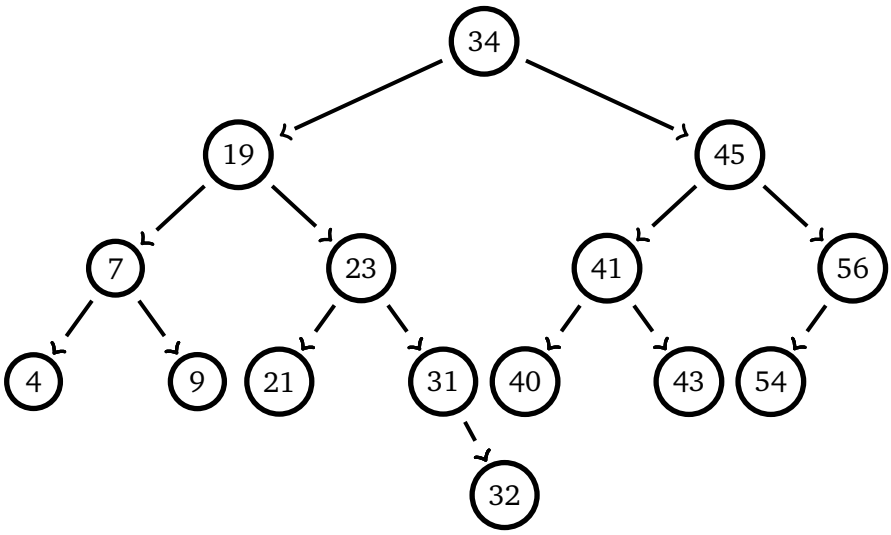


Abbildung 17.3: Der Eintrag 32 wurde hinzugefügt

Elemente aus dem Baum zu löschen ist etwas komplizierter, wenn wir die Eigenschaft, dass im linken Teilbaum nur kleinere und im rechten Teilbaum nur größere Zahlen stehen, erhalten wollen. Hier treten drei Fälle auf, je nachdem wie viele Kinder der Knoten hat, den wir löschen wollen. Zwei davon sind wiederum einfach. Wenn der Knoten keine Kinder hat, also ein Blatt ist, können wir ihn einfach löschen. Wenn er nur ein Kind hat, dann können wir den Knoten einfach löschen und sein Kind mitsamt seinem Teilbaum an seine Stelle setzen.

Wenn der zu löschende Knoten zwei Kinder hat, brauchen wir einen Schritt, der uns zu einem der beiden anderen Fälle zurückführt. Dafür suchen wir im linken Teilbaum das größte Element (indem wir vom linken Kind aus immer dem Zeiger zum rechten Kind folgen, bis dieser Zeiger auf `NULL` zeigt). Dieser Knoten hat höchstens noch ein linkes Kind, also höchstens eines. Nun kopieren wir seinen Wert in den zu löschenden Knoten. Da es der größte Knoten im linken Teilbaum war, ist damit unsere Baumeigenschaft an diesem Knoten erfüllt. Nun löschen wir den anderen Knoten. Da dieser höchstens ein Kind hat, ist das wiederum einfach. In unserem Beispiel die Knoten mit Wert 31 und 45 zu löschen führt zum Baum in Abbildung 17.4.

Damit haben wir schon alles theoretische Rüstzeug diskutiert, um eine Liste in einer Baumstruktur zu speichern und dynamisch Elemente hinzuzufügen und wieder zu löschen. Auch wie wir Elemente in dem Baum finden haben wir schon gelöst. Die Beispiele sollte auch nahegelegt haben, dass wir möglicherweise Listenoperationen effizienter auf Bäumen als auf linearen Listen ausführen können. Bevor wir das näher untersuchen, wollen wir uns erst eine Umsetzung in konkreten Code ansehen.

Für eine Implementierung in C müssen natürlich noch ein paar weitere Details beachtet werden, die in der theoretischen Betrachtung keine Rolle spielen. Wenn der zu löschende Knoten die Wurzel des Baums ist, sollte zum Beispiel auch darauf geachtet werden, dass der Zeiger auf die Wurzel am Ende wieder auf die neue Wurzel zeigt. Ebenso muss der Zeiger auf die Wurzel korrekt angepasst werden, wenn wir ein erstes Element in einen leeren Baum einfügen, oder wenn wir das letzte Element aus dem Baum löschen. Außerdem müssen wir natürlich auch entsprechend unserem Bedarf an den

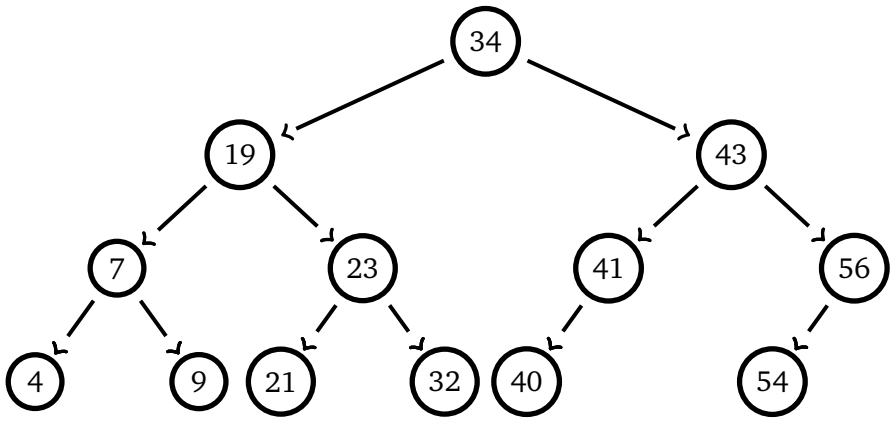


Abbildung 17.4: Löschen von 31 und 45

passenden Stellen Speicher reservieren, wenn ein neuer Knoten eingefügt wird, und wieder freigeben, wenn ein Knoten gelöscht wird.

Ein Beispiel für eine einfache Implementierung einer Baumstruktur sehen wir im nachfolgenden Code. Die Bibliothek enthält noch drei weitere Funktionen `find`, `find_min` unf `print_binary_tree_inorder`, die wir anschließend betrachten. Der Code ist wieder aufgeteilt in einen Header, der die Struktur und die öffentlichen Funktionen deklariert, und eine Codedatei, die die passenden Definitionen dazu enthält.

Programm 17.1: `kapitel_17/binary_tree.h`

```
#ifndef BINARY_TREE_H
#define BINARY_TREE_H

struct node {
    struct node * left_child;
    struct node * right_child;
    int data;
} ;

void free_bst(struct node*);

void insert(struct node**, int);
void delete(struct node**, int);

int          find_min(struct node*);
struct node* find     (struct node*, int);

void print_binary_tree_inorder(struct node*);

#endif
```

Programm 17.1:kapitel_17/binary_tree.c

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

#include "binary_tree.h"

void free_bst(struct node* node) {
```

```
7     if( node != NULL ) {
8       free_bst( node->left_child );
9       free_bst( node->right_child );
10      free( node );
11    }
12  }
13
14  struct node *newNode(int data) {
15     struct node *temp =  (struct node *)malloc(sizeof(struct node));
16     if( temp == NULL ) {
17        fprintf (stderr, "kein Speicher mehr\n");
18        exit(1);
19     }
20     temp->data = data;
21     temp->left_child = temp->right_child = NULL;
22     return temp;
23  }
24
25  void insert(struct node** node_ptr, int data) {
26      if (*node_ptr == NULL) {
27        struct node* newnode = newNode(data);
28        *node_ptr = newnode;
29        return;
30     }
31
32      if ((*node_ptr)->data >= data )
33          insert(&(*node_ptr)->left_child, data);
34      else if ((*node_ptr)->data < data )
35          insert(&(*node_ptr)->right_child, data);
36  }
37
38  void delete(struct node** node_ptr, int data) {
39     if (*node_ptr == NULL)
40        return;
41
42     struct node * node = *node_ptr;
43     if (data < node->data)
44        delete(&node->left_child, data);
45     else if (data > node->data)
46        delete(&node->right_child, data);
47     else {
48        if (node->left_child == NULL) {
49           struct node *temp = node->right_child;
50           free(node);
51           *node_ptr = temp;
52           return;
53        }  else if (node->right_child == NULL) {
54           struct node *temp = node->left_child;
55           free(node);
56           *node_ptr = temp;
57           return;
58        }
59
60        struct node* temp = find_min(node->right_child);
61
62        node->data = temp->data;
63        delete(&node->right_child, temp->data);
64     }
```

```
65     return;
66  }
67
68  int find_min(struct node* node) {
69     if ( node == NULL ) {
70        return -1;
71     }
72     while (node->left_child != NULL)
73        node = node->left_child;
74
75      return node->data;
76  }
77
78  struct node* find(struct node* node, int data) {
79     if (node == NULL) {
80        return NULL;
81     }
82
83     if ( node->data > data ) {
84        return find(node->left_child, data);
85     }  else if ( node->data < data ) {
86        return find(node->right_child, data);
87     }
88     return node;
89  }
90
91  void print_binary_tree_inorder(struct node* node) {
92    if (node != NULL) {
93      print_binary_tree_inorder(node->left_child);
94      printf("%d ", node->data);
95      print_binary_tree_inorder(node->right_child);
96    }
97  }
```

Wir wollen unsere Daten natürlich nicht nur speichern, sondern auch darauf zugreifen können. Wir wollen also insbesondere zum Beispiel feststellen, ob ein bestimmten Element Teil unserer Liste ist, oder wir wollen das kleinste oder größte Element finden.

Das kleinste Element lässt sich sehr leicht finden. Da von jedem Knoten aus alle Elemente, die kleiner sind als das aktuelle, im linken Teilbaum stehen, können wir einfach so lange im Baum immer zum linken Kind wechseln, bis wir an einem Knoten sind, der kein linkes Kind mehr hat. Der Eintrag in diesem Knoten ist das kleinste Element. Das ist in unserem Beispiel in der Funktion `find_min` realisiert. Auf ähnliche Weise können wir natürlich auch das größte Element finden, indem wir statt zum linken immer zum rechten Kind gehen.

Den Knoten eines konkreten Elements können wir ebenfalls schnell finden, indem wir je nach Vergleich mit dem aktuellen Knoten im Baum zum linken oder rechten Teilbaum übergehen. Das ist im Beispiel in der Funktion `find` enthalten. Diese Funktion gibt **`NULL`** zurück, wenn der Baum leer ist oder das Element nicht im Baum enthalten ist.

Eine sortierte Ausgabe aller Elemente können wir rekursiv ebenfalls sehr einfach realisieren. Nach Konstruktion unseres Baums stehen für jeden Knoten alle kleineren Elemente im linken Teilbaum, und alle größeren im rechten. Bei der Ausgabe müssen wir also zuerst alle Elemente des linken Teilbaums in aufsteigender Reihenfolge ausgeben, dann das Element des Knotens, in dem wir stehen, und dann alle Elemente des rechten

Teilbaums. Einen Teilbaum sortiert auszugeben ist einfach, wenn er nur ein Element enthält. Rekursiv können wir also entweder das eine Element ausgeben, wenn der Teilbaum nur eines hat, oder wir rufen unsere Ausgabefunktion mit dem Kind als neuer Wurzel erneut auf. Das macht die Funktion `print_binary_tree_inorder`.

Ein Testprogramm, das unsere Baumstruktur demonstriert, könnte zum Beispiel wie das folgende Programm aussehen.

Programm 17.2: `kapitel_17/binary_tree_main.c`

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

#include "binary_tree.h"

int main(int argc, char** argv) {

    struct node * root = NULL;

    printf("-- Alle Elemente einfuegen --\n");

    for ( int i = 1; i < argc; ++i ) {
        insert(&root,atoi(argv[i]));
    }

    printf("Liste in aufsteigender Reihenfolge: \n");
    print_binary_tree_inorder(root);
    printf("\n");

    printf("\n-- Jedes zweite Element wieder loeschen --\n");

    for ( int i = 1; i < argc; i += 2 ) {
        delete(&root,atoi(argv[i]));
    }

    printf("Liste in aufsteigender Reihenfolge: \n");
    print_binary_tree_inorder(root);
    printf("\n");

    return 0;
}
```

Um auch für unser Programm eine anschauliche Darstellung zu bekommen, machen wir hier einen kleinen Exkurs und überlegen uns eine kleine Bibliothek, mit der wir Bäume auf dem Bildschirm ausgeben können. Das ist etwas schwieriger zu lösen, da wir auf dem Terminal nur zeilenweise ausgeben können. Wenn wir uns aber darauf einigen, den Baum statt von oben nach unten auf dem Bildschirm von links nach rechts wachsen zu lassen, können wir uns wieder mit einem rekursiven Ansatz behelfen. Die folgende Bibliothek gibt ein Beispiel, wie man das lösen kann. Die Darstellung lässt sich sicherlich verbessern, und Sie sind eingeladen, das auch zu versuchen[2].

Programm 17.3: `kapitel_17/binary_tree_print.h`

```c
#ifndef BINARY_TREE_PRINT_H
#define BINARY_TREE_PRINT_H
```

[2] Eine weitere Möglichkeit, die Restriktion zeilenweiser Ausgabe zu umgehen, wäre die Verwendung der Bibliothek `ncurses`, die es erlaubt, direkt in eine konkrete Zeile und Spalte des Terminals zu springen.

```
3
4  #include "binary_tree.h"
5
6  void print_binary_tree(struct node*);
7
8  #endif
```

Programm 17.3:kapitel_17/binary_tree_print.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  #include "binary_tree_print.h"
5  #include "binary_tree.h"
6
7  struct segment {
8    struct segment *previous;
9    char * seg;
10 } ;
11
12 void print_line_segment(struct segment * segment) {
13   if ( segment == NULL )
14     return;
15   print_line_segment(segment->previous);
16   printf("%s", segment->seg);
17 }
18
19 void print_tree(struct node *node, struct segment * previous, int is_right ) {
20   if (node == NULL) {
21     return;
22   }
23
24   struct segment * segment = malloc(sizeof(struct segment));
25   segment->previous = previous;
26   char * previous_seg = "    ";
27   segment->seg        = "    ";
28   print_tree(node->right_child, segment, 1);
29
30   if ( previous == NULL ) {
31     segment->seg = "---";
32   }  else if (is_right) {
33     segment->seg = "/--";
34     previous_seg = "   |";
35   }  else {
36     segment->seg = "\\--";
37     previous->seg = previous_seg;
38   }
39
40   print_line_segment(segment);
41   printf("%d\n", node->data);
42
43   if ( previous != NULL ) {
44     previous->seg = previous_seg;
45   }
46   segment->seg = "   |";
47
48   print_tree(node->left_child, segment, 0);
49   if ( previous == NULL ) {
```

```
50      printf("\n");
51    }
52
53    free(segment);
54  }
55
56  void print_binary_tree(struct node* root) {
57    if ( root == NULL) {
58      return;
59    }
60    print_tree(root, NULL, 0);
61  }
```

Wir können auch hier ein kleines Testprogramm angeben.

Programm 17.4: `kapitel_17/binary_tree_print_main.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "binary_tree.h"
5  #include "binary_tree_print.h"
6
7  int main(int argc, char** argv) {
8
9      struct node * root = NULL;
10
11     for ( int i = 1; i < argc; ++i ) {
12         insert(&root,atoi(argv[i]));
13     }
14
15     print_binary_tree(root);
16
17     return 0;
18 }
```

Damit sollten wir folgende Ausgabe erhalten.

```
$ ./binary_tree 10 5 15 13 12 19 23 3 4 1 2
            /--23
         /--19
      /--15
     |    \--13
     |        \--12
  ---10
      \--5
         |    /--4
          \--3
             |    /--2
              \--1
```

Nachdem wir mit den Bäumen eine deutlich kompliziertere Datenstruktur aufgebaut haben als unsere verketteten Listen aus Kapitel 12 sollten wir auch noch untersuchen, ob wir dadurch einen Nutzen haben. Dafür untersuchen wir die Laufzeiten der verschiedenen Operationen. Neben der Anzahl n der Elemente in unserer Liste brauchen wir dafür die Höhe h des Baums. Um ein Element einzufügen, müssen wir zum Auffinden der richtigen Stelle in jedem Level des Baums höchstens ein Element ansehen, um zu entscheiden, ob wir im linken oder rechten Teilbaum weitermachen. Für jeden solchen

Vergleich, und für das Einfügen am Ende brauchen wir eine konstante, von der Anzahl der Elemente oder der Höhe unabhängige Anzahl von Operationen. Daher hat Einfügen eine Laufzeit von $\mathcal{O}(h)$. Das gleiche gilt im Prinzip für das Löschen, wobei wir hier in dem Fall, dass das zu löschende Element zwei Kinder hat, zweimal im Baum suchen müssen. Das ändert aber nur den Faktor c in der Definition der Menge $\mathcal{O}(.)$, so dass auch Löschen eine Laufzeit von $\mathcal{O}(h)$ hat. Suchen ist dann ebenfalls in der Laufzeit $\mathcal{O}(h)$ fertig. Die aufsteigende Ausgabe aller Elemente muss jedes Element im Baum genau einmal besuchen, daher hat die Ausgabe eine Laufzeit von $\mathcal{O}(n)$.

Vielleicht bemerken Sie hier, dass es auf den ersten Blick so aussieht, als hätten wir mit der Speicherung als Baumstruktur eine Liste schneller sortiert als in Zeit $\mathcal{O}(n \log n)$, Das ist die (worst-case) Laufzeit, von der wir in Kapitel 11 angegeben haben, dass es keinen auf paarweisen Vergleichen beruhenden Algorithmus geben kann, der diese unterschreitet. Da wir hier eine sortierte Liste durch einfaches Ablaufen des Baums erhalten, können wir hier die Liste in sortierter Form in $\mathcal{O}(n)$ ausgeben. Der Widerspruch löst sich, wenn wir beachten, dass wir, anders als beim Füllen der Liste, die wir an `MergeSort` übergeben, auch beim Einfügen der Elemente in den Baum eine nichtkonstante Laufzeit $\mathcal{O}(h)$ haben. Damit unsere bisherige Aussage richtig ist, muss also die Höhe eines Baums sich mindestens ungefähr wie $\log n$ verhalten. Um das explizit zu untersuchen, müssen wir uns also überlegen, wie die Höhe h und die Anzahl der Elemente n zusammenhängen.

Ein binärer Baum hat auf Level 0 nur seine Wurzel, und auf jedem Level k höchstens doppelt so viele Knoten wie auf Level $k-1$, für alle $k \geq 1$. Wir können in einem Baum der Höhe h also höchstens

$$1 + 2 + 2^2 + 2^3 + \cdots + 2^h = 2^{h+1} - 1$$

Knoten haben (da nicht jeder Knoten zwei Kinder haben muss, selbst wenn er kein Blatt des Baums ist, können es natürlich weniger sein). Die Höhe eines Baums mit n Knoten ist also mindestens $\lfloor \log_2 n \rfloor$. Damit erhalten wir für die Sortierung durch Einfügen in unsere Baumstruktur und anschließendes Auslesen eine Laufzeit von $\mathcal{O}(n \log n)$, allerdings nur, wenn wir beim Einfügen alle Level vollständig füllen. Es ist leicht zu sehen, dass das in der Regel nicht der Fall sein wird. Im schlechtesten Fall ist der Baum einfach nur eine Liste, und seine Höhe ist $h = n$. Wir können das zum Beispiel mit unserem Testprogramm von oben ausprobieren. Wenn wir es mit der Folge $4, 2, 6, 1, 3, 5, 7$ aufrufen, erhalten wir einen vollständig gefüllten Baum der Höhe 2.

```
$ ./binary_tree 4 2 6 1 3 5 7
        /--7
    /--6
   |    \--5
---4
   |    /--3
    \--2
        \--1
```

Wenn wir die Zahlen stattdessen in aufsteigender Reihenfolge vorgeben, dann bekommen wir eine lineare Liste der Höhe 6.

```
$ ./binary_tree 1 2 3 4 5 6 7
                        /--7
                    /--6
```

```
                /--5
            /--4
        /--3
    /--2
---1
```

Wir sind also noch nicht so richtig erfolgreich mit unserem Ziel, eine effiziente Struktur zur Speicherung einer Liste anzugeben. Bisher sind wir nur dann schneller geworden, wenn sich beim Speichern zufällig ein ausgeglichener Baum ergibt, bei dem außer den Blättern möglichst viele Knoten zwei Kinder haben. Das Beispiel zeigt allerdings auch, dass es nicht an den Daten selbst liegt, wenn der Baum eine unnötig große Höhe hat, sondern daran, wie wir die Daten in den Baum eingefügt haben. Im nächsten Abschnitt wollen wir uns daher überlegen, wie wir beim Einfügen und Löschen den Baum ausgleichen können, so dass wir dem Ziel von zwei Kindern pro innerem Knoten zumindest nahe kommen.

17.2 Balancierte Bäume

Im letzten Abschnitt haben wir gesehen, dass Bäume uns nur dann einen Vorteil bieten, wenn die Höhe aller Blätter möglichst klein ist. Wir wollen also beim Einfügen in den Baum und Löschen aus dem Baum möglichst so vorgehen, dass der Baum ausgeglichen ist.

Ein binären Baum hat auf Level k höchstens 2^k Elemente. Ein Baum mit n Elementen hat also mindestens Höhe $h := \lceil \log\, n \rceil$. Diese Höhe erreichen wir, wenn alle Level bis Höhe $h-1$ vollständig gefüllt sind und die restlichen Elemente auf dem letzten Level liegen. Man kann sich überlegen, dass wir tatsächlich unseren binären Baum auch mit der Annahme, dass links von jedem Knoten kleinere und rechts größere Elemente liegen, so umsortieren kann, dass er diese minimale Höhe hat (und sogar, dass die Elemente auf dem letzten Level möglichst weit links liegen).

Allerdings müssen wir eine solche Umsortierung nach jedem Einfügen oder Löschen von Elementen erneut durchführen. Dabei wollen wir natürlich die schnelle Laufzeit von $\mathcal{O}(log\, n)$ für Löschen und Einfügen nicht aufgeben, der Prozess des Umsortierens muss also schnell genug ablaufen. Für eine vollständige Umsortierung zu einem Baum minimaler Höhe ist das im Allgemeinen nicht möglich.

Daher wurden mehrere verschiedene Ideen entwickelt, wie wir einen binären Baum effizient so umsortieren können, dass es zumindest näherungsweise ausgeglichen ist und wir die gute Laufzeit von $\mathcal{O}(\log\, n)$ erhalten. Das führt uns zu den *balancierten Bäumen*. Wir wollen uns hier eine konkrete Möglichkeit ansehen, wie wir einen Baum effizient ausgleichen können, die sogenannten *AVL-Bäume*. Diese Datenstruktur wurde 1962 von den russischen Mathematikern Georgy Adelson-Velsky und Evgenii Landis (deren Nachnamen den Namen der Bäume erklären) entwickelt.

Wir brauchen eine neue Information, die wir einem Knoten zuweisen können, Die *Ausgeglichenheit* (oder *Balanciertheit*) $\mathrm{balance}(v)$ eines Knotens v im Baum ist die Differenz zwischen der Höhe seines linken Teilbaums und der Höhe seines rechten Teilbaums. Ein binärer Baum heißt dann *balanciert*, wenn $\mathrm{balance}(v) \in \{0, \pm 1\}$ (die *AVL-Bedingung*) liegt für alle Knoten v des Baums. Ein Beispiel eines solchen Baums ist in Abbildung 17.5.

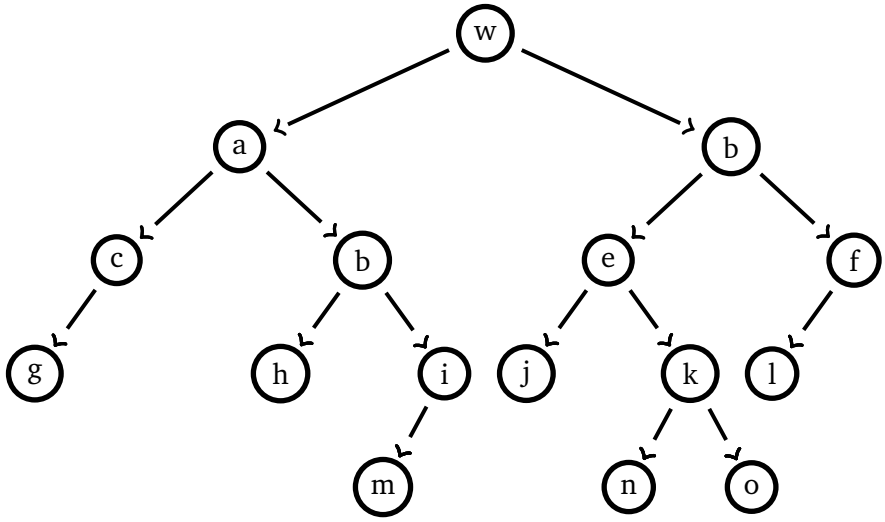


Abbildung 17.5: Ein balancierter Baum.

Sie sehen vom Bild, dass die Höhen der Teilbäume über den gesamten Baum variieren können, und auch kleinere Zwischenlevel brauchen nicht gefüllt zu sein. Da die Ausgeglichenheit aber auch an der Wurzel erfüllt sein muss, können die Höhen der Blätter sich trotzdem nicht mehr allzu sehr unterscheiden. Das reicht, um für die Prozesses des Einfügens und Löschens in den Baum eine Laufzeit von $\mathcal{O}(\log\, n)$ zu behalten.

Wir müssen uns überlegen, ob wir durch diese Relaxierung der Bedingung eines vollständig ausgeglichenen Baums tatsächlich nach jedem Einfügen und Löschen den Baum auch in der Zeit $\mathcal{O}(\log\, n)$ wieder so umsortieren können, dass die *AVL-Bedingung* wieder erfüllt ist. Bei manchen Operationen ist das automatisch der Fall, zum Beispiel wenn wir dem Knoten i oder h ein weiteres rechtes Kind anhängen. Wenn wir allerdings an g, m oder n ein weiteres Kind anhängen, ist die Bedingung verletzt (an den Knoten c, i bzw. k). Betrachten wir dazu die Situation in Abbildung 17.6. Hier ist der linke Teilbaum höher als der rechte, wobei der gezeigte Baum auch ein Teilbaum eines größeren Baums sein könnte. Wenn wir nun einen neuen Knoten einfügen, der an ein Blatt angehängt

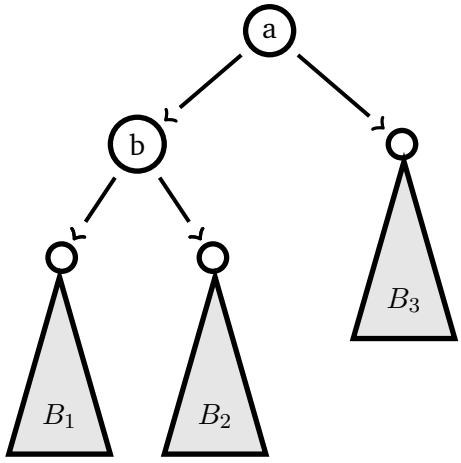


Abbildung 17.6: Ein Baum, in dem der linke Teilbaum höher ist.

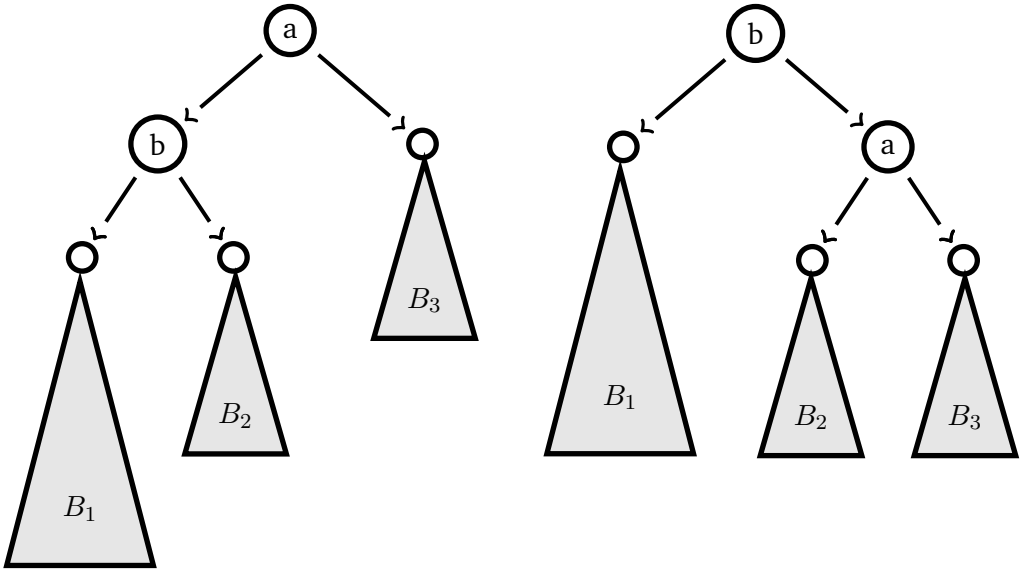


Abbildung 17.7: Rechtsrotation: Im Teilbaum mit Wurzel a drehen wir nach rechts, so dass b an die Spitze kommt, und hängen dann den Teilbaum $B2$ um.

wird, das im Teilbaum $B1$ auf der linken Seite auf der untersten Stufe liegt, steigt die Höhe um 1 und unsere *AVL-Bedingung* ist nicht mehr erfüllt. Schematisch sieht der Baum dann aus wie in Abbildung 17.7 auf der linken Seite.

Durch eine Umordnung des Baums können wir die Baumhöhen wieder anpassen. Dazu machen wir die folgenden drei Schritte, die zu dem Baum in Abbildung 17.7 auf der rechten Seite führen:

- Wir machen b statt a zum Kind des Elternknotens von a.
- Wir machen a zum rechten Kind von b.
- Wir hängen den rechten Teilbaum B_2 von b als linken Teilbaum an a.

Sie können sich leicht überlegen, dass damit die Baumhöhen wieder ausgeglichen sind. Wir müssen uns überlegen, dass auch im neuen Baum noch alle Knoten eines linken Teilbaums eines Knotens einen kleineren und alle Knoten eines rechten Teilbaums des Knotens einen größeren Wert haben.

Nach Konstruktion sind b und alle Werte in den Teilbäumen B_1 und B_2 kleiner als der Wert in a, sowie alle Werte in B_3 größer als a. Damit sind auch nach der Umsortierung alle Werte im linken Teilbaum unter a kleiner als der Wert in a, und alle Werte im rechten Teilbaum größer. Da zudem der Wert in a größer ist als b (da b ursprünglich das linke Kind von a ist), sind auch alle Werte im neuen rechten Teilbaum unter b größer als der Wert in b. Der Teilbaum B_1 hängt vorher wie nachher rechts von b, so dass sich hier nichts geändert hat. Damit haben wir auch nach der Umsortierung einen gültigen Baum, in dem links die kleineren und rechts die größeren Werte stehen.

Diese Umsortierung im Baum bezeichnen wir als als *Rechtsrotation* (*right rotation*) an a.

Auf die gleiche Weise können wir vorgehen, wenn wir in einem Baum wie in Abbildung 17.8 einen Knoten an einem Blatt maximaler Höhe im Teilbaum B_3 anhängen und auf diese Weise die Gesamthöhe von B_3 um 1 steigt. Dann haben wir die Situation

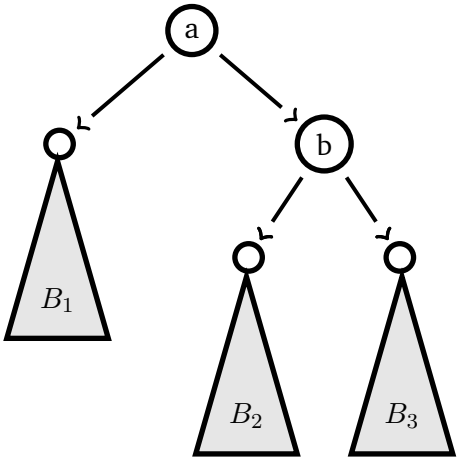


Abbildung 17.8: Ein Baum, in dem der rechte Teilbaum höher ist.

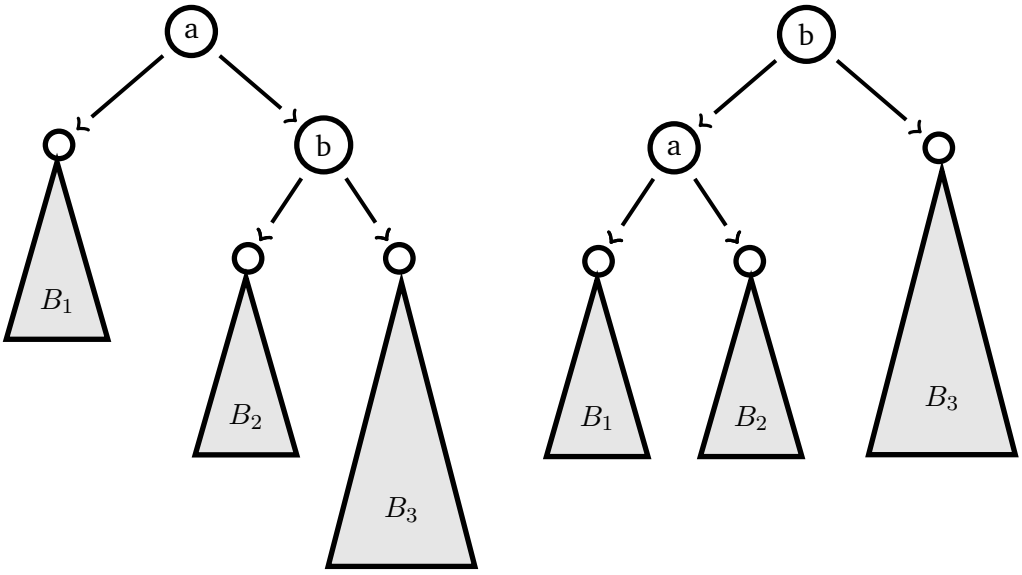


Abbildung 17.9: Linksrotation: Im Teilbaum mit Wurzel a drehen wir nach links, so dass b an die Spitze kommt, und hängen dann den Teilbaum $B2$ um.

wie in Abbildung 17.9, und wir können b zur neuen Wurzel des Teilbaums machen und den Teilbaum B_2 an die rechte Seite von a hängen. Damit erhalten wir einen Baum wie auf der rechten Seite von Abbildung 17.9.

Analog zu oben können wir uns überlegen, dass auch in diesem Baum immer links die kleineren und rechts die größeren Werte stehen. Diese Umsortierung im Baum bezeichnen wir als als *Linksrotation* (*left rotation*) an a.

Etwas schwieriger wird die Situation, wenn wir in der Situation von Abbildung 17.6 einen weiteren Knoten so an ein Blatt des Teilbaums B_2 hängen, dass sich dessen Höhe um 1 erhöht. Wir haben dann eine Situation wie in Abbildung 17.10 oder Abbildung 17.11. Wir können beide Fälle auf die gleiche Weise neu anordnen, um wieder einen ausgeglichenen Baum zu erhalten. Dafür machen wir zuerst eine *Links*rotation um den Knoten b. Damit haben wir diesen Fall auf einen Baum wie in der linken Seite von

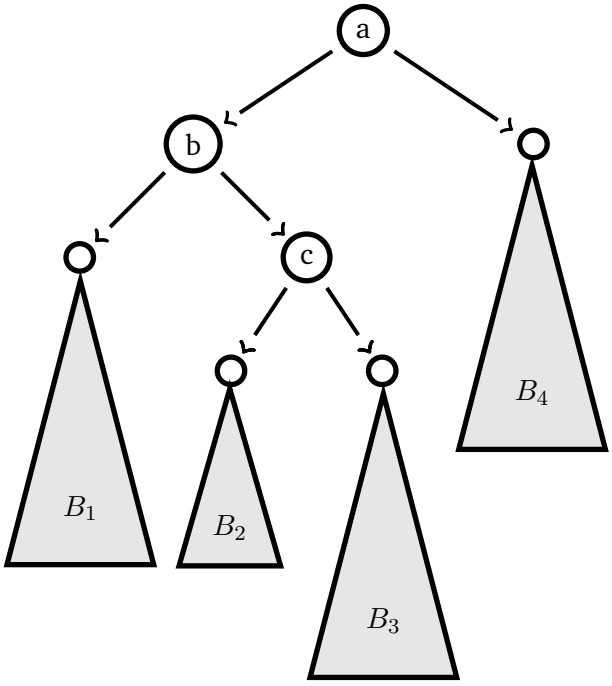


Abbildung 17.10: Ein Baum, in dem der rechte Teilbaum des linken Teilbaums höher ist, wobei sich die Höhe von B_3 (als Teil des Baums B_2 aus Abbildung 17.6) erhöht hat.

Abbildung Abbildung 17.7 zurückgeführt, und nach einer *Rechts*rotation an a haben wir wieder einen ausgeglichenen Baum. Sie können sich überlegen, dass es dafür egal ist, ob sich am Anfang die Höhe des Baums B_2 (der in der ersten Linksrotation umgehängt wird) oder die des Baums B_3 erhöht hat.

Damit sind wir auch in diesem Fall nach zwei Rotationen wieder bei einem ausgeglichenen Baum angekommen. Diese Operation wird als *Linksrechtsrotation* (*left right rotation*) bezeichnet.

Entsprechend brauchen wir natürlich noch eine *Rechtslinksrotation*, wenn sich in Abbildung 17.8 die Höhe des Baums B_2 ändert. In allen Fällen sind wir aber mit einer konstanten Anzahl (also einer von der Baumhöhe oder der Anzahl der Elemente unabhängigen Zahl von Operationen) ausgekommen um nach Anhängen eines neuen Knotens wieder zu einem *AVL*-Baum zurückzukommen.

In den Abbildungen 17.12 bis 17.14 ist ein Beispiel für eine Linksrechtsrotation. An den schwarzen Baum in Abbildung 17.12 fügen wir den roten Knoten mit Wert 23 ein. Damit ist der Baum an der Wurzel nicht mehr ausgeglichen. Da der Knoten auf der linken Seite des rechten Teilbaums von 19 eingefügt wurde, müssen wir zuerst eine Linksrotation an 19 machen, um den Baum anschließend mit einer Rechtsrotation an der Wurzel wieder ausgleichen zu können. Das Ergebnis der Rechtsrotation an 19 sehen Sie in Abbildung 17.13, während der vollständig ausgeglichene Baum in Abbildung 17.14 ist.

Wir wollen diese Operationen nun in unser Programm übernehmen. Einfügen und Löschen geht dabei erst einmal genauso wie schon im letzten Abschnitt über binäre Bäume. Das können Sie an den entsprechenden Funktionen in Programm 17.5 sehen.

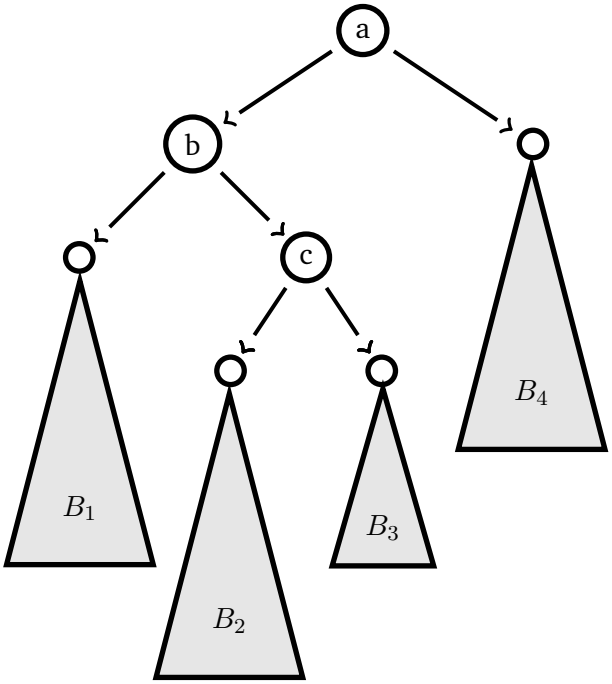


Abbildung 17.11: Ein Baum, in dem der rechte Teilbaum des linken Teilbaums höher ist, wobei sich die Höhe von B_3 erhöht hat.

Wir betrachten vorerst nur die Funktion `insert` zum Einfügen eines Knotens in den Zeilen Zeile 78 bis Zeile 106.

Um in jedem Knoten feststellen zu können, ob der Baum an dieser Stelle ausgeglichen ist, erweitern wir unseren `struct node` und speichern dort auch noch die Höhe des Teilbaums an diesem Knoten in der Variable `height` in Zeile 18 in der Datei `avl_tree.h`. Diese Information müssen wir dann natürlich bei jeder Veränderung des Baums aktualisieren. Wir werden sehen, dass wir das leicht innerhalb der Funktionen zum Einfügen und Löschen erledigen können, da sich die Höhe nur für Knoten ändern kann, die wir in diesen Funktionen sowieso verändern.

Die Balance eines Knotens ergibt sich dann aus der Differenz der Höhen seines rechten und linken Teilbaums. Da wir diese Rechnung sehr oft brauchen, ist es sinnvoll, das in eine eigene Funktion auszulagern. Wir erhalten die Höhe eines Knotens mit der Funktion `height` in Zeile 18 und die Balance mit der Funktion `balance` in Zeile 25.

In den Zeilen 79-83 behandeln wir wieder den Fall des Einfügens in eine leere Liste. Hier brauchen wir keine Rotationen betrachten, da ein Baum mit nur einem Element immer ausgeglichen ist. Nun kommen, wie schon bei den binären Bäumen, in den Zeilen 85-103 die beiden Fälle, dass wir im linken oder rechten Teilbaum einfügen müssen. Wir betrachten nur den ersten Fall, der zweite ist dann symmetrisch, wir müssen nur jeweils Rechts- und Linksdrehungen vertauschen.

Wie zuvor steigen wir erst rekursiv bis zu dem Blatt im Baum ab, an dem wir einfügen. Das passiert mit dem Aufruf von `insert` in Zeile 86. Erst wenn wir wieder aus der Rekursion zurückkommen, müssen wir an jedem Knoten, an dem wir vorbeikommen, testen, ob der Baum an diesem Knoten noch ausgeglichen ist.

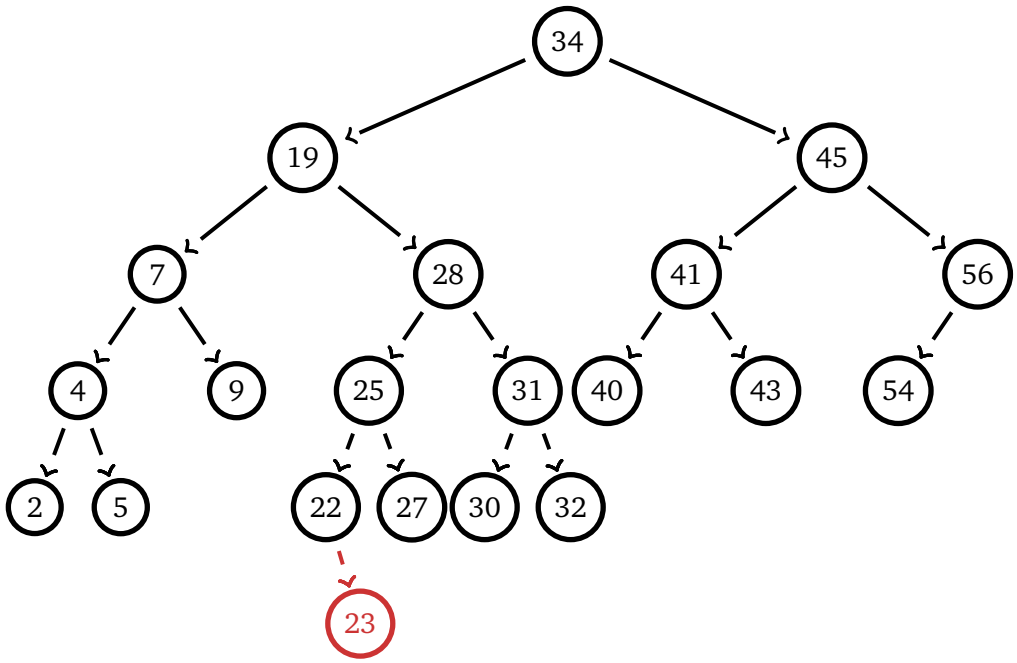


Abbildung 17.12: Dem schwarzen Baum wird der rote Knoten wir Wert 23 hinzugefügt. Damit ist der Baum nicht mehr ausgeglichen. Wir brauchen eine Linksrotation an dem Knoten mit Wert 19 und dann eine Rechtsrotation an der Wurzel.

Dafür fragen wir in Zeile 87 nach der Balance des Knotens. Durch Einfügen im linken Teilbaum kann sie nur größer werden, und da wir nur einen Knoten einfügen, kann sie auch nur um 1 steigen. Daher reicht es hier abzufragen, ob die Balance 2 ist. In diesem Fall brauchen wir auf jeden Fall eine Rechtsrotation an diesem Knoten. Wir müssen nur noch entscheiden, ob wir vorher noch eine Linksrotation am linken Kind machen müssen. Das können wir einfach durch einen Vergleich des eingefügten Werts mit dem Wert am linken Kind machen. Das passiert in Zeile 88. Die Rotation selbst haben wir nun, da wir sie an verschiedenen Stellen brauchen, in vier Funktionen `left_rotation`, `right_rotation`, `left_right_rotation` und `right_left_rotation` in Zeile 36, Zeile 46, Zeile 62 und Zeile 57 ausgelagert. Den Funktionen wird jeweils nur ein Teilbaum übergeben, die Rotation findet innerhalb der Funktion also immer an der Wurzel statt.

Die beiden letzten Funktionen rufen dabei nur nacheinander die beiden ersten auf (einmal auf dem Kind, dann auf der Wurzel des übergebenen Teilbaums). Die eigentliche Umordnung passiert also in den Funktionen `right_rotation` und `left_rotation`.

In diesen Funktionen brauchen wir nur zwei Zeiger umhängen. Da wir aber beim Verändern des ersten Zeigers den Zugriff auf seinen bisherigen Teilbaum verlieren, müssen wir diesen in einer temporären Variable zwischenspeichern. Am Ende der Funktion bestimmen wir noch die neuen Höhen der Teilbäume und aktualisieren die Variable `height` im Knoten.

Auch wenn wir in dieser Funktion rekursiv die Balance an jedem Knoten testen, an dem wir auf dem Weg zu dem Blatt, an das der neue Knoten angehängt wird, vorbeikommen, kann man sich leicht überlegen, dass wir trotzdem nur höchstens einmal tatsächlich auch mit unseren Rotationen umsortieren werden. Nach einmaliger Umsortierung wird

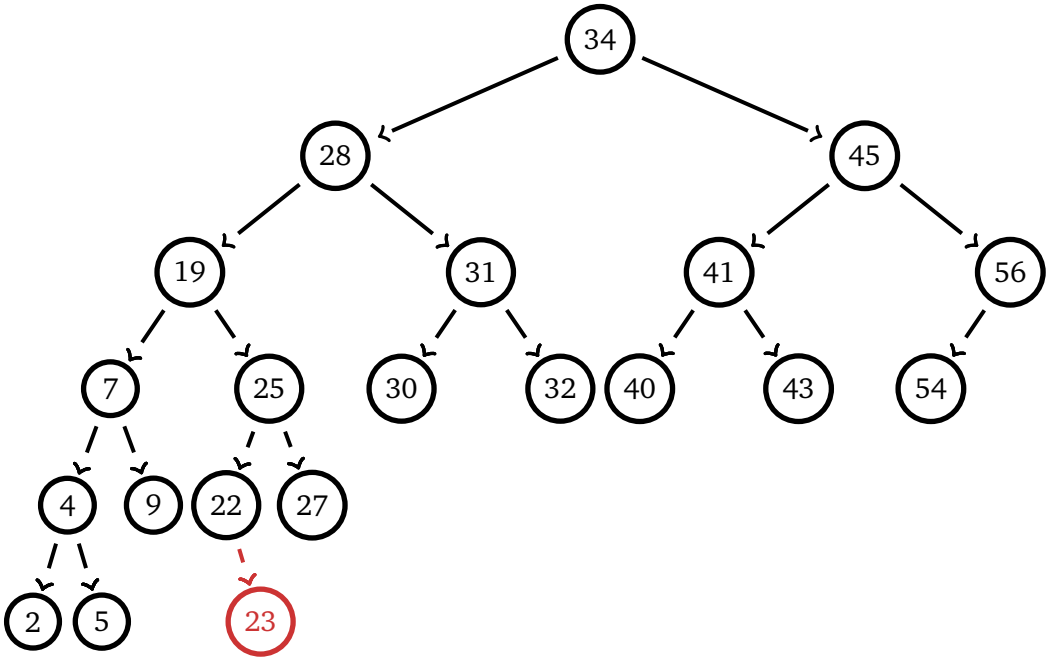


Abbildung 17.13: Der Baum aus Abbildung 17.12 nach einer Linksdrehung an 19.

der Baum an allen weiteren Knoten auf dem Weg zurück zur Wurzel wieder ausgeglichen sein.

Programm 17.5: `kapitel_17/avl_tree.h`

```
#ifndef AVL_TREE_IMPL_H
#define AVL_TREE_IMPL_H

struct node {
  struct node * left_child;
  struct node * right_child;
  int data;
  int height;
} ;

void free_tree        (struct node*);
void insert           (struct node **, int);
void delete           (struct node **, int);

struct node * find    (struct node *, int);
int           find_min(struct node *);

#endif
```

Programm 17.5:kapitel_17/avl_tree.c

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

#include "avl_tree.h"

struct node *newNode(int data) {
   struct node *temp =  (struct node *)malloc(sizeof(struct node));
```

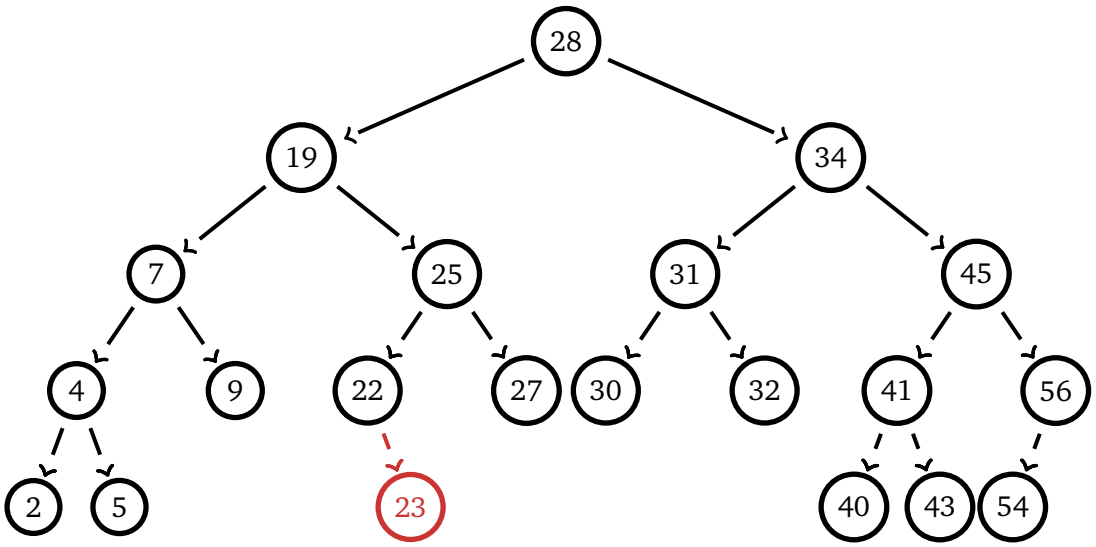


Abbildung 17.14: Der Baum aus Abbildung 17.13 nach einer Rechtsdrehung an der Wurzel. Damit ist der Baum aus Abbildung 17.12 wieder ausgeglichen.

```
8     if( temp == NULL ) {
9        fprintf (stderr, "kein Speicher mehr\n");
10       exit(1);
11    }
12    temp->data = data;
13    temp->height = 0;
14    temp->left_child = temp->right_child = NULL;
15    return temp;
16 }
17
18 static int height( struct node * node ) {
19    if( node == NULL ) {
20    return -1;
21    }
22    return node->height;
23 }
24
25 static int balance(struct node *node) {
26    if (node == NULL) {
27    return 0;
28    }
29    return height(node->left_child) - height(node->right_child);
30 }
31
32 static int max( int l, int r) {
33    return l > r ? l: r;
34 }
35
36 static void right_rotation( struct node ** root_ptr ) {
37    struct node* left_child = (*root_ptr)->left_child;
38    (*root_ptr)->left_child = left_child->right_child;
39    left_child->right_child = *root_ptr;
40
41    (*root_ptr)->height = max( height( (*root_ptr)->left_child ), height( (*root_ptr)->
      right_child ) ) + 1;
```

```
42      left_child->height = max( height( left_child->left_child ), (*root_ptr)->height ) +
          1;
43      *root_ptr = left_child;
44  }
45
46  static void left_rotation( struct node ** root_ptr ) {
47      struct node* right_child = (*root_ptr)->right_child;
48      (*root_ptr)->right_child = right_child->left_child;
49      right_child->left_child = *root_ptr;
50
51      (*root_ptr)->height = max( height( (*root_ptr)->left_child ), height( (*root_ptr)->
          right_child ) ) + 1;
52      right_child->height = max( height( right_child->right_child ), (*root_ptr)->height )
          + 1;
53
54      *root_ptr = right_child;
55  }
56
57  static void right_left_rotation( struct node ** root_ptr ) {
58      right_rotation( &(*root_ptr)->right_child );
59      left_rotation( root_ptr );
60  }
61
62  static void left_right_rotation( struct node ** root_ptr ) {
63      left_rotation( &(*root_ptr)->left_child );
64      right_rotation( root_ptr );
65  }
66
67
68  /* Public Functions */
69
70  void free_tree(struct node * node ) {
71      if( node != NULL ) {
72      free_tree( node->left_child );
73      free_tree( node->right_child );
74      free( node );
75      }
76  }
77
78  void insert(struct node ** node_ptr, int data ) {
79      if (*node_ptr == NULL) {
80          struct node* newnode = newNode(data);
81          *node_ptr = newnode;
82          return;
83      }
84
85      if( data < (*node_ptr)->data ) {
86          insert(&(*node_ptr)->left_child, data );
87          if( balance( *node_ptr ) == 2 ) {
88              if( data < (*node_ptr)->left_child->data ) {
89                  right_rotation( node_ptr );
90              }  else {
91                  left_right_rotation( node_ptr );
92              }
93          }
94      }  else if( data > (*node_ptr)->data ) {
95          insert(&(*node_ptr)->right_child, data );
96          if( balance( *node_ptr ) == -2 ) {
```

```
97              if( data > (*node_ptr)->right_child->data ) {
98                  left_rotation( node_ptr );
99              }  else {
100                 right_left_rotation( node_ptr );
101             }
102         }
103     }
104
105     (*node_ptr)->height = max( height( (*node_ptr)->left_child ), height( (*node_ptr)->
    right_child ) ) + 1;
106 }
107
108 void delete(struct node** node_ptr, int data) {
109    if (*node_ptr == NULL) {
110       return;
111     }
112
113    if ( data < (*node_ptr)->data )
114         delete(&(*node_ptr)->left_child, data);
115    else if( data > (*node_ptr)->data )
116         delete(&(*node_ptr)->right_child, data);
117    else {
118         if( ((*node_ptr)->left_child == NULL) || ((*node_ptr)->right_child == NULL) ) {
119             struct node *temp = (*node_ptr)->left_child ? (*node_ptr)->left_child : (*
     node_ptr)->right_child;
120
121             if (temp == NULL) {
122                 temp = (*node_ptr);
123                 (*node_ptr) = NULL;
124             }  else {
125                 **node_ptr = *temp;
126             }
127             free(temp);
128         }  else {
129             int val = find_min((*node_ptr)->right_child);
130             (*node_ptr)->data = val;
131             delete(&(*node_ptr)->right_child, val);
132         }
133     }
134
135    if ((*node_ptr) == NULL)
136       return;
137
138    (*node_ptr)->height = 1 + max(height((*node_ptr)->left_child), height((*node_ptr)->
     right_child));
139
140    int b = balance(*node_ptr);
141
142    if (b == 2 ) {
143          if ( balance((*node_ptr)->left_child) >= 0) {
144           right_rotation(node_ptr);
145         }  else {
146           left_right_rotation(node_ptr);
147         }
148         return;
149    }
150    if (b == -2 ) {
151         if ( balance((*node_ptr)->right_child) <= 0) {
```

```
152             left_rotation(node_ptr);
153           }  else {
154             right_left_rotation(node_ptr);
155           }
156           return;
157       }
158   }
159
160   struct node* find(struct node* node, int data) {
161       if (node == NULL) {
162           return NULL;
163       }
164
165       if ( node->data > data ) {
166           return find(node->left_child, data);
167       }  else if ( node->data < data ) {
168           return find(node->right_child, data);
169       }
170       return node;
171   }
172
173   int find_min(struct node* node) {
174       if ( node == NULL ) {
175           return -1;
176       }
177       while (node->left_child != NULL)
178           node = node->left_child;
179       return node->data;
180   }
```

Beim Löschen mit der Funktion `delete` in den Zeilen Zeile 108-Zeile 158 gehen wir ähnlich vor, allerdings ist diese Operation etwas komplizierter, da wir, wie schon bei den binären Bäumen im vorangegangenen Abschnitt mehr Fälle betrachten müssen.

Wir starten wie bei den binären Bäumen mit dem Löschen des Knotens. Das erfolgt wie beim Einfügen durch einen rekursiven Abstieg im Baum, bis wir den richtigen Knoten gefunden haben. Da wir hier aber auch Werte umkopieren und dann evtl. innere Knoten löschen, können wir hier, anders als beim Einfügen mit `insert`, nicht mehr entscheiden, ob und in welcher Richtung sich die Balance verändert. Zudem kann es sein, dass wir die Balance auf jedem Level auf dem Weg zurück zur Wurzel durch eine geeignete Rotation anpassen müssen. Daher testen wir in Zeile 142 und Zeile 150 die beiden möglichen Abweichungen der Balance, die wir korrigieren müssen (größer als 2 oder kleiner als -2 kann der Wert nicht werden, da wir nur einen Knoten löschen und sich die Balance daher um höchstens 1 in eine Richtung ändern kann).

Da wir auf dem Weg zur Wurzel höchstens h Knoten anfassen, wenn h die Höhe des Baums ist, machen wir auch höchstens h Umsortierungen. Da jede einzelne in konstanter Zeit erfolgt, bleibt auch die Gesamtzeit, die wir für das Löschen eines Knotens brauchen, bei $\mathcal{O}(h)$.

Unsere Laufzeiten für Einfügen und Löschen verändern sich gegenüber den binären Bäumen aus dem letzten Abschnitt also nicht. Da wir durch die *AVL-Bedingung*, die wir mit unseren neuen Funktionen zusätzlich erhalten, allerdings wissen, dass der Baum ausgeglichen ist und seine Höhe daher um höchstens 1 von der minimal möglichen Höhe abweicht, haben wir erreicht, dass die Laufzeit unserer Funktionen zum Einfügen

und Löschen in $\mathcal{O}(\log\, n)$ liegt, also logarithmisch in der Anzahl der Elemente unserer Liste ist.

Die Funktionen zum Finden des kleinsten oder größten Elements oder die Suche im Baum haben sich gegenüber den Funktionen für binäre Bäume nicht verändert, so dass wir auch hier eine Laufzeit von $\mathcal{O}(\log\, n)$ haben.

Damit haben wir eine Listenstruktur erhalten, in der wir alle wesentlichen Operationen in logarithmischer Zeit im Verhältnis zur Länge der Liste ausführen können.

17.3 Heaps

In diesem Abschnitt wollen wir uns eine weitere Datenstruktur ansehen, die sogenannten *(binären) Heaps*, die auf binären Bäumen aufbaut[3]. Sie ist Grundlage einer effizienten Implementierung vieler weiterer Algorithmen, unter anderem des Algorithmus von Dijkstra zur Bestimmung kürzester Wege in Netzwerken (zum Beispiel Straßennetzwerke, oder Lieferbeziehungen). Wir können auch einen weiteren Sortieralgorithmus aus dieser Datenstruktur erhalten, den sogenannten *HeapSort*-Algorithmus.

Im ersten Teil werden wir unseren *Heap* wie im vorangegangenen Abschnitt als Baumstruktur mit Knoten, die Zeiger auf ihre Kinder enthalten, modellieren. Wir werden später sehen, dass wir die zugrundeliegende binäre Baumstruktur, die wir hier verwenden, wesentlich effizienter in einem klassischen Array implementieren können, wobei sich die Eltern-Kind-Relationen in Indexzrelationen in der Liste übersetzt werden muss.

Daten, die wir in einem *Heap* speichern wollen, müssen eine vollständige Ordnung haben, wir müssen also, wie schon beim Sortieren, entscheiden können, ob von zwei Elementen eines kleiner oder größer ist, oder ob beide Elemente gleich sind. Unsere Bäume sollen zudem immer balanciert sein, wir haben also erst dann einen Knoten auf Level k, wenn in Level $k-1$ alle möglichen Plätze belegt sind. Für den letzten Level wollen wir zudem eine eindeutige Reihenfolge, in der die Plätze belegt werden, haben. Das können wir auf verschiedene Weise erreichen.

- Zum einen können wir verlangen, dass wir immer den am weitesten links stehenden Platz im letzten Level füllen, oder
- wir laufen entsprechend der Binärdarstellung von $n+1 = \sum s_i 2^i$, wobei n die aktuelle Anzahl von Knoten im Baum ist, abwärts. Dabei gehen wir in Level k nach links, wenn $s_k = 1$ und rechts sonst.

Je nach Modellierung ist die eine Variante leichter zu implementieren als die andere. Das sehen wir auch im folgenden, wenn wir zuerst Heaps als Baumstruktur wie unsere balancierten Bäume implementieren. Hier ist die zweite Variante erheblich leichter, da wir die Abstiegsfolge durch sukzessives Teilen mit Rest durch 2 bekommen. Im Anschluss überlegen wir uns, dass sich im Fall von Heaps die Baumstruktur erheblich effizienter in einem klassischen `C`-Array realisieren lässt, wenn wir die erste Regel nehmen. Für die theoretischen Überlegungen zu Heaps spielt die Wahl aber keine Rolle. Um nicht immer auf beide Fälle eingehen zu müssen, betrachten wir in den Beispielen dabei nur die erste Option. Sie können alle Aussagen leicht auf die andere übertragen.

[3] Heaps können auch auf anderen Strukturen aufbauen, aber das werden wir uns in dieser Vorlesung nicht ansehen. Wir verzichten daher im Folgenden auf den Zusatz *binär*.

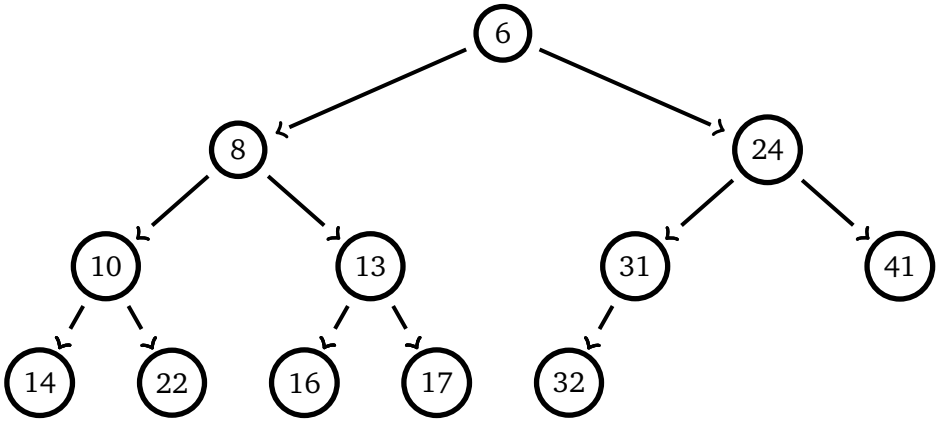


Abbildung 17.15: Ein Baum, der die (min)-Heap-Eigenschaft erfüllt.

Damit ist ein *binärer Heap* ein *binärer Baum* wir im ersten Abschnitt, mit den zwei zusätzlichen Eigenschaften, dass

- ▷ alle Level bis auf den letzten vollständig besetzt sind, und auf dem letzten Level sind die Knoten entsprechend der festgelegten Reihenfolge belegt.
- ▷ Für jeden Knoten mit Wert a erfüllen die Einträge b_1, b_2 an den Kindern entweder $a \leq b_1, b_2$ oder $a \geq b_1, b_2$. Die Relation muss an allen Knoten gleich sein.

Falls der Eintrag am Elternknoten immer kleiner oder gleich ist, spricht man von einem *min-Heap*, sonst von einem *max-Heap*. Wir wollen hier nur *min-Heaps* betrachten, aber die Übertragung auf die andere Form sollte Ihnen nicht schwerfallen. Die spezielle Anordnung der Knoten im Baum nennt man die *Heap-Eigenschaft*. Sie finden in Abbildung 17.15 ein Beispiel.

In einer Liste, die in einem *binären Heap* gespeichert ist, können wir sehr leicht das kleinste (größte im Fall von *max-Heaps*) Element finden. Dafür müssen wir nur den Eintrag an der Wurzel auslesen, den wir direkt über den Wurzelzeiger unseres Baums bekommen. Wir müssen uns allerdings überlegen, wie wir die *Heap-Eigenschaft*, die spezielle Anordnung unserer Knoten, aufrechterhalten können, wenn wir diesen Knoten entfernen, oder wenn wir neue Werte in die Liste einfügen wollen.

Wenn wir für Einfügen und Löschen schnelle Algorithmen finden, ähnlich wie für unsere balancierten Bäume, dann erhalten wir eine effiziente Struktur, mit der wir eine Liste sortieren können (erst alle Knoten einfügen, dann der Reihe nach wieder entfernen) oder im Lauf eines Algorithmus immer das kleinste (nach einer durch unser Problem vorgegebenen Bewertung) Element herausnehmen können. Wir werden sehen, dass wir die Struktur auch effizient aufrecht erhalten können, wenn sich einzelne Werte ändern. Damit bekommen eine sogenannte *Prioritätswarteschlange* (*priority queue*). Solche Strukturen gehen in viel Algorithmen ein, zum Beispiel in *Dijkstra Algorithmus* zur Berechnung kürzester Wege in Netzwerken (Straßennetze), den wir im nächsten Kapitel betrachten.

Wie können wir nun effizient ein Element einfügen und die *Heap-Eigenschaft* wieder herstellen? Dafür fügen wir das Element zuerst auf der letzten Ebene auf der ersten freien Stelle entsprechend der gewählten Reihenfolge ein. Wenn wir das Element 4 in den Baum aus Abbildung Abbildung 17.15 einfügen, erhalten wir damit den Baum Nun

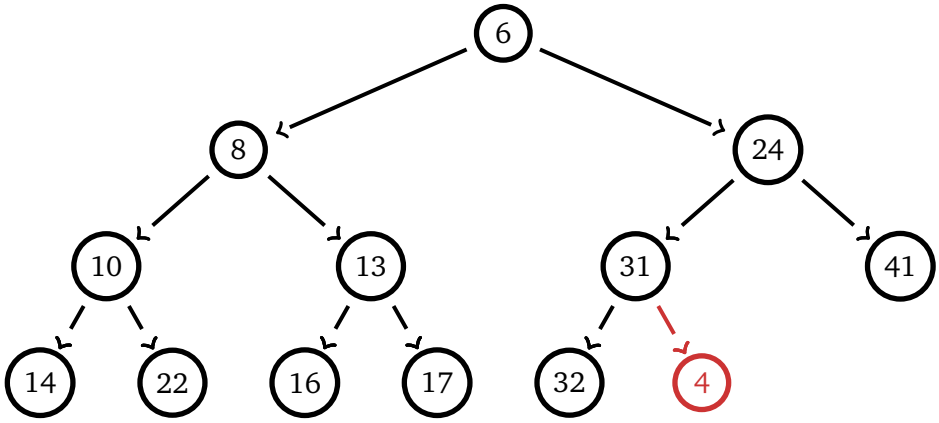


Abbildung 17.16: Erster Schritt beim Einfügen von 4 in den Baum aus Abbildung 17.15.

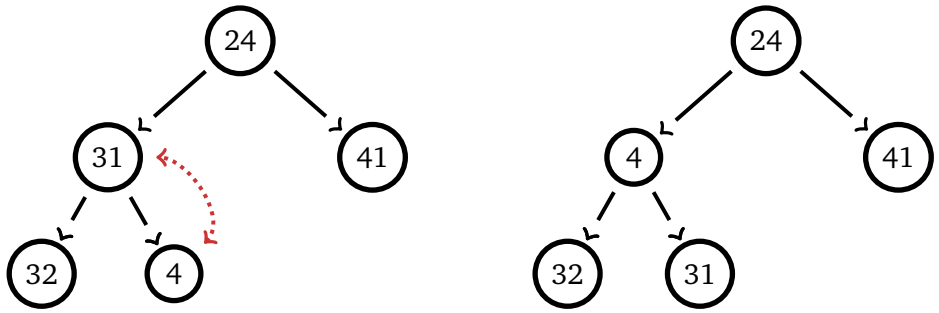


Abbildung 17.17: Rechter Teilbaum des Baums in Abbildung 17.16: Wir vertauschen 32 und 4 und erhalten den Teilbaum auf der rechten Seite

wird in der Regel die *Heap-Eigenschaft* am Elternknoten des gerade eingefügten Knoten nicht erfüllt sein, wir müssen also wie bei den balancierten Bäumen eine Methode finden, die Knoten im Baum so umzusortieren, dass die Eigenschaft wieder gilt. Dafür vergleichen wir den neuen Wert mit dem seines Elternknotens. Wenn der Wert im Elternknoten größer ist, vertauschen wir die beiden Werte. Im Beispiel vertauschen wir also 4 und 31, siehe Abbildung 17.17. Damit ist am Elternknoten die Heap-Eigenschaft wiederhergestellt, denn der Wert im anderen Kind war schon größer als der Wert, gegen den wir getauscht haben, also muss er auch größer als der neue Wert sein.

Dafür könnte jetzt die Heap-Eigenschaft am Knoten in der darüberliegenden Ebene verletzt sein. Das können wir aber dadurch korrigieren, dass wir unserer Vertauschungsoperation auf dieser Ebene wiederholen, und diesen Prozess fortsetzen, bis wir die Wurzel erreichen. Im Beispiel vertauschen wir 4 und 24. Wiederum muss damit die Heap-Eigenschaft gegenüber dem anderen Kind erhalten bleiben, da der Wert im Elternknoten kleiner wird. Nach einer weiteren Vertauschung erhalten wir den Baum aus Abbildung 17.19. Die Heap-Eigenschaft ist wiederhergestellt. Wir brauchen für diese Operation höchstens eine Vertauschung auf jedem Level. Da der Baum balanciert ist, brauchen wir also nur $\mathcal{O}(\log\, n)$ Operationen, wenn n die Gesamtzahl der Knoten im Baum (die Anzahl der Elemente unserer Liste) ist. Diese Operation wird oft *up-heap* genannt.

Auf sehr ähnliche Weise können wir auch den Wurzelknoten aus dem Baum entfernen.

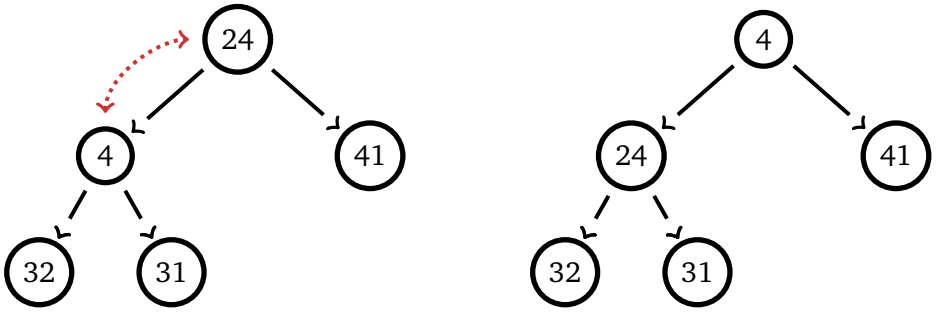


Abbildung 17.18: Fortsetzung der Vertauschung aus Abbildung 17.17 im rechter Teilbaum des Baums in Abbildung 17.16: Wir vertauschen 24 und 4 und erhalten den Teilbaum auf der rechten Seite

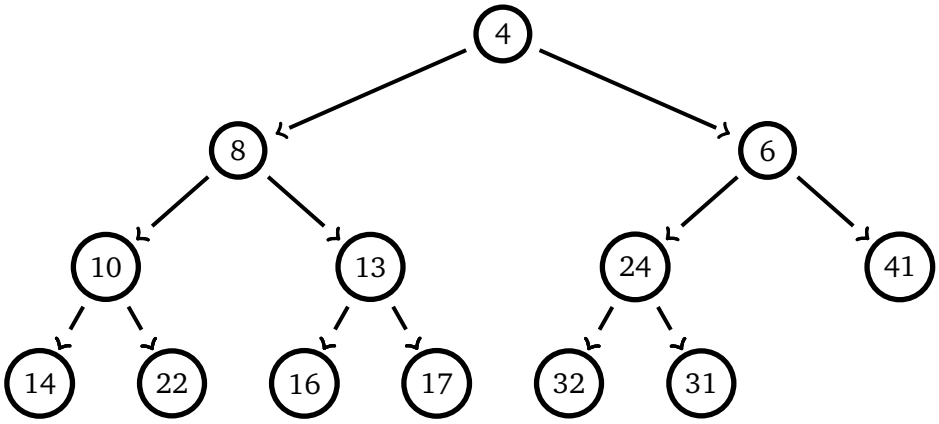


Abbildung 17.19: Einfügen von 4 in den Baum aus Abbildung 17.15.

Dazu nehmen wir den Wert aus der Wurzel heraus und ersetzen ihn durch den Wert des Knotens, der auf dem letzten Level auf dem letzten besetzten Platz in unserer Reihenfolge steht (also an der Position, an der wir eben eingefügt haben). Der Wert wird in der Regel zu groß sein und daher die Heap-Eigenschaft an dem Knoten verletzen. Aber so, wie wir eben beim Einfügen durch sukzessives Vertauschen den kleinen Wert nach oben befördert haben, können wir nun den zu großen Wert nach unten tauschen.

Wir vergleichen ihn dazu mit seinen beiden Kindern, und vertauschen mit dem kleineren. An der Wurzel ist damit die Heap-Eigenschaft wiederhergestellt. Dafür könnte sie auf dem nächsten Level an dem Knoten, mit dem wir vertauscht haben, verletzt sein. Das können wir aber beheben, in dem wir mit seinen beiden Kindern vergleichen und wieder mit dem kleineren vertauschen. Diesen Vorgang können wir wiederholen, bis die Heap-Eigenschaft erfüllt ist (was immer der Fall ist, wenn wir auf dem letzten Level angekommen sind). Wieder machen wir nur höchstens eine Vertauschung auf jeder Ebene, so dass wir nach $\mathcal{O}(\log n)$ Schritten die Heap-Eigenschaft an allen Knoten wiederhergestellt haben. Abbildung 17.20 zeigt ein Beispiel, das zum Baum in Abbildung 17.21 führt. Diese Operation wird oft *down-heap* genannt.

Damit können wir eine erste Implementierung eines binären Heaps angeben. Dafür gehen wir ähnlich wie bei den AVL-Bäumen im letzten Abschnitt vor und definieren uns eine Struktur `struct node`, die unsere Daten sowie Zeiger auf ein linkes und rechtes

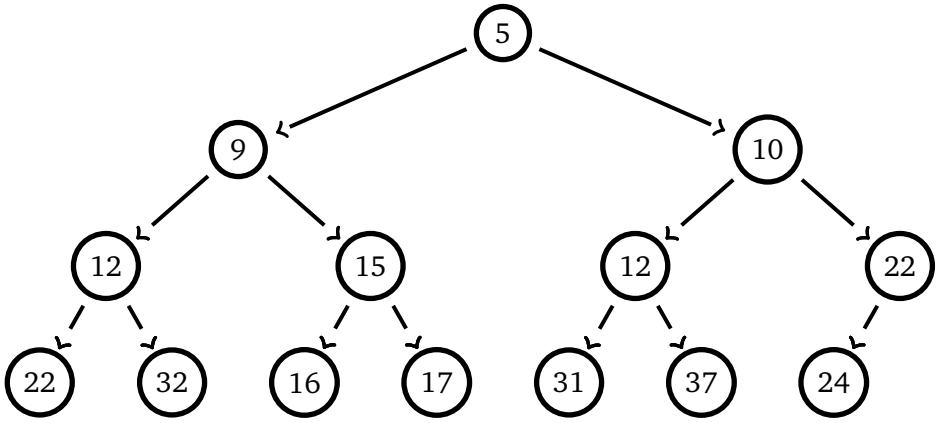


Abbildung 17.20: Löschen von 5: Wir Ersetzen mit 24 (und löschen diesen Knoten) und vertauschen dann der Reihe nach mit 9, 12, und 22 um den Baum in Abbildung 17.21 zu erhalten.

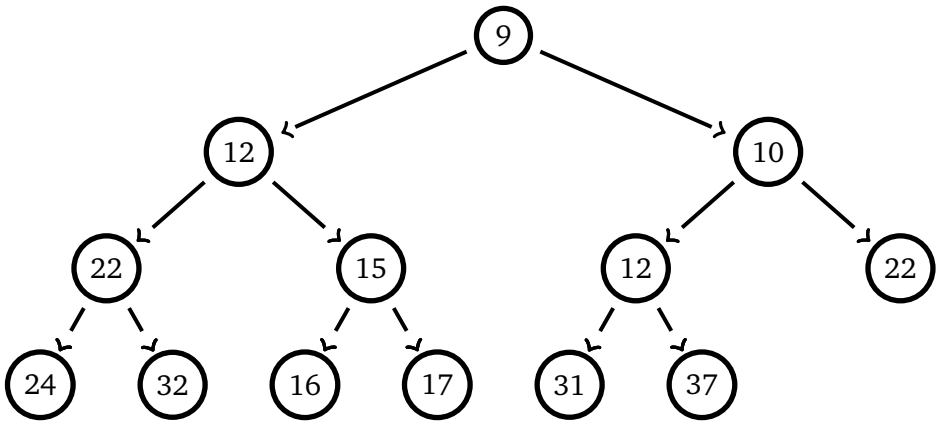


Abbildung 17.21: Der Baum aus Abbildung 17.20 nach Löschen der Wurzel und Umsortieren.

Kind enthält. Da wir bei Heaps allerdings auch die Gesamtgröße speichern müssen (um die Position des letzten Knotens zu bestimmen), führen wir eine zusätzliche Struktur `struct heap` ein, die den Kopfzeiger des Baums und die Größe zusammenfasst.

Zum Einfügen und Löschen implementieren wir dann die zwei Funktionen `heap_push` und `heap_pop`.

Programm 17.6: `kapitel_17/heap_tree.h`

```
#ifndef HEAP_TREE_H
#define HEAP_TREE_H

#include <stdio.h>

struct node {
    int data;
    struct node * left;
    struct node * right;
} ;

struct heap {
```

```
13      struct node * head;
14      int size;
15  } ;
16
17
18  struct heap * init_heap();
19  void free_heap(struct heap *);
20
21  int  heap_pop(struct heap *);
22  void heap_push(struct heap *, int);
23
24  #endif
```

Die Funktion `heap_push` muss zuerst den Spezialfall eines bisher leeren Heaps behandeln. In dem Fall initialisieren wir nur den Wurzelknoten und lassen den Kopfzeiger unserer Struktur `head` darauf zeigen. Andernfalls müssen wir unsere Operation `upheap` anwenden. Dafür steigen wir zuerst rekursiv im Baum entsprechend der Binärdarstellung der aktuellen Größe des Heaps an die Stelle ab, die durch den neuen Knoten besetzt wird. Wenn wir von einem Rekursionsschritt zurückkehren, testen wir, ob wir die Einträge von Kind und Elternknoten vertauschen müssen.

Löschen erfordert zwei Schritte. Im ersten müssen wir den Knoten auf der letzten Position finden, seinen Wert auslesen und dann löschen. Dieser Wert wird dann an der Wurzel eingetragen. Das macht die Funktion `remove_last_position`. Anschließend stellen wir mit unserer Operation `downheap` wieder die richtige Ordnung her. Da wir hier noch entscheiden müssen, ob wir mit dem linken oder rechten Kind vertauschen müssen (oder überhaupt nicht), und auf dem letzten Level nicht beide Kinder vorhanden sein müssen, ist diese Funktion etwas komplizierter als der `upheap`.

Programm 17.6:kapitel_17/heap_tree.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #include "heap_tree.h"
5
6
7   struct heap * init_heap() {
8       struct heap * heap = (struct heap *)malloc(sizeof(struct heap));
9       heap->head = NULL;
10      heap->size = 0;
11      return heap;
12  }
13
14  struct node * init_node(int data) {
15      struct node * node = (struct node *)malloc(sizeof(struct node));
16      node->left = NULL;
17      node->right = NULL;
18      node->data = data;
19      return node;
20  }
21
22  void free_node(struct node * node) {
23      free(node);
24  }
25
26  void free_node_with_children(struct node * node ) {
```

```
27      if ( node->left != NULL )
28          free_node_with_children(node->left);
29      if ( node->right != NULL )
30          free_node_with_children(node->right);
31      free_node(node);
32  }
33
34  void free_heap ( struct heap * heap) {
35      if ( heap->size > 0 ) {
36          struct node * node = heap->head;
37          free_node_with_children(node);
38      }
39      free(heap);
40  }
41
42  void upheap (struct node * node, int data, int pos) {
43      struct node * next;
44      if ( pos % 2 == 0 ) { // left child
45          if ( pos/2 == 1 )
46              node->left = init_node(data);
47          else
48              upheap(node->left,data,pos/2);
49          next = node->left;
50      }  else { // right child
51          if ( pos/2 == 1 )
52              node->right = init_node(data);
53          else
54             upheap(node->right,data,pos/2);
55          next = node->right;
56      }
57      if ( node->data > data ) {
58          next->data = node->data;
59          node->data = data;
60      }
61  }
62
63  void heap_push  (struct heap *heap, int data) {
64      if ( heap->size == 0 ) {
65          heap->head = init_node(data);
66          heap->size = 1;
67          return;
68      }
69
70      struct node * node = heap->head;
71      int size = ++heap->size;
72
73      upheap(node, data, size);
74  }
75
76  void downheap   (struct node *node, int pos) {
77      if ( node == NULL )
78          return;
79
80      if ( node->left != NULL ) {
81          if ( node->right != NULL ) {
82              if ( node->right->data < node->left->data && node->right->data < node->data
     ) {
83                  int temp = node->data;
```

```
84                  node->data = node->right->data;
85                  node->right->data = temp;
86                  downheap(node->right,pos);
87                  return;
88              }
89          }
90          if ( node->left->data < node->data ) {
91              int temp = node->data;
92              node->data = node->left->data;
93              node->left->data = temp;
94              downheap(node->left,pos);
95          }
96      }
97  }
98
99  int remove_last_position(struct node * node, int pos) {
100     struct node * next;
101     int left = pos % 2 == 0;
102     if ( left )
103         next = node->left;
104     else
105         next = node->right;
106     if ( pos/2 == 1 ) {
107         int data = next->data;
108         free_node(next);
109         if ( left )
110             node->left = NULL;
111         else
112             node->right = NULL;
113         return data;
114     }  else
115         return remove_last_position(next,pos/2);
116 }
117
118 int  heap_pop (struct heap *heap) {
119     if ( heap->head == NULL )
120         return -1;
121
122     struct node * node = heap->head;
123     int size = heap->size--;
124     int data = node->data;
125
126     if ( heap->size == 0 ) {
127         free_node(node);
128         heap->head = NULL;
129     }  else {
130         node->data = remove_last_position(node,size);
131         downheap(node,size);
132     }
133
134     return data;
135 }
```

Wir brauchen in der Bibliothek natürlich auch Funktionen, die Speicher für den Heap und seine Knoten anfordern und am Ende wieder freigeben. Dafür haben wir die Funktionen `free_node` und `free_heap`. Bei der zweiten müssen wir beachten, dass wir rekursiv zuerst alle Knoten freigeben müssen, bevor wir die Struktur `struct heap` freigeben

dürfen.

Im `git` zu diesem Skript finden sie noch eine weitere Bibliothek `heap_tree_print`, die den Baum unseres Heaps ins Terminal schreiben kann. Ein kleines Beispielprogramm zum Testen könnte dann wie folgt aussehen.

Programm 17.6:kapitel_17/heap_tree_main.c

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

#include "heap_tree.h"
#include "heap_tree_print.h"

int main(int argc, char** argv) {

    int max = argc > 1 ? atoi(argv[1]) : 10;
    srand(time(NULL));

    struct heap * heap = init_heap();

    for ( int i = 0; i < max; ++i )
        heap_push(heap,rand()%30);

    print_heap_tree(heap);

    while ( heap->size > 0 ) {
        int data = heap_pop(heap);
        printf("%d aus Baum\n", data);
    }

    free_heap(heap);

    return 0;
}
```

Unser Heap hat an allen inneren Knoten außer denen auf dem vorletzten Level immer zwei Kinder. Wenn wir alle Knoten auf dem letzten Level so weit links wie möglich anordnen (die erste der beiden Optionen, die wir am Anfang für unsere Sortierung genannt haben), dann können wir die Knoten levelweise von links nach rechts durchnummerieren. Wenn wir in der Zählung bei 0 anfangen, dann lassen sich aus dem Index k eines Knotens leicht die seiner Kinder ausrechnen. Wenn es sie gibt, dann haben sie die Indizes $2k+1$ und $2k+2$. Da wir immer so weit links wie möglich anordnen, kann dabei an der Stelle $2k+2$ nur ein Kind sein, wenn schon an $2k+1$ eines ist. Zudem ist dadurch an jedem Index zwischen 0 und der Gesamtgröße des Heaps ein Eintrag.

Wir können also unseren Heap statt in einer echten Baumstruktur auch in einer linearen Liste speichern. Abbildung 17.22 zeigt den Baum aus Abbildung 17.19 als eine solche Liste. Die Indizes k_1, k_2 der Kinder eines Knotens am Index k können wir über die Relationen $k_1 = 2k+1$ und $k_2 = 2k+2$ ausrechnen, siehe Abbildung 17.23. Da Listen in `C` erheblich effizienter sind als die für die Baumstruktur benutzten Operationen, in denen wir immer wieder neuen Speicher anfordern und rekursiv den Zeigern auf die Kinder folgen mussten, erhalten wir so eine deutlich schnellere Implementierung. Die erforderlichen Funktionen bleiben gleich, manche Aufgaben werden aber einfacher. Zum Beispiel brauchen wir beim Löschen keine gesonderte Funktion, die uns des letzte

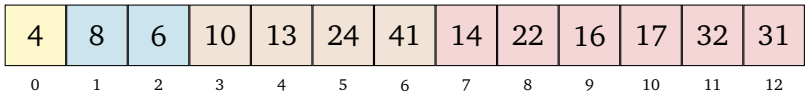


Abbildung 17.22: Der Baum aus Abbildung 17.19 als Liste. Die Farben entsprechen den Leveln im Baum.

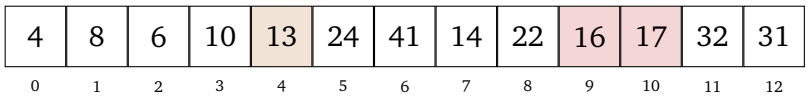


Abbildung 17.23: Der Knoten an Index 4 hat seine Kinder an den Indizes 9 und 10.

Element sucht, da wir die Anzahl n der Elemente im Heap kennen und daher wissen, dass das letzte Element am Index $n-1$ in der Liste steht.

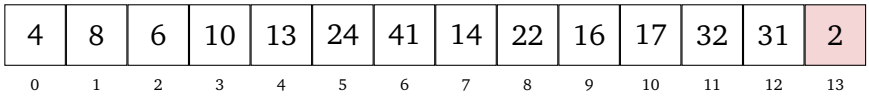
Zum Einfügen eines Elements schreiben wir das Element ans Ende der Liste, und wenden unsere Operation `upheap` an. So, wie wir aus dem Index eines Knotens den seiner Kinder bestimmen können, können wir natürlich auch den Index des Elternknotens ausrechnen. Für einen Knoten mit Index k hat dieser den Index $e = \lfloor (k-1)/2 \rfloor$. Wir vergleichen wieder die Einträge und vertauschen, wenn nötig. In Abbildung 17.24 können Sie die Veränderungen in der Liste an einem Beispiel nachverfolgen.

Ganz analog funktioniert dann auch das Löschen des kleinsten Elements aus der Liste. Es steht immer am Index 0. Wir tauschen das Element am Index $n-1$ auf diese Position und verkleinern die Länge der Liste um 1. Dann müssen wir mit der Operation `downheap` wieder die Heap-Eigenschaft herstellen. Dazu vergleichen wir das Element an der Wurzel mit seinen Kindern. Dabei sind die Kinder eines Knotens immer an den Indizes $k_1 = 2k+1$ und $k+2 = 2k+2$.

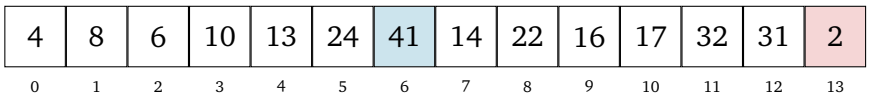
Wenn mindestens eines davon kleiner ist, vertauschen wir das kleinere mit der Wurzel. Mit diesem Prozess fahren wir fort, bis entweder beide Kinder größer sind oder wir an einem Blatt angekommen sind. Dabei haben wir ein Blatt erreicht, wenn für seinen Index k in der Liste die Indizes k_1 und k_2 größer sind als die Länge der Liste. Als Randfall müssen wir natürlich auch beachten, dass nur der zweite Index größer als die Listenlänge sein könnte, dass also der Knoten, den wir betrachten, nur noch ein Kind hat. Ein Beispiel für diese Operation können Sie in Abbildung 17.25 verfolgen.

Bei unserer Implementierung mit einer Liste müssen wir allerdings wieder mit dem Problem umgehen, dass wir die Größe einer klassischen Liste der Form `int liste[.]` in `C` nicht modifizieren können. Wir haben inzwischen zwei Wege gesehen, dieses Problem zu lösen. Zum einen können wir beim Übersetzen eine maximale Größe für unsere Liste festlegen (zum Beispiel als Präprozessorvariable), oder wir wählen den Ansatz, den wir schon bei den Stapeln benutzt haben und vergrößern die maximale Länge der Liste um einen bestimmten Faktor, wenn sie zu klein wird, indem wir mit **`realloc`** neuen Speicher anfordern und bei Bedarf umkopieren.

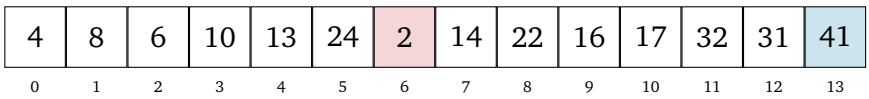
Mit einer Liste der Form `int liste[.]` können wir nur Listen von ganzen Zahlen in unsere Heap aufnehmen. Wenn Sie allgemeinere Daten haben, die Sie als Heap speichern wollen, müssen wir statt einer Liste von `int` eine Liste von Zeigern auf unsere Datenstruktur in der Liste speichern. Wenn unsere einzelnen Daten zum Beispiel



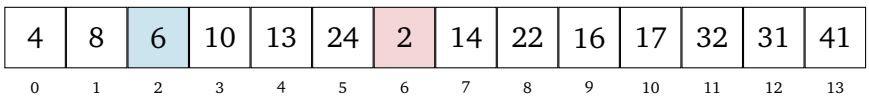
(a) 2 am Ende mit Index 13 angefügt



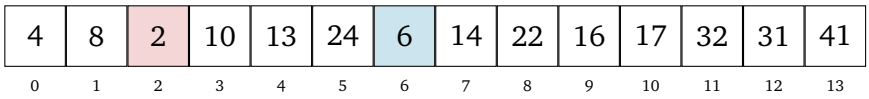
(b) Knoten 13 und sein Elternknoten 6.



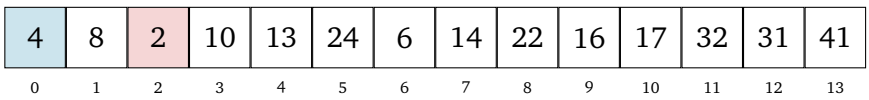
(c) Die erste Vertauschung.



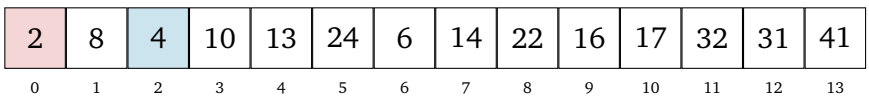
(d) Der nächste Elternknoten ist am Index 2.



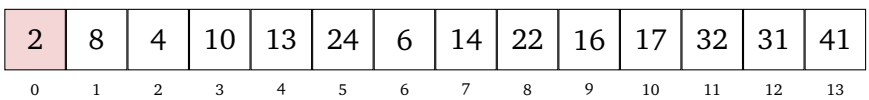
(e) Die zweite Vertauschung,



(f) Der nächste Elternknoten ist schon die Wurzel.

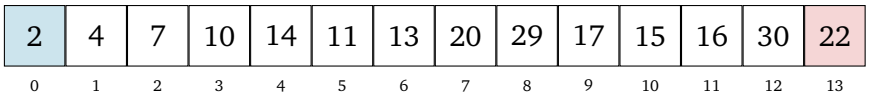


(g) Die dritte Vertauschung.

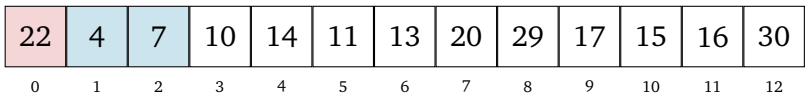


(h) Die Heap-Eigenschaft im Baum ist wieder hergestellt.

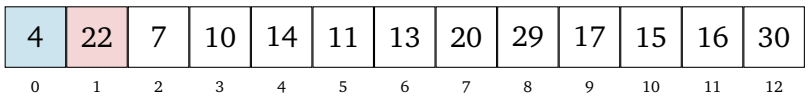
Abbildung 17.24: Einfügen von 2 und `upheap`



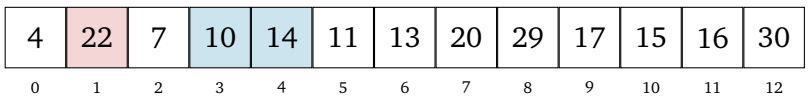
(a) 2 soll aus der Liste genommen werden. Das letzte Element wird dafür nach vorne getauscht und gelöscht.



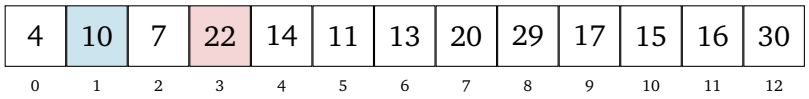
(b) Das letzte Element ist jetzt in der Wurzel. Wir müssen mit den beiden Kindern vergleichen.



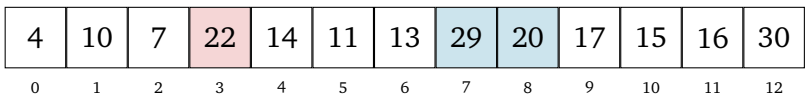
(c) Die Wurzel wird mit der 4 am Index 1 vertauscht.



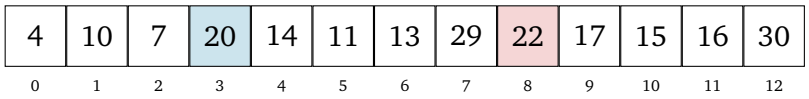
(d) Wir vergleichen wieder mit den Kindern.



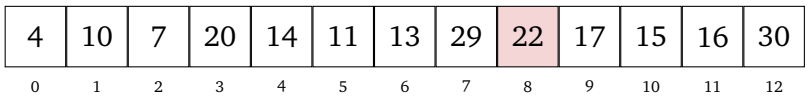
(e) Die 22 an Index 1 wird mit der 10 an Index 3 vertauscht, da die 14 am Index 4 größer ist.



(f) Wir vergleichen wieder mit den Kindern.



(g) Wir vertauschen mit der 20 am Index 8.



(h) Damit sind wir am Ende angekommen, denn der Knoten am Index 8 hat keine Kinder mehr (sie müssten an den Indizes 17 und 18 sitzen, aber so viele Einträge hat unsere Liste nicht).

Abbildung 17.25: Löschen von 2 und `downheap`

jeweils in einer Struktur `struct data` abgelegt sind, implementieren wir den Heap mit einer Liste der Form `struct data * liste[.]`. Unsere Vergleichsfunktion muss dann natürlich wieder angepasst werden und auf den passenden Eintrag in `struct data` zugreifen.

Im nachfolgenden Beispiel beschränken wir uns jedoch auf Listen von ganzen Zahlen, um die Implementierung einfacher zu halten. Wir legen auch zur Übersetzungszeit eine maximale Länge für unsere Liste fest. Die tatsächliche Länge müssen wir in jedem Fall getrennt mitführen (um zu wissen, an welchem Index aktuell das letzte Element steht). Das passiert im Beispiel mit der Variablen `size`.

Sie können das Programm aber leicht anhand der Beispiele aus dem Abschnitt zu verketteten Listen und Stapeln an einen allgemeinen Datentyp und eine Liste, deren maximale Länge wir zur Laufzeit dynamisch verändern können, anpassen.

Am Anfang müssen wir eine Eingabeliste natürlich richtig in unsere Heapstruktur schreiben. Dafür könnten wir die Struktur elementweise aufbauen, indem wir die Elemente mit `heap_push` nacheinander in die Liste einfügen. Man kann sich allerdings überlegen, dass wir in einem Baum, in dem wir die Elemente beliebig verteilt haben, der aber balanciert und die Knoten im letzten Level so weit links wie möglich hat, relativ leicht auch unsere Heap-Eigenschaft herstellen können. Dafür müssen wir nur, beginnend mit dem letzten Knoten auf dem vorletzten Level (dem letzten mit inneren Knoten), auf allen Knoten bis zur Wurzel die Operation `downheap` aufrufen. Daher nehmen wir mit der Funktion `build_heap` noch eine dritte Funktion hinzu, die diese Überlegung umsetzt.

Mit der nachfolgenden Implementierung geben wir ein Beispiel für eine Realisierung dieser Idee. Hier ist zuerst der Header.

Programm 17.7: `kapitel_17/heap.h`

```
1  #ifndef HEAP_H
2  #define HEAP_H
3
4  #include "heap_io.h"
5  struct heap {
6      int *data;
7      int length;
8      int array_size;
9  } ;
10
11 struct heap * init_heap (int);
12 struct heap * init_heap_from_array (int *, int);
13 void free_heap(struct heap *);
14 int  pop   (struct heap *);
15 void push  (struct heap *, int);
16
17 #endif
```

Die Funktionen können wir dann wie folgt ausführen. Die Funktionen `upheap` und `downheap` brauchen wir an dieser Stelle nur intern für die drei Bibliotheksfunktionen `heap_push`, `heap_pop` und `build_heap`, so dass wir diese nicht im Header aufführen müssten. Wir brauchen die Funktion `downheap` allerdings nachher für unsere Erweiterung zu einem Sortieralgorithmus, so dass wir sie schon hier in den Header aufnehmen.

Programm 17.7:`kapitel_17/heap.c`

```c

#include <stdlib.h>
#include <string.h>
#include <limits.h>

#include "heap.h"

struct heap * init_heap (int n) {
    struct heap * h = (struct heap *)malloc(sizeof (struct heap));
    h->data = (int *)malloc(n * sizeof (int));

    h->array_size   = n;
    h->length = 0;

    return h;
}

int min_index (struct heap *heap, int parent) {
    int ret = parent;
    int left_child = 2*parent+1;
    int right_child = 2*parent+2;
    if (left_child < heap->length && heap->data[left_child] < heap->data[ret]) {
        ret = left_child;
    }
    if (right_child < heap->length && heap->data[right_child] < heap->data[ret]) {
        ret = right_child;
    }
    return ret;
}

void downheap (struct heap *heap, int parent) {
    int min_child;
    while ( (min_child = min_index(heap, parent)) != parent ) {
        int temp = heap->data[parent];
        heap->data[parent] = heap->data[min_child];
        heap->data[min_child] = temp;
        parent = min_child;
    }
}

void build_heap (struct heap * heap) {
    for (int node = (heap->array_size - 2) / 2; node >= 0; node-- ) {
        downheap(heap, node);
    }
}
struct heap * init_heap_from_array (int * arr, int size) {
    struct heap * h = (struct heap *)malloc(sizeof (struct heap));
    h->data = arr;

    h->array_size = size;
    h->length = size;
    build_heap(h);
    return h;
}

void free_heap(struct heap * heap) {
    free(heap->data);
    free(heap);
```

```
59  }
60
61  void resize_heap ( struct heap * h) {
62      int new_size = h->array_size *3 / 2 >= h->length ? h->array_size *3 / 2 : h->
        array_size + 3;
63      h->data = realloc(h->data, (new_size) * sizeof(int) );
64      h->array_size = new_size;
65  }
66
67  void upheap(struct heap * heap, int node) {
68      while ( node != 0 ) {
69          int parent = (node-1)/2;
70          if ( heap->data[parent] > heap->data[node] ) {
71              int temp = heap->data[parent];
72              heap->data[parent] = heap->data[node];
73              heap->data[node] = temp;
74              node = parent;
75          }  else {
76              break;
77          }
78      }
79  }
80
81  int pop(struct heap * heap) {
82      int el = INT_MIN;
83      if ( heap->length > 0 ) {
84          el = heap->data[0];
85          heap->data[0] = heap->data[heap->length-1];
86          heap->length--;
87          downheap(heap,0);
88      }
89      return el;
90  }
91
92  void push(struct heap * heap, int data) {
93      if ( heap->array_size <= heap->length )
94          resize_heap(heap);
95
96      heap->data[heap->length++] = data;
97      upheap(heap, heap->length-1);
98  }
99
100 void upheap2(struct heap * heap, int node, int data) {
101     if ( node != 0 ) {
102         int parent = (node-1)/2;
103         if ( heap->data[parent] > data )
104             upheap2(heap, parent, data);
105         else
106             heap->data[node] = data;
107     }
108 }
109
110 void push2(struct heap * heap, int data) {
111     if ( heap->array_size <= heap->length )
112         resize_heap(heap);
113
114     heap->data[heap->length++] = data;
115     upheap2(heap, heap->length-1,data);
```

```
116 }
```

Im `git` finden Sie auch ein Programm `heap_main.c`, dass einen einfachen Test für unsere Heap-Struktur implementiert.

17.3.1 HeapSort

Mit unserer Heap-Struktur können wir einen neuen eleganten Sortieralgorithmus angeben. Dafür müssen wir die Liste, die wir sortieren wollen, nur anfangs einmal in unseren Heap schreiben, und anschließend mit `heap_pop` auslesen. Da diese Funktion immer das kleinste verbliebene Element zurückgibt, erhalten wir die Elemente in aufsteigender Reihenfolge.

Da Einfügen und Löschen von Elementen jeweils in Zeit $\mathcal{O}(\log\, n)$ möglich ist, und wir n Elemente einfügen und anschließend wieder löschen wollen, haben wir auf diese Weise einen weiteren Sortieralgorithmus erhalten, der wie *MergeSort* eine Liste in Zeit $\mathcal{O}(n \log\, n)$ sortieren kann.

Noch etwas leichter wird die Implementierung, wenn wir für eine aufsteigende Sortierung statt eines *min-Heap* einen *max-Heap* benutzen, dann können wir unseren Array sogar sortieren, ohne ihn in einen neuen Array oder in eine Heap-Struktur umkopieren zu müssen. Wir brauchen dafür auch nur die Funktionen `build_heap` und `downheap` aus unserer Heap-Implementierung (und die Hilfsunktion `min_index`, bzw. bei einem max-Heap dann eine entsprechende Funktion `max_index`, da wir nach dem Aufbau des Heaps keine weiteren Elemente mehr einfügen wollen.

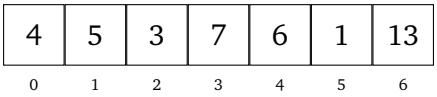
Um zu sehen, dass wir innerhalb des Arrays sortieren können, bemerken wir, dass wir beim Löschen eines Elements aus dem Heap das Element in der Wurzel (also an Index 0) auslesen und dann das Element auf der letzten Position an diese Stelle kopieren. Den Index, auf dem das letzte Element stand, brauchen wir anschließend nicht mehr. Wir können also das Element an der Wurzel dorthin kopieren. Da wir auf diese Weise den Array von hinten mit dem jeweils verbliebenen größten Element (da wir jetzt einen *max-Heap* verwenden) füllen, haben wir am Ende die Elemente in aufsteigender Folge in unserem Array stehen. In Abbildung 17.26 ist der Ablauf einmal anhand eines Beispiels dargestellt.

Basierend auf den Funktionen unserer Bibliothek `heap.c` gibt das nachfolgende Programm eine Implementierung von *HeapSort*. Wir verzichten aber auf einen eigenen `struct heap` und arbeiten direkt auf der Liste, da sich unsere Liste nach dem Einlesen nicht mehr vergrößert und wir so die eingelesene Liste nicht in den Heap kopieren müssen.

Programm 17.8: `kapitel_17/heapsort.h`

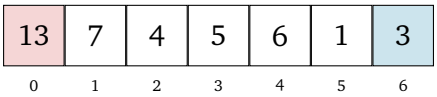
```
1 #ifndef HEAPSORT_H
2 #define HEAPSORT_H
3
4 void read_data(char *, int **, int *);
5 void heap_sort (int *, int);
6
7 #endif
```

In der Funktion `heapsort` vertauschen wir nur entsprechend der aktuellen Länge der

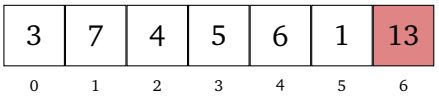


(a) Die unsortierte Liste

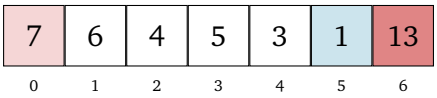
(b) Heap-Struktur aufgebaut



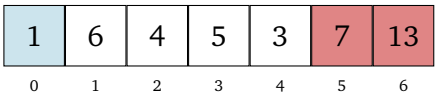
(c) Das erste Element ist das größte.



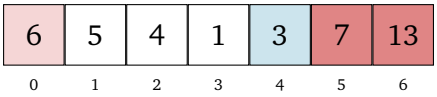
(d) Erstes und letztes Element vertauscht. Das letzte Element ist an seiner endgültigen Stelle, es verbleiben 6 Elemente in der Liste.



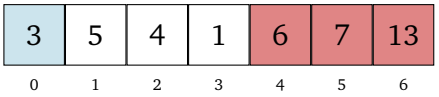
(e) Heap-Struktur auf den ersten sechs Elementen wieder hergestellt. Das erste Element ist das kleinste der verbliebenen Elemente.



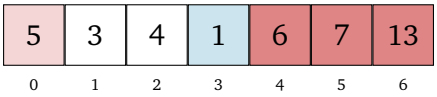
(f) Erstes und letztes Element der verkleinerten Liste vertauscht. Die letzten zwei Elemente sind an ihrer endgültigen Stelle, die Länge wird 5.



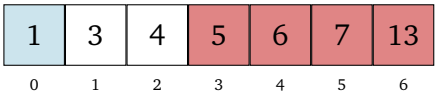
(g) Heap-Struktur auf den ersten fünf Elementen wieder hergestellt. Das erste Element ist das kleinste der noch nicht sortierten Elemente.



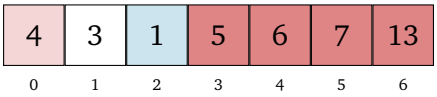
(h) Erstes und letztes Element der Liste vertauscht. Die letzten drei Elemente sind an ihrer endgültigen Stelle. Die neue Länge ist 4.



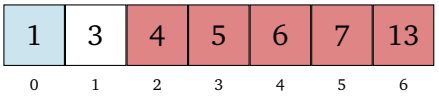
(i) Heap-Struktur auf den ersten vier Elementen wieder hergestellt. Das erste Element ist das kleinste verbliebene Element.



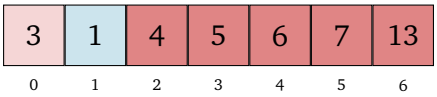
(j) Elemente vertauscht. Die letzten vier Elemente sind an ihrer endgültigen Stelle, Die Länge der zu sortierenden Liste wird zu 3.



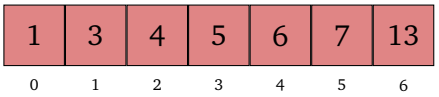
(k) Heap-Struktur auf den ersten drei Elementen wieder hergestellt. Das erste Element ist das kleinste der noch nicht sortierten Elemente.



(l) Erstes und letztes Element der verkleinerten Liste vertauscht. Es verbleiben zwei Elemente in der Liste, der Rest ist sortiert.



(m) Heap-Struktur auf den ersten zwei Elementen wieder hergestellt.



(n) Die letzte Vertauschung. Die Liste muss nun sortiert sein.

Abbildung 17.26: Die Umsortierung einer Liste durch HeapSort

noch zu sortierenden Liste das erste und letzte Element und rufen `downheap` auf. Die Länge fällt dabei von der Anfangslänge auf 0, während der sortierte Anteil von hinten nach vorne anwächst.

Programm 17.8:kapitel_17/heapsort.c

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

#include "heapsort.h"

void read_data(char* fname, int ** arr, int * size) {
    FILE *file = fopen(fname,"r");
    if ( file == NULL ) {
        printf("Datei nicht lesbar\n");
        exit(1);
    }

    *size = 0;
    int entry;
    int length = 4;
    *arr = (int *)malloc(sizeof(int) * length);
    while ( fscanf(file," %d",&entry) != EOF ) {
        if ( *size >= length ) {
            length *=2;
            *arr = realloc(*arr,length*sizeof(int));
        }
        (*arr)[(*size)++] = entry;
    }
    fclose(file);
    *arr = realloc(*arr,*size*sizeof(int));
}

 int max_index (int * heap, int size, int parent) {
    int ret = parent;
    int left_child = 2*parent+1;
    int right_child = 2*parent+2;
    if (left_child < size && heap[left_child] > heap[ret]) {
        ret = left_child;
    }
    if (right_child < size && heap[right_child] > heap[ret]) {
        ret = right_child;
    }
    return ret;
}

void downheap (int * heap, int size, int parent) {
    int max_child;
    while ( (max_child = max_index(heap, size, parent)) != parent ) {
        int temp = heap[parent];
        heap[parent] = heap[max_child];
        heap[max_child] = temp;
        parent = max_child;
    }
}

void build_heap (int * heap, int size) {
    for (int node = (size - 2) / 2; node >= 0; node-- ) {
        downheap(heap, size, node);
```

```
    }
}

void heap_sort (int * heap, int size) {

    build_heap(heap, size);

    for (int i = 0; i < size; i++) {
        int t = heap[size - i - 1];
        heap[size - i - 1] = heap[0];
        heap[0] = t;
        downheap(heap, size - i - 1, 0);
    }
}
```

Zum Ausprobieren ist hier auch ein kleines Programm, das eine Liste einliest und sortiert wieder ausgibt.

Programm 17.8:kapitel_17/heapsort_main.c

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

#include "heap.h"
#include "heapsort.h"

int main (int argc, char** argv) {

    int size = 0;
    int * arr;
    read_data(argv[1], &arr, &size);
    heap_sort(arr,size);

    for (int i = 0; i < size; i++) {
        printf("%d%s", arr[i], i == size - 1 ? "\n" : " ");
    }

    free(arr);

    return 0;
}
```

Version vom 10. Februar 2022

18 Versionskontrolle

Es ist bei größeren Projekten oft wünschenswert, Veränderungen im Code auch über eine längere Zeit nachvollziehen zu können, Teile wieder rückgängig zu machen oder sogar vollständig (vielleicht nur zu Testzwecken) zu einem älteren Stand des Codes zurückzukehren.

Wenn Sie nicht alleine an einem Projekt arbeiten, kommen Sie auch schnell an den Punkt, an dem Sie sicherstellen müssen, dass alle Änderungen im Code noch zusammenpassen, nicht mehrere Mitarbeiter*innen die gleichen Stellen im Code verändern und parallele Änderungen im Code druch verschiedene Mitarbeiter*innen am Ende korrekt zusammengefügt werden. Bei kleinere Projekten lässt sich das sicherlich dadurch lösen, dass man sich abspricht, wer wann an welcher Datei arbeitet und das Ergenis jeweils austauscht, aber bei größeren Projekten, und wenn nicht alle zeitgleich erreichbar sind (weil Sie zum Beispiel nicht am gleichen Ort arbeiten), werden solche Vereinbarungen schnell mühsam.

Beide Probleme, das der Nachvollziehbarkeit von Änderungen im Code, und die Zusammenführung von Code verschiedener Autor*inn*en, lässt sich gut mit *Versionskontrollsystemen* erreichen. Es gibt eine Reihe solcher Systeme (von denen im Moment im Wesentlichen *git* und *Mercurial* breite Verwendung finden). Allen gmeinsam ist das Prinzip, alle Änderungen im Code inkrementell zu speichern, also als Differenz zur Vorversion, und die Möglichkeit, Änderungen an verschiedenen Stellen in zwei Kopien des Codes zu einer einzigen neuen Version zusammenbauen zu können. Die Systeme können auch den Stand eines Projekts an einer bestimmten Stelle abzweigen, so dass auf dem Zweig unabhänggig vom Hauptzweig gearbeitet werden kann und bieten die Möglichkeit, solche Seitenzweige am Ende wieder durch Vereinigung der Veränderungen wieder in den Hauptzwei (oder einen anderen Zweig) zrückzuführen. Jede(r), die oder der am Code arbeitet, definiert durch eine Freigabe seiner lokalen Änderungen eine inkrementelle Veränderung des Projekts, die ab dann vom System verwaltet wird. Je nach System kann über einen Namen, den die Autorin oder der Autor vergibt, oder eine vom System vergebene Nummer auf diese konkrete Veränderung verwiesen werden.

Eines der ersten solchen Systeme war *RCS* (für *revision control system*), das ANfang der 80er Jahre aufkam. Das System war noch sehr einfach und speicherte Änderungen lokal in einem speziellen Unterverzeichnis. Weiterentwicklungen der Idee waren dann *CVS* (für *concurrent versioning system*) und *svn* (*subversion*). svn war lange Zeit das am weitesten verbreteitete System, bis es vor einigen Jahren von *git*, einem in einigen Bereichen technisch deutlich überlegenen System abgelöst wurde. Parallel dazu kam auch *Mercurial* auf, das ähnlich mächtig ist wie *git*, sich aber bisher nicht so durchsetzen konnte. svn hat noch bei älteren Projekten eine größere Bedeutung, aber immer mehr Projekte steigen inzwischen auf git oder Mercurial um. Auf *git* basieren aktuell seh

viele Projekte aus dem mathematischen Bereich, und auch viele Anbiter von cloud hosting-Lösungen für Code setzen auf git (wie zum Beispiel *github* und *gitlab*). Viele Cloud-Lösungen für gemeinsames Programmieren haben eine einfache Anbindung an Projekte, die mit *git* arbeiten (wie zum Beispiel das in diesem Kurs verwendete repl.it).

Wir wollen uns hier mit *git* eines der aktuell verbereiteten Systeme ansehen. Für die Verwendung muss das Programm `git` auf Ihrem Computer installiert sein. `git` wird wie `gcc` von der Kommandozeile aus aufgerufen und mit Optionen gesteuert[1]. Ein Projekt im Sinne von `git` sind dann alle Dateien, die in einem Verzeichnis (dem Wurzelverzeichnis des Projekts) und seinen Unterordnern liegen, und die für die Verwaltung durch `git` gekennzeichnet sind.

Wir können auf zwei verschiedene Weisen an ein solches von `git` verwaltetes Verzeichnis kommen.

- Entweder kopieren (klonen) wir ein bestehendes von `git` verwaltetes Verzeichnis von einer anderen Stelle (das wird oft eines der zentralen Archive wie github.com oder gitlab.com sein, kann aber auch ein Verzeichnis auf dem eigenen oder einem über eine remote-Verbindung erreichbaren PC sein),
- oder wir initialisieren ein bestehendes oder neues Verzeichnis als ein Verzeichnis, das ab sofort von `git` verwaltet werden soll.

Im ersten Fall müssen Sie die Adresse (was auch der lokale Pfad auf Ihrem Rechner sein kann) kennen und erzeugen die Kopie mit der Option `clone` an git. Wir haben das ganz am Anfang schon für die Programmbeispiele zu diesem Skript gemacht. Der Aufruf dafür ist der folgende.

```
\$ git clone https://git.rwth-aachen.de/programmieren_in_c/2020_21_codebeispiele
    2020_21_codebeispiele
```

Im Anschluss haben Sie ein neues, von git verwaltetes Verzeichnis, `2020_21_codebeispiele`. Wenn Sie ein bestehendes Verzeichnis von `git` verwalten lassen wollen, gehen Sie in das Verzeichnis und rufen

```
\$ git init
```

auf. Das bereitet die Versionsverwaltung vor. Mit

```
\$ git status
```

können Sie sich dann anzeigen lassen, welche von `git` verwalteten Dateien in dem Verzeichnis (und seinen Unterverzeichnissen) seit der letzten Freigabe verändert wurden und welche Dateien (noch) nicht von `git` verwaltet werden[2]. Mit

```
\$ git add <datei>
```

können Sie nun neue Dateien zur Verwaltung mit `git` hinzufügen oder veränderte Dateien auf die Warteliste für die Freigabe setzen. Sie geben die Dateien dann mit

```
\$ git commit
```

[1] Es gibt eine Reihe graphischer Programme, die die Aufrufe von `git` in einer netten Oberfläche verpacken. Alle rufen aber intern dann die zugehörigen Kommandos an `git` auf.

[2] über die Datei `.gitinit` können Sie festlegen, dass manche Dateien bei dieser Anzeige nicht berücksichtigt werden. Das ist nützlich zum Beispiel bei temporär angelegten Dateien oder, wenn Sie mit Kolleg*inn*en zusammenarbeiten, für lokale Konfigurationsdateien, die Sie nicht teilen wollen.

frei. Dabei geht ein Editor auf, in dem Sie eine kurze, aber vollständige, Beschreibung Ihrer Änderungen geben sollten, um später Ihre Arbeit leichter nachvollziehen zu können. Sie können dies Beschreibung auch mit der Option `-m` direkt an `git commit` übergeben. Damit haben Sie diese Änderung lokal in die Versionsgeschichte des Projekts eingetragen. Sie können sich alle Änderungen mit

```
\$ git log
```

ansehen. Wenn Sie Ihr Projekt am Anfang mit `git clone` von einer anderen Version des Projekts kopiert haben, können Sie jetzt noch mit

```
\$ git push origin master
```

Ihre Änderungen auf die andere Version übertragen. Dazu dürfen aber in der anderen Version keine veröffentlichten Veränderungen vorhanden sein. Andernfalls müssen Sie diese erst mit

```
\$ git pull
```

in Ihre Version übernehmen. Mit diesem Befehl können Sie auch unabhängig von Ihren Änderungen immer wieder Veränderungen seit dem letzten Aufruf von `git pull` übernehmen. Wie Sie ein lokal mit `git init` initialisiertes Verzeichnis auf einen der Cloud-basierten Dienste wie *github* übertragen, können Sie in der Dokumentation dieser Platformen nachlesen.

Wie schon am Anfang erwähnt, können Sie mehrere Versionen oder Linien Ihres Codes gleichzeitig mit `git` verwalten. Das kann zum Beispiel sinnvoll sein, wenn Sie eine sicher funktionierende Version benötigen, aber parallel an einigen Erweiterungen arbeiten wollen, oder einige Teile grundlegend umbauen wollen und dafür einen längeren Zeitraum brauchen, während Sie trotzdem mit der Software arbeiten können und kleinere Verbesserungen auch sofort benutzen können wollen. Sie können einen solchen Abzweig mit

```
\$ git branch <name>
```

erzeugen, wobei Sie diesem einen Namen geben müssen. Der Hauptzweig, der bei `git init` entsteht, heißt `master`. Mit

```
\$ git checkout <name>
```

wechseln Sie dann in diesen Zweig, und mit

```
\$ git checkout master
```

kommen Sie wieder zurück. In allen Zweigen können Sie nun getrennt Änderungen freigeben. Wenn Sie diese Änderungen wieder zusammenführen wollen, können Sie die Änderungen eines Zweigs in einen anderen übernehmen. Dazu wechseln Sie in den Zweig, der die Änderungen aufnehmen soll (mit `git checkout`) und rufen

```
\$ git merge <anderer_zweig>
```

auf.

Eine solche Zusammenführung der Änderungen, die auch vorgenommen wird, wenn sie mit `git pull` Änderungen von einer anderen Version übernehmen, ist nicht immer automatisch möglich. Das passiert insbesondere, wenn sich zwei Änderungen auf die

gleiche Stelle im Code beziehen. In diesem Fall meldet `git` zurück, dass die Zusammenführung nicht erfolgreich war und markiert im Code die Stellen, an denen das der Fall war. Dort finden Sie dann beide Versionen untereinander aufgeführt. In diesem Fall müssen Sie von Hand entscheiden, welche der Änderungen Sie übernehmen wollen (oder ersetzen die Stelle durch eine dritte Version). Wenn Sie alle Konfliktstellen abgearbeitet haben, können Sie mit

```
\$ git commit
```

`git` mitteilen, dass die aktuelle Version jetzt eine konfliktfreie neue Version des Projekts ist.

Jede Freigabe von Änderungen (ein *commit*) wird von `git` registriert und mit einer (hexadezimalen) Nummer versehen, die aus der Veränderung errechnet wird (ein sogenannter *Hashwert*). Anhand dieser Nummer lassen sich die Änderungen nachvollziehen. Sie sehen diese Nummer, wenn Sie sich mit

```
\$ git log
```

die letzten Änderungen anzeigen lassen. Jede Änderung beginnt mit dem Schlüsselwort `commit` gefolgt von seiner Nummer. Danach werden das Änderungsdatum und die Beschreibung, die sie beim Aufruf von `git commit` angegeben haben, angezeigt. Sie können Ihren Code auf auf den Stand einer solchen Änderung zurücksetzen. Dazu verwendet man, wie zum Wechsel des Zweigs, die Option `checkout`, gefolgt von der Nummer. Das kann zum Beispiel so aussehen.

```
\$ git log
commit 67033be03ff56b468253cf8edfd98a204b3151ac (HEAD -> master, origin/master)
Author: Andreas Paffenholz <paffenholz@mathematik.tu-darmstadt.de>
Date:   Mon Nov 23 12:40:05 2020 +0100

    Beschreibung von Versionskontrollsystemen

commit a14ad7ce51960e818cf63b5ae9f448b66e32996f
Author: Andreas Paffenholz <paffenholz@mathematik.tu-darmstadt.de>
Date:   Fri Nov 20 16:35:05 2020 +0100

    Kapitel über Operatoren fertiggestellt
\$ gt checkout a14ad7ce51960e818cf63b5ae9f448b66e32996f
```

Hier haben wir die Bearbeitung dieses Skripts wieder auf den Stand vor diesem Kapitel zurrückgedreht. Mit

```
\$ git checkout HEAD
```

kehren wir wieder zur aktuellen Version zurück. Wenn wir uns allerdings die alte Version nicht nur ansehen wollten, sondern alle Änderungen, die danach kamen verwerfen und die alte Version als die aktuelle nehmen wollen, dann können wir das mit

```
\$ git reset --hard <hash>
```

machen. Wenn Sie nicht alleine an einem Projekt arbeiten, sollten Sie das in der Regel aber mit Ihren Mitarbeiter*inne*n absprechen, da diese dadurch Änderungen, von denen Sie noch nichts wissen, verlieren könnten oder deren Versuch, diese Änderungen in den Hauptzweig zu übernehmen, scheitern könnten.

Es gibt noch viele weitere nützliche Optionen für git, und auch die bisher vorgestellten Befehle haben noch eine ganze Reihe nützliche Varianten, die Sie sich anschauen sollten, sobald Sie sie benötigen. Auf einen letzten wollen wir hier wenigstens noch hinweisen. Relativ häfig steht man in größeren Projekten vor dem Problem, dass nach einer Reihe von Änderungen, meistens durch unterschiedliche Personen, einige Funktionen im Programm nicht mehr so funktionieren, wie es vorgesehen war. Oft ist ein solcher Fehler dann nicht auf eine einzelne Änderung zurückzuführen, die sich leicht identifizieren liesse, sondern durch ein Zusammenspiel einiger Änderungen. In solchen Fällen ist es für die einzelnen Autor•inn*en oft schwer herauszufinden, wo der Fehler liegt. Hier ist die Option `bisect` zu `git` manchmal nützlich, mit der man die letzte Version des Codes finden kann, die noch korrekt war. Dazu identifiziert man eine Version des Codes, die korrekt ist, und eine, die nicht mehr richtig funktioniert. Diese beiden Versionsnummern (*hashes*) übergibt man nun an `git bisect`, und `git` macht dann sukzessive Intervallhalbierung auf den Versionen. Es wechselt zu einer Version zwischen den beiden aktuellen Grenzen, und je nachdem, ob Sie angeben, dass die angebotene Version korrekt ist oder nicht, passt `git` die untere oder obere Grenze an und wiederholt den Vorgang solange, bis nur noch eine Version übrig ist. Die genaue Anwendung können Sie in der Dokumentation nachlesen.

Version vom 10. Februar 2022

19 Weitere Algorithmen

19.1 Kürzeste Wege in Netzwerken

In diesem Abschnitt wollen wir die Heap-Struktur, die wir in Abschnitt 17.3 betrachtet haben, zu einer *Prioritätswarteschlange* erweitern und in einem Algorithmus zur Berechnung kürzester Wege in Netzwerken, dem *Algorithmus von Dijkstra*, einsetzen.

Netzwerke sind die vielen Anwendungen mathematischer Algorithmen auf praktische Probleme zugrundeliegenden Datenstrukturen. Ein Beispiel sind Straßennetzwerke aus Kreuzungen (den *Knoten*) und Straßen (den *Kanten*), die Kreuzungen verbinden. Jeder Straße können wir *Kosten* oder ein *Gewicht* zuordnen, zum Beispiel die Zeit, die man braucht, um auf der Straße von einer zur anderen Kreuzung zu kommen. Ein anderes Beispiel für ein Netzwerk ist der Netzplan eines Nahverkehrsverbunds. Wir haben hier Haltestellen als Knoten in unserem Netzwerk, und wir zeichnen eine Verbindung zwischen zwei Haltestellen, wenn es eine direkte Verbindung zwischen ihnen gibt (also eine Bahn oder ein Bus, der von einer zur anderen Haltestelle fährt). Auch hier können wir den Verbindungen Kosten oder Gewicht zuordnen, zum Beispiel die Fahrzeit, oder den Preis der Fahrkarte. Wenn wir nun zwei beliebige Knoten s und t in einem solchen Netzwerk markieren, können wir einem Weg zwischen diesen beiden Knoten, also einer Folge von Straßen, die wir benutzen müssen, oder einer Liste von Buslinien, die wir nacheinander benutzen und an gemeinsamen Haltestellen dazwischen umsteigen, einen Wert, zum Beispiel die Gesamtfahrzeit oder Gesamtticketkosten, zuweisen. Unser Ziel wird es in diesem Abschnitt sein, ein Programm zu erstellen, dass uns zu gegebenen Knoten einen Weg minimalen Gesamtwerts (einen *kürzesten Weg*), also kürzester Fahrzeit oder geringster Kosten für die Fahrkarte, berechnet.

Dafür müssen wir uns natürlich zuerst ein Format überlegen, wie wir unser Netzwerk in einer Datenstruktur in `C` darstellen und damit rechnen können. Außerdem brauchen wir einen Algorithmus, der uns einen solchen kürzesten Weg im Netzwerk bestimmt.

Dafür wollen wir uns zuerst etwas präziser überlegen, was wir unter einem Netzwerk verstehen wollen. Dafür führen wir einige neue Begriffe ein. Ein (*gerichteter*) *Graph* ist ein Paar $G = (V, A)$ aus einer *endlichen* Menge von *Knoten* V und einer Teilmenge A der Paare von verschiedenen Elementen aus V, also $A \subseteq V \times V \setminus \{(v, v) \mid v \in V\}$. Die Kanten können wir also als gerichtete Verbindungen zwischen zwei Knoten auffassen. Dabei nehmen wir an, dass eine Straße von einer Kreuzung zu einer anderen, von der Ausgangskreuzung verschiedenen, Kreuzung führt[1]. Wir dürfen dabei Kanten in beide Richtungen haben, für zwei Knoten a und b können wir also die Kanten (a, b) und (b, a) haben (in Straßennetzwerken ist das der übliche Fall, aber auch dort gibt

[1] Wenn Sie an Straßennetzwerke denken, ist das nicht immer der Fall, es gibt, insbesondere in Wohngebieten, manchmal Ringstraßen. Aber wir wollen unsere Überlegungen hier einfach halten

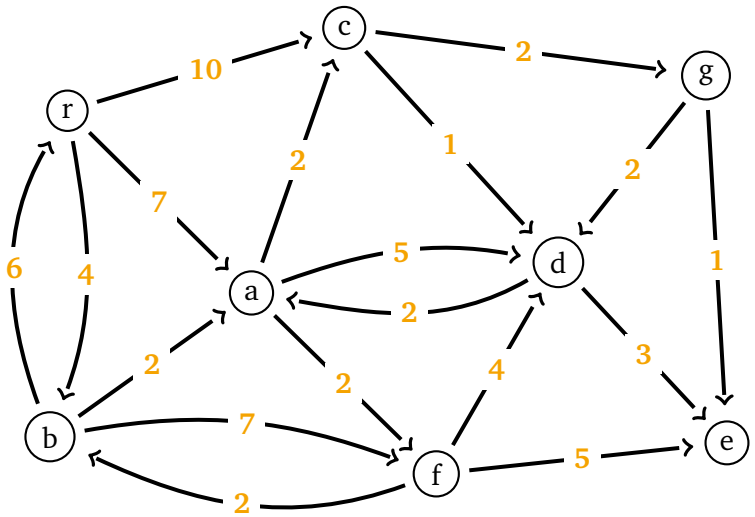


Abbildung 19.1: Ein gerichteter Graph mit 8 Knoten r, a, b, c, d. e, f und g. Eine Gewichtsfunktion ist durch die Zahlen auf den Kanten gegeben.

es Einbahnstraßen, die nur in einer Richtung benutzt werden können). Unsere Kosten oder Gewichte sind dann eine Funktion $w : A \to \mathbb{R}$ von der Menge der Kanten in die reellen Zahlen. Wir wollen im Folgenden annehmen, dass alle Gewichte nichtnegativ sind, das Bild von w also in $\mathbb{R}_{\geq 0}$ liegt. Ein Paar aus einem Graphen und einer Gewichtsfunktion wollen wir im restlichen Abschnitt als *Netzwerk* bezeichnen. Ein Beispiel ist in Abbildung 19.1.

Ein *Weg* W zwischen zwei Knoten x und y in einem Netzwerk ist dann eine Folge $x = v_0, v_1, \ldots v_k = y$ von Knoten in V, so dass $(v_i, v_{i+1}) \in A$ ist für $0 \leq i \leq k-1$. Wenn wir die Kanten, wie in der Abbildung, mit Pfeilen zwischen die Knoten zeichnen, ist ein Weg also eine Abfolge von Kanten, denen wir in Pfeilrichtung von x nach y folgen können. Das *Gewicht* eines Wegs ist nun die Summe der Gewichte auf den Kanten, die der Weg benutzt[2]. Ein Weg W zwischen zwischen zwei Knoten x und y ist ein *kürzester Weg* von x nach y, wenn seine Kosten minimal sind unter allen Wegen, die es zwischen x und y gibt. Ein solcher Weg muss natürlich nicht eindeutig sein, es kann mehrere Möglichkeiten geben, von x nach y zu kommen, die die gleichen Kosten haben. In Abbildung 19.2 ist ein Weg zwischen den Knoten r und e eingezeichnet. Wir wollen nun einen Algorithmus angeben, der einen möglichen kürzesten Weg zwischen zwei Knoten bestimmt. Unser Algorithmus wir tatsächlich etwas mehr ausrechnen. Wenn man das Problem mathematisch analysiert, kann man feststellen, dass es für die asymptotische Laufzeit im schlechtesten Fall keinen Unterschied macht, ob man die Distanz und den kürzesten Weg zwischen einem Knoten r und einem weiteren Knoten x bestimmt, oder ob man die Distanzen und Wege von r zu *allen* anderen Knoten im Netzwerk bestimmt. Das liegt daran, dass es sein kann, dass der kürzeste Weg zwischen zwei Knoten u und v alle anderen Knoten im Netzwerk durchläuft. Da dann jeder Ausschnitt aus dem Weg sicherlich auch ein kürzester Weg zwischen seinen beiden Endknoten ist, haben wir auf

[2]Falls dabei eine Kante mehrfach vorkommt, addieren wir das Gewicht der Kante entsprechend oft. Sie können sich aber leicht überlegen, dass dieser Fall nicht vorkommt, wenn wir Wege zwischen Knoten betrachten, die möglichst geringes Gewicht haben.

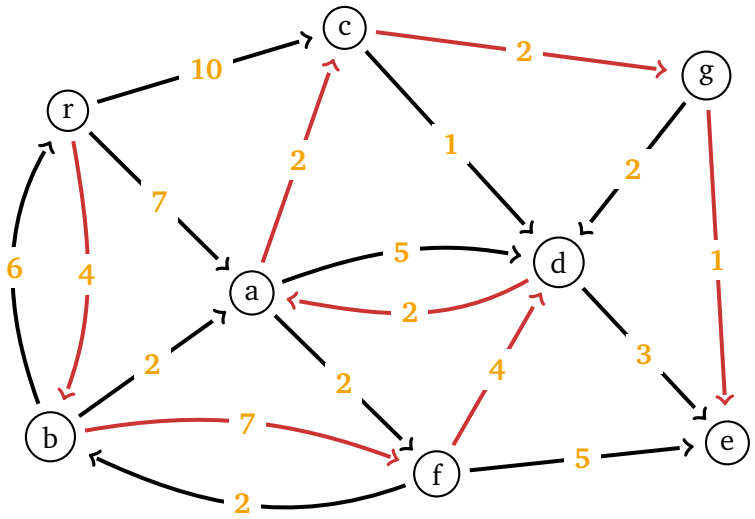


Abbildung 19.2: Die rot markierten Kanten bilden einen Weg von r nach e. Das ist jedoch kein kürzester Weg, denn der Weg von r über b, a, c und g nach e hat ein geringeres Gewicht.

diese Weise alle Distanzen von u zu anderen Knoten bestimmt. Daher wollen wir im Folgenden dieses allgemeinere Problem betrachten. Dafür zeichnen wir einen Knoten r im Netzwerk aus, den Startknoten für alle unsere Wege.

Bei der Berechnung, und dann auch bei der Ausgabe der Lösung, kommt uns noch eine weitere Beobachtung zu Hilfe, die die Speicherung und Angabe alle kürzesten Verbindungen von r zu den anderen Knoten erheblich vereinfacht. Wenn wir nämlich einen Weg minimalen Gewichts von r zu einem Knoten x kennen, und dieser Weg passiert unterwegs einen Knoten y, dann kennen wir auch einen (von vielleicht mehreren möglichen) Weg minimalen Gewichts von r nach y. Wenn wir den Weg zu x in y abbrechen, muss dieser Weg minimales Gewicht bis y haben. Wenn das nicht so wäre, und wir könnten billiger zu y kommen, dann könnten wir auch das Gewicht des Wegs zu x reduzieren, in dem wir dem billigeren Weg bis y folgen und dann auf den alten Weg zu x einschwenken.

Diese Überlegung gilt insbesondere auch, wenn y der Knoten auf dem Weg von r zu x ist, den wir als letztes vor x passieren. Um einen Weg minimalen Gewichts zu x anzugeben, reicht es uns also, anzugeben, von welchem Knoten wir auf dem Weg nach x kommen. Es reicht also, den *Vorgänger* (*predecessor*) y auf dem Weg zu x anzugeben. Bis zu y verwenden wir den kürzesten Weg zu y, der auf die gleiche Weise dadurch gegeben ist, dass wir den Vorgänger auf dem Weg zu y festhalten.

Wir können also alle kürzesten Wege von r zu allen Knoten durch eine Abbildung pred : $V \to V \cup \{\texttt{NULL}\}$ angeben (wobei wir `NULL` im Bild brauchen, da r keinen Vorgänger hat). Wenn wir die Kanten dazu betrachten, dann haben wir immer eine Kante, die vom Vorgänger zum Knoten gerichtet ist. Abbildung 19.3 zeigt diese Relation für unser Beispiel. Sie sehen in dem Beispiel auch, dass sie anhand der Relation den Kanten von r zu Ihrem Zielknoten laufen können. Durch die Vorgabe eines Vorgängers für jeden Knoten zeichnen Sie daher auch einen eindeutigen Weg von r zu jedem anderen Knoten aus, denn wir durch *backtracking* vom Ziel aus der Abbildung pred

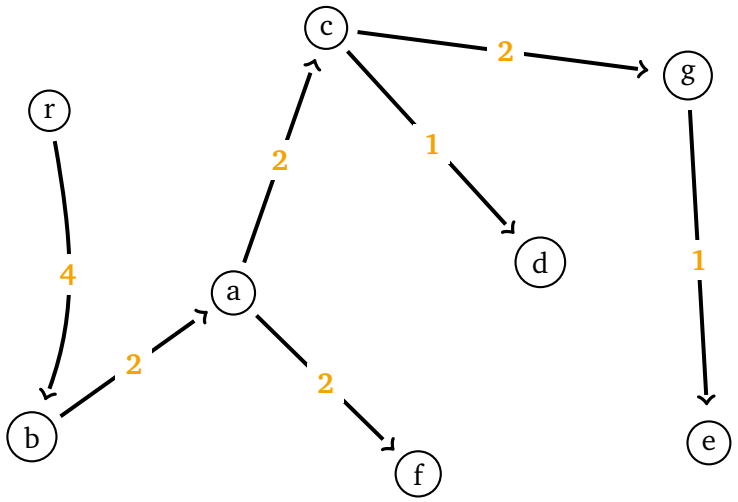


Abbildung 19.3: Die Vorgängerrelation zu den kürzesten Wegen in unserem Graphen. Wir haben eine Kante, die vom Vorgänger zum Knoten zeigt. Damit ist zum Beispiel $\mathsf{pred}(a) = b$ und $\mathsf{pred}(d) = c$.

bestimmen können.

Wir müssen uns überlegen, wie wir eine C-Struktur für ein Netzwerk aufbauen wollen. Da wir Distanzen und Wege zwischen Knoten des Netzwerks (Kreuzungen von Straßen) bestimmen wollen, sollen die Knoten die Grundlage unserer Datenstruktur werden. Alle weiteren Daten wollen wir in Abhängigkeit von den Knoten speichern. Wir könnten unsere Kanten nun als eine Liste von Paaren von Knoten speichern. Dann haben wir aber das Problem, dass wir immer die gesamte Liste durchsuchen müssten, um Verbindungen von einem bestimmten Knoten zu anderen zu finden. Bei großen Netzwerken mit vielen Kanten kann das zu sehr langen Laufzeiten führen.

Es ist daher günstiger, die Kanten, die in einem Knoten v starten, als eine zu dem Knoten gehörende Liste zu speichern. Dabei reicht es dann, nur die Zielknoten $w_1, \ldots, w_k$ zu notieren, für die $(v, w_i) \in A$. Die Notation nennt man die *Adjazenzliste* des Graphen[3]. Die Adjazenzliste unseres Beispiels finden Sie in Table 19.1a. Die Knoten in der Adjazenzliste eines einzelnen Knotens nennt man die (*ausgehenden*) *Nachbarn* des Knotens.

Wenn wir für jeden Eintrag in unserer Adjazenzliste nicht nur den Knoten, sondern auch das Gewicht der Kante speichern, die zu dem Knoten führt, dann können wir in unserer Adjazenzliste auch die Gewichtsfunktion abspeichern. Table 19.1a zeigt diese Struktur für unser Beispiel. Damit können wir eine Struktur in C für unseren Graphen entwerfen. Dabei wollen wir die Knoten im Graphen und die Liste der Nachbarknoten jedes Knotens in einem klassischen Array speichern fester Länge speichern, den wir, wie wir das bei den Stapeln in Programm 12.13 gesehen haben, in einen neuen, vergrößerten Speicherbereich kopieren, wenn unser bisheriger Platz erschöpft ist.

Hier ist eine mögliche Umsetzung dieser Ideen in ein `struct` in C. Da wir Kanten als Liste an ihrem Startknoten speichern, reicht es für eine vollständige Angabe, wenn wir

[3] Der Name kommt daher, dass man zwei Knoten in einem Graphen *adjazent* nennt, wenn es eine Kante zwischen ihnen gibt. Dabei bezieht man sich ursprünglich auf *ungerichtete* Graphen, bei denen wir den Kanten keine Richtung von einem zum anderen Knoten geben.

r:	$[a, b, c]$
a:	$[c, d, f]$
b:	$[r, a, f]$
c:	$[d, g]$
d:	$[a, e]$
e:	$[]$
f:	$[b, d, e]$
g:	$[d, e]$

(a) Die Adjazenzliste unseres Beispiels.

r:	$[(a, 7), (b, 4), (c, 10)]$
a:	$[(c, 2), (d, 5), (f, 2)]$
b:	$[(r, 6), (a, 2), (f, 7)]$
c:	$[(d, 1), (g, 2)]$
d:	$[(a, 2), (e, 3)]$
e:	$[]$
f:	$[(b, 2), (d, 4), (e, 5)]$
g:	$[(d, 2), (e, 1)]$

(b) Die Adjazenzliste unseres Beispiels mit Gewichten.

Tabelle 19.1: Adjazenzlisten

zu einer Kante Ziel und Gewicht bzw. Kosten speichern. Elemente dieser Kantenstruktur speichern wir dann als Liste an jedem Knoten, und die Knoten speichern wir dann in einer Liste an der Struktur, die das gesamte Netzwerk beschreibt.

```
1  struct edge {
2      int head;              // der Nachbarknoten
3      int weight;            // das Gewicht auf der Kante dorthin
4  } ;
5
6  struct node {
7      struct edge ** outgoing_edges;
8      int edge_struct_size; // die maximale Anzahl an Nachbarn, die wir speichern koennen
9      int n_outgoing_edges; // die Anzahl der Nachbarknoten
10 } ;
11
12 struct network {
13     struct node ** nodes;
14     int node_struct_size; // die maximale Anzahl an Knoten, die wir speichern koennen
15     int n_nodes;          // die Anzahl der Knoten des Graphen
16 } ;
```

Wir haben viele Möglichkeiten, wir wir die Übergabe eines Graphen an unser Programm organisieren wollen. In unserem Programm wollen wir den Graphen über eine Datei einlesen, in deren erster Zeile die Anzahl der Knoten und in allen weiteren ein Tripel von Zahlen (x, y, w) seht, wobei (x, y) eine Kante des Graphen ist und w ihr Gewicht. Wir haben also eine Liste der Kanten im Netzwerk. Für unser Beispiel sieht die Eingabe dann wie in Abbildung 19.4 aus. Dabei haben wir die Knoten für eine vereinfachte Notation durchnummeriert. Dabei entspricht r dem Knoten 0, und die Knoten $a, \ldots, f$ den Knoten $1, \ldots, 7$ in dieser Reihenfolge.

Eine Dateiformat in Form einer Adjazenzliste wäre sicherlich übersichtlicher, aber mit dieser Wahl haben wir den Vorteil, dass alle Zeilen die gleiche Länge und Struktur haben, so dass wir über einen Formatstring einlesen können. Um das Ergebnis unserer Berechnungen, die Distanz eines Knoten von unserem gewählten Startknoten und den Vorgänger auf dem Weg von dort, abspeichern zu können, brauchen wir noch zwei weitere Einträge in der Struktur des Knotens, die die Distanz und den Knoten speichern. Wir werden sehen, dass wir aus technischen Gründen noch zwei weitere Einträge brauchen, einen für die Kanten, und einen für die Knoten, die angeben, ob die Kante (gegeben durch den Zielknoten in der Adjazenzliste eines Knotens) eine Kante

```
8
0 1 7
0 2 4
0 3 10
1 3 2
1 4 5
1 6 2
2 0 6
2 1 2
2 6 7
3 4 1
3 7 2
4 1 2
4 5 3
6 2 2
6 4 4
6 5 5
7 4 2
7 5 1
```

Abbildung 19.4: Eingabe eines Graphen

auf einem kürzesten Weg ist, und einen für die Knoten, die angibt, ob wir die Distanz an diesem Knoten schon bestimmt haben.

Damit erhalten wir eine Deklaration von Graphen wie im folgenden Header, der auch drei Funktionen deklariert, um Speicher für einen Graphen zu reservieren und wieder freizugeben, sowie einen Graphen aus einer Datei einliest.

Programm 19.1: `kapitel_19/dijkstra/network.h`

```
1  #ifndef NETWORK_H
2  #define NETWORK_H
3
4  struct edge {
5      int head;
6      int weight;
7      int chosen;
8  } ;
9
10 struct node {
11     struct edge ** outgoing_edges;
12     int edge_struct_size;
13     int n_outgoing_edges;
14     int visited;
15     int dist;
16     int pred;
17 } ;
18
19 struct network {
20     struct node ** nodes;
21     int node_struct_size;
22     int n_nodes;
23 } ;
24
25 // initialize a network of size n
26 struct network * init_network(int n);
```

```
27
28 // read network from file
29 struct network * read_network(char*);
30
31 void free_network(struct network *);
32
33 #endif
```

Die entsprechenden Definitionen der Funktionen können wir zum Beispiel wie folgt ausführen.

Programm 19.1:kapitel_19/dijkstra/network.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <limits.h>
4
5  #include "network.h"
6
7  struct edge * init_edge(int head, int weight) {
8      struct edge * e = (struct edge *)malloc(sizeof(struct edge));
9      e->head = head;
10     e->weight = weight;
11     e->chosen = 1;
12     return e;
13 }
14 struct node * init_node() {
15     struct node * n = (struct node *)malloc(sizeof(struct node));
16     n->outgoing_edges = NULL;
17     n->n_outgoing_edges = 0;
18     n->edge_struct_size = 0;
19     n->visited = 0;
20     n->dist = INT_MAX; // no real distance should be that high
21     n->pred = -1;
22     return n;
23 }
24 struct network * init_network(int n_nodes) {
25     struct network * g = (struct network *)malloc(sizeof(struct network));
26     g->nodes = (struct node **)malloc(n_nodes * sizeof(struct node *));
27     for ( int i = 0; i < n_nodes; ++i ) {
28         g->nodes[i] = init_node();
29     }
30     g->n_nodes = n_nodes;
31     int node_struct_size = n_nodes;
32     return g;
33 }
34
35 void free_edge(struct edge * e) {
36     free(e);
37 }
38
39 void free_node(struct node * n) {
40     for ( int i = 0; i < n->n_outgoing_edges; ++i ) {
41         free_edge(n->outgoing_edges[i]);
42     }
43     free(n->outgoing_edges);
44     free(n);
45 }
46
```

```
47  void free_network(struct network * g) {
48      for ( int i = 0; i < g->n_nodes; ++i ) {
49          free_node(g->nodes[i]);
50      }
51      free(g->nodes);
52      free(g);
53  }
54
55  // checks if a given edge is in the network
56  int edge_exists(struct network * g, int tail, int head) {
57      struct node * n = g->nodes[tail];
58
59      for ( int i = 0; i < n->n_outgoing_edges; ++i ) {
60          if ( n->outgoing_edges[i]->head == head ) {
61              return 1;
62          }
63      }
64      return 0;
65  }
66
67  // adds one edge to a network if not yet present
68  int add_edge(struct network * g, int tail, int head, int weight) {
69      // if nodes don't exist do nothing
70      if ( tail >= g->n_nodes || head >= g->n_nodes ) {
71          return -1;
72      }
73      struct node * n = g->nodes[tail];
74
75      // if edge exists to nothing
76      if ( edge_exists(g,tail,head) ) {
77          return 1;
78      }
79
80      // we may have to increase the storage size
81      if (n->n_outgoing_edges >= n->edge_struct_size ) {
82          n->edge_struct_size = n->edge_struct_size ? n->edge_struct_size * 3 / 2 : 3;
83          n->outgoing_edges = realloc(n->outgoing_edges, n->edge_struct_size * sizeof (
        struct edge *));
84      }
85
86      // add the edge
87      n->outgoing_edges[n->n_outgoing_edges++] = init_edge(head,weight);
88      return 0;
89  }
90
91  /* for a network with n nodes we expect a file in the form
92  ---
93  n
94  tail-1 head-1 weight-1
95  tail-2 head-2 weight-2
96  ...
97  tail-n head-n weight-n
98  ---
99  */
100 struct network * read_network(char* filename) {
101   const char  s[2] = " ";
102   FILE *infile;
103   infile = fopen (filename, "r");
```

```
104   if (infile == NULL) {
105     fprintf(stderr, "\nError opening file\n");
106     exit (1);
107   }
108
109   int n_nodes;
110   fscanf(infile, "%d", &n_nodes);
111   struct network * g = init_network(n_nodes);
112
113   int tail, head, weight;
114   while ( fscanf(infile, "%d %d %d", &tail, &head, &weight) != EOF ) {
115     add_edge(g,tail, head, weight);
116   }
117
118   fclose(infile);
119   return g;
120 }
```

Nun müssen wir uns überlegen, wie wir unsere kürzesten Wege berechnen. Wir wollen dafür den *Algorithmus von Dijkstra* verwenden. Unserem Programm müssen wir neben dem Graphen dafür unseren gewählten Startknoten (das wird im Beispiel r bzw. 0 sein) übergeben.

Der Algorithmus berechnet die Distanzen in einer geschickten Iteration über alle Knoten des Graphen. Dabei weisen wir am Anfang jedem Knoten eine Abschätzung der tatsächlichen Distanz von oben zu, also einen Wert, von dem wir wissen, dass er die Distanz überschätzt. Dieses oberen Schranken werden wir in jeder Iteration verbessern, bis wir im Ende die wirklichen Distanzen an jedem Knoten erhalten. Dem Startknoten können wir sofort seine korrekte Distanz von 0 von sich selbst zuweisen. Bei der ersten Abschätzung aller weiteren Knoten können wir großzügig sein und jedem Knoten die Distanz ∞ zuweisen[4].

Der Algorithmus beruht auf einigen Überlegungen. Mit der ersten stellen wir fest, dass wir unsere Abschätzungen *lokal* verbessern können. Seien nämlich x und y Knoten, die durch eine Kante (x, y) von x nach y mit Gewicht w verbunden sind. Wir haben Abschätzung ihrer Distanzen d_x für x und d_y für y vom Start r. Wenn nun d_y größer ist als $d_x + w$ ($d_y > d_x + w$), dann können wir die Abschätzung der Distanz zu y verbessern, denn wir können den Weg zu x verwenden und dann über die Kante (x, y) zu y gehen. Wir können also $d_y = d_x + w$ setzen und x als den Vorgänger von y eintragen.

Wir wollen diese Überlegung ausnutzen, indem wir in geschickter Reihenfolge der Reihe nach alle Knoten des Graphen ansehen und an dem Knoten, an dem wir gerade sind, die Abschätzungen der Distanzen aller Nachbarn verbessern.

Wenn immer wir auf diese Weise einen Knoten bearbeitet haben, markieren wir den Knoten als *gesehen* (*visited*), schreiben ihn in eine Liste S, und fügen alle Nachbarn in eine Liste B von Kandidaten ein, die wir uns als nächstes ansehen könnten. In diese Liste B nehmen wir ganz am Anfang den Knoten r auf, und sobald ein Knoten *gesehen* wurde, entfernen wir ihn auch aus B. Da am Anfang nur r in B ist, wird dieser Knoten als erstes herausgenommen, bearbeitet und dann in S gelegt.

Wir müssen uns festlegen, welchen Knoten wir als nächstes aus B nehmen und

[4] Wir haben eigentlich auch keine andere Wahl, denn wenn es einen Knoten gibt, den wir vom Startknoten nicht erreichen können, gibt es auch keine endliche Distanz bis zu ihm und ∞ ist die korrekte Antwort.

ansehen. Wir nehmen immer den Knoten x, der aktuell die kleinste obere Schranke für die Distanz hat. Die zentrale Idee des Algorithmus ist nun, dass, wenn die Schranke an allen bisher *gesehenen* Knoten die korrekte Distanz ist (also nicht mehr verkleinert werden kann), dann muss auch die Schranke an diesem Knoten schon die korrekte Distanz sein.

Das können wir uns wie folgt überlegen. Da x in B war, ist sein aktueller Vorgänger ein Knoten aus der Menge S (der Knoten, die wir schon gesehen haben). Wenn das nicht schon der Vorgänger auf dem kürzesten Weg von r zu x ist, dann gibt es einen anderen Weg kürzerer Länge. Da r in S ist, x aber noch nicht, muss dieser Weg an irgendeiner Stelle eine Kante (v, w) benutzen, bei der v schon in S ist, aber w noch nicht. w muss aber in B liegen (da wir w beim betrachten von v dort abgelegt haben). Zudem ist dann v der Vorgänger auf einem kürzesten Weg zu w und die Schranke in w muss schon die wirkliche Distanz zu r sein. Nach Wahl von x ist diese Distanz aber mindestens so groß wie die aktuelle Schranke an die Distanz in x. Da alle Kanten nichtnegatives Gewicht haben und wir von w noch zu x kommen müssen auf dem Weg, kann das nicht sein. Also hat x in diesem Moment schon seine korrekte Distanz und den korrekten Vorgänger.

Wir fügen x dann in S ein, korrigieren ggf. die Distanzen seiner Nachbarn und legen diese in B, wenn sie nicht schon in S sind. Wenn wir auf diese Weise den letzten Knoten im Graphen betrachtet haben, haben alle Knoten die korrekte Distanz.

Zur Vereinfachung können wir noch die folgende Überlegung machen. Wir haben am Anfang die Schranke aller Knoten außer r auf ∞ gesetzt. r kam direkt in die Menge B, und jeder weitere Knoten, der in B aufgenommen wird, hat, da seine Schranke (mindestens) einmal angepasst wurde, eine endliche Schranke. Wir können also auch von Anfang an alle Knoten in B aufnehmen, da wir, wenn wir einen Knoten mit aktuell minimaler Schranke auswählen, nie einen Knoten nehmen, der noch eine Schranke ∞ hat. Damit bekommen wir den Algorithmus, der in Algorithmus 19.1 in Pseudocode ausgeführt ist.

Bevor wir unseren Algorithmus nun wirklich umsetzen können, brauchen wir noch einen weiteren Schritt. Wir müssen uns überlegen, wie wir die Menge B als Datenstruktur in `C` deklarieren wollen. Unsere wesentliche Anforderung ist, dass wir im Lauf des Programms immer wieder das Element mit minimaler aktueller Schranke entnehmen wollen. Hier fallen Ihnen vielleicht direkt die Heaps aus Abschnitt 17.3 als Option ein. Wenn wir hier diese Schranken an die Distanz als Ordnungskriterium für jeden Knoten nehmen, haben wir immer den Knoten mit minimaler Distanz in der Wurzel des binären Baums sitzen, und wir können die Datenstruktur in Zeit $\mathcal{O}(\log\, n)$ wieder korrigieren, wenn wir die Wurzel aus dem Baum entfernen. Es sieht also so aus als könnten wir auf diese Weise in jedem Schritt in Zeit $\mathcal{O}(\log\, n)$ den Knoten minimalen Gewichts finden und die Datenstruktur für B wieder aktualisieren.

In jeder Iteration des Algorithmus wird allerdings auch die Schranke an die Distanz in einigen Knoten verändert. Für unseren Heap bedeutet das, dass sich nach der Entfernung der Wurzel und der nachfolgenden Wiederherstellung der Heap-Struktur auch noch die Gewichte einiger Knoten im Baum ändern. Durch diese Operation könnte die Heap-Struktur wieder zerstört werden. Eine solche Operation hatten wir bei unseren Heaps bisher nicht vorgesehen.

Das Gewicht eines Knotens im Heap kann sich durch die Modifikation der aktuellen

Algorithmus 19.1: Algorithmus von Dijkstra

Eingabe : gerichteter Graph $G = (V, E)$,
Gewichtsfunktion $w : E \longrightarrow \mathbb{R}_{\geq 0}$,
Startknoten $r \in V$
Ausgabe : distance $D : V \longrightarrow \mathbb{R}_{\geq 0}$,
parent $\mathsf{pred} : V \longrightarrow V \cup \{\mathsf{nil}\}$

```
1  für jedes v ∈ V tue
2  |  D(v) ← ∞
3  |  pred(v) ← nil
4  |  visit(v)← 0
5  D(r) ← 0
6  B ← V
7  S ← ∅

8  solange B ≠ ∅ tue
9  |  u ← Knoten mit minimaler Gewichtsschranke in B
10 |  S ← S ∪ {u}
11 |  für jedes v in Nachbarschaft von u tue
12 |  |  wenn visit(v) == 0 dann
13 |  |  |  wenn D(u) + w(u,v) < D(v) dann
14 |  |  |  |  pred(v) ← u
15 |  |  |  |  D(v) ← D(u) + w(u,v)
```

Schranke jedoch nur *verkleinern*! Wenn das an einem Knoten passiert, bleibt also die Heapstruktur zu den Kindern des Knotens immer erhalten, wir müssen uns nur um die Relation zum Elternknoten kümmern. Aber da wissen wir schon, wie wir das korrigieren. Dafür haben wir schon beim Einfügen neuer Knoten die Operation `upheap` entwickelt, mit der wir einen neuen Knoten, den wir ganz unten im Baum angefügt haben, schrittweise nach oben getauscht haben, bis die Heapeigenschaft erfüllt war. Das können wir nun auch machen, wenn wir das Gewicht eines Knotens verringern, wir starten den `upheap` dann statt an dem neu eingefügten Blatt an dem Knoten, dessen Gewicht wir verändert haben.

Diese Operation für ein *Heap* nennt man oft `decrease_key`. An dieser Stelle haben wir benutzt, dass wir ein *min-Heap* haben und Gewichte nur verkleinert werden. Für ein *max-Heap* können wir die gleiche Operation betrachten, wenn Gewichte nur vergrößert werden. Diese Operation wird dann üblicherweise `increase_key` genannt. Da wir hier wie bei `push` nur die Operation `upheap` benutzen, können wir nach Anpassung eines Gewichts die *Heap*-Struktur auch in diesem Fall in Zeit $\mathcal{O}(\log n)$ wiederherstellen.

Weitere Operationen brauchen wir nicht, um B zu implementieren. Mit dieser Erweiterung können wir für die Menge B also sehr effizient einen *Heap* benutzen.

Ihnen ist vielleicht schon aufgefallen, dass wir mit dieser Datenstruktur trotzdem noch ein weiteres Problem haben, das dann auch das letzte sein wird, bevor wir daran gehen können, Dijkstras Algorithmus umzusetzen und ein Programm zur Berechnung kürzester Wege zu haben. Wir haben mit unseren Überlegungen eine effiziente Möglichkeit gefunden, das Gewicht eines Knotens zu reduzieren und anschließend die Heap-Struktur

wiederherzustellen. Dabei sind wir allerdings davon ausgegangen, dass wir wissen, welcher Knoten in der *Heap-Struktur* sein Gewicht verändert. Wenn wir nun aus der Sicht des Algorithmus von Dijkstra auf diese Reduktion sehen, dann machen wir diese nicht an einer bestimmten Stelle im Heap, sondern an einem bestimmten Knoten. Wir brauchen also einen (effizienten) Weg, wie wir herausfinden, an welcher Stelle im Heap der Knoten gespeichert ist, dessen Gewicht wir anpassen wollen. Eine Suche im Heap ist an der Stelle nicht sinnvoll. Da in einem *Heap*, anders als in *AVL-Bäumen*, keine Bedingung der Art gilt, dass im linken Teilbaum nur kleinere Knoten und im rechten nur größere stehen, haben wir in einem Heap keine effiziente Möglichkeit, einen Knoten zu suchen.

Hier können wir uns von der speziellen Struktur helfen lassen, mit der wir unser Heap realisiert haben. Wir haben dafür (in der effizienten Implementierung mit einem Array, wie wir das auch oben gemacht haben), eine klassische indexbasierte Liste in `C` gewählt, an der die Kinder des Knotens mit Index k an den Stellen $2k+1$ und $2k+2$ stehen.

Um nun einen konkreten Knoten mit Index k im Heap schnell auffinden zu können, führen wir eine zweite Liste `heap_index`, in der wir an Stelle k festhalten, welchen Index in der Liste des Heaps der Knoten k hat. Diese Liste müssen wir natürlich bei allen Heap-Operationen entsprechend aktualisieren. Das geht aber in der gleichen Laufzeit wie die Operationen zur Wiederherstellung der Heapeigenschaft, so dass wir hier keine Zeit verlieren. Der Zugriff auf einen konkreten über `heap_index` erfolgt dann sogar in konstanter Zeit.

Eine mögliche Schnittstelle könnte also so aussehen.

Programm 19.2: `kapitel_19/dijkstra/heap.h`

```
#ifndef HEAP_H
#define HEAP_H

struct heap {
    int *data;
    int *key;
    int length;
    int size;
} ;

// initialize a heap of given size
struct heap * init_heap (int);
void free_heap(struct heap *);

void print_heap(struct heap * h);

// push element
// a array of element indices,
// node node index
// key distance at node
void push(struct heap * h, int * a, int node, int key);

// return root element
int pop(struct heap *, int *);

// adjust key at element
// a array of element indices,
```

```
28 // node node index
29 // new_key adjusted distance at node
30 void decrease_key (struct heap * h, int * a, int node, int new_key);
31
32 #endif
```

Die Deklarationen könnten wir dann wie in dem folgenden Codebeispiel ausführen.

Programm 19.2:kapitel_19/dijkstra/heap.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <limits.h>
4
5  #include "heap.h"
6
7  struct heap * init_heap (int n) {
8      struct heap * h = (struct heap *)malloc(sizeof (struct heap));
9
10     h->data = (int *)malloc((n+1) * sizeof (int));
11     h->key  = (int *)malloc((n+1) * sizeof (int));
12
13     h->size   = n;
14     h->length = 0;
15
16     return h;
17 }
18
19 void free_heap(struct heap * h) {
20     free(h->data);
21     free(h->key);
22     free(h);
23 }
24
25 void resize_heap ( struct heap * h) {
26     int new_size = h->size *3 / 2 >= h->length ? h->size *3 / 2 : h->size + 3;
27     h->data = realloc(h->data, (new_size) * sizeof(int) );
28     h->key = realloc(h->key,   (new_size) * sizeof(int) );
29     h->size = new_size;
30 }
31
32 void print_heap(struct heap * h) {
33     printf("current heap (entries as (node, key)): ");
34     for ( int i = 0; i < h->length; ++i) {
35         printf("(%d, %d) ", h->data[i], h->key[i]);
36     }
37     printf("\n");
38 }
39
40 int upheap ( struct heap * h, int * heap_index, int i, int key ) {
41
42     int j = (i-1) / 2 ;
43     while (i > 0) {
44         if (h->key[j] < key)
45             break;
46         h->data[i] = h->data[j];
47         h->key[i] = h->key[j];
48         heap_index[h->data[i]] = i;
49         i = j;
```

```
50          j = (j-1) / 2;
51      }
52      return i;
53  }
54
55  void decrease_key (struct heap * h, int * heap_index, int node, int key ) {
56
57      int insert_pos = upheap(h, heap_index, heap_index[node], key);
58      h->data[insert_pos] = node;
59      h->key[insert_pos ] = key;
60      heap_index[node]    = insert_pos;
61  }
62
63  void push (struct heap * h, int * heap_index, int node, int key ) {
64
65      // adjust heap size if necessary
66      if ( h->size <= h->length )
67          resize_heap(h);
68
69      // find the new position before inserting
70      // so upheap only needs to shift elems downwards
71      int insert_pos = upheap(h, heap_index, h->length++, key);
72      // move into place
73      h->data[insert_pos] = node;
74      h->key[insert_pos ] = key;
75      heap_index[node]    = insert_pos;
76  }
77
78  int min (struct heap * h, int i) {
79      int lc = 2 * i + 1;
80      int rc = 2 * i + 2;
81      if (lc < h->length && h->key[lc] < h->key[h->length])
82          i = lc;
83      if (rc < h->length && h->key[rc] < h->key[h->length])
84          i = rc;
85      return i;
86  }
87
88  int downheap ( struct heap * h, int * heap_index, int i) {
89      int j = min(h, i);
90      if ( j != i ) {
91          h->data[i] = h->data[j];
92          h->key[i] = h->key[j];
93          heap_index[h->data[i]] = i;
94          j = downheap(h,heap_index, j);
95      }
96      return j;
97  }
98
99  int pop (struct heap * h, int * heap_index) {
100     if ( h->length == 0 ) {
101         return INT_MIN;
102     }
103
104     //store the return
105     int node = h->data[0];
106     h->length--;
107     // the element we need to downheap
```

```
108     // is in position h->length, we don't copy it into pos 0
109     // then downheap only needs to move elements upward
110     // until we have found the new position
111     int j = downheap(h,heap_index,0);
112     // move the element into position
113     // and fix head_index
114     h->data[j] = h->data[h->length];
115     h->key[j]  = h->key[h->length];
116     heap_index[h->data[h->length]] = j;
117
118     return node;
119 }
```

Damit können wir die folgende Schnittstelle für unseren Algorithmus deklarieren.

Programm 19.3: `kapitel_19/dijkstra/dijkstra.h`

```
1  #ifndef DIJKSTRA_H
2  #define DIJKSTRA_H
3
4  #include "stdio.h"
5  #include "stdlib.h"
6  #include "limits.h"
7  #include "network.h"
8  #include "heap.h"
9
10 // compute distances and store those
11 // and predecessors on the network
12 // second argument is the start node
13 void dijkstra (struct network *, int);
14
15 // extract the predecessors from a network
16 // on which Dijkstra's algorithm has run
17 // second argument returns array with predecessors
18 void predecessors(struct network *, int *);
19
20 #endif
```

Die Ausführung folgt dann einfach dem Pseudocode aus Algorithmus 19.1.

Programm 19.3:`kapitel_19/dijkstra/dijkstra.c`

```
1  #include "dijkstra.h"
2
3  void dijkstra (struct network * g, int start) {
4
5      // initializize a heap as priotity queue
6      // the heap keeps the node indices, not pointers to nodes
7      struct heap * h = init_heap(3);
8      // we need a secon list that tells us the current position of a node in the heap
9      int * heap_index = (int *)malloc(g->n_nodes * sizeof(int));
10     for ( int i = 0; i < g->n_nodes; ++i) {
11         heap_index[i] = 0;
12     }
13
14     // put root into heap
15     // the root ha no predecessor, we mark this by -1
16     // elements not in the heap implicitly have distance 0
17     push(h, heap_index, start, 0);
18     g->nodes[start]->pred=-1;
```

```
19      g->nodes[start]->dist= 0;
20
21      while (h->length) {
22          // get the element with smallest distance
23          int node_index = pop(h, heap_index);
24          struct node * node = g->nodes[node_index];
25          node->visited = 1;  // mark the current node as visited
26
27          // now update distances to neighbors if necessary
28          for (int j = 0; j < node->n_outgoing_edges; j++) {
29              struct edge * e = node->outgoing_edges[j];
30              struct node * neighbor = g->nodes[e->head];
31              // we check if we have already seen the node,
32              // then we do nothing, this node has already the correct distance
33              // else we check if going to the node via the current one
34              // creates a shorter path
35              if (!neighbor->visited && node->dist + e->weight <= neighbor->dist) {
36                  neighbor->pred = node_index;
37                  neighbor->dist = node->dist + e->weight;
38                  // if its index in the heap is still zero,
39                  // then the node is not yet in the heap (still distance 0)
40                  // In that case we push with current distance
41                  // else we adjust the distance at the node
42                  if ( heap_index[e->head] == 0 )
43                      push(h, heap_index, e->head, neighbor->dist);
44                  else
45                      decrease_key(h, heap_index, e->head, neighbor->dist);
46              }
47          }
48      }
49
50      // cleanup
51      free(heap_index);
52      free_heap(h);
53  }
54
55  // fill an array with the predecessors
56  // once the algorithm of Dijkstra has run
57  void predecessors(struct network * g, int * pred) {
58      for ( int i = 0; i < g->n_nodes; ++i ) {
59          pred[i] = g->nodes[i]->pred;
60      }
61  }
```

Zum Abschluss brauchen wir noch ein Programm, dass die verschiedenen Teile zusammenführt. Es soll also einen Graphen über unsere Bibliothek `graph.h` einlesen, mit `dijkstra.h` die kürzesten Wege bestimmen und diese zusammen mit den Distanzen am Ende ausgeben. Eine einfache Version eines solchen Programms könnte so aussehen.

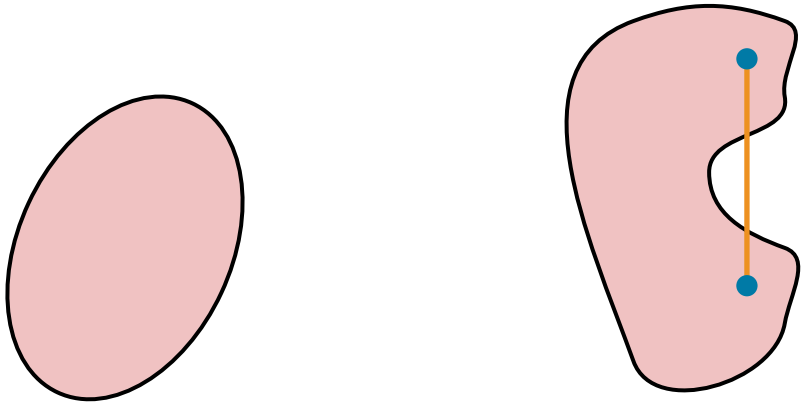
Programm 19.4: `kapitel_19/dijkstra/dijkstra_main.c`

```
1  * gcc dijkstra_main.c dijkstra.c network.c heap.c -o dijkstra
2  * Aufruf mit
3  * ./dijkstra in_network 0
4  ************************************/
5
6  #include <stdio.h>
7  #include <stdlib.h>
```

```c

#include "network.h"
#include "dijkstra.h"

int main(int argc, char** argv) {

    if ( argc < 2 ) {
        fprintf(stderr, "Netwerk und Startknoten muessen angegeben werden\n");
        exit(1);
    }

    char * infile = argv[1];
    int start = atoi(argv[2]);

    struct network * g = read_network(infile);

    // run algorithm
    dijkstra(g,start);

    // print distance for each node
    printf("The nodes have distance \n");
    for ( int i = 0; i < g->n_nodes; ++i ) {
        printf("%d: %d\n", i, g->nodes[i]->dist);
    }

    // get predecessors
    int * pred = (int *)malloc(g->n_nodes * sizeof(int));
    predecessors(g,pred);

    printf("The shortest path tree is given by the edges\n");
    for ( int i = 0; i < g->n_nodes; ++i ) {
        if ( pred[i] != -1 ) {
            printf("(%d, %d)\n", pred[i],i);
        }
    }

    free(pred);
    free_network(g);

    return 0;
}
```

Wir können dieses Programm für unser Beispiel ausführen, wobei wir den Graph aus Abbildung 19.4 in einer Datei `in_graph` abgespeichert haben.

```
./dijkstra in_graph 0
The nodes have distance
0: 0
1: 6
2: 4
3: 8
4: 9
5: 11
6: 8
7: 10
The shortest path tree is given by the edges
(2, 1)
(0, 2)
(1, 3)
```



(a) Eine konvexe Menge

(b) Eine nichtkonvexe Menge. Die Verbindung der zwei blauen Punkte verläuft nicht vollständig in der Menge.

Abbildung 19.5: Eine konvexe und eine nicht-konvexe Menge.

```
(3, 4)
(7, 5)
(1, 6)
(3, 7)
```

Wenn Sie die Korrespondenz zwischen unseren mit Buchstaben bezeichneten Knoten und den Indizes wieder auflösen (r entspricht 0, a bis g den Indizes 1 bis 7), sehen sie genau die kürzesten Wege, die wir schon in Abbildung 19.3 gesehen haben.

19.2 Konvexe Hülle in der Ebene

In diesem Abschnitt wollen wir uns ein geometrisches Problem in der Ebene anschauen, das Grundlage für viele weitere praktische Algorithmen, zum Beispiel bei der Bewegungsplanung in der Robotik, oder bei der Projektion und Darstellung von dreidimensionalen Ansichten auf einem Bildschirm ist.

Wir betrachten hier eine endliche Menge $P = \{p_1, \ldots, p_n\}$ von Punkten $p_i = (x_i, y_i)$ für $x_i, y_i \in \mathbb{R}$ und $1 \leq i \leq n$ von $n \in \mathbb{Z}_{\geq 0}$ Punkten in der Ebene $\mathbb{R}^2$, die uns mit kartesischen Koordinaten gegeben sind. Wir wollen die *konvexe Hülle* $\mathsf{conv}(P)$ dieser Punkte bestimmen. Das ist die kleinste konvexe Menge $M \subseteq \mathbb{R}^2$ (in der Regel nicht endlich), die alle Punkte aus P enthält. Dabei nennen wir eine Menge *konvex*, wenn mit zwei beliebigen Punkten immer auch ihre Verbindungsstrecke in der Menge enthält. Abbildung 19.5 zeigt ein Beispiel.

In dieser Definition ist eine Menge S *kleiner* als eine Menge T, wenn S vollständig in T enthalten ist. Diese Relation ist nur eine *partielle* Ordnung auf den Teilmengen von $\mathbb{R}^2$. Trotzdem ist die Menge K eindeutig. Wenn nämlich S_1 und S_2 zwei Mengen sind, die P enthalten, dann trifft das auch auf ihren Schnitt $S := S_1 \cap S_2$ zu, und S ist eine Teilmenge von S_1 und S_2.

Wir können uns leicht überlegen, dass es immer eine beschränkte konvexe Menge

gibt, die alle unsere Punkte enthält. Wenn nämlich $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$ das Minimum bzw. Maximum über alle x- bzw. y-Koordinaten unserer Menge ist, ist zum Beispiel das Rechteck mit den Ecklen $(x_{\min}, y_{\min})$, $(x_{\max}, y_{\min})$, $(x_{\max}, y_{\max})$ und $(x_{\min}, y_{\max})$ eine solche Menge.

In der Ebene können wir uns anschaulich die konvexe Hülle einer endlichen Punktmenge auch als diejenige Menge vorstellen, deren Rand wir erhalten, wenn wir uns unsere Punkte als eingesteckte Nadeln in der Ebene vorstellen, einen Faden als Schlaufe um alle Punkte spannen und die Schlaufe fest zuziehen. Intuitiv wird der Faden an einigen der Nadeln nachher anliegen, unsere konvexe Hülle ist also ein Polygon. Das können wir zum Beispiel an der Punktmenge aus Abbildung 19.6 sehen. Ihre konvexe Hülle ist das Polygon mit 10 Ecken aus Abbildung 19.7.

Die konvexe Hülle K einer endlichen Menge von Punkten P ist tatsächlich immer ein *Polygon*, eine von einem Kantenzug aus endlich vielen Segmenten begrenzte Fläche. Die Endpunkte der Segmente sind Punkte aus unserer Menge. Das können wir uns auf die folgende Weise überlegen. Wenn wir eine beliebige (affine) Gerade G in der Ebene betrachten, dann können wir die Punktmenge in drei Teile zerlegen, nämlich die Punktmenge G_0 der Punkte, die auf der Geraden liegen, und die beiden Mengen G_+ und G_- der Punkte, die ober- bzw. unterhalb davon liegen (dafür müssen wir der Geraden eine (beliebige) Orientierung geben. Wenn wir die Gerade verschieben, wird irgendwann eine der Mengen G_+ oder G_- leer sein, und die beiden anderen Punkte enthalten. Wenn G_0 nur einen Punkt enthält, dann können wir die Gerade ein bisschen um diesen Punkt drehen, ohne dass Punkte von einer in eine andere Menge wechseln. Wenn G_0 zwei oder mehr Punkte enthält, dann geht das nicht mehr. Wir nehmen nun alle Geraden, die in diesem Prozess zwei oder mehr Punkte enthalten. Davon gibt es offensichtlich nur endlich viele, und wir betrachten zu jeder das kleinste Segment, das alle Punkte auf der Geraden enthält. Um einen Endpunkt des Segments können wir wieder die Gerade so lange (in eine der beiden Richtungen) drehen, bis wir auf einen zweiten Punkt aus der Punktmenge treffen. Damit haben wir wieder eine Gerade die zwei (oder mehr) Punkte aus unserer Menge enthält, und wir müssen ein anderes unserer Segmente gefunden haben. Also haben wir den gesamten Rand unserer konvexen Hülle gefunden, und die Ecken dieses Polygons sind Punkte aus unserer Menge.

Wir wollen hier nun einen Algorithmus entwickeln, der uns diese konvexe Hülle ausrechnet. Betrachten wir dafür noch einmal die Punktmenge aus Abbildung 19.6 und ihre konvexe Hülle. Damit das Polygon um eine Ecke herum konvex ist, muss der Winkel, den die beiden an der Ecke benachbarten Segmente bilden, kleiner als 180 Grad sein. Das können wir auch auf die folgende Weise ausdrücken. Wenn wir in unserem Polygon gegen die Uhrzeigerrichtung auf dem Rand entlang laufen, dann machen wir an jeder Ecke des Polygons einen Knick nach *links*. An keiner Stelle biegen wir an einer Ecke nach *rechts* ab.

Diese Erkenntnis wollen wir als Grundlage unseres Algorithmus nehmen. Dabei wollen wir an einer beliebigen Ecke des Polygons starten[5] und entlang eines Randsegments zum nächsten Punkt aus unserer Menge laufen. An diesem Punkt wissen wir, dass das

[5] Bei den theoretischen Vorüberlegungen. In der Umsetzung wird es sinnvoll sein, einen Punkt auszuzeichnen und dort zu starten, nämlich den Punkt mit kleinster x-Koordinate unter allen Punkten mit kleinster y-Koordinate.

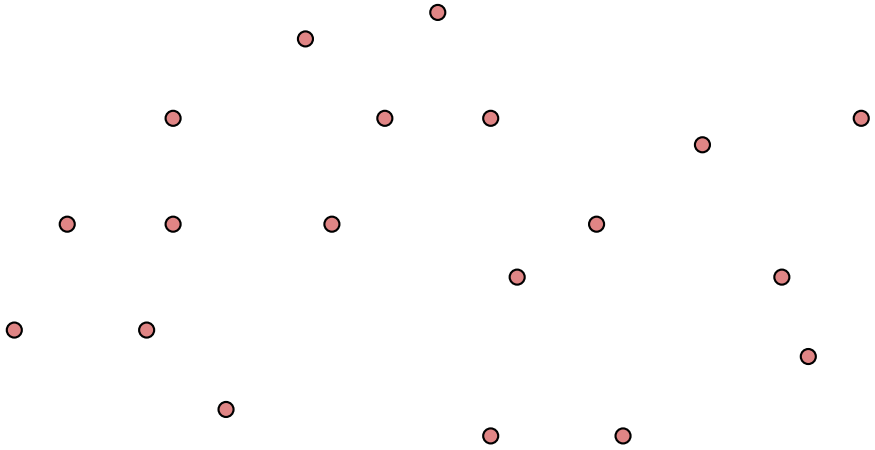


Abbildung 19.6: Eine Punktmenge in der Ebene.

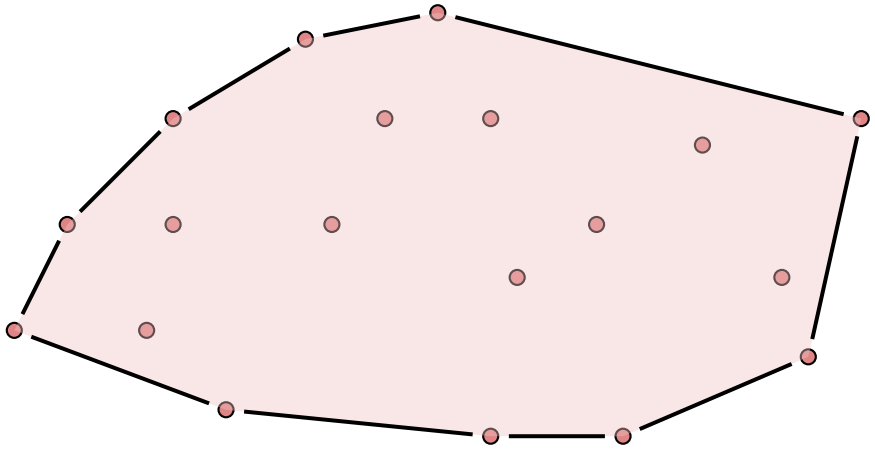


Abbildung 19.7: Die Punktmenge und ihre konvexe Hülle, ein Polygon mit 10 Ecken.

nächste Segment nach links abbiegen muss. Um die Richtung des neuen Segments zu finden, drehen wir uns nun so lange aus unserer bisherigen Laufrichtung nach links, bis wir den nächsten Punkt aus unserer Menge sehen. Wenn wir diesen Punkt gefunden haben, dann laufen wir in dieser Richtung bis zu dem Punkt und haben ein weiteres Randsegment des Polygons erhalten.

Wenn Sie versuchen, mit dieser Idee einen Algorithmus aufzuschreiben, dann fällt Ihnen wahrscheinlich auf, dass wir (mindestens) noch zwei Stellen in der Methode näher betrachten müssen. Zum einen starten wir damit, dass wir in Richtung eines Segments gehen, um dann durch Drehen nach links das nächste zu funden. Am Anfang eines möglichen Algorithmus kennen wir allerdings noch kein Segment auf dem Rand unseres Polygons. Zum anderen müssen wir uns überlegen, was es heißt, sich an einer Ecke nach links zu drehen, bis wir den nächsten Eckpunkt sehen, und wie wir das in eine algorithmische Form bringen wollen, die ein Computer für uns ausführen kann.

Um an ein erstes Segment zu kommen, wollen wir uns zuerst überlegen, wie wir eine erste *Ecke* auf dem Rand bekommen können. Dafür können wir zum Beispiel einen Punkt aus unserer Punktmenge nehmen, der die kleinste y-Koordinate hat. Es könnte mehr als einen solchen Punkt geben. Wenn wir in dann unter diesen Punkten auch noch

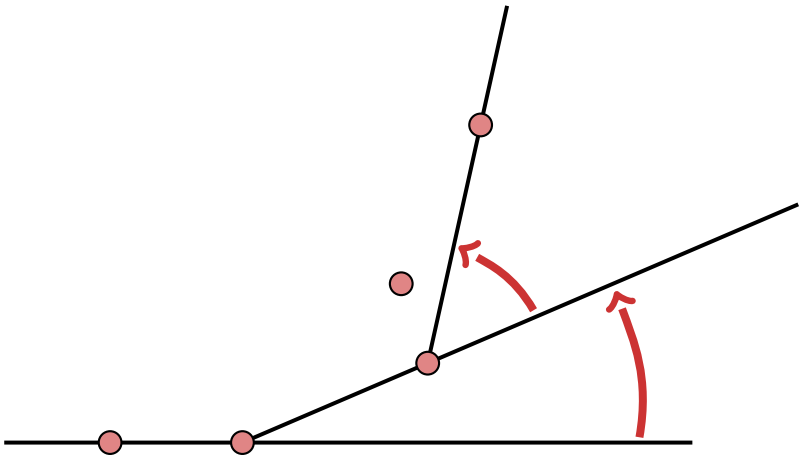


Abbildung 19.8: Auffinden des Rands: An jeder neuen Ecke drehen wir so lange nach links, bis wir wieder einen Punkt aus unserer Menge sehen.

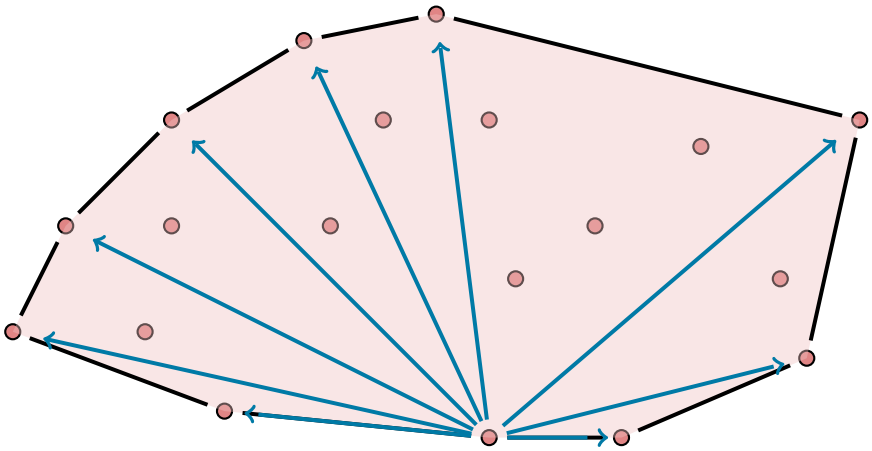


Abbildung 19.9: Eine Punktmenge in der Ebene mit ihrer konvexen Hülle und Strahlen vom Startpunkt s zu allen anderen Ecken.

den mit der kleinsten x-Koordinate auswählen, haben wir sicherlich eine erste Ecke gefunden (die anderen Punkte mit kleinster y-Koordinate liegen sicherlich mindestens auf dem Rand des Polygons, aber wenn es noch mehr als einer ist, dann sind nicht alle dieser Punkte auch Ecken.). Diesen ausgezeichneten Punkt nennen wir im Folgenden unseren *Startknoten* s und bestimmen ihn ganz am Anfang der Berechnung.

Von dieser Ecke aus können wir jetzt Strahlen zu allen anderen Ecken des Polygons ziehen. Ein Beispiel ist in Abbildung 19.9. Wenn wir nun in s die ausgehenden Strahlen gegen den Uhrzeigersinn vom am weitesten rechts abgehenden Strahl durchgehen, dann sehen wir die Strahlen in genau der Reihenfolge, wie die Ecken, zu denen sie gehören, auf dem Rand aufeinanderfolgen. Insbesondere muss also der am weitesten rechts beginnende Strahl zur nächsten Ecke im Polygon führen. Damit können wir unser erstes Anfangssegment gewinnen.

Wir sehen daran auch, dass wir die nächste Ecke zu einer gerade gefundenen Ecke v auch finden können, indem wir den Endpunkt des nächsten Strahls von s entgegen dem Uhrzeigersinn nach dem zu v nehmen. Dessen Endpunkt w ist unsere nächste Ecke.

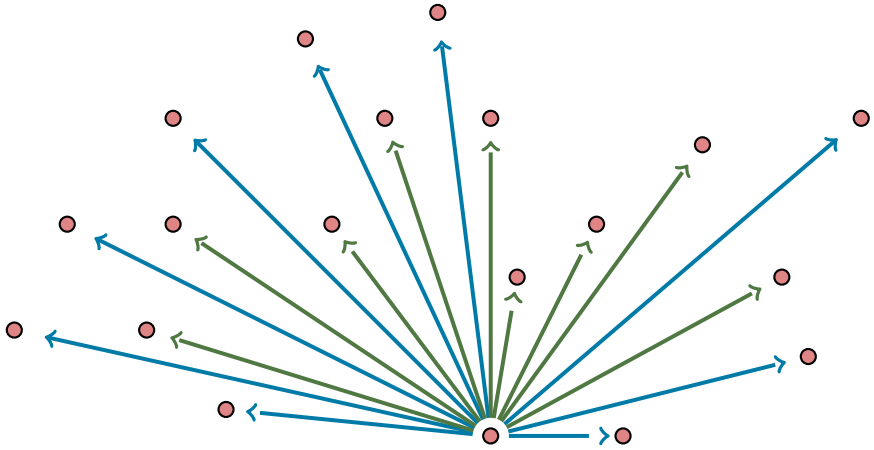


Abbildung 19.10: Eine Punktmenge in der Ebene mit allen Strahlen von s.

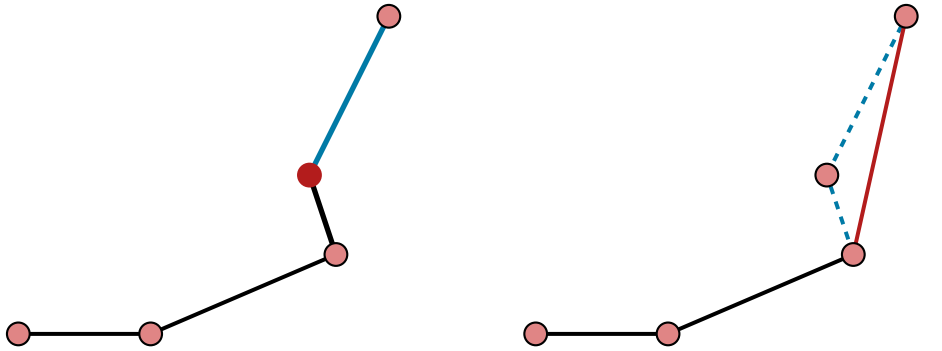
Damit das in dieser Form funktioniert, müssten wir aber schon wissen, welche der Punkte unserer Ausgangsmenge Ecken sind, und welche nachher im Inneren liegen. Wenn wir das aber schon wüssten, dann bräuchten wir unseren Algorithmus nicht. Wir können also nur die Strahlen zu allen Punkten unserer Menge einzeichnen, und müssen im Lauf des Algorithmus entscheiden, welche wir brauchen und welche nicht. Dabei bleibt aber zumindest erhalten, dass der am weitesten rechts liegende Strahl zur nächsten Ecke zeigt, und dass die Ecken immer noch beim Ablaufen der Strahlen gegen den Uhrzeigersinn in der richtigen Reihenfolge nacheinander kommen, wenn auch vielleicht nicht mehr direkt aufeinanderfolgend. Siehe Abbildung 19.10.

Wir wollen trotzdem nach der Methode vorgehen, die wir uns am Anfang überlegt haben. Wir stehen also auf einem Punkt unserer Menge und suchen nach der nächsten Ecke, oder einem Punkt, den wir dafür halten. Die Idee des Algorithmus ist nun, dass wir an dieser Stelle einfach *annehmen*, dass der Endpunkt des nächsten Strahls auch der Endpunkt der nächsten Ecke ist, und den Algorithmus damit fortsetzen, also zu diesem Punkt laufen und dort die nächste Ecke suchen.

Natürlich müssen wir irgendwann entscheiden, ob der Punkt doch im Inneren des Polygons liegt. Dafür haben wir jedoch, wenn wir einmal in dem Punkt stehen, ein einfaches hinreichendes (aber nicht notwendiges) Kriterium. Wenn wir nämlich von diesem Punkt zum nächsten Endpunkt eines Strahls laufen wollen und müssten dafür an diesem Punkt nach *rechts* statt nach *links* drehen, dann ist unser Punkt keine Ecke, und wir können ihn verwerfen und nehmen den nächsten Punkt als eine mögliche nächste Ecke. Abbildung 19.11 zeigt ein Beispiel.

In diesem Beispiel mussten wir genau einen Knoten verwerfen und zu seinem Vorgänger zurückgehen. Man kann sich leicht überlegen, dass das nicht reicht. Es könnte auch am vorhergehenden Knoten noch sein, dass wir eine *Rechts*drehung machen müssen um den nächsten Knoten zu erreichen. Dann müssen wir weiter zurückgehen, bis wir einen Knoten erreichen, von dem aus wir wieder eine *Links*drehung machen. Dieses Vorgehen bezeichnet man oft als *backtracking*. Das linke Bild in Abbildung 19.12 zeigt ein Beispiel. im rechten Bild können Sie auch sehen, dass wir in mehreren solchen *backtracking*-Schritten beim gleichen Knoten auskommen können.

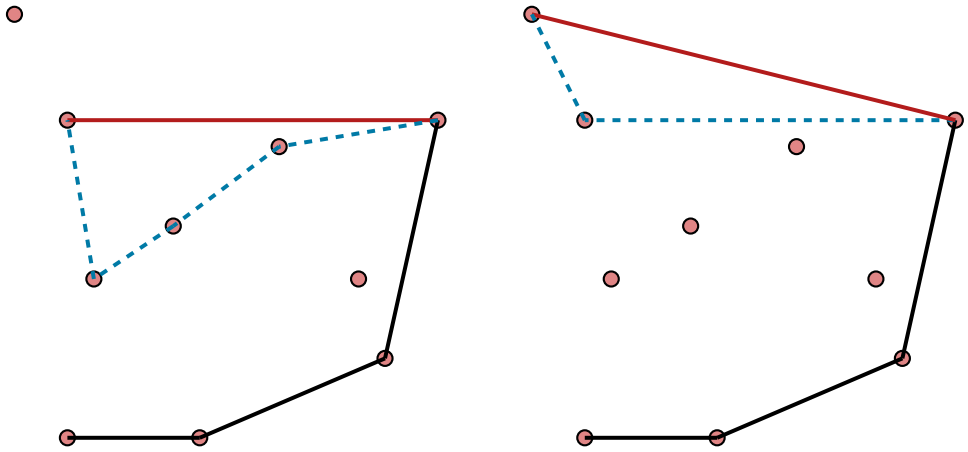
Einen Sonderfall bei der Entscheidung, ob ein Punkt eine Ecke ist oder nicht, sollte



(a) Am vorletzten Punkt drehen wir uns mit dem blauen Segment nach *rechts*. Der rote Punkt liegt daher nicht auf dem Rand.

(b) Wir ersetzen also die beiden gestrichelten Segmente durch das rote Segment.

Abbildung 19.11: Erkennen innerer Punkte.



(a) Es kann sein, dass wir mehr als einen Punkt zurückgehen müssen, bis wir einen Punkt haben, von dem aus wir wieder eine Linksdrehung machen.

(b) Hier erreichen wir im Rückwärtsabsuchen der Ecken ein zweites mal den gleichen Punkt.

Abbildung 19.12: Manchmal müssen wir mehr als einen Punkt zurückgehen.

man nun noch bemerken, bevor wir uns an die Umsetzung machen. Es könnte sein, dass wir an einem Punkt stehen, wo wir zum nächsten Punkt keine Drehung machen müssen, sondern einfach geradeaus weiterlaufen können. Diesen Fall wollen wir wie den Fall innerer Punkte behandeln und den Punkt nicht als Ecke unseres Polygons ansehen.

Wir können die Strahlen von s zu den anderen Punkten leicht entgegen des Uhrzeigersinns sortieren, wenn wir ihren Winkel zur x-Achse betrachten, also den Winkel des Strahls mit dem Einheitsvektor e_1 bestimmen. Wir haben aber bisher unterschlagen, dass in der Sortierung nach Winkeln auch zwei Punkte den gleichen Winkel haben können. In dem Fall liegen beide Punkte auf dem gleichen von s ausgehenden Strahl. Wenn wir, wie bisher zum Beispiel im Kapitel 11, nur eine Liste von Zahlen sortieren, dann ist es egal, in welcher Reihenfolge wir diese gleichen Zahlen in die Liste einsortieren, da sie am Ende nicht unterscheidbar sind. Das ist bei den Punkten unserer Punktmenge nicht richtig. Die beiden Punkte sind verschieden, auch wenn sie von s aus gesehen in

einer Sichtachse liegen. Wenn Sie sich zurückerinnern, dass wir den Rand des Polygons durch sukzessives Ablaufen der Segmente bekommen wollen, dann ist es auch relevant, welcher der beiden Punkte zuerst kommen. Wenn sie nämlich den Rand gegen den Uhrzeigersinn ablaufen, dann kommt immer der näher an s liegende Punkt zuerst (sofern er auf dem Rand liegt). Unsere Ordnung auf den Punkten durch den Winkel zur x-Achse ist also keine vollständige Ordnung. Wenn wir gleiche Winkel haben, dann brauchen wir also als weiteres Kriterium, um zu entscheiden, welcher Punkt zuerst kommt, den Abstand von s. Das wird uns bei der konkreten Implementierung und der Wahl der Funktion zum Sortieren noch einmal beschäftigen.

Zudem machen wir an einem Punkt mit unseren Segmenten genau dann eine Drehung nach links, wenn der eingeschlossene Winkel zwischen 0 und π liegt, sein $\sin$ also positiv ist. Damit können wir in Algorithmus 19.2 unseren Algorithmus in Pseudocode formulieren.

Algorithmus 19.2: Graham Scan

Eingabe : Startpunkt s
Punkteliste $P = (p_0, p_1, \ldots, p_k)$, aufsteigend sortiert nach Winkel $\angle(e_1, \overline{sp_i})$

Ausgabe : Punkteliste $q = (q_0, q_1, \ldots, q_r)$, die die Konvexe Hülle für P bildet

```
1  Q_0 ← s
2  Q_1 ← P_0
3  n ← 1                                          // letztes Element in Q
4  i ← 1                                          // nächstes Element in P
5  für jedes i ≤ |P|, i monoton steigend tue
6  |  q ← p_i
7  |  solange sin ∠(q_{n-1}q_n, q_{n-1}q) ≤ 0 und n > 0 tue
8  |  |  Lösche q_n
9  |  |  n ← (n − 1)
10 |  n ← (n + 1)
11 |  q_n ← q
```

Für die Umsetzung in Code müssen wir uns überlegen, wie wir die Punkte in der Ebene eingeben. Die erste Feststellung hier ist, dass wir hier nur rationale Koordinaten zulassen können, da wir beliebige reelle Zahlen nicht exakt im Computer darstellen können[6]. Um direkt mit den Typen aus `C` rechnen zu können und nicht auf die `gmp` zurückgreifen zu müssen, wollen wir sogar annehmen, dass alle unsere Koordinaten ganzzahlig sind. Da unter Skalierung invariant ist, welche Punkte auf dem Rand eines Polygons und welche im Inneren liegen, können wir dann auch mit rationalen Punkten rechnen, wenn wir vor der Eingabe mit dem kleinsten gemeinsamen Nenner skalieren. Wir schränken uns damit also nicht wirklich ein[7].

[6] Das ist so etwas zu pauschal formuliert, wir können auch manche andere Zahlen exakt darstellen und damit rechnen, zum Beispiel mit Zahlen aus Körpererweiterungen der rationalen Zahlen. Aber das ist deutlich aufwändiger und für diese Anwendung zu kompliziert.

[7] Das ist auch wieder nicht ganz richtig, denn unter Umständen ist der gemeinsame Nenner sehr groß, so dass wir zum einen mit viel größeren Zahlen rechnen müssen, zum anderen könnte es sein, dass die Skalierung den Bereich von `int` überschreitet, obwohl unsere Ausgangskoordinaten sehr klein waren.

Schwieriger ist die Frage, wie wir mit der Berechnung der Winkel umgehen. Die Winkel zwischen einem der Strahlen und der x-Achse werden in der Regle nicht ganzzahlig (oder wenigstens rational) sein. Für eine direkte Darstellung müssten wir daher auf den Typ `float` ausweichen. Damit können wir die Winkel jedoch nicht exakt speichern, so dass wir Probleme mit Punkten bekommen könnten, deren Strahlen dicht beieinander liegen (und auch bekommen, wie Sie leicht feststellen können, wenn Sie den Algorithmus mit Winkeln im Typ `float` implementieren und eine passende Menge Punkte vorgeben).

Wir wollen daher auf eine Darstellung ausweichen, die ebenso wie die Winkel zur x-Achse monoton wächst und die wir für unsere Sortierung heranziehen können, die aber mit Zahlen vom Typ `int` gespeichert werden kann. Im ersten Schritt stellen wir fest, dass unsere Winkel alle zwischen 0 und π liegen, und wir daher statt des Winkels auch den *Kosinus* des Winkels betrachten können, der in diesem Bereich injektiv und strikt monoton fallend ist. Zudem sollten wir uns an die Formel

$$\cos(\angle(u,v)) \;=\; \frac{\langle u,v\rangle}{\|u\|\|v\|}$$

für zwei Vektoren u und v und den von den dadurch definierten Geraden eingeschlossenen Winkel $\angle(u,v)$ erinnern. Die Norm eines Vektors bestimmt sich aus $\|u\| = \sqrt{u_1^2+u_2^2}$ und ist damit leider meistens auch nicht rational. Aber ihr Quadrat ist rational (jedenfalls wenn, wie wir angenommen haben, die Koordinaten ganzzahlig sind). Das Quadrat einer monotonen Funktion ist immer noch monoton, solange sie das Vorzeichen nicht wechselt. Das passiert zwar beim *Kosinus*, aber wir können das Vorzeichen vor dem Quadrieren abspalten und nachher wieder hinzufügen. Der Wert ist rational, wir speichern also für die Angabe eines Winkels Zähler und Nenner getrennt als

$$\begin{aligned}\text{num} &:= \epsilon\langle u,v\rangle^2\\ \text{denom} &:= \|u\|\|v\|\,,\end{aligned}$$

wobei ϵ je nach Vorzeichen von $\langle u,v\rangle$ den Wert 1 oder -1 hat. Da wir für die Sortierung lieber eine monoton steigende als eine monoton fallende Funktion haben, drehen wir hier auch noch das Vorzeichen rum. Da einer der Vektoren der erste Einheitsvektor ist, vereinfacht sich in der konkreten Anwendung die Formel auch noch etwas.

Damit können wir schon die Struktur unserer Punktmenge festlegen.

```
1  struct point {
2      int x;
3      int y;
4      int angle_num;
5      int angle_denom;
6  } ;
7
8  struct pointSet {
9      int n_points;
10     struct point* points;
11 } ;
```

Zum Sortieren der Punkte nach Winkel brauchen wir eine Vergleichsfunktion. Dabei betrachten wir zuerst den Winkel zwischen zwei Punkten u und v, und, wenn diese gleich sind, den Abstand zu s. Das heißt aber, dass, anders als bisher, in der Vergleichsfunktion

nicht nur die direkt zu vergleichenden Punkte u und v bekannt sein müssen, sondern auch der Punkt s, der also immer an die Vergleichsfunktion übergeben werden muss. Damit könnte unsere Vergleichsfunktion so aussehen.

```c
int compare(const void* a, const void* b, void* c) {
   const struct point* p1 = (const struct point*)a;
   const struct point* p2 = (const struct point*)b;

   // Die Winkel sind als rationale Zahlen gegeben
   // Wir multiplizieren aus fuer den Vergleich
   int c1 = p1->angle_num*p2->angle_denom;
   int c2 = p2->angle_num*p1->angle_denom;

   if ( c1 == c2 ) {
     // Die Winkel sind gleich
     // Wir vergleichen zuerst die Differenz in x-Richtung
     const struct point* start = (const struct point*)c;
     int xdiff1 = p1->x - start->x;
     int xdiff2 = p2->x - start->x;
     if ( xdiff1 != xdiff2 ) {
         return ( xdiff1 > xdiff2 ) - ( xdiff2 > xdiff1 );
      }  else {
        // wenn die x-Richtung übereinstimmt, dann liegen die Punkte
        // senkrecht ueber start, wir vergleichen y-Koordinaten
         int ydiff1 = p1->y - start->y;
         int ydiff2 = p2->y - start->y;
         return ( ydiff1 > ydiff2 ) - ( ydiff2 > ydiff1 );
      }
   }
   return (c1 > c2) - (c2 > c1);
}
```

`C` bietet mit `qsort_r` eine passende Funktion zum Sortieren mit *QuickSort*, die eine Vergleichsfunktion in dieser Form übernehmen kann. *QuickSort* ist ein weiterer Sortieralgorithmus, der ähnlich zu *MergeSort* funktioniert. Seine Laufzeit im schlechtesten Fall ist mit $\mathcal{O}(n^2)$ zwar schlechter als die von *MergeSort*, aber im Erwartungswert sind sie beide gleich gut. Anders als *MergeSort* ist *QuickSort* als Funktion in vielen Bibliotheken vorhanden und beliebt. Da wir selbst keine Version von *MergeSort* geschrieben haben, die mit einer erweiterten Vergleichsfunktion zurechtkommt, wie wir sie hier brauchen, wollen wir auf die Implementierung von *QuickSort* in `C` zurückgreifen.

Leider ist die Signatur dieser Funktion nicht einheitlich in den verschiedenen auf Unix basierenden Systemen. Wir müssen also unterschiedliche Signaturen für verschiedene Systeme vorhalten. Wir unterscheiden im Folgenden zwischen der Umsetzung in MacOS und den meisten Linux-Distributionen. Das können wir über eine Abfrage im Präprozessor lösen, da die verschiedenen Systeme automatisch eine Variable setzen, anhand derer wir sie unterscheiden können. Damit können wir die Signatur zum Beispiel wie folgt schreiben.

```c
#ifdef __APPLE__
int compare(void* c, const void* a, const void* b) {
#else
int compare(const void* a, const void* b, void* c) {
#endif
```

Dabei wird durch den `gcc` auf der Plattform *MacOS* die Variable `__APPLE__` gesetzt.

Wie fast alle vom System gesetzten Variablen fängt sie mit einem Unterstrich an. Das sollte man bei Namen für eigene Präprozessorvariablen vermeiden um Konflikte zu vermeiden.

Damit können wir schon ein erstes Modul mit Funktionen für unsere Punktmenge schreiben. Neben den Strukturen für Punkte und Punktmengen und unserer Sortierfunktion fassen wir Funktionen zur Ein- und Ausgabe zusammen. Außerdem nehmen wir eine Funktion auf, die die Winkel bestimmt und Punkte unserer Punktmenge vertauschen kann.

Programm 19.5: `kapitel_19/graham_scan/point_set.h`

```
#ifndef POINT_SET_H
#define POINT_SET_H

struct point {
    int x;
    int y;
    int angle_num;
    int angle_denom;
} ;

struct pointSet {
    int n_points;
    struct point* points;
} ;

#ifdef __APPLE__
int compare(void*, const void*, const void*);
#else
int compare(const void*, const void*, void*);
#endif

// read from file
struct pointSet* read_input(char*);

// store the vertices of the convex hull in a file
// p original point set
// v array of indices of the vertices among p
// nv size of nv
// f filename
void store_verts(struct pointSet* p, int* v, int nv, char* f);

// swap two points in a point set given by indices
void swap(struct pointSet* , int, int);

// compute the angles of all points in a pointset
// of the ray from the first point to all other points
// to the x-axis
void angles(struct pointSet* );

void print_points(struct pointSet*,int);

void free_pointset(struct pointSet*);

#endif
```

Für die Ausführung dieser Funktionen brauchen wir noch Funktionen, die einen einzel-

nen Punkt ausgibt und einen einzelnen Punkt freigibt, die wir aber nicht zur allgemeinen Verwendung exportieren müssen. Ebenso müssen wir Winkel nicht einzeln berechnen, so dass nur `angles()` allgemein verfügbar sein muss.

Programm 19.5:kapitel_19/graham_scan/point_set.c

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

#include "point_set.h"

#ifdef __APPLE__
int compare(void* c, const void* a, const void* b) {
#else
int compare(const void* a, const void* b, void* c) {
#endif

   const struct point* p1 = (const struct point*)a;
   const struct point* p2 = (const struct point*)b;

   // Die Winkel sind als rationale Zahlen gegeben
   // Wir multiplizieren aus fuer den Vergleich
   int c1 = p1->angle_num*p2->angle_denom;
   int c2 = p2->angle_num*p1->angle_denom;

   if ( c1 == c2 ) {
     // Die Winkel sind gleich
     // Wir vergleichen zuerst die Differenz in x-Richtung
     const struct point* start = (const struct point*)c;
     int xdiff1 = p1->x - start->x;
     int xdiff2 = p2->x - start->x;
     if ( xdiff1 != xdiff2 ) {
         return ( xdiff1 > xdiff2 ) - ( xdiff2 > xdiff1 );
      }  else {
         // wenn die x-Richtung übereinstimmt, dann liegen die Punkte
         // senkrecht ueber start, wir vergleichen y-Koordinaten
         int ydiff1 = p1->y - start->y;
         int ydiff2 = p2->y - start->y;
         return ( ydiff1 > ydiff2 ) - ( ydiff2 > ydiff1 );
      }
   }
   return (c1 > c2) - (c2 > c1);
}

// swap two points in the list
void swap(struct pointSet* pointset, int i1, int i2) {
   struct point tmp = pointset->points[i1];
   pointset->points[i1] = pointset->points[i2];
   pointset->points[i2] = tmp;
}

// compute a representative of the angle for the vector from p2 tp p1 with the x-axis
// we use the negative of the squared cosine,
// where we preserve the sign of cos before we square
// this is a rational number, which we store as (num,denom)
// special case: start node, make it smallest, so it comes first
void angle(struct point* p1, struct point* p2, int* num, int* denom) {
```

```
53     int dot = p2->x-p1->x;
54     *num = (dot<0?-1:1)*dot*dot;
55     *denom = (p2->x-p1->x)*(p2->x-p1->x) + (p2->y-p1->y)*(p2->y-p1->y);
56     if ( *num == 0 && *denom == 0 ) {
57        *num = -1;
58        *denom = -1;
59     }
60  }
61
62  // return list of all angles of
63  // a ray from the start node to another node with the x-axis
64  void angles(struct pointSet* pointset) {
65     int num, denom;
66     (pointset->points[0]).angle_num   = -1;
67     (pointset->points[0]).angle_denom =  1;
68
69     struct point* l = pointset->points;
70
71     for(int i = 1; i< pointset->n_points; i++){
72        struct point* p = pointset->points+i;
73        angle(p,l,&num,&denom);
74        p->angle_num = num;
75        p->angle_denom = denom;
76      }
77  }
78
79  // read a list of points in the plane
80  // with integral coeffs
81  // each line gives coords (x,y) for one point
82  struct pointSet* read_input(char* filename) {
83     FILE *infile;
84     infile = fopen (filename, "r");
85     if (infile == NULL) {
86        fprintf(stderr, "\nError opening file\n");
87        exit (1);
88     }
89
90     struct pointSet* points = (struct pointSet*)malloc(sizeof(struct pointSet));
91
92     const char  s[2] = " ";
93     char * line = NULL;
94     size_t len = 0;
95     size_t read;
96     char *next;
97
98     read = getline(&line, &len, infile);
99     points->n_points = atoi(line);
100    points->points = malloc(points->n_points*sizeof(struct point));
101    int i = 0;
102    struct point* point = points->points;
103    while ((read = getline(&line, &len, infile)) != -1) {
104        next = strtok(line, s);
105        if ( !next || strcmp(next,"\n")==0 ) {
106           continue;
107        }
108        point->x = atoi(next);
109        next = strtok(NULL, s);
110        point->y = atoi(next);
```

```
111         point++;
112         i++;
113     }
114
115     if( line ) {
116        free(line);
117     }
118
119     fclose(infile);
120
121     return points;
122 }
123
124 void print_point(struct point* point, int index, int show_angle ) {
125     if ( show_angle ) {
126        printf("%d: (%d %d) -- angle %f\n", index, point->x, point->y, (float)(point->
       angle_num)/point->angle_denom);
127     }  else {
128        printf("%d: (%d %d)\n", index, point->x, point->y);
129     }
130 }
131
132 void print_points(struct pointSet* pointset, int show_angle ) {
133     printf("--------------------------------\nPoint set with %d points\n", pointset->
       n_points);
134     struct point* point = pointset->points;
135     for(int i = 0; i < pointset->n_points; i++){
136        print_point(point, i, show_angle);
137        point++;
138     }
139     printf("--------------------------------\n");
140 }
141
142 // write result to a file
143 // gives a list of points on the convex hull
144 void store_verts(struct pointSet* pointset, int* verts, int n_verts, char* filename) {
145     FILE* file = fopen(filename, "w");
146     fprintf(file, "%d\n", n_verts);
147     for(int i = 0; i < n_verts; i++) {
148        struct point* point = pointset->points+verts[i];
149        fprintf(file, "%d %d\n", point->x, point->y);
150     }
151     fclose(file);
152 }
153
154 void free_point(struct point* point) {
155     free(point);
156 }
157
158 void free_pointset(struct pointSet* pointset) {
159     free(pointset->points);
160     free(pointset);
161 }
```

Nun können wir an die eigentlich Umsetzung unsere Algorithmus gehen. Wir müssen nur eine Funktion exportieren, so dass der Header sehr einfach ist.

Programm 19.6: `kapitel_19/graham_scan/graham_scan.h`

```c
#ifndef GRAHAM_SCAN_H
#define GRAHAM_SCAN_H

#include "point_set.h"

// do a graham scan on the set of points
// given in the point set
// returns indices of points on the convex hull
// and total number of vertives
// in the last two arguments
void graham_scan(struct pointSet*, int*, int*);

#endif
```

Nach den Vorbereitungen treten bei der Umsetzung der Funktion keine Probleme mehr auf. Der einzige Punkt, den man sich überlegen muss ist die Frage, wie wir erkennen, ob wir eine Rechts- oder Linksdrehung an einem Punkt machen. Dafür erinnern wir uns daran, dass wir erkennen können, ob die von zwei Vektoren u und v aufgespannte Basis mit oder gegen den Uhrzeigersinn orientiert ist, indem wir die Determinante der von den zwei Vektoren gebildeten Matrix bestimmen. Das Vorzeichen zeigt die Orientierung an.

Dabei müssen wir in unserem Fall den richtigen Bezugspunkt nehmen. Wenn wir für drei Punkte p_1, p_2, und p_3 bestimmen wollen, ob wir bei der Abfolge der Segmente von p_2 zu p_1 und p_1 zu p_3 eine Linksdrehung machen, müssen wir daher die Determinante zwischen $p_3 - p_1$ und $p_2 - p_1$ bestimmen. Das macht in unserem Programm die Funktion `ccw`.

Programm 19.6:kapitel_19/graham_scan/graham_scan.c

```c
#ifdef __linux__
#define _GNU_SOURCE
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#include "graham_scan.h"

// check for left turn
// by computing det of two vectors -> orientation
int ccw(struct point* p1, struct point* p2, struct point* p3) {
   int turn = (p2->x - p1->x)*(p3->y - p1->y) - (p2->y - p1->y)*(p3->x - p1->x);
   return turn;
}

// among those with minimal y finde the one with minimal x
int find_min(struct pointSet* pointset) {
   struct point* point = pointset->points;
   struct point* tmp   = pointset->points;
   int index = 0;

   for(int i = 0; i < pointset->n_points; i++) {
      if( point->y < tmp->y ) {
         tmp = point;
```

```
28            index = i;
29         }  else if( point->y == tmp->y ) {
30            if( point->x < tmp->x ) {
31               tmp = point;
32               index = i;
33            }
34         }
35         point++;
36      }
37      return index;
38   }
39
40   void graham_scan(struct pointSet* pointset, int* verts, int* n_verts) {
41
42      // some special cases
43      // no points -> no convex hull
44      if ( pointset->n_points == 0 ) {
45         *n_verts = 0;
46         verts[0] = 0;
47         return;
48      }
49
50      // a single point is its own convex hull
51      if ( pointset->n_points == 1 ) {
52         *n_verts = 1;
53         verts[0] = 0;
54         return;
55      }
56
57      // now start graham scan
58      // find start node
59      int index = find_min(pointset);
60      //place it at the beginning of the list
61      swap(pointset, 0, index);
62      // compute angles
63      angles(pointset);
64
65      // sort
66      // we need distance to the start node as a tie breaker
67      // if two points are collinear with the start node
68      struct point* start = (struct point*)malloc(sizeof(struct point));
69      start->x = (pointset->points[0]).x;
70      start->y = (pointset->points[0]).y;
71
72      #ifdef __APPLE__
73      qsort_r(pointset->points, (size_t)pointset->n_points, sizeof(struct point), (void *)
          start, &compare);
74      #else
75      qsort_r(pointset->points, (size_t)pointset->n_points, sizeof(struct point), &compare,
          (void *)start);
76      #endif
77
78      free(start);
79
80      // we build up a list of nodes on the convex hull
81      // head gives the current last known node
82      // of which we assume it is on the convex hull
83      // just decrease head if we need to backtrack
```

```
84      verts[0] = 0;
85      verts[1] = 1;
86      int head = 1;
87
88      for(int i = 2; i <= pointset->n_points; i++){
89
90         // backtracking until we have left turn
91         while(head >= 1 &&
92               ccw(pointset->points+verts[head-1],
93                   pointset->points+verts[head],
94                   pointset->points+(i % pointset->n_points)) <= 0 )
95               head--;
96
97         // add point, unless we have closed the polygon
98         if ( i != pointset->n_points )
99            verts[head+1] = i;
100
101        head++;
102     }
103
104     // if we have backtracked to the start node, but there is another node,
105     // then put this as the next node
106     if ( head == 1 ) {
107        if ( pointset->points->x != (pointset->points+verts[1])->x || pointset->points->y
          != (pointset->points+verts[1])->y )
108           head++;
109     }
110
111     *n_verts = head;
112  }
```

Zum Abschluss können wir noch ein kleines Programm zum Testen entwickeln.

Programm 19.7: `kapitel_19/graham_scan/graham_scan_main.c`

```
1  #ifdef __linux__
2  #define _GNU_SOURCE
3  #endif
4
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  #include "graham_scan.h"
9  #include "point_set.h"
10
11 int main(int argc, char** argv){
12    if(argc != 3){
13        printf("./%s Eingabedateil Ausgabedatei\n", argv[0]);
14        return 1;
15    }
16
17    char* input  = argv[1];
18    char* output = argv[2];
19
20    struct pointSet* points = read_input(input);
21    int * verts = (int *)malloc(points->n_points*sizeof(int));
22    int n_verts = 0;
23
24    graham_scan(points, verts, &n_verts);
```

```
25      store_verts(points, verts, n_verts, output);
26
27      free_pointset(points);
28      free(verts);
29
30      return 0;
31  }
```