How to Think Like a Computer Scientist

C Version

Thomas Scheffler, Allen B. Downey

C-Version und deutsche Übersetzung: Thomas Scheffler
Konzept und Idee: Allen B. Downey

Version 0.9.6

8. April 2019

Inhaltsverzeichnis

Kapitel 1

Achtung, jetzt kommt ein Programm!

Das Ziel dieses Buches ist es, das Verständnis dafür zu wecken, wie Informatiker denken. Ich mag es, wie Informatiker denken, weil sie sich dabei der unterschiedlichen Ansätze aus der Mathematik, der Ingenieurwissenschaften und der Sprachwissenschaften bedienen, um mit viel Kreativität, Ausdauer und Beharrlichkeit etwas Neues noch nicht Dagewesenes zu schaffen.

Informatiker benutzen dabei, wie die Mathematiker, *formale Sprachen* um ihre Ideen und Berechnungen aufzuschreiben. Sie entwerfen Dinge und konstruieren komplexe Systeme, die sie aus einzelnen Komponenten zusammenbauen, und müssen dabei verschiedene Alternativen bewerten, abwägen und auswählen. Sie beobachten das Verhalten dieser Systeme wie Wissenschaftler: sie formulieren Hypothesen und testen ihre Vorhersagen.

Die wichtigste Fähigkeit eines Informatikers besteht darin, **Probleme zu lösen**. Dazu muss er diese Probleme erkennen, geschickt formulieren, kreativ über mögliche Lösungen nachdenken und diese klar, übersichtlich und nachvollziehbar ausdrücken und darstellen können. So wie es sich herausstellt, ist der Prozess des Erlernen einer Programmiersprache eine exzellente Möglichkeit, sich in der Fähigkeit des Problemlösens zu üben. Deshalb heißt dieses Kapitel “Achtung, jetzt kommt ein Programm!”

Das Erlernen des Programmierens ist für sich allein genommen bereits eine nützliche Fähigkeit. Je mehr wir uns in der Fähigkeit üben, um so offensichtlicher wird es werden, dass wir das Programmieren auch als ein Mittel zum Zweck nutzen können. Dass es sich dabei um ein sehr leistungsfähiges Mittel handelt, wird hoffentlich im Laufe des Buches noch klarer werden.

1.1 Was ist eine Programmiersprache?

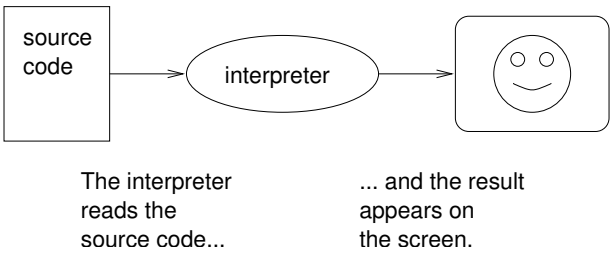
Die Programmiersprache, welche wir in diesem Kurs lernen werden, heißt C und wurde in den frühen 1970er Jahren von Dennis M. Ritchie in den Bell Laboratories entwickelt. C ist eine so genannte **Hochsprache** oder **High-level Sprache**. Andere Hochsprachen, die in der Programmierung verwendet werden, sind Pascal, Python, C++ und Java.

Aus dem Namen "Hochsprache" kann man ableiten, dass auch sogenannte **Low-level Sprachen** existieren. Diese nennt man Maschinensprache oder auch Assembler. Computer können nur Programme in Maschinensprache ausführen. Es ist daher notwendig, die Programme, welche in einer Hochsprache geschrieben wurden, in Maschinensprache zu übersetzen. Diese Übersetzung benötigt Zeit und einen zusätzlichen Arbeitsschritt, was einen klitzekleinen Nachteil gegenüber Low-level Sprachen darstellt.

Allerdings sind die Vorteile von Hochsprachen enorm. So ist es, erstens, *viel* einfacher in einer Hochsprache zu programmieren. Mit "einfacher" meine ich, dass es weniger Zeit in Anspruch nimmt ein Programm zu schreiben. Das Programm ist kürzer, einfacher zu lesen und mit einer höheren Wahrscheinlichkeit auch korrekt. Das heißt, es tut, was wir von dem Programm erwarten. High-level Sprachen verfügen über eine zweite wichtige Eigenschaft, sie sind **portabel**. Das bedeutet, dass unser Programm auf unterschiedlichen Arten von Computern ausgeführt werden kann. Maschinensprachen sind jeweils nur für eine bestimmte Computerarchitektur definiert. Programme, die in Low-level Sprachen erstellt wurden, müssten komplett neu geschrieben werden, wenn in unserem Computer statt einem Intel-kompatiblen Prozessor ein Prozessor von ARM verwendet würde. Ein Programm in einer Hochsprache muss einfach nur neu übersetzt werden.

Aufgrund dieser Vorteile wird die überwiegende Anzahl von Programmen in Hochsprachen geschrieben. Low-level Sprachen werden nur noch für wenige Spezialanwendungen verwendet.

Für die Übersetzung unseres Programms benötigen wir eine bestimmte Software auf unserem Computer - den Übersetzer. Es existieren grundsätzlich zwei Wege ein Programm zu übersetzen: **Interpretieren** oder **Kompilieren**. Ein *Interpreter* ist ein Programm welches ein High-level Programm interpretiert und ausführt. Dazu übersetzt der Interpreter das Programm Zeile-für-Zeile und führt nach jeder übersetzten Programmzeile die darin enthaltenen Kommandos sofort aus.

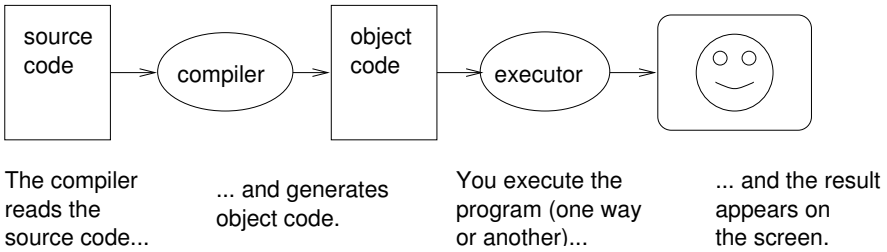


Ein *Compiler* ist ein Programm, welches ein High-level Programm im Ganzen einliest und übersetzt. Dabei wird stets das gesamte Programm komplett übersetzt, bevor die einzelnen Kommandos des Programms ausgeführt werden können. Die Programmiersprache C verwendet einen Compiler für die Übersetzung der Befehle in Maschinensprache.

Durch das Kompilieren entsteht eine neue, ausführbare Datei. Es wird dazu ein Programm zuerst kompiliert und das so übersetzte Programm in einem zweiten, separaten Schritt zur Ausführung gebracht. Man bezeichnet in diesem Fall das High-level Programm als den **Source code** oder **Quelltext**, und das übersetzte Programm nennt man den **Object code** oder das **ausführbare Programm**.

Angenommen, wir schreiben unser erstes Programm in C. Wir können dafür einen ganz einfachen Texteditor benutzen, um das Programm aufzuschreiben (ein Texteditor ist ein ganz einfaches Textverarbeitungsprogramm, welches in der Regel nicht einmal verschiedene Schriftarten darstellen kann). Wenn wir das Programm aufgeschrieben haben, müssen wir es auf der Festplatte des Computers speichern, zum Beispiel unter dem Namen `program.c`, wobei "program" ein beliebiger, selbstgewählter Dateiname ist. Die Dateiendung `.c` ist wichtig, weil sie einen Hinweis darauf gibt, dass es sich bei dieser Datei um Quellcode in der Programmiersprache C handelt.

Danach können wir den Texteditor schließen und den Compiler aufrufen (der genaue Ablauf hängt dabei von der verwendeten Programmierumgebung ab). Der Compiler ließt den Quelltext, übersetzt ihn und erzeugt eine neue Datei mit dem Namen `program.o`, welches den Objektcode enthält, oder die Datei `program.exe`, welche das ausführbare Programm enthält.



Im nächsten Schritt können wir das Programm ausführen lassen. Dazu wird das Programm in den Hauptspeicher des Rechners geladen (von der Festplatte in den Arbeitsspeicher kopiert) und danach werden die einzelnen Anweisungen des Programms ausgeführt.

Dieser Prozess klingt erst einmal sehr kompliziert. Allerdings sind in den meisten Programmierumgebungen (auch Entwicklungsumgebungen genannt) viele dieser Schritte automatisiert. In der Regel schreibt man dort sein Programm, klickt mit der Maus auf einen Bildschirmsymbol oder gibt ein einzelnes Kommando ein und das Programm wird übersetzt und ausgeführt. Allerdings ist es immer gut zu wissen, welche Schritte im Hintergrund stattfinden. So kann man, im Fall dass etwas schief geht, herausfinden, wo der Fehler steckt.

1.2 Was ist ein Programm?

Ein Programm ist eine Abfolge von Befehlen (engl.: *instructions*), welche angeben, wie eine Berechnung durchgeführt wird. Diese Berechnung kann mathematischer Art sein, wie zum Beispiel das Lösen eines Gleichungssystems oder die Ermittlung der Quadratwurzel eines Polynoms. Es kann aber auch eine symbolische Berechnung sein, wie die Aufgabe, in einem Dokument einen bestimmten Text zu finden und zu ersetzen. Erstaunlicherweise kann dies auch das Kompilieren eines Programmes sein.

Die Programmbefehle, welche sich **Anweisungen** (engl.: ***statements***) nennen, sehen in unterschiedlichen Programmiersprachen verschieden aus. Es existieren aber in allen Sprachen die gleichen, wenigen Basiskategorien, aus denen ein Computerprogramm aufgebaut ist. Es ist deshalb nicht schwer eine neue Programmiersprache zu lernen, wenn man bereits eine andere gut beherrscht. Die meisten Programmiersprachen unterstützen die folgenden Befehlskategorien:

Input: Daten von der Tastatur, aus einer Datei oder von einem angeschlossenen Gerät in das Programm einlesen.

Output: Daten auf dem Monitor darstellen, in eine Datei schreiben oder an ein angeschlossenes Gerät ausgeben.

Mathematik: Durchführen von grundlegenden mathematischen Operationen, wie zum Beispiel Addition und Multiplikation.

Testen und Vergleichen: Überprüfen, ob bestimmte Bedingungen erfüllt sind und die Steuerung der Ausführung bestimmter Abfolgen von Anweisungen in Abhängigkeit von diesen Bedingungen.

Wiederholung: Bestimmte Aktionen werden mehrfach, manchmal mit geringen Änderungen, nacheinander ausgeführt.

Das wäre dann schon fast alles. Jedes Programm, das wir in diesem Kurs kennenlernen, ist unabhängig von seiner Komplexität, aus einzelnen Anweisungen aufgebaut, welche diese Operationen unterstützen. Daher besteht ein Ansatz der Programmierung darin, einen großen komplizierten Prozess in immer kleinere und kleinere Unteraufgaben zu unterteilen, bis die einzelnen Aufgaben so klein und unbedeutend werden, dass sie mit einem dieser grundlegenden Befehle ausgeführt werden kann.

1.3 Was ist *debugging*?

Programmieren ist ein komplexer Prozess, und da es von Menschen durchgeführt wird, ist es mehr als wahrscheinlich, dass sich hier und dort Fehler einstellen. Programmierer bezeichnen einen Softwarefehler üblicherweise als **Error** oder auch **Bug** und den Prozess des Aufspürens und Korrigierens des Fehlers als **Debugging**.

Es gibt verschiedene Arten von Fehlern, die in einem Programm auftreten können. Es ist sinnvoll, die Unterscheidung zwischen diesen Arten zu kennen, um Fehler in eigenen Programmen schneller entdecken und beheben zu können. Programmfehler können sich zu unterschiedlichen Zeiten bemerkbar machen. Man unterscheidet zwischen Fehlern beim Kompilieren und beim Ausführen des Programms.

1.3.1 Fehler beim Kompilieren (Compile-time errors)

Der Compiler kann ein Programm nur übersetzen wenn dieses Programm den formalen Regeln der Programmiersprache entspricht. Diese Regeln, die **Syntax**, beschreiben die Struktur des Programms und der darin enthaltenen Anweisungen. Ein Programm muss syntaktisch korrekt sein, anderenfalls schlägt die Kompilierung fehl und das Programm kann nicht ausgeführt werden.

So beginnt zum Beispiel jeder deutsche Satz mit einem großen Buchstaben und endet mit einem Punkt. *dieser Satz enthält einen Syntaxfehler. Dieser Satz ebenfalls*

Für die meisten Menschen sind ein paar Syntaxfehler in einem Text kein größeres Problem. Wir können diese Texte trotzdem in ihrer Bedeutung verstehen, weil wir über Erfahrung und Weltwissen verfügen.

Compiler sind nicht so tolerant. Selbst ein einzelner Syntaxfehler irgendwo in unserem Programm führt dazu, dass der Compiler eine Fehlermeldung (engl.: *error message*) auf dem Bildschirm anzeigt und die weitere Arbeit des Übersetzens einstellt. Das erzeugte Programm kann nicht ausgeführt werden.

Zu allem Überfluss gibt es sehr viele Syntaxregeln in C, und die Fehlermeldungen des Compilers sind oft nicht besonders hilfreich für den Programmieranfänger. Der berühmte Physiker Niels Bohr hat einmal gesagt: “Ein Experte ist jemand, der in einem begrenzten Bereich schon alle möglichen Fehler gemacht hat.” Während der ersten paar Wochen unserer Karriere als C-Programmiererin oder C-Programmierer werden wir voraussichtlich viel Zeit damit zubringen, Syntaxfehler in selbst erstellten Programmen zu finden. In dem Maße wie unsere Erfahrung zunimmt, werden wir weniger Fehler machen und die gemachten Fehler schneller finden.

1.3.2 Fehler beim Ablauf des Programms (Run-time errors)

Eine zweite Kategorie von Programmfehlern sind die so genannten Laufzeitfehler (engl.: *run-time errors*). Sie werden so genannt, weil der Fehler erst auftritt, wenn unser Programm ausgeführt wird: zur Laufzeit.

Auch wenn wir unser Programm syntaktisch richtig aufgeschrieben haben, können Fehler auftreten die zum Abbruch des Programms führen. C ist keine **sichere** Sprache, wie zum Beispiel Java, wo Laufzeitfehler relativ selten sind. C

ist eine relativ hardwarenahe Programmiersprache. Vielleicht die hardwarenäheste von allen höheren Programmiersprachen. Die meisten Laufzeitfehler in C treten deshalb auf, weil die Sprache selbst keine Schutzmechanismen gegen den direkten Zugriff auf den Speicher des Computers bietet. So kann es vorkommen, dass unser Programm wichtige Speicherbereiche versehentlich überschreibt.

Die Hardwarenähe von C hat ihre guten, wie ihre schlechten Seiten. Viele Algorithmen sind dadurch in C besonders effizient und leistungsfähig umsetzbar. Gleichzeitig sind wir als Programmierer selbst dafür verantwortlich, dass unser Programm auch nur das tut, was wir beabsichtigen. Dafür müssen wir manchmal sorgfältiger arbeiten als Programmierer anderer Sprachen.

Bei den einfachen Programmen, die wir in den nächsten Wochen schreiben werden, ist es allerdings unwahrscheinlich, dass wir in unserem Programm einen Laufzeitfehler provozieren.

1.3.3 Logische Fehler und Semantik

Der dritte Fehlertyp, dem wir begegnen werden, ist der **logische** oder **semantische** Fehler. Wenn in unserem Programm ein logischer Fehler steckt, so wird es zunächst kaum auffallen. Es lässt sich kompilieren und ausführen, ohne dass der Computer irgendwelche Fehlermeldungen produziert. Es wird aber leider nicht die richtigen Dinge tun. Es wird einfach irgend etwas anderes tun, oder auch gar nichts. Insbesondere wird es nicht das tun, wofür wir das Programm geschrieben haben.

Das Programm, das wir geschrieben haben, ist nicht das Programm, das wir schreiben wollten. Die Bedeutung des Programms - seine Semantik - ist falsch. Die Gründe dafür können vielfältig sein. Am Anfang sind es vor allem unklare Vorstellungen darüber, was das Programm tun soll und wie ich das mit den Mitteln der Programmiersprache C erreichen kann. Es kann aber auch sein, dass unser Algorithmus fehlerhaft ist, oder wir nicht alle Voraussetzungen und Bedingungen vollständig geprüft haben.

Logische Fehler zu finden ist meistens ziemlich schwer. Es erfordert Zeit und Geduld herauszufinden, wo der Fehler steckt. Dazu kann es notwendig sein rückwärts zu arbeiten: Wir schauen uns die Resultate unseres Programms an und überlegen, was zu diesen Ergebnissen geführt haben kann.

1.3.4 Experimentelles Debugging

Eine der wichtigsten Fähigkeiten, die wir als angehende Programmierer erlernen müssen, ist das Debuggen von Programmen. Obwohl es gelegentlich auch frustrierend sein kann, so ist das Aufspühren von Programmfehlern ein intellektuell anspruchsvoller, herausfordernder und interessanter Teil des Programmierens.

In vielerlei Hinsicht ist Debugging mit der Arbeit eines Kriminalisten vergleichbar. Man muss Hinweisen nachgehen und Zusammenhänge herstellen zwischen den Prozessen innerhalb des Programms und den Resultaten, die sichtbar sind.

Debugging ist gleichfalls den experimentellen Wissenschaften ähnlich. Sobald wir eine Idee haben, was in unserem Programm falsch gelaufen sein sollte, verändern wir dieses und beobachten erneut. Wir bilden Hypothesen über das Verhalten des Programms. Stimmt unsere Hypothese, dann können wir das Ergebnis der Modifikation vorhersagen und wir sind dem Ziel, eines funktionsfähigen Programms, einen Schritt näher gekommen.

Wenn unsere Hypothese falsch war, müssen wir eine neue bilden. Wie bereits Sherlock Holmes sagte, "...when you have eliminated the impossible, whatever remains, however improbable, must be the truth" (aus Arthur Conan Doyle, *Das Zeichen der Vier*).

Einige Leute betrachten Programmieren und Debuggen als ein und dieselbe Sache. Man könnte auch sagen, Programmieren ist der Prozess, ein Programm so lange zu debuggen bis am Ende ein funktionsfähiges Programm entstanden ist, das unseren Vorstellungen entspricht. Dahinter steckt die Idee, dass wir immer mit einem funktionsfähigen Programm starten, welches *irgendeine* Funktion realisiert. Danach machen wir kleine Modifikationen, entfernen die Fehler und testen unser Programm, so dass wir zu jeder Zeit ein funktionsfähiges Programm haben, welches am Ende eine neue Funktion realisiert.

So ist zum Beispiel Linux ein Betriebssystem, welches Millionen von Programmzeilen enthält. Begonnen wurde es aber als ein einfaches Programm, dass Linus Torvalds benutzt hat um den Intel 80386 kennenzulernen. So berichtet Larry Greenfield: "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux" (aus *The Linux Users' Guide*, Beta Version 1).

In späteren Kapiteln werde ich einige praktische Hinweise zum Debugging und anderen Programmierpraktiken geben.

1.4 Formale und natürliche Sprachen

Als **natürliche Sprachen** bezeichnen wir alle Sprachen, die von Menschen gesprochen werden, wie zum Beispiel Englisch, Spanisch und Deutsch. Diese Sprachen wurden nicht von Menschen konstruiert (obwohl Menschen versuchen ihnen Ordnung und Struktur zu verleihen), sie sind das Resultat eines natürlichen Evolutionsprozesses.

Formale Sprachen sind vom Menschen für spezielle Anwendungszwecke konstruiert worden. So benutzen zum Beispiel Mathematiker spezielle Notationsformen, um den Zusammenhang zwischen Symbolen, Zahlen und Mengen darzustellen. Chemiker benutzen eine formale Sprache, um die chemische Struktur von Molekülen zu notieren. Und für uns ganz wichtig:

> **Programmiersprachen sind formale Sprachen, mit denen man das Verhalten einer Maschine steuert.**

> Programmiersprachen dienen der Beschreibung von Berechnungen und der Formulierung von Algorithmen. Ein **Algorithmus** ist eine aus abzählbar vielen Schritten bestehende eindeutige Handlungsanweisung zur Lösung einer Klasse von Problemen.

Wie ich bereits angedeutet hatte, tendieren formale Sprachen dazu, strikte Syntaxregeln zu besitzen. So ist zum Beispiel $3 + 3 = 6$ ein syntaktisch korrekter mathematischer Ausdruck, $3 =: 6\$$ hingegen nicht. H_2O ist eine syntaktisch korrekte chemische Formel, ${}_2Z$ ist es nicht.

Syntaxregeln betreffen zwei Bereiche der Sprache: deren Symbole und Struktur. Die Symbole stellen die Basiselemente einer Sprache bereit, dazu gehören die Wörter und Bezeichner, Zahlen und chemische Elemente. Eines der Probleme mit dem Ausdruck `3=:6$` ist, dass `$` kein legales Symbol in der Mathematik darstellt (so weit ich weiß, jedenfalls ...). Gleichfalls ist, ${}_2Z$ nicht legal, weil es kein chemisches Element mit der Abkürzung Z gibt.

Der zweite Fall für die Anwendung von Syntaxregeln betrifft die Struktur eines Ausdrucks, das heißt die Art und Weise, wie die Symbole der Sprache angeordnet werden.

Der Ausdruck `3=:6$` ist auch deshalb nicht korrekt, weil es nicht erlaubt ist, ein Divisionszeichen unmittelbar nach einem Gleichheitsszeichen zu schreiben. In gleicher Weise werden in molekularen Formeln die Mengenverhältnisse eines Elements als tiefer gestellte Zahl nach dem Elementname angegeben und nicht davor.

Wenn wir einen Satz in einer natürlichen Sprache lesen oder einen Ausdruck in einer formalen Sprache erfassen wollen, müssen wir seine Struktur herausfinden (bei natürlichen Sprachen macht unser Gehirn, das meistens ganz unbewußt von selbst). Diesen Prozess bezeichnen Informatiker als **Parsen**.

Wenn wir zum Beispiel den Satz hören "Die Würfel sind gefallen." erkennen wir, "Die Würfel" als das Subjekt und "sind gefallen" als das Prädikat. Nachdem wir den Satz geparst haben, können wir herausfinden, was dieser Satz bedeutet. Wir können seine Bedeutung (seine Semantik) verstehen. Angenommen wir wissen, was ein Würfel ist und was es bedeutet zu fallen, so können wir damit die generelle Bedeutung dieses Satzes verstehen.

1.4.1 Unterschiede formaler und natürlicher Sprachen

Obwohl formale und natürliche Sprachen viele Gemeinsamkeiten aufweisen: Symbole, Struktur, Syntax und Semantik, so existieren doch auch viele Unterschiede zwischen den Sprachen.

Vieldeutigkeit: Natürliche Sprachen sind voll von Mehrdeutigkeiten. Wir Menschen erkennen die Bedeutung von Aussagen in natürlicher Sprache üblicherweise anhand von Hinweisen aus dem Kontext und unserem Erfahrungswissen. So ist zum Beispiel die Aussage "wilde Tiere jagen" nicht

eindeutig. Es kann bedeuten, dass der Jäger wilde Tiere jagt, aber auch, dass wilde Tiere ihre Beute jagen. Formale Sprachen sind üblicherweise so konstruiert, dass sie keine Mehrdeutigkeiten aufweisen. Jede Aussage ist eindeutig interpretierbar. Aus `2+2` lässt sich immer und eindeutigerweise der Wert `4` ableiten. Die Bedeutung (Semantik) einer Aussage ist nicht von ihrem Kontext abhängig.

Redundanz: Um mit den vorhandenen Mehrdeutigkeiten in natürlichen Sprachen umzugehen und Missverständnisse zu reduzieren, verwenden diese Sprachen oft das Mittel der Redundanz. Das heißt, Informationen werden mehrfach wiederholt, zum Teil in anderen Formulierungen, obwohl dies eigentlich für das Verständnis der Bedeutung nicht notwendig wäre. Als Resultat sind natürliche Sprachen oft wortreich und ausschweifend. Während formale Sprachen wenig oder keine Redundanz aufweisen und dadurch knapp und präzise ausfallen.

Wortwörtlichkeit: Natürliche Sprachen beinhalten oft Redensarten und Metaphern. Wenn ich sage, "Die Würfel sind gefallen.", dann sind wahrscheinlich nirgends Würfel im Spiel, und es ist auch nichts heruntergefallen. Formale Sprachen hingegen, meinen wortwörtlich genau das, was geschrieben steht.

Viele Menschen, die ganz selbstverständlich eine natürliche Sprache verwenden (wir alle), haben oft Schwierigkeiten im Umgang mit formalen Sprachen. In vieler Art ist der Unterschied zwischen formalen und natürlichen Sprachen wie der Unterschied zwischen Poesie und Prosa – nur noch viel ausgeprägter:

Poesie: Wörter werden wegen ihrer Bedeutung, manchmal aber auch nur wegen ihres Klangs benutzt. Gedichte werden zum Teil wegen ihres Effekts oder der emotionalen Reaktion beim Leser geschrieben. Mehrdeutigkeiten kommen oft vor und werden vom Dichter stellenweise als Stilmittel bewusst eingesetzt.

Prosa: Die wörtliche Bedeutung eines Textes ist von Bedeutung und seine Struktur trägt dazu bei, das Verständnis seiner Bedeutung zu erfassen. Prosatexte lassen sich leichter analysieren als Gedichte, trotzdem enthalten sie oft Mehrdeutigkeiten.

Programm: Die Bedeutung eines Computerprogramms ist eindeutig, unzweifelhaft und wörtlich. Ein Programm lässt sich alleinig durch die Analyse der Symbole und der Struktur erfassen und verstehen.

1.4.2 Tipps zum Lesen von Programmen

Für das Erlernen einer Sprache ist es wichtig, nicht nur das Schreiben zu lernen, sondern auch viele Texte in dieser Sprache zu lesen. Im Folgenden habe ich einige Vorschläge zusammengetragen, wie man an das Lesen von Programmen (und Texten in anderen formalen Sprachen) herangehen sollte:

Zuallererst ist zu beachten, dass Texte in formalen Sprachen viel kompakter (wortärmer, weniger ausschweifend) sind als Texte natürlicher Sprachen, weil die Texte keine Redundanzen enthalten. Es dauert also in der Regel viel länger einen Text in einer formalen Sprache zu lesen, da dieser pro Texteinheit einfach viel mehr Information enthält. Hilfreich sind hier oft Anmerkungen der Programmierer die Erklärungen in natürlicher Sprache enthalten. Sie sollten sich auch angewöhnen, selbst solche Anmerkungen zu verfassen.

Wichtig ist auch die Struktur eines Programms. Es ist keine gute Idee, ein Programm als linearen Text von oben nach unten und links nach rechts durcharbeiten zu wollen. Größere Programme sind üblicherweise in sinnvolle Module gegliedert, die nicht unbedingt in der Reihenfolge des Quelltextes ausgeführt werden. Wir müssen versuchen das Programm in seiner Struktur zu erfassen. Dazu müssen wir zuerst wichtige Symbole erfassen und erkennen und uns ein mentales Bild vom Zusammenspiel der einzelnen Elemente bilden. Dabei kann es hilfreich sein, sich diese Elemente und ihre Abhängigkeiten kurz zu skizzieren.

Zum Schluss möchte ich noch darauf hinweisen, dass selbst kleine Details wichtig sind. Tippfehler und schwache Zeichensetzung, Dinge, die in natürlichen Sprachen als Formfehler gelten, können in formalen Sprachen große Auswirkungen auf die Bedeutung eines Textes haben.

1.5 Das erste Programm

Die Tradition verlangt, dass man das erste Programm, welches man in einer neuen Programmiersprache schreibt, “Hello, World!” nennt. Es ist ein einfaches Programm, welches nichts weiter tun soll, als die Worte “Hello, World!” auf dem Bildschirm auszugeben. In C sieht dieses Programm wie folgt aus:

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

/* main: generate some simple output */

int main(void)
{
      printf("Hello, World!\n");
      return EXIT_SUCCESS;
}
```

Manche Leute beurteilen die Qualität einer Programmiersprache danach, wie einfach es ist, das “Hello, World!” Programm zu erstellen. Nach diesem Standard schlägt sich C noch vergleichsweise gut. Allerdings enthält bereits dieses einfache Programm einige Merkmale, die es schwierig machen das komplette Programme einem Programmieranfänger zu erklären.

Deshalb werden wir an dieser Stelle erst einmal einige von ihnen ignorieren, wie zum Beispiel die ersten zwei Programmzeilen.

Die dritte Programmzeile fängt mit einem `/*` an und endet mit `*/`. Das zeigt uns, dass es sich bei dieser Zeile um einen **Kommentar** handelt. Ein Kommentar ist eine kurze Anmerkungen zu einem Teil des Programms (siehe Abschnitt 1.4.2). Diese Anmerkung wird üblicherweise dazu benutzt, zu erklären, was das Programm tut. Kommentare können irgendwo im Quelltext des Programms stehen. Wenn der Compiler ein `/*` sieht, dann ignoriert er von da an alles bis er das dazugehörige `*/` findet. Die enthaltenen Anmerkungen sind daher streng genommen nicht Teil unseres Programms.

In der vierten Programmzeile fällt das Wort `main` auf. `main` ist ein spezieller Name, der angibt, wo in einem Programm die Ausführung der Befehle beginnt. Wenn das Programm startet, wird die jeweils erste **Anweisung** in `main` ausgeführt, danach werden der Reihe nach alle weiteren Anweisungen ausgeführt, bis das Programm zum letzten Programmbefehl kommt und beendet wird.

Es existiert keine Beschränkung hinsichtlich der Anzahl von Anweisungen, die unser Programm in `main` enthalten kann. In unserem Beispiel sind das nur zwei Anweisungen. Die erste ist eine **Ausgabeanweisung**. Sie wird dazu benutzt, eine Nachricht auf dem Bildschirm anzuzeigen (zu "drucken"). In C wird die `printf` Anweisung benutzt, um Dinge auf dem Bildschirm des Computers auszugeben. Die Zeichen zwischen den Anführungszeichen werden ausgegeben.

Auffällig ist dabei der `\n` am Ende der Nachricht. Dabei handelt es sich um ein spezielles Zeichen, genannt *newline*, welches an das Ende einer Textzeile angefügt wird und den Cursor veranlasst auf die nächste Zeile des Bildschirms zu wechseln. Wenn unser Programm jetzt das nächste Mal etwas ausgeben möchte, so erscheint der neue Text auf einer neuen Bildschirmzeile. Am Ende der Anweisung finden wir ein Semikolon (`;`). Damit wird die Anweisung abgeschlossen – es muss am Ende jeder Anweisung stehen.

Mit der letzten Anweisung verlassen wir das Programm und geben die Kontrolle an das Betriebssystem zurück. Die `return` Anweisung wird verwendet um einen Programmteil (in C Funktion genannt) zu beenden und die Kontrolle an die Funktion zurückzugeben, welche die aktuelle Funktion gestartet (aufgerufen) hat. Dabei können wir eine Nachricht an die aufrufende Funktion übergeben (in unserem Fall das Betriebssystem) und teilen mit, dass das Programm erfolgreich beendet wurde.

Es gibt einige weitere Dinge, die wir über die Syntax von C-Programmen wissen müssen:
C benutzt geschweifte Klammern ({ und }) um Gruppen von Anweisungen zu bilden. In unserem Programm befindet sich die Ausgabeanweisung `printf` innerhalb geschweifter Klammern. Damit wird angezeigt, dass sie sich *innerhalb* der Definition von `main` befindet. Wir stellen auch fest, dass die Anweisungen im Programm eingerückt sind. Dabei handelt es sich nicht um eine strikte Vorgabe des Compilers, sondern um eine Übereinkunft zwischen Programmierern, die uns das Lesen eines Programms erleichtern soll. So kann man leichter visuell erfassen, welche Programmteile zusammengehören und von anderen Teilen abhängig sind. Im Anhang A habe ich dazu einige wichtige Regeln zusammengetragen, die man möglichst von Anfang an berücksichtigen sollte.

Zu diesem Zeitpunkt wäre es eine gute Idee sich an einen Computer zu setzen und das Programm zu kompilieren und auszuführen. Leider kann ich an dieser Stelle nicht genauer darauf eingehen, wie man das macht, da sich die einzelnen Computersysteme stark voneinander unterscheiden. Ich gehe davon aus, dass ein Seminarbetreuer, Freund oder eine Internet-Suchmaschine hier weiterhelfen können.

Wie ich bereits erwähnte, ist der C-Compiler sehr pedantisch, wenn es um die Einhaltung der Syntaxregeln geht. Wenn wir auch nur den kleinsten Tippfehler bei der Eingabe des Programms machen, ist die Gefahr groß, dass sich das Programm nicht erfolgreich kompilieren lässt. Wenn wir zum Beispiel `sdtio.h` statt `stdio.h` eingegeben haben, werden wir eine Fehlermeldung wie die folgende erhalten:

```
hello_world.c:1:19: error: sdtio.h: No such file or directory
```

Diese Fehlermeldung enthält eine Vielzahl von Informationen, leider sind sie in einem kompakten, schwer zu interpretierenden Format verfasst. Ein freundlicherer Compiler würde statt dessen schreiben:

> "In Zeile 1 des Quelltextes mit dem Dateinamen `hello_world.c` haben Sie versucht eine Headerdatei mit dem Dateinamen `sdtio.h` zu laden. Ich konnte keine Datei mit diesem Namen finden, ich habe aber eine Datei mit dem Namen `stdio.h` gefunden.
> Wäre es möglich, dass sie diese Datei gemeint haben?"

Leider sind die wenigsten Compiler so nett zu Anfängern und so geschwätzig. Der Compiler ist noch dazu nicht besonders schlau. In den meisten Fällen gibt uns die Fehlermeldung nur einen ersten Hinweis auf das mögliche Problem und manchmal liegt der Compiler mit seinem Hinweis auch gänzlich daneben und der Fehler steckt an einer ganz anderen Stelle. Vorangegangene Fehler können Folgefehler produzieren. Es ist deshalb angeraten, die Fehler in der Reihenfolge ihres Auftretens zu beheben.

Dennoch kann der Compiler auch ein nützliches Werkzeug für das Erlernen der Syntax einer Programmiersprache sein. Wir beginnen mit einem funktionsfähigen Programm (wie `hello_world.c`) und modifizieren dieses auf verschiedene Art und Weise. Sollten wir bei unseren Versuchen eine Fehlermeldung erhalten, so prägen wir uns die Nachricht des Compilers und die dazugehörige Ursache ein und falls derselbe Fehler wieder auftritt, wissen wir, was wir verändern müssen. Compiler kennen die Syntax einer Sprache sehr genau. Es wird sicher einige Zeit brauchen, bevor Sie die Nachrichten des Compilers richtig interpretieren können – es lohnt sich aber.

1.6 Glossar

Algorithmus (engl.: *algorithm*): Eine detaillierte und explizite Vorschrift zur schrittweisen Lösung eines Problems oder einer Klasse von Problemen mit folgenden Eigenschaften:

- besteht aus einzelnen Schritten
- jeder Schritt besteht aus einer einfachen und offensichtlichen Aktion
- zu jedem Zeitpunkt ist klar, welcher Schritt als nächstes ausgeführt wird

Anweisung (engl.: *statement*): Befehl oder Befehlsfolge zur Steuerung eines Computers (einzelner Schritt in einem Programm). In C werden Anweisungen durch das Semikolon (`;`) gekennzeichnet.

Bug (engl.: *bug*): Ein Fehler in einem Programm. Man unterscheidet Syntaxfehler, Laufzeitfehler und logische Fehler.

Debugging (engl.: *debugging*): Der Prozess der Fehlersuche und der Fehlerkorrektur in einem Programm.

High-level Sprache (engl.: *high-level language*): Eine Programmiersprache wie C, welche für Menschen einfach zu lesen und zu schreiben ist. Hochsprachen müssen vor der Ausführung in Maschinensprache übersetzt werden und können üblicherweise auf verschiedenen Computersystemen laufen.

Maschinennahe Sprache (engl.: *low-level language*): Eine Programmiersprache welche sich an den Befehlen und Fähigkeiten eines bestimmten Prozessors orientiert und vom Programmierer ein tiefes Verständnis des Aufbaus eines Computers verlangt (z.B. Assembler). Maschinennahe Sprachen müssen ebenfalls noch in Maschinensprache übersetzt werden. Diese Übersetzung ist aber sehr einfach und direkt. Programme in Maschinensprache laufen nur auf einem bestimmten Computersystem.

Formale Sprache (engl.: *formal language*): Eine künstliche Sprache, die von Menschen für spezielle Zwecke entworfen wurde. Mit formalen Sprachen lassen sich mathematische Ideen oder Computerprogramme beschreiben. Alle Programmiersprachen sind formale Sprachen.

Natürliche Sprache (engl.: *natural language*): Eine der Sprachen die von Menschen gesprochen wird und die sich auf evolutionäre Weise entwickelt hat.

portabel (engl.: *portability*): Der Begriff *portabel* kennzeichnet die Eigenschaft eines Computerprogramms auf unterschiedlichen Computersystemen einsetzbar zu sein (Übertragbarkeit).

interpretieren (engl.: *interpret*): Ein Programm ein einer Hochsprache ausführen, indem jede einzelne Programmzeile nacheinander durch einen *Interpreter* übersetzt und ausgeführt wird.

compilieren (engl.: *compile*): Ein Programm in einer Hochsprache mit Hilfe eines *Compilers* in ein Programm in Maschinensprache zu übersetzen. Es

wird dabei das komplette Programm übersetzt und eine neue, ausführbare Datei erzeugt, die auf einem bestimmten Computersystem ausgeführt werden kann.

Quelltext (engl.: *source code*): Ein Programm in einer Hochsprache, bevor es compiliert wurde.

Objektcode (engl.: *object code*): Das Produkt eines Compilers nach der Übersetzung des Quelltextes (Maschinencode).

Ausführbare Datei (engl.: *executable*): Der Maschinencode inklusive aller Bibliotheken. Die Datei kann durch das Betriebssystem gestartet werden und führt dann das Programm aus.

Kommentar (engl.: *comment*): Ein Teil eines Programms, welcher Informationen für den Programmierer über das Programm beinhaltet. Kommentare haben keinen Einfluss auf die Ausführung des Programms.

Syntax (engl.: *syntax*): Die Struktur eines Programms.

Syntaxfehler (engl.: syntax error): Ein Fehler in einem Programm, welches es dem Compiler unmöglich macht das Programm zu parsen (und somit zu übersetzen).

Semantik (engl.: *semantics*): Die Bedeutung eines Programms.

Parsen (engl.: *parse*): Ein Programm zu untersuchen und die syntaktische Struktur zu analysieren.

Logischer Fehler (engl.: *logical error*): Ein Fehler in einem Programm welcher dazu führt, dass das Programm zwar abläuft, aber etwas anderes macht, als das was der Programmierer beabsichtigt hatte.

1.7 Übungsaufgaben

Übung 1.1

Informatiker haben die ärgerliche Angewohnheit normale Worte einer Sprache zu benutzen und ihnen eine ganz eigene Bedeutung zu geben, die von ihrer normalen Verwendung abweicht. So haben zum Beispiel die Worte *Anweisung (statement)* und *Kommentar (comment)* normalerweise eine sehr ähnliche Bedeutung (zumindest auf Englisch). In einer Programmiersprache sind sie aber sehr unterschiedlich. Es ist wichtig, die Bedeutung der Elemente der Programmiersprache genau zu kennen und sie richtig einzusetzen, anderenfalls können sie keine korrekten Programme schreiben.

Das Glossar am Ende eines jeden Kapitels dient dazu, wichtige Begriffe und Phrasen zu rekapitulieren und die besondere Bedeutung dieser Begriffe klar zu machen.

Achten Sie darauf, dass ein Begriff, den Sie aus der Umgangssprache kennen, eine ganz eigene Bedeutung haben kann, wenn er in der Programmierung verwendet wird.

a. Was ist der Unterschied zwischen einer *Anweisung (statement)* und einem *Kommentar (comment)* in einer Programmiersprache?

b. Was bedeutet es, wenn man sagt ein Programm sei *portabel*?

c. Was bedeutet es, wenn man sagt ein Programm sei *ausführbar (executable)*?

Übung 1.2

Bevor wir uns eingehender mit der Programmiersprache beschäftigen, ist es wichtig herauszufinden, wie sich ein C-Programm auf unserem Computer kompilieren und ausführen lässt. Die erforderlichen Schritte können je nach verwendetem Betriebssystem und eingesetztem Compiler sehr unterschiedlich sein.

a. Geben Sie das "Hello World"-Programm aus Abschnitt 1.5 des Vorlesungs-Scripts (siehe Moodle) in den Computer ein, kompilieren Sie es und führen Sie es aus.

b. Fügen Sie eine zweite `printf()`-Anweisung hinzu, welche eine weitere Nachricht ausgibt. Irgendeine kurze Bemerkung, wie zum Beispiel, "How are you?" Speichern, kompilieren und führen Sie das Programm erneut aus.

c. Fügen Sie eine Kommentarzeile zu ihrem Programm hinzu (wo immer sie wollen), und kompilieren Sie das Programm erneut.

   Führen Sie das Programm aus. Wie hat sich der Kommentar auf den Ablauf des Programms ausgewirkt?

Diese Übung mag Ihnen trivial erscheinen, aber sie ist der Grundstein für all die vielen Programme, die wir in der nächsten Zeit entwickeln werden.

Um mit Vertrauen und Zuversicht die Eigenarten und Besonderheiten einer Programmiersprache zu entdecken, ist es erforderlich, dass man Vertrauen in die Programmierumgebung hat. Es ist nämlich zum Teil sehr einfach die Übersicht darüber zu verlieren, welches Programm jetzt gerade bearbeitet, übersetzt und ausgeführt wird. Und es kann leicht vorkommen, dass Sie versuchen den Fehler in einem Programm zu finden, während Sie versehentlich ein anderes Programm ausführen, oder Änderungen in einem Programm noch gar nicht gespeichert wurden.

Das Hinzufügen und Ändern von Ausgabeanweisungen (`printf()`) ist ein einfacher Weg um herauszufinden, ob das Programm, das sie ändern, auch das Programm ist, dass sie ausführen.

Übung 1.3

Es ist eine gute Idee, sich mit einer Programmiersprache vertraut zu machen, indem man viele Sachen ausprobiert.

Wir können zum Beispiel in unser Programm ganz bewusst Fehler einbauen und beobachten, ob der Compiler diese Fehler findet und wie er sie uns anzeigt. Manchmal wird der Compiler uns genau sagen, was falsch gelaufen ist und wie wir den Fehler beheben können. Manchmal bekommen wir nur eine unverständliche Meldung.

Durch einfaches Ausprobieren können wir uns einen Überblick verschaffen, wann wir dem Compiler trauen können und wann wir selbst herausfinden müssen, was falsch gelaufen ist.

Nehmen Sie ein lauffähiges Programm und probieren Sie nacheinander die folgenden Veränderung.

Achtung: Verändern Sie immer nur eine Stelle in ihrem Programm und lassen Sie es danach ausführen. Machen Sie die Änderung rückgängig, bevor Sie die nächste Änderung vornehmen.

a. Entfernen Sie die schließende, geschweifte Klammer (}).
b. Entfernen Sie die öffnende, geschweifte Klammer ({).
c. Entfernen Sie das `int` vor `main`.
d. Anstelle von `main` schreiben Sie `mian`.
e. Entfernen Sie das schließende `*/` von einem Kommentar.
f. Ersetzen Sie `printf` durch `pintf` im Quelltext des Programms.
g. Löschen Sie eine der Klammern: `(` oder `)`
h. Fügen Sie eine weitere Klammer hinzu.
i. Löschen Sie das Semikolon nach der `return` Anweisung.
j. Löschen Sie das Semikolon nach der `printf()`-Anweisung.
k. Entfernen Sie das hintere Anführungszeichen in den Klammern von `printf()`.
l. Entfernen Sie das vordere Anführungszeichen in den Klammern von `printf()`.
m. Schreiben Sie `<sdtio.h>` statt `<stdio.h>` .
n. Lassen Sie das `#` vor der Präprozessoranweisung `include` weg.
o. Kopieren Sie den gesamten Code und fügen Sie ihn noch einmal am Ende ein.
p. Schreiben Sie die Zeile `printf(...);` einmal vor `int main(void)`.

1) Nennen Sie jeweils die erste Fehlermeldung, die der Compiler ausgibt (die weiteren sind üblicherweise Folgefehler).
2) Versuchen Sie jede Fehlermeldung zu erklären. Was *'denkt'* der Compiler, was hier passiert? Warum gibt er genau diesen Fehlertext aus? Wird in der Meldung tatsächlich der von Ihnen (im Augenblick) bewusst eingebaute Fehler bemängelt? Oder stört sich der Compiler eigentlich an etwas ganz anderem?

Übung 1.4

Die `printf()`-Funktion ist eine der wichtigsten Funktionen für das Erlernen der Programmiersprache C. Mit Hilfe der Funktion lassen sich interne Zustände im Programm auf dem Bildschirm ausgeben. Leider ist die Funktion nicht besonders einfach zu benutzen und unterstützt eine Vielzahl von Optionen und Umwandlungszeichen (`%d, %f, ...`).

Probieren Sie das folgende Progamm aus und erklären Sie die beobachteten Ausgaben:

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

int main (void)
{
   printf("%d\n",5/2);
   printf("%f\n",5.0/2);
   printf("%s\n","5/2");
   printf("5/2\n");
```

```
    return EXIT_SUCCESS;
}
```

Kapitel 2

Variablen und Typen

2.1 Noch mehr Bildschirmausgaben

Wie ich bereits im letzten Kapitel erwähnte, können wir so viele Anweisungen in `main()` aufnehmen, wie wir wollen. So können wir zum Beispiel in unserem Programm auch mehr als eine Zeile ausgeben lassen:

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

/* main: generate some simple output */

int main (void)
{
      printf ("Hello World!\n");     /* output one line */
      printf ("How are you?\n");     /* output another line */
      return EXIT_SUCCESS;
}
```

Wie man sehen kann, ist es erlaubt, Kommentare auch direkt in eine Programmzeile zu schreiben und nicht nur in eine separate Zeile.

Die Ausdrücke innerhalb der Anführungszeichen werden **Strings** oder **Zeichenketten** genannt, weil sie aus einer Folge von Buchstaben bestehen. Strings können jede beliebige Kombination von Buchstaben, Ziffern, Satzzeichen und anderen speziellen Zeichen enthalten. Probleme bereiten uns nur die deutschen Umlaute und das 'ß'.

Manchmal ist es sinnvoll, den Text mehrerer Ausgabeanweisungen zusammen in einer Bildschirmzeile anzuzeigen. Wir können das ganz einfach umsetzen, in dem wir das `\n` Zeichen aus der ersten `printf()` Anweisung entfernen:

```
int main (void)
{
    printf ("Goodbye, ");
    printf ("cruel world!\n");
    return EXIT_SUCCESS;
}
```

In diesen Fall erscheint die Ausgabe auf einer einzelnen Zeile wie folgt:

```
Goodbye, cruel world!
```

Im Programm fällt auf, dass sich zwischen dem `Goodbye,` und dem Anführungszeichen noch ein Leerzeichen befindet. Dieses Leerzeichen finden wir auch in dem angezeigten Text auf dem Bildschirm wieder, das heißt es beeinflusst das Verhalten unseres Programms.

Leerzeichen ausserhalb von Anführungszeichen, irgendwo im Quelltext des Programms, haben üblicherweise keinen Einfluss auf das Verhalten unseres Programms. Ich hätte den Quelltext auch in der folgenden Form aufschreiben können:

```
int main(void)
{
printf("Goodbye, ");
printf("cruel world!\n");
return EXIT_SUCCESS;
}
```

Dieses Programm ist genauso gut kompilier- und ausführbar wie das Original. Ebenso haben die Zeilenumbrüche an den Zeilenenden keine Bedeutung. Ich hätte also schreiben können:

```
int main(void){printf("Goodbye, ");printf("cruel world!\n");
return EXIT_SUCCESS;}
```

Das funktioniert auch! Allerdings fällt auf, dass es schwerer und schwerer wird das Programm zu lesen. Zeilenumbrüche und Leerzeichen im Quelltext sind ein sinnvolles Mittel um ein Programm visuell zu strukturieren. Es wird dadurch für uns einfacher das Programm zu lesen und mögliche Fehler im Programm zu finden, beziehungsweise das Programm später zu ändern und anzupassen. Moderne Entwicklungsumgebungen können dabei die Arbeit erleichtern. Sie bieten die Möglichkeit den Quelltext automatisch formatieren zu lassen.

2.2 Bits und Bytes

Computer sind digital, das weiß heute jedes Kind. Was aber genau bedeutet das eigentlich?

Mein Computer, auf dem ich dieses Buch tippe, kann Töne, Bilder, Texte und Videos erzeugen, abspielen und verändern. Wie können so unterschiedliche Aufgaben von einer Maschine geleistet werden?

Das Rätsels Lösung ist darin zu finden wie diese Sachen gespeichert werden: als **digitale Daten**!

Computer speichern Daten in Form von Bits. Ein Bit ist die kleinste vorstellbare Informationsmenge. Es kennt nur 2 Zustände: *an* oder *aus*, *0* oder *1*, *high* oder *low*. Das ist scheinbar wenig, aber trotzdem schon recht nützlich: leuchtet die Fahrradlampe oder leuchtet sie nicht? Diese Tatsache lässt sich mit einem einzelnen Bit beschreiben.

Für das Speichern von Fotos, Musik und Abschlussarbeiten benötigen wir natürlich sehr viel mehr Bits, aber das ist für moderne Computer schon lange kein Problem mehr. Das Grundprinzip jedenfalls ist immer noch gültig.

Da es sehr unhandlich ist, mit einzelnen Bits zu hantieren, speichern Computer ihre Daten in Gruppen von Bits. Wie wir gesehen haben, trifft ein einzelnes Bit eine einfache Fallunterscheidung: *0* oder *1*, *an* oder *aus*. Gruppieren wir die Bits, so lassen sich mehr Fälle unterscheiden. Mit 8 Bit können wir bereits 256 Fälle unterscheiden – mehr als genug für die Zeichen des lateinischen Alphabets, der Ziffern und Satzzeichen.

Eine Gruppe von 8 Bit nennt man Byte und jedes Byte hat im Computer eine eindeutige Adresse. Diese ist nötig, damit keine Daten verloren gehen und unser Programm immer ganz genau weiß, wo sich welche Daten befinden.

Abbildung 2.1 zeigt uns einen Ausschnitt aus dem Speicher eines Computers. Wir sehen eine Matrix von Bits, die entweder den Wert 0 oder 1 besitzen und sind erst einmal verwirrt. Wie gelingt es hier die Übersicht zu behalten?

Für die Interpretation der Bitfolgen kommen üblicherweise Codes zum Einsatz, wie zum Beispiel der im Anhang B beschriebene ASCII Code. Dort ist festgelegt, dass die Bitfolge `00110101` das **Zeichen** *5* beschreibt. Nicht zu verwechseln mit dem **Dezimalwert** *5*, welche durch die Bitfolge `00000101` dargestellt wird.

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
`Byte 6680`	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**`Dezimalwert 5`**
`Byte 6681`	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**`ASCII-Zeichen '5'`**
`Byte 6682`	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	
`Byte 6683`	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	
`Byte ...`	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	

Abbildung 2.1: Ein hypothetischer Ausschnitt aus dem Speicher eines Computers

Als Programmierer einer Hochsprache müssen sie die genaue numerische Adresse der Daten sowie die verwendete Code-Tabelle nicht auswendig lernen. Da wir uns als Menschen besser Namen als lange Zahlenfolgen merken, werden wir in den nächsten Abschnitten kennenlernen, wie wir *Datentypen* und *Variablen* benutzen um Daten zu speichern und zu bearbeiten. Die Übersetzung in die richtige Bitfolge übernimmt dann der Compiler für uns.

2.3 Werte und Datentypen

Computerprogramme arbeiten mit Daten, die im Speicher des Computers abgelegt sind. Daten besitzen einen **Wert** – das heißt sie repräsentieren eine konkrete Zahl oder einen Buchstaben – und sind eines der fundamentalen Dinge mit denen ein Computerprogramm arbeiten kann.

Daten repräsentieren so unterschiedliche Dinge wie die ganzen Zahlen, die reellen Zahlen und Buchstaben. Man sagt die Daten haben einen bestimmten **Typ**.

Es ist wichtig, dass ein Programm ganz genau weiß, um welche Art von Daten es sich handelt, da unterschiedlichen Anforderungen für die Speicherung der Daten im Computer existieren.

Wie wir gesehen haben, werden Daten als Bitfolgen gespeichert. Für die Speicherung von *Buchstaben* des lateinischen Alphabets reichen üblicherweise 7 oder 8 Bit. Für die Speicherung von *Zahlen* werden dagegen wesentlich mehr Bit benötigt. Wir wollen ja nicht nur von 0 bis 255 zählen können.

Es macht für den Computer also einen Unterschied, ob wir den Zahlenraum der *ganzen Zahlen* oder der *reelle Zahlen* nutzen. Denn obwohl Computer mit hoher Geschwindigkeit rechnen gibt es ein Problem: Zahlenbereiche in der Mathematik sind unendlich. Unser Computerspeicher aber ist es nicht. Daher müssen wir uns bewusst machen, dass es zu Einschränkungen im Wertebereich und der Genauigkeit kommen kann.

Ein weiterer Grund für die Unterscheidung der Datentypen liegt darin begründet, dass nicht alle Operationen für jeden Datentyp sinnvoll sind. Wir können zwei Zahlen addieren, aber die Addition von Buchstaben `'a' + 'b'` ist nicht definiert. Der Datentyp legt daher fest, welche Bedeutung die Bitfolgen im Speicher des Computers haben: *Buchstabe*, *ganze Zahlen*, usw.

Die einzigen Daten, mit denen wir bisher gearbeitet haben, waren Folgen von Buchstaben, auch Zeichenketten oder Strings genannt. Wir haben zum Beispiel `"Hello, world!"` auf dem Bildschirm ausgegeben. Wir (und der Compiler) können diese Zeichenketten anhand der umschließenden Anführungszeichen erkennen.

Die *ganzen Zahlen* (beispielsweise 1 oder 17) werden in C als *integer* bezeichnet. Unser Programm kann nicht nur Zeichenketten, sondern auch ganze Zahlen auf dem Bildschirm ausgeben:

```
printf("%i\n", 16);
```

Die Ausgabe sieht auf den ersten Blick auch nicht anders aus, als wenn wir uns eine Zeichenkette ausgeben lassen:

```
printf("16\n");
```

Zahlen werden vom Computer aber anders behandelt als Zeichenketten, so kann man zum Beispiel mit Zahlen rechnen:

```
printf("%i\n", 16 + 1);
```

Schauen wir uns die `printf()` Anweisung genau an, so fällt ein `%i` zwischen den Anführungszeichen auf. Dabei handelt es sich um ein Platzhalterzeichen, welches angibt das eine ganze Zahl ausgegeben werden soll. Die auszugebene Zahl folgt erst hinter den Ausführungszeichen, durch Komma getrennt. Es gibt eine Reihe solcher Platzhalter für unterschiedliche Datentypen. Den Nächsten werden wir gleich kennenlernen.

Der Datentyp *character* repräsentiert einen Buchstaben, eine Ziffer oder ein Satzzeichen. C benutzt für die Speicherung der Werte vom Typ *character* den ASCII-Code (siehe Anhang B). In diesem Code ist leider nur das englische Alphabet definiert. Landestypische Erweiterungen des Zeichensatzes werden nicht berücksichtigt. Aus diesem Grund ist es am Anfang einfacher erst einmal komplett auf *ä*, *ö*, *ü* und *ß* zu verzichten und statt dessen *ae*, *oe*, *ue* und *ss* zu schreiben. Ein *character*-Wert in unserem Programm wird durch einfache Anführungsstriche kenntlich gemacht, wie zum Beispiel `'a'` oder `'5'`:

```
printf("%c\n", '$');
```

Für die Ausgabe von Daten vom Typ *character* benötigen wir das Platzhalterzeichen `%c`. Unser Beispiel gibt ein einzelnes Dollarzeichen in einer eigenen Bildschirmzeile aus.

Am Anfang ist es schwer, die einzelnen Typen der Werte `"5"`, `'5'` und `5` zu unterscheiden. Man muss sehr genau auf die Zeichensetzung achten, dann wird klar, dass der erste Wert ein String, der zweite Wert ein Buchstabe und der dritte eine ganze Zahl darstellt. Der Grund für diese Unterscheidung wird uns im Laufe des Kurses noch klarer werden.

2.4 Variablen

Eine der mächtigsten Fähigkeiten einer Programmiersprache ist die Möglichkeit digitale Daten zu speichern, wieder abzurufen und zu verändern. In unseren bisherigen Versuchen waren alle verwendeten Werte durch die Angaben im Quelltext des Programms statisch festgelegt. Ab jetzt werden wir oft **Variablen** benutzen um Werte dynamisch zu speichern und zu verändern. Variablen können wir uns wie die Memory-Taste an einem Taschenrechner vorstellen, nur etwa 1000x flexibler und mächtiger, weil unser Programm beliebig viele Variablen nutzen und nicht nur Zahlen speichern kann.

Eine Variable ist aber gar nichts sonderlich Geheimnisvolles. Ich hatte ja bereits erwähnt, dass die Bytes im Speicher unseres Computers Adressen besitzen. Da wir als Menschen nicht sehr gut darin sind uns lange Zahlenfolgen zu merken, verwenden wir dafür besser einen sinnvollen, selbstgewählten Namen, den **Variablennamen**.

Variablen sollen verschiedene Arten von Daten speichern können und müssen daher auch verschiedene Datentypen unterstützten. Es gibt einige Programmiersprachen, bei denen der Computer selbstständig den Typ der Variable anhand des zu speichernden Werts erkennt. In C muss der Typ immer angeben werden.

Wollen wir eine neue Variable verwenden, so müssen wir sie erst einmal *deklarieren*, das heißt in unserem Programm bekannt machen. Um eine Variable zu deklarieren die ein Zeichen speichert, muss der Typ *character* als dem Namen vorangestelltes `char` angegeben werden. Die folgende Anweisung, die man auch als **Deklaration** bezeichnet, erzeugt eine neue Variable mit dem Namen `fred` vom Typ *character*:

```
char fred;    /* creates a new character variable */
```

Der Typ einer Variable bestimmt, welche Werte gespeichert werden können. Eine `char` Variable kann genau ein Zeichen speichern. Ganze Zahlen können als `int` Variablen gespeichert werden. Um eine einfache Variable vom Typ *integer* anzulegen, verwenden wir folgende Syntax:

```
int bob;
```

Dabei ist `bob` ein beliebiger Name, den wir auswählen können um die Variable zu identifizieren. Es ist im Allgemeinen eine gute Idee Namen zu wählen, welche die Daten beschreiben, die in ihnen gespeichert werden sollen. Das erleichtert den Umgang mit Variablen und macht ein Programm leichter lesbar. Weiterhin sollten Sie sich bereits am Anfang mit den Regeln für die Namensverwendungen vertraut machen (siehe Anhang A.2).

Schauen wir uns zum Beispiel die folgenden Variablendeklarationen an:

```
char first_letter;
char last_letter;
int hour, minute;
```

Wir können wahrscheinlich eine erste, zutreffende Vermutung äußern, welche Werte in diesen Variablen gespeichert werden. Dieses Beispiel zeigt auch, wie wir einfach mehrere Variablen des gleichen Typs deklarieren können: `hour` und `minute` sind beides Variablen für ganze Zahlen (`int` Typ).

Für sehr große und komplexe Programme ist auch diese Form der Variablenbezeichnung noch zu unübersichtlich. Deswegen hat der aus Ungarn stammende Programmierer Charles Simonyi ein System der Variablenbezeichnung entworfen, indem dem Namen einer Variablen noch weitere Informationen hinzugefügt werden können.[1] Wir werden dieses System in diesem Buch aber nicht anwenden, da unsere Programme noch sehr klein und übersichtlich sind.

[1] `http://de.wikipedia.org/wiki/Ungarische_Notation`

Im Gegensatz zu anderen Programmiersprachen gibt es in C keinen eigenen Datentyp, um die Werte von Zeichenketten in einer Variable zu speichern. Das ist schade, aber wir werden lernen damit umzugehen. Leider brauchen wir dafür noch ein tieferes Verständnis der Programmiersprache, so dass ich erst später im Kapitel 8 darauf zurückkommen werde. Für den Anfang beschränken wir uns also auf Zahlen und einzelne Zeichen.

ACHTUNG: Der ältere C89 Standard erlaubt die Deklaration von Variablen nur am Anfang eines neuen Abschnitts (Block) im Quelltext. Es ist deshalb sinnvoll, alle in einer Funktion benötigten Variablen gleich am Anfang der Funktion zu deklarieren – selbst wenn wir diese Variable erst viel später in unserem Programm benutzen wollen.

2.5 Zuweisung

Nachdem wir jetzt einige Variablen erzeugt haben, möchten wir gern Werte in ihnen speichern. Dazu benutzen wir eine Anweisung, die eine **Zuweisung** vornimmt:

```
first_letter = 'a';   /* give first_letter the value 'a' */
hour = 11;            /* assign the value 11 to hour */
minute = 59;          /* set minute to 59 */
```

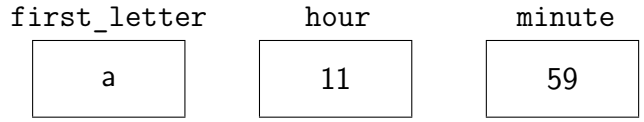
Dieses Beispiel zeigt drei Zuweisungen und die Kommentare geben uns drei Beispiele, wie Programmierer über den Vorgang des Speicherns eines Wertes in einer Variable sprechen. Das Vokabular ist vielleicht etwas verwirrend, aber die Idee ist eigentlich ziemlich einfach zu beschreiben:

1. Wenn wir eine Variable deklarieren, erschaffen wir eine benannte Speicherstelle.

2. Wenn wir eine Zuweisung zu dieser Variable vornehmen, speichern wir einen Wert in dieser Speicherstelle.

Wir können dabei den zweiten Schritt nicht vor dem ersten Schritt tun. Sollten wir den zweiten Schritt vergessen, lässt sich die Variable durchaus verwenden (zum Beispiel in einer Ausgabeanweisung), der Wert der Variable ist aber nicht bestimmt. Man sagt dazu, die Variable ist nicht initialisiert. Der Compiler legt in der Regel beim Erstellen einer Variable keinen Anfangswert fest. Es ist deshalb eine gute Idee, einer Variablen einen definierten Anfangswert zuzuweisen – anderenfalls kann die Verwendung der Variable leicht zu schwer zu findenden Programmfehlern führen.

Eine weit verbreitete Methode Variablen auf Papier darzustellen, besteht darin einen Kasten mit dem Variablennamen zu zeichnen und den Wert der Variable hier einzutragen. Das folgende Diagramm zeigt den Effekt der drei Zuweisungsanweisungen:



Wir können auf diese Weise den aktuellen Zustand einer Variablen darstellen. Dieser Zustand ist abhängig von der Ausführung von Anweisungen in unserem Programm und kann sich während des Programmablaufs ändern. Solche Veränderungen kann man in einem **Zustandsdiagramm** darstellen.

Bei der Zuweisung von Werten an Variablen müssen wir aufpassen, dass der Typ des Werts mit dem Typ der Variable übereinstimmt. So können wir zum Beispiel keine Zeichenkette in einer `int` Variable speichern. Die folgende Anweisung führt zu einer Warnmeldung des Compilers:

```
int hour;
hour = "Hello.";       /* WRONG !! */
```

Diese Regel führt manchmal zu Verwirrungen, weil es viele Möglichkeiten gibt, Werte von einem Typ in einen anderen Typ zu konvertieren. Manchmal nimmt C diese Typumwandlung auch automatisch vor. Es hilft aber sich einzuprägen, dass Variablen und Werte den gleichen Typ haben müssen. Wir werden uns später um die Spezialfälle kümmern.

Eine weitere Quelle für Verwechslungen besteht darin, dass einige Zeichenketten wie Zahlen *aussehen*, aber keine sind. So ist zum Beispiel die Zeichenkette `"123"`, aus den Zeichen `1`, `2` und `3` zusammengesetzt. Der *String* `"123"` unterscheidet sich für den Computer grundlegend von der *Zahl* `123`. Die folgende Zuweisung ist illegal:

```
minute = "59";         /* WRONG!! */
```

2.6 Variablen ausgeben

Wir können die Werte von Variablen mit den selben Kommandos ausgeben, die wir auch für die Ausgabe von einfachen Werten genutzt haben:

```
int hour, minute;
char colon;

hour = 11;
minute = 59;
colon = ':';

printf ("The current time is ");
printf ("%i", hour);
printf ("%c", colon);
printf ("%i", minute);
```

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

Dieses Programmfragment erzeugt zwei *integer* Variablen mit Namen `hour` und `minute`, und die *character* Variable `colon`. Den Variablen werden geeignete Werte zugewiesen, um danach mit einer Reihe von Ausgabeanweisungen die folgende Nachricht auf dem Bildschirm auszugeben:

```
The current time is 11:59
```

Wenn wir davon sprechen eine Variable "auszugeben", meinen wir, dass wir den *Wert* der Variable ausgeben. Der Name einer Variable ist nur für den Programmierer wichtig. Wir erinnern uns, es ist ein Name für eine Speicherstelle. Das kompilierte Programm enthält diese für Menschen lesbare Referenzen nicht mehr. Den Benutzer interessiert nur noch der dort gespeicherte Wert.

Die `printf()` Ausgabeanweisung kann mehr als einen Wert in einer einzigen Anweisung ausgeben. Dafür müssen wir so viele Platzhalterzeichen wie auszugebende Werte in die Anweisung einfügen und danach die auszugebenden Werte mit Komma getrennt anfügen. Wichtig ist es dabei, auf die richtige Reihenfolge und den Typ der Werte zu achten. Damit können wir unser Programm folgendermaßen zusammenfassen:

```
int hour, minute;
char colon;

hour = 11;
minute = 59;
colon = ':';

printf ("The current time is %i%c%i\n", hour, colon, minute);
```

In einer Programmzeile können wir jetzt einen *string*, zwei *integer* und einen *character* Wert ausgeben. Sehr eindrucksvoll!

2.7 Schlüsselwörter

Einige Abschnitte zuvor sagte ich, dass wir für unsere Variablen beliebige Namen verwenden dürfen. Das war leider nicht ganz richtig.

Es gibt in C einige Namen, die reserviert sind, weil sie bereits vom Compiler genutzt werden, um die Struktur eines C Programms zu parsen. Wenn wir diese Wörter als Variablennamen verwenden, würden Mehrdeutigkeiten entstehen und der Compiler könnte nicht mehr auseinanderhalten, ob es sich dabei um den Variablennamen oder das reservierte Wort der Sprache handelt.

Die komplette Liste der Schlüsselwörter ist im jeweiligen C Standard festgelegt. Die hier aufgeführten Schlüsselwörter entsprechen der Sprachdefinition, wie sie

die Internationale Organisation für Normung (ISO) am 1. September 1998 festgelegt hat.

Die reservierten Wörter werden **Schlüsselwörter** genannt. In der folgenden Tabelle sind alle derzeit definierten Schlüsselwörter der Sprache aufgeführt.

Reservierte Schlüsselwörter der Sprache C

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

Anstatt diese Liste jetzt auswendig zu lernen, empfehle ich einen der Vorteile moderner Entwicklungsumgebungen zu nutzen: Syntaxhervorhebungen. Wenn wir den Quelltext in einer Entwicklungsumgebung wie Code::Blocks[2] eingeben, werden unterschiedliche Teile unseres Programms unterschiedlich farblich eingefärbt. So erscheinen beispielsweise Schlüsselwörter in der Farbe blau, Zeichenketten rot und alle anderen Befehle schwarz. Wenn wir jetzt einen Variablennamen eingeben und dieser in der Farbe blau erscheint, sollten wir aufpassen! Der Compiler wird wahrscheinlich ein seltsames Verhalten zeigen.

2.8 Mathematische Operatoren

Mathematische **Operatoren** sind spezielle Symbole, die dazu benutzt werden, einfache Berechnungen wie Addition und Multiplikation darzustellen. Viele der mathematischen Operatoren in C verhalten sich genauso, wie wir das von den gebräuchlichen mathematischen Symbolen kennen. So wird zum Beispiel das Zeichen `+` für die Addition von zwei Zahlen benutzt. Für die Multiplikation wird das Zeichen `*` und für die Division das Zeichen `/` verwendet.

Wenn wir Operatoren mit Operanden kombinieren, entsteht ein sogenannter **Ausdruck** der einen Wert repräsentiert. Ausdrücke können Variablen, Werte und Operatoren enthalten. In jedem Fall werden immer die Namen der Variablen durch die Werte der Variablen ersetzt, bevor die Berechnung (Auswertung) des Ausdrucks vorgenommen wird.

Die folgenden Ausdrücke der Sprache C sind legal und ihre Bedeutung erschließt sich praktisch von selbst:

```
1+1        hour-1       hour*60+minute     minute/60
```

[2] http://www.codeblocks.org/

Addition, Subtraktion und Multiplikation funktionieren in C so, wie wir das erwarten würden. Überrascht werden wir vom Ergebnis der Division. Schauen wir uns das folgende Programm an:

```
int hour, minute;
hour = 11;
minute = 59;
printf ("Minutes since midnight: %i\n", hour*60 + minute);
printf ("Fraction of the hour that has passed: %i\n", minute/60);
```

Es erzeugt die folgende Ausgabe:

```
Number of minutes since midnight: 719
Fraction of the hour that has passed: 0
```

Die erste Zeile der Ausgabe ist korrekt, aber die zweite Zeile ist seltsam. Der Wert der Variablen `minute` ist 59 und 59 dividiert durch 60 ergibt 0,98333, nicht 0. Der Grund für dieses Verhalten liegt darin, dass C für ganze Zahlen eine **ganzzahlige Division** durchführt.

Wenn beide **Operanden** ganzzahlige Werte sind, dann ist das Resultat der Berechnung ebenfalls ein ganzzahliger Wert. Nach der Definition wird dabei eine Division mit Rest durchgeführt. Das Ergebnis ist ein Ganzzahlquotient und ein Divisionsrest und hierbei wird niemals aufgerundet, selbst dann, wenn die nächste Ganzzahl noch so nah ist.

Eine mögliche Alternative wäre in unserem Fall statt des gebrochenen Anteils den Prozentwert auszurechnen:

```
printf ("Percentage of the hour that has passed: ");
printf ("%i\n", minute*100/60);
```

Das Resultat ist:

```
Percentage of the hour that has passed: 98
```

Auch hier fehlen wieder die Nachkommastellen, aber jetzt ist die Antwort wenigstens annähernd korrekt. Um eine noch genauere Antwort zu erhalten, müssen wir einen ganz neuen Typ von Variablen für die Berechnung benutzen: Fließkommazahlen. Mit diesem Variablentyp könne auch die Resultate von Brüchen und reellen Zahlen gespeichert werden. Wir werden darauf im nächsten Kapitel zurückkommen.

2.9 Rangfolge der Operatoren

Wenn mehr als ein Operator in einem Ausdruck enthalten ist, wird die Reihenfolge der Auswertung von **Vorrangregeln** zwischen den Operatoren bestimmt. Eine komplette Erklärung dieser Regeln wäre sehr umfangreich, so dass ich hier nur die Wichtigsten erwähnen möchte:

- Multiplikation und Division werden vor Addition und Subtraktion ausgeführt (Punktrechnung geht vor Strichrechnung). `2*3-1` ergibt 5, nicht 4 und `2/3-1` ergibt -1, nicht 1
- Wenn die Operatoren den selben Rang aufweisen, werden sie von links nach rechts ausgewertet. So wird in dem Ausdruck `minute*100/60`, zuerst die Multiplikation ausgeführt, was `5900/60` ergibt, die weitere Auswertung ergibt `98`.
- Immer dann, wenn wir diese Vorrangregeln ändern wollen (oder wenn wir uns nicht komplett sicher sind), können wir Klammern setzen. Klammern setzen die automatischen Vorrangregeln außer Kraft. Ausdrücke in Klammern werden zuerst ausgewertet, so wird `2*(3-1)` zu 4 ausgewertet. Weiterhin kann man Klammern dazu verwenden einen Ausdruck einfacher lesbar zu machen, wie in `(minute*100)/60`, obwohl das natürlich keine Auswirkungen auf das Resultat hat.

2.10 Operationen über Buchstaben

Interessanterweise können wir die gleichen mathematischen Operationen für *integer* Werte auch mit *character* Werten benutzen. So gibt zum Beispiel das folgende Programm,

```
char letter;
letter = 'a' + 1;
printf ("%c\n", letter);
```

den Buchstabe `b` auf dem Bildschirm aus. Der Grund dafür besteht in der Art und Weise, wie Buchstaben im Computer gespeichert werden. So ist es erlaubt Buchstaben auch zu multiplizieren, aber es ist sehr selten sinnvoll das zu tun.

Ich habe im Abschnitt 2.5 davon gesprochen, dass wir *integer* Variablen nur *integer* Werte zuweisen können und *character* Variablen nur *character* Werte. Ich sagte aber auch, dass es viele Ausnahmen von dieser Regel gibt. So ist das folgende Beispiel völlig legal:

```
int number;
number = 'a';
printf ("%i\n", number);
```

Das Resultat unseres Programms ist 97. Dabei handelt es sich um die Zahl die von C dazu benutzt wird um den Buchstaben `'a'` darzustellen (siehe Anhang ASCII-Tabelle). Wir müssen uns immer wieder klar machen, der Computer speichert alle Daten (Zahlen, Buchstaben, Töne, Bilder) als Folgen von Bits. Diese Bitfolgen können unterschiedlich *interpretiert* werden. Es ist allerdings eine gute Idee, Zahlen als Zahlen zu behandeln und Buchstaben als Buchstaben und nur dann eine Darstellung in die andere umzuwandeln, wenn dafür ein guter Grund besteht.

Die automatische Umwandlung eines Datentyps in einen anderen (engl.: *automatic type conversion*) ist ein Beispiel für ein häufiges Designproblem bei der Erschaffung einer Programmiersprache. Dabei besteht der Konflikt zwischen **Formalismus** und **Bequemlichkeit**. Der Formalismus gebietet, dass eine Programmiersprache einfache Regeln mit wenigen Ausnahmen aufweist. Die Bequemlichkeit fordert, dass eine Programmiersprache einfach benutzbar ist.

Im Falle von C hat die Bequemlichkeit gewonnen. Das ist gut für den erfahrenen Programmierer, er wird vor rigorosen, aber manchmal unhandlichen Formalismen verschont. Für den Programmieranfänger ist das allerdings schlecht, weil er oft von der Komplexität der Regeln und der Vielzahl von Ausnahmefällen verwirrt ist. In diesem Buch habe ich versucht den Einstieg in die Programmierung mit C dadurch zu vereinfachen, dass ich die generellen Regeln hervorhebe und viele der bestehenden Ausnahmen weglasse.

2.11 Komposition

Bisher haben wir uns die Elemente einer Programmiersprache – Variablen, Ausdrücke und Anweisungen – jeweils einzeln betrachtet, ohne darüber nachzudenken, wie wir sie miteinander kombinieren können.

Eine der nützlichsten Eigenschaften einer Programmiersprache besteht darin, einzelne kleine Bausteine zu nehmen und diese zu einer mächtigeren Konstruktion zusammenzusetzen. So wissen wir zum Beispiel, wie man ganzzahlige Werte multipliziert, und wir wissen, wie man Variablen ausgibt. Es stellt sich heraus, dass wir beides zur gleichen Zeit tun können:

```
printf ("%i\n", 17 * 3);
```

Also eigentlich sollte ich nicht sagen “zur gleichen Zeit”, weil die Multiplikation bereits ausgeführt sein muss, bevor das Ergebnis auf dem Bildschirm angezeigt wird.

Worauf ich hinaus will, ist das folgende: Es ist möglich, beliebige Ausdrücke (bestehend aus Zahlen, Buchstaben, Variablen und Operatoren) in einer Anweisung zu benutzen. Der Ausdruck wird vom Computer ausgewertet und mit dem aktuell ermittelten Wert weitergearbeitet. Wir haben dafür bereits ein Beispiel gesehen:

```
printf ("%i\n", hour * 60 + minute);
```

Wir können auch beliebige Ausdrücke auf die rechte Seite einer Zuweisungsanweisung schreiben:

```
int percentage;
percentage = (minute * 100) / 60;
```

Diese Fähigkeit ist momentan noch nicht sehr eindrucksvoll, aber wir werden weitere Beispiele sehen, wo wir mit Hilfe der Fähigkeit zur Komposition komplexe Berechnungen einfach und präzise ausdrücken können.

ACHTUNG: Es existieren ein paar Einschränkungen, wo wir bestimmte Ausdrücke verwenden können. So dürfen sich auf der linken Seite einer Zuweisungsanweisung nur *Variablen* und keine zusammengesetzten Ausdrücke befinden. Der Grund dafür liegt darin, dass die linke Seite einer Zuweisung eine Speicherstelle darstellt, der ein Wert zugewiesen wird. Ausdrücke repräsentieren keine Speicherstelle, nur Werte. Somit wäre die folgende Schreibweise illegal:

```
minute + 1 = hour;      /*WRONG!! */
```

2.12 Glossar

Variable (engl.: *variable*): Eine benannte Stelle im Hauptspeicher. Dort können Werte abgelegt, verändert und wieder gefunden werden. Alle Variablen haben einen Typ. Dieser legt fest, welche Werte gespeichert werden können.

Wert (engl.: *value*): Ein anderer Begriff für digitale Daten. Das können Buchstaben, Zahlen oder andere Repräsentation von Information sein. Diese lassen sich verarbeiten und in Variablen speichern.

Typ (engl.: *type*): Die Kategorie der Daten. Datentypen mit denen wir bisher gearbeitet haben sind die ganzen Zahlen (`int` in C) und Buchstaben (`char` in C).

Schlüsselwort (engl.: *keyword*): Ein Wort, welches in dieser Programmiersprache eine bestimmte Bedeutung hat, und nicht als Name von Variablen oder Funktionen verwendet werden darf. Bisher haben wir die Schlüsselwörter `int`, `void` und `char` verwendet.

Anweisung (engl.: *statement*): Eine Befehlszeile die einen abgeschlossenen Schritt (Kommando, Aktion) in einem Programm darstellt. In C werden Anweisungen mit einem Semikolon beendet.

Deklaration (engl.: *declaration*): Eine Anweisung welche den Typ und den Namen einer Variable festlegt.

Zuweisung (engl.: *assignment*): Eine Anweisung welche einer Variablen (einer Speicherstelle) einen Wert zuweist.

Ausdruck (engl.: *expression*): Eine Kombination von Variablen, Operatoren und Werten, welche zu einem Resultat ausgewertet werden, also wiederum einen Wert darstellen. Ausdrücke haben einen bestimmten Typ, der durch die verwendeten Operatoren und Operanden bestimmt wird.

Operator (engl.: *operator*): Ein spezielles Symbol, welches eine einfache Verknüpfung oder Berechnung darstellt (z.B. Multiplikation oder Addition).

Operand (engl.: *operand*): Einer der Werte mit denen ein Operator eine Operation durchführt.

Vorrang (engl.: *precedence*): Die Reihenfolge in der die einzelnen Teilschritte einer Operation mit mehreren Operatoren ausgewertet wird.

Komposition (engl.: *composition*): Die Fähigkeit einfache Ausdrücke und Anweisungen zu komplexeren Ausdrücken und Anweisungen zusammenzufassen. Komplexe Probleme und Sachverhalte lassen sich durch die geeignete, schrittweise Ausführung einfacher Anweisungen beschreiben und lösen.

2.13 Übungsaufgaben

Übung 2.1

a. Erstellen Sie ein neues Programm mit dem Namen `MyDate.c`. Kopieren Sie dazu die Struktur des "Hello, WorldProgramms und stellen Sie sicher, dass Sie dieses kompilieren und ausführen können.

b. Folgen Sie dem Beispiel in Abschnitt 2.6 und definieren Sie in dem Programm die folgenden Variablen: `day`, `month` und `year`. `day` enthält den Tag des Monats, `month` den Monat und `year` das Jahr. Von welchem Typ sind diese Variablen?

   Weisen Sie den Variablen Werte zu, welche dem heutigen Datum entsprechen.

c. Geben Sie die Werte auf dem Bildschirm aus. Stellen Sie jeden Wert auf einer eigenen Bildschirmzeile dar. Das ist ein Zwischenschritt, der ihnen dabei hilft zu überprüfen, ob das Programm funktionsfähig ist.

d. Modifizieren Sie das Programm dahingehend, dass es das Datum im amerikanischen Standardformat darstellt: `mm/dd/yyyy`.

e. Modifizieren Sie das Programm erneut, um eine Ausgabe nach folgendem Muster zu erzeugen:

```
American format:
3/18/2009
European format:
18.3.2009
```

Diese Übung soll Ihnen dabei helfen, formatierte Ausgaben von Werten unterschiedlicher Datentypen mittels der `printf` Funktion zu erzeugen. Weiterhin sollen Sie die kontinuierliche Entwicklung von komplexen Programmen durch das schrittweise Hinzufügen von einigen, wenigen Anweisungen erlernen.

Übung 2.2

a. Erstellen Sie ein neues Programm mit dem Namen `MyTime.c`. In den nachfolgenden Aufgaben werde ich Sie nicht mehr daran erinnern mit einem kleinen, funktionsfähigen Programm zu beginnen. Allerdings sollten Sie dieses auch weiterhin tun.

b. Folgen Sie dem Beispiel im Abschnitt 2.8 und erstellen Sie Variablen mit dem Namen `hour`, `minute` und `second`. Weisen Sie den Variablen Werte zu, welche in etwa der aktuellen Zeit entsprechen. Benutzen Sie dazu das 24-Stunden Zeitformat.

c. Das Programm soll die Anzahl der Sekunden seit Mitternacht berechnen.

d. Das Programm soll die Anzahl der noch verbleibenden Sekunden des Tages berechnen und ausgeben.

e. Das Programm soll berechnen, wieviel Prozent des Tages bereits verstrichen sind und diesen Wert ausgeben.

f. Verändern Sie die Werte von `hour`, `minute` und `second`, um die aktuelle Zeit wiederzugeben. Überprüfen Sie, ob das Programm mit unterschiedlichen Werten korrekt arbeitet.

In dieser Übung führen Sie arithmetische Operationen durch und beginnen darüber nachzudenken, wie komplexere Datenobjekte, wie z.B. die Uhrzeit, als Zusammensetzung von mehreren Werten dargestellt werden können. Weiterhin entdecken Sie möglicherweise Probleme, die sich aus der Darstellung und Berechnung mit dem ganzzahligen Datentypen `int` ergeben (Prozentberechnung). Diese Probleme können mit der Verwendung von Fließkommazahlen umgangen werden (siehe nächstes Kapitel).

HINWEIS: Sie können weitere Variablen benutzen, um Zwischenergebnisse der Berechnung abzulegen. Diese Variablen, welche in einer Berechnung genutzt, aber niemals ausgegeben werden, bezeichnet man auch als temporäre Variablen.

Kapitel 3

Funktionen

3.1 Fließkommazahlen

Im letzten Kapitel hatten wir Problem mit der der Darstellung von Ergebnissen die keine ganzen Zahlen waren. Wir haben uns dadurch geholfen, dass wir keine Dezimalbrüche sondern Prozentwerte benutzt haben. Eine bessere Lösung stellt die Verwendung des Datentyps der Fließkommazahlen dar. Eine Fließkommazahl, auch als Gleitkommazahl bezeichnet, kann auch gebrochene Anteile reeller Zahlen darstellen.

In C existieren zwei Arten der Darstellung von Fließkommazahlen, genannt `float` und `double`. Für Computer ist es schwierig das mathematische Konzept der Unendlichkeit zu realisieren, da der verfügbare Speicher begrenzt ist. So weist das Ergebnis der folgenden einfachen Division, $10 : 3 = 3,\overline{3}$ unendlich viele Nachkommastellen auf. Die Datentypen `float` und `double` unterscheiden sich darin, wie genau das Ergebnis dargestellt wird. In diesem Buch verwenden wir ausschließlich `double`s.

Wir können Fließkomma-Variablen erzeugen und ihnen Werte zuweisen mit der gleichen Syntax die wir für andere Datentypen benutzen. Ein Beispiel:

```
double pi;
pi = 3.14159;
```

Es ist auch erlaubt eine Variable zu deklarieren und ihr zur gleichen Zeit einen Wert zuzuweisen:

```
int x = 1;
char first_char = 'a';
double pi = 3.14159;
```

Diese Syntax ist sogar ziemlich weit verbreitet. Die kombinierte Deklaration und Zuweisung wird auch als **Initialisierung** bezeichnet.

WICHTIG: Für die Darstellung von Dezimalbrüchen wird im englischen Sprachraum ein Punkt statt eines Kommas verwendet. C verwendet ebenfalls diese Schreibweise. Die Verwendung des Dezimalkommas stellt einen Syntaxfehler dar.

Obwohl Fließkommazahlen nützlich sind, führen Sie doch auch zu Verwirrungen, weil es eine scheinbare Überschneidung zwischen ganzen Zahlen und Fließkommazahlen gibt. Wenn wir uns zum Beispiel den Wert `1` anschauen, ist das eine ganze Zahl, eine Fließkommazahl, oder beides?

Genau genommen unterscheidet C zwischen dem Wert der ganzen Zahl `1` und dem Wert der Fließkommazahl `1.0`, obwohl beide die gleiche Zahl repräsentieren. Diese Werte gehören zu unterschiedlichen Datentypen und wir hatten die generelle Regel aufgestellt, dass es nicht erlaubt ist Zuweisungen zwischen unterschiedlichen Datentypen durchzuführen. Im folgenden Beispiel,

```
double y = 1;
```

ist die Variable auf der linken Seite ein `double` und der Wert auf der rechten Seite ein `int`. Allerdings führt die Zuweisung nicht zu einer Fehlermeldung des Compilers. C nimmt an dieser Stelle eine automatische Typumwandlung vor. Die automatische Umwandlung ist für den Programmierer bequem, kann aber zu allerlei Problemen führen. So gehen uns im folgenden Beispiel alle Nachkommastellen verloren:

```
int x = 1.98;                    /* Ergebnis: x = 1 */
```

Das Ergebnis des folgenden Programmcodes ist für Anfänger besonders verwunderlich:

```
double y = 1 / 3;                /* Ergebnis: y = 0.0 */
```

Es wäre zu erwarten, dass der Variablen `y` der Wert `0.333333` zugewiesen wird, allerdings hat sie den Wert `0.0`.
Der Grund liegt in der Art und Weise, wie der Compiler die Anweisung auswertet und ausführt. Der Compiler untersucht zuerst den Ausdruck auf der rechten Seite der Zuweisung (rechts vom `=`). Dieser Ausdruck beschreibt ein Verhältnis zwischen zwei ganzen Zahlen. Bei der Division ganzer Zahlen wird eine *Division mit Rest* durchgeführt, welche den ganzzahligen Wert `0` zum Ergebnis hat. Erst bei der darauf folgenden Zuweisung an die Variable, wird dieser Wert in eine Fließkommazahl konvertiert und das Resultat beträgt jetzt `0.0`.

Eine Möglichkeit dieses Problem zu lösen (nachdem wir die Ursache herausgefunden haben) besteht darin den Ausdruck auf der rechten Seite als Fließkommazahl darzustellen:

```
double y = 1.0 / 3.0;
```

Dadurch wird `y` der erwartete Wert `0.333333` zugewiesen. Wir müssen also beachten, dass immer erst der rechte Ausdruck einer Zuweisung komplett ausgewertet und anschließend die Zuweisung durchgeführt wird.

Alle Berechnungen die wir bisher gesehen haben – Addition, Subtraktion, Multiplikation und Division – funktionieren sowohl für Fließkommazahlen als auch für ganze Zahlen. Allerdings müssen wir wissen, dass die grundlegenden Mechanismen der Speicherung und Verarbeitung dieser Zahlen komplett verschieden realisiert sind. Die meisten modernen Prozessoren haben spezielle Hardwareerweiterungen für die Verarbeitung von Fließkommazahlen. Es ist ebenfalls wichtig zu beachten, dass Fließkommazahlen nur mit eingeschränkter Genauigkeit dargestellt werden können, was zu Rundungsverlusten bei Berechnungen führen kann.

3.2 Konstanten

Im letzten Abschnitt haben wir den Wert 3.14159 einer Variable vom Typ `double` zugewiesen. Variablen haben die wichtige Eigenschaft, dass unser Programm in ihnen unterschiedliche Werte zu unterschiedlichen Zeiten speichern kann.

So können wir zum Beispiel der Variablen `pi` aktuell den Wert 3.14159 zuweisen und ihr später erneut einen anderen Wert geben:

```
double pi = 3.14159;
...
pi = 10.999;  /* wahrscheinlich ein logischer Fehler */
```

Der zweite Wert hat nichts mehr mit dem ursprünglich `pi` genannten Wert in unserem Programm zu tun. Der Wert von π ist konstant und ändert sich nicht im Laufe der Zeit. Wenn wir die Speicherstelle `pi` dazu benutzen auch beliebige andere Werte zu speichern, kann das zu schwer zu findenden Fehlern in unserem Programm führen.

Wir benötigen eine Möglichkeit um festzulegen, dass es sich bei einer Speicherstelle um einen konstanten Wert handeln soll, der während des Programmablaufs nicht verändert werden darf. In C müssen wir dazu zusätzlich das Schlüsselwort `const` bei der Deklaration der Speicherstelle angeben. Weiterhin ist es natürlich erforderlich auch den Datentyp der Konstanten anzugeben.

Der Wert der Konstanten muss zum Zeitpunkt der Deklaration festgelegt werden (Initialisierung). Im weiteren Ablauf des Programms kann dieser Wert nicht mehr verändert werden. Um Konstanten visuell von Variablen zu unterscheiden, verwenden wir für die Namen von Konstanten ausschließlich Großbuchstaben.

```
const double PI = 3.14159;
printf ("Pi: %f\n", PI);
 ...
PI = 10.999;  /* falsch, Fehler wird vom Compiler gefunden */
```

Es ist nicht länger möglich den Wert von `PI` nachträglich zu verändern. Unabhängig davon können wir aber Konstanten genau so in Berechnungen zu verwenden wie Variablen.

3.3 Explizite Umwandlung von Datentypen

Wie wir gesehen haben wandelt C automatisch zwischen den numerischen Datentypen um, wenn nötig. Manchmal ist es jedoch sinnvoll die Konvertierung nicht dem Kompiler zu überlassen, sondern selbst zu bestimmen wann und wie eine Typumwandlung durchgeführt werden soll.

In dem folgenden Beispiel soll das Ergebnis der Berechnung als Dezimalbruch ermittelt werden:

```
int x = 9;
int y = 2;
double result = x / y;      /* result = 4.0 */
```

Wir erinnern uns, bei Operanden vom Typ `int` führt C automatisch eine Ganzzahldivision durch und das Ergebnis entspricht nicht unserer Anforderung. Wir müssten die Werte in dem Ausdruck auf der rechten Seite von `int` nach `double` wandeln, damit die Berechnung das korrekte Ergebnis liefert.

Um eine ganze Zahl in eine Fließkommazahl zu wandeln, können wir einen **Cast** oder **Typecast** benutzen. Typecasting erlaubt es uns in einem Ausdruck so zu tun, als hätten wir einen anderen Datentyp vorliegen. Der Wert selbst wird dabei nicht verändert und auch der Typ der ursprünglichen Variablen oder Konstanten ändert sich nicht.

Die Syntax für die Typumwandlung erfordert die explizite Angabe des Zieltyps `(type)` in Klammern vor dem umzuwandelnden Ausdruck. Zum Beispiel:

```
int x = 9;
int y = 2;
double result = (double) x / (double) y;     /* result = 4.5 */
```

Der `(double)` Operator interpretiert `x` und `y` als Fließkommatypen und das Ergebnis der Berechnung liefert uns den erwarteten Dezimalbruch.

Für jeden Datentyp in C, gibt es einen entsprechenden Operator welcher das Argument in den entsprechenden Typ umwandelt.

3.4 Mathematische Funktionen

Aus dem Bereich der Mathematik kennen wir Funktionen wie sin und log und wir haben gelernt mathematische Ausdrücke wie $\sin(\pi/2)$ und $\log(1/x)$ auszuwerten. Dazu wird zuerst der Ausdruck in Klammern, das **Argument** der Funktion, ausgewertet. So ist zum Beispiel, $\pi/2$ näherungsweise 1.571 und $1/x$ ergibt 0.1 (wenn wir für x den Wert 10 annehmen). Danach wird die Funktion selbst berechnet, entweder durch Nachschlagen in einer Tabelle oder über die Durchführung verschiedener Berechnungen. Der Sinus (sin) von 1.571 ist 1 und der Logarithmus (log) von 0.1 ist -1 (unter der Annahme, dass log den Logarithmus zur Basis 10 darstellt).

Dieser Prozess kann mehrfach wiederholt werden, um immer kompliziertere Ausdrücke, wie $\log(1/\sin(\pi/2))$ auszuwerten. Zuerst werten wir die innersten Funktionen aus, danach die umgebenden Funktionen und immer so weiter.

C bietet uns eine Reihe von eingebauten Funktionen, welche die meisten bekannten mathematischen Operationen unterstützen. Die mathematischen Funktionen werden in einer der mathematischen Schreibweise ähnelnden Form aufgerufen:

```
double my_log = log (17.0);
double angle = 1.5;
double height = sin (angle);
```

Das erste Beispiel weist der Variable `my_log` den Logarithmus von 17 zur Basis *e* zu. Es existiert auch eine Funktion `log10` welche den Logarithmus zur Basis 10 ermittelt.

Das zweite Beispiel findet den Sinus des Werts der Variablen `angle`. C nimmt an, dass die Werte die wir mit `sin()` und anderen trigonometrischen Funktionen (`cos`, `tan`) benutzen in *Radian* (rad) vorliegen. Um von Grad in Radiant umzurechnen müssen wir den Wert durch 360 dividieren und mit 2π multiplizieren.

Wenn uns momentan nicht einfällt wie die ersten 15 Ziffern von π lauten, so können wir diese mit der `acos()` Funktion berechnen. Der Arkuskosinus (oder invertierter Kosinus) von -1 ist π (da der Kosinus von π als Ergebnis -1 liefert).

```
const double PI = acos(-1.0);
double degrees = 90;
double angle = degrees * 2 * PI / 360.0;
```

WICHTIG: Bevor wir mathematische Funktionen in unserem Programm verwenden können, müssen wir die mathematische Bibliothek in unser Programm einbinden. Dafür reicht es die entsprechende **Header-Datei** in unser Programm einzufügen.

Header-Dateien enthalten Informationen die der Kompiler benötigt um Funktionen nutzen zu können, die außerhalb unseres Progamms definiert wurden. So haben wir zum Beispiel in unserem "Hello, world!"- Programm eine Header-Datei mit dem Namen `stdio.h` eingefügt. Zum Einfügen nutzen wir den **include** Befehl:

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

Die Header-Datei `stdio.h` enthält Informationen über die in C verfügbaren über Ein- und Ausgabefunktionen (Input und Output – I/O). Gleichermaßen enthält die Bibliothek `math.h` Informationen über mathematische Funktionen in C.
Wir können diese Datei zusammen mit `stdio.h` in unser Programm einbinden:

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

3.5 Komposition

So wie in der Mathematik, können wir in C Funktionen zusammensetzten. Das heißt, wir können einen Ausdruck als Teil eines anderen verwenden. So lässt sich zum Beispiel ein beliebiger Ausdruck als Argument einer Funktion verwenden:

```
double x = cos (angle + PI/2);
```

Diese Anweisung nimmt den Wert von `PI`, teilt ihn durch zwei und addiert das Ergebnis zum Wert von `angle`. Die Summe wird dann als Argument der `cos()` Funktion benutzt.

Wir können auch das Resultat einer Funktion nehmen und es als Argument einer weiteren Funktion benutzen:

```
double x = exp (log (10.0));
```

Diese Anweisung ermittelt zuerst den Logarithmus zur Basis e von 10 um e anschließend mit diesem Ergebnis zu potenzieren. Das Resultat wird der Variablen `x` zugewiesen. Ich hoffe Ihnen ist klar, was das Ergebnis ist.

3.6 Hinzufügen neuer Funktionen

Bisher haben wir ausschließlich Funktionen benutzt die bereits Teil der Programmiersprache sind. Es ist aber genauso gut möglich neue Funktionen hinzuzufügen. Eigentlich haben wir das auch bereits getan, ohne es zu bemerken. Wir haben unserem Programm bereits eine Funktion hinzugefügt: `main`. Die Funktion mit Namen `main` ist ganz besonders wichtig, weil sie angibt, wo in einem Programm die Ausführung der Befehle beginnt. Die Syntax für die Definition von `main()` ist aber die gleiche wie für jede andere Funktion auch:

```
DATENTYP FUNKTIONSNAME ( LISTE DER PARAMETER )
{
    ANWEISUNGEN;
}
```

Wir können einer neuen Funktion einen beliebigen Namen geben, wir dürfen sie aber nicht `main` oder nach einem Schlüsselwort der Programmiersprache benennen. Die Liste der Parameter gibt an, welche Informationen bereitgestellt werden müssen, um die neue Funktion zu nutzen (oder **aufzurufen**). Diese Liste kann auch leer sein. In diesem Fall machen wir mit dem Schlüsselwort `void` klar, dass es keine Parameter gibt, hier also nichts vergessen wurde.

Die `main()`-Funktion hat in unseren Beispielen keine Parameter, wie wir aus dem Schlüsselwort `(void)` zwischen den Klammern der Funktionsdefinition entnehmen können. Die ersten Funktionen die wir schreiben, kommen ebenfalls ohne Parameter aus, sie haben auch keinen festgelegten Datentyp, so das die Syntax wie folgt aussieht:

```
void PrintNewLine (void)
{
    printf ("\n");
}
```

Diese Funktion mit Namen `PrintNewLine()` enthält nur eine einzige Anweisung, welche eine Leerzeile auf dem Bildschirm darstellt.

Der Name unserer neuen Funktion beginnt mit einem Großbuchstaben. Die folgenden Worte des Funktionsnamens haben wir ebenfalls groß geschrieben. Dabei handelt es sich um eine oft genutzte Konvention, der wir in diesem Buch folgen werden. Die Großschreibung verhindert es, dass wir zufällig den gleichen Namen wie eine bereits existierende Funktion wählen. Der Name der Funktion selbst sollte eine treffende Beschreibung der Aufgabe dieser Funktion sein.

In unser `main()`-Funktion können wir die neue Funktion jetzt aufrufen. Dazu benutzen wir eine Syntax die vergleichbar ist mit dem Aufruf der standardmäßig in C vorhandenen Funktionen:

```
int main (void)
{
   printf ("First Line.\n");
   PrintNewLine ();
   printf ("Second Line.\n");
   return EXIT_SUCCESS;
}
```

Die Ausgabe des Programms sieht dann folgendermaßen aus:

```
First line.

Second line.
```

Haben Sie den zusätzlichen Abstand zwischen den beiden Zeilen bemerkt? Wie können wir noch mehr Abstand zwischen den Texten erzeugen? Wir rufen die Funktion einfach mehrfach nacheinander auf:

```
int main (void)
{
    printf ("First Line.\n");
    NewLine ();
    NewLine ();
    NewLine ();
    printf ("Second Line.\n");
    return EXIT_SUCCESS;
}
```

Oder wir könnten eine neue Funktion schreiben die wir `PrintThreeLines()` nennen und diese Funktion gibt drei Leerzeilen auf dem Bildschirm aus:

```
void PrintThreeLines (void)
{
  PrintNewLine ();  PrintNewLine ();  PrintNewLine ();
}

int main (void)
{
    printf ("First Line.\n");
    PrintThreeLines ();
    printf ("Second Line.\n");
    return EXIT_SUCCESS;
}
```

Ein paar Dinge in diesem Programm sollten besonders beachtet werden:

- Wir können die selbe Funktion mehrfach aufrufen. Das kommt in der Tat in den meisten Programmen ziemlich oft vor und ist ein sinnvoller Umgang mit Funktionen.
- Wir können aus einer Funktion eine andere Funktion aufrufen. In unserem Fall ruf `main()` die Funktion `PrintThreeLines()` und

`PrintThreeLines()` die Funktion `PrintNewLine()` auf. Auch das ist nützlich und verbreitet der Fall.

- In `PrintThreeLines()` habe ich drei Anweisungen in die gleiche Zeile geschrieben. Einerseits ist das syntaktisch erlaubt (Leerzeichen und Zeilenumbrüche ändern typischerweise die Bedeutung unseres Programm nicht). Andererseits ist es besser jede Anweisung in eine eigene Programmzeile zu schreiben – das Programm wird dadurch einfacher lesbar. Ich weiche manchmal von dieser Regel ab, um in diesem Buch etwas Platz zu sparen.

Bisher dürfte vielleicht noch nicht klar geworden sein, warum wir uns die Mühe mit der Erstellung dieser neuen Funktionen machen. Es gibt dafür eine ganze Reihe von Gründen, ich möchte an dieser Stelle aber nur auf zwei von ihnen eingehen:

1. In dem wir einen neue Funktion erstellen, erhalten wir die Möglichkeit einer Gruppe von Anweisungen einen Namen zu geben. Funktionen können die Struktur eines Programms vereinfachen. Wir verbergen komplexe Berechnungen hinter einem einfachen Befehl. Wir können für diesen Befehl 'sprechende' Bezeichnungen wählen, statt direkt obskuren Programmcode lesen zu müssen. Was ist klarer `PrintNewLine()` oder `printf("\n")`?

2. Durch die Verwendung von eigenen Funktionen können wir ein Programm kürzer machen, indem wir wiederholende Anweisungen eliminieren. So können wir auf einfache Weise neun aufeinanderfolgende Lehrzeilen dadurch erzeugen, dass wir `PrintThreeLines()` drei mal nacheinander aufrufen. Wie gehen wir vor, wenn wir 27 Lehrzeilen ausgeben müssen?

3.7 Definitionen und Aufrufe

Wenn wir alle Code-Fragmente der vorigen Abschnitte zusammenfügen, sieht das komplette Programm wie folgt aus:

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

void PrintNewLine (void)
{
    printf ("\n");
}

void PrintThreeLines (void)
{
    PrintNewLine ();  PrintNewLine ();  PrintNewLine ();
}

int main (void)
```

```
{
   printf ("First Line.\n");
   PrintThreeLines ();
   printf ("Second Line.\n");
   return EXIT_SUCCESS;
}
```

Das Programm enthält drei Funktionsdefinitionen: `PrintNewLine()`, `PrintThreeLine()` und `main()`.

Innerhalb der Definition von `main()` befindet sich eine Anweisung welche die Funktion `PrintThreeLine()` aufruft. Auf die gleiche Art ruft `PrintThreeLine()` die Funktion `PrintNewLine()` drei mal auf. Auffallend ist, dass die Definition jeder Funktion vor der Stelle erfolgt, an der die Funktion aufgerufen wird.

Diese Reihenfolge ist in C notwendig. Die Deklaration einer Funktion muss erfolgt sein, bevor die Funktion verwendet werden kann. Der Grund dafür liegt in der Art und Weise wie der Compiler versucht unser Programm zu übersetzen. Er geht dabei den Quellcode nur ein einziges mal durch und muss daher bereits alle Funktionen kennen, bevor sie verwendet werden.

Sie können ja einmal versuchen die Reihenfolge der Funktionen zu vertauschen und das Programm zu kompilieren. Dies wird in den meisten Fällen zu einer Fehlermeldung des Kompilers führen.

Wenn uns diese Verhalten nicht gefällt, können wir es durch das Hinzufügen eines **Funktionsprototyp** ändern. Ein Funktionsprototyp erlaub es dem Compiler zu erkennen, welche Funktionsnamen, Typen und Parameter in unserem Programm verwendet werden, bevor wir die Funktion überhaupt aufgeschrieben haben. Dazu müssen wir vor `main()` die folgenden Zeilen einfügen. Danach ist die Reihenfolge der Funktionsdefinition nicht mehr wichtig.
ACHTUNG: Funktionsprotoypen müssen mit einem Semikolon (;) abgeschlossen werden:

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

void PrintNewLine (void);
void PrintThreeLines (void);

int main (void)
...
```

3.8 Programme mit mehreren Funktionen

Wenn wir uns den Quelltext eines C Programms mit mehreren Funktionen anschauen, so ist es verführerisch, diesen von oben nach unten durchzulesen (als Menschen lesen wir Texte so).

Das kann allerdings schnell zur Verwirrung führen, da die einzelnen Funktionen in der Regel unabhängig voneinander sind. Für das Verständnis ist es deshalb besser den Quelltext in der **Reihenfolge der Programmausführung** zu lesen.

Die Ausführung beginnt immer mit der ersten Anweisung in `main()`, unabhängig, davon, wo sich die `main()`-Funktion im Programm befindet (sehr oft ist dies am Ende des Programms). Anweisungen werden der Reihe nach, eine nach der anderen ausgeführt, bis wir einen Funktionsaufruf erreichen. Funktionsaufrufe können wir uns wie eine Umleitung in der Programmausführung vorstellen. Anstatt die nächstfolgende Anweisung zu bearbeiten wird die Ausführung des Programms mit der ersten Anweisung der aufgerufenen Funktion fortgesetzt. Es werden dann alle Anweisungen der Funktion ausgeführt und erst danach wird die Programmausführung an die Stelle im Programm zurückgekehrt von der wir die Funktion aufgerufen haben.

Das hört sich einfach genug an, allerdings müssen wir bedenken, dass eine Funktion wiederum selbst andere Funktionen aufrufen kann. So kommt es, dass während wir uns mitten in der `main()`-Funktion befinden die Ausführung plötzlich in `PrintThreeLines()` fortgesetzt wird und die Anweisungen dort ausgeführt werden. Bei der Ausführung von `PrintThreeLines()` werden wir drei mal unterbrochen um Anweisungen in der Funktion `PrintNewLine()` auszuführen.

Glücklicherweise ist C geschickt darin die Übersicht über die Programmausführung zu behalten. Jedes mal, wenn `PrintNewLine()` abgearbeitet wurde, setzt das Programm genau an der Stelle fort, wo `PrintThreeLine()` verlassen wurde und kehrt schließlich zurück zu `main()` um das Programm erfolgreich zu beenden.

Was ist die Moral dieser Geschichte? Wenn wir ein Programm verstehen wollen sollten wir besser nicht von oben nach unten lesen, sondern dem Fluss der Programmausführung folgen.

3.9 Parameter und Argumente

Einige der eingebauten Funktionen die wir bisher benutzt haben hatten **Parameter**. Ein Parameter ist ein Objekt welches die Funktion benötigt um ihre Aufgabe zu erfüllen. Wenn wir zum Beispiel den Sinus einer Zahl bestimmen wollen, so müssen wir angeben um welche Zahl es sich dabei handelt. Demzufolge hat die `sin()`-Funktion einen `double` Wert als Parameter.

Einige Funktionen besitzen mehr als einen Parameter, wie zum Beispiel die `pow()`-Funktion zu Potenzberechnung, welche zwei `double`s, Basis und Exponent, für die Berechnung benötigt.

Wir müssen weiterhin beachten, dass in allen diesen Fällen nicht nur die Anzahl der Parameter eine Rolle spielt, sondern auch die Beachtung des Datentyps wichtig ist. Es sollte also nicht überraschen, dass wir in der Funktionsdefinition auch den jeweiligen Typ eines Parameters in der Parameterliste mit angeben müssen. Ein Beispiel:

```
void PrintTwice (char phil)
{
    printf("%c%c\n", phil, phil);
}
```

Diese Funktion hat einen Parameter, mit Namen `phil`, dieser ist vom Typ `char`. Je nachdem welchen Wert der Parameter währen des Funktionsaufrufs hat (und zu diesem Zeitpunkt haben wir keine Idee welcher Wert das ist), dieser Wert wird zwei mal ausgegeben, gefolgt von einem Zeilenumbruch. Ich habe den Namen `phil` für den Parameter gewählt um zu zeigen, dass der Name des Parameters völlig nebensächlich ist. Wir können den Namen frei wählen, meistens ist es sinnvoll anstelle von sinnfreien Namen wie `phil` 'sprechende' Bezeichner zu verwenden.

Um diese Funktion in unserem Programm aufzurufen, müssen wir einen `char` Wert angeben. Unsere `main()`-Funktion könnte zum Beispiel folgendermaßen aussehen:

```
int main (void)
{
    PrintTwice ('a');
    return EXIT_SUCCESS;
}
```

Der bereitgestellte `char` Wert wird **Argument** genannt und man sagt das Argument wird an die Funktion **übergeben**. In unserem Fall wird der Wert `'a'` als Argument an die Funktion `PrintTwice()` übergeben, wo er zwei mal ausgegeben wird.

Alternativ, hätten wir auch eine `char` Variable deklarieren können und diese als Argument benutzen können:

```
int main (void)
{
    char argument = 'b';
    PrintTwice (argument);
    return EXIT_SUCCESS;
}
```

WICHTIG: Wir sollten uns das Folgende gut einprägen: der Name der Variablen (`argument`) die wir als Argument an die Funktion übergeben, hat nichts mit dem Namen des Parameters der Funktion (`phil`) zu tun. Ich möchte das gleich noch einmal wiederholen:

Der Name der Variablen die wir als Argument übergeben hat nichts mit dem Namen des Parameters zu tun!

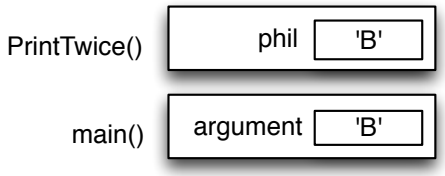
Die Namen können gleich sein, oder sie können sich unterscheiden. Wichtig ist, sie stellen unterschiedliche Objekte (Speicherstellen im Hauptspeicher des Rechners) dar und haben nur eine Gemeinsamkeit: sie besitzen den gleichen Wert (in unserem Fall das Zeichen `'b'`).

Die Werte die wir als Argumente übergeben, müssen den gleichen Typ haben wie die Parameter der aufgerufenen Funktion. Diese Regel ist wichtig, obwohl auch hier C manchmal Argumente automatisch von einen Typ in einen anderen umwandelt. Momentan ist es aber wichtig die generelle Regel zu kennen und sich später um die Ausnahmen zu kümmern.

3.10 Parameter und Variablen sind lokal gültig

Parameter und Variablen existieren nur innerhalb ihrer eigenen Funktionen in denen sie definiert wurden. Innerhalb von `main()` gibt es kein `phil`. Wenn wir dort versuchen `phil` zu verwenden wird sich der Kompiler beschweren. Gleichfalls gilt, innerhalb von `PrintTwice()` können wir nicht auf das Objekt `argument` zugreifen.

Variablen dieser Art nennt man **lokale Variablen**. Um die Übersicht über alle Funktionsparameter und lokalen Variablen zu behalten ist es nützlich ein so genanntes **Stack Diagramm** zu zeichnen. Stack Diagramme zeigen ähnlich wie Zustandsdiagramme den Wert jeder Variablen an. Der Unterschied besteht darin, dass in Stack Diagrammen die Variablen innerhalb größerer Kästchen gezeichnet werden. Diese Kästchen stellen die Funktionen dar, zu denen diese Variablen gehören. So sieht das Stack Diagram für `PrintTwice()` folgendermaßen aus:



Jedes mal, wenn eine Funktion aufgerufen wird, wird die Programmausführung in der aufrufenden Funktion unterbrochen und zur aufgerufenen Funktion gesprungen. Die aufgerufene Funktion können wir uns als eigenständiges Exemplar des Programmcodes der Funktion im Arbeitsspeicher des Computers vorstellen. Jede Funktion enthält ihre eigenen Parameter und lokalen Variablen und ist unabhängig von allen anderen Funktionen des Programms. Wird die aufgerufene Funktion beendet, kehrt das Programm zur aufrufenden Funktion zurück.

Im Diagramm wird jede aufgerufene Funktion durch einen Kasten mit dem Funktionsnamen an der Außenseite und den Variablen und Parametern im Inneren dargestellt.

In unserem Beispiel hat `main()` eine lokale Variable `argument`, und keine Parameter. `PrintTwice()` hat keine lokalen Variablen aber einen Parameter mit Namen `phil`.

Ein Stack Diagramm ist ein dynamische Diagramm. Im Programmablauf werden Funktionen aufgerufen und wieder beendet. Ein Stack Diagramm stellt also einen bestimmten Zeitpunkt in der Programmabarbeitung dar.

3.11 Funktionen mit mehreren Parametern

Die Syntax für die Deklaration und den Aufruf von Funktionen mit mehreren Parametern ist die Quelle vieler Programmierfehler. Zuerst müssen wir daran denken, dass wir bei der Definition eine Funktion den Typ jedes Parameters der Funktion mit angeben müssen. Zum Beispiel:

```
void PrintTime (int hour, int minute)
{
  printf ("%i", hour);
  printf (":");
  printf ("%i", minute);
}
```

Es erscheint verlockend hier einfach die zweite Typangabe wegzulassen und `(int hour, minute)` zu schreiben. Diese Schreibweise ist leider nur bei der Deklaration von Variablen erlaubt und nicht für Parameter.

Eine weiteres Missverständnis besteht darin, dass wir den Typ der Parameter angeben, den Typ der Argumente aber nicht angeben müssen, ja sogar einen Fehler machen, wenn wir den Typ der Argumente mit aufschreiben. Der folgende Code ist falsch!

```
int hour = 11;
int minute = 59;
PrintTime (int hour, int minute);   /* WRONG! */
```

Der Kompiler kennt den Typ von `hour` und `minute`. Wir haben ihn bei der Deklaration angegeben. Es ist daher nicht nötig und erlaubt den Typ mit anzugeben, wenn wir Variablen als Argumente einer Funktion verwenden. Die korrekte Syntax lautet: `PrintTime (hour, minute);`

3.12 Funktionen mit Ergebnissen

Es sollte mittlerweile aufgefallen sein, dass einige Funktionen die wir benutzen (wie zum Beispiel die mathematischen Funktionen) Ergebnisse liefern. Andere Funktionen wie `PrintNewLine()` führen bestimmte Aktionen durch, liefern aber kein Ergebnis an die aufrufende Funktion zurück. Dieses Verhalten lässt einigen Fragen offen:

- Was passiert, wenn wir eine Funktion aufrufen und danach nichts mit dem Resultat der Funktion machen (wir weisen es keiner Variablen zu und wir nutzen es auch nicht als Teil eines größeren Ausdrucks)?
- Was passiert, wenn wir eine Funktion die kein Resultat liefert, als Teil eines Ausdrucks verwenden, wie zum Beispiel `PrintNewLine() + 7` ?
- Können wir auch Funktionen schreiben, die uns Resultate liefern, oder müssen wir uns mit Funktionen vom Typ `PrintNewLine()` und `PrintTwice()` begnügen?

Die Antwort auf die dritte Frage ist “ja, wir können Funktionen schreiben welche ein Resultat zurückgeben,” und wir werden das in einigen Kapiteln auch tun. Ich überlasse es ihnen die Antworten auf die ersten beiden Fragen durch Probieren herauszufinden. Jedes mal, wenn die Frage auftaucht was in der Programmiersprache C erlaubt und möglich ist, ist es eine gute Ideen den C-Compiler zu fragen.

3.13 Glossar

Konstante (engl: *constant*): Eine bennante Speicherstelle, ähnlich einer Variable. Im Interschied zu Variaben können Konstanten nicht mehr verändert werden, nachdem ihr Wert festgelegt wurde:
z.B. `#define PI 3.141592`

Fließkomma (engl: *floating-point*): Der Typ einer Variable (oder eines Werts) welche reelle Zahlen speichern kann. Es existieren mehrere Fließkommatypen in C; in diesem Buch verwenden wir meistens `double`.

Initialisierung (engl: *initialization*): Eine Anweisung welche eine neue Variable definiert und gleichzeitig dieser Variable einen Wert zuweist.

Definition (engl: *definition*): Eine Deklaration (siehe 2.12) gibt nur bekannt, das eine Variable oder eine Funktion existiert (Funktionsprototyp). Eine Definition spezifiziert die Variable und Funktion und legt sie im Speicher an. Es darf dabei mehrere Deklarationen aber nur eine Definition geben.

Funktion (engl: *function*): Eine eigenständige Folge von Anweisungen die über einen Funktionsnamen aufgerufen werden kann. Funktionen können Parameters besitzen und einen Wert an die aufrufenden Funktion zurückgeben - müssen aber nicht.

Parameter (engl: *parameter*): Eine Variable in einer Funktion, deren Wert beim Funktionsaufruf durch die aufrufende Funktion bestimmt wird.
Der Name und Wert des Parameters ist nur innerhalb der Funktion gültig!

Argument (engl: *argument*): Der Ausdruck (Wert), mit dem die Funktion aufgerufen wird. Argumente müssen in Typ und Reihenfolge mit den Parametern der Funktion übereinstimmen.

Aufruf (engl: *call*): führt dazu, dass eine Funktion ausgeführt wird.

3.14 Übungsaufgaben

Übung 3.1

In dieser Übung sollen Sie das Lesen von Programmcode praktizieren. Sie sollen den Ablauf der Ausführung von Programmen mit mehreren Funktionen verstehen und nachvollziehen lernen.

a. Was gibt dieses Programm auf dem Bildschirm aus? Geben Sie präzise an wo sich Leerzeichen und Zeilenumbrüche befinden.

   HINWEIS: Beginnen Sie mit einer verbalen Beschreibung dessen was die Funktionen `Ping` und `Baffle` tun, wenn sie aufgerufen werden.

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

  void Ping (void)
  {
    printf (".\n");
  }

  void Baffle (void)
  {
    printf ("wug");
    Ping ();
  }

 void Zoop (void)
 {
    Baffle ();
    printf ("You wugga ");
    Baffle ();
  }

  int main (void)
  {
    printf ("No, I ");
    Zoop ();
    printf ("I ");
    Baffle ();
    return EXIT_SUCCESS;
  }
```

b. Zeichnen Sie ein Stackdiagram welches den Status des Programms wiedergibt wenn `Ping` zum ersten Mal aufgerufen wird.

Übung 3.2

In dieser Übung lernen Sie wie man Funktionen mit Parametern schreibt und aufruft.

a. Schreiben Sie die erste Zeile einer Funktion mit dem Namen `Zool`. Die Funktion hat drei Parameter: ein `int` und zwei `char`.

b. Schreiben Sie eine Code-Zeile in der Sie `Zool` aufrufen und die folgenden Werte als Argumente übergeben: `11`, den Buchstaben `a`, und den Buchstaben `z`.

Übung 3.3

In dieser Übung werden wir ein Programm aus einer vorigen Übung anpassen und verändern, so dass eine Funktion mit Parametern zum Einsatz kommt. Starten mit einer funktionsfähigen Programmversion.

a. Schreiben Sie eine Funktion mit dem Namen `PrintDateAmerican` diese hat die folgenden Parameter day, month und year und gibt das Datum im amerikanischen Standardformat aus.

b. Testen Sie die Funktion indem Sie diese aus `main` heraus aufrufen und die entsprechenden Parameter als Argumente übergeben. Das Ergebnis sollte folgendem Muster entsprechen:

```
3/29/2009
```

c. Nachdem Sie die Funktion `PrintDateAmerican` erfolgreich erstellt und ausgeführt haben, schreiben Sie eine weitere Funktion `PrintDateEuropean` welche das Datum im europäischen Format ausgibt.

Übung 3.4

Viele Berechnungen lassen sich übersichtlich als "multadd" Operation ausführen, dazu wird mit drei Operanden folgende Berechnung durchgeführt `a*b + c`. Einige Prozessoren bieten für diesen Befehl sogar eine Hardwareimplementierung für Gleitkommazahlen.

a. Erstellen Sie ein neues Programm mit dem Namen `Multadd.c`.

b. Schreiben Sie eine Funktion `Multadd` welche drei `doubles` als Parameter besitzt und welche das Ergebnis der Multaddition ausgibt.

c. Schreiben Sie eine `main` Funktion welche `Multadd` durch den Aufruf mit einigen einfachen Parametern testet und das Ergebnis ausgibt. So sollte zum Beispiel für die Parameter `1.0, 2.0, 3.0` als Ergebnis `5.0` ausgegeben werden.

d. Benutzen Sie `Multadd` in der `main` Funktion um den folgenden Wert zu berechnen:

$$\sin \frac{\pi}{4} + \frac{\cos \frac{\pi}{4}}{2}$$

e. Schreiben Sie eine Funktion `Yikes` welche ein `double` als Parameter übernimmt und `Multadd` für die Berechnung und Ausgabe benutzt:

$$xe^{-x} + \sqrt{1 - e^{-x}}$$

HINWEIS: Die mathematische Funktion für die Berechnung von e^x lautet `double exp(double x);`.

In der letzten Aufgabe sollen Sie eine Funktion schreiben, welche ihrerseits eine selbst erstellte Funktion aufruft. Dabei sollten Sie stets daran denken die erste Funktion ausgiebig zu testen bevor Sie mit der Arbeit an der zweiten Funktion beginnen. Ansonsten kann es vorkommen, dass Sie gleichzeitig zwei Methoden debuggen müssen - ein sehr mühsames Unterfangen.

Ein weiteres Ziel dieser Übung ist es ein spezielles Problem als Teil einer allgemeineren Klasse von Problemen zu erkennen. Wenn immer möglich sollten Sie versuchen Programme zu entwickeln, die allgemeine Probleme lösen.

Kapitel 4

Abhängigkeiten und Rekursion

4.1 Bedingte Abarbeitung

Unsere bisherigen Programme hatten eine Gemeinsamkeit. Es wurden alle Befehle in strenger Reihenfolge nacheinander abgearbeitet. Solche Programme haben die Eigenschaft, dass sie jedes mal genau die gleichen Aktionen durchführen und die gleichen Ergebnisse produzieren.

Für die meisten realen Anwendungen brauchen wir aber auch die Möglichkeit das Vorliegen bestimmter Bedingungen überprüfen zu können und die Abarbeitungsreihenfolge der Befehle (das Verhalten unseres Programms) an die jeweiligen äußeren und inneren Bedingungen anpassen zu können.

Bedingte Anweisungen geben uns diese Möglichkeit. Die einfachste Form ist dabei die `if`-Anweisung:

```
if (x > 0)
{
    printf ("x ist positiv\n");
}
```

Der angegebene Ausdruck in Klammern ist die Bedingung. Wenn er *wahr* ist, werden die Anweisungen zwischen den den geschweiften Klammern ausgeführt. Die öffnende und schließende geschweifte Klammer bildet einen so genannten **Anweisungsblock** oder einfach **Block**. Was das genau ist erkläre ich im nächsten Abschnitt genauer. Einfach gesagt handelt es sich dabei um eine Gruppierung von Anweisungen.

Wenn die Bedingung *falsch* ist, wird das Programm mit dem nächsten Befehl nach dem Block fortgesetzt. In unserem Beispiel passiert gar nichts.

Die Bedingung kann die folgenden **Vergleichsoperatoren** enthalten:

```
x == y              /* x ist gleich y */
x != y              /* x ist nicht gleich (ungleich) y */
x > y               /* x ist größer als y */
x < y               /* x ist kleiner als y */
x >= y              /* x ist größer als oder gleich y */
x <= y              /* x ist kleiner als oder gleich y */
```

Diese Operationen sollten aus dem Bereich der Mathematik bereits bekannt sein. Allerdings benutzt C eine Syntax die von den gebräuchlichen mathematischen Symbolen abweicht, wo wir $=$, $\neq$, $\geq$ und $\leq$ verwenden. Ein oft gemachter Fehler ist die Verwendung eines einzelnen `=` anstelle des doppelten `==` für den Ausdruck der Gleichheit zweier Operanden. Erinnern wir uns, `=` ist der Zuweisungsoperator und `==` ist ein Vergleichsoperator. Bitte auch beachten: `=<` or `=>` sind keine gültigen Operatoren.

WICHTIG: Die Ausdrücke auf den beiden Seiten des Vergleichsoperators sollten vom gleichen Typ sein. Wir können zwar `int`s mit `double`s vergleichen, es wird eine automatische Typumwandlung durchgeführt. Man sollte aber darauf achten, nicht auf Gleichheit zu testen (`==`) sobald ein Operator als Fließkommazahl dargestellt wird. Das gilt auch, wenn `double`s mit `double`s verglichen werden.

Unglücklicherweise, kennen wir derzeit noch keine Methode um Strings (Zeichenketten) zu vergleichen! Es gibt einen Weg das zu tun, aber wir müssen leider noch einige Kapitel darauf warten.

4.2 Anweisungsblöcke

Die Anweisung stellt einen einzelnen Befehl oder Abarbeitungsschritt in einem Programm dar. Anstelle einer einfachen Anweisung kann in einem C Programm aber immer auch eine zusammengesetzte Anweisung, ein so genannter **Block**, geschrieben werden.

Einen Anweisungsblock wird mit Hilfe der geschweiften Klammern `{` und `}` gebildet. C behandelt einen Anweisungsblock so, als wäre es eine einzelne Anweisung. Das ist besonders wichtig bei der bedingten und wiederholten Abarbeitung von Programmteilen. Wir können damit mehrere Anweisungen zusammenfassen und diese in Abhängigkeit von der zu überprüfenden Bedingungen ausführen lassen. Im folgenden Beispiel werden alle Anweisungen zwischen den geschweiften Klammern ausgeführt, wenn die Variable `x` größer als Null ist. Wenn `x` kleiner oder gleich Null ist, wird der komplette Block nicht ausgeführt:

```
if (x > 0)
{
    printf ("x hat den Wert %i\n", x);
    printf ("x ist positiv\n");
}
```

Einzelne Anweisungen werden mit einen Semikolon (`;`) abgeschlossen. Eine Blockanweisung wird durch schließende Klammer `}` beendet. Wir müssen daher am Ende eines Blocks kein zusätzliches Semikolon anfügen.

4.3 Der Modulo-Operator

Der Modulo-Operator kann auf ganzzahlige Werte (und ganzzahlige Ausdrücke) angewendete werden und liefert uns den *ganzzahligen Divisionsrest* wenn wir den ersten Operanden durch den zweiten Operanden teilen. In C wird der Modulo-Operator durch das Prozentzeichen `%` dargestellt. Die Syntax ist genau die gleiche wie bei anderen mathematischen Operatoren mit zwei Operanden:

```
int quotient = 11 / 4;
int rest     = 11 % 4;
```

Das Ergebnis der ersten Berechnung (Ganzzahldivision!) ist 2. Die zweite Berechnung liefert das Ergebnis 3, denn 11 geteilt durch 4 ergibt 2 mit dem Rest 3.

Die Modulorechnung kann erstaunlich nützlich sein. So kann man zum Beispiel auf einfache Weise überprüfen, ob eine Zahl durch eine andere Zahl teilbar ist. Wenn `x % y` den Rest Null liefert, dann ist `x` durch `y` teilbar.

Weiterhin können wir die Modulorechnung dafür verwenden um die rechts stehenden Ziffern einer Zahl zu extrahieren. So liefert uns die Operation `x % 10` die Einerstelle des in der Variable x gespeicherten Werts im Dezimalsystem. Mit der Operation `x % 100` lassen sich die letzten zwei Ziffern extrahieren.

4.4 Alternative Ausführung

Eine zweite Form der bedingten Ausführung ist die alternative Ausführung bei der zwei Möglichkeiten existieren und die Bedingung festlegt, welche davon zur Ausführung kommt. Die Syntax sieht folgendermaßen aus:

```
if (x%2 == 0)
{
    printf ("x ist gerade\n");
}
else
{
    printf ("x ist ungerade\n");
}
```

Wenn das Ergebnis der Division von `x` durch 2 Null ergibt, dann wissen wir das `x` gerade ist. Unser Programm gibt dann die Nachricht `x ist gerade` auf dem Bildschirm aus.

Wenn die Bedingung falsch ist, wird der zweite Anweisungsblock ausgeführt. Da die Bedingung nur wahr oder falsch sein kann, wird genau eine der beiden Alternativen ausgeführt. Es ist nicht notwendig im zweiten Anweisungsblock die Bedingung erneut zu überprüfen.

So nebenbei, wenn wir den Eindruck haben, dass wir in unserem Programm die Parität (Geradzahligkeit, Ungeradzahligkeit) von Zahlen öfters überprüfen müssen, dann kann es sinnvoll sein diesen Code in eine Funktion einzubetten:

```
void PrintParity (int x)
{
    if (x%2 == 0)
    {
        printf ("x ist gerade\n");
    }
    else
    {
        printf ("x ist ungerade\n");
    }
}
```

Wir haben jetzt eine Funktion `PrintParity()` die uns auf dem Bildschirm anzeigt, ob ein übergebener Integerwert gerade oder ungerade ist. Wir können die Funktion in `main()` wie folgt aufrufen:

```
PrintParity (17);
```

Wir müssen immer daran denken, dass, wenn wir eine Funktion *aufrufen* der Typ der Argumente nicht mit angeben wird. C kennt den Typ der verwendeten Variablen und Konstanten. Wir machen einen Fehler, wenn wir die Funktion in der folgenden Weise aufrufen:

```
int number = 17;
PrintParity (int number);         /* WRONG!!! */
```

4.5 Mehrfache Verzweigung

Manchmal kommt es vor, dass wir eine ganze Reihe von zusammengehörigen Bedingungen überprüfen und eine Auswahl aus mehreren möglichen Aktionen treffen wollen.

Es gibt verschiedene Möglichkeiten das zu erreichen. Eine davon ist die **Verkettung** einer Serie von `if`s und `else`s:

```
if (x > 0)
{
    printf ("x is positive\n");
}
else if (x < 0)
{
    printf ("x is negative\n");
}
else
{
    printf ("x is zero\n");
}
```

Diese Kette kann so lang werden wie wir wollen, allerdings wird es immer schwerer die Übersicht zu behalten, je länger sie wird. Eine Möglichkeit die Lesbarkeit zu erhöhen, besteht darin, mit Formatierungen durch Einrückungen zu arbeiten.

Wenn wir alle Anweisungen und geschweiften Klammern an der gleichen Stelle untereinander schreiben, kommt es viel seltener vor, dass wir einen Syntaxfehler machen und falls es doch passiert, wir würden wir ihn viel schneller finden.

4.6 Verschachtelte Abhängigkeiten

Zusätzlich zur Verkettung können wir Abhängigkeiten auch ineinander verschachteln. Wir hätten das vorige Beispiel auch in dieser Form schreiben können:

```
if (x == 0)
{
    printf ("x is zero\n");
}
else
{
    if (x > 0)
    {
        printf ("x is positive\n");
    }
    else
    {
        printf ("x is negative\n");
    }
}
```

Es gibt jetzt eine äußere Bedingung die eine Verzweigungen erzeugt. Der erste Zweig enthält eine einfache Ausgabeanweisung. Der zweite Zweig enthält eine weitere `if`-Anweisung, welche wiederum eine Verzweigung erzeugt. Glücklicherweise sind deren Zweige Ausgabeanweisungen, es hätten aber durchaus auch weitere bedingte Anweisungen folgen können.

Auch hier ist es wichtig, dass wir die Struktur des Programms durch Einrückungen sichtbar machen. Allerdings ist es prinzipiell so, dass ab einer bestimmten Verzweigungstiefe der Überblick verloren geht. Man sollte sich deshalb gut überlegen, ob die tiefe Verschachtelung von bedingten Anweisungen im eigenen Programm unbedingt nötig ist. Auf der anderen Seite finden wir ineinander **verschachtelte Strukturen** in einer Reihe von Programmen wieder, so dass es eine gute Idee ist, sich an dieses Programmierkonzept zu gewöhnen.

4.7 Die `return`-Anweisung

Die `return` Anweisung erlaubt es uns die Ausführung einer Funktion zu beenden, ohne dass alle Anweisungen der Funktion bis zum Ende ausgeführt werden müssen. Der Einsatz der `return`-Anweisung ist zum Beispiel dann sinnvoll, wenn unser Programm eine Fehlerbedingung erkannt hat:

```
#include <math.h>
```

```
void PrintLogarithm (double x)
{
    if (x <= 0.0)
    {
        printf ("Positive numbers only, please.\n");
        return;
    }

   double result = log (x);
   printf ("The log of x is %f\n", result);
}
```

Wir definieren eine Funktion namens `PrintLogarithm()` mit dem Parameter `x` vom Typ `double`. Die Funktion überprüft zu allererst, ob `x` kleiner oder gleich Null ist. In diesem Falle würde die Funktion eine Fehlermeldung anzeigen und die Funktion mittels `return` beenden. Die Abarbeitung des Programms wird in der aufrufenden Funktion fortgesetzt. Die übrigen Programmzeilen der Funktion werden in diesem Fall nicht ausgeführt.

Ich habe einen Fließkommawert auf der rechten Seite der Bedingung benutzt, weil die Variable auf der linken Seite vom Typ `double` ist.

WICHTIG: Jedes mal wenn wir eine Funktion aus der mathematischen Bibliothek benutzen, müssen wir die Headerdatei `math.h` in unser Programm einbinden.

4.8 Rekursion

Ich habe im letzten Kapitel erwähnt, dass es völlig legal ist, dass eine Funktion eine andere Funktion aufzurufen kann und wir haben bereits einige Beispiele für dieses Verhalten gesehen. Ich habe nicht erwähnt, dass eine Funktion sich auch selbst aufrufen kann – das möchte ich jetzt nachholen.

In der Tat ist es möglich und erlaubt, dass eine Funktion sich selbst aufruft. Es mag nicht offensichtlich sein, wofür das gut sein sollte, aber dabei handelt es sich um eines der merkwürdigsten und interessantesten Verhalten die ein Programm besitzen kann.

Schauen wir uns die folgende Funktion an:

```
void Countdown (int n)
{
    if (n == 0)
    {
        printf ("Start!");
    }
    else
    {
        printf ("%i", n);
```

```
        Countdown (n-1);
    }
}
```

Der Name der Funktion ist `Countdown()` und sie besitzt einen einzelnen Parameter vom Typ `int`. Wenn der Parameter den Wert Null hat, wird das Wort "Start!" ausgegeben. Anderenfalls wird der Wert des Parameters auf dem Bildschirm ausgegeben und eine Funktion mit dem Namen `Countdown()` –die gleiche Funktion– aufgerufen und `n-1` als Argument übergeben.

Was passiert wenn wir diese Funktion in der folgenden Weise aufrufen?

```
int main (void)
{
     Countdown (3);
     return EXIT_SUCCESS;
}
```

Die Ausführung von `Countdown()` beginnt mit `n=3` und weil `n` nicht Null ist, wird der Wert 3 ausgegeben und dann ruft die Funktion `Countdown()` erneut auf...

> Die Ausführung von `Countdown()` beginnt mit `n=2` und weil `n` nicht Null ist, wird der Wert 2 ausgegeben und dann ruft die Funktion `Countdown()` erneut auf...
>
> > Die Ausführung von `Countdown()` beginnt mit `n=1` und weil `n` nicht Null ist, wird der Wert 1 ausgegeben und dann ruft die Funktion `Countdown()` erneut auf...
> >
> > > Die Ausführung von `Countdown()` beginnt mit `n=0` und weil `n` jetzt den Wert Null hat, wird das Wort "Start!" ausgegeben. Danach ist die Funktion beendet und die Abarbeitung des Programms wird in der aufrufenden Funktion fortgesetzt.
> >
> > Die Funktion `Countdown()`, welche `n=1` als Argument erhalten hat, ist beendet und die Abarbeitung des Programms wird in der aufrufenden Funktion fortgesetzt.
>
> Die Funktion `Countdown()`, welche `n=2` als Argument erhalten hat, ist beendet und die Abarbeitung des Programms wird in der aufrufenden Funktion fortgesetzt.

Die Funktion `Countdown()`, welche `n=3` als Argument erhalten hat, ist beendet und die Abarbeitung des Programms wird in der aufrufenden Funktion fortgesetzt.

Und dann ist unser Programm zurück in `main()` (was für eine Reise). Die gesamte Ausgabe sieht folgendermaßen aus:

```
3
2
1
Start!
```

In der Umgangssprache der Programmierer sagt man *eine Funktion ruf sich selbst auf*. Viele Programmieranfänger (und nicht nur die) haben Probleme damit sich das vorstellen und erklären zu können.

Erinnern wir uns. Wir haben gelernt, dass, wenn eine Funktion aufgerufen wird, eine neue Instanz des Programmcodes erzeugt wird. Eine Instanz ist also eine Kopie des Programmcodes. Unsere `Countdown()` Funktion erzeugt mehrere Kopien des gleichen Programmcodes. Allerdings sind diese Instanzen nicht komplett gleich, sondern unterscheiden sich in ihren Parameterwerten. Genau genommen ist es also falsch und irreführend zu sagen, dass eine Funktion sich selbst aufruft. Sie ruft nur den gleichen Code auf.

Schauen wir uns als zweites Beispiel erneut die Funktionen `PrintNewLine()` und `PrintThreeLines()` an:

```
void PrintNewLine ()
{
    printf ("\n");
}

void PrintThreeLines ()
{
    PrintNewLine ();  PrintNewLine ();  PrintNewLine ();
}
```

Wir können die Funktionen mit unserem neuen Wissen verbessern, so dass wir so viele Zeilen ausgeben lassen können wie wir wollen, seien es 2 oder 106:

```
void PrintLines (int n)
{
    if (n > 0)
    {
        printf ("\n");
        PrintLines (n-1);
    }
}
```

Das Programm ist ähnlich wie `Countdown()` aufgebaut. So lange `n` größer als Null ist wird eine Leerzeile ausgegeben und dann wird die gleiche Funktion aufgerufen um weitere `n-1` Leerzeilen auszugeben. Somit ergibt sich die Gesamtzahl der auszugebenden Leerzeilen aus `1 + (n-1)`, was näherungsweise `n` entspricht.

Der Vorgang einer sich selbst aufrufenden Funktion wird als **Rekursion** bezeichnet – Funktionen welche diese Eigenschaft aufweisen sind **rekursive** Funktionen.

4.9 Unendliche Rekursion

In unseren Beispielen im letzten Abschnitt fällt auf, dass bei jedem rekursiven Funktionsaufruf das Funktionsargument um den Wert 1 verringert wird, bis am

Ende die Funktion mit dem Wert Null aufgerufen wird. In diesem Fall wird die Funktion beendet, ohne einen weiteren Funktionsaufruf durchzuführen. Dieser Fall –wenn die Funktion beendet wird ohne einen weiteren rekursiven Funktionsaufruf zu machen– nennt man die **Abbruchbedingung**.

Wenn eine Rekursion niemals die Abbruchbedingung erreicht, würde sich die rekursive Funktion (theoretisch) unendlich oft selbst aufrufen und das Programm würde nie beendet. Diesen Fall bezeichnet man auch als **unendliche Rekursion** und ist generell keine gute Idee. Dieser Fehler kann auch auftreten wenn eine Abbruchbedingung zwar vorhanden ist, aber während der Abarbeitung des Programms nicht erreicht werden kann.

In den meisten Fällen wird ein Programm mit unendlich rekursiven Funktionen nicht wirklich für immer laufen. Die Ressourcen eines Computers sind endlich und früher oder später wird unser Progamm mit einer Fehlermeldung beendet werden. Einen Programmfehler dieser Art bezeichnet man als *Run-time error* oder *Laufzeitfehler* (ein Fehler der erst in Erscheinung tritt, wenn das Programm ausgeführt wird – zur 'Laufzeit' eines Programms) Die Nichtexistenz einer Abbruchbedingung ist ein Beispiel für diese Art von Fehlern.

AUFGABE 1: Schreiben Sie ein kleines Programm, welches immer wieder die gleiche Funktion aufruft, ohne abzubrechen und beobachten Sie das Verhalten dieses Programms.

AUFGABE 2: Was passiert wenn wir die Funktion `Countdown()` mit dem Argument -1 aufrufen? Wie können wir diesen Fehler vermeiden?

4.10 Stack Diagramme für rekursive Funktionen

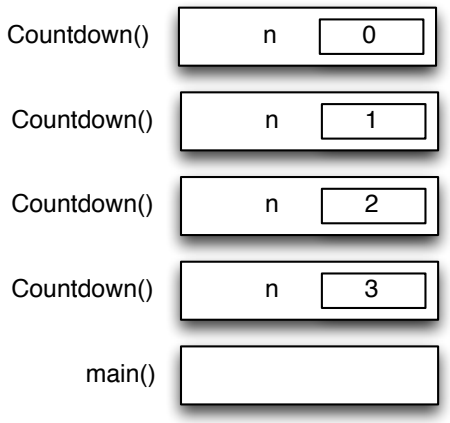
Im vorigen Kapitel haben wir ein Stackdiagramm verwendet um den Zustand eines Programms und seiner Funktionen während eines Funktionsaufrufs darzustellen.

Wir können ein Stackdiagramm auch dazu verwenden um uns noch einmal verständlich zu machen, was während eines rekursiven Funktionsaufrufs in unserem Programm vorgeht.

Es ist für das Verständnis wichtig, dass wir uns erinnern, dass jedes Mal wenn eine Funktion aufgerufen wird eine neue Instanz des Programmcodes und der lokalen Variablen und Parameter dieser Funktion erzeugt wird.

Die Abbildung zeigt und das Stackdiagramm für die Funktion `Countdown()`, wenn sie in der Hauptfunktion `main()`mit dem Argument `n = 3` wie folgt aufgerufen wird:

```
Countdown(3);
```



Es existiert genau eine Instanz von `main()` und vier Instanzen von `Countdown`, jede mit einem unterschiedlichen Parameterwert für `n`. Das unterste Element des Stapels, `Countdown()` mit `n=0` ist die Abbruchbedingung. An dieser Stelle wird kein weiterer rekursiver Funktionsaufruf durchgeführt, so dass keine weiteren Instanzen von `Countdown()` erzeugt werden. Die Instanz von `main()` ist leer, weil in `main()` keine Parameter oder lokalen Variablen definiert sind.

Versuche Sie doch einmal ein Stackdiagramm für `PrintLines()` zu zeichnen, wenn es mit dem Parameter `n=4` aufgerufen wird!

4.11 Glossar

Anweisungsblock (engl: *block*): Eine Gruppe von Anweisungen die durch eine öffnende und schließende geschweifte Klammer gebildet wird. Die Anweisungen werden gemeinsam ausgeführt und vom Compiler wie eine einzige Anweisung behandelt.

Modulo (engl: *modulus*): Der Modulo-Operator berechnet den ganzzahligen Divisionsrest bei einer Division von ganzen Zahlen. In C wird der Operator durch das Prozentzeichen dargestellt (`%`).

Bedingte Anweisung (engl: *conditional*): Ein Anweisungsblock der nur dann ausgeführt wird, wenn eine bestimmte Bedingung erfüllt ist.

Mehrfache Verzweigung (engl: *chaining*): Durch die verbundene Abfrage mehrer Bedingungen lassen sich Fallunterscheidungen durchführen. Es wird nur ein Programmabschnitt von mehreren Alternativen ausgeführt.

Verschachtelte Abhängigkeiten (engl: *nesting*): werden erzeugt in dem man eine weitere Fallunterscheidung in einem Zweig einer bedingten Anweisung durchführt.

Rekursion (engl: *recursion*): Wird aus einer Funktion heraus die gleiche Funktion erneut aufgerufen bezeichnet man dies als Rekursion. Der Aufruf erfolgt üblicherweise mit geänderten Argumenten und sollte zu einem definierten Ende führen. Anderenfalls droht die

Unendliche Rekursion (engl: *infinite recursion*): Eine Funktion ruft sich wiederholt selbst auf, ohne die Abbruchbedingung zu erreichen. Diese Verhalten belegt den kompletten Stack-Speicher und führt zu einem Laufzeitfehler.

4.12 Übungsaufgaben

Übung 4.1

Der erste Vers des Lieds "99 Bottles of Beer" lautet:

> 99 bottles of beer on the wall, 99 bottles of beer, ya' take one down, ya' pass it around, 98 bottles of beer on the wall.

Die nachfolgenden Verse sind identisch bis auf die Anzahl der Flaschen. Deren Anzahl nimmt in jedem Vers um eine Flasche ab, bis schließlich der letzte Vers lautet:

> No bottles of beer on the wall, no bottles of beer, ya' can't take one down, ya' can't pass it around, 'cause there are no more bottles of beer on the wall!

Und dann ist diese Lied schließlich zu Ende.

Schreiben Sie ein Programm, welches den gesamten Text des Lieds "99 Bottles of Beer" ausgibt. Ihr Programm sollte eine rekursive Funktion für die Ausgabe des Liedtextes verwenden. Sie können weitere Funktionen verwenden um ihr Programm zu strukturieren.

Während Sie den Programmcode schreiben und testen sollten Sie mit einer kleineren Anzahl von Versen beginnen, z.B. "3 Bottles of Beer."

Der Sinn dieser Übung besteht darin ein Problem zu analysieren und in kleinere, lösbare Bestandteile zu zerlegen. Diese kleineren Einheiten lassen sich unabhängig und nacheinander entwickeln und testen und führen im Ergebnis zu einer schnelleren und robusteren Lösung.

Übung 4.2

In C können Sie die `getchar()` Funktion benutzen um Zeichen von der Tastatur einzulesen. Diese Funktion stoppt die Ausführung des Programms und wartet auf eine Eingabe des Benutzers. Die `getchar()` Funktion ist vom Typ `int` und erfordert kein Argument. Sie liefert den ASCII-Code des eingegeben Zeichens von der Tastatur zurück.

Schreiben Sie ein Programm, welches den Benutzer auffordert eine Ziffer von 0-9 einzugeben.

Überprüfen Sie die Eingabe des Benutzers und geben Sie einen Hinweis aus, falls es sich bei dem eingegeben Wert nicht um eine Zahl handeln sollte. Geben Sie nach erfolgreicher Prüfung die Zahl aus.

Übung 4.3

Fermat's "Letzter Satz"besagt, dass es keine ganzen Zahlen a, b und c gibt, für die gilt

$$a^n + b^n = c^n \tag{4.1}$$

außer für den Fall, dass $n = 2$.

Schreiben Sie eine Funktion mit dem Namen `CheckFermat()` welche vier `int` als Parameter hat —`a`, `b`, `c` and `n`— und welche überprüft, ob Fermats Satz Bestand hat. Sollte sich für n größer als 2 herausstellen, dass $a^n + b^n = c^n$, dann sollte ihr Programm ausgeben: "Holy smokes, Fermat was wrong!" In allen anderen Fällen sollte das Programm ausgeben: "No, that doesn't work."

Verwenden Sie für die Berechnung der Potenzen die Funktion `pow()` aus der mathematischen Bibliothek. Diese Funktion übernimmt zwei `double` als Argument. Das erste Argument stellt dabei die Basis und das zweite Argument den Exponenten der Potenz dar. Die Funktion liefert als Ergebnis wiederum ein `double`.

Um die Funktion in unserem Programm nutzen zu können, müssen die Datentypen angepasst werden (siehe Abschnitt 3.3). Dabei wandelt C den Datentyp `int` automatisch in `double` um. Um einen `double` Wert in `int` zu wandeln, muss der Typecast-Operator `(int)` verwendet werden.

Zum Beispiel:

```
int x = (int) pow(2, 3);
```

weist `x` den Wert `8` zu, weil $2^3 = 8$.

Kapitel 5

Funktionen mit Ergebnissen

5.1 Return-Werte

Wir haben ja bereits einige Erfahrung bei der Verwendung von Funktionen in C. Bei einigen der Standardfunktionen, wie zum Beispiel den mathematischen Funktionen ist uns aufgefallen, das die Funktion einen Wert berechnet – die Funktion hat ein Resultat produziert.

Mit diesem Wert kann unser Programm weiterhin arbeiten. Wir können ihn in einer Variable speichern, auf dem Bildschirm ausgeben oder als Teil eines Ausdrucks verwenden. Zum Beispiel

```
double e = exp (1.0);
double height = radius * sin (angle);
```

Allerdings waren alle unsere Funktionen, die wir bisher selbst geschrieben haben **void** Funktionen, das heißt sie haben den Datentyp **void** und liefern kein Ergebnis zurück.

Wenn wir **void** Funktionen aufrufen, steht der Funktionsaufruf üblicherweise für sich allein als Anweisung in einer Programmierzeile ohne dass dabei ein Wert zugewiesen oder erwartet wird:

```
PrintLines (3);
Countdown (n-1);
```

In diesem Kapitel werden wir herausfinden, wie wir Funktionen schreiben, welche eine Rückgabe an die aufrufende Funktion erzeugt. Weil mir ein guter Name dafür fehlt werde ich sie **Funktionen mit Ergebnissen** nennen.

Das erste Beispiel ist die Funktion `CalculateCircleArea()`. Sie hat einen `double` Wert als Parameter und liefert die berechnete Fläche eines Kreises in Abhängigkeit vom gegebenen Radius:

```
double CalculateCircleArea (double radius)
{
```

```
        double pi = acos (-1.0);
        double area = pi * radius * radius;
        return area;
    }
```

Beim Betrachten der Funktionsdefinition fällt als erstes auf, dass die Funktion anders beginnt, als alle andern Funktionen, die wir bisher geschrieben haben. Der erste Begriff in einer Funktionsdefinition gibt den Datentyp der Funktion an. Anstelle von `void` steht hier `double`. Damit wird angezeigt, dass der Rückgabewert der Funktion vom Typ `double` ist.

Immer, wenn wir mit Daten arbeiten, neue Werte berechnen, Funktionen aufrufen, oder Ein- undAusgaben erzeugen, müssen wir exakt angeben um welchen Datentyp es sich dabei handelt. Nur so kann der Compiler prüfen, ob die tatsächlich verwendeten Daten dem richtigen Typ entsprechen und uns vor größeren Problemen bewahren. Das erscheint am Anfang vielleicht etwas ungewohnt und lästig, wird uns aber bald ganz selbstverständlich von der Hand gehen.

Wenn wir uns die letzte Zeile anschauen, dann fällt auf, dass die `return`-Anweisung jetzt auch einen Wert enthält. Die Bedeutung dieser Anweisung ist die folgende: "kehre unmittelbar zur aufrufenden Funktion zurück und verwende den Wert des folgenden Ausdrucks als Rückgabewert." Der angegebene Ausdruck kann dabei von beliebiger Komplexität sein. Wir hätten also die Funktion auch sehr viel knapper zusammenfassen können:

```
    double Area (double radius)
    {
        return acos(-1.0) * radius * radius;
    }
```

Auf der anderen Seite erleichtert uns die Verwendung von **temporäre** Variablen wie `area` die Suche nach Programmfehlern. Wichtig ist in jedem Fall, dass der Typ des Ausdrucks in der `return`-Anweisung mit dem angegebenen Typ der Funktion übereinstimmt.

In anderen Worten, wenn wir in einer Funktionsdeklaration angeben das der Rückgabewert vom Typ `double` ist, geben wir ein Versprechen die Funktion schließlich ein Ergebnis vom Typ `double` produziert. Wenn wir keinen Wert zurückgeben (`return` ohne Ausdruck benutzen) oder den falschen Typ zurückgeben ist das fast immer ein Fehler und der Compiler wird uns dafür zur Rede stellen. Allerdings gelten auch hier die Regeln der automatischen Typumwandlung.

Manchmal ist es nützlich mehrere `return`-Anweisung in einer Funktion zu haben. Zum Beispiel eine für jede Programmverzweigung:

```
  double AbsoluteValue (double x)
    {
        if (x < 0)
        {
            return -x;
        }
```

```
    else
    {
        return x;
    }
}
```

Da sich die `return` -Anweisungen in alternativen Zweigen unseres Programms befinden wird jeweils nur eine von ihnen auch ausgeführt. Obwohl es legal ist mehrere `return`-Anweisungen in einer Funktion zu haben, ist es wichtig daran zu denken, dass eine davon ausgeführt wird, die Funktion beendet ist, ohne noch irgendwelche anderen Anweisungen auszuführen.

Programmcode, welcher hinter einer `return`-Anweisung steht, wird nicht mehr ausgeführt und wird deshalb **unereichbarer Code** genannt. Sollte eine Funktion also nicht das erwartete Ergebnis produzieren, so sollten Sie prüfen ob die Anweisungen auch wirklich ausgeführt werden. Manche Compilers geben eine Warnung aus, wenn in einem Programm solche Codezeilen existieren.

Wenn wir `return`-Anweisungen in Programmverzweigungen benutzen, müssen wir garantieren, das *jeder mögliche Pfad* durch das Programm auf eine `return`-Anweisung trifft. Zum Beispiel gibt es ein Problem im folgenden Programm:

```
double AbsoluteValue (double x)
  {
    if (x < 0)
    {
        return -x;
    }
    else if (x > 0)
    {
        return x;
    }                 /* Fehlendes return für x==0!! */
}
```

Dieses Programm ist nicht korrekt, weil im Fall, dass `x` den Wert 0 hat, keine der beiden Bedingungen zutrifft und die Funktion beendet wird, ohne auf eine `return`-Anweisung zu treffen. Unglücklicherweise können viele C Compilers diesen Fehler nicht finden. Es ist also häufig der Fall, dass sich das Programm kompilieren und ausführen lässt, aber der Rückgabewert für den Fall `x==0` nicht definiert ist. Wir können nicht voraussagen, welcher Wert letztendlich zurückgegeben wird und es ist wahrscheinlich, dass es unterschiedliche Werte für unterschiedliche Umgebungen sein werden.

Mittlerweile haben Sie bestimmt schon die Nase voll davon Compiler-Fehler zu sehen. Allerdings kann ich versichern, dass es nur eine Sache gibt die schlimmer ist als Compiler-Fehler zu erhalten – und das ist *keine* Compiler-Fehler zu erhalten, wenn das Programm falsch ist.

Ich beschreibe mal kurz was wahrscheinlich passieren wird: Sie testen `AbsoluteValue()` mit mehreren verschiedenen Werten für `x` und die Funktion scheint korrekt zu arbeiten. Dann geben Sie ihr Programm an jemand anderen weiter und er oder sie versucht es in einer geänderten Umgebung (anderer

Compiler oder Rechnerarchitektur) laufen zu lassen. Das Programm produziert plötzlich auf mysteriöse Art und Weise Fehler.
Es wird mehrere Tage und viel Debugging-Aufwand kosten herauszufinden, dass die Implementierung von `AbsoluteValue()` fehlerhaft war - wie froh wären Sie gewesen, wenn Sie der Compiler doch nur gewarnt hätte!

Von jetzt an sollten wir nicht dem Compiler die Schuld geben, wenn er wieder auf einen Fehler in unserem Programm hinweist. Im Gegenteil, wir sollten ihm danken, dass er einen Fehler so einfach gefunden hat und uns viel Zeit und Aufwand erspart hat den Fehler selbst aufspüren zu müssen. Die meisten Compilers verfügen über eine Option mit der wir dem Compiler mitteilen können unser Programm besonders strikt und sorgfältig zu prüfen und alle Fehler zu melden die er nur finden kann. Sie sollten diese Option während der gesamten weiteren Programmentwicklung nutzen.

Ach übrigens wollte ich nur kurz noch erwähnen, es gibt in der mathematischen Bibliothek eine Funktion namens `fabs()`. Sie berechnet den Absolutwert eines `double` – korrekt und einwandfrei.

5.2 Programmentwicklung

An diesem Punkt sollten wir in der Lage sein komplette C Funktionen lesen und erklären zu können. Es ist aber sicher noch nicht so klar, wie man vorgeht um eigene Funktionen zu entwerfen und aufzuschreiben. Ich möchte deshalb an dieser Stelle eine Technik vorstellen, die ich **inkrementelle Entwicklung** nenne.

Stellen wir uns folgende Beispielaufgabe vor: Wir wollen den Abstand zwischen zwei Punkten herausfinden, deren Position jeweils durch x- und y-Koordinaten bestimmt ist. Ein Punkt hat die Koordinaten (x_1, y_1), der andere (x_2, y_2). Wir können den Abstand mit Hilfe der folgenden mathematischen Funktion ermitteln:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{5.1}$$

Wenn wir jetzt eine passende `Distance` Funktion in C entwerfen wollen, müssen wir im ersten Schritt überlegen, welche Eingaben (Parameter) und welche Ausgaben (Rückgabewerte) unsere Funktion für die Berechnung benötigt.

In unserem Fall sind die Koordinaten der zwei Punkte die Parameter. Wir müssen einen Datentyp festlegen und es ist nur natürlich hierfür reelle Zahlen vorzusehen, wir verwenden also vier `double`s. Der Rückgabewert unserer Funktion ist die Entfernung zwischen den Punkten und ist vom gleichen Typ `double`.

Damit können wir bereits die Grundzüge unsere Funktion in C aufschreiben:

```
double Distance (double x1, double y1, double x2, double y2)
{
    return 0.0;
}
```

Die `return`-Anweisung ist ein Platzhalter, so das sich die Funktion kompilieren lässt und einen Wert zurückgibt, obwohl das natürlich nicht die richtige Antwort ist.

An diesem Punkt tut die Funktion noch nichts sinnvolles, aber es ist trotzdem eine gute Idee sie bereits einmal zu kompilieren um eventuell vorhandene Syntaxfehler zu finden, bevor wir weitere Anweisungen hinzufügen.

Um die Funktion in einem Programm zu testen müssen wir sie mit Beispielwerten aufrufen. Irgendwo in `main()` könnten folgende Anweisungen hinzufügen:

```
double dist = Distance (1.0, 2.0, 4.0, 6.0);
printf  ("%f\n" dist);
```

Ich habe die Werte speziell ausgewählt, so dass der horizontale Abstand 3 und der vertikale Abstand 4 ist; auf diese Weise ergibt der korrekte Abstand den Wert 5 (die Hypotenuse eines 3-4-5 Dreiecks). Wenn wir die Rückgabewerte einer Funktion testen wollen, ist es eine gute Idee vorher die richtigen Antworten zu kennen.

Nachdem wir überprüft haben, dass die Syntax der Funktionsdefinition korrekt ist können wir anfangen weitere Codezeilen für die Berechnung hinzuzufügen. Nach jeder größeren Änderung kompilieren wir unser Programm und führen es erneut aus. Auf diese Weise ist es sehr einfach Fehler zu finden, die beim letzten Kompilieren noch nicht da waren. Sie müssen in den neu hinzugefügten Programmzeilen stecken!

Der nächste Schritt der Berechnung ermittelt die Differenz zwischen $x_2 - x_1$ und $y_2 - y_1$. Ich werde diese Werte in temporären Variablen mit Namen `dx` und `dy` speichern.

```
double Distance (double x1, double y1, double x2, double y2)
{
    double dx = x2 - x1;
    double dy = y2 - y1;
    printf  ("dx is %f\n",  dx);
    printf  ("dy is %f\n", dy;
    return 0.0;
}
```

In der Funktion habe ich noch zwei Ausgabeanweisungen hinzugefügt, so dass ich erst einmal die Zwischenergebnisse überprüfen kann, bevor ich weitermache. Ich habe es bereits erwähnt, ich erwarte an dieser Stelle die Werte 3.0 und 4.0.

Wenn die Funktion fertig ist werde ich die Ausgabeanweisungen wieder entfernen. Programmcode, der zu einem Computerprogramm hinzugefügt wird um bei der Programmentwicklung zu helfen, wird auch als **Debug-Code** bezeichnet. Dieser Programmcode sollte in der Endversion unseres Programms nicht mehr enthalten sein. Man kann Debug-Anweisungen auch im Quelltext eines Programms belassen und ihn zum Beispiel nur auskommentieren, dass heißt als Kommentar kennzeichnen. Auf diese Weise ist es einfach ihn später wieder zu aktivieren, wenn er gebraucht werden sollte.

Der nächste Schritt in unserer Berechnung ist die Quadrierung von `dx` und `dy`. Wir könnten dafür die `pow()` Funktion von C benutzen, es ist aber an dieser Stelle einfacher und schneller jeden Term einfach mit sich selbst zu multiplizieren:

```
double Distance (double x1, double y1, double x2, double y2)
{
    double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx*dx + dy*dy;
    printf  ("d_squared is %f\n", dsquared);
    return 0.0;
}
```

Es ist ratsam, an dieser Stelle das Programm erneut zu kompilieren und auszuführen. Dabei sollten wir den Wert des Zwischenergebnis kontrollieren – dieser sollte den Wert 25.0 haben.

Zum Schluss benutzen wir die `sqrt()` Funktion um das Endergebnis zu berechnen und geben dieses Resultat an die aufrufende Funktion zurück:

```
double Distance (double x1, double y1, double x2, double y2)
{
    double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx*dx + dy*dy;
    double result = sqrt (dsquared);
    return result;
}
```

In `main()` sollten wir uns diesen Wert ausgeben lassen und erneut überprüfen, ob das Resultat mit unseren Erwartungen übereinstimmt.

Im Laufe der Zeit, wenn wir unsere Programmiererfahrung verbessert haben, werden wir mehr und mehr Programmzeilen schreiben, bevor wir überprüfen, ob unser Programm fehlerfrei läuft. Trotzdem ist der inkrementelle Entwicklungsprozess auch dann noch sinnvoll und kann Ihnen helfen eine Menge Zeit bei der Fehlersuche zu sparen.

Die Schlüsselaspekte des Prozesses sind:

- Beginne die Programmentwicklung mit einem funktionsfähigen Programm und mache kleine, inkrementelle Änderungen. Kompiliere den Programmcode nach jeder Änderung. Jedes Mal wenn ein Fehler auftaucht ist sofort klar, wo dieser Fehler zu suchen ist.
- Benutze temporäre Variablen um Zwischenergebnisse zu speichern. Nutze diese Variablen um sie auf dem Bildschirm auszugeben und ihre Werte zu überprüfen.
- Nachdem das komplette Programm funktioniert, sollten Debug-Anweisungen entfernt (auskommentiert, nicht mit übersetzt) werden und

einzelne Anweisungen können zu komplexen Ausdrücken zusammengefasst werden. Es sollte aber darauf geachtet werden, dass dabei das Programm leicht lesbar bleibt. Es ist generell zu empfehlen einfachere Ausdrücke zu bevorzugen und dem Compiler die Optimierung des Programms zu überlassen.

5.3 Komposition

Wie wir bereits schon vermutet haben können wir, nachdem wir eine neue Funktion definiert haben, diese Funktion auch als Teil eines Ausdrucks verwenden. Ebenso können wir neue Funktionen mit Hilfe bereits existierender Funktionen erstellen.

Nehmen wir zum Beispiel an, dass uns jemand zwei Punkte nennt. Einer dieser Punkte sei der Mittelpunkt und der andere Punkt befindet sich auf dem Umkreis eines den Mittelpunkt umgebenden Kreises. Sie haben die Aufgabe aus diesen Angaben die Fläche des Kreises zu ermitteln.

Die Koordinaten des Mittelpunkts sollen in den Variablen `xc` und `yc` und die Koordinaten des Punkts auf dem Umkreis in `xp` und `yp` gespeichert sein. Der erste Schritt der Flächenberechnung besteht darin den Radius des Kreises zu ermitteln, welcher sich aus dem Abstand der beiden Punkte ergibt. Glücklicherweise haben wir bereits eine Funktion `Distance()`, die genau das für uns tut:

```
double radius = Distance (xc, yc, xp, yp);
```

Der zweite Schritt besteht darin den Kreisflächeninhalt auf der Basis des Radius zu berechnen und zurückzugeben (die Funktion `AreaCircle()` müssen wir noch schreiben!):

```
double result = AreaCircle (radius);
return result;
```

Wir können diese Schritte in einer neuen Funktion `AreaFromPoints()` zusammenfassen:

```
double AreaFromPoints (double xc, double yc, double xp, double yp)
{
    double radius = Distance (xc, yc, xp, yp);
    double result = AreaCircle (radius);
    return result;
}
```

Die temporären Variablen `radius` und `area` sind nützlich für die Programmentwicklung und die Fehlersuche, aber nachdem unser Programm funktioniert können wir den Programmcode knapp und präzise darstellen, indem wir die Funktionsaufrufe zu einem Ausdruck zusammenfassen:

```
double AreaFromPoints (double xc, double yc, double xp, double yp)
{
    return AreaCircle (Distance (xc, yc, xp, yp));
}
```

5.4 Boolesche Werte

Die numerischen Datentypen die wir bisher kennengelernt haben können Werte in einem sehr großen Wertebereiche speichern. Wir können sehr viele ganze Zahlen und noch mehr Fließkommazahlen darstellen. Das ist auch notwendig da die Zahlenbereiche in der Mathematik unendlich sind. Im Vergleich dazu ist die Menge der darstellbaren Zeichen vergleichsweise klein. Das hat Auswirkungen darauf, wie viel Speicherplatz ein Computer für die Speicherung dieser Werte benötigt. So benötigt ein Wert vom Typ `int` 2 bis 4 Byte, ein Wert vom Typ `double` 8 Byte und ein Wert vom Typ `char` 1 Byte Speicherplatz.

Viele Programmiersprachen implementieren noch einen weiteren fundamentalen Datentyp, der kleinste Informationseinheiten speichern kann und der noch viel kleiner ist. Es handelt sich dabei um so genannte **boolesche Werte** für deren Speicherung ein einzelnes Bit ausreicht. Boolesche Werte können nur zwei Zustände unterscheiden und werden üblicherweise für die Darstellung der Wahrheitswerte *true* und *false* genutzt.

Unglücklicherweise haben frühe Versionen des C Standards boolesche Werte nicht als separaten Datentyp implementiert. Sie benutzten statt dessen die ganzzahligen (integer) Werte 0 und 1 für die Darstellung der Wahrheitswerte. Dabei steht die 0 für den Wert `false` und die 1 für den Wert `true`. Genaugenommen interpretiert C jeden ganzzahligen Wert ungleich 0 als `true`. Das müssen wir beachten, wenn einen Wert auf `true` testen wollen. Wir dürfen ihn nicht mit `1` vergleichen sondern müssen überprüfen, ob er ungleich `!= 0` ist.

Ohne darüber nachzudenken, haben wir bereits im letzten Kapitel boolesche Werte benutzt. Die Bedingung innerhalb einer `if`-Anweisung ist ein boolescher Ausdruck. Die Vergleichsoperatoren liefern uns einen booleschen Wert als Resultat:

```
if (x == 5)
{
    /* do something*/
}
```

Der Operator `==` vergleicht zwei ganze Zahlen und erzeugt einen Wahrheitswert (boolescher Wert).

Da frühere C Standards auch keine Schlüsselwörter für die Angabe von `true` oder `false` kennen, verwenden viele Programme den C Präprozessor um sich selbst entsprechende Konstanten zu definieren. Diese können dann überall verwendet werden, wo ein boolescher Ausdruck gefordert ist.

Zum Beispiel:

```
#define FALSE 0
#define TRUE 1
 ...
if (x != FALSE)
{
    /* wird ausgeführt, wenn x ungleich 0 ist  */
}
```

5.5 Boolesche Variablen

Boolesche Werte werden in vielen C Versionen nicht direkt unterstützt. Mit dem C99 Standard wurde das geändert und der Datentyp `_Bool` eingeführt. Es gibt aber weiterhin viele Programme, deren Programmcode viel älter ist und auch nicht alle Compiler unterstützen den C99 Standard komplett. Viele Programmierer benutzen statt dessen den Datentyp `short` in Kombination mit den bereits erwähnten Präprozessordefinitionen um Wahrheitswerte zu speichern. In Variablen vom Datentyp `short` können wie bei `int` Ganze Zahlen gespeichert werden. Es aber weniger Bit zur Speicherung der Daten benutzt, das heißt, der Wertebereich von `short` ist kleiner als der von `int`.

```
#define FALSE 0
#define TRUE 1
 ...
short fred;
fred = TRUE;
short testResult = FALSE;
```

Die erste Anweisung des Programms ist eine einfache Variablendeklaration. Wir benutzen den Datentyp `short` um Speicherplatz zu sparen, wir hätten auch `int` verwenden können. Danach folgt eine Zuweisung, gefolgt von einer Kombination aus Deklaration und Zuweisung – eine so genannte Initialisierung.

Wie ich bereits erwähnte, liefern uns die Vergleichsoperatoren einen Wahrheitswert als Ergebnis. Das Resultat eines Vergleichs lässt sich in einer Variable speichern:

```
short evenFlag = (n%2 == 0);      /* true if n is even */
short positiveFlag = (x > 0);     /* true if x is positive */
```

So dass wir es später als Teil einer bedingten Anweisung nutzen können:

```
if (evenFlag)
{
    printf("n was even when I checked it");
}
```

Eine Variable die wir in dieser Art nutzen wird als **Flag** bezeichnet, weil sie uns die Anwesenheit oder Abwesenheit einer Bedingung markiert (*engl.:* to flag).

5.6 Logische Operatoren

Es existieren drei **logische Operatoren** in C: AND, OR und NOT, welche durch die Symbol `&&`, `||` und `!` dargestellt werden. Die Bedeutung (Semantik) dieser Operatoren leitet sich aus der Booleschen Algebra ab. Die logischen Operatoren haben eine geringere Priorität als arithmetischen Operatoren und Vergleichsoperatoren, dass heißt, sie werden erst nach der Auswertung von Vergleichen und Berechnungen angewendet.

AND-Operator Im folgenden Ausdruck werden zuerst die Vergleiche durchgeführt und danach werden die Wahrheitswerte der Vergleiche durch den `AND`-Operator miteinander verknüpft:

```
x > 0 && x < 10
```

Der gesamte Ausdruck ist dann *true*, wenn `x` größer als Null und (AND) kleiner als 10 ist.

OR-Operator Der `OR`-Operator wird folgendermaßen verwendet:

```
evenFlag || number%3 == 0
```

Der Ausdruck ist *true*, wenn *entweder* das `evenFlag` einen Wert ungleich Null hat oder (OR) die Variable `number` ohne Rest durch 3 teilbar ist. Aus Gründen der besseren Lesbarkeit empfiehlt es sich auch hier Klammern zu verwenden. Wir hätten den Ausdruck auch folgendermaßen aufschreiben können:

```
evenFlag || (number%3 == 0)
```

NOT-Operator Der `NOT`-Operator kehrt den Wahrheitswert eines booleschen Ausdrucks in sein Gegenteil um (er negiert den Ausdruck). Damit lässt sich in unserem vorigen Beispiel die Bedingung umkehren:

```
!(number%3 == 0)
```

Der Ausdruck wäre dann wahr, wenn `number` nicht (NOT) durch 3 teilbar ist.

Logische Operatoren werden oft dazu verwendet verschachtelte Programmverzweigungen zu vereinfachen. So könnte man in dem folgenden Beispiel das Programm so vereinfachen, dass nur eine einzige `if`-Anweisung benutzt wird. Könnten Sie dazu die Bedingung aufschreiben?

```
if (x > 0)
{
    if (x < 10)
    {
        printf ("x is a positive single digit.\n");
    }
}
```

5.7 Boolesche Funktionen

Es ist manchmal angebracht, dass eine Funktion einen Wahrheitswert an die aufrufende Funktion zurückgibt, so wie wir das bei anderen Datentypen auch tun. Das ist insbesondere dann vorteilhaft, wenn in der Funktion komplexe Tests durchgeführt werden und der aufrufenden Funktion nur mitgeteilt werden soll, ob der Test erfolgreich war oder nicht.

Zum Beispiel:

```
int IsSingleDigit (int x)
{
    if (x >= 0 && x < 10)
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}
```

Der Name der Funktion lautet `IsSingleDigit()`. Es ist üblich solchen Testfunktionen einen Namen zu geben, der wie eine Ja/Nein Frage formuliert ist. Der Rückgabewert der Funktion ist `int`, das bedeutet, dass wir erneut der Übereinkunft folgen, dass 0 – `false` und 1 – `true` darstellt. Jede `return`-Anweisung muss diese Konvention befolgen und wir verwenden dazu wieder die bereits bekannten Präprozessordefinitionen.

Der Programmcode ist unkompliziert, wenngleich etwas länger als eigentlich nötig. Wir können versuchen ihn noch weiter zusammenfassen. Erinnern wir uns, der Ausdruck `x >= 0 && x < 10` wird zu einem booleschen Wert ausgewertet. Es ist daher ohne weiteres möglich den Wert des Ausdrucks direkt an die aufrufende Funktion zurückzugeben und auf die `if`- Anweisungen komplett zu verzichten:

```
int IsSingleDigit (int x)
{
    return (x >= 0 && x < 10);
}
```

In `main` können wir die Funktion in der üblichen Weise aufrufen:

```
printf("%i\n", IsSingleDigit (2));
short bigFlag = !IsSingleDigit (17);
```

Die erste Zeile gibt den Wert *true* aus, weil 2 eine einstellige positive Zahl ist. Unglücklicherweise sehen wir bei der Ausgabe von Wahrheitswerten in C nicht die Worte `TRUE` und `FALSE` auf dem Bildschirm, sondern die Zahlen `1` und `0`.

Die zweite Zeile ist eine Zuweisung. Der Variablen `bigFlag` wird der Wert `true` zugewiesen, wenn das Argument der Funktion keine positive einstellige Zahl ist.

Boolesche Funktionen werden sehr häufig für die Auswertung der Bedingungen in Programmverzweigungen genutzt:

```
if (IsSingleDigit (x))
{
    printf("x is little\n");
}
else
```

```
{
    printf("x is big\n");
}
```

5.8 Rückgabewerte in der `main()`-Funktion

Nachdem wir jetzt wissen, dass Funktionen Werte zurückgeben können, ist es an der Zeit, dass wir uns etwas genauer mit der Funktion der `return`-Anweisung in der `main`-Funktion beschäftigen. Wenn wir uns die Definition der Funktion anschauen stellen wir fest, dass sie einen ganzzahligen Wert (integer) zurückgeben sollte :

```
int main (void)
```

Der übliche Rückgabewert aus `main` ist 0. Damit wird angezeigt, dass das Programm in seiner Ausführung erfolgreich war und genau das getan hat wozu es programmiert wurde. Wenn während der Ausführung des Programms irgend ein Fehler auftritt ist es üblich -1 oder einen anderen Wert, der angibt um welchen Fehler es sich handelt, zurückzugeben.

C stellt in der Standardbibliothek zwei Konstanten zur Verfügung `EXIT_SUCCESS` und `EXIT_FAILURE`, die wir in der Rückgabeanweisung nutzen können. Dafür müssen `stdlib.h` in unser Programm einbinden:

```
#include <stdlib.h>

int main (void)
{
    return EXIT_SUCCESS;   /*program terminated successfully*/
}
```

Natürlich werden Sie sich fragen wer diese Rückgabewerte empfängt, weil wir die `main`-Funktion niemals selbst irgendwo aufrufen. Es stellt sich heraus, dass dafür das Betriebssystem unseres Rechners verantwortlich ist. Jedes Mal, wenn das Betriebssystem ein Programm startet, ruft es `main` auf, so wie wir selbst in unserem Programm eigene Funktionen aufrufen. Ist das Programm beendet, erhält das Betriebssystem eine Mitteilung, ob das Programm erfolgreich war und könnte darauf reagieren (zum Beispiel einen Fehlerbericht verfassen).

Es ist sogar möglich in der `main`-Funktion Parameter zu verwenden, um beim Programmstart bereits Daten an das Programm zu übergeben. Leider können wir darauf an dieser Stelle noch nicht genauer eingehen.

5.9 Glossar

Rückgabewert (engl: *return value*): Der Wert der bei der Rückkehr aus einer Funktion an die aufrufende Funktion zurückgegeben wird.

Rückgabetyp (engl: *return type*): Der Typ des Werts der bei der Rückkehr aus einer Funktion an die aufrufende Funktion zurückgegeben wird.

Unerreichbarer Code (engl: *dead code*): Der Teil des Programmcodes der niemals ausgeführt wird, zum Beispiel weil sich der Code hinter einer `return` Anweisung befindet.

Debug Code (engl: *debug code*): Anweisungen innerhalb eines Programms, welche dazu genutzt werden das Programm während der Entwicklung zu testen. Diese Anweisungen sollten im fertigen Programm deaktiviert werden.

void (engl: *void*): Ein spezieller Typ, der verwendet wird um zu kennzeichnen, dass eine Funktion keinen Wert zurückliefert, und/oder keine Parameter besitzt.

Boolesche Variable (engl: *boolean*): Eine Variable, welche einen von zwei möglichen Zuständen annehmen kann, oft mit *true* und *false* bezeichnet. In C werden boolsche Werte überwiegend in Variablen vom Type `int` oder `short` gespeichert und mit Hilfe des Präprozessors definiert.
(z.B. `#define TRUE 1`)

Flag (engl: *flag*): Eine Variable, welche eine Bedingung oder einen Statuscode speichert (meistens ein boolscher Wert).

Vergleichsoperator (engl: *comparison operator*): Ein Operator, welcher zwei Werte vergleicht und als Ergebnis einen Wahrheitswert liefert, welcher die Beziehung der Operanden des Ausdrucks charakterisiert.

Logischer Operator (engl: *logical operator*): Ein Operator, der die Operanden auf Basis einer logischen Verknüfung (UND, ODER, NICHT) auswertet und einen Wahrheitswert zurückliefert.

5.10 Übungsaufgaben

Übung 5.1

Sie haben 3 Stöcke erhalten und stehen vor der Aufgabe daraus ein Dreieck zu formen. Diese Aufgabe kann lösbar oder unlösbar sein, je nachdem wie lang die zur Verfügung stehenden Stöcke sind.

Wenn zum Beispiel einer der Stöcke 12cm lang ist und die anderen Beiden je nur 2cm, so ist klar, dass diese sich nicht in der Mitte treffen werden. Es gibt einen einfachen Test, der für drei beliebige Längen ermittelt, ob sich ein Dreieck formen lässt oder nicht:

> "Wenn eine der drei Längen größer ist als die Summe der anderen beiden, dann lässt sich kein Dreieck formen. Ansonsten ist es möglich ein Dreieck zu formen."

Schreiben Sie eine Funktion mit dem Namen `IsTriangle`, welche drei `integer` als Argumente hat und entweder `TRUE` or `FALSE` zurückgibt, abhängig davon, ob sich aus Stöcken mit der gegebenen Länge ein Dreieck formen lässt oder nicht.

Der Sinn dieser Übung bestehet darin eine Funktion mit bedingten Abfragen zu schreiben, welche als Ergebnis einen Wert zurückgibt.

Übung 5.2

Schreiben Sie eine Funktion `IsDivisible` welche zwei `integer` Werte, `n` and `m` als Argumente hat und `TRUE` zurückgibt, wenn `n` durch `m` teilbar ist. Ansonsten soll die Funktion `FALSE` zurückgeben.

Übung 5.3

Der Sinn der folgenden Übung besteht darin das Verständnis für die Ausführung logischer Operatoren zu schärfen und den Programmablauf in Funktionen mit Rückgabewerten nachvollziehbar zu machen. Wie lautet die Ausgabe des folgenden Programms?

```
#define TRUE 1
#define FALSE 0

  short IsHoopy (int x)
   {
      short hoopyFlag;
      if (x%2 == 0)
      {
          hoopyFlag = TRUE;
      }
      else
      {
          hoopyFlag = FALSE;
      }
      return hoopyFlag;
  }

  short IsFrabjuous (int x)
  {
      short frabjuousFlag;
      if (x > 0)
      {
          frabjuousFlag = TRUE;
      }
      else
      {
          frabjuousFlag = FALSE;
      }
      return frabjuousFlag;
  }

  int main (void)
  {
      short flag1 = IsHoopy (202);
      short flag2 = IsFrabjuous (202);
      printf ("%i\n", flag1);
      printf ("%i\n", flag2);
      if (flag1 && flag2)
```

```
    {
        printf ("ping!\n");
    }
    if (flag1 || flag2)
    {
        printf ("pong!\n");
    }
    return EXIT_SUCCESS;
}
```

Übung 5.4

Die Entfernung zwischen zwei Punkten (x_1, y_1) und (x_2, y_2) ist

$$Distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Schreiben Sie bitte eine Funktion `Distance` welche vier `double` als Argumente erhält—`x1`, `y1`, `x2` und `y2`—und welche die Entfernung zwischen den Punkten (x_1, y_1) und (x_2, y_2) zurückgibt.

Sie sollen annehmen, dass bereits eine Funktion mit dem Namen `SumSquares` existiert, welche die Quadrate der Summen berechnet und zurückgibt.

Zum Beispiel:

```
double x = SumSquares (3.0, 4.0);
```

würde `x` den Wert `25.0` zuweisen.

Der Sinn dieser Übung besteht darin eine neue Funktion zu schreiben, welche eine bereits bestehende Funktion aufruft. Sie sollen nur die eine Funktion `Distance` schreiben. Lassen Sie die Funktionen `SumSquares` und `main` weg und rufen Sie `Distance` auch nicht auf!

Übung 5.5

Erstellen Sie eine neue Programmdatei mit dem Namen `Sum.c`, und geben Sie die folgenden zwei Funktionen ein:

```
int FunctionOne (int m, int n)
{
    if (m == n)
    {
        return n;
    }
    else
    {
        return m + FunctionOne (m+1, n);
    }
}

int FunctionTwo (int m, int n)
```

```
{
    if (m == n)
    {
        return n;
    }
    else
    {
        return n * FunctionTwo (m, n-1);
    }
}
```

a. Fügen Sie in den Funktionen eine `prinf()`-Anweisung hinzu, mit der Sie sofort nach dem Funktionsaufruf den aktuellen Wert der Funktionsparameter ausgeben. Das ist eine nützliche Technik um rekursive Programme zu debuggen, da sich auf diese Weise die Abarbeitung eines Programms besser nachvollziehen lässt.

b. Schreiben Sie in der `main`-Funktion ihres Programms einige Zeilen um diese Funktionen zu testen (rufen Sie die Funktionen einige Male mit unterschiedlichen Argumenten auf und lassen Sie sich die Rückgabewerte ausgeben, um zu sehen, was die Funktionen machen.

   Nutzen Sie eine Kombination aus gezieltem Testen und der Inspektion des Quellcodes um herauszufinden, was diese Funktionen machen. Geben Sie den Funktionen einen neuen Namen, aus dem besser hervorgeht, was die Funktionen machen. Fügen Sie Kommentare zu den Funktionen hinzu um ihre Funktion allgemeinverständlich zu beschreiben.

Übung 5.6

(This exercise is based on page 44 of Ableson and Sussman's *Structure and Interpretation of Computer Programs.*)

The following algorithm is known as Euclid's Algorithm because it appears in Euclid's *Elements* (Book 7, ca. 300 B.C.). It may be the oldest nontrivial algorithm.

The algorithm is based on the observation that, if r is the remainder when a is divided by b, then the common divisors of a and b are the same as the common divisors of b and r. Thus we can use the equation

$$gcd(a, b) = gcd(b, r)$$

to successively reduce the problem of computing a GCD to the problem of computing the GCD of smaller and smaller pairs of integers. For example,

$$gcd(36, 20) = gcd(20, 16) = gcd(16, 4) = gcd(4, 0) = 4$$

implies that the GCD of 36 and 20 is 4. It can be shown that for any two starting numbers, this repeated reduction eventually produces a pair where the second number is 0. Then the GCD is the other number in the pair.

Write a function called `gcd` that takes two integer parameters and that uses Euclid's algorithm to compute and return the greatest common divisor of the two numbers.

Kapitel 6

Iteration

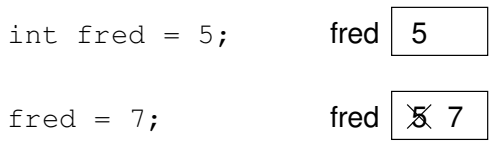
6.1 Zuweisung unterschiedlicher Werte

Ich habe noch nicht darüber gesprochen, aber es ist durchaus erlaubt einer Variablen mehr als einmal einen Wert zuzuweisen. Der Effekt der zweiten Zuweisung besteht darin, dass der *alte* Wert der Variablen durch einen *neuen* Wert ersetzt wird:

```
int fred = 5;
printf ("%i", fred);
fred = 7;
printf ("%i", fred);
```

Die Ausgabe dieses Programms ist `57`.
Wenn `fred` zum ersten Mal ausgegeben wird, hat die Variable den Wert 5. Zum Zeitpunkt der zweiten Ausgabe hat die Variable den Wert 7.

Diese Art von **aufeinanderfolgenden Zuweisungen** ist der Grund warum ich Variablen als ein *Container* für Werte bezeichnet habe. Wenn wir einer Variablen einen Wert zuweisen, wird der Inhalt dieses Containers verändert, wie in der folgenden Grafik dargestellt:



Wenn wir mehrere Zuweisungen zu einer Variable vornehmen, ist es besonders wichtig, dass wir zwischen der Zuweisungsanweisung und der Anweisung, welche die Gleichheit von Werten testet unterscheiden.

Die Programmiersprache C benutzt das Symbol `=` für die Zuweisung von Werten. Es ist daher verführerisch, die Anweisung `a = b` als eine Überprüfung der Gleichheit der Variablen `a` und `b` zu interpretieren. Was nicht der Fall ist!

Zuerst einmal können wir feststellen, dass die Gleichheitsoperation kommutativ ist und eine Zuweisungsoperation nicht. In der Mathematik gilt:

$$\text{Wenn } a = 7, \text{ dann } 7 = a$$

In C ist `a = 7;` eine gültige Anweisung. Wenn wir aber die Anweisung `7 = a;` in unser Programm schreiben erhalten wir einen Fehler. Bei der linken Seite einer Zuweisung muss es sich um einen Ort im Speicher des Computers handeln.

Weiterhin ist in der Mathematik ein Ausdruck der Gleichheit zu jeder Zeit wahr. Wenn $a = b$ ist, dann wird a **immer** den gleichen Wert besitzen wie b. In C, kann eine Zuweisung zwei Variablen den gleichen Wert geben, aber die Werte der Variablen sind damit nicht für alle Zeit festgelegt und können sich ändern!

```
int a = 5;
int b = a;      /* a und b haben jetzt den gleichen Wert */
a = 3;          /* a und b sind nicht länger gleich */
```

Die dritte Zeile ändert den Wert von `a`, der Wert der Variablen `b` ist davon aber nicht betroffen. Ab diesem Zeitpunkt im Programm sind die Variablen nicht länger gleich. In vielen anderen Programmiersprachen wird deshalb für die Wertzuweisung an eine Variable ein anderes Symbol (`:=` oder `<-`) und nicht das Gleichheitszeichen benutzt. Damit wird die Verwechslungsgefahr zwischen den Operationen verringert.

Obwohl die mehrfache Zuweisung von unterschiedlichen Werten an eine Variable oft sehr nützlich sein kann, sollten wir diese Art der Wertzuweisung mit Vorsicht benutzen. Wenn sich der Wert einer Variablen ständig an unterschiedlichen Stellen in einem Programm verändert, so wird das Lesen und die Fehlersuche in dem Programm deutlich erschwert.

6.2 Iteration - Wiederholungen im Programm

Eine der wichtigsten Aufgaben die durch Computer übernommen werden ist die Automatisierung ständig wiederkehrender Aufgaben. Die fehlerfreie Wiederholung identischer oder sehr ähnlicher Aufgaben ist ein Gebiet auf dem Computer den menschlichen Benutzern deutlich überlegen sind.

Wir haben im Kapitel 4.8 bereits Funktionen wie `PrintLines()` und `Countdown()` kennengelernt, die mit Hilfe der Rekursion wiederkehrende Aufgaben bewältigt haben. Dabei wurden Wiederholungen durch die ineinander geschachtelte Ausführung von Funktionen erreicht.

Wir werden jetzt eine neue Art der Ausführung von Wiederholungen kennenlernen bei der mit Hilfe von Kontrollstrukturen die Ausführung gesteuert werden kann. Diese Art der wiederholten Ausführung bezeichnet man auch als **Iteration**. Mit Hilfe der `while` und der `for`-Anweisung können wir die Wiederholung von Anweisungsblöcken genau steuern.

6.3 Die while-Anweisung

Ich möchte kurz an einem Beispiel zeigen, wie wir die bereits bekannte `Countdown()` Funktion mittels einer `while`-Anweisung umschreiben können:

```
void Countdown (int n)
{
    while (n > 0)
    {
        printf ("%i\n", n);
        n = n-1;
    }
    printf ("Blastoff!\n");
}
```

Was auffällt ist die gute Lesbarkeit des Quelltextes der `while`-Anweisung, welcher sich fast von selbst erklärt. Die Anweisung hat folgende Bedeutung: "Solange (engl: *while*) `n` größer als Null ist, geben wir den aktuellen Wert von `n` auf dem Bildschirm aus. Danach verringern wir den Wert von `n` um 1. Wenn `n` den Wert Null erreicht hat, wird die Schleife verlassen und das Wort "Blastoff!" auf dem Bildschirm ausgegeben.

Etwas formeller können wir die Arbeitsweise der `while`-Anweisung folgendermaßen beschreiben:

1. Die in Klammern angegebene Bedingung wird ausgewertet und der Wahrheitswert `true` oder `false` ermittelt.

2. Wenn die Bedingung falsch ist, wird die `while`-Anweisung verlassen und die Ausführung des Programms mit der nächstfolgenden Anweisung fortgesetzt.

3. Wenn die Bedingung wahr ist, werden alle Anweisungen im Anweisungsblock der `while`-Anweisung nacheinander ausgeführt und am Ende des Blocks wird zu Schritt 1 zurückgekehrt.

Diese Art des Programmablaufs wird auch als eine **Schleife** bezeichnet, weil der dritte Schritt im Ablauf wieder auf den ersten Schritt zurückführt.

Die Anweisungen im Inneren der Schleife bezeichnet man als den **Schleifenkörper**. Sollte es der Fall sein, dass gleich beim ersten Überprüfen der Bedingung der Wert `false` ermittelt wird, so werden die Anweisungen im Schleifenkörper überhaupt nicht ausgeführt.

Üblicherweise wird die Ausführungshäufigkeit der Schleife durch eine Kontrollvariable gesteuert. Der Körper einer Schleife sollte den Wert der Kontrollvariablen so ändern, dass schließlich irgendwann die Bedingung den Wert `false` erhält und die Schleife beendet wird. Anderenfalls würde die Schleife unendlich oft wiederholt werden. Eine derartige Schleife bezeichnet man dann als **Endlosschleife**. Informatiker finden daher zum Beispiel den Aufdruck auf einigen

Shampoo-Flaschen sehr amüsant: “Einseifen, Ausspühlen, Wiederholen” ist eine Endlosschleife.

Im Fall von `Countdown()` können wir beweisen, dass die Schleife irgendwann beendet sein muss, weil wir wissen dass der Wert von `n` endlich ist und wir sehen können, dass mit jedem Schleifendurchlauf (jeder **Iteration**) der Wert von `n` stetig kleiner wird. Irgendwann wird dieser Wert also Null sein und die Schleife endet.

Nicht in allen Fällen lässt sich das so einfach ermitteln:

```
void Sequence (int n)
{
    while (n != 1)
    {
        printf ("%i\n", n);
        if (n%2 == 0)       /* n ist gerade */
        {
            n = n / 2;
        }
        else                /* n ist ungerade */
        {
            n = n*3 + 1;
        }
    }
}
```

Die Bedingung dieser Schleife ist `n != 1`, dass heißt die Schleife wird fortgesetzt bis `n` den Wert 1 erhält, was dazu führt, dass die Bedingung falsch wird.
Bei jeder Iteration gibt das Programm den Wert von `n` aus und prüft dann, ob `n` gerade oder ungerade ist. Ist es gerade, so wird der Wert von `n` durch Zwei geteilt. Ist `n` ungerade ermittelt sich der neue Wert von `n` aus der Formel $3n+1$.

Nehmen wir an, der Startwert der Funktion (das Argument welches der Funktion `Sequence()` übergeben wurde) ist 3. Damit ergibt sich die folgende Sequenz:
3, 10, 5, 16, 8, 4, 2, 1.

Weil sich nun der Wert von `n` manchmal vergrößert und manchmal verringert, gibt es keinen naheliegenden Beweis, dass `n` überhaupt jemals den Wert 1 erreichen wird (und damit das Programm beendet würde). Für einige bestimmte Werte von `n` lässt sich beweisen, dass die Schleife endlich ist. Ist zum Beispiel der Startwert von `n` eine Zweierpotenz, so ist jedes Zwischenresultat gerade und die auszuführende Division führt geradewegs zum Ergebnis 1. In unserem vorigen Beispiel stellen die letzten 5 Ziffern eine solche Sequenz dar, die mit dem Wert 16 beginnt.

Wenn wir aber herausfinden wollen, ob diese Schleife für alle nur denkbaren Werte von `n` endlich ist, dann stehen wir vor einer sehr großen Herausforderung. Bisher ist es jedenfalls noch niemandem gelungen dies zu Beweisen *oder* den Gegenbeweis anzutreten!

6.4 Tabellen

Eine Sache die sich mit Hilfe von Schleifen leicht umsetzen lässt, ist die Erzeugung von tabellarischen Daten.

Bevor Menschen Computer zur Verfügung hatten, mussten Logarithmen, Sinus, Kosinus und andere mathematische Funktionen von Hand berechnet werden. Um diese Berechnungen zu vereinfachen wurden Bücher mit Tabellen der Funktionswerte von häufig genutzten Funktionen gedruckt. Die Erstellung dieser Tabellen war langwierig und langsam und die Resultate waren oftmals fehlerhaft.

Als schließlich Computer verfügbar wurden, war die erste Reaktion der Mathematiker, "Das ist großartig! Von jetzt ab werden wir die Tabellen mit Hilfe des Computer berechnen und dann gibt es keine Fehler mehr." Diese Voraussage stellte sich als richtig heraus, war aber nicht sehr visionär, denn kurze Zeit später waren Computer und Taschenrechner so weit verbreitet, dass niemand mehr die Tabellen benutzt um Funktionswerte zu ermitteln.

Na ja, jedenfalls meistens. Für einige Berechnungen benutzen selbst Computer Tabellen, um Näherungswerte zu bestimmen. Mit diesen Näherungswerten führen sie dann weitere Berechnungen durch, um das Ergebnis zu verbessern. Leider ist es aber auch schon vorgekommen, dass in diesen internen Tabellen Fehler enthalten waren. Ein bekanntes Beispiel dafür war der Fehler in den Tabellen des ersten Intel Pentium Prozessors, welcher dazu führte, dass manche Ergebnisse der Division von Fließkommazahlen nicht korrekt waren.

Obwohl eine "Logarithmentafel" heutzutage nicht mehr so nützlich ist wie früher, stellt ihre Berechnung immer noch ein gutes Beispiel für die Anwendung iterativer Algorithmen dar. Das folgende Programm stellt in der linken Spalte eine Folge von Werten und in der rechten Spalte die dazugehörigen Logarithmen dar.

```
double x = 1.0;
while (x < 10.0)
{
    printf ("%.0f\t%f\n", x ,log(x));
    x = x + 1.0;
}
```

Die Zeichenfolge `\t` steht dabei für das **Tab**-Zeichen. Die Zeichenfolge `\n` repräsentiert das Zeichen für den Zeilenumbruch (engl: *newline*). Diese Zeichenfolgen sind sogenannte Ersetzungszeichen für Zeichen aus dem ASCII-Zeichensatz, die sich nicht direkt darstellen lassen. Sie können an jeder beliebigen Stelle in einem String stehen – in unserem Beispiel sind sie die einzigen Zeichen im Formatierungsstring.

Das **Tab**-Zeichen veranlasst den Cursor um eine bestimmte Anzahl von Zeichen nach rechts zu rücken, bis der nächste **Tab Stop** erreicht ist. Normalerweise beträgt dieser Abstand 8 Zeichen. Wie wir gleich sehen werden, sind Tabulatoren sehr nützlich um die Spalten einer Tabelle gleichmäßig auszurichten. Ein Zeilenumbruch führt dazu, dass der Cursor auf die nächste Bildschirmzeile bewegt wird.

Die Ausgabe des Programms sieht folgendermaßen aus:

```
1       0.000000
2       0.693147
3       1.098612
4       1.386294
5       1.609438
6       1.791759
7       1.945910
8       2.079442
9       2.197225
```

Wenn Ihnen diese Werte seltsam vorkommen, so erinnern Sie sich bitte daran, dass die `log`-Funktion die Basis e benutzt. Da Zweierpotenzen in der Informatik so eine große Rolle spielen, kommt es oft vor, dass wir Logarithmen zur Basis 2 finden wollen. Wir können dazu folgende Formel benutzen:

$$\log_2 x = \frac{log_e x}{log_e 2}$$

Wenn wir die Ausgabeanweisung wie folgt ändern,

```
printf ("%.0f\t%f\n", x, log(x) / log(2.0));
```

ergibt sich:

```
1       0.000000
2       1.000000
3       1.584963
4       2.000000
5       2.321928
6       2.584963
7       2.807355
8       3.000000
9       3.169925
```

Wir können erkennen, dass 1, 2, 4 und 8 Zweierpotenzen sind, weil ihre Logarithmen zur Basis 2 runde Zahlen sind. Wenn wir die Logarithmen weiterer Zweierpotenzen ermitteln wollen, können wir das Programm folgendermaßen verändern:

```
double x = 1.0;
while (x < 100.0)
{
    printf ("%.0f\t%.0f\n", x, log(x) / log(2.0));
    x = x * 2.0;
}
```

Statt bei jedem Schleifendurchlauf einen festen Betrag zu `x` hinzuzuaddieren, was zu einer **arithmetischen Folge** führt, multiplizieren wir `x` mit einem Wert, woraus eine **geometrischen Folge** resultiert.

Das Resultat ist:

```
1       0
2       1
4       2
8       3
16      4
32      5
64      6
```

Auffallend ist die exakte Ausrichtung der Spalten auch bei größeren Werten. Da wir zwischen den Spalten tab-Zeichen verwenden, hängt die Position der zweiten Spalte nicht von der Anzahl der Ziffern in der ersten Zeile ab.

Logarithmentafeln mögen heutzutage nicht mehr sehr nützlich sein. Die Kenntnis der Zweierpotenzen ist allerdings für Informatiker und Elektroniker nach wie vor extrem wichtig! Modifizieren Sie daher das Programm, so dass es alle Zweierpotenzen bis zum Wert 65536 (das ist 2^{16}) ausgibt. Drucken Sie das Ergebnis aus und prägen Sie sich die Werte ein.

6.5 Zweidimensionale Tabellen

Eine zweidimensionale Tabelle ist eine Tabelle, bei der man die Zeile und Spalte auswählt und den Wert am Kreuzungspunkt ausliest. Ein gutes Beispiel dafür ist eine Multiplikationstafel.

Angenommen, wir wollen eine Multiplikationstafel für alle Werte von 1 bis 6 erstellen. Ein guter Anfang könnte darin bestehen, dass wir eine einfache Schleife schreiben, welche alle Vielfachen von 2 in einer Zeile ausgibt:

```
int i = 1;
while (i <= 6)
{
    printf("%i   ", i*2);
    i = i + 1;
}
printf("\n");
```

In der ersten Zeile wird eine Variable namens `i` initialisiert. Diese Variable ist die **Schleifenvariable**, die uns als Zähler dient. Während der Ausführung der Schleife erhöht sich der Wert von `i` von 1 bis 6. Wenn `i` den Wert 7 erreicht, wird die Schleife abgebrochen. Bei jedem Schleifendurchlauf geben wir den Wert von `i*2`, gefolgt von drei Leerzeichen aus. Indem wir in der ersten Ausgabeanweisung einfach die Zeichenfolge `\n` weg lassen, werden alle auszugebenden Werte nacheinander in einer Bildschirmzeile ausgegeben.

Das Programm erzeugt folgende Ausgabe:

```
2   4   6   8   10   12
```

So weit, so gut. Der nächste Schritt besteht darin die Funktionalität des Programms zu **kapseln** und zu **verallgemeinern**.

6.6 Modularisierung und Verallgemeinerung

Modularisierung bedeutet, dass wir unseren Quellcode in mehrere separate Programmabschnitte (Module) aufteilen. In C werden diese Module als Funktionen realisiert. Indem wir unseren Code nehmen und in einer Funktion verpacken, können wir von allen Vorteilen profitieren, die uns Funktionen bei der Programmentwicklung bieten. Unsere Programme werden übersichtlicher und im Laufe der Zeit entsteht unsere eigene kleine Funktionsbibliothek die wir immer wieder benutzen und erweitern können.

Wir haben mit `PrintParity()` in Section 4.4 und `IsSingleDigit()` in Section 5.7 bereits zwei Beispiele für modularisierte Programme kennengelernt.

Verallgemeinerung bedeutet, dass wir aus einer spezifischen, auf ein bestimmtes Problem bezogenen Lösung, eine allgemeinere Lösung entwickeln, die es uns erlaubt mehrere Probleme der gleichen Klasse zu lösen. Zum Beispiel kann unser Programm derzeit Vielfache von 2 ausgeben. Wir wollen unser Programm verallgemeinern, so dass wir die Vielfachen einer beliebigen ganzen Zahl ausgeben können.

Ich habe also unsere Schleife als eine Funktion umgeschrieben. Gleichzeitig haben wir die Eigenschaft der Funktion verallgemeinert, so dass es jetzt möglich ist, die Vielfachen von `n` auszugeben:

```
void PrintMultiples (int n)
{
    int i = 1;
    while (i <= 6)
    {
        printf("%i   ", i*n);
        i = i + 1;
    }
    printf("\n");
}
```

Um die Schleife zu modularisieren, habe ich einfach die erste Zeile hinzugefügt. Diese deklariert den Funktionsnamen und den Rückgabewert der Funktion. Der Rest der Schleife wird in eine Blockanweisung geschrieben.

Um die Funktion zu verallgemeinern ist es notwendig einen Funktionsparameter hinzuzufügen. Dieser gibt den Wert an, der in der Schleife vervielfacht wird. Im Schleifenkörper ersetzen wir dann einfach den Wert 2 mit dem Parameter `n`.

Wenn wir diese Funktion mit dem Argument 2 aufrufen, erhalten wir die gleiche Ausgabe wie zuvor. Mit Argument 3, sieht die Ausgabe folgendermaßen aus:

```
3   6   9   12   15   18
```

und Argument 4, ergibt die folgende Ausgabe:

```
4   8   12   16   20   24
```

Mittlerweile sollte klar werden, wie wir weiter vorgehen müssen, um eine komplette Multiplikationstafel zu drucken. Wir rufen einfach `PrintMultiples()`

mehrfach mit unterschiedlichen Argumenten auf. Am einfachsten ist es, wenn wir dazu wieder eine Schleife benutzen.

```
int i = 1;
while (i <= 6)
{
    PrintMultiples (i);
    i = i + 1;
}
```

Es ist auffallend, wie sehr die äußere Schleife der inneren Schleife von `PrintMultiples()` ähnelt. Ich habe nichts weiter gemacht, als den Aufruf der `printf()`-Funktion durch den Funktionsaufruf von `PrintMultiples()` zu ersetzen.

Die Ausgabe des Programms sieht folgendermaßen aus:

```
1   2   3   4   5   6
2   4   6   8   10   12
3   6   9   12   15   18
4   8   12   16   20   24
5   10   15   20   25   30
6   12   18   24   30   36
```

Wir sehen eine (leicht unordentliche) Multiplikationstafel. Wenn Sie die Unordnung stört, dann können Sie versuchen die Leerzeichen zwischen den Spalten der Ausgabe durch **Tab**-Zeichen zu ersetzen und damit die Anordnung zu verbessern.

6.7 Funktionen

Im letzte Abschnitt habe ich von den *Vorteilen, die uns Funktionen bei der Programmentwicklung bieten* gesprochen. Wahrscheinlich haben Sie sich schon gefragt, was genau ich wohl damit gemeint haben könnte. Ich möchte deshalb noch einmal genauer auf den Vorteil des Einsatzes von Funktionen bei der Programmentwicklung eingehen.

Die Aufteilung eines Programms in Funktionen hat den Vorteil, dass wir jedes Modul unabhängig vom restlichen Quellcode unseres Programms entwickeln und testen können. Möglicherweise können wir diese Funktionen auch später in anderen Projekten einfach weiterverwenden. Außerdem sind modularisierte Programme viel leichter zu durchschauen, als wenn wir alle Anweisungen einfach hintereinander in die `main()`-Funktion schreiben würden:

- Indem wir einer konzeptionell zusammengehörenden Folge von Anweisungen einen Namen geben, machen wir unser Programm einfacher lesbar. Gleichzeitig wird die Fehlersuche einfacher.
- Wenn wir ein langes Programm in Teile zerlegen, können wir diese Teile einzeln entwickeln und testen. Anschließend können wir die Funktionen wieder zu einem funktionsfähigen Programm zusammensetzen.

- Funktionen erlauben den einfachen Einsatz von rekursiver und iterativer Programmierung.
- Gut konstruierte Funktionen können in vielen künftigen Programmen weiter verwendet werden (die Bibliotheksfunktionen von C sind solche Funktionen). Nachdem wir eine Funktion geschrieben und vorhandene Fehler entfernt haben, können wir sie einfach immer wieder verwenden.

6.8 Noch mehr Modularisierung

Um ein weiteres Beispiel für die Modularisierung von Programmen zu geben, werde ich jetzt den Programmcode aus dem letzten Beispiel nehmen und in einer Funktion kapseln:

```
void PrintMultTable (void)
{
    int i = 1;
    while (i <= 6)
    {
        PrintMultiples (i);
        i = i + 1;
    }
}
```

Der Prozess, den ich hier demonstriere, ist eine ziemlich verbreitete Entwicklungsstrategie. Wir entwickeln unser Programm schrittweise, indem wir Progammzeilen zu unserer `main()`-Funktion oder einer anderen Funktion hinzufügen. Wenn wir ein lauffähiges Programm erstellt haben, versuchen wir den Code zu extrahieren und in einer eigenen Funktion unterzubringen.

Nicht immer wissen wir schon vor der Erstellung unseres Programms genau, wie wir dieses in einzelne Module strukturieren können. Mit dem gerade vorgestellten Ansatz können wir die Struktur unseres Programms entwerfen, während wir programmieren. Natürlich können wir die Funktionen auch schon vorher festlegen, aber manchmal ist das einfach nicht möglich.

6.9 Lokale Variablen

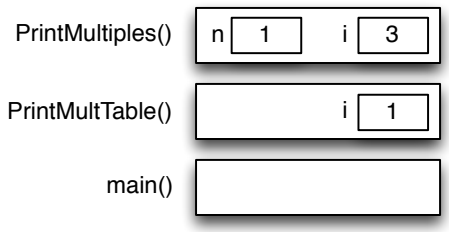
Haben Sie sich schon gefragt, wie es möglich ist, dass ich die gleiche Variable `i` in beiden Funktionen `PrintMultiples()` und `PrintMultTable()` verwenden kann?
Hatte ich nicht gesagt, das man eine Variable nur einmal deklarieren darf?
Und führt es nicht zu Problemen, wenn eine der Funktionen den Wert der Variablen verändert?

Die Antwort der letzten beiden Fragen lautet "nein," weil das `i` in `PrintMultiples()` und das `i` in `PrintMultTable()` *nicht die selbe Variable*

sind. Sie haben den selben Namen, aber sie verweisen nicht auf die gleiche Speicherstelle. Somit ist klar, dass wenn wir den Wert der einen Variable ändern, hat das keine Auswirkung auf den Wert der anderen Variablen.

Erinnern wir uns: Variablen, die innerhalb einer Funktion deklariert werden, sind so genannte *lokale Variablen*. Es ist nicht möglich auf eine lokale Variable von außerhalb ihrer "Heimatfunktion" zuzugreifen und wir können mehrere Variablen mit dem gleichen Namen haben, solange sie sich in unterschiedlichen Funktionen befinden.

Das Stackdiagramm für dieses Programm macht es völlig klar, dass die zwei Variablen mit Namen `i` an unterschiedlichen Stellen im Speicher liegen. Sie können unterschiedliche Werte besitzen und wenn wir eine der Variablen ändern hat das keine Auswirkungen auf die andere.



Der Wert des Parameters `n` in der Funktion `PrintMultiples()` ist dabei identisch mit dem Wert von `i` in `PrintMultTable()`. Der Wert von `i` in `PrintMultiples()` läuft von 1 bis 6. In unserem Diagramm, steht der Wert derzeit bei 3. Beim nächsten Durchlauf der Schleife wird der Wert 4 sein.

Es ist oft besser für unterschiedlichen Funktionen unterschiedliche Variablennamen zu verwenden, um Verwechslungen zu vermeiden. Wenn wir uns an die Richtlinien für die Verwendung von Variablen- und Funktionsnamen halten (siehe Anhang A.2) und *sprechende Bezeichner* wählen, sollten wir damit keine Probleme haben.

Es existieren aber auch gute Gründe dafür stets die gleichen Namen zu nutzen. So hat es sich zum Beispiel eingebürgert `i`, `j` und `k` für die Bezeichnung von Schleifenvariablen zu nutzen. Wenn jetzt unseren Funktionen andere Namen für Schleifenvariablen verwenden, kann es dazu führen, dass unsere Programme schwerer lesbar werden.

6.10 Noch mehr Verallgemeinerung

Wir können unser Programm aber noch weiter verallgemeinern. Stellen wir uns vor, wir brauchen ein Programm, welches eine Multiplikationstafel von beliebiger Größe und nicht nur im Format 6x6 ausgibt.

Um die Länge der Tabelle anzugeben, können wir einen Parameter zur Funktion `PrintMultTable()` hinzufügen:

```
void PrintMultTable (int high)
{
    int i = 1;
    while (i <= high)
    {
        PrintMultiples (i);
        i = i + 1;
    }
}
```

Ich haben den Wert 6 mit dem Parameter `high` ersetzt. Wenn ich jetzt `PrintMultTable()` mit dem Argument 7 aufrufe, erhalte ich:

```
1   2   3   4   5   6
2   4   6   8   10   12
3   6   9   12   15   18
4   8   12   16   20   24
5   10   15   20   25   30
6   12   18   24   30   36
7   14   21   28   35   42
```

Das sieht auf dem ersten Blick schon ganz gut aus, allerdings möchte ich, dass meine Multiplikationstafel ausbalanciert ist, die Anzahl der Spalten und Zeilen übereinstimmt. Das kann ich erreichen, indem ich einen weiteren Parameter zu `PrintMultiples()` hinzufüge, welcher angibt wie viele Spalten unsere Tabelle haben sollte.

Nur um lästig zu sein möchte ich noch einmal demonstrieren, dass auch die Parameter unterschiedlicher Funktionen den gleichen Namen besitzen dürfen (so wie lokale Variablen auch):

```
void PrintMultiples (int n, int high)
{
    int i = 1;
    while (i <= high)
    {
        printf ("%i    ", n*i);
        i = i + 1;
    }
    printf ("\n");
}

void PrintMultTable (int high)
{
    int i = 1;
    while (i <= high)
    {
        PrintMultiples (i, high);
```

```
        i = i + 1;
    }
}
```

Wenn ich einen neuen Parameter zu einer Funktion hinzufüge, muss ich die erste Zeile der Funktion anpassen und die Stelle in der Funktion ändern, wo dieser Parameter verwendet werden soll (in der Schleifenbedingung wird der Wert 6 durch `high` ersetzt). Weiterhin muss ich natürlich auch den Funktionsaufruf in `PrintMultTable()` anpassen.

Wie erwartet erzeugt unser Programm jetzt eine 7x7 Tabelle:

```
1   2   3   4   5   6   7
2   4   6   8   10   12   14
3   6   9   12   15   18   21
4   8   12   16   20   24   28
5   10   15   20   25   30   35
6   12   18   24   30   36   42
7   14   21   28   35   42   49
```

Wenn wir eine Funktion verallgemeinern, finden wir oft, dass die neue Funktion Eigenschaften aufweist, die wir so nicht unbedingt beabsichtigt haben.

Zum Beispiel ist unsere Multiplikationstafel symmetrisch, weil $ab = ba$. Daraus folgt, dass fast alle Einträge in der Tabelle doppelt auftreten. Wir könnten jetzt auf die Idee kommen, Druckkosten zu sparen, indem wir nur eine Hälfte der Tabelle drucken. Um das zu tun brauchen wir nur eine Zeile in `PrintMultTable()` zu ändern. Anstatt

```
PrintMultiples (i, high);
```

können wir

```
PrintMultiples (i, i);
```

schreiben und erhalten das folgende Ergebnis:

```
1
2   4
3   6   9
4   8   12   16
5   10   15   20   25
6   12   18   24   30   36
7   14   21   28   35   42   49
```

Ich überlasse es jetzt jedem selbst herauszufinden, wie diese Änderung funktioniert.
Allerdings geht uns durch die Änderung die ursprüngliche Definition der zweidimensionalen Tabelle verloren (siehe Abschnitt 6.5). Solche unbeabsichtigten Beeinflussungen kommen gar nicht so selten vor und man bezeichnet sie auch als das *Gesetz der unbeabsichtigten Folgen* (engl: *Law of Unintended Consequences*). Es ist daher wichtig vor der Programmierung klare Anforderungen an das Programm zu definieren und diese durch Tests sicher nachzuweisen.

6.11 Glossar

Schleife (engl: *loop*): Eine Anweisung oder Anweisungsblock, der mehrfach ausgeführt wird solange eine Bedingung *wahr* ist, oder bis irgendeine andere Bedingung erfüllt ist.

Endlosschleife (engl: *infinite loop*): Eine Schleife, deren Bedingung immer *wahr* ist.

Schleifenkörper (engl: *body*): Alle Anweisungen die innerhalb einer Schleife ausgeführt werden.

Schleifendurchlauf (engl: *iteration*): Ein Durchlauf der Anweisungen des Schleifenkörpers. Dies schließt die Auswertung der Bedingung der Schleife mit ein.

Tab (engl: *tab*): Ein spezielles Zeichen aus dem ASCII-Zeichensatz, geschrieben als `\t` in C, welches den Cursor an die nächste Tab-Stop Position auf der aktuellen Zeile bewegt.

Modularisierung (engl: *modularisation/encapsulation*): Die Zerlegung eines großen, komplexen Programms in einzelne, unabhängige Komponenten (wie z.B. Funktionen). Die Komponenten können durch die Verwendung lokaler Variablen voneinander isoliert werden.

Lokale Variable (engl: *local variable*): Eine Variable, welche innerhalb einer Funktion deklariert wird und die nur innerhalb dieser Funktion existiert. Auf lokale Variablen kann durch andere Funktionen nicht zugegriffen werden. Gleichnamige lokale Variablen in unterschiedlichen Funktionen sind voneinander unabhängig.

Verallgemeinern (engl: *generalize*): Beschreibt den Vorgang in dem wir eine spezifische Lösung die für ein eng umrissenes Problem zutrifft durch ein allgemeineres Konzept ersetzt (durch den Einsatz von Variablen oder Parameter), um damit eine ganze Klasse von Aufgabenstellungen zu bearbeiten. Die Verallgemeinerung erhöht die Nützlichkeit unseres Programms, weil es sich leichter weiterverwenden lässt und für einen breiteren Einsatzbereich verwendet werden kann.

Softwareentwicklungsprozess (engl: *software development process*): Die Vorgehensweise die es uns ermöglicht von einer Idee zu einem funktionsfähigen Programm zu gelangen. In diesem Kapitel habe ich einen Ansatz vorgestellt, bei dem man ausgehend von einfachen Programmen zur Lösung spezieller Problem durch Verallgemeinerung und Modularisierung zu umfassenden Lösungen gelangt.
Ein Teilbereich der Informatik, das Software Engineering befasst sich mit verschiedenen Methoden der Programmentwicklung.

6.12 Übungsaufgaben

Übung 6.1

```
void Loop(int n)
{
    int i = n;
    while (i > 1)
    {
        printf ("%i\n",i);
        if (i%2 == 0)
        {
            i = i/2;
        }
        else
        {
            i = i+1;
        }
    }
}

int main (void)
{
    Loop(10);
    return EXIT_SUCCESS;
}
```

a. Zeichnen Sie eine Tabelle welche die Werte der Variablen `i` und `n` während der Ausführung der Funktion `Loop()` zeigen. Die Tabelle sollte eine Spalte für jede Variable und eine Zeile für jede Iteration der `while`-Schleife enthalten.

b. Was gibt dieses Programm aus?

Übung 6.2

C stellt in der mathematischen Bibliothek die Funktion `pow()` zur Verfügung, welche die Potenz einer reellen Zahl berechnet.

Schreiben Sie Ihre eigene Version `Power()` dieser Funktion welche zwei Parameter: double `x` und integer `n` übernimmt und das Resultat der Berechnung x^n zurückliefert. Ihre Funktion soll die Berechnung iterativ (mit Hilfe einer Schleife) durchführen.

Übung 6.3

Angenommen Sie haben eine Zahl a, und Sie wollen die Quadratwurzel dieser Zahl ermitteln.

Eine mögliche Vorgehensweise besteht darin, dass Sie mit einer ersten groben Schätzung, x_0, der Antwort beginnen und diese Schätzung mit Hilfe der folgenden Formel verbessern:

$$x_1 = (x_0 + a/x_0)/2 \tag{6.1}$$

Zum Beispiel, suchen wir die Quadratwurzel von 9. Wir beginnen mit $x_0 = 6$, dann ergibt sich für $x_1 = (6+9/6)/2 = 15/4 = 3.75$, welches näher an der gesuchten Lösung liegt.

Wir können das Verfahren wiederholen indem wir x_1 benutzen um x_2 zu berechnen und so weiter... In diesem Fall ergibt sich $x_2 = 3.075$ und $x_3 = 3.00091$. Unsere Berechnung konvergiert sehr schnell hin zu der richtigen Antwort (3).

Schreiben Sie eine Funktion `SquareRoot` welche ein `double` als Parameter übernimmt und eine Näherung der Quadratwurzel des Parameters zurückliefert. Die Funktion soll dabei den oben beschriebenen Algorithmus benutzen und darf nicht die `sqrt()` Funktion der `math.h` Bibliothek verwenden.

Als erste, initiale Näherung sollten Sie $a/2$ verwenden. Ihre Funktion soll die Berechnung wiederholen, bis Sie zwei aufeinanderfolgende Näherungen erhalten, welche um weniger als 0.0001 voneinander abweichen: mit anderen Worten, bis der Absolutbetrag von $x_n - x_{n-1}$ geringer ist als 0.0001. Für die Berechnung des Absolutbetrags können Sie die `fabs()` Funktion der `math.h` Bibliothek verwenden.

Kapitel 7

Arrays

Ein **Array** steht für eine Menge von Werten, wobei jeder Wert durch eine Zahl (genannt Index) identifiziert und referenziert wird. Der Vorteil von Arrays besteht darin, dass wir eine (möglicherweise) sehr große Anzahl von Werten unter dem gleichen Namen ansprechen können.

Nehmen wir an, wir wollen in unserem Programm die Tagestemperaturen der letzten 10 Jahre auswerten – wenn wir die Werte in einzelnen Variablen speichern wollten, müssten wir dafür mindestens 3652 Variablen anlegen. Wenn wir Arrays benutzen brauchen wir nur eins.

Wenn wir ein Array deklarieren, müssen wir angeben wie viele Elemente in dem Array gespeichert werden sollen. Ansonsten sieht die Deklaration ähnlich wie für andere Variablentypen aus:

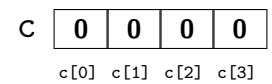
```
int c[4];
double values[10];
```

Syntaktisch sehen Array-Variablen wie andere C Variablen aus, außer dass ihnen mit `[ANZAHL_DER_ELEMENTE]` die Anzahl der Elemente des Arrays in eckigen Klammern folgt. Die erste Zeile in unserem Beispiel, `int c[4];` ist vom Typ "Array von Ganzen Zahlen" und erzeugt ein Array mit vier `int` Werten. Das Array hat den Namen `c`. Die zweite Zeile, `double values[10];` hat den Typ "Array von Fließkommazahlen" und erzeugt ein Array mit zehn Werten.

In C können wir die Elemente eines Array während der Deklaration sofort initialisieren. Die Werte der einzelnen Elemente werden dabei in geschwungenen Klammern `{}` und durch Komma getrennt angegeben:

```
int c[4] = {0, 0, 0, 0};
```

Diese Anweisung deklariert ein Array mit vier Elementen und initialisiert alle Werte des Arrays mit Null. Die folgende Abbildung zeigt wie Arrays in Zustandsdiagrammen dargestellt werden:



Die fett gedruckten Zahlen innerhalb der Kästchen sind die Werte der **Elemente** des Arrays. Die kleinen Zahlen außerhalb der Kästchen geben die Indizes an, über die wir auf die einzelnen Elemente des Arrays zugreifen können. Wenn wir ein neues Array deklarieren, ohne es zu initialisieren, dann enthalten die Elemente des Arrays beliebige, nicht vorher bestimmbare Werte. Wir müssen das Array mit sinnvollen Werten initialisieren, bevor wir sie verwenden können.

Wenn wir bei der Deklaration bereits alle Werte angeben, so können wir auf die Angabe der Größe des Arrays verzichten. Der Compiler ermittelt die benötigte Anzahl selbst:

```
int c[] = {0, 0, 0, 0};
```

Die angegebene Syntax für die Zuweisung von Werten an das Array ist nur gültig zum Zeitpunkt der Deklaration. Wenn wir später in unserem Programm die Werte des Arrays neu setzen wollen, müssen wir das einzeln für jedes Element des Arrays tun.

WICHTIG: Ein Array kann aus Elementen beliebigen Datentyps bestehen, wie zum Beipiel `int` und `double`. Es können in einem Array aber immer nur Elemente gleichen Typs gespeichert werden. Wir können die Typen innerhalb eines Arrays nicht mischen.

7.1 Inkrement und Dekrement-Operatoren

Einen Wert herauf- oder herunterzuzählen sind solche häufig vorkommenden Operationen in Programmen, dass C dafür spezielle Operatoren zur Verfügung stellt. Der `++` Operator zählt einen `int`, `char` oder `double` Wert um eins herauf (inkrement). Der `--` Operator subtrahiert (dekrementiert) den Wert um 1.

Es ist technisch legal eine Variable in einem Ausdruck zu verwenden und sie gleichzeitig zu inkrementieren. Ein Beispiel könnte folgendermaßen aussehen:

```
printf ("%i\n ", i++);
```

Wenn wir uns den Code anschauen, so ist nicht unmittelbar klar, ob der Wert um eins erhöht wird bevor oder nachdem er auf dem Bildschirm ausgegeben wird. Weil solche Ausdrücke verwirrend sein können, möchte ich von ihrer Verwendung abraten. Ich möchte Sie sogar noch mehr entmutigen, indem ich das Resultat nicht verraten werde. Wenn Sie es wirklich wissen wollen, können Sie es ja einfach selbst ausprobieren.

Mit Hilfe des Inkrement-Operators können wir die `PrintMultTable()`-Funktion aus Abschnitt Section 6.10 folgendermaßen schreiben:

```
void PrintMultTable(int high)
{
    int i = 1;
    while (i <= high)
    {
        PrintMultiples(i);
```

```
            i++;
        }
    }
```

Ein weit verbreiteter Anfängerfehler besteht darin folgenden Code zu schreiben:

```
index = index++;              /* FALSCH!! */
```

Unglücklicherweise ist diese Anweisung syntaktisch legal, so dass der Compiler uns nicht warnen wird. Der unangenehme Effekt dieser Anweisung besteht darin, dass der Wert von `index` unverändert gelassen wird. Das ist ein Fehler der in einem Programm möglicherweise nur schwer zu finden ist.

HINWEIS: Entweder schreiben wir `index = index + 1;` oder `index++;` Wir dürfen beide Ausdrücke nicht miteinander mischen!

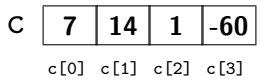
7.2 Zugriff auf Elemente eines Arrays

Der `[]` Operator erlaubt es uns einzelne Elemente eines Arrays zu lesen und zu schreiben. Dazu wird innerhalb der eckigen Klammern der Index des Elements angegeben. Hierbei müssen wir besonders aufpassen. Der Index zählt nämlich bei Null beginnend, das heißt, das erste Element im Array ist `c[0]`. Das Element `c[1]` ist bereits das zweite Element des Arrays!

Wir können den `[]` Operator in jedem beliebigen Ausdruck verwenden:

```
c[0] = 7;
c[1] = c[0] * 2;
c[2]++;
c[3] -= 60;
```

Alle diese Zuweisungen sind erlaubt. Hier ist das Ergebnis dargestellt:



Aus unserem Beispiel sollte klar geworden sein, dass die vier Elemente des Arrays über einen Indexwert identifiziert werden, der von 0 bis 3 reicht. In unserem Array gibt es kein Element mit dem Index 4.

Es ist trotzdem ein weit verbreiteter und häufiger Fehler die Grenzen eines Arrays zu überschreiten. Modernere Programmiersprachen, wie zum Beispiel Java oder C#, produzieren eine Fehlermeldung oder brechen das Programm ab, wenn versucht wird auf Elemente zuzugreifen, die außerhalb der Grenzen eines Arrays liegen. C hingegen überprüft die Grenzen eines Arrays nicht, so dass ein Programm auf Speicherstellen außerhalb des Arrays zugreifen kann, so als wären diese Elemente des Arrays. Sollte sich ihr Programm so verhalten, so ist das in den allermeisten Fällen falsch und kann zu folgenschweren Fehlern in dem Programm führen.

Es ist daher notwendig das Sie, als der Programmierer, darauf achten und sichergehen, dass der Programmcode ihres Programms die Grenzen der Arrays korrekt einhält!

Wir können beliebige Ausdrücke als Index benutzen, solange sie vom Typ `int` sind. Meistens verwenden wir eine spezielle *Schleifenvariable* um ein Array zu indizieren:

```
int i = 0;
while (i < 4)
{
    printf ("%i\n", c[i]);
    i++;
}
```

Wir benutzen die `while`-Schleife um den Wert der Schleifenvariable `i` schrittweise zu erhöhen. Wenn die Schleifenvariable `i` den Wert 4 erreicht, schlägt die Auswertung der Prüfbedingung fehl und die Schleife wird abgebrochen. Der Schleifenkörper wird also nur dann ausgeführt, wenn `i` den Wert 0, 1, 2 und 3 hat. Bei jedem Schleifendurchlauf benutzen wir `i` als Index für das Array und geben das `i`-te Element auf dem Bildschirm aus.

Wenn wir Arrays verwenden, werden wir also immer wieder auch mit Schleifen arbeiten, denn es kommt oft vor, dass ein Array von ersten bis zum letzten Element durchlaufen werden muss.

7.3 Kopieren von Arrays

Mit Arrays lassen sich eine Reihe von Programmieraufgaben sehr einfach lösen. Zum Beispiel können wir auf diese Weise sehr einfach große Datenmengen speichern und weiterverarbeiten.

Allerdings müssen wir uns daran gewöhnen, dass C sehr wenige Dinge automatisch von allein erledigt. Es ist zum Beispiel nicht möglich allen Elementen eines Arrays gleichzeitig einen neuen Wert zuzuweisen. Es ist auch nicht möglich die Werte eines Array direkt einem anderen Array zuzuweisen selbst wenn die Anzahl und der Typ der Elemente beider Arrays übereinstimmen:

```
double a[3] = {1.0, 1.0, 1.0};
double b[3];

a = 0.0;     /* Wrong! */
b = a;       /* Wrong! */
```

Um diese Aufgaben trotzdem auszuführen, müssen wir die Werte eines Arrays Element für Element bearbeiten. Das heißt, wir müssen jedem einzelnen Element eines Arrays einen neuen Wert zuweisen. Wenn wir den Inhalt eines Arrays in ein anderes Array kopieren wollen, müssen wir wieder elementweise die Werte kopieren und dazu verwenden wir am Besten eine Schleife:

```
int i = 0;
while (i < 3)
{
    b[i] = a[i];
    i++;
}
```

7.4 for Schleifen

Die Schleifen, die wir bisher benutzt haben, weisen einige Gemeinsamkeiten auf. Vor der Ausführung der Schleife wird eine Schleifenvariable initialisiert, danach wird eine Schleifenbedingung getestet, welche von der Schleifenvariable abhängt. Im Schleifenkörper wird dann die Schleifenvariable verändert, indem man sie zum Beispiel heraufzählt (inkrementiert).

Dieser Schleifentyp ist so verbreitet dass es dafür eine eigene Schleifenanweisung gibt, mit der man diesen Ablauf kürzer und klarer darstellen kann: die `for`-Schleife.
Die Syntax der `for`-Schleife sieht folgendermaßen aus:

```
for (INITIALIZER; CONDITION; INCREMENTOR)
{
    BODY;
}
```

Diese Anweisung ist äquivalent zu:

```
INITIALIZER;
while (CONDITION)
{
    BODY;
    INCREMENTOR;
}
```

Ich finde die `for`-Schleife übersichtlicher, weil jetzt alle Anweisungen, die für die Steuerung der Schleife nötig sind, in einer Anweisung zusammengefasst sind. Allerdings haben gerade Anfänger teilweise Probleme die Arbeitsweise der `for`-Schleife zu verstehen. Es ist wichtig zu wissen, wann die einzelnen Ausdrücke der Schleifenanweisung ausgeführt werden!

Der INITIALIZER Ausdruck wird in der Schleife nur ***ein einziges Mal*** ganz am Anfang ausgeführt und damit die Schleifenvariable initialisiert.

Die Schleifenbedingung CONDITION wird ***vor jedem Durchlauf*** durch die Schleife geprüft. Wenn die Bedingung *wahr* ist, wird die Schleife durchlaufen, wenn sie *falsch* ist, wird die Schleife abgebrochen. Da die Bedingung geprüft wird, bevor die Schleife ein erstes Mal ausgeführt wird, kann es passieren, dass die Schleife überhaupt nicht ausgeführt wird, weil bereits vor der ersten Ausführung der Test der Schleifenbedingung fehlschlägt.

Der INCREMENTOR Ausdruck wird ***nach jedem Durchlauf*** durch den Schleifenkörper einmal ausgeführt. In den meisten Fällen wird dann die Schleifenvariable heraufgezählt (inkrementiert). Es ist aber genauso gut möglich eine Schleife 'rückwärts' zu durchlaufen. In diesem Fall müssten wir die Schleifenvariable herabzählen.

So ist zum Beispiel:

```
int i;
for (i = 0; i < 4; i++)
{
    printf("%i\n", c[i]);
}
```

äquivalent zu:

```
int i = 0;
while (i < 4)
{
    printf("%i\n", c[i]);
    i++;
}
```

7.5 Die Länge eines Arrays

C bietet uns keinen bequemen Weg an, mit dessen Hilfe wir die Größe eines Arrays in unserem Programm ermitteln können. Die Länge des Arrays ist zum Beispiel dann wichtig, wenn wir mit Hilfe einer Schleife das Array elementweise durchgehen und die Schleife nach dem letzten Element beenden wollen.

Um die Länge des Arrays zu ermitteln könnten wir den `sizeof()` Operator benutzen. Dieser liefert uns allerdings die Größe der Datentypen in Bytes. Die meisten Datentypen in C benutzen mehrere Bytes um einen Wert zu speichern und je nach nach verwendeter Rechnerarchitektur können diese Werte sogar für den gleichen Datentyp unterschiedlich sein.

Um jetzt die korrekte Größe, das heißt die Anzahl der Elemente, eines Arrays zu ermitteln müssen wir die ermittelte Arraygröße noch durch die Anzahl der Bytes teilen die ein einzelnes Element eines Arrays belegt. Dazu benutzen wir sinnvollerweise das Element mit dem Index 0.

```
sizeof(ARRAY)/sizeof(ARRAY_ELEMENT)
```

Es ist eine gute Idee diesen Wert und keine Konstante als die obere Grenze für die Schleifenvariable zu benutzen. Auf diese Weise können wir sicherstellen, dass wenn sich irgendwann einmal die Größe eines Arrays ändern sollte, wir nicht das gesamte Programm durchforsten müssen um alle Schleifen zu ändern. Die Schleife für jede Arraygröße korrekt arbeiten:

```
int i, length;
length = sizeof (c) / sizeof (c[0]);
```

```
for (i = 0; i < length; i++)
{
    printf("%i\n", c[i]);
}
```

Besonders wichtig ist die korrekte Angabe der Schleifenbedingung. Wir erinnern uns, der Index der Elemente eines Arrays beginnt mit `0`. In unserer Schleife wird die Schleifenvariable `i` so lange erhöht, bis sie den Wert `length - 1` hat. Dies entspricht genau dem Index des letzten Element des Arrays. Anschließend wird `i` ein weiteres Mal erhöht und hat damit den gleichen Wert wie `length`. Die Bedingung unserer Schleife ist damit *falsch* und die Schleife wird abgebrochen. Würde die Schleife an dieser Stelle weiter machen, würde das Programm auf Speicherbereiche zugegreifen die nicht mehr Teil des Arrays sind und unser Programm wäre fehlerhaft.

7.6 Zufallszahlen

Die meisten Computerprogramme verhalten sich bei jeder Ausführung des Programms stets gleich. Man nennt dieses Verhalten **deterministisch**. Normalerweise ist das deterministische Verhalten eine gute Sache, denn gleiche Berechnungen sollen auch immer gleiche Ergebnisse liefern. Es gibt aber auch einige Anwendungsgebiete wo sich das Verhalten des Computers nicht vorhersagen lassen soll, zum Beispiel bei Computerspielen.

Es ist ziemlich schwierig ein Computerprogramm dazu zu bringen *echte* Zufallswerte zu erzeugen. Es gibt aber eine Möglichkeit es so aussehen lassen, als würde unser Programm zufällige Werte erzeugen. Der Computer kann so genannte **pseudozufällige Zahlen** (engl: pseudorandom numbers) erzeugen und im Programmablauf verwenden. Pseudozufällig Zahlen sind im mathematischen Sinne nicht wirklich zufällig, sie haben aber ähnliche Eigenschaften und können für viele Anwendungen anstelle von echten Zufallszahlen verwendet werden.

C stellt eine Funktion mit dem Namen `rand()` bereit, welche pseudozufällige Zahlen erzeugt. Die Funktion ist in der `stdlib.h` -Bibliothek definiert. Diese Bibliothek stellt eine Vielzahl von Standardfunktionen bereit und wird deshalb als *Standardbibliothek* (engl: standard library) bezeichnet.

Der Rückgabewert der `rand()` -Funktion ist eine ganze Zahl (`int`) zwischen `0` und `RAND_MAX`, wobei `RAND_MAX` eine große Zahl ist ($\approx$ 2 Milliarden auf meinem Computer) welche in der gleichen Headerdatei definiert ist. Jedes mal, wenn wir `rand()` aufrufen, berechnet die Funktion eine andere pseudozufällige Zahl. Um ein Beispiel zu sehen können wir die folgende Schleife ausführen:

```
for (i = 0; i < 4; i++)
{
    int x = rand();
    printf("%i\n", x);
}
```

Auf meinem Computer wird die folgende Ausgabe erzeugt:

```
1804289383
846930886
1681692777
1714636915
```

Wenn Sie das Programm ausführen, sollte das so ähnlich aussehen, es werden ihnen aber wahrscheinlich andere Zahlen angezeigt.

Natürlich wollen wir nicht immer mit gigantisch großen Zahlen arbeiten. Meistens benötigen wir Zufallszahlen zwischen `0` und einer oberen Grenze. Eine einfache Möglichkeit solche Zahlen zu erzeugen besteht darin den Modulo-Operator zu verwenden:

```
int x = rand ();
int y = x % upperBound;
```

So ist `y` der ganzzahlige Divisionsrest wenn `x` durch `upperBound` geteilt wird. Die möglichen Werte für `y` liegen damit zwischen 0 und `upperBound - 1`, inklusive beider Grenzwerte. Wir müssen beachten, dass `y` niemals den Wert von `upperBound` erreichen kann.

Manchmal benötigen wir zufällige Fließkommazahlen in unserem Programm. Wir können diese erzeugen, indem wir das Ergebnis von `rand()` durch `RAND_MAX` teilen und vorher eine Typumwandlung (engl: cast) für einen der Operanden durchführen:

```
int x = rand ();
double y = (double) x / RAND_MAX;
```

Diese Programmzeilen weisen `y` einen zufälligen Wert im Bereich zwischen 0.0 und 1.0 zu. Die Werte 0.0 und 1.0 sind in diesen Bereich eingeschlossen.

AUFGABE: Überlegen Sie sich eine Möglichkeit, wie man zufällige Fließkommazahlen in einem beliebigen Bereich erzeugen kann. Erstellen Sie ein Programm welches Zufallszahlen zwischen 100.0 und 200.0 erzeugt.

7.7 Statistiken

Die Zahlen die wir mit `rand()` erzeugen sollten eigentlich gleichverteilt sein. Das heißt, die Wahrscheinlichkeit mit der ein Wert aus dem Wertebereich gezogen wird, ist für alle Werte gleich. Wenn wir also das Vorkommen eines jedes möglichen Wertes zählen, sollten wir annähernd die gleiche Anzahl für alle Werte erhalten unter der Voraussetzung, dass wir eine sehr große Anzahl von Werten untersuchen.

In den nächsten Abschnitten werden wir ein Programm entwickeln, welches eine Folge von Zufallszahlen erzeugt und überprüft, ob diese Eigenschaft gegeben ist.

7.8 Arrays mit Zufallszahlen

Der erste Schritt besteht darin eine große Anzahl von Zufallszahlen zu erzeugen und diese in einem Array zu speichern. Ich werde dazu erst einmal mit der "großen Zahl" von 20 Zufallszahlen beginnen. Es ist eigentlich immer eine gute Idee mit einer überschaubaren Anzahl von Werten zu starten. Das macht es einfacher das Programm zu durchschauen und mögliche Fehler zu finden. Wir können dann später die Anzahl sehr einfach weiter erhöhen.

Die folgende Funktion hat die Aufgabe ein Array von `int`s mit zufälligen Werten im Bereich von 0 bis `upperBound-1` zu füllen. Die Funktion besitzt 3 Parameter, ein Array von ganzen Zahlen (int), die Größe des Arrays und einen oberen Grenzwert (upperBound) für den Wertebereich der Zufallszahlen.

```
void RandomizeArray (int array[], int length, int upperBound)
{
    int i;
    for (i = 0; i < length; i++)
    {
        array[i] = rand() % upperBound;
    }
}
```

Der Rückgabewert der Funktion ist `void`, was bedeutet, dass die Funktion keinen Wert an die aufrufende Funktion zurückgibt. Um diese Funktion zu testen, ist es bequem eine weitere Funktion zu schreiben, welche den Inhalt eines Arrays auf dem Bildschirm ausgibt:

```
void PrintArray (int array[], int length)
{
    int i;
    for (i = 0; i < length; i++)
    {
        printf ("%i ",  array[i]);
    }
    printf ("\n");
}
```

Die folgenden Programmzeilen erzeugen ein Array welches mit zufälligen Werten gefüllt wird und geben den Inhalt des Arrays auf dem Bildschirm aus:

```
int r_array[20];
int upperBound = 10;
int length = sizeof(r_array) / sizeof(r_array[0]);

RandomizeArray (r_array, length, upperBound);
PrintArray (r_array, length);
```

Auf meinem Computer sieht die Ausgabe folgendermaßen aus

```
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6
```

und scheint auf den ersten Eindruck schon ziemlich zufällig zu sein.

Wenn diese Zahlen wirklich zufällig verteilt sind, dann erwarten wir, dass jede Zahl gleich häufig auftritt. In der ermittelten Folge kommt aber die Zahl 6 fünf mal vor, die Zahl 4 und 8 hingegen überhaupt nicht.

Heißt das jetzt, das unsere Zahlenfolge keine wirklich Zufallsfolge ist? Mit so einer kleinen Stichprobe lässt sich das schwer sagen. Nur für eine sehr große Anzahl von Versuchen können wir davon ausgehen, die erwartete Gleichverteilung der Werte auch feststellen zu können.

Um unsere Theorie zu testen wollen wir daher einige Programme schreiben, die das Vorkommen jedes Werts zählen, so dass wir überprüfen können was passiert, wenn wir die Anzahl der Elemente erhöhen.

7.9 Ein Array an eine Funktion übergeben

Schauen wir uns zuerst aber die `RandomizeArray()` Funktion noch einmal etwas genauer an. Etwas an dieser Funktion ist ungewöhnlich: Wir übergeben ein Array an die Funktion und auf irgend eine Art und Weise ist es der Funktion gelungen das Array mit zufälligen Werten zu füllen ohne das die Funktion einen Wert zurückgibt, der Rückgabewert der Funktion ist nämlich `void`.

Dieses Verhalten widerspricht allem, was ich bisher über die Verwendung von Variablen in Funktionen gesagt habe. C benutzt für die Argumente einer Funktion die sogenannte **call-by-value** Auswertung des angegebenen Ausdrucks. Das heißt, es wird der Wert des Ausdruck in der aufrufenden Funktion ermittelt und anschließend in die Parametervariable der aufgerufenen Funktion kopiert. Der selbe Vorgang läuft in der umgekehrten Richtung ab, wenn eine Funktion einen Wert zurückgibt. Veränderungen in den internen Variablen der aufgerufenen Funktion haben keine Auswirkungen auf die externen Wert der aufrufenden Funktion.

Wenn wir allerdings ein Array an eine Funktion übergeben, so lässt sich nur schwer das gesamte Array als ein Wert an die aufgerufene Funktion übergeben. Dazu müsste das gesamte Array aus der aufrufenden Funktion an die aufgerufene Funktion kopiert und am Ende der Funktion vielleicht auch wieder zurückkopiert werden.

C übergibt deshalb nur einen Verweis (engl: reference) auf die Speicherstelle des Arrays an die Funktion. Die aufgerufene Funktion kann dann mit Hilfe des Verweises auf das originale Array zugreifen und dort direkt alle notwendigen Änderungen vornehmen. Dieses Verhalten ist auch der Grund dafür, dass unsere Funktion keinen Rückgabewerte hat. Die Änderungen an den Daten wurden ja bereits vorgenommen und müssen nicht noch einmal gespeichert werden. Eine solche Art der Übergabe der Funktionsargumente nennt man **call-by-reference**.

Call-by-reference macht es notwendig, dass wir der aufgerufenen Funktion die Länge des Arrays explizit mitteilen müssen. Wenn wir nämlich den `sizeof` Operator in der aufgerufenen Funktion benutzen um die Arraygröße zu ermitteln

werden wir feststellen, dass dieser uns nur die Größe des Verweises liefert. Die aufgerufene Funktion hat keine Information darüber wie unser Array in der aufrufenden Funktion definiert wurde.

Wir werden die Aufrufstrategien **call-by-reference** und **call-by-value** im Kapitel 8.7, Kapitel 9.6 und 9.7 noch genauer diskutieren.

7.10 Zählen der Elemente eines Arrays

In unserem aktuellen Progammbeispiel wollen wir eine möglicherweise sehr große Menge von Elementen durchsuchen und dabei zählen, wie oft ein bestimmter Wert darin vorkommt. Dieses Programm entspricht einem allgemeinem Muster, welches man "Durchsuchen und Zählen" nennen kann. Das Musters hat folgende Teilschritte:

- Es existiert eine Menge von Daten, zum Beispiel ein Array, die von Anfang bis Ende durchsucht werden kann.
- Es gibt einen Test der auf jedes Element der zu durchsuchenden Menge angewandt wird.
- Ein Zähler registriert wie viele Elemente den Test bestehen oder nicht bestehen.

Für unseren Anwendungsfall stelle ich mir eine Funktion mit dem Namen `HowMany()` vor, die die Anzahl der Elemente in einem Array ermittelt die mit einem vorgegebenen Wert übereinstimmen. Die Parameter der Funktion sind das Array, die Länge des Arrays und der gesuchte Wert. Der Rückgabewert der Funktion ist die Anzahl des Vorkommens des gesuchten Werts im Array.

```
int HowMany (int array[], int length, int value)
{
    int i;
    int count = 0;

    for (i=0; i < length; i++)
        {
            if (array[i] == value) count++;
        }
    return count;
}
```

Ein praktikables Vorgehen für die Lösung von solchen Programmierproblemen besteht darin sich einfache Funktionen auszudenken die eine bestimmte Teilaufgabe erledigen und einfach zu schreiben und zu verstehen sind. Anschließend nutzen wir dann mehrere dieser Funktionen um ein komplexeres Problem zu lösen. Diese Vorgehensweise wird auch **Bottom-up Entwurf** genannt.

Natürlich ist es nicht einfach immer schon im voraus zu wissen, welche Art von Funktionen wir für die Lösung unseres Problems benötigen und es ist nicht

immer offensichtlich welche Teilfunktionen einfach zu schreiben sind. Je mehr Erfahrung wir aber in der Programmierung bekommen um so leichter wird es uns fallen. Ein guter Ansatz besteht darin nach Teilproblemen zu suchen die ein bestimmtes Muster aufweisen, welches auch für andere Arten von Problemen interessant und hilfreich sein kann.

7.11 Überprüfung aller möglichen Werte

`HowMany()` zählt nur das Auftreten eines bestimmten Werts in einem Array. Wir wollen aber herausfinden, wie oft jeder mögliche Wert in dem Array vorkommt. Dazu können wir eine Schleife einsetzen:

```
int i;
int r_array[20];
int upperBound = 10;
int length = sizeof(r_array) / sizeof(r_array[0]);

RandomizeArray(r_array, length, upperBound);

printf ("value\tHowMany\n");
for (i = 0; i < upperBound; i++)
{
    printf("%i\t%i\n", i, HowMany(r_array, length, i));
}
```

Diese Programmzeilen benutzen die Schleifenvariable als ein Argument von `HowMany()` um die Anzahl aller Werte zwischen 0 und 9 in dem Array zu ermitteln, mit folgendem Resultat:

```
value   HowMany
0       2
1       1
2       3
3       3
4       0
5       2
6       5
7       2
8       0
9       2
```

In dem Beispiel ist es immer noch schwierig zu ermitteln, ob die Ziffern wirklich gleichverteilt sind. Wir erhöhen daher die Anzahl der Werte in unserem Array auf 100.000 und führen damit den Test durch:

```
value   HowMany
0       10130
1       10072
2       9990
```

```
3       9842
4       10174
5       9930
6       10059
7       9954
8       9891
9       9958
```

Jetzt lässt sich feststellen, dass die Schwankungen der Häufigkeit jede Wert circa 1% des erwarteten Werts (10,000) betragen und unsere Zufallszahlen sehr wahrscheinlich gleichverteilt sind.

7.12 Ein Histogramm

Es ist of nützlich die Tabellen aus dem vorigen Beispiel nicht nur auf dem Bildschirm auszugeben sondern diese im Computer für spätere Aufgaben vorzuhalten. Dafür müssen wir also 10 ganzzahlige Werte speichern.

Wir könnten jetzt 10 Variablen vom Typ `int` erstellen und ihnen Namen geben wie `howManyOnes`, `howManyTwos` und so weiter. Das würde eine Menge Tipparbeit abgeben und es wäre ziemlich umständlich wollten wir später zum Beispiel den Bereich der zu überprüfenden Zahlenwerte ändern.

Eine viel bessere Lösung besteht darin ein weiteres Array mit 10 Elementen zu benutzen. So können wir alle zehn Speicherplätze mit einem Mal erstellen und wir können per Index bequem auf die jeweilige Speicherstelle zugreifen ohne mit zehn verschiedenen Namen hantieren zu müssen. Hier ist das Ergebnis:

```
#define UPPER_BOUND 10
int i;
int r_array[100000];
int histogram[UPPER_BOUND];
int length = sizeof(r_array) / sizeof(r_array[0]);

RandomizeArray(r_array, length, UPPER_BOUND);

for (i = 0; i < UPPER_BOUND; i++)
{
    int count = HowMany(r_array, length, i);
    histogram[i] = count;
}
```

Ich habe das Array **histogram** genannt, weil das die Bezeichnung für eine statistische Häufigkeitsverteilung ist. In einem Histogramm werden Daten in Klassen eingeteilt und es wird erfasst, wie häufig die Ereignisse in jeder Klasse sind.

In unserem Programm ist ein kleiner Trick enthalten, denn ich benutze die Schleifenvariable für zwei verschiedene Zwecke. Zuerst wird sie als Argument der `HowMany()` Funktion verwendet um den Wert anzugeben für den wir uns

interessieren. Weiterhin verwende ich die Schleifenvariable als Index das Element des Histogramms um anzugeben, wo das Ergebnis der Funktion gespeichert werden soll.

7.13 Eine optimierte Lösung

Die vorgestellte Lösung funktioniert, allerdings ist sie nicht besonders effizient. Jedes Mal, wenn wir die Funktion `HowMany()` aufrufen durchsucht diese das gesamte Array von Anfang bis zum Ende. In unserem Beispiel müssen wir das Array zehn mal durchsuchen!

Unser Programm würde schneller arbeiten, wenn wir das Array nur ein einziges Mal durchsuchen müssten. Wir müssten nur einfach für jeden gefundenen Wert im Array den entsprechenden Zähler im Histogramm finden und inkrementieren. Da unser Histogramm die Verteilung der Werte von 0 bis 9 darstellt, können wir die gefundenen Werte unseres Arrays direkt als Index für das Histogramm benutzen, wir müssen nur vorher alle Elemente des Histogramms auf den Wert `0` setzen:

```
#define UPPER_BOUND 10
int i;
int r_array[100000];
int histogram[UPPER_BOUND] = {0};
int length = sizeof(r_array) / sizeof(r_array[0]);

RandomizeArray(r_array, length, UPPER_BOUND);

for (i = 0; i < length; i++)
{
    int index = r_array[i];
    histogram[index]++;
}
```

Wir initialisieren `histogram`, indem wir dem ersten Element den Wert `0` zuweisen. Alle nachfolgenden Elemente für die wir keinen Wert angegeben haben werden danach automatisch auf `0` gesetzt. Jetzt können wir den (`++`) -Operator innerhalb der Schleife dazu benutzen, das Vorkommen der Zufallszahlen zu zählen. Es wird oft vergessen, dass ein Zähler initialisiert werden muss, bevor er benutzt werden kann.

AUFGABE: Erstellen Sie aus den Programmzeilen des Beispiels eine eigene Funktion mit dem Namen `Histogram()`. Die Funktion soll mindestens folgende Parameter besitzen: ein Array mit den zu durchsuchenden Werten, den Suchbereich (in diesem Fall 0 bis 9) als zwei Parameter `min` und `max`, und ein zweites Array, welches groß genug ist das Histogram der Werte des ersten Arrays zu speichern.

7.14 Zufällige Startwerte

Wenn wir die Programme aus diesem Kapitel mehrmals hintereinander ausführen fällt auf, dass wir bei jedem Programmstart die gleichen 'zufälligen' Zahlenwerte angezeigt bekommen. Das ist nicht sehr zufällig!

Eine Eigenschaft der pseudozufälligen Zahlen besteht darin, dass der Algorithmus für die Erzeugung der Zufallsfolge immer wieder die gleichen Werte generiert, wenn er mit dem gleichen **Startwert** beginnt. Da wir keinen besonderen Startwert angegeben haben, erzeugt der Zufallszahlengenerator bei jedem Programmstart immer wieder die gleiche Zahlenfolge.

Während wir unser Programm entwickeln ist dieses Verhalten oft hilfreich, weil wir auf diese Weise die Ausgaben des Programms vor und nach einer Änderung miteinander vergleichen können. In vielen Anwendungen, wie zum Beispiel Computerspielen, ist es allerdings vorteilhaft, wenn unser Programm bei jedem Programmstart eine unterschiedliche Zufallssequenz verwendet.

Um dieses Verhalten zu erreichen, müssen wir einen unterschiedlichen Startwert für unsere Zufallsfunktion angeben. Wir können dafür die `srand()` Funktion benutzen. Wir müssen der Funktion ein einziges Argument mitgeben, welches eine ganze Zahl zwischen 0 und `RAND_MAX` sein muss.

Wo bekommen wir aber diesen einzigartigen Startwert her? Ein verbreiteter Weg für die Erzeugung eines Startwerts besteht darin, die Bibliotheksfunktion `time()` zu benutzen. Die Funktion liefert einen Wert für die aktuelle Zeit auf dem Computersystem. Die genaue Interpretation dieses Werts ist für uns momentan nicht wichtig. Wir benutzen den Wert nur, um unseren Zufallszahlengenerator mit einem veränderlichen Wert zu starten und können dafür den Befehl

```
srand((unsigned) time(NULL));
```

verwenden.

7.15 Glossar

Array (engl: *array*): Eine Ansammlung von mehreren Werten gleichen Typs unter einem gemeinsamen Namen. Die einzelnen Werte in einem Array werden als Elemente bezeichnet und lasst sich über einen Index identifizieren. Ein Array ist ein Beispiel für eine Datenstruktur.

Element (engl: *element*): Ein einzelner Wert in einem Array. Mit Hilfe des `[]` -Operators kann man auf die einzelnen Elemente eines Arrays zugreifen.

Index (engl: *index*): Eine ganzzahliger Wert oder Variable die dazu benutzt wird auf ein bestimmtes Element in einem Array zuzugreifen.

deterministisch (engl: *deterministic*): In einem Computerprogramm folgt auf eine Anweisung unter gleichen Voraussetzungen immer die gleiche

nächste Anweisung und es wird für jeden Durchlauf ein reproduzierbares Ergebnis erzeugt.

pseudo-zufällig (engl: *pseudorandom*): Eine Zahlenfolge die für einen außenstehenden Betrachter nicht von einer wirklich zufälligen Zahlenfolge unterschieden werden kann, die aber durch deterministische Berechnungen erzeugt wurde.

Startwert (engl: *seed*): Ein Wert der dazu benutzt wird den Algorithmus für die Erzeugung der Pseudozufallsfolge zu iniitialisieren. Wenn der gleiche Startwert verwendet wird produziert der Algorithmus die gleiche Folge von pseudo-zufälligen Werten.

Inkrementieren/Heraufzählen (engl: *increment*): Den Wert einer Variablen um 1 erhöhen. In C kann dazu der `++` Operator benutzt werden.

Dekrementieren/Herunterzählen (engl: *decrement*): Den Wert einer Variablen um 1 verringern. In C kann dazu der `--` Operator benutzt werden.

Bottom-up Entwurf (engl: *bottom-up design*): Eine Method der Softwareentwicklung bei der ausgehend von kleinen, nützlichen Funktionen durch deren Kombination ein komplexeres Programm mit vergrößertem Funktionsumfang erstellt wird. Die alternative zum Bottom-up Entwurf ist der Top-down Entwurf bei dem zuerst ein Gesamtkonzept erstellt wird, ohne die Details der programmtechnischen Umsetzung bereits zu kennen.

Histogramm (engl: *histogram*): Die Erfassung der Häufigkeitsverteilung klassifizierbarer Merkmale. Ein Histogramm kann in C als Array von ganzen Zahlen dargestellt werden, bei der in den Elementen des Arrays die Häufigkeit des Auftretens eines bestimmten Merkmals gezählt wird.

7.16 Übungsaufgaben

Übung 7.1

Schreiben Sie eine Funktion namens `CheckFactors(int, int[], int)` mit 3 Parametern. Die Funktion übernimmt einen Integerwert `n`, ein Array von Integerwerten sowie die Länge des Arrays `len` als drittes Argument.

Die Funktion soll `TRUE` zurückliefern, falls alle Zahlen in dem übergebenen Array Faktoren von `n` sind (d.h. `n` durch alle diese Zahlen teilbar ist). Für den Fall, dass mindestens eines der Array-Elemente kein Faktor von `n` ist soll `FALSE` zurückgegeben werden.

HINWEIS: Ermitteln Sie vor dem Aufruf der Funktion `CheckFactors()` die Länge des Arrays in der `main()` Funktion, siehe dazu 7.5. Vergleichen Sie ebenfalls die Lösung der Übungsaufgabe 5.2.

Übung 7.2

Schreiben Sie eine Funktion `void SetToZero(int[], int)` welche ein Array von `int` und die Länge dieses Arrays übernimmt und anschließend dieses Array für alle Elemente auf den Wert `0` initialisiert.

Für die Ermittlung der Länge des Arrays können Sie die Funktion aus dem Abschnitt 7.5 übernehmen. Testen Sie die korrekte Implementierung dieser Funktion mit Hilfe der `PrintArray()` Funktion aus Abschnitt 7.8.

Übung 7.3

Schreiben Sie eine Funktion welche ein Array von `int`, die Länge des Arrays `len` und einen `int` mit dem Namen `target` als Argumente übernimmt. Die Funktion soll das Array durchsuchen und den Index zurückliefert an dem `target` zum ersten Mal in dem Array auftritt. Sollte `target` nicht in dem Array enthalten sein soll -1 zurückgegeben werden.

Übung 7.4

Seit der Erfindung der Computer werden diese genutzt um Arrays mit Daten zu sortieren. Unzählige Algorithmen wurden entworfen und hinsichtlich ihrer Effizienz verglichen.

Eine nicht-besonders-effizienter Algorithmus läuft folgendermaßen ab: Finde das größte Element im Array und tausche es mit dem ersten Element. Finde das zweit-größte Element im Array und tausche es mit dem zweiten Element, und so weiter...

a. Schreiben Sie eine Funktion `IndexOfMaxInRange()`, welche ein Array von ganzen Zahlen (integers) übernimmt und das größte Element in einem bestimmten Bereich (range) des Arrays findet und seine Position als *index* zurückliefert.
b. Schreiben Sie eine Funktion `SwapElement()`, welche ein Array von ganzen Zahlen und zwei Indexe übernimmt und anschließend die Werte der Elemente an den gegebenen Indexen vertauscht.
c. Schreiben Sie eine Funktion `SortArray()`, welche ein Array von ganzen Zahlen übernimmt und die Funktionen `IndexOfMaxInRange()` und `SwapElement()` benutzt um das Array vom größten zum kleinsten Wert zu sortieren.

Kapitel 8

Strings and things

8.1 Darstellung von Zeichenketten

In den vorangegangenen Kapiteln haben wir vier Arten von Daten kennengelernt — Zeichen, Ganzzahlen, Gleitkommazahlen und Zeichenketten (Strings) — aber nur drei Datentypen für Variablen: `char`, `int` und `double`. Bis jetzt haben wir keine Möglichkeit kennengelernt eine Zeichenkette in einer Variable zu speichern oder Zeichenketten in unserem Programm zu manipulieren.

In diesem Kapitel soll dieser Misstand behoben werden und ich kann jetzt das Geheimnis lüften was das Besondere der Strings ist. Strings werden in C als Arrays von Zeichen gespeichert. Dabei wird das Ende der Zeichenkette in dem Array durch ein besonderes Zeichen (das Zeichen `'\0'`) markiert, was dazu führt, dass das `char`-Array immer mindestens 1 Zeichen länger sein muss als die zu speichernde Zeichenkette.

Mittlerweile können wir mit dieser Erklärung etwas anfangen und es wird klar, dass wir erst ein ganze Menge über die Funktionsweise der Sprache lernen mussten, bevor wir unsere Aufmerksamkeit den Strings und Stringvariablen zuwenden konnten.

Im vorigen Kapitel haben wir gesehen, dass Operationen über Arrays nur minimale Unterstützung durch die Programmiersprache C erhalten und wir die erweiterten Funktionen für den Umgang mit Arrays selbst programmieren mussten. Glücklicherweise liegen die Dinge etwas besser, wenn es um die Manipulation dieser speziellen `char`-Arrays geht, die wir zur Darstellung von Zeichenketten nutzen. Es existiert eine Anzahl von Bibliotheksfunktionen in `string.h` welche uns die Handhabung und Verarbeitung von Strings etwas einfacher machen, als die Verarbeitung von reinen Arrays.

Trotzdem muss man feststellen, dass die Stringverarbeitung in C um einiges komplizierter und mühsamer ist als in anderen Programmiersprachen. Die sorgfältige Beachtung dieser Besonderheiten ist notwendig um die Verarbeitung von

Zeichenketten nicht zu einer potentiellen Fehlerquelle in unseren Programmen werden zu lassen.

8.2 Stringvariablen

Wir können eine Stringvariable als ein Array von Zeichen in der folgenden Art erzeugen:

```
char first[] = "Hello, ";
char second[] = "world.";
```

Die erste Zeile erzeugt eine Stringvariable `first` und weist ihr den Wert `"Hello, "` zu. In der zweiten Zeile deklarieren wir eine zweite Stringvariable. Erinnern wir uns, die gleichzeitige Deklaration und Zuweisung eines Wertes zu einer Variablen wird als Initialisierung bezeichnet.

Nur zum Zeitpunkt der Initialisierung können wir dem String einen Wert direkt zuweisen (so wie bei Arrays im Allgemeinen). Die Initialisierungsparameter werden in der Form einer Stringkonstanten in Anführungszeichen (`"..."`) angegeben. Wie wir zu einem späteren Zeitpunkt der Stringvariablen einen neuen Wert zuweisen können, erfahren wir erst im Abschnitt 8.10.

Auffallend ist der Unterschied in der Syntax der Initialisierung von Arrays und Strings. Wir könnten den String auch in der normalen Arraysyntax initialisieren:

```
char first[] = {'H','e','l','l','o',',',' ','\0'};
```

Es ist allerdings viel bequemer einen String als Stringkonstante zu schreiben. Das sieht nicht nur natürlicher aus, sondern ist auch sehr viel einfacher einzugeben.

Wenn wir die Stringvariable direkt zum Zeitpunkt der Deklaration auch gleich initialisieren, ist es normalerweise nicht notwendig, die Größe des Arrays mit anzugeben. Der Compiler ermittelt die notwendige Größe des Arrays anhand der angegebenen Stringkonstante. Wir können allerdings die Größe der Stringvariable auch selbst definieren. Ich erkläre gleich, was wir dabei beachten müssen.

Erinnern wir uns, was ich über den Aufbau eines Strings gesagt habe. Es ist ein Array vom Typ `char` in dem ein Begrenzungszeichen das Ende der Zeichenkette markiert: das Begrenzungszeichen `'\0'`. Dieses Zeichen darf nicht mit dem Zeichen `0` verwechselt werden. Das Zeichen `0` wird in der ASCII-Tabelle (siehe Anhang B) mit dem Wert 48 kodiert. Bei dem Zeichen `'\0'` handelt es sich um den Wert 0 in der Tabelle.

Normalerweise müssen wir das Begrenzungszeichen nicht selbst mit angeben. Der Compiler versteht unseren Programmcode und fügt diese Zeichen automatisch ein. Allerdings haben wir in dem vorigen Beispiel einen String genau wie ein Array behandelt und in diesem Fall müssen wir uns selbst um das Einfügen des Begrenzungszeichen kümmern.

Wenn wir eine Stringvariable benutzen, um während des Programmablaufs unterschiedlich große Strings zu speichern, müssen wir bei der Definition des Arrays

die Größenangabe so wählen, dass auch genügend Platz für die längste Zeichenkette reserviert wird. Wir müssen weiterhin beachten, dass auch Platz für das Begrenzungszeichen übrig bleibt, dass heißt, unser Array muss exakt ein Zeichen länger sein, als die größte zu speichernde Zeichenfolge.

Wir können Strings auf dem Bildschirm ausgeben, in dem wir den Variablennamen an die `printf()` Funktion übergeben. Dazu verwenden wir den Formatierungsparameter `%s`:

```
printf("%s", first);
```

8.3 Einzelne Zeichen herauslösen

Zeichenketten werden *Strings* genannt, weil sie aus einer Folge (engl: string) von Zeichen bestehen. Die erste Aufgabe die wir lösen wollen, besteht darin aus einer Zeichenkette ein bestimmtes Zeichen herauszulösen. Da wir wissen dass die Zeichenkette in C als Array gespeichert ist können wir einen Index in eckigen Klammern (`[` und `]`) für diese Operation benutzen:

```
char fruit[] = "banana";
char letter = fruit[1];
printf ("%c\n", letter);
```

Der Ausdruck `fruit[1]` gibt an, dass ich das Zeichen mit der Indexnummer 1 aus dem String namens `fruit` ermitteln möchte. Das Resultat wird in einer `char`-Variable namens `letter` gespeichert. Wenn wir uns anschließend den Wert der Variable `letter` anzeigen lassen, so sollte es nicht überraschen, dass dabei der folgende Buchstabe auf dem Bildschirm ausgegeben wird:

```
a
```

`a` ist nicht der erste Buchstabe von `"banana"`. Wie wir bereits im letzten Kapitel besprochen haben, nummerieren Informatiker die Elemente eines Arrays immer beginnend mit 0. Das gilt auch für Stringvariablen. Der erste Buchstabe von `"banana"` ist `b` und hat den Index 0. Der zweite Buchstabe `a` den Index 1 und der Buchstabe mit dem Index 2 ist das `n`.

Wenn wir also den ersten Buchstaben eines Strings ermitteln wollen müssen wir die Null als Index in eckige Klammern setzen:

```
char letter = fruit[0];
```

8.4 Die Länge eines Strings ermitteln

Um die Länge eines Strings herauszufinden (die Anzahl der Zeichen die dieser String enthält), können wir die Funktion `strlen()` nutzen. Die Funktion wird aufgerufen indem wir den String als Argument benutzen:

```
#include <string.h>
int main(void)
{
```

```
    int length;
    char fruit[] = "banana";

    length = strlen(fruit);
    return EXIT_SUCCESS;
}
```

Der Rückgabewert von `strlen()` ist in diesem Fall 6. Wir weisen diesen Wert der Variablen `length` zu, um ihn später weiter zu nutzen.

Um dieses Programm zu kompilieren, müssen wir die `string.h` Bibliothek in unser Programm einbinden. Diese Bibliothek stellt eine große Anzahl von nützlichen Funktionen für den Umgang und die Bearbeitung von Zeichenketten bereit. Es ist sinnvoll, sich mit diesen Funktionen vertraut zu machen, weil sie uns helfen können unsere Programmieraufgaben einfacher und schneller zu lösen.

Um das letzte Zeichen in einem String zu finden mag es naheliegen die folgenden Anweisungen einzugeben:

```
int length = strlen(fruit);
char last = fruit[length];       /* WRONG!! */
```

Das funktioniert leider nicht. Der Grund dafür liegt wieder darin, dass `fruit` ein Array darstellt und einfach kein Buchstabe unter dem Index `fruit[6]` in `"banana"` gespeichert ist. Wir erinnern uns: Der Index eines Arrays wird beginnend von 0 gezählt und die 6 Buchstaben tragen deshalb die Indexnummern 0 bis 5. Um den letzten Buchstaben zu erhalten müssen wir den Wert von `length` um 1 verringern:

```
int length = strlen(fruit);
char last = fruit[length-1];
```

8.5 Zeichenweises Durchgehen

Eine häufig vorkommende Aufgabe besteht darin einen String zeichenweise durchzugehen. Wir wählen das ersten Zeichen eines Strings aus, führen irgendwelche Operationen durch und wiederholen dieses Vorgehen, bis wir beim letzten Zeichen des Strings angekommen sind.

Wir können diese Aufgabe sehr einfach mit Hilfe einer `for` Schleife erledigen:

```
int index;
for (index = 0; index < strlen(fruit); index++)
{
    char letter = fruit[index];
    printf("%i. Buchstabe: %c\n" , index+1, letter);
}
```

Der Name unserer Schleifenvariablen ist `index`. Ein **Index** ist eine Variable oder ein Wert der ein bestimmtes Element einer geordneten Menge identifiziert — in unserem Fall der Menge von Zeichen in einem String. Der Index gibt dabei an, um welches Zeichen der Menge es sich dabei handelt. Die Menge muss geordnet

sein, so dass jedem Buchstaben des Strings ein Index und jedem Index genau ein Buchstabe eineindeutig zugewiesen ist.

Die Schleife geht den String zeichenweise durch und gibt jeden Buchstaben auf einer eigenen Bildschirmzeile aus. Um die natürliche Zählweise einzuhalten, geben wir den Wert des Index erhöht um 1 aus (das hat keinen Einfluss auf den Wert von `index`).

Beachtenswert ist weiterhin die Formulierung der Bedingung unser Schleife `index < strlen(fruit)`. Wenn der Wert von `index` gleich der Länge des Strings ist, wird die Bedingung falsch und der Schleifenkörper verlassen. Das letzte Zeichen, welches wir ausgeben hat somit den Index `strlen(fruit)-1`.

AUFGABE: Schreiben Sie eine Funktion welche einen `string` als Argument übernimmt und dann alle Buchstaben des Strings auf einer Bildschirmzeile rückwärts ausgibt.

8.6 Ein Zeichen in einem String finden

Wenn wir in einem String nach einem bestimmten Zeichen suchen, müssen wir die gesamte Zeichenkette durchgehen und die Position ermitteln, an der sich das Zeichen befindet.

Hier ist eine mögliche Implementation dieser Funktion:

```
int LocateCharacter(char *s, char c)
{
    int i = 0;
    while (i < strlen(s))
    {
        if (s[i] == c) return i;
        i = i + 1;
    }
    return -1;
}
```

Wenn wir diese Funktion aufrufen, übergeben wir der Funktion den zu durchsuchenden String als erstes, und das zu suchende Zeichen als zweites Argument.

Die Funktion gibt dann die erste ermittelte Position des Zeichens zurück. Es kann allerdings auch vorkommen, dass das zu suchende Zeichen gar nicht in dem String enthalten ist. Für diesen Fall ist es sinnvoll einen Wert zurückzugeben, der normalerweise nicht vorkommt und einen Fehlerfall signalisiert. Wurde das Zeichen nicht gefunden so gibt die Funktion den Wert `-1` an die aufrufende Funktion zurück.

8.7 Pointer und Adressen

Wenn wir uns die Definition der `LocateCharacter()` Funktion anschauen werden wir feststellen, dass die Angabe des Parameters `char *s` ungewöhnlich aus-

sieht.

Erinnern Sie sich noch daran, wie wir in Kapitel 7.9 darüber gesprochen haben wie wir ein Array an eine Funktion übergeben? Ich sagte, dass anstelle einer Kopie des Arrays ein Verweis auf das Array an die Funktion übergeben wird. Ich habe aber nicht genau erklärt, worum es sich bei dem Verweis genau handelt. Das werde ich jetzt nachholen.

C ist eine der wenigen High-level Programmiersprachen welche die direkte Manipulation von Datenobjekten im Speicher des Computers unterstützt. Um den direkten Zugriff auf Speicherobjekten durchzuführen, müssen wir die Position der Objekte im Speicher kennen: ihre Adresse. Genau wie andere Daten auch, können wir diese Adressen in Variablen speichern und an Funktionen übergeben, denn so eine Adresse stellt ja selbst wieder einen Wert dar. Adressen werden in Variablen eines speziellen Typs gespeichert. Diese Variablen zeigen auf andere Objekte im Speicher, wie zum Beispiel Integer-Werte, Arrays, Strings, usw. und werden deshalb **Pointer**-Variablen genannt.

Ein Pointer (manchmal auch als Zeiger bezeichnet) verweist auf die Stelle im Speicher des Computers, wo sich das jeweilige Objekt befindet. Wenn wir einen Pointer definieren, müssen wir auch immer angeben welchen Typ das Objekt besitzt, auf das verwiesen wird. Die Definition einer Pointervariablen auf ein ganzzahliges Speicherobjekt vom Typ `int` kann folgendermaßen vorgenommen werden:

```
int *i_pointer;
```

Diese Deklaration sieht ähnlich aus wie andere Variablendeklarationen, mit einem Unterschied - dem Sternchen vor dem Variablennamen. Damit geben wir dem Compiler bekannt, dass es sich bei `i_pointer` um eine Pointervariable handelt.

Anders als bei normalen Variablen bezieht sich der angegebene Typ `int` nicht auf den Pointer selbst, sondern definiert auf welche Art von Objekten dieser Pointer zeigen soll (in diesem Fall auf Objekte vom Typ `integer`). Das ist notwendig, damit mit der Compiler weiß, wie er das Speicherobjekt behandeln soll, auf das der Pointer verweist. Der Pointer selbst ist normalerweise nur eine Adresse im Speicher des Computers.

Ein Pointer nur für sich allein hat keine sinnvolle Bedeutung. Wir benötigen immer auch ein Objekt auf das der Pointer verweist:

```
int number = 5;
int *i_pointer;
```

Diese Programmzeilen definieren eine `int` Variable und einen Pointer. Momentan besteht noch keinerlei Verbindung zwischen diesen beiden Variablen.

8.8 Adress- und Indirektionsoperator

Um eine Verbindung zwischen einem Speicherobjekt und einem Pointer herzustellen müssen wir den **Adressoperator** `&` benutzen. Damit können wir die

Speicherstelle der Variablen `number` ermitteln und dem Pointer zuweisen. Man sagt: "der Pointer `i_pointer` zeigt auf `number`":

```
i_pointer = &number;
```

Ab jetzt können wir auf den Wert der Variable `number` auch über den Pointer zugreifen. Dazu benötigen wir den **Indirektionsoperator** `*`. Damit greifen wir auf das Objekt zu, auf das unser Pointer zeigt:

```
printf("%i\n", *i_pointer);
```

Diese Programmzeile gibt `5` auf dem Bildschirm aus, welches dem Wert der Variable `number` entspricht. Bei der Verwendung des Indirektionsoperators `*` müssen wir sehr gut aufpassen. Wir dürfen niemals den Indirektionsoperator mit einem Pointer verwenden der noch nicht initialisiert wurde:

```
int *z_pointer;
printf("%i\n", *z_pointer);    /*** WRONG ***/
```

Das Gleiche gilt wenn wir über einem Pointer dem Speicherobjekt einen neuen Wert zuweisen wollen und der Pointer nicht initialisiert wurde. Hierbei kann es zu Programmfehlern oder sogar zum Programmabsturz kommen:

```
int *z_pointer;
*z_pointer = 1;                /*** WRONG ***/
```

Beim Lesen von Programmen müssen wir aufpassen, dass wir den vorangestellten Indirektionsoperators `*` nicht mit der Deklaration von Pointern verwechseln. Auch bei der Deklaration verwenden wir einen `*`, der aber nur dazu dient die Variable als Pointervariable kenntlich zu machen. Leider ergibt sich daraus für Pointervariablen der Umstand, dass eine direkte Initialisierung der Pointervariablen eine andere Syntax aufweist, als eine spätere Zuweisung im Programm:

```
int number = 5;
int *i_pointer = &number; /* Initialisierung des Pointers */
i_pointer = &number;      /* gleicher Effekt!  */
```

Mit Pointern können wir also direkt auf Speicherstellen zugreifen und diese natürlich auch verändern:

```
*i_pointer = *i_pointer + 2;
printf("%i\n", number);
```

Unsere Variable `number` hat jetzt den Wert `7` und wir fangen so langsam an zu verstehen, wie es der `LocateCharacter()` Funktion möglich ist auf die Werte unserer Stringvariablen mittels eines `char` -Pointers zuzugreifen.

Viele C Programme benutzen Pointer und wir haben gerade einmal die Oberfläche des Themas angekratzt. Durch die Verwendung von Pointern können bestimmte Probleme sehr effizient gelöst werden, allerdings kann der direkte und unkontrollierte Zugriff auf den Computerspeicher auch zu schwerwiegenden Programmfehlern führen, wenn nicht sehr sorgfältig gearbeitet wird, oder die Verwendung von Pointern nicht komplett verstanden wurde. Der Compiler hat in diesen Fällen keine Möglichkeit das richtige Programmverhalten zu prüfen. Aus diesem Grund untersagen viele andere Programmiersprachen die direkte Manipulation von Speicherstellen über Pointer.

8.9 Verkettung von Strings

Im Abschnitt 8.6 können wir sehen, wie man eine Suchfunktion implementiert die ein `character` in einem `string` findet.

Eine nützliche Operation für Strings ist deren **Verkettung**. Darunter verstehen wir die Verknüpfung des Endes eines Strings mit dem Anfang des nächsten Strings. So wird zum Beispiel aus: `auf` und `passen` wird `aufpassen`.

Glücklicherweise müssen wir nicht alle diese nützlichen Funktion selbst schreiben. Die `string.h` Bibliothek stellt verschiedene Funktionen bereit die wir für die Bearbeitung von Strings nutzen können.

Um zwei Strings miteinander zu verketten können wir die `strncat()` Funktion benutzen:

```
char fruit[20] = "banana";
char bakedGood[] = " nut bread";
strncat(fruit, bakedGood, 10);
printf ("%s\n", fruit);
```

Diese Programmzeilen erzeugen die Ausgabe: `banana nut bread`.

Wenn wir Bibliotheksfunktionen benutzen ist es sehr wichtig, dass wir alle notwendigen Argumente der Funktion kennen und ein Grundverständnis der Arbeitsweise der Funktion besitzen.

Man könnte jetzt annehmen, dass die `strncat()` Funktion zwei Strings nimmt, sie zusammenfügt und einen neuen, zusammengesetzten String erzeugt. So ist es aber nicht. Die Funktion kopiert nämlich den Inhalt des zweiten Strings an das Ende des ersten Strings.

Dieser kleine Unterschied hat für uns als Programmierer wichtige Konsequenzen. Wir selbst müssen dafür sorgen, dass die erste Stringvariable auch lang genug ist um auch den zweiten String mit aufzunehmen. Aus diesem Grund habe ich die erste Stringvariable `fruit` als Array von 20 Zeichen definiert `char fruit[20]`. Dieses Array bietet Platz für 19 Zeichen + 1 Begrenzungszeichen. Das dritte Argument von `strncat()` gibt an, wie viele Zeichen aus dem zweiten in den ersten String kopiert werden. Dabei ist darauf zu achten, dass nur so viele Zeichen in das erste Array kopiert werden wie dort hineinpassen. Weiterhin muss noch Platz für das Begrenzungszeichen bleiben. Werden mehr Zeichen kopiert können Speicherbereiche überschrieben werden.

8.10 Zuweisung von neuen Werten an Stringvariablen

Bisher haben wir gesehen, wie eine Stringvariable während der Deklaration initialisiert wird. Da es sich bei Strings im Grunde um ein mehr oder weniger normales Array handelt gilt auch hier, dass wir nicht den Zuweisungsoperator benutzen können um dem String einen neuen Wert zuzuweisen.

```
fruit = "apple"; /* Wrong: Direkte Zuweisung nicht möglich! */
```

Wir können allerdings die `strncpy()` Funktion benutzen um dem String einen neuen Wert zuzuweisen:

```
char greeting[15];
strncpy (greeting, "Hello, world!", 13);
```

`strncpy()` kopiert 13 Zeichen aus dem zweiten Argument-String in den ersten Argument-String und somit haben wir der Stringvariable einen neuen Wert zugewiesen.

Allerdings müssen wir dabei wieder einige Einschränkungen beachten. Die `strncpy()` Funktion kopiert genau 13 Zeichen aus dem zweiten String in den ersten String. Was passiert dabei aber mit dem Begrenzungszeichen `'\0'`?

Die Funktion setzt das Begrenzungszeichen **nicht** automatisch. Wir könnten unsere Kopieranweisung so ändern, dass 14 statt 13 Zeichen kopiert werden. In diesem Fall wird das unsichtbare Begrenzungszeichen einfach mitkopiert und unser String `greeting` wäre wieder korrekt:

```
strncpy (greeting, "Hello, world!", 14);
```

oder wir benutzen die `strlen()` Funktion um `strlen() + 1` Zeichen zu kopieren:

```
strncpy (greeting, "Hello, world!", strlen("Hello, world!")+1);
```

Wenn wir allerdings nur einen Teil des zweiten Strings in den ersten String kopieren wollen, so müssen wir selbst dafür sorgen, dass das Zeichen n+1 im String `greeting[15]` hinterher auf den Wert `\0` gesetzt wird:

```
strncpy (greeting, "Hello, world!", 5); /*kopiert nur Hello */
greeting[5] = '\0';         /*das 6. Zeichen hat den Index 5*/
```

Achtung! In den letzten beiden Abschnitten haben wir die `strncpy()` und die `strncat()` Funktion benutzt, bei der wir immer genau angeben müssen, wie viele Zeichen aus dem zweiten in den ersten String kopiert werden oder an diesen angehangen werden.

Die `string.h` Bibliothek stellt uns noch weitere Funktionen zur Verfügung die so ähnlich arbeiten. Es gibt zum Beispiel auch noch die `strcpy()` und die `strcat()` Funktion. Bei diesen beiden Funktionen wird die Anzahl der zu kopierenden Zeichen nicht mit angegeben. Die Funktionen kopieren so lange Zeichen aus dem zweiten in den ersten String, bis in dem zweiten String ein Begrenzungszeichen gefunden wird.

Von der Benutzung dieser Funktionen wird dringend abgeraten! Die Benutzung dieser Funktionen hat zu einer großen Anzahl von Sicherheitsproblemen in C Programmen geführt. C überprüft keine Array-Grenzen und wenn in dem zweiten String das Begrenzungszeichen zum Beispiel wegen einer fehlerhaften Benutzereingabe fehlt, so kann es leicht vorkommen, dass Zeichen über den Speicherbereich der ersten Stringvariable hinaus in den Speicher geschrieben werden und dadurch andere Daten überschrieben werden.

8.11 Strings sind nicht direkt vergleichbar

Unsere Vergleichsoperatoren die wir für den Vergleich von `int`s und `double`s benutzt haben können wir nicht auf Strings anwenden. So könnten wir auf die Idee kommen die folgenden Programmzeilen zu schreiben um zwei Strings miteinander zu vergleichen:

```
if (word == "banana")  /* Wrong! */
```

Dieser Test schlägt leider jedes Mal fehl.

Wir können aber die `strcmp()` Funktion benutzen um zwei Strings miteinander zu vergleichen. Die Funktion hat einen Rückgabewert von `0` wenn die zwei Strings identisch sind, einen negativen Wert, wenn der erste String 'alphabetisch kleiner' ist als der zweite (wenn er in einem Wörterbuch vor dem anderen String gelistet würde) oder einen positiven Wert, wenn der zweite String 'größer' ist.

Bitte beachten Sie, dass der Rückgabewert nicht der normalen Interpretation der Wahrheitswerte normaler Vergleichsoperatoren entspricht, wobei der Rückgabewert `0` als 'Falsch' interpretiert wird.

Mit der `strcmp()` Funktion können wir sehr einfach beliebige Wörter in ihre alphabetische Reihenfolge bringen:

```
if (strcmp(word, "banana") < 0)
{
    printf( "Your word, %s, comes before banana.\n", word);
}
else if (strcmp(word, "banana") > 0)
{
    printf( "Your word, %s, comes after banana.\n", word);
}
else
{
    printf ("Yes, we have no bananas!\n");
}
```

Dabei müssen wir aber wieder aufpassen. Die `strcmp()` Funktion behandelt Großbuchstaben und Kleinbuchstaben anders als wir das normalerweise gewöhnt sind. Alle Großbuchstaben kommen bei einem Vergleich vor den Kleinbuchstaben. Der Grund dafür liegt in der Art der verwendenden Zeichenkodierung (siehe Anhang: ASCII-Tabelle). Dort haben Großbuchstaben einen kleineren Wert als Kleinbuchstaben und produzieren das folgende Ergebnis:

```
Your word, Zebra, comes before banana.
```

Ein oft genutzter Ausweg aus diesem Dilemma besteht darin Strings in einer einheitlichen Formatierung zu vergleichen. Zuerst werden alle Zeichen der Strings zu Klein- oder Großbuchstaben gewandelt und erst danach wird der Vergleich durchgeführt. Der nächste Abschnitt zeigt wie das gemacht wird.

8.12 Klassifizierung von Zeichen

Es ist of nützlich den Wert, der in einer `char` -Variable gespeichert ist, im Programm untersuchen zu können und zu entscheiden, ob es sich dabei um einen Groß- oder Kleinbuchstaben, oder um eine Ziffer oder ein Sonderzeichen handelt. C stellt auch dafür wieder eine Bibliothek von unterschiedlichen Funktionen bereit, welche eine solche Klassifizierung von Zeichen ermöglichen. Um diese Bibliotheksfunktionen in unserem Programm nutzen zu können, müssen wir die Header-Datei `ctype.h` in unser Programm aufnehmen.

```
char letter = 'a';
if (isalpha(letter))
{
    printf("The character %c is a letter.", letter);
}
```

Der Rückgabewert der `isalpha()` Funktion ist ein `int` und besitzt den Wert `0` wenn das Funktionsargument kein Buchstabe ist und einen von Null verschiedenen Wert, wenn die Funktion mit einem Buchstaben aufgerufen wurde.

Wir können daher die Funktion direkt als Bedingung der `if` -Anweisung einsetzen, wie in unserem Beispiel zu sehen. Der Rückgabewert `0` wird als `false` interpretiert und lässt die Bedingung fehlschlagen. Alle anderen Rückgabewerte werden als `true` interpretiert.

Weitere Funktionen zur Klassifizierung von Zeichen sind `isdigit()`, welche dazu dient die Ziffern 0 bis 9 zu identifizieren, `isspace()` identifiziert nicht druckbare Zeichen wie zum Beispiel Leerzeichen, Tabulatoren, Zeilenumbrüche. Es gibt außerdem die `isupper()` und `islower()` Funktion, welche zwischen Buchstaben in Groß- und Kleinschreibung unterscheidet.

Schließlich gibt es auch zwei Funktionen, welche sich zur Umwandlung zwischen Groß- und Kleinschreibung nutzen lassen. Sie heißen `toupper()` und `tolower()`. Beide Funktionen erwarten ein einzelnes Zeichen als Argument und geben ein (möglicherweise umgewandeltes) Zeichen zurück.

```
char letter = 'a';
letter = toupper (letter);
printf("%c\n", letter);
```

Die Ausgabe dieser Programmzeilen ist `A`.

Aufgabe: Benutzen Sie die Funktionen der `ctype.h` Bibliothek um zwei Funktionen mit den Namen `StringToUpper()` und `StringToLower()` zu schreiben. Beide Funktionen sollen einen einzelnen Parameter vom Typ String besitzen und diesen String so modifizieren, dass im Anschluss alle Zeichen des Strings Groß- oder Kleinbuchstaben sind. Zeichen die keine Buchstaben sind, sollen nicht verändert werden. Der Rückgabewert der Funktionen soll `void` sein.

8.13 Benutzereingaben im Programm

Die meisten Programme die wir bisher geschrieben haben, verhalten sich sehr berechenbar. Sie führen bei jedem Programmstart die selben Anweisungen aus. Das liegt unter anderem daran, dass wir es bisher vermieden haben Eingaben vom Benutzer entgegenzunehmen und darauf zu reagieren.

Es gibt eine Vielzahl von Möglichkeiten mit einem Computer in Interaktion zu treten. Dazu zählen Tastatureingaben, Mausbewegungen, Sensoren sowie exotischere Mechanismen wie Sprachsteuerung und Iris-Scanning. In diesem Buch werden wir uns ausschließlich mit Tastatureingaben beschäftigen.

Für die Eingabe von Werten über die Tastatur stellt C die Funktion `scanf()` bereit, welche Benutzereingaben ähnlich behandelt wie `printf()` Programmausgaben auf dem Bildschirm darstellt. Wir können die folgenden Programmzeilen dazu benutzen um den Benutzer einen ganzzahligen Wert einzugeben:

```
int x;
printf("Please enter a number: ");
scanf("%i", &x);
```

Die `scanf()` Funktion hält die Ausführung des Programms an und wartet darauf, dass der Benutzer eine Eingabe über die Tastatur des Computers macht. Wenn der Benutzer einen gültigen ganzzahligen Wert eingegeben hat wird dieser von der Funktion in in der Variable `x` gespeichert.

Was passiert, wenn der Benutzer etwas anderes als eine ganze Zahl über die Tastatur eingibt? C gibt keine Fehlermeldung aus oder macht sonst einen Versuch das Problem zu beheben. Die `scanf()` Funktion bricht dann einfach ab und lässt den Wert von `x` unverändert.

Glücklicherweise gibt es einen Weg um herauszufinden, ob die Eingabe funktioniert hat, oder nicht. Die `scanf()` Funktion hat einen Rückgabewert, der angibt, wie viele Elemente erfolgreich eingelesen wurden (die Funktion kann auch mehrere Werte gleichzeitig einlesen). In unserem Beispiel erwarten wir einen Rückgabewert von `1` wenn erfolgreich eine ganze Zahl eingelesen werden konnte. Wenn wir einen anderen Rückgabewert als `1` erhalten, so wissen wir, dass die Eingabeoperation nicht erfolgreich war und wir nicht einfach so im Programm weitermachen können.

Das folgende Programm fragt nach einer Benutzereingabe und überprüft, ob wirklich eine Zahl eingelesen werden konnte:

```
int main (void)
{
    int success, x;

    /* prompt the user for input */
    printf ("Enter an integer: \n");

    /* get input */
    success = scanf("%i", &x);
```

```
    /* check and see if the input statement succeeded */
    if (success == 1)
    {
        /* print the value we got from the user */
        printf ("Your input: %i\n", x);
        return EXIT_SUCCESS;
    }
    printf("That was not an integer.\n");
    return EXIT_FAILURE;
}
```

8.14 Tastaturpuffer leeren

Es gibt noch eine weitere Sache die wir bei der Verwendung der `scanf()` Funktion beachten müssen. Wenn wir eine bestimmte Benutzereingabe für die Ausführung unseres Programms benötigen, könnte man auf die Idee kommen den Code zu ändern, um den Benutzer immer wieder nach einer Eingabe zu fragen, bis diese vom Programm akzeptiert wird:

```
if (success != 1)
{
    while (success != 1)
    {
         printf("That was not a number. Please try again:\n");
         success = scanf("%i", &x);
    }
 }
```

Unglücklicherweise führt dieses Programm in eine Endlosschleife wenn der Benutzer etwas anderes als eine Zahl eingibt. Sie fragen sich jetzt sicher, warum?

Die Ein- und Ausgabefunktionen unseres Programms schreiben nicht selbst Zeichen auf den Bildschirm oder beobachten wie der Benutzer einzelne Tasten drückt. Sie benutzen dazu Funktionen des jeweiligen Betriebssystems des Computers. Die Tastatureingaben des Benutzers werden unserem Programm vom Betriebssystem in einem so genannten Tastatur- oder Eingabepuffer (engl: *input buffer*) bereitgestellt.

Die `scanf()` Funktion liest die eingegebenen Zeichen aus dem Tastaturpuffer und wenn alles gut geht wird anschließend der Puffer gelöscht. Wenn allerdings, wie in unserem Beispiel, die `scanf()` Funktion keinen gültigen Wert lesen kann, dann wird der Puffer nicht geleert. Bei jedem neuen Aufruf von `scanf()` lesen wir also den zuvor eingegebenen Wert aus dem noch gefüllten Puffer und natürlich schlägt damit die Überprüfung sofort fehl - ist das Problem klar?

Wir müssen daher nach einer missglückten Eingabe den Eingabepuffer löschen um ihn für die erneute Tastatureingabe des Benutzers bereit zu machen. Leider gibt es genau dafür keine Standardfunktion in C, das heißt, wir müssen uns

selbst eine schreiben. Ich benutze dazu die `getchar()` Funktion die einzelne Zeichen aus dem Puffer holen kann. Das mache ich so lange, bis entweder ein Zeilenumbruch erkannt wird (`'\n'`), oder der Tastaturpuffer leer ist (*EOF - End-Of-File*). Die Funktion wird direkt in der Bedingung einer `while` -Schleife so oft aufgerufen bis keine Zeichen mehr im Eingabepuffer vorhanden sind. Danach sind wir bereit den Benutzer erneut um seine Eingabe zu bitten:

```
 if (success != 1)
{
    while (success != 1)
    {
        char ch;   /* helper variable stores discarded chars*/
        printf("That isn't a number. Please try again:\n");

        /* now we empty the input buffer*/
        while ((ch = getchar()) != '\n' && ch != EOF);
        success = scanf("%i", &x);
    }
}
```

Mit der `scanf()` Funktion können wir auch Strings einlesen:

```
char name[80];
printf ("What is your name?");
scanf ("%79s", name);
printf ("%s", name);
```

Hierbei ist es wieder sehr wichtig, dass wir dafür sorgen, dass unsere Stringvariable groß genug ist um die komplette Benutzereingabe aufzunehmen. Da wir nicht wissen wie viele Zeichen der Benutzer wirklich auf der Tastatur eingibt, beschränken wir die Anzahl der Zeichen die von der Funktion aus dem Eingabepuffer gelesen werden auf 79, so dass noch Platz für das Begrenzungszeichen des Strings bleibt.

Auffällig ist der Unterschied der Schreibweise des Arguments der `scanf()` Funktion, je nachdem ob wir eine ganze Zahl oder einen String einlesen. Das Funktionsargument ist keine Wert sondern ein Pointer auf eine Speicherstelle. Schließlich soll die Funktion ja die Tastatureingabe direkt in einer Variablen speichern. Wenn wir eine ganze Zahl (`int`) einlesen, müssen wir deshalb der Funktion die Adresse der Variable mitteilen, wo der eingelesene Wert gespeichert werden soll. Dazu benutzen wir den Adressoperator `&` mit dem Variablenname. Wenn wir einen String einlesen, müssen wir nur den Namen der Stringvariable selbst angeben. Wir erinnern uns: Arrays wurden per *call-by-reference* an Funktionen übergeben!

Ein letzter Hinweis gilt der Arbeitsweise der `scanf()` Funktion. Sollte sich in der einzulesenden Zeichenfolge des Eingabepuffers ein oder mehrere Leerzeichen befinden, so wird an dieser Stelle die Einleseoperation für das aktuelle Element beendet. Wenn Sie zum Beispiel im vorigen Beispiel Vor- und Nachname eingegeben haben, so wird nur das erste Wort dem String `s` zugewiesen. Der Rest

der Eingabezeile wird im Puffer gespeichert und würde von der nächsten Eingabeanweisung ausgewertet werden. Wenn Sie also das Programm wie angegeben ausführen, würde immer nur ihr Vorname ausgegeben werden.

8.15 Glossar

String (engl: *string*): Eine Variable in der eine Zeichenkette gespeichert ist. In C werden Strings in Arrays vom Typ `char` gespeichert. Das Ende der Zeichenkette wird durch den Wert `'\0'` markiert.

Zähler (engl: *counter*): Eine Variable, die genutzt wird um etwas zu zählen (z.B. die Anzahl der Schleifendurchläufe). Eine Zählervariable wird normalerweise mit 0 initialisiert und dann heraufgezählt (inkrementiert).

Verketten (engl: *concatenate*): Eine Funktion über Zeichenketten, bei der die eine Zeichenkette an eine andere angefügt wird.

Pointer (Zeiger) (engl: *pointer*): Ein Verweis auf ein Datenobjekt im Speicher. Pointer sind Variablen in denen als Wert die Adresse eines anderen Datenobjekts gespeichert ist.

Adresse (engl: *address*): Die genaue Speicherstelle eines Datenobjekts im Hauptspeicher des Computers.

8.16 Übungsaufgaben

Übung 8.1

Schreiben Sie eine Funktion `LetterHist()`, welche einen String als Parameter übernimmt und Ihnen ein Histogramm der Buchstaben in diesem String liefert.

Das 'nullte' Element des Histogramms soll die Anzahl der `a`'s (gemeinsam für Groß- und Kleinschreibung) in dem String enthalten. Das 25. Element die Anzahl der `z`'s

Zusatzaufgabe: Ihre Lösung soll den String nur genau einmal durchsuchen.

Übung 8.2

Es existiert eine bestimmte Anzahl Worte bei denen jeder Buchstabe genau zwei Mal im Wort vorkommt.

Beispiele aus einem Englisch-Wörterbuch enthalten:

> Abba, Anna, appall, appearer, appeases, arraigning, beriberi, bilabial, boob, Caucasus, coco, Dada, deed, Emmett, Hannah, horseshoer, intestines, Isis, mama, Mimi, murmur, noon, Otto, papa, peep, reappear, redder, sees, Shanghaiings, Toto

Schreiben Sie eine Funktion `IsDoubleLetterWord()` welche `TRUE` zurück liefert wenn das übergebene Wort die oben beschriebene Eigenschaft aufweist, ansonsten soll `FALSE` zurückgegeben werden.

Übung 8.3

Der Römische Kaiser Julius Cäsar soll seine geheimen Botschaften mit einem einfachen Verschlüsselungsverfahren gesichert haben. Dazu hat er in seiner Botschaft jeden Buchstaben durch den Buchstaben ersetzt, der 3 Positionen weiter hinten im Alphabet zu finden ist.

So wurde zum Beispiel aus `a` ein `d` und aus `b` ein `e`. Die Buchstaben am Ende des Alphabets werden wieder auf den Anfang abgebildet. So wird aus `z` dann ein `c`.

a. Schreiben Sie eine Funktion, welche zwei Strings übernimmt. Einer der Strings enthält die originale Botschaft, in dem anderen String soll die verschlüsselte Geheimnachricht gespeichert werden.

   Der String kann Groß- und Kleinschreibung sowie Leerzeichen enthalten. Andere Satzzeichen (Punkt, Komma, etc.) sollen nicht vorkommen. Die Funktion soll die Buchstaben vor der Verschlüsselung in eine einheitliche Darstellung umwandeln (Groß- oder Kleinschreibung). Leerzeichen werden nicht verschlüsselt.

b. Generalisieren Sie die Verschlüsselungsfunktion, so dass anstelle der festen Verschiebung um 3 Positionen, Sie die Verschiebung frei wählen können.

   Sie sollten damit in der Lage sein die Nachrichten auch wieder zu entschlüsseln, indem Sie z.B. mit dem Wert 13 verschlüsseln und mit -13 wieder entschlüsseln.

Kapitel 9

Strukturen

9.1 Aggregierte Datentypen

Die meisten Datentypen mit denen wir bisher gearbeitet haben repräsentieren einen einzelnen Wert – eine ganze Zahl, eine Fließkommazahl oder einen Zeichenwert. Arrays und Strings dagegen sind aus mehreren Elementen aufgebaut, die jedes für sich genommen einen Wert darstellen, im Falle von Strings aus den einzelnen Zeichen. Diese Art von Datentypen bezeichnet man als **aggregierte** (zusammengesetzte) Datentypen.

Je nachdem was wir in unserem Programm mit den Daten machen, können wir aggregierte Datentypen als ein einzelnes Objekt betrachten oder auf die einzelnen Elemente des Datentyps zugreifen. Diese Mehrdeutigkeit ist für die Programmierung nützlich. Beim Funktionsaufruf übergeben wir das aggregierte Objekt als einzelnes Argument. Innerhalb der Funktion greifen wir dann auf die einzelnen Elemente des Objekts zu, verbergen diese Komplexität aber vor allen anderen Funktionen in unserem Programm.

Arrays sind aggregierte Datenobjekte aus Elementen des gleichen Typs, auf deren Elemente über ihren Index zugegriffen wird. In der Programmiersprache C gibt es einen weiteren Mechanismus um aggregierte Datenobjekte zu bilden die auch aus Elementen unterschiedlichen Typs bestehen können und weitere interessante Eigenschaften besitzen. Diese Datenobjekte werden in C als `struct` (Struktur) bezeichnet und können zum Beispiel benutzt werden um eigene Datentypen zu definieren.

9.2 Das Konzept eines geometrischen Punkts

Um die das Konzept der Strukturen zu erklären, möchte ich ein einfaches Beispiel aus dem Bereich der Geometrie verwenden. Stellen wir uns die geometrische

Beschreibung eines Punkts vor. In einem zweidimensionalen, kartesischen Koordinatensystem können wir jeden Punkt durch die Angabe von 2 Zahlen (Koordinaten) beschreiben. Diese beiden Koordinaten bilden zusammen ein Objekt, welches wir für weitere Betrachtungen immer gemeinsam verwenden wollen. In der Geometrie stellen wir deshalb Punkte oft als Zahlenpaar in Klammern dar, wobei ein Komma die Koordinaten trennt. So stellt zum Beispiel $P(0,0)$ den Ursprung unseres Koordiatensystems dar und $P(x,y)$ kennzeichnet den Punkt der x Einheiten auf der X-Achse und y Einheiten auf der Y-Achse vom Ursprung des Koordinatensystems entfernt ist.

Solch einen Punkt können wir in C als zwei Werte vom Typ `double` speichern. Es bleibt aber die Frage, wie wir diese beiden Werte zu einem zusammengesetzten Objekt vom Typ `Point_t` zusammenfassen. Es kann ja vorkommen, dass wir in unserem Programm eine Vielzahl von Punkten definieren und bearbeiten wollen, für die jeweils zwei Werte zusammengenommen den Punkt definieren.

Die Antwort auf diese Frage liefern die folgenden Programmzeilen. Wir definieren einfach einen neuen `Point_t` -Datentyp als `struct`:

```
typedef struct
{
    double x;
    double y;
} Point_t;
```

Die Definition gibt an, dass es in unserer Struktur zwei Elemente mit Namen `x` und `y` gibt. Diese Elemente werden auch als **Komponenten** der Struktur bezeichet. Die `struct` Definition muss in unserem Programm außerhalb der Funktionsdefinitionen vorgenommen werden, am besten direkt nach den `#include` Anweisungen.

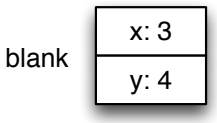
Es ist ein häufiger Fehler das Semikolon am Ende der Strukturdefinition zu vergessen. Normalerweise müssen wir hinter geschweiften Klammern kein Semikolon setzen, bei Strukturdefinitionen hingegen schon.

Nachdem wir jetzt einen neuen Struktur-Datentyp definiert haben, können wir ganz einfach neue Variablen dieses Typs erzeugen:

```
Point_t blank;
blank.x = 3.0;
blank.y = 4.0;
```

Die erste Zeile ist eine ganz normale Variablendeklaration: die Variable mit Namen `blank` hat den Datentyp `Point_t`. Die nächsten zwei Zeilen initialisieren die beiden Komponenten des `struct`. Der Punkt zwischen dem Namen der Strukturvariable und den Komponenten ist ein eigener Operator, der so genannte **Punktoperator**. Mit dem Punktoperator können wir über den Namen der Strukturvariable auf die einzelnen Komponenten der Struktur zugreifen.

Das Resultat der Zuweisung wird in dem folgenden Zustandsdiagramm gezeigt:



So wie bei anderen Variablen auch, wird der Variablenname `blank` außerhalb und der Wert innerhalb des Kastens geschrieben. In unserem Fall setzt sich der innere Teil des Kastens aus den einzelnen Werten der zwei Komponenten des aggregierten Datenobjekts zusammen.

9.3 Zugriff auf die Komponenten von Strukturen

Natürlich können wir die Werte der Komponenten einer Struktur nicht nur setzen, sondern diese auch wieder auslesen. Auch hier müssen wir wieder den Punktoperator anwenden.:

```
double x = blank.x;
```

Der Ausdruck `blank.x` bedeutet "gehe zu dem Objekt mit Namen `blank` und ermittle den Wert von `x`." In unserem Falle weisen wir dann den ermittelten Wert einer lokalen Variable mit Namen `x` zu. Dabei ist es wichtig zu verstehen, dass es keinen Konflikt zwischen der lokalen Variable `x` und der Komponente `x` der Variablen `blank` gibt. Es handelt sich um zwei voneinander verschiedene Speicherstellen und über den Punktoperator ist auch sichergestellt, dass es keinen Namenskonflikt zwischen `x` und `blank.x` gibt.

Wir können den Punktoperator in jedem normalen Ausdruck in C verwenden, wie man an den folgenden Beispielen sehen kann:

```
printf ("%0.1f, %0.1f\n", blank.x, blank.y);
double distance = sqrt (blank.x * blank.x + blank.y * blank.y);
blank.y++;
```

Die erste Programmzeile erzeugt die Bildschirmausgabe `3.0, 4.0`. Die zweite Zeile errechnet den Wert 5 und speichert diesen in der Variable `distance`. In der dritten Zeile wird der Wert von `blank.y` um `1` erhöht.

9.4 Operatoren und Strukturen

Die meisten Operatoren die wir bisher für einfache Datentypen wie `int` und `double` benutzt haben lassen sich leider nicht direkt für Strukturvariablen verwenden. Das betrifft auch die mathematischen Operatoren (`+`, `%`, etc.) und die Vergleichsoperatoren (`==`, `>`, etc.).

Der Zuweisungsoperator hingegen kann mit Strukturvariablen verwendet werden. Es gibt zwei Arten wie wir ihn einsetzen können. Zuerst lässt sich mit

dem Zuweisungsoperator eine Strukturvariable bei ihrer Definition bereits initialisieren. Darüber hinaus lassen sich Strukturvariablen gleichen Typs auch direkt kopieren, ohne dass wir jedes Element einzeln kopieren müssten. Eine Initialisierung sieht folgendermaßen aus:

```
Point_t blank = { 3.0, 4.0 };
```

Die Werte in den geschweiften Klammern werden der Reihe nach den einzelnen Komponenten der Strukturvariable zugewiesen. Wir müssen daher genau wissen in welcher Reihenfolge die einzelnen Komponenten in der Typdefinition angegeben wurden. In unserem Fall wird der erste Wert der Komponente `x` und der zweite Wert der Komponenten `y` zugewiesen.

Unglücklicherweise können wir diese Syntax nur während der Initialisierung der Variablen verwenden und nicht dazu benutzen der Variablen zu einem späteren Zeitpunkt neue Werte zuzuweisen. Die folgenden Programmzeilen sind daher fehlerhaft:

```
Point_t blank;
blank = { 3.0, 4.0 };       /* WRONG !! */
```

Jetzt habe ich aber behauptet, dass wir Strukturvariablen direkt kopieren können, warum ist dann diese Anweisung fehlerhaft? Es gibt eine kleine Einschränkung hinsichtlich der Kopierbarkeit. Strukturen können nur dann kopiert werden, wenn sie *kompatibel* sind, das heißt, wenn Sie von der gleichen Typdefinition abgeleitet sind. I unserem Falle weiß der Kompiler nicht, dass es sich bei der rechten Seite um den Inhalt einer Struktur handeln soll. Wir müssen daher noch den entspechenden Typ als explitite Typumwandlung (engl: cast) hinzufügen:

```
Point_t blank;
blank = (Point_t){ 3.0, 4.0 };
```

Das funktioniert!

Wenn wir zwei Strukturvariablen vom gleichen Typ definiert haben, können wir natürlich auch einfach eine Zuweisungsoperation ausführen:

```
Point_t p1 = { 3.0, 4.0 };
Point_t p2 = p1;
printf ("(%0.1f, %0.1f)\n", p2.x, p2.y);
```

Die Ausgabe dieser Programmzeilen beträgt `3.0, 4,0`.

9.5 Strukturen als Parameter

Wir können eine Struktur einfach als Parameter einer Funktion angeben:

```
void PrintPoint (Point_t point)
{
    printf ("(%0.1f, %0.1f)\n", point.x, point.y);
}
```

Die Funktion `PrintPoint()` wird mit einer Struktur vom Typ `Point_t` aufgerufen und gibt die Koordinaten des Punkts auf dem Bildschirm aus. Wenn

wir die Funktion folgendermaßen aufrufen `PrintPoint(blank)`, erzeugt sie die Ausgabe `(3.0, 4.0)`.

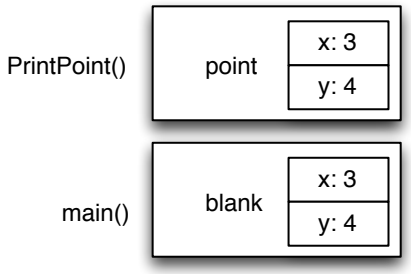
In einem zweiten Beispiel können wir die `ComputeDistance()` Funktion aus Section 5.2 so umschreiben, dass sie zwei `Point_t` -Variablen anstelle von vier `double` -Variablen als Parameter besitzt:

```
double ComputeDistance (Point_t p1, Point_t p2)
{
    double dx = p2.x - p1.x;
    double dy = p2.y - p1.y;
    return sqrt (dx*dx + dy*dy);
}
```

9.6 Call by value

Wir erinnern uns, Parameter und Argument einer Funktion sind unterschiedliche Dinge. Der Parameter ist eine Variable innerhalb der aufgerufenen Funktion. Bei dem Argument handelt es sich um den Wert den die aufrufenden Funktion an die aufgerufene Funktion übergibt. Dieser Wert wird beim Funktionsaufruf in die Parametervariable der aufgerufenen Funktion kopiert. Diesen Vorgang bezeichnet man als Parameterübergabe.

Wenn wir eine Struktur als Argument der Funktion `PrintPoint()` übergeben, sieht das Stackdiagramm folgendermaßen aus:



Wenn `PrintPoint()` jetzt den Wert der Komponenten von `point` verändert, so würde das keine Auswirkungen auf die Komponenten von `blank` haben. Da in unserem Beispiel die Funktion `PrintPoint()` die Parametervariable nicht modifiziert, ist diese Form der Parameterübergabe für das Programm angemessen.

Wie bereits erwähnt, bezeichnen wir diese Art der Parameterübergabe als *call by value*, weil nur der Wert der Struktur (oder eines anderen Datentyps) an die Funktion übergeben wird und nicht die Struktur selbst.

9.7 Call by reference

Es gibt noch eine andere Art der Parameterübergabe an die aufgerufene Funktions, das so genannte *call by reference*. In diesem Fall wird nicht der Wert des Arguments, sondern ein Verweis an die aufgerufene Funktion übergeben.

Genau genommen gibt es in C kein echtes *call by reference*. Wir behelfen uns damit, dass wir statt der direkten Übergabe des Objekts einen Pointer auf ein Speicherobjekt an die Funktion übergeben. Dabei wird Wert des Adresse des Speicherobjekts als Argument in den Pointerparameter der Funktion kopiert. Da sich dabei die Adresse nicht verändert kann die Funktion direkt auf das Speicherobjekt außerhalb der Funktion zugreifen.

Wir können mit diesem Mechanismus auch einen Pointer auf eine Struktur an eine Funktion übergeben und die Funktion dazu benutzen um die Struktur selbst zu verändern.

Stellen wir uns folgendes Beispiel vor: ein Punkt in einem Koordinatensystem soll an der 45°-Linie gespiegelt werden. Dazu müssen einfach die beiden Koordinaten des Punkts vertauscht werden.

Wenn wir jetzt eine Funktion `ReflectPoint()` in der folgenden Weise schreiben, so werden wir damit nicht den erhofften Erfolg haben:

```
void ReflectPoint (Point_t point)      /* Does not work! */
{
    double temp = point.x;
    point.x = point.y;
    point.y = temp;
}
```

Die Funktion `ReflectPoint()` arbeitet nicht korrekt, weil die Änderungen, die wir an der Strukturvariablen in der Funktion vornehmen, keine Auswirkungen auf die Strukturvariable in der aufgerufenen Funktion haben.

Wir müssen die Funktion so ändern, dass wir statt einer Kopie der Struktur einen Verweis auf die Struktur erhalten (call-by-reference). Dazu muss die Funktion einen Parameter vom Typ `Point_t *ptr` (Pointer auf die Struktur vom Typ `Point_t`) erhalten und wir müssen die Adresse des originalen Strukturobjekts an die Funktion übergeben:

```
void ReflectPoint (Point_t *ptr)
{
    double temp = ptr->x;
    ptr->x = ptr->y;
    ptr->y = temp;
}
```

Normalerweise benutzen wir den Punktoperator (`.`) um auf die Elemente einer Struktur zuzugreifen. Wenn wir aber über einen Pointer (Zeiger) auf die Komponenten einer Struktur zugreifen, müssen wir einen speziellen Operator, den **Pfeiloperator** (`->`) benutzen.

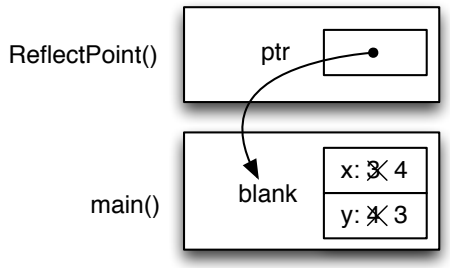
Beim Funktionsaufruf müssen wir jetzt nur noch die Adresse der Struktur ermitteln und der Funktion als Argument mitgeben. Wir benutzen dazu den *Adressoperator* (`&`):

```
PrintPoint (blank);
ReflectPoint (&blank);
PrintPoint (blank);
```

Die Ausgabe des Programms sieht folgendermaßen aus und entspricht unseren Erwartungen:

```
(3.0, 4.0)
(4.0, 3.0)
```

Das Stackdiagramm für den Funktionsaufruf sieht folgendermaßen aus:



Der Parameter `ptr` ist ein Zeiger auf die Struktur `blank`. Pointer werden in einem Stackdiagramm als ein Punkt mit einem Pfeil dargestellt. Der Pfeil zeigt direkt auf das Speicherobjekt dessen Adresse im Pointer gespeichert ist.

Der wichtige Unterschied, den dieses Diagramm zum Ausdruck bringt, ist der Fakt, dass alle Änderungen welche innerhalb der Funktion `ReflectPoint()` über den Pointer `ptr` gemacht werden, eine direkte Änderung der Strukturvariablen `blank` zur Folge hat.

Wenn wir die Adresse einer Struktur an eine Funktion übergeben haben wir mehr Möglichkeiten, da die Funktion die Struktur selbst verändern kann und nicht nur mit einer Kopie arbeitet. Außerdem ist es etwas schneller, weil der Rechner nicht erst eine Kopie der ganzen Struktur erzeugen muss. Auf der anderen Seite ist es weniger sicher und schwieriger nachvollziehbar. Wir müssen selbst nachverfolgen an welchen Stellen eine Struktur modifiziert wird und gerade wenn wir viele Funktionen benutzen kann das schnell unübersichtlich werden. Trotzdem werden in C die meisten Strukturen über Pointer an Funktionen übergeben. Die in diesem Buch beschriebenen Funktionen werden deshalb dieser Tradition treu bleiben.

9.8 Rechtecke

Ok, nachdem wir jetzt also einen Punkt als Struktur dargestellt haben, wollen wir noch etwas weiter gehen und uns überlegen wie wir ein Rechteck als Da-

tenstruktur beschreiben können. Welche Daten sind nötig, um die geometrische Figur eines Rechtecks vollständig zu beschreiben?
(Um es uns etwas einfacher zu machen wollen wir davon ausgehen, dass das Rechteck immer rechtwinklig zu unserem Koordinatensystem ausgerichtet ist und niemals verdreht.)

Es gibt mehrere Möglichkeiten: wir könnten den Mittelpunkt des Rechtecks beschreiben (ein Punkt mit zwei Koordinaten) und die Länge und Breite des Rechtecks. Eine andere Möglichkeit wäre es zwei gegenüberliegende Ecken als Punkte zu beschreiben.

Die meisten existierenden Programme geben die obere linke Ecke sowie die Länge und die Breite des Rechtecks an. Um ein solches Rechteck in C zu beschreiben brauchen wir eine Struktur die einen Punkt `Point_t` und zwei Fließkommazahlen `double` enthält:

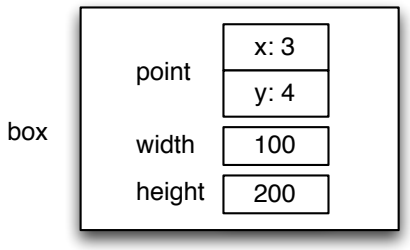
```
typedef struct
{
    Point_t corner;
    double width, height;
} Rectangle_t;
```

Wie wir feststellen könnne, ist es erlaubt, dass eine Struktur eine andere Struktur als Komponente enthalten kann. Diese Art des Aufbaus von komplexeren aus einfacheren Strukturen ist sogar ziemlich verbreitet.

Diese Definition bedeutet aber auch, dass wir bevor wir ein Rechteck `Rectangle_t` definieren können, zuerst einmal einen Punkt `Point_t` definieren müssen:

```
Point_t corner = { 0.0, 0.0 };
Rectangle_t box = { corner, 100.0, 200.0 };
```

Diese Programmzeilen erzeugen eine neue `Rectangle_t` Struktur und initialisieren die Komponenten. In der folgenden Abbildung ist diese Zuweisung als Zustandsdiagramm dargestellt:



Wenn wir auf die Komponenten `width` und `height` zugreifen wollen, so können wir das in der gewohnten Form tun:

```
box.width = box.width + 50.0;
printf("%f\n", box.width);
```

Um auf die Komponenten der Struktur `corner` zuzugreifen, könnten wir, falls wir den Wert nur auslesen wollen, eine temporäre Variable verwenden:

```
Point_t temp = box.corner;
double x = temp.x;
```

Es ist allerdings viel einfacher den Punktoperator einzusetzen und über die verkettete Angabe der beiden Strukturen direkt auf die Komponente zuzugreifen:

```
double x = box.corner.x;
```

Um diese Anweisung zu verstehen ist es am Sinnvollsten die Programmzeile von rechts nach links zu lesen: "Ermittle den Wert `x` in der Struktur `corner` in der Struktur `box` und weise ihn der lokalen Variable `x` zu."

Ich sollte vielleicht noch hinzufügen, dass man natürlich die Strukturvariable `box` auch in einem einzigen Programmschritt initialisieren kann:

```
Rectangle_t box = { { 0.0, 0.0 }, 100.0, 200.0 };
```

Die inneren geschweiften Klammern beschreiben die Koordinaten des Punkts `corner`. Die zwei anderen Werte definieren `width` und `height` des neuen Rechtecks `box`. Wir haben damit ein Beispiel für eine *verschachtelte Struktur* geschaffen. Welche Variante der Initialisierung wir dabei verwenden ist uns überlassen. Ich finde die erst Variante übersichtlicher, sie bedeutet aber auch mehr Schreibarbeit für den Programmierer.

9.9 Strukturen als Rückgabewerte

Es ist möglich Funktionen zu schreiben die eine Struktur an die aufrufende Funktion zurückgeben. Wir können zum Beispiel eine Funktion `FindCenter()` erstellen, welche ein `Rectangle_t` als Parameter besitzt und einen `Point_t` Wert zurückgibt, welcher die Koordinaten des Mittelpunkts des Rechtecks enthält:

```
Point_t FindCenter (Rectangle_t box)
{
    double x = box.corner.x + box.width/2;
    double y = box.corner.y + box.height/2;
    Point_t result = {x, y};
    return result;
}
```

Um diese Funktion aufzurufen, müssen wir ein `Rectangle_t` als Argument an die Funktion übergeben (es wird eine Kopie des Werts der Struktur übergeben: call-by-value). Den Rückgabewert der Funktion weisen wir einer `Point_t` Variable zu:

```
Rectangle_t box = { {0.0, 0.0}, 100, 200 };
Point_t center = FindCenter (box);
PrintPoint (center);
```

Die Ausgabe dieser Programmzeilen lautet: `(50, 100)`

Wir hätten natürlich auch einen Pointer auf die Struktur an die Funktion übergeben können (call-by-reference). In diesem Fall hätte unsere Funktion folgendermaßen definiert werden müssen:

```
Point_t FindCenter (Rectangle_t *box)
{
    double x = box->corner.x + box->width/2;
    double y = box->corner.y + box->height/2;
    Point_t result = {x, y};
    return result;
}
```

In diesem Fall müssen wir neben dem Parameter natürlich auch noch den Zugriff auf die Komponenten der Struktur anpassen. Das ist notwendig, weil es sich bei `box` jetzt um einen Pointer handelt. Weiterhin muss natürlich auch der Funktionsaufruf von `FindCenter()` verändert werden:

```
Point_t center = FindCenter (&box);
```

9.10 Andere Datentypen als Referenz übergeben

Wir können nicht nur Strukturen als Pointer an eine Funktion übergeben. Jeder Datentyp den wir bisher gesehen haben, kann auch als Pointer in einem Funktionsaufruf benutzt werden. So können wir zum Beispiel zwei ganzzahlige Werte in der folgenden Funktion direkt vertauschen:

```
void Swap (int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

Beim Funktionsaufruf müssen wir die Adressen der Variablen `i` und `j` als Argumente der Funktion verwenden:

```
int i = 7;
int j = 9;
printf (" i=%i, j=%i\n", i, j);
Swap (&i, &j);
printf (" i=%i, j=%i\n", i, j);
```

Die Ausgabe des Programms zeigt, dass die Werte der Variablen getauscht wurden. Sie können ja selbst ein Stack-Diagramm zeichnen um sich davon zu überzeugen, dass die Funktion direkt auf die Variablen der aufrufenden Funktion zugreift.
Wenn ich die Funktionsparameter `x` und `y` als normale `int` -Variablen (ohne `*`) deklariert hätte, wäre die Funktion `Swap()` nicht korrekt. Sie würde `x` und `y` innerhalb der Funktion vertauschen, hätte aber keinen Einfluss auf `i` und `j`.

Wenn wir Werte per *call-by-value* an eine Funktion übergeben, dann ist es durchaus üblich hier einen komplexeren Ausdruck als Argument der Funktion zu verwenden. Bei *call-by-reference* Funktionen kann aber die Verwendung

von Ausdrücken zu schwer zu findenden Programmfehlern führen. Schauen wir uns folgende Programmzeilen an:

```
int i = 7;
int j = 9;
Swap (&i, &j+1);          /* WRONG!! */
```

Vermutlich wollte der Programmierer den Wert von `j` vor dem Tausch der Variablen um `1` erhöhen. Allerdings wird hier nicht der Wert von `j` erhöht, sondern die Adresse von `j`. Es ist also gar nicht mehr klar, wohin der Pointer `*y` gerade zeigt.

Eine gute Faustregel ist daher als Argumente einer *call-by-reference* Funktion nur Variablen zu verwenden und keine Ausdrücke.

9.11 Glossar

Struktur (engl: *structure*): Ein zusammengesetztes Datenobjekt, welches im Gegensatz zu einem Array aus einer Ansammlung von Werten unterschiedlichen Typs bestehen kann. Strukturen werden in C durch das Schlüsselwort `struct` gekennzeichnet.

Komponente (engl: *member variable*): Eine benanntes Element einer Struktur auf das über seinen Namen einzeln zugegriffen werden kann.

Verweis (engl: *reference*): Ein Wert der auf ein Datenobjekt verweist. Im Stack-Diagramm werden Verweise als Pfeil von einem Datenobjekt auf ein anderes dargestellt.

Call by value (engl: *call by value*): Eine Art der Parameterübergabe beim Funktionsaufruf. Dabei wird der Wert des Arguments in den dazugehörigen Parameter der Funktion kopiert. Die Speicherstelle des Parameters befindet sich innerhalb der aufgerufenen Funktion und ist komplett unabhängig von der aufgerufenen Funktion. Dies ist die Standardform der Parameterübergabe in C.

Call by reference (engl: *call by reference*): Eine Art der Parameterübergabe beim Funktionsaufruf. Dabei wird an die aufgerufene Funktion ein Verweis übergeben. Über diesen Verweis kann die aufgerufene Funktion direkt auf Werte außerhalb ihres Speicherbereichs zugreifen.

9.12 Übungsaufgaben

Übung 9.1

Im Abschnitt 9.5 wird die Funktion `PrintPoint()` definiert. Der Parameter dieser Funktion wird als Wert (Call-by-value) übergeben.

Ändern Sie die Definition dieser Funktion, so dass nur eine Referenz auf die auszugebende Variable übergeben wird (Call-by-reference). Testen Sie die neu geschriebene Funktion.

Übung 9.2

Computerspiele werden erst dadurch interessant, dass die Aktionen ihres Gegenspielers nicht vorhersagbar sind. Im Kapitel 7.6 haben wir gesehen wie sich Zufallszahlen in C erzeugen lassen.

Schreiben Sie ein kleines Spiel, in dem der Computer eine beliebige Zahl im Bereich von 1 - 20 auswählt und Sie auffordert die gewählte Zahl zu erraten.

Falls ihre Eingabe kleiner ist als der Zufallswert soll der Computer ausgeben: 'Meine Zahl ist größer!' und Sie zu einer erneuten Eingabe auffordern. Für den Fall, dass ihre Eingabe größer ist soll die Ausgabe 'Meine Zahl ist kleiner!' lauten.

Damit das Programm bei jedem Versuch mit einem neuen Wert startet, muss der Zufallszahlengenerator am Anfang des Programms neu initialisiert werden (siehe Kapitel 7.14). Sie können dazu die Funktion `time()` verwenden, welche bei jedem Aufruf eine aktualisierte Anzahl eines Sekundenwerts zurückgibt.

```
srand(time(NULL));   /*Initialisierung des Zufallszahlengenerators*/
```

Haben Sie die Zahl richtig erraten soll der Computer ihnen gratulieren und die Anzahl der benötigten Versuche und den aktuellen 'High-Score' ausgeben.

Der Computer speichert dazu den High-Score (die Anzahl der minimal benötigen Versuche) in einem `struct` zusammen mit ihrem Namen.

Ist der aktuelle High-Score Wert größer als die Anzahl ihrer Versuche soll ihr Spielergebnis zusammen mit ihrem Namen als High-Score Wert gespeichert werden. Dazu fragt die High-Score Funktion Sie nach ihrem Namen.

Durch Drücken der Taste 'q' soll das Programm beendet werden.

Kapitel 10

Hardwarenahes Programmieren

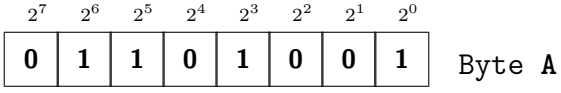
10.1 Bits and Bytes

Es ist hin und wieder notwendig, nicht nur auf einzelne Variablen im Speicher des Computers zuzugreifen, sondern sogar die einzelnen Bits dieser Werte abzufragen, zu verarbeiten oder zu verändern. Speziell wenn man mit Microcontrollern arbeitet, um Regelungs- und Steuerungsaufgaben zu automatisieren, kann es nötig sein, den Zustand angeschlossener Geräte und Sensoren bitgenau zu kennen, auszuwerten und gegebenenfalls zu ändern.

C bietet dafür eine Reihe von sogenannten *Bitoperatoren* die allerdings trotz ihres Namens nicht direkt mit einzelnen Bitwerten arbeiten sondern immer auf mehrere Bits gleichzeitig angewendet werden. Daran ist der Aufbau moderner Computersysteme schuld, welche die Daten immer parallel zwischen Prozessor und Hauptspeicher übertragen und anschließend auch verarbeiten. Es wäre viel zu langsam jedes Bits einzeln aus dem Hauptspeicher zu laden und in einer Operation des Prozessors zu verarbeiten. Weiterhin ist es nicht möglich einzelne Bits überhaupt sinnvoll zu adressieren. Der kleinste direkt adressierbare Speicherbereich ist ein Byte und dieses besteht aus einer Folge von 8 Bit.

Es hat sich eingebürgert, die zwei Zustände eines einzelnen Bits durch die numerischen Werte 0 oder 1 darzustellen. Daraus ableitend, kann der Wert eines Bytes entsprechend der Bit-Wertigkeit im Dualsystem als dezimaler Wert interpretiert werden.

So entspricht die folgende Binärdarstellung des Byte A dem dezimalen Wert `105` ($2^6 + 2^5 + 2^3 + 2^0$):



10.2 Operatoren für die Arbeit mit Bits

Die binären Operatoren ähneln in ihrer Funktion stellenweise den logischen Operatoren aus Kapitel 5.6. So existieren in C die drei **Bitoperatoren** *AND* (`&`), *OR* (`|`) und *NOT* (`~`).

Im Gegensatz zu den logischen Operatoren, arbeiten die binären Operatoren allerdings auf der Ebene der einzelne Bits. So liefert zum Beispiel der Operator für die *bitweise Negation* `~` das folgende Bitmuster:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
1	0	1	0	1	0	1	0	`Byte A`
0	1	0	1	0	1	0	1	`Byte ~A`

An jeder Stelle wo zuvor eine 1 stand, steht jetzt eine 0 und umgekehrt.

Mit Hilfe des *bitweise AND* Operators `&` können die Bitfolgen zweier Operanden UND-verknüpft werden:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
1	0	1	0	1	0	1	0	`Byte A`
1	1	1	1	0	0	0	0	`Byte B`
1	0	1	0	0	0	0	0	`Byte A & B`

Wie wir erkennen können, werden die einzelnen Bits von Byte A und Byte B UND-verknüpft. Das Ergebnis der bitweisen UND-Verknüpfung unterscheidet sich also stark von der logischen UND-Verknüpfung.

Mit Hilfe des *bitweise OR* Operators `|` können die Bitfolgen zweier Operanden ODER-verknüpft werden:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
1	0	1	0	1	0	1	0	`Byte A`
1	1	1	1	0	0	0	0	`Byte B`
1	0	1	1	1	0	1	0	`Byte A | B`

Darüber hinaus existiert ein Bitoperator, welcher keine Entsprechung als logischer Operator hat. Dabei handelt es sich um den *Exklusiv-ODER* oder XOR Operator(`^`). Er besitzt die folgende Wertetabelle:

A	B	$A \hat{} B$
0	0	0
1	0	1
0	1	1
1	1	0

Dieser Operator hat die interessante Eigenschaft, dass die zweimalige Ausführung einer XOR Operation mit dem gleichen Operanden wieder zum originalen Zustand zurückführt:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
1	**0**	**1**	**0**	**1**	**0**	**1**	**0**	`Byte A`
1	**1**	**1**	**1**	**0**	**0**	**0**	**0**	`Byte B`
0	**1**	**0**	**1**	**1**	**0**	**1**	**0**	`Byte A ^ B`
1	**0**	**1**	**0**	**1**	**0**	**1**	**0**	`Byte A ^ B ^ B`

10.3 Verschiebeoperatoren

Es ist weiterhin möglich alle Bits in einer Bitfolge nach links oder nach rechts zu schieben.

Dafür kommen die Operatoren « und » zum Einsatz. Der « Operator schiebt alle Bits entsprechend des Werts `n` des zweiten Operanden nach links. Die Operation entspricht einer binären Multiplikation mit 2^n.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
0	**0**	**1**	**0**	**1**	**0**	**1**	**0**	`Byte A (42)`
1	**0**	**1**	**0**	**1**	**0**	**0**	**0**	`Byte A « 2  (168)`

Der » Operator schiebt alle Bits entsprechend des Werts `n` des zweiten Operanden nach rechts. Die Operation entspricht einer binären Division mit 2^n.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
0	**0**	**1**	**1**	**0**	**0**	**0**	**0**	`Byte A (48)`
0	**0**	**0**	**0**	**1**	**1**	**0**	**0**	`Byte A » 2  (12)`

10.4 Anwendung binärer Operatoren

Wenn wir Byte A mit dem negierten Byte A UND-verknüpfen ist die resultierende Bitfolge komplett auf 0 gesetzt:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
1	0	1	0	1	0	1	0	`Byte A`
0	0	0	0	0	0	0	0	`Byte A & ~A`

Wir können diese Eigenschaft benutzen um aus einer Bitfolge einzelne Bits zu isolieren. Wir benutzen dafür eine sogenannte *Bitmaske*. Das ist eine Bitfolge, bei der das für uns interessante Bit den Wert 1 und alle nicht interessanten Bits den Wert 0 besitzen. Das resultierende Ergebnis hängt damit nur vom Zustand des über die Maske ausgewählten Bits ab:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
1	0	1	0	1	0	1	0	`Byte A`
0	0	1	0	0	0	0	0	`Maske` $\mathbf{2^5}$ `(32)`
0	0	1	0	0	0	0	0	`Byte A & 32`

Wenn wir Byte A mit dem negierten Byte A ODER-verknüpfen ist die resultierende Bitfolge komplett auf 1 gesetzt.:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
1	0	1	0	1	0	1	0	`Byte A`
1	1	1	1	1	1	1	1	`Byte A | ~A`

Wir können diese Eigenschaft benutzen um innerhalb einer Bitfolge einzelne Bits gezielt zu setzen. Indem wir in einer Bitfolge die zu setzenden Bits mit dem Wert 1 und alle nicht zu verändernden Bits mit dem Wert 0 kodieren lässt sich der Wert einzelner Bits gezielt setzen:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
1	0	1	0	1	0	1	0	`Byte A`
0	0	0	1	0	0	0	0	`Setze Bit` $\mathbf{2^4}$`(16)`
1	0	1	1	1	0	1	0	`Byte A | 16`

10.4.1 Variablen für Bitfolgen

Binäre Operatoren lassen sich auf beliebige, ganzzahlige Werte anwenden. In der Regel werden wir dazu Variablen zu verwenden. Daher stellt sich natürlich sofort die Frage, welchen Typ soll diese Variable haben?

Diese Frage ist nicht ganz einfach zu beantworten. Ist die Bitfolge nur 8 Bit lang, so verwenden viele Programmierer den Datentyp `unsigned char`.

ACHTUNG: Der Datentyp `char` ist in C ein ganz normaler, ganzzahliger Datentyp der auf den allermeisten Rechnern 8 Bits enthält, es können aber auch mehr oder weniger Bit sein!

Typischerweise wird in C für ganzzahlige Werte das *Most Significant Bit* (MSB) genutzt, um das Vorzeichen des Werts zu speichern. Das MSB ist das Bit mit der höchsten Wertigkeit und findet sich in unserer Bitfolge ganz links. Bitoperatoren wirken aber immer auf alle Bits der Folge gleichzeitig, was dazu führt, dass eine Änderung des MSB das Vorzeichen des gespeicherten Werts verändert. Da das in den allermeisten Fällen nicht sinnvoll ist, lassen sich mit dem Schlüsselwort `unsigned` vorzeichenlose Variablen definieren.

Eine besondere Eigenart von C ist es, dass in der Sprache nicht genau festgelegt wurde wie viele Bit für eine Variable vom Typ *int* genutzt werden. Es sind immer mindestens 16 Bit, je nach Rechnerarchitektur können es aber auch 32 oder 64 Bit sein.

Diese Ungenauigkeiten in der Sprachbeschreibung haben dazu geführt, dass in neueren Versionen der Sprache (ab C99) ganzzahlige Datentypen mit fester Breite eingeführt wurden.

In der Bibliothek `stdint.h` wurden eine Reihe neuer Datentypen mit exakter Bit-Anzahl definiert, die wir in unserem Programm verwenden können und die uns davor schützen, dass möglicherweise unser Programm auf einem Mikrocontroller zu Fehlern führt, obwohl es auf dem PC wunderbar funktioniert hat. In der folgenden Tabelle habe ich die wichtigsten Typen zusammengefasst:

Länge	Typ mit Vorzeichen	Typ ohne Vorzeichen
8 Bit	`int8_t`	`uint8_t`
16 Bit	`int16_t`	`uint16_t`
32 Bit	`int32_t`	`uint32_t`

10.4.2 Erstellung von Bitmasken

Um gezielt einzelne Bits an einer Speicherstelle verändern zu können, benötigen wir eine so genannte *Bitmaske*. Das ist nichts weiter als eine Variable oder ein Wert, dessen Bits an den entsprechenden Stellen gesetzt sind, so dass wir damit gezielte Veränderungen einzelner Bits erreichen können.

Diese Maske können wir entweder als konstanten Wert in unserem Programm speichern, oder bei Bedarf dynamisch erzeugen. Eine beliebte Methode ist es diese Maske durch die gezielte Verschiebung einzelner Bits zu generieren.

Wir hatten bereits das MSB kennen gelernt. Ein weiteres wichtiges Bit ist das *Least Significant Bit* (LSB). Das ist das am weitesten rechts stehende Bit einer Bitfolge. Es hat den Wert 2^0 (dezimal 1) und ist damit das einzige Bit mit einem ungeraden Wert.

Um die Bitmaske `my_mask` zu erzeugen, in der das Bit 2^5 gesetzt ist, können wir folgenden Code verwenden:

```
uint8_t my_mask = (1 << 5);
```

Um mehrere Bits in der Maske zu setzen, verwenden wir den bitweise-ODER Operator. So lassen sich zum Beispiel Bit 2^3 und 2^5 in einem Befehl setzen:

```
uint8_t my_mask2 = (1 << 3) | (1 << 5);
```

Eine weitere interessante Möglichkeit ergibt sich, wenn wir den C-Präprozessor benutzen, um für die Verschiebeoperation ein Makro zu definieren. Bisher haben wir `#define` nur dafür benutzt um konstante Werte zu definieren. Es ist aber auch möglich ein Makro mit Argumenten zu definieren:

```
#define BIT(x) (1 << (x))
```

Es wird das Makro `BIT(x)` definiert, bei dem der Wert `x` von dem im Programm angegebenen Wert abhängt. Dadurch können wird die Lesbarkeit unseres Programms sehr stark verbessern. So ist der Ausdruck:

```
uint8_t my_mask = BIT(5);
```

identisch zu folgendem Code:

```
uint8_t my_mask = (1 << 5);
```

10.4.3 Verwendung von Bitmasken

Um das Bit 2^5 wieder zu löschen, benutzen wir den bitweisen UND-Operator (`&`) gemeinsam mit dem NOT-Operator (`~`).

```
my_mask = my_mask & ~BIT(5);
```

Dazu invertieren wir den zweiten Operanden mit dem bitweisen NOT-Operator, dass heißt es sind jetzt alle Bits gesetzt, außer 2^5. Anschließend führen wir eine UND-Verknüpfung diese Bitfolge mit unserer Maske durch, was dazu führt, dass Bit 2^5 gelöscht wird, während alle anderen Bits ihren aktuellen Wert beibehalten.

Um ein Bit zwischen zwei Zuständen wechselweise umschalten zu lassen, benutzen wir den XOR-Operator (`^`).

```
my_mask = my_mask ^ BIT(5);
```

Bit 2^5 wird dadurch invertiert. Hatte es vorher den Wert 1, so ist dieser jetzt 0 und umgekehrt. Damit kann man zum Beispiel auf einem Mikrocontroller eine einfache Blinkschaltung einer angeschlossenen LED realisieren.

Mit dem UND-Operator lässt sich überprüfen, ob ein bestimmtes Bit gesetzt ist:

```
if (my_maks & BIT(5))
{...}
```

Dieser Ausdruck ist dann wahr, wenn Bit 2^5 gesetzt ist, anderenfalls ist er falsch.

ACHTUNG: Es ist wichtig sich daran zu erinnern, dass Informatiker beim Zählen immer mit der 0 anfangen! Wenn wir das LSB verändern wollen, müssen wir `BIT(0)` schreiben. Das ist aber eigentlich ganz logisch, denn dieses Bit hat ja den Wert 2^0.

10.5 Glossar

Bit (engl: *bit*): Ein Bit (*binary digit*) ist die kleinste Informationseinheit. Ein Bit kann zwei Zustände annehmen, die üblicherweise durch die Werte 0 und 1 dargestellt werden und die Grundlage des binären Zahlensystems bilden. In Computern werden üblicherweise mehrere Bits gleichzeitig bearbeitet.

Byte (engl: *byte*): Ein Byte ist üblicherweise eine Folge von 8 Bit und stellt die kleinste adressierbare Einheit in einem Computersystem dar. Die Festlegung der Größe eines Bytes auf 8 Bit hat historische Gründe, weil sich damit die Buchstaben des lateinischen Alphabets sowie Ziffern und Sonderzeichen darstellen lassen.

10.6 Übungsaufgaben

Übung 10.3

Es ist folgendes Programm gegeben, welches eine Zahl von der Tastatur einliest und überprüft, ob es sich um eine gerade oder eine ungerade Zahl handelt:

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

int main(void)
{
    int eingabe;
    int test;

    printf("Geben Sie eine Zahl ein: ");
    test = scanf("%i",&eingabe);
    if (test == 0)
    {
        printf("Fehler: Es wurde keine Zahl eingegeben\n");
        return EXIT_FAILURE;
    }
```

```
    if (eingabe & 1)
    {
        printf("ungerade Zahl\n");
    }
    else printf("gerade Zahl\n");
    return EXIT_SUCCESS;
}
```

a. Erklären Sie, wie das Programm herausfindet ob die eingegebene Zahl gerade oder ungerade ist.

b. Was passiert wenn der Benutzer anstelle einer *Zahl* einen *Buchstaben* auf der Tastatur eingibt? Welche Bedeutung hat dabei die Variable *test* im Programm?

c. Versehen Sie das Programm mit geeigneten Kommentaren für spätere Entwickler.

Anhang A

Guter Programmierstil

A.1 Eine kurze Stilberatung für Programmierer

Während der Beschäftigung mit der Programmiersprache C werden Sie feststellen, dass es einige Regeln gibt die Sie unbedingt beachten müssen, während andere Regeln und Designentscheidungen eher als eine Art stille Übereinkunft zwischen Programmierern getroffen werden und als die 'übliche' Art und Weise der Programmierung angesehen werden.

Viele diese Regeln sind willkürlich getroffen, trotzdem ist es sinnvoll und vorteilhaft diese Konventionen zu kennen und sich daran zu halten, weil sie ihre Programme für Sie und andere einfacher lesbar machen und ihnen helfen Fehler zu vermeiden. Es können im wesentlichen drei Arten von Regeln unterschieden werden:

Naturgesetze: Diese Art von Regeln beschreiben Prinzipien der Logik und der Mathematik und gelten damit ebenfalls für Programmiersprachen wie C (oder andere formale Systems). So ist es zum Beispiel nicht möglich die Lage und Größe eines Rechtecks in einem Koordinatensystem durch weniger als vier Angaben genau zu beschreiben. Ein weiteres Beispiel besagt, dass die Addition von zwei natürlichen Zahlen dem Kommutativgesetz unterliegt. Dieser Zusammenhang ergibt sich aus der Definition der Addition und hat nichts mit der Programmiersprache C zu tun.

Regeln von C: Jede Programmiersprache definiert syntaktische und semantische Regeln die nicht verletzt werden dürfen, da sonst das Programm nicht korrekt übersetzt und ausgeführt werden kann. Einige dieser Regeln sind willkürlich gewählt, wie zum Beispiel das `=` Symbol, dass den Zuweisungsoperator darstellt und *nicht* die Gleichheit der Werte. Andere Regeln widerspiegeln die zugrundeliegenden Beschränkungen des Vorgangs der Kompilation und Ausführung des Programms. So müssen zum Beispiel die Typen der Parameter von Funktionen explizit spezifiziert werden.

Stil und Übereinkunft: Weiterhin existieren eine Reihe von Regeln die nicht durch den Compiler vorgegeben oder überprüft werden, die aber trotzdem wichtig dafür sind, dass Programme fehlerfrei erstellt werden, gut lesbar sind und durch Sie selbst und durch andere modifiziert, getestet und erweitert werden können. Beispiele dafür sind Einrückungen und die Anordnung von geschweiften Klammern, sowie Konventionen über die Benennung von Variablen, Funktionen und Typen.

In diesem Abschnitt werde ich kurz den Programmierstil zusammenfassen, der in diesem Buch verwendet wird. Er lehnt sich lose an die "Nasa C Style Guide"[1] an und das Hauptaugenmerk ist dabei auf die gute Lesbarkeit des Codes gerichtet. Es kommt weniger darauf an Platz zu sparen oder den Tippaufwand zu minimieren.

Da C eine - für eine Programmiersprache - vergleichsweise lange Geschichte aufweist, haben sich mehrere verschiedene Programmierstile herausgebildet. Es ist wichtig, dass Sie diese Stile lesen und verstehen können und dass Sie sich in ihrem eigenen Code auf einen Stil festlegen. Das macht den Programmcode viel zugänglicher, sollten es einmal notwendig werden, dass Sie den Code mit anderen Programmierern austauschen oder auf Teile ihres Codes zugreifen wollen, den Sie selbst vor einigen Jahren geschrieben haben.

A.2 Konventionen für Namen und Regeln für die Groß- und Kleinschreibung

Als generelle Regel sollten Sie sich angewöhnen bedeutungsvolle Namen für ihre Variablen und Funktionen zu verwenden. Idealerweise können Sie durch die Verwendung so genannter *sprechender Bezeichner* für Funktionen und Variablen bereits deren Verhalten und Verwendung erkennen.

Auch wenn es vielleicht aufwändiger ist eine Funktion `FindSubString()` anstatt `FStr()` zu nennen, so ist doch der erste Name fast selbsterklärend und kann ihnen eine Menge Zeit bei der Fehlersuche und späteren Wiederverwendung des Programms sparen.

Benutzen Sie keine Variablennamen die nur aus einem Buchstaben bestehen!

Ähnlich wie bei Funktionen sollten Sie die Namen ihrer Programmvariablen für sich selbst sprechen lassen. Durch einen geeigneten Namen wird von selbst klar welche Werte in der Variable gespeichert werden.

Wie zu jeder guten Regel gibt es auch hier einige Ausnahmen: Programmierer benutzen üblicherweise `i`, `j` und `k` als Zählvariablen in Schleifen und für räumliche Koordinaten werden `x`, `y` und `z` genutzt.

[1] www.scribd.com/doc/6878959/NASA-C-programming-guide

Benutzen Sie diese Konventionen wenn Sie in ihr Programm passen. Versuchen Sie nicht eigene, neue Konventionen zu erfinden, die nur Sie selbst verstehen.

Die folgenden Regeln zur Groß- und Kleinschreibung sollten Sie für die verschiedenen Elemente in ihrem Programm nutzen. Durch die einheitliche Verwendung eines Stils können Sie als Programmierer und Leser eines Programms sehr schnell die Bedeutung und Verwendung der verschiedenen Elemente bestimmen.

variablenNamen: Namen von Variablen werden immer klein geschrieben. Zusammengesetzte Namen werden dadurch gekennzeichnet, dass der erste Buchstabe des folgenden Worts groß geschrieben wird.

KONSTANTEN: verwenden ausschließlich Großbuchstaben. Um Konflikte mit bereits definierten Konstanten aus Bibliotheksfunktionen zu vermeiden kann es notwendig sein einen Prefix wie zum Beispiel `MY_CONSTANT` zu verwenden.

FunktionsNamen: beginnen immer mit einem Großbuchstaben und sollten nach Möglichkeit ein Verb enthalten welches die Funktion beschreibt (z.B. `SearchString()`). Funktionsnamen für Testfunktionen sollten mit '`Is`' oder '`Are`' beginnen (z.B. `IsNumber()`).

NutzerDefinierteTypen_t: enden immer mit einem '`_t`'. Namen für Typen müssen groß geschrieben werden. Dadurch werden Konflikte mit bereits definierten POSIX Namen vermieden.

pointerNamen_p: um Pointer Variablen sichtbar von anderen Variablen zu unterscheiden sollten Sie Pointer mit einem '`_p`' enden lassen.

A.3 Klammern und Einrückungen

Die größte Vielfalt der Stile finden sich in C bei der Positionierung von Klammern und Einrückungen. Deren Hauptaufgabe besteht darin den Code optisch zu gliedern und funktionale Bereiche durch die konsistente Verwendung von Einrückungen sichtbar voneinander abzugrenzen.

Die einzelnen Stile unterscheiden sich hierbei in der Art und Weise wie die Klammern mit dem Rest des Kontrollblocks positioniert und eingerückt werden. In diesem Kurs wird der so genannte *BSD/Allman* Stil verwendet, weil er den lesbarsten Code produziert. Bei diesem Stil nimmt der geschriebene Code mehr horizontalen Raum ein als bei dem ebenfalls sehr weit verbreiteten K&R Stil. Der *BSD/Allman* Stil macht es allerdings sehr viel einfacher alle öffnenden und schließenden Klammern im Blick zu behalten.

Im folgenden sehen Sie ein Auflistung verschiedener gebräuchlicher Klammer- und Einrückungsstile. Die Einrückungen betragen immer vier Leerzeichen pro Level:

```
/*Whitesmiths Style*/
   if (condition)
       {
       statement1;
       statement2;
       }
```

Der Stil ist nach einem frühen kommerziellen C Compiler *Whitesmiths C* benannt, welcher diesen Stil in seinen Programmbeispielen verwendet hat. Die Klammern befinden sich auf dem äußerem Einrückungsniveau.

```
/*GNU Style*/
   if (condition)
     {
       statement1;
       statement2;
     }
```

Die Klammern befinden sich in der Mitte zwischen inneren und äußerem Einrückungsniveau.

```
/*K&R/Kernel Style*/
   if (condition) {
       statement1;
       statement2;
   }
```

Dieser Stil wurde nach den Programmierbeispielen des Buchs *The C Programming Language* von Brian W. Kernighan und Dennis Ritchie (die C-Entwickler) benannt.

Der K&R Stil ist am schwersten zu lesen. Die öffnende Klammer befindet sich an der äußersten rechten Seite der Kontrollanweisung und ist damit schwer zu finden. Die Klammern haben unterschiedliche Einrückungstiefen. Trotzdem ist dieser Stil weit verbreitet und viele C-Programme nutzen ihn. Sie sollten deshalb in der Lage sein diesen Code lesen zu können.

```
/*BSD/Allman Style*/
   if (condition)
   {
       statement1;
       statement2;
   }
```

Die Klammern befinden sich auf dem inneren Einrückungsniveau und sind damit leicht zu finden und zuzuordnen. Dieser Stil wird für alle Beispiele dieses Kurses verwendet.

Wenn Sie Programme schreiben ist es am Wichtigsten sich auf einen Stil festzulegen und diesen Stil dann konsequent beizubehalten. In größeren Softwareprojekten sollten sich alle Mitwirkenden auf einen gemeinsamen Stil einigen. Moderne

Programmierumgebungen wie zum Beispiel Eclipse[2] und Code::Blocks[3] machen es leicht durch automatische Einrückungen einen Stil durchzusetzen.

A.4 Layout

Kommentarblöcke können dazu genutzt werden die Funktion des Programms zu dokumentieren und zusätzliche Angaben zum Ersteller zu machen. Sinnvollerweise werden diese Angaben als erste Angaben noch vor den Funktionsdeklarationen vorgenommen.

Einen ähnlichen Kommentarblock können Sie vor jeder Funktion verwenden um deren Funktion zu beschreiben.

```
/*
 * File:     test.c
 * Author:   Peter Programmer
 * Date:     May, 29th, 2009
 *
 * Purpose: to demonstrate good programming
 *          practise
 * /

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

/*
 * Main function, input: none, output: 'HelloWorld'
  */

int main (void)
{
    printf("Hello World!\n");
    return EXIT_SUCCESS;
}
```

[2] http://www.eclipse.org
[3] http://www.codeblocks.org/

Anhang B

ASCII-Tabelle

Dec	Hex	Oct	Character	Dec	Hex	Oct	Character
0	0x00	000	NUL	32	0x20	040	SP
1	0x01	001	SOH	33	0x21	041	!
2	0x02	002	STX	34	0x22	042	“
3	0x03	003	ETX	35	0x23	043	#
4	0x04	004	EOT	36	0x24	044	$
5	0x05	005	ENQ	37	0x25	045	%
6	0x06	006	ACK	38	0x26	046	&
7	0x07	007	BEL	39	0x27	047	’
8	0x08	010	BS	40	0x28	050	(
9	0x09	011	TAB	41	0x29	051	)
10	0x0A	012	LF	42	0x2A	052	*
11	0x0B	013	VT	43	0x2B	053	+
12	0x0C	014	FF	44	0x2C	054	,
13	0x0D	015	CR	45	0x2D	055	-
14	0x0E	016	SO	46	0x2E	056	.
15	0x0F	017	SI	47	0x2F	057	/
16	0x10	020	DLE	48	0x30	060	0
17	0x11	021	DC1	49	0x31	061	1
18	0x12	022	DC2	50	0x32	062	2
19	0x13	023	DC3	51	0x33	063	3
20	0x14	024	DC4	52	0x34	064	4
21	0x15	025	NAK	53	0x35	065	5
22	0x16	026	SYN	54	0x36	066	6
23	0x17	027	ETB	55	0x37	067	7
24	0x18	030	CAN	56	0x38	070	8
25	0x19	031	EM	57	0x39	071	9
26	0x1A	032	SUB	58	0x3A	072	:
27	0x1B	033	ESC	59	0x3B	073	;
28	0x1C	034	FS	60	0x3C	074	«
29	0x1D	035	GS	61	0x3D	075	=
30	0x1E	036	RS	62	0x3E	076	»
31	0x1F	037	US	63	0x3F	077	?

Dec	Hex	Oct	Character	Dec	Hex	Oct	Character
64	0x40	100	@	96	0x60	140	‘
65	0x41	101	A	97	0x61	141	a
66	0x42	102	B	98	0x62	142	b
67	0x43	103	C	99	0x63	143	c
68	0x44	104	D	100	0x64	144	d
69	0x45	105	E	101	0x65	145	e
70	0x46	106	F	102	0x66	146	f
71	0x47	107	G	103	0x67	147	g
72	0x48	110	H	104	0x68	150	h
73	0x49	111	I	105	0x69	151	i
74	0x4A	112	J	106	0x6A	152	j
75	0x4B	113	K	107	0x6B	153	k
76	0x4C	114	L	108	0x6C	154	l
77	0x4D	115	M	109	0x6D	155	m
78	0x4E	116	N	110	0x6E	156	n
79	0x4F	117	O	111	0x6F	157	o
80	0x50	120	P	112	0x70	160	p
81	0x51	121	Q	113	0x71	161	q
82	0x52	122	R	114	0x72	162	r
83	0x53	123	S	115	0x73	163	s
84	0x54	124	T	116	0x74	164	t
85	0x55	125	U	117	0x75	165	u
86	0x56	126	V	118	0x76	166	v
87	0x57	127	W	119	0x77	167	w
88	0x58	130	X	120	0x78	170	x
89	0x59	131	Y	121	0x79	171	y
90	0x5A	132	Z	122	0x7A	172	z
91	0x5B	133	[	123	0x7B	173	{
92	0x5C	134	\	124	0x7C	174	|
93	0x5D	135	]	125	0x7D	175	}
94	0x5E	136	^	126	0x7E	176	-
95	0x5F	137	_	127	0x7F	177	DEL